2012

Reconfigurable framework for high-bandwidth stream-oriented data processing

Alexander Mykyta

Reconfigurable Framework for High-Bandwidth Stream-Oriented Data Processing

by

Alexander I. Mykyta

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Dr. Dorin Patru
Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY

September 2012

Approved By:

Dr. Dorin Patru
Primary Advisor – R.I.T. Dept. of Electrical and Microelectronic Engineering

---

Dr. Marcin Lukowiak
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Muhammad Shaaban
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Acknowledgements

I would like to thank:

Dr. Dorin Patru for giving me the opportunity to be a part of this research and for guidance throughout the thesis process;

Brad Larson and Gene Roylance for their insight and continued support;

Ryan Toukatly for his research efforts which lay the technical groundwork for this project;

and my committee members, Dr. Marcin Lukowiak and Dr. Muhammad Shaaban;

Abstract

Designing a digital system that implements an assortment of specialized high-performance algorithms can be costly. Considerable non-recurring engineering costs are required to develop an application specific integrated circuit (ASIC). Additionally, updating or adding features to a design requires the ASIC to be redesigned and re-fabricated. An alternative to using an ASIC is the field programmable gate array (FPGA). The modern FPGA's ability to be partially reconfigured at runtime allows for the device to have the flexibility normally associated with a processor, while also being able to implement digital logic like in an ASIC. This capability allows for multiple digital functions to be loaded into the device at runtime only as needed.

This thesis focuses on developing a reconfigurable framework that enables stream-oriented applications to make more effective use of FPGA resources and to manage partial reconfiguration operations across multiple tasks. This *multichannel framework* addresses several shortcomings of past research that evaluated various dynamic partial reconfiguration techniques using a color space conversion (CSC) engine. This framework allows for multiple different computations to be performed simultaneously, further improving throughput and flexibility of applications implemented within it. Performance of the system is evaluated by comparing its computational throughput to previous efforts using the CSC engine as well as the performance gained from the flexible scheduling that the framework allows. Implementations using the CSC engine show that performance can be improved up to 5 times faster than previous works, as a result of exploiting parallelism.

Table of Contents

Acknowledgements ..... **ii**

Abstract ..... **iii**

Table of Contents ..... **iv**

List of Figures ..... **vi**

List of Tables ..... **vii**

Glossary ..... **viii**

Chapter 1: Introduction **1**

Chapter 2: Background **5**

2.1 Related Work 6

2.2 The Color Space Conversion Engine 9

2.3 Prior Research Using the CSC as a Test Vehicle 11

Chapter 3: Proposed Methodology **14**

3.1 Requirements 14

3.2 Resulting Architecture Concept 16

3.3 Constraints 19

3.4 General Applicability of the Proposed Solution 20

Chapter 4: Implementation **22**

4.1 PCI-Express Interface 23

4.2 Global Clock Synchronization 25

4.3 Channel Multiplexing 27

4.4 Implementation of the MCF Instruction Set 31

4.5 User Circuit 37

4.6 Physical Layout 38

4.7 Software Tools 40

4.8 Validation Procedures 42

Chapter 5: Results and Discussion 46
5.1 Validation of Design 46
5.2 Performance Evaluation 48
5.3 Logic Utilization and Power Consumption 53

Chapter 6: Conclusion 55

References 57

Appendix A: Example MCF Job Script 59

Appendix B: Hardware and Software Used 60
Hardware 60
Software 60

List of Figures

Figure 2.1: Reducing Area using Module-based PR 5

Figure 2.2: CSC Engine Overview 10

Figure 2.3: Dual-Pipe PR CSC Engine 12

Figure 3.1: Basic overview of the MCF 17

Figure 3.2: Comparison of PCIe Data Formats 18

Figure 4.1: Module-level Organization of the MCF 22

Figure 4.2: Clock synchronizer waveform 26

Figure 4.3: Synchronization between clock domains 26

Figure 4.4: Input Stream Demultiplexer and its Actions 28

Figure 4.5: Input Strem Demultiplexer Action Logic Map 28

Figure 4.6: Input Demultiplexer - Example with an uninterrupted data stream 29

Figure 4.7: Input Demultiplexer - Example with a break in the data stream 29

Figure 4.8: Output Stream Multiplexer 30

Figure 4.9: General Instruction Word Format 31

Figure 4.10: Packet Format 33

Figure 4.11: ICAP Interface State Diagram 35

Figure 4.12: Floorplanned Virtex-6 FPGA 39

Figure 4.13: The MCF Toolchain 40

Figure 4.14 : MCF Verification Process 44

Figure 5.1: Effective Throughput vs. Image Size 48

Figure 5.2: Processing Time vs. Image Size 49

Figure 5.3: Overlapped Processing Test Case Scheduling 52

List of Tables

Table 4.1: Instruction Word Opcodes 32

Table 5.1: Peak pixel processing pate comparison under various throughput conditions 50

Table 5.2: Overlapped Processing Results 53

Table 5.3: FPGA Resource Utilization 54

Table 5.4: Reconfigurable Region Resource Allocation & Utilization 54

Table 5.5: Power Consumption Estimates 54

Glossary

ASIC	Application-Specific Integrated Circuit
CMYK	Cyan-Magenta-Yellow-Key
CPU	Central Processing Unit
CSC	Color Space Conversion
DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processor
FIFO	First-In, First-Out buffer
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HP	Hewlett Packard
ICAP	Internal Configuration Access Point
JTAG	Joint Test Action Group
JTRS	Joint Tactical Radio System
MCF	Multichannel Framework
MMCM	Mixed-Mode Clock Manager
NIDS	Network Intrusion Detection System
NOP	No-Operation instruction
PCI	Peripheral Component Interconnect
PCIe	PCI-Express
PLL	Phase-Locked Loop
PR	Partial Reconfiguration
PRR	Partially Reconfigurable Region
Reg-bus	CSC Register Bus
RAM	Random Access Memory
ROM	Read-Only Memory
RF	Radio Frequency

RGB	Red-Green-Blue
RR	Reconfigurable Region
SDR	Software-Defined Radio
SRAM	Static Random-Access Memory
TCP	Transmission Control Protocol

Chapter 1: Introduction

In digital design, using an application specific integrated circuit (ASIC) to process data is the solution of choice when computational speed is a priority. Although, the performance advantages of an ASIC may be appealing, development costs can be prohibitively high. If an application is required to perform a variety of specialized, time-critical computations, one or more ASICs need to be designed, fabricated, and maintained. By their very nature, ASICs are often very inflexible. The incorporation of new features requires the design and fabrication of a new device, which incur significant costs and lead times. These non-recurring engineering costs can be significant enough to make the use of an ASICs justifiable only in large quantities.

Field programmable gate arrays (FPGAs) are viable alternatives to ASICs, since they can be configured to perform a wide variety of functions directly off the shelf. Although, FPGAs have a significantly higher per-unit cost than ASICs, they are well-suited for specialized low quantity designs because of significantly lower non-recurring engineering costs [1]. Since FPGAs are purchased as completed units, a customized system can be developed far more quickly than an ASIC. SRAM-based FPGAs also present the ability to be reconfigured multiple times. This makes hardware updates easy to roll out as a product matures.

Digital logic in an FPGA is implemented using look up tables and configurable interconnects. Because of this, significantly more physical logic is required to implement a function. A design implemented on an FPGA is on average 35 times larger in area than if it was implemented in an ASIC [1], which may make it infeasible to fit all logic

functions into the FPGA fabric simultaneously. One solution is to take advantage of a feature that many modern FPGAs support: partial reconfiguration (PR). By swapping in and out modular functions over time using PR, one can implement all of the required functionalities of a device as they are needed [2]. Normally, the entire FPGA must be held in a reset state during a reconfiguration operation. Modern FPGAs eliminate this requirement by allowing portions of the device to be reconfigured while others remain active. This feature is called dynamic partial reconfiguration (DPR).

Another noteworthy feature of modern FPGAs is the inclusion of hardwired, high-speed serial transceivers. Both Xilinx's and Altera's leading FPGA models feature multiple serial transceivers capable of 28Gbps transfer speeds each [3, 4]. Combining the throughput of multiple transceivers allows for very high data transfer rates: on the order of terabits/sec. These transceivers present the opportunity to apply FPGAs in applications that work with large data sets and require high data bandwidth. Unfortunately, when migrating an ASIC design to an FPGA, it is common to experience a clock speed reduction of 3.4 to 4.6 times [1]. This means that for more complex designs, the application's computational capabilities would likely be bounded by the FPGA's internal clock speed rather than the device's data throughput capabilities.

Integration of reconfigurable processing devices such as FPGAs into computing systems has been done in a variety of ways. Frequently, the reconfigurable logic is coupled to a traditional program-executing processor. K. Compton and S. Hauck [5] classify several typical coupling methods, which range from a deeply integrated functional unit within the processor core to a stand-alone network-attached reconfigurable unit. Tightly coupled systems such as the reconfigurable functional unit

implement custom instructions within the CPU. Such a closely coupled system on chip has the advantage of extremely low-latency communication with the host CPU at the cost of being restricted to smaller working sets. Systems that are loosely coupled experience the opposite effects. A network-attached unit is not able to closely communicate with the host processor which lends itself to be used in applications that require less processor intervention such as ones with larger task sizes. One of the most common coupling methods is implementing the reconfigurable device as an attached processing unit. This type of device lies in the middle of the coupling spectrum and commonly uses interfaces that are readily-available in consumer processor platforms such as PCI or PCI-Express. Using the integrated serial transceivers available in current FPGAs, it is possible to directly interface with a host processor.

This thesis demonstrates a framework for an attached processing unit tailored for stream-oriented data processing operations. The proposed framework leverages the high data throughput of a PCI-Express interface and reconfigurable logic to facilitate the implementation of high-bandwidth stream-oriented data processing applications into an FPGA. An existing digital system, a *color space conversion engine* (CSC) that operates on large image data sets, is used to evaluate the design.

In the following chapter, several other research works are presented that have similar characteristics to the CSC engine. The characteristics of the CSC engine are also described in more detail along with other research that used it as a test vehicle for various reconfiguration methodologies. In Chapter 3, a set of requirements for a proposed DPR methodology are collected. Using this set of requirements, the concept for a proposed reconfigurable framework is developed. Chapter 4 describes, in significant detail, how

the reconfigurable framework was designed and implemented during the course of this research. In Chapter 5, the methods in which the framework were evaluated are described and the results of these tests are discussed. Finally, Chapter 6 concludes the research and also presents potential future work that could further improve the performance of the reconfigurable framework.

Chapter 2: Background

In recent years, the amount of resources available on an FPGA has increased to the point where it is possible for some systems to be implemented within a single device [6]. Such systems commonly consist of multiple independent modules that collectively perform the required tasks. If the system's requirements vary over time, some modules may remain unused. With DPR, it is possible to define a dynamic region in the FPGA fabric where modules can be loaded only when the system needs them. By time-multiplexing functions instead of implementing all of them at once, one can significantly reduce area requirements (Figure 2.1). In turn, this can also reduce static power consumption by allowing a smaller FPGA to be used [7]. Dynamic power can be reduced as well by swapping in and out high-performance modules only as needed and using lower power variants when high performance is not as necessary [2]. Also, with a reconfigurable region, the amount of functions that can be performed over time is only limited by the resources available in that region and the amount of storage space available for reconfiguration data.

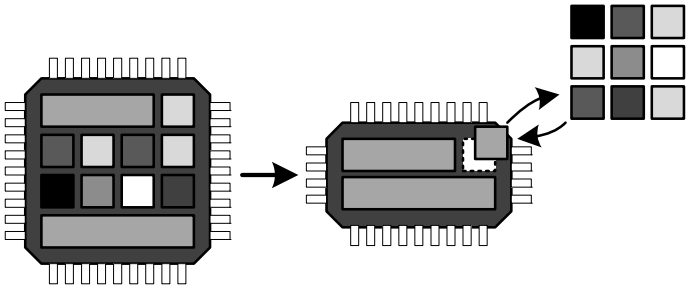


Figure 2.1: Reducing Area using Module-based PR

The implementation of DPR does not come without its challenges. The design of a system using DPR must consider the time it takes to configure a module, in addition to

the processing time. This has a significant impact on scheduling and architectural considerations of the design. Compared to traditional FPGA design, some additional steps are required during the design flow [8]. Reconfigurable regions are described in the top level of the static design as black-boxes. Doing so defines generic interfaces to the reconfigurable module. Each variant of the module is then synthesized separately and linked to the static design in a later step. Another required step in the PR design flow is the floor-planning of the implemented design where regions in the FPGA fabric are manually defined for each reconfigurable partition. Special consideration must be made so that each partition allocates sufficient FPGA resources to be able to accommodate any of the module variants.

2.1 Related Work

Partial reconfiguration of modules can be done in two ways: self or external reconfiguration. A self-reconfigurable system initiates module reconfiguration internally and often stores bitstream information locally. An external reconfigurable system uses logic outside of the FPGA such as a processor to initiate PR operations and supply bitstream data. R. Fong et. al [9] demonstrate a system that is capable of *self-reconfiguration*. In this framework, the FPGA is largely a standalone device that is loosely coupled to a host processor through a network interface. Through this interface, PR bitstreams can be sent to the device and stored locally. The system is classified as self-reconfigurable because it is designed to initiate reconfiguration of its modules on its own, and as needed. Partial bitstream configurations are cached in an off-chip *configuration data medium* and loaded into the RR as required. Using this DPR method,

it is possible to time-multiplex the implementation of a number of modules and only be limited by the amount of local storage available. By storing bitstreams locally, it is also possible to sever the link to the host processor entirely and operate as an isolated system. Although, this is an interesting solution for managing bitstream data, such a loosely coupled device would not be well suited for extremely high-bandwidth data processing.

One notable implementation of module-based dynamic partial reconfiguration is in Xilinx's Joint Tactical Radio System (JTRS) software-defined radio (SDR) kit. In an SDR system, radio frequency (RF) signals are modulated and demodulated digitally in real-time instead of using dedicated analog RF hardware [10]. This allows for a significant amount of flexibility since modulation methods can be modified in software. Traditionally, a four-channel SDR system would use up to 12 digital signal processors (DSPs) to handle the large amount of computing and data bandwidth required. By transitioning to an FPGA, both the large power consumption and cost of the DSPs could be reduced by 2-3 times [11]. The JTRS SDR system also took advantage of the FPGA's PR capability by dynamically swapping out different signal processing logic. In an implementation with multiple SDR channels, an algorithm for a video stream could be reconfigured without interrupting processing of the audio channel [11].

Other research by P. Ostler et al. [12] presented a system where a PCI-Express (PCIe) interface was used to transmit partial reconfiguration bitstreams from a host processor to an FPGA. At startup, only a PCIe core and basic support logic is configured into the FPGA fabric using traditional configuration interfaces. Once the PCIe link is initialized, user circuits can be loaded into a large reconfigurable region through the static PCIe link. This type of flexible framework presents the ability to easily time-multiplex

hardware algorithms through a widely available high-speed data interface. The PCIe interface makes the system more closely coupled to the host processor allowing for higher data bandwidth and tighter control. In this example, the external host processor both initiates the reconfiguration of the user-region and supplies the bitstream data to the FPGA. This method of external-reconfiguration allows for a simplified system that does not require additional hardware to store partial bitstreams.

Another interesting application of the FPGA is for scanning and detection of malicious content within network packets. Conventional network firewalls, which only inspect the header information in a Transmission Control Protocol (TCP) packet, are unable to detect application-level attacks which are usually hidden within the payload of the packet. Deep packet filtering systems, such as the open-source *Snort* system, perform more comprehensive scanning by matching the payload data against a set of known malicious signatures. Traditionally, the Snort network intrusion detection system (NIDS) is implemented using a Von Neuman-style architecture which does not scale well as the size of the signature database grows. For example, implementing a filter with 500 patterns is only able to achieve a throughput of 50 Mbps using a dual-core, 1 GHz Pentium III system [13]. To remedy this, Y. Cho and Mangione-Smith [14, 15] mapped the Snort rule set to pipelined pattern matching filter logic and implemented it on an FPGA. The pattern rules, stored in the FPGA's block RAM, could be updated without modifying the logic by loading a new string table into memory. Using this architecture, it was possible to match a stream of TCP packets against a set of almost 3000 rules at a sustained rate of up to 2.3 Gbps.

An additional research group [16] further enhanced the implementation of the Snort NIDS on an FPGA by adding a hashing module. Instead of checking against each pattern separately, the hashing module determines which pattern is a possible match to the TCP payload. The hashing logic is implemented as a binary hash tree within the FPGA's reconfigurable fabric as opposed to a ROM-based solution. Unfortunately, since the hashing module is tailored to the specific set of pattern strings, it is unable to be updated with just a simple RAM access. I. Sourdis et.al. [16] also increased throughput by further parallelizing the filtering process by implementing a second duplicated datapath. Although it was not attempted in this research, applying dynamic partial reconfiguration techniques to this architecture would enable the hashing module to be updated without requiring the filter to be taken offline.

2.2 The Color Space Conversion Engine

Previous research investigated various dynamic partial reconfiguration techniques in FPGAs and evaluated them using an existing color space conversion (CSC) engine. This image processing engine, provided by Hewlett Packard Company (HP), has been used numerous times as a resource to evaluate various PR methodologies [17-20]. The purpose of the CSC engine is to convert the numerical representation of a color from one coordinate system, or *color space*, to another. One conversion commonly used is from a 3-dimensional red-green-blue (RGB) color space, which is used to represent colors in additive technologies such as displays, to a 4-dimensional cyan-magenta-yellow-key (CMYK) color space. The CMYK color space is typically used in color printing that requires subtractive primary colors [21]. The CSC engine provided for this research is

being used in various HP products to perform such color conversions as well as color adjustments.

The CSC engine consists of a multi-stage, pipelined architecture that can process a stream of pixels at up to one per clock cycle. The conversions are performed on either three-channel or four-channel pixel inputs. The pixel conversions themselves are performed by using pre-computed color lookup tables that store the conversion values for the transformation. Providing an exhaustive lookup table for each possible color combination would require an impossibly large amount of storage (on the order of exabytes for a 4-channel input and 4-channel output operation). Instead, a lower precision lookup table is stored and any intermediate values are interpolated. An advantage of this method is that the time required to perform a color conversion is always constant which lends itself for implementation in a deterministic pipeline architecture.

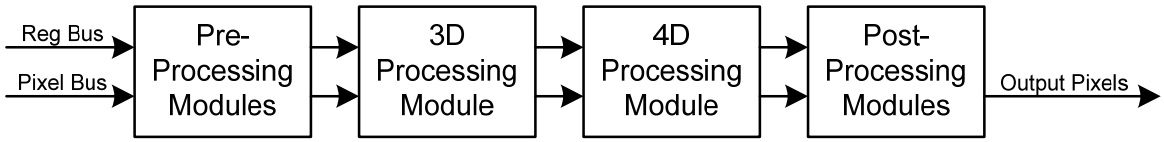


Figure 2.2: CSC Engine Overview

The pipeline architecture is comprised of several stages of processing modules which, depending on the conversion to be performed, are active or can be bypassed. At the core of the engine, two main conversion units perform the conversion between the color spaces (Figure 2.2). The 3D processing module performs conversions for pixels represented using a 3-dimensional color space, while the 4D processing module is used for 4-dimensional inputs. Depending on the input image color space, only one of the modules is required. In the ASIC implementation of the CSC engine, both modules are

present in the pipeline, but either can be bypassed by configuring internal registers through the CSC's *register bus* (reg bus).

2.3 Prior Research Using the CSC as a Test Vehicle

The CSC engine has been used in the past as a test vehicle to investigate various FPGA design techniques; specifically, dynamic partial reconfiguration (DPR). The CSC engine is an interesting candidate for testing DPR techniques because it contains two processing modules, the 3D and 4D portions, which operate mutually exclusively. With this knowledge, it is possible to merge the two into a single reconfigurable module. Loading a new configuration bitstream into this module switches its function as required by the particular image processing operation. Methods of managing these reconfiguration operations have been attempted in several different ways by making effective use of various FPGA resources.

In previous research, S. Patil [19] identified that the use of these 3D and 4D modules is mutually exclusive and, when implemented in an FPGA, can be time-multiplexed using a reconfigurable 3D/4D module. Patil's implementation achieved external partial reconfiguration by sending bitstream data for the 3D/4D module to the FPGA using the conventional JTAG programming interface. In subsequent research, J. Galindo [20] enhanced the CSC engine's register bus so that bitstreams could be transmitted through the existing data interface. The bitstream would then be passed into the FPGA's internal configuration access port (ICAP) to reconfigure the 3D/4D module.

One challenge of managing a dynamically reconfigurable system is that PR operations require a significant amount of time to complete which can incur processing

delays. In the most recent iteration of research using the CSC engine, R. Toukatly [17] investigated one method of hiding these configuration delays by taking advantage of dynamic PR and the large amount of logic resources available in modern FPGAs. In this implementation, two instances of the CSC engine, each containing a reconfigurable 3D/4D module, were instantiated in the FPGA (Figure 2.3). Since during pixel processing, the register bus remains unused, it was possible to overlap configuration time with processing time. With two independent copies of the CSC, it added the possibility to perform image processing on one pipeline while simultaneously performing partial reconfiguration of the 3D/4D region in the other pipeline. The advantage of this was that configuration operations could be overlapped with processing regardless of what kind of color conversion was required.

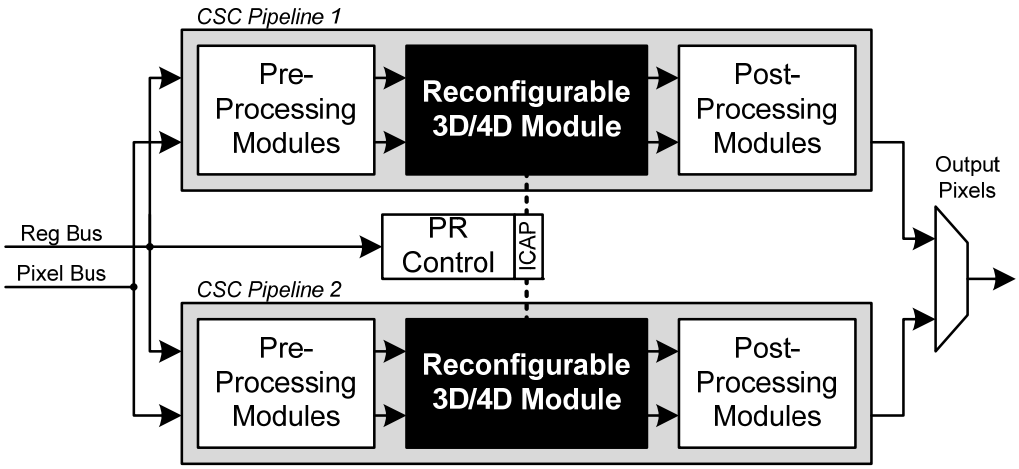


Figure 2.3: Dual-Pipe PR CSC Engine

Another notable achievement of this design was the successful coupling of the system with a host processor using a high throughput PCI-Express interface. In Toukatly's design, the register bus and pixel bus interfaces were preserved throughout the PCIe interface by transmitting a snapshot of each of the CSC's signals for every clock

cycle. Although, the 128-bit wide snapshots contained a significant amount of redundant information, this simplified the migration of the interface to a PCI-Express bus.

The dual-pipeline implementation of the CSC was able to improve performance by reconfiguring portions of the engine while performing pixel conversion at the same time. However, for all of the tests performed, reconfiguration time was always significantly shorter than image processing time. Because of this, hiding configuration operations only resulted in minimal speedup and became negligible as image sizes increased. Although, the design allowed for reconfiguration to be completely hidden, it nearly doubled logic utilization while providing no benefit to actual image processing time. A more efficient approach would be to allow both pipelines to be able to process image data simultaneously after being configured.

Although this research will continue using the CSC engine as a test vehicle, the DPR techniques developed can also be applied to other computational engines with similar characteristics. Systems such as the JTRS software defined radio [11] and the deep network packet filters [13-16] are all implemented using some form of a deterministic pipeline that operates on a stream of data much like the CSC. All of them, if enhanced with DPR techniques, require some method of managing their reconfiguration. Additionally, the throughput of these systems can be further enhanced by exploiting the inherent parallelism of the operations performed. This research will continue work on further enhancing the partial reconfiguration techniques used in the CSC engine as well as improving its overall throughput by making efficient use of the FPGA resources available.

Chapter 3: Proposed Methodology

In order to improve upon DPR methodologies, the shortfalls of the approaches used in prior research were examined along with the computational characteristics of the CSC engine and other similar applications. This information was used to infer a set of requirements that the proposed solution must meet. Using this set of requirements, the concept for a reconfigurable framework was developed. This chapter describes some of the benefits and shortfalls of prior methodologies that influenced the development of the requirements as well as the general concept of the proposed reconfigurable framework.

3.1 Requirements

The ultimate goal is to develop a generalized framework that enables stream-processing applications, such as the CSC engine, to take advantage of FPGA features such as dynamic partial reconfiguration to increase the application's overall processing throughput.

Prior work by Toukatly [17] developed a reconfiguration latency hiding method targeted for the CSC engine. When implemented using an FPGA, computational latencies were successfully hidden by using two identical pipelines and overlapping configuration and computation operations. For test cases using small images, where configuration time was comparable to image processing time, an overall speedup of up to 1.3 was achieved when enabling overlapping using the two pipelines. Unfortunately, tests using large images resulted in negligible speedup since hiding the comparably small reconfiguration latencies became insignificant. Since only one of the two CSC engines could process

image data at a time, the other would remain idle, needlessly occupying logic resources within the FPGA.

Research groups working with the Snort packet filter described in section 2.1 also experimented with the concept of datapath duplication [16]. However, instead of duplicating logic for reconfiguration delay hiding, an additional processing pipeline was added to increase computational throughput by enabling both to process data in parallel. It is clear that, if managed properly, multiple processing datapaths can be used to parallelize certain computations as well as hide latencies related to partial reconfiguration.

In the JTRS software defined radio [11], multiple processing datapaths are also used but to perform *different* operations concurrently. Parallel signal processing datapaths are used to decode multiple radio receive channels such as an audio and video channel. This concept can be applied to the CSC engine to allow for unrelated images to be processed at the same time as well. To accomplish this, these processing channels should be logically isolated in order to allow multiple tasks to be performed independently. Doing so will allow for more application flexibility as well as more effective utilization of the available FPGA resources.

Another drawback of the dual-pipe CSC architecture [17] was that the system's total throughput was dictated by the processing speed of a single processing pipeline. This was due to the limitation that only one pipeline could process pixel data at a time. The slower pipeline speed is due to the CSC engine reference design's clock to be constrained to a maximum of 50 MHz. This clock speed correlates to a maximum data throughput of 3.1 Gbps which is considerably less than the PCIe interface's peak of 10

Gbps [22]. Combined with an inefficient data structure of the input data stream, the PCIe interface was left to be severely underutilized. To achieve improved performance, the proposed architecture should be able to accommodate for processing engines that require lower datapath clock speeds. Additionally, the data rate of the architecture's data interface should be able to scale to higher throughput protocols such as PCIe and beyond.

3.2 Resulting Architecture Concept

In order to address the above requirements, the concept of a multichannel framework (MCF) has been developed (Figure 3.1). The MCF allows for multiple instances of a stream-oriented processing core to operate simultaneously. This is advantageous for the CSC engine because pixel conversions are completely independent from each other, allowing greater parallelism in image processing. With multiple instances of the CSC engine, it is possible to not only overlap configuration operations between pipelines, but image processing as well. These processing datapaths are also logically isolated from each other which allows for multiple unrelated image processing operations to occur concurrently. Due to the isolated nature of the user circuit channels in the MCF, the efficiency of logic utilization can be greatly improved. In Toukatly's design, only one of the two instances of the CSC could perform image processing operations at any time. Since each channel within the MCF is independent of the others, it is possible to process image data on all of them simultaneously, resulting in more efficient logic utilization.

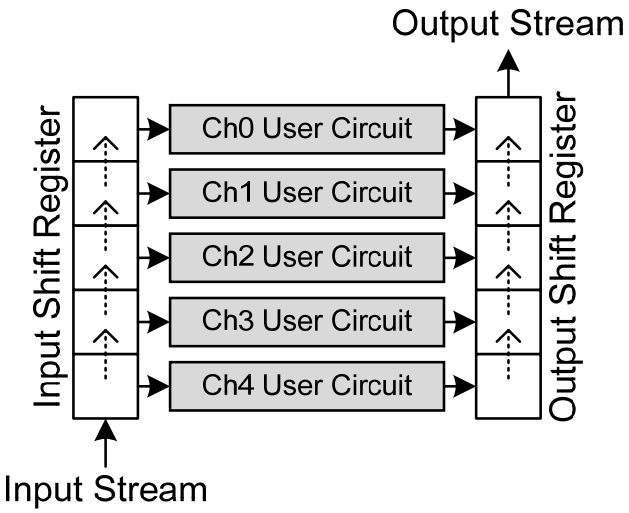


Figure 3.1: Basic overview of the MCF

Within the framework, the individual clock speeds for the user circuits are fixed at 50 MHz which allows for ASIC cores such as the CSC engine to continue to operate without requiring modifications to satisfy timing constraints. The PCIe interface used in Toukatly's research and in this thesis buffers input and output data through a FIFO that can be read and written to 8 bytes at a time, at a rate of 250 MHz. Coincidentally, during image processing, the CSC engine requires at most 8-bytes of unique pixel data at each clock cycle as well. With five instances of the CSC engine accepting 8-byte packets of pixel data each 50 MHz clock cycle, the collective throughput of all the channels can be matched to the limit of the PCIe FIFO interface of 8-bytes at every 250 MHz clock edge.

Distribution and clock conversion to each channel is done by sending a sequentially interleaved stream of data for each channel through the PCIe interface. Channel data packets are shifted into an 8-byte wide shift register and latched for synchronization with the channels' clocks. Resulting data from the user circuits is also latched and shifted out of the framework. By using this architecture, it is possible to

preserve the maximum clock rate of 250 MHz for the PCIe FIFO interface, while seamlessly allowing a slower clock frequency for the user circuits.

Command and control of the MCF is performed through the same interleaved data stream using an instruction-based interface. As described in Section 2.2, the data format through the PCIe interface in Toukatly's implementation preserved numerous redundant signals. During pixel processing, which consisted of the majority of the device's operation, half of these signals remained unused. The MCF eliminates these unnecessary data transfers by utilizing an instruction-based data format which allows for large payloads of data to be transmitted without any overhead. Figure 3.2 illustrates the difference between the two data formats. In the Snapshot of signals method used in Toukatly's research, each 64-bit block of pixel data (white) was transmitted along with two bits of control signals (dark grey). The remaining data (light grey) remained static during the course of a pixel processing operation. With an instruction-based interface, each pixel processing operation is initiated with an instruction header. This type of interface is useful for not only the CSC engine, but other processing systems that operate on large sets of data. The specifics of the instruction-based interface is described in further detail in section 4.4.

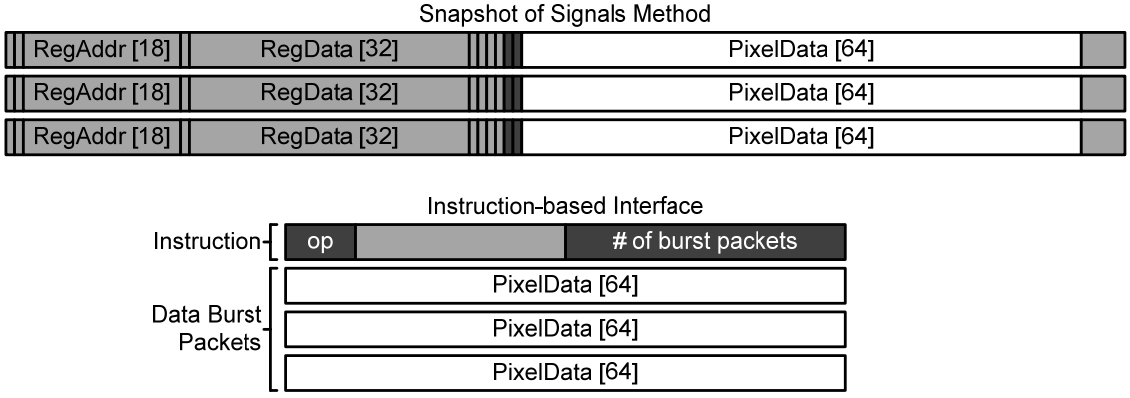


Figure 3.2: Comparison of PCIe Data Formats

3.3 Constraints

Although, the concept of the MCF emphasizes its flexibility, there are a few design constraints to which the framework must conform. First, applying the multichannel framework is only beneficial for applications that have a slower operating data throughput than the interface they are connected to. For the CSC engine, the datapath clock frequency is limited to a maximum of 50 MHz. This reduced clock frequency is a result of the CSC engine being a design originally intended for implementation in an ASIC. When first implemented in an FPGA by S. Patil [19], the engine was only modified to enable partial reconfiguration, not to enhance the clock rate.

Another, more stringent, constraint is that the data rate of the streaming interface must be an integer multiple of an individual channel's data rate. This multiple must also be equal to the number of channels used. For example, the PCIe's FIFO interface is able to exchange 8-byte packets every 250 MHz clock cycle. The CSC engine operates on 8-byte pixels on every 50 MHz cycle. This combination results in a stream to channel clock ratio of 5:1 and requires 5 channels to be used. Once implemented, this clock ratio is fixed at runtime and must be equivalent for each channel.

The last constraint relates to the dynamic partial reconfiguration capability of the framework. Although, several datapaths within the MCF can process data simultaneously, only one reconfiguration operation can be performed at a time. This constraint will be discussed in further detail in section 4.4.

3.4 General Applicability of the Proposed Solution

The major benefit of the multichannel framework is that it's methodology can potentially be applied to many stream-oriented data processing applications. In the case of the JTRS software defined radio, each RF decode filter could be mapped directly to a processing channel within the MCF. By doing so, stream processing and reconfiguration for the audio or video channels can be managed independently. I. Sourdis et. al. demonstrated that their implemetation of the deep network packet filter benefited from datapath duplication to improve performance. Using the MCF, it would be possible to manage multiple instances of the filter as well as enable it to be updated dynamically using the MCF's reconfigurability. With a generalized instruction-based interface, it is possible to add instruction words in order to accommodate a variety of other processing engines. Commanding and controlling the deep packet filter or the JTRS software defined radio could be done by adding several application-specific opcodes to the instruction set.

Different applications are likely to require a variety of data bandwidth requirements. With the proposed MCF concept, the clock ratios between the processing channel and the data stream interface is configurable at design-time. An application that is not able to operate a high clock rate (a high stream-to-channel clock ratio) can be parallelized into numerous channels while an application able to operate at a high frequency (low stream-to-channel clock ratio) would not require as many channels.

Another feature of the proposed MCF concept is that the array of processing channels does not have to be homogeneous. If desired, several entirely different processing applications can operate simultaneously. In addition, each channel's partially

reconfigurable partition is not required to be the same. One channel may not require a partially reconfigurable region at all while another channel may be entirely contained in a PRR. With a fully reconfigurable channel, one could swap out entire applications instead of just modifying them. The potential heterogeneity of the MCF is not limited to just the individual channel. Adjacent processing channels can potentially be merged together to create a single, wider datapath if the application requires it. Although possible, methods of merging channels will not be explored in this thesis.

Chapter 4: Implementation

In the previous chapter, the areas of improvement in past and related works were investigated and a set of desired system requirements were generated. These requirements have been used to develop a general concept for a reconfigurable multichannel framework. In this chapter, the process of implementing every aspect of the MCF is discussed in detail.

To simplify the hardware description language (HDL) implementation of the MCF, the design is divided into several distinct functional units as shown in Figure 4.1. By dividing up the framework in such a way, each component can be designed and tested separately. Breaking down a system into smaller components also helps the designer by reducing the individual task size, making the overall design process quicker. This chapter describes, in detail, the design and implementation of various HDL and software components that make up the Multichannel Framework.

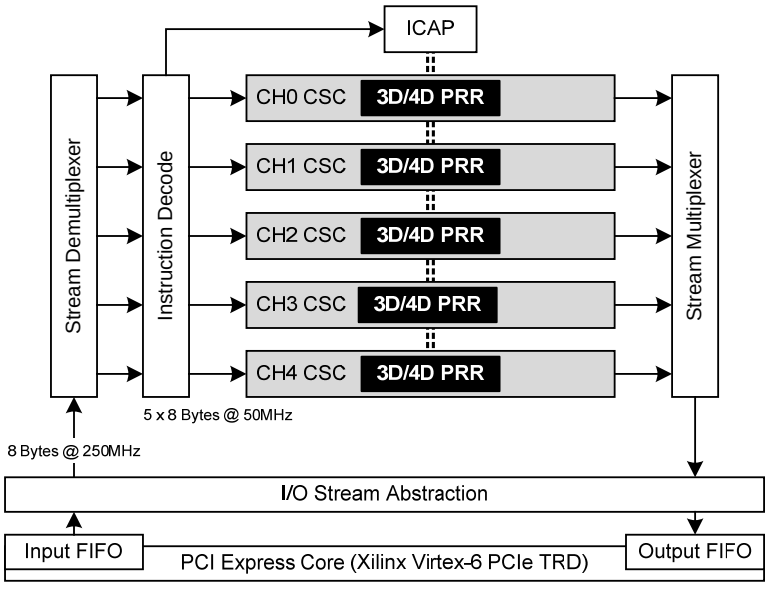


Figure 4.1: Module-level Organization of the MCF

4.1 PCI-Express Interface

To provide an interface between the host processor and the MCF, an existing PCIe interface based on the reference design provided by Xilinx's ML605 Connectivity Kit is used. Originally, the HDL and software for the PCIe reference design only implemented a loopback. Previous work by R. Toukatly had already modified the design to enable a useable data path. To further improve the interface's performance, several modifications to the software drivers were made.

First, the Linux driver was further streamlined. Originally, an intermediate ring-buffer was used to store outgoing packets before committing them to the direct memory access (DMA) buffer. Subsequently, it was discovered that the ring buffer was not necessary and was therefore eliminated. Second, the *vecSender* program written by R. Toukatly, which communicates with the PCIe driver to send and receive test data, was also enhanced to improve performance. Originally the program used two threads: one to write to the PCIe device, and another one to poll the device for data received. When experimenting with the software, it was discovered that the program would spend an excessive amount of time polling the PCIe device without sending additional packets. Since the MCF operates constantly on a stream of data, additional output data cannot be produced if no input data is received. To alleviate this condition, the *vecSender* program was consolidated into a single thread that would switch between writing to the PCIe device and polling it in a far more predictable fashion.

Several aspects of using the PCIe reference design were simplified as well. Originally, the HDL code consisted of several sub-modules along with numerous

interface-related signals instantiated at the top-level of the design. To ease the process of merging the PCIe reference code with the MCF code, the former was compartmentalized by placing it into a wrapper module.

The PCIe reference design provides a data interface to the user using two first-in first-out (FIFO) buffers. To handle communication between the PCIe's FIFOs and the MCF, a new HDL module, called the *I/O Stream Abstraction unit*, was created. Both of the PCIe's FIFO buffers are designed to provide the number of bytes available to be read or written. Unfortunately, documentation for the reference design indicates that these values are delayed by a clock cycle, making the interface slightly cumbersome. Also, since the MCF does not require input data to be stalled at any time, the abstraction unit forces data to be read from the input FIFO as soon as it is available. Doing so further simplifies communication between the MCF and PCIe. Whenever a new 8-byte packet is available and has been read, the abstraction unit asserts a `data_valid` signal indicating to the MCF that a new packet is available. The output FIFO is handled similarly: If data from the MCF is available, it is written to the output immediately. In order to prevent an output FIFO overflow, the abstraction unit monitors the number of free bytes remaining. If the value reported from the FIFO drops below a threshold, the abstraction unit throttles back the amount of data being read from the input FIFO. Since the MCF is designed for pipelined systems, reducing the data flow on the input will reduce the amount at the output, allowing the output FIFO to recover.

Embedding an input/output (I/O) abstraction unit is also advantageous because it can simplify the migration of the MCF to another interface. Future adaptation to different

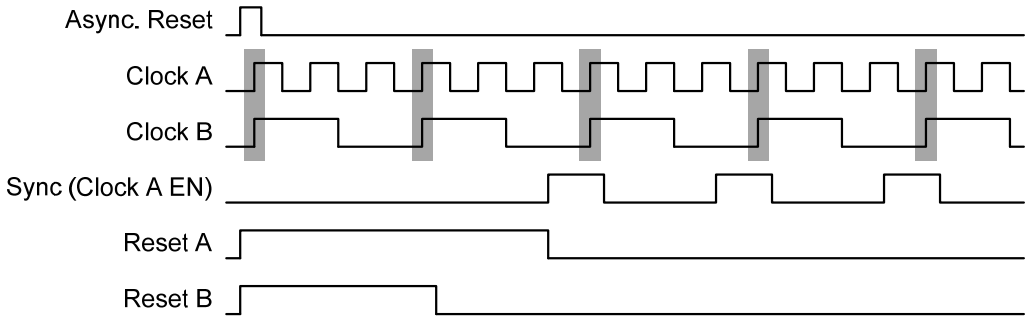
buffer styles or DMA-like interfaces is greatly simplified by abstracting the MCF's inputs and outputs to a single data bus and a corresponding `data_valid` signal.

4.2 Global Clock Synchronization

To drive the logic within the MCF and its user circuits, three main clock signals are used. A 250 MHz clock is used for the PCIe FIFOs as well as the channel multiplexing logic for the MCF. Another 50 MHz clock is used for all channel-based logic including the user circuits. Lastly, a 100 MHz clock is used to drive the ICAP interface which will be described in greater detail in section 4.4. All three of these clocks are derived from the same external 200 MHz clock source. This differential clock is connected to a Mixed-Mode Clock Manager (MMCM) which uses a PLL to synthesize multiple output clocks. One useful feature of these output clocks is that they all have a fixed phase relationship. For example, every fifth rising edge of the 250 MHz clock will be aligned with each rising edge of the 50 MHz clock. This phase relationship is valuable when moving data between clock domains because it eliminates the requirement for multi-staged synchronizers to account for metastability.

When moving data between clock domains, transfers are only done when the two clocks' rising edges are aligned. Doing so allows signals in the slower clock domain to propagate for the entire clock period, resulting in the longest setup time possible. Unfortunately, the MMCM does not provide information on when the clocks share a rising edge so a synchronization module is developed to do so. The clock synchronization module is designed to generate a sync pulse whenever a pair of phase-locked clocks are going to have matching rising edges. In order to facilitate reliable resetting of the MCF,

the module also translates an asynchronous reset pulse into corresponding synchronous reset signals for each clock domain. The synchronous reset signals ensure that the first rising edge in each clock domain is the one that is also synchronized with the other clock. Figure 4.2 illustrates an example of the synchronizer's startup procedure with a hypothetical pair of clocks with a 3:1 frequency ratio. In the MCF clock ratios of 5:1 and 2:1 are used when synchronizing from the 250 MHz to 50 MHz domains and 100 MHz to 50 MHz domains.



(Example using 3:1 clock ratio)

Figure 4.2: Clock synchronizer waveform

When moving signals between clock domains, registers using the faster clock use the generated sync signal as a clock-enable (Figure 4.3). Registers are also reset using the corresponding generated reset signal. In addition to generating the sync and reset signals, the synchronizer module monitors the status of the clocks. If for whatever reason synchronization is lost, the module will reset the system.

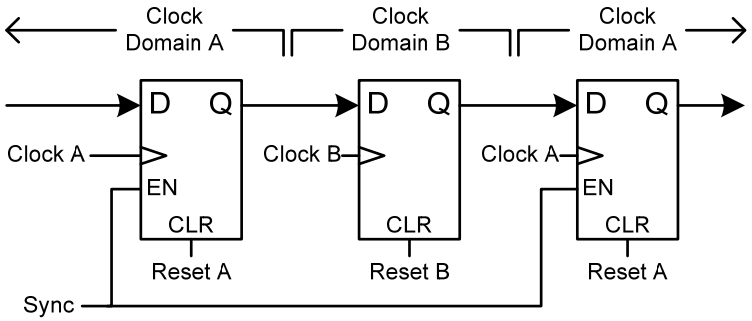


Figure 4.3: Synchronization between clock domains

4.3 Channel Multiplexing

Data is distributed to and from each channel's user circuit using a process called demultiplexing and multiplexing. In communication systems, a demultiplexer is defined as a mechanism that breaks a high-rate data stream into multiple lower-rate channels. A multiplexer performs the opposite operation and recombines the channels back into a single higher-rate stream. For the MCF, the input data from the host processor is organized as a time-divided stream of 8-byte packets. Every consecutive set of five packets, each of which corresponds to a separate channel, is called a *frame*. The packets arrive sequentially from the PCIe link and are read from the FIFO buffer one at a time at a clock rate of 250 MHz. The demultiplexer must then ensure that each frame of packets must arrive at the channel inputs both aligned with the proper channels and synchronized with the slower 50 MHz *channel clock*.

The MCF's demultiplexer essentially consists of two stages of shift registers. As described in section 4.1, the input stream can generate up to one packet available at every clock cycle. Occasionally, due to latencies within the PCIe interface, gaps in the input stream occur and packets may not be available for several clock cycles. Additionally, the PCIe abstraction unit simplifies the interface by enforcing the restriction that if data is available to be read, it must be accepted by the input demultiplexer on the same clock cycle. The demultiplexer must be able to buffer these packets so that it can correct for any breaks in the transmission and pass along the packet frames to the channels intact. All of this is done by implementing a primary shift register and a secondary shift register as shown in Figure 4.4. The figure also describes every possible data transfer within the demultiplexer architecture. These actions are executed depending on the state of the

primary shifter, whether input data is available, and if the next 250 MHz clock edge coincides with the rising edge of the 50 MHz channel clock. Every possible case and the corresponding actions are presented in a logic table in Figure 4.5.

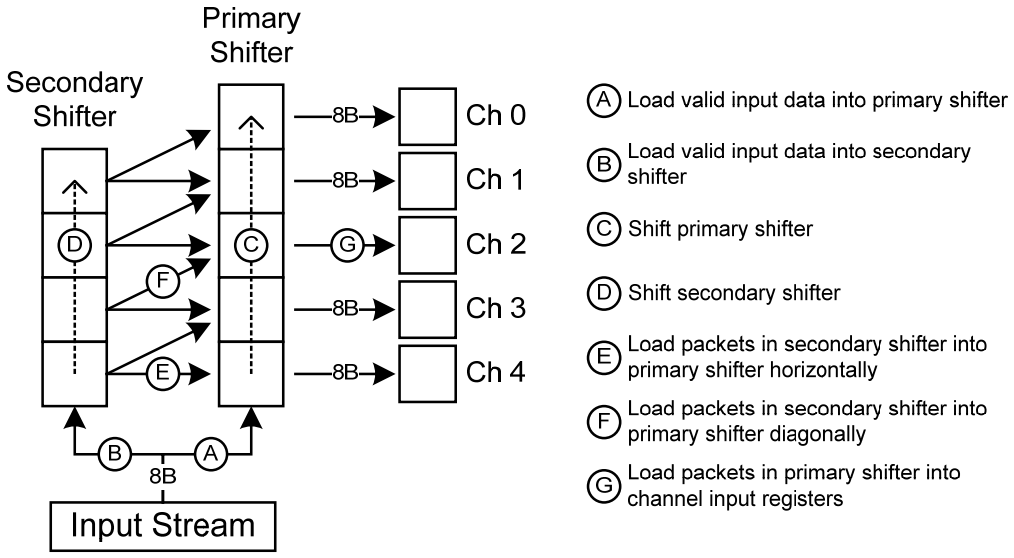


Figure 4.4: Input Stream Demultiplexer and its Actions

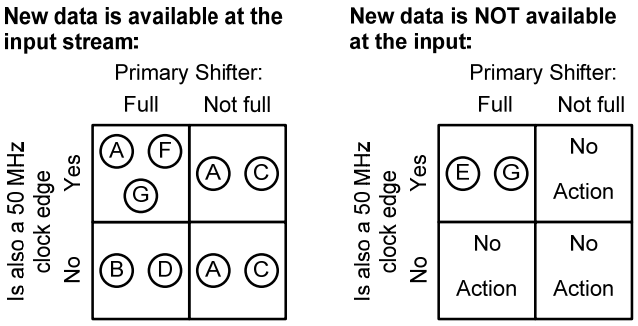


Figure 4.5: Input Strem Demultiplexer Action Logic Map

To further illustrate how the input stream demultiplexer operates, two examples showing packet data flow are included below. In Figure 4.6, an input packet is available at every cycle and fills up the primary shift register (a-e). Since the packet frame is already aligned with the phase of the 50 MHz channel clock, the primary shifter contains an entire frame (e) which can be transferred to the channel input registers in time for the

rising edge of the channel clock. At the same time, the first packet in the following frame can be loaded into the now-empty primary shifter (f).

If an input packet is not available to be read by the demultiplexer at any cycle, the packet frame may become desynchronized with the channel clock. Figure 4.7 illustrates the worst-case example of how the MCF's demultiplexer compensates for any misalignment. Once again, input packets begin to fill the primary shift register. On the fourth rising edge of the figure (d), an input packet was not available and the primary shifter was not able to be filled in time for the rising edge of the channel clock. To correct this, the frame is stored until the following rising edge of the 50 MHz clock where it can be transferred to the channel registers in its entirety (j). During this time, the following packets are stored in the secondary shifter until the primary shifter has been emptied.

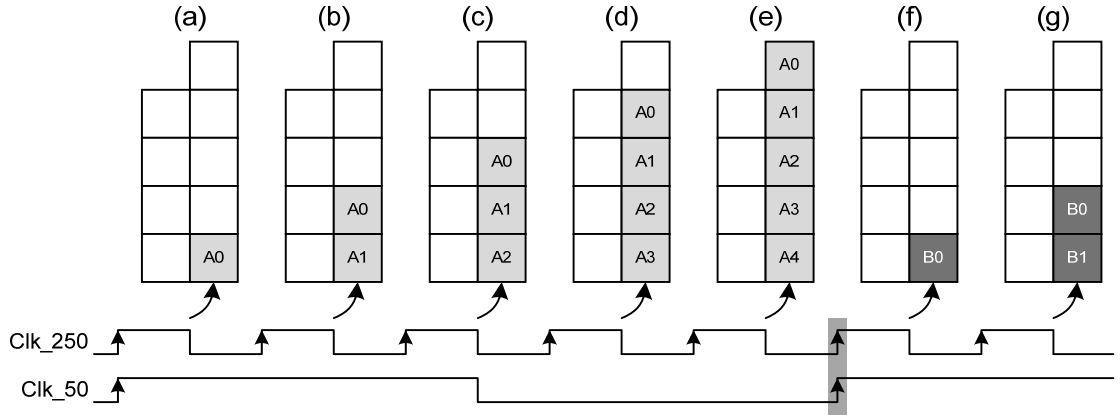


Figure 4.6: Input Demultiplexer - Example with an uninterrupted data stream

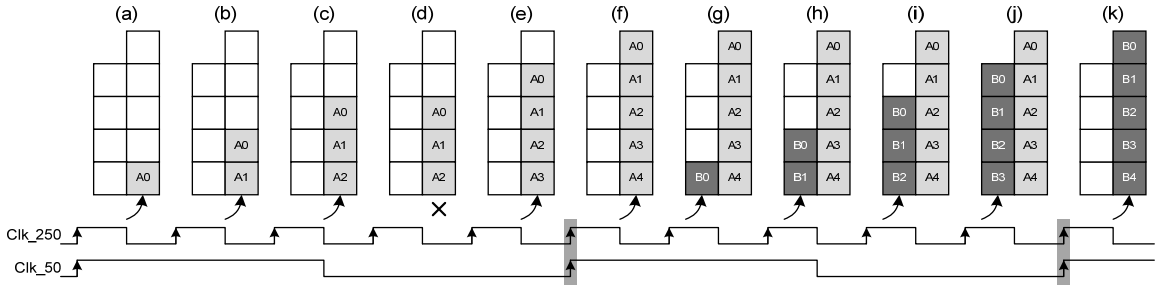


Figure 4.7: Input Demultiplexer - Example with a break in the data stream

The MCF's output multiplexer operates significantly differently from the input demultiplexer (Figure 4.8). Because the outputs from the channels' user circuits do not always generate data, it is not possible to simply interleave each channel's packet sequentially back into the output stream. Instead, when a new frame is loaded from the channels, the frame is compacted by shifting valid packets into slots that did not generate an output. The compacted frame is then transferred to another shift register that transfers the frame to the output stream 8-bytes at a time. A 1-byte bit flag header is prepended to each frame that indicates which channels generated a valid output. In this implementation of the MCF, output packets are only 6-bytes in size. This is because the converted pixel data from the CSCs output requires at most 48-bits. Using the optimal packet size eliminates any overhead from the bit-stuffing that would be required if using an 8-byte packet.

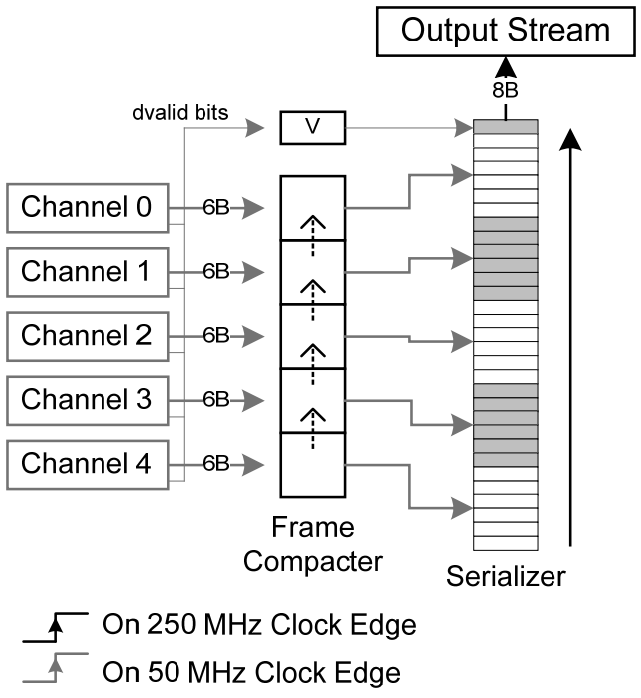


Figure 4.8: Output Stream Multiplexer

4.4 Implementation of the MCF Instruction Set

The instruction-based interface organizes the input data stream into 8-byte packets. Each packet can serve as an instruction word or a payload of raw data. The instruction words are used to control either the MCF itself or execute user-defined actions within the user circuits. Additionally, an instruction word can initiate a burst transfer when a large payload of data is required for a particular command. After a burst instruction, the specified number of packets of raw data are transmitted and are not interpreted as instruction words. This allows large amounts of data to be transferred with minimal overhead.

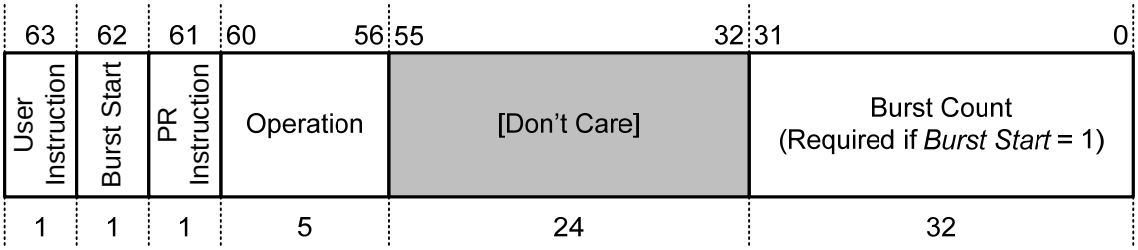


Figure 4.9: General Instruction Word Format

Each packet that is to be interpreted as an instruction word uses its most significant 8 bits as the opcode (Figure 4.9). The upper three bits of the opcode are used to indicate what kind of instruction it is. The first bit, if set, indicates that the instruction is user-defined and intended to be decoded by the user circuit. Otherwise, the instruction is a non-processing command that controls the operation of the framework itself.

Both processing and non-processing commands may require a large payload of data. If the operation requires a burst of data, the *burst start* bit in the opcode is set. This indicates that subsequent packets will be raw data and are not to be interpreted as instructions. When the *burst start* bit is set, the number of raw data packets following the

instruction is indicated by the *burst count* value in the lower 32-bits of the instruction word. If the *burst start* bit is not set in the opcode, the operation is atomic and the following packet for the channel is interpreted as a new instruction word.

The third bit in the opcode, when set, indicates that the FPGA's internal configuration interface is required for a reconfiguration operation. The remaining 5-bits of the opcode are used to further identify which operation is to be performed. Except for burst instructions, the remaining 56 bits can be used as needed if additional information is to be passed along with the instruction word. Table 4.1 summarizes all of the MCF's opcodes used for both non-processing commands and CSC-specific processing operations.

Bit position	*63* **User Instruction**	*62* **Burst Start**	*61* **PR Instruction**	*60 … 56* **Operation**	**Resulting Opcode**
Non-Processing Commands:					
No Operation	0	0	0	0x0	0x00
Start PR Data Burst	0	1	1	0x1	0x61
Flush MCF	0	0	0	0x2	0x02
Channel Sync	0	0	0	0x8	0x08
CSC Commands:					
RegBus Write	1	0	0	0x1	0x81
Start Pixel Burst	1	1	0	0x2	0xC2

Table 4.1: Instruction Word Opcodes

After the input stream is demultiplexed, each channel's data passes through an instruction-decode stage which contains the logic necessary to interpret non-processing command opcodes. In order to keep track of which packets are raw data and which are to be interpreted as instructions, a burst counter generates an `is_burst` signal for each channel's instruction decode logic. If the burst counter detects a packet with the *burst start* bit set, it latches the *burst count* value and counts packets until the counter expires. During this time, the `is_burst` signal is asserted.

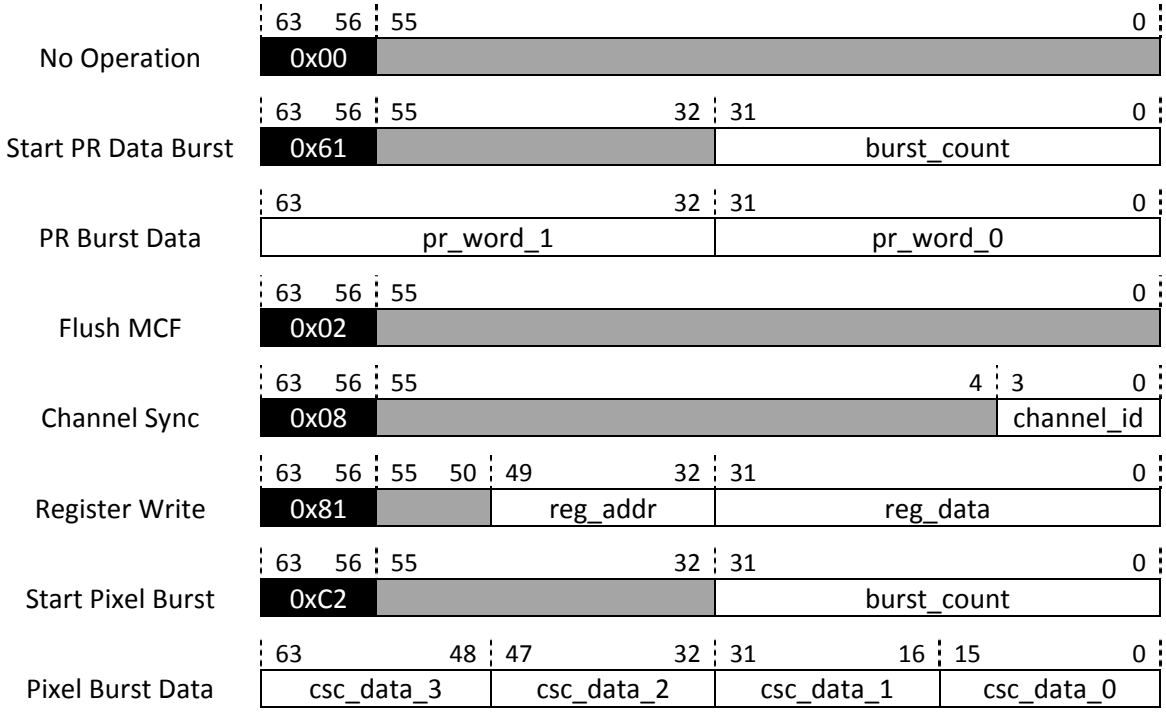


Figure 4.10: Packet Format

Several non-processing instructions are defined to facilitate control of the MCF. First, a synchronization instruction ensures that all subsequent interleaved channel packets are aligned to the corresponding user-circuit channel. Next, to facilitate partial reconfiguration of the user-circuits, a PR burst instruction is included so that a configuration bitstream can be loaded directly into the FPGA's configuration interface. Finally, a *flush* instruction is used at the end of a computation stream to ensure that any remaining data inside the user-circuit pipelines is propagated fully to the host machine through the PCIe interface.

To control logic within the channels' user-circuits, custom instruction words can be defined provided they do not conflict with existing non-processing MCF instruction words. For the implementation of the CSC engine into the MCF, two additional instruction words are defined. One allows data to be written into the CSC's internal configuration registers through its native Reg-bus. The other initiates a pixel-processing burst. Once initiated, an image can be processed with minimal overhead. Packet formatting for both non-processing MCF commands and CSC operations is included in Figure 4.10.

At the start of a processing job, it is necessary to ensure that the sequence of packets in the interleaved stream is aligned and synchronized with the input demultiplexer. This is done using the *channel sync* instruction that verifies frame alignment and re-establishes it if necessary. First, the host processor sends a single frame consisting entirely of sync commands. Along with its opcode, each sync packet includes the channel ID that it is intended for. Each channel compares the ID that it receives with its actual ID. If any of the channels' IDs do not match, the demultiplexer is notified and it discards the appropriate number of input packets to re-establish frame alignment.

Since several packets may be discarded in the process, the host processor must also send at least 3 frames of no-operation (NOP) instructions following a sync operation. This ensures that no actual commands get discarded. The sync command also performs a hard reset of all of the logic within the user-circuits.

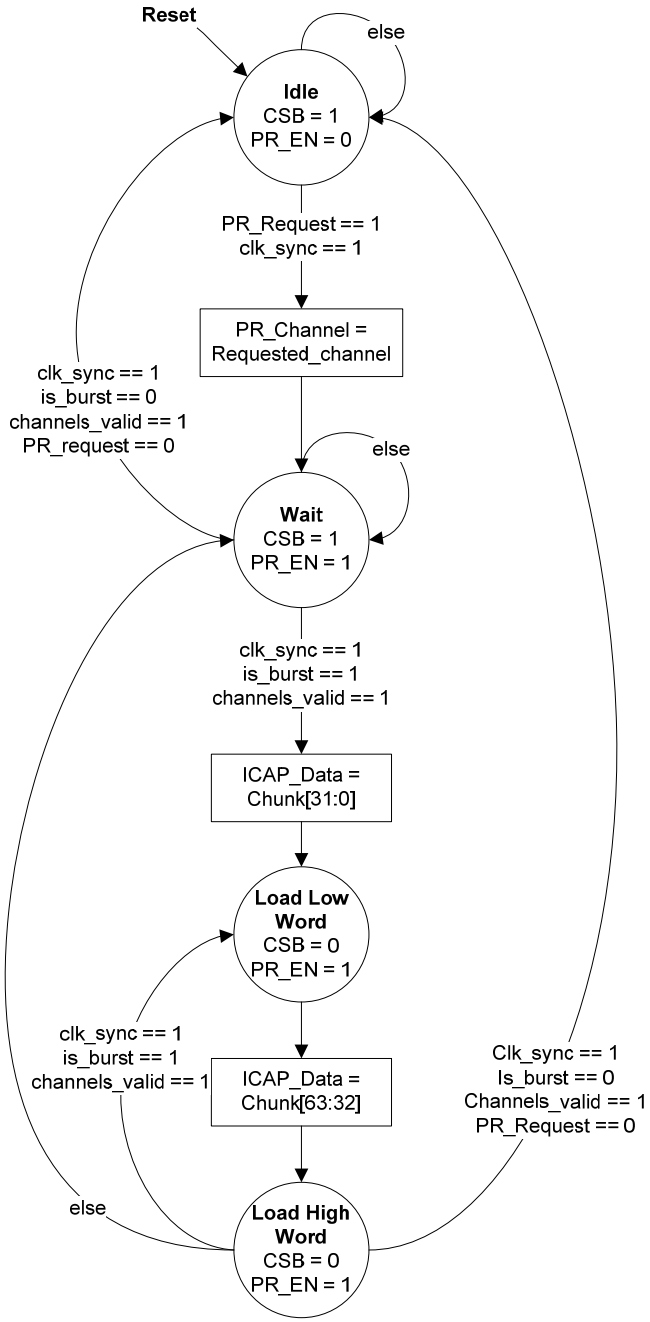


Figure 4.11: ICAP Interface State Diagram

The Virtex-6 FPGA being used contains an *Internal Configuration Access Port* (ICAP) that allows the device to be configured using a data path other than the traditional JTAG interface. The MCF provides user access to this port through another instruction type. The *PR Burst Start* instruction is used to initiate a burst of PR bitstream data. Since

the ICAP interface can accept 32-bit words at a speed of up to 100 MHz, each incoming channel packet is split and multiplexed to this higher frequency. Figure 4.11 illustrates the ICAP control logic state machine which operates on the 100 MHz clock. The ICAP clock (100 MHz) and the channel data clock (50 MHz) are edge-synchronized using a `clk_sync` signal. During a PR operation, the corresponding channel's output signals are overridden to prevent invalid data to be latched into the output multiplexer.

One important limitation of the PR instruction is that only one channel can perform a PR operation at a time. This is because the Virtex-6 only has one reconfiguration data path available[1]. If the host processor erroneously issues two PR instructions, the operation that is already in progress or the channel with the lower ID number takes priority while the other operation is ignored.

In the PCIe reference design, input and output stream data is transmitted in 4kB packets. Because of this, all transactions between the MCF and the host processor must be a multiple of 4096-bytes long. To satisfy this restriction, a *flush* instruction is implemented. This instruction ensures that all pending output data from the user circuits make it back to the host processor at the end of a processing job. To implement this, each user circuit generates a `pending_data` signal which indicates if, based on recent instructions received, valid output data will be generated. When the MCF receives a flush command, the output multiplexer waits until all channels indicate that no more data will

[1] The Virtex-6 actually contains two identical instances of the ICAP that are both connected to the same reconfiguration data path. Only one ICAP can be active at a time. The other instance exists solely for redundancy purposes.

be generated. Once this condition is met, the output PCIe stream is padded with null data until the total length of the output is a multiple of 4kB.

4.5 User Circuit

To evaluate the effectiveness of the MCF, each user circuit was populated with an instance of the CSC engine. Each channel's CSC engine can be divided into three parts: reconfigurable CSC region, static CSC logic, and the CSC abstraction layer. The reconfigurable CSC region is a black-box module reserved for the CSC's 3D/4D modules. Originally identified by S. Patil, the 3D and 4D modules of the CSC engine are never used simultaneously and can be swapped in and out of the engine using partial reconfiguration. The reconfigurable region's logic can be changed using the MCF's PR instruction which feeds a partial bitstream into the ICAP. The static CSC logic contains all non-reconfigurable portions of the original CSC engine. Finally, a new CSC abstraction layer is added to the user circuit.

The abstraction layer decodes two additional MCF instructions and generates the appropriate signals native to the CSC engine. First, to configure the engine's internal registers, a `Register_`Write instruction is used. Each instruction word contains the internal address of the CSC register as well as the data to be written to it. A second command, called the `Pixel_Burst` instruction, initiates a burst of pixel data that is to be converted. This instruction includes the *burst count* value (bits [31:0]) described in section 4.4 which indicates the number of pixels that will be sent following the command.

Instead of implementing a fixed CSC engine in the user circuit, it is also possible to define the entire channel as a PR region. By doing so, any type of stream-oriented

processing engine can be loaded into the channel at run time provided that enough logic resources have been allocated.

4.6 Physical Layout

After completing the capturing the MCF logic into Verilog HDL, the source code is synthesized into netlists. All of the static logic (PCIe, MCF, and static CSC modules) are synthesized into one static netlist. Both 3D and 4D variants of the reconfigurable CSC module are synthesized into their own respective netlists as well. These netlist are all imported into Xilinx's PlanAhead tool which facilitates partitioning of the FPGA's resources for partially reconfigurable designs.

The five reconfigurable black-box regions defined in the static netlist must first be constrained to physical regions within the Virtex-6 FPGA. The size of each reconfigurable partition is defined by the maximum amount of resources required by the 3D or 4D modules. Because both modules require heavy-use of block RAMs, this is the driving factor of the size of the reconfigurable region. During the implementation of the design, it was found that the placement of reconfigurable regions (RRs) had a significant influence on whether or not the PCIe reference design's logic would be able to meet its timing constraints. The best results occurred when the RRs were placed on the outskirts of the FPGA fabric as shown in Figure 4.12. Doing so prevented timing-critical PCIe logic from being bisected during placement.

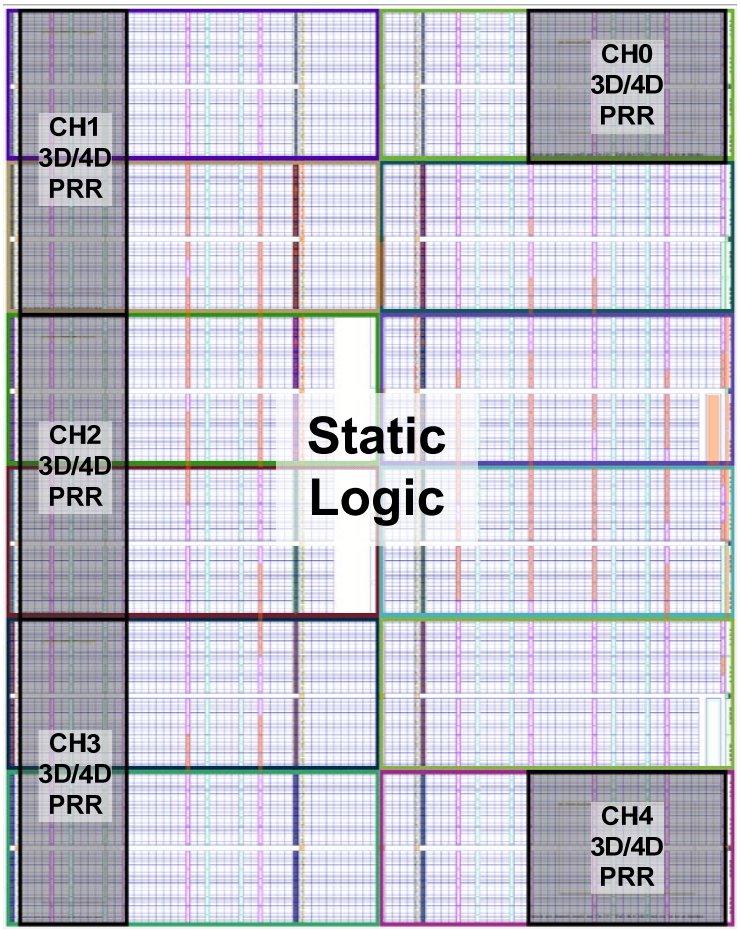


Figure 4.12: Floorplanned Virtex-6 FPGA

Another version of the MCF was attempted where the entirety of each channel was a reconfigurable region. Although this would inevitably increase the sizes of partial reconfiguration bitstreams, it would enable the function of the user circuits to be changed entirely. Unfortunately, sizing the PRRs to be able to fit an entire CSC engine into each resulted in insufficient resources for the remainder of the static logic. By constraining each CSC engine within a partition, the design tools were no longer able to pack the logic together as tightly as they did with the 3D/4D PRR version.

4.7 Software Tools

To facilitate development and testing of the MCF, several C programs were written. Each of the programs is designed to handle a specific portion of the MCF test process: Generation of the test vector, transmission of the vector to the FPGA, and the interpreting the resulting output vector (Figure 4.13). All of the programs (except for *vecSender*) are written using standard C libraries so they can be compiled without modification for both Windows and Linux hosts.

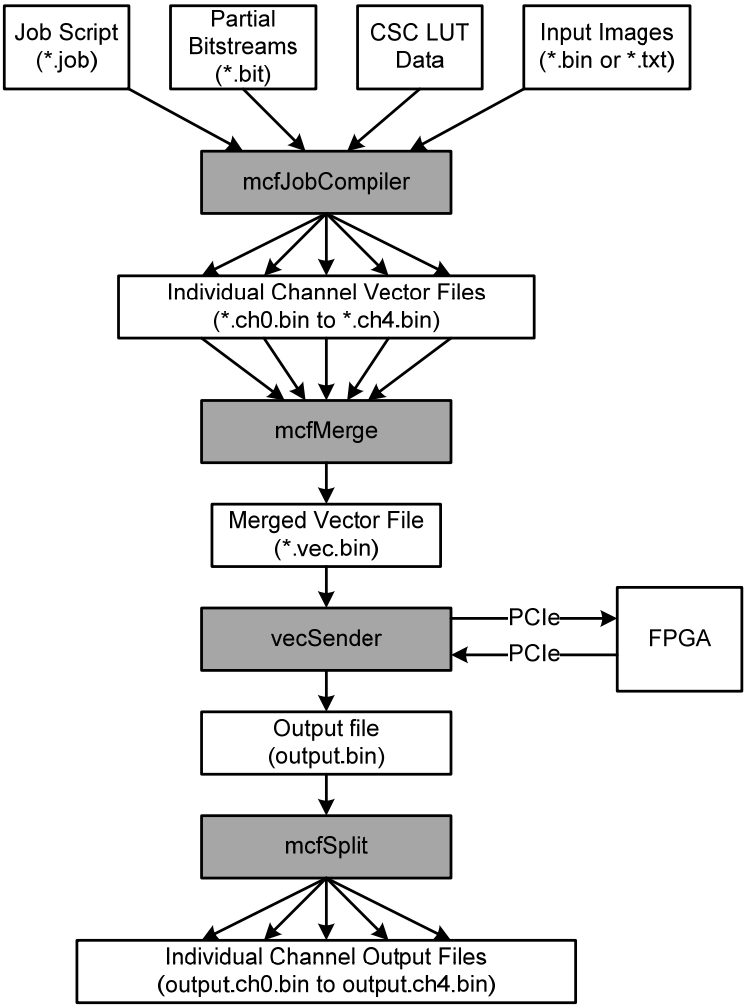


Figure 4.13: The MCF Toolchain

One of the most significant programs written for the MCF Toolchain is the *mcfJobCompiler* program. The program allows processing jobs for the MCF to be easily scheduled and described textually using basic commands. In this *job script*, operations to be performed for each channel are described. If an operation requires data from an external source, such as an image file or a reconfiguration bitstream, the relative path to the file is included. An example of a generic job script is included in Appendix A. The *mcfJobCompiler* program interprets the provided job script and combines any external data sources to generate individual vector files for each channel. These vector files contain a binary stream of packets that each channel will receive.

After the job compiler generates the channel vector files from the job script, they are fed into the *mcfMerge* program. This program sequentially interleaves the packets from each channel vector and combines them into a single vector file. The resulting vector file contains the exact data that is sent through the PCIe link to the MCF. In addition to merging the channel vector files, *mcfMerge* performs several other operations. Before copying packets from each channel vector, the program prepends a channel sync command along with the appropriate number of no-operation (NOP) instructions to the beginning of the stream. This ensures that all of the following packets are properly aligned to their respective channels. The *mcfMerge* program also interprets each channel's vector files as it copies them and ensures that simultaneous PR operations do not collide. If this condition is detected, NOP are inserted to stall the start of another channel's PR command. Finally, the end of the merged vector file is padded with additional NOP instructions to ensure that the total length is a multiple of 4096 bytes, the packet size defined by the PCIe driver.

Next, the *vecSender* program which was originally written by R. Toukatly during his research, sends the merged vector files through the PCIe link to the MCF. Any data received from the MCF is also written to a binary output file for later interpretation. Aside from performance enhancements described in section 4.1, only one major change was made to the *vecSender* program to facilitate the evaluation of the MCF's performance. This will be discussed in further detail in section 5.2.

The output stream received from the MCF consists of a collection of packets with header bytes followed by packets of data from the corresponding channels. To decode this output stream, the *mcfSplit* program is written. *mcfSplit* de-packetizes the output file and splits into a separate file for each channel's output. Doing so makes interpretation of each channel's output more straightforward.

The last program in the MCF Toolchain (not shown in Figure 4.13) compares the output vector files with known good results in order to verify that the MCF is operating properly. In the original job script, an enhancement to the syntax allows for the paths to the expected output files to be captured. The *mcfJobVerify* program interprets the Job script and compares the split output files with the expected outputs.

4.8 Validation Procedures

One important aspect of the design methodology was that the components of the MCF were tested as they were completed. Since the MCF was divided into several functional units, each could be simulated upon completion using Xilinx's ISim to verify proper operation. This verification process was done by writing custom behavioral Verilog testbenches that stimulate the input signals of the component under test and

verify the outputs. Being able to simulate the MCF's components also provides virtually unlimited visibility of internal behavior which proved to be invaluable during any debugging processes. To take full advantage of the simulation software, the MCF code was written to be able to be simulated in its entirety as well.

Unfortunately, not every aspect of the MCF can be simulated. The PCIe reference design used to send data to and from the framework was not designed to be simulated. To remedy this, a behavioral Verilog emulation of the PCIe link was written that reads and writes vector files to and from the file system just like the *vecSender* program does. Vector files are streamed in and out of the MCF through an interface identical to the actual PCIe FIFO interface. The PCIe emulator code also was designed to mimic the latencies and gaps in the data stream observed in the actual hardware. Using this PCIe emulator, it is possible to easily integrate the HDL simulations of the MCF with the MCF software toolchain.

Another feature of the MCF that cannot be simulated is the partial reconfiguration of user circuits. Because of this, the reconfigurable regions could only be simulated with a static configuration.

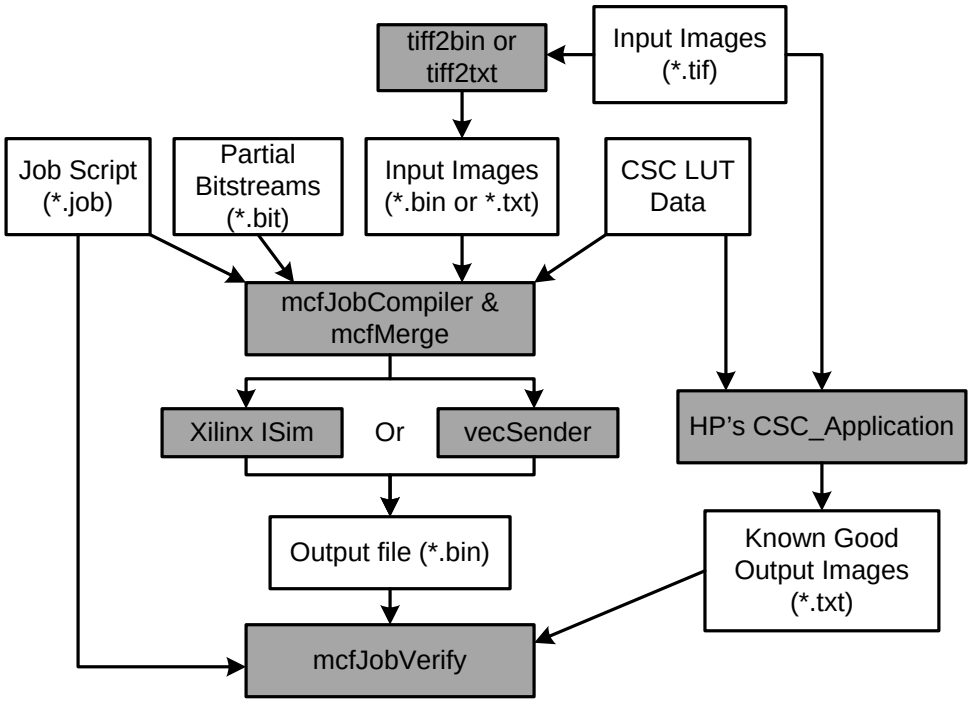


Figure 4.14 : MCF Verification Process

Now that the entire MCF can be simulated using Xilinx's ISim, the framework's functionality can be verified in both software and in hardware. In order to fully verify that the CSC within the MCF is functioning properly, a verification software flow is developed (Figure 4.14) where converted images are compared to known good outputs. These verification output images are generated using a software version of the CSC engine that was provided by HP. This *CSC_Application* accepts a set of register configuration data and an input image file. The application processes the image's pixels according to the configuration and outputs the converted image file in a TIF format. To simplify the verification process, all images are converted into a hexadecimal text file format using a MATLAB conversion utility. These "known good" outputs are generated for each configuration to be tested and saved for future verification.

Since an output vector from an MCF channel can contain multiple converted images within a single job, an additional program is developed to simplify the

verification process. The original job script file used to generate the input vectors also includes paths to the expected "known good" output files for each channel's operations. The *mcfJobVerify* program extracts these paths from the job script and uses them to verify the computed images received from the MCF. The program reads the split output channel vector files and compares the images to the known good ones. Verification results are printed to the terminal, indicating if any invalid packets were received.

Chapter 5: Results and Discussion

In this chapter, various aspects of the multichannel framework are evaluated. In the first section, methods and procedures used for verifying correct functioning of the MCF are discussed. Next, the computational performance of the MCF is measured and analyzed using several different operating conditions and scheduling methods. In the last section, a brief overview of the logic utilization of the MCF along with the CSC engine is presented. Power consumption estimates generated by the Xilinx tools are tabulated as well.

5.1 Validation of Design

The actual verification of the MCF was done in three phases. First, a static version of the MCF with the CSC engine was tested. Each channel's user circuit contained a fixed configuration of the CSC engine using only the 3D processing module. By doing so, the entire project could be simulated using Xilinx's ISim to confirm that the MCF's logic is behaviorally correct. One shortfall of a behavioral simulation is that it does not simulate the effects of timing delays for signals. For this reason, it was still necessary to perform a hardware test of the MCF using the physical Virtex-6 FPGA. Once the design was verified in simulation, the MCF was synthesized and implemented. Since this variant of the MCF did not contain any reconfigurable partitions, the entire compilation process could be done within Xilinx's ISE suite without requiring the use of the PlanAhead software. The hardware implementation of the MCF was tested from the Linux host machine and verified using the procedures described in section 4.8.

The static hardware test phase was able to verify that the MCF and CSC engine functioned correctly in hardware. The only component that was not verified was the MCF's PR capabilities. For this testing phase, a stripped down version of the MCF is implemented that does not contain the CSC engine in its user circuits. Instead, the user circuits are synthesized as black boxes that can be loaded with a configuration at runtime. For simplicity sake, two trivial variants of the user circuit logic are designed: one that bypasses any burst data through the channel and another that doesn't. When partitioning these regions in PlanAhead, only several slices were required to implement these variants which greatly reduced the time taken for the implementation phase of compilation. To test the PR capabilities of the MCF, a job script was written that effectively turns on and off a data loopback through the MCF by sending the partial bitstreams for the reconfigurable regions. By observing which data was returned through the PCIe link, it was possible to verify that the MCF's PR burst commands were functioning properly.

The final phase of verifying the MCFs functionality was testing the completed system using the CSC engine and reconfiguring each channel's 3D/4D module. The job script written for this test was not written for optimal scheduling, but so that each channel would experience a reconfiguration from the 3D variant to the 4D variant and vice versa. Once again, since PR operations cannot be captured by a behavioral simulation, ISim was not able to be used to verify the design. Instead, the implemented design was tested only by the hardware and verified against the expected output images using the process described in section 4.8.

5.2 Performance Evaluation

Computational performance of the MCF was evaluated by observing the effective throughput when processing a single image of various sizes. The worst case scenario was assumed where each channel's reconfigurable region must be configured prior to the start of pixel conversion. Since pixel operations are data-independent of each other, a single image can be divided between the five available channels to maximize the throughput. The architecture's theoretical throughput capability has been calculated by assuming that the input stream can always provide packets of data to the MCF at its maximum rate of 250 MHz. As expected, small images did not result in high processing throughput due to the significant configuration overhead. In addition, images smaller than 460k pixels cannot be scheduled to use all 5 available processing channels. As the image size increases, configuration overhead becomes less significant and the processing throughput approaches its theoretical maximum of 250 megapixels per second.

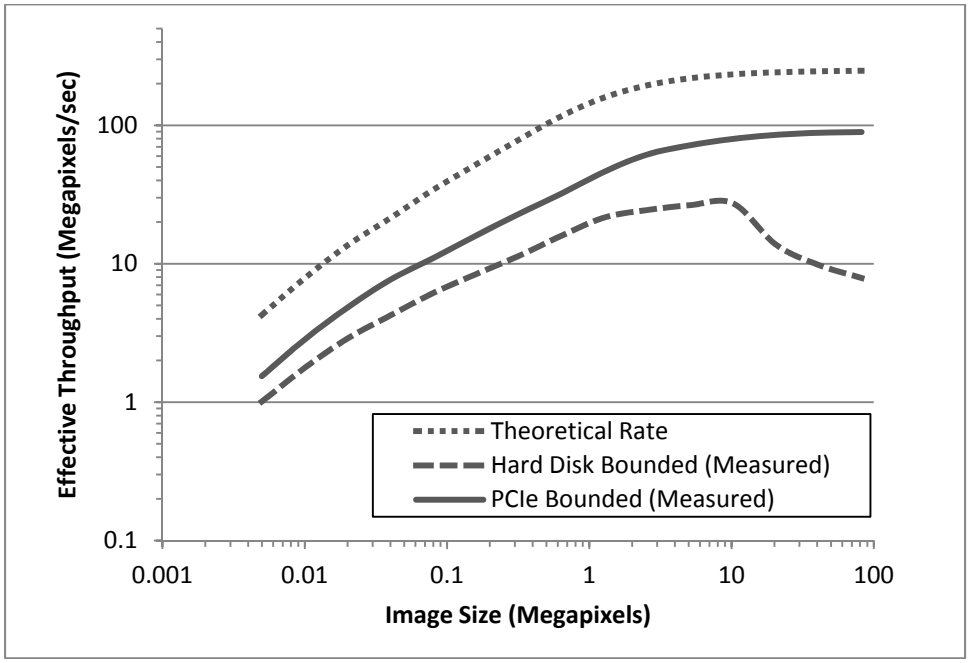


Figure 5.1: Effective Throughput vs. Image Size

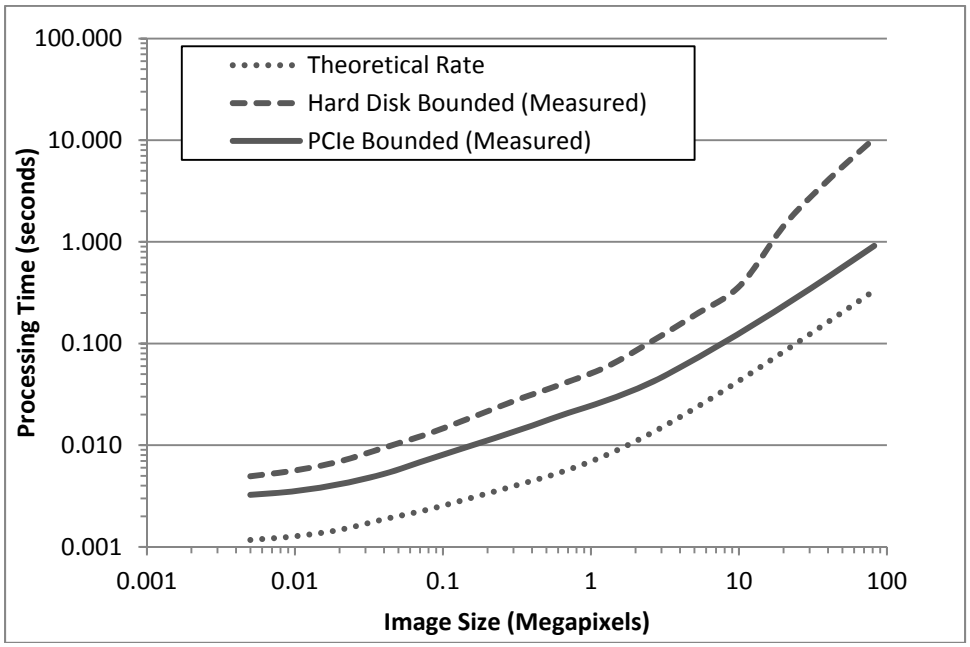


Figure 5.2: Processing Time vs. Image Size

The same test cases were performed in hardware by sending the packet stream to the MCF from a Linux host machine using the PCIe interface. As shown in Figure 5.1 and Figure 5.2, actual hardware latencies result in overall performance nearly an order of magnitude less than the ideal case. This occurred because input image data had to be read directly from the host machine's hard disk (HD) instead of the main memory. This became especially significant in images larger than 10 megapixels where performance dropped off significantly. The performance peaked at 27 megapixels per second and fell off rapidly due to the exhaustion of buffered stream data. After this point, the host machine had to fetch the stream data directly from the hard disk which has significantly larger access times than main memory. Performance in previous designs not using the MCF has suffered similarly from host-machine latencies. To circumvent the HD latencies, the *vecSender* program is modified so that instead of reading a vector file from the file system, it can generate pseudorandom image data on the fly and send it to the

MCF. Any data returned from the MCF is discarded. Since *vecSender* no longer performs any file operations, this completely eliminates any hard-drive accesses. The resulting performance is consistently 1/3 the speed of the theoretical architecture capability as a result of PCIe-related latencies.

Framework Architecture	*Active Pipelines*	*Operating Frequency*	*Throughput (Mpix/sec)*		
			Arch. Capability	*PCIe*	*PCIe+HD*
ASIC (no framework)	1	167 MHz	167	-	-
R. Toukatly's Dual-pipe	1 (2) [2]	50 MHz	50	15.9	7.9
MCF	5	50 MHz	250	89.6	27.3

Table 5.1: Peak pixel processing pate comparison under various throughput conditions

Due to the pipelined design of the CSC engine, its performance is directly proportional to the data rate in and out of the core. For all implementations discussed, the CSC reads a single 8-byte pixel input and produces a converted 6-byte pixel output in each clock cycle. As shown in Table 5.1, the original ASIC CSC is capable of processing up to 167 million pixels per second which directly correlates to the pixel-bus clock of 167 MHz. In previous designs not using the MCF, the CSC engine is constrained to a clock speed of 50 MHz. Even with the dual-pipe design proposed in R. Toukatly's research, only one CSC pipeline is able to process pixel data at a time. One significant advantage of the MCF is that user circuits in one channel can operate completely independent of user circuits in any other channel. With five instances of the CSC engine instantiated within the MCF running at a 50 MHz clock each, an effective pixel processing throughput of 250 megapixels per second can be achieved. Measured performance with PCIe and HD latencies are also included in Table 5.1.

[2] Two CSC pipelines were implemented however only one could process image data at a time

Another advantage of the MCF is that by using an instruction-based design, the CSC's control and register bus signals are handled within the IO abstraction unit without consuming extra data bandwidth from the PCIe interface. Since pixel data is sent to the CSC core using a burst command, the only data overhead is from the instruction word that initiates the processing operation. Data sent through the PCIe interface in previous designs not using the MCF not only included pixel data, but also a snapshot of all of the CSC's control and register bus signals. While this simplified migration from ASIC to FPGA, each clock cycle snapshot required 128-bits with at most 64 of the bits being pixel signals. This resulted in half of the data sent from the host machine being redundant during regular pixel processing.

Lastly, tests were performed on the MCF evaluating the effectiveness of being able to use multiple channels to process images. For this test set, three identically sized images (images A, B and C) are scheduled to be processed using different CSC operations. Image A is processed using a 1D conversion which does not require the use of the 3D/4D module (it is bypassed) and images B and C are processed using a 3D and 4D conversion respectively. Since images B and C require the use of the 3D/4D reconfigurable module, it must be loaded with the proper bitstream prior to image processing. It is assumed that the configuration of the module is not known at the start of the test.

Figure 5.3 illustrates the channel scheduling for each test case. PR operations are indicated with grey boxes and image processing operations are labeled with their respective image name. For the small 50k pixel images, startup configuration for images B and C takes only slightly less time than the image processing itself. Because of this, it

cannot be justified to schedule the use of more than two channels for such small images. The 500k and 10M pixel sizes for the other two test cases are large enough to justify utilizing all five available processing channels. Each test case is also compared against a scheduling using only one channel. Based on the resulting stream lengths, the theoretical pixel processing rates were calculated for each case. Actual computation times in hardware were measured as well.

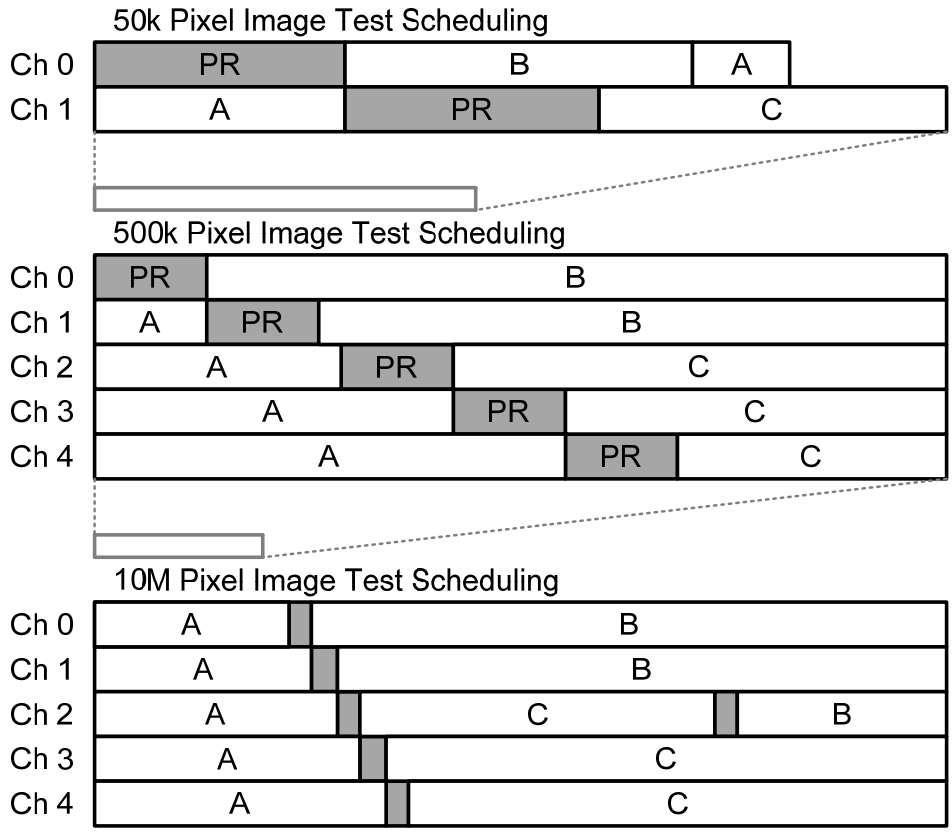


Figure 5.3: Overlapped Processing Test Case Scheduling
(Dashed lines illustrate relative task scale)

As shown in Table 5.2, significant theoretical improvements to processing throughput can be achieved by utilizing the parallelism of the MCF. In each case, increasing the number of channels used for an operation improved processing throughput. With all five channels in use, processing time was reduced by nearly 5 times when

converting large images compared to using just a single pipeline. As expected, processing speed suffered when tested in hardware due to interface and hard-disk latencies. The test case with 10 megapixel images on a single pipeline suffered especially from these latencies due to the large size of the file that had to be read from the hard disk and transferred. This resulted in a significantly slower processing speed for that particular test causing the relative speedup to be greater than the theoretical maximum of 5.

Image Size	*# Ch. Used*	*Theoretical Mpix/sec*	*Theoretical Rel. Speedup*	*Measured Mpix/sec*	*Measured Rel. Speedup*
50k pixels	1 2	28.67 48.89	1.71	5.34 7.91	1.48
500k pixels	1 5	46.54 197.51	4.24	9.98 24.17	2.42
10M pixels	1 5	49.81 246.39	4.95	1.45 11.17	7.72

Table 5.2: Overlapped Processing Results

5.3 Logic Utilization and Power Consumption

When observing the total resource utilization of the implemented design (Table 5.3), it is clear that logic overhead for the MCF itself is minimal. The MCF support logic, consisting of the input & output shifters and instruction decode logic, accounts for approximately 2% of the design and only 1% of the Virtex-6 ’s (xc6vlx240t) resources (Flip-flops and LUTs only). The PCIe adds considerable resource requirements due to the added complexity of the interface.

Table 5.4 further breaks down the logic distribution for the reconfigurable regions and their utilizations for the 3D and 4D module variants. Due to floor plan geometries, it was required to allocate an excess amount of slices to also enclose the required block RAM primitives. It is also important to note that due to the different partition geometries of channels 0 and 4, the logic utilizations differ slightly.

	Slices	FFs	LUTs	BRAM	DSP48	BUFG	BUFR	MMCM
MCF	2546	1857	2447	0	0	0	0	0
PCIe	12094	26721	20568	75	0	11	2	2
User Circuits: Static	11265	12865	25385	160	0	0	0	0
User Circuits: PRRs	9440	75520	37760	160	256	0	0	0
Total Used:	35345 (94%)	116963 (39%)	86160 (57%)	395 (95%)	256 (33%)	11 (34%)	2 (6%)	2 (17%)
R. Toukatly:	17138 (45%)	35297 (12%)	42463 (28%)	195 (47%)	64 (8%)	10 (31%)	2 (6%)	2 (17%)
Available in xc6vlx240t:	37680	301440	150720	416	768	32	36	12

Table 5.3: FPGA Resource Utilization

	Slices	FFs	LUTs	BRAM	DSP48
RR: Ch 0&4	1840	14720	7360	32	32
RR: Ch 1-3	1920	15360	7680	32	64
Utilized: 3D	952	908	3805	32	16
Utilized: 4D	1208	1298	4828	32	24

Table 5.4: Reconfigurable Region Resource Allocation & Utilization

After the implementation of the MCF, static and dynamic power approximations were collected from post-implementation reports. The estimated power consumption of the Virtex-6 FPGA under projected clock conditions for each module is shown in Table 5.5. These values are calculated automatically by Xilinx's tools during implementation and make assumptions on logic activity based on their own internal heuristics. Once again, the MCF support logic incurs minimal power consumption overhead to the design, requiring only 0.34% of the total dynamic power.

		mW
Dynamic Power (By component)	MCF	9
	PCIe	2590
	User Circuits: Static	33
	User Circuits: PRRs	24
Totals	Dynamic Power	2656
	Quiescent Power	6756
	Total	**9412**

Table 5.5: Power Consumption Estimates

Chapter 6: Conclusion

In this thesis, a methodology for managing multiple partially reconfigurable datapaths has been developed and evaluated. In order to evaluate the multichannel framework developed, a color space conversion engine is implemented in an FPGA. By leveraging the framework's features, image processing is able to be improved when compared to previous efforts using reconfigurable techniques. The main contribution to previous research efforts is that the framework provides a lower-overhead method of implementing the CSC engine in an FPGA. This is in regards to both FPGA logic utilization and overhead in the data stream. The multichannel framework developed also makes high-bandwidth data processing possible without requiring the user-logic to operate at a prohibitively high clock rate. This reduces the need for increasing the pipeline depth of an ASIC core being migrated to an FPGA. Because of this, the multichannel framework is able to improve previous efforts' performance by up to 5x and allows for significantly more flexibility by including fully-reconfigurable user circuit channels.

The MCF was evaluated using a complete testing platform using an FPGA connected to a Linux host PC using a PCI-Express link. The platform was used to both verify proper operation of the developed framework as well as measure its performance. Both theoretical ant measured results show that the MCF is able to drastically increase processing speeds using multiple simultaneous channels while overlapping reconfiguration operations.

Some disadvantages of the design were observed as well. One shortfall discovered was that the Virtex-6 FPGA is only capable of performing a single partial reconfiguration operation at a time. This restriction can result in a significant delay in the MCF in the event that all of the channels must be reconfigured upon startup. During hardware tests, significant latencies from the PCIe interface and hard-disk access times within the Linux host machine were observed. This resulted in significantly decreased MCF performance. For this reason, a more closely coupled embedded system is preferable.

In order to further improve the MCF, future research will target Xilinx's upcoming ZYNQ platform. ZYNQ combines reconfigurable FPGA fabric along with a dual core ARM Cortex CPU on the same silicon die. The reconfigurable logic in ZYNQ contains several times more resources than the Virtex-6 allowing for significantly larger designs. By including a hard-wired CPU on the same silicon die, the FPGA fabric can be coupled significantly closer to the host processor. This would significantly reduce latencies and would enable the MCF to operate at its full potential.

References

[1] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 26, pp. 203-215, 2007.

[2] D. Dya, "Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite," ed: Xilinx Inc., 2011.

[3] "Stratix V Device Overview," ed. San Jose, CA: Altera Corp., 2012.

[4] "7 Series FPGAs Overview," in *DS180*, ed: Xilinx, Inc., 2012.

[5] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.,* vol. 34, pp. 171-210, 2002.

[6] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," *Computers and Digital Techniques, IEE Proceedings -,* vol. 153, pp. 157-164, 2006.

[7] E.-A. Esam, G. Ivan, and E.-G. Tarek, "Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing," *ACM Trans. Reconfigurable Technol. Syst.,* vol. 1, pp. 1-23, 2009.

[8] D. Dye, "Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite," ed: Xilinx, Inc., 2011.

[9] R. J. Fong, S. J. Harper, and P. M. Athanas, "A versatile framework for FPGA field updates: an application of partial self-reconfiguration," in *Rapid Systems Prototyping, 2003. Proceedings. 14th IEEE International Workshop on*, 2003, pp. 117-123.

[10] M. W. Chamberlain, "A software defined HF radio," in *Military Communications Conference, 2005. MILCOM 2005. IEEE*, 2005, pp. 2448-2453 Vol. 4.

[11] A. Malagamba. (2006, *ISR and Xilinx Roll Out Ready-to-Wear SDR*. Available: http://www.eejournal.com/archives/articles/20060228_sdr/

[12] P. S. Ostler, M. J. Wirthlin, and J. E. Jensen, "FPGA Bootstrapping on PCIe Using Partial Reconfiguration," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, 2011, pp. 380-385.

[13] Y. Cho, S. Navab, and W. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering." vol. 2438, M. Glesner, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 2002, pp. 337-357.

[14] Y. H. Cho and W. H. Mangione-Smith, "Fast reconfiguring deep packet filter for 1+ gigabit network," in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, 2005, pp. 215-224.

[15] H. C. Young and H. M.-S. William, "Deep network packet filter design for reconfigurable devices," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 1-26, 2008.

[16] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," in *Field Programmable Logic and Applications, 2005. International Conference on*, 2005, pp. 644-647.

[17] R. Toukatly, "Dynamic Partial Reconfiguration for Pipelined Digital Systems: A Case Study Using A Color Space Conversion Engine," Master's Thesis, Dept. of Electrical Engineering, Rochester Institute of Technology, 2011.

[18] J. Hibbits, "An Evaluation of the Application of Partial Evaluation on Color Lookup Table Implementations," Master's Thesis, Dept. of Electrical Engineering, Rochester Institute of Technology, 2012.

[19] S. Patil, "Reconfigurable Hardware for Color Space Conversion," Master's Thesis, Dept. of Electrical Engineering, Rochester Institute of Technology, 2008.

[20] J. Galindo, "A Novel Partial Reconfiguration Methodology for FPGAs of Multichip Systems," Master's Thesis, Dept. of Computer Engineering, Rochester Institute of Technology, 2008.

[21] R. S. Berns, F. W. Billmeyer, and M. Saltzman, *Billmeyer and Saltzman's principles of color technology*, 3rd ed. New York: Wiley, 2000.

[22] "Virtex-6 FPGA Targeted Reference Design User Guide," ed: Xilinx Inc., 2010.

Appendix A: Example MCF Job Script

```
# This is a line-comment.
# Any blank lines with or without comments will be ignored
# Each line describes an operation to be sent to a single channel.
# A line starts with the name of the operation followed by any related parameters.

# Job files can also be interpreted by mcfJobVerify which compares the output with the expected
# output defined in the job file. mcfJobVerify commands are preceded by a '#@'
# Available verification commands are:
# #@outputs <file> - Compares output with specified file
# #@skip <number n> - N packets are expected to be generated. Skip them in verification.
#
# See channel 3 for examples.

#=================================================================================================
channel 0                       # All the following commands are for channel 0
PR channel0_bitstream1.bit      # Generates vector data that loads the specified PR bit file
cscLUT ../HP_3d                 # Loads CSC LUT data. Point to a directory of text hex files
cscImage test_image.bin         # Sends image data to CSC from a binary file.
cscRandomImage -n 1000 -s 1234 # Sends a pseudorandom image. 1000 pixels, using a seed of 1234
NOP 256                         # Insert 256 NOP cycles. (In case other ICAPs would still be busy)
PR channel0_bitstream2.bit      # Another PR operation
cscImage test_image2.bin        # Another Image

#=================================================================================================
channel 1    # Descriptions of tasks for another channel
PR channel1_bitstream1.bit
cscImage test_image.bin

#=================================================================================================
channel 2    # generates the channel file but it will be empty since no tasks were assigned to it

#=================================================================================================
channel 3
cscImage input_image.bin    #@outputs output_image.bin # comments can be added after another '#'
cscImage ignored_image.bin  #@skip 512 # skips 512 packets
#=================================================================================================
channel 4
```

Appendix B:
Hardware and Software Used

Hardware

- FPGA Development Board
  - Xilinx ML605
  - FPGA Family: Virtex-6 LXT
  - Device: xc6vlx240t-1ff1156-1
  - Programming Interface: JTAG over USB
  - Debugging Interface: UART over USB
- Development and Implementation PC:
  - OS: Microsoft Windows 7 (x86, SP1)
  - CPU: Intel Core 2 Duo, 2.66 GHz
  - RAM: 3 GB
- Testing PC:
  - OS: Linux Fedora 10 (2.6.27.5 Kernel version)
  - CPU: Intel Core 2 Duo, 2.40 GHz
  - RAM: 2 GB
  - PCI-Express slot populated with ML605 FPGA card.

Software

- Windows 7 Development PC:
  - Xilinx ISE Design Suite: 13.1 System Edition (incl. April 2011 patch)
  - ISE Project Navigator
  - PlanAhead (incl. PR license)
  - iMPACT
  - Cygwin
    - GNU C Compiler
    - GNU Make
  - Tera Term v4.64
- Linux Fedora Testing PC:
  - GNU C Compiler
  - GNU Make