DEVELOPMENT OF A RECONFIGURABLE MULTI-FACETED COMMUNICATIONS DEVICE USING PARTIAL DYNAMIC RECONFIGURATION

by

Richard Dunkley

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

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Dr. Jacob Gunther Major Professor	Dr. Scott Budge Committee Member

Dr. Wei Ren
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2011

Abstract

Development of a Reconfigurable Multi-Faceted Communications Device Using Partial Dynamic Reconfiguration

by

Richard Dunkley, Master of Science

Utah State University, 2011

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

Supporting a variety of communication protocols has typically required extensive hardware and Input/Output (I/O) interfaces targeting each protocol specifically. Recent designs in the past ten years have created more dynamic approaches by using Field Programmable Gate Arrays (FPGAs) and embedded hardware to implement or simulate previous hardware I/O designs. With the constant increase in FPGA and embedded technologies the capabilities of dynamic implementations have expanded. This report addresses the design of an up-to-date reconfigurable multi-faceted embedded device targeting recent technological advances in FPGAs.

(60 pages)

Contents

List of Tables

List of Figures

Acronyms

API	Application Programming Interface
AXI	Advanced eXtensible Interface
BAR	Base Address Register
CAN	Controller-Area Network
CAPRE	Common Aircraft Portable Reprogramming Equipment
DDR3	Double Data Rate 3
DIP	Dual In-Line Package
DMA	Direct Memory Access
ECU	Engine Control Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIFO	First In First Out
FPGA	Field Programmable Gate Array
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
IO	Input/Output
IP	Intellectual Property
IM	Interface Module
JTAG	Joint Test Action Group
LED	Light-Emitting Diode
PCI	Peripheral Component Interconnect
PCIE	Peripheral Component Interconnect Express
PIO	Programmed Input Output
RMCD	Reconfigurable Multi-Faceted Communication Device
SM	System Module
TLP	Transaction Layer Packet
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

Chapter 1

Introduction

This document outlines the design of a Reconfigurable Multi-Faceted Communication Device (RMCD) targeting military and commercial communication interfaces. The device is reconfigurable because it can be reconfigured by software to support different interfaces or different combinations of communication interfaces on the fly. The device is multi-faceted because it supports a variety of military and commercial protocols and interfaces. The purpose for researching this design is to outline a possible next generation RMCD platform targeting the growing demand for higher performance devices in the market. RMCDs are typically used were it is advantageous to support a variety of communication interfaces without requiring numerous or redundant support equipment.

1.1 Background

Large complicated systems such as automobiles, military vehicles, and airplanes typically are made up of smaller independent systems working as a whole. Each component of the system may be developed by various commercial manufacturers with various motivations and requirements driving a variety of communication interfaces. For example, a Global Positioning System (GPS) unit for an automobile would not require the same speed and reliability in its communication as an Engine Control Unit (ECU). This typically is not a problem for end users but is a large problem for maintenance personnel who may need to interface to the individual components for software loading, test, or diagnostic purposes. This document refers to the various units contained in automotive and avionics systems as System Modules (SMs) in order to generalize the broad range of components in this category. SMs would refer to units within the system that personnel may need to connect to for software loading, test or diagnostic purposes. A few examples of SMs would be: ECUs,

line replaceable units, and weapons replaceable units.

The automotive industry has tried to mitigate the complication of various communication interfaces by standardizing their communication protocols and interfaces. As of 2008 all cars manufactured in the United States (US) support the Controller-Area Network (CAN) bus interface for on-board diagnostics [1]. The aerospace and defense industry has had a lot more difficult time standardizing due to the large number of manufacturer's involved and the lengthy life span of some of the systems. Furthermore, upgrades to older systems make it difficult to standardize new protocols without creating incompatibilities with existing ones.

Not only has the variety of communication interfaces increased but also the speed of communication. Ten years ago, most of the bus speeds used to communicate with SMs varied from KHz to MHz [2–4]. That range has now expanded into the GHz range with the potential for much faster bus speeds using fiber-optics [5,6]. This change in speed has been driven by more powerful capabilities of embedded systems. More powerful embedded devices have a larger capacity to capture and store data. As the amount of captured and stored data increases the amount of data communicated from the SM also increases. The speed of the interface is then increased as well in order to better accommodate larger data transfers. The requirements for increased speed and performance have introduced a variety of new higher speed communication protocols and interfaces into the industry, further complicating the problem.

1.2 Current Solutions

With a vast number of communication interfaces and protocols in the market a large number of interface modules have been developed in order to interface with the various SMs. An interface module is a bridge of communication between a computer system and the SM. The computer system may be based on an embedded system, but is usually a larger more capable personal computer. A Personal Computer (PC) is typically used to provide more advanced processing of the data pulled from the system and can be upgraded or exchanged to prevent obsolescence [7]. This document refers to the bridge between a computer system

and an SM as an Interface Module (IM). To clarify, RMCDs are IMs that are reconfigurable and target a variety of communication interfaces and protocols, whereas an IM may only target one.

Different organizations have attempted to mitigate some of the issues of developing IMs to connect to a variety of communication interfaces and protocols by developing RMCDs. The 309th Software Maintenance Group at Hill Air Force Base, Utah is one of those organizations. They are the primary provider of loader/verifier support equipment to the Air Force. They provide Common Aircraft Portable Reprogramming Equipment (CAPRE) to allow hookup to embedded systems aboard various military platforms. The equipment is typically composed of a ruggedized laptop and RMCD device. The 309th Software Maintenance Group has a direct interest in research in this area and has funded this project.

The purpose of this project is to develop a prototype RMCD that is a platform for providing various communication protocols and interfaces. Developing the various communication protocols and interfaces is not part of this project; however, a few example designs were implemented in order to prove the functionality and features of the platform.

Chapter 2
Limitations of Current Multi-Faceted Communication Devices

2.1 Bus Controllers

Most IMs currently on the market are hardware based. The IMs contain hardware to provide specific communication interfaces and protocols. These IMs are typically not reconfigurable. IMs which are reconfigurable typically use a series of relays to switch communication lines to different transceivers for various communication protocols. Devices of this type are typically limited to very few specific communication protocols. For higher speed applications this can get complicated and expensive. High-speed applications require high-performance relays that can maintain a connection without affecting the low voltage high-speed signals passing through them. Depending on the speed requirements of the device, this could increase the relay's cost from less than a dollar to thousands of dollars. The communication device may also require additional environmental constraints making it difficult, if not impossible, for a series of relays to meet the requirements.

Another limitation of hardware IMs is that SMs may require coordination of multiple communication protocols. For example, a module may require communication to take place over RS-422 [8] and MIL-STD-1553 [9] simultaneously. The commands are sent over RS-422 and the data is transferred over MIL-STD-1553. Current hardware-based solutions with separate drivers and software APIs may not be able to accommodate systems where coordination must be made between the two protocols at a low level. Recent devices have mitigated this problem by using Field Programmable Gate Arrays (FPGAs) for reconfigurable logic to allow for a broader range of custom catered communication protocols [10]. FPGAs also have a growing number of Intellectual Property (IP) communication cores that

target various communication protocols and interfaces. SMs requiring low-level coordination between two protocols can be implemented by combining multiple IP cores into a single design. FPGAs can also be interfaced with proprietary or custom interfaces that break standard protocol rules or implement non-standard communication. Changing the hardware to support common or custom interfaces is done by simply reconfiguring the FPGA.

The problem with FPGAs in an RMCD is that typically the user needs the ability to reconfigure the FPGA to transition from one communication protocol to the next. This becomes a difficult problem depending on how the FPGA is interfaced to the computer. The bus interface from the computer to the RMCD could be one of a variety of peripheral buses supported by typical computer systems. For example, Peripheral Component Interconnect (PCI) [11], PCI Express (PCIE) [12], Universal Serial Bus (USB) [13, 14], RS-232 [15], IEEE-1394 [16], and Ethernet [17] are all common computer system peripheral buses. Connecting the FPGA directly to the bus interface does not work because the FPGA must be reconfigured and once the FPGA begins reconfiguration the bus interface is no longer valid. Some bus interfaces such as PCI and PCIE require that the bus interface is stable within a certain period of time after the computer system is powered on [11, 12]. Dropping a bus interface such as PCI or PCIE after the system boot-up to allow for reconfiguration may cause the computer system to lockup or crash. These limitations require that the device contain a bus interface controller that sits between the FPGA and the computer's communication bus. This setup allows the device to remain connected to the PC while it reconfigures the FPGA. Figure 2.1 shows an example of this type of system. This design has been used for a variety of applications involving FPGAs [10], but it also has some inherent limitations. Introducing a bus controller into the system increases the latency, bottlenecks the maximum bandwidth of the system, and may limit performance based on cache requirements. The following sections give a more in depth explanation of these issues.

2.1.1 Latency

By adding an additional layer to the system the overall latency of the system increases. The bus controller must process every byte of data that is passed through it. In addition to

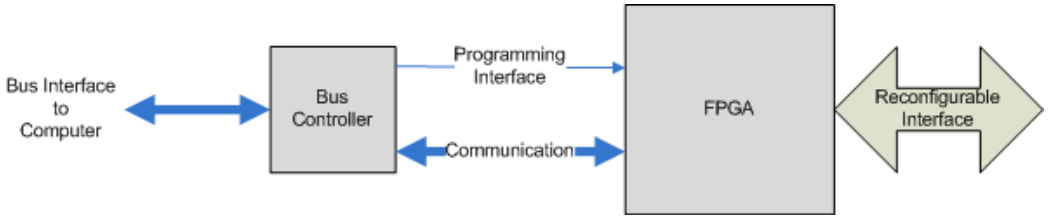

Fig. 2.1: Interfacing an FPGA to the computer system's bus using a bus controller.

this it may also be required to transfer the format of the data. For example, let's assume the bus interface is USB. The bus controller would be required to process the incoming USB packet and strip off all the USB specific packet information. It then may need to repackage the data into a different packet prior to forwarding the data to the FPGA. Even with a simple interface between the bus controller and the FPGA, the bus controller may still be required to handshake signals or verify available space in the FPGA prior to sending the information. This increases the overall latency of the system.

Most computer peripheral buses contain mechanisms to control the flow of data or determine prior to sending whether the receiver is capable of receiving the data. USB and Ethernet perform this mechanism by dropping incoming packets [13, 14, 17]. Dropping incoming packets forces the sender to resend the packet later. PCIE has the ability to do this by exchanging tokens between the sender and receiver [12]. The sender tracks the number of tokens the receiver has available and therefore knows when it will be able to receive a certain amount of data. Adding a bus controller in between the recipient and sender breaks these mechanisms and requires that the mechanism either be repeated between the bus controller and the FPGA or mandate that the FPGA always perform real-time processing of the data. The later is not really possible for a device targeting a variety of communication protocols.

Adding a data flow control mechanism between the FPGA and bus controller converts the bus controller into a form of router or store-and-forward device. Figure 2.2 shows two flow diagrams that illustrate the additional latency added with the bus controller. The diagram on the left shows a packet transfer to and from the device directly without the use of a bus controller. The diagram on the right shows a packet transfer to and from the device with the use of a bus controller. Adding the bus controller in the middle adds an additional

layer and may double the overall latency of the system. For large packet transmissions the additional latency may be negligible, but for single byte transactions the additional latency could be crippling. Even with the added latency, FPGA systems of this type are commonly used, but finding a solution to this problem would be beneficial.

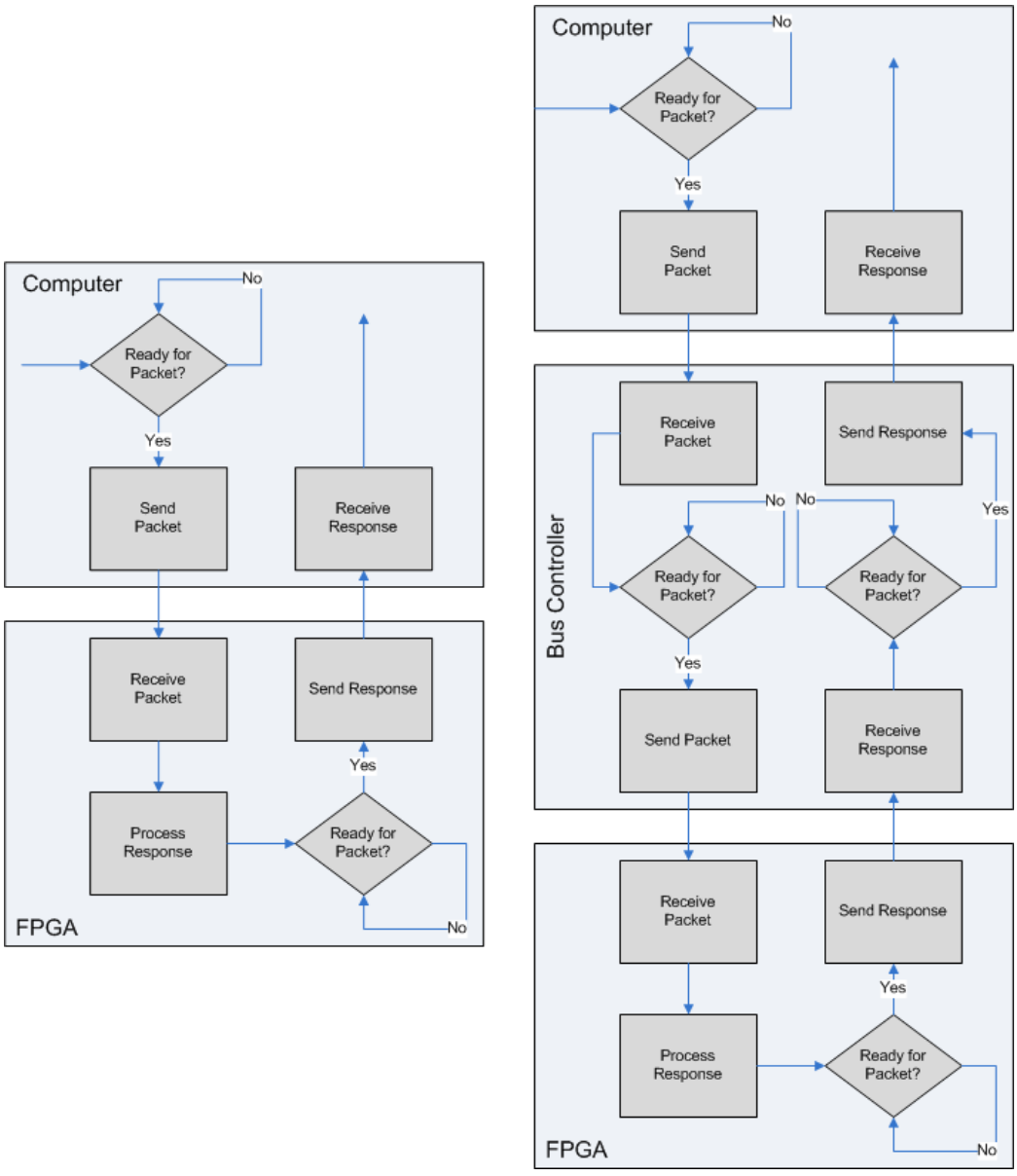

Fig. 2.2: Packet flow from the computer to the FPGA logic with and without a bus controller.

2.1.2 Maximum Bandwidth

Adding a bus controller in between the FPGA and the computer's bus interface typically requires a microprocessor to perform the data processing prior to sending the data to the FPGA. Bus controllers that are designed to perform this type of interface between an FPGA and computer peripheral bus are sparse. The reason for this is typically the programming interface to the FPGA can vary from one FPGA to another. Bus controllers require built-in transceivers and hardware to perform the duties required for the bus transactions. It is not cost effective to design a hardware chip that both interfaces to a peripheral bus and a specific group of FPGAs. A variety of manufacturers provide bus controllers that include embedded microprocessors to allow the bus to interface to custom logic. This makes for a low-cost solution to interfacing with the FPGA, but typically limits the speed of the FPGA. Embedded microprocessors are driven by software that consumes many more clock cycles than hardware implementations. The difference in consumed clock cycles can quickly create a large performance gap between hardware and software driven Integrated Circuits (ICs). A single routine in software may require hundreds of overhead processor cycles before the code inside the routine is run. For example, a microprocessor typically pushes all its internal registers to the stack prior to jumping to an interrupt service routine. The hardware logic in the FPGA does not have these limitations. In addition the FPGA logic may process information in parallel or in a pipelined fashion making its overall bandwidth greater than a microprocessor with sequential constraints.

Embedded microprocessors are typically limited to a small number of speeds at which they are capable of running. For example, a microprocessor embedded in a bus controller IC may have a low-power sleep state, a lower-power slow speed, and a higher-power full-speed mode. This is a poor match to an FPGA which runs at a speed based on the current logic it is programmed with. Adding a high-speed microprocessor as a bus controller gives a higher throughput with less chance of bottlenecking the FPGA but also consumes a large amount of power when used for slower FPGA designs. Higher-speed microprocessors increase the cost of the design and may be difficult to find depending on the targeted bus. Even if the

Central Processing Unit (CPU) cycle overhead of the microprocessor is relatively low, it would be required to run at a lot higher clock speed than the FPGA in order to keep up. For example, if the microprocessor processed one byte of data every four CPU clock cycles it would have to have four times the clock speed of the FPGA to keep up. If the FPGA had a clock speed of 400MHz, the microprocessor would need a 1600MHz clock. This can quickly push the requirements of the microprocessor to beyond what is currently capable. A number of fibre-channel routers have used high performance FPGAs as data processors in the system to avoid the performance limitations of microprocessors.

2.1.3 Performance Limitations Due to Cache Constraints

Using a bus controller as a store-and-forward device requires that the controller is capable of caching packets of data flowing through the system. For low-speed applications this can be limited to a relatively small amount of cache, but for higher performance scenarios larger transfers are required to decrease the ratio of packet overhead to packet data.

Direct Memory Access (DMA) transfers would also be required for high performance applications. Most high-speed computer peripheral buses accommodate DMA transfers, but adding a bus controller in between the FPGA and the computer would complicate the transactions. The data from the DMA transfer would either have to be cached in the bus controller or the latency of the overall system would have to be increased in order to avoid reversing the performance advantages of using DMA.

The above reasons have made it difficult to design an RMCD with a bus controller IC that does not bottleneck the system at some point. For previously designed RMCDs, this has not been a problem since the speed and performance requirements were low compared to the speed and performance of the components within the system; however, the next generation of RMCDs could benefit from avoiding any performance degradation in the system.

2.2 High-Speed Peripheral Bus

One of the problems with developing an RMCD is that the bus interface that connects

the device to the computer can quickly become obsolete. Typically, the lifespan of computer peripheral buses is small compared to the lifespan of an RMCD target application. This problem is especially prevalent with aerospace designs where the lifespan could be 20 or 30 years. RMCDs that were developed to interface with the USB revision 1.1 may have been faster than required at the time of their release, but are now severely limited. Designing an RMCD to target future applications while implementing a current peripheral bus can be difficult.

Chapter 3

Design of a Reconfigurable Multi-Faceted Communication Device

3.1 Dynamic Partial Reconfiguration

One of the hot topics of FPGA technology in the last four years is partial dynamic reconfiguration. Partial dynamic reconfiguration is the ability to dynamically reconfigure part of an FPGA during runtime. This means that it is no longer required to configure the entire FPGA prior to starting its logic. In addition, portions of the FPGA logic can be reconfigured without shutting down the FPGA or affecting other parts of the FPGA currently running.

The previous chapter addressed limitations of the system by requiring the use of a bus controller. Using partial dynamic reconfiguration, the FPGA can be directly connected to the computer's peripheral bus. On power up, the bus controller portion of the FPGA would be initialized. This is referred to as the static portion. The user's software application could communicate with the FPGA through the static portion and then partially reconfigure the remaining portion of the FPGA to provide the various communication protocols and interfaces required on demand. This would give direct communication with the configurable hardware from the computer and remove some of the limitations discussed previously. Removing the need for an additional bus controller chip within the RMCD would also reduce the cost of the device.

Implementing an RMCD that uses partial reconfiguration to reconfigure the non-static portion of the chip is the primary objective of this design project. The additional problems of integrating the FPGA output transceivers with various physical layer requirements is beyond the scope of this project and is left for future work.

3.2 Computer Peripheral Bus

The previous chapter addressed problems with current peripheral bus obsolescence when targeting future RMCD applications. For this design, a peripheral bus was chosen that would allow support for current high performance communication protocols while minimizing future obsolescence. Three standard bus interfaces were chosen that are currently backwards compatible and also provide for increased speed in the future with minimal impact to the design. The interfaces are USB, Ethernet, and PCIE. Each bus interface currently has support for gigabit speeds and is commonly found on most computers [12, 14, 17]. The following sections contain the advantages and disadvantages of each.

3.2.1 USB

Advantages

USB is found on almost all computers manufactured within the last 10 years. The latest version (3.0) provides a pin compatible interface to USB 2.0 and USB 1.1. USB is a common interface and a variety of bus transceivers and IP cores are available at a relatively low cost.

Disadvantages

USB was originally designed for low-speed applications such as mice and keyboards. Data flow control and efficiency were not major motivating factors during its initial design. Subsequently the protocol has been known to contain a lot of overhead which reduces its performance when compared with slower protocols. USB is not a point-to-point protocol. A USB device cannot directly transfer information to the host without having the host request the information first. This requires a polling mechanism on the PC side which typically adds additional latency to the system. Additional caching on the device may be required depending on the speed of the polling mechanism. USB 3.0 contains a point-to-point mechanism in its protocol [14], but this may eliminate some backwards compatibility with USB 2.0 and USB 1.1 interfaces.

3.2.2 Ethernet

Advantages

Ethernet protocols can be implemented using IEEE standards. Ethernet can be found on almost all computer and laptops and is backwards compatible with older slower speed protocols (100BASE-T, 10BASE-T) [17]. Ethernet can be used as point-to-point interface which means the device can transfer the data immediately back to the host computer. This reduces the cache requirements of the device and provides almost unlimited storage on the PC side. If required ethernet can use longer cables than USB and PCIE.

Disadvantages

Different firewalls and security software running on the host PC may block software from using Ethernet protocols to communicate with the RMCD. The security software may require the user to intervene to configure the firewall to allow the communication to take place. In order to mitigate conflicts with firewalls and security software when communicating with the device, it would be preferred to use a higher level TCP/IP protocol such as HTTP or HTTPS. These protocols are not typically blocked by firewalls and could allow the communication to take place. Higher level protocols would add a lot of complexity to processing the data packets on the FPGA side. The bus controller on the device side would have to perform a lot of processing to remove the data from the packets. This would increase the overall latency of the system.

3.2.3 PCI Express

Advantages

PCIE is one of the few protocols which provide high bandwidth and low latency [18]. It is designed to be scalable. The speed of the protocol is increased by adding additional lanes to the device. The bus controller on the device would need to modify its data link layer, but would not need to modify the above logic to increase the speed. PCIE is also the

only bus out of the three that uses tokens to track the amount of space available on the other end [12]. It does not send a packet until it knows the packet can be received. This cuts down on wasted bandwidth. Both Ethernet and USB buses send the packet and then resend if the packet was dropped [13, 17].

Disadvantages

PCIE is typically used as an internal peripheral bus. The PCI consortium has approved an external cabling specification, but its commercial applications have been sparse. Currently using PCIE would limit the computer's connections to inside the computer or through an ExpressCard Interface [19].

3.2.4 Selection

This design uses PCIE to interface the RMCD design to the computer system. In addition to the advantages stated above, most recent high end FPGAs have built in transceivers for PCIE so interfacing with the bus would not require additional transceiver ICs.

3.3 Development Board

The development board used for this project is one provided by Xilinx as an evaluation module for their Virtex 6 line of FPGAs. The board was chosen because of its various capabilities and interface options. It contains a PCIE interface which is the primary means chosen for communication between the computer system and RMCD. It also has a variety of other interfaces which will allow demonstration and evaluation of the capabilities of the design. Table 3.1 contains the development board specifications. The board also contains a number of additional peripherals that are interfaced to the FPGA. Table 3.2 lists some of those additional peripherals.

3.4 Hardware Overview

The hardware design of the RMCD is the hardware logic contained in the FPGA. All the hardware logic was developed using VHDL. The Xilinx Embedded Development Kit was used to synthesize and route the logic.

3.4.1 Static vs Dynamic

Using dynamic partial reconfiguration on an embedded FPGA within the device allows the FPGA to connect directly to the computer's peripheral bus so that the FPGA acts as both the bus controller and the data processing system. Figure 3.1 shows a basic overview of the system. The static portion of the FPGA is the part which is configured on boot up and interacts with the system's bus. The static portion may also have some additional logic to control additional features of the RMCD (voltage regulators, cable detection, etc.). The dynamic portion of the FPGA is the portion that is reconfigured by the static portion to provide the various communication protocols and interfaces.

The interface between the static and dynamic portion of the FPGA is the interface required by all communication protocols implemented in the system. Since the target applications of the device range from simple serial communication to complex embedded systems, the communication interfaces would need to accommodate both targets. Making a complicated interface would drive up the development costs of simple applications and making an interface that was too simple may limit the capabilities of the device to implement more complicated designs. In the end, the interface between the static and dynamic portions of the FPGA was designed similar to the I/O interface used by the processor to access peripheral devices. This would allow a driver to be developed that is transparent to the endpoints of the system and provides relatively simple interfaces for implementation of

Table 3.1: Development board specifications.

Description	Virtex 6 FPGA Embedded Kit
Part Number	DK-V6-EMBD-G-XP1
Target FPGA	Virtex-6 LX240T (XC6VLX240T-1FFG1156)

Table 3.2: Development board features.

Feature	Description
PCI Express	Provides 8-lanes of Generation 1 PCI Express or 4-lanes of Generation 2 PCI Express
External Memory	Provides the FPGA access to 512MB of DDR3 Memory
Compact Flash	Provides the FPGA with access to a Compact Flash card through a CF interface.
Ethernet	10/100/1000 Tri-Speed Ethernet
Serial Communication	UART 16550, I2C and SPI communication interfaces
USB	On-The-Go (OTG) and Host Interfaces
Other I/O	DIP Switches, LEDs and push-buttons for user interaction with the board

both complicated and simple designs. The bus protocol between the computer and the RMCD could be replaced in the future without requiring the interface between the static and dynamic portions of the FPGA to change. If the bus protocol used became obsolete the RMCD could be modified without requiring changes to the dynamic logic.

All peripheral devices in a computer system interact with the processor by reserving blocks of addresses in the computer system's address space. The blocks map to registers or memory located within the device. The operating system then interacts with the device through a driver that reads and writes to the memory addresses to control the hardware. The device would contain a range of registers which could be accessed similar to memory.

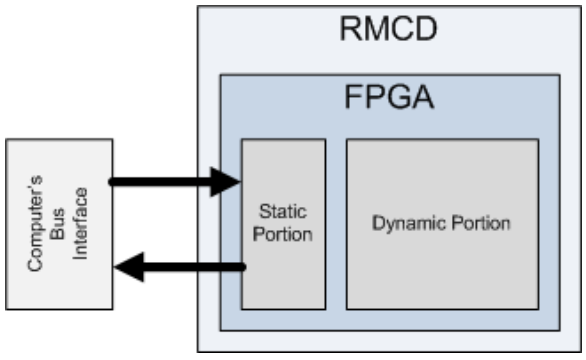

Fig. 3.1: Basic overview of an RMCD design using partial reconfiguration.

The device would also need to cater to simple designs which only need to handle the transfer of data and do not depend on a large amount of registers for control. Having all the data transfers addressable would give additional complexity and overhead to future simple designs, so all the communication to the device was categorized into two types. The two different types of communication are addressed-based transfers and packet-based transfers.

Address-Based Transfers

Address-based transfers are transfers of data to or from addressable registers on the device. Addressed-based transfers are used to control the device and are not the primary means of transferring data (although they could be if the communication protocol required it). The control registers on the device are addressed based on a base address of 0x000000, but can be mapped externally in the computer's memory system to various locations (See Fig. 3.2). The address range is byte addressable and has a range of 0x000000 to 0x1FFFFF giving an address space of 2MB. A basic overview of the signals required to implement the interface can be seen in Fig. 3.3.

The addressable registers on the device are used by both the static portion and the dynamic portion. The static portion uses the registers to provide the computer system with access to various features that are provided to all communication protocols implemented on the device. For example, the dynamic portion of the FPGA is reconfigured using registers in the static portion. The static portion also contains status registers that describe the state and configuration of the dynamic portion. To accommodate the need for register access in both the static and dynamic portions, the address space of 2MB was split with the lower addresses corresponding to the static portion and the higher addresses corresponding to the dynamic portion. Splitting the memory space in half allows the driver to isolate the static region to kernel mode access only. This prevents unauthorized access to RMCD capabilities by end user applications.

Registers located in the address space (dynamic and static) are accessed by the computer system using a polling mechanism. The host application is limited to reading and writing of the registers only. The device does not delay reads based on data availability.

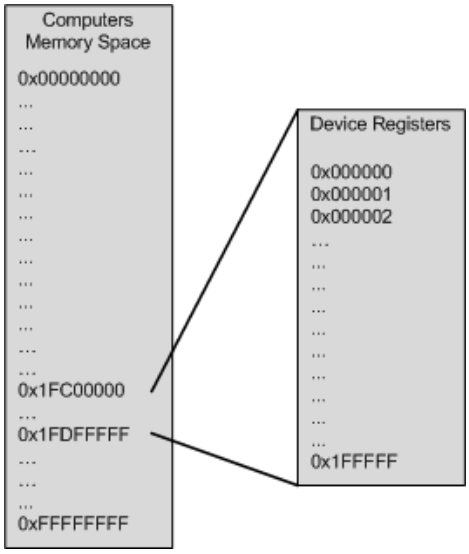

Fig. 3.2: External mapping of internal device registers.

Packet-Based Transfers

This type of transfer is recommended to be the primary method of transferring data to and from the device. The data contained in the packet does not correspond to specific register addresses. The data may be organized into packets complete with headers, checksums, or other packet information or it may be organized into simple chunks of data ready to be transmitted. The organization of the data is dependent on the user application and communication logic running in the dynamic portion of the FPGA. Control information may be embedded within the data if required by the end communication protocol. Packet transfers are performed by the driver on the computer side similar to address based transfers. A block of addresses is reserved in the system's main memory to provide the capability of writing to the device, but when the writes occur the addresses are abstracted from the associated data and any writes to the address block regardless of the location of the write will send data to the device in the order of the time of the writes instead of the address location. The address range in the system's main memory consists of an address space of 2KB. A basic overview of the signals required to implement the interface can be seen in Fig. 3.4.

Future users of the device may need to have multiple communication interfaces running

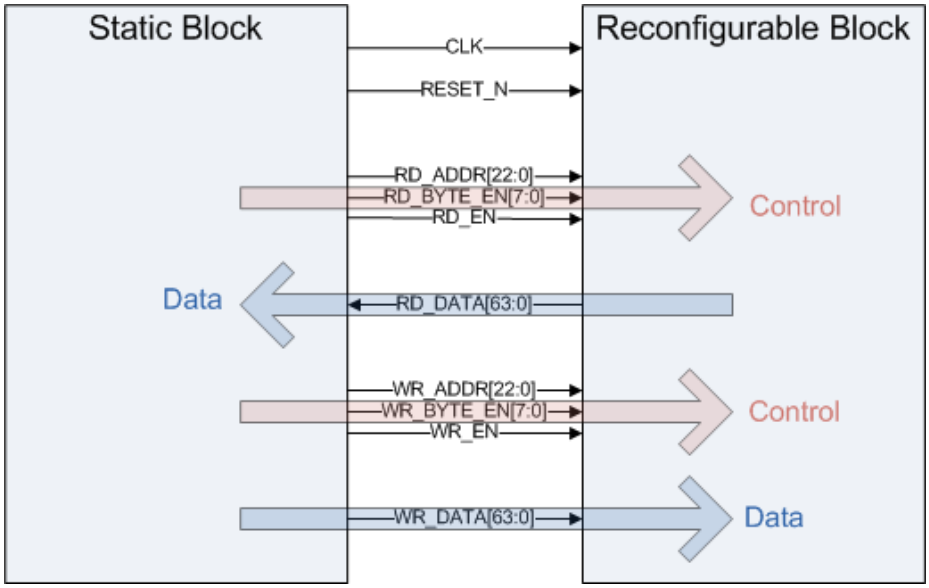

Fig. 3.3: Communication lines required for address-based transfers with data flow represented by transparent arrows.

simultaneously on the device, so four packet-based transfer channels were added to the device to allow for up to four independent channels of packet transmission. Each channel will have the ability to transmit and receive packets simultaneously (full duplex).

Packet-based transfers are different from address-based transfers because the interface may or may not be ready to receive or transmit data. A simple byte count mechanism was developed in order to prevent buffer over-runs or under-runs on either side of the interface between the static and dynamic portions. The mechanism gives status to the static portion on how many bytes can be read or written from the interface.

Figure 3.5 shows an overview of the system with the four packet transfer channels and the address space in the dynamic section. The PCIE bus controller is shown by using the PCIE Endpoint Block IP core provided by Xilinx [20]. The endpoint block uses PCIE transceivers built into the Virtex 6 processor hardware. The PCIE endpoint block exposes two transaction interfaces for transmitting and receiving information. Each transaction interface has a corresponding controller in the static logic that drives the interface.

The reader should refer to the PCIE specification [12] for detailed information on the implementation of the protocol; however, there are a couple of points worth mentioning.

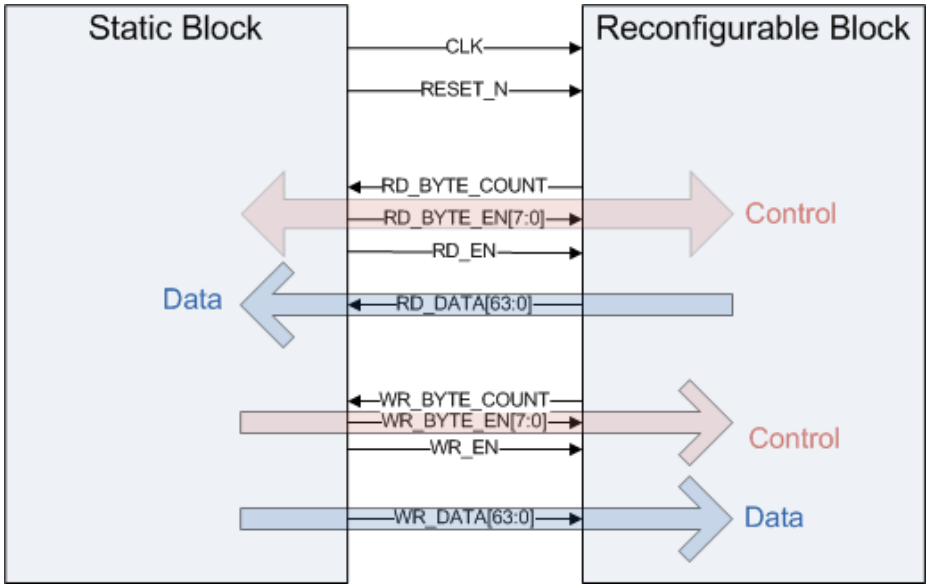

Fig. 3.4: Communication lines required for packet-based transfers with data flow represented by transparent arrows.

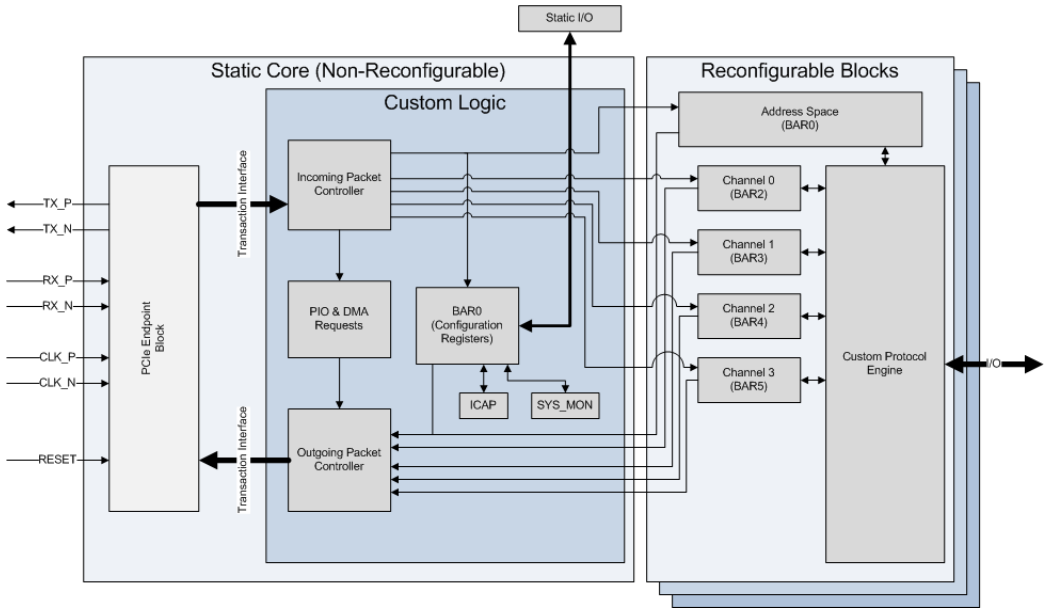

Fig. 3.5: Overview of the FPGA hardware logic design.

The PCIE specification uses Base Address Registers (BARs) to map the devices block memory into the systems main memory. The specification allows for six BARs which in turn allows the device to map six regions into system memory. The BARs for this project correspond with the device block memory regions. BAR0 is the address space corresponding to the addressable registers. BAR2, BAR3, BAR4, and BAR5 correspond to the packet transfer channels as shown in the figure. Both the static and dynamic portions have access to portions of the BAR0 space. The ICAP shown in the figure represents the Internal Configuration Access Port (ICAP) inside the FPGA that is used to reconfigure the dynamic portion of the FPGA. Future reference to the BAR2, BAR3, BAR4, and BAR5 blocks and associated logic will be referred to as channels representing the packet communication channels that they implement.

Typical partial reconfiguration designs that use the ICAP to reconfigure portions of the FPGA use a MicroBlaze soft IP core processor to provide a reconfiguration Application Programming Interface (API). The MicroBlaze IP core is provided by Xilinx for use with designs that target their line of FPGAs. Xilinx also provides an IP core for accessing the ICAP interface from within a MicroBlaze. The ICAP IP core comes complete with an embedded driver and software API to abstract a lot of the complications of the interface. The MicroBlaze design provided by Xilinx was avoided because of the FPGA resources that it would consume. A simple hardware interface to the static configuration registers would consume the least amount of resources and the least amount of power when running. The amount of resources consumed by the static portion of the FPGA was a concern because increasing its resources would decrease the amount of resources that could be used by the dynamic portion. Future designs may push the limit of resources in the dynamic portion so minimizing the static portion would be advantageous.

3.4.2 DMA

Figure 3.5 shows a PIO & DMA block in between the incoming and outgoing PCIE packet controllers. Accesses to the device from the computer system's processor can be classified into two categories: Programmed Input Output (PIO) and DMA. PIO is used

when the processor performs a memory read or memory write to an address corresponding to the device's mapped registers. The processor waits until the read or write has completed before continuing on. PIO is the simplest way of accessing the device. It is also limited to bit sizes small enough to fit into the processor's internal registers (for x86 this is 8-bit, 16-bit, or 32-bit accesses). Because the processor stalls, a large amount of processor clock cycles are consumed when transferring a large amount of data. This usually results in delayed performance of the system and stutters in the responsiveness of the operating system. DMA, on the other hand, gives the responsibility to the device to transfer the data. The CPU sets up a block of memory to store the transferred data and then continues on performing other tasks while the device transfers the data to or from the block. The device signals the processor with an interrupt when it has completed and the processor is then able to access the data.

The RMCD device developed for this project only provides PIO access. DMA would be required to create a marketable device for high performance use. Due to the additional complexities of implementing DMA, it was left for future work. The DMA is listed in Fig. 3.5 to represent the portion of the logic that would handle the requests and provisions were made in the current implementation of that logic to allow for further addition of a DMA controller in the future.

3.4.3 Incoming Packet Controller

The incoming packet controller is responsible for receiving and processing Transaction Layer Packets (TLPs) from the PCIE endpoint block. Figure 3.6 shows a block diagram of the incoming packet controller. The controller portion of the module determines the destination of the packet and enables the appropriate processor. The processors are divided up according to destination. The Request Processor handles incoming packets that are requests for data. These requests are sent to the PIO module for ordering and then finally passed to the outgoing packet controller to allow the reply to be transmitted. The BAR0 processor handles incoming packets that contain data to be written to the BAR0 address registers. It serves both static and dynamic registers. The BAR2 - BAR5 processor handles

packets that contain data to be written to the various communication channels.

The module also contains a data realign module. Since the dynamic portion is unknown, the amount of data transferred should not be limited to specific packet sizes. Similar designs require certain data aligned packets in order to simplify the logic in pulling data from TLPs. The complication arises from the PCIE endpoint block's 64-bit bus. The PCIE specification breaks the data up into 32-bit words similar to PCI. In order to process large packets quickly the PCIE endpoint block uses a 64-bit bus. 64-bit buses are used through the static portion to the address registers and communication channels of the dynamic portion to provide maximum bandwidth. Depending on the address the incoming data may not be correctly aligned with the appropriate data lines.

For example, if a TLP packet was received that writes two 32-bit words to address 0x000000 in the device the 64-bit incoming transaction bus from the PCIE endpoint block would contain the byte addressed to 0x000000 in data lines 63 to 56. Lines 55 to 48 would contain the byte for address 0x000001 all the way down to lines 7 to 0 representing address 0x000007. There would be no need to realign the data. If a packet was received that contained a starting address of 0x000004 the incoming transaction bus would contain the first byte on data lines 63 to 56, but these would need to be transitioned to 31 to 24 prior to writing the value to the registers. The fourth byte coming in on lines 31 to 24 would need to be stored until the next write since it would align to 63 to 56. To complicate things even more, TLPs can be received with three or four word headers. A three word header would contain the first byte of data on lines 31 to 24 and a four word header would contain the first byte of data on lines 63 to 56 regardless of the target address.

In order to rearrange the data lines to the appropriate location prior to processing the data, the Data Realign module was added between the TLP transaction bus and the destination processors. The data is then rearranged and possibly stored till the next clock cycle if required.

3.4.4 Request Processor

Request processing is handled by the PIO & DMA requests module (as seen in Fig. 3.7).

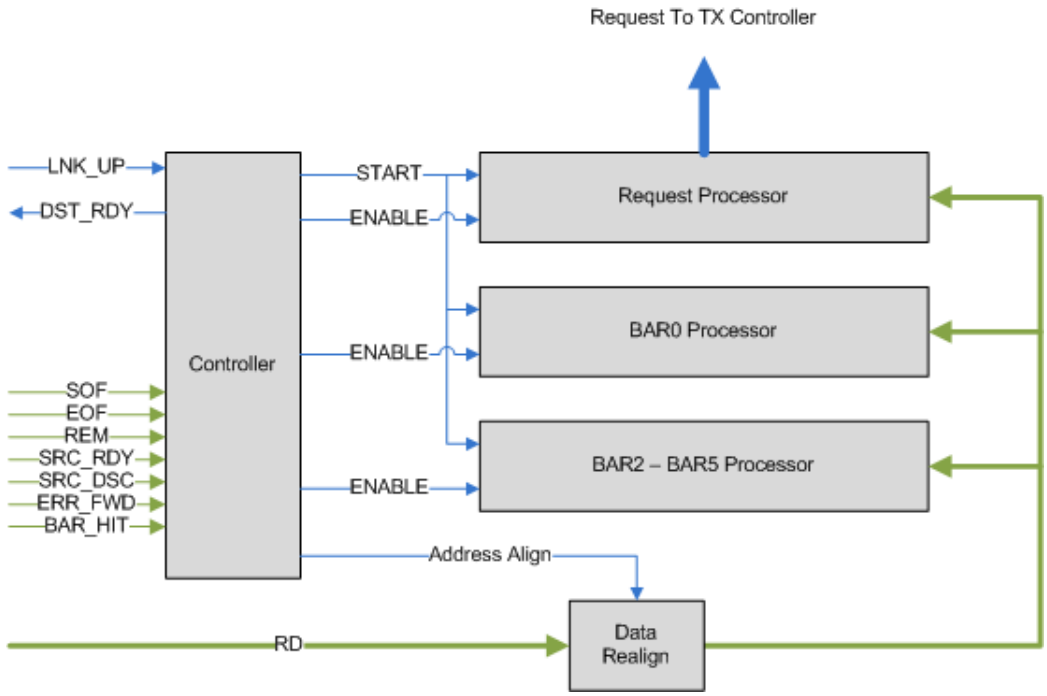

Fig. 3.6: Overview of the Incoming Packet Controller hardware logic design.

It receives the request information from the incoming packet controller and then prioritizes the requests prior to submitting the ordered requests to the outgoing packet controller. It may seem that providing multiple communication channels might lead to starvation of certain channels. For example, if the device processes the requests on a first come first serve basis a communication channel that is relatively slow such as a 1200 baud Universal Asynchronous Receiver/Transmitter (UART) would bottleneck the entire system. To prevent this, the system was designed to allow the request that is currently ready (has the requested amount of data) to preempt an earlier received request that is still waiting for the incoming data to be received. This solved the bottleneck problem but could also introduce an out of order problem. If a channel received a request for 64 bytes and then a request for 1 byte, the initial 64-byte request should not be preempted with the 1-byte request on the same channel. Received requests had to be ordered for a given channel based on the time that they were received. Another scenario could arise where status or control requests may need to preempt large data packet transfers. For example, if a user needed to cancel a blocked packet transfer. To correct this problem, BAR0 requests are given higher priority

over packet requests. BAR0 requests could then be used for status or control requests.

In order to achieve the desired results, the incoming packets would need to be separated into individual queues for each of the corresponding destinations. The separate queues would maintain order within a specific destination based on the order the requests were received. The selector would then prioritize the next request for each destination and select the appropriate one for transfer. If a BAR0 request was made the selector would always select it as the next packet to be transmitted. This would give higher priority to BAR0. Selecting BAR0 first also makes sense because it is the only type of request that does not need to wait until the requested amount of data has been received. If a BAR0 request was not currently pending then the selector would choose the next channel request that was ready.

Giving BAR0 the highest priority could cause starvation of the other channels. Back-to-back requests to BAR0 would always be processed causing the other channels to continue to wait. This is why BAR0 was designated for status and control transfers and should not be used for large data transfers.

If no BAR0 requests are pending, but multiple channel requests are ready then the selector needs to know which request has been in the system the longest. A queue would need to be implemented to provide the ordering of the channel requests as they arrived in the system. Storing the requests in multiple queues would consume a lot of FPGA resources so a virtual queue system was developed. The virtual queue contained a head controller and a tail controller.

When a channel request arrives, the head controller assigns an index value to the incoming request prior to placing it in its corresponding channel queue. The head controller would then increment its stored value in preparation for the next incoming request. This would maintain an ordering of the requests in the system. This system was simple, but would function incorrectly when the value in the head controller hit its maximum value and rolled over. To allow for rollover in the system the tail controller was implemented.

The tail controller tracks the value of the channel request that has been in the system

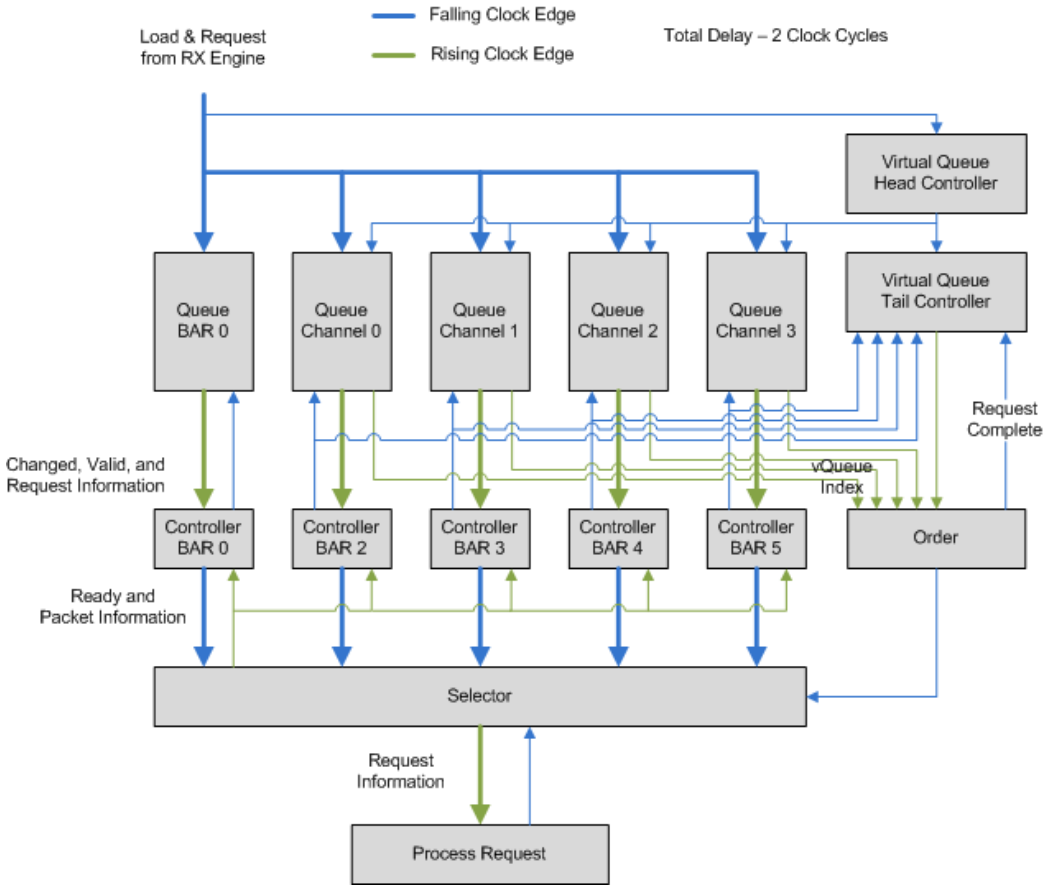

Fig. 3.7: Overview of the request processing hardware logic design.

for the longest amount of time. Determining the order of the requests is then determined by the order module. It does this by subtracting the tail value from all the queue indexes of the requests of each channel queue that is next in line. It then orders the four channels based on their normalized queue index. This maintains order even when rollover occurs. This system is slightly different than a typical implementation of a queue because the tail value may or may not be incremented when a request is transmitted. The requests can be transmitted out of order based on whether the channel's data is ready to be transmitted or not. When a request is transmitted the tail controller looks at the remaining requests at the top of each channel queue and assigns its value to the request whose index value is the next in line after the transmitted request's index value.

If no BAR0 request is pending the selector module uses the prioritized order from the

order module to determine which channel request should be transmitted. The selector steps though each request in the prioritized order and transmits the first request that it finds that is ready to be transmitted. The readiness of a request to be transmitted is determined using a variety of factors. The PCIE specification allows for partial packets to be returned when a request is made. This feature was implemented to allow part of a request to be transmitted in between other requests in order to keep the system responsive. This reduces the cache requirements of dynamic designs and maximizes the data flow in the system. The reader should refer to the PCIE specification [12] for the requirements of partial packets for request replies.

The virtual queue system does have an inherent flaw. If the system has one slow communication protocol on one channel and more than one fast communication protocol on the other channels, a situation could arise where a request is pending in the slow channel's queue and numerous other fast channel requests pass through the system. This could cause the value in the head controller to rollover and pass the value in the tail controller. A request assigned to an index value larger than the current tail controller value might then have higher priority than previously assigned fast requests. The consequences of this scenario did not seem to cause any real problems in the system since it would only reverse the priority of the fast channels which would only be incorrect if both the fast channels were ready to transmit. Starvation would not occur since the header would continue to count causing the order to correct itself eventually. This problem could also be mitigated by increasing the count size of the head and tail values. Speed and simplicity of the current design outweighed the possible detrimental effects of such a rare occurrence so the virtual queue system was implemented.

In the end, the implementation of the request processing system contained a delay of only two clock cycles. This is a significant performance improvement over similar designs and significantly reduces the latency of the system.

3.4.5 Outgoing Packet Controller

The outgoing packet controller, as seen in Fig. 3.8, receives requests from the PIO and

DMA request module when they are ready to be transmitted. The controller module waits for the PCIe Endpoint Block transmit transaction interface to signal that it is ready for a Transaction Layer Packet (TLP) to be transmitted. When this occurs the controller enables the desired processor to read data from the corresponding location. The various processors are in charge of aligning the data with the appropriate data lines on the transaction data bus similar to the data realign module in the incoming packet controller. The Data Bus Switch shown in the figure connects the appropriate outgoing data to the transaction interface data bus.

3.4.6 ICAP Controller

The ICAP interface used to reconfigure the dynamic portion of the FPGA is controlled using custom logic driven by a 50MHz clock. The ICAP interface requires a different clock than the PCIe Endpoint Block clock provided to the static portion system logic. The ICAP interface also requires some special handling in order to prevent incorrect usage that could inadvertently damage the FPGA [21]. Figure 3.9 shows an overview of the custom ICAP logic. The registers located in the system clock domain are the ICAP interface registers mapped into the static portion BAR0 registers. The ICAP interface registers in the BAR0 address space allow the driver and application software residing on the computer system to reconfigure the dynamic portion of the FPGA. They consist of a control, status, read number, write count, read count, write and read register. First In First Out (FIFO) queues are used in the FPGA to translate the address registers from the system clock domain into the 50MHz clock domain. The FIFOs prevent problems with transferring the signals across different clock domains and allow us to change the system's clock without affecting the function of the ICAP logic.

The Control register is used to send commands to the ICAP controller. The commands tell the controller what mode to put the ICAP interface in and can signal to begin reading and writing or to abort the current mode and return to an idle state. The controller reflects its current mode of operation through the status register by sending status updates through the mode FIFO. The commands are only a request to change the mode and the mode is

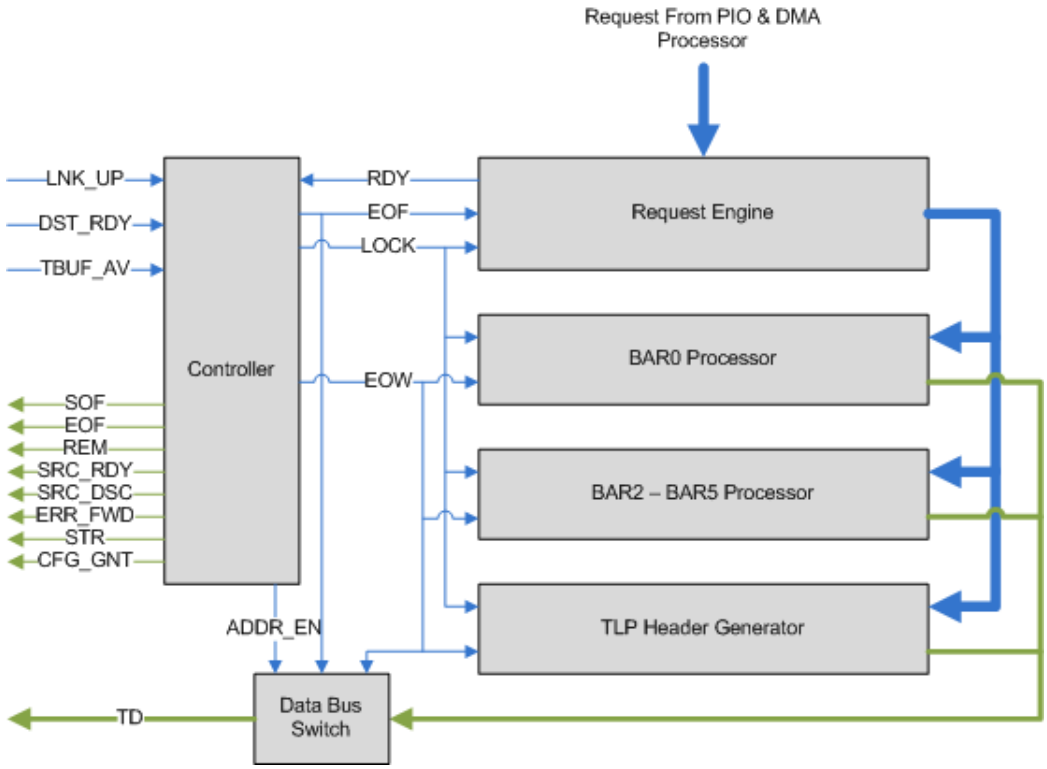

Fig. 3.8: Overview of the Outgoing Packet Controller hardware logic design.

updated according to the rules outlining the use of the ICAP interface. The mode controller and processor ensure that when a command is issued to the ICAP interface the command is either completed or aborted. The read number register updates and reflects the status of a count value register located within the mode controller that determines when a read command has completed. While in write mode, the mode controller and processor will continue to write to the ICAP interface as long as the incoming data FIFO contains data. While in read mode, the logic will continue to read data from the ICAP interface until the number of words read matches the count value register in the mode controller. This ensures that all writes and reads to the interface must be completed or an abort must be issued. The write and read count register allow the software to determine the read and write FIFO statuses when sending or receiving data during reconfiguration. The reader should refer to the FPGA Configuration User Guide for further information on the details of the ICAP interface.

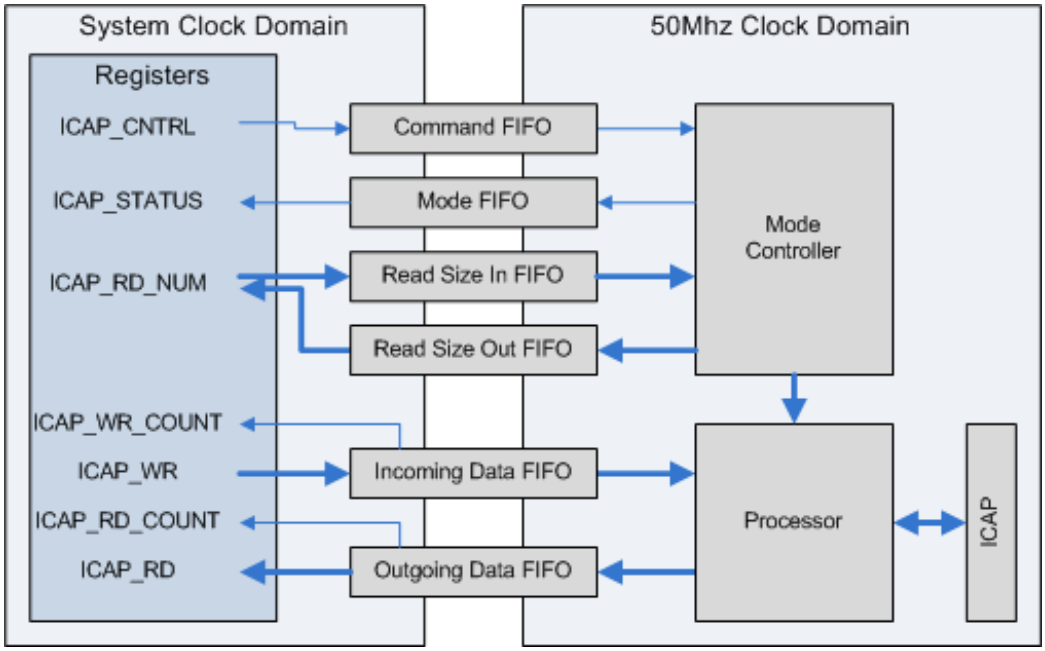

Fig. 3.9: Overview of the ICAP controller hardware logic design.

3.5 Driver Overview

The driver was designed using Windows Driver Foundation and was written exclusively for windows operating systems. The target operating system for the design is the x86 version of Windows XP. PCIE is software compatible with PCI [12]. This allowed the driver design to leverage off of PCI driver examples provided with the Windows Driver Development Kit (WDK). The driver is a kernel mode driver. A kernel mode driver was required in order to obtain access to the system mapped memory locations (BARs). A user application running in user mode on the OS can then access the device by making I/O calls into the device's instantiated driver.

The Windows Operating System allows user mode applications to transfer data to and from a kernel mode driver using two different methods. Both methods require obtaining a handle to the device first. Once the handle is obtained the first method for reading and writing to the device is performed by making Win32 API calls to read and write to a file. The device handle previously obtained is used as the file handle. This method works well for large data transfers and was not implemented in the design. It was left for

DMA implementation in future work. The second method is to make DeviceIOControl calls to the device using a Win32 provided API. The driver was designed to provide custom DeviceIOControl call handlers. These custom handlers allow restricted access to certain features of the RMCD as well as provide a mechanism for reading and writing data using PIO.

Table 3.3 contains a list of the custom device I/O control calls that are supported by the driver. In order to protect access to the registers located in the static portion of the BAR0 address space only the driver can access this area. The PIO_WRITE and PIO_READ control calls can only read and write data to or from the dynamic portion of the BAR0 address space. The driver provides individual I/O control calls for each of the features located in the static portion of the BAR0 address space. This allows each user request to be validated by checking security permissions. It also allows for strict rules of how specific data is read or written to specific locations.

Each communication protocol or interface that is developed to reside in the dynamic portion of the chip contains a unique 32-bit identifier. The identifier is used by the system software to determine what is currently loaded in the dynamic portion of the FPGA. The GET_RP_STATUS device I/O control call was added to provide that 32-bit identifier to the user. The GET_RP_STATUS call also returns additional status information about the loaded partition. For example, the call returns information on whether the loaded partition is currently enabled or not.

When performing partial reconfiguration of an FPGA it is recommended that the logic in the dynamic portion be held in a reset state for a short period of time prior to and after reconfiguration. To accommodate this recommendation, the logic in the static portion of the FPGA sends a separate reset signal to the dynamic portion. The reset signal is a combination of the reset signal in the static portion and a control register bit value. The dynamic portion logic is reset if the static portion is reset or by enabling a bit in the static portions BAR0 addressable registers. The RP_ENABLE and RP_DISABLE calls enable and disable the dynamic portion. This is used when the device is reconfigured but can also

be used by the user at any time they wish to reset their logic without resetting the entire FPGA.

The GET_CH_STATUS control call returns the status of the communication channels read and write interfaces. Currently this provides the byte counts of each interface but could be expanded for future use to include additional features.

3.6 Software Framework Overview

3.6.1 I/O Library

An important part of the design was to provide software libraries as a programming API for user applications. An I/O library was written in C# to provide access to the features provided by the driver. The library contained a device class that would abstract a lot of the Win32 API from user software and provide additional processing to simplify the data transfers. When instantiated, the device class would obtain a handle to the device and store it privately for use throughout the lifetime of the object. It would also handle splitting up data transfers into block sizes that were allowed by the DeviceIOControl calls.

3.6.2 Tester Application

In addition to creating an I/O library, a tester application was developed as part of the project. This allowed tests to be run on certain aspects of the device and provided a platform for regression and robust testing. The aspects of the design of this software are left out of this report since it is not the objective of the design, but some of the features of the software are identified in the following sections.

ICAP Configuration Register Access

The ICAP interface provides access to the internal configuration registers of the FPGA which give status information about the actual chip. The software test application provides the ability to read the different configuration registers and display the information in a user readable format for evaluation of the internal chip.

Table 3.3: Custom device I/O control calls.

Control Call Identifier	Description
PIO_WRITE	Performs a PIO write of the device's addressable registers. Only the dynamic portion registers can be accessed
PIO_READ	Performs a PIO read of the device's addressable registers. Only the dynamic portion registers can be accessed
ICAP_CTRL	Writes to the ICAP control register
ICAP_STATUS	Reads the ICAP status register
ICAP_WR	Writes data to the ICAP interface
ICAP_RD	Reads data from the ICAP interface
GET_RP_STATUS	Returns the status of the dynamic portion of the FPGA
RP_ENABLE	Enables the dynamic portion of the FPGA
RP_DISABLE	Disables the dynamic portion of the FPGA
GET_CH_STATUS	Get the status of the channels
CH0_WRITE	Writes data to channel 0 using PIO
CH0_READ	Reads data from channel 0 using PIO
CH1_WRITE	Writs data to channel 1 using PIO
CH1_READ	Reads data from channel 1 using PIO
CH2_WRITE	Writes data to channel 2 using PIO
CH2_READ	Reads data from channel 2 using PIO
CH3_WRITE	Writes data to channel 3 using PIO
CH3_READ	Reads data from channel 3 using PIO
PCIE_STATUS	Obtains internal PCIE bus status information

Addressable Memory Space

The tester application provided the ability to perform reads and writes to specific addresses in the dynamic portion of the device's addressable registers. This allows the functionality of custom communication designs to be tested by future developers without having to write any specific software.

Channel Communication

The tester has the ability to read and write to the different communication channels on separate threads. The data transmitted can be of various or random packet sizes with data generated randomly or read in from a file. It also generates metrics such as average write and read speed and total byte counts for analysis of possible bottlenecks in the system or starvation of certain communication channels.

Memory Testing

In order to test the accuracy and capability of the outgoing and incoming controllers in the PCIE transaction portion of the design, a variety of memory tests were implemented that would spawn off on a specified number of threads and perform read and write access to different portions of memory. A test design was developed to run on the dynamic portion of the FPGA that would simulate memory using block rams contained in the device. The memory tests ensured that requests to the device from numerous threads and device handles were handled appropriately when tested over a period of time.

Xilinx FPGA Bit Files

Understanding Xilinx Bit files was an important aspect in designing a controller that would perform reconfiguration of the device. Analyzing the Xilinx generated bit files in a hex editor proved to be very time consuming due to their size (approximately 9MB). To prevent countless hours of file analysis, a bit file parser was implemented as part of the tester application. The tester parses the bit files and displays a breakdown of the configuration register and memory locations of the FPGA that are read or written during configuration in a readable format. The tester does not display the data contained in the encapsulated packets in order to generate a simple overview of the communication without a large amount of data being displayed.

Chapter 4

Implementing the Design

The design described so far in this report is a framework for implementing a variety of communication protocols and interfaces in an RMCD. The project would not be complete without implementing a few communication protocols in order to prove the concept. The RMCD was designed to allow simple implementation of communication protocols, but also be robust enough to handle large complicated designs. A couple of example implementations were chosen that would test and prove the simple capabilities of the design as well as the complex capabilities.

4.1 Simple Example

For the simple example, a serial communication interface was implemented. The development board contains an RS-232 to USB bridge chip (CP2103GM) connected directly to I/O pins on the FPGA. The serial communication interface would contain a transmitter and receiver module implemented in hardware logic in the dynamic portion of the FPGA. The modules would drive two I/O pins on the FPGA, respectively. The I/O pins would connect directly to the bridge chip to allow serial communication with another PC.

4.1.1 Transmitter

The design can be broken down into two separate portions: the transmitter and the receiver. The data flow through the transmitter is separate from the data flow through the receiver in order to provide full duplex communication. Figure 4.1 shows a block diagram of the design. The transmitter contains a FIFO queue (Incoming FIFO) that is connected directly to channel 0's write interface. The FIFO stores the incoming data pushed from the channel and maintains the byte count for use by the software application driving the

interface. A transmitter module is then connected to the output port of the FIFO. The transmitter module pulls each byte from the FIFO and sends the byte one bit at a time serially out of the FPGA. The serial communication uses one start bit and one stop bit.

4.1.2 Receiver

The receiver portion of the design contains a receiver module that connects to the input pin of the FPGA. The receiver module receives the bits serially one at a time with one start bit and one stop bit. When an entire byte has been received it is then pushed to the Outgoing FIFO. The Outgoing FIFO stores the received bytes until the user application is able to read the bytes from the device. The Outgoing FIFO connects directly to the channel 0 read interface.

4.1.3 Implementation

No control registers were used for the design and the remaining channels (1-3) were also unused. After the design is loaded into the dynamic portion of the FPGA it is accessed using the DeviceIOControl requests specified in Chapter 3. The tester application also described in Chapter 3 is used to transmit the data to and from the serial interface. The design was simple and straightforward to implement. It took about two hours to implement and verify functionality.

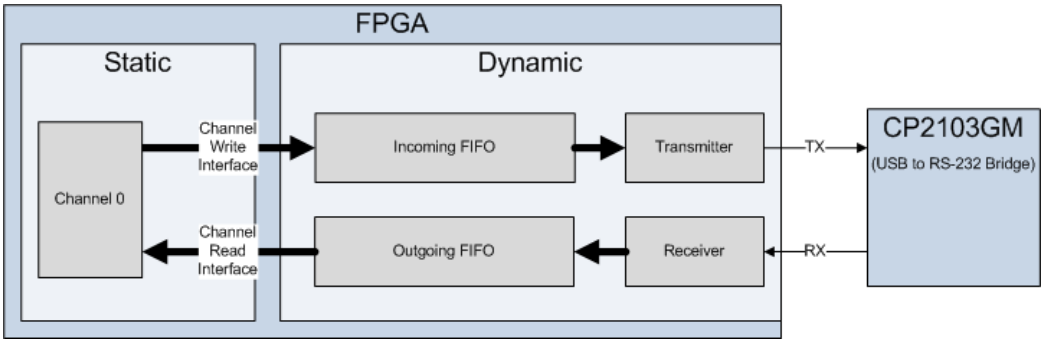

Fig. 4.1: Overview of the simple example hardware logic design.

4.2 Complex Example

The simple example represented a communication interface that is driven by incoming and outgoing FIFOs. The complex example should represent a design that has a lot more processing to do on the dynamic side and may actively drive communication without having to be driven by the user software first. Developers would need to debug their communication designs so implementing something that required a debugger would be beneficial. The development board contains a Tri-mode Ethernet interface and Xilinx provides an example embedded web server design [22, 23]. Xilinx's example web server design was modified to run in the dynamic portion of the FPGA. The design uses a MicroBlaze processor to run the web server application. This would demonstrate a complicated self-contained design. It would also demonstrate the usage of a software debugger to debug the software web server application after it is loaded into the dynamic portion of the FPGA. This proved to be a much more difficult task and pointed out some drawbacks with the system.

4.2.1 Required Modifications to the Static Portion

Two current limitations of Partial Dynamic Reconfiguration on Xilinx FPGAs are clock generation modules and global buffers cannot be reconfigured, and bidirectional signals in the FPGA cannot cross reconfigurable partition boundaries. The first limitation was not initially seen as a problem since the clock generation modules can be programmed to provide various clock frequencies to drive various requirements within the dynamic portion of the FPGA. This is not a problem with custom logic, but when trying to implement 3rd-party IP cores the cores may have clock generation modules or buffers instantiated within their design. This would require significant modification to the IP core in order to use external clocks instead of internally generated ones. The web server example provided by Xilinx used an IP core to interface with the Double Data Rate 3 (DDR3) memory modules on the board. The DDR3 IP core contained bidirectional I/O to connect to the DDR3 memory module and instantiated clock generation primitives within the core.

To get around this problem, the only possible solution was to move the DDR3 IP core to the static portion and provide access to it from the dynamic portion with a standard bus

interface. This is not unreasonable since a production RMCD containing DDR3 memory would not interface it to reconfigurable I/O pins anyway so it could remain in the static portion without affecting the dynamic designs. The only limitation would be if a design required a different IP core for the memory or a different standard bus to interface with it. The bus interface used to interface to the DDR3 IP core is the Advanced eXtensible Interface (AXI) bus. Xilinx has recently adopted the AXI interface for all its future IP cores and have transitioned many of the old cores over to AXI. Using AXI may limit certain non-AXI implementations from using the DDR3 memory, but would provide a bus that is most likely to be compatible with future Xilinx IP Cores. The debug module and Ethernet IP Cores also had to be moved into the static portion. The debug core needed to interface with the Joint Test Action Group (JTAG) [24] ports and required certain clock buffers. The Ethernet IP Core had similar limitations. The Ethernet IP Core uses AXI and a Xilinx defined slave version of the bus called AXI4-Lite. The debug module uses AXI4-Lite as well. An overview of the RMCD design can be seen in Fig. 4.2 that includes the added static IP cores.

4.2.2 Design of the Dynamic Portion

The final design of the dynamic portion can be seen in Fig. 4.3. The software running on the MicroBlaze processor can also be used to access the additional peripherals on the development board such as the Light-Emitting Diodes (LEDs), push buttons, Dual In-line Package (DIP) switches, Electrically Erasable Programmable Read-Only Memory (EEPROM), Compact Flash, and UART. These interfaces and IP cores had no difficulty running in the reconfigurable portion of the FPGA. Xilinx Platform Studio was used to synthesize the MicroBlaze project provided in the Xilinx example design. Synthesizing the design in Xilinx Platform Studio created netlist files for all the individual modules. It also created a VHDL system file that combines all the modules into the example system. The generated system VHDL file was then modified to contain outside interfaces to the modules that would reside in the static portion (DDR3, Ethernet MAC, debug module, and clock generator) instead of instantiating these modules from within. The I/O pins used in the design also

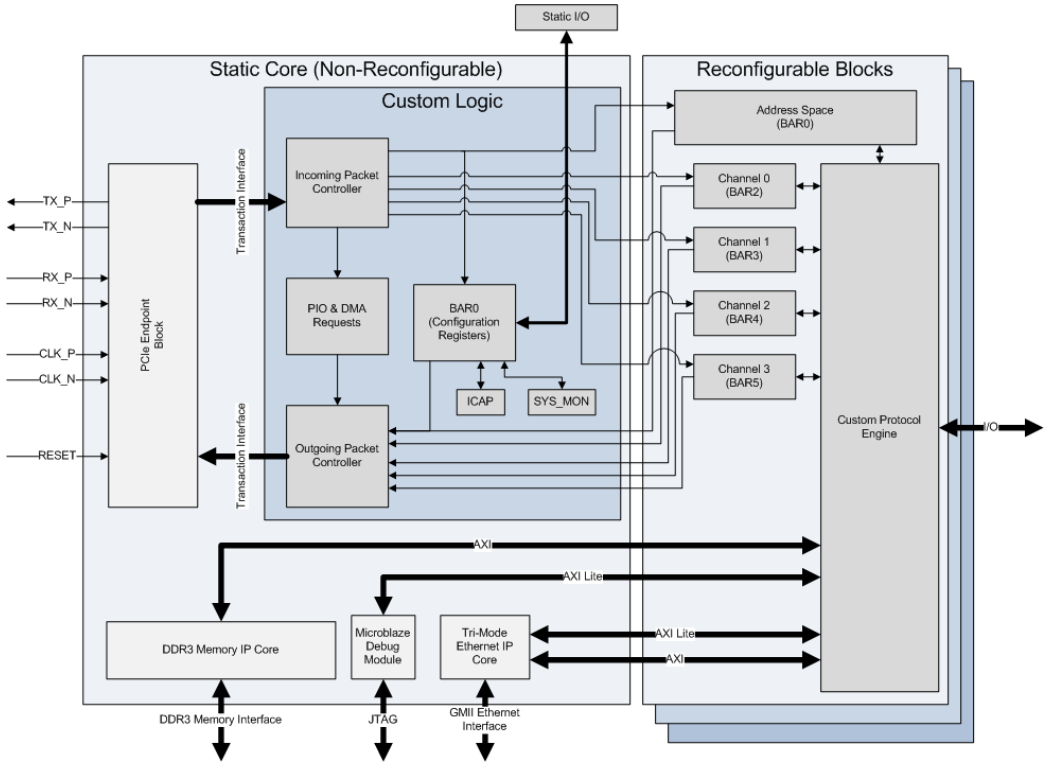

Fig. 4.2: Overview of the RMCD hardware logic design with additional static IP cores added.

had to be separated out to ensure that the buffers driving the I/O pins were instantiated in the same partition as the corresponding pins. Some additional modifications were required in order to get the system VHDL file to synthesize using the partial reconfiguration design flow as opposed to the Xilinx Platform Studio. The system VHDL file could then be included in a higher level VHDL file which represented the reconfigurable design.

4.2.3 Functionality of the Design

One of the problems with implementing a MicroBlaze core in the design is that the core contains a hardware and a software component. The hardware is the functioning processor and the software is the logic that runs on the processor. There is no purpose in loading a pure hardware MicroBlaze design into the reconfigurable partition if there is not a way to load software as well. To handle this problem a boot loader needed to be included in the design so that the processor could start up after reset with some basic functionality. The boot loader is similar to the BIOS on a PC. A boot loader for MicroBlaze processors is provided by the Xilinx Platform Studio. The boot loader needs to reside in the processor when it boots up so the boot loader code must be programmed into the FPGA logic at the same time the hardware logic is programmed. Xilinx provides the ability to load startup values into block rams contained in the FPGA logic. These values are incorporated into the bit stream file that is used to program the FPGA. Using a command line utility called Data2Mem, the boot loader machine code could be included in the generated bitstreams. When the bitstreams are loaded into the FPGA the hardware logic is configured and the block rams are programmed with the boot loader code. The block rams represent the MicroBlaze reset vectors and instruction cache blocks. The block rams are represented by the Internal BRAM module in Fig. 4.3. When the processor is started, the boot loader code is run and the processor is then ready to be accessed using the JTAG debugger.

4.2.4 Running the Web Server

Xilinx provides a JTAG debugger that can be used to connect into a MicroBlaze processor once the hardware and boot loader have been configured. The JTAG debugger can

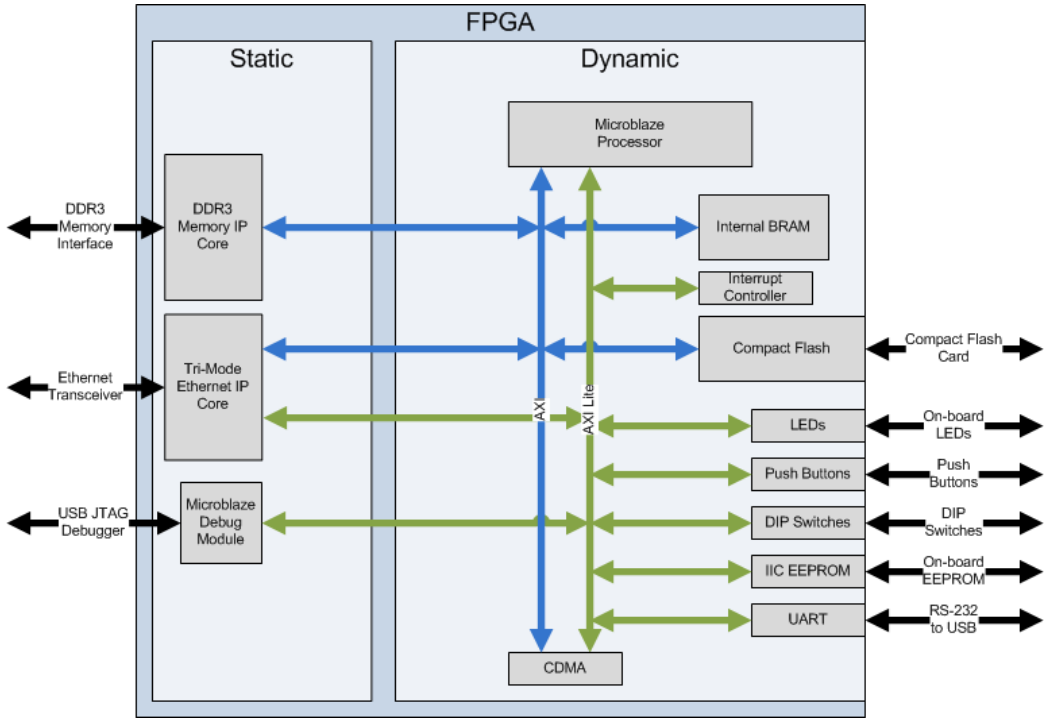

Fig. 4.3: Overview of the complex example hardware logic design.

be accessed using a command line interface or accessed using the Xilinx Software Development Kit Integrated Development Environment. Using the JTAG debugger, a compiled executable can be sent to the MicroBlaze core to run on the processor. The debugger transfers the executable machine code to the MicroBlaze system using the JTAG interface. It writes the code into the addressable memory region of the processor that resides in DDR3 memory. After the executable has been transferred the debugger resets the processor and the processor begins to execute the loaded executable. The web server demo executable provided by Xilinx contains a TCP/IP protocol stack for the HTTP communication. When an HTTP request comes in, the processor generates the HTML files and then serves them out to the requester.

4.2.5 Implementation

The design took about a month to implement. The length of time was partially due to the modifications that were required in the static portion and partially because the original

VHDL code generated by Xilinx Platform Studio had to be modified in order to split the design across the static and dynamic portions of the FPGA. The actual communication protocol implemented for the design (web server) is done in software so additional communication protocols could be implemented without requiring modification to the hardware design of the dynamic portion. This would reduce the amount of development time in the future to only that which is required to develop the software. The development would be independent of the RMCD hardware platform.

Chapter 5

Analysis of Results and Comparison with Similar Designs

Both the complicated and simple example designs function properly and reconfiguration of the dynamic portion of the FPGA functions as it should. A developer was able to load the complicated example and connect to the MicroBlaze core using the JTAG debugger. The developer was then able to load and run the web server application over the JTAG interface. A separate computer was setup to receive serial communication from the simple example when loaded. The separate computer was also connected to the development board using a CAT6 cable in order to access the web server when the more complicated example is run. Reloading the simple and complex examples allowed the separate computer to communicate with the development board according to the currently loaded protocol. The following sections outline the advantages and disadvantages of the completed system.

5.1 Advantages

The design reduces the cost of an RMCD by eliminating the need of a separate bus controller in the design. The FPGA is connected directly to the computer system so the bottlenecks explained in Chapter 2 no longer apply. The static portion of the design processes data in real time from the PCIE bus removing any possible bottlenecks from the bus controller logic.

The design provides simple addressable memory and packet transfer interfaces that should require minimal development time for simple designs and provide the necessary capabilities for more complex designs. The communication channel interface from the static to the dynamic portion can operate independently allowing low level coordination between standard and custom interfaces.

The transparent nature and small footprint of the Windows driver and static portion

of the FPGA allow for additional Windows drivers to be developed that sit on top of the current driver. For example, a network driver could be written that interfaces with this driver to perform the data transfers. A communication protocol loaded into the dynamic portion of the FPGA could then act as if it was a network card plugged into the host operating system. Standardized software and APIs could then be used to access the device similar to other commercial hardware.

Embedded system designs can be implemented and debugged in the dynamic portion to cover a wide range of possibilities. MicroBlaze cores can currently run a variety of operating systems including various implementations of Linux and μC/OS-II [25]. Linux can also be used to run a variety of communication protocols and applications including web servers, network file systems, file transfer protocols, telnet, etc.

5.2 Disadvantages

Most of the major drawbacks to the system are a result of using partial reconfiguration in the FPGA. IP cores that use FPGA primitives that cannot be instantiated in reconfigurable logic would require significant modification and may not work at all.

The partial reconfiguration development flow provided by Xilinx targets applications where the reconfigurable logic is developed at the same time the static logic is developed [26]. A design may contain a reconfigurable portion, but the possible reconfigurable designs meant to reside in that portion are known at development time and are developed with the static portion. The reason for this is that when the initial static portion is mapped, placed, and routed it relies on the currently selected dynamic portion to determine how the static portion is routed. Xilinx recommends that the most complicated dynamic logic is used to initially implement the design so that the timing can be assured for the remaining less complicated designs. A limitation of partial reconfiguration for an RMCD is that a future complicated design may struggle to meet timing constraints due to how the static design was previously routed.

Another problem with assuming that all dynamic reconfigurations are known at development time is that the partial reconfiguration design flow requires the FPGA resources

used by the dynamic portion be explicitly set. The remaining logic of the FPGA would then belong to the static portion. In an RMCD the static portion would be known and the dynamic portion unknown. It would suite the design better if the static portion resources could be explicitly set and the remaining resources left to the dynamic portion. This may initially seem irrelevant because it seems like if one side is explicitly assigned then the remaining resources would go to the other side. Numerous problems were run into while getting the reconfigurable design to route. Space must be left between static and dynamic partitions in order to allow for proxy logic to be inserted. Large portions of the FPGA were left open in order to get the device to route properly. The slack space not consumed by the proxy logic was then left to the static portion which was unusable for future dynamic designs. The initial expectation of the design was that partitions designating what is static and what is dynamic could be strictly defined. Due to limitations in the placement and routing of the designs with the Xilinx software, strict definitions were not possible.

5.3 Performance

An analysis of the design would not be complete without evaluating the performance of the design. A platform that could be used to implement high speed or high bandwidth protocols in the future would need to be capable of handling such high performance. It was difficult to compare this design with other designs on the market because, as stated in previous chapters, DMA was not implemented in the design due to time constraints. PIO severely limits performance and it would not be realistic to compare a PIO design to a DMA one. This section gives a baseline for the design's performance using PIO in order to remove any doubts of possible limitations or bottlenecks in the system due to the design itself.

The performance tests were performed by loading in a reconfigurable design that interfaces the BAR0 configuration registers to block rams. The block rams were combined to form a 64-bit read/write interface with 64K memory locations giving an address space of 524,288 bytes. A benchmark algorithm was then added to the tester application to write 524,288 bytes to the configuration registers and track the amount of time the operation

took to complete. A second benchmark was added to read the 524,288 bytes. A series of 100 benchmarks was run and the results tallied to try and minimize deviations due to external affects (other software running, CPU interrupts, etc.). Address-based transfers were used for the benchmark instead of packet-based because the PIO performance would be the same for each and targeting the configuration registers would provide a larger address space in which to average the transfers. Table 5.1 contains the results of the benchmarked writes when run on two separate computer systems. Table 5.2 contains the results of the benchmarked reads.

The first system had an average write duration of 152 milliseconds and an average read duration of 1,618. Figure 5.1, Fig. 5.2, Fig. 5.3, and Fig. 5.4 show captures of the request transactions on the bus interface between the static logic and Xilinx's PCI Express IP core. The captures were made using Xilinx's ChipScope Pro analyzer. The waveforms were captured during the benchmarks of laptop 1. Figure 5.1 shows the latency of the RMCD logic when a read request is made to the configuration registers located in the reconfigurable partition's BAR0 address space. The waveform shows a latency of seven clock cycles between the last byte of the received frame and the first byte of the response frame. This shows the efficiency of the design to handle the request within a small amount of clock cycles. If the packet duration is taken into account and the host computer system was assumed to not have any delays between a response from the device and the next request to the device it would mean that each byte could be sent and received within eight clock cycles. The clock is running at 62.5 Mhz which means the duration to send one byte would be 128ns. Sending 524,288 bytes would then take approximately 67ms. This would provide a theoretical PIO read transfer speed of 7,812,500 bytes per second. This is much larger than the measured speed (323,947).

Figure 5.2 shows the latency of the computer system when making the next read request to the device after a previous response. The system takes 195 clock cycles between the last byte of the previous response and the first byte of the next request. This is a significantly larger amount of time when compared to the turnaround time of the device

Table 5.1: Measured write durations for 100 runs of 524,288 bytes each (values are in milliseconds unless specified).

System	Minimum	Maximum	Standard Deviation	Average	Throughput (B/s)
Laptop 1	140.625	156.25	7.051	151.875	3,452,102.058
Laptop 2	203.125	265.625	9.447	212.344	2,469,053.127

Table 5.2: Measured read durations for 100 runs of 524,288 bytes each (values are in milliseconds unless specified).

System	Minimum	Maximum	Standard Deviation	Average	Throughput (B/s)
Laptop 1	1,609.375	1,687.500	11.578	1,618.438	323,947.017
Laptop 2	2,281.250	2,359.375	11.650	2,291.250	228,821.822

(seven clock cycles). Using 196 clock cycles (including the packet width) and eight clock cycles for the turnaround time means the device processes a byte every 3.2 microseconds which would transfer the 524,288 block in 1,677 milliseconds which is approximately the results obtained through the benchmark. Because of the low turnaround time of this design, the PIO performance is bottlenecked by limitations in the computer system's ability to handle PIO. The differences between the PIO performance of laptop 1 and laptop 2 is around 30% which shows that PIO performance also varies quite a bit between different computer systems.

One interesting point brought out by the bus captures is the difference between PIO reads and writes. The latency between each read stays pretty routine with only minor fluctuations in the idle latency. Figure 5.3 and Fig. 5.4 show captures of the PIO write benchmark. The latency between each write request switches back and forth between a long latency and a short one. The long and short latencies remain fairly constant throughout the benchmark. The short latencies are around five clock cycles and the long latencies are around 27 clock cycles. PIO writes will naturally be faster than reads since the RMCD is designed to handle the requests in real-time so the bus packets could be transferred back-to-back. This would take two clock cycles per packet so theoretically the host system could write a byte every 32ns. A block of 524,288 bytes would take 16ms. This is much faster than

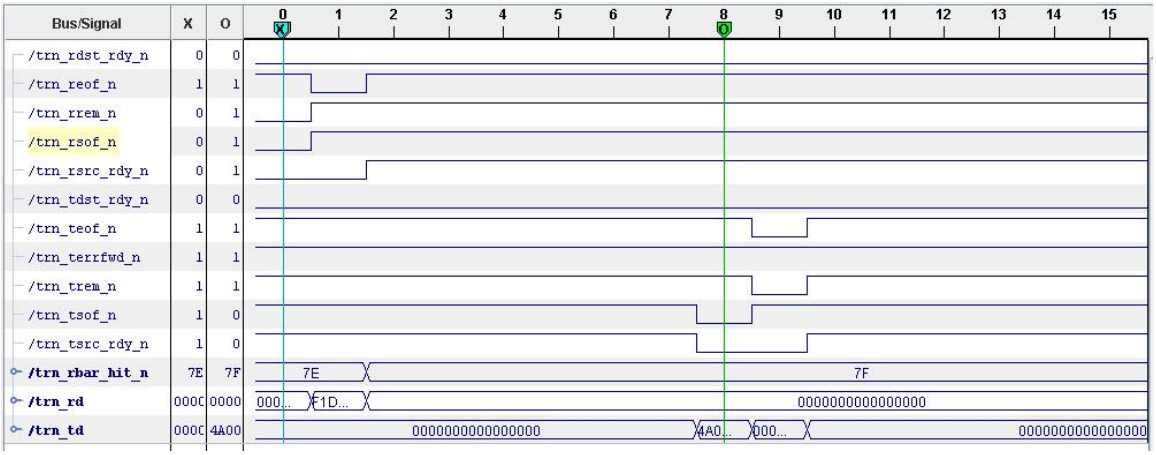

Fig. 5.1: Latency of the RMCD logic to respond to a read request.

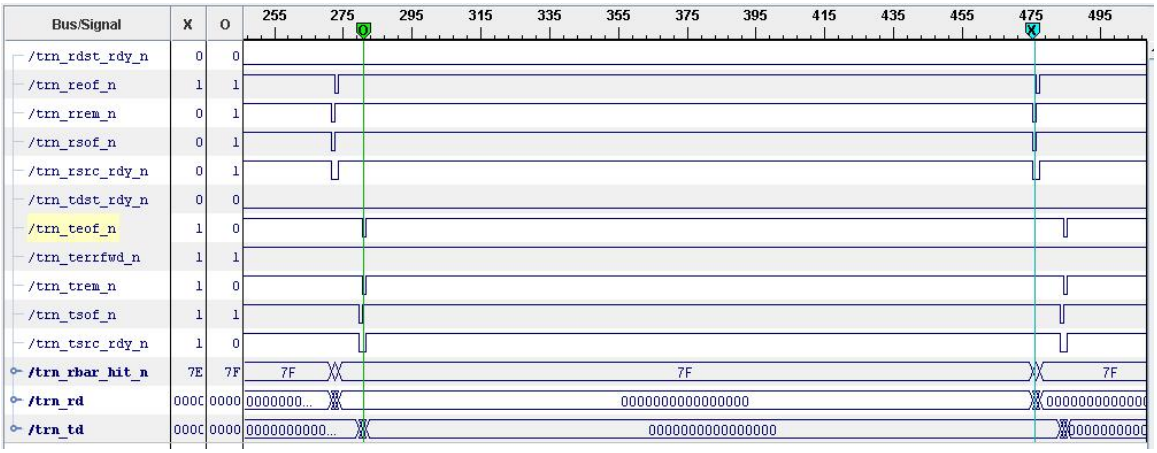

Fig. 5.2: Idle time of the RMCD logic while waiting for next read request.

the benchmarks measured and is once again due to latencies between requests sent from the host computer system. The reason for the alternating idle times is currently unknown and may have something to do with how the packets are cached along the data path.

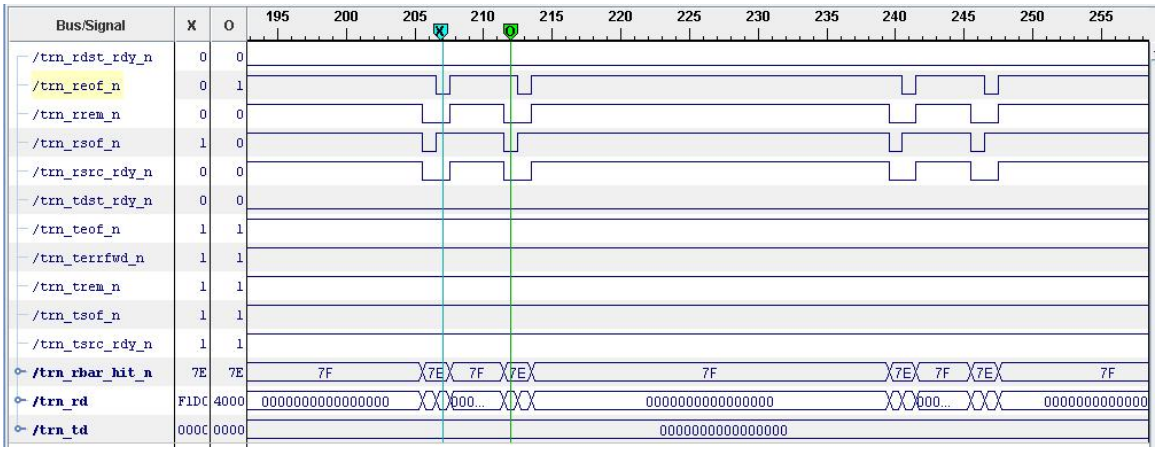

Fig. 5.3: Short idle time of the RMCD logic while waiting for the next write request.

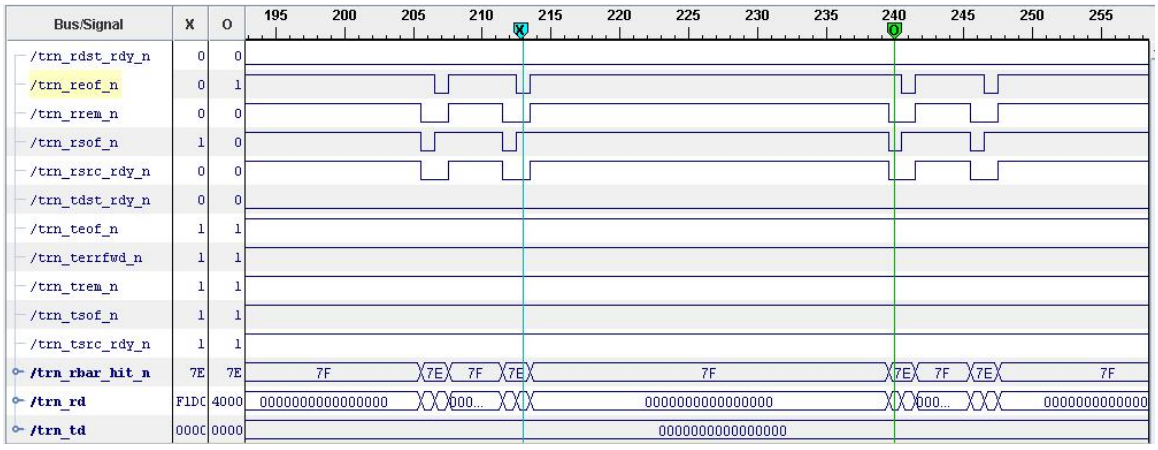

Fig. 5.4: Long idle time of the RMCD logic while waiting for the next write request.

Chapter 6

Conclusion

Using partial reconfiguration proved to be a viable option for next generation RMCD designs; however, it does have limitations. Partial reconfiguration eliminates the need for a separate bus controller in the system, but also provides some restrictions to what can be implemented in the dynamic portion of the FPGA. Adding a separate bus controller to the system increases the cost and may introduce a bottleneck into the system, but it would also allow the entire FPGA to be used for the communication protocol. A bus controller would remove the need for partial reconfiguration and reduce the complexity of routing high performance tightly constrained IP cores.

The best solution depends on the target communication applications of the RMCD. If low production costs are a driving factor in the design and most of the dynamic logic would contain custom protocols then partial reconfiguration may be the best option. If high performance and versatility is the driving factor in the design and high speed tightly constrained IP cores will be implemented than a different design with a bus controller may be a more suitable platform.

References

[1] International Organization for Standardization, "ISO 15765-4:2005," [http://www.iso.org], 2005.

[2] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," in *Proceedings of the IEEE*, pp. 1204–1223, June 2005.

[3] T. Nolte, H. Hansson, and L. Lo Bello, "Automotive communications - past, current and future," in *Proceedings of the 10th IEEE Conference on Emerging Technologies and Factory Automation*, pp. 985–992, Sept. 2005.

[4] J. Zhang and A. Pervez, "Avionics data buses: An overview," *IEEE Aerospace and Electronic Systems Magazine*, vol. 18, pp. 18–22, 2003.

[5] J. K. Murdock and J. R. Koenig, "Open systems avionic network to replace MIL-STD-1553," in *Proceedings of the 19th Digital Avionics Systems Conference*, pp. 4E5/1–4E5/6, Oct. 2000.

[6] M. W. Beranek, T. P. Curran, A. S. Glista Jr. and M. J. Hackert, "Avionics fiber-optic and photonics network preliminary technology readiness assessment," in *Proceedings of the 23rd Digital Avionics Systems Conference*, pp. 9.B.3–9.1–8, Oct. 2004.

[7] K. F. Roosendaal and D. W. Christenson, "Embedded computer software loader/verifier implementation using a hardware and software architecture based upon best practices derived from multiple spiral developments and the joint technical architecture," in *Proceedings of the 2003 IEEE Systems Readiness Technology Conference*, pp. 496–501, Sept. 2003.

[8] Telecommunications Industry Association, "EIA/TIA-422-B," [http://www.tiaonline.org/standards/], 2005.

[9] As-1a Avionic Networks Committee, "AS15532 - Data Word and Message Formats," [http://www.sae.org/technical/standards/AS15532], 1999.

[10] D. W. Christenson, "Developing a stable architecture for interfacing aircraft to commercial personal computers," in *Proceedings of the 2000 IEEE AUTOTESTCON*, pp. 559–563, Sept. 2000.

[11] Peripheral Component Interconnect Special Interest Group, "PCI Local Bus Specification Revision 3.0," [http://www.pcisig.com/], 2004.

[12] Peripheral Component Interconnect Special Interest Group, "PCI Express Base Specification Revision 2.1," [http://www.pcisig.com/], 2009.

[13] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips, "Universal serial bus specification revision 2.0," [http://usb.org/], 2000.

[14] Hewlett-Packard, Intel, Microsoft, NEC, ST-NXP Wireless, and Texas Instruments, "Universal serial bus 3.0 specification revision 1.0," [http://usb.org/], 2008.

[15] Telecommunications Industry Association, "EIA/TIA-232-E," [http://www.tiaonline.org/standards/], 1991.

[16] The Institute of Electrical and Electronics Engineers, "1394a-2000 IEEE Standard for a High Performance Serial Bus (Amendment)," [http://standards.ieee.org/], 2000.

[17] The Institute of Electrical and Electronics Engineers, "Standard 802.3," [http://standards.ieee.org/], 2002.

[18] A. B. McCarthy and F. Peng, "Comparing GPIB, LAN/LXI, PCI/PXI measurement performance in hybrid systems," in *Proceedings of the 2006 IEEE Autotestcon*, pp. 122–128, Sept. 2006.

[19] Personal Computer Memory Card International Association, "ExpressCard standard release 2.0," [http://www.usb.org/developers/expresscard/], 2009.

[20] Xilinx Inc., "Virtex-6 FPGA integrated block for PCI Express user guide," [http://www.xilinx.com/support/documentation/], 2010.

[21] Xilinx Inc., "Virtex-6 FPGA configuration user guide," [http://www.xilinx.com/support/documentation/], 2010.

[22] Xilinx Inc., "AXI interface based ML605/SP605 MicroBlaze processor subsystem hardware tutorial," [http://www.xilinx.com/support/documentation/], 2010.

[23] Xilinx Inc., "AXI interface based ML605/SP605 MicroBlaze processor subsystem software tutorial," [http://www.xilinx.com/support/documentation/], 2010.

[24] The Institute of Electrical and Electronics Engineers, "1149.1-2001 (Reaff 2008) IEEE Standard Test Access Port and Boundary-Scan Architecture," [http://standards.ieee.org/], 2001.

[25] Xilinx Inc., "Third party embedded solutions providers," [http://www.xilinx.com/ise/embedded/epartners/listing.htm#RTOS].

[26] Xilinx Inc., "Partial reconfiguration user guide," [http://www.xilinx.com/support/documentation/], 2010.