HARDWARE AND SOFTWARE FOR WINC2R COGNITIVE RADIO PLATFORM

BY SHALINI JAIN

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Wade Trappe

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2008

ABSTRACT OF THE THESIS

HARDWARE AND SOFTWARE FOR WINC2R COGNITIVE RADIO PLATFORM

by Shalini Jain

Thesis Director: Professor Wade Trappe

Emerging wireless technologies and standards require inter-operability between devices whose infrastructure has been built using different radio access technologies and which operate in different spectrum bands. In this thesis, we study the feasibility of building such a multi-protocol platform - the WINC2R platform. WINC2R is a platform for cognitive radio applications that has the agility needed for the per-packet protocol adaptation across the protocol layers and flexible enough to support the future evolution of the wireless communication protocols.

The WINC2R platform concept is based on the generic processing engines that each handles processing for the computationally intensive physical and MAC layer functions. They are augmented with a software programmable processor that can handle the differences between the standards and future changes of those functions. The microprocessor environment forms one of the processing engines and can support additional functions with a sufficiently low complexity without sacrificing the performance in terms of latency and throughput. The WINC2R architecture satisfies the requirements of low latency and fast context switching as required by the multi-layer protocol processing.

The WINC2R system consists of three distinct layers: the data layer, the interconnection layer and the control layer. The data layer handles all the processing engines

and its functionalities. Flexibility and programmability is achieved through the interconnections between the processing engines. In this thesis, we work with the control layer which consists of the underlying hardware that supports the control functions like handling interrupts from the processing engines, synchronizing the tasks, etc. The underlying hardware is customizable for system connectivity, Digital Signal Processing (DSP), and data processing applications which are required for the physical layer. The control functions are used to maintain the WINC2R system integrity and manage the wireless protocols. They also enable communication among the functional modules themselves. We examine the overall system and task processing flow. The interconnection layer deals with interconnecting the processing engines together which is handled by the Unit Control Module (UCM).

At the heart of every processing engine lies the UCM that assigns and schedules tasks to and from the functional blocks with the help of a centralized data structure. It is in charge of scheduling the tasks to the module that it is associated with, assigning the task, monitoring the task completion, and communicating with the other modules in the system for task sequencing. The UCM is analyzed, and we study the feasibility of various tasks occurring in time. We propose a timeline for the occurrence of data and control tasks in different scenarios and study the feasibility of these scenarios.

Acknowledgements

As it always happens, with the completion of a very important chapter in our lives, we think of all the people who helped us get to that point. For me too, the completion of my master's thesis marks the end of a very exciting and insightful experience. And I would like to thank everyone who made my experience so.

First and foremost, I would like to thank my advisor, Prof. Trappe. He accepted me the way I am and encouraged me at every point of my thesis. He took time to give me helpful suggestions and tips on a regular basis. It has been been a very pleasant experience working under his supervision.

I am deeply grateful to Mr. Khanh Le, a research staff member of the WINC2R team, for his constant support and patience. His domain of knowledge is immense and I thank him for giving me insightful tips whenever I got stuck on some problem. He has always been extremely helpful taking out time to help everyone on the team. I would like to thank Ms. Renu Rajnarayan for her guidance and understanding. I am grateful to Prof. Zoran Miljanic and Ivan Seskar for their support. I would like to acknowledge the Cognitive Radio Team for having worked very hard for almost a year and working together at every point.

I will always remember the good times with my friends - Tejaswy, Sumit, Ayesha, Tanuja, Ben and Abhishek and many more.

Dedication

To my family, for giving me the sense of right and wrong.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Current commodity radio devices and the infrastructure supporting them use different radio and higher layer communication protocols, e.g. 802.11a/b/g, 802.16, 3G, etc. They use different radio access technologies and operate in different spectrum bands. Standardization efforts are already in place to make these devices inter-operable with each other [1]. Support for new protocols like vehicular telemetry requires flexibility and agility for per-packet protocol selection at very high rates.

Achieving the desired inter-operability and flexibility with high data rate requirements is very challenging specially at the physical layer that is computationally intensive. High data rates provide very short time window for processing and protocol selection.

We need a radio that is "intelligent", which can change its transmission or reception parameters (such as power, frequency and modulation) according to the changing user or network requirements [2]. It should be flexible to support new protocols. It should be able to scan the unused spectrum and dynamically reuse them.

There have been several approaches in this direction as described in the next section.

1.1 Current Cognitive Radio Approaches

To date, the approaches considered for supporting inter-operability can be classified as follows:

- *Multi-modal hardware based platforms*,
- *Portable software platform*,
- *Reconfigurable platforms*,

- *SoC (System on Chip) Programmable Radio Processors*

Multi-modal hardware based platforms: These are the platforms that implement multiple physical and higher layer protocols in distinct sub-modules that are implemented in hardware. To select a particular protocol, a corresponding sub-module can be activated. Thus, the support for multiple protocols is restricted to the selection of a sub-module. They lack flexibility as there is a restriction as to how many protocols can be implemented in hardware at a time due to the increase in complexity. An example radio using such a platform would be multi-mode cell phones [3]. These platforms are useful where the functions do not need to change in future.

Portable software platforms: These platforms have complete software implementation for all the physical and MAC layer communication protocols above the RF and A/D. The protocols run on an operating system (usually Linux) and hence, are portable across different platforms. They use hardware accelerators for computationally intensive functions to improve the processing time. The programming is done at a high level, in a language like C++ or Java. Some examples of such platforms are Vanu Inc. [4] and GNU Radio [5] public domain platforms. They do not provide sufficient processing speeds for emerging WLAN applications based on OFDM based physical layer and IP packet based protocols for the higher layers. These platforms are useful as infrastructure devices like base stations.

Reconfigurable platforms: These platforms implement the physical and MAC layer communication protocols using tightly coupled combination of FPGA, DSP processor(s), and other embedded RISC processors. This type of platform supports a limited set of protocols at a time. To support a new protocol, embedded system programming has to be done and a new FPGA bitmap and DSP code needs to be generated. Programming is very tough and this does not provide the developer with flexibility. The underlying FPGA devices, DSP processors, and clock rates limit their performance. Examples of such a platform are Rice University Warp platform [6] and JTRS development kit designed by ISR Technologies from Canada in partnership with Xilinx [7]. These platforms are useful for research and development work.

SoC Programmable Radio Processors: These platform consist of semiconductor devices that are completely software programmable for the physical and MAC layer communication functions. Typically, the platform architecture ia based on an array of special purpose processors, and also use hardware accelerators for the PHY layer functionalities like FFT, Viterbi decoding, etc. Pipelining is generally employed to improve the over-all system performance. The number of processors in the array varies significantly: the Sandbridge SB3000 has 4 processors while the picoChip PC20x family has 248 processors [8]. The processing speed is limited by the number of processors and the hardware accelerators used. The software is not portable on different platforms or even the different generations of the same platform. The massive parallel processing is not practical for hand held devices because of the cost and power limitations.

The gap between processing complexity of high speed wireless protocols and processing power of SoC devices is increasing [9]. In this thesis, we focus on the challenge of creating a radio device which provides inter-operability and flexibility while maintaining the power and capabilities of a traditional ASIC (Application Specific Integrated Circuit) approach.

1.2 WINC2R Platform Concept

WiNC2R [10] is based on the concept of programmable platform which allows flexibility for per-packet protocol selection at high data rates. It resolves the requirement of the protocols for speed and flexibility. This is a different approach than the traditional ASIC or SoC approach of building a radio.

The ***ASIC*** approach to radio design addresses application requirements using a set of functional blocks with dedicated, point-to-point, connections between them. The functional blocks are configurable as required by the protocol. The ASIC approach provides the required speed for the application and is optimized for complexity, but provides no flexibility beyond the requirements of the particular application. An example ASIC architecture for an 802.11a receiver is shown in the diagram of Figure 1.1 [11].

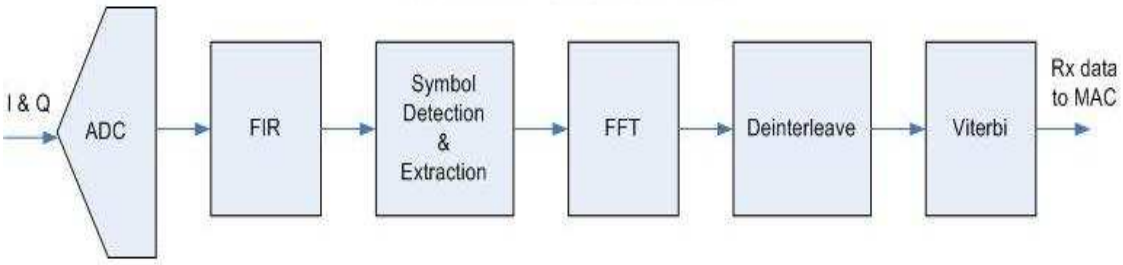


Figure 1.1: 802.11a Receiver Block Diagram (ASIC Approach)

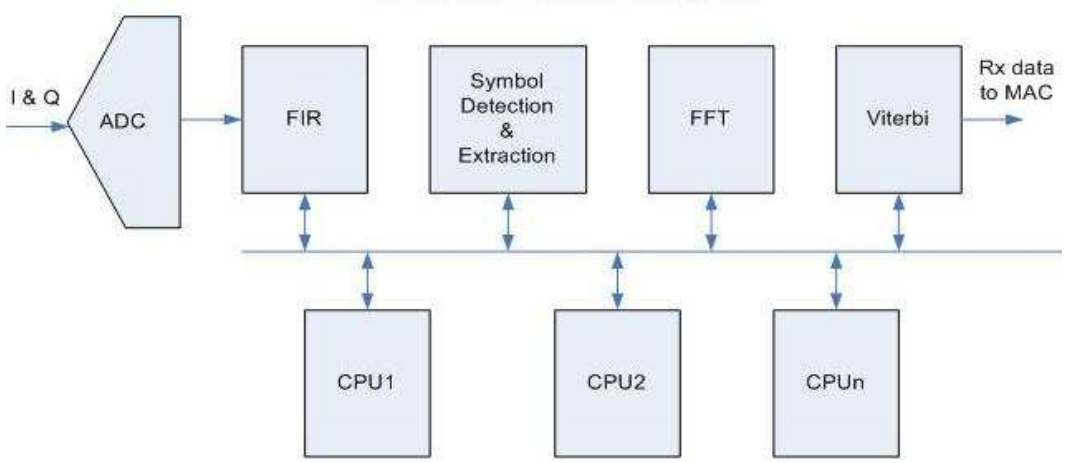


Figure 1.2: 802.11a Receiver Block Diagram (SOC Approach)

On the other hand, radios built using an ***SoC*** approach are based on an array of processors which are software programmable to perform the physical and higher layer processing. The processing is assisted by hardware accelerators to improve performance. The processing times are limited by the number of processors and the type of hardware accelerators. Using this approach, it is very difficult to meet the timing constraints of the physical layer functions in cognitive radio applications due to the amount of interrupts, input and output handling, and memory accesses required. Increasing the number of processors makes the system very complex and the programming becomes very difficult. An SoC architecture for a 802.11a receiver is shown in Figure 1.2 [11].

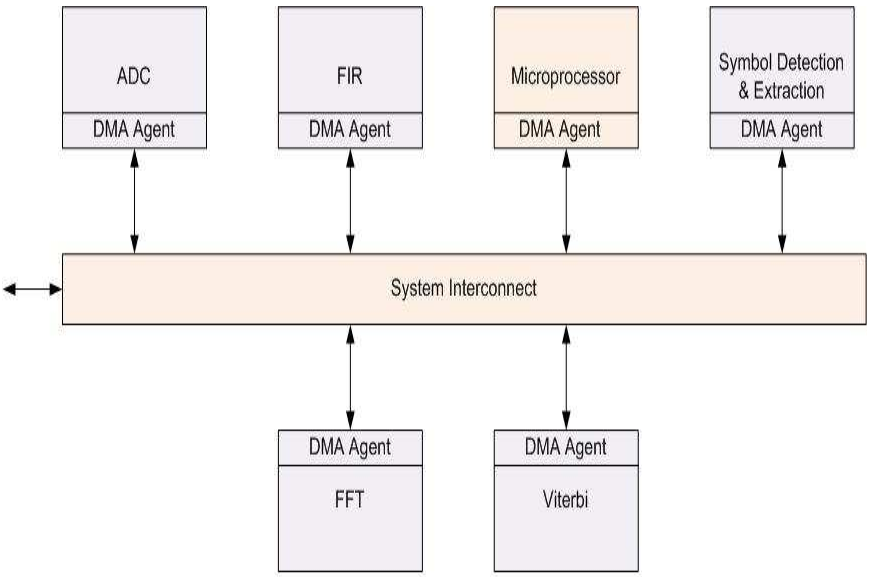


Figure 1.3: 802.11a Receiver Block Diagram (Platform Approach)

Platform Organization is a bus based architecture connecting autonomous functional modules that can have a fixed function, be parameterized or be fully programmable (Microprocessor) for supporting the target application. The global control mechanisms like a Global Task Descriptor Table (GTT) are used for inter-module communication and to schedule tasks to and from these modules. They also help in meeting the throughput and latency constraints of the application. A bus based architecture provides low latency communication and any-to-any connectivity. A microprocessor acts as one of the functional modules, hence providing the required flexibility, and communicates with the other modules through the exchange of parameters.

A control module assigns and schedules tasks for processing to the functional modules and the microprocessor. The scheduling needs to satisfy protocol throughput and latency constraints. The control module should allow efficient multiplexing of the different tasks being processed and should handle the real time scheduling of the tasks.

The diagram depicting a platform approach is shown in the Figure 1.3 [11].

1.3 WINC2R System

WINC2R is a hardware platform for cognitive radio applications that is being designed to be flexible enough to support new wireless communication protocols and will be able to offer per-packet protocol selection for improved data rates.

WINC2R is based on generic functional modules which handle the processing of the most computationally intensive physical layer functionalities. The programmable microprocessor (Microblaze) handles the differences between the different protocol standards and can support new protocols with sufficiently low complexity without sacrificing performance. It satisfies the low latency, fast context switching, and deterministic timing requirements required by multi-layer protocol processing. The data flow and allocation of tasks to the functional modules is handled by a control module called the Unit Control Module (UCM). It uses a centralized data structure called the Global Task Descriptor Table (GTT) to schedule the tasks and handle the task processing.

Hence, in contrast to the Software Defined Radio (SDR) which are completely software based and augmented by hardware accelerators for speed, WINC2R is hardware based and augmented by software for flexibility. The platform provides a flexible MAC layer that can support different MAC algorithms.

1.4 Problem Statement and Thesis Overview

To achieve this platform architecture, we have developed a FPGA based system which connects the processing engines, the CPU (which is the Microblaze), RS232, Dual Data Rate (DDR), Flash Memory and various other components provided by the Xilinx board using the IBM system interconnect called the PLB (Processor Local Bus). Every processing engine performs a specific function. A Unit Control Module (UCM) is the global control module that assigns and schedules the tasks to and from the processing engines and the Microblaze with the help of a centralized Global Task Descriptor Table. The UCM is associated with each processing engine. It is in charge of scheduling the tasks to the unit that it is associated with, assigning the command (task), monitoring the task completion, and communicating with the other units in the system for task

sequencing. The task scheduling and sequencing has to be done in a pipelined fashion to meet the the strict throughput and latency constraints of the protocol.

The FPGA based architecture achieves our goals of flexibility along with being customizable for system connectivity, DSP, and/or data processing applications. The FPGA devices are supported by easy-to-use design and development tools like Xilinx Synthesis and Place and Route tools. Using these tools system we can define a complete system, from hardware to software, within one tool and in a fraction of the time of traditional system-on-a-chip (SOC) design.

In our research, we focus on creating the Microblaze environment and the software for the WINC2R system, putting together all the different components and handling the communication between them to guarantee low latency and maximum utilization of the FPGA resources. An interrupt handler handles the interrupt from the functional modules and the Xilinx blocks based on a fixed priority assigned to each block. We develop the software for communication between the functional modules and also between Microblaze and functional modules. The Microblaze environment is created as a separate functional module and invokes the other functional modules by passing parameters. We calculate and analyze the memory and FPGA resource requirement of the WINC2R system and based on our results, we decide whether to go for the Virtex 4 or the Virtex 5 board. We propose the overall system and task processing flow - as to how the functional module would behave when a task is received. We analyze the UCM, the heart of the functional module and propose the task sequencing for the data and control tasks. We analyze the possible scenarios and begin the simulations to test them and in the future optimize the task processing to reduce the latency and maximize optimization.

In Chapter 2, we describe the WINC2R hardware platform. We discuss the board and the various hardware components that have been used for a specific purpose in the system. We present the FPGA utilization of the Virtex 4 board and justify the need for the Virtex 5 board in future versions of WINC2R. We also discuss how the Microblaze processing engine is integrated into the whole system.

In Chapter 3, we look at the WINC2R system architecture. We describe the concept

of a functional module and its memory map. In the process, we explain the various memory regions of the functional module. Finally, we explain the system flow and describe how the task processing occurs.

In Chapter 4, we detail the system software and its functions. We explain how the tasks get activated and terminated, how the Microblaze interfaces with the other functional modules, the different levels of interrupts and the communication through the command line interface needed.

In Chapter 5, we explain the structure of the UCM and the various functions it calls for the task processing. We analyze the time lines associated with different tasks sequences.

In Chapter 6, we draw conclusions and discuss the future work.

Chapter 2

WINC2R Hardware Platform

This chapter describes the WINC2R hardware platform - the Xilinx FPGA boards, the microprocessor (Microblaze), various Xilinx blocks and the functional modules. The embedded platform has been created using the Xilinx Platform Studio (XPS) integrated development environment which contains a wide variety of embedded tools, IP, libraries, wizards, and design generators [12]. The chapter also describes how the various blocks are connected together by the system interconnect. It also gives an estimate of the BRAMs required for the WINC2R system, and justifies the need for the Virtex 5 board. Section 2.5 of the chapter explains the process of bootloading for the FPGA platform. Section 2.6 describes the FPGA resources used for the Microblaze environment for both Virtex 4 and Virtex 5. The chapter also describes how the microblaze functional module is integrated with the other modules.

2.1 WINC2R Board

The WINC2R platform architecture is being implemented using Xilinx Development Board [12]. We use the Virtex-4 family, XCV4SX35 board which is a high performance solution for DSP applications. It has 192 DSP slices, 192 18KB Block RAMs, 8 Digital Clock Managers (DCM), high speed memory interface for DDR and Flash Memory, JTAG-Connector for Virtex and PROM programming (JTAG-Chain) and 4 free usable LEDs, all of which makes it a powerful FPGA platform. It has 4MB of flash memory and 64MB of DDR SDRAM external memory. Currently, we use two Virtex-4 boards - one for the transmit chain and the other for the receive chain as one board cannot accommodate the whole design.

We will migrate to Virtex 5, XC5VSX95T board in the immediate future which

would accommodate the whole design easily. It has 640 DSP slices, 244 36KB Block RAMs, 12 Digital Clock Managers (DCM), high speed memory interface for DDR2 SDRAM, JTAG-Connector for Virtex and PROM programming (JTAG-Chain) and 2 free usable LEDs, which makes it a more powerful FPGA platform than Virtex 4. It has 64 bit 256MB of DDR2 SDRAM external memory.

The microprocessor on these FPGA platforms is Microblaze which is described in Section 2.2.

2.2 Microblaze

The Microblaze embedded processor soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx Field Programmable Gate Arrays (FPGAs). It is implemented with a Harvard memory architecture. It is a 32-bit processor with 32-bit address bus and 32 or 64-bit data bus. It provides three interfaces for memory accesses - Local Memory Bus (LMB), Processor Local Bus (PLB) or On-Chip Peripheral Bus (OPB). It provides a debug interface connected to the Xilinx Microprocessor Debug Module (MDM) core, which interfaces with the JTAG port of Xilinx FPGAs. It supports reset, interrupt, user exception, break, and hardware exceptions [13].

Figure 2.1 shows the functional block diagram of the Microblaze core.

Hence, Microblaze soft processor is highly configurable allowing us to select a specific set of features as required for our custom design.

2.3 System Overview

The WINC2R system is shown in Figure 2.2.

The various blocks are explained as follows:

- ***General Purpose Input Output (GPIO)***: The XPS GPIO design provides a general purpose input/output interface to a PLB. The GPIO can be configured as either a single or a dual channel device. The channels may be configured to generate an interrupt when a transition on any of their inputs occurs. In our

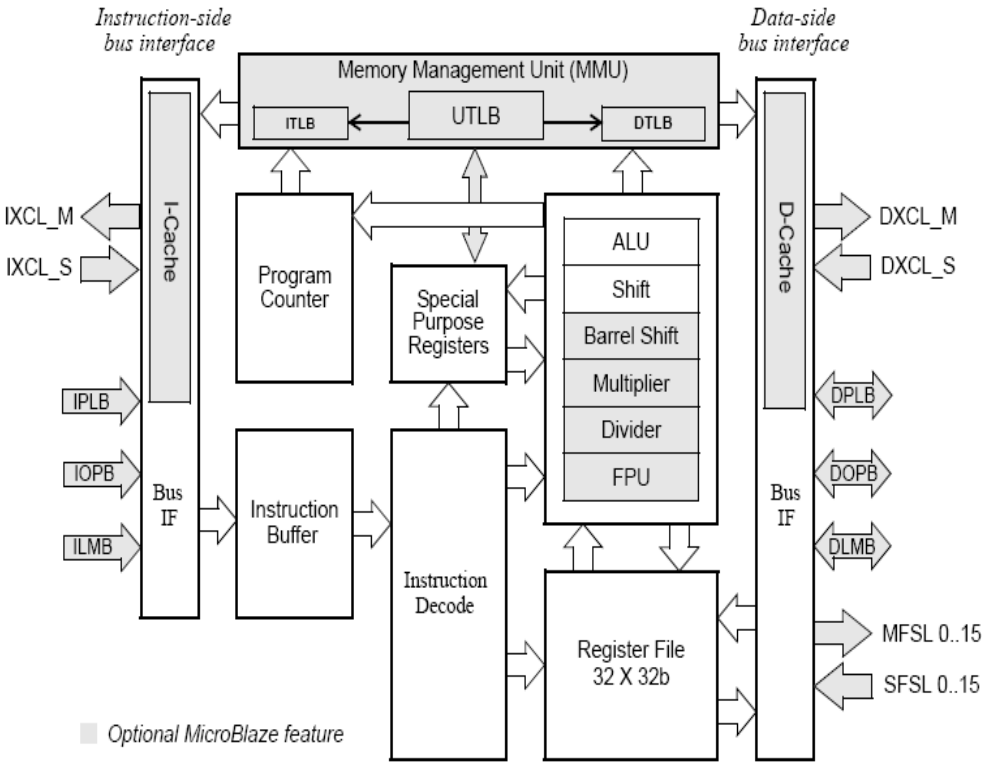


Figure 2.1: The Microblaze core block diagram

system we have used the GPIO to glow the LEDs for easy debugging purposes [14].

- ***Interrupt Controller (INTC)***: The XPS INTC concentrates multiple interrupt inputs from peripheral devices to a single interrupt output to the system processor. The registers for checking, enabling and acknowledging interrupts are accessed through a slave interface for the PLB v4.6. The number of interrupts and other aspects have been tailored to our system. There are upto 32 configurable, priority based interrupts. In WINC2R, we have used 12 interrupts, one each from the eight functional modules, RS232, the timer, debug module and the PLB to PLB bridge [15].

- ***Timer***: The Timer/Counter is organized as two identical timer modules. Each timer module has an associated load register that is used to hold either the initial

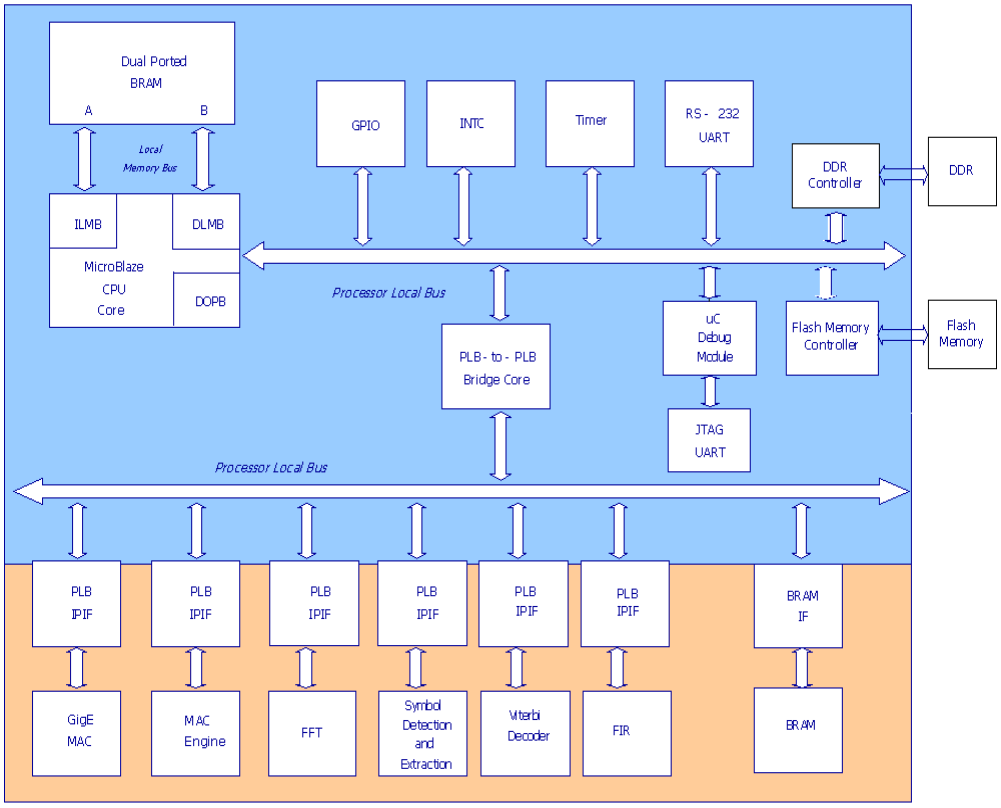


Figure 2.2: The WINC2R System Overview

value for the counter for event generation, or a capture value, depending on the mode of the timer. The generate value is used to generate a single interrupt at the expiration of an interval, or a continuous series of interrupts with a programmable interval. The capture value is the timer value that is latched on detection of an external event. All of the Timer/Counter interrupts are ORed together to generate a single external interrupt signal. The interrupt service routine reads the control/status registers to determine the source of the interrupt. In WINC2R, the two timers are used in the generate mode and the two external signals GenerateOut0 and GenerateOut1 are used to synchronize all the other modules in the WINC2R system. They are also used for the synchronous task processing [16].

- ***RS232/Universal Asynchronous Receiver/Transmitter (UART)***: The XPS (UART) Lite connects to the primary PLB and provides the controller interface

for asynchronous serial data transfer. It performs parallel to serial conversion on characters received through PLB and serial to parallel conversion on characters received from a serial peripheral. The XPS UART Lite is capable of transmitting and receiving 8, 7, 6 or 5 bit characters, with 1 stop bit and odd, even or no parity. It can transmit and receive independently. The device can be configured and its status can be monitored via the internal register set. The XPS UART Lite generates an interrupt when data is present in the Receive FIFO or when the Transmit FIFO becomes empty. This interrupt can be masked by using interrupt enable/disable signal. The device contains a 16-bit programmable baud rate generator and independent 16 word Transmit and Receive FIFOs. The FIFOs can be enabled or disabled through software control. We use the RS232 for communication through the Command Line Interface (CLI) [17].

- ***Microblaze Debug Module (MDM)***: The MDM enables JTAG-based debugging of up to eight MicroBlaze processors. It also provides JTAG-based communication to the Xilinx Microblaze Trace Core. The MDM includes an UART with a configurable slave bus interface. It can be configured for a PLBv46 bus or an OPB bus. The UART TX and RX signals are transmitted over the FPGA JTAG port to and from the Xilinx Microprocessor Debug (XMD) tool. It is also required to program the flash memory and the DDR SDRAM memory. MDM is also used for debugging using chipscope [18].

- ***PLB to PLB Bridge***: The PLBv46 to PLBv46 Bridge allows us to connect two PLB buses. The PLBv46 to PLBv46 Bridge can be used to isolate slow PLB peripherals from the primary PLB and improve the system performance. The PLBv46 to PLBv46 Bridge is a slave on the primary PLB and is a master on the secondary PLB. It can generate an interrupt to identify abnormal terminations. A reset logic provides exclusive resets to primary PLB and secondary PLB. We used two PLB buses - primary and secondary. The primary bus connects to all the Xilinx provided blocks and memory controllers, while the secondary connects to the FUs [19].

- ***XPS Multi Channel External Memory Controller (XPS MCH EMC)***: The Xilinx XPS MCH EMC provides the control interface for external synchronous, asynchronous SRAM and flash memory devices through the PLB interfaces. It supports multiple (up to 4) external memory banks. It also supports single-beat and burst transactions. On the WINC2R platform, the EMC is used to connect the flash memory through the PLB interface [20].

- ***Multi Port Memory Controller (MPMC)***: MPMC is a fully parameterizable memory controller that supports Double Data Rate (DDR and DDR2) memory and Single Data Rate (SDRAM). MPMC provides access to memory for one to eight ports, where each port can be chosen from a set of Personality Interface Modules (PIMs) that permit connectivity into PowerPC 405 processor, MicroBlaze, CoreConnect, and the MPMC Native Port Interface (NPI) structures. MPMC also supports the Soft Direct Memory Access (SDMA) controller that provides full-duplex, high-bandwidth, LocalLink interfaces into memory. Additionally, the MPMC supports optional Error Correcting Code (ECC) and Performance Monitoring (PM). On the WINC2R platform, the MPMC is used to connect a DDR synchronous dynamic RAM through the PLB interface [21].

- ***XPS BRAM Interface Controller***: The XPS BRAM Interface Controller is a Xilinx IP module that incorporates a PLB v4.6. This controller is designed to be byte accessible. Any access size (in bytes) up to the parameterized data width of the BRAM is permitted. The XPS BRAM Interface Controller is the interface between the PLBV46 and the BRAM block peripheral. A BRAM memory subsystem consists of the controller along with the actual BRAM components that are included in the BRAM block peripheral [22].

- ***BRAM Block***: The BRAM Block is a configurable memory module that attaches to a variety of BRAM Interface Controllers. The BRAM Block structural HDL is generated by the EDK design tools based on the configuration of the BRAM interface controller IP. All BRAM Block parameters are automatically calculated and assigned by the EDK tools Platgen and Simgen. We used two BRAMs for

WINC2R connected to primary and secondary PLB respectively [23].

- ***Flash Memory***: The flash memory provided on the Virtex 4 board is Atmel AT49BV322AT6. It is a 2.7-volt 32-megabit Flash memory organized as 2,097,152 words of 16 bits each or 4,194,304 bytes of 8 bits each. We used the flash memory to store our application software. Flash memory is non-volatile [24].

- ***DDR SDRAM***: The DDR SDRAM provided on the Virtex 4 board is a one developed by HYNIX - HYB25D512160BC6. The 512-Mbit Double-Data-Rate SDRAM is a high speed CMOS, dynamic random-access memory containing 536,870,912 bits. It is internally configured as a quad-bank DRAM. The 512-Mbit Double-Data-Rate SDRAM uses a double-data-rate architecture to achieve high-speed operation. We used the DDR to execute our application software. DDR is volatile memory storage [25].

The user uses the Command Line Interface and a terminal to interface directly to the hardware.

2.4 Bus Structure/Interconnect

With the advancement of System on Chip designs, numerous peripherals are integrated on the same board along with the processor. As a result, the on-chip buses used in such designs have to be sufficiently flexible and robust in order to support a wide variety of embedded system requirements.

The IBM Blue Logic cores program provides the framework to efficiently realize complex system-on-chip (SOC) designs. Typically, a SOC contains a large number of functional blocks represented by a very large number of logic gates. The blocks are generally in the form of common reusable macros. However, many current applications use single chip solutions which are designed as custom chips, each with its own internal architecture. Logical units that can be re-used in different applications are difficult to extract from such chips. As a result, many times the same functions are redesigned from the different applications.

Using common buses for inter-modular communications promotes macro reuse. For this purpose, IBM CoreConnect architecture [27] provides three buses for interconnecting cores, library macros, and custom logic:

- ***Processor Local Bus (PLB)***: The PLB is a synchronous, high performance bus used to inter connect high performance processor, ASIC and memory cores. It provides the infrastructure for connecting an optional number of PLB masters and slaves into an overall PLB system. It consists of a bus control unit, a watchdog timer, and separate address, write, and read data path units. The main features of the PLB bus are:
  - PLB arbitration support for up to 16 masters with number of PLB masters configurable via a design parameter
  - PLB address and data steering support for up to 16 masters
  - 128-bit, 64-bit, and 32-bit support for masters and slaves
  - PLB address pipelining
  - Four levels of dynamic master request priority
  - PLB Reset generated synchronously to the PLB clock from external reset when external reset provided
  - DMA support for buffered, fly-by, peripheral-to-memory, memory-to-peripheral, and memory-tomemory transfers
- ***On-Chip Peripheral Bus (OPB)***: OPB is a secondary bus used to connect high latency peripherals and alleviate system performance bottlenecks by reducing capacitive loading on the PLB. Peripherals suitable for attachment to the OPB include serial ports, parallel ports, UARTs, GPIO, timers and other low-bandwidth devices. A bridge module is used to interface the OPB bus with the PLB bus. The main features of the OPB bus are:
  - Upto 64 bit address bus width
  - 32 or 64 bit read, write data bus width support

  - Support for multiple masters
  - Bus parking for reduced transfer latency
  - Single cycle data transfer between OPB masters and slaves

- ***Device Control Register (DCR) Bus***: Lower performance status and configuration registers are typically read and written through the Device Control Register (DCR) Bus. The DCR provides a maximum throughput of one read or write transfer every two cycles and is a fully synchronous bus. It allows the lower performance status and control read/write transfers to occur separately and concurrently with high speed transfers on PLB and OPB buses, thus improving system response time and overall performance.

As we can see in the Figure 2.2, WINC2R design uses two PLB buses connected by a PLB-PLB bridge. The first PLB has memory controllers connected to it and the second PLB has the PEs connected to it. We cannot put all the components on one PLB bus as it causes capacitative loading on the PLB which increases signal propagation delay and reduces performance. It also decreases the throughput as all the components would be accessing the same bus. Hence, we use two PLBs. The two PLBs are connected using the PLB to PLB bridge. The bridge logic components allow better signal propagation management since signal needs to traverse smaller wire lengths.

2.5 Bootloader

When the Microblaze in the WINC2R design comes out of reset, it starts executing the code stored in BRAM at the processor reset location. Typically, the BRAM size is only a few kilobytes and is therefore too small to accommodate the entire software application image. Therefore, the software application image (typically, a few megabytes-worth of data) can be stored in a non-volatile memory like the flash memory. With the increasing speed of modern SRAM, executing code from the flash memory is very slow, hence, typically a faster volatile external memory like DDR SDRAM is used to execute the program. A small bootloader is designed to fit inside the BRAM. The processor executes the bootloader on reset, which copies the software application image from the

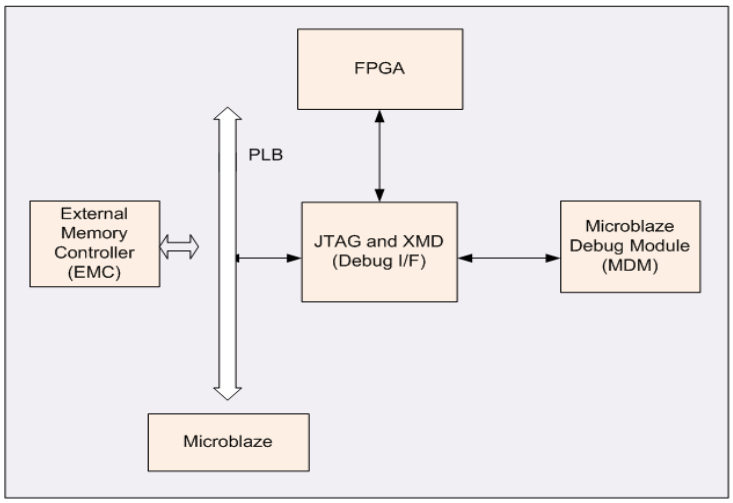


Figure 2.3: Programming Flash Memory

flash memory into the external memory. The bootloader then transfers control to the software application, which continues execution from the external memory [26].

The software application created is in Executable Linked Format (ELF). When bootloading a software application from flash, an ELF image is converted to one of the common bootloadable image formats, such as SREC (Motorola S Record). This keeps the bootloader simpler and smaller than the original format. As shown in Figure 2.3, the programming is achieved through the debugger connection to the Microblaze. XPS downloads and executes a small in-system flash programming stub on the target processor. The in-system programming stub requires a minimum of 8 KB of memory to operate. A host Tool Command Language (TCl) script drives the in-system flash programming stub with commands and data and completes the flash programming. While running, the flash programmer erases only as many flash blocks as required in which to store the image. When the programming is done, the flash programmer TCl sends an exit command to the flash programmer and terminates the debug (XMD) session.

Table 2.1: FPGA Utilization for Virtex 4

Resource	Used	Max Available	% Utilization
Slice FFs	5603	30720	18%
4 I/P LUTs	7702	30720	25%
Slices	5368	15360	34%
BRAM	75	192	39%
DSP	3	192	1%
IOB	638	448	142%
DCM	1	8	12%

2.6 FPGA Utilization

We refer to the BRAM estimate given in the Appendix A before discussing the FPGA utilization of Virtex 4 and Virtex 5 boards. The estimate tells us the requirement of BRAMs for the WINC2R system and justifies the need to migrate to Virtex 5. The utilization of the FPGA resources for the Virtex 4 is given in the Table 2.1. This utilization is for the Microblaze system environment only, which excludes the resources required for the PE as discussed in Appendix A.

We can infer that putting the eight FU's, the FPGA resources available on Virtex 4 especially the BRAM are not sufficient for the WINC2R system. As we can see in the table, the IO Buffers are already over utilized by 142%. The FU's would require a lot of FPGA resources as some of them perform a large amount of calculations. Hence the need to migrate to Virtex 5. The utilization of the FPGA resources for the Virtex 5 is given in the Table 2.2. Again, this utilization is for the Microblaze system environment only which excludes the resources required for the PE. Virtex 5 has much more FPGA resources for our use which will fulfill all our resource requirements.

2.7 FPGA Top and System Integration

As we have seen in Figure 2.2, the eight functional modules are connected to the Microblaze environment through the IPIF interface. All the different modules are implemented as independent modules and are put into one larger design called the FPGA top. The FPGA top is as shown in the Figure 2.4. The

Table 2.2: FPGA Utilization for Virtex 5

Resource	**Used**	**Max Available**	**% Utilization**
Slice FFs	5603	58880	9%
4 I/P LUTs	7703	58880	13%
Slices	2457	10849	22%
BRAM	38	244	15%
DSP	3	640	0%
IOB	638	640	99%
DCM	1	12	8%

Every independent module, including the Microblaze environment is in the form of a ngc file which is the output of Xilinx synthesis tool. We call the ngc file for the Microblaze environment *pcore*.

An ngc file is a binary output file, the result of generating the netlist for the FPGA. This file can be used as a top level design file or a module file. The module file contains the implementation of a module in the design. If an ngc file exists for a module, NGDBuild reads this file directly, without looking for the source of the design. Hence, the ngc file is sort of a black box for the underlying design. In WINC2R system, the functional modules and the Microblaze environment all have a corresponding ngc file which contains their underlying design. The FPGA top also has an ngc file. The FPGA top connects all of the different ngc design files, as shown in Figure 2.5.

The various ngc files are combined using the NGDBuild command of the Xilinx synthesis tool. It reads the netlist file in the NGC format and creates an NGD file that describes the logical design (a logical design is in terms of logic elements such as AND gates, OR gates, decoders, flip-flops, and RAMs). The NGD file resulting from an NGDBuild run contains both a logical description of the design reduced to Xilinx Native Generic Database (NGD) primitives and a description in terms of the original hierarchy expressed in the input netlist files. The output NGD file can be easily mapped to the Virtex 4 or Virtex 5 board.

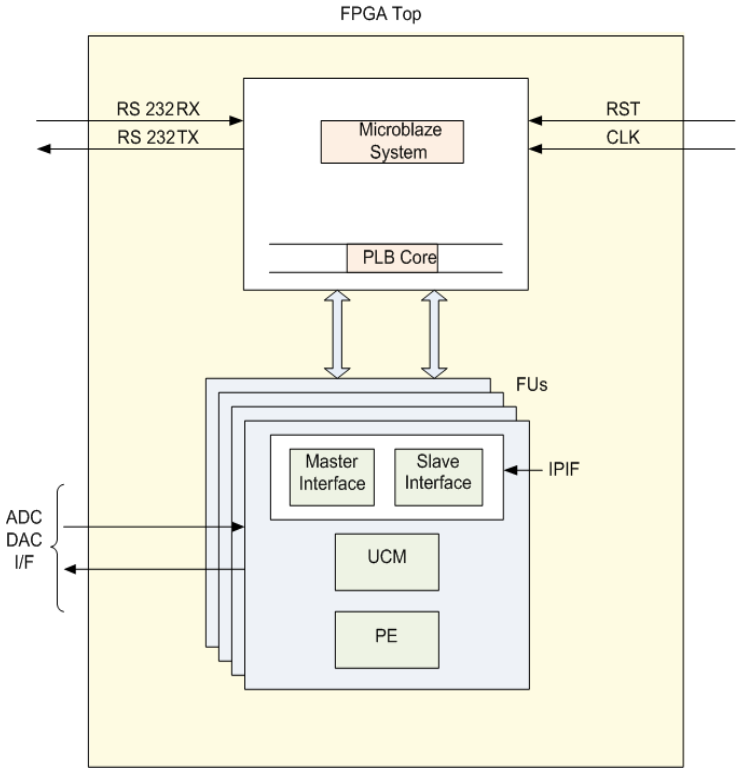


Figure 2.4: FPGA Top

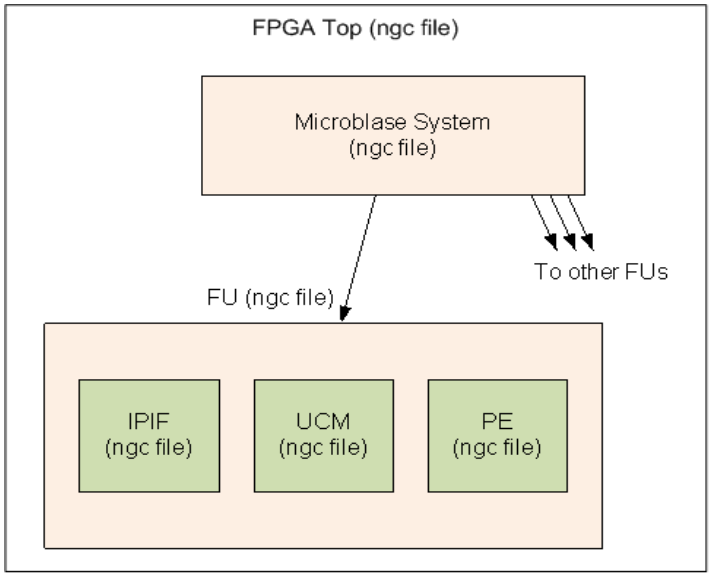


Figure 2.5: System after combining ngc files

Chapter 3

WINC2R System Architecture

This chapter describes the system architecture of the WINC2R platform. Section 3.1 details the memory map for the NCP platform. Section 3.2 describes the task flow processing by the UCM using the GTT and how the various functional modules communicate amongst themselves.

3.1 Memory Map

The blocks in the NCP platform like GPIO, Timer, INTC, and the FUs have a size of 64KB, each allocated within a certain address range. The Microblaze can access these address ranges to read and write from the various registers available. Microblaze also handles the interrupts, the initialization and the various control parameters by accessing these memory locations. Since the Microblaze is a 32-bit microprocessor, the address ranges can range from 0x00000000 to 0xFFFFFFFF. The memory map of the NCP platform can be visualized in the Figure 3.1.

There are currently 8 functional modules in the WINC2R platform - MAC Tx, Header, Modulator, FFT, Synchronization block, Demodulator, Checker and the MAC Rx. The MAC Tx, Header, Modulator and FFT form the transmit chain and the Synchronization block, Demodulator, Checker and MAC Rx form the receive chain. As we saw in Chapter 1, every FU performs specific functions which can be common for all the protocols.

Each FU is again divided into sub blocks. The FU essentially consists of the following major sub-blocks : IPIF, UCM, PE and local memory. A functional module architecture diagram is shown in the Figure 3.2. The FU is connected to the PLB bus using the Xilinx IPIF interface module. This module supports both slave and master bus access

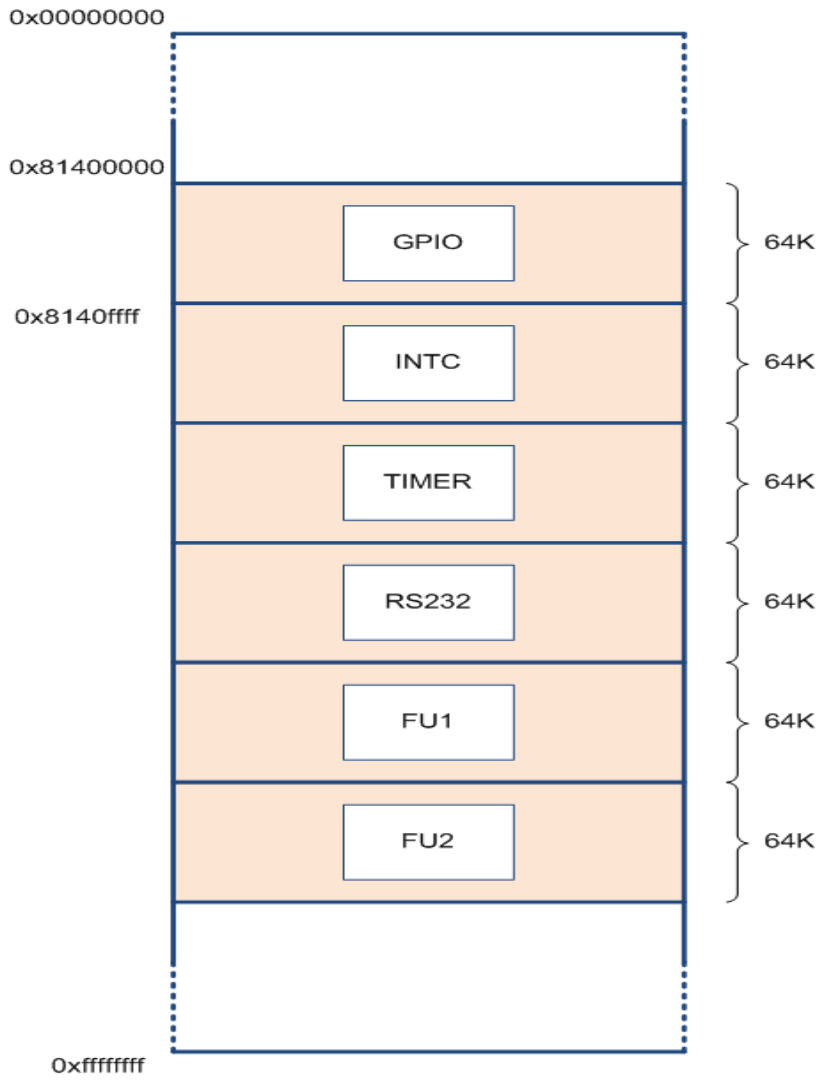


Figure 3.1: NCP Platform Memory Map

schemes. A DMA engine permits automated data transfer between FUs without any processor interventions. The UCM block is essentially a wrapper that wraps its diverse underlying functionalities into a single entity - UCM, DMA engine, Task Descriptor table, Task Scheduler Queues, In/Out buffers, register map and the control logic, which are required to glue all the functional blocks together (address decoder, bus muxes, memory arbiters).

The address decoding for the various sub blocks in the FU is shown in the FU memory map in Figure 3.3. The IPIF RMAP at offset 0x0000 is generated using the

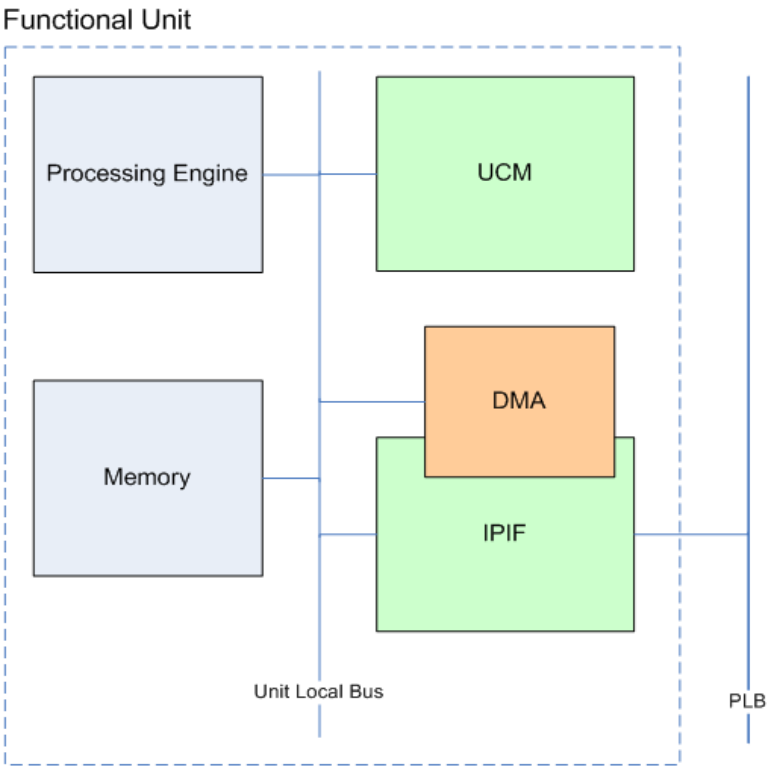


Figure 3.2: Functional Unit

Xilinx IPIF block. The address decoding function for the rest of the sub blocks are maintained by the FU itself and ranges from 0x1000 to 0xFFFC (UCM RMAP to PE Output Buffer). The UCM TD Table describes the task associated to a particular FU. The UCM Task Scheduler Queue specifies which tasks are in the asynchronous or the synchronous queue. PE RMAP maintains the control and status information, version control and output buffer pointer sets. The PE input and output buffers serves as the local memory for the PEs.

3.1.1 Global Task Descriptor Table

The Global Task Descriptor Table (GTT) resides in the BRAM connected to the secondary PLB bus. The Microblaze creates and initializes the GTT at the start. It is centralized in the sense that it is used by the UCMs of all the PEs in order to decode the FU in charge of task execution and insert the asynchronous target (consumer) tasks

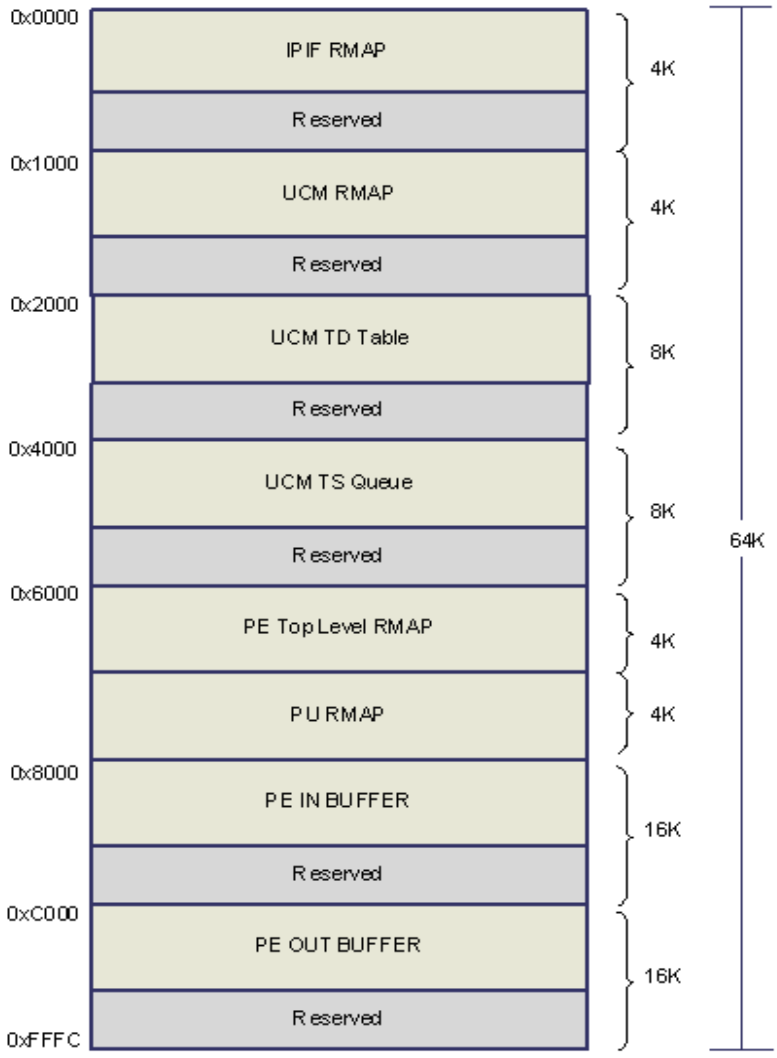


Figure 3.3: FU Memory Map

to the FU's queues. It synchronizes task execution with the completion of all producer tasks. The UCM in every PE accesses the GTT. It is an array that is indexed by a 15-bit TaskID in the Task Descriptor Table (TD) which is explained in Section 3.1.2. The values in the GTT are modified by the UCM during the task execution. The GTT contains the information about the different tasks and which FU the tasks are associated to. It also helps in the task synchronization through the enable flag processing. It also contains configuration settings for the tasks.

A Func Unit ID field in the GTT identifies the functional unit to which the task is

allocated. A task active bit specifies whether the task is inactive (0) or active (1). The TD pointer points to the start of the TD table in the corresponding PE as is shown in the Figure 3.6. The TSQ pointer points to the task scheduler queue to which the task will be allocated depending on whether the task is synchronous or asynchronous. A Busy vector field in the GTT indicates which consumer input buffers are being processed by the consumer PE. Every bit of the vector refers to a corresponding buffer. The Enable Flag acts like a semaphore to the number of tasks and is explained in detail in Section 4.1 of Chapter 4. The SynchAsynch bit specifies whether the task is synchronous (0) or asynchronous (1). The task priority bits specify the priority of the tasks which is 00 for control tasks and 01 for data tasks. Scheduled time shift is used to calculate the start time of the synchronous task. Guard time is used to schedule the synchronous tasks. Reschedule period is also used to schedule synchronous tasks when for some reason the task has not been processed till now. If the Dynamic data size is 1 then this means that the input/output data sizing for the task is dynamic.

3.1.2 Task Descriptor Tables

A Task Descriptor table (TD) is local to every PE and handles the task flow execution within the PE. The UCM fetches the information related to a task belonging to a particular PE from the TD table of that PE. A TD table is shown in the Figure 3.6. The TD table contains the information regarding the number of input/output buffers used by the task, the next tasks triggered after the successful execution of that task and the information about whether a task is a chunking/dechunking task or not.

3.1.3 IO Buffers

The Input and Output buffers in a PE form the PE local memory. A producer PE while transferring the data, performs the DMA transfer from the output buffer of the producer PE to the input buffer of the consumer PE. Both the buffer regions are divided into two parts - the pointer set region and the data buffer region. The pointer set region contains pointers to the data buffer region. After the DMA transfer, the producer PE updates the input buffer pointers and the size in the pointer set region of the consumer PE. The

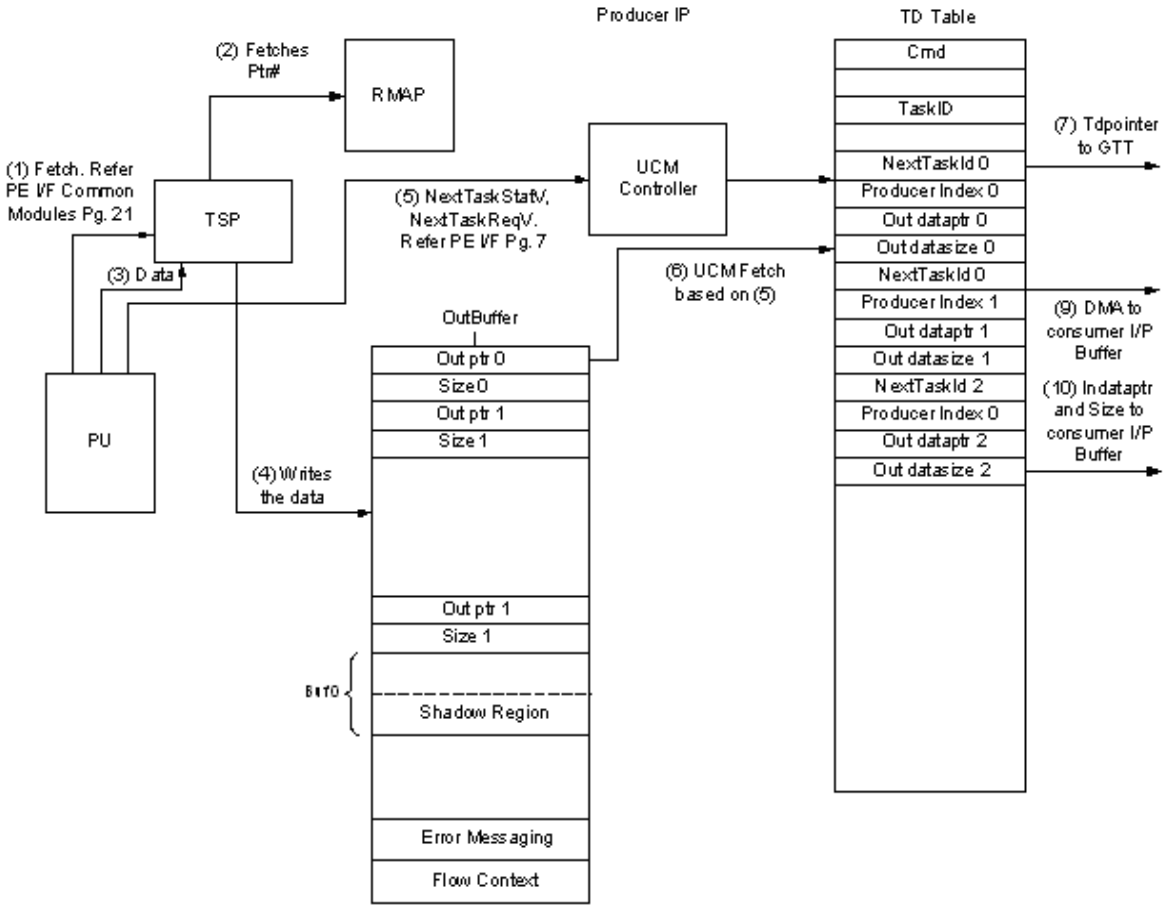


Figure 3.4: System Flow

last data buffer region in he IO buffers are reserved for flow context information. The Buffer region is also reserved for error messaging in case of some erroneous scenarios.

3.2 System Flow

As depicted in Figure 3.4 and Figure 3.5, the system flow describes the sequence of command flow originating from Producer PE to Consumer PE. The sequence has been marked by numbers.

When the producer PE has some data in its input buffers, it calls the Task Spawn Processor (TSP) and requests the output buffer pointer sets. The TSP fetches the pointers to the output buffer regions from the PE top Register Map (RMAP) of every

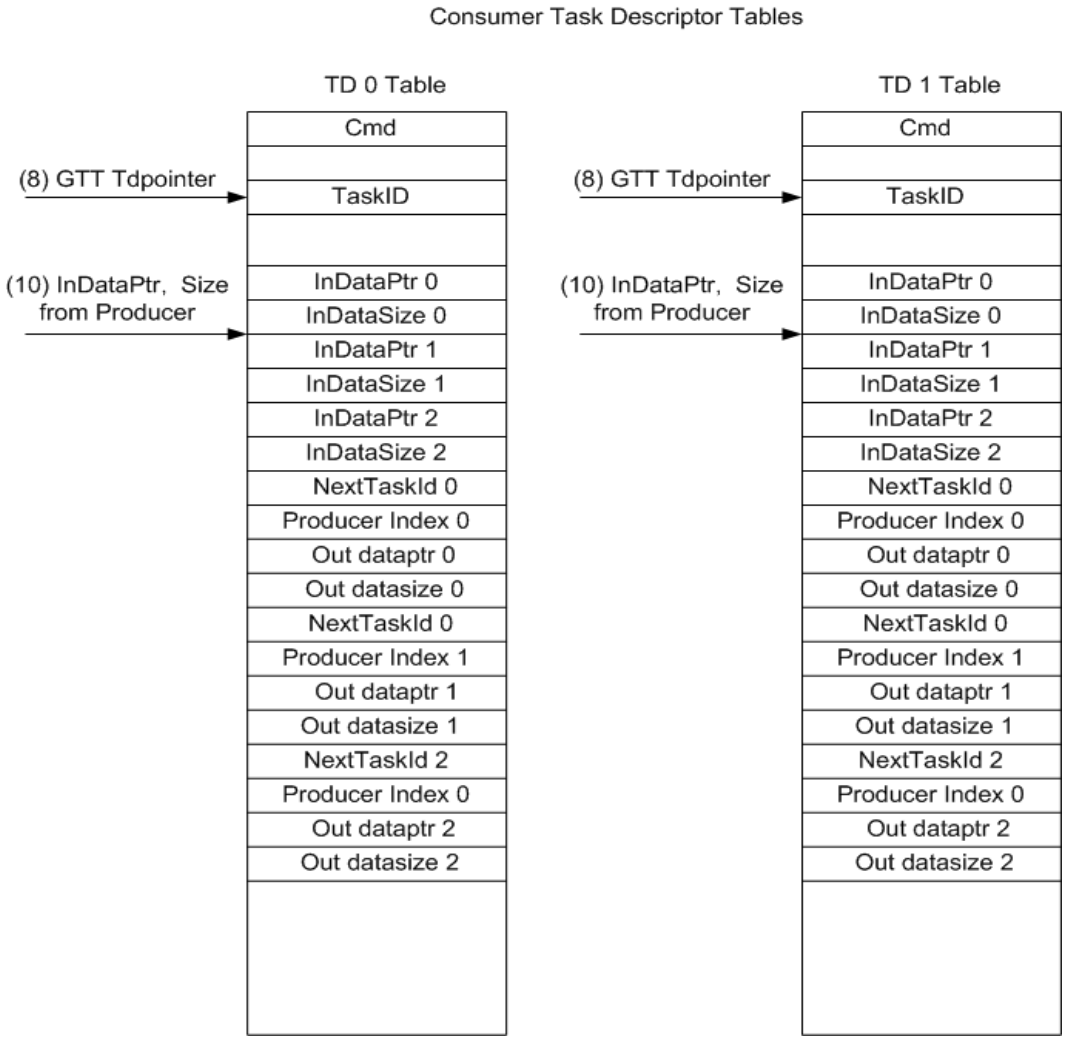


Figure 3.5: System Flow Ctd.

PE. The producer PE may request several output buffer regions depending on the number of tasks. The output buffers required for a particular task are hard-coded in the PE at the hardware level. The TSP then receives the data from the input buffer of the producer PE and writes it to the Output Buffer region as specified by the buffer pointers obtained from the PE RMAP. The PE then sends the NextTaskStatus vector and NextTaskReq vector to the UCM controller. Based on the encoding of these two vectors, the UCM activates the specified tasks. The encoding scheme also specifies the buffer regions required for the next task triggered.

The UCM fetches the output data pointers and output data size from the output

buffer of the PE and puts it in the Task Descriptor Table (TD) of the producer PE. The Task ID in the producer TD table identifies a particular task. It indexes the Global Task Table (GTT) to point to the entry for that particular task. The next task id in the TD table identifies the next task triggered at the consumer PE. Next task ID also indexes the GTT to point to the entry of the next task. A single task at the producer PE can trigger multiple tasks belonging to the same or different consumer PEs.

The DMA engine transfers the data from the output buffer of the producer PE to the input buffer of the consumer PE. The OutDataPtrs in the Producer PE and the InDataPtrs in the Consumer PE are fixed after the initialization. The OutDataSize are taken from the producer Output Buffers and written on to the consumer TD tables as InDataSize.

3.2.1 GTT and TD Interface

The NextTaskID in the TD table indexes the GTT by pointing to the start of the entry for that particular task in the GTT. The Tdpointer (32-bits) in the GTT in turn contains the physical address of the first field in the consumer TD table.

As shown in the Figure 3.6, the producer TD table has the NextTaskID 0 field which serves as an offset to the GTT table. It points to the start of the entry in the GTT table (FuncUnitID) for that particular task. If we add 4 to this offset, it points to the field TD Pointer A. The TD pointer has the 32-bit physical address of the start of the consumer TD table. The InDataSize in the consumer TD table is copied from the OutDataSize of the producer TD table and provides the offset to the input buffer of the consumer.

The figure shows one producer PU which has 2 tasks as pointed to by NextTaskID 0 and NextTaskID 2. The NextTaskID 0 has two buffers associated to it as pointed to by OutDataPtr 0 and OutDataPtr 1. Since these two buffers belong to the same task, the NextTaskID 0 for both of them will point to the TD Pointer A. The NextTaskID 2 has only one buffer associated to it and points to the TD Pointer B in the GTT Table. The TD Pointer A and TD Pointer B in the GTT point to the consumer TD Tables.

The interface between the TD tables and GTT can have the following cases based

on the number of tasks triggered and the number of buffers associated with it:

- Single task, Single buffer,
- Single task, Multiple Buffers,
- Multiple tasks, Single buffer,
- Multiple tasks, Multiple buffers,
- Multiple Tasks in different PEs

In Figure 3.6, the producer tasks belong to the same PE, but the multiple tasks triggered belong to different PEs.

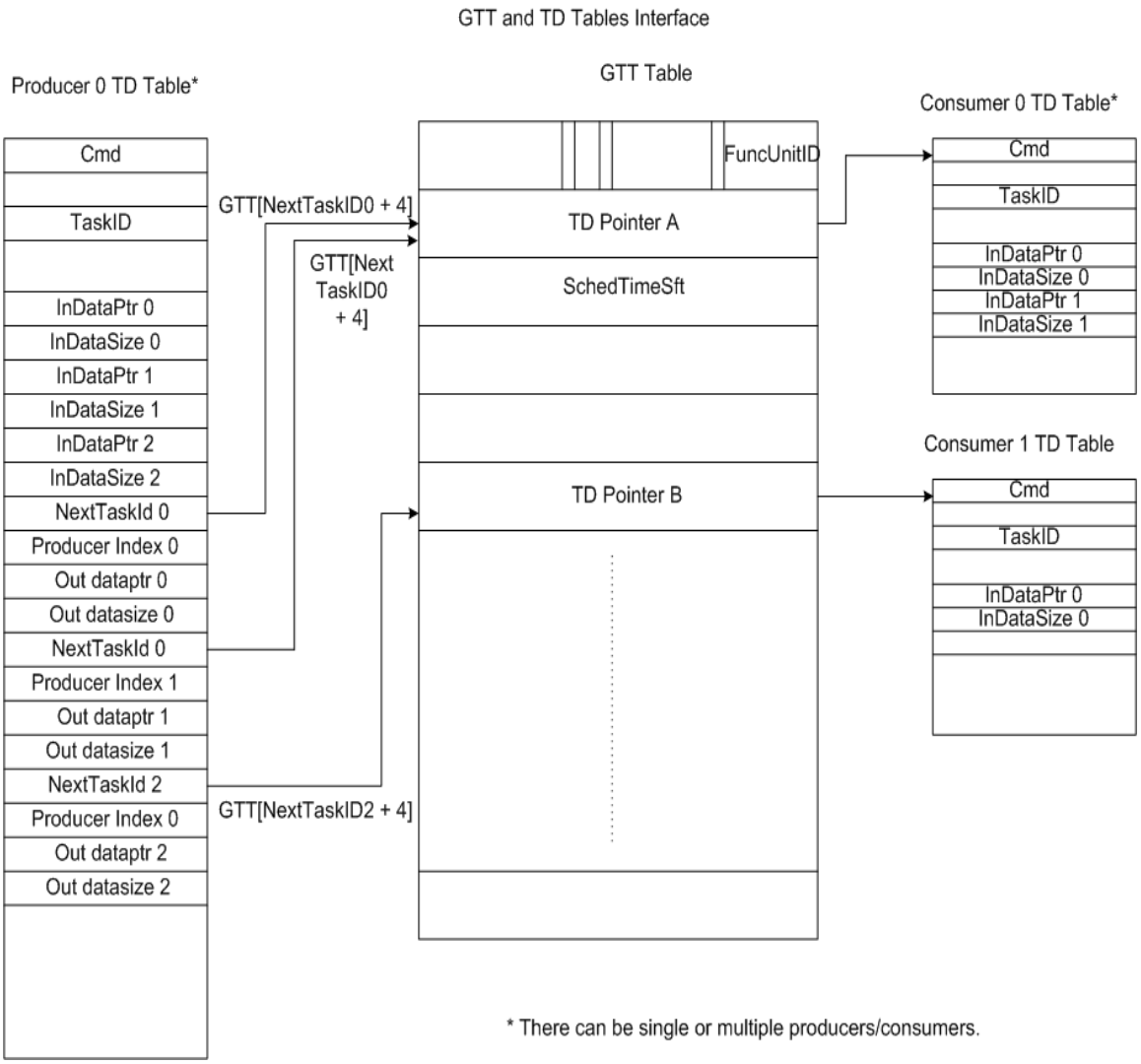


Figure 3.6: Interface between TD table and GTT

Chapter 4

WINC2R System Software

4.1 Task Activation

As we have seen in the previous chapter, when the Producer PE has data to transfer, a new task is invoked based on the encoding of the NextTaskStatus and NextTaskReq vectors. When this happens, the UCM takes control of the data transfer and scheduling and processing of the task. As we saw, the NextTaskId points to the Tdpointer in the GTT. This Tdpointer is sent to the Task Controller of the UCM. Task Controller fetches the various fields like FlowContextSize, NumInBufs, etc. from the TD table which are required to process the current task. It also fetches the input buffer pointers and the context information from the TD table. After fetching, the task controller updates the input buffer pointer, input data size and the context information in the input buffer of the PE. Based on the SyncAsync and task priority, the UCM writes the task to the correct scheduler queue.

The enable flag is one of the fields in the GTT table. It has the value of the number of producer PEs that want to talk to a consumer PE at the same time. If there are 3 PEs who want to talk to a consumer PE, then the enable flag has the value 3. If the same task is associated to multiple buffers, then the enable flag has value equal to the number of buffers associated.

Every PE is both a producer and a consumer. The tasks are activated and put on the scheduler queues based on the Enable Flag. The flowchart depicting the flow of events on the producer side for task scheduling is shown in Figure 4.1.

When a PE needs to do a data transfer, it checks the BusyVector. If it is 0, it does the DMA transfer and updates the InDataPtr and InDataSize of the consumer PE. It then checks the Enable Flag. If its value is 0xFF, that implies that the task is always enabled and the PE does not need to decrement the Enable Flag. If the value of Enable Flag is between 0x01-0xFE, then the PE decrements it one by one. If the enable flag becomes 0, the task is scheduled to be activated at the consumer. If the enable flag does not become 0, the PE does not so anything. Hence, the PEs, keep decrementing the enable flag after the DMA transfer but the task is scheduled at the consumer only if enable flag becomes 0. The SynchAsync = 1, that means the task is asynchronous and the task is put on the appropriate asynchronous queue based on the priority. If the task is synchronous, then the ScheTimeSft is checked. If it is 0, that means a synchronous task is already scheduled on the synchronous queue and nothing needs to be done. If it is not 0, that means that a task is being scheduled to be put on the synchronous queue at some later time. At this point the producer's work is done.

The task is scheduled when the producer UCM writes a Task Descriptor Format (TDF) into the Task Scheduler Queue (TSQ) of the consumer. The task descriptor format is different for synchronous and asynchronous tasks as shown in the Figure 4.2. After writing the TDF, a FIFO empty flag gets de-asserted and this indicates to TSQ Controller that a task is ready for the en-queuing process. The TSQ Controller then proceeds with the task en-queuing process.

In the Synchronous Task Descriptor, task id is the Task identifier used to index GTT in order to get task enable status (EnableFlag) which determines if the task is

eligible for activation. Tdpointer points to the TD table. Reschedule period gives the rescheduled time for the task. Start time gives the task start time. Guard time is used to limit the wait for scheduling the synchronous task. Processing time is used to determine the busy time slots for the functional units in order to determine if the new task can be scheduled. Asynch Move Flag indicates that the task should be moved to the asynchronous queue if it misses the synchronous activation. Next pointer points to the next descriptor in a scheduler queue. It is used for the insertions and deletions in the queue.

In the Asynchronous Task Descriptor, queue id describes the queue the task belongs to. There are 4 asynchronous queues: AsynchQ0, AsynchQ1, AsynchQ2 and AsynchQ3. Tdpointer, processing time and the next pointer points to the next descriptor in the scheduler queue. It is used for the insertions and deletions in the queue.

4.2 Task Termination

If a task is present in the queues of the consumer, the enable flag is validated. If it is 0, the task is activated. If it is not 0, this represents an erroneous scenario as the task should be scheduled at the consumer only when the enable flag is 0. The flowchart depicting the flow of events on the consumer side is shown in Figure 4.3. Once the task has been activated, the consumer stays in idle state till it receives the command done signal. When the consumer PE gets the command done or the task completed signal, it checks if the Reload Enable Flag bit is set. If it is, then if the task is not in self repeating mode, then the enable flag is re-initialized with the initial value. If the Reload Enable Flag bit is not set, then it goes back to check whether it has received command done signal or not. The Enable Flag is never reloaded if the task is in self repeating mode. After the Enable Flag has been reloaded, the consumer's work is done.

4.3 The Interrupt Structure

The interrupt structure for the WINC2R system follows a hierarchy as shown in the Figure 4.4. The interrupts arise at the PE level. There are 2 PE interrupts and 1

UCM interrupt. These interrupts are OR'd together to form one interrupt. This is essentially the signal IP2bus_intr which goes from the PU to the IPIF. The IPIF and the PU together is called a Functional Unit (FU). A single interrupt line goes out from the FU to the interrupt controller (INTC) connected to the primary PLB bus. Since there are 8 FU's, there are 8 interrupt lines going in to the INTC. Every FU has a fixed priority. Microblaze services the interrupt request based on those priorities. The interrupts raised by the PU register map and the PE top register map are control and debug interrupts. The ucm register map interrupt is the data related interrupt. When an interrupt is raised to the Microblaze, it checks whether it is an interrupt coming from the UCM or the PE and services it.

The interrupts are enabled at three levels - the interrupt controller level, the IPIF level and the PE level. At the boot up, the delta register in the PE RMAP needs to be reset and the enable register needs to be enabled by writing 1 to these addresses.

4.4 Microblaze MAC Tx IF

For MAC Tx to start transmitting frames, the Microblaze has to write the frames into the input buffer of the MAC Tx. The MAC Tx interface to the Microblaze environment is shown in the Figure 4.5. The figure also details the steps required. The Microblaze first writes the acknowledge, beacon and the data header into the header RAM of the MAC Tx which is a part of MAC Tx register map. The first word of the data header specifies the size of the data header. For 802.11b, the acknowledgement header is 10B, beacon header is 24B and the data header is 28B. The Microblaze then writes the payload information into the input buffer of the MAC Tx and updates the InDataPtr and InDataSize in the TD table for MAC Tx for the task *SendDataFrame*. To schedule this task, Microblaze writes a TDF into the asynchronous data queue of the TSQ.

To send multiple frames, the MAC Tx raises an interrupt once a frame has been transferred from its input buffer to the output buffer. This is the interrupt specified in the PE RMAP. The interrupt signals to the Microblaze to send the next frame. A second frame can be sent only when the interrupt has been received. A timer generates a

timeout if the interrupt does not arrive in time and the frame needs to be retransmitted.

4.5 Microblaze Sync IF

The Microblaze writes a control word into the input buffer of the sync PE. The control word provides the parameters required by the task RxStartRcvCtrl. To schedule this task, the Microblaze writes into the asynchronous control queue of the TSQ. The task processing in WINC2R system is command based, hence by writing the control word into the input buffer of the sync PE, we essentially trigger that block to start processing. At boot up, the block is in a non-processing state.

4.6 Microblaze Mac Rx IF

To interface the MAC Rx to the microblaze, the BRAM connected to the primary PLB behaves like just another PE. The BRAM is divided into TSQ, TD, Input Buffer and BRAM regions as shows in the Figure 4.6. The steps involved in interfacing them are also shown. After the MAC Rx finishes writing data into its output buffer, it raises the next task request to the UCM. The next task in this case in the *ProcessDataFrame*. The UCM updates the input data pointer and input data size in the TD region of the BRAM. The DMA transfer occurs between the MAC Rx and the BRAM and the data is transferred from the output buffer of the Mac Rx to the input buffer region of the BRAM. To schedule the task *ProcessDataFrame*, the UCM writes the TSF into the AsyncQ which is the TSQ region of the BRAM. When this is done, the UCM raises an interrupt to the Microblaze so that Microblaze can read the data from the input buffer region of the BRAM and process it.

4.7 Boot up sequence

When the Microblaze comes out of reset, it executes a series of reset and initialization routines required for the overall system flow. They are summarized as follows:

1. Read feature vector in PE Top RMAP: The feature vector contains information about the type of FPGA load - transmit or receive. The transmit/receive FPGA

load consists of the FU's in the transmit/receive chain respectively.

2. Initialize the ADCs, DACs and RF settings: These are the settings required for the RF front end.

3. Initialize the Global Task Descriptor Table (GTT): This the the global table that is common to all the PEs. It contains the configuration settings for the tasks as detailed in the Section 3.1.1

4. Initialize the Task Descriptor (TD) Table for every PE: The TD tables are local to every PE. Every task has an entry in the TD table. It contains the information about the buffers being used and the next tasks triggered by that task as detailed in the Section 3.1.2

5. Initialize the RMAP of every PE: Every PE requires certain settings as soon as the system boots up. These settings have been specified in the registers local to every PE and specified in the PE register map.

6. Initialize the PE Top RMAP: It contains the pointers to the output buffer regions in the PE.

7. Initialize UCM RMAP: UCM RMAP contains some configuration settings for the UCM.

8. Reset the IPIFs (Software Reset): The software reset of all the IPIFs is done by writing 0xA to a particular location in the register space of every IPIF.

9. Reset the Timer in the Microblaze environment: The timers in the Microblaze environment are initialized to a default value. They can be set through the command line as well by writing the command *SetTimer*.

10. Send packets to MAC Tx: This is done by writing a command *send_packet* through the command line. We can specify the number of packets that we want to send on the command line. How the packets are sent has been detailed in the Section **??**.

11. Trigger the Sync PE: The interface between the Microblaze and the Sync has been defined in the Section 4.5.

12. MAC Rx IF: The interface between the Microblaze and the MAC Rx has been detailed in the Section 4.6.

13. Interrupt handing: The interrupt raised by the MAC Tx on transferring a frame and the interrupt raised by the MAC Rx UCM after writing to the BRAM is handled by the software.

4.8 Command Line Interface

Currently, there are three commands which can be typed in from the command line which will execute the software.

- *GlobalReset*: This command executes the reset and the initialization routines of all the RMAPs, IPIFs, GTT, TDs and other components.

- *SetTimer*: This command sets the value of the timer 0 or 1. The usage is *SetTimer (value in hex) (Timer number).*

- *send_packet*: This command is used to start sending the frames. The usage is *send_packet (Number of packets).*

- *mr* and *mw*: These commands are used to read or write to a particular memory location on the FPGA. The usage of *mr* is *mr (location in hex) (No. of bytes).* The usage of *mw* is *mw (location in hex) (Value in hex).*

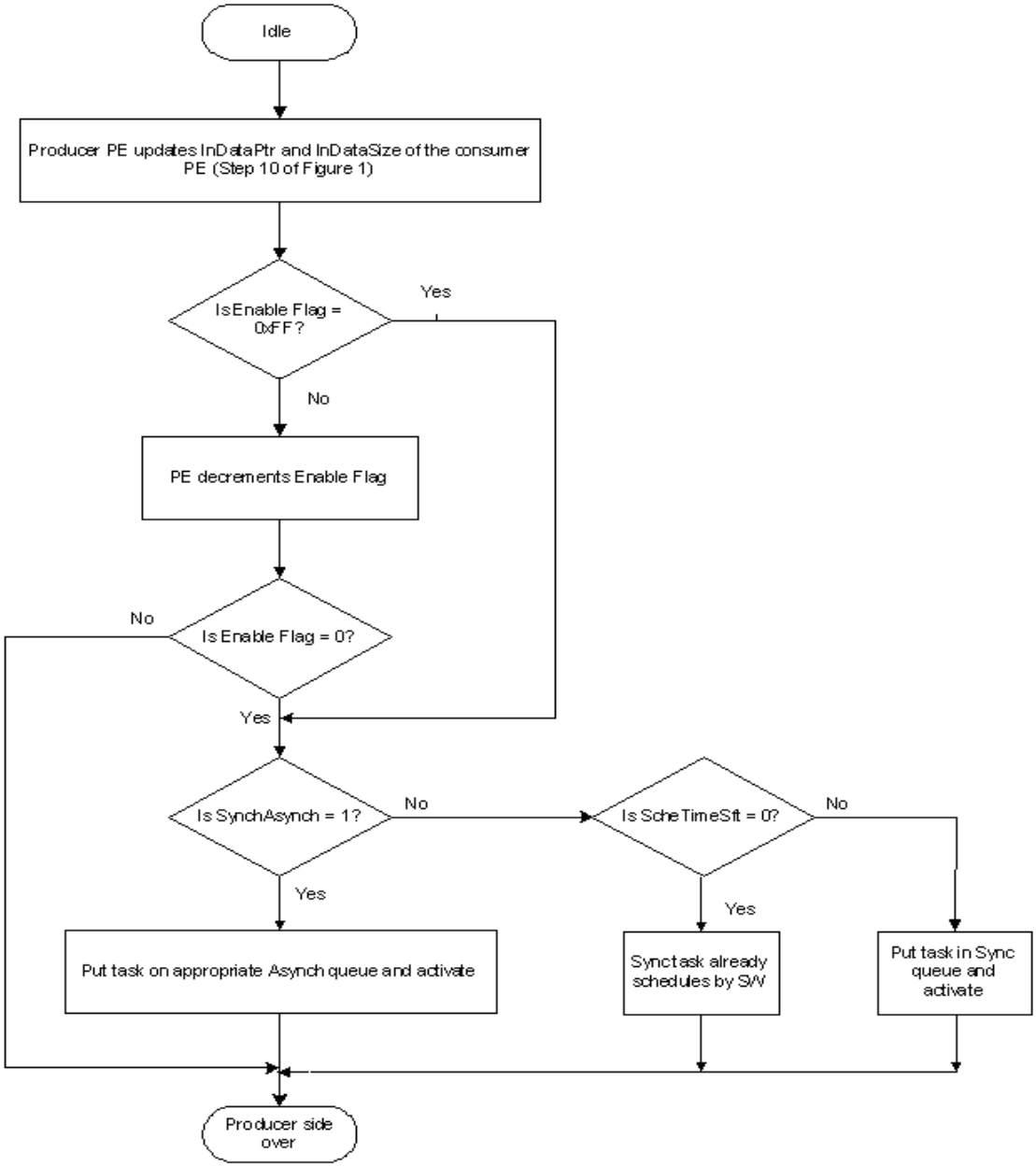


Figure 4.1: Producer side enable flag processing

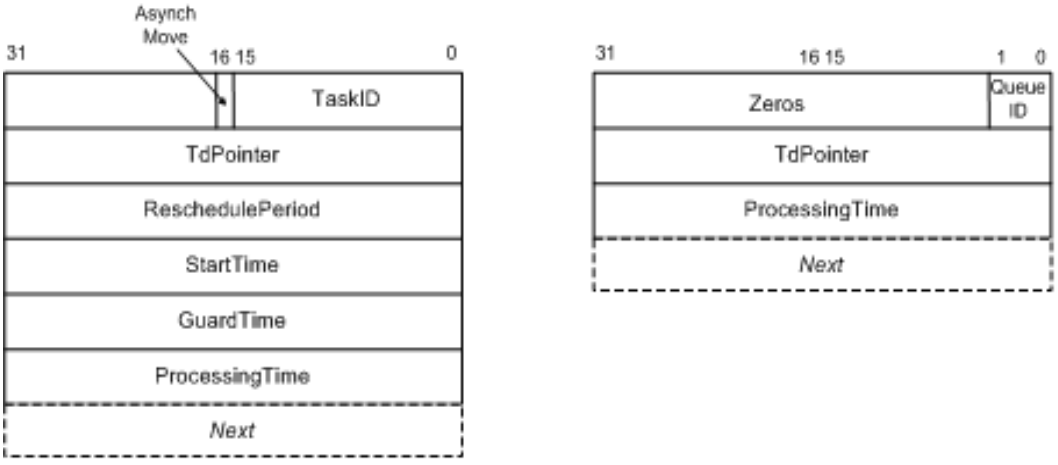


Figure 4.2: Task Descriptor Formats

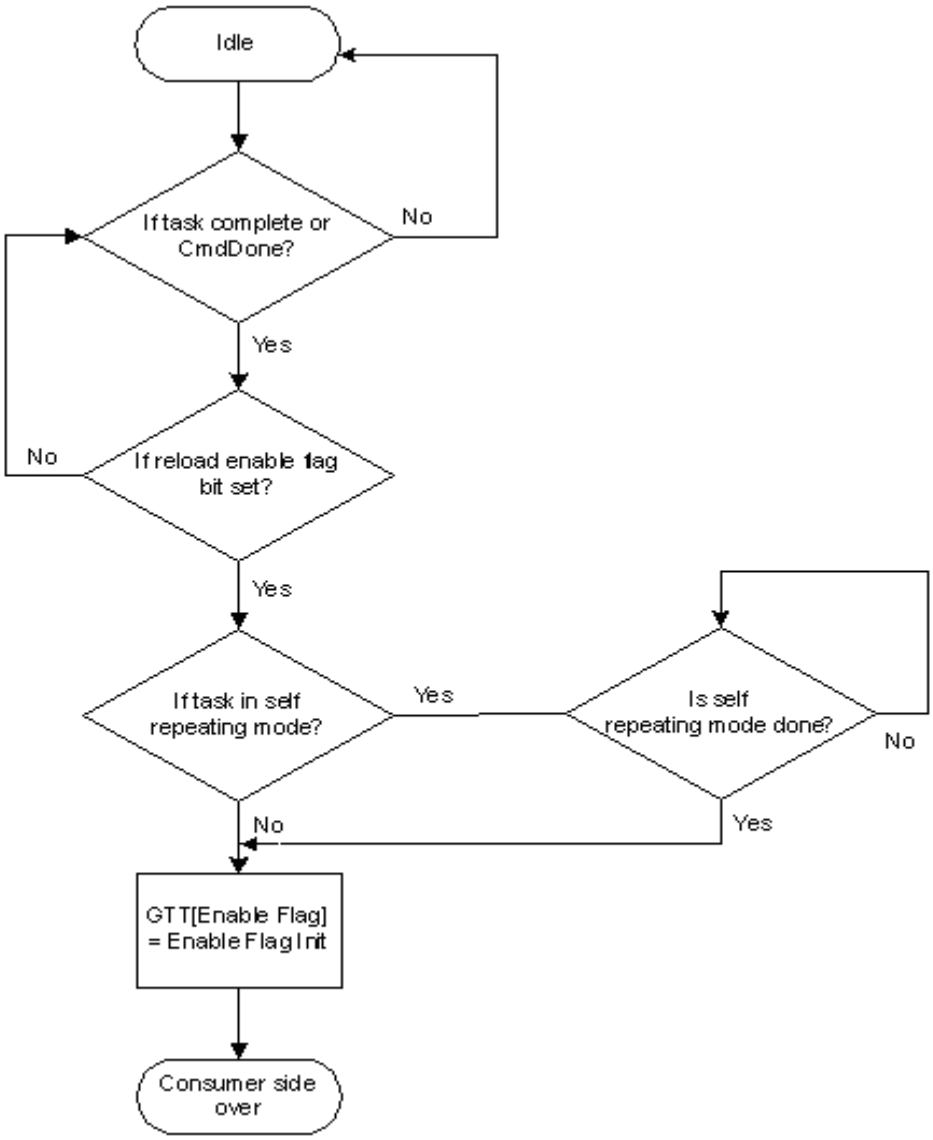


Figure 4.3: Consumer side enable flag processing

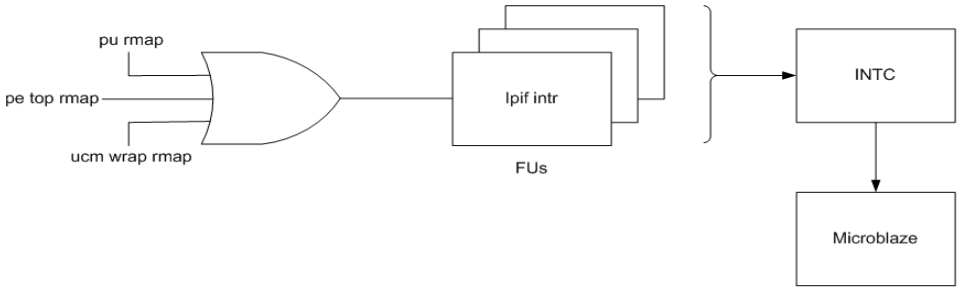


Figure 4.4: Interrupt Structure

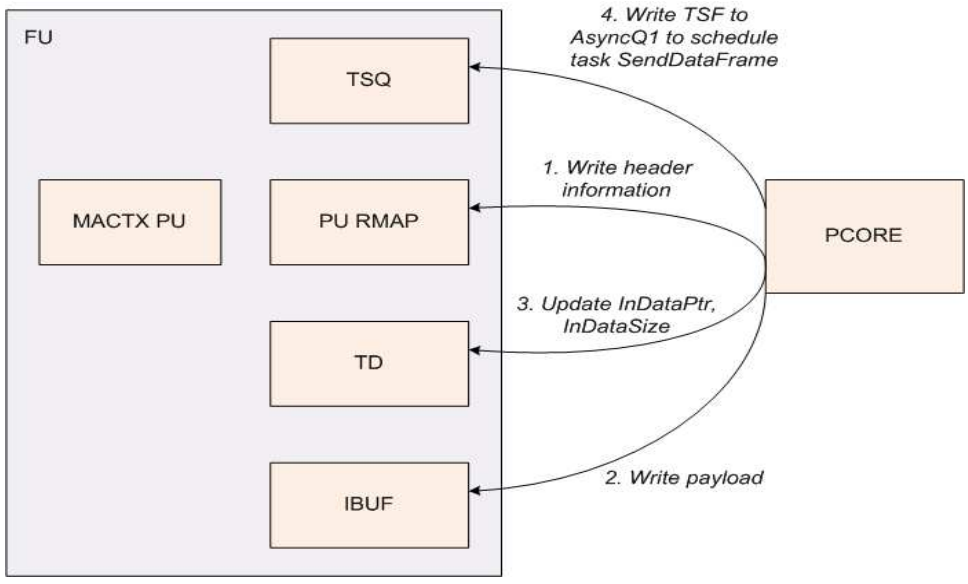


Figure 4.5: Interface between Mactx and pcore

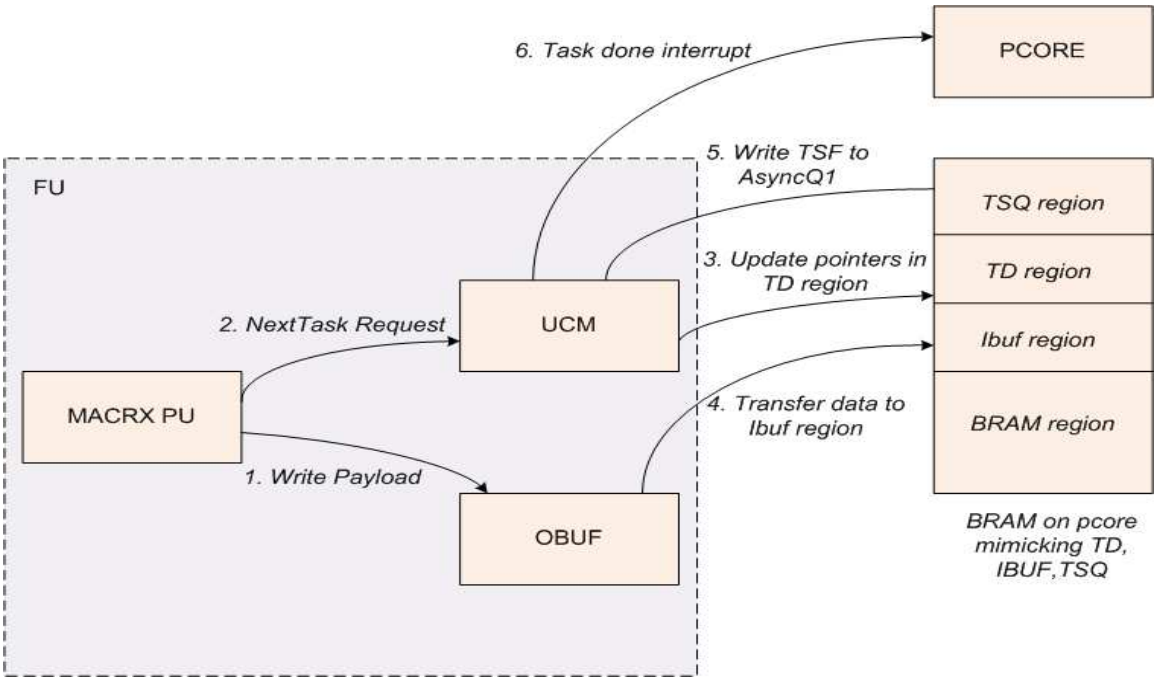


Figure 4.6: Interface between Macrx and pcore

Chapter 5

UCM Analysis

As shown in Figure 3.2, every functional module consists of a block called UCM (Unit Control Module). It is in charge of scheduling the tasks to the unit that it is associated with, assigning the task, monitoring the task completion, and communicating with the other units in the system for task sequencing. The task scheduling and sequencing have to be performed in a pipelined fashion - under the strict time and throughput constraints of the protocol.

5.1 UCM Architecture

The UCM architecture is as shown in the Figure 5.1.

The various blocks comprising the UCM and their functions are explained below:

- PE Controller: It activates the PE by sending a command. It acts as an interface between the PE and the other sub-blocks of the UCM.
- Task Flow Manager (TFM): It activates the next tasks as given by the Task Controller. It handles the enable flag processing for task synchronization. It also queues the asynchronous tasks into the Asynchronous Scheduler Queue.
- Scheduler: Processes the synchronous and asynchronous queues. The synchronous tasks have guarantied execution time slots with respect to the global timer. The asynchronous tasks are selected based on the fixed priority arbitration of 4 queues, where each queue is served in FIFO manner.
- DMA Arbiter: It arbitrates between the different blocks that need to perform the data transfer as each block (TC, TFM, Scheduler) operates independently.

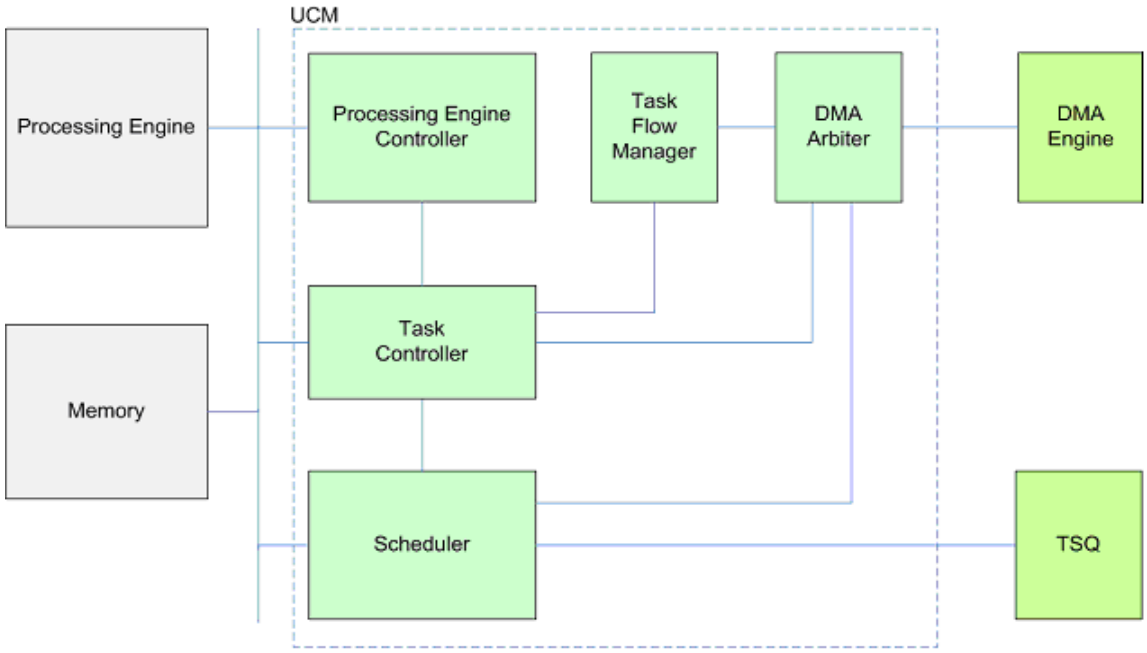


Figure 5.1: UCM Architecture

- Task Controller: Communicates with the other units in the system to move the data and control messages that facilitate execution through task sequencing.

UCM uses the global and local data structures for controlling its operation. It uses GTT which has been explained in Section 3.1.1. The GTT is shared by all the FUs. Every FU has its own scheduler queues which are dynamic data structures in the form of link lists. The FUs also have a local task flow graph which is a set of local data structures linked together through the Global Task Table. All the sub-modules in the UCM have a global address space allocated to them to make them easily accessible as shown in the Figure 3.3.

5.2 Task Processing

There can be two types of tasks that the UCM can process: data and control. The type of task is specified by the parameter TaskType. A 0 value indicates a data task and a 1 value indicates a control. Two data tasks cannot be processed simultaneously as

the second set of data would overwrite the previous one in the input buffer. The UCM calls several processing functions while processing a task. The functions in the order of being called are:

- Supervisory function (SUF): Supervisor maintains state information on which task is currently being processed. It provides the TdPointer and TaskType values along with the Activate signal. The individual processing functions latch the TdPointer and TaskType for internal use.
- Pre-fetch (pfetch) function: It fetches all the TD parameters. Those parameters are then distributed to Command Activation, Next Task and Command Termination functional modules.
- Command Activation (CA) function: It fetches the input data pointers and flow context. pFetch and CA functions are tightly coupled e.g. pFetch is always followed by CA processing.
- Next Task (NT) function: It is required for the data transfer between the producer and consumer. It fetches the next task parameters and the output data pointers and size.
- Command Termination (CT) function: It resets the flags to terminate the current task processing.

5.3 Analysis

Based on the system flow, there can be several possible task sequencing scenarios. A data/control task can be followed by data/control task. We have considered four possible scenarios which are: data task followed by control task, a data task followed by data task, a data followed by data followed by a control task, and control followed by control followed by another control task.

5.3.1 Task sequence - Data, Control

Figure 5.2 shows the time line for a data task followed by a control task.

As we can see, as soon as the scheduler schedules the data task, the supervisory function activates the data p-fetch function. After the task parameters have been fetched, the SUF activates the data CA function. The CA function updates the input buffer pointer sets and asserts the data command and sends an ack to the scheduler that the data pre-fetch was completed successfully. The data task processing now begins.

At this point, the scheduler schedules a control task. The SUF activates the control pre-fetch function and then activates the control command activation function. After the input buffer pointer sets have been updated, the control command is asserted and an ack sent to the scheduler that the control pre-fetch was completed successfully. After the control task has been processed, the PE controller sends a Next Task Request command which activates the NT function for the control task. After the NT function has been completed, an ack is sent back to the PE controller. The PE controller then sends a CmdAck command for control task, which triggers the CT function. On completion, a control command done ack is sent to the scheduler. Now, the NT function for the data task can be invoked followed by the CT function.

In this case, the NT function is followed by a CT function. There could be situations where NT and CT happen at the same time like when there is no next task to be triggered for a task. A data and a control task can be processed at the same time. However, two data tasks or two control tasks cannot be processed at the same time as the current input buffer pointers might get overwritten if the second task is being processed. The p-fetch function for two tasks also cannot happen at the same time. The scheduler shall not assert a new task processing request after receiving the control task, as the Task controller is currently handling two tasks already. If it ever happens then Task Controller shall flag an error condition and shall not process the request. This will result in a system error. Scheduler also maintains state information of which and how many tasks are currently being processed. Scheduler asserts a new task only after PE CmdDone flag has been set.

5.3.2 Task sequence - Data, Data

Figure 5.3 shows the time line for a data task followed by another data task.

As we see in the Figure, when the Data task 1 is scheduled by the scheduler, the p-fetch for the task is activated. After this the CA function is called and the input buffer pointers are updated. At this point the data command 1 is activated and the processing begins. At this point, the scheduler schedules a second data task. The p-fetch occurs for the second data task. Now the CA function cannot be invoked for the second data task as this will overwrite the input buffer current task pointer set region. Hence, the data 2 command assertion is stalled. The NT function followed by CT function for data 1 is called and on completion a command done ack is sent to the scheduler. At this point, the CA function for data 2 task is called and the data 2 task is processed in the similar manner to the data 1 task.

5.3.3 Task sequence - Data, Data, Control

Figure 5.4 shows the time line for the sequence - data, data, control tasks.

As we can see in the figure, once the task processing of the second data task has begun, if a control task is scheduled, its pre-fetching can start immediately. The data 2 task processing and control task processing can occur simultaneously. The PE controller sends the command done or command ack signal to the SUF for data and control tasks respectively. If the command done or command ack occur in the same clock cycle, then the PE controller will give priority to the control task. If the command done occurs few clock cycles before command ack, then the data task will be processed.

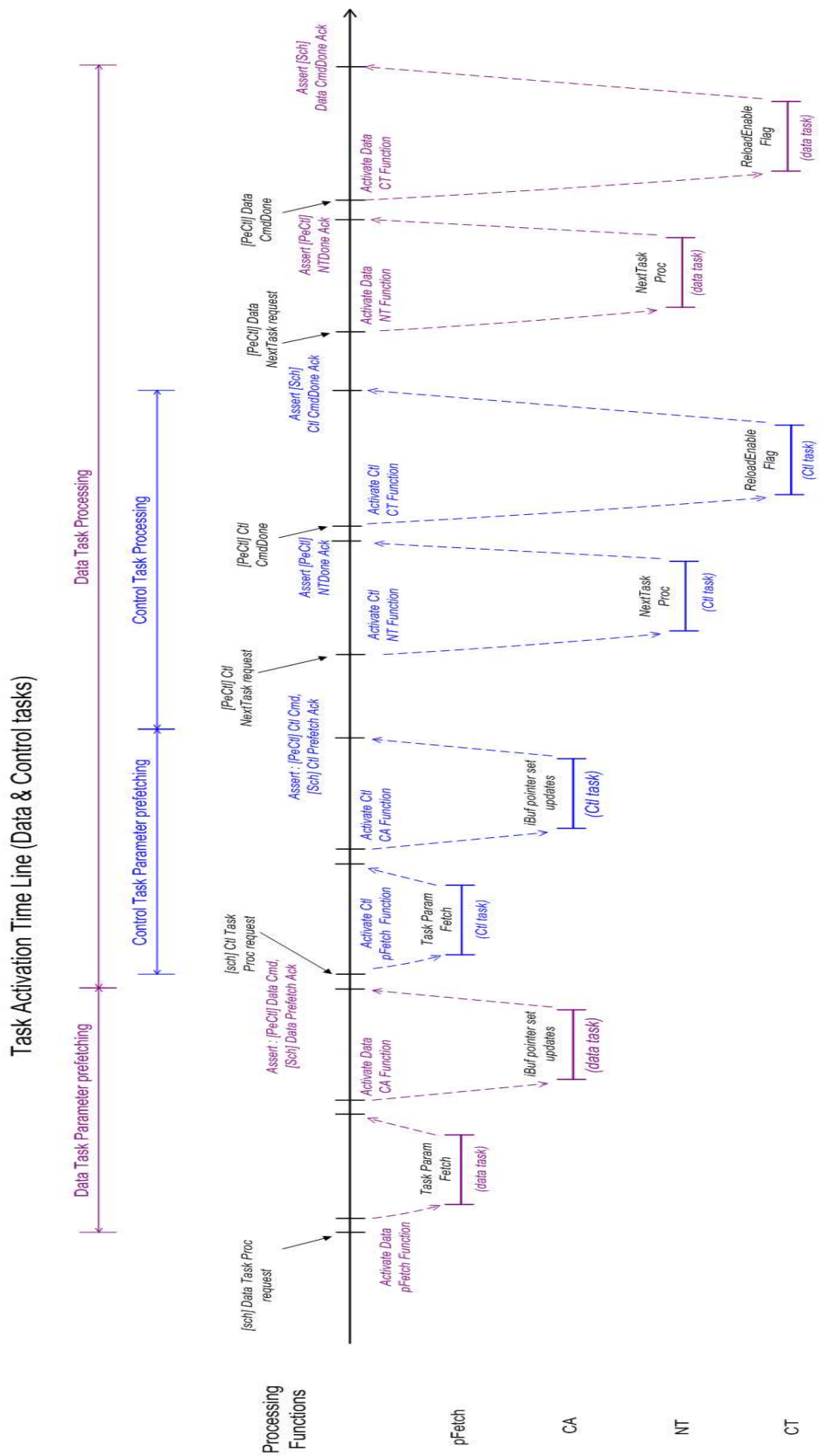
5.3.4 Task sequence - Control, Control, Control

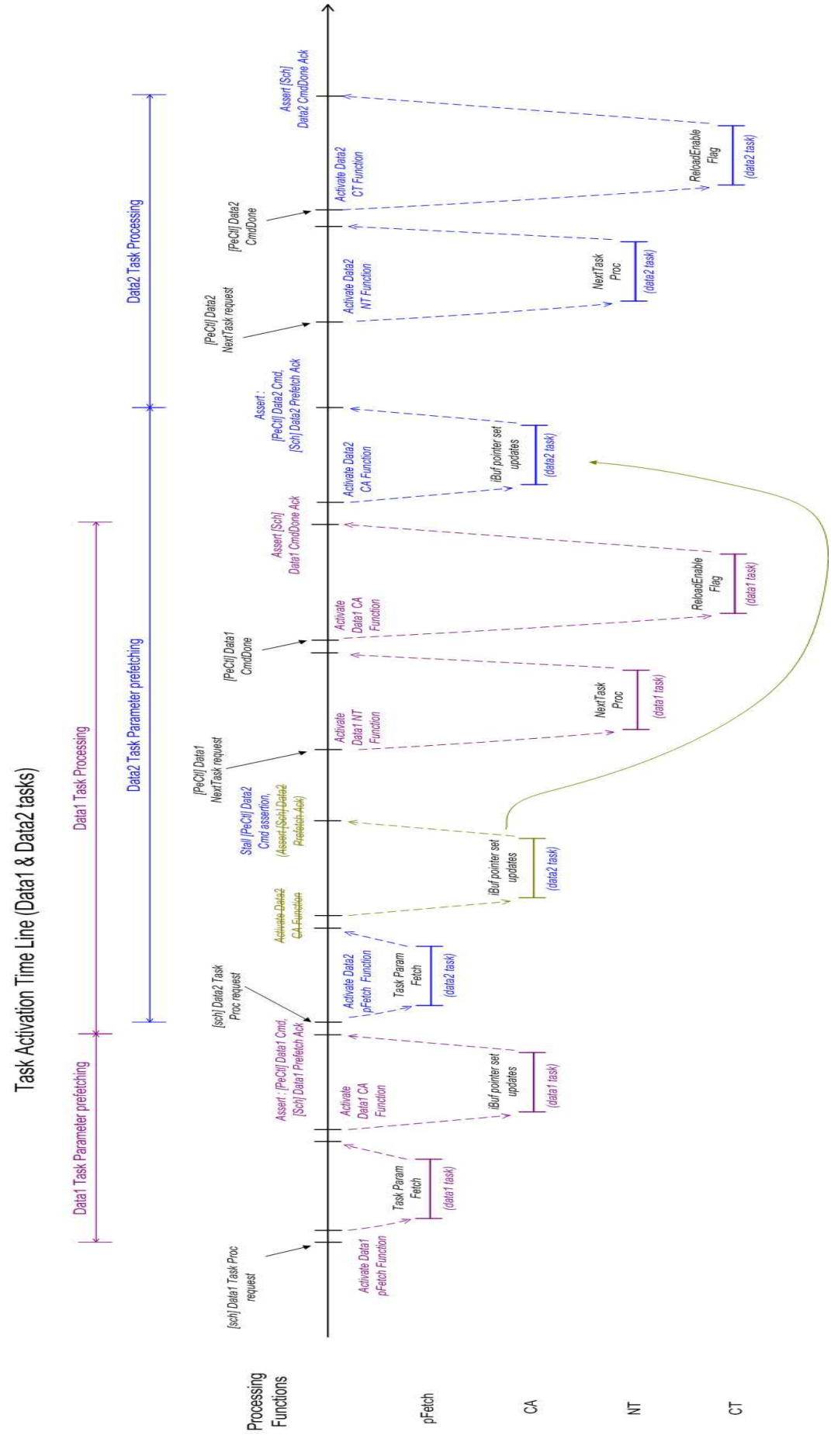
Figure 5.5 shows the time line for the sequence - control, control and another control tasks. The time line would be similar for control, control and a data task.

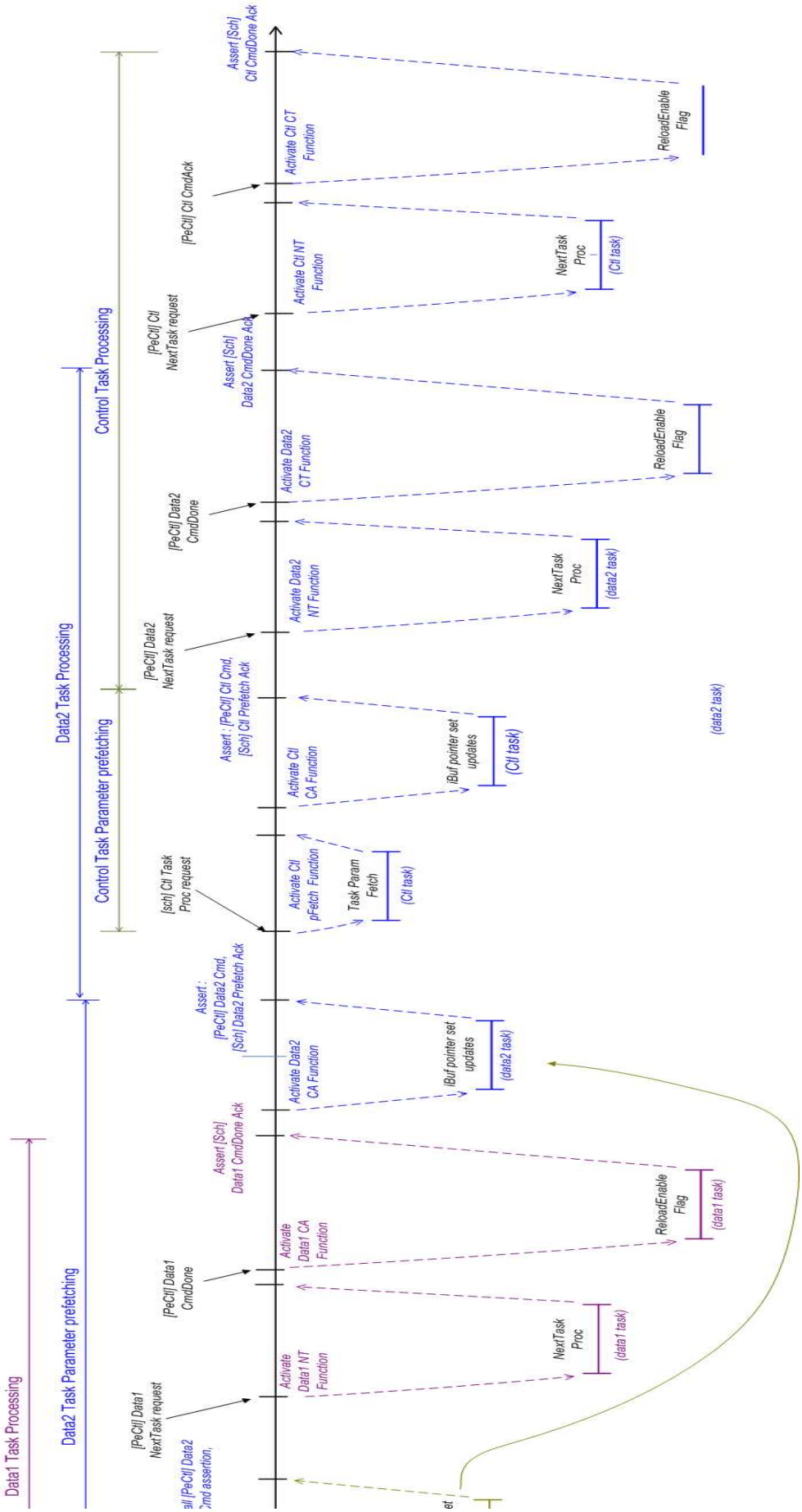
As we can see in the figure, the pre-fetch of control 3 task can occur simultaneously with the NT function processing of control 2 task since pre-fetch and NT functions are independent functions. However, the control task 3 cannot be activated until the PE controller sends the command ack signal. Pipelining the pre-fetch and NT function would save a few clock cycles.

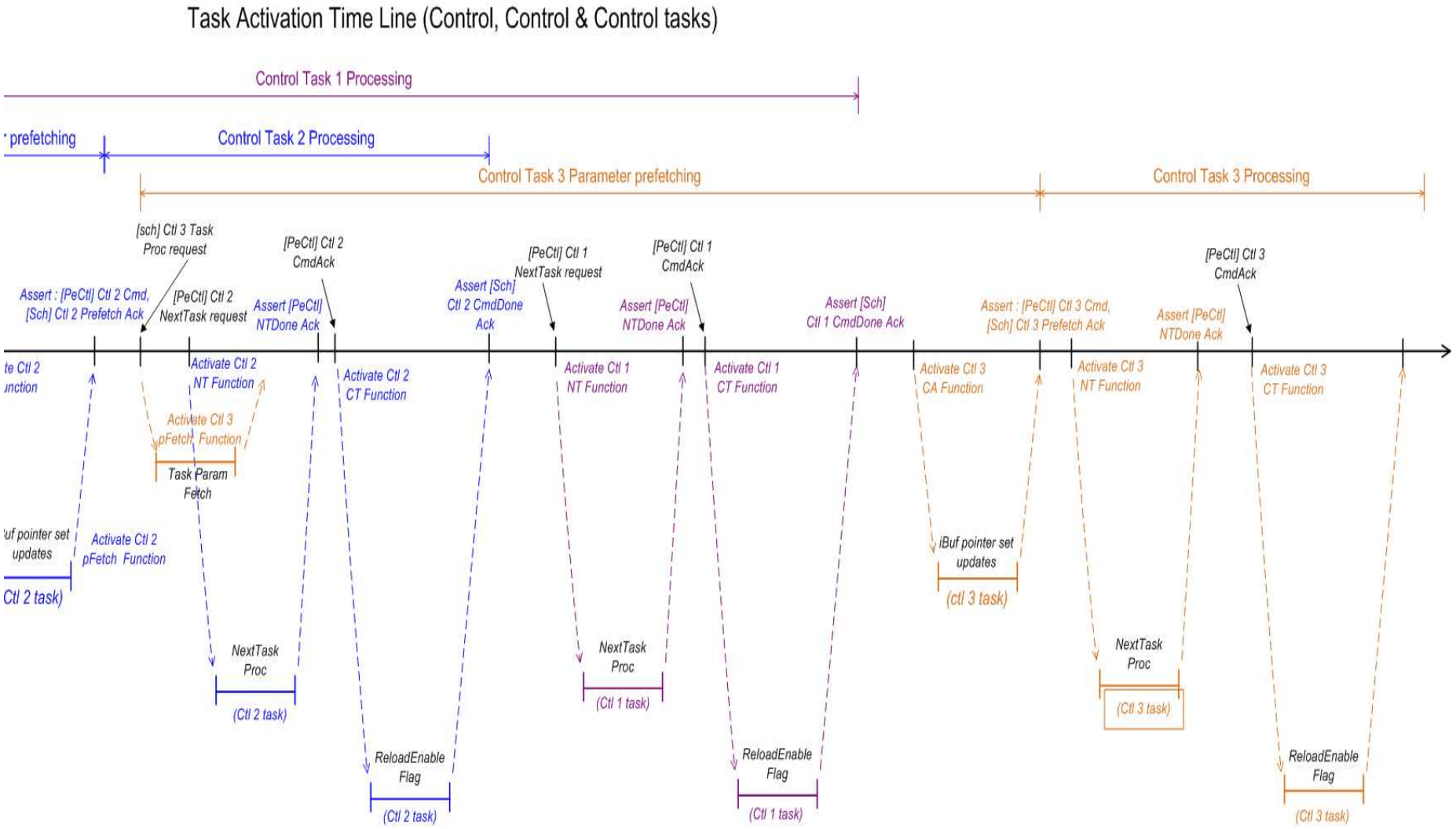
These scenarios will eventually be simulated and tested using the BFM simulations.

The simulations are already in progress. We can test completely once the UCM block is ready.


Task Activation Time Line (Data & Control tasks)
Data Task Parameter prefetching
Data Task Processing
Control Task Parameter prefetching
Control Task Processing
Processing Functions
pFetch
CA
NT
CT
[sch] Data Task Proc request
Activate Data pFetch Function
Task Param Fetch
(data task)
Activate Data CA Function
iBuf pointer set updates
(data task)
[sch] Ctl Task Proc request
Assert : [PeCtl] Data Cmd, [Sch] Data Prefetch Ack
Activate Ctl pFetch Function
Task Param Fetch
(Ctl task)
Activate Ctl CA Function
iBuf pointer set updates
(Ctl task)
[PeCtl] Ctl NextTask request
Assert : [PeCtl] Ctl Cmd, [Sch] Ctl Prefetch Ack
Activate Ctl NT Function
NextTask Proc
(Ctl task)
[PeCtl] Ctl CmdDone
Assert [PeCtl] NTDone Ack
Activate Ctl CT Function
ReloadEnable Flag
(Ctl task)
[PeCtl] Data NextTask request
Assert [Sch] Ctl CmdDone Ack
Activate Data NT Function
NextTask Proc
(data task)
[PeCtl] Data CmdDone
Assert [PeCtl] NTDone Ack
Activate Data CT Function
ReloadEnable Flag
(data task)
Assert [Sch] Data CmdDone Ack


Task Activation Time Line (Data1 & Data2 tasks)
Data1 Task Parameter prefetching
Data1 Task Processing
Data2 Task Parameter prefetching
Data2 Task Processing
Processing Functions
pFetch
CA
NT
CT
[sch] Data1 Task Proc request
Activate Data1 pFetch Function
Task Param Fetch
(data1 task)
Activate Data1 CA Function
iBuf pointer set updates
(data1 task)
Assert : [PeCtl] Data1 Cmd, [Sch] Data1 Prefetch Ack
[sch] Data2 Task Proc request
Activate Data2 pFetch Function
Task Param Fetch
(data2 task)
Activate Data2 CA Function
iBuf pointer set updates
(data2 task)
Stall [PeCtl] Data2 Cmd assertion, (Assert [Sch] Data2 Prefetch Ack)
[PeCtl] Data1 NextTask request
Activate Data1 NT Function
NextTask Proc
(data1 task)
[PeCtl] Data1 CmdDone
Activate Data1 CA Function
Assert [Sch] Data1 CmdDone Ack
ReloadEnable Flag
(data1 task)
Assert : [PeCtl] Data2 Cmd, [Sch] Data2 Prefetch Ack
[PeCtl] Data2 NextTask request
Activate Data2 NT Function
NextTask Proc
(data2 task)
[PeCtl] Data2 CmdDone
Activate Data2 CT Function
Assert [Sch] Data2 CmdDone Ack
ReloadEnable Flag
(data2 task)


Task Activation Time Line (Data1 & Data2 & Control tasks)
Data1 Task Processing
Data2 Task Parameter prefetching
Control Task Parameter prefetching
Data2 Task Processing
Control Task Processing
[PeCtl] Data1 NextTask request
[PeCtl] Data1 CmdDone
Activate Data1 NT Function
NextTask Proc
(data1 task)
Activate Data1 CA Function
Assert [Sch] Data1 CmdDone Ack
ReloadEnable Flag
(data1 task)
Activate Data2 CA Function
Assert : [PeCtl] Data2 Cmd, [Sch] Data2 Prefetch Ack
iBuf pointer set updates
(data2 task)
[Sch] Ctl Task Proc request
Activate Ctl pFetch Function
Task Param Fetch
(Ctl task)
Activate Ctl CA Function
Assert : [PeCtl] Ctl Cmd, [Sch] Ctl Prefetch Ack
iBuf pointer set updates
(Ctl task)
[PeCtl] Data2 NextTask request
Activate Data2 NT Function
NextTask Proc
(data2 task)
[PeCtl] Data2 CmdDone
Activate Data2 CT Function
Assert [Sch] Data2 CmdDone Ack
ReloadEnable Flag
(data2 task)
[PeCtl] Ctl NextTask request
Activate Ctl NT Function
NextTask Proc
(Ctl task)
[PeCtl] Ctl CmdAck
Activate Ctl CT Function
Assert [Sch] Ctl CmdDone Ack
ReloadEnable Flag


Task Activation Time Line (Control, Control & Control tasks)
Control Task 1 Processing
prefetching
Control Task 2 Processing
Control Task 3 Parameter prefetching
Control Task 3 Processing
[sch] Ctl 3 Task Proc request
[PeCtl] Ctl 2 CmdAck
[PeCtl] Ctl 1 NextTask request
[PeCtl] Ctl 1 CmdAck
[PeCtl] Ctl 3 CmdAck
Assert : [PeCtl] Ctl 2 Cmd, [Sch] Ctl 2 Prefetch Ack
[PeCtl] Ctl 2 NextTask request
Assert [PeCtl] NTDone Ack
Assert [Sch] Ctl 2 CmdDone Ack
Assert [PeCtl] NTDone Ack
Assert [Sch] Ctl 1 CmdDone Ack
Assert : [PeCtl] Ctl 3 Cmd, [Sch] Ctl 3 Prefetch Ack
Assert [PeCtl] NTDone Ack
te Ctl 2 unction
Activate Ctl 2 NT Function
Activate Ctl 2 CT Function
Activate Ctl 1 NT Function
Activate Ctl 1 CT Function
Activate Ctl 3 CA Function
Activate Ctl 3 NT Function
Activate Ctl 3 CT Function
Activate Ctl 3 pFetch Function
Task Param Fetch
uf pointer set updates
Activate Ctl 2 pFetch Function
Ctl 2 task)
NextTask Proc
(Ctl 2 task)
ReloadEnable Flag
(Ctl 2 task)
NextTask Proc
(Ctl 1 task)
ReloadEnable Flag
(Ctl 1 task)
iBuf pointer set updates
(ctl 3 task)
NextTask Proc
(Ctl 3 task)
ReloadEnable Flag
(Ctl 3 task)

Chapter 6

Conclusion

In this research, we have designed a WINC2R hardware platform using Xilinx FPGA board. The FPGA based architecture achieves our goals of flexibility along with being customizable for system connectivity, DSP, and/or data processing applications. The Xilinx Platform Studio provides us with an interface to IBM system interconnect - PLB bus structure. The various components were connected together and an interrupt handler was written to handle the communication between them. The design was made so as to minimize the latency and maximize the utilization. The initial design was built on Virtex 4 using the internal memory. To overcome the constraint of the small internal memory size, we used the external memories. This allowed us to conserve the resources on Virtex 4. We developed the software for interfacing the functional modules and the Microblaze. Software was also written to assist the UCM to perform the task processing and the DMA transfer in between the PE's.

The Mac layer functionalities have not been incorporated into the software as yet. The command SendDataFrame from the Mac Tx should involve the processing of the ack received from the Mac Rx. It should have the capability of retransmissions on the loss of frames. Thus, the Mac and higher layers still need to be worked on. The design and development will be undertaken as part of future work.

We proposed the entire system and task processing flow as explained in the Section **??**. The Microblaze environment was created as a separate functional module and invokes the other functional modules by passing parameters. After doing a BRAM utilization analysis and the FPGA resource utilization analysis, we decided to migrate to Virtex 5 board to meet our WINC2R design requirements. The migration of the design to the Virtex 5 board is in the process. Virtex 5 board has more memory and

higher FPGA resources available which can easily contain the WINC2R platform.

The UCM is a part of every FU. It is in charge of scheduling the tasks to the unit that it is associated with, assigning the command (task), monitoring the task completion, and communicating with the other units in the system for task sequencing. The sequence of tasks have to complete under strict time frame constraints. We analyzed the UCM block and analyzed the possible task sequencing scenarios which could occur in the WINC2R system. We have proposed the timelines for the task processing. These time lines would be simulated and analyzed with respect to the number of clock cycles used as part of the future work. The simulations are already in progress to test the task scheduler queue block. In the future all the blocks of the UCM would be connected together and analyzed.

Appendix A

BRAM Estimate for Virtex 4 and Virtex 5

A.1 BRAM Estimate

Xilinx Virtex 4 board, XC4VSX35 has 192 Block RAMs (BRAM), 18Kbits each. The Microblaze design and the PEs need a certain number of BRAM depending upon their memory requirements. An estimate of the BRAM utilization in the WINC2R system for Virtex 4 board is shown below.

PE BRAM utilization:

- In Buffer (16Kx8) : 8
- Out Buffer (16Kx8) : 8
- Task Descriptor table (8Kx8) : 4
- Task Scheduler Queue (8Kx8) : 4

Total per PE :24

Taking 4 PEs, we need 4 x 24 = 96 BRAMs.

It should be kept in mind that this is the common BRAM usage by the PE, the individual PU may need additional BRAMs for its register map, but it will be lesser in comparison to the common BRAM requirement. Also, right now we are only considering 4 PEs, whereas in the WINC2R system we have 8 PEs.

Microblaze Design, Instruction/data BRAM utilization: 104 BRAMs

Hence, with this NCP configuration, the total BRAM utilization = 104 + 96 = 200 BRAMs (+ a few BRAMs for the PU specific applications).

We can see that already the BRAM utilization exceeds the number of available BRAMs in Virtex 4 which is 192. Hence, the BRAMs are over mapped and the utilization is 104%. There is also no room for internal probing using Chipscope for debugging. Chipscope uses BRAM resources to store the internal sample values. An important thing to note about here is that in this case we were using the internal memory for Microblaze instructions and data. This is because EDK tool has a default linker script and boot code when using the internal memory, hence making it convenient.

To overcome this over utilization, we decided to use the external memory on the board (DDR SDRAM and Flash Memory). By using them we move the instruction code onto the flash memory. To execute the code, we copy the code from the flash memory to the DDR SDRAM and execute it. We use a new linker script and a bootloader which accomplishes the task of moving the code from flash memory to DDR SDRAM.

After using the external memories, the BRAM utilization for the Microblaze design becomes 83 BRAMs. Hence, the total BRAM requirement for the microblaze and the PEs becomes 179 BRAMs which is still very close to the limit of 192.

Hence, the need to migrate to the Virtex 5 board which has much more number of BRAMs available so that our design can fit in comfortably with a lot of margin. The Virtex 5 board that we are in the process of migrating to is XC5VSX95 which has 244 BRAMs, 36 Kbits each.

An estimate of the BRAM utilization in the WINC2R system for Virtex 5 board is shown below.

PE BRAM utilization:

- In Buffer(16Kx8) : 4
- Out Buffer(16Kx8) : 4
- Task Descriptor table (8Kx8) : 2
- Task Scheduler Queue (8Kx8) : 2

Total per PE :12

Taking 4 PEs, we need 4 x 12 = 48 BRAMs.

Microblaze Design, Instruction/data BRAM utilization: 38 BRAMs

With this NCP configuration, the total BRAM utilization = 38 + 48 = 86 BRAMs (+ a few BRAMs for the PU specific applications). This is much less than 244 and the utilization is only 36%.

References

[1] IEEE Standard 802.20 *http://grouper.ieee.org/groups/802/20/*

[2] J. Mitola, Cognitive Radio: An Integrated Agent Architecture for Software Radio, PhD thesis, Royal Institute of Technology, Sweden, May 2000.

[3] Multimode Cell Phones *http://electronics.howstuffworks.com/cell-phone8.htm*

[4] Vanu *http://www.vanu.com/*

[5] GNURadio, *http://www.gnu.org/software/gnuradio/*

[6] Rice University, *http://warp.rice.edu/*

[7] ISR Technologies, *http://www.isr-technologies.com/JTRS_SDR_eng.shtml*

[8] picoChip User Manual, picoChip Designs Ltd., UK.

[9] Steve Liebson, *The End of Moore's Law*, Embedded System Design

[10] Z. Miljanic, I. Seskar, K. Le, and D. Raychaudhuri, WINLAB Network Centric Cognitive Radio Hardware Platform - WiNC2R, in *Proceedings of International Conference on Cognitive Radio Oriented Wireless Networks and Communications*, Orlando, Florida, 2007.

[11] *https://local.winlab.rutgers.edu/projects/cognitive/wiki/WiNC2R*

[12] Embedded System Tools Reference Manual, EDK 9.2i.

[13] Xilinx documentation, UG081, *Microblaze Processor Reference Guide, EDK 9.2i*

[14] Xilinx documentation, DS569, *XPS General Purpose Input/Output v1.00a*

[15] Xilinx documentation, DS572, *XPS Interrupt Controller v1.00a*

[16] Xilinx documentation, DS573, *XPS Timer/Counter v1.00a*

[17] Xilinx documentation, DS571, *XPS UART Lite v1.00a*

[18] Xilinx documentation, DS641, *XPS Microblaze Debug Module v1.00a*

[19] Xilinx documentation, DS618, *XPS PLB v46 to PLB v46 Bridge v1.00a*

[20] Xilinx documentation, DS575, *XPS Multi Channel External Memory Controller v1.00a*

[21] Xilinx documentation, DS643, *XPS Multi Port Memory Controller v3.00.a*

[22] Xilinx documentation, DS596, *XPS Block RAM Interface Controller v1.00a*

[23] Xilinx documentation, DS444, *XPS Block RAM v1.00a*

[24] Atmel documentation, AT49BV322A, *Atmel 32-Mbit 3-Volt only Flash Memory*

[25] Infineon Technologies, HYB25D512160BC-6, *Hynix 512-Mbit Double Data Rate (DDR) SDRAM*

[26] Xilinx Application Notes: Flash Memory Bootloading Using SPI with Spartan-3A DSP 1800A Starter Platform.

[27] IBM Core-Connect Bus Architecture, *http://www-03.ibm.com/technology/power/licensing/coreconnect/index.html*