Framework for a Context-Switching Run-Time Reconfigurable System

David I. Lehn

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
Computer Engineering

Dr. Peter M. Athanas, Chair
Dr. James R. Armstrong
Dr. Mark T. Jones

May 3, 2002
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: Configurable Computing, CCM, FPGA, Run-Time Reconfiguration,
Context Switching

Framework for a Context-Switching Run-Time Reconfigurable System

David I. Lehn

(ABSTRACT)

The reprogrammable nature of configurable computing machines has led to a wealth of research in run-time reconfigurable systems and applications. A limitation often encountered in this research is the slow configuration time with respect to the system clock speed. One technique to deal with these configuration delays has been to develop devices that can hold multiple rapidly interchangeable configurations. This technique is known as context-switching.

This thesis discusses the development of a framework to support applications which execute on a run-time reconfigurable system containing context-switching devices. The framework is divided into a number of layers: hardware, middleware, software, and applications. The design, implementation, and details of each layer are presented.

Dedication

To my parents and teachers.

Acknowledgements

I would like to thank my graduate advisor, Dr. Peter M. Athanas, and the rest of my committee, Dr. James R. Armstrong and Dr. Mark T. Jones, for their support and time. Special thanks goes to Kiran Puttegowda and to Dr. Jae H. Park who were vital in helping to complete the research leading to this thesis. Many thanks to my other fellow lab rats for moral support and entertainment. I also wish to thank Rhett Hudson for interesting discussion and encouragement through my early days as a graduate student and continual encouragement to finish. Thanks must also go to all my friends over the years. And finally, a very big thanks goes of course to my loving family for their continual encouragement and support.

Contents

List of Figures

List of Tables

Terms

API Application Program Interface

ASIC Application Specific Integrated Circuit

CCM Configurable Computing Machine

CPU Central Processing Unit

CSRC Context Switching Reconfigurable Computer

EEPROM Electrically Erasable Programmable Read-Only Memory

FIFO First-In First-Out

FPGA Field-Programmable Gate Array

FSM Finite State Machine

HDL Hardware Description Language

IEEE The Institute of Electrical and Electronics Engineers

RAM Random Access Memory

RCM Reconfigurable Computing Module

RCMOS RCM Operating System

RTR Run-Time Reconfigurable

SRAM Static RAM

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuits

Chapter 1

Introduction

The configurable computing community has demonstrated that field programmable gate array (FPGA) technology provides many of the benefits of both application specific integrated circuits (ASICs) and general purpose processors. The commercially available configurable computing devices and systems have traditionally had long configuration times. This property of the devices has limited the potential of run-time reconfigurable (RTR) systems. Research has generated a number of novel approaches to conceal this configuration time from applications. One solution has been to embed multiple configurations in a device and enable fast switching times between configurations. Various research systems based on these devices have become available.

Applications based on these new systems can become complex to develop. Research devices are generally coupled with other hardware that also must be controlled. The development process requires support at many levels from the configurable device designs, to system level firmware, to data passing interfaces, to host software and high level application interfaces.

At a high level, the application programmer wants simplicity and ease of use. The application programmer also wants the ability to control the low level details of a system. The flexibility must exist to adapt to the requirements of multiple types of applications while allowing full

access to the full capabilities of research hardware. This thesis discusses such a framework that supports a context switching run-time reconfigurable system.

1.1 Motivation

The work presented in this thesis was driven by the need to develop applications that target an available context switching device based system. The various types of applications implemented have different requirements for system level support. The framework presented has been developed to support all the requirements of these applications.

The hardware used has much complexity and many features beyond the actual research devices it contains. The development approach taken involved various components and multiple levels of support. The requirement to support various access technologies and interfaces also directed the framework to have many layers of abstraction that each build on lower level layers.

The research nature of the hardware and applications also directed the framework to allow components to be easily testable. Of general use are built-in application debugging and hardware state inspection features.

An attempt has also been made to encapsulate low level details in a high level interface. This is made available to the application programmer to hide many of the details of the entire framework.

1.2 Contributions

This thesis covers the implementation details of a framework for a context switching run-time reconfigurable system. The view from a system framework perspective can provide insight into both future system and device development. Application developers can also

make better judgments on the suitability of such a system by understanding system level issues.

1.3 Thesis Organization

Chapter 2 begins by presenting a background of configurable computing and the evolutionary steps towards context-switching based systems.

The organization of the main focus of this thesis follows a path from a description of the hardware used, through the layers of a support framework, to the final applications that can be built using this framework. Each major layer of the framework developed is discussed on its own. Chapter 3 first describes the hardware platform used in this research. Chapter 4 then describes the middleware support between the hardware and software interfaces. This includes descriptions of the configurations of various programmable components of the hardware. Chapter 5 gives details of the host-based support software required for applications to interface with the context-switching application framework. The software can be separated into various levels each providing the application programmer with various degrees of functionality and access to the middleware and hardware. Each will be discussed.

Chapter 6 describes various types of applications that can use the developed framework and context-switching hardware. Specific applications implemented in this research are presented. Details of how these application use features from the hardware, middleware, and software layers are discussed.

Chapter 7 presents results obtained from the implementation of the applications and framework. Finally, conclusions of this research and suggestions for future work are presented in Chapter 8.

Chapter 2

Background

This chapter gives an overview of the concepts on which this thesis is based. First is an introduction to the type of processing hardware used: configurable computing machines (CCMs). The next concept introduced is run-time reconfiguration, which utilizes configurable properties of CCMs while an application is running. Run-time reconfiguration is then further explored with a technique known as virtual hardware. A multiple hardware configuration acceleration for run-time reconfiguration and virtual hardware will be discussed. Finally, there will be an introduction to higher level software tools that can take advantage of the hardware presented.

2.1 Configurable Computing

Traditional digital logic devices can be roughly separated into two major types. One type is device designed for a specific task. These devices offer high performance for their designed functionality. Flexibility in such devices is sacrificed at design time. Often this type of device is referred to as an Application Specific Integrated Circuit (ASIC). Another type of device is a general purpose processor. These devices trade off performance for a high level

of user flexibility. The common Central Processing Unit (CPU) is an example of this. It can perform a wide variety of tasks but it does not utilize its resources on any one task as efficiently as an ASIC.

A device that offers both the ability to obtain high performance for specific tasks while allowing a high degree of flexibility is a programmable logic device. This device is an integrated circuit that provides a way to implement a wide variety of custom hardware designs. Devices are typically composed of a large set of programmable blocks of logic and programmable signal routing resources. These devices provide performance at a level approaching ASICs while allowing users the ability to program custom functionality after devices have been manufactured. The performance gain is due to the user's ability to make use of a high percentage of the available resources to complete a specific task.

Programmable devices are controlled by a means of a set of programmable points. Logic and signal routing is controlled at these points. The set of data needed to specify all the programmable points is known as a *configuration*. A configuration may also be used to refer to the *design* that has been implemented on a device rather than the data used to represent the design.

Programmable logic device configurations can be implemented in a variety of ways. A one-time programmability can be achieved with the use of fuses and anti-fuses. Electrical component connections in the device can be broken or created, respectively, a single time with such technology. These connections determine logic function and signal routing. This allows a certain amount of design flexibility but limits reuse of the device. A positive aspect to this technology is that it is non-volatile. Configurations can persist without a constant electrical power source. Re-programmable non-volatile devices also exist. They use memory technologies such as EEPROM or Flash to store the device configuration.

Modern technology has allowed a higher degree of flexibility with the use of memory cells to store the logic and routing configuration. This memory is typically Static RAM (SRAM) based. This allows an effectively infinite number of reconfigurations of the device. These

devices often require external non-volatile memory for storage of configuration data. Programming time for configurations varies widely among devices. Tradeoffs are made between configuration time and resources used for the configuration. Traditionally, devices have favored applications that require infrequent configuration events. This has led to devices with long configuration times.

Configurable computing uses programmable logic devices to achieve a goal of hardware programmability and reprogrammability. The earliest known exploration into a computing system based on configurable hardware was proposed and implemented at UCLA [1]. This system augmented a general purpose processor with high speed logic resources. The two were connected by means of an application specific interconnect. In the mid 1980s Xilinx introduced the Field-Programmable Gate Array (FPGA) device. A research community grew around FPGA computing systems and applications. The PRISM [2] project suggested methods of combining reconfigurable logic with a general purpose processor.

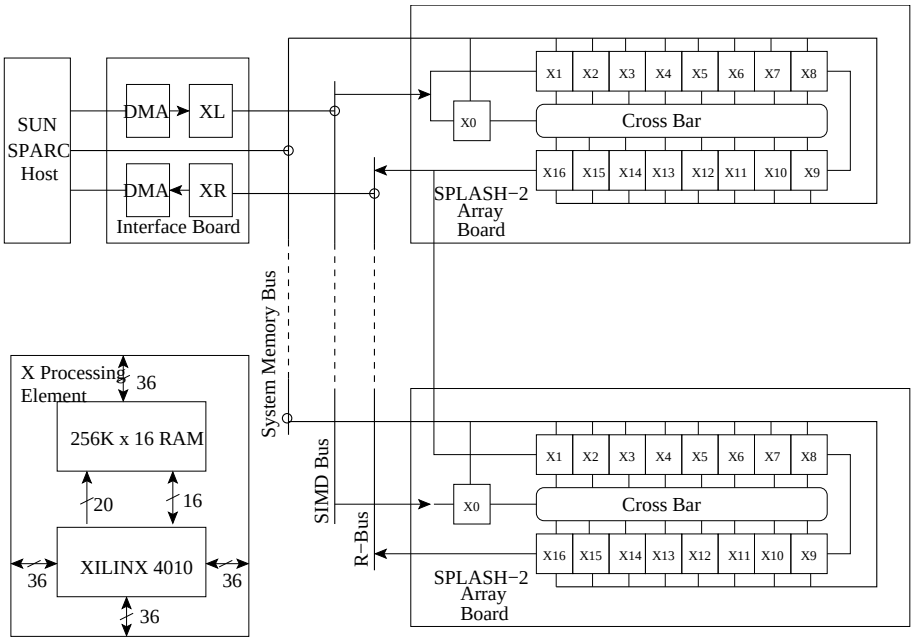


Figure 2.1: SPLASH-2: System architecture

SPLASH-2 [3] [4] is an implementation of configurable computing resources used as a co-

processor board to a Sun SPARC workstation. SPLASH-2 is specifically designed to support high-performance linear systolic applications. The SPLASH-2 architecture is shown in Figure 2.1. Each SPLASH-2 board has sixteen Xilinx XC4010 FPGAs connected in a linear systolic array. Complementing the systolic data-path are two other methods of FPGA communication. The first is a global broadcast data-path and the second is a common user-defined cross-bar data-path. These allow a good degree of on-board flexibility. The system is designed to be scalable by adding more boards and interconnecting the ends of the systolic array as shown in Figure 2.1. Each FPGA processing unit on the board is connected to a local 256K $\times$ 16bit memory. Each memory is mapped to the address space of the host workstation processor. FIFOs at each end of the systolic array is used to provide data buffering for the host processor and increase throughput. Data-paths between neighboring processing units and to the cross-bar are 36 bits wide. This is to allow a 32-bit data path as well as a 4-bit data tag. Control of the system is done through an additional FPGA.

Development of SPLASH-2 applications starts by first manually partitioning the design to fit into individual FPGAs. The design for each FPGA in the system is then described in VHSIC Hardware Description Language (VHDL). Initial debugging is done in a simulation environment. Then the designs are synthesized and the configurations loaded on the hardware. Debugging continues with run-time debugging tools. The partitioning of an application to the various components of the system provided the most difficulty. The resulting applications were shown to have a significant performance speedup over other computing systems of the time. SPLASH-2 achieved two orders of magnitude speedup on genome sequence matching compared to supercomputers of that time such as the Cray2. Image processing applications using a number of algorithms also reported significant performance gains [5].

2.2 Run-Time Reconfiguration

Applications often only require that configurable computing resources be configured once. This is known as *static reconfiguration*. In this model, the entire design must fit within one configuration. The SPLASH-2 is an example of such as system. An application configuration is loaded onto the hardware a single time as discussed in Section 2.1. This technique is also known as rapid prototyping. Configurable resources can be used in the development process of designs that will later be used in non-reconfigurable devices or systems. This offers the flexibility to quickly change hardware designs during the development cycle.

The nature of configurable computing devices does not limit them to static reconfiguration. Another technique is to allow applications to dynamically change configurations at run-time. This is known as *run-time reconfiguration*. In this model the configurable hardware resources can be reused over time. Control of Run-Time Reconfigurable (RTR) systems can be divided into three categories. The first is *application-driven RTR* in which other application logic controls reconfiguration. Another is *host-driven RTR* in which the host or the user provides the control. Finally is *data-driven RTR*. Control of reconfiguration can be determined by the data passing through or generated by the system.

Various approaches have been researched for run-time reconfiguration. One effort of note uses RTR techniques to achieve instruction-set metamorphosis [2]. This technique attaches a reconfigurable functional unit to a general purpose microprocessor. The microprocessor's instruction set has some control over the configurable resources and the ability to implement custom instructions with the configurable logic.

An example of a host-driven RTR system is the Gate Array Reconfigurable Processor (Garp) [6] developed by the Berkeley Reconfigurable Architectures, System and Software (BRASS) group. Garp combines reconfigurable hardware with general purpose processor on the same die. Figure 2.2 shows the Garp block diagram. The processor core uses a standard MIPS-II instruction set augmented with custom instructions. The reconfigurable array has

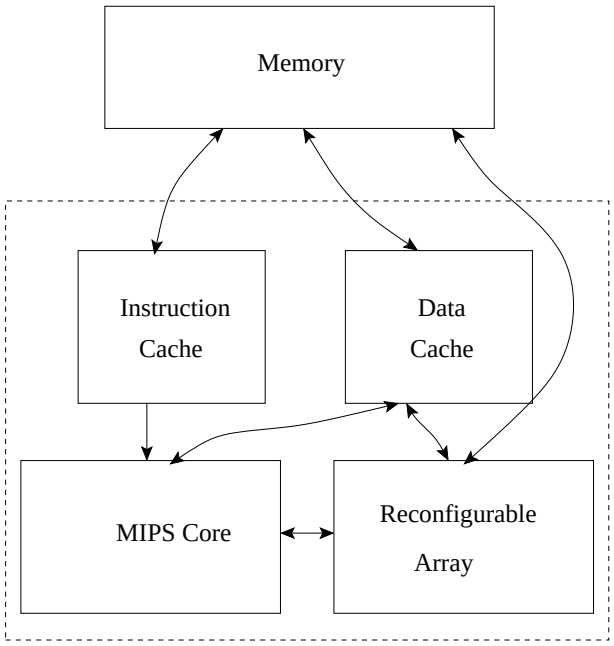


Figure 2.2: Garp: Block diagram

data registers that the processor can access. The array has no direct access to the processor registers. The reconfigurable array can access the same external memory hierarchy as the main processor. Memory consistency between the array and the processor is guaranteed with this technique.

The Garp reconfigurable array is a two-dimensional array of entities called *blocks*. One block in each row is known as a *control block*. All other blocks are *logic blocks*, which correspond roughly to the CLBs for the Xilinx 4000 series [7]. The Garp architecture fixes the number of block columns such that it fits a common data path width. However, the number of rows is implementation specific and the architecture is defined so that the number of rows can scale in an upward-compatible fashion.

The main processor has a number of special instructions for controlling the array. These instructions are used for loading configurations, for copying data between the array and the processor registers, for manipulating the array clock counter, and for saving and restoring

array state on context switches. This gives an application writer simple direct access to configurable logic configuration and control.

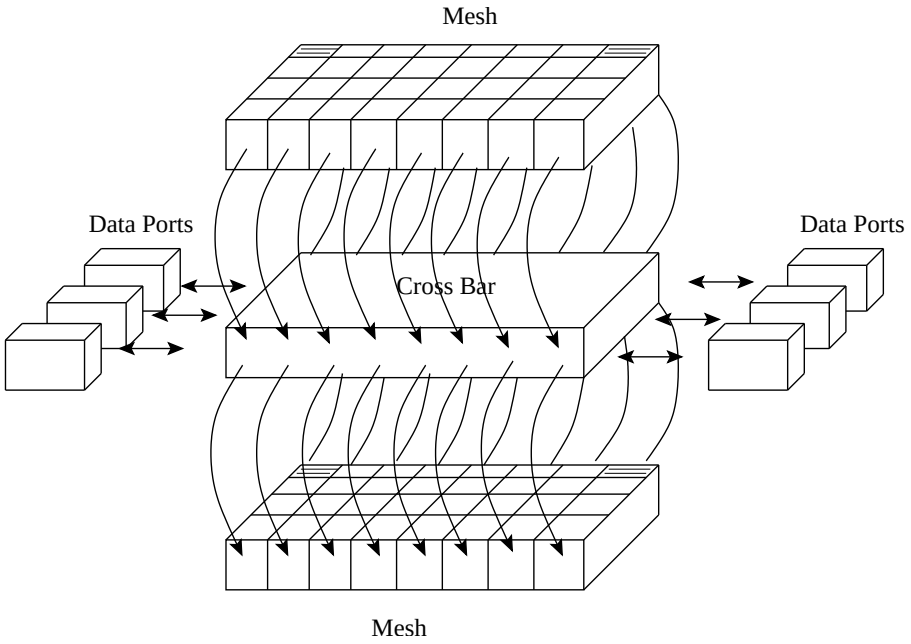


Figure 2.3: Stallion: System architecture

Another RTR architecture approach is wormhole run-time reconfiguration [8] [9]. This approach is based on a data-flow computer architecture. Wormhole run-time reconfiguration uses custom computational pathways through a configurable mesh. These pathways are rapidly created and modified using a data-driven partial run-time reconfiguration scheme.

Stallion [10] is a processor based on wormhole run-time reconfiguration. The Stallion architecture is shown in Figure 2.3. Data input and output are done through data ports. Data is processed in a mesh composed of an array of interconnected Functional Units (IFUs). These units are the basic programmable processing element of the Stallion architecture. A cross-bar gives full connectivity between the IFUs in the mesh and the data ports. Computational streams are used to reconfigure the device. The wormhole RTR concept involves independent streams composed of both configuration data and operand data that move and interact within the architecture to perform a computation.

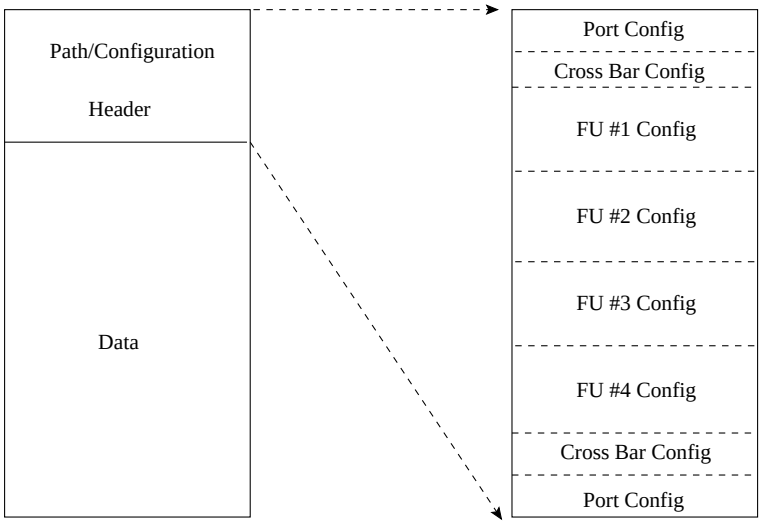


Figure 2.4: Stallion: Stream format

Figure 2.4 illustrates the stream format for the Stallion. A stream is composed of a header segment and a data segment. The configuration programming information is in the header followed by the operands. The programming information configures various resources inside the Stallion device as the stream creates a computational path. This path is followed by the operands in the stream and determines what data processing will occur. As the header travels through resources in the device each unit gets configured. The unit then directs the rest of the stream to other blocks inside the device according to its own configuration. As this process continues the programming header is stripped of the stream. After an entire data path has been configured no header remains. The order and length of programming information in the header is not fixed. The stream, its 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 [11].

2.3 Virtual Hardware

Run-time reconfiguration allows an application to change configurations of a configurable computing device as often as required. This property can be used to implement a concept known as *virtual hardware*. This concept is similar to the virtual memory system found in common computing systems. Virtual hardware reuses configurable computing resources to realize a large configurable resource with a small real hardware. This can be implemented by swapping configurations in and out of the hardware as needed at run-time.

The concept of virtual hardware was first introduced by Ling et al. [12]. 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 configuration and *preloading* for the process of loading a configuration before it is needed for computation.

Xilinx developed the XC6200 [13] which allows configuration and data registers to be memory mapped. This feature allows for fast reconfiguration as well as partial reconfiguration. A number of virtual hardware RTR systems have been developed using this device.

Brebner introduced the Swappable Logic Unit (SLU) [14] which performs the same role in hardware systems that pages or segments do in virtual memory systems. An 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. Efficient use of SLUs involves two specific hardware design restrictions. They must have a fixed area in their FPGA implementation and have fixed input and output interfaces. Software using SLUs has a more flexible view. It can view SLUs as functions or sub-routines in traditional sequential programming languages such as C or Fortran. Object-oriented environments can use a related set of SLUs which correspond to a set of object methods. Parallel programming environments can map SLUs to parallel constructs.

Two virtual hardware models for SLUs have been proposed [14] [15]. First is a *sea of accelerators* model which consists of a collection of independent SLUs. This model is intended

for SLUs that have host-addressable register interfaces. A second model is a *parallel harness* model which consists of interconnected SLUs. This model is intended for SLUs that have signal interfaces on their perimeter. The operating system supplies the routing between the SLUs. The SLU work suggests a uniform three level interface between SLUs and their environment: the physical interface, the bit-level interaction, and the functional interaction with the environment. This allows flexibility and modularization at a low level and the possibility of high level software libraries to use SLUs.

2.4 Multiple Contexts

One of the major limitations of many attempts at a virtual hardware system is reconfiguration time. Traditional configurable computing devices have long configuration times with respect to clock speeds. An effective virtual hardware implementation would require reconfiguration times in the nanosecond range. An application has to take into account long reconfiguration times as periods when the hardware resources are unavailable for computation. A solution to this problem is to have multiple configurations, also known as *contexts*, stored in the device. The processes of rapidly reconfiguring can then be seen as changing from one configuration to another, known as *context switching*. This technique can accelerate both run-time reconfiguration and virtual hardware applications.

The multiple hardware configuration can be stored in the device in SRAM or possibly in more dense DRAM. There will be a practical limit to the number of contexts which can be stored in the device. An advantageous feature of hardware devices is support for background configuration loading for inactive contexts. This makes it possible to support transparent switching between an infinite number of contexts without stalls for configuration loading. If background loading is not possible then only the on-device contexts will have fast switching times.

WASMII [12] was the first publicly proposed system based on the concept of a multi-context

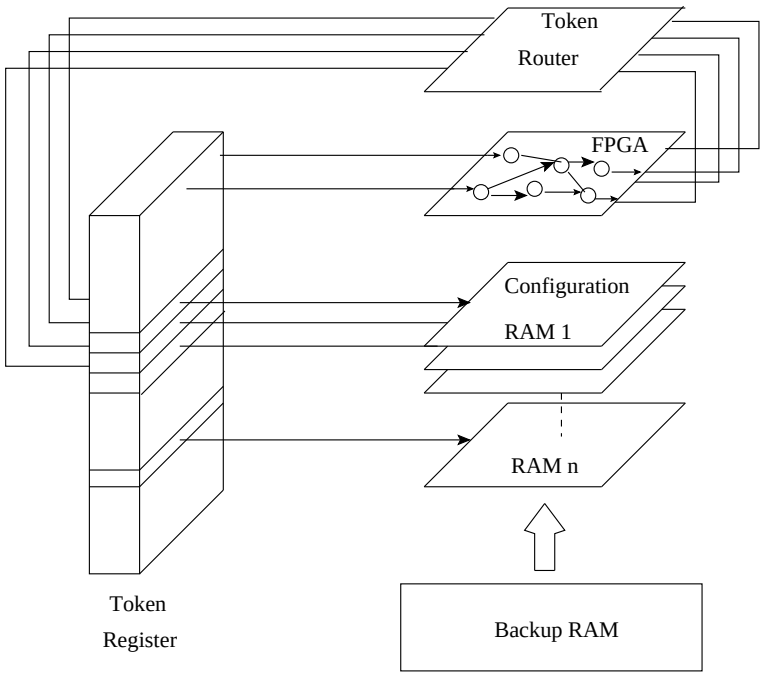


Figure 2.5: WASMII: Device architecture

device. It is a data-driven computational system which uses the concept of virtual hardware to emulate infinite hardware for applications. To reduce the effects of device configuration delays it uses a multiple context architecture. WASMII was later implemented on an experimental multiple context device called Dynamically Reconfigurable Logic Engine (DRLE) developed by NEC [16].

The WASMII device architecture is shown in Figure 2.5. A token router and input token register are attached to virtual hardware composed of multiple contexts and a FPGA programmable logic device. The token router is a packet switching system for transferring tokens between the configurations, known as pages. It receives tokens for the active page and transfers them to the input registers. These token registers are external to the configurations. Every page in the WASMII device requires its own set of token registers.

WASMII is designed to execute data dependent computations. A data-flow graph is partitioned into sub-graphs which fit into a single FPGA configuration. The order in which the

sub-graphs are used for computation is determined by a static scheduling preprocessor. This allows an external RAM to be used to swap in configurations at the appropriate time. A computation is carried out by waiting for input tokens to arrive at the input register. As all the required tokens for a page arrive the page can then be activated. As all the tokens are flushed out of the active page the next ready page is activated in the scheduled order. Each page's logic nodes and wires are realized by the FPGA resources. Nodes start their computation in a data driven manner. Tokens sent out to the activated page are then routed back to the input registers. This enables other pages to be activated. This process continues until the computation is complete.

During the same period as WASMII, the concept of a Dynamically Configurable Gate Array (DPGA) was proposed by Bolotski, et al. [17]. This work also discussed context swapping within an FPGA. The DPGA concept programs several different sets of configuration RAMs for each logic function. The appropriate configuration set could be selected at run-time using global signal wires to select the appropriate context at any given time. 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 device in 1995 [19] [20] and presented the work as a time-multiplexed FPGA [21]. The patented device has an architecture similar to the Xilinx XC4000E [7] and has multiple configuration planes. The reconfigurable communication processor [22] developed by Chameleon Systems, Inc. is the first known commercial multi-context programmable device. It has a reconfigurable fabric with two configurable planes. One is used for executing while the other configures the next part of the algorithm or application. Scalera and Vásquez presented the Context Switching Reconfigurable Computing (CSRC) device in [23]. This device has been used to implement a prototype research platform called the Reconfigurable Computing Module (RCM). This device and platform were used for the research in this thesis and are discussed in Chapter 3.

2.5 Software Tools and Application Frameworks

The choice to use a context-switching device is only part of the design process. Application level development issues must also be considered. Research has explored tools for use at all levels of application design. The choice of a Hardware Description Language (HDL) and associated tools can directly effect the time it takes to generate new configurations. Different HDLs also support various levels of simulation. Some are only circuit level while others allow full simulation from the application down to the hardware. Research has also gone into application frameworks that give a level of hardware system independence. Examples of these software tools follow.

2.5.1 JBits

Another partially reprogrammable set of devices in the FPGA market is the Xilinx Virtex family [7]. Programming characteristics of this family enable virtual hardware to be implemented more efficiently than with other devices such as those from the XC4000 family. The availability of tools is important to enable designers to effectively take advantage of partial reconfiguration. JBits [24] is a Java-based interface to FPGA configuration bit-streams. It enables the designer to significantly reduce the time required to synthesize a custom design compared to traditional partition, place, and route tools. The delay using these tools makes run-time configuration generation impractical even using techniques such as incremental design flow [25]. The introduction of JBits made this run-time capability possible. JBits also allowed the introduction of Run-Time Parameterizable cores [26] which enabled fast parameter-based core customization. Compared with traditional design synthesis speeds this enables a high degree of designer flexibility for virtual hardware applications.

2.5.2 JHDL

JHDL [27] is an exploratory system for the development of reconfigurable system tools. It allows the designer to specify circuits in Java [28]. Java was chosen because it is a high level object oriented language. This allows the designer to leverage software expertise and tools such as debuggers to design hardware. One motivation of JHDL is to allow platform and device independent designs where possible.

JHDL targets a unified simulation and execution environment. A full application can be written in Java and JHDL that can transparently switch between using hardware or software. This allows a high degree of flexibility when debugging a design, when hardware is not currently available, or when exploring new hardware designs from a systems perspective.

JHDL uses traditional tools to convert netlists it creates into device configurations. It supports simulation and targeting of the RCM platform and CSRC devices. Device specific features such as register sharing are also supported.

2.5.3 Janus

The scalable virtual hardware characteristic of context switching reconfigurable systems makes it attractive for users of some types of application frameworks. One such framework that attempts to support applications that are portable between RTR systems is Janus [29]. It provides a high level framework in which an application is divided into hardware and software stages by the user. Temporal ordering and modules that can execute in parallel are explicitly declared. A scheduler and run-time environment determines how to execute these stages on a target platform. The platform can be as simple as only a host processor or as complex as a distributed network of hosts each with local CCM hardware. The Janus project is based on Java and JHDL, and inherits and builds on JHDL strengths, such as the ability to do complete software simulations of a circuit. Janus can use a simulated target architecture just as easily as real hardware. This allows full application simulation from the

user interface to the hardware logic.

Janus was originally based on an attempt to automatically temporally partition and schedule fine-grained data-flow graphs [30]. The scheduling issues proved too complex to complete in reasonable times. This led to the approach of using modules with a higher level of functionality which could adapt to a known target. Work has also been presented to virtualize memory resources which a Janus application targets [31]. This allows an architecture that is memory port poor to execute multiple modules each needing memory access.

Application structure and target platform properties can significantly determine the resulting performance. One example used in the Janus work is the image interpolator algorithm [32] [33]. This algorithm presented a highly regular structure and ability to process entire rows and columns of an image in parallel. This property enables a framework such as Janus to scale with the available resources. A large number of processing elements can process more application modules in parallel. A limiting factor of such systems is the data-transfer rate which also scales with the amount of available processing elements. Applications with little parallel structure or which have many data dependencies may be difficult to scale with additional hardware using a framework such as Janus.

The Janus approach works well on reconfigurable systems as the hardware can be reconfigured for each piece of the application. However, there can be a significant performance degradation due to reconfiguration time. Depending on the application framework run-time implementation, hardware characteristics, and application properties, this can range from an insignificant to an overwhelming computational cost [29] [33]. Context switching platforms and configuration caching techniques can accelerate application frameworks by improving performance of their virtual hardware characteristics.

This chapter gave an overview of the concepts on which this thesis is based. First introduced was the type of processing hardware used: configurable computing machines (CCMs). Building on this configurable hardware is the application level concept of run-time reconfiguration. Run-time reconfiguration is then further explored as virtual hardware. To accelerate

virtual hardware, context-switching architectures use multiple hardware configurations. Finally there was an overview of software tools available to take advantage of this configurable hardware.

The above research does not give a detailed study of the system level support required for a context switching RTR system. The details of such work are often trivial in the case of simple hardware resources or are abstracted into an opaque hardware driver sub-system. This thesis presents a study of the system level details required to support a variety of application styles on a complex hardware context switching RTR system.

Chapter 3

Hardware

The first step in the development of the framework presented in this thesis is to introduce the context switching hardware used. The beginning of the framework is shown in Figure 3.1. Later chapters will build upon this base.

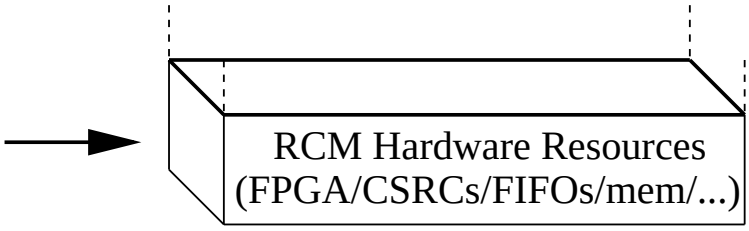


Figure 3.1: Framework: hardware layer

The Context Switching Reconfigurable Computing (CSRC) device presented by Scalera and Vásquez [23] has been developed and integrated into a context switching run-time reconfigurable system by Sanders (now BAE-Systems). A prototype research platform hosting the CSRC devices is called the Reconfigurable Computing Module (RCM). It is the hardware used as the basis of the work presented in this thesis.

3.1 CSRC Architecture

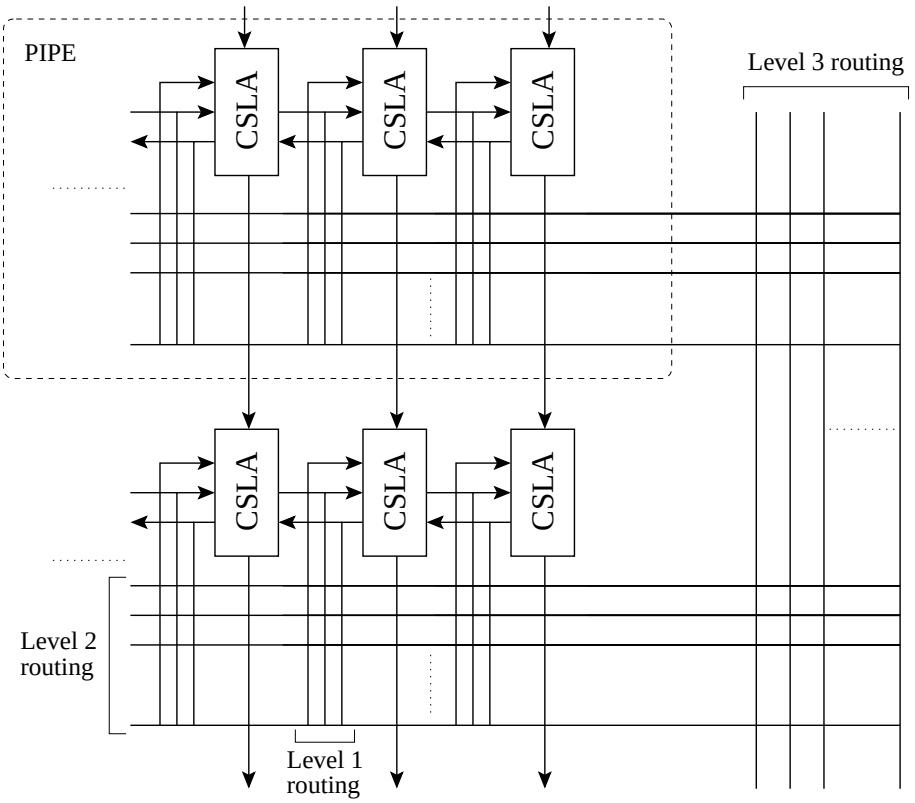


Figure 3.2: CSRC architecture

Figure 3.2, taken from [23], shows the architecture of a single CSRC. The CSRC device 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 (CSLCs) and is capable of processing two 16-bit words to produce 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. If in one context data is processed from left to right, the next context can take that result and process it from right to left.

The CSLC 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 carry

logic. The carry chain can be connected, disconnected or fed a logic zero or one every four bits. This enables a pipeline granularity of four-bits. Each configurable resource in the CSLC along with each routing resource has four configuration bits; a single bit is selected as the current configuration. 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.

The prototype CSRC consists of eight pipes stacked one above the other with eight CSLAs each. Each CSLA has sixteen CSLCs; for a total of 1k CSLCs. 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.

3.2 Reconfigurable Computing Module (RCM)

The RCM is a PCI card that houses the CSRC chips and other necessary hardware used to demonstrate the operation of the context switching reconfigurable computing. The basic architecture is shown in Figure 3.3. Central data communication for the board is handled with an IBM CPC700 PCI bridge. This devices provides a connection between the host machine, an on-board PowerPC 750 microprocessor, secondary cache, RAM, CSRC input and output First-In First-Outs (FIFOs), and the Xilinx XC4085 FPGA. The RCM's main memory is composed of SDRAMs arranged in up to four banks on-board and a 168-pin DIMM connector to accept commercially available SDRAM modules. A pair of synchronous SRAM chips provide 1 megabyte of cache for the PowerPC processor.

The CPC700 provides a one-chip approach to bus and memory handling for the the PowerPC. The RCM firmware programs the CPC700 on power-up for various memory timings and locations which devices are memory mapped to the host PCI bus. All communication to the FIFOs and FPGA are done through these memory mapped regions.

The RCM contains two CSRC devices. Each device is connected to private SSRAM. The two CSRC devices are directly connected with 144 signal lines and there are 48 signal lines from each CSRC to a support FPGA. Both CSRCs are connected through a 36-bit wide FIFO to the processor bus. The assumed data flow is from the processor through the input FIFO to CSRC-A to CSRC-B to the output FIFO and back to the processor bus. The 36-bit wide input and output FIFOs are composed of a pair of 18-bit wide 16k word deep IDT 72V265 hardware FIFO ICs. The processor to CSRC communication is primarily through these two sets of FIFOs. The FIFO control and status signals are connected to the FPGA. This FPGA has a connection to the main processor bus and CSRCs and can be configured to provide access to these signals as needed. The Xilinx FPGA is intended to provide a variety of support functions. The FPGA contains at minimum the ability to receive interrupt requests from the host processor, manage FIFO control and status signals, program and context switch the CSRC devices, clock the CSRCs, and serve as a DMA controller to move data to and from the CSRC devices.

There are a few aspects of the RCM board that the designer must be aware of. One unfortunate design decision is the limited 8-bit wide data path from the FPGA to the processor bus. This limits configuration and data that can be passed to the FPGA. There is also no direct method of accessing the CSRCs local memory except through the CSRC. A workaround for this is to load a CSRC context that routes signals specifically for the purpose of accessing the memory. The unified main bus architecture also is a limitation on its own. The maximum bandwidth of this bus is shared by the PowerPC, main memory, input and output FIFOs, and the FPGA.

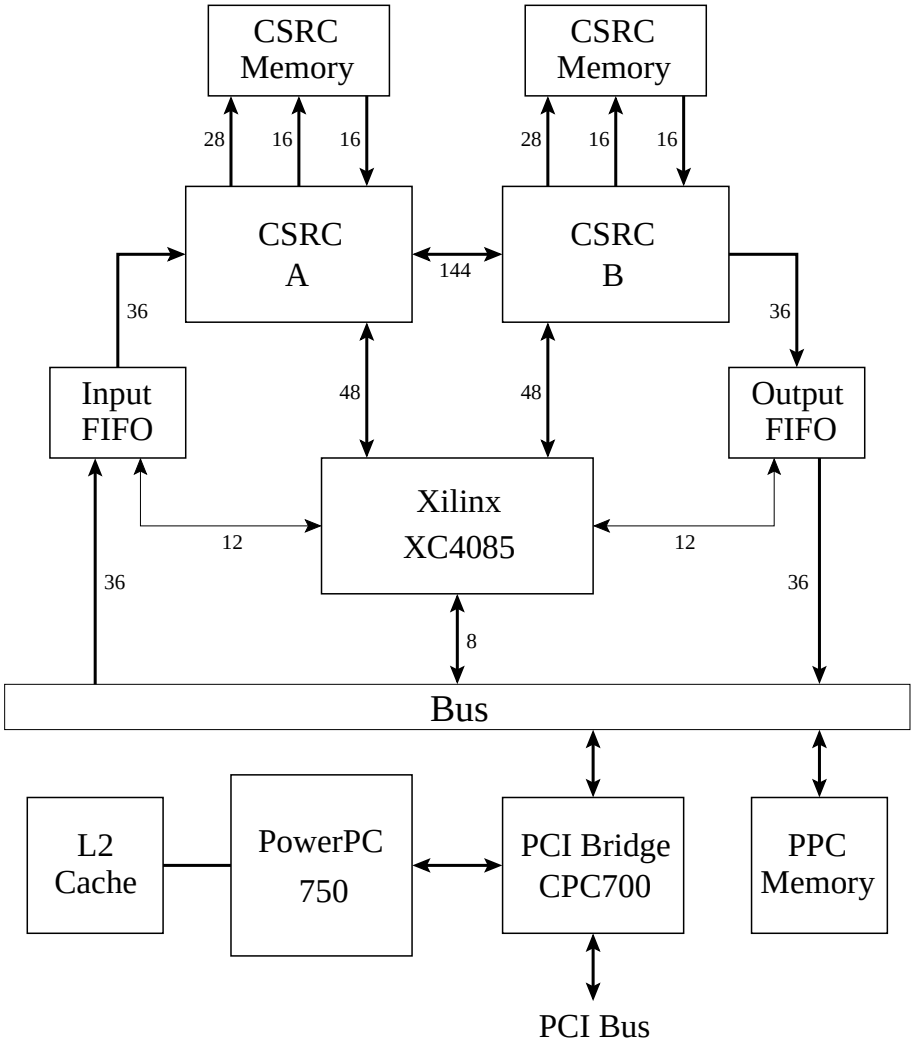


Figure 3.3: RCM architecture

Chapter 4

Middleware

Once the capabilities of the RCM platform are understood, the next layer of the framework can be designed. This layer has two main responsibilities. First is to provide a means for the host to access all the features of the hardware. Second is to provide additional convenience and performance features which applications can use. This chapter introduces the framework features designed for the PowerPC and the FPGA. It also introduces features to simplify application control structures by using the FPGA and PowerPC in conjunction with the CSRCs.

The RCM platform has two locations which this *middleware* can be located. First is in the PowerPC and second is in the FPGA. The PowerPC is examined first because it is easily bootstrapped with a basic program. The PowerPC 750 is capable of running programs from small control loops to full common operating systems. In this research a simplified custom control program is used called the RCM Operating System (RCMOS). The RCMOS will be discussed in Section 4.1. The second location for on-board logic is in the Xilinx XC4085 FPGA. Many control and data signals are connected to this device. It's ability to be customized by the user makes it an ideal location for much of the systems high speed synchronous control logic. The FPGA configuration is discussed in Section 4.2.

Figure 4.1 shows how the middleware layer builds on the hardware. It should be noted that the RCMOS and FPGA configuration need to cooperate to provide complete control to higher layers of the framework.

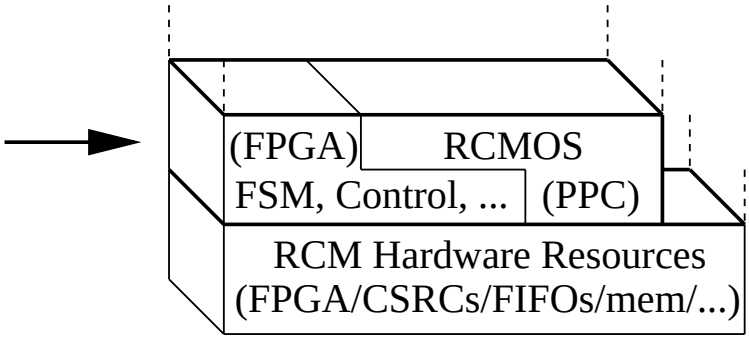


Figure 4.1: Framework: middleware layer

4.1 RCMOS

There is a small amount of on-board firmware. This is mainly used to bootstrap the CPC700. It also sets up the initial code the PowerPC is running. This code is simple. It only polls a control value at a known memory location until it is non-zero. Once it is non-zero the code rewrites this value back to zero and jumps to the memory location specified in an address value stored at another known memory location. The host can load software for the PowerPC by writing it to the memory mapped RCM RAM. This program is then activated by the host writing its address into the address value and signaling the initial firmware to jump to that address with a non-zero write to the control value.

Next, a number of steps are taken to properly initialize the PowerPC. Interrupts are set up, data and code memory sections are initialized, and other low level details. Then the main RCMOS code can take control. Exiting the RCMOS is also implemented which unwinds all the initialization steps. This allows new RCMOS code to be loaded as required. This is especially useful while developing the RCMOS code itself.

The RCMOS provides many features:

- Xilinx FPGA configuration,
- CSRC configuration,
- CSRC configuration caching,
- CSRC control (reset, context switching, clocking, etc),
- streaming data management,
- memory copying, and
- custom application functionality.

This code running on the PowerPC is used to implement many of the low-level API functions, discussed in Chapter 5, in an efficient manner. Programming of the configurable hardware resources involves bit manipulation that would be too slow if done from the host over the PCI bus. Loading the configurations into the board memory and letting software in the PowerPC do the work is more efficient. Many of the CSRC operations such as control-driven context switching and clocking are also more efficiently implemented closer to the hardware than across the PCI bus.

4.1.1 Communication

Two methods of communicating with the RCM board from the host are available: a low bandwidth serial line, used primarily for debugging, and memory mapped reads and writes over the PCI interface. Application interfaces are built on top of the memory-based communication.

The memory mapped RCM RAM is used to provide the host with an interface that appears similar to a local function call. Various commands can be sent to the RCMOS from the host

with parameters. The RCMOS then executes the function and provides return values. The procedure used to perform this shared memory-based communication is shown in Figure 4.2.

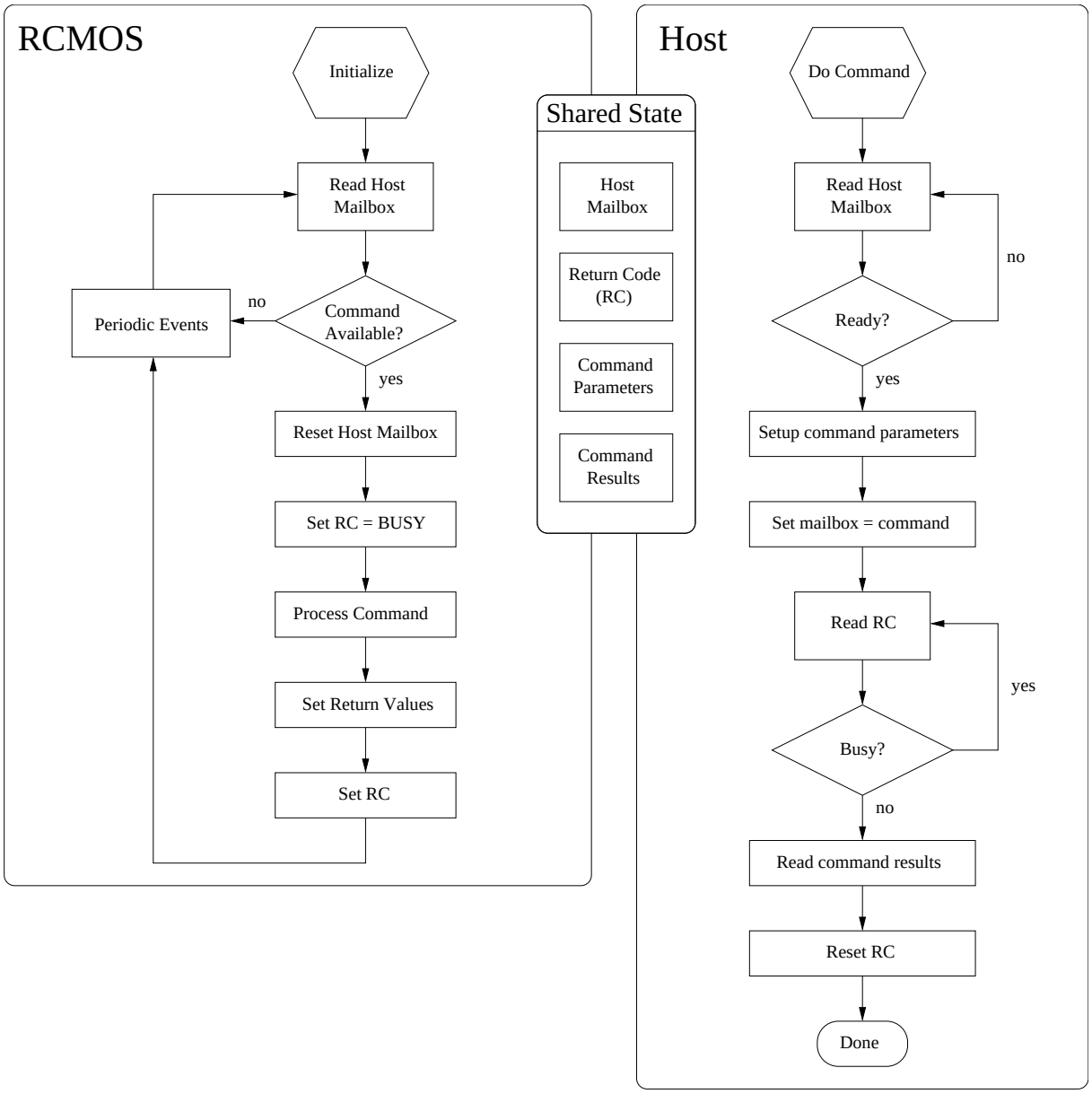


Figure 4.2: Shared memory communication

There exists memory mapped locations that are shared between both the PowerPC and the host. The host mailbox is used to send commands to the RCM and used for acknowledgement

codes. The return code is used to signal the RCMOS is busy processing a command and the result code of that command. There are also memory areas for transferring data to and from the RCMOS associated with a command. These features allow memory-based handshaking synchronization and a simple communication protocol.

The RCMOS is in an infinite loop. It reads the shared host mailbox for a host command. If none exists a few periodic events are potentially run (discussed below) and the command read repeats. If a command is available the host mailbox is immediately rest. Next the shared return code for the host is set to BUSY. The command is then executed. It can read parameters out of shared memory and write results to shared memory. Once the command finishes executing it sets the return code to whatever is appropriate for the command that was executed. Finally, the periodic events are potentially run and the whole command cycle repeats with another command read.

Some tasks need to be run periodically. Streaming data support, discussed below in Section 4.1.3, requires that data be periodically written to the input FIFO and read from the output FIFO. The RCMOS is not have a full multi-tasking design as common operating systems do. Therefore a simple counter is kept each time the code enters the periodic event block. The user can control how often these events are run based on this counter. The user control allows the hardware to be more efficiently used for applications that infrequently, or never, require this functionality.

The host executes single commands on the RCMOS. It executes only as requested rather than in an infinite loop. The first step that is run is to read the shared host mailbox memory location and verify that the RCMOS is ready to accept a command. Next the input parameters for the command are set in the shared memory. The command is then executed by writing the command id to the shared host mailbox. At this point the host loops reading the return code location until it is not in the BUSY or reset state. As soon as the value is valid command return code the loop stops and the output values are read if necessary. Finally, the return code value is reset.

There are some inefficiencies in this communication technique. The polling loops use processing time that could be needed for other tasks. It is assumed that commands do not take a significant time to complete.

4.1.2 Configuration

A primary task of the RCMOS is to move the processing required to configure the programmable logic devices on the RCM from the host to the RCM. The Xilinx XC4085 FPGA must be configured first. The FPGA programming pins are accessed through a CPC700 UART port. The host issues a command to the RCMOS with the FPGA bitstream as a parameter. The RCMOS then does the control sequence to load this configuration on the FPGA.

The CSRC configuration signals are accessed through the FPGA. This requires an FPGA configuration with some known functionality. Similarly to the FPGA configuration, the host issues a command to the RCMOS to configure one of the CSRCs. The configuration data is passed in the parameter memory. The RCMOS then writes to memory mapped registers in the FPGA to toggle CSRC programming data and clock signals.

For RCM board level configuration caching, the host uses an API that stores configurations on the board via the RCMOS. The application uses high level functionality which requests a certain configuration to be loaded. The RCMOS handles the details of keeping track of which configurations are loaded into which contexts in the hardware. If the requested configuration is currently loaded in a context, then a simple high speed context switch will effectively load the configuration from the applications viewpoint. If the requested configuration is not currently in a context, then the RCMOS uses standard replacement algorithms to determine which context it will replace with the requested configuration. Performance could be improved by applications supplying hints on which configurations will be needed in the near futures so that the context can be preloaded in the background before it is needed.

4.1.3 Streaming Data

The RCMOS handles streaming of data to and from various *ports*. Each CSRC has a 32 bit bidirectional port used for both reading and writing between the FPGA and CSRC. A control register determines the direction of the port with the use of tri-state buffers. Output values for this port are stored in registers. There are also a number of other signals available from the FPGA to CSRCs. In this research they are largely unused and are only readable from the host and RCMOS.

These input and output signals provide simple direct access between FPGA and CSRC. It is possible to build FIFO structures in the FPGA but the design and intended use of the CSRC chip could make CSRC designs difficult to implemented due to routing issues. A much better approach is to use these signals for debugging and simple control and the hardware FIFO ICs for data streams.

The RCMOS does, however, support reading and writing streams to most of these ports. The RCMOS code makes a simplifying assumption that when writing to or reading from these ports the CSRCs should be clocked between every data item. On its own this functionality is not that useful. In association with the hardware FIFOs it is possible to write simple processing code. The input hardware FIFO can be filled and a configuration loaded that writes data to the non-FIFO ports. As the data is read the CSRCs will be clocked and can do useful processing.

The computational model used in this research uses the hardware FIFOs. They provide the ability to let the CSRC clock run on it's own. Larger blocks of data can be written to the input FIFO and read from the output FIFO. The FIFOs provide capacity status signals to the FPGA: empty, partially empty, half full, partially full, and full. The FPGA or CSRCs can use these to shut the CSRC clock off to avoid underflow or overflow.

The availability of large memory resources on the RCM board allows for additional software-based FIFO storage. The 16k entry hardware FIFOs are expanded by using this memory.

These software FIFOs are used when enabled by the application and use the exact same streaming interface. Memory copying from the hardware to memory is avoided until needed. Read and write operations will first attempt to directly access the hardware FIFOs. This system is shown in Figure 4.3.

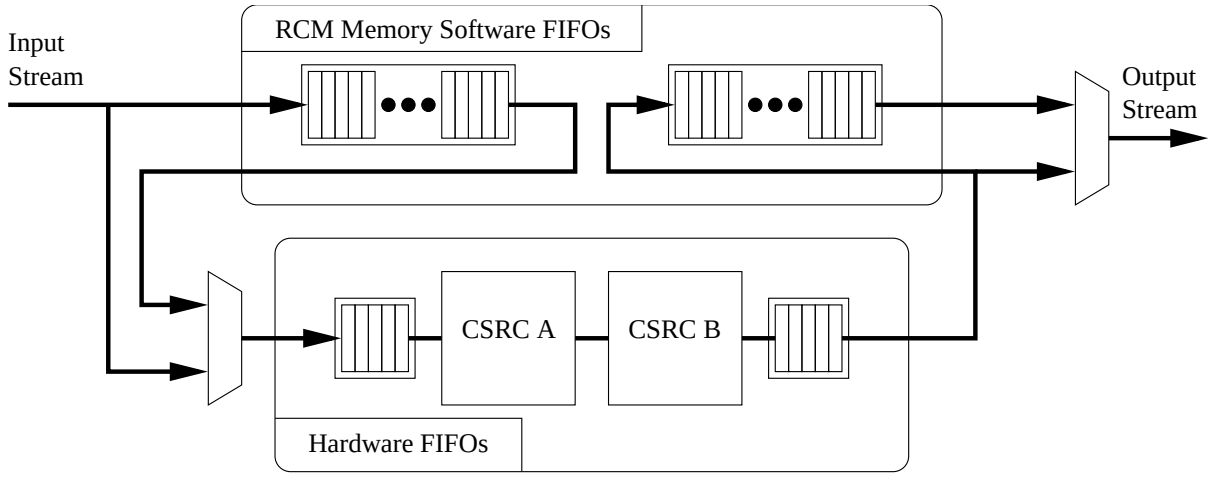


Figure 4.3: Software FIFOs

The RCMOS attempts to keep the input hardware FIFO from becoming empty and the output hardware FIFO from becoming full. The host application can write large amounts of data to the input software FIFO or wait for a large amount of data to be available on the output software FIFO. The RCMOS will handle the data transfer independently of the application. This could be used, for example, to process images that are much larger than the hardware FIFOs without the application handling the memory management details. This mode of operation does increase data input to output latency so it is not appropriate for all applications.

The RCMOS uses polling to watch for underflow or overflow in the hardware FIFOs and performs reads and writes to keep data flowing. The possibility also exists to use PowerPC interrupts for this detection. In this research it was not necessary due to the light workload placed on the processor. If more complex or multi-tasking programs are used then interrupts would be more efficient than polling. The polling occurs at a user adjustable rate as a periodic

event (see Figure 4.2). User settings are available to control the block size of data that are read and written to the hardware FIFOs. This block size is useful to control the situation when the RCMOS is writing to the input FIFO as the CSRCs read data out. A continuous write could potentially never return if the CSRCs read the data fast enough that the FIFO full flag is never active.

4.1.4 Other Functionality

It is possible to set up the CPC700 memory controller such that resources on the RCM board are not directly accessible over the PCI bus from the host. This can happen due to the choice of a memory range. The PowerPC may be able to access the full address space but the host may only be able to access a portion of it. In order to fully access the available address space, memory copying commands are implemented. The write operation first writes the source data into an accessible memory area on the board. Next the write command is issued which tells the PowerPC to copy the data to an address in its address space. This address space is able to fully access all resources on the board. A similar procedure is used for a read operation. A command is issued for the PowerPC to copy data at a location in its address space to a shared data buffer that the host can access.

The PowerPC is able to run custom code for an application. A full operating system and file system are not available so this code is directly integrated with the RCMOS. A command is implemented that allows the host to call arbitrary application code with parameters. It can then return special return codes or allow access to result data.

4.2 FPGA Configuration

The Xilinx XC4085 on the RCM board is used to implement control logic for applications. The FPGA is a normal programmable logic device and the hooks exist to reprogram it as

is needed by the application. In this research there is one common configuration for all applications. The configuration is designed to provide flexible control of the hardware for various types of applications.

A high level block diagram of the FPGA configuration is shown in Figure 4.4. The CSRCs each have their own control logic, switching logic, and switching log. The control logic handles which signals are routed to various outputs and which signals control context number source and context switch signals. The switching logic handles switching control that can originate from the host writing to an FPGA request register, loopback signals from a CSRC, or a number of internal FPGA signals.

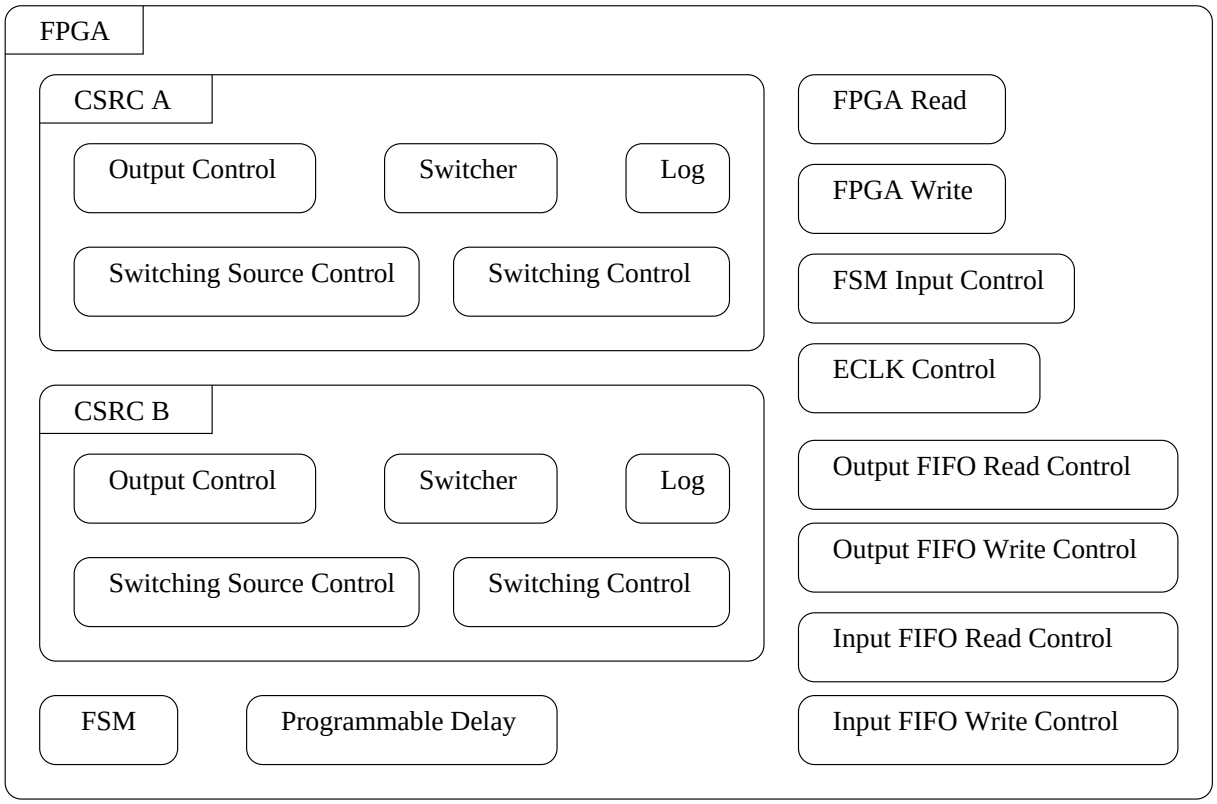


Figure 4.4: FPGA configuration block diagram

The CSRC log is a feature to keep track of context switching activity. The context number that is being sent to the CSRCs is saved when a context switching signal occurs. In this

research the log is implemented as a 128 element memory. This memory tracks only up to the last 128 contexts. Due to it observing the CSRC signals directly it will capture all activity. The context number and switch signal can be configured to come from any source and they will be similarly logged. One limitation is that the CSRCs do contain the functionality to switch their context internally. The log will not be able to detect this switch. This feature is not yet supported by the tools used in this research.

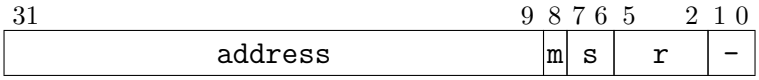
A Finite State Machine (FSM) is included to implement various application control logic. The limited resources on the CSRCs can be more fully utilized for data processing using this FPGA based logic. This is discussed in Section 4.3 and Section 5.4.

A small programmable delay exists for simple pipelines. It delays the input FIFO empty flag from zero to 31 clocks. This signal can optionally be used to control the write enable on the output FIFO. This allows for CSRC pipelines with static delays to easily control the FIFO read and write signals.

4.2.1 FPGA Addressing

The FPGA read and write interface is controlled by two blocks of logic. The FPGA is memory mapped by the CPC700. The address format is shown in Figure 4.5. The whole FPGA is inside the address range determined by the static `address` field. The `m` field is used to explicitly specify read or write mode. The `s` field selects which major group of functions are being accessed. The "board" selections cover a number of miscellaneous features. The `r` field determines the specific register within a major selection.

A write to the FPGA address range will cause a FPGA select signal and a FPGA write signal to become active. When this occurs the logic waits for a clock and looks at which address was written. This write can cause a number of registers to be written. These are discussed in Section 4.2.2. A FPGA read is similar. The logic waits for the FPGA output enable to be active then uses the address signals to determine what value to write on the



field	*width*	*value*	*description*
address	23	0x15000000	FPGA system address
m	1	0	write mode
		1	read mode
s	2	00	CSRC A select
		01	CSRC B select
		10	board section 0 select
		11	board section 1 select
r	4	$r_3r_2r_1r_0$	register selection
-	2	--	unused

Figure 4.5: FPGA address format

shared system bus. The read and write functionality can also be used to perform tasks other than register access.

4.2.2 Control Registers

The FPGA configuration has many control registers. These are accessed through the FPGA read and write mechanism discussed in the previous section. The system bus to FPGA data path width is only 8 bits. Some of the register operations would be more easily implemented with a wider data path. This has resulted in some pairs of ADDR and DATA registers where the ADDR controls what a DATA write will effect. The following sections list are a listing of the registers along with brief descriptions.

Writable CSRC Registers

The first section of writable registers are for the CSRCs. These are shown in Table 4.1. Similar registers exist for both CSRCs. The CSRC is selected with a different `s` field value. The `EXEC_CTXT` register write is implemented with a small state machine which does the proper sequence of events to perform a switch and reset the value. This avoid locking the CSRC in a reset state. `EXEC_CTXT_SRC_CTRL` and `EXEC_CTXT_SW_CTRL` select the source of the next context value and context switching signals. They can select the host value written into `EXEC_CTXT` and an automatic switching signal, constant values, any of a range of output signals from the FSM, or any of a range of signals from either FPGA to CSRC buses. `CSRC_CTXT_LOG_ADDR` sets up a register which selects the log entry to read from `CSRC_CTXT_LOG_DATA`. The other registers offer access to raw signals that connect to various CSRC ports.

`RESET`	Reset
`PROG_DATA`	Program data port
`CSRC_PCLK`	Pulse CSRC program clock
`PROG_CTXT`	Program context select
`EXEC_CTXT`	Execution context register
`EXEC_CTXT_SRC_CTRL`	Exec context source control
`EXEC_CTXT_SW_CTRL`	Exec context switch control
`CSRC_CTXT_LOG_RST`	Log reset
`CSRC_CTXT_LOG_ADDR`	Log read address
`WCSRC_0`	FPGA bus output register (7:0)
`WCSRC_1`	FPGA bus output register (15:8)
`WCSRC_2`	FPGA bus output register (23:16)
`WCSRC_3`	FPGA bus output register (31:24)

Table 4.1: Writable CSRCs registers

Writable Board Registers

Table 4.2 and Table 4.3 show the writable registers available when address select field `s` is set to board section 0 or board section 1. `PORT_DIR` controls tri-state buffers for the FPGA

to CSRC busses. The inverted value of data written to `WTESTREG` is available for reading from `RTESTREG`. This is a useful simple tool to verify the configuration has loaded properly.

`ECLK` is used to manually clock the CSRCs. It will override and reset any free running clock mode. `ECLK_CTRL` lets the application chose between disabling the clock, manually setting its value, using the same clock as the FPGA uses, or tapping off a pattern register. This registers also has flags to determine if the clock is disabled when the input FIFO is empty or the output FIFO full. This allows for a basic level of automatic flow control. The pattern register can be loaded with from `ECLK_PATTERN`. In this implementation the pattern is only 8 bits wide. The CSRC clock can cycle through the bits of this register at the FPGA clock rate. This allows the CSRC clock to be slowed down with some coarse grained control. A 50% duty cycle clock divider can be generated with various patterns. A pattern of `01010101` will divide the clock by 2, `00110011` by 4, and `00001111` by 8. Other duty cycles are possible as well as an even slower clock if the pattern register is expanded beyond 8 bits.

A number of registers are available to control aspects of FIFO behavior. `FIFO_IN` is a somewhat special register. When the FPGA detects a write to this register it toggles the write enable of the input FIFO for one clock. This allows 36 bits of the system bus value to be written to the FIFO. The FPGA ignores the actual data of this write.

The FIFO input write enable control and FIFO output read enable control allow for those signals to be disabled. The FIFO input read enable control and FIFO output write enable control select the source of those signals. Due to routing issues a limited selection is available but could be changed for application specific purposes. The common choices available are disabling the signal, using a constant, CSRC A to FPGA bus bit 0, CSRC B to FPGA bus bit 0, FSM output bit 0, and FSM output bit 1. Additionally, the input FIFO read enable can be enabled when the input FIFO is not empty. The output FIFO write enable can additionally use the programmable delay value.

The CSRC to FPGA bus has fine control over which values are available on the CSRC. `CSRC_OUT_CTRL_ADDR` selects a sub-configuration register for a CSRC part and bus bit.

CSRC_OUT_CTRL_DATA configure that bit to be a constant, use the WCSRC_x value, or use one of the FSM output bits.

The FSM registers are discussed in Section 4.3.

ECLK	CSRC A and B execution clock
ECLK_CTRL	Clock control
ECLK_PATTERN	Clock pattern register
PORT_DIR	CSRC (A/B) Output register enables
FIFO_IN	PowerPC to FIFO write
FIFO_PIPE_DEL	Programmable pipeline delay
FIFO_IN_CTRL	Input FIFO flag control
FIFO_IN_WEN_CTRL	Input FIFO write enable control
FIFO_IN_REN_CTRL	Input FIFO read enable control
FIFO_OUT_CTRL	Output FIFO flag control
FIFO_OUT_WEN_CTRL	Output FIFO write enable control
FIFO_OUT_REN_CTRL	Output FIFO read enable control
WTESTREG4	Test register
WTESTREG3	Test register
WTESTREG2	Test register
WTESTREG	Test register (used with RTESTREG)

Table 4.2: Writable board registers (section 0)

Readable CSRC Registers

Readable registers related to the CSRCs are listed in Table 4.4. PROG_ACK is a programming acknowledgement signal from the CSRCs used when during configuration. CSRC_STAT returns other CSRC status signals. The CSRC_x registers each return a byte from the various FPGA to CSRC buses. The log data register returns the data at the location in the log specified by CSRC_CTXT_LOG_ADDR. The context log count returns the number of elements in the log since the last reset and not more than the value in CSRC_CTXT_LOG_SIZE.

FSM_CTRL	FSM control
FSM_LD_ADDR	FSM programming address
FSM_LD_DATA	FSM programming data
FSM_LD_IN	FSM forced programming load
FSM_INPUT_IN	FSM forced input
FSM_STATE_IN	FSM forced state
FSM_INPUT_CTRL_ADDR	FSM input control address
FSM_INPUT_CTRL_DATA	FSM input control data
CSRC_OUT_CTRL_ADDR	CSRCx_OUT control address
CSRC_OUT_CTRL_DATA	CSRCx_OUT control data

Table 4.3: Writable board registers (section 1)

PROG_ACK	Read APACK signal
CSRC_0	FPGA bus (7:0)
CSRC_1	FPGA bus (15:8)
CSRC_2	FPGA bus (23:16)
CSRC_3	FPGA bus (31:24)
CSRC_STAT	Status
CSRC_CTXT_LOG_DATA	Read context log entry at CSRC_CTXT_LOG_ADDR
CSRC_CTXT_LOG_CNT	Read context log count
CSRC_P3BL	CSRCx_P1B3B (7:0)
CSRC_P3BH	CSRCx_P1B3B (15:8)
CSRC_P3TL	CSRCx_P1B3T (7:0)
CSRC_P3TH	CSRCx_P1B3T (15:8)
CSRC_P5BL	CSRCx_P1B5B (7:0)
CSRC_P5BH	CSRCx_P1B5B (15:8)

Table 4.4: Readable CSRC registers

Readable Board Registers

Table 4.5 and Table 4.6 show the readable registers available when address select field `s` is set to board section 0 or board section 1. `RTESTREG` returns the inverse of the value written to `WTESTREG`. `RADDR` returns the value on the address bus which should be the address of `RADDR`. `RCONST` returns a known constant. These features allow some basic sanity checks of the loaded FPGA configuration. `CSRC_CTXT_LOG_SIZE` returns the context logs maximum size as determined at design time.

The FIFO status flags return empty, partially empty, half full, partially full, and full flags as well as the programmable delay line status. `FIFO_OUT` is similar to `FIFO_IN`. It only controls the output FIFO and does not write a FPGA value to the system bus. The output FIFO read enable is pulsed for one clock if enabled. This will cause the FIFO to write a dequeue an item onto the system bus.

The readable FSM registers are discussed in Section 4.3.

`FIFO_IN_STAT`	Input FIFO status
`FIFO_OUT_STAT`	Output FIFO status
`FIFO_OUT`	FIFO to PowerPC read
`CSRC_CTXT_LOG_SIZE`	Context log maximum size
`RADDR`	Current CPU address (debug tool)
`RCONST`	Constant value (debug tool)
`RTESTREG`	Test register (returns inverted `WTESTREG`)

Table 4.5: Readable board registers (section 0)

`FSM_STATE_SIZE`	Number of FSM state bits
`FSM_INPUT_SIZE`	Number of FSM input bits
`FSM_OUTPUT_SIZE`	Number of FSM output bits
`FSM_STATE`	FSM state
`FSM_INPUT`	FSM input
`FSM_OUTPUT`	FSM output

Table 4.6: Readable board registers (section 1)

4.3 Finite State Machine

The CSRCs have a limited amount of logic and routing resources. Any complex control logic uses large amounts of this valuable resource. The Xilinx XC4085 FPGA is ideal as a location to implement much of this logic. The FPGA has access to signals from both CSRCs as well as the both hardware FIFO units.

Application control logic can be either hard-coded for a specific task or have some host application controllable flexibility. One flexible approach is a host programmable finite state machine (FSM). Each state can control user selectable signals on the RCM board. The state transitions are also controlled by user selectable signals on the RCM board. This is implemented as a table based design in the Xilinx FPGA. An abstract description of a FSM can be converted from a high level description into a memory based table format. This is then loaded into the FPGA with a protocol of low level register writes. This allows flexible application control with the advantage of moving logic to high speed hardware. The prototype CSRC parts have limited logic and routing resources which require some applications to depend on this external control logic.

The implementation details of the FSM used in this research are presented in Section 4.3.1. Programming the FSM using the FPGA register interface is discussed in Section 4.3.2. Application usage of the FSM in terms of higher level software access is discussed in the next chapter in Section 5.4.

4.3.1 Implementation

The FSM is implemented in the FPGA as a dual-ported RAM. A behavioral VHDL representation of the RAM is used. It is written such that synthesis tools can recognize the dual-ported RAM structures and use an optimal implementation for the target hardware.

As shown in Figure 4.6 the FSM has inputs for the current state, the current input, various

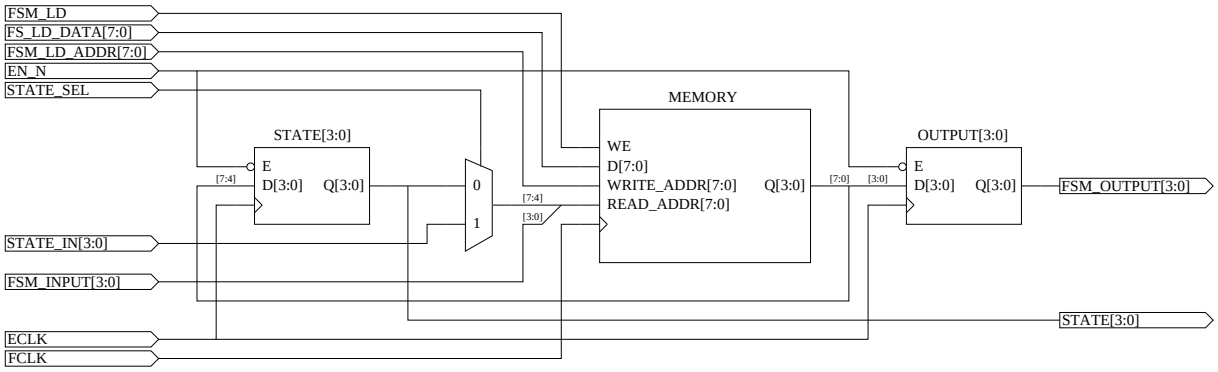


Figure 4.6: Finite state machine circuit

programming signals, forced input signals, and control signals. It has two outputs: the current state and that state's output.

Internally the current state and input are used to create an address into the RAM. The value at that RAM location is both the next state and the output of the FSM for that next state. On each clock the FSM registers this memory value and effectively outputs the current state and output value associated with that state.

Programming the RAM is done off the Xilinx FPGA clock `FCLK`. This clock is not user adjustable. The FSM itself runs off the CSRC clock `ECLK`. This clock speed can be adjusted with control registers as discussed in Section 4.2.2.

The current size of the hardware FSM is fixed at VHDL compile time. The various sizes can be found at runtime by reading the following values:

`FSM_STATE_SIZE` number of bits for state representation

`FSM_INPUT_SIZE` number of bits for input representation

`FSM_OUTPUT_SIZE` number of bits for state output representation

The number of bits needed in an address is:

$$\texttt{ADDRESS\_BITS} = \texttt{FSM\_STATE\_SIZE} + \texttt{FSM\_INPUT\_SIZE}$$

The number of bits needed in a data word is:

$$\texttt{DATA\_BITS} = \texttt{FSM\_STATE\_SIZE} + \texttt{FSM\_OUTPUT\_SIZE}$$

The size of RAM required is therefore:

$$\texttt{RAM\_BITS} = 2^{\texttt{ADDRESS\_BITS}} \times \texttt{DATA\_BITS}$$

For each combination of state and input that can occur in the application the associated memory location must be set. The location can be found by concatenating state address and input bit vectors together. The value at that location is found by concatenating together the next state address and output bit vector for that state.

Note that it is not required to set all locations. If, for instance, only a few states and inputs are required only RAM locations related to them need to be set. Defaults can be provided to allow invalid states to transition to a known state.

4.3.2 Programming

The first step in using the FSM is *programming* it. Programming usually starts by disabling the FSM. Next, the memory is loaded. Then, a pseudo next state and input are forced in order to force the initial state. The initial state and output is loaded on the next clock edge. Finally all signals are reenabled for normal operation.

During programming, it is often required to stop the FSM from changing states. This is not strictly required however. Disabling is done by setting `FSM_CTRL`(1) to 1 and `FSM_STATE_IN` to whatever state you which to use while programming.

There are two programming modes. In the first, FSM memory is automatically updated whenever FSM_LD_DATA is written to. The second is manual writes, performed by toggling FSM_LD_IN. This mode is chosen by setting FSM_CTRL(3) to 0 and 1 respectively.

The procedure is as follows:

1. disable FSM: set FSM_CTRL(0) to 1
2. for each state and input combination used by the application program the FSM:
   (a) set the memory location at FSM_LD_ADDR
   (b) set the memory value at FSM_LD_DATA
   (c) if FSM_CTRL(3) is 1 then toggle FSM_LD_IN
3. initialize:
   (a) use forced input: set FSM_CTRL(2) to 1 and FSM_INPUT_IN
   (b) use forced state: set FSM_CTRL(1) to 1 and FSM_STATE_IN
   (c) clock the FSM via the CSRC clock
4. enable FSM: set FSM_CTRL(0) to 0

The FSM output bits are available to various other configurable sections of the control logic. They can be routed to the CSRC inputs as well as FIFO controls.

The FSM inputs can come from various locations. They are programmed in a similar address/data fashion as the FSM memory. For each bit of the input vector set FSM_INPUT_CTRL_ADDR to the bits address then set FSM_INPUT_CTRL_DATA to select that bits source. The source can be disabled and set to 0, set to a specific value, or set to a bit from the CSRC A or CSRC B output bus.

4.4 Debug Features

Debugging is an important step in the development of any application. The hardware should support this process as much as possible without interfering in the normal operation or performance of the resulting application. The RCMOS and FPGA configuration have a number of these features. The CSRCs can also be loaded with special debugging contexts. These can route signals to debug ports or compute data checks.

One of the easiest ways to debug an application is single stepping the clock. The various clock control features allow this. All the pieces of the system are able to function with manual clock toggling. This is of course very slow but is invaluable. While single stepping the clock a number of signals can be observed. The FSM current state, input, and output are all available. The context log can be read at any time to examine the sequence of all context switches that have occurred.

In summary, the middleware layer of the framework provides a number of services that utilize the programmable hardware as well as a means for higher layers of the framework to access this functionality. An operating system, known as RCMOS, was developed for the PowerPC to provide a convenient high-speed means to access many capabilities of the hardware. A shared memory communication protocol was developed to allow the host to access the features of the RCMOS. A configuration for the Xilinx FPGA was developed that allows flexible access to the available RCM component signals. A method to move high speed control from the CSRCs to the FPGA was developed as a programmable finite state machine. Debugging features were also implemented.

Chapter 5

Software

A layer of software is required to use the hardware and middleware features discussed in Chapter 3 and Chapter 4. This software provides higher level functionality that reduces the need for applications to directly access the middleware and hardware layers. The most basic application must at a minimum load the RCMOS. After this, it is possible to do all FPGA register manipulation and data processing through the RCMOS. In this research, the RCMOS is intended to be simple and most of the application software is run on the host.

This chapter introduces details of the software layers developed on top of the hardware and middleware. These layers build on the framework being developed and are shown in Figure 5.1. The first layer, discussed in Section 5.1, adds a raw access API for the host to RCM board interface. Section 5.2 builds on the raw access layer, FPGA configuration, and RCMOS to provide a low level API. Section 5.3 introduces a high level API which allows applications to implement context switching applications on the RCM board in a more intuitive manner than with the other APIs.

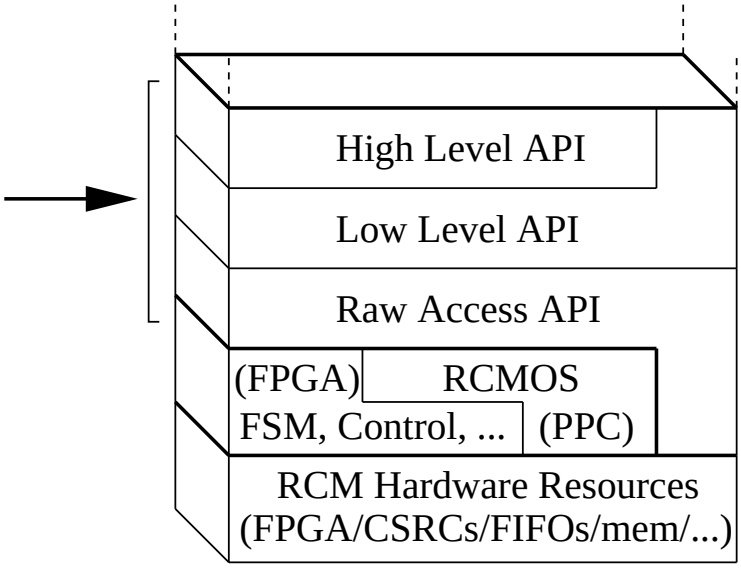


Figure 5.1: Framework: software layer

5.1 Raw Access

As shown in Figure 5.1 the top levels of the framework must access the hardware through a *Raw Access API*. This is the most basic interface to the hardware. It consists of a host operating system device driver which memory maps the RCM board and provides very basic services:

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

The raw access API makes no assumptions regarding the hardware configuration. It does not access special features of any software running on the PowerPC or any features of configurations loaded in the CSRCs or FPGA. The initial setup of software on the PowerPC must be done with this API and board firmware. The memory controller on the RCM board is configured to map memory ranges to various devices. This is used to provide some direct access to the FIFOs and FPGA. This layer also takes responsibility for byte swapping the

data to the proper byte order if required. These architecture specific byte order issues often require that data pass through this extra level of manipulation rather than allowing the data to be directly written to the memory mapped area.

5.2 Low Level Interface

All complex communication and functionality is implemented with a memory-based handshaking protocol with software on the PowerPC. This was described in Section 4.1.1 and shown in Figure 4.2. The *Low Level API*, as shown in Figure 5.1, implements this protocol using the raw access API. It is used to provide transparent access to the hardware and RCMOS from the host application. The API consists of a number of features:

- PowerPC configuration,
- RCMOS communication primitives,
- RCMOS control,
- indirect RCM memory access,
- FPGA configuration,
- CSRC configuration (basic and caching),
- CSRC context switching, clock, reset, and
- data streaming.

The procedure to configure the PowerPC is discussed in Section 4.1. The low level API provides a convenience function to read an S-record file and properly load it into the RCM memory with the raw access API. This function works for any PowerPC configuration. Much of the rest of the low level API assumes the RCMOS developed in this research has

been loaded. A few functions are available to provide the functionality for memory based RCMOS communication. Other functions exist to access various RCMOS features. These include control of buffer sizes, debug flags, periodic event control, software FIFO buffer sizes and control, and various other miscellaneous features. There is also a function to exit the RCMOS and return the PowerPC to an uninitialized state.

As discussed in Section 4.1.4 there is a need to do indirect RCM memory access from the PowerPC for the host. Basic read and write functions are implemented to perform this access.

The Xilinx FPGA is used to access and control many signals on the board: the CSRCs clock and resets, context switching control, programming control, many other CSRC data connections, and FIFO status and control. The low level API makes very few assumptions about how all of these signals are used; it only accesses the FPGA features through RCMOS commands. Custom FPGA control and more complex features are up to either the application or another layer of API. The low level API could be used from the host to emulate most of the RCMOS functionality with FPGA register writes. For efficiency reasons this is discouraged in favor of using the RCMOS functions directly. Using the RCMOS also reduces the dependency of the application on the specific details of the FPGA configuration.

The low-level API for the RCM board exists in two forms. One is a basic C API. The other is through the ACS API [34]. The basic C API is suitable for high speed direct access. The ACS API allows the RCM board to be accessed in the same uniform method as other ACS supported devices in a distributed dynamic network of heterogeneous reconfigurable hardware. In Figure 5.1 both these APIs are represented by the *Low Level API* layer. The ACS API driver for the RCM board is implemented using the C API. However, the application can only access the ACS API so it's interface can be considered an alternative low level API.

5.3 High Level Interface

Applications can also take advantage of a *High Level API*. This API provides an object-oriented view of the board representative of its physical parts. This API includes the functionality needed to take full advantage of the hardware. Access to the low level API is also available when more direct application control is required. This layer is dependent on specific features of the FPGA configuration, features of the RCMOS, and all of the low level API. Applications using this higher level API are isolated from many of the details of register access and bit manipulation. Complex functionality involving many low level API calls can be wrapped up into an easier to use interface.

The implementation of a high level API used in this research was done using Python [35] [36]. Python is a high-level, interpreted, interactive, object-oriented programming language. It is similar to languages such as Tcl, Perl, Scheme, and Java. Access to external C and C++ libraries is possible by means of a Python extension module facility [37]. In this research extension modules were built for both the C low level API and the ACS low level API. A tool known as SWIG [38] was used to create the extension modules. It greatly simplifies the complexities involved in creating a high quality mapping of C and C++ functions to a scripting language extension module.

A high level API could be written in any language. Python was chosen because it provides a good modern object-oriented model and has features that make it ideal for rapid development of applications. Python is not as efficient as a pure C interface; however, in this research the Python code is used primarily for control, initialization, and other low frequency events. The majority of the performance critical work is done by more efficient C code. This design methodology provides the user with a cleaner more intuitive interface layer while utilizing the benefits of optimized code where needed.

The C API and ACS API are interchangeable with a few restrictions discussed below. Figure 5.2 illustrates that the high level API uses an intermediate low level interface. This

interface is mapped to both the C API and ACS API. When the application is started it can select which implementation of the low level interface it wishes to use. The ACS implementation uses the C API to access the lower layers of the framework on the host with the RCM board. The application can then access this host transparently over the network using the ACS API on another system.

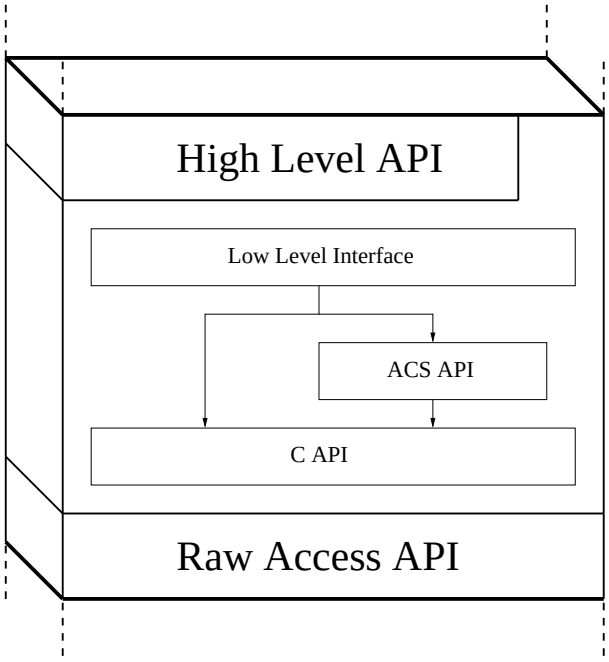


Figure 5.2: C vs ACS low level API

Initialization of the ACS API is slightly more complex due to the possibility of using a RCM board node on a remote system from where the application is running. The C API requires the application be run on the same host as the RCM board. Due to the design of the ACS API the only ports that can be used for streaming data are the input and output FIFOs. Finer control involving interleaving clocking and streaming is needed to make practical use of the other non-buffered ports.

The high level API developed in this research uses an object-oriented representation of the full RCM system. Starting with the RCM board the system is decomposed into sub-components

as shown in Figure 5.3. Each component is representative of a logical part of the board that be controlled. In general each component maps onto a physical part of the hardware. The *Clock* and *FSM* are not physical parts but interfaces to circuits in the FPGA configuration. The *Cache* is an interface to the RCMOS CSRC configuration caching API. Each component is an instance of a class that contains methods that provide application level behavior. This representation allows an interface to the lower layers of the framework without introducing low level details in the application.

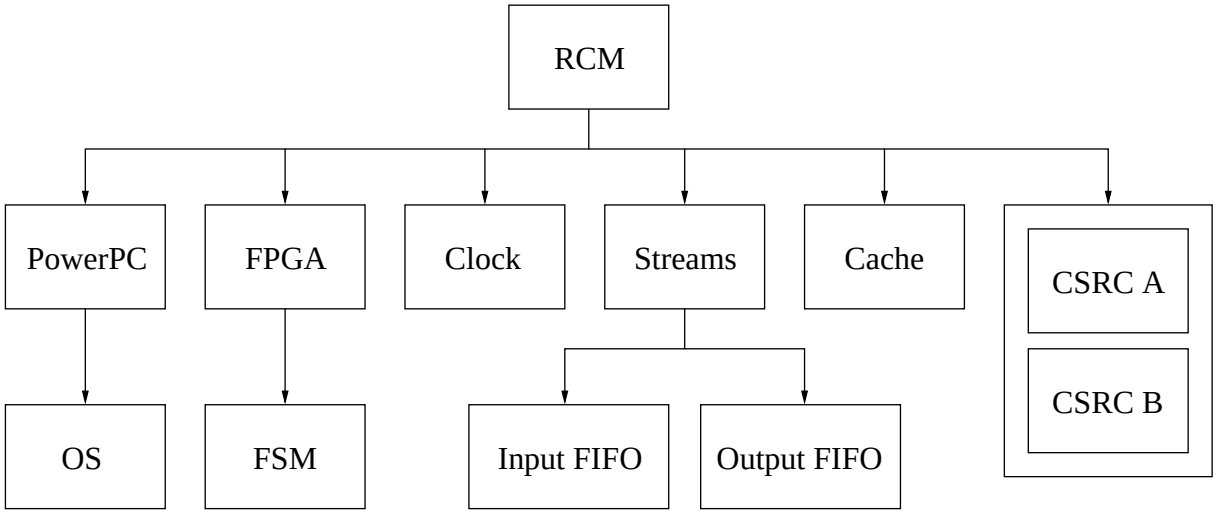


Figure 5.3: Component representation of RCM system

The *RCM* component handles opening and closing of the board, memory read and writes, an overall reset of the board, and some miscellaneous features. Each component has it's own reset interface which calls the reset of it's sub-components. All components also have simplified interfaces to various bit field configurations. This avoids the need for applications to do bit manipulation and direct FPGA register access.

The *PowerPC*, *FPGA*, and *CSRC* components have interfaces for configuration and other control. The *OS* component has interface to access RCMOS specific features that are not specific to any one other component. The *FSM* component is an interface to the FSM described in Section 4.3.1. It's features are described below in Section 5.4. The *Clock* component pro-

vides convenience interfaces for the clock control features in the FPGA configuration. The *Streams* component interfaces into the streaming data features of the RCMOS and hardware. The *FIFO* components have interfaces to the specific stream features and configuration bits of the FIFOs. The *Cache* component interfaces into the caching features of the low level API. In the case of the C API this caching will be handled in the RCMOS. For the ACS API this caching can also occur at a host level to avoid sending configurations over the network. The *CSRC* components have interfaces for context switching and access to various input and output signals.

5.4 Finite State Machine Usage

The FSM implementation described in Section 4.3.1 has a very low level interface: registers are read and written, addresses manipulated, and the details of the memory based implementation must be understood. A simpler interface for high level applications was developed to ease the use of this framework feature.

A simplified object-oriented symbolic finite state machine interface was developed. Each state, input bit vector, and output bit vector are given a symbolic name. Then each state is defined as a named output bit vector and a transition table to other states based on named input bit vectors. The states are added to a FSM object along with the default state for unknown transitions, the default output, and the reset state. The bit width of the input bit vector, output bit vector, and state must also be set. Optionally, the states can be given explicit values, otherwise the state values are automatically determined.

The FSM description is then converted into the memory based format required by the FPGA configuration. This will adapt for the specified state, input, and output sizes. Then convenience functions can be used to load this FSM into the hardware. Other setup may also need to be done to direct the proper signals to the FSM inputs and direct the outputs to the proper location.

In summary, the software layer of the framework provides applications with various methods to access the RCM hardware. Basic hardware access is obtained with the *Raw Access API*. The *Low Level API* builds up a layer to provide a simple application interface to the RCMOS capabilities. The *High Level API* adds another layer that provides applications with an abstract object-oriented view of the RCM board components and the specific middleware RCMOS and FPGA configuration features.

Chapter 6

Applications

The motivation for using any configurable hardware system is, of course, to provide a means to execute an application. Figure 6.1 shows the completed framework with the final application layer. This chapter will discuss various types of applications that can use this framework and present the specific applications that have been implemented as a part of this research. Section 6.1 will introduce the application types and Section 6.2 will preset example applications.

6.1 Application Types

One way to classify various context switching based applications is by their control structure. Figure 6.2 shows a simplified block diagram representative of control paths on the RCM platform. As control structures move from the CSRC devices towards the host there is an increase in latency.

The paths of control signals are application specific. Signals from the CSRCs may need to travel back to logic on the host or anywhere in between. Applications with user input may require the full control path from host to CSRC. Applications can improve performance by

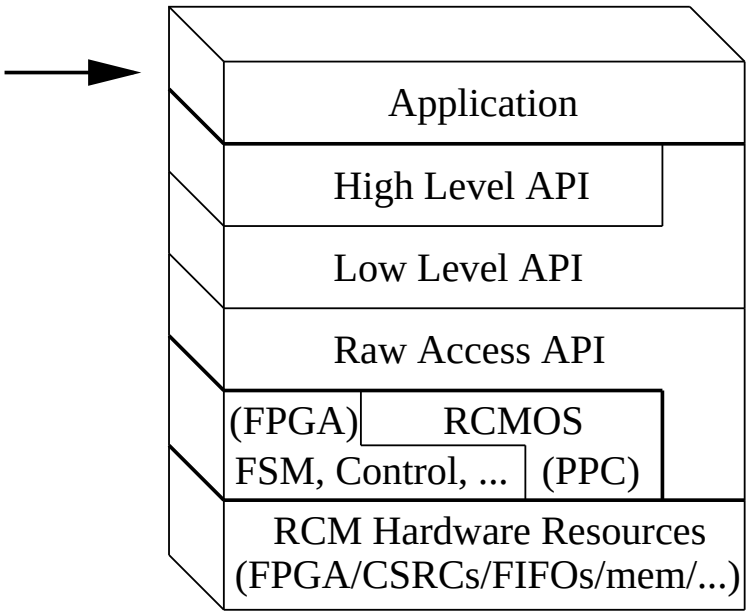


Figure 6.1: Framework: application layer

moving appropriate logic closer to the CSRCs. One possibility is to locate FSM structures on the FPGA. Data are input from the CSRCs to the FSM which will control switching signals. It is also possible that the CSRCs can control their own context switching and not require any external logic.

The latency involved in a context switch is directly related to the number of layers that signals have to pass through. If the CSRCs require host processing for a decision the latency could be large. This communication can be used during debugging to stall the CSRC logic. If a switching decision is made in logic close to the CSRCs, then the switch is done relatively fast. On the RCM board this switching time is as low as one clock cycle for internal CSRC switching and two clocks if external logic is required.

The control latency can divide context switching applications into a number of types. First is *host-driven* context switching. In this type of application the control signals come from a source on the host. This source could be generated by a software event executing on the host or from a user's input. In either case the latency to the context switching logic can be

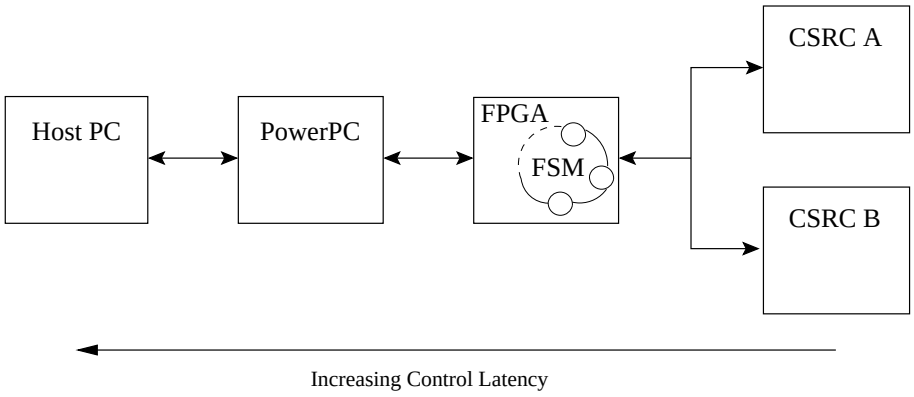


Figure 6.2: RCM control paths

very high. The application designer must recognize that waiting for signals from the host could add long stalls in data processing.

Moving context switching control on to the RCM board removes one layer of communication. Control logic can be added to code running on the PowerPC. This type of control may be beneficial for some *data-driven* context switching applications. This type of application performs analysis on the data to determine when context switching will occur. The data analysis can be performed on the PowerPC rather than the host. The CSRCs can be run at full speed if the data analysis is done on data before it is pushed into the input FIFO. If the output data is used to make control decisions then care must be taken with queued data. The FIFOs and CSRC logic can have partially processed data. The control logic must be able to handle this situation in some manner. One method is to single-step the CSRC clock and read items as they appear on the output FIFO. Output FIFO flags could also be used to stop the CSRC clock. This would require polling of the flags to determine when to read data. These or other methods are likely to dramatically slow computation speed.

Another method of *data-driven* context switching control is to move data analysis logic to the FPGA or CSRCs. Using the FPGA allows data analysis at the same clock speed as the CSRCs. This method can also have issues with partially processed data in the FIFOs. Using the CSRCs themselves for all control and context switching logic allows direct control

of data flow, data analysis at the CSRC clock speed, and full speed data processing.

If the application has a known switching sequence it may be possible to use *control-driven* context switching with control logic in the FPGA. This can be done in the FSM introduced in Section 4.3. The FPGA is running at the same or a faster clock rate than the CSRCs so context switching control is almost immediate. Due to the limited resources on the prototype CSRC devices moving control logic to the FPGA may be necessary.

6.2 Example Applications

This section presents applications implemented as a part of this research. Examples of control-driven, host-driven, and data-driven applications were developed. These applications were specifically targeted to the RCM context switching platform described in Chapter 3. However, these applications are intended to be representative of the application types discussed in Section 6.1. Development of these examples verifies that the framework developed has all the features required for actual application implementation. More detail on implementation of these applications can be found in [39].

6.2.1 Motion Detection Algorithm

The motion detection algorithm implementation demonstrates the control of CSRC contexts through the host programmable finite state machine. The motion detection algorithm [40] basically consists of capturing an image, processing it, and sending out a cropped image where there is motion. Such an application is useful for power critical remote sensing motion detection. The algorithm mapped onto the system is shown in Figure 6.3. 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 is

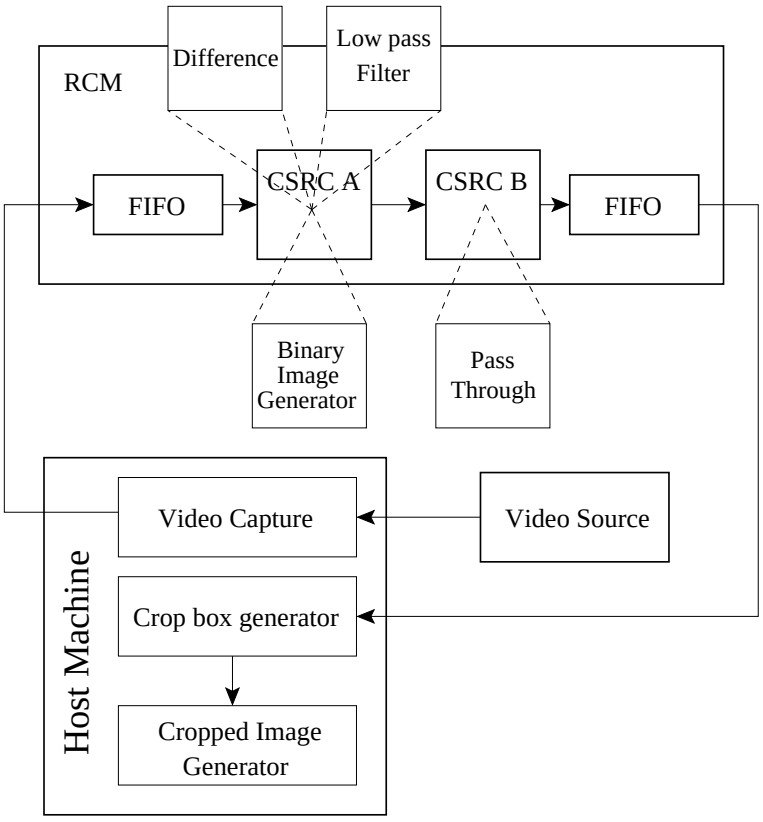


Figure 6.3: Image processing application

used to generate the cropped image by the host machine. A description of the pieces of the application follows.

This application demonstrates a number of pieces of the framework. First is the streaming data capabilities. The application on the host uses the high level API to write images through to the input FIFO. Processed data is read from the output FIFO also using the stream API. The context switching between stages of the application is done with a FSM. The FSM is described and loaded onto the hardware with the high level API using a symbolic description. Using the FSM for context switching control results in a 3-clock switching time.

Video Capturing

A Bt848 video capturing card with a Video4Linux driver is used for generating a video stream. The video stream consists of a sequence of 160×120 8-bit gray scale images. The image data is stored in the shareable memory between the host PC and the PowerPC on 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 local 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

Ideally, the difference image should not have any spot noise and only the moving object on the image should be highlighted as long as the background remains constant. However, there are many factors that generate small background changes, such as wind, ground vibration, etc. In experiments executing this application the camera would also often return many very slightly changed pixels on even static scenes. A low-pass filter eliminates the spot noise and smooths out the image. The low-pass filter is implemented as a 5×5 averaging filter that reads the differenced image from the CSRC memory.

Threshold Estimator

The overall image intensity can change from one image to the next. This can be caused by environmental changes or events that cause the video camera to adjust its exposure. This change in intensity can cause the difference block to produce non-zero pixels throughout the difference image. The pixel differences are compared with a threshold and non-zero pixels due to intensity variations are detected. Ideally a dynamic threshold value should be 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 computational block can be implemented as a separate context.

Binary Image Generator

The binary image generator block compares the filtered image with the threshold and generates a binary image with a '1' for pixels above the threshold and a '0' for those below the threshold. 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 objects that are moved without the spot noise and disturbances due to intensity variations.

This image can be used to clip only the part of the original image frame where there is movement.

6.2.2 Video Filter

A variation on the motion detection application is a host-driven video processing application. A continuous video stream is processed by filters stored in contexts. A user request from the host causes the active filter context to switch. The requested context id is queued and the switch occurs in a single cycle between images. If more filters are needed than available contexts than an algorithm can be used to replace a current context with a new algorithm.

This could be done with special application logic or as part of the configuration caching system. Although not implemented in this research, the loading of new filters into contexts that are currently inactive could take place while another filter is running. This would allows unlimited "virtual hardware" filters. Simple filters have been implemented on the CSRCs that include a basic passthrough, the motion detection difference filter, and a delay filter that provides a "fade out" effect of previous images.

6.2.3 Enigma Encryptor

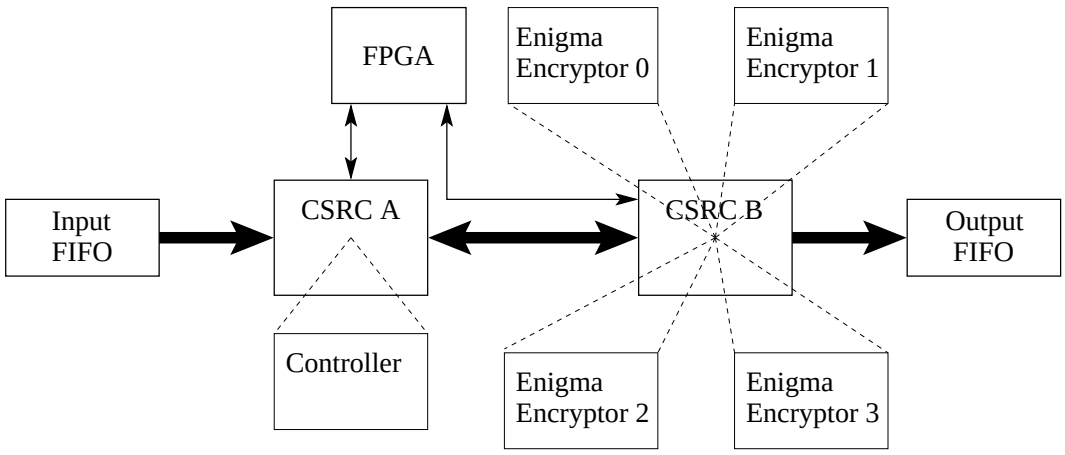


Figure 6.4: Enigma application

This application is a demonstration of the CSRC for data-driven network processing in which the context to be made active is dependent upon the data that is about to be processed. The application is basically a demonstration of simple encryption and decryption of network traffic for multiple channels. Because the resources on the prototype CSRC are limited, a simple Enigma-like encryption scheme was selected.

The Enigma Machine [41], built in the 1920s by the Germans and used in World War II, 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. 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 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.

Network Processing Model

Using the available RCM prototype board, the following system was implemented. FIFOs are used to stream data into the processing element. The processing elements consist of the two CSRCs and the Xilinx support FPGA. Output FIFOs are used to stream data out of the processing elements. The system design partition is shown in Figure 6.4.

Input is a data packet with a header. The header specifies what channel the packet should use and the packet data length. Output is just the processed data without a header.

Each of the four contexts on the CSRC B contain an enigma encryptor with a particular rotor configuration and key. The key could be made programmable using a header field. Each of these individual rotor configurations is used for a specific communications channel. The CSRC A has a controller context on it which processes the incoming packet headers. It determines the channel for which the packet is intended and the number of bytes in the packet. This number is stored in a down-counter register. The controlling context then signals the support FPGA to set the proper context on the CSRC B. CSRC A's controller context now acts as a passthrough and down-counts the number of bytes. After the counter reaches zero, the controller context resets itself and 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. With the hardware cache it 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.

6.2.4 Multiple Filter Signal Processing

Another application similar to the Enigma application was developed. This application performs signal processing on an input signal to determine what filters should further process the data. This application uses the input and output FIFOs as in the Enigma application. Instead of the data-dependent processing occurring on the input data in CSRC A, as in the Enigma application, it is done on the output data. Ideally this processing would occur in CSRC B. This would allow synchronous data flow control and make it much easier to switch CSRC A to a new context immediately as required.

For this research the output data was processed with specialized application code in the RCMOS on the PowerPC. This poses one complex issue. The most efficient way to use the FIFOs from the host is to use large blocks of data. This avoids overhead. But a simple set of filters in a CSRC could potentially process all the input data and put all the results in

the output FIFO at full clock speed. If this occurs then code running on the PowerPC may only be able to process data in large blocks. If the intent was to switch filters on CSRC A at some point before a full block was processed this method will not work.

The solution used for this example is application specific. The input data is single clocked through the system. The data on the output FIFO is read one item at a time. This effectively view the output FIFO as a single register. As soon as the output data signals that a context switch could occur in CSRC A it is switched. For this application the clock speed can then be increased and large blocks of data are processed more efficiently. However, in other applications this may not be possible.

In summary, this chapter first classified context-switching applications into various types: *host-driven*, *data-driven*, or *control-driven*. An application of each type was then presented. Each of the applications shares the common property of processing a data stream. The RCM hardware simplifies these types of applications by including hardware FIFOs. The applications all adapted well to the run-time reconfiguration and virtual hardware properties of the RCM and CSRC hardware.

Chapter 7

Results

This chapter presents the results from the development of a framework for context-switching run-time reconfigurable system. First the results of the framework are presented. Next are observations on the overall configuration caching system. Then is a quantitative discussion of the effects of context-switching hardware design on average context switching time. Lastly are general observations on the CSRC and RCM hardware.

7.1 Framework

Starting with the hardware discussed Chapter 3 a full framework has been developed to support the various types of applications discussed in Chapter 6. The major layers of this framework are shown in Figure 7.1.

The hardware used in this research provided an excellent platform to explore aspects of a run-time reconfigurable context-switching framework and applications. The RCM and CSRCs include a rich set of features to support. Utilizing the RCM board required the development of the middleware layer. This includes development of the RCMOS code running on the PowerPC and a full Xilinx FPGA configuration. This layer includes most of the logic involved

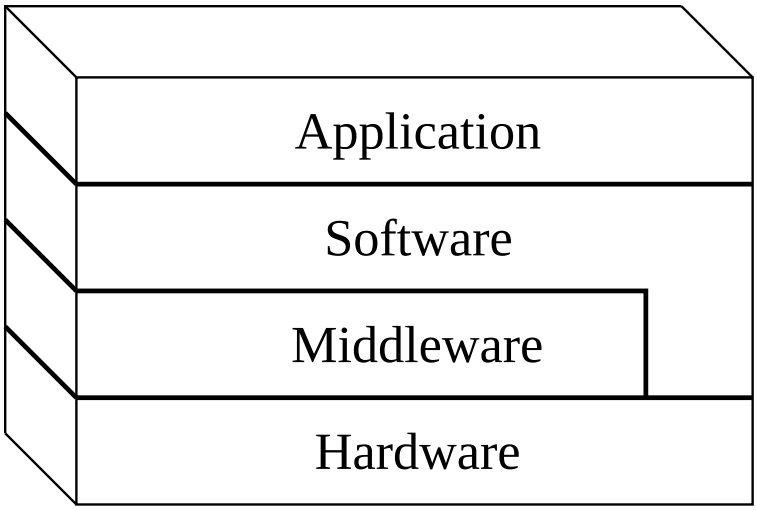


Figure 7.1: Framework: overview

in controlling many of the RCM board signals. The FPGA configuration also includes the finite state machine. This feature became extremely useful during the development of applications. The software layer adds a few sub-layers. First is the direct raw access API to communicate with the RCM board. Next is the low level API which provides a convenient interface to various RCM features through an interface to the RCMOS. The high level API allows applications to access the RCM board in simplified object-oriented way in a high-level language. The applications drove the design of many of the other features of the framework and provided their own challenges. Supporting the various context-switching control methodologies opened up possibilities for large classes of applications.

7.2 Caching

Context switching can be seen as part of a configuration caching hierarchy as shown in Figure 7.2. At the lowest level, configurations are stored in mass storage. This storage can be persistent physical media such as a host hard disk or remote storage accessed over a network. Access speed increases and capacity decreases as the configurations move from

mass storage to host memory to RCM board memory to to device local memory to the device contexts. If any level cannot hold the number of configurations required for an application they are stored in the next higher capacity level and loaded on demand. For such applications caching affects the average switching time.

Cache replacement algorithms can be customized for specific application needs. In this research the only the least recently used (LRU) algorithm is used. Known configuration access patterns may require the use of different algorithms.

The device memory in the figure is representative of the CSRC local memory. Using this memory to store configurations could be more efficient than using the main RCM board memory. The CSRC could reprogram itself or it's neighbor CSRC at the CSRC clock rate without any board system bus access required.

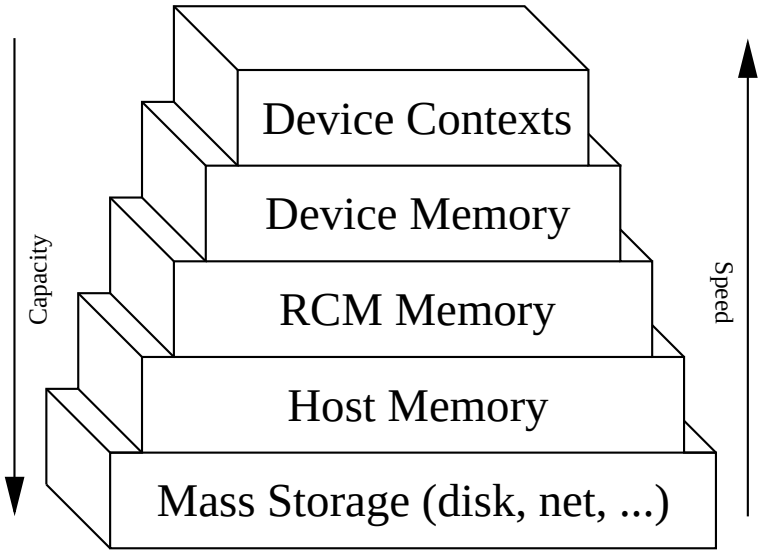


Figure 7.2: Configuration caching hierarchy

7.3 Switching Time

Some applications require more than the available number of contexts in a device. The caching structure described above can be used to store additional contexts. Configurations are moved up the storage hierarchy and swapped into hardware contexts as needed. This will effectively implement a virtual hardware environment. The context switch time in such a system would have a significant latency if the context is not currently available on the device. Assuming that each context for an application has an equal probability of being requested when a context switch is requested, an expression for the average context switching time can be expressed as:

$$t_{\text{avg}} = \begin{cases} t_{\text{s}} & \text{if } 1 < n \leq k \\ p_{\text{s}} t_{\text{s}} + p_{\text{c}} t_{\text{p}} & \text{if } n > k \end{cases} \tag{7.1}$$

where;

t_{avg}	average switching time
t_{s}	context switch time
t_{p}	context program 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 the average context switch time with numbers of application contexts for different number of device contexts is plotted in Figure 7.3. 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}. For illustrative purposes t_{s} is plotted such that it appears significant on the same scale as t_{p}. In an actual device such as the CSRC these values are very far apart. t_{s} can be as low as one clock cycle and

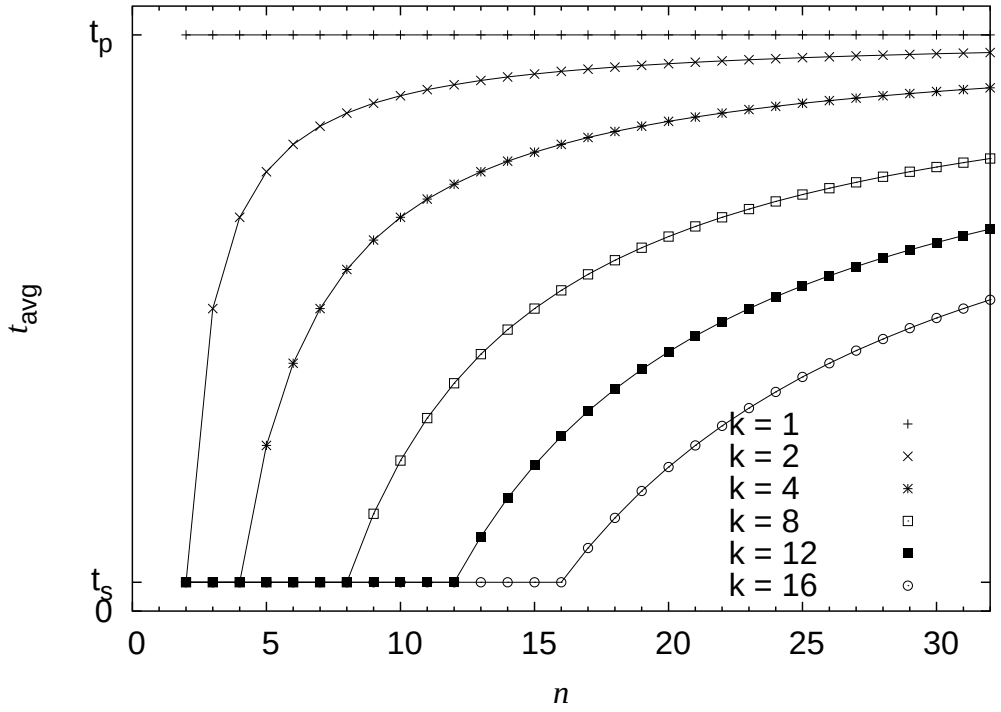


Figure 7.3: Average context switch time

t_{p} nearly three orders of magnitude larger. This difference would only be magnified in a larger device with more configuration data. The plot is still representative of the relative relationship between the variables.

Adding a new context is expensive in terms of additional logic area. It would involve adding new RAM cells for context storage, multiplexers, control logic, and the necessary routing for each configurable resource. RAM cells take up the most significant additional area out of these changes. So the VLSI area in adding new contexts increases nearly linearly. Continuing to add more device contexts results in 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}. For very high number of application contexts the curves come closer. This implies that when an application uses many 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 more intelligent algorithms the probability that the context requested is on the device will be much higher. This would greatly decrease the average context switch time observed in the plot. This locality-of-reference is highly dependent on application behavior. 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 time and hence the average reconfiguration time. This capability can be utilized in applications that can anticipate the next context required.

7.4 Hardware

The CSRC device used is suitable for research into context-switching run-time reconfigurable applications. The logic resources of the prototype device used are very small. A larger device would allow more complex applications to be developed. The limited ability to put control logic on the CSRCs did lead to the necessity of developing the finite state machine that otherwise might not have been as important.

One difficulty with using prototype devices, such as the CSRC, is the availability of tools to support their unique features. For instance, traditional tools were not designed to match register placement between multiple configurations. The CSRC design flow requires such features to be able to properly support data sharing between contexts.

The RCM board is also very suitable for research. It has many major components of a full stand-alone computer. It also has features such as hardware FIFOs to support data streaming applications. The applications in this research used many features of the hardware but did not take full advantage of their processing power. One reason for this is that the prototype CSRC devices are limited in comparison to the other components. However, the concepts presented would apply to a system with larger CSRC devices.

Chapter 8

Conclusions

This chapter summarizes the work presented and gives suggestions for possible future work.

8.1 Summary

A framework to support context-switching run-time reconfigurable systems was designed and implemented. The framework makes use of all aspects of the specific RCM hardware system used. The framework is divided into a number of layers which each provide an interface to higher layers abstracting away implementation details.

Applications were written to demonstrate context-switching run-time reconfiguration. Various types of applications were developed that explore context-switching technology and stress all features of the framework. Using the high level API the applications were resilient to minor changes in lower layers during development.

8.2 Future Work

This research brings forth a number of possibilities for future work:

- Explore background context loading. Additional logic is needed to handle concurrent operations without effecting normal operation.
- Caching algorithm enhancements. Improved or application specific cache replacement algorithms could offer a significant improvement to average context switching time.
- Interrupt usage. The RCMOS uses polling to check FIFO status when needed. For software FIFOs a PowerPC interrupt could be used when specific hardware signal conditions occur. This could reduce unneeded idle activity.
- Improve the RCMOS. The PowerPC on the RCM board is capable of running much more complex programs than the RCMOS. More advanced traditional operating systems could be ported to this system.

Bibliography

[1] G. Estrin, "Organization of computer systems - the fixed plus variable structure computer," in *Proceedings of the Western Joint Computer Conference*, 1960, pp. 33–40.

[2] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, no. 3, pp. 11–12, March 1993.

[3] 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*, June 1992, pp. 316–324.

[4] Duncan A. Buell, Jeffrey M. Arnold, and Walter J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, 1996.

[5] P. M. Athanas and A. L. Abbott, "Real-time image processing on a custom computing platform," *IEEE Computer*, vol. 28, no. 2, pp. 16–24, February 1995.

[6] John R. Hauser and John Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proceedings of IEEE symposium on FPGAs for custom computing machines*, April 1997.

[7] Xilinx Inc., *The Programmable Logic Data Book*, San Jose, CA, 1999.

[8] R. Bittner, M. Musgrove, and P. Athanas, "Colt: An experiment in wormhole run-time reconfiguration," in *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*. SPIE, November 1996, pp. 187–194.

[9] R. Bittner, *Wormhole Run-Time Reconfiguration: Conceptualization and VLSI Design of a High Performance Computing System*, Ph.D. thesis, Virginia Polytechnic Institute and state University, Jan 1997.

[10] Maneesh Soni, "VLSI Implementation of a Wormhole Run-time Reconfigurable Processor," M.S. thesis, Virginia Polytechnic Institute and state University, June 2001.

[11] 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*, 2001, pp. 3370–3374.

[12] X.-P. Ling and H. Amano, "WASMII: a data driven computer on a virtual hardware," in *IEEE Workshop on FPGAs for Custom Computing Machines*, Duncan A. Buell and Kenneth L. Pocek, Eds., Napa, CA, April 1993, pp. 33–42, IEEE Computer Society Press.

[13] Xilinx, Inc., "XC6200 Advance product information," 1996.

[14] Gordon Brebner, "A virtual hardware operating system for the Xilinx XC6200," in *Proceedings of 6th Intl. workshop on Field Programmable Logic and Applications, Springer*, 1996, pp. 327–336.

[15] Gordon Brebner, "The swappable logic unit: a paradigm for virtual hardware," in *Proceedings of IEEE symposium on FPGAs for custom computing machines*, April 1997.

[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] Andre DeHon, "DPGA Utilization and Application," in *MIT Artificial Intelligence Laboratory, Transit Note 129*, September 1995.

[18] Andre DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," in *Proceedings of IEEE Workshop on FPGAs for custom computing machines*, 1994, pp. 31–39.

[19] Xilinx Inc., "Time multiplexed programmable logic device," July 1997, Patent no. 5646545.

[20] Xilinx Inc., "Configuration modes for a time multiplexed programmable logic device," February 1997, Patent no. 5600263.

[21] Steve Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong, "A Time-Multiplexed FPGA," in *Proceedings of IEEE symposium on FPGAs for custom computing machines*, April 1997.

[22] Chameleon Systems, Inc., "CS2000 Reconfigurable Processor," 2000, CS2000 Advance product information.

[23] Stephen M. Scalera and Jóse R. Vázquez, "The design and implementation of a context switching FPGA," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[24] Steven A. Guccione and Delon Levi, "XBI: A Java-based interface to FPGA hardware," in *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, John Schewel, Ed., Bellingham, WA, November 1998, pp. 97–102, SPIE - The International Society for Optical Engineering.

[25] Xilinx Inc., *Using Xilinx and Synplify for Incremental Designing (ECO)*, August 1999, XAPP164.

[26] S. Guccione and D. Levi, "Run-Time Parameterizable Cores," in *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, 1999, pp. 215–222.

[27] P. Bellows and B. L. Hutchings, "JHDL - an HDL for reconfigurable systems," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. M. Arnold and K. L. Pocek, Eds., Napa, CA, April 1998, pp. 175–184.

[28] James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*, Addison-Wesley Publishing Company, 1996.

[29] Rhett D. Hudson, *Architecture-Independent Design for Run-Time Reconfigurable Custom Computing Machines*, Ph.D. thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA USA, August 2000.

[30] Rhett Hudson, David Lehn, Jason Hess, James Atwell, David Moye, Ken Shiring, and Peter Athanas, "Spatio-temporal partitioning of computational structures onto configurable computing machines," in *SPIE Proceedings*, October 1998, vol. 3526, pp. 62–71.

[31] James Atwell, "A multiplexed memory port for run time reconfigurable applications," M.S. thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA USA, December 1999.

[32] Leonard A. Ferrari and Jae H. Park, "An efficient spline basis for multi-dimensional applications: Image-interpolation," in *IEEE International Symposium on Circuits and Systems*, 1997, vol. 1, pp. 757–760.

[33] Rhett D. Hudson, David I. Lehn, and Peter M. Athanas, "A run-time reconfigurable engine for image interpolation," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998, pp. 88–95.

[34] Mark Jones, Luke Scharf, Jonathan Scott, Chris Twaddle, Matthew Yaconis, Kuan Yao, Peter Athanas, and Brian Schott, "Implementing an API for distributed adaptive computing systems," in *Proceedings of the IEEE International Conference on Communications*, April 1999, pp. 222–230.

[35] Guido van Rossum, "Python Reference Manual," March 2000, Corporation for National Research Initiatives, Reston, VA 20191, USA, http://www.python.org/doc/ref/ref.html.

[36] Guido van Rossum, "Python Library Reference," March 2000, Corporation for National Research Initiatives, Reston, VA 20191, USA, http://www.python.org/doc/lib/lib.html.

[37] Guido van Rossum, "Python/C API Reference Manual," March 2000, Corporation for National Research Initiatives, Reston, VA 20191, USA, http://www.python.org/doc/api/api.html.

[38] D. M. Beazley, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Proceedings of the 4th USENIX Tcl/Tk Workshop*, 1996.

[39] Kiran Puttegowda, "Context Switching Stratagies in a Run-Time Reconfigurable System," M.S. thesis, Virginia Polytechnic Institute and State University, April 2002.

[40] Namrata Vaswani and Rama 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.

[41] Deutsches Museum, "Enigma encryption machine," http://www.deutsches-museum-bonn.de/ausstellungen/meisterwerke/2_3enigma/enigma_e.html.

[42] David I. Lehn, Kiran Puttegowda, Jae H. Park, Peter M. Athanas, and Mark 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.

Vita

David I. Lehn

David Ilan Lehn was born in Fairfax, Virginia on January 10, 1975. He enjoyed life in Annandale, Virginia; graduating from Thomas Jefferson High School for Science and Technology in 1993. Along the way David earned the rank of Eagle Scout during his involvement in the Boy Scouts of America. David enrolled into the Engineering program at Virginia Tech in the fall of 1993. He obtained a Bachelor of Science in Computer Engineering Degree with a Minor in Computer Science in 1997. He then continued his education at Virginia Tech working towards a Master of Science in Computer Engineering Degree. David has been involved in reconfigurable computing research since 1997. His hobbies include music, hiking, biking, and software development. His technical interests include reconfigurable computing, computer architecture, software engineering, and hardware and software development tools.