The University of Texas at
TYLER
The Graduate School

University of Texas at Tyler
Scholar Works at UT Tyler

Electrical Engineering Theses

Electrical Engineering

Fall 8-2011

A Study of Multiprocessor Systems using the Picoblaze 8-bit Microcontroller Implemented on Field Programmable Gate Arrays

Venkata Mandala

Follow this and additional works at: https://scholarworks.uttyler.edu/ee_grad

Part of the Electrical and Computer Engineering Commons

Recommended Citation
Mandala, Venkata, "A Study of Multiprocessor Systems using the Picoblaze 8-bit Microcontroller Implemented on Field Programmable Gate Arrays" (2011). *Electrical Engineering Theses.* Paper 21.
http://hdl.handle.net/10950/59

This Thesis is brought to you for free and open access by the Electrical Engineering at Scholar Works at UT Tyler. It has been accepted for inclusion in Electrical Engineering Theses by an authorized administrator of Scholar Works at UT Tyler. For more information, please contact tbianchi@uttyler.edu.

The University of Texas at
TYLER
The Graduate School

A STUDY OF MULTIPROCESSOR SYSTEMS USING THE PICOBLAZE 8-BIT MICROCONTROLLER IMPLEMENTED ON FIELD PROGRAMMABLE GATE ARRAYS

by

VENKATA CHANDRA SEKHAR MANDALA

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Electrical Engineering
Department of Electrical Engineering

David H. K. Hoe, Ph.D., Committee Chair

College of Engineering and Computer Science

The University of Texas at Tyler
August 2011

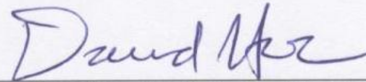
The University of Texas at Tyler
Tyler, Texas

This is to certify that the Master's thesis of

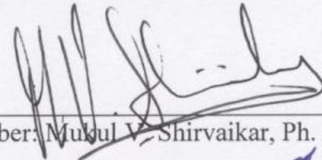
VENKATA CHANDRA SEKHAR MANDALA

has been approved for the thesis requirements on
July 13, 2011
for the Master of Science Degree in Electrical Engineering

Approvals:

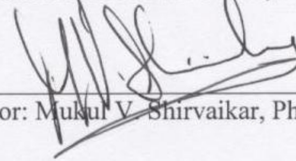


Thesis Chair: David H. K. Hoe, Ph.D.

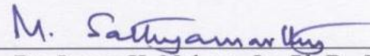


Member: Mukul V. Shirvaikar, Ph. D.

Member: Hector A. Ochoa, Ph.D.



Chair and Graduate Coordinator: Mukul V. Shirvaikar, Ph. D.



Dr. James K. Nelson, Jr., Ph.D., P.E.,
Dean, College of Engineering and Computer Science,
Brazzel Professor of Engineering

Acknowledgements

First of all, I am thankful to my father Mandala Maheswara Rao and mother Suguna Kumari and sister Geetha Devi for making my dream come true. I would like to thank Dr. David Hoe, my thesis advisor, for his support in my research in different areas of Electrical Engineering. I would also thank him for his encouragement, patience and supervision from the preliminary stages to the concluding level in my thesis. Without his guidance and persistent help this thesis would not have been possible.

I would like to thank my committee members, Dr. Hector A. Ochoa and Dr. Mukul V. Shirvaikar for taking the time to review my work. I am thankful to Dr. Mukul Shirvaikar for supporting and guiding me throughout my Master's degree. I would like to thank the entire Electrical Engineering department and the University of Texas at Tyler for supporting me throughout my Master's degree. Also, I would like to thank Chris Martinez, an undergraduate student in Electrical Engineering, for helping me in my thesis. Finally, I would like to thank all those who supported me in any respect during the completion of the thesis.

Table of Contents

List of Figures iv
List of Tables vi
Abstract vii
Chapter One: Introduction 1
1.1 Soft Processor Cores on FPGA 1
1.2 Research Objectives 2
1.3 Research Method 3
1.4 Thesis Outline 3
Chapter Two: Background 4
2.1 Trend Towards Multicore Processors 4
2.1.1 Power Wall Problem 4
2.2 Multicore Processor 7
2.2.1 Array of Processors 8
2.2.2 Array of Functional Units 9
2.3 Reconfigurable Computing 11
2.3.1 Advantages of Reconfigurable Computing 11
2.3.2 FPGAs 11
2.3.2.1 Microcontroller within an FPGA 11
2.3.3 Array of Soft Processors 12
2.3.4 PicoBlaze Microcontroller 13
2.4 Interprocessor Communication 15
2.5 Summary 16
Chapter Three: FIFO Style Interprocessor Communication 17
3.1 FIFO Style Communication 17
3.1.1 FIFO 17
3.1.2 PicoBlazes with a FIFO Buffer 19
3.1.3 Experimental Verification of the FIFO with Two PicoBlazes 20

3.2 Array of PicoBlazes 20
3.2.1 The Wrapper 20
3.2.2 Need for an Array of PicoBlaze Units 24
3.2.3 Details of the Array 25
3.2.4 Routing/Logic Cell Resources Comparison 26
3.3 Experiment with Array of PicoBlazes 26
3.3.1 Counting Numbers Experiment 27
3.3.2 Status Signals and Read/Write Strobe Conditions Setup 27
3.3.3 Validation and Results 28
3.4 Application (A 4-tap FIR Filter) 29
3.4.1 Algorithm and Simulation of FIR 29
3.4.2 Array Structural Flow For FIR 29
3.4.3 Validation and Results 31
3.5 Summary 33
Chapter Four: Shared Memory Inter-Processor Communication 34
4.1 Shared Memory 34
4.1.1 Shared Memory Mode of Communication 34
4.1.2 Sharing BlockRAM Between Two PicoBlazes 35
4.1.3 Shared Memory for Communication Between PicoBlaze Processors 36
4.2 Need for an Arbiter 36
4.2.1 Arbitration 37
4.3 Round-Robin Arbiter 37
4.3.1 Implementation of a Round-Robin Bus Arbiter 38
4.3.2 Round-Robin Arbiter with Four Processors 39
4.4 PicoBlazes Using Round-Robin Arbiter 40
4.4.1 Four PicoBlazes Using Round-Robin Arbiter 40
4.4.2 Implementation Description of Shared Memory with Round-robin Arbiter 40
4.4.3 Observations 42
4.5 Summary 43
Chapter Five: Conclusions and Future Work 44

5.1 Conclusion 44

5.2 Future Work 44

References 46

Appendix A: Assembly Code for Button Press Experiment 50

A.1 PicoBlaze 1 to FIFO Assembly Code for Button Press Experiment 50

A.2 FIFO to PicoBlaze 2 Assembly Code for Button Press Experiment 51

Appendix B: Assembly Code for Counting Numbers Experiment 53

B.1 PicoBlaze 4 Assembly Code for Counting Numbers Experiment 53

B.2 PicoBlaze 1 (P1) Assembly Code for Counting Numbers Experiment 55

B.3 PicoBlaze 2 Assembly Code for Counting Numbers Experiment 56

B.4 PicoBlaze 3 Assembly Code for Counting Numbers Experiment 57

B.5 PicoBlaze 5 Assembly Code for Counting Numbers Experiment 58

Appendix C: Code for 14 PicoBlazes FIR Filter Experiment 59

C.1 N-tap and T-input FIR Filter Code in C Language 59

C.2 PicoBlaze Array Code in VHDL for FIR Filter Design 60

C.3 Four FIFO Wrapper Design Code in VHDL 79

C.4 PicoBlaze 1 to PicoBlaze 10 Assembly Code for FIR Filter Design 83

C.5 PicoBlazeU0 to PicoBlazeU3 Assembly Code for FIR filter Design 86

Appendix D : *Spartan 3E* and *Virtex 5* Statistics 87

D.1 Logic Cells LEGEND Color on *Virtex 5* and *Spartan 3E* FPGA 87

D.2 Occupation of Logic Cell Resources on *Virtex 5* FPGA 87

D.3 Occupation of Logic Cell Resources on *Spartan 3E* FPGA 88

D.4 Table Showing General Comparison of *Virtex 5* and *Spartan 3E* FPGAs 88

Appendix E: Arbitration Schemes 89

E.1 Different Arbitration Schemes: (A Section from Chapter4) 89

Appendix F: Delay Statistics for *Spartan 3E* and *Virtex 5* 91

F.1 Place and Route Delay Statistics for *Spartan 3E* and *Virtex 5* FPGAs 91

F.2 Synthesis (Pre-Routing) Report Statistics for *Spartan 3E* and *Virtex 5* FPGAs 92

List of Figures

Figure 2.1: Trend in clock rate and power for Intel uniprocessors (single core)........... 5

Figure 2.2: Plot showing supply voltage (V_{dd}) and threshold voltage (V_{th}) scaling transistor size (L_{eff})……………………………………………………………….. 7

Figure 2.3: Array of processors where PR is Processor and MEM is memory............. 9

Figure 2.4: Array of Functional Units (FU)…………………………………………... 10

Figure 2.5: Array of Soft Processors (CN stands for Computation Nodes).................. 13

Figure 2.6: PicoBlaze Microcontroller Block Diagram………………………………. 14

Figure 2.7: KCPSM3 Assembler Files……………………………………………….. 15

Figure 3.1: Theoretical view of a FIFO Buffer……………………….……………… 18

Figure 3.2: Three Possible Cases of a circular FIFO Buffer………………………….. 18

Figure 3.3: Pseudo code for FIFO *write* and *read* cases……………………………… 19

Figure 3.4: Two PicoBlazes (PB1 and PB2) communicating through a FIFO Buffer... 19

Figure 3.5: Different topologies in designing a Wrapper…………...………………... 21

Figure 3.6: Wrapper design view in schematic editor (1, 4 and 8 FIFOs from left)… 21

Figure 3.7: Detailed Wrapper with four FIFO buffers and a PicoBlaze Processor…… 22

Figure 3.8: Port map for the Wrapper entity in VHDL…………………………….. 23

Figure 3.9: Wrapper with port id coded in VHDL…………………………………… 24

Figure 3.10: A wrapper that connects to four neighboring Processors……………….. 25

Figure 3.11 Experiment setup with 5 PicoBlazes where PicoBlaze 4 is destination…. 27

Figure 3.12: Destination PicoBlaze holding data in FIFOs…………………………... 28

Figure 3.13: A 4-tap FIR Filter Application on Array of PicoBlaze Units…………… 31

Figure 4.1: Two PicoBlazes sharing one Block RAM……………………………….. 35

Figure 4.2: Two PicoBlazes using two separate 512- instruction memory sections from BRAM……………………………………………………………………... 36

Figure 4.3: Round-robin Arbiter……………………………………………………… 39

Figure 4.4: Round-robin Arbiters with Four Processors……………………………… 40

Figure 4.5: Four PicoBlazes with Shared RAM mode communication using Bus

Arbiter………………………………………………………………………………… 41
Figure D-1: Legend colors for 14 PicoBlaze cores on FPGA...………………………. 87
Figure D-2: Occupation of logic cells on *Virtex 5* FPGA……………………………. 87
Figure D-3: Occupation of logic cells on *Spartan 3E* FPGA………………………… 88
Figure F-1: delay for *Spartan 3E* with 14 PicoBlazes (FIFO) setup…………………. 91
Figure F-2: delay for *Virtex 5* with 14 PicoBlazes (FIFO) setup…………………….. 91
Figure F-3: delay for *Spartan 3E* with 4 Picobalzes (shared memory) setup………… 91
Figure F-4: delay for Virtex 5 with 4 PicoBlazes (shared memory) setup…………… 91
Figure F-5: Synthesis delay report for *Spartan 3E* with 14 PicoBlaze setup……….... 92
Figure F-6: Synthesis delay report for *Virtex 5* with 14 PicoBlazes setup…………… 92
Figure F-7: Synthesis delay report for *Spartan 3E* with Shared memory setup……… 93
Figure F-8: Synthesis delay report for *Virtex 5* with 4 Shared memory setup……….. 93

List of Tables

Table 3.1: Statistics for the 1, 4, and 8 FIFO Wrapper in *Spartan 3E*………………….. 26

Table 3.2: Statistics for the 1, 4, and 8 FIFO Wrapper in *Virtex 5*……………………... 26

Table 3.3: Logic cell Statistics of the *Spartan 3E* and *Virtex 5* FPGAs implementing an array of fourteen PicoBlazes………………………………………………………………. 32

Table 4.1: Logic cell Statistics of the *Spartan 3E* and *Virtex 5* FPGAs implementing four PicoBlaze processors and a shared memory……………………………………………. 42

Table D-1: Comparison of *Spartan 3E* and *Virtex 5* FPGA……………………………. 88

Abstract

A STUDY OF MULTIPROCESSOR SYSTEMS USING THE PICOBLAZE 8-BIT MICROCONTROLLER IMPLEMENTED ON FIELD PROGRAMMABLE GATE ARRAYS

Venkata Chandra Sekhar Mandala
Thesis Chair: David H. K. Hoe, Ph.D.

The University of Texas at Tyler
August 2011

As Field Programmable Gate Arrays (FPGAs) are becoming more capable of implementing complex logic circuits, designers are increasingly choosing them over traditional microprocessor-based systems for implementing digital controllers and digital signal processing applications. Indeed, as FPGAs are being built using state-of-the-art deep submicron CMOS processes, the increased amount of logic and memory resources allows such FPGA-based implementations to compete in terms of speed, complexity, and power dissipation with most custom-built chips, but at a fraction of the development costs. The modern FPGA is now capable of implementing multiple instances of configurable processors that are completely specified by a high-level descriptor language. Such arrays of soft processor cores have opened up new design possibilities that include complex embedded systems applications that were previously implemented by custom multiprocessor chips. As the FPGA-based multiprocessor system is completely configurable by the user, it can be optimized for speed and power dissipation to fit a given application.

The goal of this thesis is to investigate design methods for implementing an array of soft processor cores using the Xilinx FPGA-based 8-bit microcontroller known as PicoBlaze. While development tools exist for the larger 32-bit processor from Xilinx known as MicroBlaze, no such resources are currently available for the PicoBlaze microcontroller. PicoBlaze benefits in applications that requires only less data bits (less than 8 bits). For example, consider the gene sequencing or DNA sequencing in which the processing requires only 2 to 5 bits. In such an application, PicoBlaze can be a simple processor to produce the results. Also, the PicoBlaze unit offers a finer level of granularity and hence consumes fewer resources than the larger 32-bit MicroBlaze processor. Hence, the former will find applications in embedded systems requiring a complex design to be partitioned over several processors but where only an 8-bit datapath is required.

The main challenge of this research is evaluating efficient schemes for interprocessor communication suitable for use with an 8-bit microcontroller. Two standard communication schemes were considered: the Mailbox method and Shared memory design. The former was found to be suitable for an array of PicoBlaze units that require nearest neighbor communication. This topology is sufficient for many signal processing applications. Various degrees of interconnect capability were considered for the Mailbox method and a wrapper that enables efficient array design was implemented. A simple Finite Impulse Response (FIR) filter was implemented to verify the design. For the Shared memory design, a round-robin arbiter design was considered. and four PicoBlaze units were connected to a single shared memory over a common bus.

This research provides some guidelines for implementing a multiprocessor system using the PicoBlaze processors. A comparison of logic and memory resource utilization indicates that a wrapper design using four FIFO (First In First Out) buffers provides a good tradeoff between interconnectivity and routing complexity, allowing many common signal processing applications to be implemented. The shared memory approach is more suited for designs where scaling to a large number of processors that need to transfer arbitrary and large blocks of data is required.

Chapter One
Introduction

Field Programmable Gate Arrays (FPGAs) are becoming a viable alternative for digital controllers and Digital Signal Processing (DSP) applications which were previously dominated by microprocessors and dedicated chips. The primary reason for the recent increased popularity of FPGA-based designs is that the modern FPGA chip now has the logic cell and memory resources implemented on advanced processes that allow them to realize many complex designs that can run more efficiently than regular microprocessor-based systems. This efficiency is seen in lower power consumption while maintaining the required clock speed. Also the growth in logic cell resources have allowed soft processor cores to be implemented on an FPGA.

Today, multiple soft processor cores can be implemented on an FPGA due to the increase of logic cells and memory unit resources. The ability to specify the hardware for implementing an array of processors through the use of a high-level descriptor language (HDL) allows a high-degree of flexibility for such FPGA-based designs. In general, Multiprocessor systems-on-a-chip (MPSoC) are being developed to support multiple applications on modern embedded systems. The ability to create an array of soft processor cores on a reconfigurable fabric with the associated advantages of flexibility and optimized run-time efficiency has made the idea of implementing complex embedded systems with FPGAs an attractive design option [1, 2].

1.1 Soft Processor Cores on FPGA

MPSoC designs are required in applications that perform multiple operations (e.g. smart phones, PDAs, set-top boxes). The challenge for design occurs when hardware and software implementations are considered. Due to the increased number of logic cells, speed and performance, FPGAs can implement multiple soft processor cores. MPSoC is beneficial in terms of power consumption, as they can be clocked at lower speeds thereby

reducing the power consumed. Also, a high degree of parallelism can be maintained when using soft processor cores [3].

There have been several VLSI prototypes of arrays of simple processors developed at both the industrial and university level. Each simple processor in an array typically has local memory and specialized interconnections to communicate with neighboring processors. Some design examples include the PicoChip from PicoArray, the RAW processor developed at MIT, and the Am2045 core from Ambric [4, 5, 6]. Also, there has been recent research into an asynchronous array of simple processors (AsAP) chip consisting of processors with an array size of 6×6 that is designed to compute DSP tasks efficiently [7]. Soft processor cores like MicroBlaze from Xilinx, Nios from Altera, and the Mitrion Virtual Processor from Mitrionics have been implemented as an array of soft processor cores for various applications [2, 4].

1.2 Research Objectives

This thesis develops a methodology for implementing an array of soft processor cores using a simple 8-bit microcontroller known as PicoBlaze. While Xilinx provides a design flow for implementing an array of MicroBlaze processors (a 32-bit RISC-style processor), no such resources exist for the PicoBlaze processors. The tasks for this thesis include understanding the utilization of resources available on an FPGA and developing an application which involves programming the PicoBlaze processor using assembly language coding and specifying the hardware implementation using a high-level descriptor language. The templates of different topologies from different architectures will be examined and the best architecture for the design of an array of soft processor core will be determined. In summary, a systematic study of an array of soft processor cores using the PicoBlaze 8-bit microcontroller is undertaken. Unlike the Xilinx Microblaze processor, which has the resources for optimizing interprocessor communication, there is a need to build the necessary logic on an FPGA for communicating between arrayed PicoBlaze processors. A wrapper needs to be designed that should be flexible enough to be used for a wide range of signal processing applications.

1.3 Research Method

The research method includes the following. Two Xilinx development boards (*Spartan* 3E and *Virtex* 5 – see Appendix D-4 for a specification of the two FPGAs) are chosen and a PicoBlaze microcontroller is studied and implemented using VHSIC (very high speed integrated) Hardware Description Language (VHDL) on the two FPGAs. Then, different architectures for an array of processors and different communication schemes are studied. A wrapper is designed that is used to produce an array as well used to keep the flow of resources constant for every unit. Then, an array of PicoBlazes is designed and implemented on the development boards. Finally, a simple FIR filter is implemented on the array to verify the functionality of the design. At all stages, the results are observed on the LEDs of the *Spartan* 3E and *Virtex* 5 boards.

1.4 Thesis Outline

The flow of the thesis is as follows: Chapter 2 gives the background information on arrays built from simple processors, functional units, and soft processor cores. Details ofthe PicoBlaze soft processor core are also discussed in this chapter. In Chapter 3, communication between the Picoblaze processor cores is performed using FIFO buffers. Also, this chapter gives details of the wrapper and array of PicoBlaze units. In Chapter 4, the shared memory mode of communication for the PicoBlaze processor using an arbiter is described. Chapter 5 provides some conclusions and gives an overview of future work.

Chapter Two

Background

This chapter gives the background information on the need for multicore processors. The first section of the chapter describes the trend towards multicore processors. The second section is a discussion about an array of different processors in different categories. The next section gives the details of the PicoBlaze soft processor. And, the last section talks about the different modes of interprocessor communication for a PicoBlaze microcontroller. Finally, a summary is provided for this whole chapter.

2.1 Trend Towards Multicore Processors

This section deals with the issues encountered with advanced processor design. First, the power wall problem is considered in which different factors and equations are discussed. This provides the basis for understanding the trend towards multicore processors.

2.1.1 Power Wall Problem

Since the invention of the microprocessor in 1971, clock rates for processors have continued to increase until early in the 2000's. This was accomplished by technology scaling, where the decrease in integrated circuit (IC) feature sizes have made making Field effect transistors (FETs) faster and have allowed more transistors to be implemented on a single chip. In 1965, Gordan E. Moore predicted that the numbers of transistors on an IC will double every eighteen months. This has become known as "Moore's Law," and has held true to this point. Figure 2.1 illustrates the increase in clock rate with each succeeding generation of Intel single core processors. What is notable is the peak frequency of 3.6 GHz reached by the Pentium 4 in 2004. Intel had planned to implement a 4 GHz version of its Pentium 4 processor but ended up cancelling its release [8]. The problem is the increasing power dissipation that accompanies each increase in clock rate.

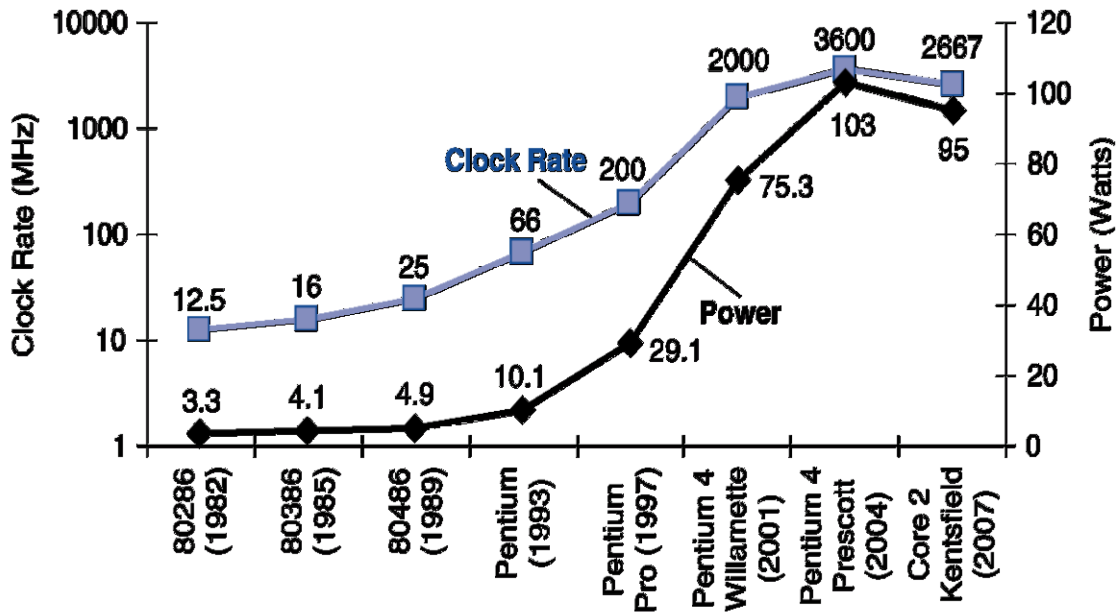


Figure 2.1: Trend in clock rate and power for Intel uniprocessors (single core) [9]

There is a limit to how much power can be dissipated by an integrated circuit due to the maximum rate of heat removal from an air-cooled micro-chip. This problem, known as the “power wall,” can be modeled theoretically [10]. For CMOS circuits (currently the most common technology for implementing ICs), the primary source of power dissipation is the dynamic power, the power consumed during switching. The dynamic power dissipation depends on the capacitive loading of the logic gate, the voltage applied, and the frequency of operation:

$$P = C \cdot {V_{dd}}^2 \cdot f \tag{2.1}$$

where,

P = dynamic power of the circuit,

C = capacitive load of the circuit

V_{dd} = supply voltage

f = frequency (clock rate)

As feature sizes decrease, the trend is to decrease the power supply voltage V_{dd} (see Figure 2.2). The main reason is to keep the electric field in the transistors to a manageable level. This also helps with minimizing the power dissipation. However, lowering the supply voltage also has a negative impact on circuit speed. This can be seen by the following sets of equations:

$$I_{DS} \propto (V_{dd} - V_T)^{\boldsymbol{n}} \quad (2.2)$$

$$Delay \propto C\frac{V_{DD}}{I_{DS}} \quad (2.3)$$

where I_{DS} is the drain-to-source current of a MOSFET and V_T is its threshold voltage. To improve I_{DS} as the supply voltage is decreased, the threshold voltage can be decreased. However, lowering V_T below 0.5 V makes it increasingly difficult to turn the transistor off, especially for deep submicron transistors. Indeed, for circuits implemented on a process in the nanoscale regime, up to 40% of the power dissipation can be due to "leaky" transistors. The following equation summarizes the total power dissipation for a CMOS circuit:

$$P = \alpha \cdot C \cdot {V_{dd}}^2 \cdot f + V_{dd} \cdot I_{st} + V_{dd} \cdot I_{leak} \quad (2.4)$$

where,

α = activity factor

V_{dd}= supply voltage

I_{st}= static current

I_{leak} = leakage current

Power per transistor scales with frequency and also scales with V_{dd}. Lower V_{dd} can be compensated with increased pipelining to keep throughput constant. Power per transistor is not equal to power per area. Therefore, power density is a serious issue.

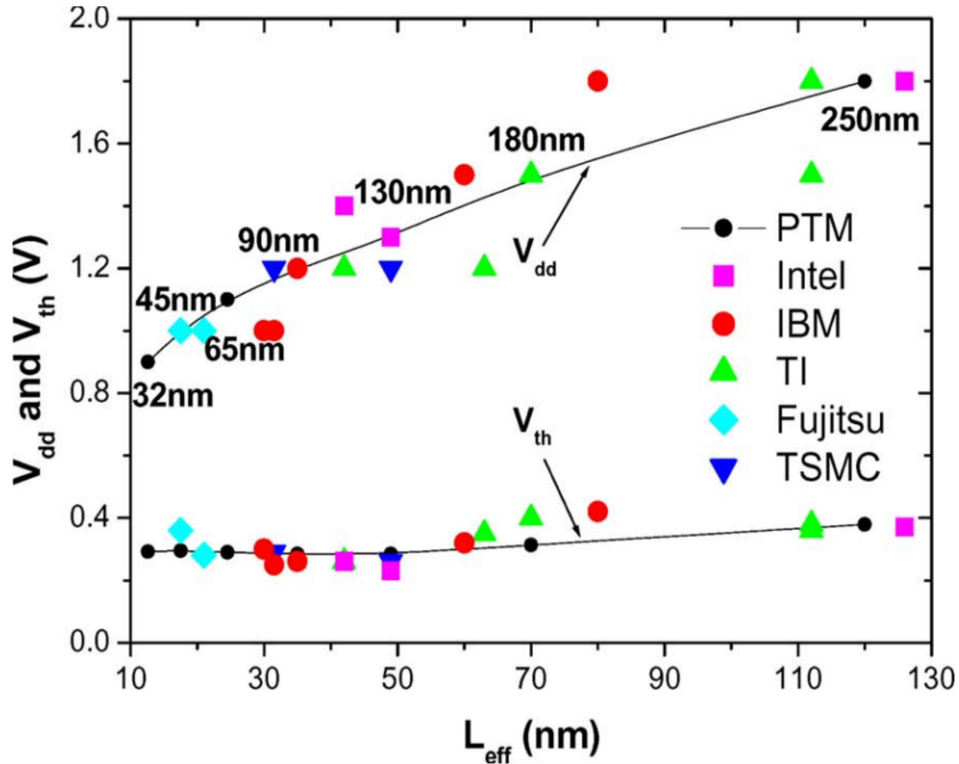


Figure 2.2: Plot showing supply voltage (V_{dd}) and thresold voltage (V_{th}) scaling with transistor size (L_{eff}) [11]

There are two solutions to power wall. The first is to introduce high-performance cooling technologies as implemented by IBM on their z/10 and z/11 servers [12]. The second solution is to implement multiple processors per chip. Each processor can run at lower frequencies, but due to parallel processing, the overall performance is increased. These ICs that have multiple processors per chip are called "multicore" processors. Multicore designs have been implemented by Intel and AMD beginning in 2006. The following subsections review the trend towards multicore processor design.

2.2 Multicore Processor

Intel's Many Integrated Core (MIC) design has many cores on a single chip which is used to run platforms at trillions of calculations per second. This architecture is designed to meet lower power dissipation and achieve higher levels of parallelism and is targeted for highly parallel applications [13]. Intel's Tera-scale computing program is researching methods to build processors with hundreds of cores [14].

In addition, Intel Labs is building a microprocessor called Single-chip Cloud Computer (SCC) that contains 48 cores (most Intel architecture cores) on a single silicon chip. Researchers have demonstrated message-passing and shared memory mode of communication using the SCC architecture [15]. The following section describes the arrays of different processors.

2.2.1 Array of Processors

In this subsection, recent research into implementing an array of processors on a single chip is reviewed. An array of processors has simple processors with local memories and interconnect that helps inter-processor communication as shown in Figure 2.3.The Reconfigurable Architecture Workstation (RAW) from MIT has 16 tiles that are arranged in 2D-mesh in which each tile consists of a MIPS style compute processor (i.e., a pipelined RISC processor) . Each RAW processor consists of a single-issue 8-stage pipeline, a 32-bit floating point unit, 32KB of instruction and data cache memory in a RAW processor [4].

The picoArray is a high performance communication processor developed by Picochip for wireless signal processing applications. PC102, one of the implementations of picoArray, consists of a heterogeneous array of 322 16-bit Reduced Instruction Set Computer (RISC) processor Array Elements (PAE) and 14 Functional Accelerator Units (FAU). Each processor has separate local instruction and data memories organized in a Harvard Architecture. The array elements in PC102 are arranged in a 2D grid and communicate over a network consisting of 32-bit buses and programmable bus switches. Application development for the picoArray involves specifying interactions between processes in the form of signal flows [4].

AMBRIC, from Ambric Inc. whose basic unit is called a bric. Each bric has two pairs of units, each pair consisting of a compute unit and a memory unit. The compute unit consists of two 32-bit Streaming RISC (SR) processors with 64 words of local memory, two 32-bit streaming RISC processors with DSP extensions, and a 32-bit channel interconnect for inter-processor and inter-compute unit communications. Each RAM unit consists of four banks of 1KB RAM and dynamic channel interconnect to communicate with these memories. The Am2045 core from Ambric consists of 45 brics

arranged in a 5 x 9 array. Since each bric has eight processors, the Am2045 encapsulates a total of 360 processors with 585 KB of on-chip memory [4].

Discussion:

In terms of granularity, the basic building block has one or many simple processors. RAW has an individual tile which is designed in a MIPS style processor, while, the picoArray consists of 16-bit RISC processor and Ambric's bric have two 32-bit Streaming RISC (SR) processors and two 32-bit SRD processors. Overall, Ambric gives more computational capacity.

In terms of interconnection network, RAW allows packet-oriented routing, while Ambric has circuit switch interconnect channels and picoArray has programmable switched buses. In all the above processors, communication patterns are found before compilation providing flexibility to the compiler for scheduling fine-grain parallelism. Optimization in terms of power consumption is done by keeping the units in sleep mode or by varying the clock frequency in different units to save energy [4].

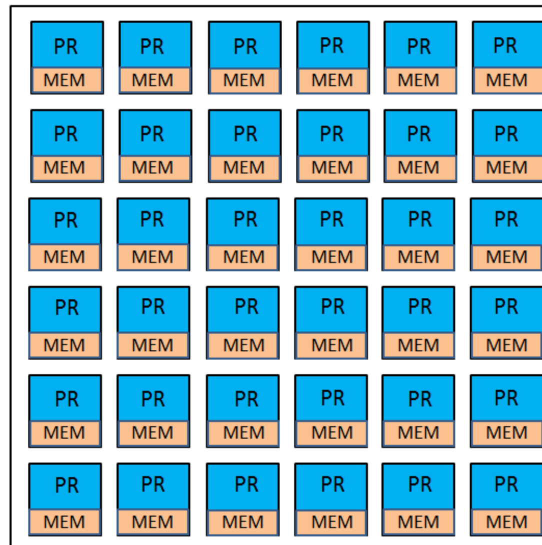


Figure 2.3: Array of processors where PR is Processor and MEM is memory

2.2.2 Array of Functional Units

An array of functional units architecture lacks an on-chip control processor and there is no central processing unit (see Figure 2.4). In this model, complex algorithms are

partitioned into a sequential flow of configurations to be dynamically configured into the functional units. The dynamic configuration is controlled by the configuration manager, which does not perform any computations by its own. Some examples include the following: Multiple ALU architecture with Reconfigurable Interconnect (MATRIX) from MIT, eXtreme Processing Platform (XPP) from PACT XPP Technologies, and the family of Field-Programmable Object Array (FPOA) from MathStar.

In all these examples, there is a basic unit called the functional unit (FU), consisting of a simplified ALU and a small amount of local memory. Each example has different strategies for configuring and managing the interconnections between functional units. MATRIX supports nearest neighbor connectivity as well as bypass connections. It also has the flexibility for arranging functions depending on the application requirement. The configuration sequencing of the processing array elements in XPP is done by a configuration manager. In FPOA, eight nearest neighbors can communicate with each other [4, 5, 6, 7].

Discussion:

In terms of architecture, MATRIX has local memory for each functional unit while XPP and FPOA have memory banks. In terms of interconnection network, MATRIX and FPOA uses nearest neighbor connectivity, while XPP communicates through buses.

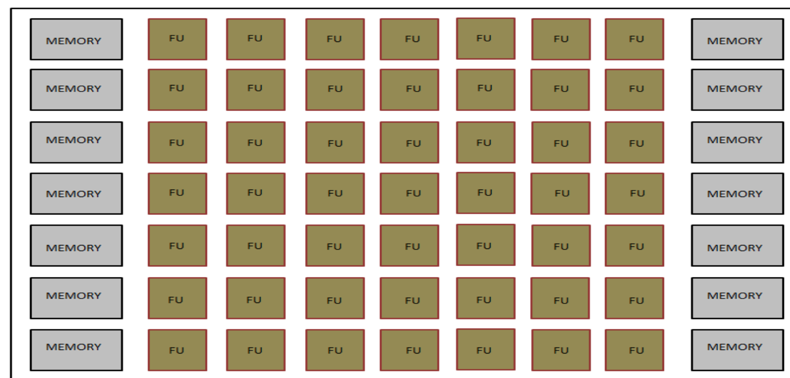


Figure 2.4: Array of Functional Units (FU)

2.3 Reconfigurable Computing

This section deals with reconfigurable computing, FPGAs and soft processor cores. PicoBlaze, a soft processor core from Xilinx Corporation is discussed. Finally, the interprocessor communication modes and schemes are reviewed.

2.3.1 Advantages of Reconfigurable Computing

Reconfigurable computing has many advantages over Application Specific Integrated Circuits (ASICs) and software-programmed processors. The reasons include: they fill the gap between hardware and software; higher performance than software; and higher level of flexibility than hardware.

Reconfigurable computing allows users to write, download and run the program on a hardware chip. This kind of re-use is achieved by using a special hardware known as Field-Programmable Gate Arrays (FPGAs) which is discussed in the next section (section 2.3.2).

2.3.2 FPGAs

FPGAs were first introduced in 1986 by the Xilinx Corporation. As denoted by its name, FPGAs can be programmed in the "field" by the user based on the type of the application desired. FPGAs are preferred over application specific integrated circuits (ASICs) for several reasons: they are cost effective for low volume, high capacity and offer more flexibility than programmable logic devices (PLDs). The reason for selecting a FPGA over an ASIC is that complex designs can be implemented at lower engineering costs with FPGAs.

2.3.2.1 Microcontroller within an FPGA

While both dedicated microcontrollers and FPGAs are able to successfully implement practically any digital logic function, the former have a fixed hardware configuration and can become obsolete with changing application requirements. On the other hand, as an FPGA-based microcontroller is described by VHDL code. It's architecture is flexible and can easily be modified by simply updating the code and reconfiguring the device. In addition, any support logic required by the microcontroller

can easily be added within the FPGA by simply adding the pertinent VHDL code [18] [19].

2.3.3 Array of Soft Processors

An array of soft processors, often called soft instruction processors, consists of programmable instruction processors implemented in reconfigurable logic. As the cost and time to market for ASICs has increased, soft processors have found increased deployment in FPGA-based embedded systems. Even though the soft processors cannot meet the area, power and speed characteristics of their corresponding hardware solutions, they can make use of the reconfigurability option in FPGAs to match the complexity and implementation requirements of the application. Major FPGA vendors have provided their own soft processor cores that can be used on their own FPGA platforms. Xilinx FPGAs are widely used in soft multiprocessor implementations. It supplies three main processors: a 32-bit RISC soft processor core known as MicroBlaze, the 8-bit soft microcontroller known as PicoBlaze, and a hard processor core, which is a PowerPC processor. Soft processor cores can also be used as elements in larger processor arrays which gives solutions similar to the class of array of processors presented earlier, but in a more flexible way. A brief overview of the Mitrion platform, which is portable to a number of FPGA computer platforms, is given below.

The Mitrion virtual processor from Mitrionics has a cluster of soft computing nodes placed on the same FPGA. It differs from the conventional soft processor cores in the way it offers parallelism. The Mitrion-C programming language is used to extract parallelism based on data dependencies rather than on the order of execution. The optimizations in hardware are achieved by providing virtualized processor architecture in the form of soft IP cores that are configurable. Figure 2.5 shows an array of different computing nodes which are implemented as soft processor cores [4].

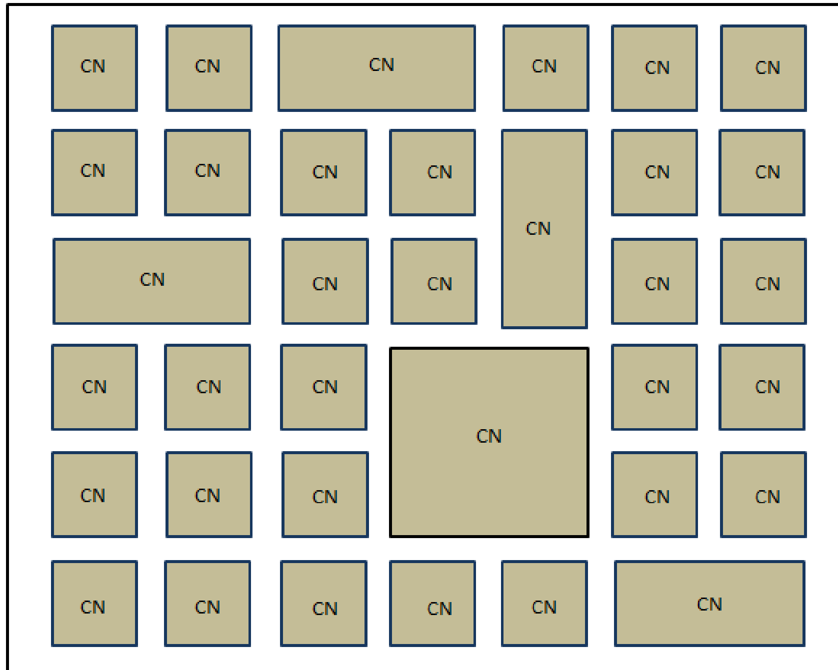


Figure 2.5: Array of Soft Processors (CN stands for Computation Nodes)

2.3.4 PicoBlaze Microcontroller

There are dozens of 8-bit microcontroller architectures and instruction sets. Modern FPGAs can efficiently implement any 8-bit microcontroller, and available FPGA soft processor cores support popular instruction sets such as the PIC, 8051, AVR, 6502, 8080, and Z80 microcontrollers.

The PicoBlaze soft processor core is called the constant (K) coded Programmable State Machine (KCPSM3) also known as Ken Chapman's PSM, after its creator. The PicoBlaze microcontroller is specifically designed and optimized for the *Spartan 3* family. Versions also exist for the *Spartan 6* and *Virtex 6* family of FPGAs. Its compact architecture consumes considerably less FPGA resources than comparable 8-bit microcontroller architectures implemented on an FPGA. Also, the PicoBlaze microcontroller is provided as a free, source-level VHDL file with royalty-free reuse on Xilinx FPGAs [20].

The PicoBlaze microcontroller has many advantages over standalone microcontrollers. Since it is specified in VHDL code [21], the PicoBlaze microcontroller can be easily updated to allow the microcontroller to be retargeted for future generations

of Xilinx FPGAs, enabling future cost reductions and feature enhancements. Furthermore, the PicoBlaze microcontroller is expandable and extendable.

Before the advent of the PicoBlaze and MicroBlaze embedded processors, the microcontroller resided externally to the FPGA, limiting the connectivity to the other FPGA functions and restricting overall interface performance. By contrast, the PicoBlaze microcontroller is fully embedded in the FPGA allowing extensive on-chip connectivity to other FPGA resources. Signals remain within the FPGA, improving overall system performance. The PicoBlaze microcontroller reduces system cost because it offers a single-chip solution that is fully integrated within the FPGA [21].

The PicoBlaze architecture consists of an 8-bit datapath with a 64-byte scratchpad RAM, 16 registers and a 1K word (18 bits wide) Instruction PROM. A block diagram view of the PicoBlaze embedded microcontroller is shown in Figure 2.6 [20].

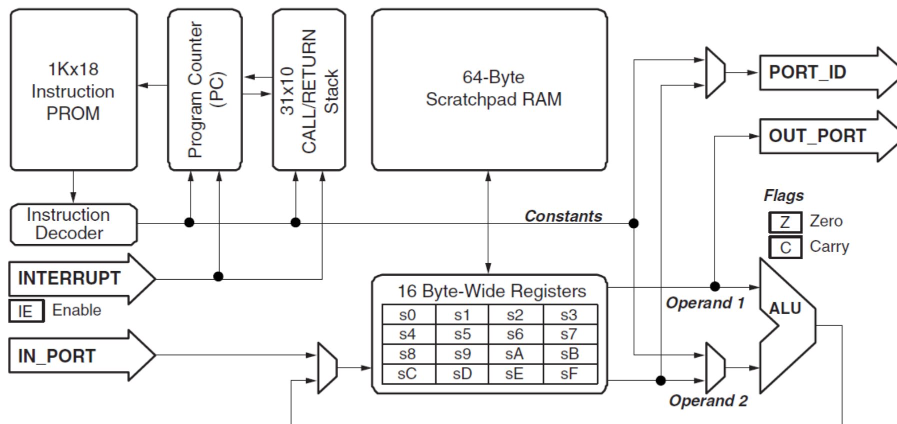


Figure 2.6: PicoBlaze Microcontroller Block Diagram [20]

The latest version of the PicoBlaze microcontroller is called KCPSM3 which can be downloaded from the Xilinx PicoBlaze lounge. The PicoBlaze design suite of tools consists of *"KCPSM3.vhd"* (the main VHDL microcontroller code) and an assembler folder containing "*KCPSM3.exe*", *"ROM_form.vhd"*, *"ROM-form.coe"* and *"assemblycode.psm"*. The assembler is used to generate the VHDL code for the Block RAM from the *psm* code as shown in Figure 2.7.

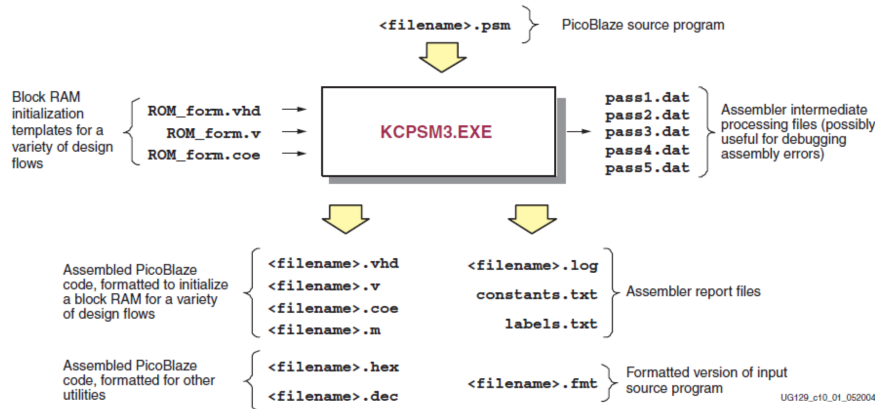


Figure 2.7: KCPSM3 Assembler Files [20]

The PicoBlaze microcontroller is resource efficient allowing multiple PicoBlaze microcontrollers to be implemented on a single FPGA. Consequently, complex applications can be portioned across several PicoBlaze microcontrollers with each controller implementing a particular function (for example, keyboard and display control, and system management) [19].

2.4 Interprocessor Communication

The key challenge for a multicore processor design is developing efficient means of communication among the cores. In a multiprocessor system, things get difficult when embedded designers encounter interprocessor connections designed to take advantage of a particular processor environment. On FPGA processors, the most common communication schemes are Shared Memory and Mailbox based message passing. Each interprocessor communication has its advantages and disadvantages. System designers must choose the most appropriate method for their application. The mailbox tends to be used to exchange smaller packets of information. It also provides synchronization between processors. The mailbox forms a channel through which messages are queued in a First in First Out (FIFO) mode from one end by the senders, and then dequeued at the other by the receiver [1, 18].

After multiple processors are connected to the interconnect infrastructure, the system designer is concerned with the primary modes of communication and synchronization between the processors. Mailboxes are used to pass messages between

senders and a receivers. Typical usage of a mailbox is to send actual data between processors. There are again two methods of communication in mailbox mode. One is the synchronous method in which the receiver actively keeps polling the mailbox for new data. The other method is an asynchronous method where the mailbox sends an interrupt to the receiver once the data is present in the mailbox. Xilinx provides the XPS Mailbox inter-processor communication core which has a pair of mailbox FIFOs in each mailbox. The recommended connection from Xilinx is to use a single mailbox between a pair of processors. FIFOs are implemented either using distributed Random Access Memory (RAM) or Block RAM (BRAM) resources [19].

Shared memory is the other mode of interprocessor communication between processing subsystems. Any processor can reference any shared memory location directly in a shared memory scheme. Data could be distributed across multiple processors, whose details could be abstracted by some particular software Application Program Interface. Shared memory can be built out of on-chip local memory or external memory. This scheme is more suitable for sharing large blocks of data between processors [19].

The following chapters will discuss in detail the implementation of the Mailbox and Shared Memory schemes suitable for communicating between multiple PicoBlaze microcontrollers implemented on a single Xilinx FPGA.

2.5 Summary

Overall, this chapter provided background information of the problems and issues with various types of processor implementations. Also, this chapter explained PicoBlaze: a soft core processor. The discussion of the chapter ends with the different communication schemes available between processors. Different interprocessor communication mechanisms will be addressed in future chapters.

Chapter Three
FIFO Style Interprocessor Communication

Having discussed the need for multicore processors and various soft processor core implementations, the Mailbox scheme for interprocessor communication using the PicoBlaze microcontroller on Xilinx FPGAs is discussed in this chapter. First, the basic FIFO buffer is explained and then the implementation of two processors communicating with a single FIFO buffer is described. Then the wrapper with four FIFO's, one processor and four latches is discussed. Finally, two applications that demonstrate the working of the FIFO buffer and wrapper are described: the counting of numbers and an FIR filter.

3.1 FIFO Style Communication

Mailboxes are used to send and receive data or messages between systems. Messages are queued in a FIFO fashion from the sender and then dequeued by the receiver. The mailbox can be taken as a simplified Transmission Control Protocol/Internal Protocol (TCP/IP)-like message channel between systems or processors. Message reception is addressed synchronously or asynchronously. In the synchronous method, a receiver continuously polls the mailbox for new data, while, in the asynchronous method, the mailbox sends an interrupt to the receiver once the data is present in the mailbox.

3.1.1 FIFO

A FIFO buffer is an "elastic" storage between two subsystems (see Fig. 3.1). It typically has two control signals, *rd* and *wr*, for read and write operations, respectively. When *wr* is high, the input data is written into the buffer, while the *rd* signal is used to read data from the FIFO buffer or flush the data out from the FIFO buffer.

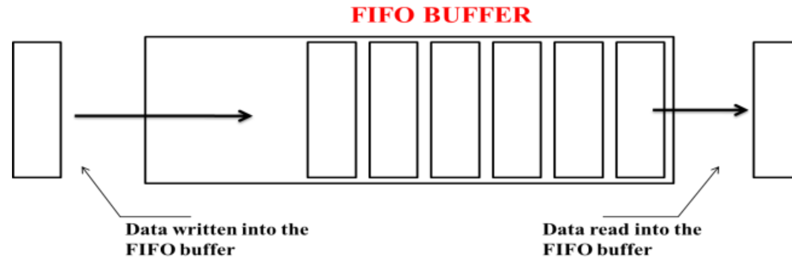


Figure 3.1: Theoretical view of a FIFO Buffer

In many applications the FIFO buffer is a critical component and the optimized implementation can be complex. In the next section, a simple circular-queue-based design is introduced. More efficient, device-specific implementations can be found in the Xilinx literature [22].

A FIFO buffer can be implemented in various ways. A straightforward approach is a circular queue-based implementation. In this method, a control circuit is added to a register file and the registers are arranged in a circular queue with read and write pointers (see Figure 3.2). The write pointer is at the head of the queue and the read pointer is at the tail of the queue. The pointer shifts one position for each operation, read or write.

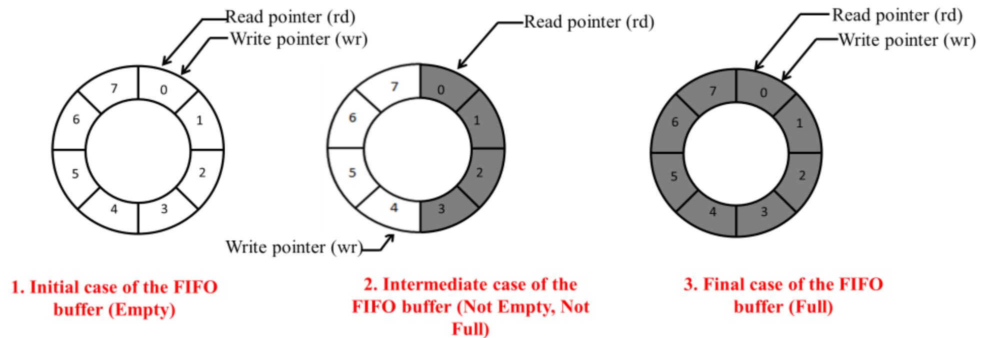


Figure 3.2: Three Possible Cases of a circular FIFO Buffer

A FIFO buffer has full and empty status signals. A full signal indicates that the FIFO is full and that data cannot be written. An empty signal states that the FIFO is empty and data cannot be read from the FIFO [22].

The pseudo code listing algorithm for reading and writing to FIFO is given in Figure 3.3.

Write to FIFO	Read from FIFO
• read full_flag ; • while full_flag = = 01 • read full_flag ; • write data ;	• read empty_flag ; • while empty_flag = = 01 • read empty_flag ; • read data ;

Figure 3.3: Pseudo code for FIFO *write* and *read* cases

3.1.2 PicoBlazes with a FIFO Buffer

An initial implementation has two PicoBlazes that are communicated using a single FIFO buffer. The write signal of the FIFO is connected to the write strobe of the PicoBlaze 1 and the read signal of the FIFO is connected to the read strobe of the PicoBlaze 2. Also, the output of the PicoBlaze 1 unit is connected to the data input line of the FIFO buffer and the output of the FIFO buffer is connected to the input signal of the PicoBlaze 2 unit. Figure 3.4 shows the connections between the PicoBlazes and the FIFO.

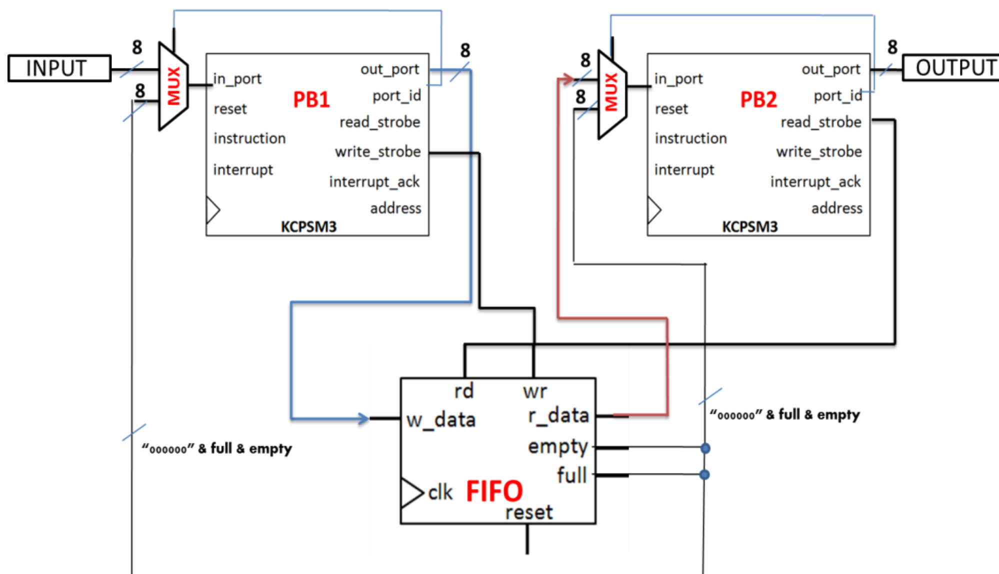


Figure 3.4: Two PicoBlazes (PB1 and PB2) communicating through a FIFO Buffer

3.1.3 Experimental Verification of the FIFO with Two PicoBlazes

The operation of the FIFO buffer is checked by implementing a small program on two processors. In the first processor, natural numbers are continuously produced and sent to the FIFO buffer and the status of the FIFO buffer is continuously verified. When the FIFO buffer is full, it sends a Full status signal to processor PB1. Then, the processor PB1 pauses at that point and waits until the data is read by processor PB2. The second processor is programmed so that when a button is pressed, it reads 16 bytes of data from the FIFO. Now, the first processor can send 16 more bytes of data into it as some data taken out by processor PB2. This experiment illustrates the working of FIFO as well as verifies the Mailbox method of interprocessor communication. This whole experiment is verified on the Xilinx *Spartan* 3E FPGA development boards. (See Appendix A for the assembly code.)

3.2 Array of PicoBlazes

In this section, various nearest neighbor interconnection schemes are considered for an array of PicoBlazes. A wrapper is designed to allow different topologies. An experiment is performed to validate the wrapper and at last an array of PicoBlazes are produced and validated.

A wrapper is designed in order to allow repeated use of the same structure. The wrapper is coded in VHDL and its entity is called for *n* number of times in the top level file (main file). This facilitates saving time in code development and maintenance. For example, any change in the wrapper description will automatically makes changes in all the units of the array which are instances of this wrapper component.

3.2.1 The Wrapper

Before designing the wrapper there are different topologies to be discussed. Consider the design of the wrapper with one, four and eight FIFOs, as shown in the Figure 3.5. The wrapper with one FIFO buffer can practically be used from one direction only as one wrapper can communicate with one PicoBlaze only. A wrapper with 8 FIFO buffers can communicate with eight neighboring processors. Analyzing data in Table 3.1 taken from *Spartan 3E* and Table 3.2 taken from *Virtex 5*, a wrapper designed with eight FIFO buffers occupies a lot of logic cell resources. The design of a wrapper with four FIFO

buffers provides a good balance between routing complexity and the ability to maintain adequate communication between processors for most signal processing applications. For example, in an FIR filter implementation [23], a topology which allows data to flow in the horizontal, vertical and diagonal directions through different blocks allows the filter to be easily partitioned into an efficient pipelined architecture [24]. Such a scheme will be described in detail at the end of this chapter. Shown below in Figure 3.6 is the screen shot of the wrapper in the schematic editor for three cases (1, 4 and 8 FIFO buffers).

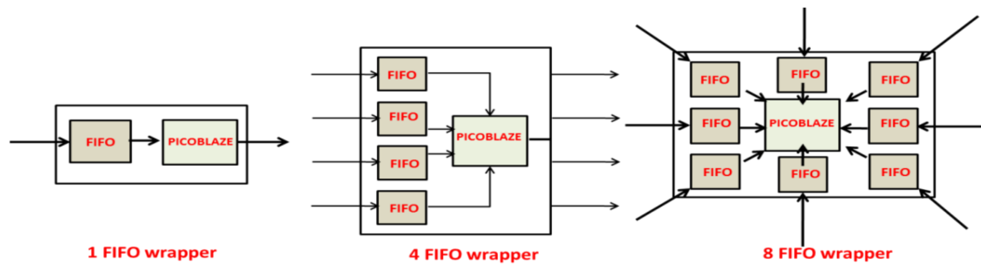


Figure 3.5: Different topologies in designing a Wrapper

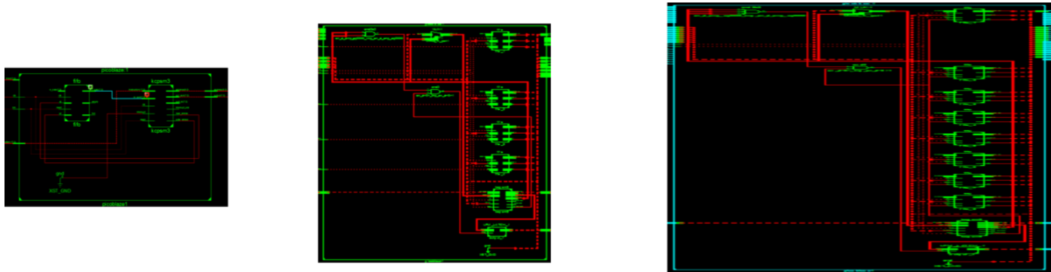


Figure 3.6: Wrapper design view in schematic editor (1, 4 and 8 FIFOs from left)

Hence, the design of a wrapper that can communicate with four PicoBlaze units around it is the focus of this work. The designed wrapper has four FIFO buffers for each of the four inputs and four latches for latching the data at the output. Different read and write signals are taken in and out of the wrapper to maintain communication with the other processors. Details of the wrapper are shown in the Figure 3.7.

Figure 3.7: Detailed wrapper with four FIFO buffers and a PicoBlaze Processor

The signals are defined as follows:

1, 3, 6, 8 are the write strobes from previous PicoBlaze processor

2, 4, 7, 9 are inputs to the current PicoBlaze processor

5 is the set of inputs of FIFO status signals from neighboring PicoBlaze processor

10, 11, 12, 13 are the outputs

Shown in Figure 3.8 are the input (IN) and output (OUT) ports for wrapper, described in a VHDL entity statement. For example, consider *ws_s*, *ws_e*, *ws_d* and *ws_w* in the port map which are called as write strobe in south, write strobe in east, write strobe in southeast (called as 'diagonal' in the code) and write strobe in south directions respectively. Consider two PicoBlaze units, PB1 and PB2 where PB1 is writing data in

the south direction to PB2. Here, the PB2 south FIFO is written with PB1 south output and therefore, PB1 write strobe should be sent to PB2 in order to strobe the south FIFO of PB2. Hence, *ws_s* is important for sending data from one Output of PB1 to input of PB2 south side FIFO. A similar flow occurs for the remaining directions as well.

```
entity PicoBlaze is
  port(
        input_1: in std_logic_vector(7 downto 0);            -- IN
        input_2: in std_logic_vector(7 downto 0);            -- IN
        input_3: in std_logic_vector(7 downto 0);            -- IN
        input_4: in std_logic_vector(7 downto 0);            -- IN
        output_1: out std_logic_vector(7 downto 0);          -- OUT
        output_2: out std_logic_vector(7 downto 0);          -- OUT
        output_3: out std_logic_vector(7 downto 0);          -- OUT
        output_4: out std_logic_vector(7 downto 0);          -- OUT
        clk : in std_logic;                                  -- IN
        address : inout std_logic_vector(9 downto 0);        --INOUT
        instruction : inout std_logic_vector(17 downto 0);   --INOUT
        status1, status2, status3, status4: out std_logic_vector(7 downto 0);          -- OUT
        status1_in, status2_in, status3_in, status4_in: in std_logic_vector(7 downto 0);
        write_strobe_west_in, write_strobe_south_in, write_strobe_diagonal_in,
        write_strobe_east_in: in std_logic;                  -- IN
        ws_s, ws_e, ws_d, ws_w: out std_logic;               -- OUT
        btn : in std_logic                                   -- IN
        );
end PicoBlaze;
```

Figure 3.8: Port map for the Wrapper entity in VHDL

Figure 3.9 contains a code snippet showing the selection of lines depending on the port id of the associated PicoBlaze processor. In the PicoBlaze assembly program, data is sent to a particular port and also received from a particular port. For example, if the

data is to be taken from FIFO3, *port_id* of "0010" in binary notation or "2" in decimal notation should be called.

```
with port_id(3 downto 0) select

in_port <=
        fifo_out1 when "0000",          -- data from FIFO1
        fifo_out2 when "0001",          -- data from FIFO2
        fifo_out3 when "0010",          -- data from FIFO3
        fifo_out4 when "0011",          -- data from FIFO4

        "000000" & full1 & empty1 when "0100", -- FIFO1 empty/full status
        "000000" & full2 & empty2 when "0101", -- FIFO2 empty/full status
        "000000" & full3 & empty3 when "0110", -- FIFO3 empty/full status
        "000000" & full4 & empty4 when "0111", -- FIFO4 empty/full status

        status1_in when "1000",         -- status of FIFO1 of neighbor PicoBlaze
        status2_in when "1001",         -- status of FIFO2 of neighbor PicoBlaze
        status3_in when "1010",         -- status of FIFO3 of neighbor PicoBlaze
        status4_in when "1011",         -- status of FIFO4 of neighbor PicoBlaze
        "00000000" when others;
```

Figure 3.9: Wrapper with port id coded in VHDL

A multiplexer (MUX) is implemented in VHDL code taking port id as the select line. The PicoBlaze assembly level code uses the port ids defined here and thus sends the data to that particular port id.

3.2.2 Need for an Array of PicoBlaze Units

The need for constructing an array comes when Digital Signal Processing (DSP) applications like an Infinite Impulse Response (IIR) filter or Finite Impulse Response (FIR) filter design are considered. An array of processors or subsystems is required for a filter design as the data is needed to be processed in different number of stages.

3.2.3 Details of the Array

An array of PicoBlazes is arranged by using the wrapper designed with four FIFO buffers. The four FIFO buffers are sufficient to design an FIR filter. Four FIFO buffers (FIFO1, FIFO2, FIFO3 and FIFO4) are considered to take data coming from the four directions WEST, SOUTH, South_East (DIAGONAL), and EAST, respectively. An array of PicoBlaze units is arranged as shown in Figure 3.10

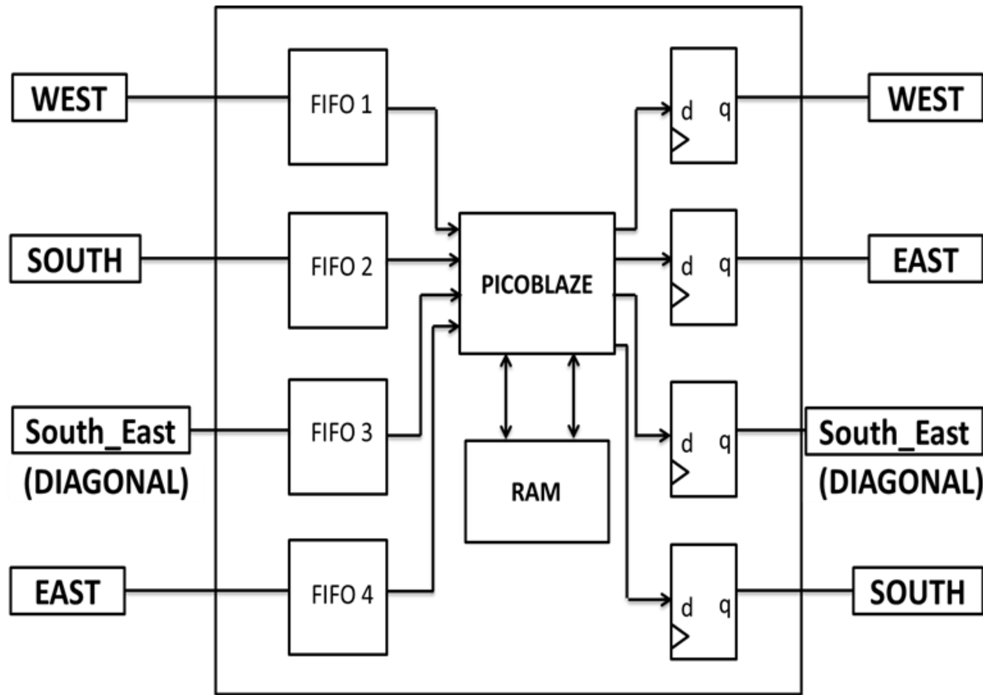


Figure 3.10: A Wrapper that connects to four neighboring Processors

The data coming from the neighboring PicoBlaze in the south direction is connected to the FIFO buffer wired to the SOUTH input port. Similarly, the neighbors in the South_East (DIAGONAL), WEST and EAST directions send the data into those FIFOs in those directions, respectively.

Each PicoBlaze has four inputs and four outputs. Four inputs are named as *in_west*, *in_south*, *in_diagonal* and *in_east* and, four outputs are named as *out_east*, *out_west*, *out_south* and *out_diagonal*. Connections between PicoBlazes depend on the direction of the data flow (see Figure 3.11 for details).

3.2.4 Routing/Logic Cell Resources Comparison

Table 3.1 and 3.2 below gives the statistics for 1-FIFO, 4-FIFOs and 8-FIFOs wrappers implemented on the *Spartan 3E* and *Virtex 5* FPGAs, respectively.

Table 3.1: Statistics for the 1, 4, and 8 FIFO Wrapper in *Spartan 3E*

SPARTAN-3E	Number of Slice LUTS (= Logic + Memory + route-thru) (Used/Available)	Number used for Logic	Number used for Memory	Number used for route-thru	Number of Block RAM (Used/Available)	Number of Slices (Used/Available)
1- FIFO	192 / 9,312 (2%)	102	68	22	1 / 20 (5%)	100 / 4,656 (2%)
4 - FIFO	196 / 9,312 (2%)	106	68	22	1 / 20 (5%)	107 / 4,656 (2%)
8 - FIFO	271 / 9,312 (3%)	181	68	22	1 / 20 (5%)	149 / 4,656 (3%)

Table 3.2: Statistics for the 1, 4, and 8 FIFO Wrapper in *Virtex 5*

VIRTEX -5	Number of Slice LUTS (= Logic + Memory + route-thru) (Used/Available)	Number used for Logic	Number used for Memory	Number used for route-thru	Number of Block RAM (Used/Available)	Number of Slices (Used/Available)
1- FIFO	145 / 69,120 (1%)	116	26	3	1 / 148 (1%)	89 / 69,120 (1%)
4 - FIFO	149 / 69,120 (1%)	120	26	3	1 / 148 (1%)	98 / 69,120 (1%)
8 - FIFO	199 / 69,120 (1%)	170	26	3	1 / 148 (1%)	138 / 69,120 (1%)

3.3 Experiment with Array of PicoBlazes

For the verification of consistency in the array as well as FIFO buffers inside the wrapper, a sample counting of numbers application is implemented on the Array of PicoBlaze.

3.3.1 Counting Numbers Experiment

The goal of this experiment is to check the FIFO buffer usage and the control that the PicoBlazes have on FIFOs. The data is continuously fed into four FIFOs of one destination PicoBlaze wrapper and only eight bits of data at a time is chosen out of four FIFOs. The FIFOs are fed from four other PicoBlazes. Here, consistency in holding data by FIFO buffers and status signals are verified at different clock speeds.

3.3.2 Status Signals and Read/Write Strobe Conditions Setup

Let us consider the communication between PicoBlaze 1 and PicoBlaze 4. The diagonal output of PicoBlaze 1 is taken into the diagonal input FIFO of the PicoBlaze 4. The data should be written if and only if the diagonal input FIFO of PicoBlaze 4 is not FULL. In order to check the condition, PicoBlaze 1 must be able to know the status signal of FIFO of PicoBlaze 4. Hence, there is a signal connection going out of the PicoBlaze 4 and also a signal connection going into the PicoBlaze 4. Similarly, if we consider the other cases too, there will be six more connections: three for the status in and status out of three other FIFOs; south, west and east.

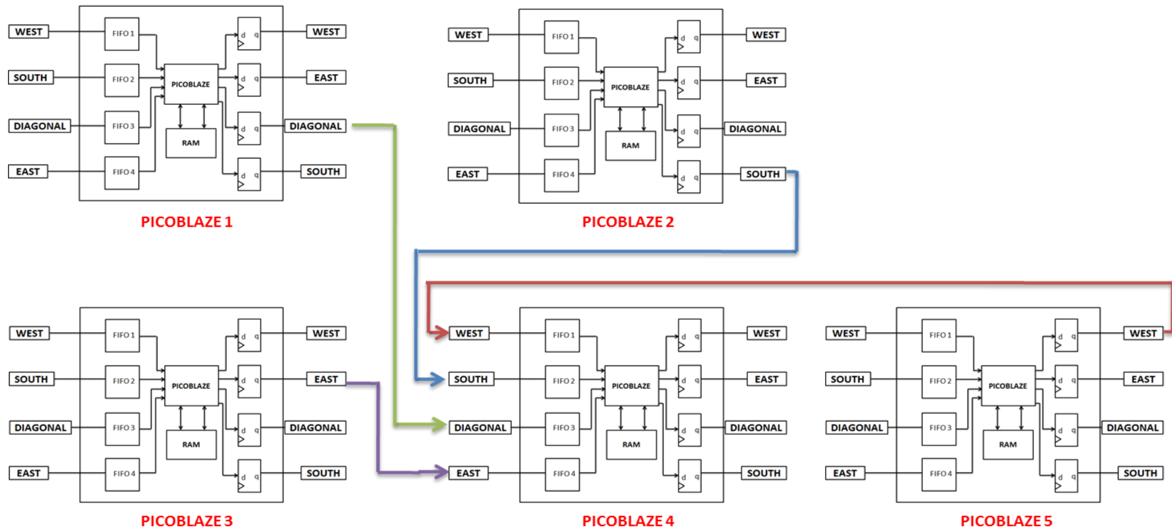


Figure 3.11 Experiment setup with 5 PicoBlazes where PicoBlaze 4 is destination

Also, data can be written into FIFO using the write enable signal, which should be connected to the write strobe of the PicoBlaze that is writing the data into FIFO. So, there

is one more connection that is taken in and out of the wrapper for exchanging the strobe signals. (See Appendix B for the code.)

3.3.3 Validation and Results

First, five processors named PicoBlaze 1, PicoBlaze 2, PicoBlaze 3, PicoBlaze 4, and PicoBlaze 5, are assumed to have their places on an FPGA as shown in the Figure 3.12. PicoBlaze 1 is programmed to send numbers starting from one and incremented by four and so on. Similarly, PicoBlaze 2 starts with two and increments by four and it continues for PicoBlaze 3 and PicoBlaze 5 counting numbers from three and four and incrementing by 4 on both PicoBlazes. The output of four PicoBlazes: PicoBlaze 1, PicoBlaze 2, PicoBlaze 3, PicoBlaze 5 are hardwired to the input of the destination processor, PicoBlaze 4 as shown in Figure 3.12 below.

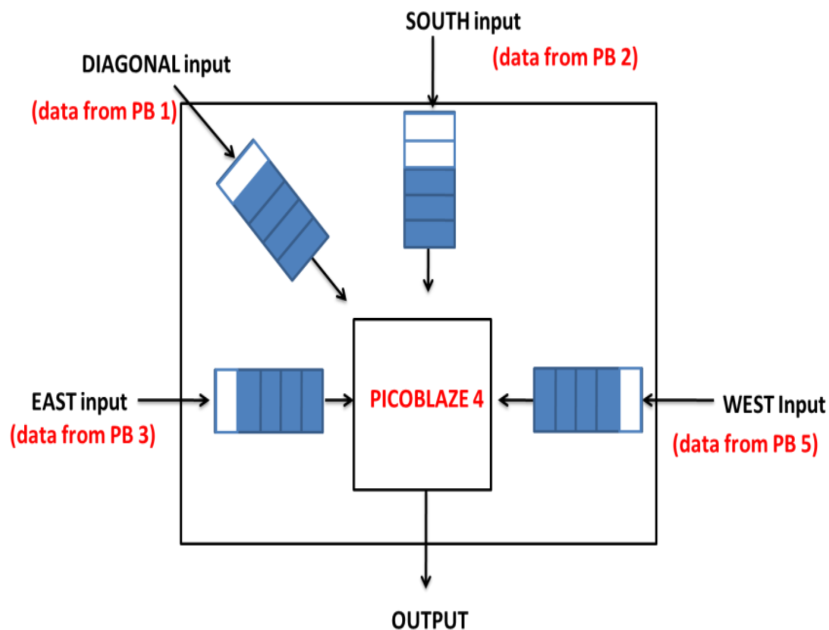


Figure 3.12: Destination PicoBlaze holding data in FIFOs

In this process, the destination processor, PicoBlaze 4 receives the output as one, two, three, four, five, six and so on. Hence, this is proof of work for array as the data is taken from PicoBlazes one after the other, holding the data in FIFO until it is being read by PicoBlaze 4. This example verifies the functionality of the Array, FIFO buffers and the wrapper.

3.4 Application (A 4-tap FIR Filter)

Digital signal processing is utilized in almost all aspects of today's modern life from cell phones to HDTV. The structure of many digital filters allows parallelism to be exploited for efficient hardware implementation. Also, the common computational units found in most of the signal processing algorithms (e.g., the Multiply and Accumulate Unit) helps in implementing them on hardware. The Finite Impulse Response (FIR) filter is a popular digital filter topology due to case of design and guarantee of stability [23, 24]. A simple FIR filter is implemented as an example.

The FIR filter is described by the simple difference equation shown in equation 3.1.

$$y(i) = \sum_{j=0}^{i} a(j) \cdot u(i-j) \qquad (3.1)$$

where,

$a(j)$ denotes the filter coefficients,

$u(i)$ are the filter inputs, and

$y(i)$ are filter outputs.

3.4.1 Algorithm and Simulation of FIR

An N-tap, T-input FIR is implemented in the C language. The complete C code is found in the file "*FIR.c" (*see Appendix C.1). The results of the C program are compared against the output of the arrayed PicoBlaze microcontrollers to validate the functionality of the FPGA implementation.

3.4.2 Array structural flow for FIR

An array of five rows and four columns (5× 4=20) PicoBlaze microcontrollers is designed to implement a simple FIR filter with 4 inputs (i=3). In this particular implementation, six of the PicoBlaze microcontrollers are not required for the signal flow and therefore are not included in the design. Figure 3.13 illustrates the array in which the six deleted PicoBlaze microcontrollers are shown in a light gray color).

The PicoBlaze microcontrollers labeled PicoBlazeU0, PicoBlazeU1, PicoBlazeU2 and PicoBlazeU3 can generate inputs $U[0]$,$U[1]$, $U[2]$ and $U[3]$respectively, while, PicoBlaze 1 to PicoBlaze 10 are used to process the data. Also, PicoBlaze 1, PicoBlaze 2, PicoBlaze 4 are PicoBlaze 5 fed with coefficient values $A[3]$, $A[2]$, $A[1]$ and $A[0]$ respectively. Output values of the FIR filter are $Y[0]$, $Y[1]$, $Y[2]$ and $Y[3]$ which are taken from PicoBlaze 7, PicoBlaze 8, PicoBlaze 9 and PicoBlaze 10 as shown in Figure 3.13.

The four output values $Y[0]$, $Y[1]$, $Y[2]$ and $Y[3]$ for the simple FIR filter are given by equations 3.2-3.5 respectively.

$$Y[0] = A[0] \cdot U[0] \tag{3.2}$$

$$Y[1] = A[0] \cdot U[1] + A[1] \cdot U[0] \tag{3.3}$$

$$Y[2] = A[0] \cdot U[2] + A[1] \cdot U[1] + A[2] \cdot U[0] \tag{3.4}$$

$$Y[3] = A[0] \cdot U[3] + A[1] \cdot U[2] + A[2] \cdot U[1] + A[3] \cdot U[0] \tag{3.5}$$

The PicoBlaze units U0, U1, U2 and U3 continuously generate the filter inputs $U[0]$, $U[1]$, $U[2]$ and $U[3]$ values, respectively, and send the data to PicoBlaze units 1, 3, 6 and 10. PicoBlaze units 1, 2, 4, and 7 take the filter coefficients and the data is processed in stages. For example, PicoBlaze unit 3 takes the input $U[1]$ (from the south direction), $A[2]$ (from the east direction) and multiplies the value and adds it to the diagonal FIFO buffered data (i.e., the data in diagonal buffer will be will be the output of P1 = $A[3] \cdot U[0]$) and outputs the value in the diagonal direction as shown in Figure 3.13. This is repeated in stages and the outputs ($Y[0], Y[1], Y[2], Y[3]$) can be observed at the bottom of the PicoBlaze units.

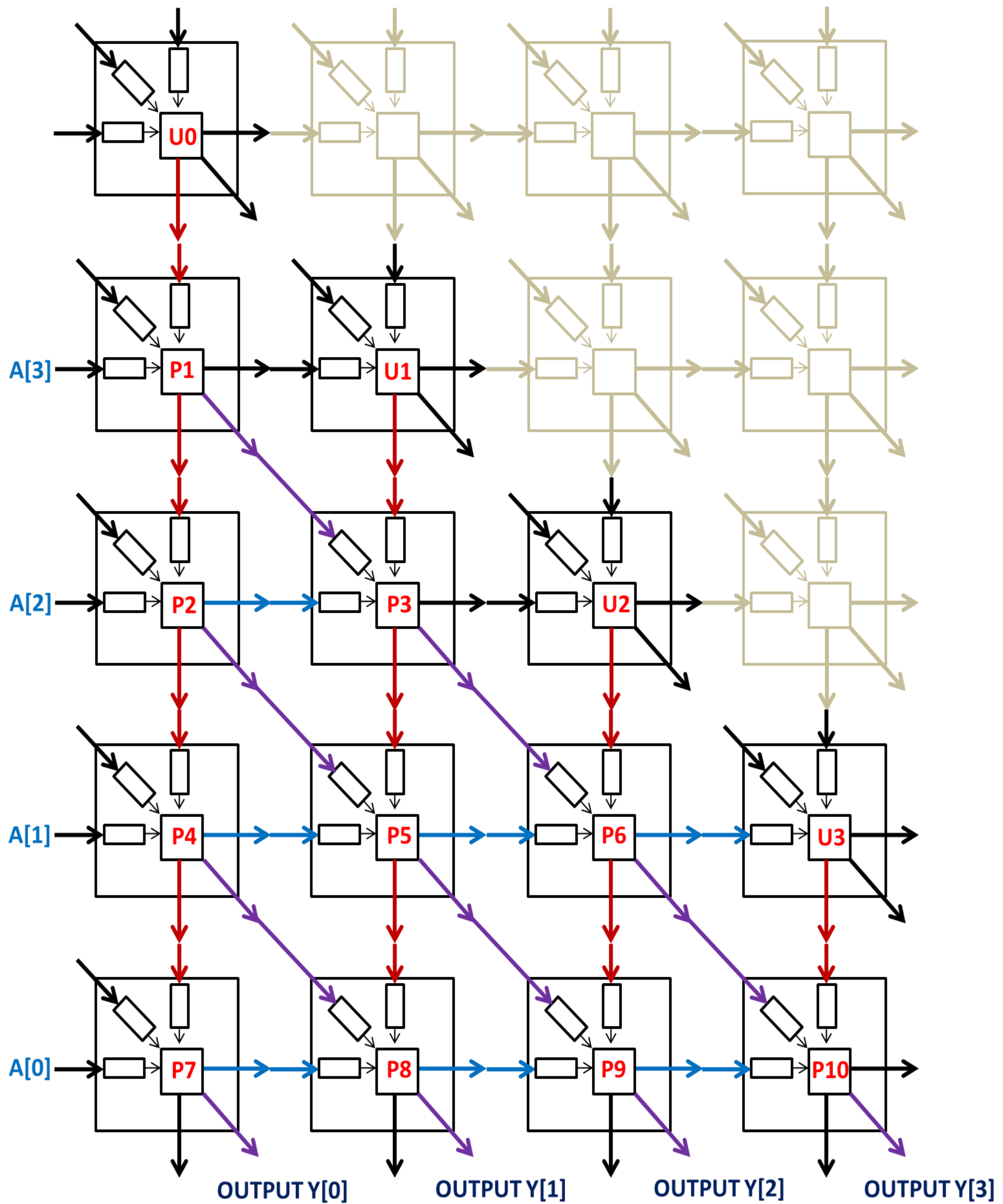


Figure 3.13: A 4-tap FIR Filter Application on an Array of PicoBlaze Units

3.4.3 Validation and Results

The 4-tap FIR filter was implemented on the Xilinx *Virtex 5* and *Spartan 3E* development boards. The outputs were connected to the LED displays on the development boards [25, 26] in order to verify the correct functionality of the filter

design. The statistics taken from *Virtex 5* and *Spartan 3E* synthesis reports are given in the table below. Also, see Appendix D.4 for the difference between the *Spartan 3E* and *Virtex 5* in terms of logic resources memory resources and I/O resources [27]. Delay considered in Figure 3.3 is taken from synthesis report rather than the report after place and route (See Appendix F-1) as our experience with testing complex adder circuits indicates that the former report provides a more accurate match with experimental measurements. (See Appendix F-2 for detailed Synthesis report on delay)

Table 3.3: Logic cell Statistics of the *Spartan 3E* and *Virtex 5* FPGAs implementing an array of Fourteen PicoBlazes

BOARDs used	Number of Slice LUTS (= Logic + Memory + route-thru) (Used/Available)	Number used for Logic	Number used for Memory	Number used for route-thru	Number of BRAM Units	Number of Slice Registers (Used/Available)	Average Fanout of Non-Clock Nets	Total Delay = (delay for Logic + route)	Delay for Logic	Delay for Route
SPARTAN-3E	4,320/ 9,312 (45%)	3,320	952	48	14/20 (20%)	2,890 / 9,312 (31%)	3.83	11.739ns	8.138ns (69.3%)	3.601ns (30.7%)
VIRTEX - 5	2,975 / 69,120 (4%)	2,582	364	29	14/148 (9%)	2,890 / 69,120 (4%)	4.04	5.915ns	3.272ns (55.3%)	2.643ns (44.7%)

The array of fourteen PicoBlaze microcontrollers consumes half of the available logic and memory resources of the *Spartan 3E* FPGA. This is close to the limit of what can be achieved on this FPGA chip (for efficient placement and routing on an FPGA, somewhere between 50% and 70% resource utilization is considered a reasonable upper limit). As expected, the *Virtex 5* FPGA, with its more complex configurable logic block implementation is able to implement this design using fewer logic slices. In addition, as the *Virtex 5* FPGA has roughly seven times more logic slices available compared with the *Spartan 3E* FPGA, only 4% of the logic and memory resources are consumed. Thus, the *Virtex 5* FPGA is able to implement a much larger array of PicoBlaze microcontrollers.

3.5 Summary

This chapter has explained the method of using FIFO buffers for communicating between arrayed PicoBlaze units. The optimum number of FIFO buffers in a wrapper design were considered and the approach for implementing an array PicoBlaze microcontrollers was explained. Finally, an FIR filter application was implemented and its synthesis statistics for two types of Xilinx FPGAs are reported. The next chapter will discuss the shared memory approach for communicating between soft processor cores.

Chapter Four
Shared Memory Inter-processor Communication

In this chapter, the shared memory mode of communication between processors is discussed. First, PicoBlaze processors with two different options of sharing Block RAM are described. Second, the need for arbitration is discussed. Third, different arbitration schemes are shown. Finally, a setup is designed which has an efficient arbiter, shared memory and four PicoBlaze microcontrollers. Also, different statistics from *Spartan 3E* and *Virtex 5* FGPA development boards are tabulated from the implemented experiment.

4.1 Shared Memory

Multiprocessor systems may be further divided into either loosely or tightly coupled systems. In a loosely coupled system, each processor has a private local memory and the messages including commands and data; communication among processors is performed by means of message-passing technique using FIFOs as explained in Chapter 3. On the other hand, in a tightly coupled system, i.e., a multiprocessor system, each processor may access data from a global memory, also called shared memory. It is also used as a means of communication among the processors [19, 23].

4.1.1 Shared Memory Mode of Communication

Shared memory is the most common way of passing information between processing subsystems. A shared memory system has different set of properties compared to normal (e.g. FIFO style) communication modes available. In a shared memory scheme, any processor can reference any shared memory location directly. Communication takes place through processor load and store instructions. A programmer is able to see the location of the data in memory easily. Data could be distributed across multiple processors, whose details could be abstracted away by some software, as in an Application Programming Interface (API). Access to the shared memory segment must be synchronized by some hardware/software protocol between the processors.

Shared memory is typically the fastest asynchronous mode of communication, especially when the information to be shared is large (> 1000 bytes). Shared memory also allows possible zero-copy or in-place message processing schemes. Shared memory can be built out of on-chip local memory or external memory [19].

4.1.2 Sharing BlockRAM Between Two PicoBlazes

Two PicoBlazes can share a single Block RAM (BRAM) using two procedures: using a single Block RAM sharing a common code image and using a single BRAM with separate 512-instruction memories in one BRAM.

Two PicoBlazes with a common code image is shown in Figure 4.1 below. This model can be implemented in an FPGA using the core generator in Xilinx ISE [28]. The Core Generator has different options when creating memory in which the dual port shared memory is one such option that creates a memory core with two ports. Using the dual ports, two PicoBlazes can use a common code saving an extra BRAM.

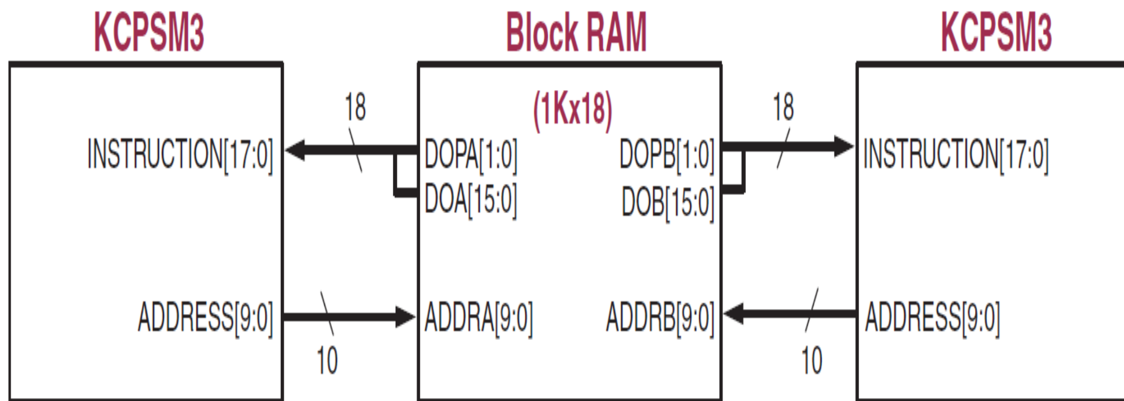


Figure 4.1: Two PicoBlazes sharing one Block RAM [20]

A second procedure is to use a separate 512-instruction memory in one Block RAM as shown in Figure 4.2. This approach can also be implemented on an FPGA using the core generator facility available in the Xilinx ISE suite. From the Figure 4.2, it can be seen that a single Block RAM is partitioned into two 512-instruction memory sections.

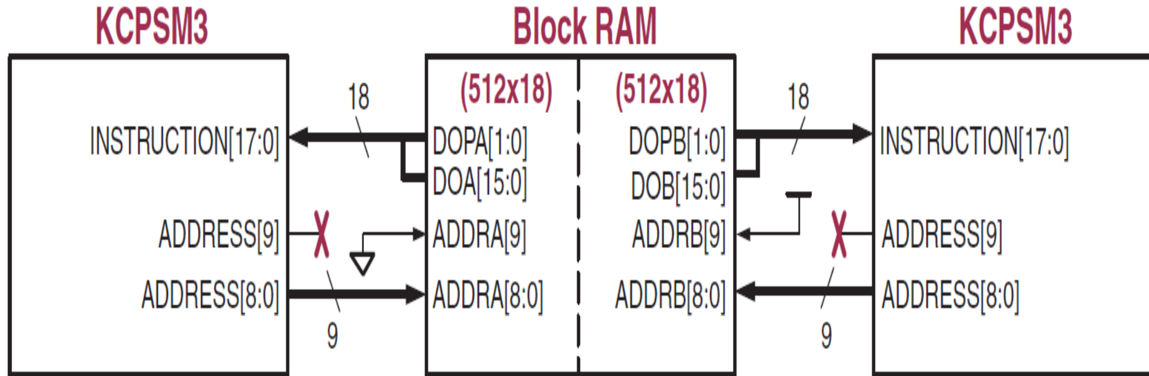


Figure 4.2: Two PicoBlazes using two separate 512- instruction memory sections from BRAM [20]

4.1.3 Shared Memory for Communication Between PicoBlaze Processors

The two techniques explained above can be used only for sharing instruction memory, but not for communication between two PicoBlaze microcontrollers. Consider multiple PicoBlaze units where the units need to read or write to a single location. Internally, each PicoBlaze processor has a 64-byte scratch pad memory that can be shared. Access to the scratch pad memory can be done using a special set of Load/Store instructions that need to guarantee that only one processor modifies the memory at a time. This approach is not pursued further since it would require changes to the internal structure of the PicoBlaze architecture and is beyond the scope of this present work. Another possible approach for customizing a shared memory mode communication is to use a single custom BRAM connected on a bus having multiple PicoBlaze processors. This approach is considered further, since it does not require changes to the PicoBlaze design itself.

4.2 Need for an Arbiter

If a shared memory for data is implemented, there is a problem when two or more processors may simultaneously request using this shared resource. Therefore, an arbitration circuit is required to resolve the contention among the competing processors and allocate the resource to the appropriate requesting processor [29]. Also, an arbiter judgment scheme and effective interrupt access mechanisms are needed to avoid memory

access problems [30]. The arbiter technique or procedure is explained in the following sections.

4.2.1 Arbitration

Arbitration is a technique introduced to reduce the burden of data exchange and also to eliminate competitions between subprocessors. A suitable arbitration scheme that will assign priority to the memory requesting systems. Good arbitration techniques are required to avoid the conflict between units accessing the shared memory units.

Consider a shared memory multiprocessor system, in which there are large numbers of processors and memory modules. Data is transferred between processor-to-processor (P2P) or processor-to-memory (P2M) via a single bus or multiple buses. The activated bus is supervised and arbitrated by a bus arbiter. In a single bus multiprocessor system, only one processor is allowed to use the bus at any time though a number of processors may request to use this bus, simultaneously. A predictable arbiter is required to schedule the memory access groups in order to bound latency [31]. Arbiter should be designed considering the effective access mechanism and improving resource utilization ratio. Today, there are plenty of arbitration schemes on the market, out of which a few are selected and discussed in Appendix E.1.

In this chapter, the focus will be on a simple scheme known as the round-robin arbiter. It has two kinds of approaches: the Bus Arbiter and the Switch Arbiter. This arbiter is considered as the efficient arbiter for implementation with the PicoBlaze processors and a shared memory scheme. Details of this arbiter model are discussed in following section.

4.3 Round-Robin Arbiter

A round-robin Bus Arbiter or the Switch Arbiter design guarantees the system will not suffer from any kind of starvation among the masters (i.e., the units requesting access to the shared memory). They allow any unused time slot to be allocated to a master whose turn is later and who is ready in the unused time slot. For several reasons the Bus arbiter model of Round-robin arbiter is preferred over the Switch arbiter and

hence the future sections will deal more with the Bus Arbiter model. The working protocol of a round-robin Bus Arbiter is described below [32].

In each cycle of the Bus Arbiter, one of the masters (processors) who are in round-robin order is given a priority. The particular master has to wait until time for the priority arises for accessing the shared memory resource. In case the token-holding master does not require access in that cycle, the master or the processor with the next highest priority (the processor who sends the next request) is chosen and is granted the permission to access the shared memory. After completing the cycle, the highest priority master then passes the token in round-robin order to the next master which then will have the next time slot. This process repeats until all the requests are granted completely.

4.3.1 Implementation of a Round-Robin Bus Arbiter

The Bus Arbiter consists of a D-flip-flop, logic blocks that set the priority, and M M-input OR gates. M x M priority logic block is implemented in combinational logic implementing the logic function of Table 1. Also an M-bit ring counter is present in the Arbiter. The M value as 4 as the experiment to be conducted has four processors and one global memory unit. Therefore, a 4×4 priority block and 4-bit ring counter are used in arbiter. The priority of inputs is placed in descending order from *in[0]* to *in[3]* in the priority logic blocks having logic 1 through 3. Thus, *in[0]* always has the highest priority, *in[1]* has the next priority, and *in[2]* has the third highest priority and *in[3]* has the lowest priority. In order to implement a Bus Arbiter, a token concept from a token ring in a network is employed. The possession of a token allows a priority logic block to be enabled. As the priority logic block has a different order of inputs (request signals), the priority of request signals varies with the chosen priority logic block [32]. The token is implemented in a 4-bit ring counter. A Bus Arbiter with four requests is coded in VHDL. Figure 4.3 shows the Round-robin Arbiter unit with four priority logic blocks.

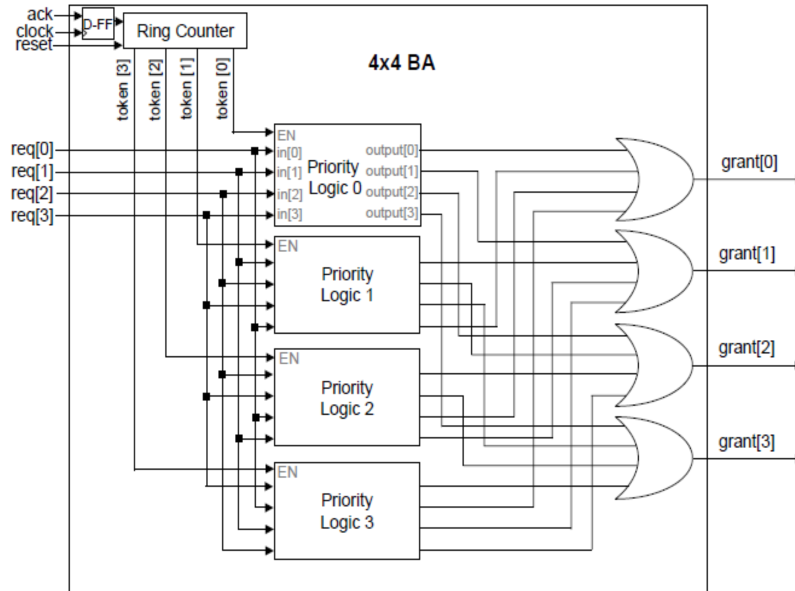


Figure 4.3: Round-robin Arbiter [28]

The four output bits of the ring counter act as the enable signals to the priority logic blocks called *token[0]*, *token[1]*, *token[2]* and *token[3]*. Here, only one enabled priority logic block can get a grant signal. A D flip-flop is used to delay the *ack* signal by one arbitration cycle. The delayed *ack* signal pulls a trigger to the ring counter in order to make the content of the ring counter get rotated by one bit. Thus, for every cycle the token bit is rotated left, with 1000 rotating to 0001 and the token is initialized to one at the reset phase. For a four bit ring counter it will be 0001. The overall concept here is to have a single '1' output by the ring counter. In the round-robin algorithm, each master must wait no longer than *(M-1)* time slots, the period of time allocated to the chosen master, until the next time it receives the token. If the owner of the time slot has nothing to send then the time slot can also be used by another master. This protocol guarantees a dynamic priority assignment to the bus requesters without starvation. The Generated Bus Arbiter is fair, fast, and has a low and predictable worst-case wait time. Therefore, the Bus Arbiter is well preferred over the available arbiter techniques [32, 33].

4.3.2 Round-Robin Arbiter with Four Processors

Four processors sharing a memory can be implemented using the round-robin Bus Arbiter. In this case, there will be four requests and four grants as described above.

Consider the case where all processors are requesting the shared memory. At a point in time only the processor with the highest priority in the ring has the opportunity to access the shared memory. From Figure 4.4 it can be seen that processor 1 has the higher priority in the ring counter and hence is granted permission to access the memory. All the other processors on the bus have to wait until the processor 1 clears the grant or accesses the memory location.

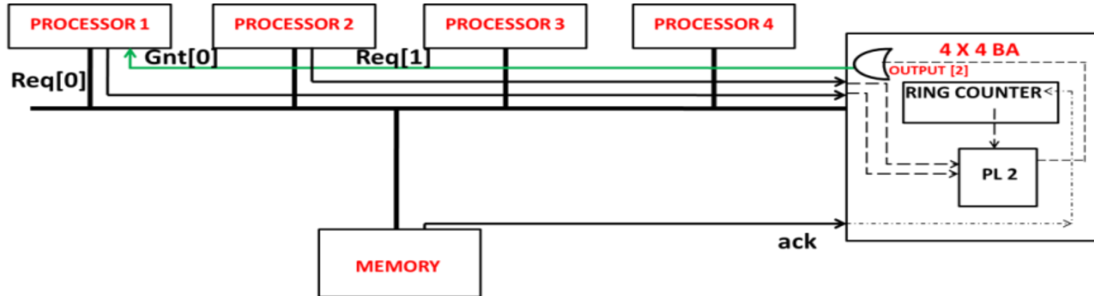


Figure 4.4: Round-robin Arbiters with four Processors [32]

The same setup can be used for the PicoBlaze processors as well. The whole setup for the system is explained in the next section.

4.4 PicoBlazes Using Round-Robin Arbiter

This section explains the setup taken as an experiment to make use of the shared memory mode of communication using the round-robin Bus Arbiter scheme.

4.4.1 Four PicoBlazes Using Round-Robin Arbiter

Consider a setup where four PicoBlaze processors are trying to access a shared memory using the Round-robin Bus Arbiter architecture. The four processors can send four requests, *request1*, *request2*, *request3*, and *request4*, to the arbiter. In return, the Bus Arbiter can send four grants, *grant1*, *grant2*, *grant3* and *grant4*, to the four PicoBlaze units.

4.4.2 Implementation Description of Shared Memory with Round-robin Arbiter

Consider four PicoBlaze processors trying to access the common shared memory. Arbitration is maintained for a case where all the four PicoBlaze units should access the shared memory at the same point in time. The Round-robin Bus Arbiter is connected to the PicoBlaze units using their I/O ports. Requests and grants are sent and received

through the output and input ports respectively. A tri-state buffer is implemented and is replicated to all four processors. A tri-state buffer is chosen which allows the data and address lines of the PicoBlaze processor to be connected to the data bus and address bus, respectively. A memory core is created in Xilinx ISE core generator and is used as a shared or global memory for all the four PicoBlazes.

The PicoBlaze processors are programmed to access the memory at the same instant. In this case, the arbiter judges the requests and grants the request to the PicoBlaze having higher priority in the ring and keeps the other PicoBlaze units in wait state. Once the PicoBlaze processors receive the grant, they complete the communication with the memory and clear the request. Once the request is cleared, the arbiter jumps to the next PicoBlaze processor in the ring counter and grants request to it. This happens in a round robin cycle mode. The first PicoBlaze unit can expect the grant again after going round through all the other PicoBlaze units. Thus, the cycle in a circle or ring mode is obtained. In the case there is no request sent by a particular PicoBlaze unit, the arbiter just skips that unit and gives the priority to the next PicoBlaze unit.

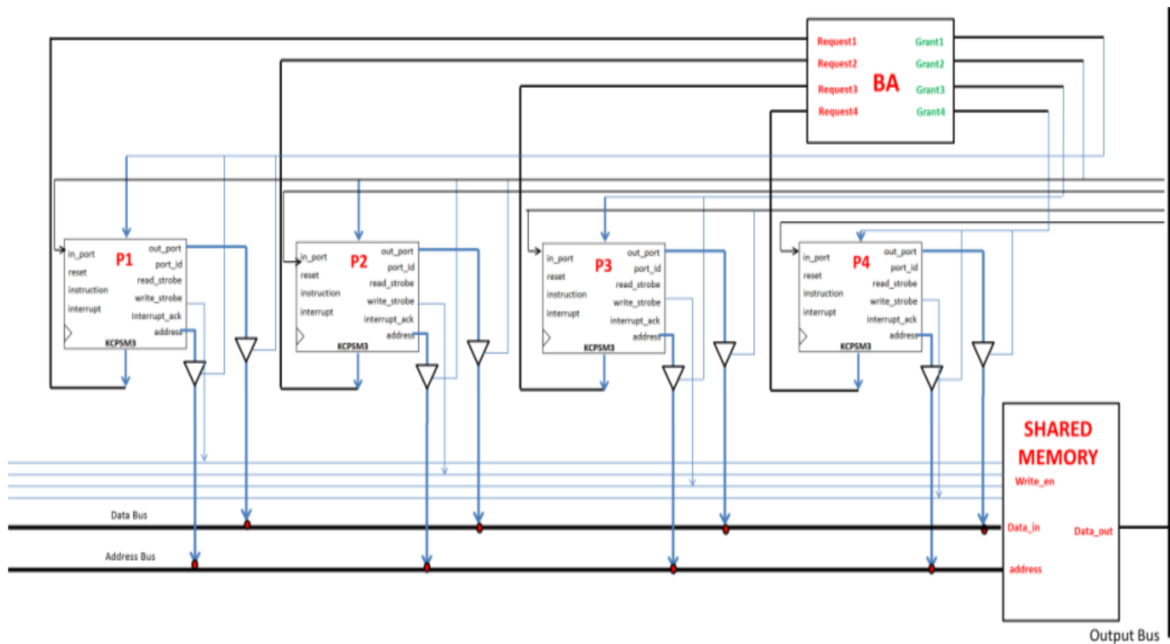


Figure 4.5: Four PicoBlazes with Shared RAM mode communication using Bus Arbiter

Using the Bus arbiter, the shared memory should be accessible by the PicoBlaze units without any conflicts. But, there are pros and cons involved in this shared memory mode of communication. Delay considered in Figure 4.1 is taken from Synthesis report as the delay after place and route (See Appendix F-1) is more and not the best case possibility in this experiment as there are different resources considered to be optimized in routing. (See Appendix F-2 for detailed Synthesis report on delay)

Table 4.1: Logic cell Statistics of the *Spartan 3E* and *Virtex 5* FPGAs implementing four PicoBlaze processors and a Shared memory

BOARDs used	Number of Slice LUTS (= Logic + Memory + route-thru) (Used/Available)	Number used for Logic	Number used for Memory	Number used for route-thru	Number of Block RAM (Used/Available)	Number of Slice Flip Flops (Used/Available)	Average Fanout of Non-Clock Nets	Total Delay = (delay for Logic + route)	Delay for Logic	Delay for Route
SPARTAN-3E	783 / 9,312 (8%)	503	272	8	5 / 20 (25%)	352 / 9,312 (3%)	3.95	8.611 ns	6.055ns (70.3%)	2.566ns (29.7%)
VIRTEX - 5	516 / 69,120 (1%)	403	104	9	5 / 148 (3%)	352 / 69,120 (1%)	3.64	4.586 ns	2.937ns (64%)	1.649ns (36%)

The summary of the two interprocessor communications, FIFO and shared memory is discussed below.

4.4.3 Observations

After implementing the design on the *Spartan 3E* and *Virtex 5* boards, it can be concluded that the FIFO-style of communication is generally preferred over the Shared memory mode of communication (provided occupation of resources in the FPGA is not a restriction). In detail, utilization of routing cell resources and memory units in a FIFO-style of communication is more as each PicoBlaze unit has its own resources like FIFO buffers and memory units. If the resources are not a factor, the FIFO-style of communication is preferred over shared memory. Also, designing a wrapper and replicating them to form an array is easier than analyzing the details involved in implementing shared memory, such as optimizing the arbitration scheme. The neighboring PicoBlaze processors can communicate better using a FIFO buffer. Furthermore, parallelism is highly maintained in the FIFO-style of communication and not in the shared memory mode as only one PicoBlaze can access the memory per clock

cycle when using shared memory. Furthermore, shared memory reduces the amount of BRAM available on an FPGA chip. This can be an important consideration as BRAM is often a scarce resource on an FPGA chip. However, the shared memory scheme does allow large blocks of data to be shared easily between processors. Unlike the FIFO-style of communication, shared memory communication is not restricted to the nearest neighbors allowing this scheme to be easily scaled to a larger number of processors that share a common memory.

Overall, the PicoBlaze array with a FIFO-style of communication is considered more efficient in terms of routing logic cell resources and is preferred if nearest neighbor communication is acceptable. If BRAM resources are not scarce and large amounts of data need to be shared or scalability is important, then the shared memory mode of communication should be implemented.

4.5 Summary

This chapter on the whole discussed the shared memory mode of communication. This chapter also described the different arbitration schemes available. Also, an experimental setup for shared memory with PicoBlaze processors using a round-robin arbiter was discussed. Finally, some observations were given and different statistics were tabulated for *Virtex 5* and *Spartan 3E* boards. The advantages and disadvantages of Shared memory and FIFO-mode of interprocessor communications were considered.

Chapter Five
Conclusions and Future Work

5.1 Conclusion

This thesis has investigated methods for enabling efficient interprocessor communication between soft processor cores implemented on FGPAs. The 8-bit PicoBlaze soft processor core from Xilinx was selected as the soft processor core since it has established itself as a reliable microcontroller for use in embedded systems implemented on reconfigurable fabrics. The two most common communication schemes were examined: Mailboxes and Shared Memory.

In the FIFO-style of communication, a wrapper is designed and an array of PicoBlaze units is produced. A simple FIR filter is also implemented and different statistics are tabulated. In the shared memory style of communication, details of the required arbitration are studied and an arbiter suitable for use with the PicoBlaze processor (i.e., round-robin arbiter) is explained. Also, the shared memory mode study concludes by considering a setup with four PicoBlaze units and a shared memory having a round-robin bus arbiter as the arbitration scheme. Different statistics between the FIFO style of communication and shared memory are discussed in which each has their own advantages depending on the resources and nearest neighbor connection requirements. Statistics from the *Spartan* and *Virtex* FPGAs were presented. Overall, this thesis has provided guidelines for interprocessor communication for an array of soft processor cores which will enable a designer to efficiently produce multiprocessor implementations on FPGAs.

5.2 Future work

The work in this thesis can be extended in several ways. First, the array of PicoBlaze units should be implemented in an embedded system and analyzed for performance under real-time conditions. The design method developed for the array of

PicoBlazes will allow the designer to easily insert an array of PicoBlazes into such embedded system applications. In particular, the wrapper for the FIFO-style of communication allows arrays of arbitrary sizes to be implemented, constrained only by the resources of the target FPGA. Second, using an array of PicoBlaze processors on more advanced FPGA platforms can be examined. The new *Virtex 6* FPGA boards with PCI-e interfaces recently acquired by our lab will allow an array of soft processor cores to communicate easily and rapidly with the host computer. This opens up the possibility of studying the partitioning of algorithms between a traditional microprocessor-based system and an array of reconfigurable processors. In addition, Xilinx has also recently introduced a *Virtex 7* FPGA (built at the 28 nm technology node) as well as an upgraded version of the PicoBlaze processor (*KCPSM6*). As the PicoBlaze arrays have been specified in VHDL, the designs can be easily ported to other families of FPGAs. Third, the development of fault-tolerant systems using an array of PicoBlaze microcontrollers should be investigated. Fault tolerance is vital for mission critical systems such as military and space-based systems. An idea is to use one PicoBlaze in an array to monitor its neighboring PicoBlaze units for correct performance and to take corrective action when a fault is detected.

References

[1] B. Othman, S. Salem, and B. Saoud, "MPSoC design of RT control applications based on FPGA SoftCore processors," in *15th IEEE International Conference on Electronics, Circuits and Systems,* pp. 404-409, 2008.

[2] M. Hubner, K. Paulsson, and J. Becker, "Parallel and Flexible Multiprocessor System-On-Chip for Adaptive Automotive Applications based on Xilinx MicroBlaze Soft-Cores," in *19th IEEE International Conference on Parallel and Distributed Processing Symposium*, p. 149, 2005.

[3] A. Kumar, S. Fernando, H. Yajun, B. Mesman, and H. Corporaal, "Multi-Processor System-Level Synthesis for Multiple Applications on Platform FPGA," in *International Conference on Field Programmable Logic and Applications,* pp. 92-97, 2007.

[4] Zain-ul-Abdin and S. Bertil, "Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing," in *Microprocessors and Microsystems*, vol. 33, no. 3, pp. 161-178, 2009.

[5] XPP, XPP III Processor Overview, http://www.pactxpp.com/, 2008.

[6] M.B. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, and J. Kim, "Evaluation of the Raw microprocessor: an exposed-wire-delay architecture for ILP and streams," in *31st Annual International Symposium on Computer Architecture*, pp. 2- 13, 2004.

[7] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, and B. Baas, "Architecture and Evaluation of an Asynchronous Array

of Simple Processors," in *Journal of VLSI Signal Processing Systems,* vol. 53, no. 3, pp. 243-259, 2008.

[8] Intel Microprocessor, *Intel Pentium cancels 4 Ghz P4,* http://www.mpronline.com, 2011.

[9] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*. Morgan kaufmann, Fourth Edition, 1993.

[10] T. Kuroda, "CMOS design challenges to power wall," in *Microprocessors and Nanotechnology Conference,* pp. 6-7, 2001.

[11] W. Zhao and Y. Cao, "New Generation of Predictive Technology Model for Sub-45 nm Early Design Exploration," in *IEEE Transactions on Electron Devices,* vol. 53, no. 11, pp. 2816-2823, 2006.

[12] IBM, Z OS, http://www-03.ibm.com/systems/z/os/zos/, 2010.

[13] Intel, Many Integrated Core Architecture, http://www.intel.com/technology/architecture-silicon/mic/index.htm, 2011.

[14] Intel, Teraflops Research Chip, http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf, 2011.

[15] Intel, Many Integrated Core Architecture, http://www.intel.com/technology/architecture-silicon/mic/index.htm, 2011.

[16] E. Mirsky, and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 157-166, 1996.

[17] FPOA Design Flow, Revision 1.5, MathStar, 13th March 2008.

[18] T. Dorta, J. Jimenez, J. L. Martín, U. Bidarte, and A. Astarloa, "Reconfigurable Multiprocessor Systems: A Review," in *International Journal of Reconfigurable Computing*, vol. 2010.

[19] Xilinx, Designing Multiprocessor Systems in Platform Studio, http://www.xilinx.com/support/documentation/white_papers/wp262.pdf, 2007.

[20] Xilinx, PicoBlaze 8-bit Embedded Microcontroller User Guide, http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf, 2011.

[21] Xilinx, *PicoBlaze Lounge*,http://www.xilinx.com/products/intellectual-property/picoblaze.htm, 2011.

[22] P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. Hoboken, NJ: Wiley-Interscience, 2008.

[23] H. Ruckdeschel, H. Dutta, F. Hannig, and J. Teich, "Automatic FIR Filter Generation for FPGAs," in *SAMOS*, pp. 51-61, 2005.

[24] H. Dutta, D. Kissler, F. Hannig, A. Kupriyanov, J. Teich, and B. Pottier, "A holistic approach for tightly coupled reconfigurable parallel processors", in *presented at Microprocessors and Microsystems - Embedded Hardware Design*, pp. 53-62, 2009.

[25] Xilinx, *Spartan 3E FPGA Starter Kit Board User Guide*, www.xilinx.com, June 20, 2008.

[26] Xilinx, *Virtex 5 Product Specification*, www.xilinx.com, February 6, 2009.

[27] Xilinx, Product Selection Guide, http://www.xilinx.com/publications/matrix/Product_Selection_Guide.pdf, 2011.

[28] Xilinx, *Xilinx Integrated Software Environment (ISE)*,http://www.xilinx.com/itp/xilinx5/help/iseguide/iseguide.htm, 2008.

[29] J. Tu and C. Chen, "An effective bus-based arbiter for processors communication," in *18th International Conference on Advanced Information Networking and Applications*, vol. 2, pp. 236- 240, 2004.

[30] Y. Hu, Y. Liu, and W. Jing, "Design of SSCMP with Arbiter and Shared Data Memory Interrupt," in *International Symposium on High Density packaging and Microsystem Integration,* pp. 1-4, 2007.

[31] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable SDRAM memory controller," in *5th IEEE/ACM/IFIP International Conference onHardware/Software Codesign and System Synthesis,* pp. 251-256, 2007.

[32] E.S. Shin, V.J. Mooney, and G.F. Riley, "Round-robin Arbiter Design and Generation," in *15th International Symposium on System Synthesis,* pp. 243- 248, 2002.

[33] A. Zitouni, M. Abid, and R. Tourki, "Arbitration schemes synthesis approach for multiprocessor systems," in *IEEE Workshop on Signal Processing Systems,* pp. 499-508, 1998.

Appendix A: Assembly Code for Button Press Experiment

A.1 PicoBlaze 1 to FIFO Assembly Code for Button Press Experiment

```
; PicoBlaze_to_FIFO.PSM
; Button Press Experiment Assembly level Programming code
; Description:
; 1. Generate numbers
; 2. Check FULL flag
; 3. Send data into FIFO
; 4. Jump again to send next data
;---------------------------------------------------

namereg s0, k
namereg s3, success
namereg s4, fdata
namereg sf, sw_in
namereg s6, dout
;------------input port definitions---------------------
constant sw_port, 02 ;8-bit switches
constant flags, 01 ;
;------------output port definitions---------------------
constant dout_port, 06
;-------------------

constant MAX, 7F
load k, 00
constant MAX1, FF
loop_body:
        load dout,k
fail:                           ; Load with a value
        call write_to_fifo  ; write it into FIFO
        compare success, 00 ; if written proceed next
        jump z, fail        ; if failed to write jump to fail,
        add k, 01           ; send next number
        compare k, MAX          ; if number reaches 7F halt program there
        jump z, forever
        jump loop_body          ; Else be back to loop
forever:
        jump forever
;---------------------

write_to_fifo:
          input fdata, flags     ; load flags input to fdata
          and fdata, 02          ; masking the lower bit
```

```
            compare fdata, 02       ; test full flag
            jump nz, ok_to_wr  ; if not full -->write to fifo
            load success, 00       ; else load failure
            return
ok_to_wr:

            output dout, dout_port;send switch input to out_port1
            load success, 01          ; success!!
            return
;---------------------------------------------------------------------------------------
```

A.2 FIFO to PicoBlaze 2 Assembly Code for Button Press Experiment

```
; Fifo_to_PicoBlaze 2.PSM
; Button Press Experiment Assembly level Programming code
; Description:
; 1. For first time Check for Full case of FIFO
; 2. After FIFO is full, Check for Button is pressed or not
; 3. Read data when button pressed
; 4. Check Empty status and read data until it is not empty.
;
namereg s3, success
namereg s4, fdata
namereg sf, f_in
namereg s6, bpress
;------------input port definitions---------------------
constant fifo, 02 ;8-bit switches
constant flags, 01 ;
constant btn, 04
;------------output port definitions---------------------
constant dout_port, 06
;-----------------------------------------------------
full_test:
            input fdata, flags ; check full flag
            and fdata, 02
            compare fdata, 02
            jump z, btn_chk    ; check button status if Pressed
jump full_test

empty_test:
            input fdata, flags ; load flags input to fdata
            and fdata, 01      ; masking the lower bit
            compare fdata, 01  ; test empty flag
```

```
        jump nz, btn_chk   ; if not empty go to loop
jump empty_test

btn_chk:
        input bpress, btn  ; input button status
        compare bpress, 01 ; if pressed goto input_fifo
        jump z, input_fifo
jump btn_chk

input_fifo:
        input f_in, fifo   ; input data if pressed
        output f_in, dout_port ; Send data out
jump empty_test
```

Appendix B: Assembly Code for Counting Numbers Experiment

B.1 PicoBlaze 4 Assembly Code for Counting Numbers Experiment

```
; Code for Destination PicoBlaze (PicoBlaze 4.psm)
; This code continuously reads numbers coming from all directions
; The directions are West, South, East, Diagonal
; Description:
; 1. Check Empty status of all FIFOs(one after the other)
; 2. If not empty proceed next; else be in loop until not empty
; 3. Write data
; 4. go to next FIFO
namereg s1, addr  ;reg for temporary mem & i/o port addr
namereg s2, i     ;general-purpose loop index
namereg s5, j
namereg s3, success
namereg sf, sw_in
namereg s6, dout
namereg s8, m
namereg s9, n
namereg sa, data1
namereg sb, data2
namereg sc, data3
namereg sd, data4
namereg s4, fdata1
namereg se, fdata2
namereg s7, fdata3
namereg s0, fdata4
;-----------input port definitions--------------------
constant west, 00 ;
constant south, 01 ;
constant diagonal, 02 ;
constant east, 03 ;
constant flag_w, 04 ;
constant flag_s, 05 ;
constant flag_d, 06 ;
constant flag_e, 07 ;
;-----------output port definitions--------------------
constant dout_port, 00
clr_data_mem:
  load i, 40            ;unitize loop index to 64
  load sw_in, 00
clr_mem_loop:
  store sw_in, (i)
```

```
sub i, 01              ;dec loop index
  jump nz, clr_mem_loop   ;repeat until i=0

;-------------------

constant MAX1, FF

wes:
        input fdata4, flag_w  ; check FIFO 2 for Empty Condition
        and fdata4, 01
        compare fdata4, 01
        jump z, wes
        input data4, west         ; Input data from FIFO2 if not empty
        output data4, dout_port ; output FIFO1 data


sout:
        input fdata2, flag_s  ; check FIFO 2 for Empty Condition
        and fdata2, 01
        compare fdata2, 01
        jump z, sout
        input data2, south        ; Input data from FIFO2 if not empty
        output data2, dout_port ; output FIFO1 data
diag:
        input fdata1, flag_d    ; Check FIFO 1 for Empty condition
        and fdata1, 01
        compare fdata1, 01
        jump z, diag
        input data1, diagonal   ; Input data from FIFO1 if not empty
        output data1, dout_port ; output FIFO1 data

eas:

        input fdata3, flag_e  ; check FIFO 2 for Empty Condition
        and fdata3, 01
        compare fdata3, 01
        jump z, eas
        input data3, east          ; Input data from FIFO2 if not empty
        output data3, dout_port ; output FIFO1 data


jump wes
;----------------------------------------------------------------------------------------------------
```

Appendix: B (Continued)

B.2 PicoBlaze 1 (P1) Assembly Code for Counting Numbers Experiment

```
;PICOBLAZE 1.psm
; description: This Program sends the data from PB1(Diagonal) to Destination (P4)
; 1. generate numbers
; 2. check FULL case
; 3. If not FULL send data into FIFO
namereg s0, k
namereg s3, success
namereg s4, fdata
namereg sf, sw_in
namereg s6, dout
;------------input port definitions---------------------
constant flags, 0A
;------------output port definitions---------------------
constant dout_port, 02
;-------------------
constant MAX, 17
load k, 03
loop_body:
        load dout,k
fail:
        call write_to_fifo      ; call routine write into fifo
        compare success, 00   ; if data sent proceed to next number
        jump z, fail            ; if not written try to write
        add k, 04               ; increment number by integer 4 everytime
        compare k, MAX                  ; if maximum value reached then terminate
        jump z, forever
        jump loop_body                  ; else be in loop "loop_body"
forever:
        jump forever
;---------------------

write_to_fifo:
            input fdata, flags     ; load flags input to fdata
            and fdata, 02          ; masking the lower bit
            compare fdata, 02   ; test full flag
            jump nz, ok_to_wr  ; if not full -->write to fifo
            load success, 00      ; else load failure
            return
ok_to_wr:
        output dout, dout_port;send switch input to out_port1
        load success, 01         ; success!!
        return
```

B.3 PicoBlaze 2 Assembly Code for Counting Numbers Experiment

```
;PICOBLAZE 2.psm
; description: This Program sends the data from PB1(South) to Destination (P4)
; 1. generate numbers
; 2. check FULL case
; 3. If not FULL send data into FIFO
namereg s0, k
namereg s3, success
namereg s4, fdata
namereg sf, sw_in
namereg s6, dout
;------------input port definitions---------------------
constant flags, 09
;------------output port definitions---------------------
constant dout_port, 03
;-------------------
constant MAX, 16
load k, 02
loop_body:
        load dout,k
fail:
        call write_to_fifo      ; call routine write into fifo
        compare success, 00   ; if data sent proceed to next number
        jump z, fail               ; if not written try to write
        add k, 04                  ; increment number by integer 4 everytime
        compare k, MAX                  ; if maximum value reached then terminate
        jump z, forever
        jump loop_body                   ; else be in loop "loop_body"
forever:
        jump forever
;---------------------
write_to_fifo:
            input fdata, flags     ; load flags input to fdata
            and fdata, 02          ; masking the lower bit
            compare fdata, 02   ; test full flag
            jump nz, ok_to_wr  ; if not full -->write to fifo
            load success, 00      ; else load failure
            return
ok_to_wr:
        output dout, dout_port;send switch input to out_port1
        load success, 01          ; success!!
        return
```

Appendix: B (Continued)

B.4 PicoBlaze 3 Assembly code for Counting Numbers Experiment

```
;PICOBLAZE 3.psm
; description: This Program sends the data from PB3(East) to Destination (P4)
; 1. generate numbers
; 2. check FULL case
; 3. If not FULL send data into FIFO
namereg s0, k
namereg s3, success
namereg s4, fdata
namereg sf, sw_in
namereg s6, dout
;------------input port definitions---------------------
constant flags, 0B
;------------output port definitions---------------------
constant dout_port, 01
;-------------------
constant MAX, 18
load k, 04
loop_body:
        load dout,k
fail:
        call write_to_fifo      ; call routine write into fifo
        compare success, 00   ; if data sent proceed to next number
        jump z, fail               ; if not written try to write
        add k, 04                  ; increment number by integer 4 everytime
        compare k, MAX                  ; if maximum value reached then terminate
        jump z, forever
        jump loop_body                  ; else be in loop "loop_body"
forever:
        jump forever
;---------------------

write_to_fifo:
            input fdata, flags     ; load flags input to fdata
            and fdata, 02          ; masking the lower bit
            compare fdata, 02   ; test full flag
            jump nz, ok_to_wr  ; if not full -->write to fifo
            load success, 00      ; else load failure
            return
ok_to_wr:
        output dout, dout_port;send switch input to out_port1
        load success, 01          ; success!!
        return
```

Appendix: B (Continued)

B.5 PicoBlaze 5 Assembly code for Counting Numbers Experiment

```
;PICOBLAZE 5.psm
; description: This Program sends the data from PB5(West) to Destination (P4)
; 1. generate numbers
; 2. check FULL case
; 3. If not FULL send data into FIFO
namereg s0, k
namereg s3, success
namereg s4, fdata
namereg sf, sw_in
namereg s6, dout
;-----------input port definitions--------------------
constant flags, 08
;-----------output port definitions--------------------
constant dout_port, 00
;-------------------
constant MAX, 15
load k, 01
loop_body:
        load dout,k
fail:
        call write_to_fifo      ; call routine write into fifo
        compare success, 00     ; if data sent proceed to next number
        jump z, fail            ; if not written try to write
        add k, 04               ; increment number by integer 4 everytime
        compare k, MAX                  ; if maximum value reached then terminate
        jump z, forever
        jump loop_body                  ; else be in loop "loop_body"
forever:
        jump forever
;---------------------
write_to_fifo:
            input fdata, flags      ; load flags input to fdata
            and fdata, 02           ; masking the lower bit
            compare fdata, 02       ; test full flag
            jump nz, ok_to_wr       ; if not full -->write to fifo
            load success, 00        ; else load failure
            return
ok_to_wr:
        output dout, dout_port;send switch input to out_port1
        load success, 01        ; success!!
        return
;----------------------
```

Appendix C: Code for 14 PicoBlazes FIR Filter Experiment

C.1 N-tap and T-input FIR Filter Code in C Language

```
#include<stdio.h>                    //Header files
//#include<conio.h>
//venkata mandala Code for FIR FILTER Design May/27/2011
main()
{
int T,N;
int i=0,j=0,k,l;
float Y[127], A[127], U[127], Z[127], Y_OUT[127];

printf("---FIR FILTER Design--- \n");

printf("Enter the Number of Filter Inputs \n");          //accepting Taps
scanf("%d",&T);

printf("Enter the Number of Filter Taps \n");            //accepting Inputs
scanf("%d",&N);

        for(i=0; i<T; i++)
    {
        printf("Enter coefficient for U[%d]",i);
        scanf("%f",& U[i]);
    }

        for(j=0; j<N; j++)
   {
        printf("Enter coefficient for A[%d]",j);
        scanf("%f",&A[j]);
   }

                for(i=0; i<T; i++)
                 {
                  for(j=0; j<N; j++)
                   {

                    k=i-j;
                        if(i<j)
                        {
                        U[k] =0;
                        }
                        if(i==0 && j==0)
                        {
```

```
                Z[j] = A[j] * U[k];
                 Y_OUT[j] = Z[j];
                }

                if(i==0 && j>0)
                {
                Z[j] = A[j] * U[k];
                Y_OUT[j] = Y_OUT[j-1] + Z[j];
                }


                if(i>0)
                {

                Z[j] = A[j] * U[k];
                Y_OUT[j] = Y_OUT[j-1] + Z[j];
                }

                if(j==N-1)
                {
                printf("Output Y[%d] --> %f  \n",i,Y_OUT[j]);
                printf("---------------- \n");
                }

            }
        }
}
```

C.2 PicoBlaze Array Code in VHDL for FIR filter Design

```
---Thesis Report by Venkata Mandala----------
-- This is the Top level PicoBlaze Array declaration VHDL file--
-- Final Version Submitted on July 8th 2011-----
--
-- Name: picoArray. Vhdl------------------------
--DESCRIPTION: This top level Array should have different sources given below.
-- 1.picoblaze.vhd (Wrapper code used for Wrapper instantation)
-- 2. KCPSM3. vhd (Picoblaze 8-bit microcontroller code)
-- 3. FIFO.vhd    (main FIFO file that holds the Data)
-- 4. CLKDIV.vhd (clk_rate settings file)
-- 5. PINS.UCF (pin declaration FILE)


--DETAILS: This array has 14 picoblazes
```

Appendix: C (Continued)

```
--picoblazeU0, picoblazeU1, picoblazeU2,picoblazeU3--> generates the FIR "U[x]"
values
--PicoBlaze 1 to picoblaze 10 are different stages of PicoBlazes interconnected
--OUPUTs Y [0], Y[1], Y[2], Y[3] :taken from PicoBlaze 7 to PicoBlaze 10 Respectively

-- Declaration of Libraries
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-- Main Pin declaration File--
entity picoArray is
  port(
      clk     : in std_logic;
                    btn      : in STD_LOGIC;
                    led     : out std_logic_vector(7 downto 0)
                   );
end picoArray;
-- Architecture declaration of File starts here

architecture arch of picoArray is
-- Input Signals of Each picoblaze is given below--

signal in_westU0, in_westU1, in_westU2, in_westU3, in_west1, in_west2, in_west3,
in_west4, in_west5, in_west6, in_west7, in_west8, in_west9, in_west10 :
std_logic_vector(7 downto 0);
signal in_eastU0, in_eastU1, in_eastU2, in_eastU3, in_east1, in_east2, in_east3, in_east4,
in_east5, in_east6, in_east7, in_east8, in_east9, in_east10 : std_logic_vector(7 downto 0);
signal in_southU0, in_southU1, in_southU2, in_southU3, in_south1, in_south2,
in_south3, in_south4, in_south5, in_south6, in_south7, in_south8, in_south9, in_south10
: std_logic_vector(7 downto 0);
signal in_diagonalU0, in_diagonalU1, in_diagonalU2, in_diagonalU3, in_diagonal1,
in_diagonal2, in_diagonal3, in_diagonal4, in_diagonal5, in_diagonal6, in_diagonal7,
in_diagonal8, in_diagonal9, in_diagonal10 : std_logic_vector(7 downto 0);

-- Output Signals of Each picoblaze is given below--

signal out_westU0, out_westU1, out_westU2, out_westU3, out_west1, out_west2,
out_west3, out_west4, out_west5, out_west6, out_west7, out_west8, out_west9,
out_west10 : std_logic_vector(7 downto 0);
signal out_eastU0, out_eastU1, out_eastU2, out_eastU3, out_east1, out_east2, out_east3,
out_east4, out_east5, out_east6, out_east7, out_east8, out_east9, out_east10 :
std_logic_vector(7 downto 0);
```

```
signal out_southU0, out_southU1, out_southU2, out_southU3, out_south1, out_south2,
out_south3, out_south4, out_south5, out_south6, out_south7, out_south8, out_south9,
out_south10 : std_logic_vector(7 downto 0);
signal out_diagonalU0, out_diagonalU1, out_diagonalU2, out_diagonalU3,
out_diagonal1, out_diagonal2, out_diagonal3, out_diagonal4, out_diagonal5,
out_diagonal6, out_diagonal7, out_diagonal8, out_diagonal9, out_diagonal10 :
std_logic_vector(7 downto 0);
-- Address and Instruction Declaration for each ROM
signal addressU1, addressU0, addressU2, addressU3, address1, address2, address3,
address4, address5, address6, address7, address8, address9, address10 :
std_logic_vector(9 downto 0);
signal instructionU1,instructionU0, instructionU2, instructionU3, instruction1,
instruction2, instruction3, instruction4, instruction5, instruction6, instruction7,
instruction8, instruction9, instruction10 : std_logic_vector(17 downto 0);
-- Clocks used

signal clr, clk190, clk48: STD_LOGIC;
signal led_reg: std_logic_vector(15 downto 0);
-- Status signals taken from Wrapper--

signal status_west1_in, status_south1_in, status_diagonal1_in, status_east1_in:
std_logic_vector(7 downto 0);
signal status_west2_in, status_south2_in, status_diagonal2_in, status_east2_in:
std_logic_vector(7 downto 0);
signal status_westU0_in, status_southU0_in, status_diagonalU0_in, status_eastU0_in:
std_logic_vector(7 downto 0);
signal status_westU1_in, status_southU1_in, status_diagonalU1_in, status_eastU1_in:
std_logic_vector(7 downto 0);
signal status_westU2_in, status_southU2_in, status_diagonalU2_in, status_eastU2_in:
std_logic_vector(7 downto 0);
signal status_westU3_in, status_southU3_in, status_diagonalU3_in, status_eastU3_in:
std_logic_vector(7 downto 0);
signal status_west3_in, status_south3_in, status_diagonal3_in, status_east3_in:
std_logic_vector(7 downto 0);
signal status_west4_in, status_south4_in, status_diagonal4_in, status_east4_in:
std_logic_vector(7 downto 0);
signal status_west5_in, status_south5_in, status_diagonal5_in, status_east5_in:
std_logic_vector(7 downto 0);
signal status_west6_in, status_south6_in, status_diagonal6_in, status_east6_in:
std_logic_vector(7 downto 0);
signal status_west7_in, status_south7_in, status_diagonal7_in, status_east7_in:
std_logic_vector(7 downto 0);
signal status_west8_in, status_south8_in, status_diagonal8_in, status_east8_in:
std_logic_vector(7 downto 0);
```

```
signal status_west9_in, status_south9_in, status_diagonal9_in, status_east9_in:
std_logic_vector(7 downto 0);
signal status_west10_in, status_south10_in, status_diagonal10_in, status_east10_in:
std_logic_vector(7 downto 0);
-- Write strobes sent/recieved from wrapper for neighbour picoblaze status's

signal write_strobe_west1_in, write_strobe_south1_in, write_strobe_diagonal1_in,
write_strobe_east1_in: std_logic;
signal write_strobe_west2_in, write_strobe_south2_in, write_strobe_diagonal2_in,
write_strobe_east2_in: std_logic;
signal write_strobe_westU0_in, write_strobe_southU0_in, write_strobe_diagonalU0_in,
write_strobe_eastU0_in: std_logic;
signal write_strobe_westU1_in, write_strobe_southU1_in, write_strobe_diagonalU1_in,
write_strobe_eastU1_in: std_logic;
signal write_strobe_westU2_in, write_strobe_southU2_in, write_strobe_diagonalU2_in,
write_strobe_eastU2_in: std_logic;
signal write_strobe_westU3_in, write_strobe_southU3_in, write_strobe_diagonalU3_in,
write_strobe_eastU3_in: std_logic;
signal write_strobe_west3_in, write_strobe_south3_in, write_strobe_diagonal3_in,
write_strobe_east3_in: std_logic;
signal write_strobe_west4_in, write_strobe_south4_in, write_strobe_diagonal4_in,
write_strobe_east4_in: std_logic;
signal write_strobe_west5_in, write_strobe_south5_in, write_strobe_diagonal5_in,
write_strobe_east5_in: std_logic;
signal write_strobe_west6_in, write_strobe_south6_in, write_strobe_diagonal6_in,
write_strobe_east6_in: std_logic;
signal write_strobe_west7_in, write_strobe_south7_in, write_strobe_diagonal7_in,
write_strobe_east7_in: std_logic;
signal write_strobe_west8_in, write_strobe_south8_in, write_strobe_diagonal8_in,
write_strobe_east8_in: std_logic;
signal write_strobe_west9_in, write_strobe_south9_in, write_strobe_diagonal9_in,
write_strobe_east9_in: std_logic;
signal write_strobe_west10_in, write_strobe_south10_in, write_strobe_diagonal10_in,
write_strobe_east10_in: std_logic;
-- Write strobes of wach picoblazes are communicated using the below signals

signal ws_sU0, ws_dU0, ws_eU0, ws_wU0: std_logic;
signal ws_sU1, ws_dU1, ws_eU1, ws_wU1: std_logic;
signal ws_sU2, ws_dU2, ws_eU2, ws_wU2: std_logic;
signal ws_sU3, ws_dU3, ws_eU3, ws_wU3: std_logic;
signal ws_s1, ws_d1, ws_e1, ws_w1: std_logic;
signal ws_s2, ws_d2, ws_e2, ws_w2: std_logic;
signal ws_s3, ws_d3, ws_e3, ws_w3: std_logic;
signal ws_s4, ws_d4, ws_e4, ws_w4: std_logic;
```

Appendix: C (Continued)

```
signal ws_s5, ws_d5, ws_e5, ws_w5: std_logic;
signal ws_s6, ws_d6, ws_e6, ws_w6: std_logic;
signal ws_s7, ws_d7, ws_e7, ws_w7: std_logic;
signal ws_s8, ws_d8, ws_e8, ws_w8: std_logic;
signal ws_s9, ws_d9, ws_e9, ws_w9: std_logic;
signal ws_s10, ws_d10, ws_e10, ws_w10: std_logic;
--status signals taken out from each Picoblaze here

signal status_west1_out, status_south1_out, status_diagonal1_out, status_east1_out:
std_logic_vector(7 downto 0);
signal status_west2_out, status_south2_out, status_diagonal2_out, status_east2_out:
std_logic_vector(7 downto 0);
signal status_westU0_out, status_southU0_out, status_diagonalU0_out,
status_eastU0_out: std_logic_vector(7 downto 0);
signal status_westU1_out, status_southU1_out, status_diagonalU1_out,
status_eastU1_out: std_logic_vector(7 downto 0);
signal status_westU2_out, status_southU2_out, status_diagonalU2_out,
status_eastU2_out: std_logic_vector(7 downto 0);
signal status_westU3_out, status_southU3_out, status_diagonalU3_out,
status_eastU3_out: std_logic_vector(7 downto 0);
signal status_west3_out, status_south3_out, status_diagonal3_out, status_east3_out:
std_logic_vector(7 downto 0);
signal status_west4_out, status_south4_out, status_diagonal4_out, status_east4_out:
std_logic_vector(7 downto 0);
signal status_west5_out, status_south5_out, status_diagonal5_out, status_east5_out:
std_logic_vector(7 downto 0);
signal status_west6_out, status_south6_out, status_diagonal6_out, status_east6_out:
std_logic_vector(7 downto 0);
signal status_west7_out, status_south7_out, status_diagonal7_out, status_east7_out:
std_logic_vector(7 downto 0);
signal status_west8_out, status_south8_out, status_diagonal8_out, status_east8_out:
std_logic_vector(7 downto 0);
signal status_west9_out, status_south9_out, status_diagonal9_out, status_east9_out:
std_logic_vector(7 downto 0);
signal status_west10_out, status_south10_out, status_diagonal10_out, status_east10_out:
std_logic_vector(7 downto 0);
-- Main Program with entity declaration begins here

begin
-- Picoblaze U0 used for sending the U[0] numbers for FIR--

picoblazeU0: entity picoblaze
        port map(
                input_1 => in_westU0,
```

```
                    input_2 => in_southU0,
                    input_3 => in_diagonalU0,
                    input_4 => in_eastU0,
                    output_1 => out_westU0,
                    output_2 => out_eastU0,
                    output_3 => out_diagonalU0,
                    output_4 => out_southU0,
                    clk => clk190,
                    address => addressU0,
                    instruction => instructionU0,
                    status1 => status_westU0_out,
                    status2 => status_southU0_out,
                    status3 => status_diagonalU0_out,
                    status4 => status_eastU0_out,
                    status1_in => status_westU0_in,
                    status2_in => status_southU0_in,
                    status3_in => status_diagonalU0_in,
                    status4_in => status_eastU0_in,
                    write_strobe_west_in => write_strobe_westU0_in,
                    write_strobe_south_in => write_strobe_southU0_in,
                    write_strobe_diagonal_in => write_strobe_diagonalU0_in,
                    write_strobe_east_in => write_strobe_eastU0_in,
                    ws_s => ws_sU0,
                    ws_d => ws_dU0,
                    ws_e => ws_eU0,
                    ws_w => ws_wU0,
                    btn => btn
                    );
U0_RAM: entity U0_RAM
   port map(
   clk => clk190,
address=>addressU0,
   instruction=>instructionU0
);
-- Picoblaze U1 used for sending the U[1] numbers for FIR--

picoblazeU1: entity picoblaze
        port map(
        input_1 => in_westU1,
         input_2 => in_southU1,
                    input_3 => in_diagonalU1,
                    input_4 => in_eastU1,
                    output_1 => out_westU1,
                    output_2 => out_eastU1,
```

```
                output_3 => out_diagonalU1,
                output_4 => out_southU1,
                clk => clk190,
                address => addressU1,
                instruction => instructionU1,
                status1 => status_westU1_out,
                status2 => status_southU1_out,
                status3 => status_diagonalU1_out,
                status4 => status_eastU1_out,
                status1_in => status_westU1_in,
                status2_in => status_southU1_in,
                status3_in => status_diagonalU1_in,
                status4_in => status_eastU1_in,

                write_strobe_west_in => write_strobe_westU1_in,
                write_strobe_south_in => write_strobe_southU1_in,
                write_strobe_diagonal_in => write_strobe_diagonalU1_in,
                write_strobe_east_in => write_strobe_eastU1_in,
                ws_s => ws_sU1,
                ws_d => ws_dU1,
                ws_e => ws_eU1,
                ws_w => ws_wU1,
                btn => btn
                );
U1_RAM: entity U1_RAM
    port map(
    clk => clk190,
address=>addressU1,
    instruction=>instructionU1
);
-- Picoblaze U2 used for sending the U[2] numbers for FIR--
picoblazeU2: entity picoblaze
        port map(
                input_1 => in_westU2,
                input_2 => in_southU2,
                input_3 => in_diagonalU2,

                input_4 => in_eastU2,
                output_1 => out_westU2,
                output_2 => out_eastU2,
                output_3 => out_diagonalU2,
                output_4 => out_southU2,
                clk => clk190,
                address => addressU2,
```

```
                instruction => instructionU2,
                status1 => status_westU2_out,
                status2 => status_southU2_out,
                status3 => status_diagonalU2_out,
                status4 => status_eastU2_out,
                status1_in => status_westU2_in,
                status2_in => status_southU2_in,
                status3_in => status_diagonalU2_in,
                status4_in => status_eastU2_in,
                write_strobe_west_in => write_strobe_westU2_in,
                write_strobe_south_in => write_strobe_southU2_in,
                write_strobe_diagonal_in => write_strobe_diagonalU2_in,
                write_strobe_east_in => write_strobe_eastU2_in,

                ws_s => ws_sU2,
                ws_d => ws_dU2,
                ws_e => ws_eU2,
                ws_w => ws_wU2,
                btn => btn
                );
U2_RAM: entity U2_RAM
    port map(
    clk => clk190,
    address=>addressU2,
    instruction=>instructionU2
);
-- Picoblaze U3 used for sending the U[3] numbers for FIR--
picoblazeU3: entity picoblaze
        port map(
                input_1 => in_westU3,
                input_2 => in_southU3,
                input_3 => in_diagonalU3,
                input_4 => in_eastU3,
                output_1 => out_westU3,
                output_2 => out_eastU3,
                output_3 => out_diagonalU3,
                output_4 => out_southU3,

                clk => clk190,
                address => addressU3,
                instruction => instructionU3,
                status1 => status_westU3_out,
                status2 => status_southU3_out,
                status3 => status_diagonalU3_out,
```

```
status4 => status_eastU3_out,
                    status1_in => status_westU3_in,
                    status2_in => status_southU3_in,
                    status3_in => status_diagonalU3_in,
                    status4_in => status_eastU3_in,
                    write_strobe_west_in => write_strobe_westU3_in,
                    write_strobe_south_in => write_strobe_southU3_in,
                    write_strobe_diagonal_in => write_strobe_diagonalU3_in,
                    write_strobe_east_in => write_strobe_eastU3_in,
                    ws_s => ws_sU3,
                    ws_d => ws_dU3,
                    ws_e => ws_eU3,
                    ws_w => ws_wU3,
                    btn => btn
                    );
U3_RAM: entity U3_RAM
     port map(
     clk => clk190,
address=>addressU3,
     instruction=>instructionU3
           );
-- PicoBlaze 1 declaration--
PicoBlaze 1: entity picoblaze
          port map(
           input_1 => in_west1,
           input_2 => in_south1,
                    input_3 => in_diagonal1,
                    input_4 => in_east1,
                    output_1 => out_west1,
                    output_2 => out_east1,
                    output_3 => out_diagonal1,
                    output_4 => out_south1,
                    clk => clk190,
                    address => address1,
                    instruction => instruction1,
                    status1 => status_west1_out,
                    status2 => status_south1_out,
                    status3 => status_diagonal1_out,
                    status4 => status_east1_out,
                    status1_in => status_west1_in,
                    status2_in => status_south1_in,
                    status3_in => status_diagonal1_in,
                    status4_in => status_east1_in,
                    write_strobe_west_in => write_strobe_west1_in,
```

```
                write_strobe_south_in => write_strobe_south1_in,
                write_strobe_diagonal_in => write_strobe_diagonal1_in,
                write_strobe_east_in => write_strobe_east1_in,
                ws_s => ws_s1,
                ws_d => ws_d1,
                ws_e => ws_e1,
                ws_w => ws_w1,
                btn => btn
                );
p1_RAM: entity p1_RAM
    port map(
    clk => clk190,
address=>address1,
    instruction=>instruction1
        );
-- PicoBlaze 2 declaration--
PicoBlaze 2: entity picoblaze
        port map(
                input_1 => in_west2,
            input_2 => in_south2,
                input_3 => in_diagonal2,
                input_4 => in_east2,
                output_1 => out_west2,
                output_2 => out_east2,
                output_3 => out_diagonal2,
                output_4 => out_south2,
                clk => clk190,
                address => address2,
                instruction => instruction2,
                status1 => status_west2_out,
                status2 => status_south2_out,
                status3 => status_diagonal2_out,
                status4 => status_east2_out,
                status1_in => status_west2_in,
                status2_in => status_south2_in,
                status3_in => status_diagonal2_in,
                status4_in => status_east2_in,
                write_strobe_west_in => write_strobe_west2_in,
                write_strobe_south_in => write_strobe_south2_in,
                write_strobe_diagonal_in => write_strobe_diagonal2_in,
                write_strobe_east_in => write_strobe_east2_in,
                ws_s => ws_s2,
                ws_d => ws_d2,
                ws_e => ws_e2,
```

```
                    ws_w => ws_w2,
                    btn => btn
                    );
p2_RAM: entity p2_RAM
    port map(
    clk => clk190,
address=>address2,
    instruction=>instruction2
            );
-- PicoBlaze 3 declaration--
PicoBlaze 3: entity picoblaze
        port map(
                    input_1 => in_west3,
          input_2 => in_south3,
                    input_3 => in_diagonal3,
                    input_4 => in_east3,
                    output_1 => out_west3,
                    output_2 => out_east3,
                    output_3 => out_diagonal3,
                    output_4 => out_south3,
                    clk => clk190,
                    address => address3,
                    instruction => instruction3,
                    status1 => status_west3_out,
                    status2 => status_south3_out,
                    status3 => status_diagonal3_out,
                    status4 => status_east3_out,
                    status1_in => status_west3_in,
                    status2_in => status_south3_in,
                    status3_in => status_diagonal3_in,
                    status4_in => status_east3_in,
                    write_strobe_west_in => write_strobe_west3_in,
                    write_strobe_south_in => write_strobe_south3_in,
                    write_strobe_diagonal_in => write_strobe_diagonal3_in,
                    write_strobe_east_in => write_strobe_east3_in,
                    ws_s => ws_s3,
                    ws_d => ws_d3,
                    ws_e => ws_e3,
                    ws_w => ws_w3,
                    btn => btn
                    );
p3_RAM: entity p3_RAM
    port map(
    clk => clk190,
```

```
address=>address3,
     instruction=>instruction3
    );
-- PicoBlaze 4 declaration--

PicoBlaze 4: entity picoblaze
          port map(
             input_1 => in_west4,
                    input_2 => in_south4,
                    input_3 => in_diagonal4,
                    input_4 => in_east4,
                    output_1 => out_west4,
                    output_2 => out_east4,
                    output_3 => out_diagonal4,
                    output_4 => out_south4,
                    clk => clk190,
                    address => address4,
                    instruction => instruction4,
                    status1 => status_west4_out,
                    status2 => status_south4_out,
                    status3 => status_diagonal4_out,
                    status4 => status_east4_out,
                    status1_in => status_west4_in,
                    status2_in => status_south4_in,
                    status3_in => status_diagonal4_in,
                    status4_in => status_east4_in,
                    write_strobe_west_in => write_strobe_west4_in,
                    write_strobe_south_in => write_strobe_south4_in,
                    write_strobe_diagonal_in => write_strobe_diagonal4_in,
                    write_strobe_east_in => write_strobe_east4_in,
                    ws_s => ws_s4,
                    ws_d => ws_d4,
                    ws_e => ws_e4,
                    ws_w => ws_w4,
                    btn => btn
                    );
p4_RAM: entity p4_RAM
     port map(
     clk => clk190,
address=>address4,
     instruction=>instruction4
              );
-- PicoBlaze 5 declaration--
PicoBlaze 5: entity picoblaze
```

```
          port map(
                    input_1 => in_west5,
                    input_2 => in_south5,
                    input_3 => in_diagonal5,
                    input_4 => in_east5,
                    output_1 => out_west5,
                    output_2 => out_east5,
                    output_3 => out_diagonal5,
                    output_4 => out_south5,
                    clk => clk190,
                    address => address5,
                    instruction => instruction5,
                    status1 => status_west5_out,
                    status2 => status_south5_out,
                    status3 => status_diagonal5_out,
                    status4 => status_east5_out,
                    status1_in => status_west5_in,
                    status2_in => status_south5_in,
                    status3_in => status_diagonal5_in,
                    status4_in => status_east5_in,
                    write_strobe_west_in => write_strobe_west5_in,
                    write_strobe_south_in => write_strobe_south5_in,
                    write_strobe_diagonal_in => write_strobe_diagonal5_in,
                    write_strobe_east_in => write_strobe_east5_in,
                    ws_s => ws_s5,
                    ws_d => ws_d5,
                    ws_e => ws_e5,
                    ws_w => ws_w5,
                    btn => btn
                    );
p5_RAM: entity p5_RAM
     port map(
     clk => clk190,
address=>address5,
     instruction=>instruction5
              );

-- PicoBlaze 6 declaration--

PicoBlaze 6: entity picoblaze
          port map(
                     input_1 => in_west6,
                     input_2 => in_south6,
                    input_3 => in_diagonal6,
```

```
                    input_4 => in_east6,
                    output_1 => out_west6,
                    output_2 => out_east6,
                    output_3 => out_diagonal6,
                    output_4 => out_south6,
                    clk => clk190,
                    address => address6,
                    instruction => instruction6,
                    status1 => status_west6_out,
                    status2 => status_south6_out,
                    status3 => status_diagonal6_out,
                    status4 => status_east6_out,
                    status1_in => status_west6_in,
                    status2_in => status_south6_in,
                    status3_in => status_diagonal6_in,
                    status4_in => status_east6_in,
                    write_strobe_west_in => write_strobe_west6_in,
                    write_strobe_south_in => write_strobe_south6_in,
                    write_strobe_diagonal_in => write_strobe_diagonal6_in,
                    write_strobe_east_in => write_strobe_east6_in,
                    ws_s => ws_s6,
                    ws_d => ws_d6,
                    ws_e => ws_e6,
                    ws_w => ws_w6,
                    btn => btn
                    );
p6_RAM: entity p6_RAM
    port map(
    clk => clk190,
address=>address6,
    instruction=>instruction6
            );
-- PicoBlaze 7 declaration--
PicoBlaze 7: entity picoblaze
        port map(
                    input_1 => in_west7,
                     input_2 => in_south7,
                    input_3 => in_diagonal7,
                    input_4 => in_east7,
                    output_1 => out_west7,
                    output_2 => out_east7,
                    output_3 => out_diagonal7,
                    output_4 => out_south7,
                    clk => clk190,
```

```
                    address => address7,
                    instruction => instruction7,
                    status1 => status_west7_out,
                    status2 => status_south7_out,
                    status3 => status_diagonal7_out,
                    status4 => status_east7_out,
                    status1_in => status_west7_in,
                    status2_in => status_south7_in,
                    status3_in => status_diagonal7_in,
                    status4_in => status_east7_in,
                    write_strobe_west_in => write_strobe_west7_in,
                    write_strobe_south_in => write_strobe_south7_in,
                    write_strobe_diagonal_in => write_strobe_diagonal7_in,
                    write_strobe_east_in => write_strobe_east7_in,
                    ws_s => ws_s7,
                    ws_d => ws_d7,
                    ws_e => ws_e7,
                    ws_w => ws_w7,
                    btn => btn
                    );
p7_RAM: entity p7_RAM
    port map(
    clk => clk190,
address=>address7,
    instruction=>instruction7
                          );
-- PicoBlaze 8 declaration--
PicoBlaze 8: entity picoblaze
        port map(
                    input_1 => in_west8,
                    input_2 => in_south8,
                    input_3 => in_diagonal8,
                    input_4 => in_east8,
                    output_1 => out_west8,
                    output_2 => out_east8,
                    output_3 => out_diagonal8,
                    output_4 => out_south8,

                    clk => clk190,
                    address => address8,
                    instruction => instruction8,
                    status1 => status_west8_out,
                    status2 => status_south8_out,
                    status3 => status_diagonal8_out,
```

```
                    status4 => status_east8_out,
                    status1_in => status_west8_in,
                    status2_in => status_south8_in,
                    status3_in => status_diagonal8_in,
                    status4_in => status_east8_in,
                    write_strobe_west_in => write_strobe_west8_in,
                    write_strobe_south_in => write_strobe_south8_in,
                    write_strobe_diagonal_in => write_strobe_diagonal8_in,
                    write_strobe_east_in => write_strobe_east8_in,
                    ws_s => ws_s8,
                    ws_d => ws_d8,
                    ws_e => ws_e8,
                    ws_w => ws_w8,
                    btn => btn
                    );
p8_RAM: entity p8_RAM
    port map(
                    clk => clk190,
                    address=>address8,
                     instruction=>instruction8
                                );
-- PicoBlaze 9 declaration--
PicoBlaze 9: entity picoblaze
        port map(
                      input_1 => in_west9,
                     input_2 => in_south9,
                    input_3 => in_diagonal9,
                    input_4 => in_east9,
                    output_1 => out_west9,
                    output_2 => out_east9,
                    output_3 => out_diagonal9,
                    output_4 => out_south9,
                    clk => clk190,
                    address => address9,
                    instruction => instruction9,
                    status1 => status_west9_out,
                    status2 => status_south9_out,
                    status3 => status_diagonal9_out,
                    status4 => status_east9_out,
                    status1_in => status_west9_in,
                    status2_in => status_south9_in,
                    status3_in => status_diagonal9_in,
                    status4_in => status_east9_in,
                    write_strobe_west_in => write_strobe_west9_in,
```

```
                    write_strobe_south_in => write_strobe_south9_in,
                    write_strobe_diagonal_in => write_strobe_diagonal9_in,
                    write_strobe_east_in => write_strobe_east9_in,
                    ws_s => ws_s9,
                    ws_d => ws_d9,
                    ws_e => ws_e9,
                    ws_w => ws_w9,
                    btn => btn
                    );
p9_RAM: entity p9_RAM
    port map(
    clk => clk190,
address=>address9,
    instruction=>instruction9
        );
-- PicoBlaze 10 declaration--
PicoBlaze 10: entity picoblaze
        port map(
          input_1 => in_west10,
    input_2 => in_south10,
                    input_3 => in_diagonal10,
                    input_4 => in_east10,
                    output_1 => out_west10,
                    output_2 => out_east10,
                    output_3 => out_diagonal10,
                    output_4 => out_south10,
                    clk => clk190,
                    address => address10,
                    instruction => instruction10,
                    status1 => status_west10_out,
                    status2 => status_south10_out,
                    status3 => status_diagonal10_out,
                    status4 => status_east10_out,
                    status1_in => status_west10_in,
                    status2_in => status_south10_in,
                    status3_in => status_diagonal10_in,
                    status4_in => status_east10_in,
                    write_strobe_west_in => write_strobe_west10_in,
                    write_strobe_south_in => write_strobe_south10_in,
                    write_strobe_diagonal_in => write_strobe_diagonal10_in,
                    write_strobe_east_in => write_strobe_east10_in,
                    ws_s => ws_s10,
                    ws_d => ws_d10,
                    ws_e => ws_e10,
```

```
                    ws_w => ws_w10,
                    btn => btn
                    );
p10_RAM: entity p10_RAM
      port map(
      clk => clk190,
address=>address10,
      instruction=>instruction10
                );
clkdiv: entity clkdiv
            port map(
            mclk =>clk,
            clr =>clr,
            clk190 =>clk190,
            clk48 =>clk48
  );
--Picoblazes strobes are sent other picoblazes who are needed
-- below are write strobes that are required when the FIFO need the data from neighbour
write_strobe_south1_in <= ws_sU0;
write_strobe_south2_in <= ws_s1;
write_strobe_south3_in <= ws_sU1;
write_strobe_south4_in <= ws_s2;
write_strobe_south5_in <= ws_s3;
write_strobe_south6_in <= ws_sU2;
write_strobe_south7_in <= ws_s4;
write_strobe_south8_in <= ws_s5;
write_strobe_south9_in <= ws_s6;
write_strobe_south10_in <= ws_sU3;
--diagonal strobes
write_strobe_diagonal1_in <= ws_sU0;
write_strobe_diagonal2_in <= ws_s1;
write_strobe_diagonal3_in <= ws_d1;
write_strobe_diagonal4_in <= ws_s2;
write_strobe_diagonal5_in <= ws_d2;
write_strobe_diagonal6_in <= ws_d3;
write_strobe_diagonal7_in <= ws_s4;
write_strobe_diagonal8_in <= ws_d4;
write_strobe_diagonal9_in <= ws_d5;
write_strobe_diagonal10_in <= ws_d6;
--east strobes

write_strobe_east1_in <= ws_sU0;
write_strobe_east2_in <= ws_s1;
write_strobe_east3_in <= ws_e2;
```

```
write_strobe_east4_in <= ws_s2;
write_strobe_east5_in <= ws_e4;
write_strobe_east6_in <= ws_e5;
write_strobe_east7_in <= ws_s4;
write_strobe_east8_in <= ws_e7;
write_strobe_east9_in <= ws_e8;
write_strobe_east10_in <= ws_e9;
--connection between the picoblazes for data transfer
in_south1 <= out_southU0;    -- U[0]
in_south2 <= out_south1;
in_south3 <= out_southU1;    -- U[1]
in_south4 <= out_south2;
in_south5 <= out_south3;
in_south6 <= out_southU2;    -- U[2]
in_south7 <= out_south4;
in_south8 <= out_south5;
in_south9 <= out_south6;
in_south10 <= out_southU3; -- U[3]

in_east1 <= "00000010";                --A[3]
in_east2 <= "00000011";                --A[2]
in_east3 <= out_east2;
in_east4 <= "00000001";                --A[1]
in_east5 <= out_east4;
in_east6 <= out_east5;
in_east7 <= "00000010";                --A[0]
in_east8 <= out_east7;
in_east9 <= out_east8;
in_east10 <= out_east9;

in_diagonal1 <= "00000000";
in_diagonal2 <= "00000000";
in_diagonal3 <= out_diagonal1;
in_diagonal4 <= "00000000";
in_diagonal5 <= out_diagonal2;
in_diagonal6 <= out_diagonal3;
in_diagonal7 <= "00000000";
in_diagonal8 <= out_diagonal4;
in_diagonal9 <= out_diagonal5;
in_diagonal10 <= out_diagonal6;
--Outputs are seen from LEDS
--NOTE: watch for one output at once as we dont have more than 8 LEDs

--led <= out_diagonal10;        -- Y[3] series
```

```
led <= out_diagonal9; -- Y[2] series
--led <= out_diagonal8;         -- Y[1] series
--led <= out_diagonal7;         -- Y[0] series
--End of program
end arch;
```

C.3 Four FIFO Wrapper Design Code in VHDL

```
;--------------------------------------------------------------------------------------------------------
-- Thesis report by Venkata Mandala
-- Submitted: July 8th 2011
--picoBlaze.VHD

--DESCRIPTION: below vhdl code is the code acting as a wrapper
-- The wrapper is used as an entity in the main file for multiple times
-- 4 FIFOs are created in Wrapper designed for West, South, Diagonal, East directions
-- Wrapper takes different Signals in/out
-- For Example Signals like Write-Strobes, Data, FIFO status of neighbours taken

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

--Wrapper IN/OUT entity declarations

entity picoblaze is
  port(
    input_1: in std_logic_vector(7 downto 0);
    input_2: in std_logic_vector(7 downto 0);
input_3: in std_logic_vector(7 downto 0);
input_4: in std_logic_vector(7 downto 0);
output_1: out std_logic_vector(7 downto 0);
output_2: out std_logic_vector(7 downto 0);
output_3: out std_logic_vector(7 downto 0);
output_4: out std_logic_vector(7 downto 0);
clk : in std_logic;
address : inout std_logic_vector(9 downto 0);
   instruction : inout std_logic_vector(17 downto 0);
status1, status2, status3, status4: out std_logic_vector(7 downto 0);
status1_in, status2_in, status3_in, status4_in: in std_logic_vector(7 downto 0);
write_strobe_west_in, write_strobe_south_in, write_strobe_diagonal_in,
write_strobe_east_in : in std_logic;
ws_s, ws_e, ws_d, ws_w: out std_logic;
btn : in std_logic
```

```
);
end picoblaze;

--Architeture declaration is given below
architecture arch of picoblaze is

--Signals are given below

-- Below are signals in, out respectively from Picoblaze
   signal in_port, out_port: std_logic_vector(7 downto 0);
   signal port_id: std_logic_vector(7 downto 0);
signal read_strobe, write_strobe: std_logic;
signal interrupt, interrupt_ack: std_logic;

--Below are the signals for 4 FIFO's used--
signal fifo_out1, fifo_out2, fifo_out3, fifo_out4: std_logic_vector(7 downto 0);
signal rv: std_logic_vector(3 downto 0);
         signal rv1, rv2, rv3, rv4: std_logic;
         signal empty1, full1 : std_logic;
         signal empty2, full2 : std_logic;
         signal empty3, full3 : std_logic;
         signal empty4, full4 : std_logic;

--Process Signals
         signal en_d: std_logic_vector(3 downto 0);
         signal en_d1: std_logic_vector(11 downto 0);

begin
---Unused  inputs on processor
         interrupt <= '0';

--KCPSM3 is declared below which is an entity taken from KCPSM3.VHD file
kcpsm3: entity kcpsm3
     port map(
       clk=>clk, reset=>btn,
       address=>address, instruction=>instruction,
       port_id=>port_id, write_strobe=>write_strobe,
       out_port=>out_port, read_strobe=>read_strobe,
 in_port=>in_port, interrupt=>interrupt,
       interrupt_ack=>interrupt_ack
                              );

--write strobes of each side (south, diagonal, west east) are taken and send out
respectively
```

Appendix: C (Continued)

```
process(clk)
        begin
        if port_id(1 downto 0) = "00" then ws_w <= write_strobe; end if ;
        if port_id(1 downto 0) = "01" then ws_e <= write_strobe; end if ;
        if port_id(1 downto 0) = "10" then ws_d <= write_strobe; end if ;
        if port_id(1 downto 0) = "11" then ws_s <= write_strobe; end if ;
end process;

-- FIFO 1 is declared below which is for the west side
fifo_1: entity fifo
    port map(
                        clk=>clk, reset=>btn,
                        rd=>rv(0), wr=>write_strobe_west_in,
                        w_data=>input_1, r_data=>fifo_out1,
                        empty=>empty1, full=>full1
                          );
-- FIFO 2 is declared below which is for the south side

fifo_2: entity fifo
    port map(
                        clk=>clk, reset=>btn,
                        rd=>rv(1), wr=>write_strobe_south_in,
                        w_data=>input_2, r_data=>fifo_out2,
                        empty=>empty2, full=>full2
                          );
-- FIFO 3 is declared below which is for the diagonal side

fifo_3: entity fifo
    port map(
                        clk=>clk, reset=>btn,
                        rd=>rv(2), wr=>write_strobe_diagonal_in,
                        w_data=>input_3, r_data=>fifo_out3,
                        empty=>empty3, full=>full3
                          );
-- FIFO 4 is declared below which is for the east side

fifo_4: entity fifo
    port map(

                        clk=>clk, reset=>btn,
                        rd=>rv(3), wr=>write_strobe_east_in,
                        w_data=>input_4, r_data=>fifo_out4,
                        empty=>empty4, full=>full4
                          );
```

Appendix: C (Continued)

```
--FIFO's Status(Empty and Full) made as a 8-BIT by concatinating zeros for MSB's;
	status1 <= "000000" & full1 & empty1;
	status2 <= "000000" & full2 & empty2;
	status3 <= "000000" & full3 & empty3;
	status4 <= "000000" & full4 & empty4;

-----------Read Strobe--------------------
--
-- Input to the PicoBlaze is given with FIFO data, FIFO status and status from neighbours
with port_id(3 downto 0) select
	in_port <=	fifo_out1 when "0000",
					fifo_out2 when "0001",
					fifo_out3 when "0010",
					fifo_out4 when "0011",

					"000000" & full1 & empty1 when "0100",
					"000000" & full2 & empty2 when "0101",
					"000000" & full3 & empty3 when "0110",
					"000000" & full4 & empty4 when "0111",

					status1_in when "1000",
					status2_in when "1001",
					status3_in when "1010",
					status4_in when "1011",
					"00000000" when others;
--
-- "rv" enables when the data is coming into the FIFO
-- Below code does the bit high where the particular FIFO is to be write enabled
process (read_strobe, port_id)

	begin
		if read_strobe = '0' then
			rv <= "0000";
		else
			case port_id(3 downto 0) is

			when "0000" =>
				rv <= "0001";
			when "0001" =>
				rv <= "0010";
			when "0010" =>
				rv <= "0100";
			when "0011" =>
				rv <= "1000";
```

```
                    when others =>
                            rv <= "0000";

            end case;
        end if;
end process;

--- Write Strobe to different directions given below-
process (clk)
  begin
      if en_d(0)='1' then output_1 <= out_port; end if;
      if en_d(1)='1' then output_2 <= out_port; end if;
      if en_d(2)='1' then output_3 <= out_port; end if;
      if en_d(3)='1' then output_4 <= out_port; end if;
  end process;
process(port_id,write_strobe)
  begin
    en_d <= (others=>'0');
    if write_strobe='1' then
      case port_id(1 downto 0) is
        when "00" => en_d <="0001";
        when "01" => en_d <="0010";
        when "10" => en_d <="0100";
        when others => en_d <="1000";
      end case;
    end if;
  end process;
---------------------
--END of Wrapper program
end arch;
----------------------------------------------------------------------------------------------------
```

C.4 PicoBlaze 1 to PicoBlaze 10 Assembly Code for FIR Filter Design

```
;------------------------------------------------------------------------------------------
;- (PICOBLAZE 1 - PICOBLAZE 10).PSM file for FIR FILTER design
;------------------------------------
; Thesis report PSM file
; Submitted by Venkata Mandala
; July 8th 2011
; Below Code is a Picoblaze Assembly code (for PicoBalzes 1 to 10)
;------------------------------------
;------------------------------------
; DESCRIPTION:
;1. data from i_sou,i_dia and i_eas ports taken into in_s, in_d, in_e reg's
```

```
;2. in_s, in_d data is again loaded in s3, s4 reg's and multiplied
;3. result is stored in s6
;4. s6 is added with in_d(diagonal data) and stored in in_d
;5. in_d is outputed (sent) to o_dia (diagonal output)
;6. in_s is outputed (sent) to o_sou (south output)
;7. in_e is outputed (sent) to o_eas
;NOTE: The below code is used for PicoBlaze 1 to PicoBlaze 10
;-----------------------------------
;DETAILS
; First the FIFO status is checked
; If the FIFO is empty stay in loop else proceed next step
; Extract data from input ports
; do multiplication for south and east sides
; do addition for diagonal input and muliplied value
; sent the result obtained after addition above to diagonal outport
; sent the east and south inputs to east and south outputs resp'ly
;-----------------------------------
; registers
namereg s0, data
namereg s1, i
namereg s2, count
namereg sa, flag_s
namereg sb, flag_d
namereg sc, flag_e
namereg sd, in_s
namereg se, in_d
namereg sf, in_e
;-----------input port definitions---------------------
constant f_sou, 05
constant f_dia, 06
constant f_eas, 07
constant i_sou, 01
constant i_dia, 02
constant i_eas, 03

;-----------------------------------------------------
;------------output port definitions---------------------
constant o_sou, 03
constant o_dia, 02
constant o_eas, 01
constant o_wes, 00
;-----------------------------------------------------
main:
check_s:                ; Check the FIFO status of South
```

```
input flag_s, f_sou      ; Input the Flag status of South
and flag_s, 01
compare flag_s, 01       ; Compare with Empty Flag
jump z, check_s                    ; If empty be in loop "check_s"
check_e:                 ; check the FIFO status of East
input flag_e, f_eas      ; Input the Flag status of East
and flag_e, 01
compare flag_e, 01       ; Compare with Empty Flag
jump z, check_e                    ; If empty be in loop "check_e"
check_d:                 ; check the FIFO status of Diagonal
input flag_d, f_dia      ; Input the Flag status of Diagonal
and flag_d, 01
compare flag_d, 01       ; Compare with Empty Flag
jump z, check_d                    ; If empty be in loop "check_d"
input in_s, i_sou        ; Input data from South
input in_e, i_eas        ; Input data from east
input in_d, i_dia        ; Input data from diagonal
load s3, in_s            ; Load s3 with south value
load s4, in_e            ; Load s4 with east value
;--Below does the Multiplication of s3 and s4
;-- Product is stored in "s6" register
;NOTE: the product is 8-bit value, make sure not to exceed the value
calculate:
  load s6, 00
  load s5, 00             ;clear s5
  load i, 08              ;initialize loop index
mult_loop:
  sr0  s4                 ;shift lsb to carry
  jump nc, shift_prod     ;lsb is 0
  add s5, s3              ;lsb is 1
shift_prod:
  sra s5                  ;shift upper byte right,
                        ;carry to MSB, LSB to carry
  sra s6                  ;shift lower byte right,
                        ;lsb of s5 to MSB of s6
sub i, 01               ;dec loop index
  jump nz, mult_loop      ;repeat until i=0


; addition of muliplied value to the diagonal input obtained from diagonal
add in_d, s6
; Output all the ports to corresponding directions
output in_s, o_sou        ; sending the input of south to output of south
output in_e, o_eas        ; sending the input of east to output of east
```

```
output in_d, o_dia        ; diagonal = previous diagonal + (south * east)
load s3, 00               ; clear s3 register
load s4, 00               ; clear s4 register
jump main                 ; Jump to start of program and be in loop
```

C.5 PicoBlazeU0 to PicoBlazeU3 Assembly Code for FIR Filter Design

```
;-----------------------------------------------------------------------------------------------------
; PicoBlaze U0, PicoBlaze U1, PicoBalze U2, Picoblaze U3 PSM file codes
;-----------------------------------------------------------------------------------------------------
; Thesis report submitted by Venkata Mandala
; Date: July 8th 2011
; NOTE: can change input values {U1, U2, U3}for U1_RAM, U2_RAM, U3_RAM
; Description: This file generates U[0] Input values and outputs to south ports.
namereg s2, in_s
;------------output port definitions---------------------
constant o_sou, 03
;-------------------
; Main program starts here
load in_s, 01           ;Load series of U0 values
output in_s, o_sou      ; output to south port
load in_s, 02
output in_s, o_sou
load in_s, 03
output in_s, o_sou
load in_s, 04
output in_s, o_sou
load in_s, 03
output in_s, o_sou
forever:          ; be in loop after sending U0's
jump forever
;-----------------
```

Appendix D : *Spartan 3E* and *Virtex 5* Statistics

D.1 Logic Cells LEGEND Color on *Virtex 5* and *Spartan 3E* FPGA

picoblaze1 (picoblaze_NO9_picoblaze1)
picoblaze2 (picoblaze_NO8_picoblaze2)
picoblaze3 (picoblaze_NO7_picoblaze3)
picoblaze4 (picoblaze_NO6_picoblaze4)
picoblaze5 (picoblaze_NO5_picoblaze5)
picoblaze6 (picoblaze_NO4_picoblaze6)
picoblaze7 (picoblaze_NO3_picoblaze7)
picoblaze8 (picoblaze_NO2_picoblaze8)
picoblaze9 (picoblaze_NO1_picoblaze9)
picoblaze10 (picoblaze_picoblaze10)
picoblazeU0 (picoblaze)
picoblazeU1 (picoblaze_NO12_picoblazeU1)
picoblazeU2 (picoblaze_NO11_picoblazeU2)
picoblazeU3 (picoblaze_NO10_picoblazeU3)

Figure D-1: Legend colors for 14 PicoBlaze cores on FPGA

D.2 Occupation of Logic Cell Resources on *Virtex 5* FPGA

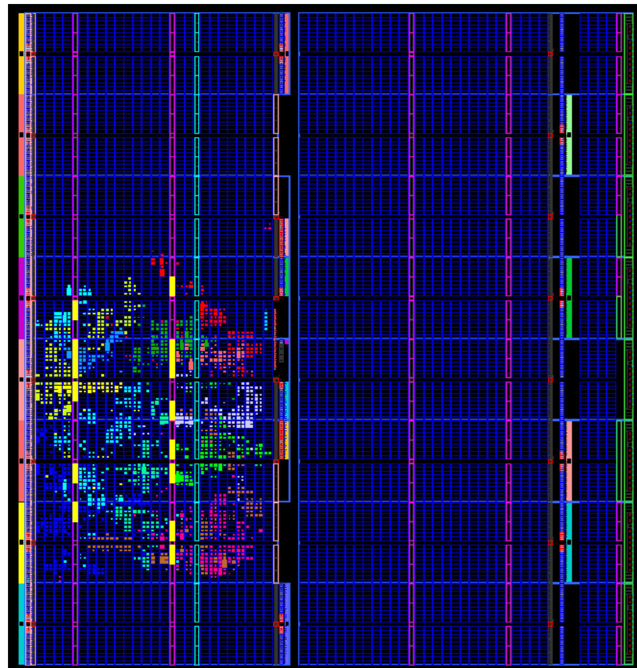


Figure D-2: Occupation of logic cells on *Virtex 5* FPGA

Appendix: D (Continued)

D.3 Occupation of Logic Cell Resources on *Spartan 3E* FPGA

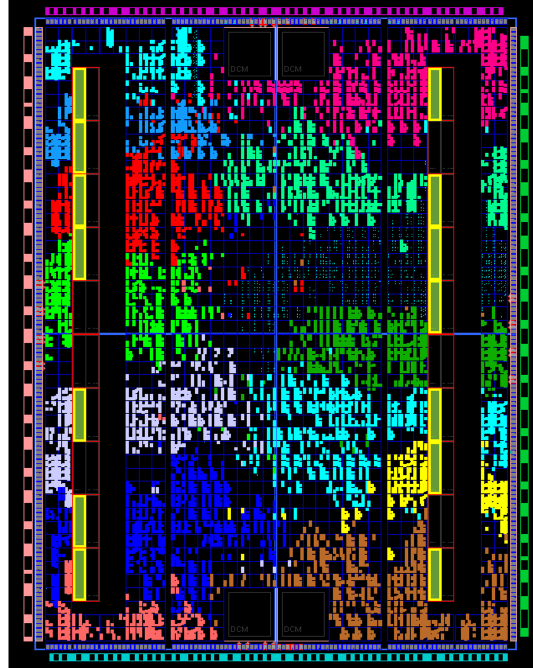


Figure D-3: Occupation of logic cells on *Spartan 3E* FPGA

D.4 Table Showing General Comparison of *Virtex 5* and *Spartan 3E* FPGAs

Table D-1: Comparison of *Spartan 3E* and *Virtex 5* FPGA

	Spartan-3 FPGA XC3S500E	Virtex-5 LXT FPGA XC5VLX110T
Slices	4,656	17,280
Logic Cells	10,476	110,592
CLB Flip-Flops	9,312	69,120
Maximum Distributed RAM (Kbits)	73	1,120
Total Block RAM (Kbits)	360	5,328
Block RAM as 36Kbits each	-	148
Block RAM as 18Kbits each	20	-
Maximum Single-Ended I/Os	232	680

Appendix E: Arbitration Schemes

E.1 Different Arbitration Schemes: (A Section from Chapter4)

In the available arbitration schemes, consider an arbitration technique used for design of Symmetric Single-Chip Multiprocessor (SSCMP). The arbiter has two different processes; scanning unit and arbitral process. The latter is executed by Arbitral Process Controlling Unit (APCU) and Interfacing Switching Unit (ISU) [30].

APCU is the most important part, and actually is a Finite State Machine (FSM), with three states which are IDLE, FETCH and BUSY. When the FSM detect that the FIFO is not empty, it shifts its state from IDLE to FETCH. In the state of FETCH, FSM fetches the address data of the processors from the FIFO, then it transfer to the next state, BUSY. The state of BUSY takes charge of waiting for the processor, which is accessing the shared data memory, to finish accessing operation.

ISU controls the connection of shared memory interface, in response to the output variable of ASCU (fetch data). ISU must cooperate with APCU [30].

Bus based arbiter is the other scheme available. A multiple bus multiprocessor system with N processors, M memory modules, and B buses are considered in the system. Four important bus control signals are present in a bus arbiter of multiple bus multiprocessor system. They are, bus request signal (RequestR_i), bus grant signal (GrantG_i), bus selection line (i), and memory-module enable signal. The bus selection line (i) is used to send the identification number of the grant bus to a requesting processor, where i =$log_2[B]$ and B is the total number of buses in the multiprocessor system. The memory-module enable line (E_n) is used to indicate the requested destination memory of the needed data from to the requesting processor, where E_n line is decoded for the address lines [25].

The arbitration algorithm followed by the bus arbiter is in executed by a state machine, which has four states: W (wait), G (grant) state, I (idle) state, and E (exclude) state. In wait state, the processor requests are suspended and queued in request queue until bus is released. If the state enters a grant state as a response is delivered from arbiter

to the requesting processor of this system. The idle state indicates that the processor is free. In exclude state, the bus is used to transfer data processor from/to other processors or memory modules [29].

The algorithm follows the state transition of first request first grant (FRFG) priority policy. StateW_j,G_j,I_j and E_j represent the wait state, grant state, and exclude state of processor j, respectively. If a request (R) is issued, if the destination is idle then the arbiter receives this request and responses a grant to the requesting processor; else this request is entered into wait state (W) until a free bus and the detonation are released, thus the wait-request can entry into grant state, furthermore entry exclude state [29].

Appendix F: Delay Statistics for *Spartan 3E* and *Virtex 5*

F.1 Place and Route Delay Statistics for *Spartan 3E* and *Virtex 5* FPGAs

Timing Results - Report Timing - results_1 (10 paths)

Name	Type	Slack	From	To	Total Delay	Logic Delay	Net %	Stages
Unconstrained Paths (10)								
Path 1	Setup	∞	clkdiv/q_20/C	picoblazeU3/kcpsm3/pc_loop[9].register_bit/D	16.419	12.549	23.6	18
Path 2	Setup	∞	clkdiv/q_20/C	picoblaze4/kcpsm3/pc_loop[9].register_bit/D	16.369	12.549	23.3	18
Path 3	Setup	∞	clkdiv/q_20/C	picoblaze10/kcpsm3/zero_flag_flop/D	16.217	11.357	30.0	17
Path 4	Setup	∞	clkdiv/q_20/C	picoblaze9/fifo_3/array_reg_0_0/CE	16.202	7.782	52.0	11
Path 5	Setup	∞	clkdiv/q_20/C	picoblaze9/fifo_3/array_reg_0_1/CE	16.202	7.782	52.0	11
Path 6	Setup	∞	clkdiv/q_20/C	picoblaze9/fifo_3/array_reg_0_6/CE	16.202	7.782	52.0	11
Path 7	Setup	∞	clkdiv/q_20/C	picoblaze9/fifo_3/array_reg_0_7/CE	16.202	7.782	52.0	11
Path 8	Setup	∞	clkdiv/q_20/C	picoblaze9/fifo_3/array_reg_1_4/CE	16.202	7.782	52.0	11
Path 9	Setup	∞	clkdiv/q_20/C	picoblaze9/fifo_3/array_reg_1_5/CE	16.202	7.782	52.0	11
Path 10	Setup	∞	clkdiv/q_20/C	picoblaze9/fifo_3/array_reg_4_4/CE	16.202	7.782	52.0	11

Figure F-1: Delay for *Spartan 3E* with 14 PicoBlazes (FIFO) setup

Timing Results - Report Timing - results_1 (10 paths)

Name	Type	Slack	From	To	Total Delay	Logic Delay	Net %	Stages
Unconstrained Paths (10)								
Path 10	Setup	∞	clkdiv/q_20/C	picoblaze6/kcpsm3/zero_flag_flop/D	8.195	3.219	60.7	16
Path 9	Setup	∞	clkdiv/q_20/C	picoblaze2/kcpsm3/zero_flag_flop/D	8.223	3.334	59.5	16
Path 8	Setup	∞	clkdiv/q_20/C	picoblaze4/kcpsm3/zero_flag_flop/D	8.448	3.334	60.5	16
Path 7	Setup	∞	clkdiv/q_20/C	picoblaze5/kcpsm3/zero_flag_flop/D	8.468	3.334	60.6	16
Path 6	Setup	∞	clkdiv/q_20/C	picoblaze7/kcpsm3/zero_flag_flop/D	8.551	3.330	61.1	16
Path 5	Setup	∞	clkdiv/q_20/C	picoblaze8/kcpsm3/zero_flag_flop/D	8.639	3.330	61.5	16
Path 4	Setup	∞	clkdiv/q_20/C	picoblaze3/kcpsm3/zero_flag_flop/D	8.650	3.219	62.8	16
Path 3	Setup	∞	clkdiv/q_20/C	picoblaze10/kcpsm3/zero_flag_flop/D	8.677	3.334	61.6	16
Path 2	Setup	∞	clkdiv/q_20/C	picoblaze9/kcpsm3/zero_flag_flop/D	8.702	3.334	61.7	16
Path 1	Setup	∞	clkdiv/q_20/C	picoblaze1/kcpsm3/zero_flag_flop/D	8.707	3.334	61.7	16

Figure F-2: Delay for *Virtex 5* with 14 PicoBlazes (FIFO) setup

Timing Results - Report Timing - results_1 (10 paths)

Name	Type	Slack	From	To	Total Delay	Logic Delay	Net %	Stages
Unconstrained Paths (10)								
Path 1	Setup	∞	p1_RAM/ram_1024_x_18/CLK	picoblaze1/pc_loop[9].register_bit/D	13.114	11.974	8.7	16
Path 2	Setup	∞	p3_RAM/ram_1024_x_18/CLK	picoblaze3/pc_loop[9].register_bit/D	13.114	11.974	8.7	16
Path 3	Setup	∞	p4_RAM/ram_1024_x_18/CLK	picoblaze4/pc_loop[9].register_bit/D	13.114	11.974	8.7	16
Path 4	Setup	∞	p2_RAM/ram_1024_x_18/CLK	picoblaze2/pc_loop[9].register_bit/D	12.994	11.974	7.8	16
Path 5	Setup	∞	p4_RAM/ram_1024_x_18/CLK	picoblaze4/arith_loop[7].msb_arith.arith_carry_flop/D	12.836	11.016	14.2	15
Path 6	Setup	∞	p1_RAM/ram_1024_x_18/CLK	picoblaze1/arith_loop[7].msb_arith.arith_carry_flop/D	12.576	11.016	12.4	15
Path 7	Setup	∞	p2_RAM/ram_1024_x_18/CLK	picoblaze2/arith_loop[7].msb_arith.arith_carry_flop/D	12.576	11.016	12.4	15
Path 8	Setup	∞	p1_RAM/ram_1024_x_18/CLK	picoblaze1/pc_loop[8].register_bit/D	12.400	11.260	9.2	15
Path 9	Setup	∞	p3_RAM/ram_1024_x_18/CLK	picoblaze3/pc_loop[8].register_bit/D	12.400	11.260	9.2	15
Path 10	Setup	∞	p4_RAM/ram_1024_x_18/CLK	picoblaze4/pc_loop[8].register_bit/D	12.400	11.260	9.2	15

Figure F-3: Delay for *Spartan 3E* with 4 Picobalzes (shared memory) setup

Timing Results - Report Timing - results_1 (10 paths)

Name	Type	Slack	From	To	Total Delay	Logic Delay	Net %	Stages
Unconstrained Paths (10)								
Path 1	Setup	∞	p1_RAM/ram_1024_x_18/CLKA	picoblaze1/zero_flag_flop/D	4.873	2.380	51.2	10
Path 2	Setup	∞	p4_RAM/ram_1024_x_18/CLKA	picoblaze4/zero_flag_flop/D	4.763	2.296	51.8	10
Path 3	Setup	∞	p3_RAM/ram_1024_x_18/CLKA	picoblaze3/zero_flag_flop/D	4.708	2.300	51.1	10
Path 4	Setup	∞	p2_RAM/ram_1024_x_18/CLKA	picoblaze2/zero_flag_flop/D	4.671	2.380	49.1	10
Path 5	Setup	∞	p4_RAM/ram_1024_x_18/CLKA	picoblaze4/pc_loop[9].register_bit/D	4.633	2.466	46.8	12
Path 6	Setup	∞	p4_RAM/ram_1024_x_18/CLKA	picoblaze4/pc_loop[8].register_bit/D	4.587	2.420	47.2	11
Path 7	Setup	∞	p4_RAM/ram_1024_x_18/CLKA	picoblaze4/pc_loop[7].register_bit/D	4.581	2.513	45.1	14
Path 8	Setup	∞	p2_RAM/ram_1024_x_18/CLKA	picoblaze2/pc_loop[9].register_bit/D	4.557	2.469	45.8	16
Path 9	Setup	∞	p1_RAM/ram_1024_x_18/CLKA	picoblaze1/pc_loop[9].register_bit/D	4.556	2.469	45.8	16
Path 10	Setup	∞	p3_RAM/ram_1024_x_18/CLKA	picoblaze3/pc_loop[9].register_bit/D	4.541	2.469	45.6	16

Figure F-4: Delay for *Virtex 5* with 4 PicoBlazes (shared memory) setup

Appendix -F (Continued)

F.2 Synthesis (Pre-Routing) Report Statistics for *Spartan 3E* and *Virtex 5* FPGAs

```
Data Path: p5_RAM/ram_1024_x_18 to picoblaze5/kcpsm3/zero_flag_flop
                                Gate     Net
  Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
  ----------------------------------------  ------------
   RAMB16_S18:CLK->DO4   10   2.436   0.750  ram_1024_x_18 (instruction<4>)
   end scope: 'p5_RAM'
   begin scope: 'picoblaze5'
   begin scope: 'kcpsm3'
   RAM16X1D:DPRA0->DPO    1   0.612   0.387  reg_loop[0].register_bit (sy<0>)
   LUT3:I2->O            92   0.612   1.119  reg_loop[0].operand_select_mux (port_id<0>)
   end scope: 'kcpsm3'
   LUT3:I2->O             1   0.612   0.000  Mmux_in_port_8 (Mmux_in_port_8)
   MUXF5:I0->O            1   0.278   0.387  Mmux_in_port_6_f5 (Mmux_in_port_6_f5)
   LUT4:I2->O             1   0.612   0.426  port_id<3>1 (in_port<0>)
   begin scope: 'kcpsm3'
   LUT3:I1->O             1   0.612   0.000  alu_mux_loop[0].mux_lut (input_group<0>)
   MUXF5:I1->O            2   0.278   0.532  alu_mux_loop[0].shift_in_muxf5 (alu_result<0>)
   LUT4:I0->O             1   0.612   0.000  low_zero_lut (low_zero)
   MUXCY:S->O             1   0.404   0.000  low_zero_muxcy (low_zero_carry)
   MUXCY:CI->O            1   0.051   0.000  high_zero_cymux (high_zero_carry)
   MUXCY:CI->O            0   0.051   0.000  zero_cymux (zero_carry)
   XORCY:CI->O            1   0.699   0.000  zero_xor (zero_fast_route)
   FDRE:D                     0.268          zero_flag_flop
  ----------------------------------------
  Total                      11.739ns (8.138ns logic, 3.601ns route)
                                      (69.3% logic, 30.7% route)

      Minimum period: 11.739ns (Maximum Frequency: 85.189MHz)
      Minimum input arrival time before clock: 3.100ns
      Maximum output required time after clock: 4.114ns
      Maximum combinational path delay: No path found
```

Figure F-5: Synthesis delay report for *Spartan 3E* with 14 PicoBlaze setup

```
Data Path: p5_RAM/ram_1024_x_18 to picoblaze5/kcpsm3/zero_flag_flop
                                Gate     Net
  Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
  ----------------------------------------  ------------
   RAMB16:CLKA->DOA4     10   1.892   0.269  ram_1024_x_18 (instruction<4>)
   end scope: 'p5_RAM'
   begin scope: 'picoblaze5'
   begin scope: 'kcpsm3'
   RAM16X1D:DPRA0->DPO    1   0.086   0.361  reg_loop[1].register_bit (sy<1>)
   LUT3:I2->O            29   0.086   0.525  reg_loop[1].operand_select_mux (port_id<1>)
   end scope: 'kcpsm3'
   LUT6:I4->O             1   0.086   0.436  Mmux_in_port_7 (Mmux_in_port_7)
   LUT4:I2->O             1   0.086   0.436  port_id<3>1 (in_port<0>)
   begin scope: 'kcpsm3'
   LUT3:I1->O             1   0.086   0.000  alu_mux_loop[0].mux_lut (input_group<0>)
   MUXF5:I1->O            2   0.214   0.615  alu_mux_loop[0].shift_in_muxf5 (alu_result<0>)
   LUT4:I0->O             1   0.086   0.000  low_zero_lut (low_zero)
   MUXCY:S->O             1   0.305   0.000  low_zero_muxcy (low_zero_carry)
   MUXCY:CI->O            1   0.023   0.000  high_zero_cymux (high_zero_carry)
   MUXCY:CI->O            0   0.023   0.000  zero_cymux (zero_carry)
   XORCY:CI->O            1   0.300   0.000  zero_xor (zero_fast_route)
   FDRE:D                    -0.022          zero_flag_flop
  ----------------------------------------
  Total                       5.915ns (3.272ns logic, 2.643ns route)
                                      (55.3% logic, 44.7% route)

      Minimum period: 5.915ns (Maximum Frequency: 169.055MHz)
      Minimum input arrival time before clock: 1.547ns
      Maximum output required time after clock: 2.989ns
      Maximum combinational path delay: No path found
```

Figure F-6: Synthesis delay report for *Virtex 5* with 14 PicoBlazes setup

Appendix -F (Continued)

```
Data Path: p4_RAM/ram_1024_x_18 to picoblaze4/pc_loop[9].register_bit
                                Gate     Net
  Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
  ----------------------------------------  ------------
   RAMB16_S18:CLK->DO14   27   2.436   1.224  ram_1024_x_18 (instruction<14>)
   end scope: 'p4_RAM'
   begin scope: 'picoblaze4'
   LUT4:I0->O            1   0.612   0.387  move_group_lut (move_group)
   LUT3:I2->O           11   0.612   0.945  normal_count_lut (normal_count)
   LUT3:I0->O            1   0.612   0.000  pc_loop[0].value_select_mux (pc_value<0>)
   MUXCY:S->O            1   0.404   0.000  pc_loop[0].pc_lsb_carry.pc_value_muxcy (pc_value_carry<0>)
   MUXCY:CI->O           1   0.051   0.000  pc_loop[1].pc_mid_carry.pc_value_muxcy (pc_value_carry<1>)
   MUXCY:CI->O           1   0.051   0.000  pc_loop[2].pc_mid_carry.pc_value_muxcy (pc_value_carry<2>)
   MUXCY:CI->O           1   0.051   0.000  pc_loop[3].pc_mid_carry.pc_value_muxcy (pc_value_carry<3>)
   MUXCY:CI->O           1   0.051   0.000  pc_loop[4].pc_mid_carry.pc_value_muxcy (pc_value_carry<4>)
   MUXCY:CI->O           1   0.051   0.000  pc_loop[5].pc_mid_carry.pc_value_muxcy (pc_value_carry<5>)
   MUXCY:CI->O           1   0.051   0.000  pc_loop[6].pc_mid_carry.pc_value_muxcy (pc_value_carry<6>)
   MUXCY:CI->O           1   0.051   0.000  pc_loop[7].pc_mid_carry.pc_value_muxcy (pc_value_carry<7>)
   MUXCY:CI->O           0   0.051   0.000  pc_loop[8].pc_mid_carry.pc_value_muxcy (pc_value_carry<8>)
   XORCY:CI->O           1   0.699   0.000  pc_loop[9].pc_msb_carry.pc_value_xor (inc_pc_value<9>)
   FDRSE:D                   0.268          pc_loop[9].register_bit
  ----------------------------------------
  Total                      8.611ns (6.055ns logic, 2.556ns route)
                                     (70.3% logic, 29.7% route)

              Minimum period: 8.611ns (Maximum Frequency: 116.136MHz)
              Minimum input arrival time before clock: 2.544ns
              Maximum output required time after clock: 4.754ns
              Maximum combinational path delay: No path found
```

Figure F-7: Synthesis delay report for *Spartan 3E* with Shared memory setup

```
Data Path: p4_RAM/ram_1024_x_18 to picoblaze4/pc_loop[9].register_bit
                                Gate     Net
  Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
  ----------------------------------------  ------------
   RAMB16:CLKA->DOA14   27   1.892   0.700  ram_1024_x_18 (instruction<14>)
   end scope: 'p4_RAM'
   begin scope: 'picoblaze4'
   LUT4:I0->O            1   0.086   0.361  move_group_lut (move_group)
   LUT3:I2->O           11   0.086   0.588  normal_count_lut (normal_count)
   LUT3:I0->O            1   0.086   0.000  pc_loop[0].value_select_mux (pc_value<0>)
   MUXCY:S->O            1   0.305   0.000  pc_loop[0].pc_lsb_carry.pc_value_muxcy (pc_value_carry<0>)
   MUXCY:CI->O           1   0.023   0.000  pc_loop[1].pc_mid_carry.pc_value_muxcy (pc_value_carry<1>)
   MUXCY:CI->O           1   0.023   0.000  pc_loop[2].pc_mid_carry.pc_value_muxcy (pc_value_carry<2>)
   MUXCY:CI->O           1   0.023   0.000  pc_loop[3].pc_mid_carry.pc_value_muxcy (pc_value_carry<3>)
   MUXCY:CI->O           1   0.023   0.000  pc_loop[4].pc_mid_carry.pc_value_muxcy (pc_value_carry<4>)
   MUXCY:CI->O           1   0.023   0.000  pc_loop[5].pc_mid_carry.pc_value_muxcy (pc_value_carry<5>)
   MUXCY:CI->O           1   0.023   0.000  pc_loop[6].pc_mid_carry.pc_value_muxcy (pc_value_carry<6>)
   MUXCY:CI->O           1   0.023   0.000  pc_loop[7].pc_mid_carry.pc_value_muxcy (pc_value_carry<7>)
   MUXCY:CI->O           0   0.023   0.000  pc_loop[8].pc_mid_carry.pc_value_muxcy (pc_value_carry<8>)
   XORCY:CI->O           1   0.300   0.000  pc_loop[9].pc_msb_carry.pc_value_xor (inc_pc_value<9>)
   FDRSE:D                  -0.022          pc_loop[9].register_bit
  ----------------------------------------
  Total                      4.586ns (2.937ns logic, 1.649ns route)
                                     (64.0% logic, 36.0% route)

              Minimum period: 4.586ns (Maximum Frequency: 218.076MHz)
              Minimum input arrival time before clock: 1.424ns
              Maximum output required time after clock: 2.864ns
              Maximum combinational path delay: No path found
```

Figure F-8: Synthesis delay report for *Virtex 5* with 4 Shared memory setup