FPGA Ethernet Platform for Communication, Debugging, Testing, and Rapid Prototyping

Peter Andrew Thorpe Lieber

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Brad L. Hutchings, Chair
Brent E. Nelson
Michael J. Wirthlin

Department of Electrical and Computer Engineering

Brigham Young University

August 2011

ABSTRACT

FPGA Ethernet Platform for Communication, Debugging,
Testing, and Rapid Prototyping

Peter Andrew Thorpe Lieber
Department of Electrical and Computer Engineering
Master of Science

FPGA-CF is an open-source, portable, extensible communications framework that consists of a small hardware core (less than 600 slices) and and a host-software library/API (Java and C++). It enables a host PC to transmit data at 120 Mb/s to Xilinx-based FPGA boards via Ethernet using standard inter-networking protocols (UDP/IP). A custom lightweight connection-oriented protocol guarantees reliability. The hardware core is directly connected to the Xilinx internal configuration port (ICAP) and supports all ICAP functionality. The core also provides an extensible user-channel interface and provides up to 15, 8-bit user-data channels that can be connected to user circuitry (configurable by the user). The host software API supports both Java and C++ and provides high-level functionality for making connections and transmitting data. The utility of the system is demonstrated by implementing an on-chip test/debug system using FPGA-CF.

Keywords: FPGA, communication, ethernet, network, stack, debug, prototyping

ACKNOWLEDGMENTS

I would like to acknowledge those who have helped me complete this work. First, I thank University of California's Information Sciences Institute East for their generous financial support. I could not have completed this work without it. I would also like to thank the Electrical and Computer Engineering department at Brigham Young University for their equipment and facilities.

I express my appreciation for the time my committee chair, Dr. Hutchings, spent with me reading, editing, and improving this work. I thank my other committee members, Dr. Nelson and Dr. Wirthlin for their time reviewing my manuscript. I also thank my fellow lab members in the Configurable Computing Lab for putting up with me.

I especially thank my wife, who's relentless encouragement and love made it possible for me to keep going and not loose sight of the goal. I thank my mother, who puts everything into perspective. I thank my family and friends for their words of encouragement. I thank God for blessing me with this opportunity to have a great education and learn from the best.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Most commercially available Field Programmable Gate Array (FPGA) development boards provide a relatively standard set of I/O interfaces such as Ethernet, USB, and RS232. During initial development of FPGA designs, developers need some way to communicate with the FPGA for on-chip testing; however, there is no widely-used, open-source communication framework that can be easily integrated to facilitate communication between a hardware application on an FPGA and a software application running on a host computer. In FPGA development, a communication framework would include the hardware and firmware necessary for the FPGA as well as the software necessary for the host computer. Due to the lack of such a framework, developers typically develop a unique communication system for each application, often in an ad hoc manner. Developing communication circuitry and software on an app-by-app basis is often wasted effort given that many applications can be covered by the same subset of communication functionality. A framework that has the features needed for most applications would increase developers' efficiency. A typical developer or researcher wants a communication facility that can

- integrate quickly and easily with their application,,
- reconfigure and partially reconfigure the FPGA quickly,
- transmit data at high speed,
- control the application running on the FPGA,
- and observe the application's internal state.

A standard framework for communication that has these features would ease the overall development effort, provide a standard interface on which many tools and components could be built, enable advanced research into reconfiguration strategies, and make it easier to

share efforts between research groups. A well-designed framework would include a broader range of functionality. The framework's designer can devote more time to consider functionality and problems that may otherwise be overlooked when the communication system is developed from scratch for every project. The developer who uses the framework will then have more time to focus on his or her application. A simple interfacing mechanism should be included so the framework's learning curve is much less than the time to develop a custom communication system. The framework can be built with extendability in mind to allow new features to be added without sacrificing compatibility. Care can be taken by the framework designer to provide a space efficient and good performing communication system.

In addition to addressing the limitations of application specific and ad hoc communication systems, there are additional benefits of a standard framework. Testing of even small FPGA designs can be done easily in hardware. To do this, the standard interface of the framework can be quickly connected to the module being tested. Then, the host can send test data and verify results. This allows small portions of a design to be tested independently of the input/output (I/O) system of the application. The status of complex modules can be monitored. In many cases, status is indicated by LEDs built into the development board. However, many applications have very complex states which can be difficult to monitor using only the limited number of LEDs and DIP switches on a development board. This complex status can be sent through the communication framework. With advanced configuration access, new debugging tools can be built. These can be as simple as reading the state of flip flops in the design or implementing a complete debugging system with clock control. This debugging system, when implemented using configuration read-back supported by Xilinx, does not require intrusive instrumentation as with tools such as Chipscope [1].

The FPGA Communication Framework (FPGA-CF) is a general-purpose Ethernet-based communication framework that can be used to communicate with FPGA development boards. It includes both a hardware network stack for the FPGA and software for the host computer. The framework has the following attributes:

- supports 1000Base-T Ethernet,
- can be used with a variety of Xilinx FPGA devices,
- uses a standard socket-based Internet protocol,

- occupies a very small hardware footprint on the FPGA,
- for Xilinx devices it provides full access to all configuration modes, including partial reconfiguration, via Xilinx's Internal Configuration Access Port (ICAP),
- and can be easily augmented by the end-user.

The FPGA-CF is open source, extensible, and can easily be adapted to most Xilinx-based FPGA boards that provide a fast Ethernet port (100BaseT, 1000BaseT). The open source project can be found on Opencores.org at `http://opencores.org/project,fpga-cf`. The main hardware is implemented with generic hardware description language (HDL), so it can be used with other vendors' programmable chips such as Altera. The configuration features, however, are only supported for Xilinx devices. It can transmit user, configuration, and internal state data over a single Ethernet connection. The hardware in this framework includes the processing core that implements the network stack and the interfacing mechanism for transferring data to and from user logic. The software includes an application programming interface (API) to connect to and communicate with the FPGA and its resources.

1.1 Background Information

Field Programmable Gate Arrays (FPGAs) are digital logic chips that can be programmed in the field. Currently, FPGAs are produced by companies such as Altera, Lattice, Tabula, and Xilinx. FPGAs are widely used in many industries. They have especially had an impact in communications, aerospace, medical imaging, and Application Specific Integrated Circuit (ASIC) prototyping. One of the biggest advantages of using FPGAs is very fast and low cost development, especially compared to ASIC development. When developing a product that is not mass produced, FPGAs become very valuable because they can achieve much higher performance than software solutions, but avoid the high cost of ASICs.

To program FPGAs, users must provide a description of the circuit to the software tool flow provided by the chip vendor. This tool flow is designed to produce the programming files for a specific device (see Figure 1.1). The user provides their design as a hardware description language (HDL) or a schematic. The chip vendor's tool then generates a programming file called a bitstream. This bitstream is then sent to the FPGA through a configuration port on

the chip. To test the user's design, the user can either use this flow to run the design on the hardware, or simulate the circuit using a simulation program such as ModelSim. Designs are normally simulated many times before testing on hardware because internal signals within a circuit are not readily accessible on a clock by clock basis. Furthermore, simulation can be faster than running an entire tool flow followed by testing in hardware.

Figure 1.1: FPGA Tool Flow

Xilinx FPGAs feature a special Internal Configuration Access Port (ICAP). This port, rather than being accessed off-chip, can be connected internally to user designs. The ICAP port can be used to configure memory and on-chip resources. It also can be used to read the state of flip flops within the FPGA and the current configuration of the chip. A common use of this interface is to program parts of the FPGA with new circuitry without affecting the entire chip. This process is called partial reconfiguration. Although not in wide use, partial reconfiguration has been a popular topic among researchers. The ICAP also allows the current state of the FPGA to be read. While this feature has received less attention than writing configuration data, it can be used to implemented in-hardware debugging capabilities.

Xilinx provides a partial reconfiguration flow in their tools starting in version 12. This flow allows a user to specify dynamic modules within their design, and generate partial bit-streams to program these modules with different behavior. This programming is commonly done completely within the FPGA using an embedded processor, or over the JTAG port. While this work is primarily dealing with FPGA communication, the networking protocols used must be understood. Next, I will introduce the basics of networking terminology and protocols.

The networking protocols in use today are the results of years of experimentation and experience with the Advanced Research Projects Agency Network (ARPANET) [2]. Starting

out as many isolated networks, ARPANET needed a way to connect all these networks into a single network of networks, or Internet. Layers of protocols were introduced to separate the various networks' physical differences from the data being transmitted. Now, the Internet Protocol Suite [3] [4] defines each layer, its purpose and relationships between them. There are four defined network layers: link, Internet, transport, and application. Each layer is encapsulated within the next lower layer, hiding the details that are not relevant to higher layers. Figure 1.2 illustrates encapsulation and how these layers relate to one another.

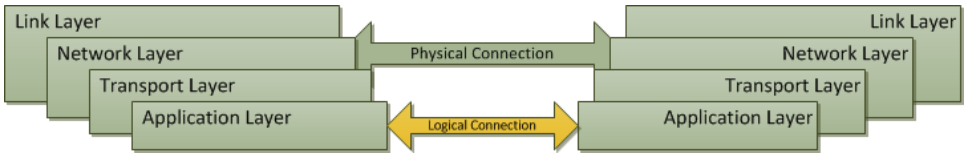

Figure 1.2: Network Layers

The link layer is the lowest level protocol that is transmitted directly over the physical network. A common example and the protocol used in FPGA-CF is Ethernet. This layer hides the details of the physical network from all other layers. The Internet layer is used to provide communication between any two hosts regardless of the link layer protocol used. The only network layer protocol defined in the Internet Protocol Suite [3] is the Internet Protocol (IP). The Internet layer provides logical addressing, packet fragmentation and reassembly, as well as type of service and security information. It is the transport layer's job to provide end-to-end support such as reliability and flow control. User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) are the two primary transport layer protocols used. The application layer is the top layer and provides the highest level of functionality and is closest to the user. Application level protocols include Telnet, SMTP, HTTP, DNS, etc.

Ethernet defines both the physical interconnect and the link layer protocol. It uses a 48-bit Media Access Control (MAC) address space to uniquely identify devices. Data is sent in frames which have the structure shown in Table 1.1. A source and destination MAC addresses identify what devices should respond and where to direct their responses. The Ethertype field indicates the type of data within the payload. At the end of each

Ethernet frame, there is a CRC32 field that is a value calculated from the entire frame, excluding preamble and start of frame (SOF) bytes. This field allows transmission errors to be detected. If an error is detected, the payload data is considered corrupted and is thrown away. The payload can contain a maximum of 1500 bytes. This maximum payload size is referred to as the maximum transmission unit (MTU).

Table 1.1: MAC Packet Structure

Preamble	*SOF*	*MAC dest*	*MAC source*	*Ethertype/Length*	*Payload*	*CRC32*
1 byte	1 byte	6 bytes	6 bytes	2 bytes	46 - 1500 bytes	4 bytes

In the Internet Protocol Suite, the Internet Protocol (IP) is the Internet, or network, layer protocol used to logically join many networks (which may or may not have similar link level protocols) into a single, unified, internetwork, or Internet. This protocol contains source and destination address fields using the 32-bit IP addressing scheme. Error checking is done with the header checksum field (see Table 1.2). As its name implies, the header checksum is only calculated over the header. The protocol assumes that the payload protocol will ensure data integrity. There are other capabilities included in IP that a link level protocol does not need to consider. The Time to Live field indicates how many hops, or routing nodes, a packet may traverse before it is considered expired and thrown out. The Fragment offset and flags fields allow a single IP packet to be "fragmented" over several link layer packets. This capability allows hosts to ignore the protocol restraints of the link layer. For example, IP packets have an MTU of about 65,000 bytes, while Ethernet has an MTU of 1500 bytes. An IP packet cannot fit into a single Ethernet frame, therefore, the IP packet is split into small fragments and sent over multiple Ethernet frames. There are other features of IP that are not included here for brevity [5].

User Datagram Protocol (UDP) is a transport layer protocol that provides connectionless communication between hosts. This protocol, and all others in the transport layer and up, use the IP layer for addressing purposes. This protocol uses a port number to get

Table 1.2: IP v4 Packet Structure

DWord	*Byte 0*	*Byte 1*	*Byte 2*	*Byte 3*
0	Version, Header Length	Differentiated Services	Total Length	
1	Identification		Flags, Fragment Offset	
2	Time to Live	Protocol	Header Checksum	
3	Source Address			
4	Destination Address			
5	Options (optional)			
6+	Data ...			

data to the correct application on the destination host (see Table 1.3). Applications on the host register with the operating system to receive data over a specified port.

Table 1.3: UDP Packet Structure

DWord	*Byte 0*	*Byte 1*	*Byte 2*	*Byte 3*
0	Source port		Destination port	
0	Length		Checksum	
5	Data ...			

Transmission Control Protocol (TCP) is the connection oriented protocol to compliment UDP. It incorporates flow control, connections, and guarantees delivery. The structure of a TCP is show in Table 1.4, but a discussion is out of the scope of this work. TCP is the most common protocol because of its ubiquity and ability to guarantee delivery. This is also a very complex protocol and therefore has much longer latency when compared with UDP. IP, UDP, and TCP form the basis of the Internet and are implemented by all modern computer systems. This common ground provides a way to connect any device with any other. This common protocol stack is useful for FPGA developers in communicating between chips and from host to chip.

The network, transport, and application level protocols all use the network level address (IP address) for identifying each other. Therefore, when initiating communication, only the IP address is known, not the link layer or hardware address. For this reason, a special protocol that is aware of both the link and network layers is used. Address Resolution

Table 1.4: TCP Packet Structure

DWord	*Byte 0*	*Byte 1*	*Byte 2*	*Byte 3*
0	Source port		Destination port	
1	Sequence number			
2	Acknowledgment number			
3	Data offset, reserved	Flags	Windows Size	
4	Checksum		Urgent pointer	
5	Data ...			

Protocol (ARP) allows hosts to request the hardware address when only the network address is known. Usually a ARP request is broadcast to all devices on a network requesting information about a certain IP address. Anyone who is aware of the correct hardware address (usually the host with the IP address in question) then replies with a link to network level address mapping. Another protocol, Internet Control Message Protocol (ICMP), is used to transmit IP network level status and error information. A host supporting ICMP can take advantage of its features, but can still function in an IP network without it. ARP is more critical to smooth operation because without it, applications must know both the IP and hardware address, which defeats much of the purpose of the network level abstraction.

1.2 Previous Work

No current framework exists that allows FPGA developers to easily send configuration and data to and from the FPGA over a network. Various TCP/IP and UDP/IP implementations for FPGAs have been proposed and reported in the literature. Some are implemented in software running on a soft or embedded processor, while others are custom logic implementations. These implementations are not, however, part of a communication framework. They are described here and compared to FPGA-CF with respect to their ease of use and flexibility to show that there is a need for a framework aimed at speedy integration with a user design.

Custom logic implementations of network stacks can be very fast and space efficient; however, much more time is required to develop custom hardware. In software solutions, the solution can be much more flexible and take less time to develop, at the cost of resources and performance. If a soft processor is used to implement the network stack, it will take

much more FPGA resources than a hardware implementation of the same stack. In addition, using a soft processor will most likely not reach the performance level of a good hardware implementation. There are small soft processors available (i.e. Picoblaze [6]), but they sacrifice performance for a small footprint. In the case of the Picoblaze processor, it has two cycle instructions and only has an 8-bit data path. At the other end of the spectrum, using an embedded processor can save FPGA fabric resources, but may increase either the price of the FPGA (when the processor is a hard resource built into the FPGA) or increase system cost and complexity (when using a separate processor chip).

Table 1.5 lists previous implementations of FPGA network stacks and their features and performance. The first metric used is the number of slices. This metric can be misleading due to the differences in architecture between FPGA families. It is, however, important to see relative difference in area consumed by each implementation. The type of slice is noted in parentheses. The next metric is the number of BRAMs used in the design. Again, each device family has different sizes of BRAMs, but this gives an idea of what kind of resources beside logic are needed for each design. In some instances, this speed is measured in an ideal environment, or there was no measurement taken. A '*' denotes a design that did not report measured speeds. The TP/area column is a measurement of throughput normalized by the area of the implementation. This metric gives a better comparison between implementations. TP/area is also important when a high throughput is desired with a low footprint, as is the case for my work. In addition to standard metrics, it is useful to compare the type and number of features that are supported and the ease of use for developers wishing to integrate the network stack into an application. A short list of features provided by each implementation is included in the last column.

Network stacks implemented directly in hardware can achieve very high space efficiency and/or performance. Dollas et. al. [9] proposes an open-source implementation of TCP/IP on an FPGA. This core implements the link layer (Media Access Control) as well as the network and transport layers (TCP/UDP/IP). It includes ICMP, TCP, UDP, and ARP support. An MII interface block connects the network logic to the physical interface chip (PHY). There are separate hardware modules implementing each protocol. The Ethernet send module and Ethernet receive module buffer then send or received then buffer

Table 1.5: Previous Work Comparison

Source	*# Slices*	*# BRAM*	*TP/area*	*Notes*
Bomel et al [7]			N/A	Power PC
Herrmann et al [8]	1580 (S3E)	2+	N/A *	UDP/IP, Includes MAC
Dollas et al [9]	10,007 (V2)	5	0.03	Complete TCP/IP Stack
Uchida [10]	2285 (V4)	22	0.42	TCP/IP, MAC, ICMP
Loefgren et al [11]	1584 (S3E)	5	N/A *	UDP/IP
Alachiotis et al [12]	91 (V5)	0	9.94	UDP/IP
Xilinx AppNote [13]	3737 (V4)	8	0.001	Microblaze Webserver

Ethernet frames. The ARP and ICMP modules construct and send replies to the Ethernet send module. The IP receive module routes the encapsulated data to the correct transport protocol (UDP or TCP) from the Ethernet receive module. The UDP and TCP modules have an application interface that can connect multiple hardware applications on the chip to the network core. The author mentions a limitation of his TCP implementation which is that it cannot accept and rearrange out of order datagrams. While packet reordering is a large part of TCP, without it, TCP can still be an effective protocol that guarantees and confirms delivery. It is unclear whether IP fragmentation is supported in this work. All this functionality comes at the price of FPGA resources. It consumes 10,007 Virtex-2 slices. The TCP subsystem consumes 84.4% of those slices. This demonstrates the cost of implementing TCP in hardware, even with the limitations imposed by this implementation.

Loefgren et. al. [11] presents three implementations of a basic UDP/IP stack: a minimal solution that supports 10/100 Mbps speeds and occupies 517 slices, a medium performing solution that supports more protocols such as ARP and ICMP and occupies 1,022 slices, and an advanced solution supporting 10/100/1000 Mbps speeds that occupies 1,584 slices. These speeds are the physical speeds of the connection, but not the performance of the entire stack as implemented. The actual throughput was not given by the author, but the maximum frequency of the circuit is given. With this frequency, a theoretical throughput can be inferred for comparison(see Table 1.5). The minimal implementation supports only basic UDP/IP communication with no support for ARP or ICMP. It also sacrifices gigabit Ethernet support and has a smaller MTU of 256 bytes. While the network layer supports

full duplex, the UDP logic is shared between the send and receive functions, making its performance between half and full duplex.

This type of implementation, according to Loefgren, is well suited for transmit-intensive applications such as data acquisition systems. The medium implementation supports many of the necessary functions to perform well in a normal network environment. It includes ARP, ICMP, and full duplex mode. However, it does not support gigabit Ethernet, RARP, or "TCP channel". The advanced implementation supports ARP, ICMP, RARP as well as "TCP channel" communication. "TCP channel" is simply a feature that recognizes TCP packets, and forwards them directly to the user application for processing, bypassing the built-in UDP processing. This fast implementation, while much larger, supports gigabit Ethernet and the maximum Ethernet MTU of 1518 bytes. Loefgren uses these implementations as a way to analyze UDP/IP stack parallelism, but does not provide a way for others to incorporate these cores into other applications. While they are not meant to be used by others, these implementations show that requirements can have an impact on the size and complexity of the network interface. Therefore, when high performance is not necessary, space can be saved in the network logic.

Herrmann et. al. [8] proposes a gigabit UDP/IP stack implemented on an FPGA. This implementation provides a smaller implementation than Dollas [9] or Loefgren [11], while still attaining the speed necessary for a physical link speed of 1000 Mbps. As with Loefgren, Herrmann did not report the actual measured speed of the stack. Herrmann implements the link layer through the network layer and includes support for UDP/IP only. The transmit and receive functions are independent and enable full duplex operation. Similar to Dollas and Loefgren, Herrmann implements each layer of the network stack as separate blocks. He achieves a system clock frequency of 125 MHz which has the potential to achieve maximum throughput on a gigabit network. The speed is due to a very simple and compact implementation of only the basic UDP/IP network stack. Without support of ARP, however, it is more difficult to communicate with the FPGA using a regular operating system network stack.

Alachiotis el. al. [12] has released an extremely small and fast UDP/IP implementation on opencores.org [14]. Their core operates at a measured raw throughput of 113 -

111 MB/s. They claim an extraordinarily small footprint of 67 occupied slices. It must be noted that this small footprint is achieved with two main drawbacks. First, it is only the UDP and IP protocols that are implemented. This core does not include the Ethernet MAC (link) layer nor does it include interfacing logic to a hard Ethernet MAC. It also does not implement any kind of ARP reply capability. This requires all software sending to their device to know the MAC address of the FPGA. Normal operating system network functions cannot be used, making software interaction a more complex task. Second, the design is not flexible. The hardware interface only supports one application, and the protocol implementation requires changes at the HDL level. For applications where raw speed is the major concern, and network compatibility is not, this is a very good implementation of UDP/IP.

Tomohisa Uchida [10] presents an implementation of TCP/IP that is substantially smaller than Dollas's work. Uchida accomplishes this by only implementing the minimum protocol set required to allow a PC to communicate with it using standard operating system socket functions. His implementation, called SiTCP, is targeted for devices with limited hardware size like detectors and other front-end devices. Much like other implementations described in this section[11] [9], the MAC layer is implemented as part of the system as well as ARP and ICMP protocols. Each protocol and their respective transmitters function independently. An arbiter block to the MAC layer transmitter is used to combine all these transmitters. Uchida has the netlist available for academic use. This is a very sophisticated and high performance solution. It can achieve a claimed 949 Mb/s throughput. Like most network interface implementations reviewed, it only provides the network stack. It does not provide FPGA specific features such as configuration access. There is also no API provided to easily communicate with FPGA resources and user logic.

Xilinx provides an example implementation of a simple web server, including the TCP/IP stack [13]. The web server as well as the entire network stack is implemented on an embedded Microblaze soft processor. The main advantage is that most of the code needed for network communication from user logic to a host PC is provided by Xilinx. Using the examples provided as a template, users can connect their own hardware and control it from a webpage. Xilinx's integrated development environment, EDK, is used to generate all the hardware and program the FPGA. This method uses a lot more resources

than dedicated hardware implementations. Unlike other processor based implementations of network communication systems [7], this uses a soft processor, which resides in regular FPGA resources to implement the processor, rather than using an embedded PowerPC processor. Perhaps the biggest disadvantage of using this example design is the transfer rate. Xilinx claims only 500 Kb/s throughput, which is three orders of magnitude less than most other network stacks for FGPAs.

Network based communication links have also been developed as part of reconfiguration frameworks. These have been implemented using an embedded or soft processor. Bomel's implementation [7] specifically supports his partially reconfigurable platform. Tan [15] and Blodget [16] present frameworks for reconfiguration of FPGAs that are implemented using embedded microprocessors. In both cases, an on-chip microprocessor manages reconfiguration of the device. In cases where the systems can be accessed via a network, this can be a waste of space because much of the configuration management code can reside on a PC rather than in a processor embedded in the FPGA [15] [16], leaving more logic for use by the application. The configuration bitstream storage and configuration command generation could all be implemented on the PC, with raw data being transferred from the network directly into the ICAP. For autonomous systems, however, a self-contained configuration management system, like Blodget's, is very useful.

Bomel et. al. [7] uses an embedded PPC405 processor to implement the network stack. This is a common method to speed the development process because many of the components are already available. However, this approach may poorly utilize hardware if the processor's sole function is to implement a network stack. Bomel's implementation is part of a reconfiguration framework. This framework allows areas of the chip to be reconfigured using the on board PPC405 processor. Bitstreams are sent from a host computer through the network interface using a custom network level protocol. The bitstream data and custom protocol headers are inserted directly into Ethernet frame payloads. This has the advantage of less latency due to the absence of other layers. This could present problems, however, if a user wants the device to coexist with other IP level devices on the same network.

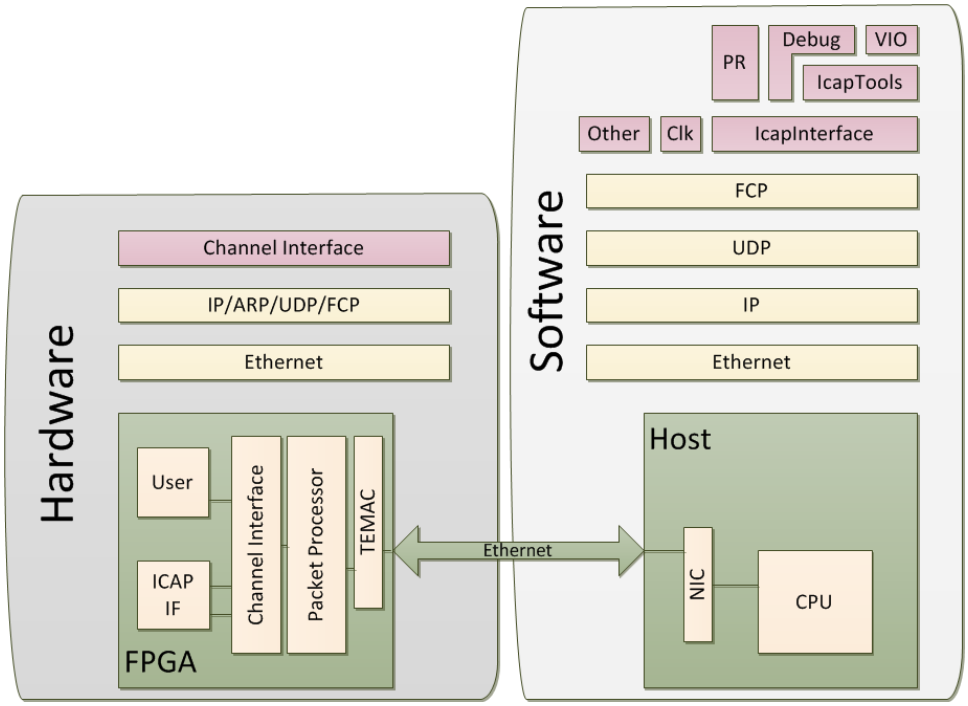

Figure 1.3: FPGA Communication Framework

1.3 Overview

Previous network stack implementations in HDLs and embedded processors are too large or too specialized. Using large network interfaces on small chips leaves little room for the application logic when embedded processors are not available or affordable. One common method to reduce resources in a hardware implementation is to eliminate features or protocols that are not needed for the application. Reliable transfer is a feature that takes up a lot of space, and so it is often not included. While very fast, specialized interfaces may not work on some networks or may be tied to certain applications. Many of the previous implementations were implemented and optimized on old (Virtex 2) architectures. These architectures also did not provide a way to port the design between parts and development boards. They also did not provide source or hardware cores with an API to aid host-side application development. More importantly, previous network interface implementations for FPGAs do not provide a well defined, complete, and general framework for common tasks such as sending data to and from hardware on chip, and reading and writing configuration data.

My solution is to use standard network protocols to connect the host to the FPGA over Ethernet. It uses a special-purpose packet processor to provide the network stack. Designed specifically for this communication framework, the processor retains low-level programmability, achieves high performance and occupies much less hardware than a Microblaze, for example. The main goals and features of this system are:

- Transfer arbitrary data to and from the FPGA
- Control applications running on the FPGA
- Access to a configuration interface
- Low latency
- High throughput
- Simple host API
- Simple hardware interface
- Extensible
- Minimal area and complexity

The packet processor implements a standard UDP stack and also allows the user to easily define dedicated hardware ports that transmit data directly to user hardware. These user-defined hardware ports follow the Xilinx Local-Link interface [17]. The packet processor communicates to the outside world via the FPGA Communication Protocol (FCP), a connection-oriented protocol that is layered upon UDP, that was created specifically for this project. UDP was chosen as the base protocol because it is fast, standard, simple, and because a hardware implementation of it will occupy far fewer resources than more complex protocols such as TCP. FCP guarantees order and delivery of packets sent to the packet processor from the host via a sequence number. It is not necessary for users to know anything about FCP because a simple host-side API is provided. However, an assembler for the packet processor is provided to allow users to change or augment FCP and its implementation without direct HDL modification.

Chapter 2

Implementation

The goals listed in the previous chapter have a large impact on how each part of my framework, FPGA-CF, is implemented. Achieving low latency, high throughput, and minimal area requires that all parts of the system should be simple. The network stack needs to avoid complex features such as packet fragmentation. The processor implementation requires careful consideration of the balance between performance and simplicity. In the sections that follow, I discuss how each part of FPGA-CF is implemented and, when appropriate, how these decisions impact the goals of the framework.

2.1 Network Protocol

The two most common protocols used in networks today are UDP and TCP over IP. While TCP is very reliable and connection oriented, it is very complex and can consume many resources [9]. UDP, on the other hand, is very simple and small but does not guarantee delivery and is connectionless. The use of IP for the network layer is required in FPGA-CF to create a system that would connect easily with most host computers. For the transport layer, I needed to devise a way to achieve reliability but not consume excess resources.

I considered two main approaches to achieve the reliability of TCP, and the simplicity of UDP. The first method is to develop a new transport layer protocol to replace UDP that preserves simplicity but has some sort of delivery guarantees similar to TCP. This approach would introduce a very limiting factor: the built in network protocols of most operating systems could not be used unless special network drivers were written. The very common and easy to use socket programming would also not be useful because it is based on UDP/TCP/IP network stacks. The second method is to develop a new application layer protocol. This method allows the host API implementation to use the standard network

interfaces of the operating system and add functionality on top of them. I chose the second method and developed the FPGA Communication Protocol (FCP).

FCP is implemented as an application layer protocol over UDP/IP. The packet structure of FCP is shown in Table 2.1. FCP communicates over UDP using port 12289. This protocol has commands for connecting to the FPGA, sending data, and receiving data. The channel field allows the FPGA to route communication among different hardware channels connected to user hardware. A sequence number enables packet order and delivery guarantees to be implemented.

Table 2.1: FCP Packet Structure

Version	*Command*	*Channel*	*Sequence*	*Length*	*Data*
4 bits	4 bits	1 byte	2 bytes	2 bytes	0 - 1024 bytes
0	0 - Data 1 - Ack 2 - Connect 3 - Con Ack 4 - Data Req 5 - Data Ack	1 - 255 0 for commands	Sequence Number	# data bytes	The data

To simplify the FPGA implementation of FCP, the protocol is designed to provide host-initiated communication only. All data transfers, both to and from the FPGA device, are initiated by the connected host. To send data, the host sends a Data packet with the data and the destination channel number. The FPGA then responds with an Ack packet. To receive data from the FPGA, the host sends a Data Req packet. In this case, the Length field contains the number of bytes requested, not number of bytes in the packet. The FPGA then responds with a Data Ack packet, which contains the data requested.

Before sending any data packets, the host first must send a Connect packet to the FPGA. When the FPGA receives a Connect packet, it stores the connecting host's IP-MAC address pair. It uses this information to filter subsequent packets by IP address, only accepting those from the connected host. The connected host's IP-MAC address pair is the

only pair stored by the network stack and assists the Address Resolution protocol at the network layer.

The fact that only the host can initiate transfers can present a problem when the device's data availability is not known. It is left up to the user application to account for this. If the host-side application can be programmed to know exactly how much data should be available on each channel on the FPGA, this does not present a problem. However, if the amount of data cannot be known, the user logic can implement a simple polling feature. For example, one channel can be used to retrieve the amount of bytes available on other channels. The host application can then first request a fixed amount of data from this "polling channel" then request the right amount of data from the channel of interest.

Throughput can be limited if the host must wait for an acknowledgement after every packet sent to the FPGA before sending another. To increase throughput, a sliding window protocol is used for FCP. To conserve FPGA resources, only the host side will have a window greater than 1. This eliminates the need to store unacked packets sent from the FPGA. Moreover, in the present implementation, the FPGA only sends acknowledgements to host requests, so it will not benefit from a sending window greater than 1.

The data size of each FCP packet could theoretically be as large as UDP will allow. However, to save even more resources, I elected to limit FCP data packets to 1500 bytes. The main purpose for the limit is to ensure that each FCP packet is encapsulated in a single link layer frame. With the assumption that the entire packet will reside in a single frame, data integrity issues can be offloaded to the lower layer protocols.

The UDP and IP layers both include a checksum in their headers to protect against data corruption. The Ethernet layer includes a CRC value at the end of the frame for the same reason. The hardware Ethernet MAC calculates the CRC automatically. The IP checksum is calculated only over the IP header, so it can be calculated without buffering the packet. The UDP checksum, however, is calculated over the entire packet, including data. To calculate this, the packet would need to be buffered and then the checksum inserted into the header. To save FPGA resources, the UDP checksum is omitted and replaced with zeros as allowed in RFC 768 [19]. The data is still validated by the Ethernet CRC value calculated

over the entire frame. This is sufficient because each FCP packet will always fit within a single Ethernet frame.

The IP protocol is the most complex in the stack. With careful selection of implemented features, it can be simplified to save resources. Since FCP's maximum size is less than Ethernet's MTU, we can assume that packet fragmentation will never be needed. This eliminates a lot of buffering to reorder and reassemble IP packets. The IP protocol does, however, allow a packet to be marked "Don't Fragment." My IP implementation asserts this flag on all outgoing packets and does not support IP fragmentation. All IP packets containing FCP payload should, if possible, set IP's Don't Fragment flag (DF). This flag prohibits fragmentation and ensures that data received by the FPGA will not be fragmented [5].

To support operations on normal IP networks, Address Resolution Protocol (ARP) must be implemented. This allows a host connecting via FCP to find the FPGA's MAC address when only the IP address is known. The FPGA can reply to arbitrary ARP requests but only stores the connected host's information. For this reason, we call this approach to ARP "Connection-oriented ARP." This limits the FPGA to sending packets to the connected host. This is not a problem because in a host-initiated protocol such as FCP, the client (FPGA) will only be sending packets to the connected host.

2.2 Packet Processor

The packet processor implementing the FCP/UDP/IP protocol stack is a custom programmable state machine. Other options, such as a soft microcontroller or microprocessor, were considered. Implementing the stack on a microprocessor would be an easy method since much of the code has already been written and is widely available. However, to use an entire processor for a simple protocol would waste resources. A microcontroller is small, but it can lag in performance compared to a custom state machine. My packet processor combines the speed and application specificity of state machines, while retaining the flexibility of a microcontroller.

2.2.1 Feature Selection

Initial feature selection was done by enumerating the main, high-level operations needed to implement the network protocols. An advantage to designing a custom processor is the ability to create custom units for particular tasks. The protocols implemented were analyzed and only the minimum required features are implemented while ensuring correct functionality.

The processor has some standard features such as an ALU and register file. In addition to the standard hardware units, some custom functions are implemented in the processor. A FIFO is included to store values temporarily when they will be used unchanged and in the same order later. A checksum unit is included for fast calculation of the IP checksum.

2.2.2 Hardware Units

Although most of the work that is done by the packet processor is data movement, there are some simple calculations to be done as well as checksums. A 16-bit Arithmetic Logic Unit (ALU) is implemented. As shown in Figure 2.1, it has two 16-bit operands and one 16-bit result output. The ALU supports four operations: addition, subtraction, bitwise OR, and bitwise AND. The operation is selected with a 2-bit mode signal.

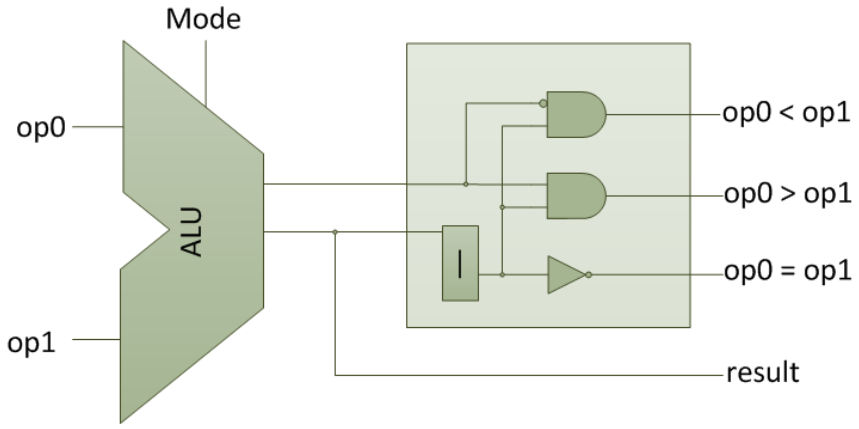

Figure 2.1: ALU Comparator Logic

To augment the ALU, a comparator takes the result of the ALU and a "compare mode" signal as input (see Figure 2.1). The result indicates whether the selected compare operation is true. The result, when the ALU is performing a subtraction, compares $op0$ with $op1$. The compare modes are: equal to ($op0 = op1$), greater than ($op0 > op1$), less than ($op0 < op1$), or not equal to ($op0 \neq op1$).

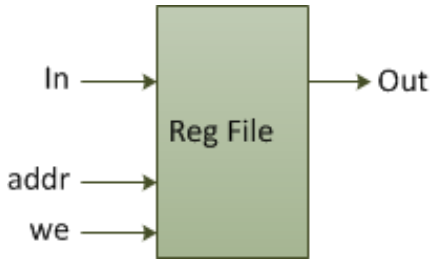

Figure 2.2: Register File

A register file and small SRL FIFO buffers are used for temporary storage. The register file holds sixteen 16-bit words. A single address line selects what word should be read or written. The "we" input causes the specified location to be overwritten. It is a read first architecture where the previous value is read when a read and write happen simultaneously. The register file is used to store connection state which includes the connected host's IP and MAC addresses as well as the current sequence numbers.

The FIFO buffers serve as short-term storage for the data that is used in the same order it is received. These buffers have a capacity of sixteen 16-bit words. They are built to take advantage of the shift register logic available in Xilinx parts. The SRLs are inferred with generic Verilog in order to support non-Xilinx parts. BRAM FIFOs are not used because only 16 words of storage is required. The SRL architecture, however, supports fixed length shift registers rather than FIFO behavior. In order to enable FIFO behavior, some extra logic is added to the basic shift register (see Figure 2.3). The address input of the shift register specifies the length, or what bit in the shift register should be presented at the output. To get FIFO behavior from the shift register, the amount of words currently in the FIFO is used as the address input. This allows the oldest (first) word to be retrieved when

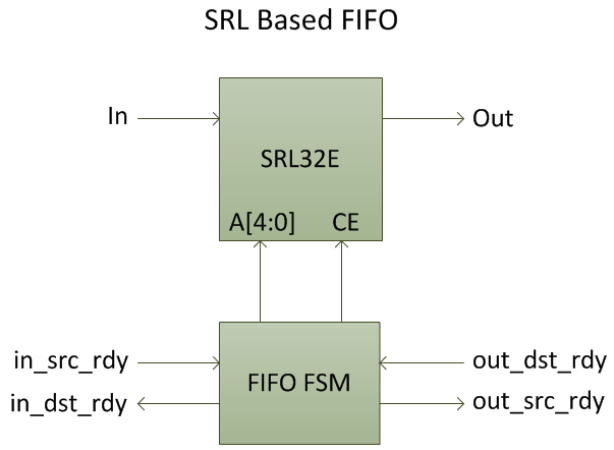

Figure 2.3: SRL Based FIFO

the FIFO is read. A small state machine as seen in Figure 2.4 increments the address when a word is added and decrements it when one is removed. When a word is written and read simultaneously, the address remains unchanged.

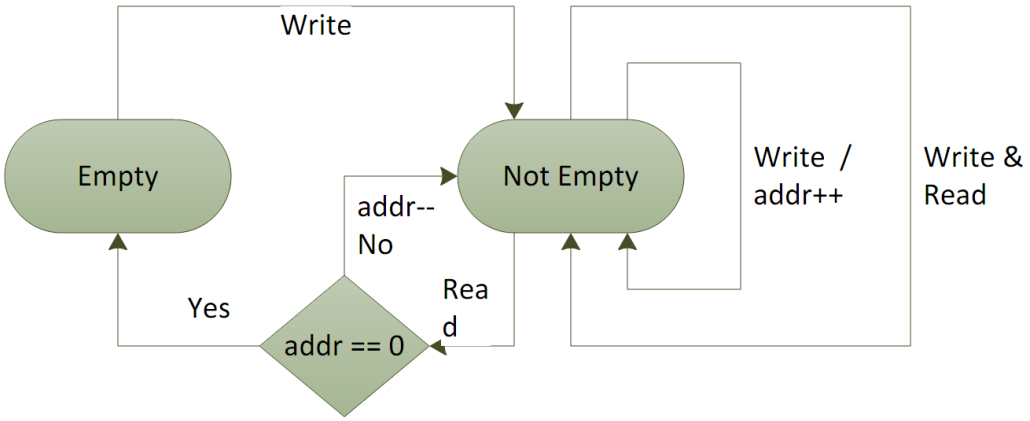

Figure 2.4: FIFO FSM

A special logic unit is available to calculate the finite checksum (FCS) field of the Internet Protocol header. The checksum used in the UDP/IP network protocol is calculated by finding the one's complement of the one's comlement sum of all 16-bit words. The checksum unit performs each ones complement addition in one clock cycle. If the ALU (which is a two's complement machine) were used, it would take two clock cycles. Having a

dedicated logic unit for checksums also allows the ALU data-path to be used for something else. The FCS unit has a data input to feed new 16-bit values. The 16-bit result output is the current checksum calculated thus far. There is also a single bit output that indicates that the current checksum is zero. This single zero test bit is useful for verifying checksums. To verify the checksum of the IP header, for example, the checksum is calculated over the header, including the checksum value. Once completed, if that checksum value was correct, then the result should be zero. Rather than comparing the 16-bit result with zero, the processor can simply use the zero test bit.

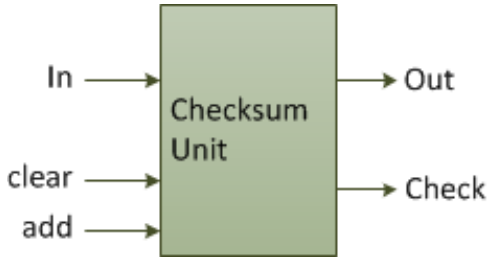

Figure 2.5: Checksum Unit

2.2.3 Data Path

The general structure of the processor data path is shown in Figure 2.6. There is a main data bus that serves as the input to all of the logic units (e.g. ALU, register file, etc.). All of the logic units' data outputs are fed to a master data multiplexer. This multiplexer controls what data is on the main data bus. This architecture allows for simple movement of data between any two logic units or ports. Using a single multiplexer is not sufficient for some operations, such as incrementing a value in a register. This is because the multiplexer would need to select the register file for the ALU operand, but also need to select the ALU result to write back into the register file. To solve this issue, the operands to the ALU are not taken from the main bus. Instead, each operand has a dedicated multiplexer.

A minimal selection of operands minimizes both multiplexer size and microcode width. Figure 2.7 shows the available data sources for each operand. The result from

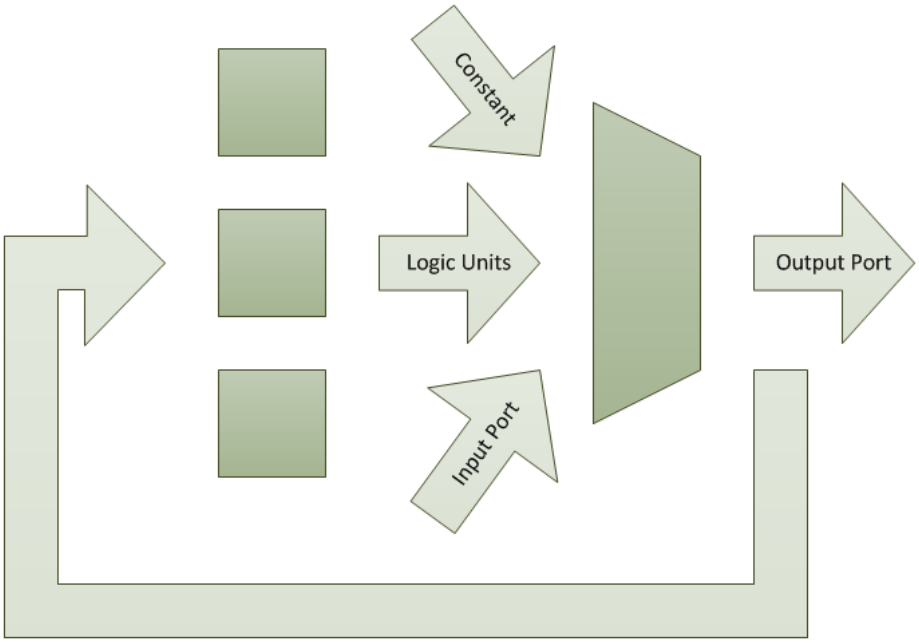

Figure 2.6: General Data Path

the ALU operation is also available to the register file through a dedicated feedback loop. This feedback data path allows a register to be altered while another instruction uses the main bus, such as data-movement. This feature is used to enable a Direct Memory Access (DMA) transfer of an arbitrary amount of bytes from one port to another. For this DMA transfer function, a register containing the amount of bytes left to transfer is decremented while simultaneously transferring the data from the input port to the output port. This instruction is executed until the register value is zero.

The processor transfers data to and from other logic via ports. The ports follow the LocalLink specification from Xilinx [17]. Each port has separate input and output addresses. These addresses control what port number should be active for input and output. Initially, only one address specified both the input and output address and was hard-coded into the microcode. This lengthened the microcode unnecessarily and did not allow for data to be passed directly from one port to another. To solve this problem, port address registers store the current active input and output ports (see Figure 2.8). These registers are loaded with the desired port numbers before doing port operations. This does require extra instructions

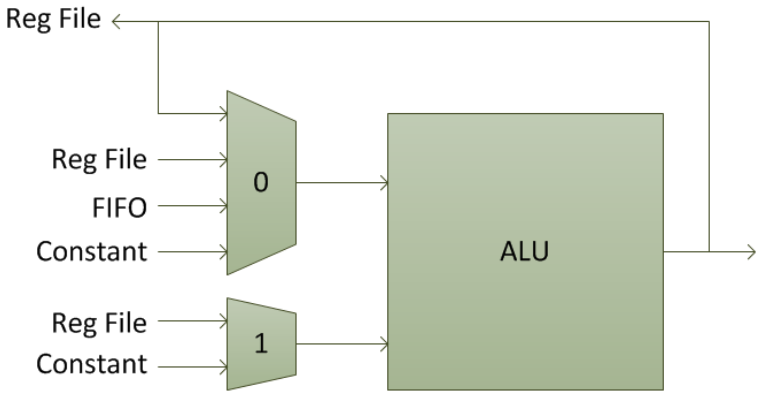

Figure 2.7: ALU Operands

to load the port address registers, but for long data transfers this happens only at the beginning.

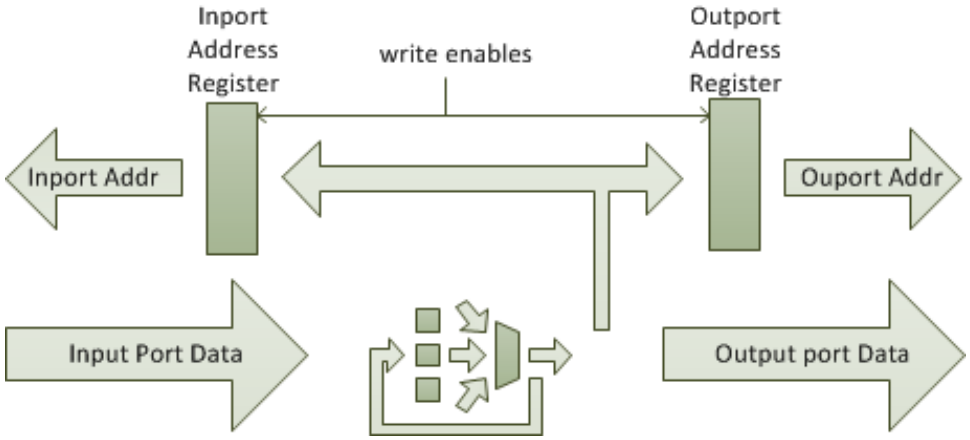

Figure 2.8: Input/Output Port Addresses

2.2.4 Microcoded Architecture

A microcoded architecture is used to implement the custom, programmable packet processor. Other methods were considered such as an encoded instruction memory and a standard state machine. Changes and additions to the protocol needed to be easily implemented. For these reasons, a standard state machine design is not used. Moving to an

encoded instruction architecture might introduce performance penalties and adding functionality still would require changes to the hardware and assembler. A hybrid approach is used that takes advantage of the speed and simplicity of a horizontal microcoded state machine, while allowing for easier development.

A traditional state machine moves through states in a predetermined manner. The outputs of the state machine control various hardware units within the processor, and the next state is determined by next-state logic. The packet processor has all these components. The instruction pointer is the state; the microcode is the outputs; and the condition logic is the next-state logic (see Figure 2.9). The packet processor differs from a traditional state machine only in the way the state machine is written. Rather than writing the HDL for the microcode (outputs) and the condition logic (next-state logic) by hand, it is generated by an assembler from a programming language developed specifically for the packet processor.

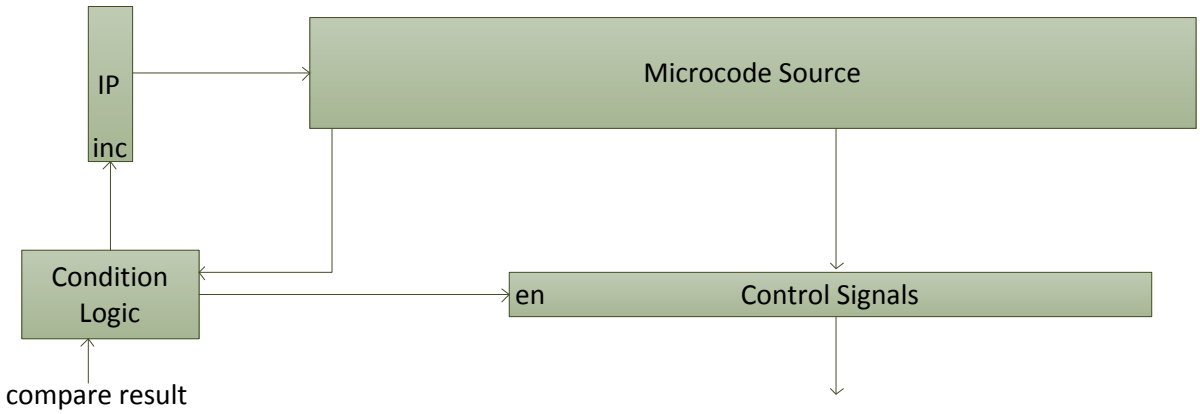

Figure 2.9: State Control

The microcode generated by the assembler specifies all control signals for the hardware units (ALU, Register File, etc.) as well as special signals to control the condition logic. Normally, the instruction pointer is incremented by 1 every clock cycle. Various condition and jump operations are available to control the state transitions. Conditions can be set to advance the state when it is true. There are three condition modes: 'when', 'if', and 'until'. The 'when' mode disables the microcode instruction (effectively executing a NOOP) until

the condition is true. The instruction is enabled once the condition is met, then the state is incremented. The 'if' mode only enables the instruction when the condition holds. Whether or not the condition held, state is incremented. The 'until' mode is the dual of the 'when' mode. In this mode, the instruction is enabled, rather than disabled, until the condition is met. Once the condition is met, the state is incremented as usual. In addition to controlling the instruction pointer, the condition logic can disable many of the control signals when a condition is not met to execute the current instruction. This prevents registers from being loaded or port from being accessed when, for instance, an 'if' condition is not met. A special jump instructions load a new instruction pointer value. Conditions can be associated with a jump instruction to create conditional jump behavior.

Figure 2.10: Microcode Development Flow

To control all the data paths and control lines for each unit, all the signals are fed from the microcode source. This microcode can be stored in either block RAM or distributed RAM (see Figure 2.9). Some additional logic is provided to allow for arbitrary state changes (jumps and resets). To ease the development process and allow for future upgrades, a simple assembly language is provided to generate the microcode source. This language is an embedded language based on Python. Each source and sink (ALU, Checksum, etc.) is presented as a Python class and each basic state type (instruction) as a Python function. The result of each function is a Verilog line of code representing the microcoded version of the function. For example, if we wanted to bring in a byte from port 4 and place it in register 8, we would first set the input port address to 4, then bring in the byte:

```
1: MOV( C(4) , IPR() )
2: IN( R(8) )
```

In this case, MOV and IN are functions that generate Verilog. C, IPR, and R construct class instances that the function uses to generate the correct microcode. The assembler supports instructions for data movement and for operating the ALU, checksum unit, FIFOs, and compare logic. The instructions supported are documented in Appendix B.

2.3 Channel Interface

An easy to use hardware interface is crucial to making a system that will save development time. User hardware modules can interface with the FPGA Communication Framework via an interface based on Xilinx's LocalLink interface. The packet processor presents a channel address along with a single bidirectional LocalLink interface. The channel interface (see Figure 2.11) decodes the channel address to enable signals for each channel. For each channel, a LocalLink in each direction and channel write and read enable signals are provided by the channel interface.

It is not necessary to use all 15 available channels for every design. A channel interface generator script is provided. This tool, chifgenpy, accepts one parameter: the number of channels to expose. The output of the tool is the channel interface implementation, channelif.v. After generating this file, the user can wire it to the FPGA-CF and the user logic.

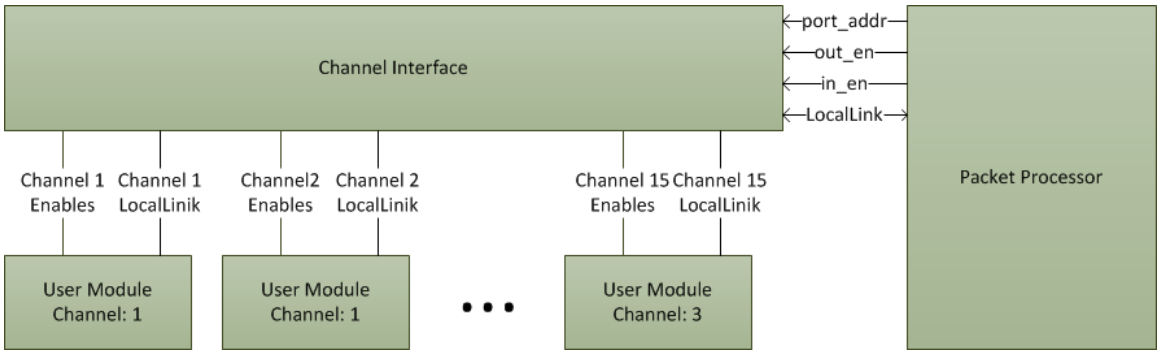

Figure 2.11: Channel Interface

Each channel's LocalLink interface functions as specified in Xilinx App Not XAPP691 [17] with the addition of global read and write enable signals for each channel. Data is

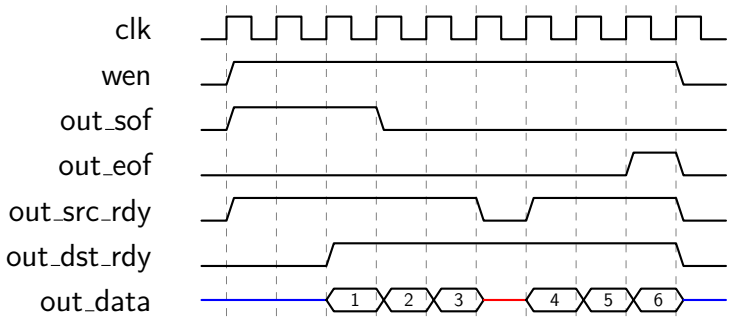

Figure 2.12: From Channel Interface

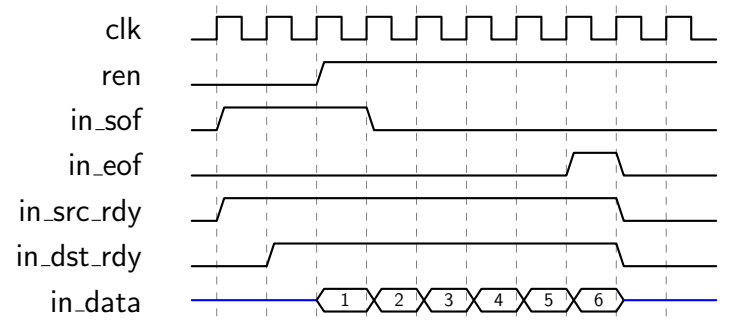

Figure 2.13: To Channel Interface

transferred to and from the channel interface via separate 8-bit input and output buses. Each direction has four control signals. The out_src_rdy and out_dst_rdy signal control when data is valid and data is accepted, respectively. Only when both signals are asserted is data considered transferred. The out_sof and out_eof signals are asserted on the first and last byte, respectively. These signals are often used to trigger user designs to begin processing. The read and write enable signals are global enables for that channel. For data to be valid coming from the channel interface, wen and out_src_rdy must both be asserted. Figure 2.12 shows an eight byte sequence being sent to the user logic.

The user logic generated signal, out_dst_rdy, gets asserted when it is ready for data. At that point, the eight bytes of data are transferred to the user logic. If, at any time, the out_src_rdy signal is de-asserted, the data is not valid and the user logic must wait for that signal to continue receiving data. For data transfers in the other directions, from user logic to the channel interface, the procedure is identical (see Figure 2.13).

2.4 ICAP Interface

Although the FPGA-CF is a complete communication framework by itself, the ICAP Channel Interface provided connects the ICAP to the FPGA-CF API over two FCP channels to provide easy access to this port. This will allow even more researchers to access this FPGA configuration port but with a higher level of access. For instance, rather than learning the ICAP protocol, if a user simply instantiates the ICAP Channel Interface, they can immediately have access to this port over an Ethernet connection. The API provides functions to download a bitstream, or read and write configuration registers. One might use this capability for partial reconfiguration. With the FPGA-CF and the ICAP Channel Interface, the user can do partial configuration over Ethernet. Using the partial reconfiguration tools provided by Xilinx, users can generate the partial bitstreams then download them using FPGA-CF.

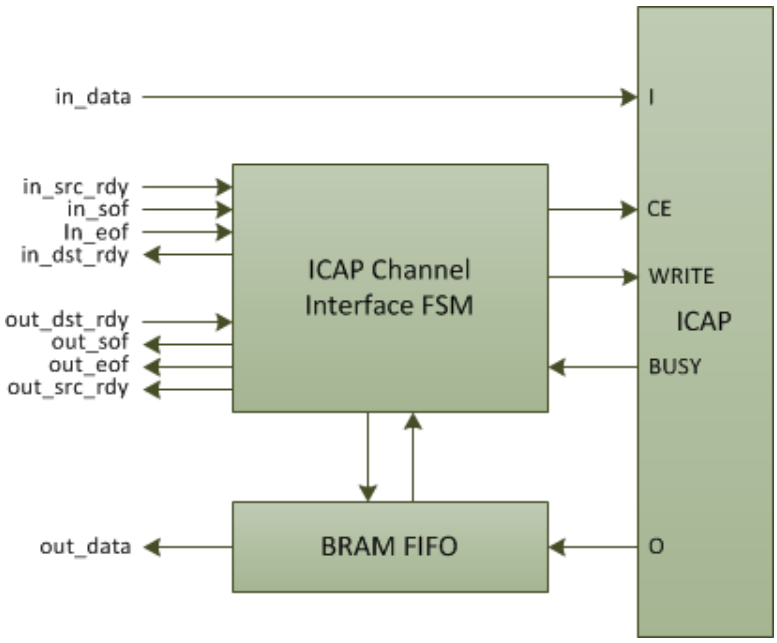

Figure 2.14: ICAP Channel Interface

FPGA-CF connects to the ICAP via two channels (see Figure 2.14). On one channel, called the write channel, ICAP data is transfered to the ICAP. On the other channel, the

read channel, data is requested and read from the ICAP. It is important to note that the ICAP interface is not part of the FPGA-CF itself but is an interfacing mechanism to allow communication with the ICAP through FPGA-CF. The ICAP Channel Interface takes care of the special signalling required by the ICAP. When writing to the ICAP, the write channel enable and src_rdy signals are simply combined to generate the clock enable and write signals for the ICAP. This direct connection to FPGA-CF is possible because the ICAP logic allows the CE signal to be lowered and pause writes.

To read from the ICAP, however, requires reading all available bytes without pausing. For this reason, a second channel is needed to send read requests. To read from the ICAP, the number of bytes to be read is sent over the read channel to the ICAP Channel Interface. A state machine then reads all the requested data from the ICAP into a BRAM FIFO. The data can then be read from the read channel to the host application. More details can be found in Appendix C.

2.5 Software API

Two APIs are provided with FPGA-CF. One, written in C++, runs on Linux. The second, written in Java, runs on any machine that supports the Java runtime. Both APIs were written to provide a simple interface with a low learning curve. An easy-to-learn software API will make it easier to communicate with hardware designs and allow more time to be spent in hardware development.

The C++ API is a low overhead implementation that does not support sliding windows. It requires the user to know the IP and MAC address of the FPGA. This lowers the time spent in OS calls and lowers latency of transfers. The C++ API has four functions: connect, disconnect, sendData, and requestData. These functions are all that is needed to do anything with FPGA-CF. The `connect` function sends a connection command, then receives the connection acknowledgement. It returns true when connected or false if it failed. The `disconnect` function simply cleans up host side memory.

The `sendData` function sends some number of bytes to the specified channel on the FPGA. It returns true if the correct acknowledgement is received, false otherwise. Similarly,

`requestData` sends a data request packet then receives the data acknowledgement packet. When the data is successfully received, the function returns true, otherwise it returns false.

I spent more time making the Java API more sophisticated. This is the suggest language to use with FPGA-CF. The Java implementation supports a sliding window, and runs on any operating system supporting Java. The API consists of a single, base API that can send and receive data over FCP channels. Higher level and application specific APIs are built to provide more intuitive interfaces to specific hardware modules such as the ICAP channel interface. The base API consists of five functions that support all operations in higher level APIs:

- `void connect(InetAddress address);`
- `void sendData(int channel, ArrayList<Byte> data, int numBytes);`
- `void sendDataRequest(int channel, int numBytes);`
- `ArrayList<Byte> getDataResponse();`
- `void disconnect();`

Connecting to the FPGA is very simple with this framework. First, an FCPProtocol object is created and then the connect function is called. Before any data transfer functions are called, the `isConnected()` function should return true. Below is an example:

Listing 2.1 Connecting

```
FCPProtocol protocol = new FCPProtocol();
protocol.connect(InetAddress.getByName(192.168.1.222));
while (!protocol.isConnected());
```

Sending data is the simplest operation with a single function call: `sentData(3, data)`, where data is to be sent to the hardware connected to channel 3. Because the FPGA only responds to requests from the host, receiving data involves two function calls:

The first function call sends a data request packet. The FPGA responds with a data acknowledgement packet containing the data from the requested channel. This data

Listing 2.2 Requesting Data

```
protocol.sendDataRequest(1, 8);
Byte[] response = protocol.getDataResponse();
```

can then be retrieved from the protocol by the `getDataResponse()` function. When the program exits, the disconnect function is called to terminate the send and receive threads on the host.

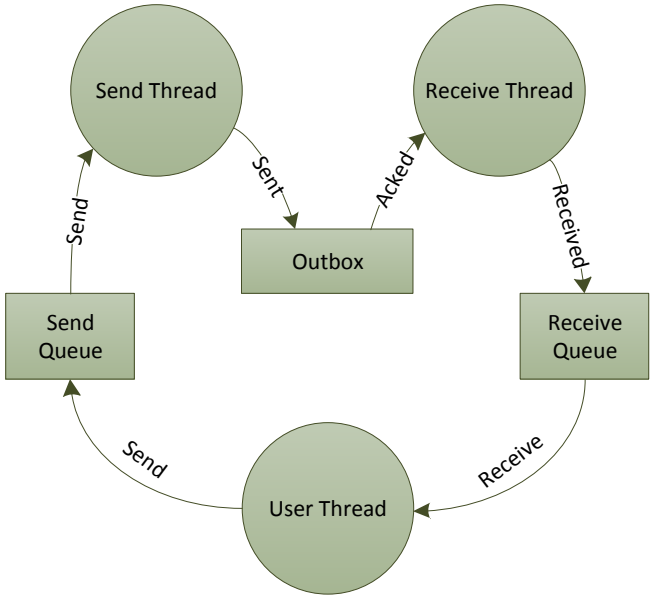

Figure 2.15: Send and Receive Thread Interaction

Unlike the C++ API, sending the data request and receiving the data acknowledgement is separated into two functions: `sendDataRequest` and `getDataResponse`. This allows data to be requested and then other work to be performed while waiting for the data to arrive. This is made possible through a send and receive thread. When the API functions send data, they do not access Java I/O library directly but place the packet to be sent into

a send queue. The send thread then takes packets from this queue, attempts to send the packet, then places the packet in an out-box. Meanwhile, the receive thread will at some point receive the acknowledgement. When this happens, the receive thread removes the acknowledged packet from the out-box. If the send thread notices that a packet has been in the out-box for too long, it will attempt to resend the packet, up to 5 times. Figure 2.15 illustrates how these two threads interact with each other and with the user thread.

Chapter 3

Performance and Functional Analysis of FPGA-CF

3.1 Footprint and Performance

To quantify the improvements made over previous implementations of FPGA network connectivity systems, I compared the performance and footprint metrics of each implementation. In addition, the feature set of each system is listed to provide context for the comparison. Table 3.1 compares the resources, measured speed, and the feature sets of each system reviewed with my system. Both TCP and UDP comparisons are made because FCP + UDP is as reliable as TCP. The throughput numbers marked with a '*' were not published. All of the network stack implementations listed are not readily available for a more controlled comparison. The TP/area column in Table 3.1 lists a metric that tries to equalize the throughput by the number of slices. My implementation is faster, per slice, than all but that reported by Uchida [10]. Non-standard test setups make it difficult to compare throughput. For example, the TCP/IP implementation presented by Uchida reports the throughput achieved between two FPGAs rather than between a host PC and an FPGA, as our tests were done. Comparing portability, extendability, and usability can be even more difficult because these are often subjective. A list of features are noted in the table to get a sense of what each implementation can do.

Using a microcoded architecture and through selective feature implementation of UDP/IP, the hardware footprint is kept small. To get a sense of its size, it is comparable to that of two Picoblaze [6] microcontrollers. The logic portion of the packet processor consumes only 120 Virtex 5 slices, compared to the 94 Virtex 4 slices for the Picoblaze core. Both the Picoblaze and my processor need instruction or microcode memory to support their cores. The microcode ROM takes one full BRAM or 271 slices when using distributed ROM. The

Table 3.1: Comparison

Source	*# Slices*	*# BRAMs*	*TP/area*	*Notes*
Bomel et al			N/A	Power PC
Herrmann et al	1580 (S3E)	2+	N/A *	UDP/IP, Includes MAC
Dollas et al	10,007 (V2)	5	0.03	Complete TCP/IP Stack
Uchida	2285 (V4)	22	0.42	TCP/IP, MAC, ICMP
Loefgren et al	1584 (S3E)	5	N/A *	UDP/IP
Xilinx AppNote	3737 (V4)	8	0.001	Microblaze Webserver
Lieber	589 (V5)	5	0.2	ConARP, Java API

Picoblaze also consumes one full BRAM for this purpose. FPGA-CF uses Xilinx's Ethernet MAC wrapper which consumes another 200 slices and 4 BRAM primitives.

To be usable across a wide range of applications, FPGA-CF needs to achieve reasonably high throughput. The logic design achieved a maximum operating frequency of 75Mhz. I calculated that with a 20% overhead and processing 1 byte per clock, the design should achieve about 480 Mb/s. However, initial throughput test results were less than projected. Increasing the send window size on the host sliding window protocol improved the throughput considerably, and approaches optimal throughput on a 100Base-T, local network. Further improvements gave end-to-end throughput results of around 120 Mb/s on a gigabit Ethernet link. It is end-to-end because it is the throughput measured in a real world situation from a host running a full operating system through a switch to an FPGA as shown in Figure 3.1.

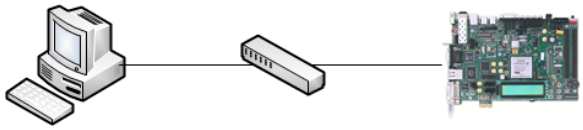

Figure 3.1: Test Setup

Two methods were used to measure throughput. The first method is through direct hardware observation, while the second is using software to measure latency. To measure hardware throughput, I captured the FPGA state with Chipscope while it was receiving a packet. The FPGA Ethernet Platform can receive a maximum size packet (1024 data

bytes), send an acknowledgment packet, and be ready for a new packet in 1189 cycles. The maximum data (not raw) throughput with 1024 data bytes per packet is

$$\frac{75Mcycles/s}{1189cycles/pkt} = 63078pkts/s \approx 63MB/s = 504Mb/s.$$

The throughput measured using the second method on the host includes all software, operating system, and transport delays. Two measurements were taken. First, the Java code was instrumented to measure the throughput and average packet-to-acknowledgment time. Second, Wireshark was employed to measure the packet to acknowledgment time as seen by the host. Figure 3.2 shows the relative placement of the measured latencies and the stack operations.

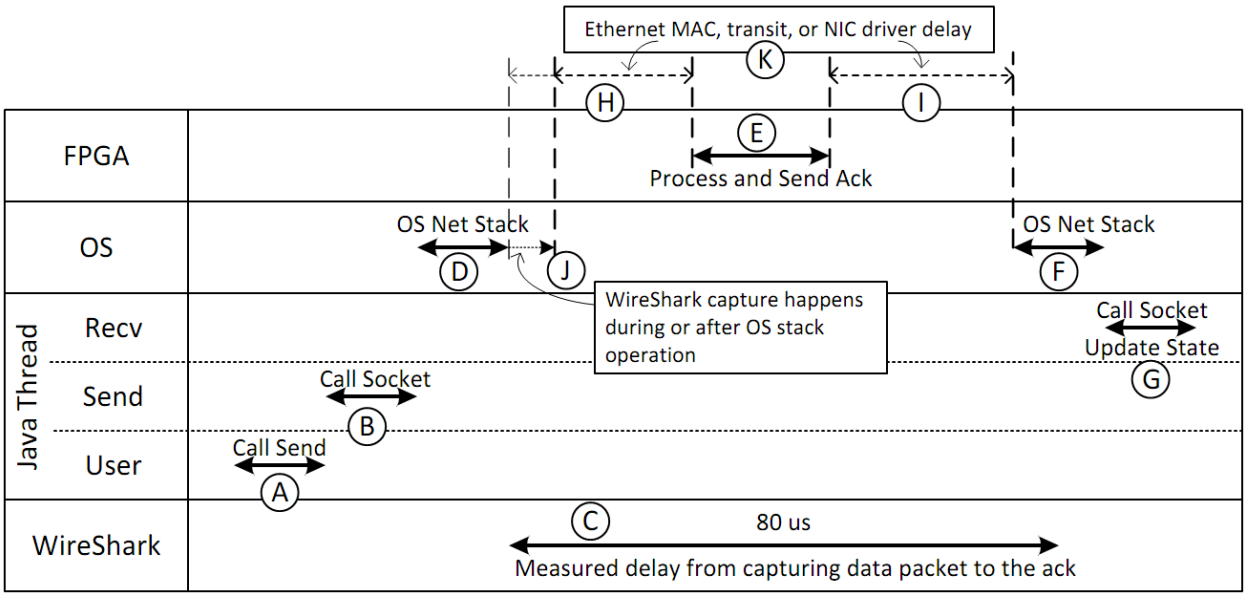

Figure 3.2: Latency Analysis

The average latencies measured with Java instrumentation and with Wireshark were all between 60 and 80 us. The interval (C) represents this measurement. The user call to send and the send thread call to the socket interface happen before the packet is captured by Wireshark (A,B). After the call to the socket interface, the operating system network stack prepares the packet for the network interface driver and is represented by interval (D).

Wireshark captures the outgoing packet at some point during or after interval (D). The time from the receipt of the the first byte to the sending of the last byte of the acknowledgment by the FPGA is 1189 cycles, or about 16 us and is represented by interval (E). Intervals (H) and (I) represent the time that is unaccounted for. This time could be in the Ethernet MAC, transit, or NIC driver. With a low overhead C++ implementation of FCP, the throughput did not change, which confirms that (H) and (I) times cannot be improved with host software improvements.

3.2 Portability and Extensibility

The FPGA Communication Framework is portable between various Xilinx devices, development boards, and workstations. The only required component of the system that is device specific is the hardware Ethernet MAC to LocalLink Interface wrapper. When changing the device or the Phy connection method, the Ethernet wrapper can be regenerated with the Xilinx Coregen tool. When changing development boards, the constraints file may also need to be changed. Chips from other vendors could also potentially be targeted because the core logic is part-independent.

For example, when porting from the XUPV5 board with a Virtex 5 LX110T part, to an ML403 board with a Virtex 4 FX35 part, these changes are needed:

1. Regenerate the Xilinx Ethernet MAC Wrapper.
2. Regenerate the UCF file.
3. In the ICAP channel interface, Update the BRAM FIFO instantiation (this change is only needed when the ICAP is used).

Only a few hours were spent performing this conversion. Ports to other boards and devices will also need the above changes. Porting to another Vendor's chips would only required a redesign of the MAC (link) network layer that interfaces with the specific chip's Ethernet controller. The memories and shift registers in the processor are all generic HDL that synthesis can infer to device specific structures such as an Shift Register LUT (SRL).

The host API can be used with most operating systems because it uses standard UDP/IP communication and it is written in Java. The Java API has been tested on Windows

and Linux operating systems, with no changes needed between them. Included with the framework is a C++ implementation of the protocol. The C++ implementation only works on Linux and only implements the lowest level API commands (send and receive data over FCP channels). It also only supports a window size of one. The C++ implementation is limited because it was a feature requested for a specific case. Despite these limitations, the C++ API is useful in applications where latency is more critical than throughput.

This project is open source, and therefore can be shared and extended. It has been used by multiple Universities for research projects. The framework includes an assembler to generate new microcode and a channel interface generator to change the number of channels exposed to user logic. These tools allow the user to add functionality to the protocol and processor easily without requiring an understanding of or direct changes to the Verilog. Furthermore, the user can add more advanced functionality with minor Verilog changes. For example, during development a new instruction was needed that would move a byte from a port to another while decrementing a counter, and repeating until the counter was zero. Using normal instructions, it takes the following four instructions to do this.

Listing 3.1 Data Transfer Loop

```
#: BEG\_DAT             (label definition)
IN( SR(1))               (move byte from input channel to shift register 1)
OUT(SR(1))               (move byte from shift register 1 to output channel)
SUB( R(0), C(1), R(0) )  (subtract 1 from register 0)
JMP( label["BEG\_DAT"], IF(NEQ(R(0), C(0))) )  (jump if register 0 is zero)
```

The new instruction enables a stream of bytes to be transferred at an average rate of one byte per cycle. To add this instruction, about 5 lines of code were added and 10 were changed in the Verilog. To accommodate the new instruction in the assembler, a new python class of about 10 lines of code along with a few changes were needed. With these changes, the assembly code was simply recompiled, generating a new state machine ROM. The new microcode for the packet processor decreased the packet acknowledgment delay by

a factor of almost four. Similar to this example, further improvements and additions are made possible by the microcoded architecture and the assembler.

Chapter 4

Applications

From the basic input and output system provided by FPGA-CF, many interesting and useful applications can be built in a relatively short amount of time. I will present three application examples that demonstrate the various features and strengths of FPGA-CF. Each example implements user hardware connected to the channel interface. Any user logic that interfaces with a FPGA-CF channel is referred to as a channel module. The software to control the hardware is also described. The first example is a simple hardware register that can be read and written from the host. The next is an example of taking an existing core, md5 [20], and writing a channel module for it. The channel module allows the core to be used from a host machine through the FPGA-CF framework. The last example is much more complex. It uses the ICAP port and a clock control module to implement a rapid on-chip testing and debugging system.

4.1 Hardware Register

To demonstrate basic data input and output, the register channel module allows an on-chip register to be controlled over FCP. In this example, the register module is connected to channel 2. Channel 1 controls the general purpose LEDs and reads the DIP switch position on the development board. This control of the LEDs and DIP switches is included in all application examples to provide immediate confirmation that the communication link is working correctly. Figure 4.1 illustration the structure of this example system.

To control the LEDs, we write a single byte to channel 1. First, I create an array containing a single byte that represents the desired value of the LEDs. Then, this array is sent to the FPGA. Listing 4.1 illustrates this. To get the value of the DIP switches, we read a single byte from channel 1 as shown in Listing 4.2.

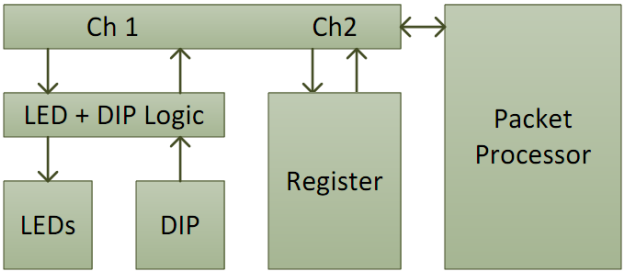

Figure 4.1: Hardware Register

Listing 4.1 Set LED Value

```
ArrayList<Byte> data = new ArrayList<Byte>();
data.add(0xa3);
fcpprotocol.sendData(1, data, 1);
```

Listing 4.2 Get DIP Value

```
fcpprotocol.sendDataRequest(1, 1);
ArrayList<Byte> data = fcpprotocol.getDataResponse();
```

The hardware register is controlled through channel 2. The register is 32 bits wide. Data, however, comes from the channel interface one byte at a time. To load all four bytes of the register, the bytes are transferred least-significant byte first. To hide these details, I created an API class called RegisterInterface. The constructor takes the FCPProtocol object and a channel number as arguments. With these, it can access the correct hardware on the FPGA. This class exposes these two functions:

```
public void writeRegister(int value);
public int readRegister();
```

These functions accept and return a 32-bit integer, respectively. The write function extracts the four bytes from the int. Then, it sends them in the correct little-endian order to the channel specified in the constructor (in this example, channel 2). The read function reads four bytes from this channel. It them assembles them into the 4-byte integer to be

returned. Using this API, reading or writing the register is a single function call. Using this example as a starting point, one can create a register interface to read status from and control hardware running on the FPGA from a java program.

Having an API for each channel module encourages reuse and simplifies software development. It also provides a simple mechanism for automated hardware regression tests. This pattern of first creating a hardware module that interfaces with the FPGA-CF channel interface, then creating a small API to hide lower level details is used throughout the examples. This is the preferred method for using FPGA-CF and will simplify both hardware and software development.

4.2 MD5 Hash Calculator

The MD5 hash calculator logic is taken from the opencores.org project, md5 [20]. I implemented the interface logic necessary to translate the signals from FPGA-CF's channel interface to those used by the md5 logic. This example demonstrates how to use existing logic within the FPGA-CF. The md5 module accepts blocks of 512 bits via a 128-bit interface. The interface logic buffers 16 bytes (128 bits) of data from the FCP channel, then clocks the 128 bits into the md5 module. Figure 4.2 describes the way in which the md5 opencores module interfaces with the FPGA-CF channel.

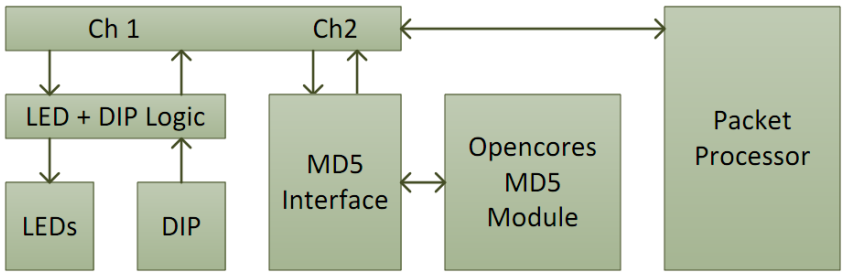

Figure 4.2: MD5 Application

According to the MD5 specifications, there is special padding that must be added to the end of the text being hashed [21]. Rather than adding padding in hardware, this padding is done in software. Following the same design pattern used in the hardware register example,

I created a java class that exposed application specific functions for the MD5 calculator. The code within the class then performs the necessary padding before sending data to the FPGA.

There is only one public function exposed in the MD5 API: `string calculateMD5(string text)`. This function accepts a string and returns the MD5 hash sum of that string. This hides the details of sending and receiving data over FCP, packetizing the data so that it is meaningful to the MD5 module, and the generation of the padding as required by the MD5 calculation. The main purpose of this example is to show how my framework can be used to test and use any hardware module implemented for an FPGA.

4.3 Rapid Testing System

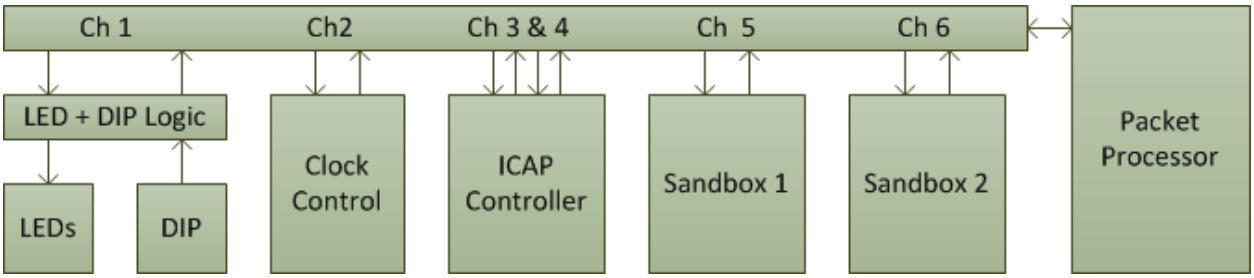

Figure 4.3: Rapid Testing System

The Rapid Testing System uses channel modules to interface with the ICAP port and a clock-control module to control a clock on the FPGA. This controlled clock is used by two sandbox modules connected to other FCP channels, as seen in Figure 4.3. The Rapid Testing System includes APIs and a hardware interface to make the ICAP port more accessible. Using the provided API for ICAP commands, you can:

- write partial bitstreams,
- perform configuration read-back,
- obtain state values,
- read and write individual frames,
- and read and write configuration registers.

Using this API along with a clock control module and a custom API written for it, I was able to create a debugger. With this debugger, you can:

- control a clock (step, run, run for X cycles),
- assert a local reset signal,
- read all state values within the design,
- and send test vectors to/from the design under test.

For example, I can test the sha1 modules created from an opencores project, similar to the md5 module (see Section 4.2). After creating and simulating the sha1 module, I then synthesize the module and use Planahead to generate a partial bitstream for it. I program channel 5 with the sha1 module using the code in Listing 4.3.

Listing 4.3 Send Bitstream

```
icapinterface.sendBitstream("sha1\_channel5.bit");
```

Now, I can test it by sending a sample string to channel 5. We first must create a sha1 API, then use it to send a string to the SHA1 hash calculator that is now on the FPGA. The first line in Listing 4.4 tells the sha1 interface that the hardware is on channel 6. The next sends the string and receives the result.

Listing 4.4 Calculate SHA1 Hash

```
SHA1Interface sha1 = new SHA1Interface(fcpprotocol, 6);
sha1.calculateSHA1("Sample text.");
```

Connectivity with the internal configuration port allows for interesting partial reconfiguration and debugging applications. Modules to be tested can be connected to the channel interface to accept test vectors and send results. Using a clock control module, the design

under test can be controlled to see the clock-by-clock progress. This progress can be viewed by reading the state bits through the ICAP port. An on-demand prototyping system can be built using partial reconfiguration. All these functions can be performed over a single Ethernet connection. This paves the way for FPGAs to used as computing resources similar to server farms. Modules can be dispatched to available FPGAs, similar to the way jobs are distributed to servers. The user can then send data to be processed or test the module over the same interface.

Chapter 5

Conclusion

The FPGA Communication Framework provides an all in one solution for FPGA to PC communication. A standard hardware and software interface allows system designers to place more focus on the application rather than the mode of communication. The framework provides a generic data link to connect a software application to multiple hardware resources on chip. These channels can transport data, control, and configuration information to and from user hardware modules. This information can be sent and received using the included Java or C++ application programming interfaces. This framework:

- hides communication implementation details,
- allows researchers to focus on application and not communication,
- provides a reasonably fast connection for more data-intensive applications,
- is extensible,
- supports multiple operating systems and devices,
- and gives researchers access to the ICAP port for Xilinx devices without the need for a full processor.

This system is implemented using a small, extensible packet processing core that occupies minimal FPGA resources. This processor is designed using a python-based embedded language. This language produces the Verilog ROM that controls the processor. This method allows changes to be made more easily and in less time than a traditional state machine design. It also provides a path for users to extend the capabilities without the need to understand the Verilog.

FPGA-CF uses standard network protocols to allow the FPGA to coexist with other networked workstations. A small, lightweight protocol, FPGA Communication Protocol

(FCP) is implemented over UDP/IP. FCP implements a simple sliding window protocol to guarantee in-order delivery. UDP/IP is used as the base protocol, which not only allows the FPGA to coexist in a standard network as stated, but the software can then use standard operating system libraries to send and receive packets. This makes the Java API portable to most processors and operating systems that support Java.

User hardware is easily connected to the framework. The channel interface presents a bidirectional, 8-bit interface based on Xilinx's LocalLink interface for every channel. The user simply wires their modules to the channel interface. Software can then be written using the API to read and write values to FPGA-CF channels. Software reads and writes are then routed to the corresponding hardware connected to each channel.

I have demonstrated a communication framework that can be both flexible and small, while giving good performance. The hardware is capable of 500Mbps and the measured throughput of an end-to-end application was 120Mbps. This throughput is sufficient for control and status monitoring as well as many data transfer applications. The packet processor consumes only 120 Virtex 5 slices, which is just under the size of two Picoblaze microcontrollers [6]. Its small size allows for the majority of the FPGA to be dedicated to the main application.

Many applications can benefit from this framework. Reading the status of and controlling FPGA resources from software is a simple, but powerful example of its utility. Modules can be testing on-chip more easily by providing a standard way to send test vectors and receive results. Using the ICAP channel interface, more complex and powerful testing debugging systems can be made. FPGA-CF provides the tools necessary for a wide variety of applications and environments.

Bibliography

[1] (2011, Mar.) ChipScope Pro 13.1 Software and Cores User Guide. Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/chipscope_pro_sw_cores_ug029.pdf 2

[2] R. Hauben. (1998) From the ARPANET to the Internet. Columbia University. [Online]. Available: http://www.columbia.edu/~rh120/other/tcpdigest_paper.txt 4

[3] R. Braden. (1989, Oct.) Requirements for Internet Hosts – Communication Layers. IETF RFC 1122. [Online]. Available: http://tools.ietf.org/html/rfc1122 5

[4] ——. (1989, Oct.) Requirements for Internet Hosts – Application and Support. IETF RFC 1123. [Online]. Available: http://tools.ietf.org/html/rfc1123 5

[5] J. Postel. (1981, Sep.) Internet Protocol. IETF RFC 791. [Online]. Available: http://www.ietf.org/rfc/rfc791.txt 6, 20

[6] K. Chapman, "PicoBlaze 8-bit Embedded Microcontroller User Guide," Xilinx, Jan. 2010. 9, 37, 50

[7] P. Bomel, G. Gogniat, and J.-P. Diguet, "A networked, lightweight and partially reconfigurable platform," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, R. Woods, K. Compton, C. Bouganis, and P. Diniz, Eds. Springer Berlin / Heidelberg, 2008, vol. 4943, pp. 318–323. 10, 13

[8] F. Herrmann, G. Perin, J. de Freitas, R. Bertagnolli, and J. dos Santos Martins, "A Gigabit UDP/IP network stack in FPGA," in *Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on*, Dec. 2009, pp. 836 –839. 10, 11

[9] A. Dollas, I. Ermis, I. Koidis, I. Zisis, and C. Kachris, "An open TCP/IP core for reconfigurable logic," in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, April 2005, pp. 297 – 298. 9, 10, 11, 12, 17

[10] T. Uchida, "Hardware-Based TCP Processor for Gigabit Ethernet," *Nuclear Science, IEEE Transactions on*, vol. 55, no. 3, pp. 1631 –1637, Jun. 2008. 10, 12, 37

[11] A. Lofgren, L. Lodesten, S. Sjoholm, and H. Hansson, "An analysis of FPGA-based UDP/IP stack parallelism for embedded Ethernet connectivity," in *NORCHIP Conference, 2005. 23rd*, Nov. 2005, pp. 94 – 97. 10, 11, 12

[12] N. Alachiotis, S. Berger, and A. Stamatakis, "Efficient PC-FPGA Communication over Gigabit Ethernet," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, 29 2010. 10, 11

[13] R. Armstrong, M. Muggli, M. Ouellette, and S. Thammanur. (2006, Oct.) XAPP433: Embedded System Example: Web Server Design Using MicroBlaze Soft Processor. Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp433.pdf 10, 12

[14] N. Alachiotis. (2011, May) UDP/IP Core. [Online]. Available: http://opencores.org/project,udp_ip__core 11

[15] H. Tan and R. DeMara, "A Multilayer Framework Supporting Autonomous Run-Time Partial Reconfiguration," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 5, pp. 504 –516, May 2008. 13

[16] B. Blodget, S. McMillan, and P. Lysaght, "A lightweight approach for embedded reconfiguration of FPGAs," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 399 – 400. 13

[17] W. Y. Wei and D. Huang. (2007, May) XAPP691: Parameterizable LocalLink FIFO. Xilinx. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp691.pdf 15, 25, 29, 57

[18] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," *SIGCOMM Comput. Commun. Rev.*, vol. 30, pp. 309–319, August 2000. [Online]. Available: http://doi.acm.org/10.1145/347057.347561

[19] J. Postel. (1980, Aug.) User Datagram Protocol. IETF RFC 768. [Online]. Available: http://www.ietf.org/rfc/rfc768.txt 19

[20] J. C. Villar. (2010, Apr.) SystemC/Verilog MD5. [Online]. Available: http://opencores.org/project,systemcmd5 43, 45, 78

[21] R. Rivest. (1992, Apr.) The MD5 Message-Digest Algorithm. IETF RFC 1321. [Online]. Available: http://www.ietf.org/rfc/rfc1321.txt 45, 79

Appendix A

Users Guide

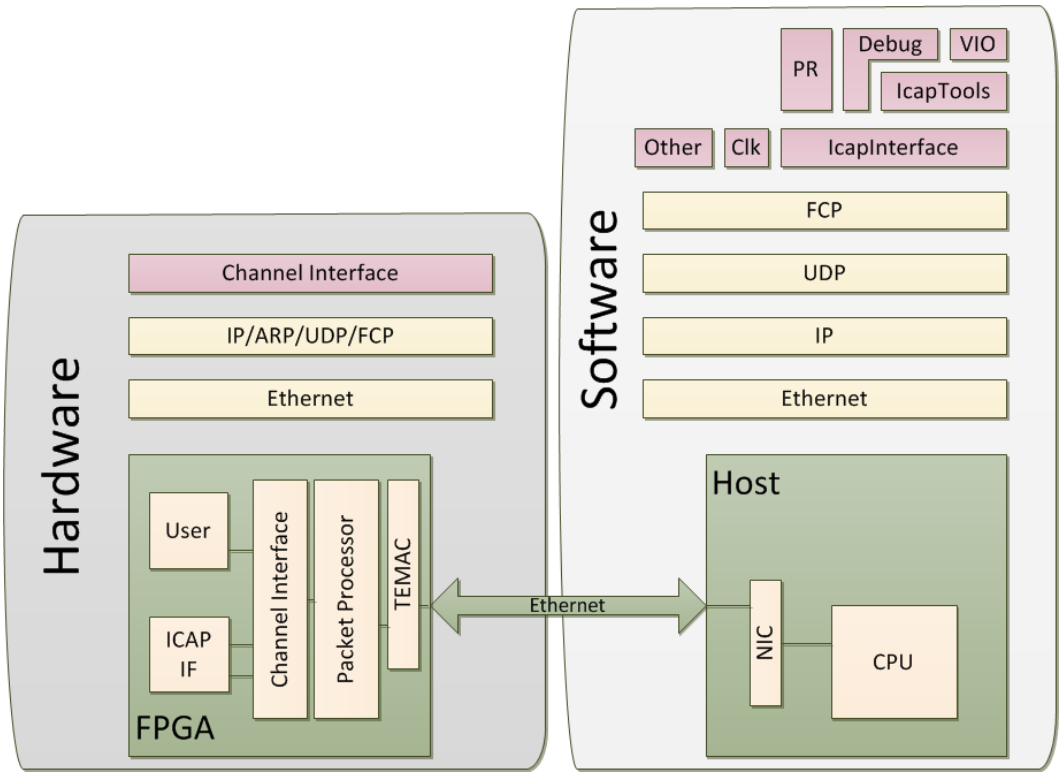

Figure A.1: FPGA Communication Framework

A.1 Introduction

The FPGA Communication Framework (FPGA-CF) is a communications framework for FPGA to host PC communication. It uses standard network protocols to communicate between an FPGA and any host computer. The only requirements for the computer are

that it must have a network adapter and Java support. This system can be used to send test vectors and receive results for in-hardware verification. It can also be used as the main communication mechanism when appropriate. The use of standard network protocols eases the deployment process because standard networking components can be used. Having a Java based API allows for the use of many host systems (Windows, OS X, Linux, etc.).

A.2 Overview

During the FPGA development cycle, in-hardware testing must be performed. Portions of the users design are often completed before any communication system is done. To test the design, a simple communication system needs to be used temporarily to get test vectors and sample data to the circuit. This framework can serve as that system.

User modules are connected to the communication system through the channel interface. This interface can present one to fifteen channels for general use. The user module(s) are then wired up to their assigned channel numbers. The Java API is used to connect to and send data to/from the user module(s). The API classes are employed by the user software to send generated data and test vectors to the hardware. Any Java program can make use of this API to connect to hardware resources. The basic flow for integrating this framework is shown in Figure A.2.

A.2.1 Requirements

This framework can be used with many Xilinx FPGAs and development boards. Some boards are supported without change. Others, including those with Altera chips, can be supported if the user provides the necessary circuitry, such as the Ethernet MAC wrapper. The ICAP configuration interface is only available on Xilinx chips. Table A.1 shows the chips and boards supported by each main component.

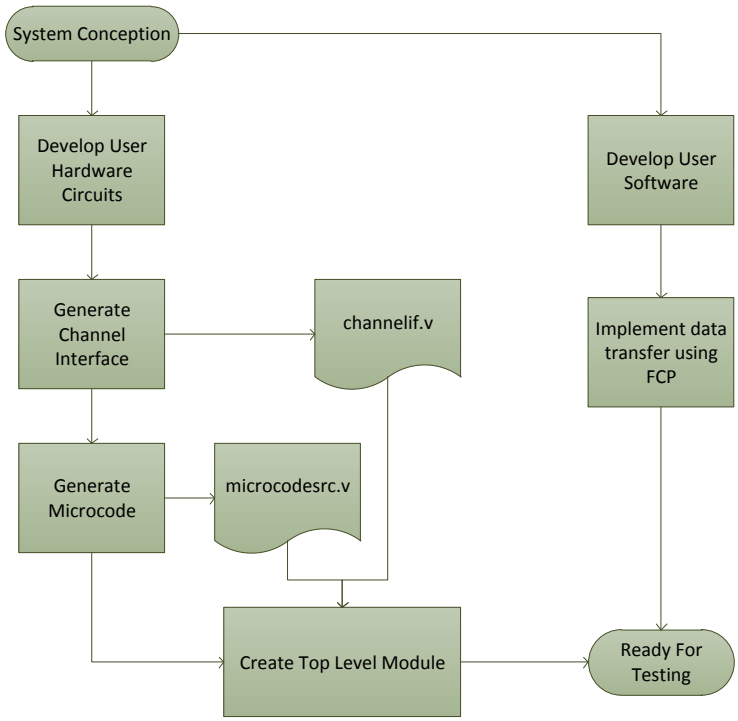

Figure A.2: Basic Flow for Incorporating this communication system into the user's design

Table A.1: Supported Chips and Boards

Chip	*Board Support*	*Packet Processor*	*EMAC Wrapper*	*ICAP Interface*
Virtex 4	ML403	Supported	Supported	Supported
Virtex 5	ML50x, XUPV5	Supported	Supported	Supported
Spartan 6		Supported	User Provided	User Provided
Virtex 6		Supported	User Provided	User Provided
Other Vendors		Supported	User Provided	Not Available

A.2.2 Hardware

The hardware framework consists of a packet processing circuit, Ethernet MAC wrapper, and a channel interface (see Figure A.3). The packet processing circuit implements the entire network stack from layer 1(IP) to 3(FCP).

The processor is a micro-coded state machine that is generated from a python script. The script, fcpudpip.py, generates the micro-code ROM file in Verilog, microcodesrc.v. This

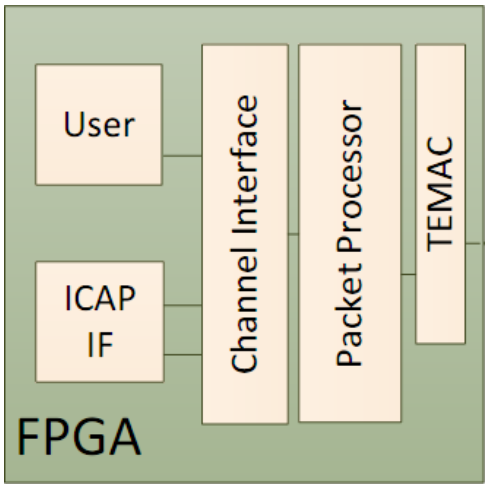

Figure A.3: FPGA-CF Hardware

file needs to be generated by the user when the IP address needs to change. By default, it is hard-coded to 192.168.1.222. To change the IP address, first locate the IP definition at the top of fcpudpip.py:

```
IP = [192, 168, 1, 222]
```

Change each number, remembering to keep the commas instead of periods. After making the change, generate the Verilog from the command line (python 2.6 required):

```
>  python pasm.py fcpudpip.py > microcodesrc.v (Linux)
>  python pasm.py fcpudpip.py | Out-File Encoding ascii microcodesrc.v (PS)
```

Replace the old microcodesrc.v with the newly generated one. Now, the FPGA will respond to the new IP address.

A.2.3 Software

The software is written entirely in Java and therefore portable to any operating system that supports Java. The software consists of a base API, FCPProtocol, and sub-APIs to enable higher level functionality for specific modules (see Figure A.4). User programs can be written using the base API, a premade sub-API for existing modules, or a user-created

sub-API. The suggested method is to first create a sub-API for your module, then use it to communicate with your hardware (see the example design in Section A.5).

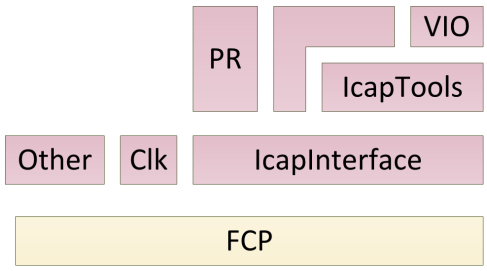

Figure A.4: FPGA-CF Software API

A.3 Hardware Interface

User hardware is connected to the framework via the channel interface module which is based on Xilinx's LocalLink interface. The packet processor presents a channel address along with a single bidirectional LocalLink interface. The channel interface decodes the channel address into enable signals. For each channel, a LocalLink in each direction and a channel enable signal is exposed to the user. The user can generate a custom channel interface with only the needed channels exposed. The tool chifgen.py accepts one parameter: the number of channels to expose. The output of the tool is the channel interface, channelif.v.

Each channel gets an in and out LocalLink interface. The LocalLink interface is specified in Xilinx App Note XAPP691 [17]. It is repeated here for convenience. One design consideration should be followed when using the channel interface: in order to prevent timing violations, the control signals generated by the user module (out_dst_rdy, in_src_rdy, etc.) should not be derived from control signals of the channel interface without registering them first.

Data is transferred to and from the channel interface via an 8-bit input and output bus. Each direction has four control signals governing the operation. The out_src_rdy and

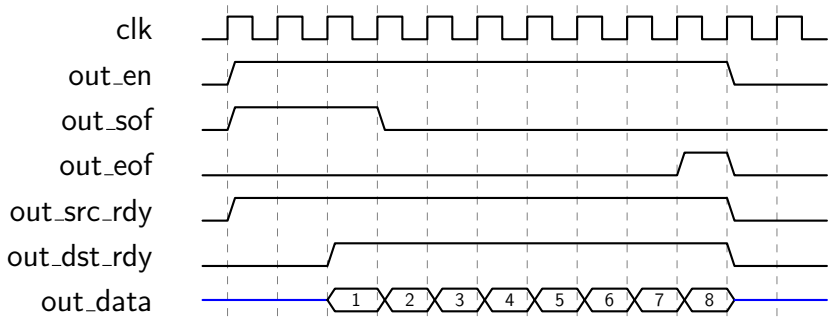

Figure A.5: From Channel Interface

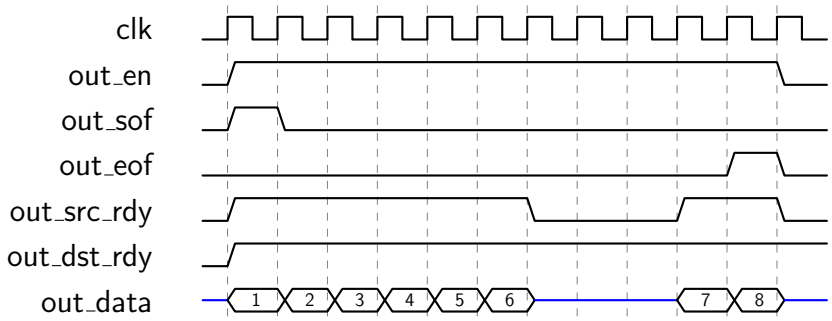

Figure A.6: From Channel Interface Flow Controlled

out_dst_rdy signals control when data is valid and accepted, respectively. Data is only considered transferred when both signals are asserted. The out_sof and out_eof signals are asserted on the first and last byte, respectively, of each data packet. These signals are often used to trigger user designs to begin. The enable signal is a global enable for that channel. For data to be valid coming from the channel interface, en and out_src_rdy must both be asserted. Figure A.5 shows an eight byte sequence being sent from the channel interface to the user logic.

As seen in Figure A.5, the user logic generated signal, out_dst_rdy, is asserted when it is ready for data. At that point, the eight bytes of data are transferred to the user logic. If at any time the out_src_rdy signal is deasserted, the data is not valid and the user logic must wait for the signal to be asserted to continue receiving data. Figure A.6 shows an example where the user logic is expecting eight bytes, but must wait midstream for the last two.

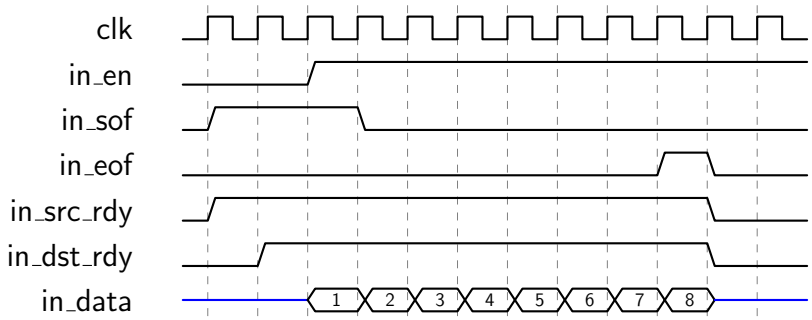

Figure A.7: To Channel Interface

For data transfers in the other direction (from user logic to channel interface) the procedure is identical. Figure A.7 shows an example of this. Flow control can also cause pauses in communication similar to those shown in Figure A.6.

A.4 Software API

All capabilities of the FPGA Ethernet Framework are encompassed in the FCPProtocol.java API. Five basic functions support all operations:

1. void connect(InetAddress address);
2. void sendData(int channel, ArrayList>Byte <data, int numBytes);
3. void sendDataRequest(int channel, int numBytes);
4. byte[] getDataResponse();
5. void disconnect();

To connect to the FPGA, the correct IP address is needed. An FCPProtocol object is created, then the connect function is called. Before any data transfer functions can be called, the isConnected() function should return true. Listing A.1 is an example of the connection procedure.

Sending data consists of a single function call. The data, an ArrayList or array of bytes, is passed along with the channel number and number of bytes to send. The code in Listing A.2 would result in a waveform on the channel interface similar to Figure A.6.

Listing A.1 Connection Procedure

```
FCPProtocol protocol = new FCPProtocol();
protocol.connect(InetAddress.getByName(192.168.1.222));
while (!protocol.isConnected());
```

Listing A.2 Sending Data to Hardware

```
ArrayList<Byte> data = new ArrayList<Byte>();
for (int i=0; i<8; i++) {
    data.add(new Byte((byte)(i+1)));
}
protocol.send(1, data, 8);
```

Receiving data is split into two function calls: requesting the data and getting the response (see Listing A.3). The data request sends a request packet to the FPGA, which in turn responds with the data. The data request function is non-blocking. The FCPProtocol receives the response some time later. This response is retrieved with the getDataResponse() function. This function blocks until a response is in the receive buffer. It then returns an array of bytes containing the data sent from the FPGA. The responses are given by getDataResponse() in the same order that the requests were sent.

Listing A.3 Receiving Data from Hardware

```
byte[] response;
protocol.sendDataRequest(1, 8);
response = protocol.getDataResponse();
```

When the program exits, the disconnect function should be called to terminate the send and receive tasks. Listing A.4 shows the complete connect, send, receive, then disconnect session.

Listing A.4 Complete Session

```
FCPProtocol protocol = new FCPProtocol();
protocol.connect(InetAddress.getByName(192.168.1.222));
while (!protocol.isConnected());

ArrayList<Byte> data = new ArrayList<Byte>();
for (int i=0; i<8; i++) {
    data.add(new Byte((byte)(i+1)));
}
protocol.send(1, data, 8);

byte[] response;
protocol.sendDataRequest(1, 8);
response = protocol.getDataResponse();

protocol.disconnect();
```

A.5 Example Design

Included with this framework is a complete and simple example design. The design targets an XUPV5 development board, and it exposes two FCP channels in the hardware. When written to, channel 1 sets the eight LEDs to the binary representation of the byte written to the channel. When read from, channel 1 reads the eight DIP switches. Channel 2 is connected to a 32-bit register module. This module allows a 32-bit value to be read and written in little endian order. The hardware is accompanied by an API for the 32-bit register and a command line program to read and write the register, LEDs, and DIP switches.

A.5.1 Hardware

The top-level module, top_simple.v, instantiates the Ethernet platform, the channel interface, and the register modules. It also includes code for controlling the DIP switches and LEDs. The code for channel 1 operation of the DIP switches and LEDs is shown in Listing A.5.

Listing A.5 LED and DIP Logic

```
reg [7:0] DIP_r;
reg [7:0] LEDr;
wire [7:0] LEDnext;

assign LEDS = LEDr; // Assign the LED register to the LED pins of the FPGA

always @(posedge clk_local)
begin
    DIP_r <= DIP; // Register the DIP switch contents
end

always @(posedge clk_local)
begin
    if (rst_local)
        LEDr <= 0;
    else if (ch1_wen & ch1_out_src_rdy) //If ch1 is enabled & src is ready
        LEDr <= LEDnext; // Write the next value to the LED register
end

assign ch1_in_sof = 1; // Only 1 byte is read at a time from channel 1,
assign ch1_in_eof = 1; // both start and end of frame are always asserted.
assign ch1_in_src_rdy = 1; // Always ready with DIP switch value
assign ch1_out_dst_rdy = 1; // Always ready to receive writes to LEDs
assign ch1_in_data = DIP_r; // Connect registered DIP to channel 1.
assign LEDnext = ch1_out_data; // Connect channel 1 to next LED value
```

The LED and DIP switch functions for channel 1 are simple. Since only one byte is read or written, the hardware is always ready to send or receive. Therefore, the control signals are all tied high. The start and end of frame signals are tied high because all reads are only one byte long. For writing to the LEDs, whenever the source is ready, the new value is clocked into a register. This register is directly wired to the pins for the LEDs. For reading the DIP switches, the value from the pins are clocked once, then tied directly to the data port.

Channel 2 is used to read and write a 32-bit register. To illustrate the suggested flow, the module, port_register.v, implements this functionality. It presents the interface signals

that are connected directly to channel 2 of the channel interface. The port list is shown in Listing A.6.

Listing A.6 Register Channel Module Port List

```
module port_register(
    input clk,
    input rst,
    input wen,
    input ren,
    input in_sof,
    input in_eof,
    input in_src_rdy,
    output in_dst_rdy,
    input [7:0] in_data,

    output reg out_sof,
    output reg out_eof,
    input out_dst_rdy,
    output out_src_rdy,
    output reg [7:0] out_data
);
```

Four bytes are sent over channel 2 to write the register. A simple state machine shifts the first three bytes in, then on the forth bytes, it writes the whole word to the register to avoid invalid values (see Figure A.8). Another state machine controls the reading. It reads each value directly from the register and selects a byte with a mux using the state as the select line.

A.5.2 Software

A simple API wraps the low level calls to the FCP layer in four functions:

- byte getDIP()
- void setLED(byte value)

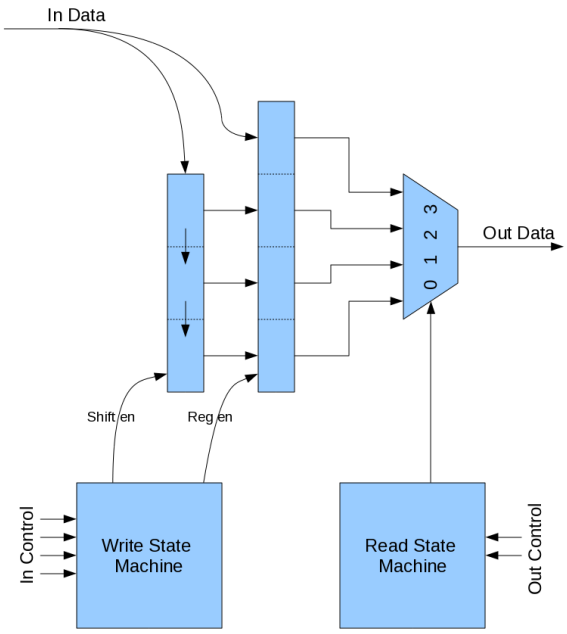

Figure A.8: Register Logic

- int getRegister()
- void setRegister(int value)

The first two functions read and write the DIP switches and LEDs, respectively. The getRegister function reads four bytes from channel 2 then assembles them into a 32-bit integer in little endian order. Its implementation is shown in Listing A.7.

Listing A.7 Read Register

```
public int getRegister() {
protocol.sendDataRequest(2, 4);
byte[] bytes = protocol.getDataResponse();
int res = (((int)bytes[3] & 0xff) << 24) | (((int)bytes[2] & 0xff) << 16) |
          (((int)bytes[1] & 0xff) << 8) | (((int)bytes[0] & 0xff));
return res;
```

The setRegister function sends four bytes in little endian order from a 32-bit integer. It must first mask and shift each byte of the given integer and assemble it into an array. Then it calls the send function of the FCP protocol (see Listing A.8).

Listing A.8 Write Register

```
public void setRegister(int value) {
    ArrayList<Byte> bytes = new ArrayList<Byte>();
    bytes.add(new Byte((byte) (value & 0xff)));
    bytes.add(new Byte((byte) ((value >> 8) & 0xff)));
    bytes.add(new Byte((byte) ((value >> 16) & 0xff)));
    bytes.add(new Byte((byte) ((value >> 24) & 0xff)));
    protocol.sendData(2, bytes);
}
```

These four functions are then used by the example command line program to allow a user to read the DIP switch values, change the LED states, and read and write the 32-bit register. There are four commands:

- w <integer>: right the integer to the register
- r: read the value of the register
- l <byte>: set the LEDs to byte
- d: get the DIP switch positions
- quit: exit the program

An example transcript of the command line interface is shown in Listing A.9.

Listing A.9 Command Line Transcript

```
  Received Connection Ack
  Connected to: 192.168.1.222 on port 12289
  Welcome to the Example Design
>   r
  Register Value: 0
>   w 4923
>   r
  Register Value: 4923
>   d
  DIP Switch Value: 27
>   d
  DIP Switch Value: -39
>   l 185
>   r
  Register Value: 4923
>   quit
  Goodbye!
```

Appendix B

Packet Processor Instructions

Creating state machines for the FPGA-CF packet processor is similar to writing assembly code for standard processors. There are registers, operations, and conditionals that can be used. After writing the code, it is assembled into a form of machine language, which can then be used in a FPGA-CF design.

As mentioned in Section 2.2.4, the assembly language is an embedded language based on Python. The assembly instructions are actually Python functions that print the Verilog microcode value. The INITCODE and ENDCODE are special instructions that must be placed at the beginning and end of the program. These print the Verilog module beginning and ending. The module produced by running the assembly code with python is a Verilog description of the microcode ROM. For example, Listing B.1 shows some assembly code written for the packet processor. The code simply moves a byte from port 4 to register 8, add 2, then sends it out port 2.

Listing B.1 Sample Assembly Code

```
INITCODE()
MOV( C(4), IPR() )
IN( R(8) )
ADD( R(8), C(2), R(8) )
MOV( C(2), OPR() )
OUT( R(8) )
ENDCODE()
```

When run by a python interpreter, this code prints the Verilog in Figure B.1. Each case of the case statement is printed by each instruction of the assembly program. The rest of the code is printed by the INITCODE and ENDCODE calls. This Verilog module is used by the packet processor as the microcode ROM. The 67-bit code output drives the control lines of the various units within the processor, producing the desired behavior.

There is a jump (JMP) instruction available that jumps to the specified address in the microcode. This address is simply the 0-based order of the instructions in the assembly code. It is common practice to allow labels to be created to abstract the notion of instruction address. This allows more code to be added without changing all jump instructions that refer to addresses after the added code. To accommodate labels, the assembly code python script is not run directly. Instead, an assembler script, pasm.py, preprocesses the assembly source. Instead of passing a constant to the jump instruction, a label map is used as shown in Listing B.2. The added conditional jump will skip the add instruction if the value in register 8 is equal to 2.

Listing B.2 Label Example

```
label = globals()

INITCODE()
MOV( C(4), IPR() )
IN( R(8) )
JMP( label["Output"], IF( EQU(C(2),R(8)) ) )
ADD( R(8), C(2), R(8) )
#: Output
MOV( C(2), OPR() )
OUT( R(8) )
ENDCODE()
```

In addition to the standard set of instructions, custom instructions can be created. There are two ways of doing this. The first way is to use the `MAN` custom instruction. It takes

```
module microcodesrc
(
input wire [8:0] addr,
output reg [66:0] code
);

always @(addr)
begin
case (addr)

// code: {          <jmp,rst>
//          |      <in_rdy,out_rdy,aeof,asof>
//          |      |        <predmode>
//          |      |        |     <pred: fcs,eof,sof,equ,dst,src>
//          |      |        |     |          <High Byte Reg En>
//          |      |        |     |          |     <Output Byte Select>
//          |      |        |     |          |     |     <Outport_reg_en, Inport_reg_en>
//          |      |        |     |          |     |     |      <Data Mux Select>
//          |      |        |     |          |     |     |      |     <Op 0 Select>
//          |      |        |     |          |     |     |      |     |     <Op 1 Select>
//          |      |        |     |          |     |     |      |     |     |     <Register Address>
//          |      |        |     |          |     |     |      |     |     |     |       <Register Write Enables>
//          |      |        |     |          |     |     |      |     |     |     |       |     <FCS Add, FCS Clear>
//          |      |        |     |          |     |     |      |     |     |     |       |     |      <sr1ie,sr2ie,sr1oe,sr2oe>
//          |      |        |     |          |     |     |      |     |     |     |       |     |      |        <Flag Register>
//          |      |        |     |          |     |     |      |     |     |     |       |     |      |        |     <Compare Mode>
//          |      |        |     |          |     |     |      |     |     |     |       |     |      |        |     |     <ALU Op>
//          |      |        |     |          |     |     |      |     |     |     |       |     |      |        |     |     |     <Byte Constant>
//          |      |        |     |          |     |     |      |     |     |     |       |     |      |        |     |     |     |       <Word Constant> }
000: code <= {2'b00, 4'b0000, 2'd0, 6'b000000, 1'b0, 1'd0, 2'b01, 3'd0, 2'd0, 1'd0, 4'd00, 2'd0, 2'b00, 4'b0000, 1'b0, 3'd0, 2'd0, 9'd000, 16'd00004};
001: code <= {2'b00, 4'b1000, 2'd0, 6'b000001, 1'b0, 1'd0, 2'b00, 3'd3, 2'd0, 1'd0, 4'd08, 2'd3, 2'b00, 4'b0000, 1'b0, 3'd0, 2'd0, 9'd000, 16'd00000};
002: code <= {2'b00, 4'b0000, 2'd0, 6'b000000, 1'b0, 1'd0, 2'b00, 3'd5, 2'd3, 1'd0, 4'd08, 2'd3, 2'b00, 4'b0000, 1'b0, 3'd0, 2'd0, 9'd000, 16'd00002};
003: code <= {2'b00, 4'b0000, 2'd0, 6'b000000, 1'b0, 1'd0, 2'b10, 3'd0, 2'd0, 1'd0, 4'd00, 2'd0, 2'b00, 4'b0000, 1'b0, 3'd0, 2'd0, 9'd000, 16'd00002};
004: code <= {2'b00, 4'b0100, 2'd0, 6'b000010, 1'b0, 1'd0, 2'b00, 3'd2, 2'd0, 1'd0, 4'd08, 2'd0, 2'b00, 4'b0000, 1'b0, 3'd0, 2'd0, 9'd000, 16'd00000};

default: code <= 0;
endcase

end
endmodule
```

Figure B.1: Sample Output Verilog

a single argument, an instance of the `Instruction` class. This class has members for each of the control lines within the packet processor. The second, preferred method is to define a new instructions. Recall that the standard instructions are simply python functions. These function simply create an instance of the Instruction class, set the control lines appropriately, then print the instruction (which produces Verilog case statements). Before the INITCODE instruction, any number of new instructions can be defined in this way. For example, Listing B.3 shows how the clear checksum (CSC) instruction is implemented. It asserts the fcs_clr signal which clears the checksum result.

Listing B.3 Label Example

```
def CSC():
    global pc
    inst = Instruction("CSC()")
    inst.fcs\_clr = True
    print inst
    pc += 1
```

The following tables list the instructions and syntax supported by the packet processor. Table B.1 lists the instructions available. Table B.2 shows the valid source, destination, and op code values. Some resources are not supported for all three. For example, it does not make sense to use the checksum result as a destination because it is a calculated value and cannot be directly changed. Table B.3 gives an overview of the three condition modes available as explained in section 2.2.4. Table B.4 lists the signals that can be used as a condition, and Table B.5 lists the possible flags that can be manually asserted.

B.1 Packet Processor Assembly Reference

Table B.1: Packet Processor Instructions

Name	*Syntax*	*Description*
Initialize Code	`INITCODE();`	This must be placed before the first instruction
Input	`IN( dest[, cond][, [flags...]] );`	Move one byte from input port to `dest`
Output	`OUT( source[, cond][, [flags...]] );`	Move one byte from `source` to output port
Bypass	`BYP( [cond][, [flags...]] );`	Move one byte from the input port directly to the output port
Checksum Add	`CSA( source );`	adds the value at `source` to the running finite checksum
Checksum Clear	`CSC();`	Clears the running finite checksum
Jump	`JMP( loc[, cond][, [flags...]] );`	Jumps to the specified location, `loc`
Reset	`RST( [cond][, [flags...]] );`	Resets the processor to the first instruction (equivalent to `JMP(0)`)
Add	`ADD( op0, op1, dest[, cond][, [flags...]] );`	Adds `op0` to `op1` and stores the result in `dest`
Subtract	`SUB( op0, op1, dest[, cond][, [flags...]] );`	Subtracts `op0` to `op1` and stores the result in `dest`
Move	`MOV( source, dest[, cond][, [flags...]] );`	Copies `source` to `dest`
Custom	`MAN( inst );`	Creates a microcode state based on the supplied instruction class, `inst`
Finish Code	`ENDCODE();`	This must be placed after the last instruction

Table B.2: Packet Processor Sources, Destinations, and Operands

Name	*Syntax*	*Valid For*	*Description*
Register	R(reg_num [, hbs])	Any	A register from the Register File hbs (High Byte Select): selects the most significant bit when used as the source for an OUT instruction
Port	P()	Src, Dst, Op0	The current input / output port. This behaves differently than IN and OUT instructions because it does not signal the port that a bytes has been received.
Constant	C(value)	Src, Op0, Op1	A constant value
FIFO Register	SR(sr_num)	Src, Dest	The FIFO temporary storage (numbered 0 - 1)
Checksum Result	CS(hbs)	Src	The current checksum result. hbs: same as for Register
Flag Register	FR()	Src, Op0	(Future) The previous instruction's condition signals
High-byte Reg	HBR()	Dst	Temporary register that is the most most significant byte of the next IN
Output Port Reg	OPR()	Dst	Register that controls what output port is used for OUT and BYP
Input Port Reg	IPR()	Dst	Register that controls what input port is used for IN and BYP

Table B.3: Packet Processor Condition Modes

Name	*Syntax*	*Description*
If	IF(signal)	Executes instruction if the conditions signal is asserted, moves on whether the signals was asserted or not
When	WHEN(signal)	Executes no-ops until the signal is asserted, at which time it executes the instruction a single time
Until	UNTIL(signal)	Executes the instruction continuously up to and including the cycle in which the signal is asserted, then moves on

Table B.4: Packet Processor Condition Signals

Name	*Syntax*	*Valid*	*Description*
Start-of-frame	SOF()	Any	This signal can be used to indicate the start of a packet or frame of data. It is used by the Ethernet interface for FPGA-CF.
End-of-frame	EOF()	Any	This signal can be used to indicate the end of a packet or frame of data. It is used by the Ethernet interface for FPGA-CF.
Equality	EQU(op0, op1 [, bytewide])	JMP	Uses the ALU to compare op0 to op1, and is asserted if they are equal. If bytewide is true, then it only compares the least significant bytes
Inequality	NEQ(op0, op1 [, bytewide])	JMP	Same as Equality, but tests for inequality
Destination Ready	DST()	Any	True when the destination of the current output port is ready
Source Ready	SRC()	Any	True when the source of the current input port is ready
Checksum Zero	FCS()	JMP	True when the checksum result is zero

Table B.5: Packet Processor Flags

Name	*Syntax*	*Use*	*Description*
Assert end-of-frame	AEOF()	OUT	Asserts the end-of-frame signal to the output channel
Assert start-of-frame	ASOF()	OUT	Asserts the start-of-frame signal to the output channel

Appendix C

Implementation Details

In this appendix, I will describe some details of FPGA-CF that did not receive attention. First many of the channel interface modules developed for the example applications are described. An explanation of the packet processor and its control signals and data-path is also given. Finally, the implementation of the packet processor embedded assembly language is discussed in more detail.

C.1 Channel Interface Modules

C.1.1 ICAP Channel Module

The ICAP channel module enables the use of Xilinx's ICAP module over FPGA-CF. At first glance, it would seem simple to connect the channel interface to the ICAP. Each have an 8-bit bus, and each have flow control. The flow control for the ICAP module, however, only functions for writing to it. When the ICAP has data and a read operation is initiated, the ICAP outputs all bytes available without pause. In addition to this limitation, the write enable signal is synchronous to the ICAP. This requires the write enable signal to be driven high one cycle before data is sent. The ICAP channel module accommodates the differences between interfaces (see Figure C.1.)

To accommodate for writing to the ICAP, the channel module FSM simply delays writing to the ICAP for one clock cycle to allow the ICAP to register the write enable signal. As seen in figure C.2, when the source (the packet processor) is ready, the FSM moves to the start-write state (S_WRITE) for one cycle, during which icap_wr is asserted. In the WRITE state, the icap_en signal is controlled by the source ready signal. The ICAP accepts data

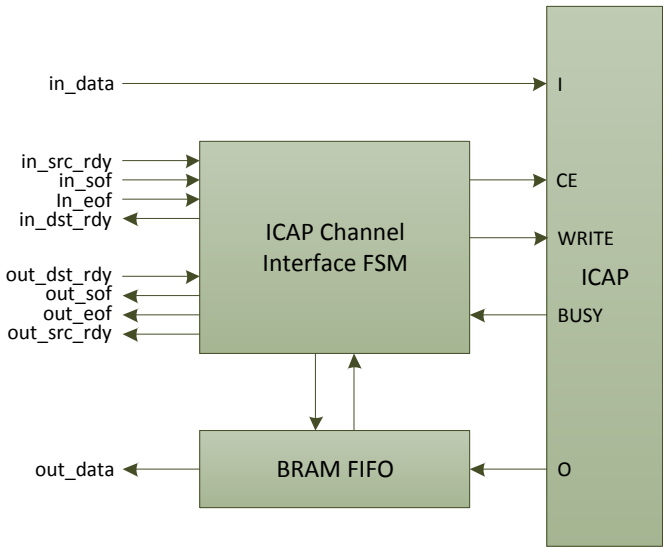

Figure C.1: ICAP Bridge Block Diagram

only when the icap_en is asserted. Finally, when the end-of-frame is signaled, the ICAP write enable signal is held high for one last clock (E_WRITE), after which the FSM returns to the IDLE state.

Reading from the ICAP is much more difficult. The ICAP does not announce when it has data available. To read data, the FSM must assert icap_en, then monitor icap_busy. While icap_en is high and icap_wr is low, the data is valid when the icap_busy signal is low. The data must immediately be consumed whether or not the packet processor is ready. For this, a FIFO holds the read data until the packet processor retrieves it.

The last problem is that the state machine must know how many bytes is expected from the ICAP so it can go back to the IDLE state. To solve this, I split the read and write operations between two channels. The first, write channel is only used for writing data to the ICAP. When in the IDLE state, only bytes coming from the write channel will trigger the aforementioned write sequence. The second channel is for reading data from the ICAP. First, a two byte read count is sent over the read channel. The state machine is then in the READ state. In this state, the ICAP is free to send bytes into the FIFO. Once the

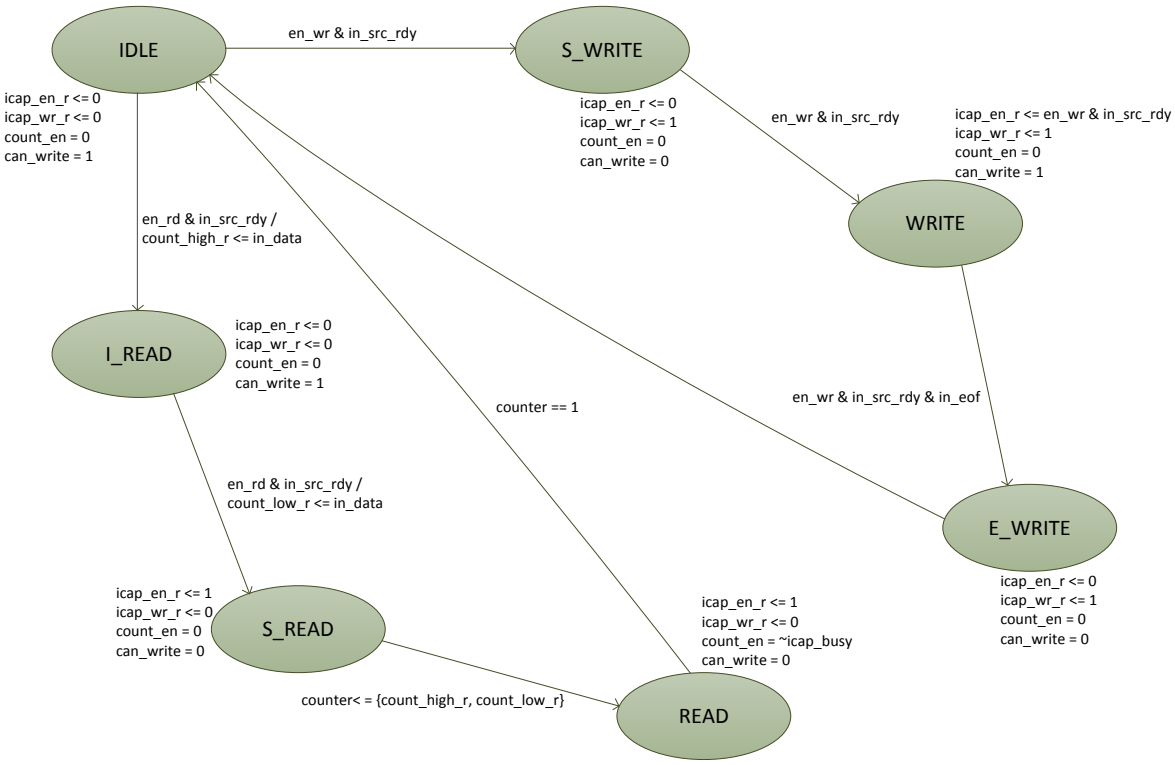

Figure C.2: ICAP Bridge State Machine

prescribed amount of bytes have been transferred to the FIFO, the state machine returns to IDLE. Software can now read (from the read channel) the bytes contained in the FIFO.

C.1.2 Clock Control Module

The clock control module allows a single clock and a local reset to be controlled over FPGA-CF. The clock can be stopped, run for 1 to $2^{32} - 1$ clock cycles, or allowed to freely run. The reset can be asserted and deasserted. There are two registers that control the operation: the control register and the termination count register.

Figure C.3 illustrates a use case for this module. First, you set the control register to assert the reset and the termination count to run for a few cycles. Next, you allow the clock to run for a number of cycles. Now, assuming you are controlling a clock to a module you are testing, you would send data to the channel to which the module is connected. You can

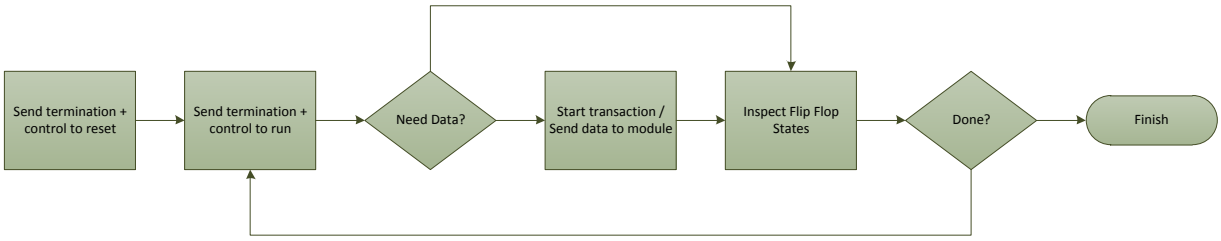

Figure C.3: Clock Control Use Case

then inspect the flip flops within the module being tested using the ICAP tools discussed in Section 4.3. With this, you can step through your design in hardware.

The control register has three bits of interest. Bit 0 allows the clock to run. When set, the clock will run until either the termination count register becomes zero or the start bit is overwritten to 1. Bit 1 allows the clock to free-run. When set, the clock will only stop if the bit is cleared. Bit 2 drives the local reset line.

Loading the control and termination count registers over FPGA-CF involves sending 1 or 5 bytes. Sending 1 byte loads only the control register. This is useful to free-run the clock or to lower the reset during a free-run. To load the termination and control byte, the four byte termination count is first sent, most significant byte first, followed by the control byte. This must be done in a single packet (i.e. not in two separate calls to FCPProtocol.send()).

Figure C.4 shows how these registers are connected. The control register only is loaded when the end-of-frame is signaled. This signifies the last byte of the data. If the termination count value is sent with the control frame, the first four bytes (the termination count value) are shifted into that register, while the last byte is loaded into the control register.

C.1.3 MD5 Channel Module

The MD5 channel module does what its name implies; it calculates the MD5 hash sum. The core functionality was obtained from opencores [20]. The way the opencores

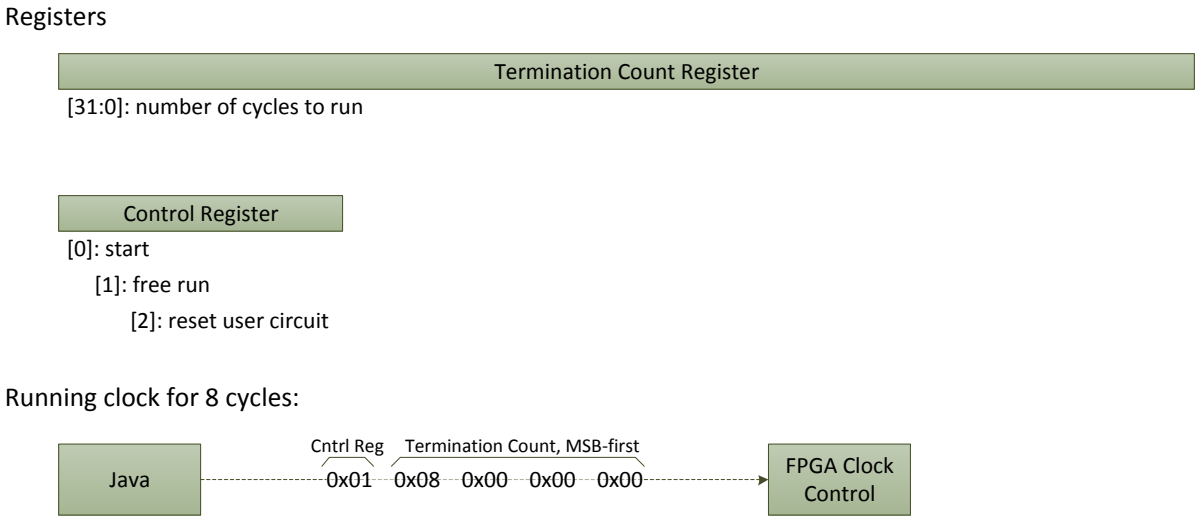

Figure C.4: Clock Control Registers

module accepts data is drastically different from FPGA-CF. However, having a software front end allowed the preparation of the data to be handled in software.

The MD5 core has a 128-bit interface. Buffering data from FPGA-CF to provide 128 bits at a time is trivial. There are, however, some control signals that need to be asserted at various times. The newtext_en signal is asserted to clear the core and prepare for a new string. The MD5 calculation works on blocks of 512 bits [21]. Each block is loaded 128 bits at a time with a load signal. After loading 512 bits, the logic must wait for the ready signal to go high before starting the next block. This is done until the end of the string.

Figure C.5 is the state diagram for the interface between the channel and the MD5 module. The state machine on the right loads the 128 bit result when ready, then simply shifts out the 128-bit result one byte at a time to the channel interface. On the left is the write state machine. When a new packet comes in, it first asserts the newtext_en signal. Then, it shifts the next 16 bytes into a 128-bit temporary register. When filled, the 128-bit value is loaded into the MD5 module. When 4 of these values have been loaded, the state

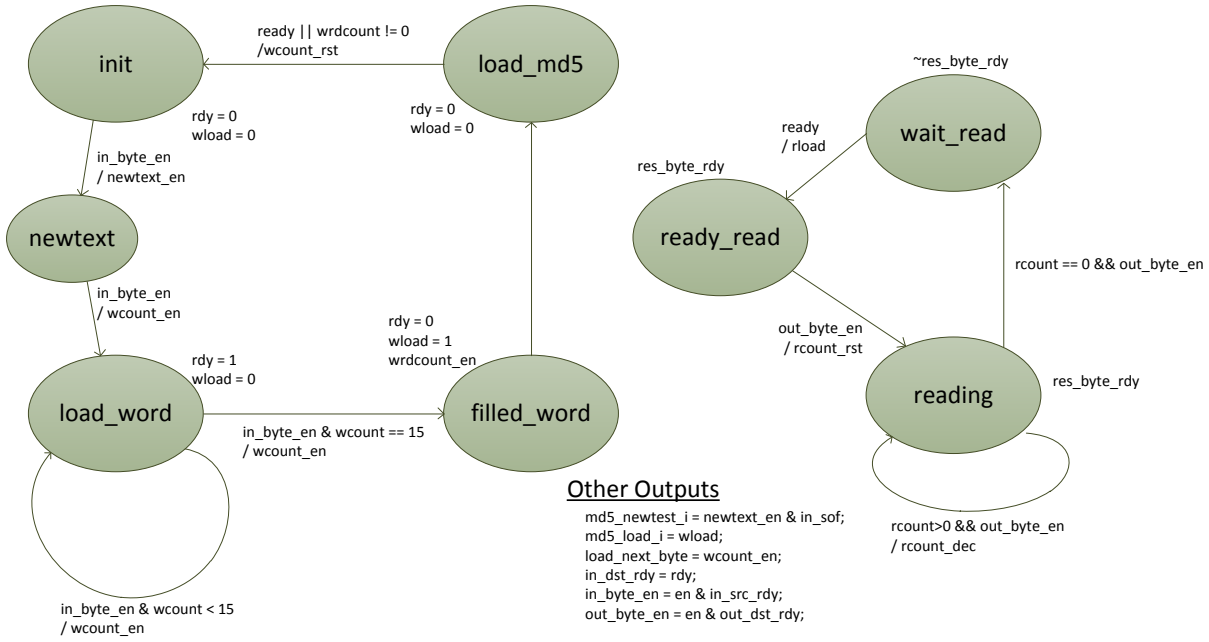

Figure C.5: MD5 Channel Interface State Machine

machine waits until the MD5 ready signal is asserted before allowing another 512 bits to come from the channel interface.

As mentioned, it is left up to software to preprocess a string from calculation. This involves padding the string in the specified way so that its length is a multiple of 512 bits. This could have been done in hardware, but it is a good example of how using FPGA-CF, hardware and software can work together easily.

C.1.4 SHA1 Channel Module

The SHA1 module has similar issues as the MD5. Its data is padded in software. The calculation is done in 512-bit chucks. The interface, however, is a 32-bit interface and the result is a 160-bit word. All other aspects are virtually identical. Figure C.6 describes the state machine that interfaces between the packet processor and the opencores SHA1 module.

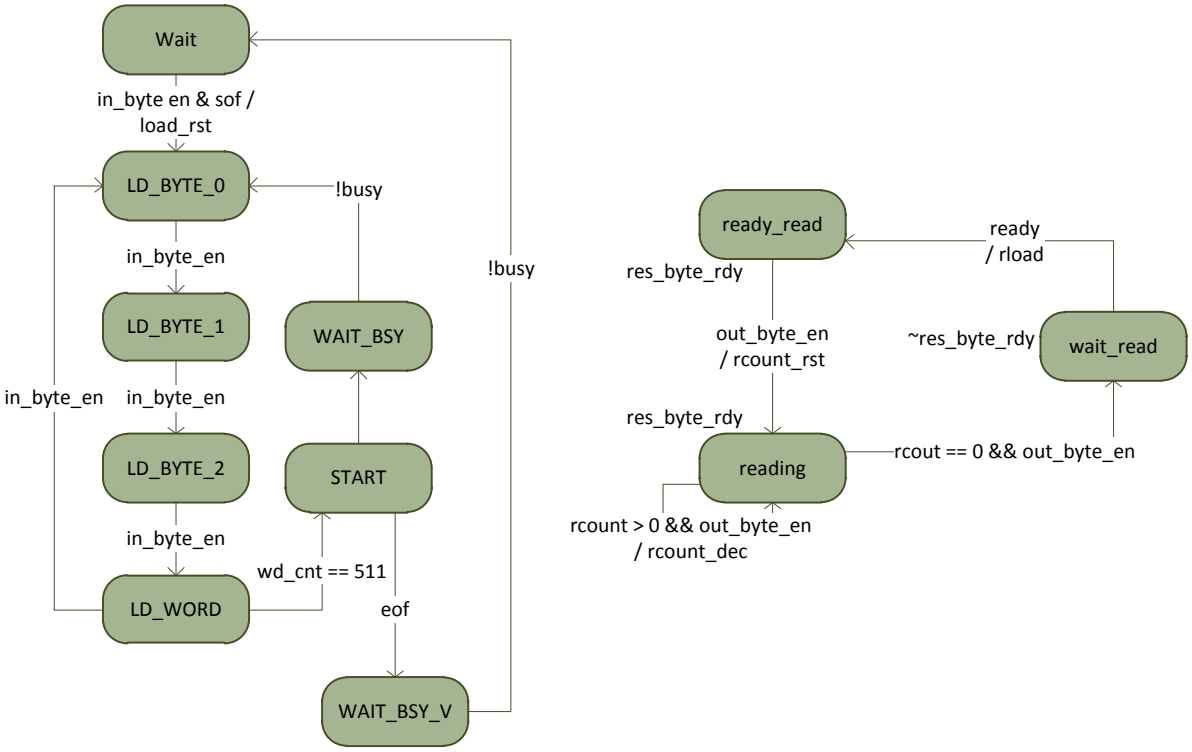

Figure C.6: SHA1 Channel Interface State Machine

C.2 Packet Processor

A lot has been discussed regarding the packet processor, but the details of all the control lines are presented at the end of this appendix in Figure C.7. Each state of the microcode drives all these control lines to produce the desired behavior. Other methods could be used to drive the logic units such as a finite state machine or a tradition processor instruction set with instruction decode. Section C.3 describes the way the microcode is generated from an assembly code based in python.

C.3 Embedded Assembly Language

In Appendix B I explained how to use the assembly language and what instructions were available. Here, I will detail the instruction class and the assembler.

The instruction class implementation is shown in the Instruction Class Listing at the end of this section. Each member variable represents one of the control lines in the packet process (see Figure C.7.) The `loc` variable is the one exception, which stores the line of assembly code that corresponds with this instruction (this helps in debugging.) The `__str__` function is an automatic function that gets called when converting an instance of Instruction to a string (i.e. print instr.) This is how the Verilog code is generated.

The `processCond` function takes a list of condition class objects (see section B.1) and sets the correct member variables (control lines) to implement that condition. For example, say we have the instruction, `RST( IF( EQU(R(4), C(135)) ) )`. This should reset if register 4 is equal to 135. The EQU object would be passed to the `processCond` function of the instruction. This set the predicate, pred_cmp, and sets the ALU up to compare the register with the constant. With the predicate set, the condition logic (see Figure 2.9) the control signal will only be enabled **IF** the condition holds. Similar to `processCond`, the `processFlags` function takes a list of Flag objects and enables the correct flags in the instruction.

The assembler script is shown in Listing C.1. Most of the work is already done because the instructions produce their own Verilog microcode with the `__str__` function. The assembler enables one important feature: jump labels. Jumps, via the `JMP` instruction, require the state number, not the line of code. Without labels, this required the developer to calculate the instruction number for every jump location. At first, it is not hard, because it is the line number minus the number of blank lines, comments, and initialization python code. But as the assembly gets longer, this can be cumbersome, especially when changes are made in the middle of the program (causing all future jump points to change.)

Listing C.1 Packet Processor Assembler

```
if (len(sys.argv) < 2):
    print 'Usage: pasm <input file> [output file]'
    exit()

infilename = sys.argv[1]
if (len(sys.argv) > 2): outfilename = sys.argv[2]
else: outfilename = "out.v"

infile = open(sys.argv[1])
pc = 0
labels = dict()

for line in infile:
    line = line.strip()

    if line.startswith(('IN(','OUT(','BYP(','CSA(','CSC(','JMP(','RST(',
    'ADD(','SUB(','MOV(','SRAP2R(')):
        pc += 1
    elif line.startswith('#:'):
        print "// ", line
        label = line[2:]
        label = label.strip(' ')
        labels[label] = pc

print "// labels: ", labels
execfile(sys.argv[1], labels)
```

The assembler first parses through the assembly python file, keeping track of how many actual instructions it encountered. It also looks for labels, which are special comments of the form, `#: <label name>`. When it encounters one, it adds it to a label hash map, with the label name as the key, and the current instruction number as the value. After parsing through the file, the assembler executes the same file, passing the label hash map as a global argument.

Listing C.2 Label Example

```
label = globals()

INITCODE()
MOV( C(4), IPR() )
IN( R(8) )
JMP( label["Output"], IF( EQU(C(2),R(8)) ) )
ADD( R(8), C(2), R(8) )
#: Output
MOV( C(2), OPR() )
OUT( R(8) )
ENDCODE()
```

In the python assembly file, the built-in function, `globals`, contains this label map. Listing B.2, repeated here as C.2, demonstrates this. Globals is given the more intuitive name, `label`, and then it is used to reference labels within the program.

Instruction Class Listing

```
class Instruction:
    jump = False
    reset = False

    input_rdy = False
    output_rdy = False
    sof_out = False
    eof_out = False

    pred_mode = 0 # 0: when, 1: until, 2: if
    pred_fcs = False
    pred_eof = False
    pred_sof = False
    pred_cmp = False
    pred_dst = False
    pred_src = False

    highbyte_reg_en = False
    output_byte_s = 0

    outport_reg_en = 0
```

```
inport_reg_en = 0

data_mux_s = 0
op0_mux_s = 0
op1_mux_s = 0

reg_addr = 0
reg_wen = 0

fcs_add = False
fcs_clr = False

sr1_in_en = False
sr2_in_en = False
sr1_out_en = False
sr2_out_en = False

flag_reg_en = False

comp_mode = 0

alu_op = 0

const_byte = 0
const_word = 0

loc = ""

def __init__(self, loc):
    self.loc = loc

def __str__(self):
    ret = "\t\t%03d:\t\t\tcode <= {" % pc
    ret += "2'b%1d%1d, " % (self.jump, self.reset)
    % (self.input_rdy, self.output_rdy, self.sof_out, self.eof_out)
    ret += "4'b%1d%1d%1d%1d, "
    ret += "2'd%1d, " % self.pred_mode
    % (self.pred_fcs, self.pred_eof, self.pred_sof, self.pred_cmp,
    % self.pred_dst, self.pred_src)
    ret += "6'b%1d%1d%1d%1d%1d%1d, "
    ret += "1'b%1d, " % self.highbyte_reg_en
    ret += "1'd%1d, " % self.output_byte_s
    ret += "2'b%1d%1d, " % (self.outport_reg_en, self.inport_reg_en)
    ret += "3'd%1d, " % self.data_mux_s
    ret += "2'd%1d, " % self.op0_mux_s
```

```
        ret += "1'd%1d, " % self.op1_mux_s
        ret += "4'd%02d, " % self.reg_addr
        ret += "2'd%1d, " % self.reg_wen
        ret += "2'b%1d%1d, " % (self.fcs_add, self.fcs_clr)
        % (self.sr1_in_en, self.sr2_in_en, self.sr1_out_en, self.sr2_out_en)
        ret += "4'b%1d%1d%1d%1d, "
        ret += "1'b%1d, " % self.flag_reg_en
        ret += "3'd%1d, " % self.comp_mode
        ret += "2'd%1d, " % self.alu_op
        ret += "9'd%03d, " % self.const_byte
        ret += "16'd%05d" % self.const_word
        ret += "}; // %s" % self.loc

        return ret

    def processCond(inst, cond):
        if (cond != None):
            if (cond.preds == None): preds = []
            elif not isinstance(cond.preds, list): preds = [cond.preds]
            else: preds = cond.preds
            for pred in preds:
                if (isinstance(pred, EOF)):
                    inst.pred_eof = True
                elif (isinstance(pred, SOF)):
                    inst.pred_sof = True
                elif (isinstance(pred, EQU)):
                    inst.pred_cmp = True
                    inst.alu_op = 1
                    if (pred.bytewide):
                        inst.comp_mode += 4
                    if (isinstance(pred.op0,C)):
                        inst.op0_mux_s = 0
                        inst.const_word = pred.op0.value
                    elif (isinstance(pred.op0,P)):
                        inst.op0_mux_s = 1
                    elif (isinstance(pred.op0,FR)):
                        inst.op0_mux_s = 2
                    elif (isinstance(pred.op0,R)):
                        inst.op0_mux_s = 3
                        inst.reg_addr = pred.op0.address
                    if (isinstance(pred.op1,C)):
                        inst.op1_mux_s = 0
                        inst.const_word = pred.op1.value
                    elif (isinstance(pred.op1,R)):
                        inst.op1_mux_s = 1
```

```
                inst.reg_addr = pred.op1.address
            elif (isinstance(pred, NEQ)):
                inst.pred_cmp = True
                inst.alu_op = 1
                inst.comp_mode = 3
                if (pred.bytewide):
                    inst.comp_mode += 4
                if (isinstance(pred.op0,C)):
                    inst.op0_mux_s = 0
                    inst.const_word = pred.op0.value
                elif (isinstance(pred.op0,P)):
                    inst.op0_mux_s = 1
                elif (isinstance(pred.op0,FR)):
                    inst.op0_mux_s = 2
                elif (isinstance(pred.op0,R)):
                    inst.op0_mux_s = 3
                    inst.reg_addr = pred.op0.address
                if (isinstance(pred.op1,C)):
                    inst.op1_mux_s = 0
                    inst.const_word = pred.op1.value
                elif (isinstance(pred.op1,R)):
                    inst.op1_mux_s = 1
                    inst.reg_addr = pred.op1.address
            elif (isinstance(pred, DST)):
                inst.pred_dst = True
            elif (isinstance(pred, SRC)):
                inst.pred_src = True
            elif (isinstance(pred, FCS)):
                inst.pred_fcs = True

    def processFlags(self, flags):
        if (flags != None and len(flags) > 0):
            for flag in flags:
                if (isinstance(flag, AEOF)):
                    self.eof_out = True
                elif (isinstance(flag, ASOF)):
                    self.sof_out = True
```

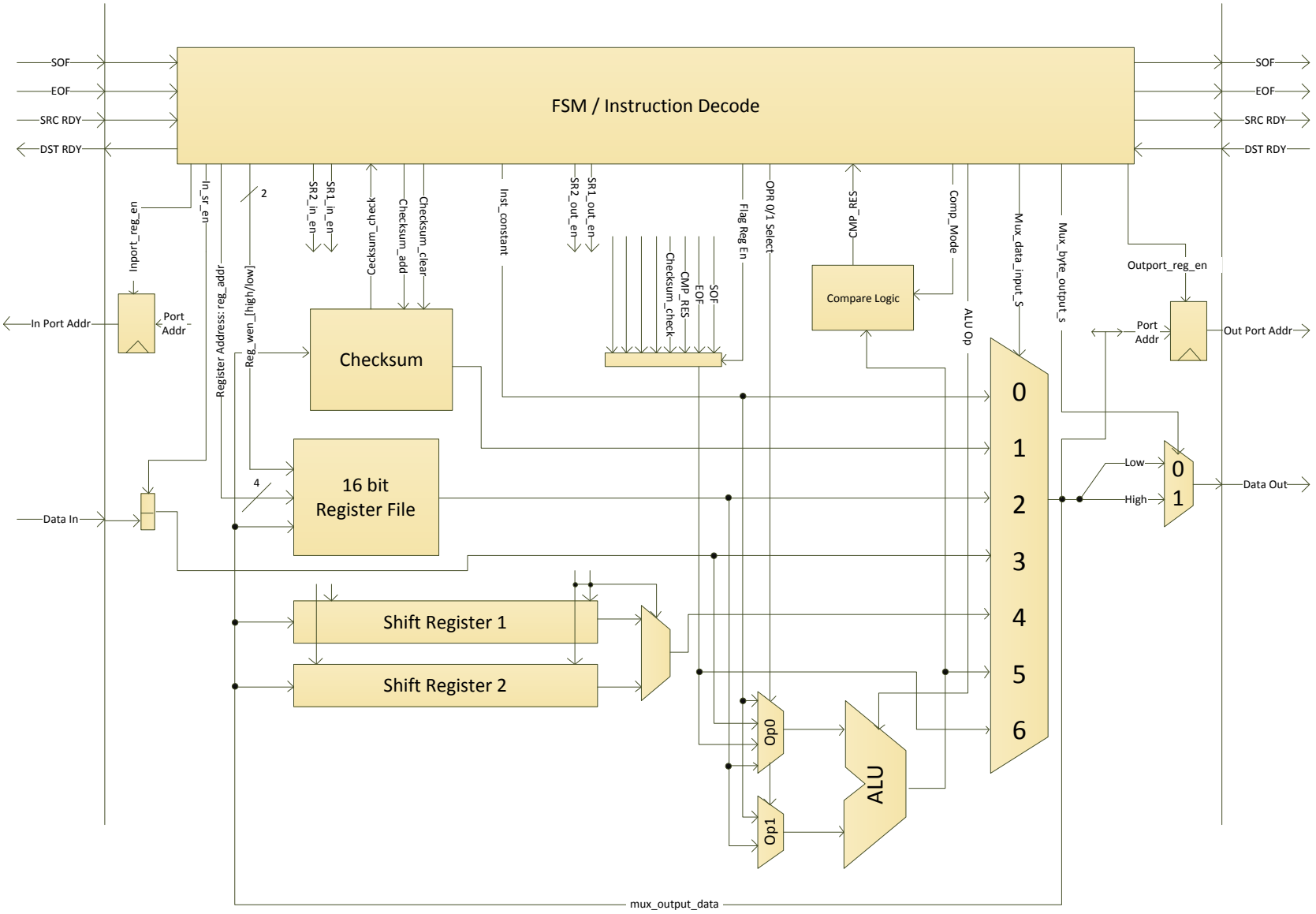

Figure C.7: Packet Processor