NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Enabling Communications Between an FPGA's Embedded Processor and its Reconfigurable Resources

Author: Joshua Noseworthy

Department: Electrical and Computer Engineering

Approved for Thesis Requirement of the Master of Science Degree

_____ _____
Thesis Advisor: Prof. Miriam Leeser Date

_____ _____
Thesis Reader: Prof. Stefano Basagni Date

_____ _____
Thesis Reader: Sarah Leeper Date

_____ _____
Thesis Reader: Prof. Xinping Zhu Date

_____ _____
Department Chair: Prof. Stephen McKnight Date

Graduate School Notified of Acceptance:

_____ _____
Dean: Prof. Yaman Yener Date

Copy Deposited in Library:

_____ _____
Reference Librarian Date

Enabling Communications Between an FPGA's Embedded Processor and its Reconfigurable Resources

A Thesis Presented

by

Joshua Noseworthy

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical Engineering

in the field of

Electrical Engineering

Northeastern University

Boston, Massachusetts

August 2005

Acknowledgements

I would like to extend my deepest gratitude and appreciation to my advisor Professor Miriam Leeser. Her guidance and instruction has played an invaluable part in both my undergraduate and graduate studies.

It has been a pleasure to work with my colleagues in the Rapid Prototyping Lab at Northeastern University. They have provided a friendly, encouraging, and supportive environment for me to work in.

I would also like to extend my appreciation to Mercury Computer System for funding my research. I would like to specifically recognize Graham Bardouleau and Sarah Leeper for their willingness to answer any questions that I may have had.

Finally I would like to recognize the best family anyone could ever ask for, especially my parents Ed and Karen, and girlfriend Jennifer. I could not have done this without their unconditional love, support, and understanding.

Abstract

Increasing device densities allow designers to integrate more functionality onto a single piece of silicon. Many chip manufactures are using this flexibility to offer complete solutions that can be integrated onto a single device. FPGA manufacturers, such as Xilinx and Altera, have introduced FPGA architectures that contain a variety of embedded processing elements along with the device's reconfigurable logic. One of the more recent processing elements that has been introduced by Xilinx is the PowerPC405 hard-core processor.

One of the challenging aspects of developing applications that target the PowerPC is the interfacing of the processor with the surrounding reconfigurable logic. We have implemented several versions of a FM3TR Waveform Application to exercise the various interfaces that enable communication between the processor and the surrounding FPGA fabric. These interfaces can be either shared or dedicated. Shared interfaces enable communication between the processor and multiple peripherals. Dedicated interfaces provide dedicated communication links between the processor and a single peripheral. Dedicated interfaces are less flexible, but can deliver higher performance than shared interfaces. Our results indicate that the performance of the FM3TR Waveform Application can be increased by as much as 60% just by choosing the interfaces that is most appropriate for the implementation. This demonstrates that the performance of FPGA applications that use the embedded processor are dramatically effected by the

mechanisms that are chosen to enable communication between the processor and its surrounding resources.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advancements in silicon technologies continually increase the number of transistors that can be integrated into a single device. Many designers have begun using this new integration potential to fabricate complete systems on a single silicon fabric. This new design practice, referred to as System-on-Chip (SoC), aims to integrate multiple board level components into a single silicon die.

As SoC architectures continue to receive attention from the embedded systems community, FPGA manufacturers such as Xilinx are responding with a new generation of FPGA architectures that contain a variety of embedded resources. One of several recentadditions to Xilinx's Virtex family architecture is the embedded PowerPC405 core. The motivation for the integration of the processor core onto the fabric of the FPGA is the idea that most FPGAs contained within an embedded system require some level of interaction with an external processor. Moving this processor into the fabric of the FPGA eliminates bottlenecks associated with communicating through off-chip interfaces.

Integrating a general-purpose processor into the fabric of the FPGA eliminates the need for off-chip interfaces, but creates a need for on-chip interfaces that provide efficient communication between the processor and the reconfigurable resources. The existence of such an interface is critical to the successful integration of the processor core into an FPGA implementation.

In this thesis, we present a study that investigates various mechanisms that interface the Virtex II Pro's PowerPC405 with the surrounding FPGA fabric. These mechanisms utilize the On-Chip Memory (OCM) and the Processor Local Bus (PLB) interfaces of the PowerPC405. Both of these interfaces enable communication between the PowerPC405 and its surrounding reconfigurable resources. However, the mechanisms used by each interface are very different. For instance, the OCM interface provides a dedicated interface to the surrounding FPGA fabric, while the PLB provides a shared interface. A dedicated interface provides a higher performance relative to the shared interface, however, the share interface offers greater flexibility in interfacing peripherals to the processor. These are the types of tradeoffs that will be the focus of this investigation.

The results of the investigation provide an accurate characterization of the advantages and disadvantages of the interfaces that provide the communication fabric between the PowerPC405 and the surrounding architecture on a Xilinx Virtex-II Pro FPGA. To serve as a basis for comparison we present multiple implementations of an FM3TR waveform application, each using a different interface.

The outline of the remainder of this thesis is as follows.

Chapter 2 begins with an introduction to FPGAs and how they compare to other available technologies. It then gives an introduction to SoC, including the various mechanisms used to interface entities that exist within SoC architectures.

It then describes the extension of SoC into platform FPGAs, specifically, the Virtex-II Pro FPGA. The reader is then introduced to the concept of Software Defined Radios (SDRs) and how using SoC architectures can facilitate their development. Finally, a survey of related work on SoC and the Virtex-II Pro FPGA is presented.

Chapter 3 begins with a detailed discussion of the FM3TR waveform application. We then discuss the platform that was used to implement multiple FM3TR waveform application implementations. Finally, we present the design and performance expectations of each implementation.

Chapter 4 presents the performance results of the interfaces that were discussed in Chapter 3. These performance results are presented in terms of speed, complexity and resource utilization. The results are then analyzed to determine circumstances that warrant the use of a particular interface.

Chapter 5 gives conclusions and suggestions for future work.

Chapter 2

Background

This chapter establishes fundamental concepts necessary to understand the focus of this research. The topics that will be discussed include system on chip, FPGA technologies, general-purpose processors, on chip communication architectures, and software defined radios.

2.1 Modern Day Processing Elements

Modern day engineers have several devices to choose from as the implementation fabric for their application. These devices can be classified as either general purpose, application specific hardware, or reconfigurable hardware.

General-purpose hardware is a term used to describe devices that are capable of understanding instructions that are issued by a programmer. The hardware contained within the device is designed to provide moderate performance in a wide range of applications. A programmer can issue a command to tell the device to perform any one of its pre-determined instructions at any given time. This type of hardware comes in many different flavors, some of which are more heavily optimized for specific application domains. For instance, a general-purpose processor (GPP) is a microprocessor that has been optimized to offer moderate performance in a wide range of application domains. A

digital signal processor (DSP) is a microprocessor that has been optimized to offer better than moderate performance for a narrow range of digital signal processing applications.

General-purpose hardware is suitable for a variety of applications. As a result of this flexibility, general-purpose hardware may fail to provide an implementation platform that is capable of meeting the system requirements for higher performance applications. In instances that require the highest performance, designers use application specific hardware. Application specific hardware usually takes the form of an application specific integrated circuit (ASIC). ASICs are optimized for high performance with respect to a specific application. Unlike general-purpose hardware, ASICs can only perform the specific function they were designed to perform.

Reconfigurable hardware attempts to couple the performance of ASICs with the flexibility of general-purpose hardware. The most common type of reconfigurable hardware uses an array of field programmable gates. These gates can be configured to perform specific boolean operations. The gates are interconnected through the device's reprogrammable interconnect fabric.

2.1.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be customized by the end user for a specific application. Figure 1 presents a general architecture overview of Xilinx's Virtex-II Pro Platform FPGA. The fundamental

building block of an FPGA is the configurable logic block (CLB). CLBs are blocks of logic whose functionality can be changed by reconfiguring the contents of the block itself. A single CLB contains 4 slices and 2 tri-state buffers. Each slice is identical to the others slices that are contained within the CLB. A single slice, seen in Figure 2, provides two function generators, two storage elements, arithmetic logic gates, multiplexers, and fast carry logic. The function generators may be configured as 4-input look-up tables (LUTs), as 16-bit shift registers, or as 16-bit distributed SelectRAM+ memory. In addition, either storage element may be configured as an edge-triggered D-type flip-flop or a level sensitive latch. Each CLB has its own internal interconnect, as well as the ability to access general routing resources.

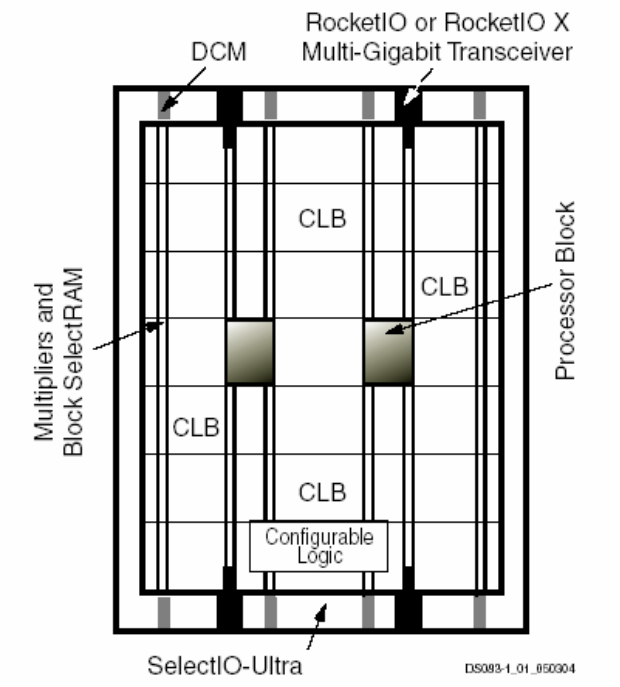

Figure 1: Virtex-II Pro Architecture Overview [20]

Although Configurable Logic Blocks remain the fundamental building block of an FPGA, increasing device densities has allowed manufacturers to integrate additional heterogeneous resources into their FPGA architectures. Modern FPGA devices, such as the Virtex-II Pro, contain other reconfigurable elements such as BlockRAMs, multipliers, and general-purpose processors. Each BlockRAM provides an 18Kb dual-ported memory structure with two independently clocked and independently controlled synchronous ports that access a common storage area. Each multiplier element provides an 18-bit by 18-bit signed multiplier. They are optimized for high speed operations and

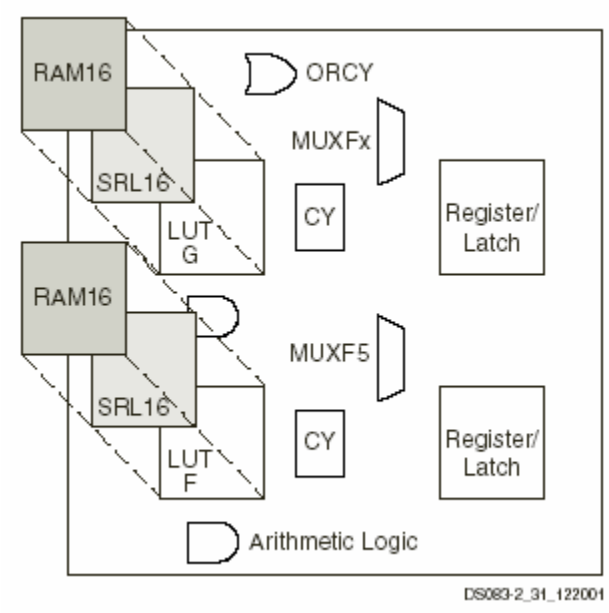

Figure 2: Virtex-II Pro Slice Configuration [17]

have low power consumption compared to an equivalent multiplier implementation using CLBs. Finally hard processor macros, such as the PowerPC405, provide a multi-stage instruction pipeline capable of executing stored instructions.

These heterogeneous elements are interconnected using the FPGAs general routing matrix. The general routing matrix is made up of a network of interconnected routing switches. Programming elements that wish to communicate connect to a routing switch. Paths are then established between these routing switches. The connection of programmable elements to routing switches, as well as the establishment of routes between them, is the responsibility of the CAD tools.

All reconfigurable elements contained within the FPGA are controlled by values that have been stored in static memory cells. These values are loaded into the memory cells at the time of configuration. To reconfigure elements within the FPGA, new values must be loaded into the appropriate memory cells [17].

2.2 System-On-Chip

Advances in the semi-conductor industry continue to increase circuit density of silicon devices. This increasing potential has prompted many designers to consider the integration of multiple board level components onto a single silicon device, a concept that has been termed System-On-Chip. This type of component integration has the potential to offer increased reliability, increased performance, lower resource utilization, and lower

cost. However, such a high level of transistor density makes successful design and verification of these systems difficult.

To facilitate the design of SoC systems, many designers are steering clear of full custom design approaches. Instead, designers are choosing to build their systems using existing components that have well-defined contents and interfaces. This re-use of existing components lowers development costs and time to market.

One of the major challenges associated with SoC design methods is providing efficient communication between the components. Developing efficient communication architectures for SoC systems is a challenging task.

2.2.1 Communication Architectures for SOC

In order to effectively use the hardware components contained within a SoC, an efficient SOC communications architecture is critical. SoC communication architectures provide a medium that on-chip components can use to communicate with each other. Although this communication fabric is critical to the operation of the device, its existence does not provide any additional computational functionality.

2.2.2 Shared Interfaces Versus Dedicated Interfaces

SoC communication architectures can consist of shared interfaces, dedicated interfaces, or a mixture of both. A shared interface is an interface that is shared between multiple entities. An example is the bus topology, seen in Figure 7, which uses a data bus to serve as the vehicle that two entities use to communicate. One entity acts as the bus master, and the other as a slave. The bus master is the controlling entity and is the only device capable of presenting commands. The slave responds to commands presented to it, either by accepting data from, or presenting data to the master. Multiple entities that assume master or slave roles can connect to a single bus. However, communication may only occur between a single master and a single slave at any given instance in time.

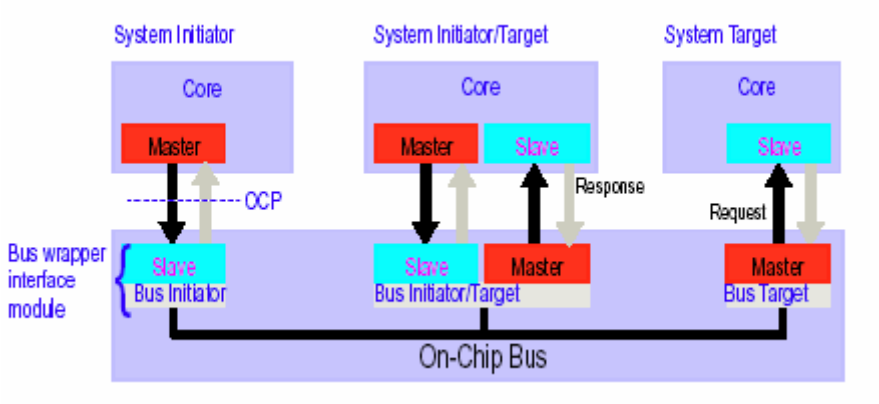

Figure 3: Shared Bus Topology [7]

One problem that is evident with shared interfaces is the fact that not all entities can communicate at the same time. In many implementations of bus topologies, several

entities are capable of mastering the bus. However, it is impossible for more than one of those to master the bus at any given time. As a result, if two bus masters need to communicate with other entities at the same time, one would have to wait for the other to finish before it could start its communication. This uncertainty makes the performance of bus topologies extremely undeterministic.

In a dedicated interface, there is a dedicated communication link between a single master and a single slave. As a result of this dedicated link, the hardware required to manage multiple connections to the interface is not needed. This makes the management of the connection simpler, and the performance of the connection higher when compared to shared interfaces.

2.3 Platform FPGAs

FPGA manufactures, such as Xilinx, have begun introducing FPGAs with architectures capable of providing complete on-chip solutions. In addition to the more traditional CLB arrays, these platform FPGA architectures contain embedded memories, processors, clock managers, arithmetic units, high speed i/o etc. These heterogeneous architectures benefit from the extension of traditional SoC techniques onto the fabric of the FPGA. The Virtex-II Pro FPGA is one of the more recent platform FPGA architectures that have been introduced by Xilinx. In addition to containing the traditional

elements that are characteristic of previous Platform FPGA generations, the Virtex-II Pro contains the PowerPC405 Processor Block.

2.3.1 The PowerPC405 Processor Block

The processor block, seen in Figure 4, contains the PowerPC405 core, specially designed logic that interfaces the core with the surrounds CLBs, block RAMs, and general-purpose routing resources. The number of processor blocks that are present is dependent on the specific Virtex-II Pro device.

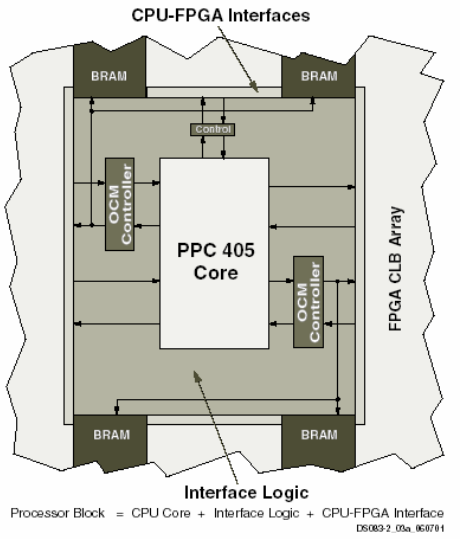

Figure 4: Processor Block Architecture [20]

2.3.2 The PowerPC405 Processor Core

The PowerPC405 (PowerPC405), seen in Figure 5, is a 0.13um implementation of IBM's PowerPC 405D4 core that is engineered for low power consumption at a clock speed of up to 300MHz. This embedded core implements the PowerPC User Instruction Set Architecture (UISA), user-level registers, programming model, data types, and addressing modes for 32-bit fixed point operations.

The central processing unit (CPU) of the PowerPC405 implements a 5-stage pipeline consisting of fetch, decode, execute, write-back, and load write-back stages. The CPU has a single-issue execute unit containing the general-purpose register file, the arithmetic logic unit, and the multiply-accumulate unit. The execute unit supports all 32-bit PowerPC UISA integer instructions in hardware. Floating-point calculations are not supported in hardware, but can be emulated using software.

The PowerPC405 contains a Memory Management Unit (MMU) that is capable of supporting a 4GB logical address space. The MMU is responsible for translating between logical and physical memory addresses. The address translations are handled by the MMU's 64-entry translation look-aside buffer (TLB). To prevent TLB contention between data and instruction accesses, a 4-entry instruction and an 8-entry data shadow TLB are maintained.

The PowerPC405 has separate instruction and data cache units. Each cache unit includes a processor local bus (PLB) master, cache arrays, and a cache controller. Cache

hits appear to the CPU as a single cycle memory access. Cache misses are translated into transactions over the PLB that are serviced by PLB devices.

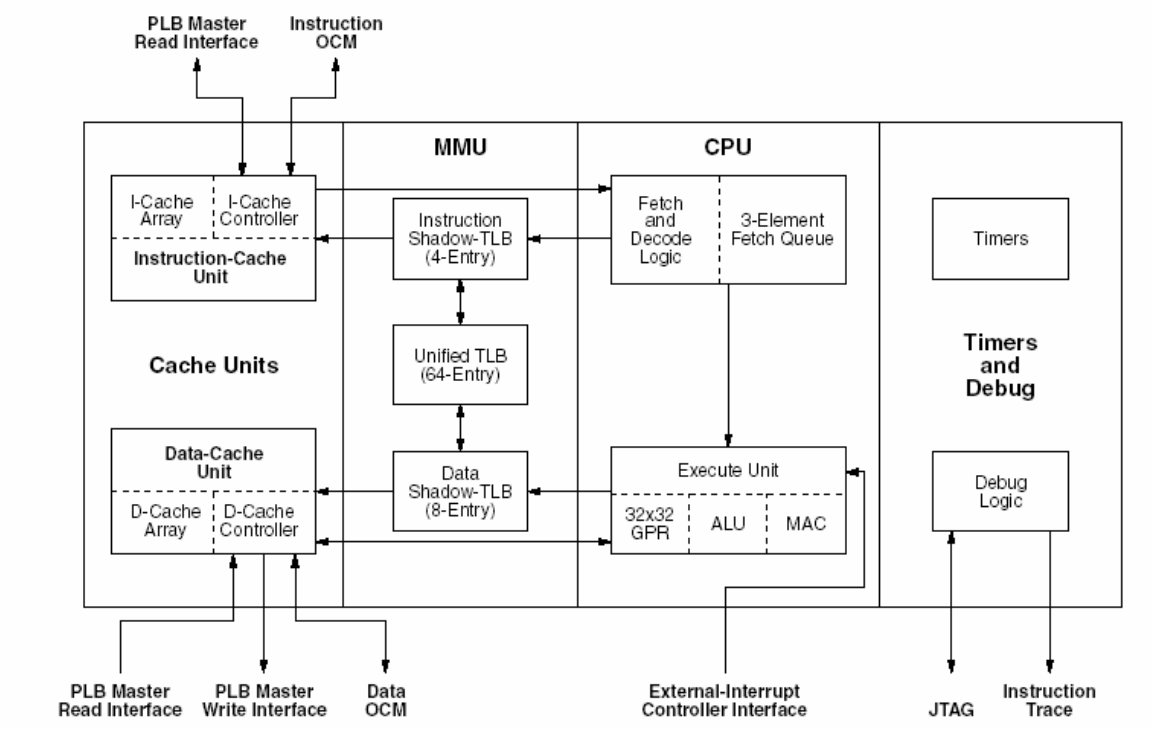

Figure 5: PowerPC Hardware Organization [15]

2.3.3 The On-Chip Memory Controller

The On-Chip Memory (OCM) Controller is responsible for generating the processor's OCM signals. These signal enable communication between the processor and the OCM BRAM. The PowerPC's OCM signals are engineered to provide quick access

to a fixed amount of data and instruction memory. The OCM controller provides the processor with access to both the 64-bit Instruction Side BlockRAMs (ISBRAM) and the 32-bit Data Side BlockRAMs (DSBRAM). The designer may choose to implement designs with various combinations of ISBRAM or DSBRAM.

2.3.4 Interfacing to the Processor Block

The two interfaces that allow the processor block to communicate data to and from the FPGA fabric are the PLB and OCM interfaces. Each interface has advantages and disadvantages. For instance, although the PLB interface is capable of addressing 4GB of memory, the interface is share by multiple components. The OCM interface can provide higher performance through its sole dedication to a single device. This higher performance is contingent upon the interface being configured to address 64 kb of memory or less. If the OCM address space is configured to support anything more than 64 kb, the performance advantage of the interface may be compromised.

2.3.5 The CoreConnect Architecture

The PLB interface is engineered to communicate with a third party on-chip bus architecture. Developed by IBM, this architecture is referred to as the CoreConnect architecture. The CoreConnect Architecture is a hierarchical bus topology that has been

designed to provide efficient on-chip communications. The topology consists of 3 buses: the Processor Local Bus(PLB), the On-Chip Peripheral Bus (OPB), and the Device Control Register (DCR) Bus. Each bus and its associated IP are implemented on the FPGA's fabric using re-configurable resources. Either the PLB or the DCR can be used in an implementation without the existence of the other. The OPB interfaces to the PowerPC405 through the PLB. Therefore, the PLB must be instantiated in designs that require use of the OPB.

The PowerPC405 uses the PLB to access devices that demand high performance, such as memory controllers. The PLB is a fully synchronous 64-bit data bus that supports read and write transfers between master and slave devices. Each PLB master is attached to the PLB through separate address, read-data, and write-data buses. PLB slaves are attached to the PLB through shared but decoupled, address, read-data, and write-data buses and a plurality of transfer and status control signals. Devices that wish to communicate over the PLB must first contact the PLB arbiter. The PLB arbiter will grant the device access to the PLB if either the bus is not in use, or the priority of the requesting device is higher than the priority of the device that is currently mastering the bus. If neither of those two conditions is true, the device that is requesting access to the bus will need to wait until either of those conditions becomes true.

The PowerPC405 uses the OPB to communicate with low speed devices, such as a Universal Asynchronous Receiver/Transmitter (UART). The OPB is a fully synchronous 32-bit data bus that functions independently of the PLB on a different level

of the bus hierarchy. The OPB does not interface directly to the PowerPC405 core. Instead an OPB to PLB bridge provides the interface between the two levels of hierarchy. Therefore, if an OPB master needs to communicate with the processor, it must do so by using the OPB to PLB bridge to generate the appropriate request on the PLB. Similarly, if the processor wants to communicate with a device on the OPB, it must do so by using the PLB to OPB bridge to generate the appropriate request on the OPB.

The DCR bus provides the ability to transfer data between the CPU's general purpose registers and the DCR slave logic's device control registers. The DCR bus allows control and status information to be communicated over a dedicated bus. Without the existence of the DCR, the OCM and PLB interfaces would be responsible for communicating the applications data as well as status and control information. This would cause the performance of the implementation to suffer. The existence of the DCR reduces contention over the OCM, PLB, and OCM interfaces,

2.3.6 The On-Chip Memory Interface

The On-Chip Memory (OCM) interface serves as the interface between the Block RAMs and the OCM signals that are contained within the PowerPC405 processor core. An OCM controller provides non-cacheable access to instruction-side and data-side on-chip memory. The data-side OCM supports a 32-bit, bi-directional memory interface, and the instruction-side supports a 64-bit unidirectional interface. Unlike the PLB, the

OCM interface is a dedicated interface that has separate interfaces for instruction and data accesses. As a result, there is no arbiter that mediates connections to the bus. Each OCM controller is capable of addressing up to 16 MB of memory; however, the amount of BRAM contained within the device will limit the amount of memory than can be used. Furthermore, as the amount of memory connected to the OCM controller increases, the performance of the interface decreases. This performance decrease is a direct result of the increase in resources that are required to accommodate the additional BRAMs used to supply the memory.

The primary advantage of the OCM interface is that it guarantees a fixed latency of execution. This provides a higher level of determinism, making communication over the OCM interface a good choice for processor applications that must guarantee a specific rate of communication with the fabric. This of course assumes that the OCM interface is capable of delivering the required rate of communication.

Another advantage of the OCM interface is that it handles separate data and address for both the instruction and data side. This allows the processor to fetch both instructions and data through the OCM interface simultaneously.

The disadvantage of the processor's OCM interface is that the interface is only capable of addressing a relatively small amount of memory. The data-side OCM (DSOCM) and instruction-side OCM (ISOCM) interfaces are physically capable of addressing 16MB of memory. However, the complexity of the routing that is needed to connect the BRAM to the OCM interface is directly proportional to the amount of OCM

that is used. As a result, the OCM interface performs best when it only has to service a small amount of BRAM.

Applications utilize the dual ported feature of the OCM's BRAMs to communicate data between the processor and the fabric of the FPGA. This feature allows both the processor and hardware modules implemented on the fabric to independently write and read data to and from the BRAM (with some restrictions). Data is communicated between the two entities when one entity, such as the processor or a block of configured logic, reads data that has been placed in BRAM by the other entity.

2.3.7 The Software/Hardware Interface

Previous sections have discussed the physical interfaces provide the connections between the PowerPC405 processor and the FPGA's fabric. Almost equally important is initiation of communications through the interface. Specifically, how software instructions are able to drive logic that exists within the fabric of the FPGA. The PowerPC405 and its surrounding communication architecture is no different then any other processor system in the sense that it can be modeled as a single processing element that is capable of performing mathematical operations, writing data to memory, and reading data from memory. The processor assumes that the devices that are connected to it are some form of memory element. When the processor executes a software instruction that requires it to communicate with one of its devices, the processor will present data

and its associated address on one of its buses. The higher bits of the address value will determine which bus the information appears on. Once the information has appeared on the appropriate bus, the devices connected to that bus are responsible for determining which device the processor wants to interact with. Each device that is connected to the processor has its own address decoder. When an address is presented on its interface, each device's address decoder examines the higher bits of the address value. If the address value falls within the address range of the device that the address decoder is servicing, the address decoder will signal the device. The device will then collect the data off the bus and store the data in the location indicated by the lower bits of the address value. This storage location could reside in a register, on-chip memory, or even off-chip memory. The device will then will then perform the action that is indicated by the recently loaded values. This model is valid for both the OCM and PLB interfaces.

2.3.8 Processor Centric versus Logic Centric

Processor centric and logic centric are terms used to characterize the flow of processed data within a specific implementation. In a processor centric system, the logic serves as the computational adjunct to the processor. This means that the logic will perform based on the commands issued by the processor. Processor centric systems are sequential machines that execute instructions in succession. As a result, scenarios will

arise where logic entities will exist in an idle state awaiting further instructions from the processor.

The goal of any FPGA implementation is to maximize the amount of processing that is occurring at any given time. In many instances, it would be unacceptable for logic within an FPGA to be idle unnecessarily. Furthermore, the processing capabilities of the FPGA could be limited by the processing capability of the processor that is at the center of the processor centric system. This would be extremely inefficient since the processing capabilities of the logic as a whole is far greater than a single processor core. As a result, processor center designs do not map well to FPGAs in most instances.

Alternatively, designers should consider a logic centric approach that uses the processor as a computational adjunct to the logic. Logic centric approaches facilitate the exploitation of parallelism that is often critical to achieving the most efficient implementations. In a logic centric approach, the logic issues commands to the processor. While the processor is responding to those commands, the logic can continue to work concurrently.

The type of architecture that is chosen to provide communications between the processor and the FPGA's fabric is often determined by the centricity of the design. The bus topology, discussed in Section 2.2.2, is representative of processor centric models. Processor centric implementations use the processor as the computational center of the application. As a result, the rest of the entities that exist within the implementation must be able to communicate with the processor so that they may respond appropriately to the

actions of the processor. The shared interface that is provided by the bus topology suits this model well because it gives a single processor the ability to communicate with multiple devices.

Dedicated interfaces are representative of logic centric implementations. Logic centric applications use the processor as a computational adjunct to the processing that is occurring within the logic of the FPGA. The role of the processor as an adjunct processor is specific and does not require that the processor be able to communicate with multiple devices. The use of dedicate interfaces for these types of scenarios saves resources that would otherwise be consumed by the mechanisms that enable the sharing of a single interfaces by multiple devices. Furthermore, use of a dedicated interface in the logic centric approach provides a higher level of determinism for the communications that occurs between the processor and the surrounding FPGA fabric.

2.4 Software Defined Radios

A Software Defines Radio (SDR) is a collection of hardware and software technologies that enable reconfigurable system architectures for wireless networks and user terminals. SDR provides an efficient and comparatively inexpensive solution to the problem of building multimode, multiband, and multifunctional wireless devices that can be updated, adapted, or enhanced using software upgrades [10].

SDR has generated a tremendous interest in the wireless communications industry for the wide ranging economic and deployment benefits it offers. Many companies struggle to keep up with the diverse technologies that exist within the industry and the high rate at which these technologies evolve. SDR aims to reduce these problems by providing radio platforms that can be easily reconfigured. Such a platform would allow companies to build radios that could survive generations of evolving communication standards. This saves companies enormous amounts of financial resources by allowing them to update existing infrastructures instead of replacing them.

In addition to providing a means to update existing systems, reconfigurable radios are beneficial in scenarios where radio users may be traveling between different radio domains. For instance, consider a cellular phone user that is traveling between the US and Europe. If a SDR implementation of the cellular phone were possible, a person traveling from the US to Europe could simple reconfigure his/her phone to work with the cellular phone networks in the country that he/she is traveling to.

The high performance and high reconfigurability requirements of SDR make FPGA architectures a good implementation fabric for many SDR applications. SDR processing tasks such as modulation, de-modulation, up-conversion, and down-conversion benefit from the high performance that is offered by modern FPGA architectures. In addition, the high level of reconfigurability associated with FPGAs makes it possible to configure the FPGA with instances of these processing tasks that are most appropriate for the current communications environment.

In addition to FPGAs, general-purpose processors (GPPs) play an important role in most software defined radio systems. In addition to executing data processing tasks that contribute to the actual output of the radio, GPPs are used in the initialization and configuration of the various hardware components that constitute the radio. The FPGA and GPP usually exist with an SDR system as two separate entities interconnected by an off-chip communications architecture. However, the introduction of FPGA architectures that contain embedded processors, such as the Virtex-II Pro, has prompted some SDR system designers to beginning integrating functionality of the two separate entities onto a single piece of silicon.

2.4.1 The FM3TR Proposed Reference Waveform

Future Multi-Band, Multi-Waveform, Modular, Tactical Radio Waveform (FM^3TR) is an international cooperative effort between the United States, Germany, France, and the United Kingdom to develop a reconfigurable communications system for ground and airborne applications. The result of this effort is a relatively simple and unclassified waveform that can be used to demonstrate interoperability between various software defined radios [1]. The processing requirement of the waveform itself is relatively low when compared to industrial waveforms, such as WCDMA. The simplicity and low processing requirements of FM3TR are appropriate for our application because

they allow us to focus on studying the architectural issues associated with FPGA development using embedded processors and not implementation issues.

2.5 Related Work

As the complexity of IC designs continue to increase, more and more designers are adopting SoC design methodologies to accommodate the integration of components into a single piece of silicon. This is true for both FPGA and ASIC designers. The following discussions present a brief survey of applications that have been mapped to SoC platforms.

2.5.1 The Single-chip Gigabit Mixed-version IP Router

Gordon Brebner has developed an SOC application intended for use with computer networks that contain a mixture of IPv4 and IPv6 workstations. The platform makes extensive use of the PowerPC405 core on the Virtex-II Pro FPGA. [2]

The purpose of the platform is to harness programmable logic in novel ways to build a Mixed-version IP Router (MIR) that will route packets between IPv4 and IPv6 networks. Prototype A uses a processor centric architecture, illustrated in Figure 6. That basic organization is that there are four gigabit Ethernet ports implemented in logic and attached to Rocket I/O transceivers via a 32-bit interface, and one processing hub that is

implemented using the PowerPC. All packets received by the router must pass through the hub before any further transmission. Communication of packets between ports and the hub is via dedicated dual ported buffers that have been connected to the processor's OCM interface.

Prototype B, seen in Figure 7, uses a logic centric architecture to process the system's incoming packets. In this scenario, a subset of the packet processing occurs in the exchange module. Specifically, IPv4 packets that just need to pass through or IPv4 packets that are headed for IPv6 tunnels are handled in logic. All other IPv4 packets, and all IPv6 packets are handled by the processor.

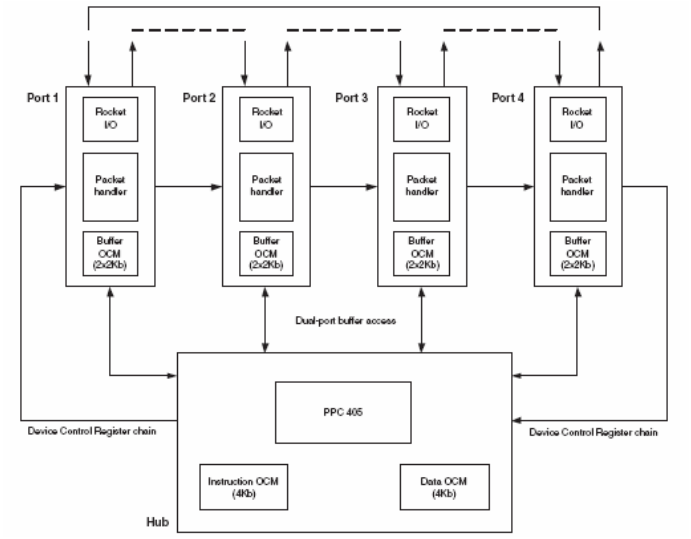

Figure 6 : Single-chip Gigabit Mixed-version IP Router Prototype A [2]

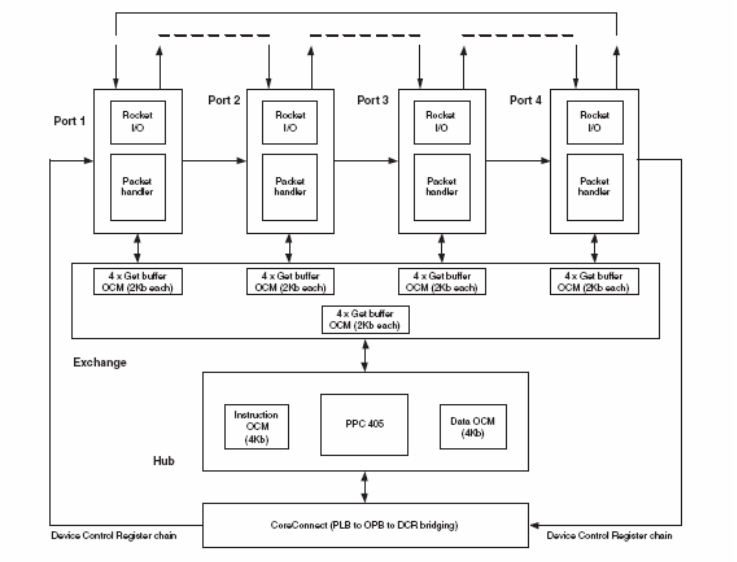

Figure 7 : Single-chip Gigabit Mixed-version IP Router Prototype B [2]

The difference between the implementations of Prototypes A and B is in how the packet processing is handled. Prototype A uses a processor centric approach that requires all of the incoming packets to be processed by the processor. In this scenario, the rate at which the system can process data is limited by the rate at which the processor can process data. Furthermore, the processor core is an inherently sequential device, meaning that the possibility of exploiting computational parallelism doesn't exist. Although the model may work well for the requirements of the MIR implementation, scalability may be an issue due to the fact that the processing capabilities of the implementation is limited by a single processing entity.

Prototype B uses a logic centric approach. In this approach, the processor performs a subset of the overall packet processing. If the packet falls into a specific

category, the packet is processed by the processor. Otherwise the packet gets processed by logic. In this scenario, the processing capabilities of the system are not limited by the sequentiality of the processor. The processing of packets by the processor and the logic can occur concurrently. This allows the implementation to take advantage of the parallelism that is inherently characteristic of FPGA architectures.

2.5.2 Software Decelerators

In the paper, titled *Software Decelerators*, the authors describe a logic centric design technique where processing tasks are offloaded from the logic to the processor of a Virtex-II Pro FPGA [4]. This technique is described by the authors using the term "software decelerator". The term implies that a processing task offloaded from logic to a processor yields no gain in performance, but may result in an implementation that uses the FPGA's resources more efficiently. The argument for software decelerators is a consequence of the fact that the PowerPC core exists in the fabric of the FPGA regardless of whether or not it is used in a given design implementation. As a result, using the processor for processing tasks with liberal timing constraints may free up logic for use elsewhere in the design.

To ensure that software decelerators provide their anticipated benefits, the authors present a list of considerations that should be reviewed prior to the design of the software decelerator. Some of these considerations are:

1. The overall area consumed by the software decelerator should not exceed the area consumed by the equivalent logic implementation unless there are valid arguments to do so.
2. The resource consumption of the interface between the processor and the logic should be minimized.
3. The designer should be able to acquire accurate timing and resource utilization information for the logic centric system. The information that is acquired should include those tasks that are being performed by the processor.

The authors provide a case study to illustrate the benefits of software decelerators. The case study emphasizes the use of the embedded processor as finite state machine. Finite state machines (FSMs) play critical roles in the control of many hardware implementations. Their complexity is a function of the number of states they implement, as well as the complexity of the logic equations used to trigger state changes. Despite their potential complexity, their timing constraints, in comparison to the rest of the system, are relatively relaxed. As a result, FSMs can be good candidates for execution on the Virtex-II Pro's embedded processor.

The case study presented by the authors describes three FSMs that are used for network related applications. The first FSM application, rs232echo, is a RS232 protocol-handling machine. This application serves as a repeater that broadcasts all of its received inputs on its output. The second application handled the Media Independent Interface

(MII) of an Ethernet MAC. Finally, the third application FSM, tx_host_io, handles the host interface to a 10G Ethernet MAC.

The results of the logic implementations of the FSMs are shown in Table 1. These figures show the amount of resources needed to guarantee the FSMs were able to perform at the rate required by the application. Table 2 shows the resource savings that result from executing the FSMs in the embedded processor. These values are relative to the respective values in Table 1. Although the processor implementations consistently result in resource savings, Table 3 indicates that the performance requirement for the tx_host_io application cannot be met by a software implementation. This leads the authors to conclude that FSMs with high performance constraints are not candidates for execution on the embedded processor and should be implemented using only the logic resources of the FPGA, while FSMs with low performance requirements are good candidates for execution on the embedded processor.

Machine	Input width	Output width	Number of states	Registers	LUTs	Required frequency
rs232echo	3	1	12	92	111	115 kHz
miim	33	20	33	26	61	2.5 MHz
tx_host_io	90	94	5	142	320	156 MHz

Table 1 : Resource Consumption of FSM Logic Implementations [4]

	OCM			DCR			PLB		
Machine	Registers	LUTs	Ratio	Registers	LUTs	Ratio	Registers	LUTs	Ratio
rs232echo	1	4	3.6%	2	6	5.4%	4	8	7.2%
miim	20	38	62.3%	21	40	65.6%	23	42	68.9%
tx_host_io	94	75	23.4%	95	77	24.1%	97	79	24.7%

Table 2 : Resource Savings Relative to the Equivalent Logic Implementation [4]

Machine	Worst-case performance (cycles)	Worst-case performance (MHz)	% of time in I/O	Code size (kbytes)	Code size as % of cache
rs232echo	40	8.75	30.95%	1416	8.6%
miim	74	4.730	25.22%	2968	18.1%
tx_host_io	135	2.593	33.99%	1952	11.9%

Table 3 : Performance Results for Software Implementations of FSMs [4]

2.5.3 PLB vs. OCM Comparison Using The Packet Processor System

The Xilinx application note, *PLB vs. OCM Comparison Using The Packet Processor Software,* uses a packet processing application to compare the performance tradeoffs of the PowerPC405 Processor Local Bus and On-Chip Memory interfaces [5]. The application's implementation is a modified implementation of the application presented in [11]. The objectives for the study of the packet processor application include:

1. Identify performance penalties for the OCM and PLB interfaces that result from the system's current configuration.
2. Compare the performance that results from fetching instructions through the PLB interface versus fetching instructions through the OCM interface.
3. Compare the performance of the PLB-attached packet buffers versus the OCM-attached packet buffers.
4. Compare the application performance of the PowerPC405 processor when operating at the same clock frequency as the PLB and OCM interfaces versus the performance of the application when the processor is operating at a faster clock frequency then the PLB and OCM interfaces.

Figure 8 shows the architecture of the packet processor system used in the study. The system uses two Packet Processing Engines (PPEs): one connected to DSOCM BRAM and the other connected to PLB BRAM. Depending on which processor is producing packets, the PowerPC405 will receive packets through either its PLB or OCM interface. Upon reception of a packet, the PowerPC405 will examine the contents of the packet to determine the packet's destination. If the packet is destined for the PowerPC405, the processor will send an acknowledgement to the packet processor that the packed originated from. If the

packet is not destined for the PowerPC405, the processor forwards that packet to the other packet processor.

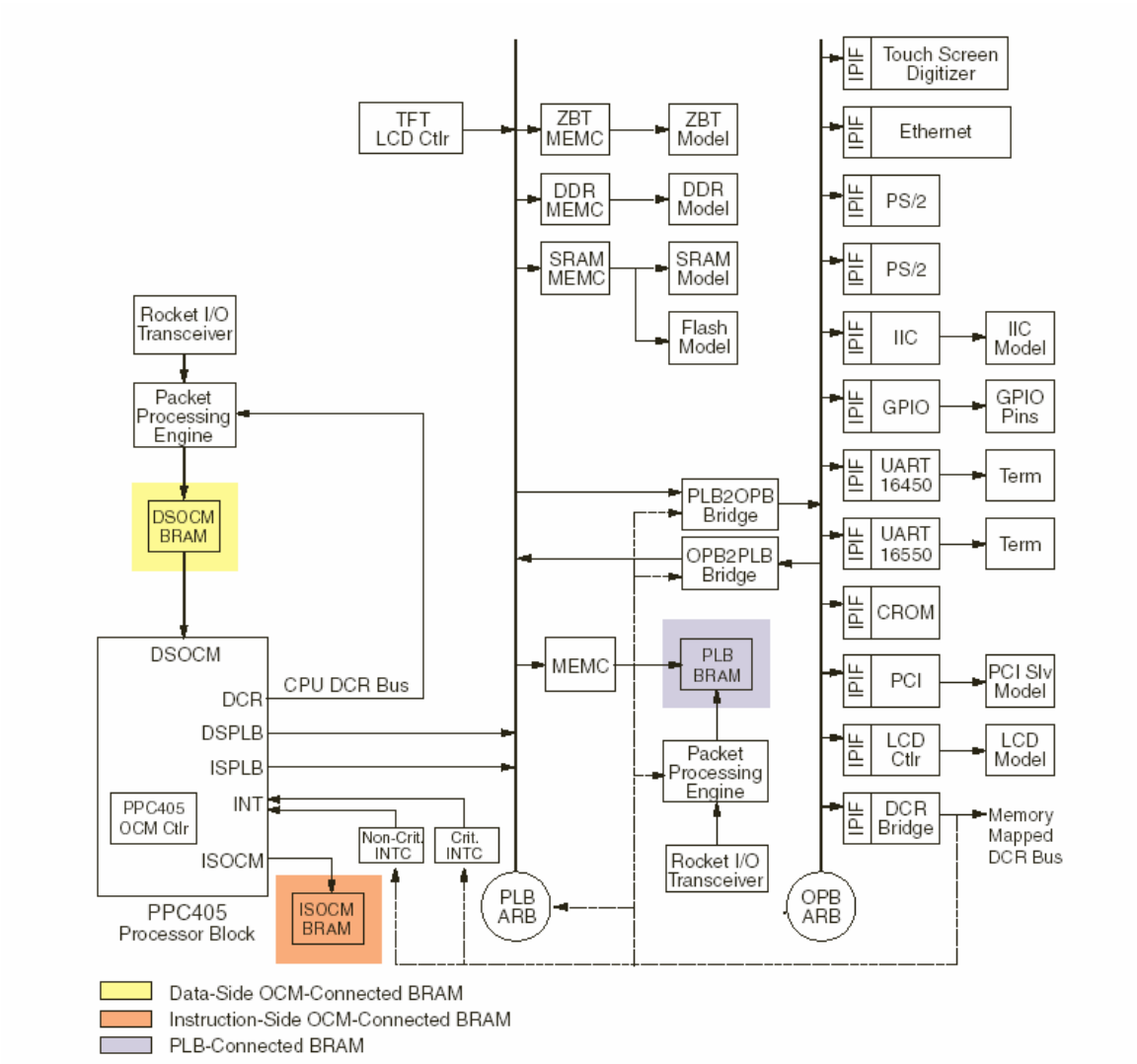

Figure 8: Architecture of the Packet Processor Reference System [5]

Each experiment that is performed in this study uses one of three different test cases. Each test case uses a different memory organization. The first test case stores the

instruction and program data for the PPC in PLB BRAM. The packet processor's data is stored in ISOCM. The second test case stores all of the instructions in ISOCM BRAM. The program data and the packet processor's data are stored in DSOCM BRAM. Finally, the third test case stores the instruction, program, and packet processor data in PLB BRAM. In all cases the D-Cache for the PLB interface has been disabled. The I-Cache for the PLB interface is enabled when appropriate. The different test cases and the clock frequencies at which they operate are summarized by Table 4 and Table 5.

Table 4 : Packet Processor Reference System Design Details By Case [5]

Case	Instruction Location	Program Data	Packet Proc Memory	D-Cache	I-Cache	Software & Mapfile Used
Case 1A, 1B	PLB BRAM	PLB BRAM	DSOCM	OFF	ON	pkt_proc mapfile1
Case 2A, 2B	ISOCM(1)	DSOCM	DSOCM	OFF	n/a	pkt_proc mapfile2
Case 3A, 3B	PLB BRAM	PLB BRAM	PLB BRAM	OFF	ON	pkt_proc mapfile3

Table 5 : Operating Frequencies of the Packet Processor Test Cases

Case	processor	OCM	PLB/OPB	DCR	MGT/PPE
All "A" Cases	100 MHz	100 MHz	100 MHz	100 MHz	156 MHz
All "B" Cases	300 MHz	150 MHz	100 MHz	100 MHz	156 MHz

Comparison 1 measures the elapsed time from the deassertion of the system reset to the high state of the transmission's control signal. The measurement takes into account

all packet transfers as well as instruction fetching, and execution, which are common between all test cases.

The analysis of Comparison 1 is summarized by Table 6. Case 1B is the fastest overall; however, Case 2B completes the first transfer quicker. Case 2B is initially faster because of the time that elapses before all of Case 1B's instruction have made their way into the PPC cache. Once its instructions have been loaded into the cache, Case 1B operates slightly faster than Case 2B, and significantly faster than Case 3B. This suggests that for the clock ratios used in Cases *B, the PLB interface is comparable to the ISOCM interface for applications where the program's instruction can fit entirely in cache.

Table 6 : Overall Performance Measurements

	1st "transmit" (a + c)	2nd "transmit" (a)	3rd "transmit" (c)	4th "transmit" (a)	Units
All Clocks 100 MHz					
1A PLB + DSOCM	16.471	48.668	63.577	78.512	µs
2A All OCM	16.567	51.635	68.814	85.852	µs
3A All PLB	29.252	81.878	107.307	132.460	µs
Clock Frequencies: Processor, OCM, PLB = 300 MHz,150 MHz,100 MHz					
1B PLB + DSOCM	12.048	29.829	39.111	48.232	µs
2B All OCM	11.830	36.329	48.732	61.200	µs
3B All PLB	23.566	61.501	82.237	102.890	µs

Comparison 2 measures the time it takes for one packet to be transferred from one area of memory to another. The move will occur through either the PLB or the OCM

interface, depending on the case. The measurement is made starting with the first packet byte on the read bus and ending with the last packet byte on the write bus.

The results of comparison 2 are summarized in Table 7. Once again, fetching instructions out of the i-cache is shown to deliver performance that is slightly greater than instruction fetches over the ISOCM interface. The results also suggest the data transfers through the DSOCM interface deliver greater performance than transfers through the PLB. This can be seen by comparing Cases 3A and 3B against the rest of the cases.

Cases 3A and 3B do not reflect the true bandwidth of the PLB. In this particular case, the 64-bit PLB is transferring only one word (32 bits) of data at a time. If 64-bit transfers were done, the numbers in rows 3A and 3B would be approximately half of what they are. Making 64-bit transfers, however, requires assembly language instructions that some compilers do not take advantage of. This is a great example of why the processor should not be used for large data transfers. Instead, use the FPGA to do a DMA transfer.

Table 7 : Data Movement Resultsfor TC2 [8]

	1st "transmit" (a + c)	2nd "transmit" (a)	3rd "transmit" (c)	4th "transmit" (a)	Units
All Clocks 100 MHz					
1A PLB + DSCOM	7.7	10.24	10.24	10.22	µs
2A All OCM	10.22	12.76	12.76	12.76	µs
3A All PLB	20.48	20.47	20.48	20.48	µs
Clock Frequencies: Processor, OCM, PLB = 300 MHz,150 MHz,100 MHz					
1B PLB + DSCOM	5.126	5.143	5.143	5.123	µs
2B All OCM	6.816	8.510	8.51	8.51	µs
3B All PLB	16.640	16.640	16.640	16.640	µs

Finally, Comparison 3 measures the time it takes for the system to complete the diagnostic routine that is part of the implementation. The routine contains a mixture of data accesses that must be performed over both the PLB and OCM interfaces. In addition, some of the instructions that are fetched over the PLB will only be executed once. Other instructions will be frequently accessed, allowing the benefits of the cache to be utilized.

The results of Comparison 3 are summarized in Table 8. Cases 1B and 2B execute in roughly the same amount of time. This suggests that for applications where only a fraction of the instructions will fit in cache the ISOCM will provide comparable performance when the PLB operates at 100 MHz and the ISOCM operates at 150 MHz.

Table 8 : Test Completion Measurment Results

	Diagnostic Routine FInished	Units
All Clocks 100 MHz		
1A PLB + DSOCM	36.659	µs
2A All OCM	37.589	µs
3A All PLB	59.649	µs
Clock Frequencies: Processor, OCM, PLB = 300 MHz,150 MHz,100 MHz		
1B PLB + DSOCM	23.423	µs
2B All OCM	26.985	µs
3B All PLB	43.593	µs

The PLB versus OCM study discusses the main differences between the PLB and OCM interfaces by contrasting their attributes, as well as comparing the interfaces using a hardware reference system. The results of this study show the following:

1. The OCM interface can provide comparable performance to the PowerPC‘s caches at 1:1 and 2:1 processor-to-clock-to-OCM clock ratios.
2. Using OCM to reduce the amount of traffic on the PLB reduces cache thrashing in large applications.
3. Operations that require determinism benefit from using the OCM interface because of its deterministic behavior.
4. Applications that use less than 16KB of program and instruction data should run completely out of the PowerPC’s cache.

2.5.4 Energy Efficient Application Synthesis Using Platform FPGAs

In the paper, titled *A Methodology for Energy Efficient Application Synthesis Using Platform FPGAs*, the authors argue that the design choices that an FPGA application designer makes can have a dramatic impact on the power consumption of his/her application. The author's analysis focuses on the buses that enable communication between the Virtex-II Pro's PowerPC405 and the surrounding FPGA fabric. Specifically, the Processor Local Bus (PLB) and the On-Chip Memory (OCM) Bus [22].

The paper presents an FFT software program that executes on the Virtex-II Pro's PowerPC405. This program serves as a vehicle that enables power analysis of the processor's PLB and OCM interfaces.

The study shows that executing the same FFT program on different configurations of the PowerPC405 processor core results in configurations that consume five times the time and energy costs of alternative configurations. Implementations that used the PLB bus with caching enabled dissipated the least amount of energy while implementations that use the PLB bus with caching disabled dissipate the highest amount of energy. Caching reduces the devices power consumption since most of the FFT processing executes within loops. The data that is operated on within an execution loop fits entirely in cache, This means that instruction and data can be fetched from the cache without generating transactions on the PLB. Communications over the PLB also introduces overhead that is not present with communications through the OCM interface. This

overhead increases the energy dissipation on the PLB further as the PowerPC spends more time waiting for instructions and data to be fetched over the PLB bus.

2.5.5 The Novelty of our Approach

The Virtex-II Pro Mixed-version IP Router, discussed in Section 2.5.2, illustrates the differences between processor and logic centric processing models. Prototype A requires all incoming packets be processed by the PowerPC405. This is done to show that the performance a processor centric system is clearly bounded by the capabilities of the processor. Prototype B uses a logic centric approach where the processor services specific requests that are generated in logic. These requests are generated for specific types of packets that are received by the system. If a packet requires the processor, the logic can dispatch that packet to the processor and begin processing the next packet. As a result, the logic and the processor can operate concurrently. This approach eliminates the processor as the limiting factor of the system's performance.

Prototype A of the Mixed-version IP Router is similar to our implementations in the sense that the processor can be the limiting factor in the performance of our application. The input samples of our application are modulated by the PowerPC405 before being converted to a passband signal by the DUC. However, the PowerPC405 used in our implementations is responsible for servicing a single data buffer, while the PowerPC405 used in Prototype A services multiple data input buffers. In addition, our

implementations consider the possibility of using the PLB interface to the processor to service the application's input buffers in place of the OCM interface. The Mixed-version IP Router only considers the processor's OCM interface in either Prototype A or Prototype B.

The FSM based design technique using software decelerators, discussed in Section 2.5.3, describes scenarios where logic centric systems can use embedded processors to execute FSMs. Application designers who choose to adopt this design technique must ensure the processor is capable of delivering the performance needed to operate the FSM at the correct rate. Studies show that in most cases the performance delivered by the Virtex-II Pro's PowerPC405 is more than adequate for a variety of FSM processor implementations. This is a consequence of the fact that the rate at which most applications transition between states is lower than the rate at which data is processed.

Using the FSM as a software decelerator is similar to our approach. Much like the authors of [4], we do not expect the use of the PowerPC405 yield any performance gains for our implementation. However, the possibility does exist that an equivalent hardware implementation of the software application could consume more of the FPGA's resources. A motivation for using the software implementation in such a scenario is that it frees up the FPGA's logic for tasks that have higher performance requirements than those tasks executing on the PowerPC405.

The comparison study that is performed using the packet processor software is similar to our research. Both studies are intended to evaluate the performance of the

PowerPC405's PLB and OCM interface. The comparison study, presented in [5], uses the PowerPC405 to execute a packet processing application that will either forward an incoming packet or acknowledge the receipt of a packet. The packet is forwarded by copying the value of the packet to a new location in memory. The computational requirements of our FM3TR modulation application are greater than those of the packet processing application. As a result of this, we feel that the results of our investigation will offer comparison results from an applicat7ion that is more representative of the types of real-world applications that might target the PowerPC405. Furthermore, the results that are presented in [5] are based on simulation models. Our results are obtained through measurement using actual hardware.

The power analysis study, discussed in section 2.5.5, presents a comparison of the PLB and OCM interfaces from the perspective of their power consumption. While our investigation does not include power measurements, it is important to note that the configuration of a particular interface may alter the applications power consumption. Using the work presented in [22], the power consumption of an interface can be used as the deciding factor between two interfaces that perform satisfactory.

2.6 Summary

This chapter begins with an introduction to FPGAs and how they differ from other modern processing elements. It then introduces System-on-Chip architectures and discusses how the concept of SoC is extended to the architecture of an FPGA. Next, the architecture of the Virtex-II Pro is introduced, and the devices' embodiment of SoC concepts is discussed. The chapter then introduces the notion of a Software Defined Radio and the roles that FPGAs play in the implementation of an SDR. Finally, a survey of related work is presented.

Chapter 3

Experimental Setup

In this chapter, we describe an application that performs FM3TR's modulation and digital up-conversion (DUC) functions. We then describe the algorithms that are used to modulate and up-convert the digital data that is fed into our application. Next we describe the software implementation of the modulator and the hardware implementation of the digital up-converter. We then describe the interfaces that are used to interface the modulator with the digital up-converter. Finally we describe multiple instances of the same application that were developed to analyze tradeoffs associated with the use different interfaces.

3.1 Development Tools

The implementation of the FM3TR Waveform Application was subdivided into two separate problems: the implementation of the modulator and the implementation of the DUC. Each implementation utilizes a separate design flow. In the final stages of the design process, the implementation of the modulator and the DUC were merged together to form a complete application.

The implementation of the modulator required the use of Xilinx's Embedded Development Kit (EDK). The EDK is a development environment that provides application designers with the tools necessary to build embedded processor systems that make use of the Virtex-II Pro's PowerPC405.

The steps within the EDK that are necessary to build the embedded processor system include: hardware platform creation, software platform creation, and software application creation. The hardware platform is defined by the Microprocessor Hardware Specification (MHS) file. The MHS file defines our system architecture, memory modules, and embedded processors. It also defines the system's connectivity as well as the configurable options and the address map for each memory module in our system.

The Platform Generator (platgen) parses the MHS file and generates the appropriate netlists and HDL wrappers. These files are then imported into Xilinx's ISE Project Navigator, where they are instantiated in the application.

The software platform is defined by the Microprocessor Software Specification (MSS) file. The MSS file defines driver and library customization parameters for peripherals, standard input/output devices, interrupt handler routines and other software features. The Library Generator (libgen) tool parses the MSS file and configures the libraries and drivers that are required for the application.

Software application creation involves the creation of the FM3TR Modulator that executes on the embedded processor. The code is written in C. Once the source files are created, they are compiled and linked to generate executables in the Executable and Link

(ELF) Format. GNU compiler tools for the PowerPC405 are used in our implementations, but tools from other vendors are available as well.

The FM3TR Digital Up-Converter (DUC) is developed using a different design flow. First, Simulink is used to develop a model that is representative of our system. System Generator is then used to generate a VHDL description of our Simulink model, as well as to create the appropriate project files that enable the design to be imported into the Xilinx's ISE Project Navigator.

In the final stages of the design flow the FM3TR Modulator and the FM3TR DUC are imported into Xilinx's ISE Project Navigator. Implementation specific interfaces are then instantiated to connect the modulator and the DUC. Finally the design is synthesized, placed and routed.

3.2 The FM3TR Waveform Application

Our application aims to demonstrate the advantages and disadvantages of the different interfaces that enable communication between the Virtex-II Pro's FPGA fabric and its embedded processors. The demonstration will perform a subset of the processing tasks that are required in FM3TR Waveform Processing. Specifically, the application will perform the modulation and digital up-conversion that is associated with an FM3TR transmitter. The data that is to be modulated will be preloaded into the FPGA's BlockRAM. Similarly, data that has resulted from this waveform processing will be

retrieved from the FPGA's Block SelectRAM+. It should be noted that the goal of this thesis is to provide an accurate characterization of the advantages and disadvantaged of the interfaces that provide a communication fabric between the PowerPC405 and the surrounding FPGA fabric on a Xilinx Virtex-II Pro. The FM3TR Waveform Application is used as a vehicle to assist in the characterization of the different interfaces. As a result, the implementation details of the modulation and DUC will not be the main focus of this discussion.

3.2.1 FM3TR Modulation

Digital modulation is the process by which digital information is used to alter the characteristic of a given carrier signal. A digital word is communicated to the receiver by transmitting its corresponding symbol on the channel for a predefined time interval, known as the symbol interval. The symbol transmission conveys a unique phase pattern that is associated with a specific digital word. The demodulator is able to determine which symbol was transmitted by examining the phase characteristics of the received signal. Since there is a one-to-one correspondence between transmitted symbols and digital words, the receiving side is able to determine which digital word was transmitted.

The specification for the FM3TR Waveform requires the use of Minimum Shift Keying (MSK) modulation. MSK is a continuous phase modulation technique that uses four signals (4-ary) each with a phase difference of $\pi/2$ [1]. Thus a digital word

consisting of 2 bits can be transmitted during the symbol interval T_b. The input to the MSK modulator consists of a stream of bits. Each bit is delivered to the modulator every T_b interval and can assume a value of either *–1* or *1* (as opposed to *0* or *1*). This representation allows the receiver to distinguish between the reception of a binary *0* and the absence of a signal. The modulator divides its input stream into in-phase and quadrature components. The in-phase component, a_{2n}, consists of all of the input's even bits. The quadrature component, a_{2n+1}, consists of all of the input's odd bits. During each symbol interval, the modulator acquires an in-phase and quadrature value from its input. These in-phase and quadrature bits are use to produce a modulated signal according to the following equation:

$$s(t) = \left\{ \left[\sum_{n=-\infty}^{N} a_{2n} g_T(t - 2nT_b) \right] + \left[\sum_{n=-\infty}^{N} a_{2n+1} g_T(t - 2nT_b - T_b) \right] \right\} \quad (1)$$

where $g_T(t)$ equals

$$g_T(t) = \begin{cases} \sin \dfrac{\pi t}{2T_b} & 0 \le t \le 2T_b \\ 0 & otherwise \end{cases}$$

The phase continuity of an MSK modulated signal, as seen in Equation 1, makes this modulation technique difficult to implement. It can be seen that the maintenance of phase continuity requires the storage of all previous input values. This is due to the fact

that the instantaneous phase value of the carrier signal is a function of all previous phase values. As the length of the transmission grows, the required storage capacity of the system will approach infinity. As a result, the MSK modulation scheme in its current representation is not practical. We use a representation of MSK modulation that describes the modulated signal as a function of the modulator's instantaneous input value and its last calculated phase value. This representation requires that one examine the complex envelope of the modulated signal [8].

The complex envelope describes the in-phase and quadrature components that are used to create the modulated signal. The complex envelope of the signal $s(t)$ is

$$
\begin{aligned}
s'(t) &= \sum_{n=-\infty}^{N} e^{j(\pi/2)A_n} g_T(t - nT_b) = \sum_{n=-\infty}^{N} z_n g_T(t - nT_b) \\
A_n &= \sum_{l=-\infty}^{N} a_l \\
z_n &= e^{j(\pi/2)A_n}
\end{aligned}
\qquad (1)
$$

Instead of expressing z_n as a function of all the modulators previous input values, z_n can be expressed in terms of the modulator's instantaneous input value and the last calculated value of z_n. This is done by using Euler's notation to express z_n as the summation of a sine and a cosine as follows:

$$z_n = e^{j(\pi/2)} A_n$$
$$z_n = \cos\left(\frac{\pi}{2}\sum_{l=-\infty}^{N} a_{2l}\right) + j\sin\left(\frac{\pi}{2}\sum_{l=-\infty}^{N} a_{2l+1}\right) \quad (3)$$

Notice than z_n = *{-j,j}* for even values of N, and z_n = *{-1,1}* for odd values of N. Thus the value of z_n will change by a factor of $\pm j$ each time its value is calculated. The factor by which z_n changes by is positive if $\sum_{l=-\infty}^{N} a_l \geq 0$ and negative if $\sum_{l=-\infty}^{N} a_l < 0$. As a result, $z_n = e^{j(\pi/2)} A_n = j z_{n-1} a_n$ and Equation 2 becomes

$$s'(t) = j z_{n-1} a_n g_T(t - nT_b) \quad (2)$$

where n is the current bit number ranging from $-\infty \leq n \leq \infty$. The complex envelope can then be applied to the carrier multiplying $s'(t)$ and $e^{j2\pi f_c}$ together. The real part of this product is the signal that gets transmitted over the airwaves. One can see that MSK modulation technique described by Equation 4 only requires the storage of the most recent value of z_n. This technique allows the MSK modulation scheme to be implemented on a platform that contains a finite amount of memory.

3.2.2 Digital Up Conversion

Digital Up-Conversion is the process by which a complex digital baseband signal is converted to a real passband signal. The device that is responsible for this conversion is

referred to as a Digital Up-Converter (DUC). In a DUC, the input signal is sampled at a relatively low sampling rate, typically T_b, the symbol rate of the digital modulator. The baseband signal is filtered and converted to a higher sampling rate before being modulated onto the higher frequency carrier.
Figure 9 shows the architecture of the DUC that was used in our investigation. The complex input signals are passed through three stages of filtering, each of which performs a sampling change and the associated low pass interpolation filtering . The three filtering stages include:

- **Pulse Shaping FIR Filter** P(z) provides a sampling rate increase of 2 and performs Nyquist pulse shaping.
- **Compensation FIR Filter** C(z) provides a sampling rate increase of 2 and is used to compensate for the passband distortion of the 3rd stage's cascaded integrator-comb (CIC) filter.
- **Cascaded Integrator-Comb Filter** The CIC is used to cause a sampling rate increase from 4 to a maximum of 1448.

The complex output stream of the filtering stages is up-converted to its final frequency band by passing the complex output stream through a mixer. The mixer multiplies each value of the complex output stream with the appropriate output value of the local oscillator. The sinusoidal signal values that are produced by the local oscillator are generated using a Direct Digital Synthesizer (DDS). The outputs are then combined to form the final DUC passband result [14].

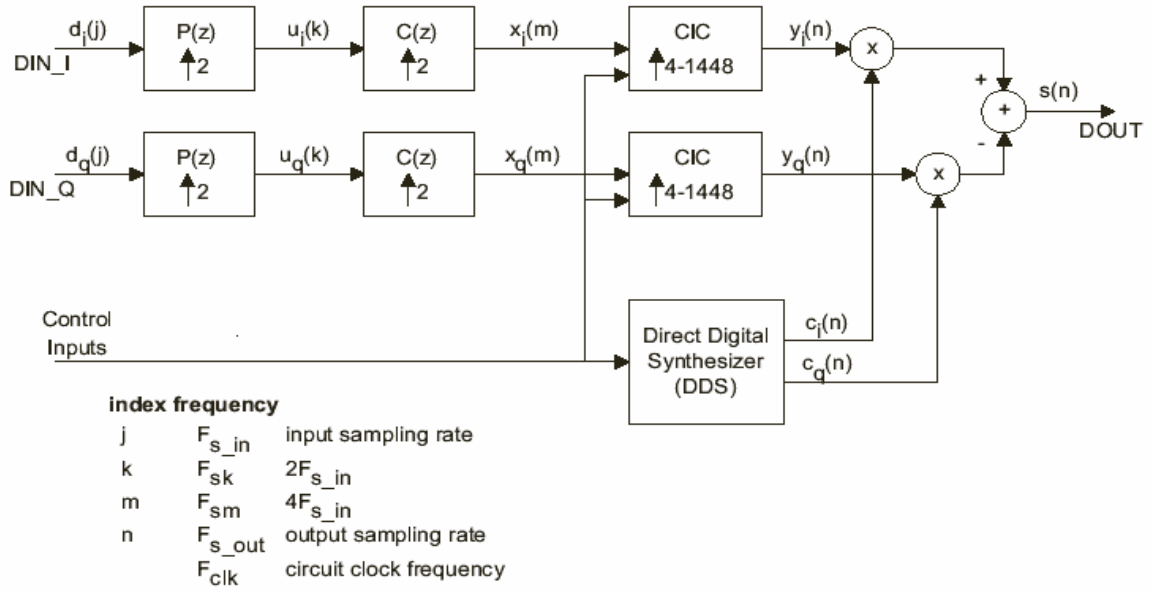

Figure 9: DUC Core Architecture [14]

3.3 Application Overview

Our application, seen in Figure 10, implements FM3TR modulation and its associated digital up conversion using the reconfigurable logic and embedded processors contained within a Virtex-II Pro Platform FPGA. Data is place into the initial First In First Out (FIFO) Queue 0 via a source that is external to the FPGA, such as a waveform encoder. The modulator then pulls the data out of the FIFO 0 and performs MSK modulation. The signal values that result from the modulation are pushed into FIFO 1.

The DUC pulls the modulated signal values out of FIFO 1 and converts them to a real passband signal. Finally, the DUC pushes the real passband signal into FIFO 2. The data can then be used to drive the inputs of a device external to the FPGA, such as a digital-to-analog converter.

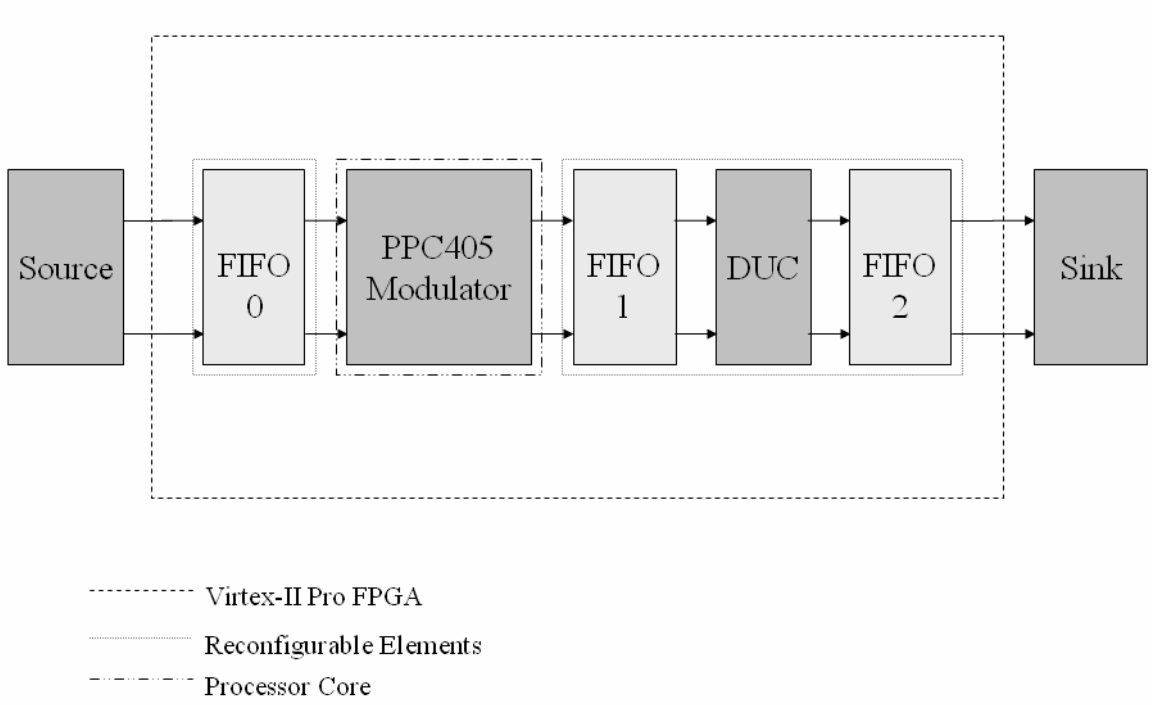

Figure 10: FM3TR Application Architecture

3.4 Data Formatting and Storage

In our experimental setup, the data samples that feed the input to our FM3TR modulator are generated using a MATLAB script. The script produces an array of

numbers that can assume the value of either -1 or 1. The input samples are represented as 8-bit two's complement values which are stored on the PowerPC405 using the *char* type. The input samples are packed into 32-bit words prior to being stored in BlockRAMs. This allows 4 sample values to be stored in each location in BlockRAMs. In our experiments, the bitstream used to configure the FPGA contains the information necessary to initialize the appropriate BlockRAMs with the correct input sample values. As a result, the loading of the input sample values into the FPGA's BlockRAMs is implicit with the configurations of the FPGA.

The output values of the modulator are represented using 16-bit two's complement values with a 15-bit fractional part. These values are represented on the PowerPC405 using the *short* type.

3.4.1 Calculation and Storage of Pulse Values

The calculation of the sinusoidal pulse that is used to modulate the incoming data samples is performed according to the following equation:

$$g_T(t) = \begin{cases} \sin\dfrac{\pi t}{2T_b} & 0 \le t \le 2T_b \\ 0 & otherwise \end{cases}$$

where T_b equals the reciprocal of the modulation. The modulation rate is mandated by the FM3TR specification to be 25,000 symbols per second. The number of samples per symbol is chosen as 4 for this application.

The values of the sinusoidal pulse are specific to the modulation parameters of a specific implementation and only need to be re-calculated if those parameters change. The same pulse values will be used in all subsequent implementations of the FM3TR Application. As a result, we initialize the pulse values during the configuration of the FPGA on all implementations.

3.5 Asynchronous First-In-First-Out Queues

Applications that use the embedded processor often clock the reconfigurable logic and the processor core at different rates. In addition, the execution time of any given software application can differ between successive executions as a consequence of traffic on the PLB and OPB, software interrupts, etc. This makes synchronous data transfers between the processor's memory and other logic entities difficult. As a result, we use asynchronous FIFOs to mediate connections between the processor's memory and the surrounding logic

Asynchronous FIFO's can be implemented using the Virtex-II Pro's dual-ported BlockRAM [18]. The entity that produces data writes the data into the BlockRAMs through the appropriate port. The entity that consumes data reads the contents of the BlockRAMs through the appropriate port. Additional logic is needed to maintain flags that will tell either entity when the FIFO is full, empty, etc. The independence between the two ports of the BlockRAMs allows data to be produced and consumed at different

clock rates. This can be seen in Figure 11. Data that is pushed into the FIFO will be presented on the output port of the FIFO after N clock cycles. Notice that in this example the rate at which data is being pushed into the FIFO is less than the rate at which data is being pulled from the FIFO. This corresponds to a consumption rate that is higher than the production rate. Eventually the FIFO will empty, making any additional data that gets consumed invalid. To avoid this situation, asynchronous FIFOs have FULL and EMPTY flags. If the FIFO's FULL flag is asserted, the producer will know that the FIFO cannot accept any additional data. The producer will then respond appropriately. Similarly, if the FIFO's EMPTY flag is asserted, the consumer will know that the FIFO has no additional data to present. The consumer will then respond appropriately. This description of the Asynchronous FIFO is based on an ideal model. The actual Asynchronous FIFO implementations that exist within our implementation differ in their use of memory and handshaking.

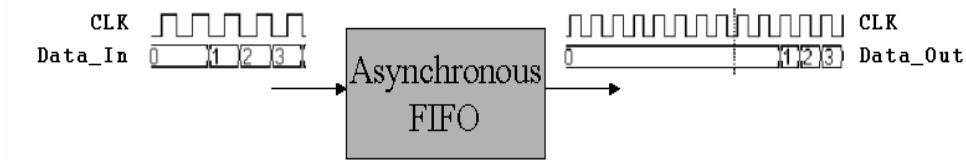

Figure 11: Asynchronous FIFO Example

Our application uses three Asynchronous FIFOs. FIFO 0, seen in Figure 10, mediates transfer of data between the application's data source and the modulator that is

running on the PowerPC405. In a production version of this application, this source would most likely be a waveform encoder. However, this investigation focuses solely on the modulation and up-conversion stages of waveform processing. In our experiments, this FIFO is modeled using BlockRAMs. The contents of the BlockRAMs are initialized with the appropriate sample values at the time of configuration. The initialized values are representative of the sample values that would be produced by an FM3TR encoder. The values are stored as 8-bit 2's compliment numbers. As a result, no handshaking is required for this FIFO.

FIFO 1, seen in Figure 2, mediates the transfer of symbols between the modulator and the DUC. FIFO 1 exists because of the possibility that the modulator and the DUC could produce and consume symbols at different rates. FIFO 1 is implemented by connecting one of the processor's memory interfaces to port A of FIFO 1's BlockRAMs. The software executing on the PPC writes symbols into FIFO 1 through the appropriate interface. When all symbols have been written to BlockRAMs, the DUC is signaled to begin processing.

The DUC is connected to Port B of FIFO 1's BlockRAMs through an interface that we designed. In its initial state, the memory interface continuously polls a specific location in this BlockRAMs. When the value that is stored at this location changes to the appropriate value, the memory interface begins presenting the DUC with valid data.

FIFO 2 is implemented using a customized memory interface and BlockRAMs. In its initial state, the memory interface waits for the qualification of the first DUC output

value. This qualification tells the memory interface that the DUC's output data is valid. Once this initial qualification has occurred, the interface continually writes qualified output values to the BlockRAMs on successive ticks of the clock. This continues until the DUC begins presenting unqualified output values or the storage capacity of the BlockRAMs is reached.

3.6 The FM3TR Modulator Implementation

The FM3TR modulator is implemented in software using the embedded PowerPC processor that is contained within the Virtex II-Pro FPGA. The functionality of the FM3TR modulator is described using the C Programming language. This description is then compiled by a GNU C compiler and the resulting code is downloaded into the BlockRAMs of the FPGA. This compiled code can then be executed on the embedded processor.

The FM3TR modulator is implemented using the three-state state machine seen in Figure 12. In its initial state, the modulator presents the initial pulse values on the output of the modulator. This pulse serves as the reference point, allowing the phase difference between the initial pulse and subsequent pulses to be known. After the presentation of the initial pulse values, the modulator waits to be signaled to advance into the next state. In a production implementation of this application, this signal would be generated by some entity that is controlling then processing for the entire radio. Since this entity does not

exist in our implementations, this signal value is mapped to a specific location in BlockRAMs. The BlockRAMs location value is initialized at the time of configuration to the value that will initiate the state change. This initialization will cause the modulator to advance into the wait state immediately following the presentation of the initial pulse value.

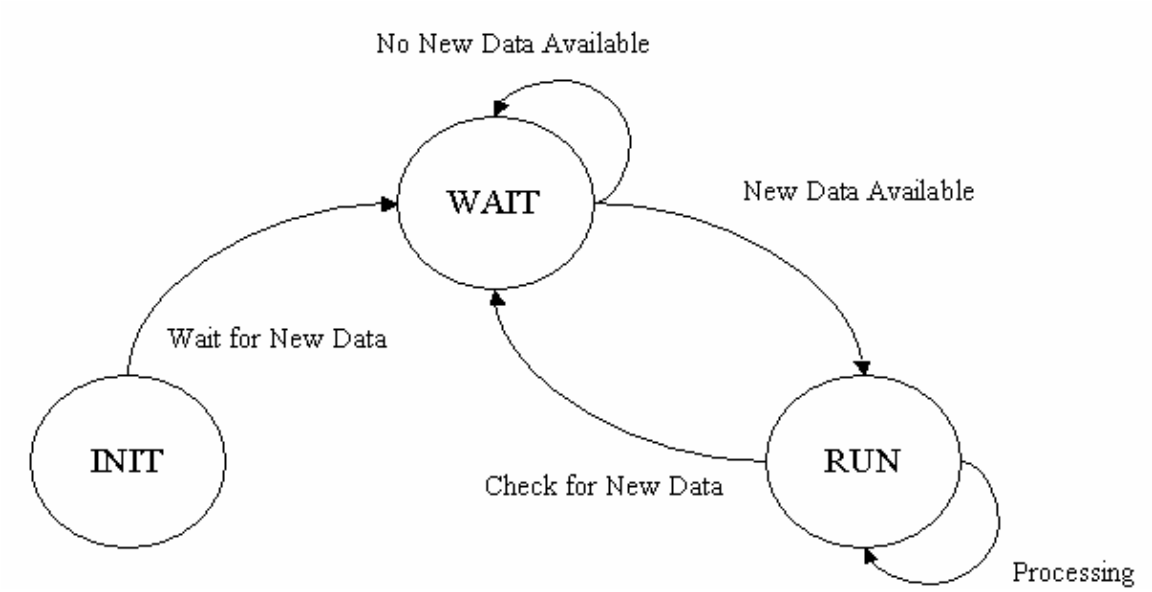

Figure 12: FM3TR Modulator State Machine

While in the wait state, the modulator waits for input samples to become available. Once this occurs, the modulator advances into its run state.

Upon entering its run state, the modulator begins modulating the input samples. Two samples are removed from FIFO 0 once every $2T_b$ intervals. Input samples removed from FIFO 0 during even multiples of T_b are used in the creation of the in-phase

component of the output signal. Input samples removed from the FIFO during odd multiples of T_b are used in the creation of the quadrature component of the output signal.

Each sample value is mapped to a positive or negative pulse. The sign of the pulse depends on whether the current input sample value causes a positive or negative phase change on the output of the modulator. The transmission of a single pulse requires that 9 output sample values be communicated through the appropriate modulator output in a time period equal to $2T_b$, or 4 samples per symbol interval.

After processing of the sample window is complete, the modulator returns to its wait state where it waits for new data to become available. This process repeats itself until the user terminates the application.

3.7 The Digital Up-Converter Implementation

The implementation of the Digital Up-Converter is provided by v1.4 of the Digital Up-Converter IP Core from Xilinx. This core is made available through v6.3 of Xilinx's System Generator (SysGen) software. The SysGen software allows the DUC to be parameterized using MathWork's Simulink Environment. The core is then translated into a VHDL description by the SysGen software. All parameters that are set in the Simulink environment are maintained through the translation process. Once translation is complete, the core is then instantiated in the design.

Symbol values are read out of FIFO 1 and presented to the in-phase and quadrature inputs of the DUC for processing. When processing is complete, qualified values of the passband signal will appear on the output of the DUC on successive clock cycles. The values that appear on the output of the DUC are pushed into FIFO 2.

3.8 Experiments

Section 3.7 presented the reader with a logical explanation of our FM3TR Waveform Application. Section 3.8 explains the application's implementations at the physical level. Factors that effect the physical level implementation include simplifications that can be made as a result of this being a controlled experiment and the FPGA's board environments.

3.8.1 Simplifications

Implementations of the FM3TR Waveform Application are intended to serve as a vehicle for the study of the various interfaces that enable communication between the PowerPC and the FPGA's reconfigurable elements. As a result, simplifications were made to reduce the complexity of our implementations' design. Although they are valid for the purpose of our investigations, the simplifications could not be made if any of our implementations were to be deployed in production systems.

Any implementation that is intended for use in a SDR environment should be capable of stream based processing. In its current form, our application is not capable of processing in stream-based environments. Implementations that implement applications that are representative of the FM3TR Wave Application organize the incoming data stream into successive windows of samples. These windows are then operated on sequentially by a single processor, or concurrently by multiple processors. The windows of data are then organized back into a single stream of data once processing is complete. It should be noted that in order for this type of processing to work, the processing entities must be able to accept data from their input and present data to their output at rates consistent with those in the application specifications.

In their current forms, our implementations are only capable of processing the initial window of the stream based processing technique described by the previous paragraph. It was determined that the processing of the initial window was sufficient for our study. The ability to process successive sample windows after the initial sample window would need to be incorporated into our implementations before they could be deployed in production systems.

The FIFOs used in our implementations behave differently at the physical level then those described in Section 3.4. The FIFO implementations described in Section 3.4 maintain pointers to the newest and the oldest elements in the FIFO. These pointers are called PTR_LAST, and PTR_FIRST respectively. When the consuming entity asks the FIFO for the next element, the FIFO will present the element that is pointed to by

PTR_LAST. PTR_LAST is then incremented and now points to the second oldest point in the FIFO. Similarly, when the producing entity wishes to add an element to the FIFO the element is stored in the location pointed to by PTR_NEW. PTR_NEW is then incremented. When PTR_NEW= PTR_LAST, the EMPTY flag is asserted and the FIFO is considered EMPTY. When PTR_NEW = PTR_LAST – 1 the FULL flag is asserted and the FIFO is considered EMPTY. If either pointer equals the last address managed by the BRAM providing the FIFO implementation, the next incrementation of that pointer will cause the pointer to wrap around to the first address of the BRAM. The management of these pointers and status flags is performed by the FIFO's control logic.

In our implementations, the FIFOs' control logic is different. The window of samples that are operated on fit entirely in one BRAM module. As a result, the control logic needed to implement the circular buffer and generate the status flags does not exist.

3.8.2 Implementation Platform

This application was implemented using a ML310 Development Board from Xilinx. The ML310 contains a single Virtex-II Pro P30 FPGA device. The limitations of the ML310 that were most influential in the design of our application was the lack of board I/O, inability to create a shared memory space between the FPGA and a host computer, and the existence of a single non-programmable oscillator.

In a production version of this implementation, data would need to be streamed into and out of the FPGA. Without the purchasing of additional equipment, the ML310 development board cannot support input or output from any external I/O source or sink. As a result, any data that is to be processed must be loaded into the FPGA during the time of configuration. Once the data has been processed, it is stored in BlockRAMs. The data can then be transferred to a PC for verification through a RS232 connection. These limitations make it difficult to support the stream-based environment that is required for FM3TR Waveform Processing. As a result, our implementation can only process a relatively small amount of data at a time.

The inability to create a shared memory space between a host computer and the FPGA can make development difficult. The presence of a shared memory space between a host machine and the FPGA would facilitate the transfer of processed data back to the host machine for verification. Since this shared memory space does not exist the transfer is performed using the FPGA's Universal Asynchronous Receiver Transmitter (UART). Using the UART requires use of the Virtex-II Pro's second PowerPC processor, a second PLB instance, additional BlockRAMs, and an OPB instance. Although the use of these additional resources may have negatively impacted the performance of this application, they are necessary in order to overcome the limitations of the ML310 development board.

The ML310 Development Board contains a single 100 MHz on board oscillator. This oscillator is not programmable. Since the Digital Up-Converter Core is dependent on the frequency of the clock, the existence of a single non-programmable clock heavily

influenced our implementations. The output rate of the DUC's output data is the product of the output rates of its Pulse Shaping Filter, Compensation Filter, and Cascaded-Integrator Comb Filter. The Pulse Shaping Filter and the Compensation Filter each increase the data rate of the DUC by a factor of 2. The Cascaded-Integrator Comb Filter increases the data rate further by a factor of R, where R can range from 4 to 1448 depending on the requirements of the implementation. Thus the total data rate increase of the DUC is equal to 4R.

A requirement of the DUC is that its clock pin must be driven at a rate that is greater than or equal to 4R. Since the frequency of our clock is fixed, the range of values that R can assume is limited. In our implementations R is set to 4, making the total rate change of the DUC equal to 16.

The rate at which data enters the DUC is a fraction of the rate of date leaving the DUC. Since FIFO 1 is responsible for presenting data to the DUC, its interface to the DUC must be clocked at the rate at which the DUC is accepting data. Since only one clock exists, a second clock had to be synthesized. This was done using one of the Virtex-II Pro's Digital Clock Managers (DCM). The new clock oscillates at one-sixteenth the rate of the ML310's onboard oscillator's .By clocking FIFO 1's interface to the DUC at 6.25 MHz, we were able to deliver the modulated samples to the DUC at the appropriate rate.

3.8.3 Objectives

The goal of any real time application implementation is the provision of interfaces between data intensive processing modules that supply seemingly infinite bandwidth and consume no device resources. Of course this goal is unachievable. The reality is that higher bandwidth interfaces consume more device resources than lower bandwidth interfaces. An interface design that enables communication between two data intensive processing modules is a trade off between bandwidth, resource consumption, and development time.

The purpose of our application is to serve as a vehicle for the study of the interfaces that enable communication between the PowerPC and the surrounding FPGA logic, as well as factors that can affect the performance of these interfaces. Earlier sections describe the interfaces in terms of their functionality and the types of data that they are capable of moving. Using our application, we created an experimental setup that allows us to observe the performance of each interface, and the effects that the interfaces have on each other. This experimental setup consists of multiple implementations of the same application. Each instance differs in the mechanisms that are used to transfer data between the processing components (i.e. the modulator and the digital-up-converter). .

The first step is identifying the areas in our design where efficient communication between the PowerPC and the FPGA's logic is critical. In our application, there is at least one interface that exchanges data between the PowerPC and the surrounding logic. This

interface is responsible for presenting data to the digital up-converter input at an appropriate rate. Depending on the implementation, the potential exists for other interfaces to play a critical role in the performance of this type of application.

The data transfer between the modulator and the DUC occurs when the DUC pulls modulated signal values from FIFO 1, as described in earlier sections. The BlockRAMs that implement FIFO 1 have the ability to be connected to either the PowerPC405's OCM or PLB interface. BlockRAMs that are connected to the PowerPC405 through the PLB interface can reside on either the PLB or the OPB. Our experiment includes three different implementations that implement FIFO 1 using BlockRAMs that have been connected to the processor through the DSOCM, PLB, or OPB. With these implementations, we are able to compare these interfaces in terms of available bandwidth, resource consumption, and development complexity.

The interface mechanism is not the only design choice that affects the performance of the data transfer between the modulator and the DUC. In addition to having access to a sufficient amount of memory for the storage of both the modulated and unmodulated data, the PowerPC405 must also be able to store the instructions that are performing the modulation. Furthermore, the processor requires use of a stack and/or a heap to maintain the state of the processor, as well as to accommodate potential memory allocations. The maintenance of these data structures results in the consumption of additional memory beyond what is needed for the storage of both data and instruction. The allocation of memory for instruction storage and the maintenance of the stack and/or

heap can impact the performance of the interface that is communicating signal values to FIFO 1. This performance impact occurs when the modulated data is communicated to FIFO 1 through an interface that is being used for instruction fetches, stack/heap management, or both. This scenario exists when the PLB interface has been configured to communicate instruction and data over the same bus. If the situation were to arise where both instruction and data need to be communicated at the same time, the processor would stall. This stall results from the fact the either data or instruction can be communicated through the shared interface at any given instance in time, but not both. As a result, the PLB arbiter executes requests for transfers of both instructions and data sequentially. This forces the processor to wait until both requests are executed.

We now describe the implementations that have been developed to enable the comparison of the various interfaces to the PowerPC405. Each implements the FM3TR Waveform Processing Application. The differences in the implementations are the mechanisms that are used to communicate information between the processor and the FPGA's reprogrammable resources.

3.8.4 Implementation Class 1

Implementation class 1 uses the processor's OCM interface to communicate modulated data to BlockRAM. Data samples and processor instructions are loaded into the appropriate memory at the time of configuration. The data samples are read from

DSOCM by the PowerPC405 and modulated according to the instructions being stored in instruction-side memory. After modulation, the data is written back into the DSOCM. The DUC removes the modulated signal values from Port B of the DSOCM beginning with the first value written to DSOCM and proceeding in succession from there on. The modulated signal is processed by the DUC and pushed into FIFO 2.

Implementation class 1 contains three implementations, a, b and c, seen in Figures 13 and 14. Implementations a, b, and c use the same memory to store the applications' instructions and the modulated data. The difference between the three implementations is the storage locations of the program data that is responsible for maintaining the processor's stack and heap. Implementation subclass 1.b stores the stack/heap data in OCM BRAM. Implementation subclass 1.a store the stack/heap data in PLB BRAM. Implementation 1.c is identical to 1.a except that the cache is enabled in 1.c, but not in 1.a.

A performance analysis of implementation subclass 1.b will demonstrate the performance that one can expect from an application that runs completely out of OCM BRAM. A performance analysis of implementation class subclass 1.a and 1.c will demonstrate how the relocation of the program data can affect the performance of an application.

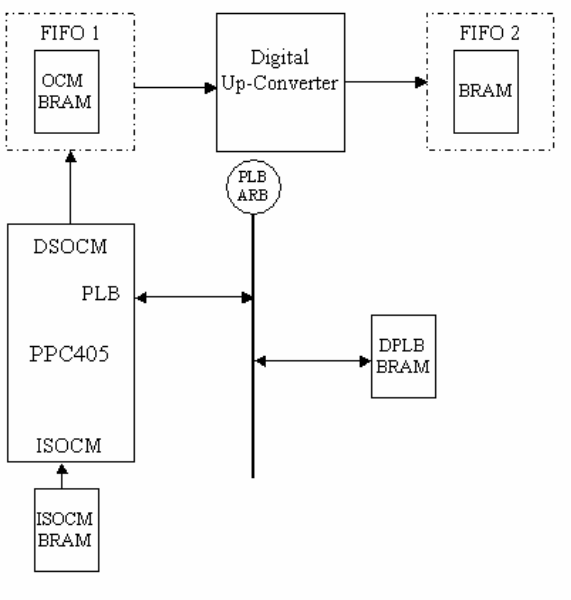

Figure 13 : Architecture Overview for Implementation Subclass 1.a and 1.c

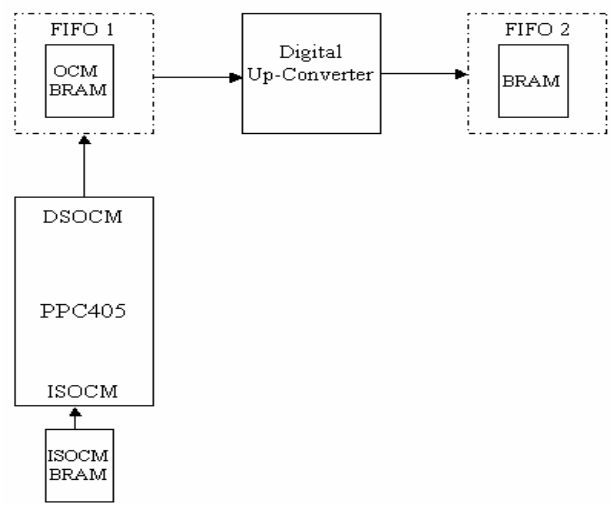

Figure 14 : Architecture Overview for Implementation Subclass 1.b

3.8.5 Implementation Class 2

Implementation class 2 uses the PLB to transfer data between the modulator and FIFO 1. Data samples and instructions are loaded into the appropriate on-chip memory at the time of configuration. The modulator removes the sample values from DSOCM and performs the modulation. The sample values of the modulated signal are pushed into FIFO 1 through the PLB interface. The DUC removes the modulated signal values from FIFO 1 beginning with the first value written and proceeds in succession from there on. The modulated signal is processed by the DUC and pushed into FIFO 2.

Implementation class 2 contains 6 implementations. Implementation 2.a, shown in Figure 15, stores its program data in DSOCM BRAM. Implementation 2.b, shown in Figure 16, stores its program data in PLB BRAM. Instructions and program data for both implementations are stored in ISOCM BRAM and PLB BRAM respectively.

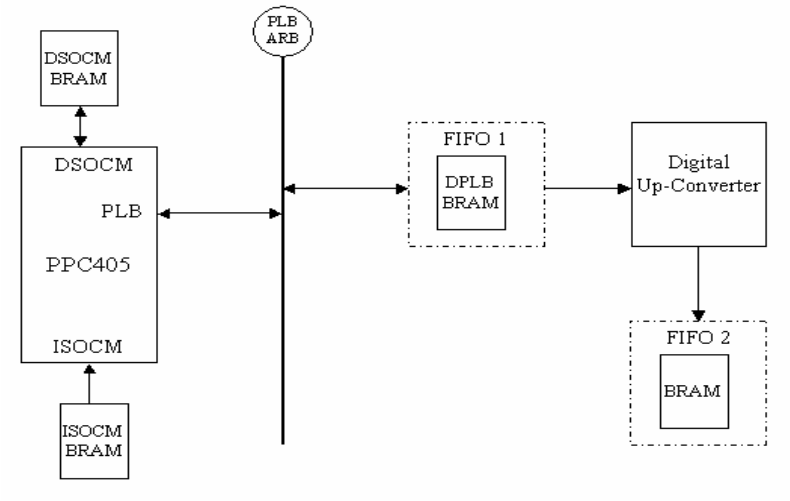

Figure 15 : Architecture Overview for Subclasses 2.a

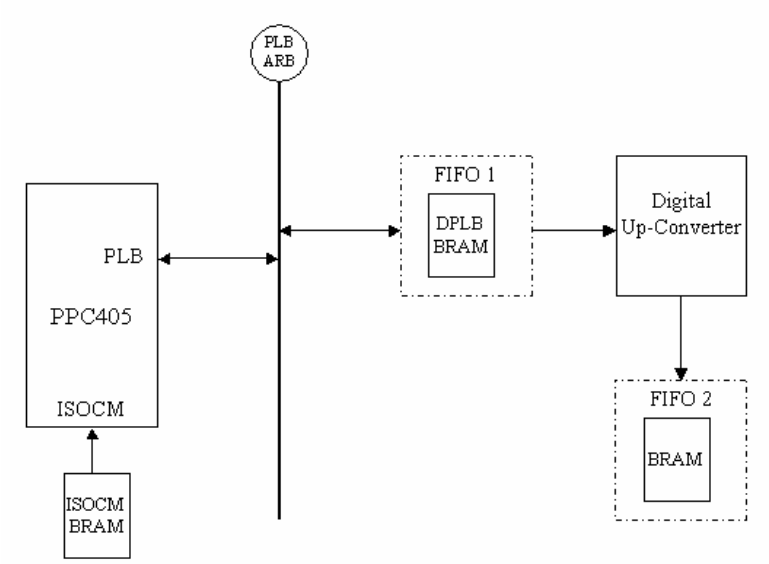

Figure 16 : Architecture Overview for Subclass 2.b

Implementations 2.c, 2.d, 2.e, and 2.f, seen in Figure 17, demonstrate the performance consequences of storing both data and instruction in PLB BRAM. The sharing of a single PLB between multiple memories will cause contention if the processor

attempts to communicate data and instruction over the PLB at the same time. This contention will need to be resolved by the PLB arbiter before either communication can proceed. Use of the processor's cache can be used to speed up the performance of these implementations. By enabling the cache, communication over the PLB can be reduced significantly. This communication reduction is dependent on the locality of the data stored in PLB BRAM.

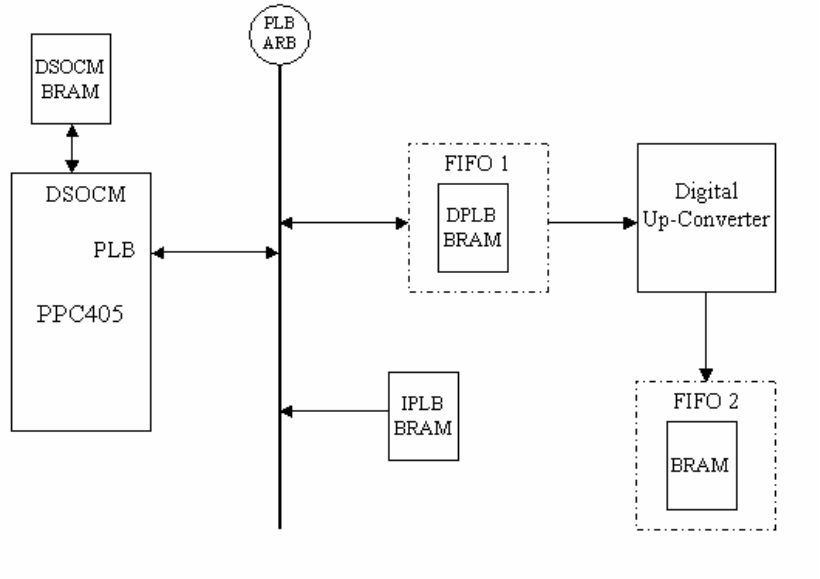

Figure 17 : Architecture Overview For Subclasses 2.c, 2.d, 2.e, and 2.f

3.8.6 Implementation Class 3

Implementation class 3 demonstrates the performance of the application when the modulated data is communicated to memory over the OPB. Data samples and instructions are loaded into the appropriate on-chip memory at the time of configuration. The

modulator removes the sample values from either PLB or ISOC BRAM and performs the modulation. The sample values of the modulated signal are pushed into FIFO 1 through the PLB interface. The PLB/OPB recognizes that the destination address of the modulated data and generates the appropriate transaction on the OPB. The DUC removes the modulated signal values from FIFO 1 beginning with the first value written and proceeds in succession from there on. The modulated signal is processed by the DUC and pushed into FIFO 2.

Implementation 3.a, shown in Figure 18, stores the processors instructions in ISOCM BRAM, the program data in DSOCM BRAM, and the modulated data in OPB BRAM.

Implementations 3.b and 3.c, shown in Figure 19, demonstrates the consequences of sharing the PLB, while communicating modulated data over the OPB. In Implementation 3.a, all of the memories are connected to separate interfaces, allowing for concurrent memory transfer. This is the best possible scenario given the implementation specification requiring the modulated data to be communicated over the OPB. The moving of instruction storage to the PLB in Implementation 3.b will create contention for the PLB bus. This contention is resolved in Implementation 3.c by enabling the i-cache.

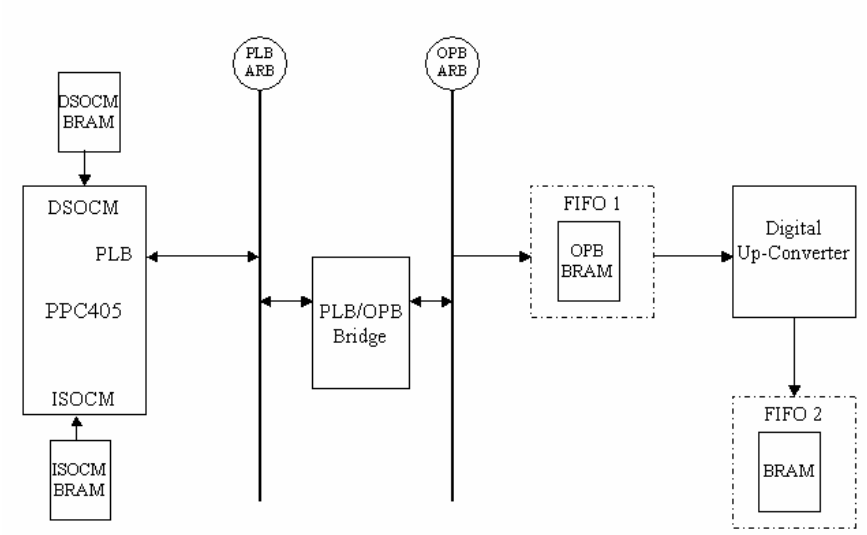

Figure 18 : Architecture Overview of Implementation 3.a

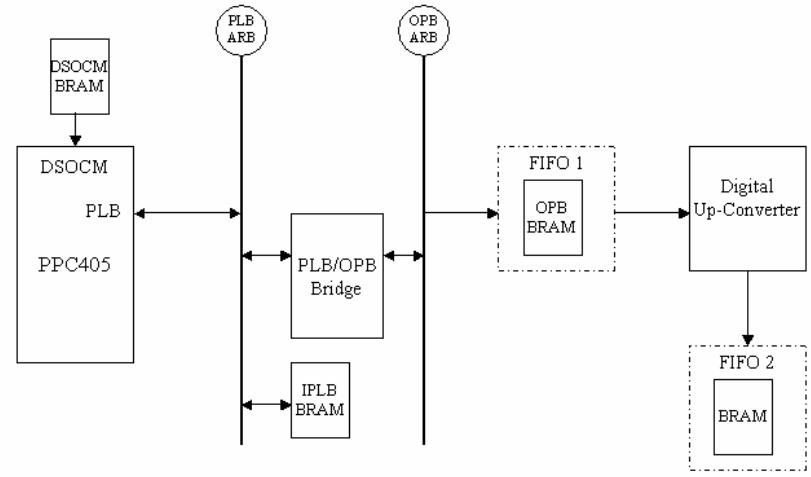

Figure 19 : Architecture Overview of Implementation 3.b and 3.c

3.9 Summary

This chapter begins with an explanation of FM3TR modulation and digital up-conversion. We then give an overview of our FM3TR application including the software implementation of our FM3TR modulator and the hardware implementation of the digital

up-converter. Finally we describe several implementations of our FM3TR application used to study the various interfaces that enable communication between the PowerPC405 and the surrounding FPGA fabric on a Xilinx Virtex-II Pro.

Chapter 4

Experimental Results

In this chapter we describe the results of this investigation. We begin by describing the technique that was used to measure the performance of our implementations. Next we present the performance results for each implementation that was discussed in Chapter 3. Finally we analyze the similarities and differences between the performance results for each implementation.

.

4.1 The Programmable Interval Timer (PIT)

Performance measurements are made for each implementation using the PowerPC405's Programmable Interval Time. This timer is controlled by various software instructions that are executed on the PowerPC. Physically, the timer is a 32-bit register that is incremented synchronously with the PowerPC's clock. A software task can be timed by initializing the timer with a value of 0, executing the software task, and then reading the value of the timer. This value will indicate the number of clock cycles that have elapsed since the timer was initialized to a zero value. The product of the timer's values and clock period is the execution time of the software task.

An implementation's performance is measured in terms of the time it takes for the implementation's modulation code to execute on the PowerPC processor. The computations that are performed within the modulation code are identical among the implementations. The only difference is the technique used to enable communications between the processor and the surrounding FPGA fabric.

The modulation code's execution time is measured by resetting the Programmable Interval Timer (PIT) to 0, executing the modulation code, and then reading the value of the PIT. The value of the PIT after the modulation code executes indicates the number of clock cycles that elapsed during the modulation.

Using the PIT to measure the execution time of the implementations effects the actual timing measurement. The discrepancy is due to the clock cycles that are consumed in the reading of the PIT register value. The number of clock cycles consumed by this operation is on the order of several cycles. Since the application's cycle consumption is on the order of several thousand clock cycles, the error introduced by this timing method is negligible. Furthermore, since each execution is timed using the same technique, a relative comparison of all the implementations need not consider the performance penalties of the timing technique provided that the penalties incurred in each execution are identical.

4.2 Implementation Results

Once the clock cycle consumption of each implementation is measured, they are compared to one another. Since the computation in each implementation is identical, differences in execution cycles are attributed to the performance differences of the PowerPC's interfaces. By comparing the execution cycles that are consumed by each implementation, we are able to determine which interfacing techniques work best. Tables 8 and 9 show the results that were obtained from each of the implementations. Table 8 shows the how each implementation's memory is organized. Table 9 gives the performance results for the implementations of Table 8. It should be note that since the OCM interface is a non-cacheable interface, the enabling or disabling of the cache effects the PLB interface only.

The values in the column titled *Computation Cycles* are obtained by measuring the number of clock cycles consumed by the execution of a modulation code version that does not save the result of each computation to memory. The values in the column titled *Total Cycles* are obtained by measuring the number of clock cycles consumed by the execution of a modulation code version that saves the result of each computation to memory. The values of the *Communication Cycles* are the difference between the values in the *Total Cycles* column and the *Computation Cycles* column. It should be noted that the values in the *Communication Cycles* column do not reflect the communication costs associated with instruction fetches. The values indicate the number of clock cycles

required to communicate the modulated data to memory. Instruction transfers are required for computation and thus are reflected by the values in the *Computational Cycles* column. Differences in the number of computational cycles between implementations are due to differences in instruction fetch times.

Table 9 : Implementation Configurations

Implementation	Instruction Location	Stack/Heap	Application Data	I-Cache	D-Cache
1.a	ISOCM	DPLB BRAM	DSOCM	N/A	DISABLED
1.b	ISOCM	DSOCM	DSOCM	N/A	N/A
1.c	ISOCM	DPLB BRAM	DSOCM	N/A	ENABLED
2.a	ISOCM	DSOCM	DPLB BRAM	N/A	ENABLED
2.b	ISOCM	DPLB BRAM	DPLB BRAM	N/A	ENABLED
2.c	IPLB BRAM	DSOCM	DPLB BRAM	DISABLED	DISABLED
2.d	IPLB BRAM	DSOCM	DPLB BRAM	DISABLED	ENABLED
2.e	IPLB BRAM	DSOCM	DPLB BRAM	ENABLED	DISABLED
2.f	IPLB BRAM	DSOCM	DPLB BRAM	ENABLED	ENABLED
3.a	ISOCM	DSOCM	OPB BRAM	DISABLED	DISABLED
3.b	IPLB BRAM	DSOCM	OPB BRAM	DISABLED	DISABLED
3.c	IPLB BRAM	DSOCM	OPB BRAM	ENABLED	DISABLED

Table 7 : Implementation Performance Results

Implementation	Computation Cycles	Communication Cycles	Total Cycles
1.a	7828	20566	28394
1.b	7818	13468	21286
1.c	7816	13462	21278
2.a	8430	17752	26182
2.b	8430	27716	36146
2.c	16638	36887	53525
2.d	15561	40400	55961
2.e	6519	17659	24178
2.f	6519	17742	24261
3.a	7779	29225	37004
3.b	17891	46238	64129
3.c	16766	21312	38078

4.3 Analysis of Results

4.3.1 The OCM Interface Analysis

The results show that communicating to the fabric through the processors On-Chip Memory (OCM) interfaces delivers the highest performance. This is illustrated by implementation 1.b, which communicates both instruction and data through the processor's OCM interfaces. This implementation executes the entire processor application in under 23000 clock cycles. This is comparable to implementation 1.c, which executes in approximately the same amount of time using PLB BRAM to maintain the processor's stack and heap. However, the performance delivered by 1.c is dependent on the processor's ability to use its internal cache. When the cache is disabled, the processor

must use the fabric of the FPGA to communicate with PLB BRAM. The need for this communication is the reason why implementation 1.a's performance is poor when compared to 1.b and 1.c. When compared to other implementations of the same class, the results of implementations 2.a and 3.a are consistent with the results described in the analysis of Implementation Class 1. This leads us to conclude that the OCM interface provides the fastest communications between the processor and the fabric.

It is interesting to note that the communication performance of a cache enabled PLB interface is comparable to the performance of the OCM interface. This observation may depend on the fact that the cache is capable of storing all of the data communicated through the PLB interface. If the size of the stored data is larger than the cache, the need to fetch data from BRAM may widen the performance gap between the OCM and the cache enabled PLB interface. Furthermore, if the locality of the data that is being communicated through the PLB interface is poor, the cache will not be of any use.

4.3.2 The PLB Interface Analysis

4.3.2.1 The Processor Local Bus

The results indicate that using the PowerPC's cache appropriately can drastically improve the performance of implementations that use the Processor Local Bus (PLB) interface to communicate to the FPGA's fabric. Implementations that illustrate this fact

include 2.c, 2.d, 2.e, 2.f, 3.b, and 3.c. When we compare implementations 2.c, 2.d, 2.e, and 2.f to each other we see that the enabling of the I-Cache results in a reduction in execution time by more than 50%. Before the enabling of the I-Cache in 2.e and 2.f, the IPLB and the DPLB were fighting for control of the PLB. Enabling the I-Cache allows for the instructions to be fetched almost exclusively form the processor cache. This increases the performance of the application. Furthermore, the investigations in [21] show that using a cache enabled PLB interface to communicate data to the fabric consumes 5 times less power than an equivalent implementation that uses a PLB interface with the cache disabled, and 3 times less power than an implementation that uses the OCM interface. Though this analysis was done using a slightly different application, it is evident that using the cache to reduce communication over the FPGA fabric can result in a substantial power savings.

Performance improvements that result from the I-Cache are a consequence of the instruction's temporal locality. Temporal locality is a consequence of the frequent execution of one instruction within a given window of instruction code. The modulation code executes a loop that maps each input sample to a pulse. Since the instructions of successive loop iterations are identical, the execution frequency of the instructions that exist within the loop are high. Furthermore, all of the instructions that execute within the loop can fit inside the I-Cache. If the loop code did not fit entirely in the cache, each loop iteration will result in several cache misses. These cache misses cause the application's

performance to suffer. In our implementations the PowerPC can take advantage of temporal locality, but not spatial locality.

Spatial locality is a consequence of the fact that instructions are stored contiguously in memory. A cache can take advantage of spatial locality by loading the values of several contiguous memory locations into the cache each time a cache miss occurs. Since instructions are stored contiguously, the probability of the next instruction fetch resulting in a cache hit is high. Although the processors instructions exhibit spatial locality, the processor is configured by default to only load one word per cache miss. However, the user can change this cache policy so that up to eight words can be loaded into the cache each time the cache misses. We chose to allow the cache to follow its default policy of loading one word per cache miss.

Implementations 2.d and 2.f are instances where use of the PowerPC's cache can hurt. When compared to implementations 2.c and 2.e, it can be seen the enabling of the data-side cache hurts the performance of the implementations. This drop in performance is a consequence of the processor's efforts to maintain cache coherency. An input sample, and the values of the modulated signal produced as a result of the input sample, plays no part in future computations. As a result, it can be said that the application's data exhibits poor temporal locality. This poor locality makes the performance costs from having to write data to the cache and to the BRAM greater than any performance gain that results from the enabling of the cache. It should be noted that the application's data does exhibit

spatial locality. However, our implementations configure the PowerPC to take advantage of temporal locality, but not spatial locality.

In addition to data locality, another factor that makes maintaining cache coherency difficult is the fact that PLB BRAM can be shared between the processor and the logic. In instances where the processor communicates data to logic via PLB BRAM, the possibility exists that the data in the PLB BRAM is inconsistent with the data in the cache. A similar scenario exists when logic communicates data to the processor via the PLB BRAM. When logic writes data into PLB BRAM it may create an inconsistency with the values that are currently stored in the processors' cache. To resolve these inconsistencies and still make use of the D-Cache, the processor's cache should be flushed before the communication between the processor and the logic occurs.

Our experiments show that the D-Cache should be disabled in streaming applications that used the PLB BRAM to store application data. Configurations that use PLB BRAM to store the application's stack/heap can benefit from enabling the D-Cache. This can be seen in a comparison of implementations 1.a and 1.c. Both implementations are identical except that 1.a executes the application with the D-Cache disabled, while 1.c executes the application with the cache enabled. It can be seen that the enabling of the D-Cache reduces the execution time by approximately 7000 cycles. This suggests that spatial locality of the program data is sufficient to make its placement in cacheable memory beneficial to the performance of the application.

4.3.2.2 The On-Chip Peripheral Bus

Our experiments show that the On-Chip Peripheral Bus (OPB) is capable of offering reasonable performance when there are no devices that have to fight for control of the PLB. Implementations 3.a and 3.c indicate scenarios where the OPB offers performance that is comparable to the PLB. However, neither implementation execution suffers from bus contention. In 3.a, all data except for the application's data is store in OCM BRAM. As a result, traffic appearing on the PLB and the OPB is always destined for OPB BRAM. In implementation 3.c, the instruction data has been moved to PLB BRAM. However, implementation 3.c's enabling of the cache nearly eliminates any traffic on the PLB related to the communication of processor instructions. This is a consequence of the fact that all of the processor's instructions are capable of fitting inside the PowerPC's cache. Since the I-Cache is capable of retaining all of the processor's instructions, the need to fetch instructions over the PLB does not exist.

Implementation 3.b is identical to 3.c except for the instruction cache in 3.b being disabled. This disabling of the cache forces both the processor instruction and application's data to be fetched through the PLB interface of the processor. This creates contention on the PLB, drastically lowering the performance of the design.

Using the results of the execution of Implementation Class 3, we conclude that the performance OPB is reasonable provided that contention for the PLB is minimized. In

general , the PLB should be used to communicate data to and from the processor. Devices that only support OPB interfaces can be used with the OPB and still be expected to deliver reasonable performance. However, as contention over the buses increases, the performance of the application will drop considerably.

4.4 Summary

In this chapter we presented the performance results for all of our implementations. It can be seen that the OCM interfaces provide the fastest communication with the fabric. The PLB provides comparable performance to the OCM when the cache is enabled and the locality of the data is good. We also saw the using the PLB to service multiple peripherals lowers the performance of the interface. This performance hit can be reduced through use of the cache. Finally, we saw that the OPB delivers reasonable performance when the traffic on the PLB bus is low.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

The emergence of FPGA architectures that contain embedded processor leaves many designers with questions of how to use the various memories that exist within an embedded processor system efficiently. In this investigation we used a FM3TR Waveform application to provide a vehicle for the use of investigating the various interfaces between the FPGA and the processor. The results can be used to guide future applications that have similar data attributes.

This investigation shows that the performance of an application that uses the Virtex-II Pro's embedded processor is affected by the types of interfaces that are chosen to communicate data between the processor and the fabric of the FPGA. It was seen that the OCM and PLB are both capable of providing high performance if used appropriately. Data that requires a high performance interface, but does not exhibit good locality should be communicated through the OCM interface. Data that exhibits good locality should be communicated through a cache-enabled PLB interface. Application designers that have reasons for communicating data with poor locality over the PLB should consider disabling the cache to eliminate the overhead associated with the maintenance of cache coherency. Communication over OPB delivers reasonable performance provided that the number of devices contending over the bus is low.

5.2 Future Work

The results of our experiments show that data locality can dramatically increase the performance of a given implementation. Our implementations are configured to load a single word each time a cache miss occurs. Since the modulator's input samples are stored in contiguous memory locations, loading multiple successive words into the cache each time a miss occurs increases the number of future memory accesses that result in cache hits. These performance benefits associated with spatial locality have not yet been investigated. Future implementations will investigate the performance benefits of configuring the PowerPC's cache to take advantage of spatially local data.

This investigation describes the performance of the PowerPC's interfaces in terms of the number of clock cycles required to communicate data from the processor to the surrounding FPGA fabric. In future studies of the PowerPC interfaces we hope to study the amount of FPGA resources that are consumed by the use of a specific interface. Such a study would enable application designers to decide if the performance benefits from using a particular interface outweigh the area costs.

We have seen that the application data's lack of temporal locality hurts the performance of the application when the cache is enabled. The application data for image processing applications exhibit good temporal locality. This is because a single input sample contributes to multiple calculations. To explore these benefits, we hope to develop an image processing application that makes use of the PowerPC processor.

Finally, the implementations presented in this investigation use the PowerPC to modulate data. Alternatively, the modulation could be performed entirely in logic. Such an implementation would allow us to compare a processor application with an equivalent logic implementation. This would help us determine scenarios in which the use of the processor application consumes less of the device's area than the equivalent logic implementation.

Bibliography

[1] AFRL. Description of the FM3TR Proposed Reference Waveform, August 2001. Last Accessed August 2005. http://www.sdrforum.org/MTGS /mtg_25_sep01 /01_i_0056_v0_00_fm3tr_97_09_10_01.pdf

[2] Gordon Brebner. Single-Chip Gigabit Mixed-Version IP Router on Virtex-II Pro. Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002.

[3] Gordon Brebner. Eccentric SoC Architectures as the Future Norm. Proceedings of the Euromicro Symposium on Digital Design, pages 2-9, Sept 2003.

[4] Gordon Brebner Eric Keller and Phil James-Roxby. Software Decelerators. 12th Annual IEEE Symposium on FCCM, pages 3-12, 2004.

[5] Kraig Lund. PLB vs. OCM Comparison Using The Packet Processor Software. The Xilinx Corporation, v1.1 edition, October 2004. Last Accessed July 2005. http://direct.xilinx.com/bvdocs/appnotes/xapp644.pdf.

[6] John G. Proakis and Masoud Salehi. Communication Systems Engineering, chapter 7. Prentice Hall, 2002.

[7] OCP International Partnership. Open Core Protocol Specification, v2.1 edition, 2005. Last Accessed May 2005. http://www.ocpip.org.

[8] Pierre A. Laurent. Exact and Approximate Construction of Digital Phase Modulation by Superposition of Amplitude Modulated Pulses. IEEE Transactions on Communications, 2(COM-34):150-160, February 1996.

[9] Punit Kalra. UltraController-II: Minimal Footprint Embedded Processing Engine. The Xilinx Corporation, v1.0.1 edition, February 2005. Last Accessed July 2005. http://direct.xilinx.com/bvdocs/appnotes/xapp575.pdf.

[10] Software Defined Radio Forum . FAQ . http://www.sdrforum.org/faq.html

[11] Steve Trynosky, Matt Dipaolo, Kraig Lund, Ryan Laity. PowerPC405 PPE Reference System Using Virtex-II Pro RocketIO Transceivers. The Xilinx Corporation, v1.1 edition, July 2003. Last Accessed July 2005. http://direct.xilinx.com/bvdocs

/appnotes/xapp669.pdf

[12] The Xilinx Corporation. Chipscope Pro Software and Cores User Guide, v6.3.1 edition, October 2004. Last Accessed July 2005. http://www.xilinx.com/ise /verification/chipscope_pro_sw_cores_6_3i_ug029.pdf.

[13] The Xilinx Corporation. Embedded System Tools Reference Manual, v3.0 edition, August 2004. Last Accessedf July 2005. http://www.xilinx.com/ise/ embedded/edk6_3docs/est_rm.pdf.

[14] The Xilinx Corporation. Digital Up Converter Product Specication, v1.4 edition, May 2005. Last Accessed July 2005. http://www.xilinx.com/bvdocs/ipcenter /data_sheet /DUC.pdf

[15] The Xilinx Corporation. PowerPC405 Block Reference Guide. v2.0 edition, July 2005. Last Accessed July 2005. http://direct.xilinx.com/bvdocs/userguides ug018.pdf

[16] The Xilinx Corporation. ML310 User Guide. v1.1.4 edition, January 2005, Last Accessed July 2005. http://www.xilinx.com/products/boards/ml310/current/pcb /sch/ug068.pdf

[17] The Xilinx Corporation. Virtex-II Pro and Virtex-II Pro X FPGA User Guide. v4.0 edition, March 2005. Last Accessed July 2005. http://direct.xilinx.com /bvdocs/ userguides/ug012.pdf .

[18] The Xilinx Corporation. Asysnchronous FIFO. v6.1 edition, November 2004. Last Accessed July 2005. http://www.xilinx.com/ipcenter/catalog/logicore/docs/ sync_fifo.pdf.

[19] The Xilinx Corporation. The Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. V4.3 edition, June 2005. Last Accessed July 2005. http://direct.xilinx.com/bvdocs/publications/ds083.pdf.

[20] Wipro Technologies. Software Defined Radio White Paper, August 2002. Last Accessed May 2005. http://www.broadcastpapers.com/broadband WiproSDRadio.pdf.

[21] Jingzhao Ou and Viktor K Prasanna. A Methodology for Energy Efficient Application Synthesis Using Platform FPGAs. Engineering of Reconfigurable Systems and Algorithms (ERSA), pages 280-283, 2004.