DESIGN OF AN ADAPTABLE RUN-TIME RECONFIGURABLE SOFTWARE-DEFINED RADIO PROCESSING ARCHITECTURE

by

Joshua R. Templin

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Jacob Gunther Major Professor	Dr. Koushik Chakraborty Committee Member
Dr. Todd Moon Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2010

Abstract

Design of an Adaptable Run-Time Reconfigurable Software-Defined Radio Processing Architecture

by

Joshua R. Templin, Master of Science

Utah State University, 2010

Major Professor: Dr. Jacob Gunther
Department: Electrical and Computer Engineering

Processing power is a key technical challenge holding back the development of a high-performance software defined radio (SDR). Traditionally, SDR has utilized digital signal processors (DSPs), but increasingly complex algorithms, higher data rates, and multi-tasking needs have exceed the processing capabilities of modern DSPs. Reconfigurable computers, such as field-programmable gate arrays (FPGAs), are popular alternatives because of their performance gains over software for streaming data applications like SDR. However, FPGAs have not yet realized the ideal SDR because architectures have not fully utilized their partial reconfiguration (PR) capabilities to bring needed flexibility. A reconfigurable processor architecture is proposed that utilizes PR in reconfigurable computers to achieve a more sophisticated SDR. The proposed processor contains run-time swappable blocks whose parameters and interconnects are programmable. The architecture is analyzed for performance and flexibility and compared with available alternate technologies. For a sample QPSK algorithm, hardware performance gains of at least 44x are seen over modern desktop processors and DSPs while most of their flexibility and extensibility is maintained.

(63 pages)

To my wife and son who have patiently borne all the negatives resulting from my academic pursuits and still encourage me.

Acknowledgments

My efforts could not have been possible without the help of my thesis committee, especially Dr. Aravind Dasu and Dr. Jacob Gunther. Dr. Dasu was key to introducing me to reconfigurable computing and giving me critical experience and tools to be able to conceptualize the problem and realize a solution. Without Dr. Dasu, my interest in reconfiguration computing would never have existed, nor would this thesis. Dr. Gunther has been compassionate beyond degree to counsel me and chair my committee, even though much of my work was outside his realm of expertise. Dr. Gunther provided valuable advice along the path and sparked my interest in communication systems that led to this particular application of reconfigurable computing. I especially appreciate Dr. Gunther's selfless efforts to acquire funding for me to support my family while I pursued this research of my own interest. Dr. Todd Moon must also be credited as the teacher that, against all odds, captured my interest in signal processing, which was ultimately fundamental to steering me into digital communications. His challenging coursework caused me to think and explore and simply made life satisfying.

Gratitude is due to my wife, who encouraged me and provided greater help than imaginable to finish this degree, and my parents, who counseled and advised me along the path and listened to my concerns and struggles. My appreciation extends to my entire family, who perpetually showed interest and concern in my work and made me feel loved, even though I know they didn't understand what I was trying to accomplish nor could I explain it to them very well.

Josh Templin

Contents

List of Tables

List of Figures

Acronyms

3G	3rd generation
4G	4th generation
ADC	analog-digital converter
ALU	arithmetic/logic unit
AM	amplitude modulation
ASIC	application-specific integrated circuit
CCM	configurable computing machine
CDFG	control- and data-flow graph
CDMA	code division multiple access
CPU	central processing unit
DAC	digital-analog converter
DLP	data-level parallelism
DRM	data router and manager
DSP	digital signal processor, digital signal processing
DSP48E	digital signal processing 48 extension
EDK	embedded development kit
EMAC	ethernet media-access controller
EPIC	external interface and control
FM	frequency modulation
FPFA	field-programmable function array
FPGA	field-programmable gate array
FSL	Fast simplex link
FU	functional unit
GPP	general purpose processor
GPS	Global positioning system
GSM	Global system for mobile communications or group spécial mobile

HDL	hardware description language
HWICAP	hardware ICAP
ICAP	Internal configuration access port
ILP	instruction-level parallelism
IO	input-output
LED	light-emitting diode
LUT	look-up table
MAC	multiply-accumulate
Mbps	megabits per second
MIMO	multiple-input, multiple-output
MIPS	Microprocessor without interlocked pipeline stages
MSPS	million samples per second
MODEM	modulator/demodulator
OPB	On-chip peripheral bus
PAR	place and route
PDR	partial, dynamic reconfiguration
PE	processing element
PR	partial reconfiguration
QPSK	quadrature phase-shift keying
RAM	random-access memory
RaPiD	Reconfigurable pipelined datapath
RAW	Reconfigurable architecture workstation
RCA	Radio corporation of America
ROM	read-only memory
RPE	reconfigurable processing element
RSFMD	reconfigurable SFMD
SDR	software-defined radio
SFMD	single-function, multiple-data
SIMD	single-instruction, multiple-data
SOC	system-on-a-chip

SRAM	static RAM
SRIF	static-reconfigurable interface
TI	Texas Instruments
TLP	thread-level parallelism or task-level parallelism
USB	Universal serial bus
VHDL	VHSIC HDL
VHSIC	very-high-speed integrated circuit
VLSI	very-large-scale integration
XUP	Xilinx University Program

Chapter 1

Introduction

1.1 Motivation

Software-defined radios (SDRs) [1][1] are becoming an ever-increasingly popular platform for signal processing implementations [2, 3]. Unlike hardware radios, which build systems from primitive hardware elements, SDRs typically build systems from programmable processors like general purpose processors (GPPs) and digital signal processors (DSPs), instruction-based processors that execute software algorithms. SDRs are well liked because of their ease of development and their reconfigurability. Hardware radios, on the other hand, are statically designed for a specific use, have little adaptability, and require a new hardware design for each new application.

Most modern wired and wireless communication devices incorporate software performing radio algorithms, from cellular phones to Wi-Fi routers. These devices generally have microprocessors performing critical modulator/demodulator (MODEM) functionality in software. Most of these systems' radios are little beyond a software implemented radio, that is, a radio implemented in software with little or no run-time re-programmability. True software-defined radios (SDRs) go beyond a simple software implementation; they entail the ability to configure and manipulate the performance of the radio to a significant degree at run time. In addition to manipulating MODEM parameters such as carrier frequency, error coding, filter types, etc, SDRs are capable of changing MODEM standards and schemes altogether. A true SDR forms a suitable platform for a cognitive radio.

[1]The term software-defined radio or software radio was first coined by J. Mitola in a 1992 IEEE publication, although software radios had existed for decades prior. Since then the term's usage has become muddy, often blurring the line with cognitive radio, which Mitola also introduces. In this paper, "software radio" or "software-implemented radio" is used to define a basic software implementation of a radio while "software-defined radio" is used to identify a software radio with substantial run-time re-programmability. "Cognitive radio" is used to distinguish software-defined radios that possess some degree of intelligence and autonomous optimization capabilities.

From a consumer's perspective, SDRs have the ability to compact all of their wireless devices into a single receiver/hand-held unit. Consumers often have dozens of software-implemented radios, each performing a specific function and only occasionally needed. Examples of such devices are: GPSes, provider-specific national cellular phones, international cellular phones, satellite phones, frequency modulation (FM) radios, amplitude modulation (AM) radios, HAM radios, XM radios, mobile hotspots, and wireless access points. Generally each device is packaged separately, creating a large assortment of radios that must be transported to provide access to all of the desired radio services. Ideally, all of these devices would be combined into a single package [1]. A consumer demands an SDR that is widely versatile while simultaneously handling multiple modes/uses.

With the progression from 3rd generation (3G) to 4th generation (4G) wireless cellular networks and the ever changing standards and technologies of wireless communication, a consumer SDR allows upgrade-ability and increases future-proofing. A consumer desires each purchase to have a long lifetime. Almost every cellular phone, for example, is optimized and tuned to a particular standard. For example, there are two main cellular phone communication standards: code division multiple access (CDMA) and Global System for Mobile Communications (GSM). While some phones support both, typically the process of changing standards requires a hardware upgrade (the purchase of a different device), not a software upgrade. International travelers have frequently been frustrated by this incompatibility. Consumer demand exists for an SDR with feature upgrades and some degree of future-proofing. Ultimately what is desired is an any-mode, any-band, run-time reconfigurable software-defined radio.

Such an SDR has never before been realized. One of the challenges hindering the development of this commercial SDR is the lack of proper processing power. As modern, complex, high-data-rate modulations arise, researchers find themselves turning to processors other than SDRs and DSPs [4]. Consequently, SDRs targeting high-data-rate modulations have shifted hardware into the FPGA realm, which have proven more efficient computers for streaming data applications [5, 6]. However, tweaking an FPGA configuration is not as

simple as changing the software of a GPP or DSP.

The emergence of configurable computing machines (CCMs) indicate the trend to solve the SDR problem by abandoning the GPP and DSP and creating custom hardware architectures for signal processing/communication algorithms. These CCMs implement processing architectures that are configurable, adaptable for improved performance for a subset of applications. An excellent overview of CCM efforts targeting SDRs has already been presented [7]. Some of these CCMs are implemented on custom application-specific integraded circuit (ASIC) hardware, while others are prototyped on or targeting FPGAs. While custom ASIC CCMs can easily outperform an FPGA, their utilization requires the fabrication of a new (and generally expensive) silicon chip and development of a suitable evaluation platform. Custom ASIC CCMs also generally lack sufficient software support, which renders their use very difficult. FPGA-based CCMs are able to leverage the pre-designed evaluation platforms and the substantial software support of the FPGA vendors, saving decades of work and millions of dollars. Also, as FPGA technology continues to progress, their performance margin over ASIC implemented designs continues to narrow, lending the FPGA as a suitable target platform and not just as a prototyping platform.

FPGAs are becoming ever more popular as the hardware of choice for SDR CCMs. Their ability to capitalize on thread-, instruction-, and data-level parallelism have allowed them to achieve incredibly high performance in the realm of stream processing. But developing for FPGAs has been infamously difficult in the past. However, modern open-source collections of cores [8] along with improving C/Matlab/C++-to-hardware-description-language (HDL) compilers [9–11] have drastically improved developer efficiency and reduced development time by eliminating a bulk of the required user-generated HDL. Reconfiguring FPGAs allows for platform reusability so that a single FPGA platform can be used for a large set of drastically different algorithms. Thus, an FPGA configuration can be modified and updated as required by future needs and applications, achieving any-band and any-mode within the confines of the FPGA size and speed. This also achieves run-time reconfigurability but only for mono-mode use.

Ultimately, FPGAs lack the run-time flexibility of GPPs and DSPs. Simultaneous multi-mode execution on a GPP is simply accomplished with multiple threads but at the cost of deteriorated performance. Simultaneous multi-mode execution in an FPGA is more difficult and cannot be feasibly accomplished by standard reconfiguration (this would require a collection of configurations for every desired combination of modes). Fortunately, some FPGAs have the ability to dynamically reprogram only a portion of the FPGA with new configurations. The process of dynamically reprogramming portions of the FPGA is called partial, dynamic reconfiguration (PDR), or more simply partial reconfiguration (PR) [12]. PR allows for only a portion of the FPGA to be reconfigured while the rest of the FPGA remains active and continues with its current configuration. While this PDR can be slow (on the order of 1-100 ms, depending on the device, speed, and bit-stream size [13–15]), it does yield the ability to adapt the FPGA to a simultaneous multi-mode application. In addition to PDR, each FPGA-based core can be SDR conscious, designed specifically to allow programming of the core's functionality and parameters. This provides run-time reconfigurability via a much faster software reprogramming. When possible, utilizing this software reprogramming can yield instantaneous reconfiguration and, when combined with the more extensive capabilities of PDR, can yield an FPGA-based stream processing framework that is both computationally efficient and greatly adaptable.

1.2 Software-Defined Radio Architectures

As the processing needs of software defined radio and cognitive radio exceed the capabilities of GPPs and DSPs, alternative processors need to be designed. SDR poses unique requirements to the design of a CCM. The class of algorithms found in SDR is somewhat narrow, but broad enough that a single functional unit pipeline is not practical (one cannot build every SDR algorithm from a practical-sized collection of functional units with programmable interconnects). Typically, in SDR only a few tasks need to simultaneously execute, (for instance, running a Global Positioning System (GPS) demodulation and HD radio demodulation at the same time).

In light of these unique requirements, much research has been performed to identify

optimal architectures for SDR CCMs. A sampling of the more well-known architectures is presented in this section. These architectures can be roughly divided into three categories: instruction-based processors, hardware co-processors, and hardware processors. Figure 1.1 visualizes the extensibility vs. performance trade-offs for each type of CCM. Each CCM is able to nick the desired realm of SDR in extensibility or performance, but none are currently able to offer satisfactory coverage.

1.2.1 Instruction-Based Processors

Instruction-based processors are those processors which operate on a software program consisting of a sequence of instructions coupled with data operands. GPPs and DSPs fall into this category of processor. When used as a CCM architecture, these processors generally have very reduced, optimized instruction sets targeting a specific application. Instruction-based processor CCMs often seek to improve performance via instruction-level parallelism (ILP) by implementing a very-long instruction word (VLIW) architecture as well as to accomplish task-level- parallelism (TLP) by networking an array of processors with a configurable array.

The most popular instruction-based processor CCM targetting SDR is the Reconfigurable Architecture Workstation (RAW) [16]. RAW consists of a 2D array of processor tiles, each tile having its own instruction/data memory and registers, interconnected by programmable switches, also having their own program memories. One of RAW's main performance advantages over other multi-core processor architectures is its incorporation of a block of configurable logic for optimized execution of commonly occurring routines. The field-programmable functional array (FPFA) [17] is another well-known architecture. Its 2D array of processor tiles with localized memories are optimized for streaming data, being programmed with configurations and not by instructions.

The advantage of instruction-based processors is their support for a more general class of applications and their ability to switch to a wildly different class of applications with little to no overhead. The disadvantages of instruction-based processors for SDR are twofold: 1. They are always limited by a discrete instruction set which restricts the amount of

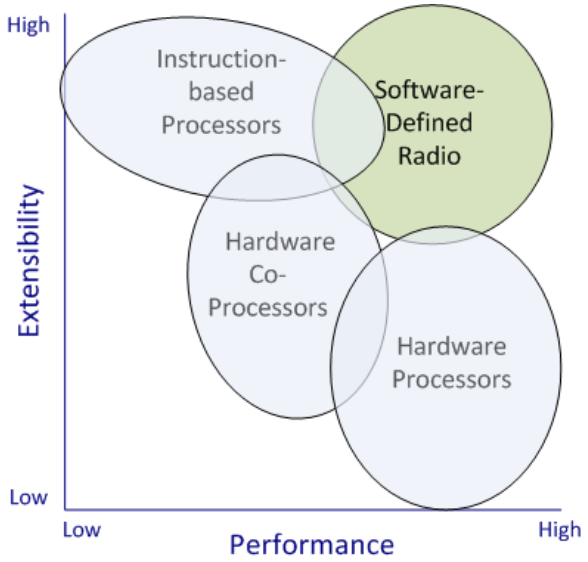

Fig. 1.1: Extensibility vs. performance for each CCM category. The desired realm of SDR is both high-performance and high-extensibility.

doable work each cycle; and 2. They often suffer from low computation power per circuit size because of the need to have a static, general purpose arithmetic/logic unit (ALU) for which only a few functionals are active during a given cycle. Hence, instruction-based processors have a comfortable niche in applications that require quickly or very frequently changing tasks. For applications where only a single or a very small collection of tasks need to be executed for long periods of time, a more targeted hardware implementation can always perform better. SDR uses generally fall into the latter.

1.2.2 Hardware Co-Processors

In an attempt to keep the general nature of instruction-based processors and the advantage of optimized hardware, many SDR CCMs couple a DSP or GPP and a custom hardware acceleration fabric together into a single processor. These hardware co-processor CCMs usually rely on the presence of a master, instruction-based processor which manages the configuration of the hardware acceleration fabric. A key attribute of hardware co-processor systems is that algorithms are partitioned across both the instruction-based processor and the hardware accelerator. Typically, the presence of an instruction-based

processor for synchronization and control while the hardware fabric performs the entire algorithmic computation does not constitute a hardware co-processor system.

Garp [18] is one such architecture that contains a single MIPS-II processor with its own instruction memory and sharing a data memory with a single, custom FPGA. The instruction-based processor is able to populate the FPGA with a configuration to accelerate a portion of the algorithm, like an inner-loop. Such hardware co-processor systems present an elegantly simple solution to algorithm acceleration. However, the continual passing of data to/from the accelerator can significantly hinder the system's real-time performance, which is critical in SDR applications. If the application is restricted to the hardware acceleration fabric then the system is no longer a hardware co-processor system and the cost in area and power of a separate, full-fledged instruction-based processor is not merited.

1.2.3 Hardware Processors

Hardware processors offer the most potential for meeting the processing power needs of SDR. The challenge of designing an effective SDR hardware processor is being ASIC enough to meet the processing requirements of all of the algorithms yet GPP enough to meet the flexibility introduced by SDR. Stallion [19] is a novel hardware solution consisting of a 2D mesh of signal processing oriented functional units (FUs). The FUs are statically implemented, reconfiguration is achieved by altering the configurations of the FUs and the routing of data through the mesh via wormhole run-time reconfiguration [20]. RaPiD [21] is a clever cascade of ALUs, random-access memories (RAM)s, multipliers, and registers interconnected to a collection of buses which allows rapid, streaming reconfiguration of custom pipelines. RSFMD [22] is a recent architecture which implements a collection of reconfigurable processing elements (RPEs) which can be loaded at run-time with optimized logic for a particular application. Each RPE is loaded with an identical configuration and different data is routed to each RPE enabling large parallel data processes or single-function, multiple-data (SFMD).

Hardware processors, such as those presented, are able to achieve large performance gains over other types of processors for a particular class of application and have even re-

tained a significant amount of flexibility, but are still sub-optimal. In the case of Stallion and RaPiD, the architectures are statically implemented and, while general and optimized for a class of applications, non-optimal for a particular application. These architectures implement all reconfiguration as "soft" reconfiguration (run-time programming of pre-designed functionality, no hardware design alteration). The RSFMD implements all reconfiguration as "hard" reconfiguration (run-time modification of processing hardware, no adaptability of a hardware configuration to a similar but slightly different application). RSFMD also allows only a single hardware configuration at a time, requiring custom-built hardware configurations for every application or combination of applications, regardless of existing designs, each of which configuration must be stored locally for run-time access.

In hardware processors, the lack of a "hard" reconfiguration requires that all applications map onto a pre-defined, sub-optimal architecture. The lack of a "soft" reconfiguration requires that a new, time-consuming hard reconfiguration be performed for every application, even those that use the same architecture but with different parameters or in a slightly different configuration.

1.3 Contributions

The goal of this research is to provide better coverage of the SDR region represented in figure 1.1. The approach that has been chosen is to improve the extensibility of hardware processors. For this research, a new hardware processor was developed that utilizes optimized logic for each application while maintaining great flexibility in the class of applications executable on the processor. The proposed architecture improves the hardware processor group's extensibility, as shown in figure 1.2, providing a better platform for realizing a larger range of both high-performance and highly-extensible applications. The scope of this research is limited to the deduction and definition of a reconfigurable computer architecture optimized for software-defined radio applications as well as a demonstration implementation sufficient to prove the architectural concept. Important implementation specifications are presented when applicable, but detailed specifications on the usage and

performance of technology-specific partial reconfiguration and bitstream relocation (including partial reconfiguration timing) are beyond the scope of this work.

The key novelties of this architecture are: an array of programmable sockets coupled with partial, run-time reconfiguration to achieve, to a large degree, the flexibility of instruction-based processors; a high-speed, programmable data routing and socket interconnect; and a high-speed, programmable socket interface. These contributions together allow arbitrary combinations of applications to simultaneously execute at the discretion of a run-time controller. Run-time reconfiguration in the processor is achieved via both soft reconfiguration and hard reconfiguration. The soft reconfiguration allows the controller to capitalize on reconfigurability built-into the individual processing elements (PEs) and is implemented as part of the processor. Hard reconfiguration allows a limitless possibility of compatible PEs to be "plugged in" and utilized as needed and is realized by the underlying technology. The developed architecture is described in a hardware descriptive language and is portable to different platforms. The design and implementation target Xilinx's Partial Reconfiguration [12] as the hard reconfiguration technology, though the architecture could easily be ported to other FPGAs featuring a run-time partial reconfiguration technology and even other CCMs, such as hardware/software co-processor systems.

1.4 Thesis Organization

The remainder of the document presents the results of the performed research. Chapter 2 describes the architecture of the processor and discusses its capabilities. Chapter 3 discusses a test case implementation on a Xilinx FPGA and presents the results of its performance. Chapter 4 concludes the discussion with a summary and a discussion of future possibilities. Appendix A gives an overview of necessary background topics on reconfigurable computing utilized by the processor and its implementation. Appendix B provides additional details on the system as designed using Xilinx EDK. Appendix C contains all of the source code developed as part of this effort.

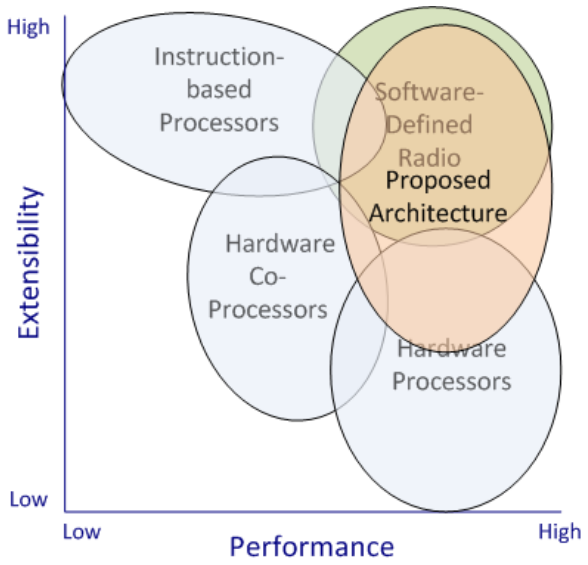

Fig. 1.2: Extensibility vs. performance of the proposed architecture. The desired realm of SDR is both high-performance and high-extensibility and experiences greater coverage.

Chapter 2

Architecture

2.1 Design Goals

The goal of this architecture is to provide a flexible processing environment for radio applications. While the architecture should remain as general as possible so as to be useful for a larger class of applications, many design decisions and optimizations need to specifically target radio applications. The following considerations are important for specifying the design details of varying parts of the architecture from a radio applications perspective.

2.1.1 Streaming Data

Communication algorithms are streaming by nature. Instead of executing only occasionally when certain conditions are met they generally execute continuously as periodic data arrives. A communication algorithm represents a function to be executed every time a new data sample is available. Hence, each algorithm follows a single-function, multiple-data (SFMD) [22] model. These functions are much coarser grained than single-instruction, multiple-data (SIMD) models. In an instruction-based processor, a function may consist of hundreds or thousands of instructions. Each instruction delays the beginning of processing on a subsequent data sample, only to execute the same instruction again. Because of its time-periodic nature, streaming lends itself better to parallelized, pipelined hardware implementations than sequential, instruction-based implementations. Thus, the architecture should be optimized for streaming data by being well-pipelined.

2.1.2 Data Widths

In instruction-based processors, data operands are generally a fixed width, such as 32-bits, since registers are statically sized. Conversely, signal processing algorithms generally

require smaller data widths and have varying requirements for different data. For example, a lot of resources are wasted mapping 12-bit samples produced by an analog-to-digital converter (ADC) onto 32-bit arithmetic operations. Likewise, a function might take 12-bit inputs and generate 23-bit outputs or a single-bit output. Throughout the function processing chain, data widths may grow or shrink based on the operations being performed and the data ranges of interest. Also, configuration parameters of a function may vary from a single bit to 64 or more bits. Hardware implementations are free to use whatever precision desired for an implementation. However, data width requirements vary from implementation to implementation and this variability is important to realizing optimized implementations. In order to not unnecessarily restrict the range of possible algorithms and the optimality of their implementations, the architecture should allow for arbitrary data width ranges.

2.1.3 Programmability

Many communication algorithms consist of a common set of signal processing operations. Often two algorithms will have a large portion of common operations, varying only in their parameters. Filtering is one of the most fundamental and common operations performed in communications and often the filters are identical in function and architecture and vary only in their coefficients. Taking advantage of these commonalities means that switching between two similar algorithms does not require hard reconfiguration of the common parts, but only a soft reconfiguration, which can be substantially faster, depending on the underlying technology. Thus these operations can be defined as generically as possible and programmed at run-time to extend their application, achieving a soft reconfiguration. Varying operations will provide varying programmability with each programming port having different data width requirements. The architecture should be considerate and aware of and allow this programmability.

2.1.4 Speed

Communications systems are real-time systems requiring the ability to begin another iteration within a certain period of time, (generally determined by the period of the data

samples). In instruction-based processors this means the function must begin and complete in the alloted amount of time. Real-time requirements can cause substantial problems in instruction-based processors and algorithms must be carefully designed and even more carefully analyzed to ensure that real-time constraints are satisfied. Mixing and matching functions at run-time only further complicates guaranteeing and meeting real-time performance. When hardware implementations closely emulate software implementations, hardware implementations can also suffer from this problem. However, in hardware, real-time issues can be alleviated or eliminated by properly pipelining and adjusting the clock frequency of the implementation.

Generally, a trade-off exists between the area consumed by a pipeline and the clock frequency needed to meet real-time needs. When a pipeline contains replicated operations, those operations can be time-shared, but the clock frequency needs to be increased proportionally to maintain real-time performance. Similarly, pipeline stages may need to become larger, simultaneously reducing clock speed, in order to reduce overall circuit size. Also, the same algorithm may run at different frequencies to achieve different performance or data rates. Each of these scenarios demonstrates the varying needs of potential algorithms' operation speeds and even the ability to adjust these speeds via clock frequency at run-time. The architecture should provide a mechanism for varying functional pipelines to execute at varying speeds.

2.2 Overview

In light of the design goals, a novel hardware processor architecture was created which allows arbitrary communication algorithms to be executed at hardware efficiency while maintaining a great degree of flexibility. The architecture consists of a system controller and a 1D array of sockets hanging from a configurable signal routing block. Each socket can be populated at run-time with any of a variety of processing elements (PEs). Figure 2.1 presents a high-level view of the processor architecture. Many architecture parameters are left unspecified to allow for proper adaptation to different underlying technologies (ie. a Xilinx FPGA vs other reconfigurable logic technologies, commercial and academic, present

and future). Thus, the architecture can be realized on and a library of PEs developed for any reconfigurable logic technology that supports a sufficient partial, dynamic reconfiguration.

The system is dichotomized into static logic and reconfigurable logic. The static portion of the architecture is specified and defined at the time of porting the architecture onto a hardware technology. When implementing the architecture on a target, the architecture parameters, such as number (and location) of sockets, number (and location) of signal sources and sinks, and data widths of internal buses, must be concretized. The reconfigurable portion, aside from the socket boundaries and locations, is specified completely at run-time. That is, after the static portion of the architecture has been established, then, at run-time, the controller populates the sockets with meaningful PEs from the library of available PEs.

The PEs are coarser than a multiply-accumulate (MAC) or even a filter and are finer than a complete, complex algorithm. Each PE can implement either an entire algorithm, if the algorithm is simple enough, or a collection of common processing operations, such as timing and phase error correction, or algorithm specific logic. The size of the sockets is implementation dependent, but optimally they achieve balance between fine and coarse granularity. Fine grained sockets see great reuse and little waste between different configurations but result in large overhead both within the socket and in the system. Coarse grained sockets accomplish more work with less wasted overhead (because of the reduced number of sockets) but are more likely to see increased waste between configurations (not all parts of the PE being used for every application). Ultimately the architecture leans towards larger sockets in order to reduce system complexity, accomplish more work, and improve PE relocatability. An alternate version of this architecture features a more sophisticated 2D array of smaller processing elements, useful if the underlying technology can support the finer-grained PE relocation. The former architecture was pursued and is presented here because of the coarser relocation requirements of Xilinx FPGAs, which is the technology of focus for this implementation.

2.3 System Controller

The system controller manages the entire system. It is responsible for interpreting

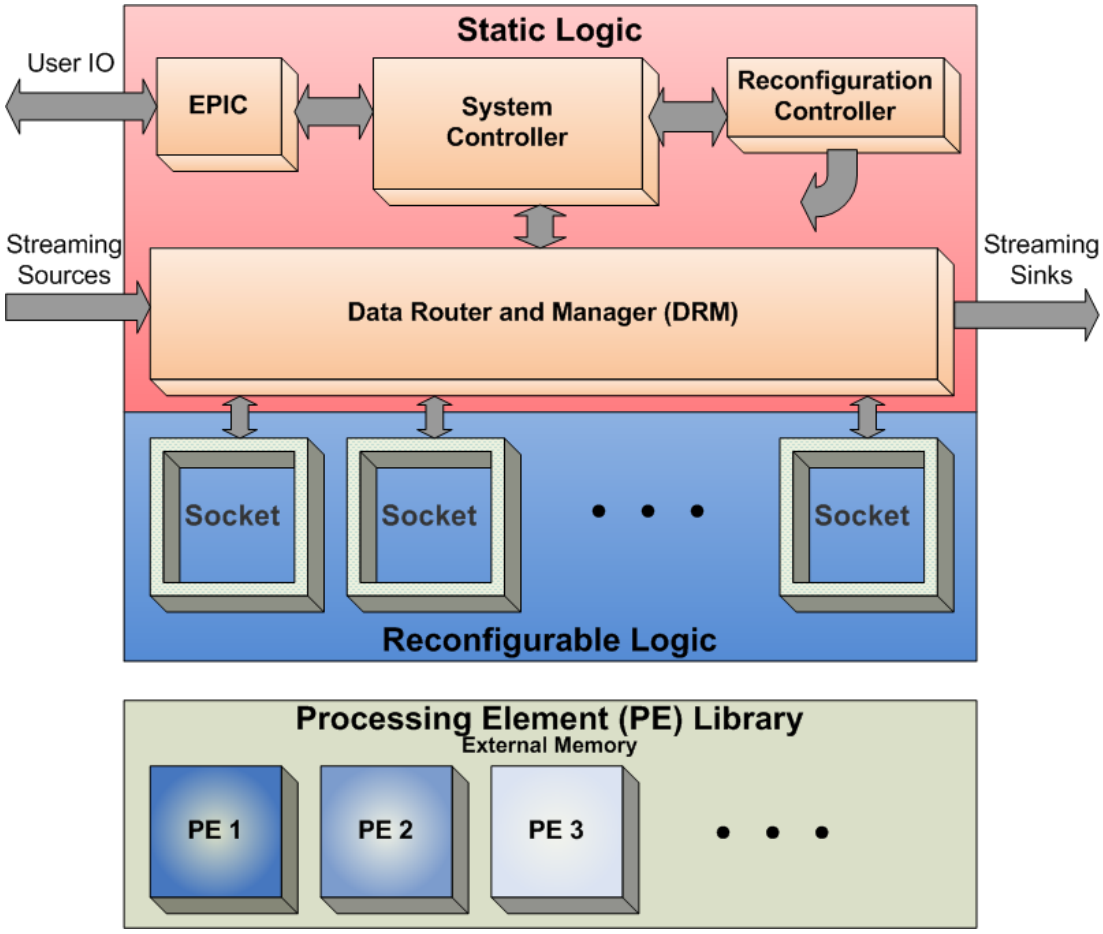

Fig. 2.1: The architecture - a top-level view. The purposed architecture contains a 1D array of sockets with configurable interconnects and sources/sinks. The sockets can be reconfigured with PEs at run-time as specified by a user or controller.

user instructions into system configurations and programming the necessary components accordingly. Socket reconfigurations are initiated by the system controller. All virtualization for over-partitioned sockets is managed by the system controller. The system controller interfaces with all of the other static logic modules but does not have any communication directly with any reconfigurable logic. The only communication between the system controller and reconfigurable logic is communicating programming addresses and data which get intercepted and manipulated by a reconfigurable logic interface. Because of the varied and sporadic duties of the system controller it is well-suited for implementation in a simple, embedded instruction-based processor. However, even when implemented in an

instruction-based processor, the architecture does not represent a hardware/software co-processor system because the system controller does not execute any algorithmic software; it executes strictly control logic software.

Because of the complexity of the system controller logic, it is desirable to separate it from real-time sensitive parts of the architecture. Hence the routing of data to/from the socket array is implemented in programmable hardware instead of being handled by the system controller. Because of separation of concern, the system controller adopts a laissez-faire interaction with the socket array, only communicating when the configuration needs to be altered. Once configured by the system controller, the socket array and all of the encompassing signal routing are completely self-sustaining.

2.4 Reconfiguration Controller

Technology dependent partial reconfiguration is abstracted into the reconfiguration controller. The system controller issues reconfiguration commands to the reconfiguration controller which translates the commands into platform-specific actions. This also frees the system controller of the responsibilities of managing reconfiguration. While the system controller knows about the contents of the PE library, only the reconfiguration controller interfaces with the PE library, likely stored in external, persistent memory. The reconfiguration controller also gives status updates to the system controller when requested.

Communication of addresses between the system controller and reconfiguration controller is virtualized to simplify porting the architecture onto different technologies. The system controller identifies both PEs and destination sockets to the reconfiguration controller using virtual addresses. The reconfiguration controller then translates the virtual addresses into physical addresses for communicating with the various physical interfaces. For the PE library, this address mapping takes a unique PE identification number input and generates a beginning memory address for the PE's storage location within the library. For the sockets, this address mapping takes a unique socket identification number input and generates technology-specific address for the socket's configuration data. In Xilinx FPGAs, the addresses are the horizontal and vertical locations of each configuration frame within

the socket's boundary.

The reconfiguration controller also manages relocation of PEs from one socket to another. For Xilinx FPGAs, this means modifying the addresses of the frames in the PE bitstream and performing any other relocation required bitstream modifications. This relocation logic may be implemented in hardware or in software. When implemented in software it makes sense to incorporate the relocation portion of the controller with the system controller on the GPP.

2.5 External Programming Interface and Controller

Interfacing the system with the user is the responsibility of the external programming interface and controller (EPIC). This module's sole functionality is to bridge the user's world with the system's internal world. Thus, the EPIC consists mostly of physical interfaces and their drivers. User IO may come in the form of a serial port, push buttons and LEDs, a touchscreen display, a keyboard, etc. The EPIC implements these system-specific interfaces so as to abstract them from the system controller. Thus, the EPIC delivers to the system controller a standard set of messages directing the radio how to perform. In many implementations, the EPIC module will be implemented as peripherals hanging off of a bus of the system controller's GPP.

2.6 Data Router and Manager

The data router and manager (DRM) is the most interesting component in the static portion of the architecture and is diagrammed in figure 2.2. Its responsibility is mapping streaming data between the sockets and the IO signal sources as well as delivering programming commands from the system controller to the socket array. Input sources may be ADCs connected to antennas or digital sources such as ethernet or USB or another processor running the same architecture. Outputs may be to DACs connected to RCA or headphone jacks or digital sinks such as ethernet or USB or another processor.

In the current implementation each socket is able to take its data from only a single source and output its results to a single sink. Data from any external input can be routed

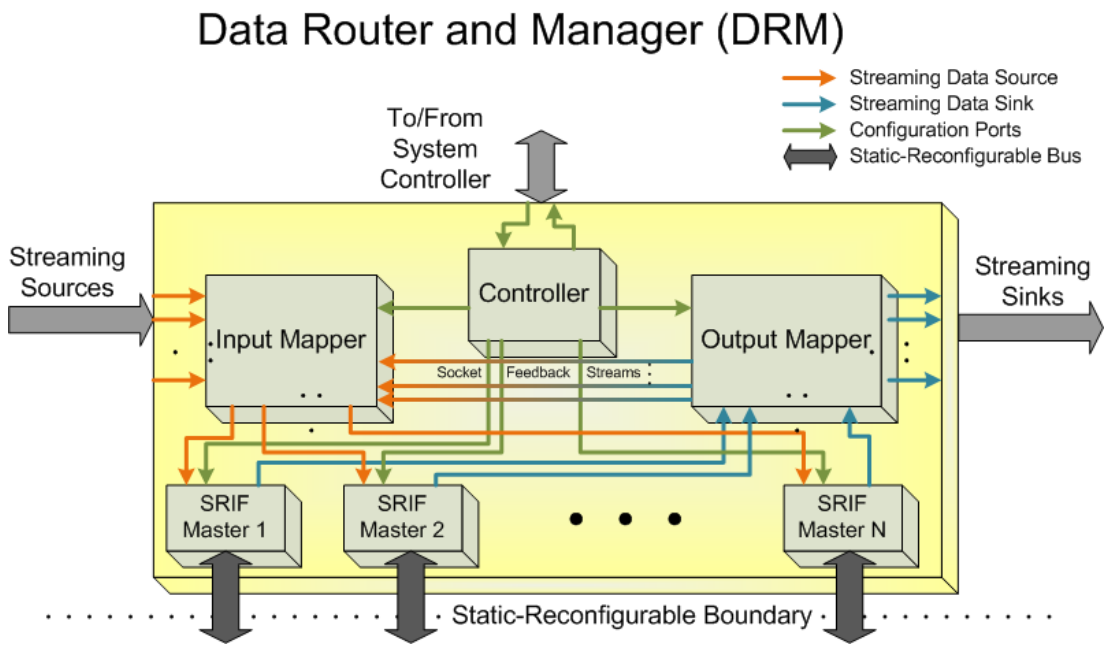

Fig. 2.2: Data router and manager. The DRM consists of programmable routing logic to interface the socket array with available streaming data sources. For each socket there is an interface master that the PE connects to.

to any or all of the sockets. Data to any external output can come from any single socket. Sockets can also route data to each other but only in a streaming simplex fashion and only to the nearest neighbor in a single direction. These limitations are imposed for the purpose of simplifying the configurable routing logic of the DRM and can be eased/eliminated in future revisions.

2.7 Static-Reconfigurable Interface

The most important and critical component of the entire system is the interface between the static region and the reconfigurable region (SRIF), shown in figure 2.3. This link needs to be flexible enough to meet the varying data width needs of streaming data interfacing with the IO as well as the streaming data flowing between sockets and also the programming parameters coming from the controller. Because of logic overhead in connecting signals between static logic and reconfigurable logic in many reconfigurable platforms, including Xilinx FPGA, signals crossing the boundary should be minimized to free up logic for use

by the PE. Thus the SRIF should be as serial and as fast as possible.

The SRIF consists of a static-side master and a reconfigurable-side slave. The master initiates programming the slave while both the master and the slave communicate data across the interface in full-duplex. Each side of the interface contains a disassembler, which takes in words and fragments them for sending across the interface, and an assembler, which receives fragments from the interface and assembles them into words. The size of the words and the fragments must be specified at design time because the logic exists in the static region. The slave side of the interface also contains programmable clock decimation logic for providing custom clock speeds to the PE.

Because of the general nature of the SRIF (it has no knowledge of the PE's function), it does not perform any interpretation or manipulation of data. The same SRIF is used for every PE so it must be capable of meeting the needs of PEs in general. Thus, the SRIF is completely agnostic of any particular radio functionality and merely provides a common interface for pushing data from multiple sources and various widths across a simplistic, small-footprint, serialized interface. Interpreting the data and pumping multiple inputs/outputs through the interface is the responsibility of the source/destination logic.

2.8 Sockets

Sockets are the workhorse of the architecture. All radio functionality is implemented in the sockets. Sockets can be populated with any PE at any time. PEs are implementations of customized/programmable radio logic that are compatible with the available sockets. The architecture contains an array of sockets and, based on the current needs of the user, any, all or none of the sockets may be utilized and populated with PEs. At any given time a PE from the library of available PEs may be found in none, one, or multiple sockets. To make this possible, and as required by the requirements of partial reconfiguration on many platforms, the sockets are all constructed of regions of identical shape and resource structure in the underlying reconfigurable fabric. If the sockets were of varying sizes or constitution then relocation of PEs across sockets may not be possible and multiple versions of a given PE may be required in the PE library. Keeping sockets equal allows for the use of relocation

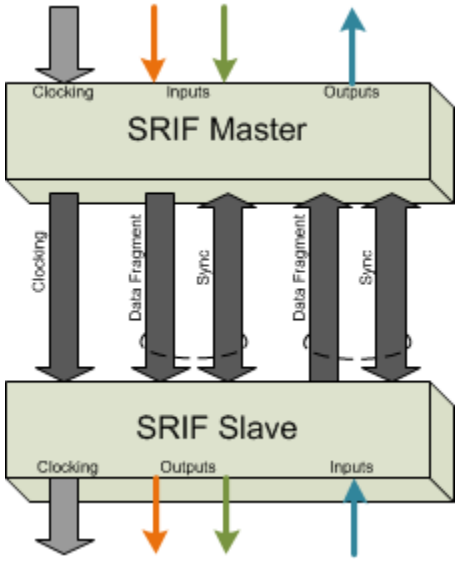

Fig. 2.3: Static reconfigurable interface. The master SRIF multiplexes different-sized data from several sources through a small interface to be reassembled on the slave side. It also receives disassembled data from the slave side and reassembles it for delivery.

and reduces the development time and memory requirements of the PE library.

Because all PEs communicate identically with the architecture through the SRIF, all PEs have at least some common logic. This portion of the logic is pre-placed-and-routed into a template to ensure common footprints and routings to the static logic as well as the meeting of timing constraints. Developers then use this template and amend it with custom logic to form a relocatable PE that becomes added to the PE library, as seen in figure 2.4.

The slave SRIF contains streaming data ports and configuration ports. The configuration ports consist of an address port and a data port. The configuration data then is processed by custom logic which maps the configuration data to the proper port of the PE logic based on the address. In the case that the configuration words transmitted across the SRIF are larger than the SRIF port word size, this logic will be aware of this and properly reassemble the words to construct the full-length configuration parameter. Thus, between the fragmentation performed by the SRIF and the fragmentation performed by the user logic, configuration parameters of any bit-width can be communicated while maintaining a common, low-footprint interface.

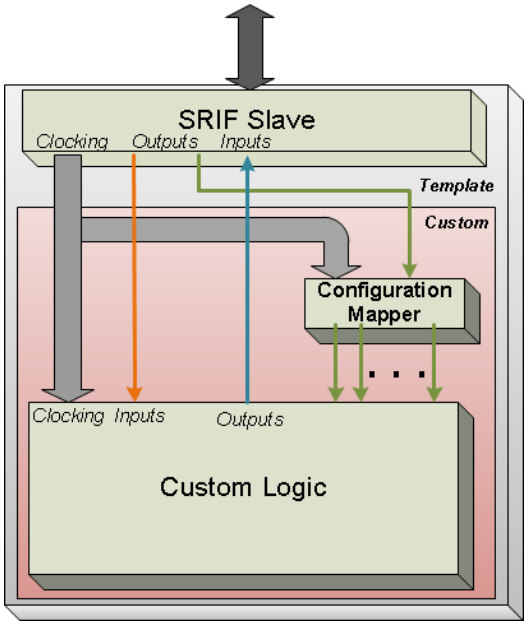

Fig. 2.4: Processing element structure. Each processing element (PE) for a given architecture implementation must have a common structure. Fundamentally this common portion consists entirely of the slave SRIF. However, other templates might include some signal pre-processing operations such as downsampling.

Chapter 3

Implementation and Results

3.1 Virtex-5 Implementation

To demonstrate the feasibility of the proposed architecture, it was implemented on a Xilinx Virtex-5 LX110T FPGA (xc5vlx110t), found on the Virtex-5 Xilinx University Program (XUP) Evaluation Platform. The system was described using the VHDL language. Once the system was developed, a small collection of PEs were developed to demonstrate run-time swapping of PEs.

3.1.1 Architecture

Before implementation on a particular platform can begin, the system parameters must be determined. The most critical of these parameters is the number, size, and placement of sockets in the underlying fabric. These parameters are all determined based on the structure of the underlying fabric while the remainder of the parameters (such as IO) are fixed, in this case, by the available peripherals. The xc5vlx110t has the floorplan shown in figure 3.1.

Because of the frame-based reconfiguration approach of Xilinx, sockets should consume the full height of a clock row. In the xc5vlx110t part has is no repeating pattern of configurable logic blocks (CLBs), block RAMs (BRAMs), and DSP48Es within a given clock row. Consequently, a clock row cannot be divided into multiple, identical sockets.[1] Some logic is needed for the static logic portion of the architecture and for interfacing with the columns of hard cores and IO banks. Thus, some of the logic immediately surrounding these devices is reserved for the static logic, as well as one entire central clock row for the

[1]The typical modern techniques of bitstream relocation in Xilinx FPGAs require relocation among identical regions. This issue is discussed more in Appendix A.3.

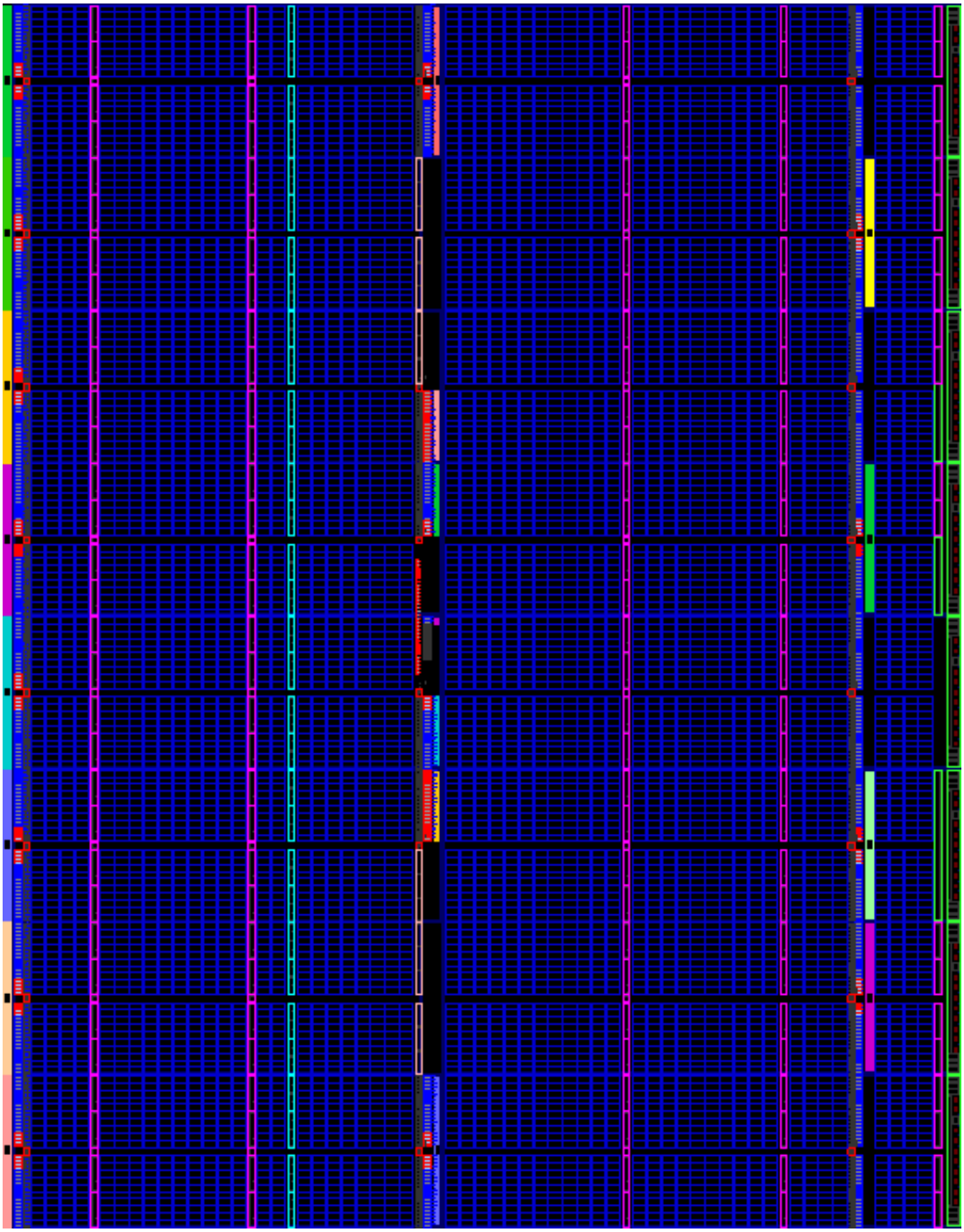

Fig. 3.1: Floorplan of the xc5vlx110t chip. The xc5vlx110t has CLBs divided into eight clocking rows (distinguished by the dividers in the rightmost column). Interspersed are five columns of BRAMs (pink), one column of DSP48Es (light blue), and four other columns of hard cores and IO banks.

purpose of horizontally bridging these four columns (and for interfacing with the Internal configuration access port (ICAP), which is located in the center of the FPGA).

Based on these restrictions, the architecture is able to contain seven large sockets. The size and location of these sockets are shown in figure 3.2. Each socket represents about 1/9th of the entire circuit, as reported in table 3.1. The static controller overhead consumes about 2/9ths of the logic while the remaining 7/9ths are available for optimized PE logic.

The DRM is a hardware module described in VHDL. Being the most performance critical part of the static logic, it is important to make this as fast as possible. A parameterizable DRM module was created which allows generic specification of the number of IO ports, number of sockets, and data widths. In this example the only data sources and sinks are canned data RAMs, but to create a realistic scenario the number of sources was set to three and the number of sinks set to four. Finally, specifying that this implementation contains seven sockets produced a fully-pipelined DRM entity preliminarily capable of running at 300 MHz.

The availability of the MicroBlaze processor for Xilinx FPGAs makes it a natural choice for implementing the system controller, the reconfiguration controller, and the EPIC. When an embedded instruction-based processor is chosen for implementing the controllers the architecture begins to take the form of a system-on-a-chip (SOC). In a custom silicon design, this could be realized as a single die with a small-scale instruction-based processor core coupled with one or more reconfigurable hardware cores.

Communication of the system controller with the DRM occurs via a custom peripheral attached to a Fast simplex link (FSL). This interface allows custom, asynchronous communication across the clock domains of the slower MicroBlaze and the faster DRM.

Table 3.1: Architecture resource distribution. Distribution of native FPGA resources to each socket and to the static logic.

Resources	Slices		BRAMs		DSPs	
	Socket	Static	Socket	Static	Socket	Static
Used	1960	3560	16	36	8	8
Available	17280		148		64	
Proportion	11.3%	20.1%	10.8%	24.4%	12.5%	12.5%

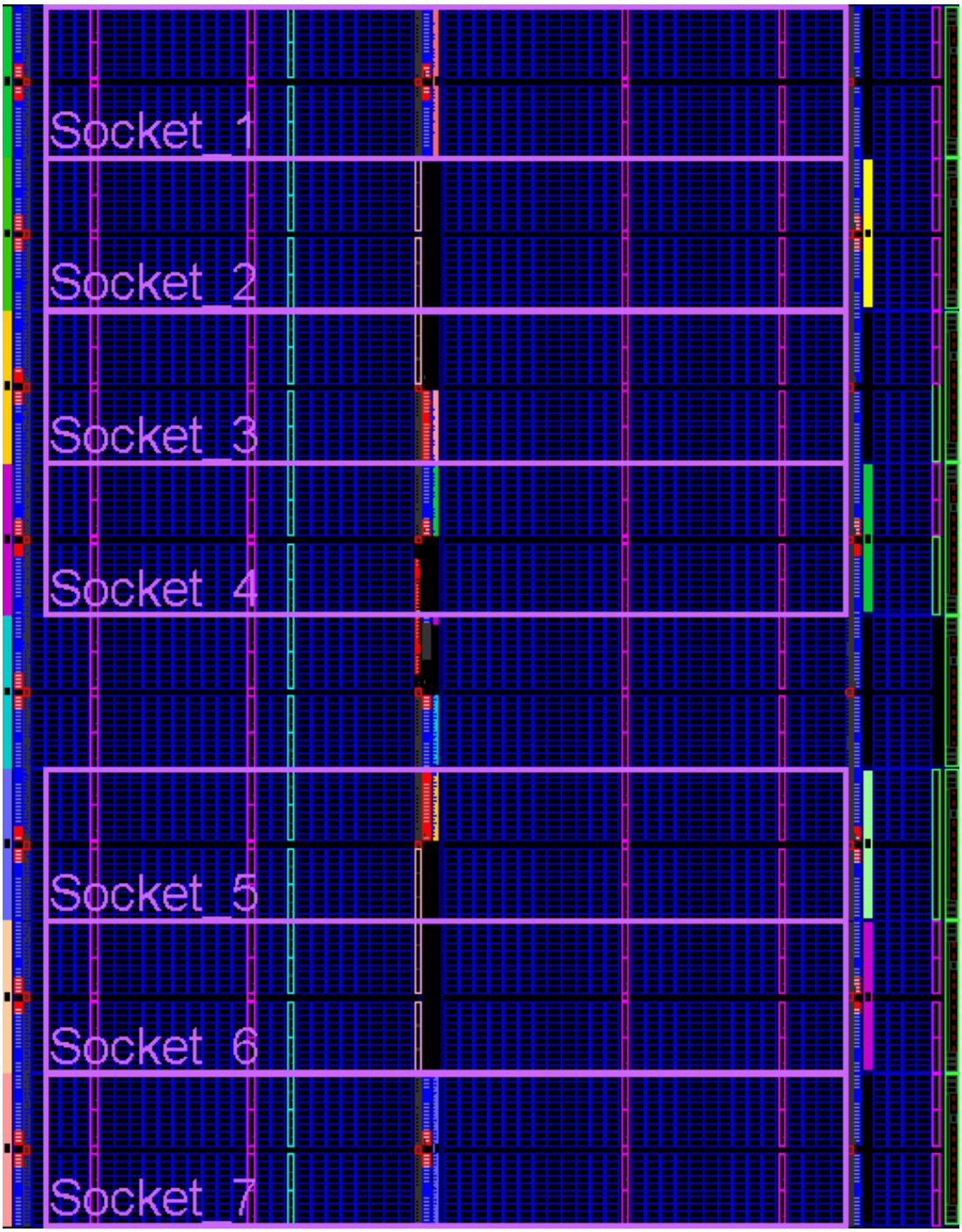

Fig. 3.2: Layout of the sockets. The xc5vlx110t fits seven large sockets divided by clock rows. Each socket consumes most of the clock row with a little logic reserved for use by the static-logic controller.

The Reconfiguration Controller is realized via a software program and the Hardware ICAP (HWICAP) interface provided by Xilinx for managing partial, dynamic reconfiguration. Xilinx offers the HWICAP as a core connecting to a MicroBlaze via the On-chip peripheral bus (OPB). EPIC is also realized as a partial software/hardware peripheral solution. Thus the single MicroBlaze hosts the software for all three controllers and manages the physical interfaces via hardware peripherals connected by buses.

MicroBlaze offers two different pipelines: a 3-stage pipeline targeting low area and a 5-stage pipeline targeting high performance [23]. Since these controllers are not performance critical the low-area pipeline was chosen. This produces a slower but smaller controller to ensure the maximal amount of logic available to the socket array.

Altogether the static-logic portion of the architecture requires the resources shown in table 3.2. The architecture runs off two clocks: the slower MicroBlaze clock and the faster socket array clock. In this implementation the MicroBlaze clock is 100 Mhz and the socket array clock was chosen as 300 Mhz.

3.1.2 Algorithms

The sample architecture implementation was tested using a typical QPSK system. The QPSK system consists of a modulator and a demodulator, as shown in figure 3.3. As in most communication systems, the modulator is lightweight compared with the demodulator, where most of the processing occurs. The process of demodulation entails a linear series of sample-based operations, ideally suited for a linear, streaming architecture such as this. The demodulation process contains some highly-regular, largely parallel, arithmetic functions,

Table 3.2: Architecture resource consumption. Resources required by the entire static-logic portion of the architecture including the MicroBlaze, its peripherals, and the DRM and master socket interfaces.

Resources	Slices	BRAMs	DSPs
Controller	1720	3	3
DRM	882	0	0
Total	2602	3	3
Available	17280	148	64
Proportion	15.1%	2.1%	4.7%

such as filters, as well as logical and branching functions, as found in the timing and phase correction loops.

Each algorithm was created in VHDL for implementation on the architecture and in C for benchmarking on an Intel central processing unit (CPU) and a Texas Instruments (TI) DSP. Both implementations were kept functionally similar, including integrating similar degrees of run-time re-programmability.

3.2 Performance

Performance of the architecture is bound by the socket array's clock and the size of the fragments. Ultimately the architecture's performance is limited by the PE element implementation. If the inputs to a PE are n SRIF fragments large then samples are only available at most every $^1/_n$ cycles of the 300 MHz array clock. Similarly, if a PE is designed to run at a maximum clock frequency of 50 MHz then its performance will be limited to 50 million samples per second (MSPS). Thus, in this implementation, the maximum sample rate of any PE in the array is 300 MSPS which is possible when the PE is designed to run at a 300 MHz clock frequency and the input samples are only one SRIF fragment large.

For this implementation of the architecture, a QPSK system was developed. The modulator fits within one socket while the demodulator consumes two sockets. Both are able to run at 100 MHz. Running at an upsample factor of 2, the system is able to process 50 million symbols per second, resulting in a performance of 100 Mbps. With seven sockets the system is able to simultaneously run 7 100 Mbps modulators or 3 $^1/_2$ 100 Mbps demodulators.

3.3 Results

For purposes of benchmarking the architecture, its performance for the sample algorithms is compared with the two next most common, likely alternative processors: a desktop CPU and a Texas Instruments DSP. Due to the lack of availability, a comparison with similar reconfigurable architectures is not currently feasible. For each processor the estimated

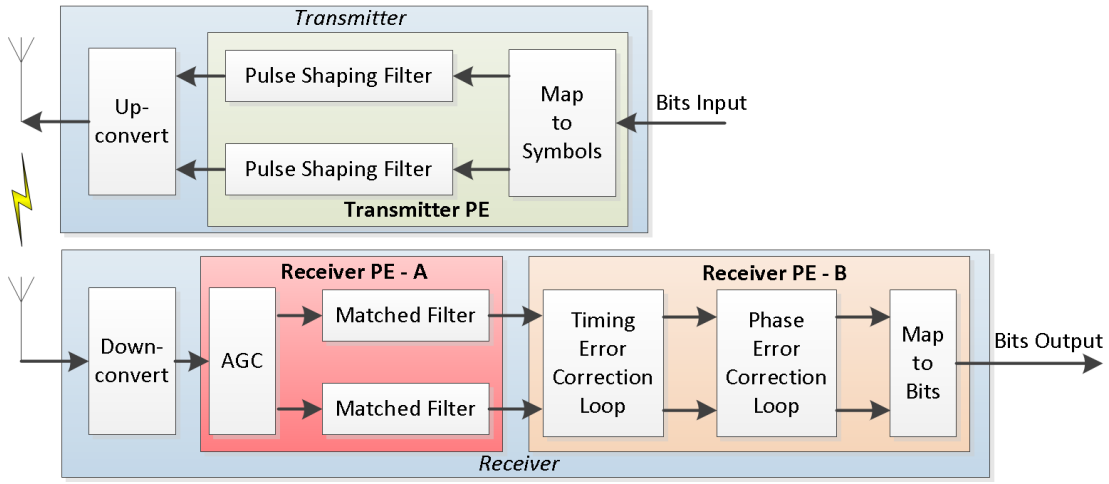

Fig. 3.3: QPSK communication system. The QPSK communication consists of a modulator and a demodulator. All together the system requires three sockets.

maximum data rate for each algorithm was calculated based on experimental data gathered from profiling reasonably-optimized C code. Table 3.3 shows the performance comparison.

CPU results were obtained by profiling the C code on a Pentium 4 3.4 GHz CPU. Only a single thread was used, but the added performance of multiple cores is accounted for by treating each core as a separate instance of the algorithm and calculating the combined data rate. Instrumentation profiling was used to capture accurate execution times, simulating stand-alone algorithm execution. Execution times were averaged over 1,600 generated/received signal samples. DSP results were obtained by simulating DSP-optimized C code on a TI TMS320C6713 DSP. This floating-point DSP has eight execution units and operates at 225 MHz, achieving a maximum 1800 MIPS. The simulator delivers a precise instruction count and function call count. Based on this information the maximum sample rate can be calculated.

Table 3.3: Performance comparison. Comparison of the sample algorithm performance between the developed architecture, a CPU, and a DSP, in MSamples/s for each instance of the algorithm and for all simultaneous instances supported by the processor.

Algorithm	Proposed		CPU		DSP	
	Each	Total	Each	Total	Each	Total
QPSK Mod	100	700	.520	1.040	2.227	2.227
QPSK Demod	100	300	.352	.704	.202	.202

As the table indicates, the implementation handily outperforms both the CPU and the DSP, achieving at least 44x greater data rates for the QPSK system. In addition to higher data rates, the implementation is able to simultaneously execute up to seven such algorithms, all with equivalent performance. The CPU is able to execute two algorithms with similar performance because of hyper-threading and the DSP is able to execute only one.

3.4 Summary

This Virtex-5-based implementation validates the concept of the proposed architecture. A reconfigurable architecture for the domain of signal processing algorithms can be designed that utilizes hardware-optimized logic and enables dynamic reconfiguration as in a software implementation. More importantly, the architecture can be realized on existing, modern reconfigurable fabrics and realize substantial performance improvement while maintaining sufficient flexibility.

Chapter 4

Conclusions

4.1 Summary

A flexible, reconfigurable processor architecture has been designed and presented which targets the SDR domain. The need for a more optimized, extensible architecture was motivated and an overview of existing architectures was presented. These architectures were categorized based on architecture features. Advantages and disadvantages of each category and each architecture were discussed. The conclusion was reached that strictly hardware processors are most likely to be the solution to the processing challenges of SDR applications. This constituted Chapter 1.

Chapter 2 presented the architecture and presented details on its derivation, composition, and execution. The architecture presentation was kept separate from underlying reconfigurable technologies discussions to emphasize the realizability of the architecture on arbitrary technologies, only mentioning specific technologies when they influence the design of the architecture. A sample implementation of the architecture in a modern, mid-sized Xilinx Virtex-5 FPGA is presented in Chapter 3 to help concretize the design and presentation of the architecture. From this specific implementation, performance metrics such as sizing, speed, and throughput are gathered for comparison with alternative architectures.

4.2 Conclusions

Custom reconfigurable processor architectures, such as the architecture presented in this document, are powerful alternatives to general processor architectures for a given domain of applications. This architecture can achieve greater performance for streaming data applications, such as SDR, than standard GPPs and DSPs. Moreover, the architecture is

defined generically enough that it can be realized on different reconfigurable technologies. FPGAs, such as those offered by Xilinx, are particularly good targets for this architecture. Perhaps most importantly, the architecture is not tied to a particular technology which means that it can be used now, on existing technologies, without the need for an expensive, custom-made silicon very-large-scale integration (VLSI) design.

Using this architecture, radio algorithms can be implemented with the performance of a hardware FPGA implementation but still maintain the flexibility provided by software implementations. The architecture can allow, at run-time, arbitrary algorithms and even arbitrary combinations of algorithms to execute. For low-performance algorithms the system can even be over-partitioned via virtualization (socket paging). This architecture makes novel strides to meet the processing requirements needed by future software defined radios and cognitive radios.

4.3 Future Works

This presentation outlines an initial proof of concept for an interesting architecture. However, much more work can and should be done to verify the design decisions and improve the architecture. Such future work regarding this architecture can be classified into two categories: improvements and support. Improvements are modifications to the architecture to increase performance and extensibility. Support is development of external resources to enhance the development for and use of the architecture.

4.3.1 Architecture Improvements

Currently the architecture is not as extensible as desired. The limitation that each PE can only receive data from and send data to one source excludes a large number of applications, including multiple-input, multiple-output (MIMO) applications. Also limiting is the restriction of communication paths amongst sockets. Work can be done to improve the DRM module to allow for multiple sources/destinations for PE data as well as allowing more socket-socket communication paths. Also, while multiple inputs/outputs from/to a single source (such as two-channel operation) can be accomplished by treating each port's data

as separate, consecutive words when going through the SRIF and being reassembled on the other side, this approach feels like a kludge and work should be done on increasing support for single-source multiple inputs/outputs in the form of either architecture modification or protocol specification.

Along with the limitations previously discussed, the limited routing presents another restriction; it can only handle linear data flows. Because of this, loops or any feedback paths cannot cross PEs; they must be implemented entirely within a PE. This unnecessarily limits the possible size of feedback paths realizable on the architecture and restricts the domain of possible application. Performing the stated work to increase the routing capabilities of the architecture would allow for implementations of nonlinear data flow between sockets and increase the usability of the architecture.

4.3.2 Architecture Support

This architectures suffers the same problem as almost every custom processor architecture discussed; it lacks sufficient development support. No work has been done on creating a generalized process to port algorithms onto the new architecture. As a result all work done for the demonstration implementation has been done by hand and configurations have been hard-coded into the system controller's program or the PE's default configurations have been used. Ultimately a more sophisticated user interface could be developed which would allow for large-scale, preset configurations to be loaded with minimal input as well as providing access to the full configuration details of each PE.

Optimally, the development of a compiler for this architecture would most greatly increase the usability of the architecture. While the architecture is not an instruction-based architecture, it does have an available library of functional units which are capable of being combined to accomplish complex algorithms. A standard should be developed for describing the functionality and programmability of a given PE, perhaps as a data flow graph. Then high-level descriptions of algorithms, perhaps in a software language like C or in a functional description like a control- and data-flow graph (CDFG), could be input to the compiler which would map the algorithm onto the PE library. This mapping would

then generate an alternate description of the algorithm based on combinations of PEs and their required configurations. In addition, the compiler would indicate which portions of the algorithm do not current map onto any available PEs so that a sufficient PE could be created.

References

[1] J. Mitola III, "Software radios-survey, critical evaluation and future directions," in *National Telesystems Conference, 1992. NTC-92*, pp. 13/15–13/23, May 1992.

[2] S. Hasan, "A low cost multi-band/multi-mode radio for public safety," in *Software Defined Radio Technical Conference, SDR'06*. SDR Forum, Nov. 2006.

[3] Y. Tachwali and H. Refai, "Implementation of a bpsk transceiver on hybrid software defined radio platforms," in *Proceedings of the 3rd International Conference on Information and Communication Technologies: From Theory to Applications ICTTA 2008*, pp. 1–5, Apr. 2008.

[4] J. Wells, "Faster than fiber: the future of multi-g/s wireless," *Microwave Magazine, IEEE*, vol. 10, no. 3, pp. 104–112, May 2009.

[5] Z. He, J. Chen, Y. Li, and H. Zirath, "A novel fpga-based 2.5gbps d-qpsk modem for high capacity microwave radios," in *2010 IEEE International Conference on Communications (ICC)*, pp. 1–4, May 2010.

[6] C. Dick, F. Harris, and M. Rice, "Synchronization in software radios. Carrier and timing recovery using FPGAs," in *2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 195–204, 2000.

[7] S. Srikanteswara, R. C. Palat, J. H. Reed, and P. Athanas, "An overview of configurable computing machines for software radio handsets," *IEEE Communications Magazine*, vol. 41, no. 7, pp. 134–141, July 2003.

[8] Opencores.org [Online]. Available: `www.opencores.org`.

[9] Impulse accelerated technologies [Online]. Available: `www.impulseaccelerated.com/`.

[10] Altium [Online]. Available: `http://wiki.altium.com/display/ADOH/Introduction+to+C-to-Hardware+Compilation+Technology+in+Altium+Designer`.

[11] Simulink hdl coder [Online]. Available: `http://www.mathworks.com/products/slhdlcoder/`.

[12] Xilinx, *Partial Reconfiguration User Guide*, Xilinx, May 2010.

[13] T. Raikovich, "Dynamic reconfiguration of fpga devices," in *Proceedings of the 15th PhD Mini-Symposium*, pp. 72–73, Feb. 2008. [Online]. Available: `http://www.mit.bme.hu/events/minisy2008/papers/15Minisymp_proceedings.pdf`.

[14] J. P. Delahaye, G. Gogniat, C. Roland, and P. Bomel, "Software radio and dynamic reconfiguration on a dsp/fpga platform," *Frequenz, Journal of Telecommunications*, vol. 58, pp. 152–159, 2004.

[15] A. Sudarsanam, R. Barnes, J. Carver, R. Kallam, and A. Dasu, "Dynamically reconfigurable systolic array accelerators: a case study with extended kalman filter and discrete wavelet transform algorithms," *IET Computers Digital Techniques*, vol. 4, no. 2, pp. 126–142, 2010.

[16] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, Sept. 1997.

[17] P. M. Heysters, H. Bouma, J. Smit, G. J. M. Smit, and P. J. M. Havinga, "A reconfigurable function array architecture for 3g and 4g wireless terminals," in *World Wireless Congress, San Francisco, California*, pp. 399–405. Cupertino, CA: Delson Group, May 2002.

[18] J. Hauser and J. Wawrzynek, "Garp: a mips processor with a reconfigurable coprocessor," in *Proceedings of The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 12–21, Apr. 1997.

[19] S. Srikanteswara, M. Hosemann, J. Reed, and P. Athanas, "Design and implementation of a completely reconfigurable soft radio," in *RAWCON 2000: IEEE Radio and Wireless Conference*, pp. 7–11, 2000.

[20] R. Bittner and P. Athanas, "Wormhole run-time reconfiguration," in *FPGA '97: Proceedings of the 1997 ACM fifth international symposium on Field-programmable gate arrays*, pp. 79–85. New York: Association for Computing Machinery, 1997.

[21] C. Ebeling, D. C. Cronquist, and P. Franklin, "Rapid - reconfigurable pipelined datapath," in *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pp. 126–135. London, UK: Springer-Verlag, 1996.

[22] A. Saha and A. Sinha, "An fpga based architecture of a novel reconfigurable radio processor for software defined radio," in *Proceedings of the 2009 International Conference on Education Technology and Computer*, pp. 45–49. Los Alamitos: IEEE Computer Society, 2009.

[23] Xilinx, *MicroBlaze Processor Reference Guide*, 11th ed., July 2010.

[24] G. Estrin, "Organization of computer systems-the fixed plus variable structure computer," *International Workshop on Managing Requirements Knowledge*, p. 33, 1960.

[25] T. Becker, W. Luk, and P. Y. K. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM 2007*, pp. 35–44, Apr. 2007.

[26] A. Sudarsanam, R. Kallam, and A. Dasu, "PRR-PRR dynamic relocation," *Computer Architecture Letters*, vol. 8, no. 2, pp. 44–47, Feb. 2009.

[27] S. Corbetta, M. Morandi, M. Novati, M. Santambrogio, D. Sciuto, and P. Spoletini, "Internal and external bitstream relocation for partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 11, pp. 1650–1654, 2009.

[28] A. Flynn, A. Gordon-Ross, and A. George, "Bitstream relocation with local clock domains for partially reconfigurable fpgas," in *Design, Automation Test in Europe Conference Exhibition DATE '09*, pp. 300–303, Apr. 2009.

[29] Xilinx, Inc. (Sept 2009) Embedded system tools reference guide [Online]. Available: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/est_rm.pdf`.

Appendices

Appendix A

Reconfigurable Computing

Reconfigurable computing is the term ascribed to digital computer processing where the processor adapts itself to provide an optimized architecture for each task. In virtually all modern computers (both personal and embedded), the central processor consists of a statically implemented architecture. This architecture is typically an instruction-based processor which is generally optimized for the full spectrum of possible requests. These processors are useful in that they perform all tasks well. Clearly, application-specific architectures achieve greater performance but they are then limited in their applicability. The goal of reconfigurable computing is not to map all possible tasks onto a generally optimized architecture. Instead, reconfigurable computing seeks a balance between hardware and software, providing the optimized performance of hardware for each task with the flexibility of software.

In his 1960 article [24], Gerald Estrin initially proposed the possibility of reconfigurable computing by creating a "fixed plus variable structure" computer. This architecture is basically a software-hardware co-processor architecture with an instruction-based processor used for managing and controlling several hardware-implementation fabrics which are customized for a particular task. Estrin uses polynomial evaluation as a demonstration for extracting hardware-optimized functions for implementation in the variable portion of the computer and for implementing system control in the fixed portion. With the presentation of this concept, no longer were computers limited to generally-optimized instruction-based processors but could obtain acceleration outside of software. Care must be taken, though, when designing the variable computer as too-much flexibility tends towards instruction-based processors and we have come full-circle.

Initially, it was not exactly clear how such a architecture proposed by Estrin should be

realized. Decades of research has been performed trying to accomplish a practical reconfigurable computer. Many efforts have produced prototype reconfigurable computing technologies, but almost all have failed to achieve any commercial success. The main barriers have typically been maintaining performance while still allowing substantial configurability. The most successful effort, the Field-Programmable Gate Array (FPGA), capitalizes on the proposal of Estrin by integrating programmable routing, flip-flops, shift registers, and counters.

A.1 Field Programmable Gate Arrays

In 1985, Xilinx created the world's first FPGA. The concept behind the FPGA was to create an array of logic that circuits of digital gates could be mapped onto and programmed in the field (post-manufacturing and by the user). Creating a literal array of gates was impractical and wasteful, so the first FPGA capitalized on look-up tables (LUTs). LUTs are simply very small, read-only memories (ROMs) that are indexed by a certain number of bits (typically fewer than ten) and generate a small number of outputs (typically two or one). Digital logic can be mapped onto LUTs because all combinatorial circuits of digital gates generate deterministic outputs based solely on inputs, meaning that the inputs are like addresses into a space of pre-calculated outputs. By creating arrays of LUTs with distributed registers and programmable interconnects, any arbitrary sequential digital circuit limited to a certain size can be mapped onto an FPGA by properly programming the LUT contents and the interconnects. This groundbreaking revelation allowed Xilinx to construct the world's most successful reconfigurable computing technology.

Many early FPGAs were single-burn only. Their programmability was implemented by physical connections that would either be kept or destroyed by flowing electrical current, literally a "burning." This resulted in devices that, once burned, kept their configuration permanently and were not intended to be modified. Later, physical connections were replaced by tiny static random-access memories (SRAMs) coupled with multiplexers to create programmable routing that could be "burned" in a non-destructive process. This allowed FPGAs to be not only field-programmable but also field-re-programmable. Field-

re-programmability became popular not just because it was more forgiving of programmer mistakes, but because it also enhanced the reconfigurability of the computer, allowing for the FPGA's configuration to be modified at run-time based on user needs.

Estrin, in the same article, indicated that separating the fixed from the variable logic is a trial process and fixed-plus-variable computers will adapt over time as fixed and variable needs are more fully understood from experience. As FPGAs became more and more used, common themes kept arising resulting in Xilinx moving more and more functionality into the fixed side, the hard cores embedded within the FPGA fabric. Modern Xilinx FPGAs contain not only large arrays of configurable logic blocks (CLBs), which contain 4- or 6-input LUTs and multiple registers, but also a variety of fixed hard cores connected to the programmable routing. These hard cores include: DSPs (optimized fixed-point arithmetic processors), large RAMs, PowerPC processors, ethernet media-access controllers (EMACs), and high-speed serial interfaces. A typical FPGA architecture is shown in figure A.1.

Xilinx FPGAs are broken into rows and columns. Columns are distinguished by (generally) homogeneous elements. Rows are grouped by clocking resources and are called "clock rows." Each clock row shares a common clock distribution tree and can contain multiple sub-rows of element types. Figure A.1 shows a simple FPGA with 13 columns and five clock rows with each clock row having two rows of CLBs. An element's row and column becomes a primary addressing mechanism for FPGA configurations.

Xilinx is not the only modern manufacturer of FPGAs, but they are the most prevalent and are the product of choice for this thesis, so discussion of FPGAs in this appendix will be limited in scope to their products.

Programming FPGAs can begin with several different methods, including schematic capture, but typically is done through a hardware-description language (HDL). Figure A.2 shows the Xilinx tool flow that gets input HDL onto the FPGA. Synthesis verifies HDL syntax and performs device-specific optimizations, generating a netlist description of the circuit. Translate translates the behavioral primitives in the netlist into an NGD file containing implementation primitives. Map maps the NGD description of the circuit onto

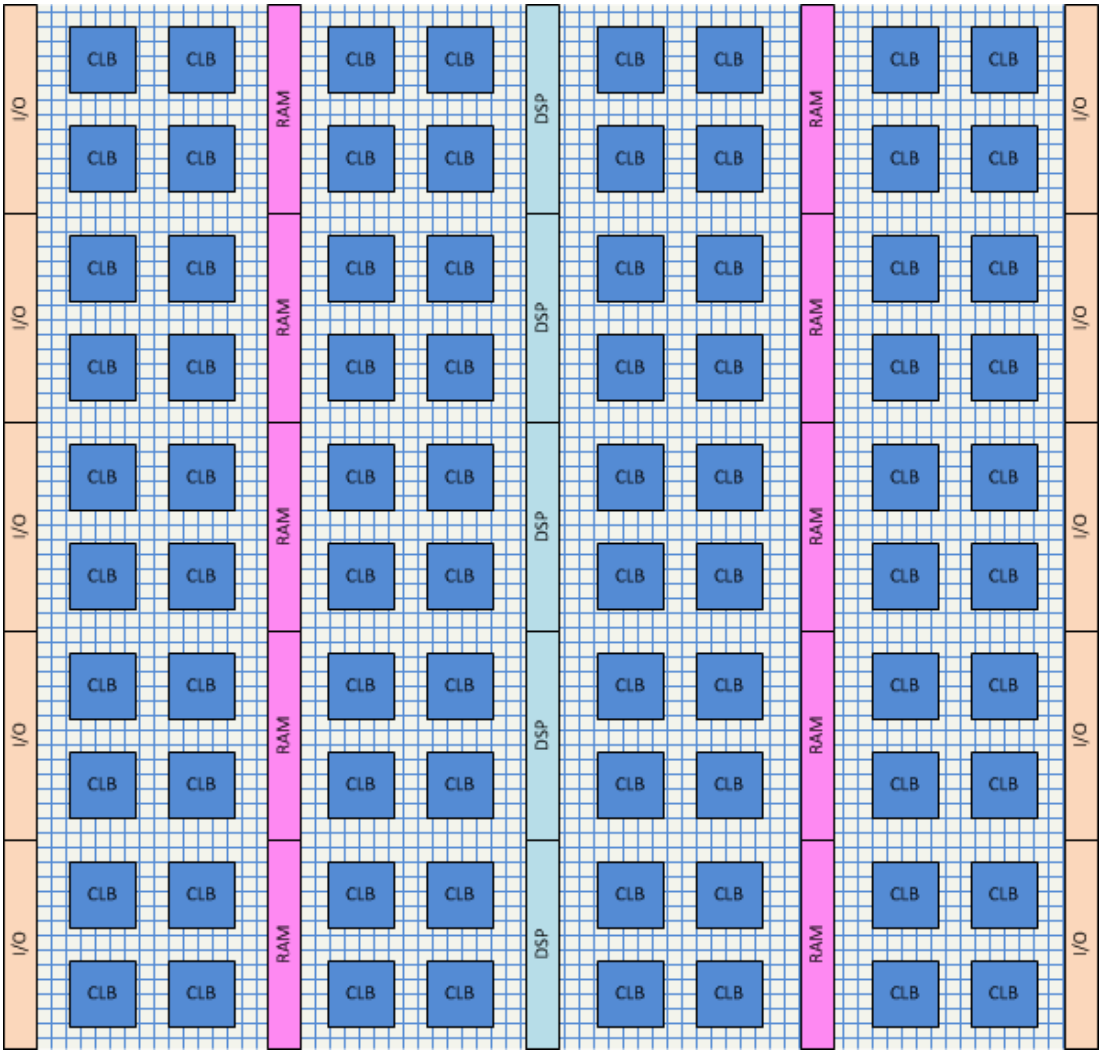

Fig. A.1: Typical FPGA architecture. Xilinx FPGAs consist of a fabric of programmable routing with interspersed logic and hard core components such as multipliers and memories.

device-level components and generates an NCD file. Place and Route (PAR) places the device-level components identified by Map onto specific components of the specified device, routes the programmable interconnects, and outputs an NCD file. Finally, Bitgen generates the bitstream configuration file from the fully-specified NCD file. The bitstream is then stored and loaded onto the FPGA for configuration.

A.2 Partial Reconfiguration

Modern Xilinx FPGAs provide a means of reconfiguring only a portion of the FPGA

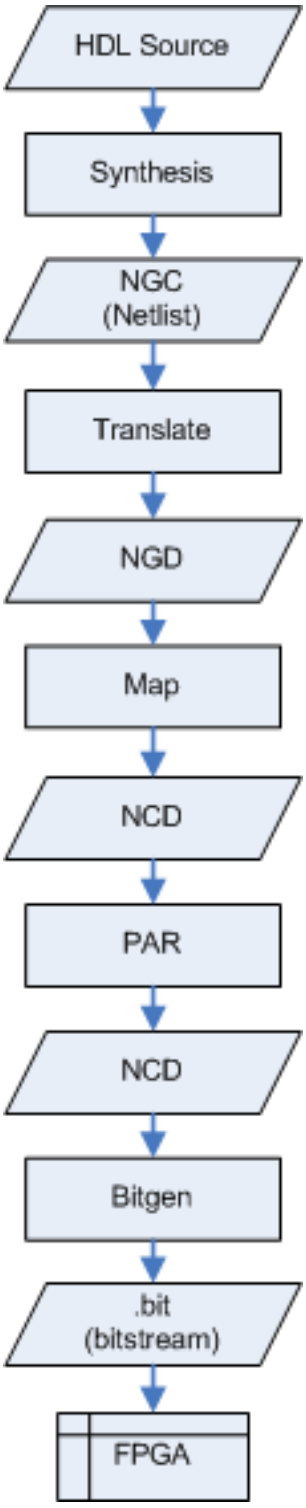

Fig. A.2: Xilinx tool flow. The process of translating an HDL circuit into a bitstream configuration involves five different tools.

instead of reprogramming the entire FPGA. This functionality is called partial reconfiguration (PR) or partial, dynamic reconfiguration (PDR) [12]. Instead of generating full-device bitstreams, partial reconfiguration allows smaller, partial bitstreams to be generated which modify the configuration of only the region of the FPGA addressed by the bitstream, as in figure A.3. Additionally, during the process of partial reconfiguration the non-reconfigured regions are able to remain active, meaning the entire design does not need to stop working because one portion of the design is being swapped out with an alternate configuration.

Partial reconfiguration has natural application to reconfigurable computing, allowing the existence of fixed logic on the FPGA with various variable logic blocks that are populated at run-time based on current algorithmic needs. Typical applications of partial reconfiguration target time-sharing and hardware paging. For instance, in linear algebra applications, partial reconfiguration can be used to swap linear algebra kernels, where only a single kernel needs to be operating on data at a given time. Matrix data can be stored in a fixed-side memory with an attached variable-side partial reconfiguration block. The partial reconfiguration block is then populated at run-time with the desired linear algebra kernel. Throughout the life of the circuit, various operations in various sequences may be performed on the data including transpositions, inversions, determinant calculation, domain mapping and, and least squares solving. Without partial reconfiguration each of these tasks would need to statically exist in the fabric of the FPGA. With partial reconfiguration they all utilize a common area. Thus, partial reconfiguration allows the use of smaller, lower power FPGAs than might otherwise be required.

Like as in linear algebra applications, partial reconfiguration is naturally applicable to signal processing and software-defined radio. Signal processing generally consists of a collection of kernel operations that are performed on data in various sequences to perform various tasks. Statically enumerating each possible operation in the FPGA circuit is physically unpractical, especially since only one or a handful of operations are ever active on a given stream of data. Partial reconfiguration allows the design of a reconfigurable computer where data is fed to signal processing operational blocks, but the specific operation

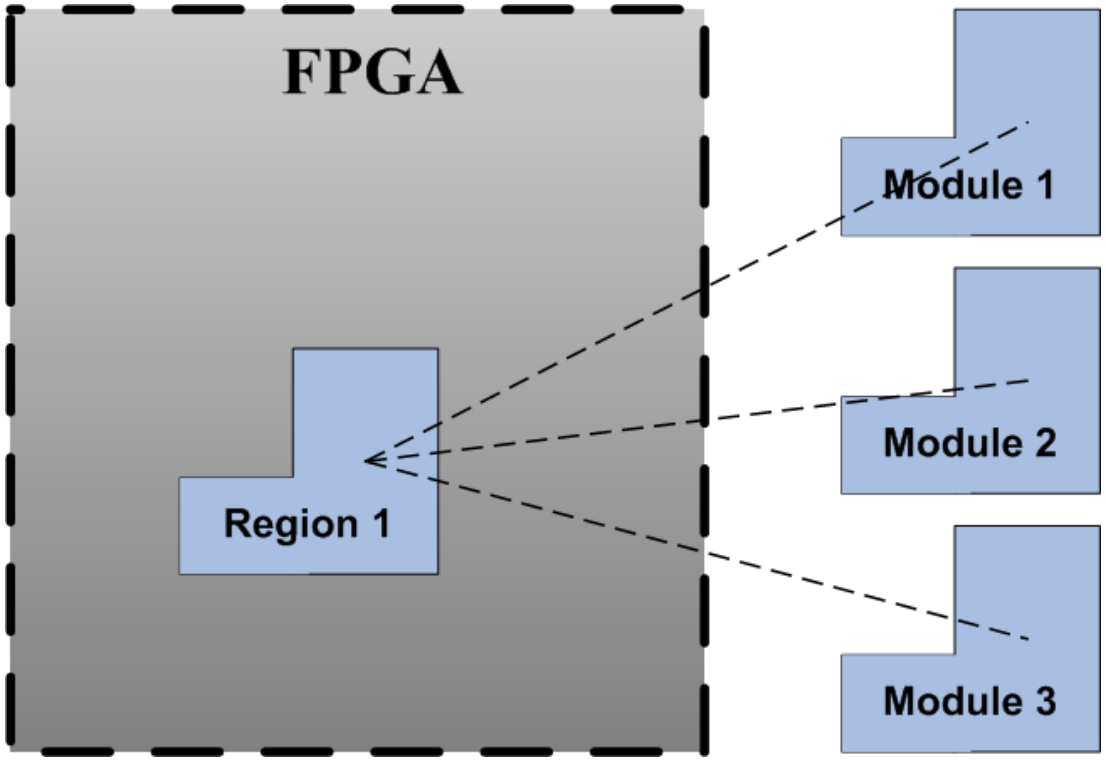

Fig. A.3: Partial reconfiguration. In Xilinx FPGAs, partial reconfiguration allows reconfiguring only a portion of the FPGA with different configurations while the remainder of the FPGA remains active.

being performed by each block is varying and not specified until run-time. A library of signal processing kernels is then created that can be used to populate each operational block according the temporally varying needs of the system.

An alternative to partial reconfiguration is total reconfiguration, where the entire device is reprogrammed. Total reconfiguration is appropriate in many cases and, because of simplicity, is generally used whenever possible. Some applications utilizing partial reconfiguration can still work with total reconfiguration, but full-bitstreams must be generated for every desired combination of modules in each of the reconfigurable regions. This may prove unbearable in terms of bitstream storage memory. Partial reconfiguration allows the storage of only the individual modules and eliminates the need to enumerate each combination of modules in multiple-reconfigurable-region systems.

A.3 Bitstream Relocation

In Xilinx FPGAs, bitstream configurations are generated for specific locations (addresses) on the device, not for specific resource regions across the device. However, the bitstream configurations between two different regions of identical composition is generally the same. Researchers have utilized this structure to allow relocation of a bitstream initially built for one address to be moved to another address on the FPGA [25–28], as in figure A.4.

While bitstream relocation is not generally required for a particular task, bitstream relocation does offer some substantial advantages over the alternative, strictly partial reconfiguration. Without relocation, separate configuration bitstreams must be generated for each possible placement location of a module. This can result in libraries of several to dozens of bitstreams describing the same circuit and only varying, typically, in their destination addresses. This results in a large amount of replication and memory waste. Also, the process of generating each bitstream is laborious and time-intensive; generating only one instance per module saves time and effort.

While the key functionality is provided by partial reconfiguration, bitstream relocation assists in a practical realization of partial reconfiguration. Bitstream relocation creates a more stream-lined, efficient method of managing partial reconfiguration modules and, once implemented and integrated, greatly reduces the complexity and memory requirements of partial reconfiguration use.

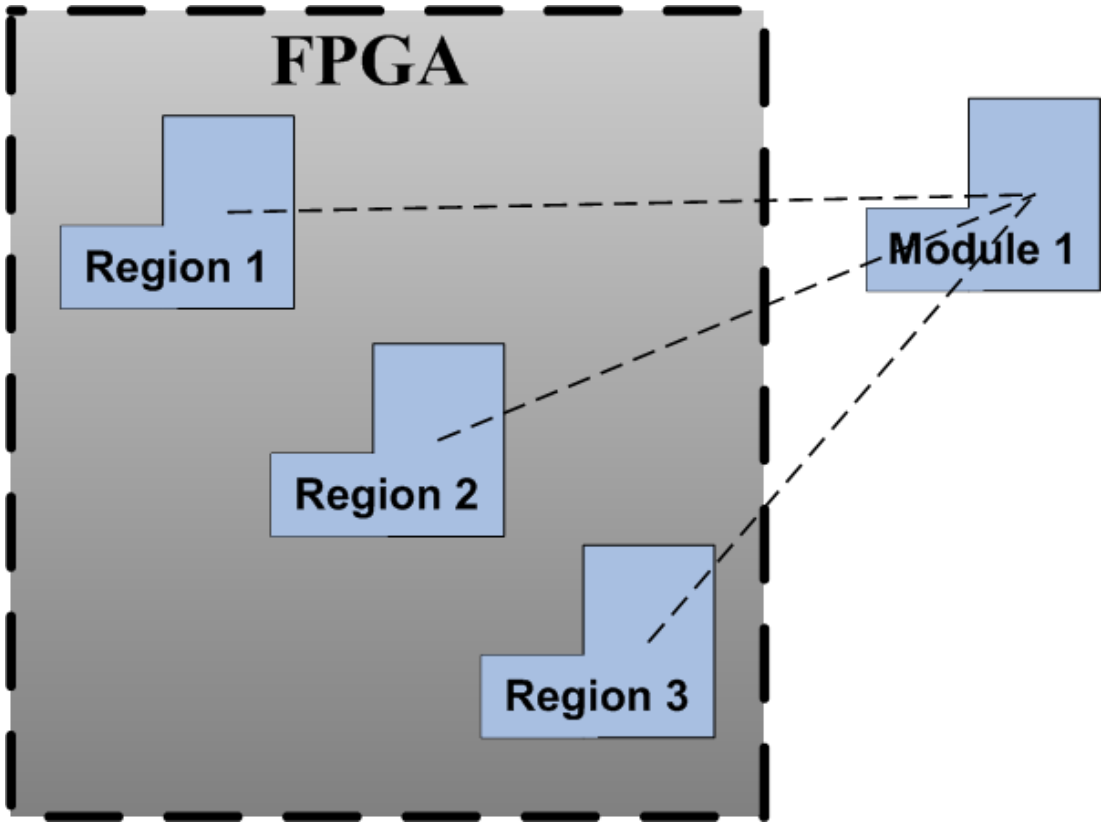

Fig. A.4: Relocation. While not officially supported by Xilinx, relocation allows moving a module generated for one region to other identically shaped and constituted regions.

Appendix B

Xilinx EDK Design

The System Controller, Reconfiguration Controller, and EPIC blocks were all implemented as a Microblaze soft-processor system using Xilinx's EDK tool [29]. Due to lack of time, the software for the processor was not implemented, but the system was completely designed and integrated using EDK in order to get accurate sizing estimates. Figures B.1 and B.2 show the block diagram of the designed system.

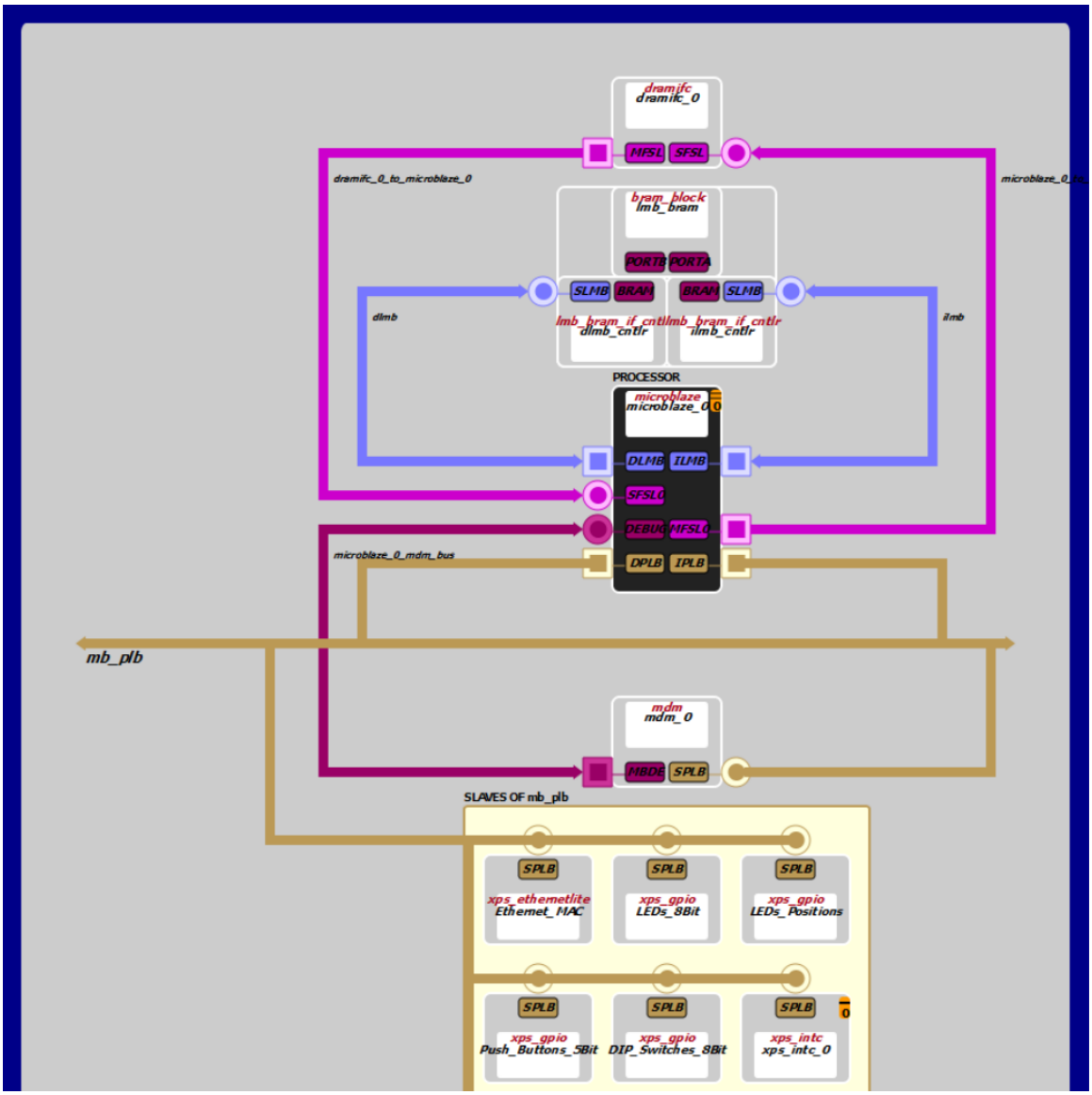

Fig. B.1: EDK controller system block diagram part 1. EDK was used to design and implement the static controller logic including the system controller, the reconfiguration controller, and the EPIC. This block diagram, generated by EDK, shows the architecture of the generated soft-processor, its peripherals and busses (continued to figure B.2).

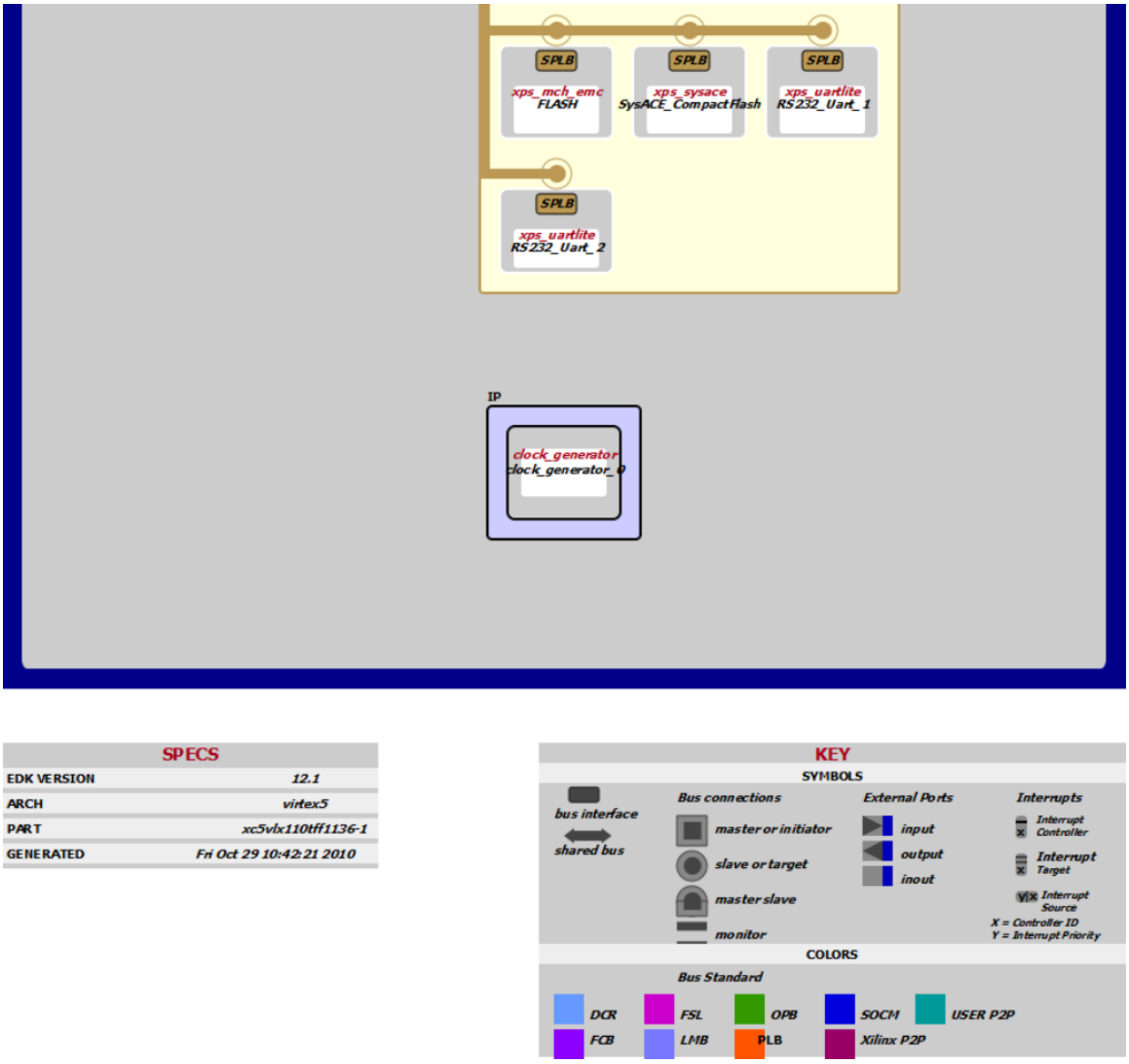

Fig. B.2: EDK controller system block diagram part 2. EDK was used to design and implement the static controller logic including the system controller, the reconfiguration controller, and the EPIC. This block diagram, generated by EDK, shows the architecture of the generated soft-processor, its peripherals and busses (continued from figure B.1).

Appendix C

Architecture Source Code

Complete source code for the xc5vlx110t implementation of the architecture at the time of publication is available on the Appendix CD.