Library Functions in Reconfigurable Hardware for Matrix and Signal Processing Operations in MATLAB

Suresh Babu Periyacheri

Technical Report No. CPDC-TR-9905-008

May 1999

Center for Parallel and Distributed Computing

Department of Electrical & Computer Engineering

Northwestern University

Technological Institute

2145 Sheridan Road, Evanston IL-60208

Library Functions in Reconfigurable Hardware for Matrix and Signal Processing Operations in MATLAB

by

Suresh Babu Periyacheri

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate School of
Northwestern University, 1999

Evanston, Illinois

Abstract

Reconfigurable or adaptive computing is an emerging area of computer science, which promises super-computing performance from small, relatively low cost systems composed of re-programmable hardware such as field-programmable gate arrays (FPGA). Reconfigurable computing blurs the traditional boundaries between software and hardware by introducing the concept of hardware/software co-design where the underlying hardware to execute an application changes with the needs of the application. In effect, it provides the programmer with a custom processor for each of his/her applications. But large scale reconfigurable computing is yet to become a reality, mainly due to the limitations of available reconfigurable hardware and the lack of the tools to aid in the fast development of applications to run on them. The MATCH project at Northwestern University is an attempt to make this application development process easier by providing tools that can quickly map programs written in a high-level language (MATLAB) to code that can be executed simultaneously on a collection of commercial-off-the-shelf (COTS) components that include DSPs and embedded processors, apart from FPGAs. The test bed for such a system has been built and one of its components is the WILDCHILD multi-FPGA system from Annapolis Microsystems based on the SPLASH-2 custom computer built at Supercomputing Research Center in Bowie, MD. As part of the MATCH project, a compiler is being developed that would read in a high-level application written in MATLAB and generate code running on a host processor that would spawn of parallel tasks on the FPGAs, DSPs and the embedded processors. The current effort on the FPGA side is to develop library functions for matrix and signal processing operations in hardware on the FPGAs and to provide simple interfaces to them for the MATLAB compiler. This thesis outlines these efforts and describes the functions and their implementation on the WILDCHILD system.

Acknowledgements

I wish to thank my advisors Prof. Prithviraj Banerjee and Prof. Nagaraj Shenoy for the guidance and advice they provided me throughout this work. Without their support and encouragement this thesis would not have materialized. It has indeed been a pleasure working with them. I also wish to thank Prof. Alok Choudhary and Prof. Valerie Taylor for participating in the examination committee and Prof. Scott Hauck for his valuable comments during the project group meetings.

Finally, I wish to thank all my colleagues in the lab, especially those who are part of the MATCH project team, for providing inputs about different aspects of the project that were very valuable to the successful completion of this thesis.

To my parents and sisters

Contents

CHAPTER 1 INTRODUCTION **10**

1.1 THE MATCH PROJECT: AN OVERVIEW 11
1.2 THE COMPILER AND ITS LIBRARY INTERFACES 12
1.3 RELATED WORK 14
1.4 CONTRIBUTIONS 18
1.5 THESIS OUTLINE 18

CHAPTER 2 RECONFIGURABLE COMPUTING WITH THE WILDCHILD SYSTEM **19**

2.1 THE RECONFIGURABLE COMPUTING SYSTEM 19
2.2 THE WILDCHILD MULTI-FPGA SYSTEM 20
2.3 THE WILDCHILD HOST INTERFACE 23
2.4 FPGA FUNCTION CODING IN VHDL 26
2.5 THE WILDCHILD DESIGN CYCLE 28
2.6 SUMMARY 32

CHAPTER 3 SCALED MATRIX ADDER FUNCTION **33**

3.1 FUNCTION DESCRIPTION 33
3.2 DESIGN ALTERNATIVES 34
3.2.1 Alternative 1: Block Partitioning 35
3.2.2 Alternative 2: Matrix-wise Distribution 36
3.3 IMPLEMENTATION OF MATRIX ADDER ON WILDCHILD 36
3.3.1 Architecture of Matrix Adder on WILDCHILD 37
3.3.2 Reader Module 37
3.3.3 Adder Module 38
3.4 HOST INTERFACE TO MATRIX ADDER 40
3.5 RESULTS 44
3.6 SUMMARY 44

CHAPTER 4 FILTER FUNCTION **46**

4.1 FUNCTION DESCRIPTION 46
4.2 DESIGN ALTERNATIVES 47
4.2.1 Alternative 1: Direct Form I 47
4.2.2 Alternative 2: Direct Form II 48
4.2.3 Alternative 3: Cascade Form 49
4.3 IMPLEMENTATION OF FILTER ON WILDCHILD 49
4.3.1 Architecture of Filter on WILDCHILD 50
4.3.2 One-tap Filter Module 50
4.4 HOST INTERFACE TO FILTER 52
4.5 RESULTS 55
4.6 SUMMARY 56

CHAPTER 5 MATRIX MULTIPLIER FUNCTION **57**

5.1 FUNCTION DESCRIPTION 57
5.2 DESIGN ALTERNATIVES 58
5.2.1 Alternative 1: Block Partitioning 59
5.2.2 Alternative 2: Row-wise Partitioning 59
5.3 IMPLEMENTATION OF MATRIX MULTIPLIER ON WILDCHILD 62
5.3.1 Architecture of Matrix Multiplier on WILDCHILD 62

5.3.2 Master Module ... *62*
5.3.3 Slave Module ... *63*
5.4 HOST INTERFACE TO MATRIX MULTIPLIER ... 65
5.5 RESULTS ... 69
5.6 SUMMARY ... 69

CHAPTER 6 FAST FOURIER TRANSFORM FUNCTION ... 71

6.1 FUNCTION DESCRIPTION ... 71
6.1.1 The FFT algorithm: ... *72*
6.1.2 The Memory Bandwidth Constraint ... *73*
6.2 DESIGN ALTERNATIVES ... 74
6.2.2 Alternative 1: Design without On-chip RAM ... *77*
6.2.3 Alternative 2: Design with On-chip RAM ... *78*
6.3 IMPLEMENTATION OF FFT ENGINE ON WILDCHILD ... 79
6.3.1 Architecture of FFT engine on WILDCHILD ... *79*
6.3.2 Master Module ... *80*
6.3.3 RAM Module ... *81*
6.3.4 Butterfly Module ... *83*
6.4 HOST INTERFACE OF FFT ENGINE ... 85
6.5 RESULTS ... 89
6.6 SUMMARY ... 90

CHAPTER 7 CONCLUSION ... 91

BIBLIOGRAPHY

List of Figures

FIGURE 1.1 THE MATCH SYSTEM 11
FIGURE 1.2 OVERVIEW OF TASKS IN THE MATCH PROJECT 12
FIGURE 1.3 THE MATCH COMPILER 13
FIGURE 1.4 COMPILER-LIBRARY INTERFACING 14
FIGURE 2.1 VME-BASED RECONFIGURABLE COMPUTING SYSTEM 20
FIGURE 2.2 WILDCHILD SYSTEM ARCHITECTURE 20
FIGURE 2.3 BUS DRIVER CIRCUIT 21
FIGURE 2.4 CROSSBAR CONFIGURATION SPECIFICATION 22
FIGURE 2.5 A STATE MACHINE PROCESS IN VHDL 28
FIGURE 2.6 INTERFACING OF MODULE WITH VHDL MODEL OF PEX FPGA 29
FIGURE 2.7 APPLICATION DESIGN CYCLE 30
FIGURE 3.1 EXAMPLE OF SCALED VECTOR ADDITION 34
FIGURE 3.2 (A) MATLAB CODE FOR SCALED VECTOR ADDITION (B) CORRESPONDING C CODE 34
FIGURE 3.3 BLOCK PARTITIONING 35
FIGURE 3.4 ARCHITECTURE OF MATRIX ADDER FUNCTION 37
FIGURE 3.5 SCHEMATIC OF READER 38
FIGURE 3.6 SCHEMATIC OF ADDER 39
FIGURE 3.7 MATRIX ADDER HOST INTERFACE 41
FIGURE 3.8 FPGA CONFIGURATION LOADER FUNCTION FOR MATRIX ADDER 42
FIGURE 3.9 (PEX) MEMORY DATA ORGANIZATION 42
FIGURE 3.10 EVALUATOR FUNCTION FOR MATRIX ADDER 43
FIGURE 4.1 GENERAL FILTER EQUATION 46
FIGURE 4.2 (A) MATLAB CODE FOR FILTER (B) CORRESPONDING C CODE 47
FIGURE 4.3 DIRECT FORM I OF 4TH ORDER IIR FILTER 47
FIGURE 4.4 DIRECT FORM II OF 4TH ORDER IIR FILTER 48
FIGURE 4.5 CASCADE FORM OF 4TH ORDER IIR FILTER 49
FIGURE 4.6 ARCHITECTURE OF FILTER 50
FIGURE 4.7 SCHEMATIC OF FIRST ORDER FILTER 51
FIGURE 4.8 FILTER HOST INTERFACE 53
FIGURE 4.9 FPGA CONFIGURATION LOADER FUNCTION FOR FILTER 53
FIGURE 4.10 (PEX) FILTER MEMORY DATA ORGANIZATION 54
FIGURE 4.11 EVALUATOR FUNCTION FOR FILTER 55
FIGURE 5.1 (A) MATLAB CODE FOR MATRIX MULTIPLICATION (B) CORRESPONDING C CODE 58
FIGURE 5.2 BLOCK PARTITIONING OF RESULT MATRIX 59
FIGURE 5.3 STAR AND RING TOPOLOGIES FOR COMMUNICATION 60
FIGURE 5.4 ARCHITECTURE OF MATRIX MULTIPLIER 62
FIGURE 5.5 SCHEMATIC OF MASTER 63
FIGURE 5.6 SCHEMATIC OF SLAVE 64
FIGURE 5.7 MATRIX MULTIPLIER HOST INTERFACE 65
FIGURE 5.8 FPGA CONFIGURATION LOADER FUNCTION FOR MATRIX MULTIPLIER 66
FIGURE 5.9 PE0 (MASTER) AND PEX (SLAVE) MEMORY DATA ORGANIZATION 67
FIGURE 5.10 EVALUATOR FUNCTION FOR MATRIX MULTIPLIER 68
FIGURE 6.1 (A) BASIC DFT EQUATION 72
FIGURE 6.2 BUTTERFLY GRAPH FOR COMPUTING FFT (A) AND A RE-ORDERED GRAPH (B) 73
FIGURE 6.3 PIPELINING WITH DUAL SET OF REGISTERS 75
FIGURE 6.4 INTER-PROCESSOR COMMUNICATION PATTERN 75
FIGURE 6.5 DATA POINT TRANSMIT/RECEIVE MODE DETERMINATION 76
FIGURE 6.6 ADDRESS GENERATION USING 1-BIT BARREL-SHIFT 77
FIGURE 6.7 FFT ENGINE WITHOUT BUFFER RAM 77

Figure 6.8 FFT Engine with 32-bit Buffer RAM 78
Figure 6.9 Architecture of FFT Computer 80
Figure 6.10 Schematic of Master Module 81
Figure 6.11 Schematic of RAM Module 82
Figure 6.12 Schematic of 4-Clock Butterfly Module 83
Figure 6.13 FFT Host Interface 86
Figure 6.14 Crossbar Connections for Last Two Passes of FFT 86
Figure 6.15 FPGA Configuration Loader Function for FFT 87
Figure 6.16 PE0 (Master) and PEX (RAM/Butterfly) Memory Data Organization 87
Figure 6.17 Evaluator Function for FFT 89

List of Tables

TABLE 3.1 PERFORMANCE OF MATRIX ADDER FUNCTION ON WILDCHILD 44
TABLE 4.1 PERFORMANCE OF FILTER FUNCTION ON WILDCHILD 56
TABLE 5.1 PERFORMANCE OF MATRIX MULTIPLIER FUNCTION ON WILDCHILD 69
TABLE 6.1 PERFORMANCE OF FFT FUNCTION ON WILDCHILD 90

Chapter 1 Introduction

Reconfigurable or adaptive computing is an emerging area of computer science, which promises super-computing performance from small, relatively low cost systems composed of re-programmable hardware such as field-programmable gate arrays (FPGA). Reconfigurable computing blurs the traditional boundaries between software and hardware by introducing the concept of hardware/software co-design where the underlying hardware to execute an application changes with the needs of the application. In effect, it provides the programmer with a custom processor for each of his/her applications.

But large scale reconfigurable computing is yet to become a reality, mainly due to the limitations of available reconfigurable hardware and the lack of the tools to aid in the fast development of applications to run on them. The MATCH project [1] at Northwestern University is an attempt to make this application development process easier by providing tools that can quickly map programs written in a high-level language (MATLAB) to code that can be executed simultaneously on a collection of commercial-off-the-shelf (COTS) components that include DSPs and embedded processors, apart from FPGAs.

The test bed for such a system has been built and one of its components is the WILDCHILD multi-FPGA system [2] from Annapolis Microsystems based on the SPLASH-2 [3] custom computer built at Supercomputing Research Center in Bowie, MD. As part of the MATCH project, a compiler is being developed that would read in a high-level application written in MATLAB and generate code running on a host processor that would spawn of parallel tasks on the FPGAs, DSPs and the embedded processors. The current effort on the FPGA side is to develop library functions for matrix and signal processing operations in hardware on the FPGAs and to provide simple interfaces to them for the MATLAB compiler. This thesis outlines these efforts and describes the functions and their implementation on the WILDCHILD system.

1.1 The MATCH Project: An Overview

MATCH - A MATLAB Compilation Environment for Distributed Heterogeneous Adaptive Computing Systems - is an ongoing project at Northwestern University. The goal of the project is to demonstrate a heterogeneous computing environment where user programs written in MATLAB are compiled into multi-platform target code that can run simultaneously on a collection of different COTS hardware like DSPs, embedded processors, FPGAs and general purpose processors. The chief focus of this project is the development of a compiler, which can automatically extract parallelism and perform optimized mapping of user applications in MATLAB to the target set of hardware. This allows the programmer to focus on the adaptive computing application instead of the details of its implementation on the heterogeneous system. An outline of the adaptive computing environment is shown in Figure 1.1.

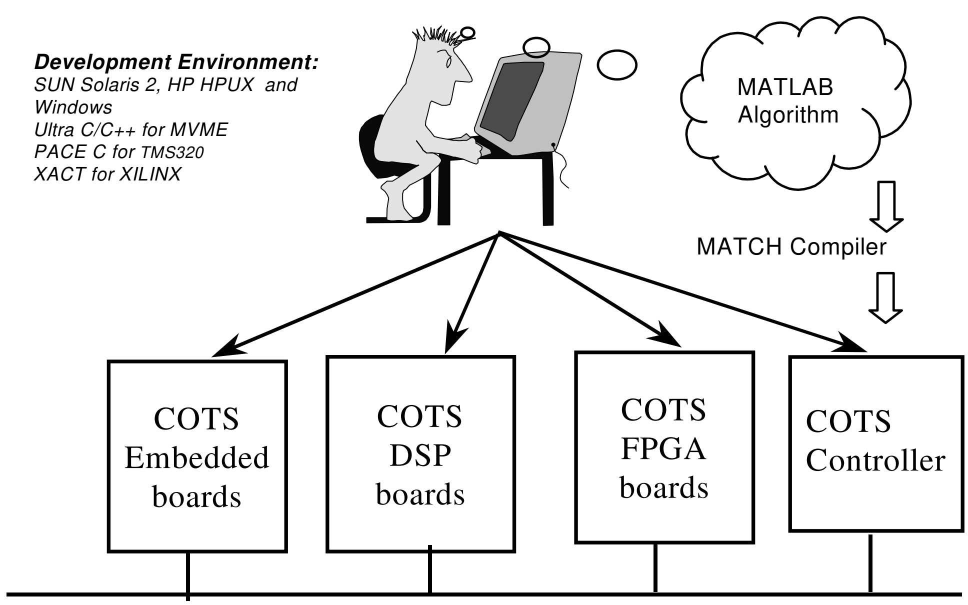


Figure 1.1 The MATCH System

Figure 1.2 shows the tasks to be accomplished in the MATCH project in a 3-year time frame. The first version of the MATCH compiler relies on a library-based approach where the compiler generates target code that is mostly a collection of different function calls to libraries that have been pre-developed for each component of the heterogeneous system. Thus there are matrix, signal processing and other common functions encountered in MATLAB programs implemented on the DSPs, FPGAs and the embedded board.

The ultimate goal of the project is to have the compiler automatically map program tasks to the target hardware component most suited for the task with little programmer intervention but the first version relies on user-specified guidelines in the form of directives in the MATLAB program to perform the mapping.

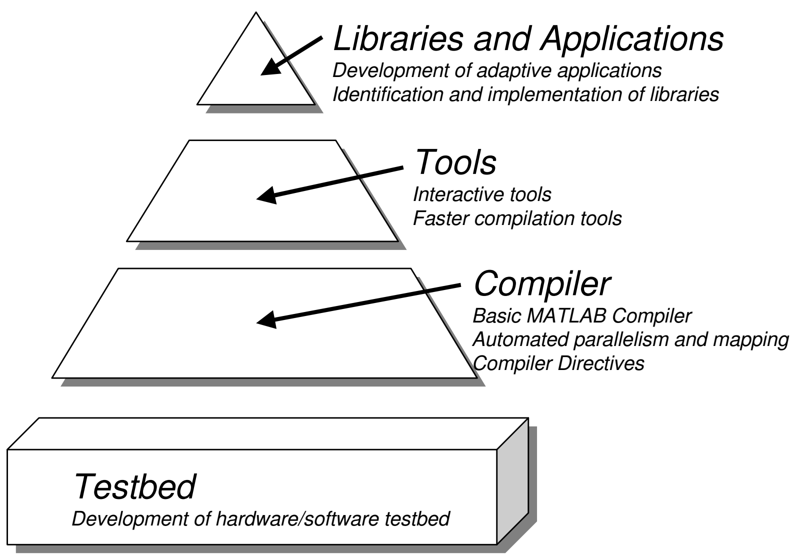


Figure 1.2 Overview of tasks in the MATCH Project

1.2 The Compiler and its Library Interfaces

As mentioned in the last section, the initial version of a MATLAB compiler for the MATCH system relies on a library-based approach to execute sections of the program in FGPA-based hardware. The libraries have a C language interface that consists of a single function call with arguments of a common pre-defined data type used by the compiler to ship data and results between different functions. In the initial version of these library interfaces, the FPGA configuration loading and evaluating functions occur in sequence within a single interfacing function. This is not necessary because the FPGA configuration required to implement a function is independent of the data and hence can be loaded much in advance of the actual computation. Potentially, it is useful in hiding the long loading times by executing the loads in parallel with other operations. This will be included in the next version of the compiler and library interfaces.

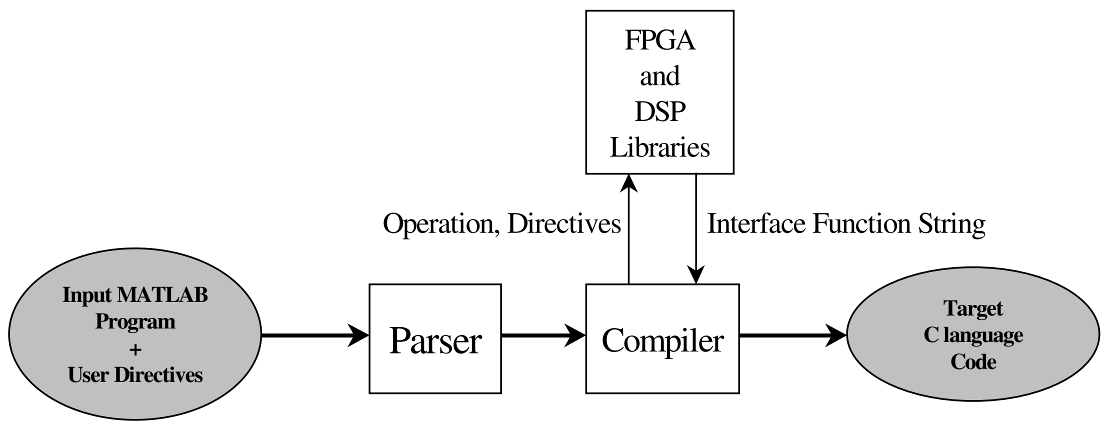
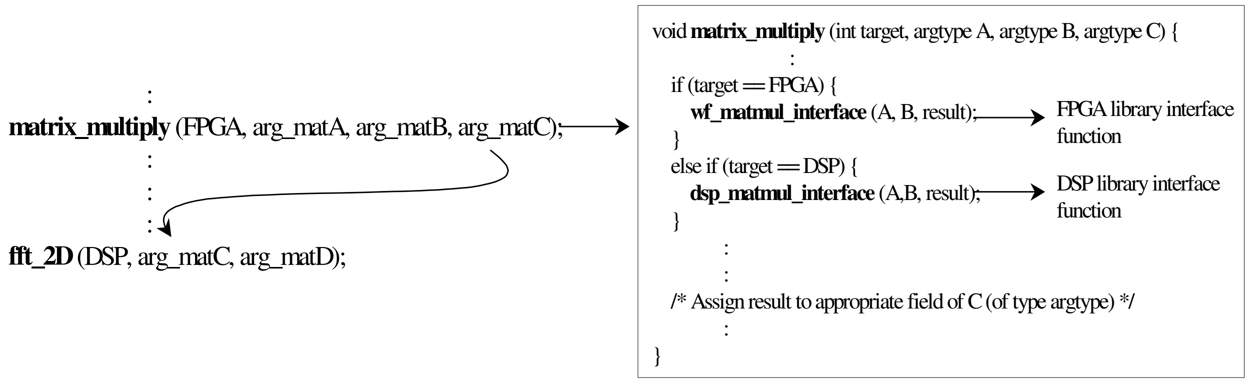


Figure 1.1 The MATCH Compiler

The initial stages of the compiler identify tasks in the input MATLAB program (assisted by directives entered by the user) that are suitable for execution in FPGAs. This information is passed through several phases of the compiler while extracting dependencies among the various tasks to be executed in the FPGAs and other components of the system. The final phase of the compiler generates target C code embedded with the function calls to the FPGA and DSP libraries. Sample compiler generated C code showing this functional interface to the FPGA and DSP libraries is shown in Figure 1.4.



Compiler generated C code

Figure 1.2 Compiler-Library Interfacing

The purpose of this thesis is to describe the development of the library units on the FPGA-based component of the system. In the following chapters we describe these functions and their hardware versions along with their C language interface to the compiler.

1.3 Related Work

In the past few years there have been significant developments in the general area of adaptive and reconfigurable computing architectures, systems and software platforms. These developments could be broadly categorized into two.

The first category includes developments at the architecture level. Many of these new architectures combine the advantages of a general-purpose processor with the reconfigurability of an attached co-processor. The RAW project [4] at MIT is an attempt at building a simple and wire-efficient architecture that scales with increasing VLSI gate densities. A RAW machine is configurable in that it exposes its low-level hardware details completely to the software system, eliminating the traditional compact instruction-set interface (ISA) between the hardware and the compiler. Traditional ISAs are unable to communicate much of the information available to the compiler to the runtime execution.

In RAW machines, the hardware and the compiler are closely coupled, and together provide a compact, higher level interface to rest of the software system.

The BRASS group [5] at the University of California, Berkeley, is investigating the integration of processors and reconfigurable logic. Their first pass at such an architecture is GARP [6] which combines a MIPS-II processor with a fine-grained FPGA coprocessor on the same die. They are also working on more efficient reconfigurable array designs, techniques to simplify and accelerate the mapping process, and strategies for building reconfigurable applications.

The IRAM project [7], also at Berkeley, proposes to architect, design, fabricate and evaluate a single chip supercomputer combining a configurable processor and high capacity DRAM to deliver vector supercomputer-style sustained floating point and memory performance, at vastly reduced power. The goal is to demonstrate that a single chip with a simple processor and a very high bandwidth local memory can be faster on memory-intensive problems as well as be a much better match to real-time applications.

The CHIMAERA project [8] here at Northwestern University is developing a complete adaptive computing system, including both hardware resources and software mapping tools, which tightly integrates reconfigurable logic with standard processors. By removing the communication bottleneck, and by virtually eliminating the reconfiguration time overhead, it produces a system that can support general-purpose computing. Hand-mapped algorithms to such a system is expected to gain a performance improvement of more than two orders of magnitude.

The second category include the developments in software technology that enable easy mapping of applications on reconfigurable systems and the development of such applications themselves. Some of these software developments are explained below.

The Cameron project [9] at the Colorado State University is an attempt to develop an automatic tool for image processing applications in *Khoros,* an advanced and widely used software development environment for signal processing. By targeting *Khoros*, they hope to extend the user base by several orders of magnitude. The objective of this project is to shift the programming paradigm of adaptive computing systems (ACS) from hardware-centered to software-centered, making them accessible to software engineers and application programmers as well as portable across platforms. It is specifically focused on Image Processing (IP) applications. Current adaptive computing systems present a very low-level programming abstraction as they are programmed using sophisticated but vendor-specific hardware design tools and rely on hardware description languages (VHDL) to describe algorithms. The goal of the Cameron project is to provide ease of programming of IP applications on ACS. The approach taken is to use dataflow graphs as an intermediate form between high level language and adaptive hardware. The SA-C (single assignment C) language allows the creation of such dataflow graphs, which are ideal for optimization and performance modeling. SA-C is a sub-set of C, with extensions that facilitate mapping high level code onto adaptive computing systems. Their approach involves an optimized compiler for SA-C that has been extended to support both IP applications and the mapping to reconfigurable hardware. The compiler targets an abstract architecture that consists of a set of parameterized hardware modules implemented in VHDL and mapped to a variety of FPGA-based reconfigurable platforms. Towards this end, they have implemented the IP components of a standard signal-processing library called VSIPL (Vector Signal Image Processing Library) in hardware using FPGAs. This is quite like our efforts in the MATCH project where we focus on developing library units on FPGAs for matrix and signal processing applications.

Annapolis Micro Systems Inc., the manufacturer of the custom computer board used in the MATCH project, intends to develop a hybrid computing engine called WASPP (WILDFORCE(TM)-based Adaptive Digital Signal Processing Project). WASPP [10] combines the performance advantages of a Field Programmable Gate Array (FPGA)-based computer to the floating-point arithmetic and complex algorithm execution capabilities of a Digital Signal Processor (DSP). They intend to execute an adaptive image thresholding algorithm, thereby demonstrating the substantial performance gains that result from the combination of the two computing hardwares. They also intend to develop a new design methodology, leveraging the robustness, availability and industry-wide acceptance of commercial off-the-shelf FPGA and DSP development tools, allowing the user to design applications more efficiently and reliably. The goal of this research is to define a methodology for developing applications and to prove that a FPGA/DSP hybrid architecture is better suited for solving certain classes of problems than FPGAs or DSPs alone can solve. The design methodology will involve the use of COTS tools in such a way so as to allow the designer to take advantage of the FPGA and DSP components of the WASPP architecture. They plan to integrate the COTS FPGA and DSP development tools into a seamless design environment and a library of design components will be provided to make FPGA/DSP algorithm co-design easy to use and understand.

The CHAMPION project [11] at University of Tennessee, Knoxville, also focuses on providing tools to automate the process of mapping image processing applications in *Khoros* onto reconfigurable systems. The goal of this project is to automate the mapping of *Khoros*-based applications onto adaptive computing systems to improve designer productivity by 100 times. Thus, more application designers will be able to achieve higher quality implementations in less time and adaptive computing systems will be utilized more effectively and by a wider audience. The approach taken is to build a library of pre-compiled primitives that can be used as building blocks of image processing applications. The *Khoros* flow graph will be automatically mapped onto the adaptive computing hardware by partitioning and floor-planning software that will be developed as part of the project. The library primitives in *Khoros* will be precompiled into FPGA configuration files and then characterized for area, delay, etc. in advance. This will not only accelerate the mapping process but will assist the designer in exploring candidate solutions before final implementation.

1.4 Contributions

The main contribution of this thesis has been the hardware implementation of a set of library functions for matrix and signal processing operations, which occur frequently in MATLAB signal processing kernels, along with their software interfaces to the compiler. These functions are useful for prototyping the MATCH compiler which can execute core routines in the MATLAB program in FPGA hardware by simply interfacing to these functions. Matrix multiplication, matrix addition, filtering and Fast Fourier transform are some of the most computationally intensive operations encountered in signal processing. By implementing these in hardware and benchmarking their performance gains over traditional methods, we have provided the necessary elements to demonstrate the effectiveness of reconfigurable computing and the usefulness of the MATCH compiler in automating the process of adaptive application development.

1.5 Thesis Outline

We first describe in Chapter 2 the reconfigurable computing system composed of the WILDCHILD multi-FPGA board and the Force 5V host. We describe the hardware components of the WILDCHILD board and its software interface to the host. We also describe the VHDL coding methodology and the design cycle involved in developing applications for the WILDCHILD system. Chapter 3 describes the S*caled Matrix Adder* function, its design and implementation. In Chapter 4 we describe the design and implementation of the *FIR/IIR filter* function. Chapter 5 describes *the Matrix Multiplier* function. We describe the design and implementation of the *Fast Fourier Transform* function in Chapter 6. We conclude in Chapter 7.

Chapter 2 Reconfigurable Computing with the WILDCHILD System

In this chapter we describe the WILDCHILD multi-FPGA system from Annapolis Microsystems and the process involved in the design, implementation and execution of reconfigurable computing applications on the system. We first describe the composition of the WILDCHILD reconfigurable computer in terms of the hardware units, the host environment and software interfaces that link the host and the hardware units. Next we describe the process involved in developing an application for the system using VHDL to describe the hardware functional units of the application and C to code the host interfaces to the hardware. The coding style and methodology that is employed in the development of the VHDL descriptions and the design cycle involving simulation and testing and optimizations to improve hardware resource usage and performance will be described in detail. We also describe the generic method employed in interfacing with the host using API calls in C.

2.1 The Reconfigurable Computing System

The computer described here is an integrated system consisting of the WILDCHILD multi-FPGA reconfigurable hardware unit and a host that controls it and provides a conventional software interface to the complete system. The WILDCHILD FPGA unit is a VME-compatible board that is installed in a standard chassis along with a VME-compatible host computer. The host, *Force 5V,* is a SPARC processor-based system running a version of the Solaris OS. The host and the WILDCHILD unit communicate through the VME bus, which is also shared by other components of the MATCH adaptive computing system as shown in Figure 2.1.

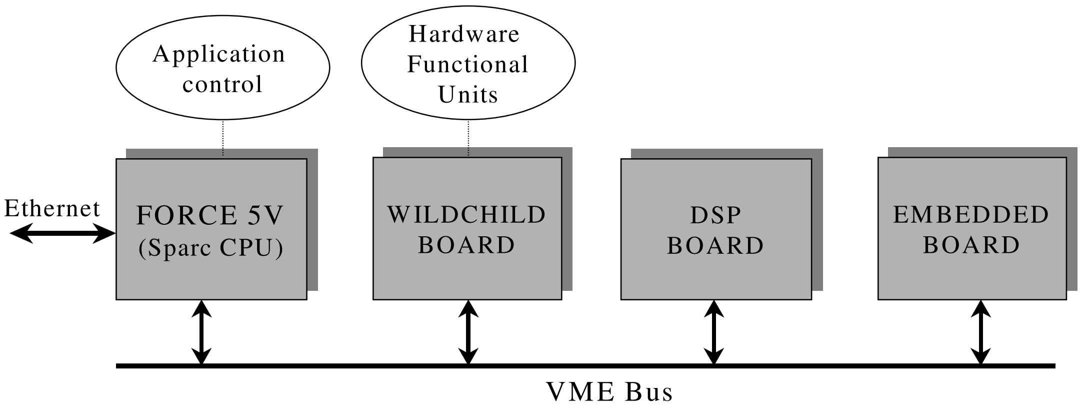


Figure 2.1 VME-based Reconfigurable Computing System

2.2 The WILDCHILD Multi-FPGA System

The WILDCHILD system consists of 9 FPGAs of the popular Xilinx 4000 family. Eight of these are 4010 FPGAs, referred to as Processing Elements 1 through 8 (PE1-PE8), with 400 CLBs (about 10,000 gates) each and are identical. The ninth is a 4028 FPGA referred to as Processing Element 0 (PE0) with 1024 CLBS (about 30,000 gates). The nine FPGAs are arranged in a master-slave configuration as shown in Figure 2.2.

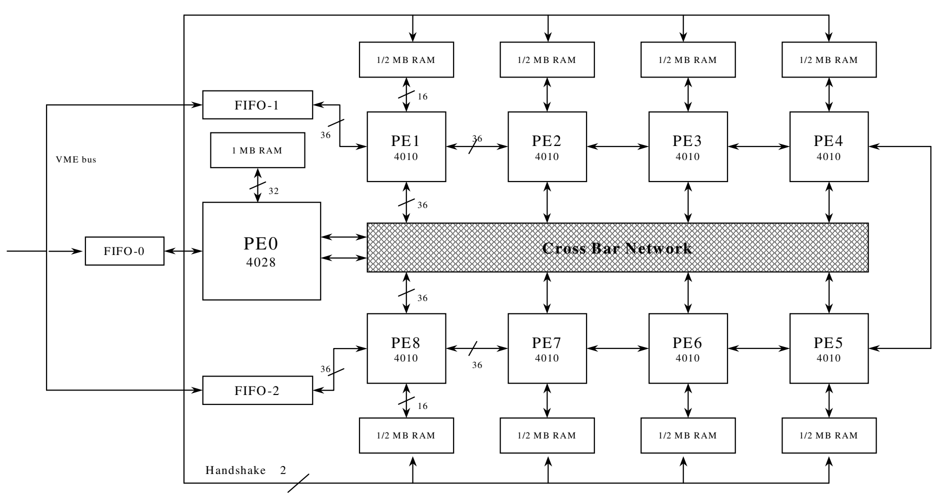


Figure 2.1 WILDCHILD System Architecture

Each FPGA on the board is connected to a pseudo dual-ported memory, which can be accessed by both the FPGA and the host. The memories connected to the 4010 FPGAs are 16 bits wide and contain 2^{18} addressable locations. The memory connected to PE0 is 32 bits wide and contains 2^{18} addressable locations. The dual port memory controller for each of these memories can be configured to give exclusive or arbitrated access to the host or the FPGA. In exclusive mode, the host is in complete control of the memory while in arbitrated mode, the FPGA is given priority in memory access and the memory is available to the host when the FPGA is not requesting access.

The architecture of the system enables high-throughput systolic computation using the 36-bit bus that connects each 4010 FPGA to its neighbors and also to the on-board FIFOs. Each bit of this bus can be configured to be driven by either of the FPGAs connected to it and this configuration can be changed during execution of the application. The bus driver circuit on the FPGAs is shown below in Figure 2.3.

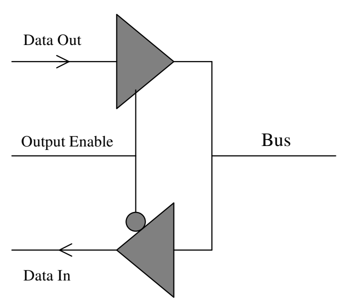


Figure 2.2 Bus Driver Circuit

There is also a crossbar network that can be used to realize any arbitrary interconnection of the FPGAs thus enabling irregular computations. The crossbar connection to each FPGA is also 36 bits wide and each nibble (4 bits) of the connection can be configured separately. But a nibble can be connected only to a corresponding nibble on another FPGA. The bus driver circuit for the crossbar on each FPGA is similar to the one in Figure 2.3. A 4-bit register on PE0 controls the crossbar configuration. It is possible to load 16 different configurations at any time and change the configuration by setting the register on the fly. Also, there is a separate broadcast mode where PE0 can transmit data to the crossbar ports of all the FPGAs at the same time irrespective of the currently active configuration. The crossbar configurations are loaded from a text file where nibble-to-nibble connections for each port are specified as shown in Figure 2.4.

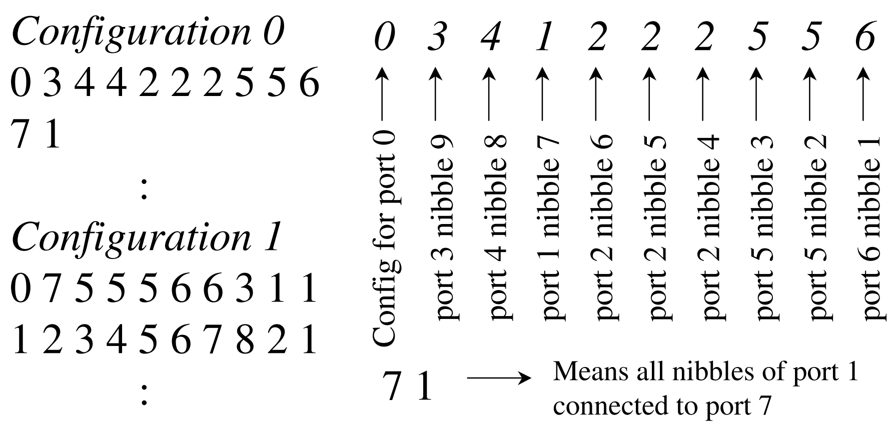


Figure 2.3 Crossbar Configuration Specification

As shown in Figure 2.2, two of the 3 on-board input/output FIFOs are connected to the systolic bus at each end of the PE1-PE8 chain. FIFO-1 is connected to the left bus of PE1 and FIFO-2 is connected to the right bus of PE8. The FIFOs are 16K deep and 36-bit wide and can be read or written directly by the host program. FIFO-0 is a larger FIFO that is 32K deep and 36-bit wide and is connected to PE0. All the three FIFOs on the board are controlled by PE0. In addition to the crossbar, PE0 can communicate control and status information to each FPGA using a set of global handshake signals. This is useful, for example, to communicate the reset and completion status to all FPGAs so that they can be synchronized.

All input and output ports (to the systolic bus and the crossbar) on each FPGA are buffered and the designer may choose to use the buffered or un-buffered version for a signal. The system clock rate is completely configurable to operate at any frequency though the board can only support operations up to a maximum frequency of 40 MHz. The system also features a global reset signal but the reset of each FPGA does not occur at the same time. This makes it necessary to include a handshaking mechanism in some designs to ensure that the FPGAs are synchronized at reset.

2.3 The WILDCHILD Host Interface

The host environment on the *FORCE 5V* consists of a set of drivers that communicate with the WILDCHILD board using other drivers that talk to the VME back-plane. A set of custom headers and library functions for the WILDCHILD system are installed for application development in C using the gcc compiler.

The software interface to the WILDCHILD system consists of a set of application program interfaces (APIs) provided by Annapolis Microsystems. These are provided in the form of a library of function calls in C that are included in the program running on the host. These functions perform various tasks like loading a circuit on an FPGA to implement a function, configuring the crossbar, loading memories with data etc. Some of the important functions that are used to perform the basic tasks involved in running any application are listed below. For a more comprehensive list and a more detailed description of each of the functions, please refer to the WILDCHILD user manual.

- *WFireOpen*

This function opens an interface to the WILDCHILD board in a specified unit and slot in the VME chassis. The main arguments are the unit number, slot number and mode. The function returns a handle to the board of type WF_BOARD.

- *WFResetBoard*

This function resets the specified board, the handle to which is provided as an argument.

- *WFireMaskPEInt*

This function masks the interrupt from a specified FPGA. The arguments are the board handle and FPGA (PE) number.

- *WFireProgramPE*

This function programs the specified PE with the bit stream that configures it to implement a circuit. The PE number and the bit stream file (*.m68) are the arguments.

- *WFireXBarConfig*

This function loads in a set of 16 configurations for the crossbar from a text file provided as the argument. One of these 16 configurations is active at any time depending on the 4-bit crossbar configuration register in PE0.

- *WFireClearPEInts*

This function clears any pending interrupts on the host from the PE. The PE number is provided as an argument.

- *WFireUnMaskPEInts*

This function unmasks PE interrupts to enable them to alert the host. The set of PEs is provided as the argument.

- *WFireSetMemoryMode*

This function sets the mode of operation of the dual-port memory controller for the specified PE. The mode is either BLOCKED where the PE is denied access to its local memory or ARBITRATE where the PE gets access to its memory on a priority basis. The arguments are the PE number and mode.

- *WFireClearMem*

This function clears the specified PE's memory i.e. sets all memory locations to 0. The PE number is provided as an argument.

- *WFireWrite*

This function loads the specified PE's local memory with data. The PE number, the starting location for loading data, the number of locations to load and a pointer to the host memory buffer that holds the data are the arguments to this function.

- *WFireRead*

This function reads the specified PE's local memory. The PE number, the memory offset, the number of elements to read and a pointer to a host memory buffer to hold the data read are provided as arguments. The pointed memory location is filled with requested data when the function returns.

- *WFireClkSetFreq*

This function configures the system clock to operate at a particular frequency. The clock frequency in Megahertz is provided as an argument.

- *WFireResetPE*

This function resets the specified PE or set of PEs. The set is provided as an argument.

- *WFireGetPEIntEvents*

This function reads the interrupt status register on the board. A bit in the status register is set if the corresponding PE has issued an interrupt. The argument to this function is the board handle.

- *WFireCloseBoard*

This function closes the WILDCHILD board. The handle to the board is provided as the argument.

2.4 FPGA Function Coding in VHDL

The VHSIC Hardware Description Language or VHDL is one of the most popular languages used to model digital systems. Entire systems can be described using this language and can be simulated to verify their correctness before actually building the systems in hardware. The language supports a very hierarchical design style where systems are built up from subsystems that have previously been tested for their functionality.

Systems can be described in VHDL using varying levels of abstraction. At one end of the spectrum is the purely behavioral description where the behavior of the system to external and internal stimulus is described using a set of equations in straight code like any other programming language. This is usually the initial model of a system where the details of implementation are not known but the general expected behavior is known. At the other extreme is the purely structural description where the system is modeled as an interconnection of components using wires like in a real digital system. This often represents a system in the final stages of its implementation and is usually more difficult to code than the behavioral model. The individual components can themselves be behavioral or structural descriptions.

Often, most systems are designed using a combined approach - modeling known components in structural VHDL and describing the yet unknown components using behavioral VHDL. Purely structural models can be easily synthesized by an automatic tool into actual hardware while automatically synthesizing purely behavioral models is still one of the most difficult problems in synthesis.

The register transfer level (RTL) description of digital systems is a very popular approach that employs a level of abstraction that is lower than purely behavioral systems but higher than gate-level structural descriptions. They are often used to describe behavior of systems that progress from one state to another (usually on a global clock signal) in a controlled manner where information is transferred between registers from state to state. Such systems can be viewed as consisting of a data path where registers transfer data on each edge of the clock signal and are controlled by a set of signals generated by a control unit operating on the edge of the same clock signal. We have adopted the RTL methodology for describing our systems, as it is abstract enough for easy design and simple enough for automatic synthesis.

The RTL model in VHDL essentially consists of a **Process** construct, which describes a state machine usually operating at the positive or negative edge of the global clock signal. The process is 'sensitized' to two global signals - the system reset and clock. When the reset signal is asserted, the process executes actions that initializes the subsystem, usually by assigning initial values to registers and signals and resetting the state machine to start from the first state. When the reset signal is not asserted, the process executes the state machine for every positive (or negative) edge of the clock signal. Registers and control signals are assigned for the current state and the state variable is set to the next state to be executed on the next edge of the clock. A sample process in VHDL that describes a state machine is shown below in Figure 2.5. Refer to [12] for a better understanding of the RTL methodology for FPGA synthesis.

```
controller : process (clock, reset)                          -- Sensitive to clock
and reset
begin
        if reset = '1' then                                  -- Assign reset values
                ram_data_out     <= (others => '0');
                done             <= '0';
                ram_addr_sig     <= DATA_ADDR;
                RAM_REQn         <= '1';
                controller_state <= init_state;
        elsif rising_edge(clock) then                        -- Execute for rising
edge of clock
                done             <= '0';                     -- Default
                assignments
                RAM_REQn         <= '0';
                case controller_state is
                        when init_state  =>
                                ram_data_out     <= (others => '0');
                                ram_addr_sig     <= DATA_ADDR;
                                if strobe_in = '1' then
                                        controller_state <= state0;     -- Assign next state
                                else
                                        controller_state <= init_state;
                                end if;
                        when state0     =>
```

```
                    ram_data_out      <= data_in;
                    controller_state  <= state1;
               when state1     =>
                    ram_data_out      <= data_in;
                    controller_state  <= state2;
                    :
                    :
               when end_state =>
                    RAM_REQn          <= '1';
                    controller_state  <= end_state;        -- Loop back to end
               state
               when others =>
                    controller_state  <= init_state;       -- Initial state for
               everything else
          end case;
     end if;
end process controller;
```

Figure 2.1 A State Machine Process in VHDL

2.5 The WILDCHILD Design Cycle

Annapolis Microsystems, the manufacturer of the WILDCHILD board, provides a complete description of the entire WILDCHILD board in VHDL down to the level of each individual FPGA. So the application design process simply involves coding the RTL description of the digital system on each FPGA in VHDL and connecting this system to the interfaces on the FPGA provided by the manufacturer. The entire system could then be simulated on a VHDL simulator like *ModelSim* and the functionality of the design can be verified. Figure 2.6 shows a sample design on the PEX FPGA with its interface to the VHDL model of the FPGA provided by the manufacturer.

```
architecture my_design of PEX_Core is
     component my_module              -- A pre-defined module and its interfaces
       port (
          clock           : in  std_logic;
          reset           : in  std_logic;
          data_in         : in  std_logic_vector (15 downto 0);
          strobe_in       : in  std_logic;
          ram_data_out    : out std_logic_vector (15 downto 0);
          ram_addr        : out std_logic_vector (17 downto 0);
          RAM_REQn        : out std_logic;
          done            : out std_logic);
     end component;
begin
     --  "Inactive" output port signal assignments
     Left_Out         <= ( others => '0' );       -- Set left systolic bus to zero
```

```
        Left_OE         <= ( others => '0' );      -- Disable left systolic bus output
        Right_Out       <= ( others => '0' );      -- Set right systolic bus to zero
        Right_OE        <= ( others => '0' );      -- Disable right systolic bus output
        XB_Out          <= ( others => '0' );      -- Set crossbar data to zero
        XB_OE           <= ( others => '0' );      -- Disable data to crossbar
        XB_En_n         <= ( others => '1' );      -- Disable data from crossbar
        LED             <= '0';                    -- Turn off LED
        HS0_Out         <= '0';                    -- Set handshake '0' to host to '0'
        HS0_OE          <= '0';                    -- Disable handshake '0' to host
        HS1_Out         <= '0';                    -- Set handshake '1' to host to '0'
        HS1_OE          <= '0';                    -- Disable handshake '1' to host
        GORR_Out        <= '0';                    -- Set GOR result to PE0 to '0'
        GORR_OE         <= '0';                    -- Disable GOR result to PE0
        GORV_Out        <= '0';                    -- Set GOR valid to PE0 to '0'
        GORV_OE         <= '0';                    -- Disable GORR valid to PE0
        MwriteEn_n      <= '0';

        my_label : my_module                       -- Connect ports of the module to ports of
PEX FPGA
                port map (
                        clock           => Pclk,            -- System clock signal
                        reset           => Reset,           -- System reset signal
                        data_in         => Right_In(15 downto 0),
                        strobe_in       => Right_In(16),
                        ram_data_out    => Mdata_OutFF,
                        ram_addr        => Maddr_OutFF,
                        RAM_REQn        => Mreq_n,
                        done            => Intr);           -- PEX interrupt
end my_design;
```

Figure 2.1 Interfacing of Module with VHDL model of PEX FPGA

Once the design has been successfully simulated and verified, the VHDL description of the entire FPGA, including the manufacturer-supplied description of the interface logic on the FPGA is synthesized using the *Synplify* tool from Synplicity Corporation. Most of the interface logic on the FPGAs have been pre-compiled into gates by the manufacturer and so the VHDL description of the interfaces merely refer to these compiled modules instead of actually describing them. The Synplify tool processes the design in two steps. The compilation phase checks the design for syntax and logical errors like combinational loops, extracts state machines and counters and optimizes them and analyzes possibilities for sharing resources among states. A logic implementation for the VHDL description, taking into account any timing goal specified, is built during this phase. In the mapping phase, the tool tries to map generated logic into basic units on the target FPGA (Xilinx 4000 series in our case) trying to optimize the use of resources. The result of the synthesis is a netlist for the design in Xilinx's .xnf format.

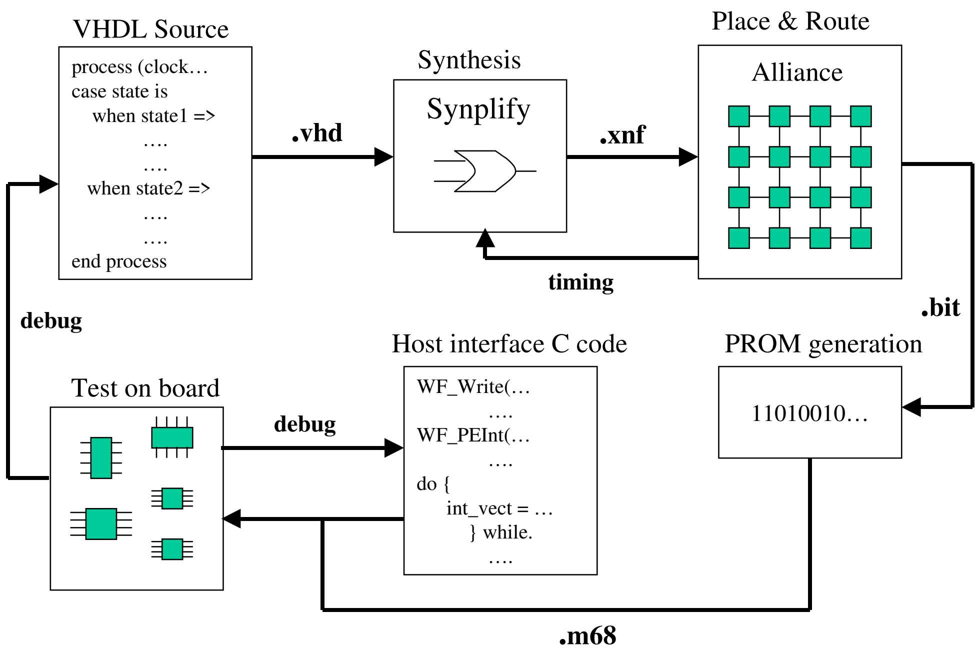


Figure 2.2 Application Design Cycle

The synthesis tool reports the resource requirements for the design (in terms of CLB count on the target FPGA) and also the results of the timing driven synthesis (in terms of delay of critical path). If the generated logic does not fit on the target FPGA, the designer has the choice of altering the design to optimize it for resource utilization, or specifying a lower timing goal to let the synthesis tool trade clock speed for area. Often small changes in design strategy lead to big improvements in speed and/or resource utilization.

After synthesis, the output netlist is loaded into Xilinx's design manager. Here the constraint file for the FPGA which specifies the mapping from ports in the design to pins on the FPGA (provided by the manufacturer) and also the timing goal (from the synthesis result) is specified along with the desired place and route effort level. The design manager runs the mapper (to load in pre-compiled logic and do any final mapping), the placer and then the router on the netlist. Since the pre-compiled interface logic and routing requirements are unaccounted for during the synthesis phase, the actual resource requirements during place and route differ significantly from the results reported from synthesis. So a design that seems to 'fit' after the synthesis step may not do so after place and route. The result of the place and route process is an FPGA bit stream that corresponds to the circuit on the FPGA. The bit stream (*.bit) is then converted to a PROM format using the **promgen** tool from Xilinx. The result (*.mcs) is then converted by the **mcs2bin** tool from Annapolis Microsystems into a special format (*.m68) for downloading to the WILDCHILD board.

A program in C on the host is then developed using the WILDCHILD library function calls described in section 2.3 to download the configurations into the FPGAs, the data into the memories and the crossbar configuration for the design. After configuring the clock and interrupts, the reset signal on the board is de-asserted to begin computation in hardware. The program waits in a tight loop reading the interrupt status register to detect end of computation. Once the interrupt is detected, the program reads the result data from the FPGA memories.

Some Tips

If the design simulates fine in the simulator but does not work on the board, it is likely that the C program is the culprit. Debugging is much easier if this is the case, as the changes need to be made only to the C program, which can be re-compiled quickly. Any changes made to the original VHDL code will need much longer to test on the board, as the design cycle is long - sometimes nearly an hour for large designs. Hence it is important to test the design exhaustively in the simulator before proceeding with Synthesis. Also, it is often useful to employ simple debugging techniques like adding a few extra states that can provide more information about the status of the system. Some special data could be provided in memory that is read during initialization and controls the condition for entry into these states.

To improve design performance, it is sometimes necessary to break complicated logic into simpler steps. Adding more states does not necessarily lead to an increase in resource requirements. In fact, many times they lead to a decrease due to some other side effects of simplification of per-state computations. Also, due to limitations in automatic synthesis and sometimes bugs in the synthesis tool, manual instantiation of repetitive logic using simple structural instantiations leads to better results than automatic synthesis.

2.6 Summary

In this chapter we described the hardware and software components of the WILDCHILD reconfigurable computing system. We described the common design methodologies used in VHDL and choice of RTL for our designs. We then described the design cycle involved in the development of an application on the WILDCHILD board starting from VHDL all the way down to the FPGA bit streams and offered some tips for improving performance and resource utilization of the designs. In the following chapters we describe the design and implementation of various library functions on the WILDCHILD system.

Chapter 3 Scaled Matrix Adder Function

In this chapter we describe the design and implementation of a 16-bit fixed-point matrix adder function on the WILDCHILD. Using this function up to 7 matrices, each with a maximum of 250,000 elements, can be added in parallel at hardware speeds. We first describe the mathematical form of the function and its MATLAB and C language versions. We then look at some of the design alternatives and analyze them for their merits. We then describe our implementation of the different modules of the function along with the host interfaces. Finally, we tabulate the execution times of the hardware implementation.

3.1 Function Description

Several situations arise in the computation of matrix equations where there is a need to find the sum or difference of a set of matrices, often their scaled versions i.e. each element of every matrix is pre-multiplied by a simple constant before the addition or subtraction is performed. An example of such a situation is the solution of linear equations where the set of equations is represented as a single scaled matrix (vector) equation as shown below in Figure 3.1. Once the equation is solved, the variables are substituted in the equations on the right-hand-side to compute the left-hand-side values.

$$
\begin{aligned}
z_1 &= x \cdot a_{11} + y \cdot a_{12} + z \cdot a_{13} \\
z_2 &= x \cdot a_{21} + y \cdot a_{22} + z \cdot a_{23} \\
&\vdots \\
z_n &= x \cdot a_{n1} + y \cdot a_{n2} + z \cdot a_{n3}
\end{aligned}
\qquad
\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} = x \cdot \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{bmatrix} + y \cdot \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{n2} \end{bmatrix} + z \cdot \begin{bmatrix} a_{13} \\ a_{23} \\ \vdots \\ a_{n3} \end{bmatrix}
$$

Figure 3.1 Example of Scaled Vector Addition

Figure 3.2 shows the MATLAB and C versions of this operation.

$$
\begin{aligned}
A_1 &= [a_{11}\ a_{21}\ a_{31} \cdots a_{n1}]; \\
A_2 &= [a_{12}\ a_{22}\ a_{32} \cdots a_{n2}]; \\
A_3 &= [a_{13}\ a_{23}\ a_{33} \cdots a_{n3}]; \\
Z &= x * A_1 + y * A_2 + z * A_3;
\end{aligned}
$$

(a)

```
for (i=0; i<n; i++) {
   z[i] = x*a[i][1] + y*a[i][2] + z*a[i][3];
}
```

(b)

Figure 3.2 (a) MATLAB code for Scaled Vector Addition (b) Corresponding C code

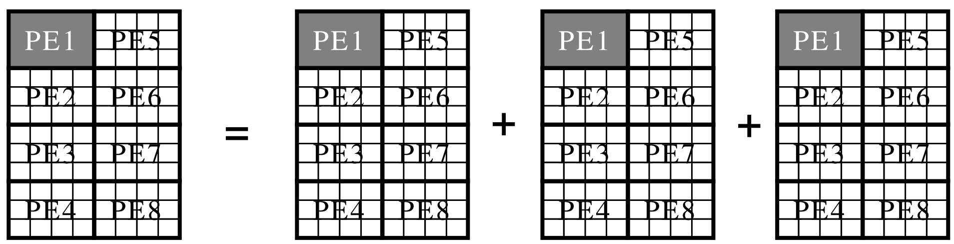
The scaling of the corresponding elements of each matrix can be done in parallel and also all the elements of the resultant matrix can be computed in parallel after they have been scaled. The manner in which the operation is parallelized will have strong implications for the resource requirements and performance of any design that implements it. In the next section we explore the different design options and their implications and justify our approach.

3.2 Design Alternatives

There are several ways one could implement the addition function on a parallel platform. These implementations generally differ in the manner in which data and computation tasks are distributed among the multiple processors, which in turn depend on the architecture of the underlying system and the communication mechanisms. For the case of multi-FPGA systems, there are additional constraints like resource usage and communication bandwidths that affect the decision. We explore some of the alternatives next and evaluate their merits.

3.2.1 Alternative 1: Block Partitioning

The output matrix is partitioned into N chunks, where N is the number of processors, and each processor performs the addition on its portion of the output matrix. This requires that each input matrix is partitioned in the same manner and the chunks distributed to the processors' local memories. Also each processor needs a copy of the set of scaling constants. The data distribution and flow for this scheme is shown in Figure 3.3.



$$S = k_1*A + k_2*B + k_3*C$$

Figure 3.1 Block Partitioning

In the WILDCHILD system, each PEX FPGA could act as a processor, reading matrix elements from its local memory, multiplying each with the corresponding constant and accumulating the result in a local register. The result can then be written back to memory. This way each processor functions independently of the other processors.

The advantage of this design is that the size of the operation is limited only by the size of each processor's memory and does not depend on the number of processors available. The disadvantage of this design arises from the following: The memories in the WILDCHILD system have a non-zero bus turnaround time. This means that for each change between read and write modes, a clock cycle is lost in turning the bus around. This may not be significant for matrix additions involving several matrices, as the percentage of wasted clock cycles may not be large. But for smaller number of matrices this wastage could be significant. For example, for addition of five matrices, there are 2 wasted clock cycle for every 5 clock cycles which leads to a 30% loss in performance. One way to reduce this wastage is to buffer the results of several sets of additions and write the entire buffer in one shot.

3.2.2 Alternative 2: Matrix-wise Distribution

An alternative data partitioning involves distribution of entire matrices on separate processor memories. Each processor reads a matrix element from its local memory, scales it with the locally stored constant and adds it to the sum it receives from the previous processor. It then passes this sum on to the next processor. The last processor reads the incoming sum and writes it to its local memory. The processors could then be classified as Reader, Adder and Writer.

The preceding design was implemented on the WILDCHILD system. The design and implementation is described in detail in the next section. The advantage of this design is that since each FPGA will either always read or write memory, there is no penalty for bus turn-around as in the previous design. The disadvantage of this design is that one of the FPGAs does the trivial operation of only writing incoming data to memory and does not do any processing. The more serious limitation is that the size of the matrix equation is limited to a maximum of seven terms.

3.3 Implementation of Matrix Adder on WILDCHILD

In this section we describe the architecture of the function on the WILDCHILD and the design and implementation of the different components of the function in hardware.

3.3.1 Architecture of Matrix Adder on WILDCHILD

The architecture for the scaled matrix adder function is shown below in Figure 3.4. The matrix Adder is comprised of three separate designs - the Reader, Adder and Writer. Each of these units was targeted for the PEX FPGAs and the PE0 FPGA was left blank. The next section described each of the modules in detail.

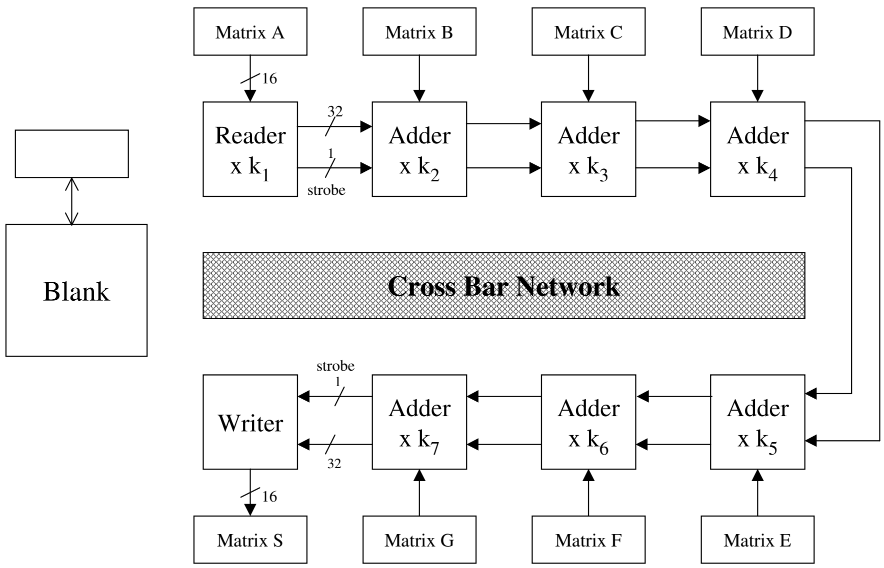


Figure 3.1 Architecture of Matrix Adder Function

3.3.2 Reader Module

The Reader reads in a 16-bit matrix element every clock cycle from memory, multiplies it with a constant and sends out the 32-bit product on the right systolic bus. The constant is read from memory during the initialization phase and stored in a local register. The multiplier is a single clock cycle multiplier generated by the synthesis tool. Refer to Appendix A for the RTL description of the Reader module. The Reader asserts the strobe line on the right bus to signal to the next processor that valid data is available on the bus. Figure 3.5 shows the Reader circuit on PEX.

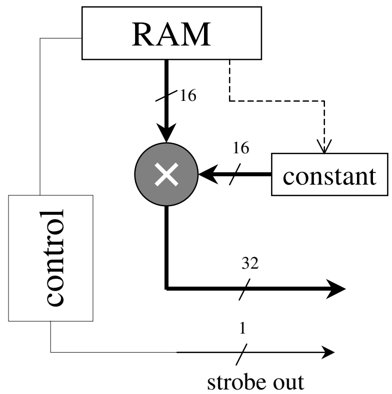


Figure 3.1 Schematic of Reader

Resource Requirements and Performance of Reader

The synthesis of the RTL description resulted in a Xilinx netlist that required at least 277 CLBs (70%). The timing goal in synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 18 MHz. The final placed and routed version used 309 CLBs (70%) and could be clocked at a maximum frequency of 9 MHz.

3.3.3 Adder Module

The Adder reads in a 16-bit matrix element from the memory every clock cycle and multiplies it with a constant. It then adds this 32-bit product to the 32-bit number read from the left systolic bus during the same clock cycle and sends the 32-bit sum to the right systolic bus. The constant is read from memory during the initialization phase and stored in a local register. Refer to Appendix A for the RTL description of the Adder module. The Adder waits on the input strobe signal from the preceding processor to start processing. The output strobe signal is asserted as soon as an input strobe is detected. Though the start of computation is dependent on the input strobe from the previous processor, the end of computation is determined by a count that is loaded from memory during the initialization phase. Figure 3.6 shows the Adder circuit on PEX.

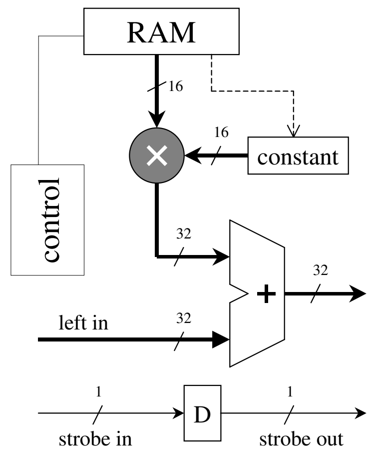


Figure 3.1 Schematic of Adder

Resource Requirements and Performance of Adder

The synthesis of the RTL description resulted in a Xilinx netlist that required at least 296 CLBs (74%). The timing goal in synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 18 MHz. The final placed and routed version used 330 CLBs (82%) and could be clocked at a maximum frequency of 8 MHz.

3.3.3 Writer Module

The Writer reads in a 32-bit number every clock cycle from the left systolic bus and sends out the most significant 16 bits on the local memory bus. Refer to Appendix A for the RTL description of the Writer module. The Writer waits on the input strobe signal from the preceding processor to start writing data to memory and stops as soon as the input strobe is turned off. It also asserts the interrupt signal briefly to signal the end of computation. This interrupt is used by the host to determine that data is ready to be read from memory.

Resource Requirements and Performance of Writer

The synthesis of the RTL description resulted in a Xilinx netlist that required at least 25 CLBs (7%). The timing goal in synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 52 MHz. The final placed and routed version used 30 CLBs (7%) and could be clocked at a maximum frequency of 46 MHz.

3.4 Host Interface to Matrix Adder

The interface to the host is the function wf_matadd_interface() which performs the mapping between the host data structures and the structures specific to the FPGA function. This interface function contains within it two separate functions - the function loader and function evaluator. A part of the interface code is shown in Figure 3.7.

```
void wf_matadd_interface (argtype *src_matrices, int num_matrices, argtype constants, void
**dest)
{
            :
  WF_MATRIX *out_mat;
  WF_MATRIX_ARRAY *matrices;
  WF_INTEGER_ARRAY *consts;
            :
  /* Set clock frequency */
  clk_freq = FLOAT(9.0);
  WFireClkSetFreq( hBoard, &clk_freq );

  /* Free run clock */
  WFireClkFreeRun( hBoard );

  /* Load configuration */
  wf_matadd_load();                                   /* Function loader is called */

  /* Re-assign input vector from arg_type */
  matrices->length = num_matrices;
  for ( i = 0; i<num_matrices; i++) {
```

```
   matrices->matrix[i].data.length = src_matrices[i].diminfo[0].bound*
src_matrices[i].diminfo[1].bound;
   matrices->matrix[i].data.byte = (unsigned char*) src_matrices[i].data;
   consts-> value = constants.data;
 }

 /* Call function */
 wf_matadd (matrices, consts, out_matrix);                    /* Function evaluator is called */

 /* Assign destination */
 *dest = (void*)out_matrix.data.byte;
}
```

Figure 3.1 Matrix Adder Host Interface

The loader configures each FPGA (except PE0) on the WILDCHILD system with one of Reader, Adder or Writer configurations and also loads in the default configuration of the crossbar. PE0 is loaded with the 'blank' configuration. If, for example, the matrix equation involves 5 terms, PE1 will be loaded with the Reader configuration and would hold data for the first matrix. PE2 through PE5 will be configured as Adders and will hold the data for the 4 other matrices. PE6 will be configured as Writer and will hold the result of the computation. PE0, PE7 and PE8 will be configured as 'blank' modules. A part of the function loader routine is shown below in Figure 3.8.

```
void wf_matadd_load ()
{
        :
        :
 /* Program PE0 with blank design */
 WFireProgramPE ( hBoard, WF_PE0_BIT, "pe0blank.m68");

 /* Program all PEXs with adder */
 WFireProgramPE (hBoard, WF_ALL_PE1_16, "pex_adder.m68");

 /* Program PE1 with reader */
 WFireProgramPE (hBoard, WF_PE1_BIT, "pex_reader.m68");

 /* Program PE8 with writer */
 WFireProgramPE (hBoard, WF_PE8_BIT, "pex_writer.m68");

 /* Load XBar configuration */
 WFireXBarConfig ( hBoard, "default.cfg");

 /* Clear PE interrupts */
 WFireClearPEInts ( hBoard, WF_ALL_PEs );
```

```
 /* Block memory access to all PEXs */
 WfireSetMemoryMode ( hBoard, WF_ALL_PE1_16, WF_BLOCKED);
}
```

Figure 3.2 FPGA Configuration Loader Function for Matrix Adder

The evaluator function is used to download data to the FPGA memories and to actually evaluate the matrix expression. The first eight locations in memory are special purpose locations that contain some configuration information and the 16-bit scaling constant. The memory data organization is shown below in Figure 3.9.

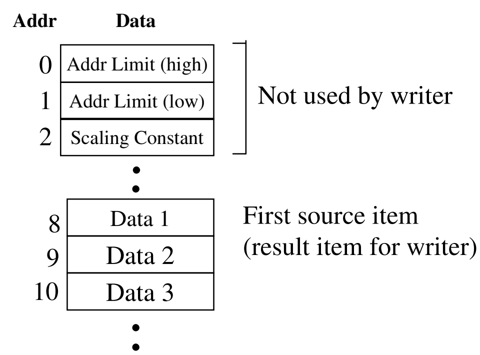


Figure 3.3 (PEX) Memory Data Organization

The evaluator function packs the configuration data and the input data into a single buffer and performs one write operation to the target FPGA memory. Similarly, after receiving the function completion interrupt, it reads back the FPGA memory, unpacks result data from the data buffer and returns the result in the appropriate data structure. This data structure is then mapped into the compiler's data structure by the interface function. A part of the evaluator routine is shown below in Figure 3.10.

```
void wf_matadd (WF_MATRIX_ARRAY *mat_array, WF_INTEGER_ARRAY *const_array,
                WF_MATRIX *mat_out)
{
        :
        :
 /* Inactivate unused PEs */
 buf[0] = 262143;
 buf[1] = 0;
 for (i=mat_array->length+1; i <=8; i++) {
```

```
  /* Write PEX memory */
  WFireWrite ( hBoard, i, 0, 3, buf);
}

index = 0;
for (i=1; i<mat_array->length+1;  i++) {
  /* Set last data address */
  buf[0] = (mat_array->matrix[index].data.length/2 + 8)/65536 * 65536;
  buf[0] += (mat_array->matrix[index].data.length/2 + 8)%65536;
  buf[1] = const_array->values[index] * 65536;
  byte_loc = (unsigned char*)buf;
  byte_loc += 16;

  /* Assemble data for matrix */
  /* - assumed to be in column major order and little end-ian */
  for (j=0; j<mat_array->matrix[index].data.length; j++) {
    *byte_loc = mat_array->matrix[index].data.byte[j];
    byte_loc++;
  }

  /* Write PEX memory */
  WFireWrite ( hBoard, i, 0, j/4+4, buf);
  index++;
}

/* Give memory access to PEs */
WFireSetMemoryMode( hBoard, WF_ALL_PE1_16, WF_HW_ARBITRATE );

/* Start processors */
WFireResetPE(hBoard, WF_ALL_PEs, WF_DISABLE);

/* Wait for PE8 interrupt */
do {
  interrupt_vect = WFireGetPEIntEvents(hBoard);
} while(!(interrupt_vect & WF_PE8_BIT));

/* Read back PE8 memory */
WFireSetMemoryMode(hBoard, WF_PE8_BIT, WF_BLOCKED);
read_count = mat_array->matrix[0].data.length/4;
WFireRead (hBoard, WF_PE8_NUM, 4, read_count, buf);

/* Assign result */
byte_loc = (unsigned char*)buf;
mat_out->num_rows = mat_array->matrix[0].num_rows;
mat_out->num_cols = mat_array->matrix[0].num_cols;
mat_out->data.length = mat_array->matrix[0].data.length;
mat_out->data.byte = byte_loc;
}
```

Figure 3.4 Evaluator Function for Matrix Adder

3.5 Results

The 16-bit fixed-point Matrix Adder function was tested on various data set sizes and the configuration and compute times are shown in Table 3.1. The timing results are also compared against C implementations of the matrix equations run on the host processor. Since the Adder module can run only at 9 MHz, it determine the system clock. It is evident from the data that the configuration and data transfer times dominate the evaluation time by several orders of magnitude. While pre-loading of functions could potentially hide the configuration times, the FPGA function time is still largely due to the data transfer times.

Data Size (Total elements)	Configuration Time (ms)	Data Transfer Time (Download+Readback) (ms)	Compute Time (ms)	Compute Time of C code on host (ms)
31250	3300	195+28=223	0.8	186
62500	3300	386+55=441	1.8	361
125000	3300	793+110=903	3.7	730
250000	3300	1600+220=1820	7.5	1462

Table 3.1 Performance of Matrix Adder Function on WILDCHILD

3.6 Summary

In this chapter we have described the design and implementation of a matrix adder/subtractor function on the WILDCHILD FPGA system. We have described the design options available based on different methods of partitioning matrix data and the advantage and disadvantage of each method with respect to implementation. We then described the architecture of our implementation and its components and reported the

FPGA resource usage and clock performance of each component. We also described the C application program interface to the matrix adder, which consists of a function loader and function evaluator. Finally we presented the performance results for computation of various size of matrix equations. In the next chapter we describe the IIR/FIR filter function, its design and implementation.

Chapter 4 Filter Function

In this chapter we describe the design and implementation of an 8-bit fixed point filter function on the WILDCHILD. Using this function, FIR or IIR filtering of up to order 64 can be performed on vectors of maximum size 250,000 elements at hardware speeds. We first describe the mathematical form of the function and its MATLAB and C language versions. We then look at some of the design alternatives and analyze them for their merits. We then describe our implementation of the different modules of the function along with the host interfaces. Finally we tabulate the execution times of the hardware implementation.

4.1 Function Description

Filtering is one of the most common operations performed in signal processing. MATLAB provides the ***filter*** function that can be used to perform different filtering operations on vectors of signal data. All filters belong to one of two classes - FIR for Finite Impulse Response and IIR for Infinite Impulse Response filter. While the actual implementation of these two classes of filters is different, mathematically they are quite similar. The general filter equation is shown below in Figure 4.1.

$$y(n) = \sum_{i=0}^{N} a_i x(n-i) + \sum_{i=1}^{N} b_i y(n-i)$$

Figure 4.1 General Filter Equation

An IIR filter can be used to perform FIR filtering by setting all of the constants 'b' to 0. MATLAB and C codes for a 4-tap IIR filter operation is shown below in Figure 4.2.

$$X = [x_0\ x_1 \cdots x_n]$$
$$a = [a_0\ a_1\ a_2\ a_3\ a_4]$$
$$b = [b_0\ b_1\ b_2\ b_3\ b_4]$$
$$Y = filter(a, b, X)$$

a and b are coefficient vectors
X is the input vector and Y is the output vector

(a)

```
x4=x3=x2=x1=0;
y4=y3=y2=y1=0;
for (i=0; i<n; i++) {
   y(i) = a(0)*x(i)+a(1)*x1+a(2)*x2+a(3)*x3+a(4)*x4
            - b(1)*y1 - b(2)*y2 - b(3)*y3 - b(4)*y4;
   x4=x3;  x3=x2;  x2=x1;  x1=x(i);
   y4=y3;  y3=y2;  y2=y1;  y1=y(i);
}
```

(b)

Figure 4.2 (a) MATLAB code for filter (b) Corresponding C code

4.2 Design Alternatives

There are several schemes for implementing the general filter equation in Figure 4.1. The different schemes have different implications for the hardware complexity and performance of the filter. Also, the best hardware scheme may not be suitable for implementation on the WILDCHILD FPGA system due to resource constraints. Some of the schemes are described below.

4.2.1 Alternative 1: Direct Form I

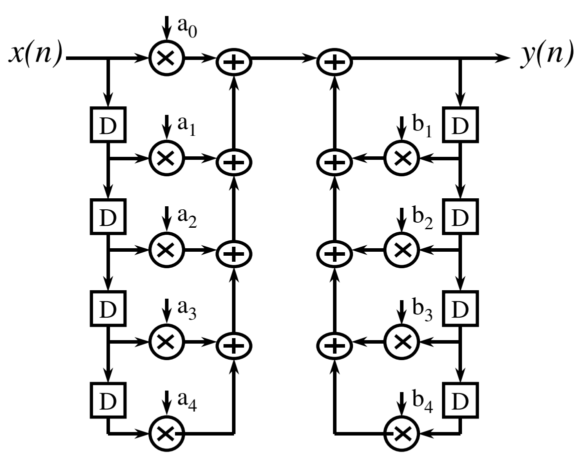


Figure 4.1 Direct Form I of 4th Order IIR Filter

Figure 4.3 shows the direct implementation of the basic delay-tap IIR filter equation. As can be seen from the figure, this would require as many delay blocks (registers) as there are delay elements in the equation. But a simple manipulation of the basic equation results in a different form of the filter that takes up fewer resources as shown below in Figure 4.4.

4.2.2 Alternative 2: Direct Form II

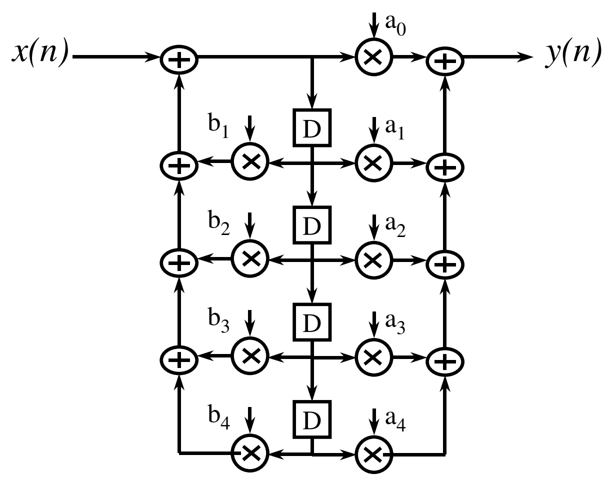


Figure 4.1 Direct Form II of 4th Order IIR Filter

In the direct II form, the delay units are shared between the recursive and non-recursive paths and thus the number of delay registers required is reduced.

The above forms of the IIR filter, though very efficient, are difficult to parallelize using multiple processors. Also, they are not scalable for higher-order filters that involve more delay taps. What is needed is a multi-processor design consisting of repeated units that is scalable. This makes the following form of the filter attractive for our implementation.

4.2.3 Alternative 3: Cascade Form

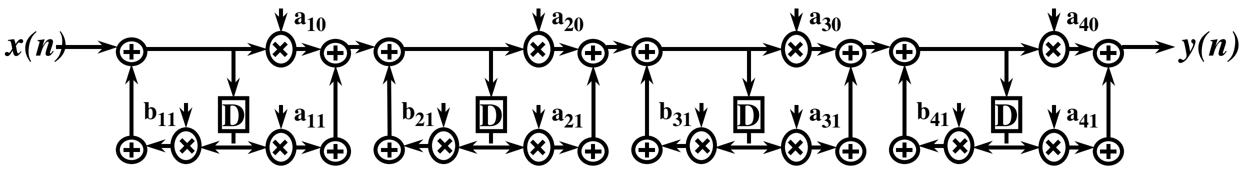


Figure 4.1 Cascade Form of 4th Order IIR Filter

The basic filter equation, through some transformations, can be factorized into a product of many smaller sized terms. This corresponds to a cascade of several lower-order filters to simulate the effect of the initial higher-order filter. For example, a 16-tap (16th order) filter can be constructed using a cascade of four 4-tap filters or eight 2-tap (second order) filters. Note that the filter coefficients for the new cascaded units are different from the original filter.

4.3 Implementation of Filter on WILDCHILD

The cascaded form of the IIR filter lends well to implementation on the multi-FPGA architecture of the WILDCHILD system due to the presence of near-neighbor communication capability. Several FPGAs could be strung together in series to implement the required filter operation.

The primary constraint in our implementation has been the resource requirements of the filter unit on each FPGA. Since the FPGAs on the WILDCHILD board are very modest, we were able to fit only a 1-tap filter (first order filter or 'integrator') on an FPGA. This is indeed a trivial filter by signal processing standards where a second-order filter is usually considered the basic building block. But this was not a serious concern to us because in the following section we describe some tricks that were employed to enable the filter design to handle any-order filtering.

4.3.1 Architecture of Filter on WILDCHILD

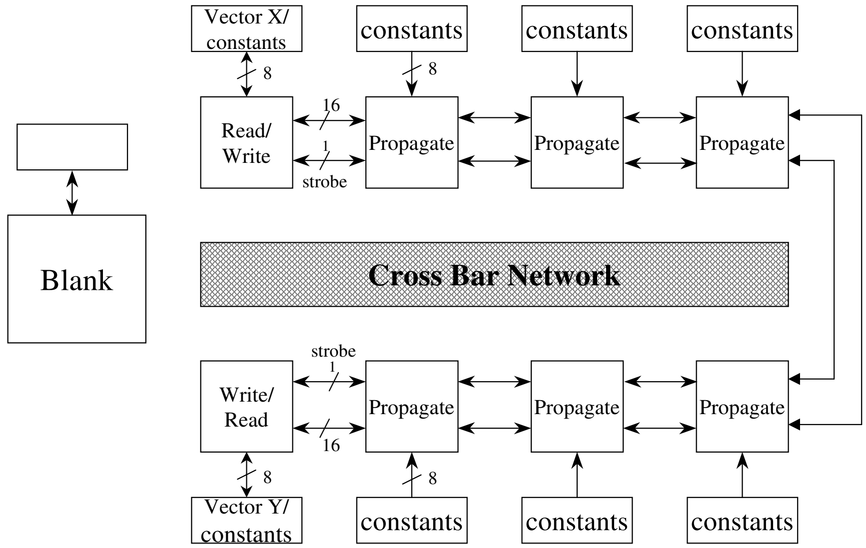


Figure 4.1 Architecture of Filter

Each 1-tap filter unit on the FPGAs is reversible. It can accept input from the left or right systolic bus and output the result on the right or left bus respectively. The input vector is initially stored in the local memory of PE1 and fed into the 1-tap filter on PE1. The partially filtered data then moves on to PE2, then to PE3 and so on in a pipelined manner until it reaches PE8 which writes its result to its local memory. Thus an 8-tap filtering has been performed on the input vector. For higher-order filters, all PEs are reversed and the partially filtered data is read from PE8's local memory, passed through PE7, PE6 and so on until it reaches PE1 where it gets written to PE1's local memory. Thus a 16-tap filtering has been performed. This process of passing data back and forth can be repeated as many times as required to implement filters of any size. PE0 is not used in this design and is configured as a 'blank' FPGA.

4.3.2 One-tap Filter Module

The 1-tap filter is a reversible structure that can process data left-to-right or right-to-left. It consists of a delay unit, three 8-bit multipliers, a 16-bit adder, a 16-bit subtractor and registers to hold the filter coefficients. The filter coefficients for each iteration of the filter are loaded from memory into the registers between iterations. The filter also contains logic to detect and assert strobe signals and multiplexers to route source and result data. A schematic of the 1-tap filter unit is shown below in Figure 4.7. Refer to Appendix B for the RTL description of the 1-tap filter.

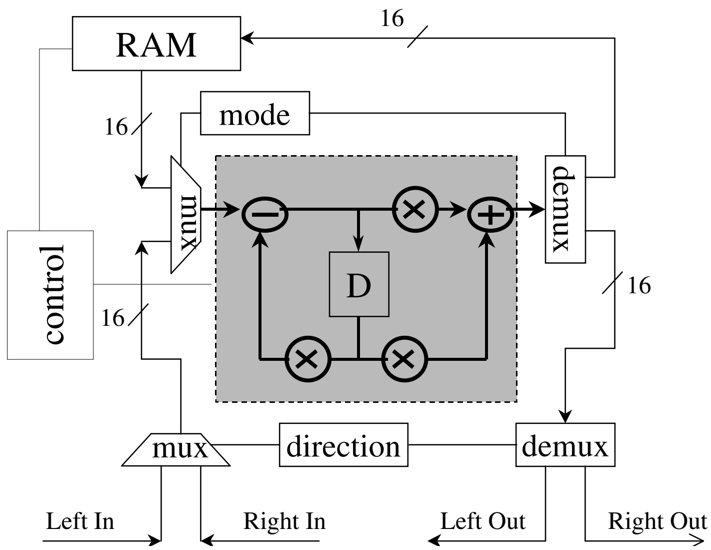


Figure 4.1 Schematic of First Order Filter

Each 1-tap filter could be in one of three different states - Read, Write or Propagate. In addition there is another state bit which determines the direction of flow of data i.e. left-to-right or right-to-left. The initial state of each processor is determined by a 'configuration' byte in the beginning of memory. For the usual case, the direction bit is initially set to left-to-right. PE1 is initially configured as Read, PE2 to PE7 are configured as Propagate and PE8 is configured as Write. After the first iteration, all PEs change their direction bit to right-to-left, PE8 goes into Read mode, PE1 goes to Write mode and all the other PEs remain in Propagate mode. This process of alternating between modes and directions is repeated until all iterations are complete. The result is read back from the local memory of the PE that was last in Write mode. In any iteration, the PE in the Read mode indicates start and end of processing and presence of valid data on its outgoing bus by asserting or de-asserting its strobe line. The PEs in the Propagate and Write mode start processing as soon as their incoming strobe line is detected high and stop processing when it is detected low. They pass the strobe state on to the next PE in the chain.

Resource Requirements and Performance of Filter

The synthesis of the RTL description resulted in a Xilinx netlist that required at least 329 CLBs (83%). The timing goal in synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 10 MHz. The final placed and routed version used 388 CLBs (97%) and could be clocked at a maximum frequency of 8 MHz.

4.4 Host Interface to Filter

The interface to the host is the function wf_filter_interface() which performs the mapping between the host data structures and the structures specific to the FPGA system. This interface function contains within it two separate functions - the function loader and function evaluator. A part of the interface code is shown in Figure 4.8.

```
void wf_filter_interface (argtype src_vector, argtype coeff_a, argtype coeff_b, void **dest)
{
        :
  WF_ARRAY input_vector, output_vector;
  WF_ARRAY coeffa, coeffb;
        :
  /* Set clock frequency */
  clk_freq = FLOAT(8.0);
  WFireClkSetFreq( hBoard, &clk_freq );

  /* Free run clock */
  WFireClkFreeRun( hBoard );

  /* Load configuration */
  wf_filter_load();                              /* Function loader is called */

  /* Re-assign input vector from arg_type */
  input_vector.length = src_vector.diminfo[0].bound;
  input_vector.byte = (unsigned char*) src_vector.data;

  /* Re-assign coefficient data */
  coeffa.length = coeff_a.diminfo[0].bound;
```

```
    coeffa.byte = (unsigned char*) coeff_a.data;
    coeffb.length = coeff_b.diminfo[0].bound;
    coeffb.byte = (unsigned char*) coeff_b.data;

    /* Call function */
    wf_filter (&input_vector, &const_a, &const_b, &output_vector);      /* Function evaluator is
   called */

    /* Assign destination */
    *dest = (void*) output_vector.byte;
}
```

Figure 4.1 Filter Host Interface

The loader configures all FPGAs (except PE0) on the WILDCHILD system with the 1-tap filter configuration and also loads in the default configuration of the crossbar. PE0 is loaded with the 'blank' configuration. A part of the function loader routine is shown below in Figure 4.9.

```
void wf_filter_load ()
{
            :
            :
  /* Program PE0 with blank design */
  WFireProgramPE ( hBoard, WF_PE0_BIT, "pe0blank.m68");

  /* Program all PEXs with filter */
  WFireProgramPE (hBoard, WF_ALL_PE1_16, "pex_filter.m68");

  /* Load XBar configuration */
  WFireXBarConfig ( hBoard, "default.cfg");

  /* Clear PE interrupts */
  WFireClearPEInts ( hBoard, WF_ALL_PEs );

 /* Block memory access to all PEXs  filters */
  WfireSetMemoryMode ( hBoard, WF_ALL_PE1_16, WF_BLOCKED);
}
```

Figure 4.2 FPGA Configuration Loader Function for Filter

The evaluator function is used to download data to the FPGA memories and to actually perform the filter operation. The first eight locations in memory are special purpose locations that contain some configuration information and the sets of filter coefficients. The configuration information includes the initial mode, the iteration count and the last data address. The memory data organization is shown below in Figure 4.10.

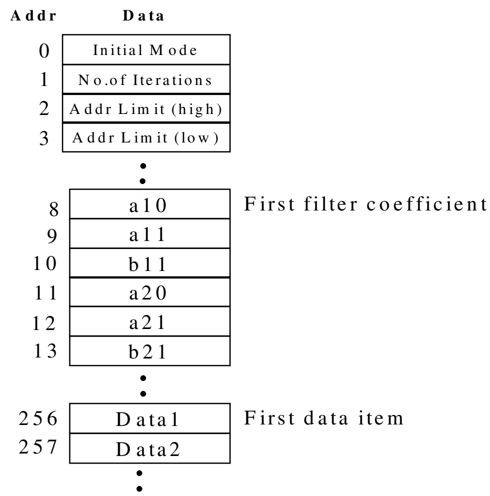


Figure 4.3 (PEX) Filter Memory Data Organization

The evaluator function packs the configuration data, filter coefficients and the input data into a single buffer and performs one write operation to the target FPGA memory. Similarly, after receiving the function completion interrupt, it reads back the FPGA memory, unpacks result data from the data buffer and returns the result in the appropriate data structure. This data structure is then mapped into the compiler's data structure by the interface function. A part of the evaluator routine is shown below in Figure 4.11.

```
void wf_filter (WF_ARRAY *in_vect, WF_ARRAY *const_A, WF_ARRAY *const_B,
WF_ARRAY
                    *out_vect)
{
          :
          :
  num_iterations = const_B->length/8;
  buf[0] = 1*65536 + num_iterations;                /* Mode and num of iterations */

  buf[1] = in_vect->length+256;                     /* Last address */
  byte_loc = (unsigned char*)buf;
  byte_loc += 32;
  factorize_coeff (byte_loc, 1, const_A, const_B);  /* Compute and load coefficients */
  byte_loc += 1024;
  for (j=0; j<in_vect->length; j++) {
    *byte_loc = in_vect->byte[j];
    byte_loc++;
```

```
        }

        /* Write PE1 memory (reader)*/
        WFireWrite ( hBoard, 1, 0, 128+j/4, buf);

        for (i=2; i<=7; i++) {
          buf[0] = 3*65536 + num_iterations;
          factorize_coeff (byte_loc, i, const_A, const_B);
          /* Write PEX (propagate) memory */
          WFireWrite ( hBoard, i, 0, 128, buf);
        }

        buf[0] = 2*65536 + num_iterations;

        /* Write PE8 memory (writer)*/
        WFireWrite ( hBoard, 8, 0, 128, buf);

        /* Give memory access to PEs */
        WfireSetMemoryMode ( hBoard, WF_ALL_PE1_16, WF_HW_ARBITRATE );

        /* Start processors */
        WfireResetPE (hBoard, WF_ALL_PEs, WF_DISABLE);

        /* Wait for PE interrupt */
        do {
          interrupt_vect = WFireGetPEIntEvents (hBoard);
        } while(!(interrupt_vect & result_pe (num_iterations)));

        /* Read back PE memory */
        WfireSetMemoryMode (hBoard, result_pe(num_iterations), WF_BLOCKED);
        WFireRead (hBoard, result_pe(num_iterations), 128, in_vect->length/4, buf);

        /* Assign result */
        byte_loc = (unsigned char*)buf;
        out_vect->length = in_vect->length;
        out_vect->byte = byte_loc;
}
```

Figure 4.4 Evaluator Function for Filter

4.5 Results

The 8-bit fixed-point filter function was tested on various data set sizes and for different filter taps (order). The configuration and compute times are shown in Table 4.1. The filter was configured to run at 8 MHz, which is the maximum speed of the 1-tap filter module. The timing results are also compared against C implementations of the filter equation running on the host processor.

Filter Taps	Vector size	Configuration Time (ms)	Data Transfer Time (ms)	Compute Time (ms)	Compute Time of C code on host (ms)
16	16K	2600	132+15=147	3	483
	64K	2600	188+58=246	13	1936
	256K	2600	440+230=670	52	7764
64	16K	2600	132+15=147	13	1813
	64K	2600	188+58=246	52	7290
	256K	2600	440+230=670	210	29219
256	16K	2600	132+15=147	52	7130
	64K	2600	188+58=246	210	28770
	256K	2600	440+230=670	840	115618

Table 4.1 Performance of Filter Function on WILDCHILD

4.6 Summary

In this chapter we have described the design and implementation of an IIR filter function on the WILDCHILD FPGA system. Some of the common schemes used in building IIR filters like Direct I form, Direct II form and Cascade form were described and the choice of the cascade form for our implementation was justified. We described the architecture of the filter design on the multi-FPGA system. We then described the 1-tap filter module and reported its FPGA resource usage and clock performance. We also described the C application program interface to the filter, which consists of an interface function, which included a function loader and function evaluator. Finally we presented the performance results for computation of various filter data set sizes and different filter orders. In the next chapter we describe the Matrix Multiplier function, its design and implementation.

Chapter 5 Matrix Multiplier Function

In this chapter we describe the design and implementation of a 16-bit fixed point matrix multiplier function on the WILDCHILD. If used with square matrices, this function can multiply two matrices of size up to 500X500 elements at hardware speeds. We first describe the mathematical form of the function and its MATLAB and C language versions. We then look at some of the design alternatives and analyze them for their merits. We then describe our implementation of the different modules of the function along with the host interfaces. Finally we tabulate the execution times of the hardware implementation.

5.1 Function Description

Matrix multiplication is one of the computationally intensive matrix operations and involves $O(n^3)$ complexity. In MATLAB, multiplication of matrices is simply specified by the use of the multiply operator '*'. The MATLAB and C codes required for multiplication of two matrices is shown below in Figure 5.1. The three nested loops in the C code makes this operation $O(n^3)$.

$$A = [a_{11}\ a_{12}\ a_{13}; a_{21}\ a_{22}\ a_{23}];$$
$$B = [b_{11}\ b_{12}; b_{21}\ b_{22}; b_{31}\ b_{32}];$$
$$C = A * B;$$

(a)

```
for (i=0; i<2; i++) {
  for (j=0; j<2; j++) {
    for (k=0; k<3; k++) {
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
    }
  }
}
```

(b)

Figure 5.1 (a) MATLAB code for Matrix Multiplication (b) Corresponding C code

The matrix multiplication operation exhibits a high level of parallelism and this can be exploited using several methods of data partitioning. Some of these methods are attractive for multi-FPGA implementation and others are not. The factors that influence the decision to adopt a particular strategy in a shared or distributed memory parallel processing computer are different from that of a multi-FPGA system. Since the WILDCHILD system is a collection of FPGAs with separate local memories and there is no mechanism to share these memories across the FPGAs, the system is closer to a distributed memory parallel machine than to a shared memory machine. In the next section, we describe some common matrix multiplication schemes and evaluate their suitability for our system.

5.2 Design Alternatives

There are several ways one could implement the multiplication function on a parallel platform. These implementations generally differ in the manner in which data and computation tasks are distributed among the multiple processors, which in turn depend on the architecture of the underlying system and the communication mechanisms. For the case of multi-FPGA systems, there are additional constraints like resource usage and communication bandwidths that affect the decision. We explore some of the alternatives next and evaluate their merits.

5.2.1 Alternative 1: Block Partitioning

The output matrix C is partitioned into N chunks, where N is the number of processors, and each processor performs the operations required to compute its portion of the output matrix. The set of rows of matrix A and columns of matrix B required for the purpose are stored locally in the processor's memory. This means row and column information is duplicated in the processors to minimize communication and increase performance. For example, for a matrix multiplication involving matrix A with 64 rows and matrix B with 32 columns, the output matrix C will have 64 rows 32 columns. If there are 4 processors, each processor computes the following sections of the result matrix C: Processor 1 - rows 1:32 and columns 1:16, processor 2 - rows 1:32 and columns 17:32, processor 3 - rows 33:64 and columns 1:16 and processor 4 - rows 33:64 and columns 17:32. This partitioning scheme is shown below in Figure 5.2.

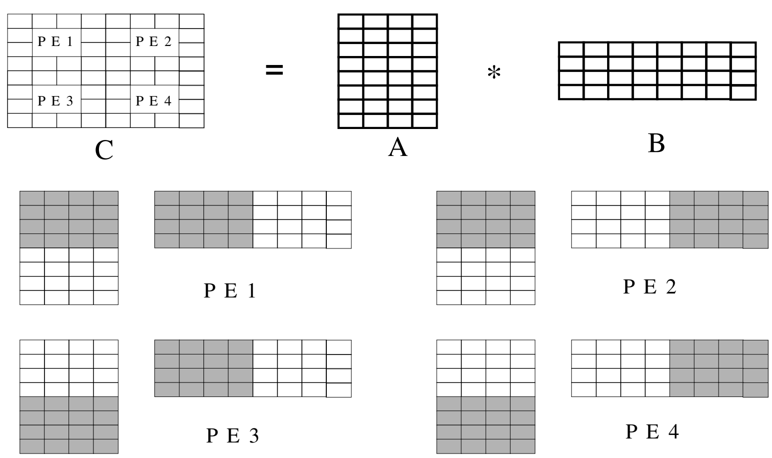


Figure 5.1 Block Partitioning of Result Matrix

5.2.2 Alternative 2: Row-wise Partitioning

The result matrix C is partitioned row-wise (or column-wise) and each processor is responsible for computing its set of rows (or columns). This requires the processor to store only the required rows (or columns) of the input matrix A in its local memory but it will need to store all the columns (or rows) of matrix B unless it can receive it from others through some sharing mechanism.

One way to implement this sharing is to have a 'master' processor containing only data for matrix B to broadcast each of its columns on a broadcast network to all the 'slave' processors. The slaves then multiply the received column with their row data to generate the corresponding entries of the result matrix. This is not usually considered very attractive in distributed computing due to the very high costs involved in message broadcasts. Another way to implement sharing is to use a 'ring' topology where the master sends out a column to the first processor in the ring. This processor uses the column data to compute the corresponding result matrix elements and passes the column on to the next processor in the ring and so on. The 'star' and 'ring' topologies are shown below in Figure 5.3.

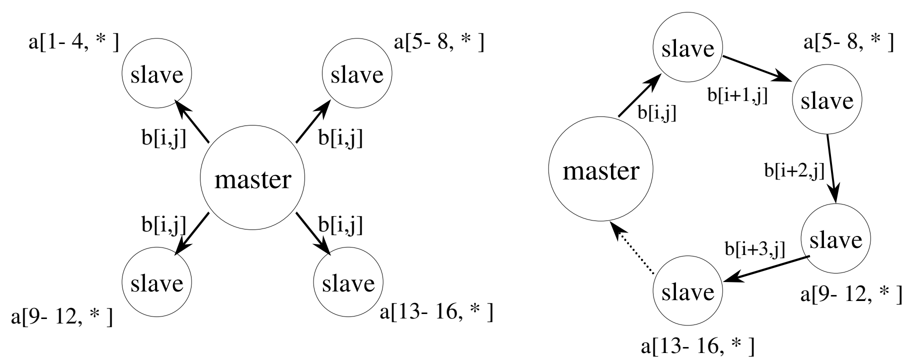


Figure 5.1 Star and Ring Topologies for Communication

For our implementation we chose the star topology for the following reasons: A broadcast in the WILDCHILD system is not an expensive process. All processors can be connected physically to each other using the network of wires that constitute the crossbar. A broadcast involves simply putting data on this mesh of wires and each processor can read it simultaneously during the same clock cycle. Since the interconnection network is only 36 bits wide and the processors can process only one multiply-accumulate operation per clock cycle, the columns of matrix B are broadcast one element every clock cycle. The broadcast based design does not involve any latency but the ring based design involves a latency to 'fill' the pipeline. Also, the logic that is required for the broadcast based design is simpler than that required for a ring topology.

In the star topology, each PEX FPGA acts as a slave processor and the PE0 FPGA acts as the master processor. The input matrix A is row partitioned into as many partitions as there are slaves (max 8). Each slave processor is responsible for computing the corresponding row elements of the output matrix C. Thus, if matrix A has 64 rows and there are 8 slaves, rows 1 through 8 go to slave 1, rows 9 through 16 go to slave 2 and so on. The master processor (PE0) stores the entire input matrix B. The master broadcasts each column of matrix B, element by element during every clock cycle. The slave processor reads the column element from the crossbar, multiplies it with the corresponding row element and accumulates the result in a local register. Thus at the end of *n* clock cycles, where *n* is the number of elements in a column of B, each slave processor has accumulated the result for the respective element of the output matrix C. The accumulated result could be written to the local memories of the slave or they could be passed back to the master to be stored in a contiguous location. The memory utilization efficiency of either method would depend on the actual sizes of the input matrices. But since PE0 memory is twice as big as PEX memory, we predicted that the second method would lead to a better utilization of memory most of the time. The second method involves a more complex design compared to the first one. Since PE0 FPGA can only read from one PEX FPGA at a time, the slaves have to be scheduled to send their accumulated result to the master one after another on the crossbar. This involves each slave having to know its 'id' so that it knows when it is its turn to send its data on the bus to the master. This is achieved by having the configuration information in the beginning of the PEX memories to include the particular slave's id. So at the end of accumulation of a result matrix element, the slave waits on the crossbar tag bits until its id appears on it and then sends out the result

5.3 Implementation of Matrix Multiplier on WILDCHILD

In this section we describe the architecture of the function on the WILDCHILD and the design and implementation of the different components of the function in hardware.

5.3.1 Architecture of Matrix Multiplier on WILDCHILD

Figure 5.4 below shows the architecture of the matrix multiplier on the WILDCHILD FPGA system.

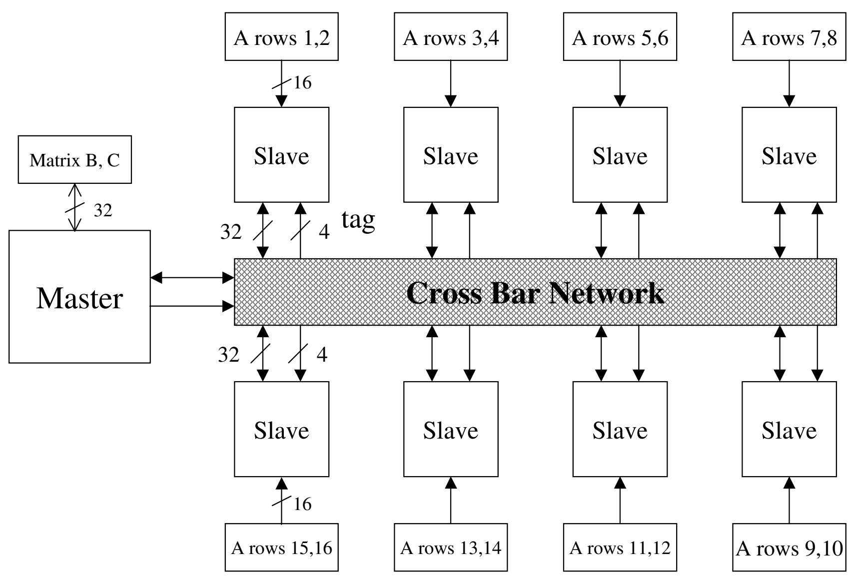


Figure 5.1 Architecture of Matrix Multiplier

5.3.2 Master Module

The master holds the entire matrix B in its memory. During initialization, the slaves read information about the source matrix sizes etc from the master, which broadcasts them on the crossbar. The tag consists of 4 bits with 3 bits used for the slave id and bit 4 used to signal a broadcast. The master broadcasts the column elements of matrix B one every clock cycle. When all the elements in a column have been sent, it sends out the id of the first slave (000) on the tag bits to signal to slave 0 to send out its accumulated result matrix element on the crossbar. It also sets the crossbar configuration so that PE0 (master) is connected to PE1 (slave). The data on the crossbar is written to memory in the next clock cycle. This process is repeated for all the slaves. Refer to Appendix C for the RTL description of the master module. Figure 5.5 shows the master circuit on PE0 FPGA.

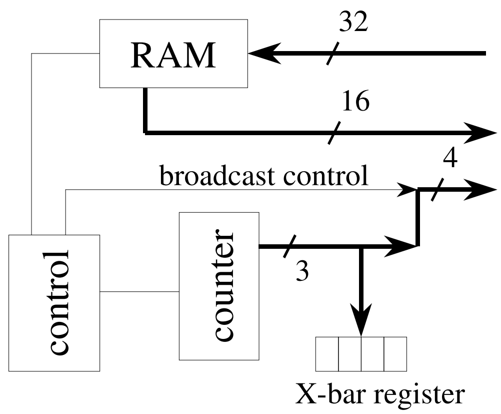


Figure 5.1 Schematic of Master

Resource Requirements and Performance of Master

The synthesis of the RTL description resulted in a Xilinx netlist that required at least 163 CLBs (16%). The timing goal for synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 28 MHz. The final placed and routed version used 192 CLBs (48%) and could be clocked at a maximum frequency of 25 MHz.

5.3.3 Slave Module

Each slave is responsible for computing a section of the output matrix. During initialization, the slave reads its id from the first location in its memory. It then reads information about the size of the source matrices etc from the master, which broadcasts them on the crossbar. The slave then waits for a broadcast signal from the master to begin reading column data from the crossbar. Every clock cycle, the slave reads a row element from its local memory, multiplies it with the column element on the crossbar and accumulates the result in a register. At the end of *n* clock cycles, where *n* is the number of elements in a column of matrix B, an element of the result matrix C has been computed. The slave then continuously reads the crossbar tag waiting for its id to appear on it. When it detects its id on the tag, it sends out the accumulated result on the crossbar. Refer to Appendix C for the RTL description of the slave module. Figure 5.6 shows the slave circuit on PEX FPGA.

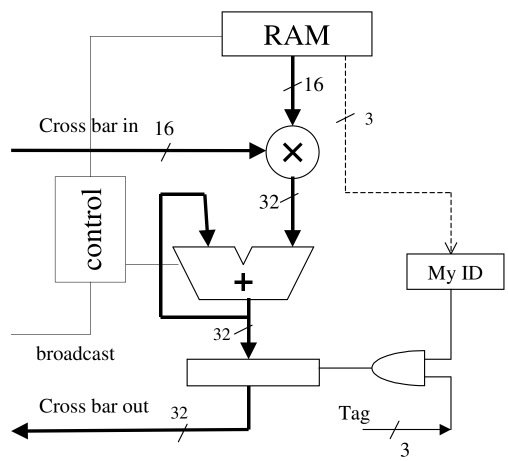


Figure 5.1 Schematic of Slave

Resource Requirements and Performance of Slave

The synthesis of the RTL description resulted in a Xilinx netlist that required at least 192 CLBs (48%). The timing goal in for synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 26 MHz. The final placed and routed version used 215 CLBs (53%) and could be clocked at a maximum frequency of 20 MHz.

5.4 Host Interface to Matrix Multiplier

The interface to the host is the function wf_matmul_interface() which performs the mapping between the host data structures and the structures specific to the FPGA system. This interface function contains within it two separate functions - the function loader and function evaluator. A part of the interface code is shown in Figure 5.7.

```
void wf_matmul_interface (argtype arg_srcA, argtype arg_srcB, void **dest)
{
        :
 WF_MATRIX  matA, matB, matC;
        :
 /* Set clock frequency */
 clk_freq = FLOAT(20.0);
 WFireClkSetFreq( hBoard, &clk_freq );

 /* Free run clock */
 WFireClkFreeRun( hBoard );

 /* Load configuration */
 wf_matmul_load();                               /* Function loader is called */

 /* Obtain row and column information of source matrices */
 num_rowA  = arg_srcA.diminfo[0].bound;
 num_colA   = arg_srcA.diminfo[1].bound;
 num_rowB  = num_colA;
 num_colB   = arg_srcB.diminfo[1].bound;

 /* Fill matrix A */
 matA.num_rows  = num_rowA;
 matA.num_cols   = num_colA;
 matA.data.length = 2*num_rowA*num_colA; /* Assuming 16 bit data, 2 bytes per element */
 matA.data.byte    = (unsigned char*) arg_srcA.data;

 /* Fill matrix B */
 matB.num_rows  = num_rowB;
 matB.num_cols   = num_colB;
 matB.data.length = 2*num_rowB*num_colB; /* Assuming 16 bit data, 2 bytes per element */
 matB.data.byte    = (unsigned char*) arg_srcB.data;

 /* Call function */
 wf_matmul (&matA, &matB, &matC);                /* Function evaluator is called */

 /* Assign destination */
 *dest = (void*)matC.data.byte;
}
```

Figure 5.1 Matrix Multiplier Host Interface

The loader configures the PE0 FPGA with the master circuit and configures each of the PEX FPGAs with the slave circuit. It also loads the crossbar configurations from the file xbar_matmul.cfg. The crossbar configurations allow the master to communicate individually with each of the slaves and also broadcast to all the slaves. A part of the loader code is shown in Figure 5.8.

```
void wf_matmul_load ()
{
        :
        :
  /* Program PE0 with pe0_master */
  WFireProgramPE ( hBoard, WF_PE0_BIT, "pe0_master.m68");

  /* Program all PEXs with pex_slave */
  WFireProgramPE (hBoard, WF_ALL_PE1_16, "pex_slave.m68");

  /* Load XBar configuration */
  WFireXBarConfig ( hBoard, "xbar_matmul.cfg");

  /* Clear PE interrupts */
  WFireClearPEInts ( hBoard, WF_ALL_PEs );

  /* Block memory access to master  PE0*/
  WfireSetMemoryMode ( hBoard, WF_PE0_BIT, WF_BLOCKED);

  /* Block memory access to all PEXs  slaves*/
  WfireSetMemoryMode ( hBoard, WF_ALL_PE1_16, WF_BLOCKED);
}
```

Figure 5.2 FPGA Configuration Loader Function for Matrix Multiplier

The evaluator function is used to download data to the FPGA memories and to actually evaluate the matrix product. The first eight 16-bit locations in PE0 memory are loaded with configuration information which includes the number of slave processors (max 8), the number of columns in matrix B and the rows allocated per slave processor. The first memory location in PEX memories is loaded with the id for the slave processor on the FPGA. The memory data organization is shown below in Figure 5.9.

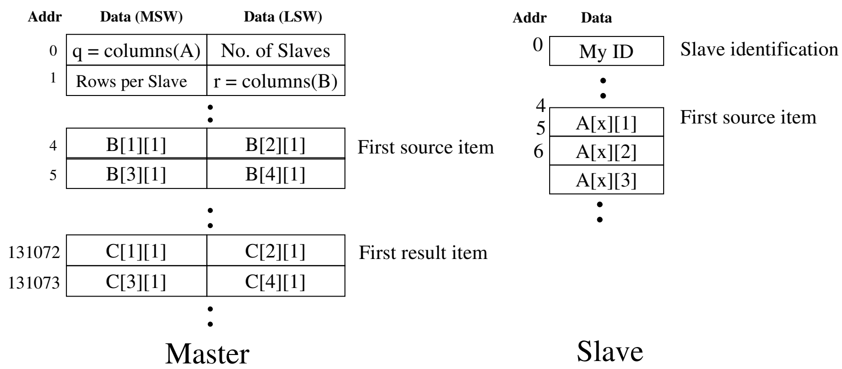


Figure 5.3 PE0 (Master) and PEX (Slave) Memory Data Organization

The evaluator function packs the configuration data and the input data into a single buffer and performs one write operation to the target FPGA memory. Similarly, after receiving the function completion interrupt, it reads back PE0 FPGA memory, unpacks result data from the data buffer and returns the result in the appropriate data structure. This data structure is then mapped into the compiler's data structure by the interface function. A part of the evaluator routine is shown below in Figure 5.10.

```
void wf_matmul (WF_MATRIX *mat_A, WF_MATRIX *mat_B, WF_MATRIX *mat_C)
{
            :
            :
 /* Determine partitioning */
 p     = mat_A->num_rows;
 rem = p%num_slaves;
 if (rem) rows_per_slave = p/num_slaves + 1;
 else rows_per_slave        = p/num_slaves;

 /* Master processor */
 /* Set number of rows, number of columns and rows per slave */
 buf[0]       = q * 65536 + num_slaves;
 buf[1]       = rows_per_slave * 65536 + r;
 byte_loc    = (char*)buf;
 byte_loc   += 16;

 /* Assemble input matrix B array */
 /* - assumed to be in column major order and little end-ian */
 for (i=0; i<mat_B->data.length; i++) {
   *byte_loc = mat_B->data.byte[i];
   byte_loc++;
 }

 /* Write PE0 memory */
 WFireWrite (hBoard, WF_PE0_NUM, 0, mat_B->data.length/4+8, buf);
```

```
    /* Slave processors */
    for (id=0; id<num_slaves; id++) {
      /* Set slave id */
      buf[0]      = id * 65536;
      byte_loc  = (char*)buf;
      byte_loc  += 8;

      /* Assemble row partitions of input matrix A */
      /* - assumed to be in column major order and little end-ian */
      for (i=id; i<p; i+=num_slaves) {
        for (j=0; j<q; j++) {
            *byte_loc = mat_A->data.byte[(j*p+i)*2];
            byte_loc++;
            *byte_loc = mat_A->data.byte[(j*p+i)*2+1];
            byte_loc++;
        }
      }
      /* Write PEX memory */
      WFireWrite ( hBoard, (id+1), 0, (rows_per_slave*q+4)/2+1, buf);
    }

    /* Start processors */
    WfireResetPE (hBoard, WF_ALL_PEs, WF_DISABLE);

    /* Wait for PE0 interrupt */
    do {
      interrupt_vect = WFireGetPEIntEvents (hBoard);
    } while(!(interrupt_vect & WF_PE0_BIT));

    /* Read back PE0 memory */
    WfireSetMemoryMode (hBoard, WF_PE0_BIT, WF_BLOCKED);
    read_count = (rows_per_slave * num_slaves * r)/2+1;
    WFireRead (hBoard, WF_PE0_NUM, RESULT_ADDRESS, read_count, buf);

    /* Assemble result */
    index = 0;
    byte_loc = (char*)buf;
    mat_C->data.byte = (unsigned char*) malloc (p*r*2*sizeof(unsigned char));
    for (j=0; j<r; j++) {
      for (i=0; i<p; i++) {
        mat_C->data.byte[index++] = byte_loc[2*(j*rows_per_slave*num_slaves+i)];
        mat_C->data.byte[index++] = byte_loc[2*(j*rows_per_slave*num_slaves+i)+1];
      }
    };

    /* Assign result */
    mat_C->num_rows = p;
    mat_C->num_cols = r;
    mat_C->data.length = p*r*2;
}
```

Figure 5.4 Evaluator Function for Matrix Multiplier

5.5 Results

The Matrix Multiplier function was tested on various input matrix sizes and the configuration and compute times are shown in Table 5.1. The timing results are also compared against C implementations of the matrix products run on the host processor. The WILDCHILD system clock was set at 20 MHz, which was the maximum clock rate of the slave processor. It is evident from the data that the configuration and data transfer times dominate the evaluation time by several orders of magnitude. While pre-loading of functions could potentially hide the configuration times, the FPGA function time is still largely due to the data transfer times.

Matrix Size (Multiplication of square matrices)	Configuration Time (ms)	Data Transfer Time (Download+Readback) (ms)	Compute Time (ms)	Compute Time of C code on host (ms)
64 X 64	2620	31+7=38	1.95	300
128 X 128	2620	58+24=82	15	2380
248 X 248	2620	155+91=246	103	17702
496 X 496	2620	603+358=961	795	142034

Table 5.1 Performance of Matrix Multiplier Function on WILDCHILD

5.6 Summary

In this chapter we have described the design and implementation of a matrix multiplier function on the WILDCHILD FPGA system. We have described the design options available based on different methods of partitioning matrix data. We described the star and ring topology for master-slave communication and the advantage and disadvantage of each method with respect to implementation. We then described the logic components of the master and slave circuit and presented the FPGA resource usage and performance report for each component. We also described the C application program interface to the matrix multiplier, which consists of a function loader and function evaluator. Finally we presented the performance results for computation of various size of matrix products. In the next chapter we describe the Fast Fourier Transform function, its design and implementation.

Chapter 6 Fast Fourier Transform Function

In this chapter we describe the design of a 16-bit fixed point and the design and implementation of an 8-bit fixed point FFT function on the WILDCHILD. Using this function, FFT for up to 1024 points can be computed at hardware speeds. We first describe the mathematical form of the basic FFT algorithm. We then look at some of the design alternatives and analyze them for their merits. We describe our implementation of the different modules of the function along with the host interfaces. Finally we tabulate the execution times of the hardware implementation.

6.1 Function Description

One of the key processes involved in signal processing is the convolution operation. Convolution is a computationally expensive process taking $O(n^2)$ time where n is the number of samples used in the computation. Hence the signal is transformed using mathematical techniques into a domain where the convolution turns into a less expensive process. One such domain is the frequency domain where convolution is accomplished by multiplication which is O(n). The Discrete Fourier Transform (DFT) is used to transform the signal representation from the time domain to the frequency domain. The Fast Fourier Transform (FFT) is an algorithm that allows quick computation of the DFT when the number of samples is a power of 2.

In MATLAB, the Fast Fourier transform on a vector of input data is computed by using the *fft* function as shown below.

$$Y = fft(X)$$
$$where\ X\ is\ vector.$$
$$Y = fft(X, n)$$
$$returns\ the\ n-POINT\ \ fft\ of\ \ vector\ X$$

The FFT operation exhibits a high level of parallelism during the initial stages of computation but communication between processors becomes high during the final stages. Since communication between processors in the multi-FPGA WILDCHILD system is easily achieved using a mesh of interconnecting wires, there is no extra cost in communication though it does lead to more complex logic on the processors. In the next section, we analyze the FFT operation in more detail and describe the design options for implementing the operation on the WILDCHILD board.

6.1.1 The FFT algorithm:

If $x(n)$, for different values of n, represents the N samples of the input time signal and $X(k)$ represents its N-point DFT, they are related as:

$$X(k) = \sum_{j=0}^{N-1} x(n) \times e^{-\frac{j2\Pi k}{N}} \quad = \quad \sum_{j=0}^{N-1} x(n) \times \omega_N^k$$

Figure 6.1 (a) Basic DFT equation

where $e^{-j2\pi k/N}$ is the complex number $\cos\theta + j\sin\theta$ and $\omega_N^k = e^{-j2\pi k/N}$

By using some transformations this can be written as a weighted sum of two DFTs of half the size i.e. *N/2* point DFTs as:

$$X(k) = G(k) + H(k)\omega_N^k \ \ for\ k = 0\ to\ \frac{N}{2} - 1$$

$$X(k) = G(k) - H(k)\omega_N^k \ \ for\ k = \frac{N}{2}\ to\ N - 1$$

Figure 6.1 (b) DFT decomposition

Doing this process recursively, we can express the complete DFT in terms of 2-point DFTs, which are almost trivial to compute. Thus the FFT is a 'divide-and-conquer' approach to computing the DFT. Notice that the above are equations in complex numbers and therefore involve simultaneous processing of real and imaginary components.

The FFT computation can be represented graphical as shown in Figure 6.2. This structure is called a 'butterfly graph'. Figure 6.2 (a) shows the original graph and Figure 6.2 (b) shows the graph re-ordered to exhibit repetitive functional blocks and the corresponding data flow.

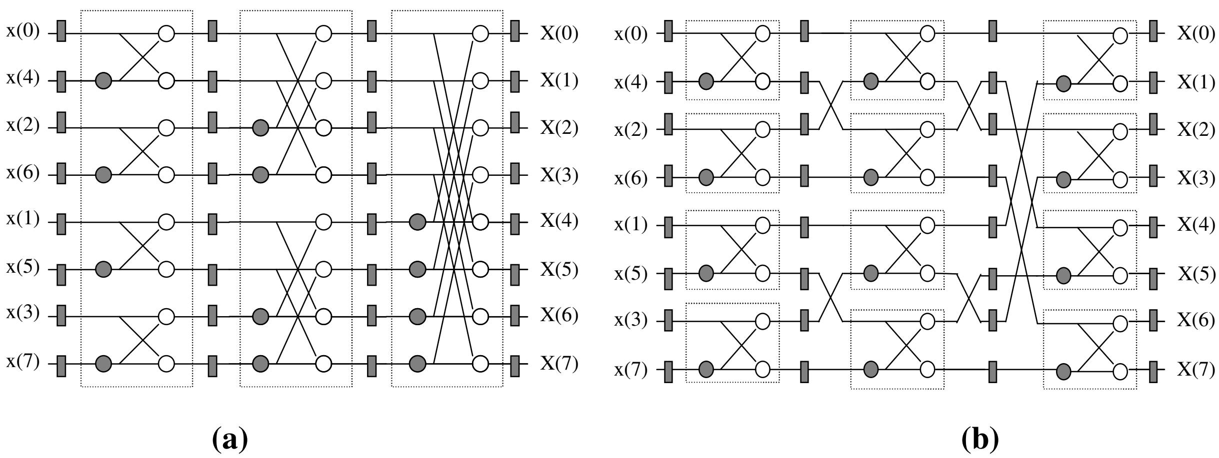


Figure 6.2 Butterfly Graph for Computing FFT (a) and a Re-ordered Graph (b)

An FFT of size *N* has *N * log(N)* computation 'cells'. Each cell is comprised of a complex multiplier (one input is a constant), one complex adder and one complex subtractor. Potentially, each of the constants in the complex multipliers is a distinct non-trivial quantity and the constants are different for different sized FFTs. Each complex multiplication involves 4 real multiplications and 2 real additions. Each complex addition involves 2 real additions.

6.1.2 The Memory Bandwidth Constraint

Looking at Figure 6.2 above, it might be tempting to throw a lot of resources at the butterfly cell to perform all the operations in the lowest number of clock cycles.

Ideally, one would want to parallelize all the multiplications by using 4 multipliers so that the complex multiplication is achieved in one clock cycle. Also, one could use 6 adders to complete the additions in one clock cycle. If the multiplication and addition stages are not pipelined, one could complete the entire butterfly operation in one long clock cycle.
But a closer analysis shows that the operations are constrained by the memory data bandwidth. The PEX memories on the WILDCHILD board are only 16 bits wide. Assuming a 16-bit fixed-point representation for the real and imaginary components of each complex number, it would take at least 4 clock cycles to read the required data for each butterfly from memory. Also, each butterfly operation needs the complex 'twiddle factor' to be loaded from memory. Thus at least 6 clock cycles will be spent reading the data before computation can start. Another 4 clock cycles will be spent writing the result to memory. Also, the memories on the board are non-zero-bus-turn-around. That means one clock cycle is lost in the change of mode between read and write, thus resulting in at least 12 clock cycles per butterfly. So there is no real advantage in having parallel multipliers and adders as they will be idle most of the time waiting for input data to be ready.

6.2 Design Alternatives

The alternative designs for the FFT stem from the different ways one could handle the bandwidth constraint described in Section 6.1.2. In Section 6.2.1 we describe a basic approach that is used for both the alternative designs described in Sections 6.2.2 and Section 6.2.3.

6.2.1 Basic Approach

The following basic strategy was adopted for all designs. The FFT is computed using a set of slave processors controlled by a master processor. Each slave processor operates on its subset of data points until the master signals it to begin communicating with other neighboring processors to exchange data points and results.

To overlap memory accesses with computation, two sets of source and result registers were used. Computation is performed on one set of registers while the result from the previous computation of the other set is written to memory followed by reading of its source registers as shown below in Figure 6.3.

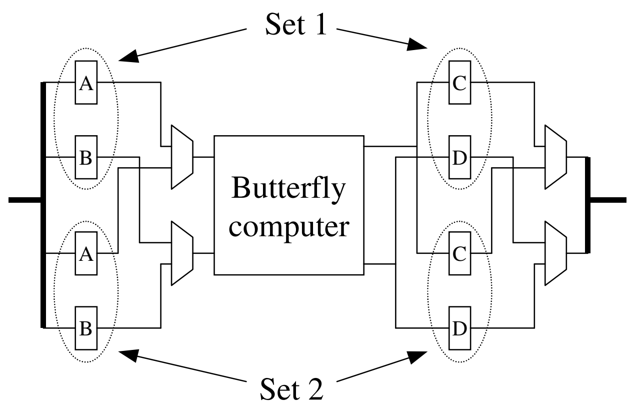


Figure 6.1 Pipelining with Dual Set of Registers

An N-point FFT needs $log(N)$ iterations or 'passes' to compute. Using p processors, the processors operate in parallel on their (N/p) local data points without any need for communication for the first $log(N/p) = log(N) - log(p)$ iterations. In the remaining $(log(N) - log(N/p) = log(p))$ iterations, the processors communicate with increasingly distant processors to exchange source and result data points as shown in Figure 6.4.

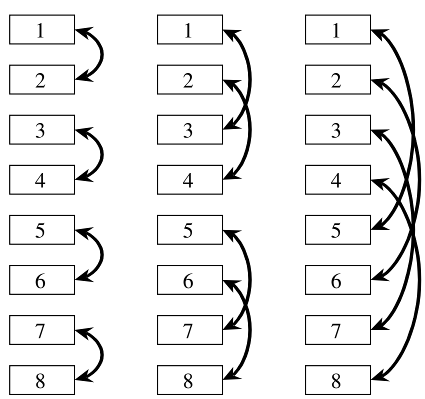


Figure 6.2 Inter-processor Communication Pattern

The data points are exchanged using the crossbar network. The required crossbar configuration for each communication stage is set by the master processor, which also signals the slaves to begin communicating. In each communication between two processors, one of the processors sends the first of the pair of data points it reads from memory and also receives the first data point from its partner during the same clock cycle. The second processor sends the second of the pair of data points it reads from memory and also receives the second data point from its partner in the same clock cycle. This communication is also repeated for the result pair after computation. In the different iterations, a processor exchanges either the first or second data point depending on its id and also the iteration step. For example, processor with id 0 always exchanges its second data point in all iterations while processor with id 7 (in an eight-processor configuration) always exchanges its first data point in all iterations. Processors 1 through 6 exchange the first or second data point depending on its id and the iteration step. This is shown in Figure 6.5.

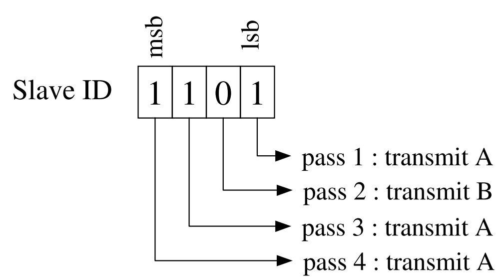


Figure 6.3 Data Point Transmit/Receive Mode Determination

A special bit-slice barrel shifting process generates the address for the pair of data points. A slice (of size determined by the iteration stage) of an ordinary address counter is barrel shifted and concatenated with the remaining counter bits to generate the required address for the read and write pair as shown in Figure 6.6.

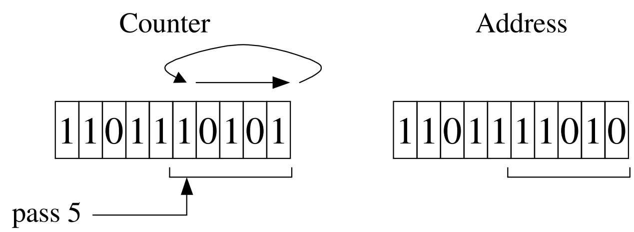


Figure 6.4 Address Generation Using 1-bit Barrel-Shift

6.2.2 Alternative 1: Design without On-chip RAM

In this design, all data will be read and written directly to memory. Using one 16-bit multiplier and one 16-bit adder/subtractor, the entire butterfly computation involving 4 multiplications and 6 additions/subtractions can be completed in a maximum of 10 clock cycles. This is less than the 12 clock cycles needed for data transfer from and to memory. Thus this design uses much less logic compared to the next design. The advantage of using fewer resources on the FPGA is that the place and route tool has more flexibility. This usually results in a better placement and routing meeting higher timing goals and the entire design will have a better clock performance. This design is shown in Figure 6.7.

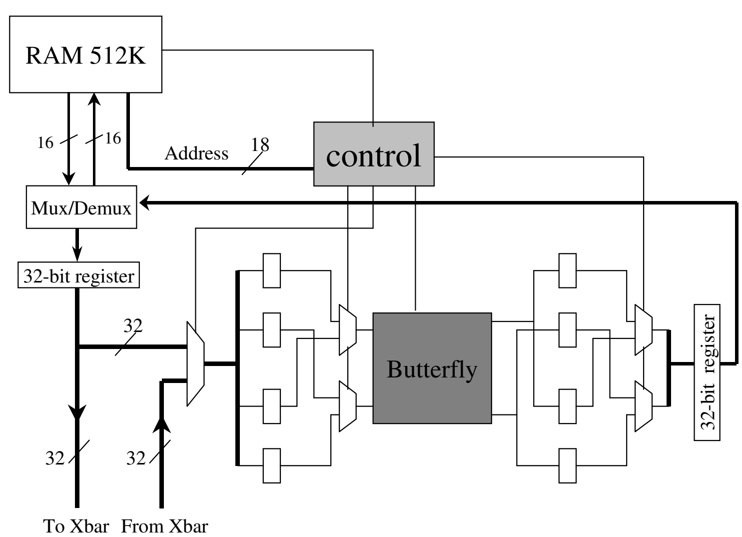


Figure 6.1 FFT Engine without Buffer RAM

6.2.3 Alternative 2: Design with On-chip RAM

An alternative design (an 8-bit version of which was implemented) uses a local on-chip RAM to buffer the data points during processing. Xilinx provides customizable 'cores' for RAM that is implemented on the FPGA using the CLB resources. These RAMs are highly optimized and are fast. At the expense of increased resource requirements, the on-chip RAM provides the following advantages that drastically reduces the computation time for a butterfly and alters the bandwidth limitations that dictated the last design.

The on-chip RAMs are 32-bit wide. Thus an entire 32-bit complex number could be read in one clock cycle. Also, this RAM does not have a bus turn-around penalty for mode changes like the off-chip memory. If the data points are read from this local RAM, the off-chip RAM is free and so could be used to store only the twiddle constants. This means the data points and twiddle constant could be read simultaneously during the same clock cycle. The result of all this is that data required for a butterfly computation can be read in only 2 clock cycles and the result can also be written back in 2 clock cycles! This design is shown in Figure 6.8.

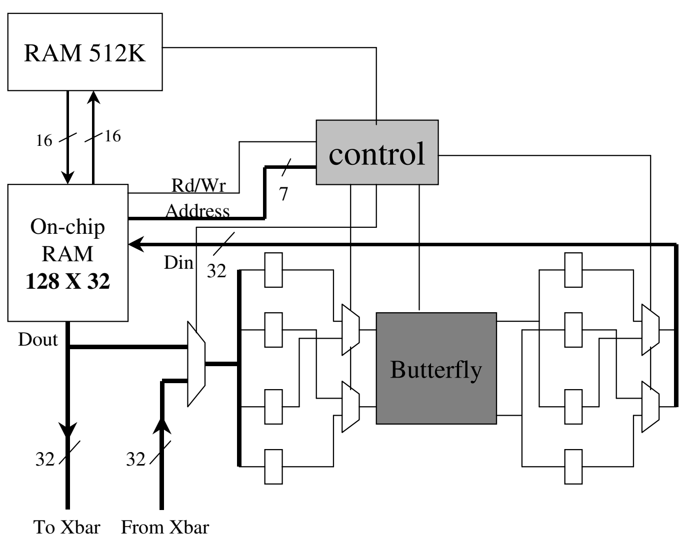


Figure 6.1 FFT Engine with 32-bit Buffer RAM

This gives us an opportunity to put more logic on the FPGA to compute a butterfly in only 4 clock cycles compared to 12 in the previous design using the same two sets of input and result data registers to buffer data as shown in Figure 6.3. This is a good example of the classic area-speed tradeoff. It turns out that we need one multiplier and two adders to achieve a 4-clock cycle butterfly. This will be described in detail in the next section. The on-chip RAM is loaded with data points from the off-chip RAM during initialization and the contents are written back after completion of the FFT. The clock cycles spent in this process are small compared to the savings achieved by including the RAM.

6.3 Implementation of FFT engine on WILDCHILD

The initial design was a 16-bit version where each complex number was a 32-bit number with 16-bit real and imaginary parts. After simulation and verification of the design in a simulator, we realized that the FPGAs were too small to accommodate both the on-chip RAM and the functional units. So we decided to modify the design so that it spanned two FPGAs instead of one. One FPGA entirely contained the on-chip RAM with some communicating logic while the other FPGA contained the butterfly computation units. Also, the design was scaled down from 16 bits to 8 bits i.e. each complex number was a 16-bit number with 8-bit real and imaginary parts. Though the following architecture and module descriptions assume a 16-bit implementation, the final results were obtained for an 8-bit version. The resource usage (after synthesis) for the modules are reported for both versions but post place and route resource and timing information is reported only for the 8-bit version.

6.3.1 Architecture of FFT engine on WILDCHILD

Figure 6.9 below shows the master-slave architecture of the FFT engine on the WILDCHILD system. The slave consists of two modules - a RAM module and a butterfly module. The master module is targeted to the PE0 FPGA while the slave modules are targeted to the PEX FPGAs. The following sections describe the master and slave modules in detail.

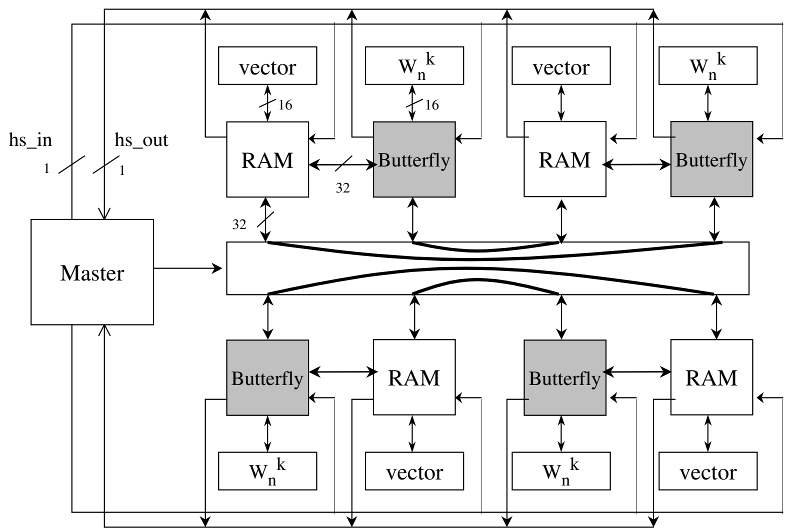


Figure 6.1 Architecture of FFT Computer

6.3.2 Master Module

The master synchronizes the slaves after reset and signals them to begin communication during the later stages of the FFT computation. During initialization, the master reads information about the size of the FFT and the number of required iterations to complete the FFT from its memory. It then waits for handshake signals from all slaves to arrive indicating their reset state. Once all the slaves are reset, it sends a 'start' handshake signal to all slaves to initiate computation. It then sets up counters to sequence through the iterations in step with the slaves. When the stage is reached where inter-processor communication becomes necessary, the master sends another handshake signal to the slaves to initiate communication. It also sets the crossbar configuration register to enable appropriate linkages between the processors for each iteration. When the end of computation is reached, the master alerts the host through its interrupt signal. Refer to Appendix D for the RTL description of the master module. Figure 6.10 shows the master circuit on PE0.

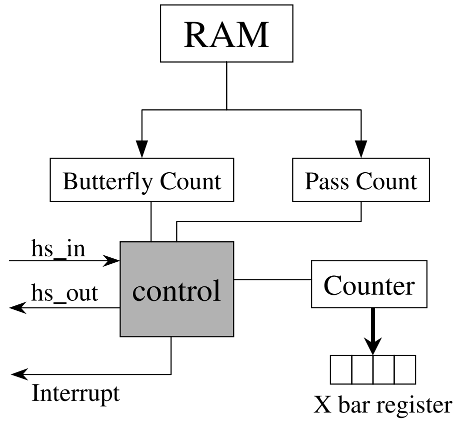


Figure 6.1 Schematic of Master Module

Resource Requirements and Performance of Master

The synthesis of the RTL description resulted in a Xilinx netlist that required at least 36 CLBs (4%). The timing goal for synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 53 MHz. The final placed and routed version used 50 CLBs (4%) and could be clocked at a maximum frequency of 26 MHz.

6.3.3 RAM Module

This module is simply a buffer to store the data points during computation. The module essentially consists of an instance of a Xilinx 'core' memory component. The 32 X 256 synchronous RAM was generated using Xilinx's LogiBlox tool, which generates optimized netlists (and their VHDL models for simulation) for parameterized functions. The RAM is implemented using available resources (CLBs) on the FPGA and is fast. The RAM module also contains addressing logic and logic to interface with the butterfly processor on the adjoining FPGA using the systolic bus and with the crossbar to communicate directly with other butterfly processors. Since the RAM module runs synchronously with the butterfly processor, the address generation logic is local to the RAM module and the data on the data bus of the RAM is directly connected to the right systolic bus and the crossbar port. The RAM read/write and bus direction control signals are also generated locally. During the stages involving communication, the RAM sends and receives data both from the systolic bus and the crossbar. Refer to Appendix D for the RTL description of the RAM module. Figure 6.11 below shows the RAM circuit on the PEX FPGA.

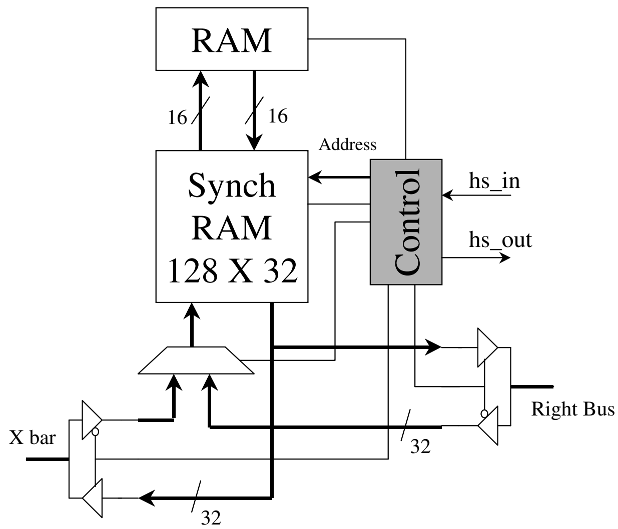


Figure 6.1 Schematic of RAM Module

Resource Requirements and Performance of RAM

As mentioned earlier, though the 16-bit design was simulated, only an 8-bit version was implemented. The synthesis results for both versions and the final (place and route) results for the 8-bit version are reported here.

The synthesis of the RTL description of the 16-bit version (32 X 256 RAM) resulted in a Xilinx netlist that required at least 290 CLBs (73%). The timing goal for synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 30 MHz. The synthesis of the 8-bit version (16 X 256 RAM) resulted in a Xilinx netlist that required at least 118 CLBs (30%). The timing goal for synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 36 MHz. The final placed and routed version used 279 CLBs (69%) and could be clocked at a maximum frequency of 27 MHz.

6.3.4 Butterfly Module

This module computes a butterfly for a pair of data points. The complex butterfly operation involves 4 real multiplications and 6 real additions. Using a 32-bit on-chip RAM, two data points can be read and written in exactly 4 clock cycles. Hence the design goal for the butterfly unit is to exactly match this performance by computing a butterfly in 4 clock cycles so that the memory bus is constantly kept busy. It is also necessary to use as little FPGA resources as possible so that the post place and route clock performance is good. This is achieved using some data buffering and employing parallel functional units as shown in Figure 6.12.

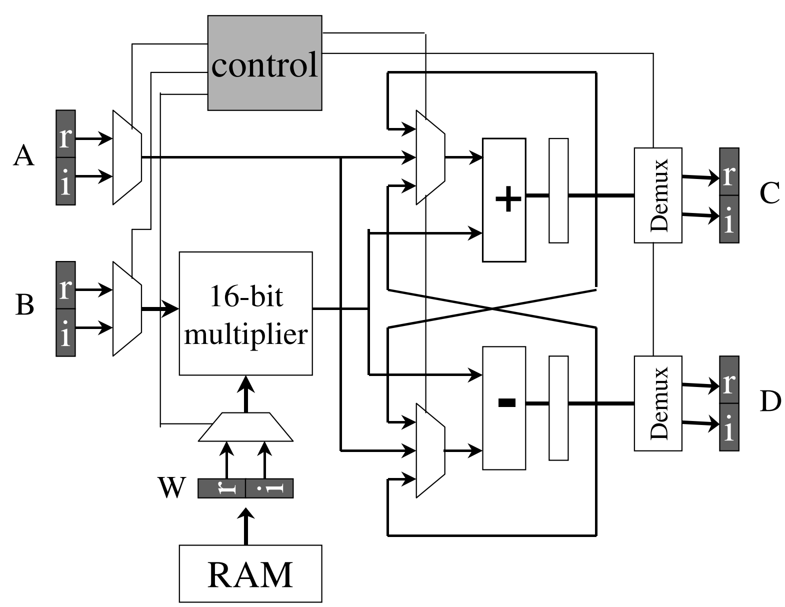


Figure 6.1 Schematic of 4-Clock Butterfly Module

A snap shot of the actions scheduled in each clock cycle is shown below.

:

*Clock (**n-1**) →   Read data point B into set 2*

:

:

*Clock (**n+0**) →* *Write data point C1 from set 1*

:

*Clock (**n+1**) →* *Write data point D1 from set 1*

:

*Clock (**n+2**) →* *Read data point A1 into set 1*

:

*Clock (**n+3**) →* *Read data point B1 into set 1*

Read twiddle constant W1 into set 1

:

*Clock (**n+4**) →* *Write data point C2 from set 2*

Compute $(B1_r * W1_r)$

*Clock (**n+5**) →* *Write data point D2 from set 2*

Compute $(B1_i * W1_i)$

Compute $A1_r + (B1_r * W1_r)$

Compute $A1_r - (B1_r * W1_r)$

*Clock (**n+6**) →* *Read data point A2 into set 2*

Compute $(B1_i * W1_r)$

Compute $C1_r = A1_r + (B1_r * W1_r) - (B1_i * W1_i)$

Compute $D1_r = A1_r - (B1_r * W1_r) + (B1_i * W1_i)$

*Clock (**n+7**) →* *Read data point B into set 2*

Compute $(B1_r * W1_i)$

Compute $A1_i + (B1_i * W1_r)$

Compute $A1_i - (B1_i * W1_r)$

*Clock (**n+8**) →* *Write data point C from set 1*

Compute $(B2_r * W2_r)$

Compute $C1_i = A1_i + (B1_i * W1_r) + (B1_r * W1_i)$

Compute $D1_i = A1_i - (B1_i * W1_r) - (B1_r * W1_i)$

*Clock (**n+9**) →* *Write data point D from set 1*

:

:

Refer to Appendix D for the RTL description of the butterfly module.

Resource Requirements and Performance of Butterfly

As mentioned earlier, though the 16-bit design was simulated, only an 8-bit version was implemented. The synthesis results for both versions and the final (place and route) results for the 8-bit version are reported here.

The synthesis of the RTL description of the 16-bit version resulted in a Xilinx netlist that required at least 373 CLBs (94%). The timing goal for synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 14 MHz. The synthesis of the 8-bit version resulted in a Xilinx netlist that required at least 273 CLBs (69%). The timing goal for synthesis was set at 100 MHz. The resultant circuit synthesized could potentially be clocked at a maximum frequency of 17 MHz. The final placed and routed version used 336 CLBs (84%) and could be clocked at a maximum frequency of 10 MHz.

6.4 Host Interface of FFT engine

The interface to the host is the function wf_fft_interface() which performs the mapping between the host data structures and the structures specific to the FPGA system. This interface function contains within it two separate functions - the function loader and function evaluator. A part of the interface code is shown in Figure 6.13.

```
void wf_fft_interface (argtype src_vector, void **dest)
{
          :
  WF_ARRAY input_vector, output_vector;
          :
  /* Set clock frequency */
  clk_freq = FLOAT(8.0);
  WFireClkSetFreq( hBoard, &clk_freq );

  /* Free run clock */
  WFireClkFreeRun( hBoard );

  /* Load configuration */
  wf_fft_load();                                        /* Function loader is
called */

  /* Re-assign input vector from arg_type */
  input_vector.length = src_vector.diminfo[0].bound;
  input_vector.byte = (unsigned char*) src_vector.data;
```

```
 /* Call function */
 wf_fft (&input_vector, &output_vector);                              /* Function evaluator is
called */

 /* Assign destination */
 *dest = (void*) output_vector.byte;
}
```

Figure 6.1 FFT Host Interface

The loader configures the PE0 FPGA with the master circuit, configures PE1, PE3, PE5 and PE7 FPGAs with the RAM circuit and PE2, PE4, PE6 and PE8 with the butterfly circuit. It also loads the crossbar configurations from the file xbar_fft.cfg. The crossbar configurations specify the necessary linkages between the slave butterfly units and the RAM units during different stages of the FFT. Since there are a total of 4 butterfly processors, only the last two passes of the FFT involve communication. For an *N*-point FFT, this would be iterations [$log(N)-1$] and [$log(N)$]. During iteration [$log(N)-1$], RAM-0 (on PE1) is connected to Butterfly-1 (on PE4) and RAM-1 (on PE3) is connected to Butterfly-0 (on PE2). Similarly RAM-2 (on PE5) is connected to Butterfly-3 (on PE8) and RAM-3 (on PE7) is connected to Butterfly-2 (on PE6). During iteration [$log(N)$], RAM-0 (on PE1) is connected to Butterfly-2 (on PE6) and RAM-1 (on PE3) is connected to Butterfly-3 (on PE8). Similarly RAM-2 (on PE5) is connected to Butterfly-0 (on PE8) and RAM-3 (on PE7) is connected to Butterfly-1 (on PE4). This is shown in Figure 6.14.

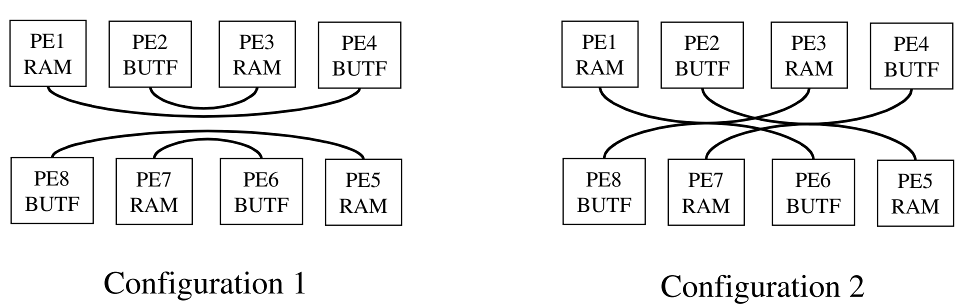


Figure 6.2 Crossbar Connections for Last Two Passes of FFT

A part of the loader code is shown in Figure 6.15 below.

```
void wf_fft_load ()
{
        :
        :
 /* Program PE0 with pe0_master */
 WFireProgramPE ( hBoard, WF_PE0_BIT, "pe0_master.m68");

 /* Program all PEXs with pex_butterfly */
 WFireProgramPE (hBoard, WF_ALL_PE1_16, "pex_butterfly.m68");

 /* Re-program odd PEXs with pex_ram */
 for(i=1; i<=7; i+=2) {
   WFireProgramPE (hBoard, 1<<i, "pex_ram.m68");
 }

 /* Load XBar configuration */
 WFireXBarConfig ( hBoard, "xbar_fft.cfg");

 /* Clear PE interrupts */
 WFireClearPEInts ( hBoard, WF_ALL_PEs );

 /* Block memory access to all PEXs */
 WFireSetMemoryMode( hBoard, WF_ALL_PE1_16, WF_BLOCKED);
}
```

Figure 6.3 FPGA Configuration Loader Function for FFT

The evaluator function is used to download data to the FPGA memories and to actually evaluate the FFT. The first eight 16-bit locations in the PE memories are loaded with configuration information which includes the processor id (for slaves), the number of data points and the number of passes (iterations = *log(N)*). The memory data organization is shown below in Figure 6.16.

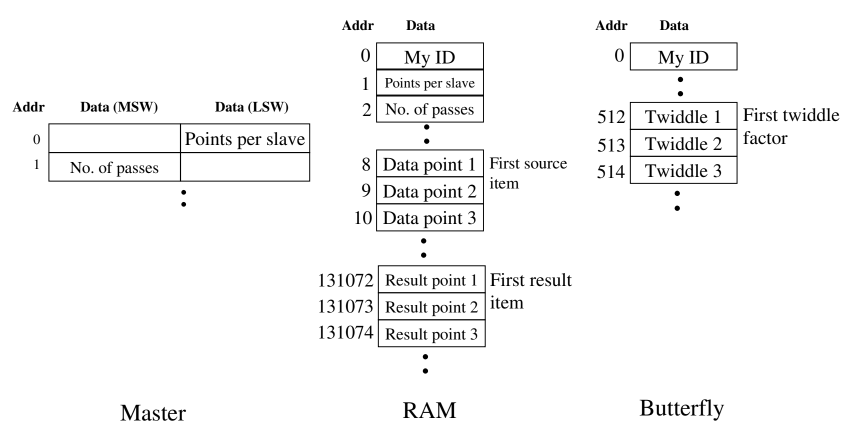


Figure 6.4 PE0 (Master) and PEX (RAM/Butterfly) Memory Data Organization

The evaluator function packs the configuration data and the input data into a single buffer and performs one write operation to the target FPGA memory. Note that only the PEX memories that hold the RAM module need to be loaded with sections of the input vector data. After receiving the function completion interrupt from PE0, it reads back the PEX FPGA (with RAM) memories, unpacks result data from the data buffers and returns the result in the appropriate data structure. This data structure is then mapped into the compiler's data structure by the interface function. A part of the evaluator routine is shown below in Figure 6.17

```
void wf_fft (WF_ARRAY *in_data, WF_ARRAY *out_data)
{
            :
            :
  size = in_vect->length/2;
  points_per_slave = size/4;          /* Size should be a power of 2 */
  num_of_iterations = log2(size);

  /* Write PE0 memory (master) */
  buf[0] = points_per_slave;
  buf[1] = num_of_iterations * 65536;
  WFireWrite ( hBoard, WF_PE0_NUM, 0, 10, buf );

  /* Set memory mode for PE0 */
  WFireSetMemoryMode( hBoard, WF_PE0_BIT, WF_HW_ARBITRATE );

  /* Write PEX memories (RAM and butterfly) */
  for(i=1; i<=8; i++) {
    id = ((i+1)/2-1);
    /* RAM's and butterfly's id are the same - range from 0 to 3 */
    buf[0] = id*65536+points_per_slave;
    byte_loc = (unsigned char*)buf;
    /* Two bytes per data point */
    if (i mod 2) { /* Only for odd PEXs (local ram) */
      byte_loc += 16; // Points to buf[4]
      /* Assign data points */
      for (j=2*id*points_per_slave; j<2*(id+1)*points_per_slave; j++) {
            *byte_loc = in_data->byte[j];
            byte_loc++;
      }
    }
    else { /* Only for even PEXs (butterflies) */
      byte_loc += 1024;
      twiddle_factors (byte_loc, size, id);       /* Computes and loads twiddle factors */
    }
    WFireWrite ( hBoard, i, 0, 131072, buf);
    WFireSetMemoryMode( hBoard, 1<<i, WF_HW_ARBITRATE );
  }
```

```
/* Give memory access to PEs */
WFireSetMemoryMode( hBoard, WF_ALL_PE1_16, WF_HW_ARBITRATE );
/* Start processors */
WfireResetPE (hBoard, WF_ALL_PEs, WF_DISABLE);
/* Wait for master interrupt */
do {
  interrupt_vect = WFireGetPEIntEvents (hBoard);
} while(!(interrupt_vect & WF_PE0_BIT));

/* Read back and assemble result */
out_data->length = in_data->length;
for(i=1; i<=8; i+=2) {
  id = ((i+1)/2-1);
  WFireSetMemoryMode( hBoard, 1<<i, WF_BLOCKED );
  WFireRead (hBoard, i, 0, 131072, buf);
  byte_loc = (unsigned char*)buf;
  byte_loc += 262144;
  for (j=2*id*points_per_slave; j<2*(id+1)*points_per_slave; j++) {
    out_data->byte[j] = *byte_loc;
    byte_loc++;
  }
}
}
```

Figure 6.5 Evaluator Function for FFT

6.5 Results

The 8-bit FFT function was tested on various input vector sizes and the configuration and compute times are shown in Table 6.1. The clock speed was 8 MHz. It is evident from the data that the configuration and data transfer times dominate the evaluation time by several orders of magnitude. While pre-loading of functions could potentially hide the configuration times, the FPGA function time is still largely due to the data transfer times.

FFT size	Configuration Time (ms)	Data Transfer Time (ms)	Compute Time (us)	Compute Time of C code on host (us)
128	3656	105+46=151	50	510
256	3656	106+47=153	130	1111
512	3656	107+48=155	290	2528
1024	3656	109+48=157	640	5624

Table 6.1 Performance of FFT Function on WILDCHILD

6.6 Summary

In this chapter we described design and implementation of the Fast Fourier Transform. We first described the basic algorithm and the decomposition of its data-flow graph into computation cells. We then described the memory bandwidth constraint for a 16-bit FFT computation, which made it necessary to explore the need for an on-chip RAM to increase the bandwidth. Both designs - with and without on-chip RAM - were analyzed in terms of the number of clock cycles required for computation and the resource requirements. We then described the version with on-chip RAM in detail. We presented the architecture of the design on the WILDCHILD board with the butterfly and RAM on separate FPGAs and described the separate modules along the mechanisms for inter-processor communication. Due to resource constraints we were forced to implement only an 8-bit version of the design. We then described the functions interface to the host program. Finally we reported performance results of the implementation in comparison to implementations in C and MATLAB.

Chapter 7 Conclusion

In this thesis, the design and implementation of a set of library functions for matrix and signal processing operations and their interfaces to a MATLAB compiler was presented. The compiler, the focus of the MATCH project, compiles a source program in MATLAB into parallel code that can be executed simultaneously on a collection of COTS hardware like FPGAs, DSPs, and embedded processors that form an experimental heterogeneous testbed.

The FPGA component of the system comprised of the WILDCHILD system, a multi-FPGA reconfigurable computer with 9 Xilinx FPGAs on a single VME-compatible board. We described the architecture and host interfaces of this system in detail. We then described the process involved in the design of hardware functions on this reconfigurable computer and presented the VHDL coding style and methodology that we adopted for our designs.

We described in detail our implementations of the library functions for matrix and signal processing operations - the Matrix Adder, Filter, Matrix Multiplier and Fast Fourier Transform. For each function, we presented the design of its WILDCHILD-based architecture and described the hardware implementations of its modules and also its interface to the MATCH compiler. Finally, for each function, we presented the execution times that reflected the performance gains of the function in hardware compared to its traditional implementations.

Bibliography

[1] The MATCH Project
http://www.ece.nwu.edu/cpdc/Match/Match.html

[2] The WILDCHILD Custom Computing Engine **http://www.annapmicro.com/wfhtm.html**

[3] "Splash 2: FPGAs in a Custom Computing Machine" by Duncan A. Buell, Jeffrey M. Arnold, and Walter J. Kleinfelder
http://www.computer.org/cspress/catalog/bp07413/spls-tc.htm

[4] "Baring it all to Software: Raw Machines" by Elliot Waingold et al (*IEEE Computer*, September 1997, pp. 86-93.)
http://www.cag.lcs.mit.edu/raw/

[5] The BRASS Research Group
http://http.cs.berkeley.edu/projects/brass/

[6] "Garp: A MIPS Processor with a Reconfigurable Coprocessor" by John R. Hauser and John Wawrzynek. (FCCM April 1997).
http://http.cs.berkeley.edu/projects/brass/garp.html

[7] The Berkeley Intelligent RAM (IRAM) Project
http://iram.cs.berkeley.edu/

[8] "The Chimaera Reconfigurable Functional Unit" by S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao (FCCM 1997 , pp. 87-96).
http://www.ece.nwu.edu/~hauck/chimaera.html

[9] "The Cameron Project: High-Level Programming of Image Processing Applications on Reconfigurable Computing Machines" by W. Najjar et al (PACT Workshop on Reconfigurable Computing, October 1998)
http://www.cs.colostate.edu/cameron/

[10] WASPP: WILDFORCE(tm)-based Adaptive Signal Processing Project
http://www.annapmicro.com/WASSP.html

[11] CHAMPION: A Software Design Environment for Adaptive Computing Systems
http://microsys6.engr.utk.edu/~bouldin/darpa/

[12] "Essential VHDL: RTL Synthesis Done Right" by Sundar Rajan
http://www.clbooks.com/specials/vhdl/

[13] "The Designer's Guide to VHDL" by Peter J.Ashenden, Morgan Kaufmann

[14] "Digital Signal Processing in VLSI" by Richard J.Higgins, Prentice Hall

[15] "Digital Signal Processing with the TMS320C25" by Chassaing, Wiley-Interscience

[16] "HDL Chip Design" by Smith, Doone Publications