NORTHEASTERN UNIVERSITY

DESIGN AND EVALUATION OF MATLAB FUNCTIONS ON FIELD PROGRAMMABLE GATE ARRAYS (FPGAs)

THESIS

SUBMITTED TO THE GRADUATE SCHOOL IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Electrical and Computer Engineering

By

David C. Zaretsky

EVANSTON, ILLINOIS
June 2001
ABSTRACT

The use of Field Programmable Gate Arrays (FPGAs) in signal processing and related applications has become increasingly popular, as they achieve higher performance results over general-purpose processors. Additionally, the FPGAs can be used in parallel architectures allowing even greater performance improvements. However, application design for these systems is a long and complicated process. The objective of MATCH is to build a compiler to alleviate the arduous task of developing efficient code for systems such as FPGAs, DSPs and embedded processors, by automatically translating high-level applications written in MATLAB and mapping them to these systems. Several benchmark evaluations were performed on two platforms of the Match testbed, the Annapolis WILDCHILD and WILDSTAR multi-FPGA systems. The Wavelet Transform algorithm demonstrated that there exists the possibility to obtain performance improvements in orders of magnitude between a general-purpose processor and special purpose hardware. Single processor and parallel processor implementations of the Matrix Multiplication algorithm were designed for the WILDSTAR board and compared to that of the WILDCHILD and several other platforms of the MATCH testbed. The new compiler interface for the WILDSTAR system was evaluated by comparing compiler-generated Matrix Multiplication designs in MATLAB with the hand-coded designs. Finally, several benchmark designs written in MATLAB were compiled to VHDL and tested on the WILDCHILD and WILDSTAR boards, and performance results were compared. Conclusions show that although the WILDSTAR system has an advanced architecture, its performance is nevertheless limited by its lack in ability to pipeline data.
ACKNOWLEDGEMENTS

I wish to thank my advisor, Professor Prithviraj Banerjee, for allowing me the opportunity to work under his guidance on the MATCH project. His support and encouragement has led to the accomplishments of my work as described in this thesis. I also wish to thank Professor Alok Choudhary and Professor Yehea Ismail for participating on the examination committee.

To my colleagues, especially those who are part of the MATCH project team, I wish to thank you for all your helpful insights in the different aspects of the project that were invaluable to the successful completion of this thesis.

Finally, I wish to thank my parents for teaching me… the sky is the limit!
CONTENTS

CHAPTER 1: INTRODUCTION ................................................................................................. 1
  1.1 MOTIVATION ...................................................................................................................... 1
  1.2 MATCH .................................................................................................................................. 2
  1.3 MATCH TESTBED ............................................................................................................. 3
  1.4 THE MATCH COMPILER .................................................................................................... 4
  1.5 RELATED WORK ................................................................................................................. 5
  1.6 THESIS OUTLINE ............................................................................................................. 6

CHAPTER 2: THE WILDCHELDR AND WILDSSTAR SYSTEMS............................................ 8
  2.1 THE WILDCHELDR MULTI-FPGA SYSTEM ........................................................................ 8
  2.1.1 HARDWARE ARCHITECTURE .......................................................................................... 9
  2.1.2 SOFTWARE ................................................................................................................... 11
  2.1.2.1 HOST INTERFACE ......................................................................................................... 11
  2.1.2.2 VHDL MODELING FOR WILDCHELDR FPGAS ............................................................ 12
  2.2 THE WILDSSTAR MULTI-FPGA SYSTEM ........................................................................ 15
  2.2.1 HARDWARE ARCHITECTURE ........................................................................................ 15
  2.2.2 SOFTWARE ................................................................................................................... 18
  2.2.2.1 HOST INTERFACE ......................................................................................................... 18
  2.2.2.2 VHDL MODELING FOR WILDSSTAR FPGAS .............................................................. 19
  2.3 WILDCHELDR VS. WILDSSTAR ..................................................................................... 23
  2.4 DESIGN PROCESS .............................................................................................................. 24
  2.5 SUMMARY ........................................................................................................................ 26

CHAPTER 3: WAVELET TRANSFORM................................................................................ 27
  3.1 OVERVIEW OF THE WAVELET TRANSFORM ...................................................................... 27
  3.2 WAVELET TRANSFORM ALGORITHM IN C LANGUAGE ..................................................... 28
  3.3 WAVELET TRANSFORM ALGORITHM IN MATLAB .......................................................... 31
  3.4 IMPLEMENTATION OF THE WAVELET TRANSFORM ON THE WILDCHELDR SYSTEM..... 32
  3.5 RESULTS ........................................................................................................................... 35
  3.6 SUMMARY ........................................................................................................................ 36

CHAPTER 4: MATRIX MULTIPLICATION ......................................................................... 37
  4.1 FUNCTION DESCRIPTION ................................................................................................... 37
  4.2 MATRIX MULTIPLICATION ON THE WILDCHELDR SYSTEM ....................................... 38
  4.2.1 MATRIX MULTIPLICATION USING A SINGLE PROCESSOR ........................................ 38
  4.2.1.1 ARCHITECTURE FOR THE SINGLE PROCESSOR ......................................................... 39
  4.2.1.2 PERFORMANCE ............................................................................................................ 40
  4.2.2 MATRIX MULTIPLICATION USING PARALLEL PROCESSORS ..................................... 41
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.3</td>
<td>Motion Estimation</td>
<td>219</td>
</tr>
<tr>
<td>E.4</td>
<td>128x128 Matrix Multiplication</td>
<td>221</td>
</tr>
<tr>
<td>E.5</td>
<td>256x256 Matrix Multiplication</td>
<td>221</td>
</tr>
<tr>
<td>E.6</td>
<td>Sobel Transform</td>
<td>222</td>
</tr>
<tr>
<td>E.7</td>
<td>Vectorsum</td>
<td>223</td>
</tr>
</tbody>
</table>

REFERENCES ..................................................................................................................... 224
FIGURES

FIGURE 1.1: ... THE DEVELOPMENT PROCESS OF CODE FOR A HETEROGENEOUS COMPUTING-ENGINE USING THE MATCH COMPILER ................................................................. 2
FIGURE 1.2: THE MATCH TESTBED .............................................................................................. 3
FIGURE 1.3: THE MATCH COMPILER .................................................................................................. 5
FIGURE 2.1: THE WILDCHILD SYSTEM ARCHITECTURE ........................................................................... 9
FIGURE 2.2: CROSSBAR CONFIGURATION SPECIFICATION ................................................................. 10
FIGURE 2.3: A SAMPLE PE0 ENTITY COMMONLY USED FOR FPGA DESIGNS ON THE WILDCHILD SYSTEM ......................................................................................................................... 12
FIGURE 2.4: A STATE MACHINE IMPLEMENTATION OF A FOR-LOOP IN WHICH DATA IS READ FROM THE LOCAL MEMORY ............................................................................................................ 13
FIGURE 2.5: A SAMPLE CORE ARCHITECTURE FOR PE0 THAT INCORPORATES A USER’S FPGA DESIGN AS A COMPONENT, AND THEN CONNECTS THE SIGNALS TO THE FPGA INTERFACES IN A PROCESS NAMED “MASTER_PROCESS” ........................................................................................................ 14
FIGURE 2.6: THE WILDSTAR SYSTEM ARCHITECTURE ............................................................................... 15
FIGURE 2.7: FOUR CROSSBAR CONFIGURATIONS FOR THE MEZZANINE CARDS, “00” PEX ACCESSES MEM 0, PE0 ACCESSES MEM 1; “01” PEX ACCESSES MEM 1, PE0 ACCESSES MEM 0; “10” PEX ADDR/CTRL & LOW DATA SHUNTED TO PE0; “11” PEX ADDR/CTRL & HIGH DATA SHUNTED TO PE0 .............................................................................................................................................................................. 16
FIGURE 2.8: INTERRUPT PROCESS USED IN WILDSTAR DESIGNS, WHERE THE HOST READS THE INTERRUPT FROM A REGISTER AT ADDRESS INTERRUPT_BASE ................................................................................................. 17
FIGURE 2.9: A SAMPLE PEX ENTITY COMMONLY USED FOR FPGA DESIGNS ON THE WILDSTAR SYSTEM ................................................................................................................................................................ 20
FIGURE 2.10: A STATE MACHINE IMPLEMENTATION OF A FOR-LOOP IN WHICH DATA IS READ FROM THE PEX’S LEFT LOCAL MEMORY ........................................................................................................ 21
FIGURE 2.11: A SAMPLE CORE ARCHITECTURE FOR PEX THAT INCORPORATES A USER’S FPGA DESIGN AS A COMPONENT, AND THEN CONNECTS THE SIGNALS TO THE FPGA INTERFACES IN A PROCESS NAMED “PEX_INSTANCE” ......................................................................................................................................................................................... 22
FIGURE 2.12: FPGA DESIGN PROCESS ........................................................................................................ 25
FIGURE 3.1: THE FILTER OPERATIONS OF THE WAVELET TRANSFORM ......................................................... 27
FIGURE 3.2: FULL IMPLEMENTATION OF THE WAVELET TRANSFORM ON A 512X512 IMAGE .......................................................................................................................... 28
FIGURE 3.3: THE MAIN FUNCTION OF THE WAVELET TRANSFORM IN C LANGUAGE..................................................................................................................................................................................... 29
FIGURE 3.4: THE $f_{cdf22}$ FUNCTION OF THE WAVELET TRANSFORM WHICH PERFORMS THE HIGH-PASS AND LOW-PASS FILTERS ........................................................................................................... 29
FIGURE 3.5: THE REDESIGNED $f_{cdf22}$ FUNCTION BY MEANS OF LOOP-FUSION ........................................ 30
FIGURE 3.6: (A) C CODE FOR THE $f_{cdf22}$ FUNCTION. (B) CORRESPONDING MATLAB CODE ........................................................................................................................................................................................................ 31
FIGURE 3.7: PSEUDO VHDL MODEL OF THE $fwd\_wavelet$ FUNCTION .................................................. 32
FIGURE 3.8: MEMORY READ AND WRITE FOR ROW OPERATIONS (ST = 1) .................................................................................................................................................................................................................. 33
FIGURE 3.9: MEMORY READ AND WRITE FOR COLUMN OPERATIONS (ST = 512) ........................................ 34
FIGURE 3.10: ONE POSSIBLE FPGA IMPLEMENTATION OF THE $f_{cdf22}$ FUNCTION ........................................ 34
FIGURE 3.11: THREE 512X512 GRAYSCALE IMAGES - BARBARA (LEFT), GOLDHILL (MIDDLE), LENA (RIGHT) ........................................................................................................................................................................... 35
FIGURE 4.1: (A) MATRIX MULTIPLICATION CODE. (B) ROW-COLUMN OPERATION FOR MATRIX MULTIPLICATION................................................................................................................................. 37
FIGURE 4.2: SINGLE PROCESSOR MATRIX MULTIPLICATION IMPLEMENTED ON PE0 OF THE WILDCHILD BOARD........................................................................................................................................... 38
FIGURE 4.3: PERFORMANCE RESULTS VERSUS TIME FOR THE MATRIX MULTIPLICATION FUNCTION USING A SINGLE PROCESSOR ON THE WILDCHILD SYSTEM. ........................................................................... 41
FIGURE 4.4: MATRIX MULTIPLICATION IMPLEMENTED ON THE WILDCHILD BOARD USING MULTIPLE PROCESSORS ........................................................................................................................... 42
FIGURE 4.5: PERFORMANCE VERSUS TIME FOR PARALLEL MATRIX MULTIPLICATION ON WILDCHILD WITH 16-BIT FIXED-POINT INPUT AND 32-BIT FIXED POINT OUTPUT ......................................................................................................................................................... 45
FIGURE 4.6: PERFORMANCE VERSUS TIME FOR PARALLEL MATRIX MULTIPLICATION ON WILDCHILD WITH 16-BIT FIXED-POINT INPUT AND 16-BIT FIXED POINT OUTPUT ......................................................................................................................................................... 45
FIGURE 4.7: SINGLE PROCESSOR MATRIX MULTIPLICATION IMPLEMENTED ON PE1 OF THE WILDSTAR BOARD ......................................................................................................................................................... 47
FIGURE 4.8: PERFORMANCE RESULTS VERSUS TIME FOR THE MATRIX MULTIPLICATION FUNCTION USING A SINGLE PROCESSOR ON THE WILDSTAR SYSTEM. ........................................................................... 49
FIGURE 4.9: PARALLEL MATRIX MULTIPLICATION IMPLEMENTED ON PE1 AND PE0 OF THE WILDSTAR BOARD......................................................................................................................................................... 50
FIGURE 4.10: PERFORMANCE RESULTS VERSUS TIME FOR THE MATRIX MULTIPLICATION FUNCTION USING PARALLEL PROCESSORS ON THE WILDSTAR SYSTEM................................. 53
FIGURE 4.11: PERFORMANCE COMPARISON OF MATRIX MULTIPLICATION ALGORITHMS ON A SINGLE PROCESSOR ......................................................................................................................................................... 55
FIGURE 4.12: PERFORMANCE COMPARISON OF MATRIX MULTIPLICATION ALGORITHMS ON PARALLEL PROCESSORS. ......................................................................................................................................................... 56
FIGURE 5.1: SYNTHESIS FLOW OF THE MATCH COMPILER [25]........................................................................................................................................................................................................ 58
FIGURE 5.2: MATLAB CODE USING DIRECTIVES TO SPECIFY THE MEMORY LOCATION ........................................................................................................................................................................ 59
FIGURE 5.3: SYNTHESIZED RTL VHDL OF THE MATLAB CODE IN FIGURE 5.2 BY THE MATCH COMPILER. ........................................................................................................................................................... 60
FIGURE 5.4: MATLAB CODE FOR 64x64 MATRIX MULTIPLICATION. ........................................................................................................................................................................................................ 62
FIGURE 5.5: PERFORMANCE RESULTS VERSUS TIME FOR THE COMPILER-GENERATED MATRIX MULTIPLICATION DESIGNS ON PEX OF THE WILDSTAR SYSTEM......................... 64
FIGURE 5.6: PERFORMANCE COMPARISON OF THE COMPILER-GENERATED, SINGLE PROCESSOR AND MULTIPLE PROCESSOR MATRIX MULTIPLICATION DESIGNS ON THE WILDSTAR BOARD. ........................................................................................................................................................... 65
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Computation results for the Wavelet Transform implemented in C and MATLAB. Times are</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>given in seconds.</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>Hardware implementation of the Wavelet Transform performed on the WILDCHILD multi-FPGA</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>System at 21 MHz. Times are given in seconds.</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Performance results of the Matrix Multiplication function using a single processor on</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>the WILDCHILD system. Times are listed in seconds.</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Performance of Matrix Multiplication using parallel processors on the WILDCHILD system.</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>Data uses 16-bit fixed-point input and 32-bit fixed-point output. Times are listed in</td>
<td></td>
</tr>
<tr>
<td></td>
<td>seconds.</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Performance of Matrix Multiplication using parallel processors on the WILDCHILD system.</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>Data uses 16-bit fixed-point input and 16-bit fixed-point output. Times are listed in</td>
<td></td>
</tr>
<tr>
<td></td>
<td>seconds.</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Performance comparison between single and multiple processor designs on the WILDCHILD</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>board. Times are given in seconds.</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Performance results of the Matrix Multiplication function using a single processor on</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>the WILDSTAR system. Times are listed in seconds.</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Performance results of the Matrix Multiplication function using parallel processors on</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>the WILDSTAR system. Times are listed in seconds.</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>Performance comparison between single and multiple processor designs on the WILDSTAR</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>board. Times are given in seconds.</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>Performance comparison of Matrix Multiplication Algorithms [20].</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance results of the compiler-generated Matrix Multiplication designs on PEX of</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>the WILDSTAR system. Times are listed in seconds.</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Performance comparison between the compiler-generated designs and the single and</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>multiple processor designs on the WILDSTAR board. Times are given in seconds.</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Performance comparison of compiler-generated MATLAB benchmarks for the WILDCHILD and</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>WILDSTAR systems. Times are given in seconds.</td>
<td></td>
</tr>
</tbody>
</table>
INTRODUCTION

1.1 Motivation

Reconfigurable or adaptive computing is an emerging area of computing [18,19] that promises high performance from small, relatively low cost systems composed of re-programmable hardware such as field-programmable gate arrays (FPGAs). 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 tools to aid in the fast development of applications. The MATCH project [1,15] 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 on a collection of commercial-off-the-shelf (COTS) components including DSPs, FPGAs, and embedded processors.

By taking advantage of these resources, portions of complex computations can be distributed or assigned to these components, accelerating computation time and efficiency. Additionally, while implementing these resources in parallel, the computing engine can take advantage of course-grain parallelism in the computation. Furthermore, by using multi-FPGA and multi-DSP components it
is also possible to take advantage of the fine-grain parallelism in the computation as well.

1.2 MATCH

It seems apparent that developing high-speed heterogeneous computing engines is of great benefit to computationally intensive scientific applications. One major disadvantage of such an engine is the difficulty in programming the engine for the computation. Currently, many of the devices involved in these heterogeneous computing platforms require the programs be written in relatively low level languages such as assembly language or C, and C+MPI for parallel devices. The FPGAs require a hardware description language such as VHDL or VERILOG. There may also be difficulties to overcome with communication between resources. In many cases, specialists may be required to design these codes and the design time can be very long [20].

These restrictions make these systems difficult to use for scientists and engineers. They do not have direct access to the systems, and codes. Errors in the programs could take extremely long times to debug. For image and signal processing applications, most scientists would much rather code in a high-level language such as MATLAB and still have access to the processing power of the distributed heterogeneous computing engine. This is focus of MATCH (MATlab Compiler for Heterogeneous computing systems) [1,20].
The objective of MATCH is to build a compiler to automatically translate high-level applications written in MATLAB and map them to distributed heterogeneous computing systems, such as FPGAs, DSPs and embedded processors. The desire is to alleviate the arduous task of developing efficient code for such systems. As is depicted in Figure 1.1, MATCH will target not a single custom-computing engine, but a computing engine built from commercial off-the-shelf (COTS) components. The end goal of the compiler is to generate efficient code automatically while optimizing under two objectives, minimizing resources under performance constraints, and maximizing performance under resource constraints. The benchmark of these goals will be to reach within a factor of two to four the performance of a hand-implemented approach [20].

To aid in the development of MATLAB-based applications, a library of matrix and signal processing functions is being developed for reconfigurable hardware that can be used as building blocks for these applications.

1.3 MATCH Testbed

To prove the technology behind MATCH, an example testbed was compiled containing four types of compute resources. These resources are examples of COTS devices connected together using a VME cage. The VME bus provides the communication backbone for the components to work in concert.

![Figure 1.2: The MATCH testbed.](image)

The testbed uses two off-the-shelf multi-FPGA boards from Annapolis Micro Systems, the WILDCHILD and WILDSTAR systems, as the reconfigurable units. The WILDCHILD system contains eight Xilinx 4010 FPGAs, each with 400
CLBs and 512KB local memory, and one 4028 FPGA with 1024 CLBs and 1MB local memory. The WILDSTAR system contains three Xilinx Virtex XCV400 series FPGAs, each with 2400 CLBs. Two of the FPGAs each contain two 1MB local memories. Additionally, four Mezzanine memory cards, each containing two 2MB memory banks, are shared between the three FPGAs. A Transtech TDM-428 board is used as a DSP resource. This board has four Texas Instruments TMS320C40 processors each operating at 60MHz, with 8MB RAM. The interconnection is an on-board 8-bit wide 20MB per second communication network. The testbed also employs a pair of Motorola MVME2604 boards, each hosting a PowerPC-604 processor operating at 200 MHz with 64MB local memory running Microware’s OS-9 operating system. These processors can communicate among themselves via a 100BaseT Ethernet interface. Finally, a Force 5V board with a MicroSPARC-II processor operates as the main controller of the testbed. This processor runs Solaris 2.6 operating system and can communicate with the other computing resources via the VME bus or an Ethernet interface. The MATCH testbed is depicted above in Figure 1.2.

1.4 The MATCH Compiler

One major component of the heterogeneous computing engine, and perhaps the most interesting part of MATCH, is the utilization of the Field Programmable Gate Arrays (FPGAs). The FPGA is an attempt at the best of both worlds of computing, by combining the speed of hardware with the flexibility of software. For MATCH, the utilization of FPGAs is optimal for supporting code that is often executed but has little control, such as loops. Often, these code kernels represent the majority of the computation time, yet a relatively small portion of the code [20].

The functionality of the MATCH compiler in utilizing FPGAs in the testbed can be divided into two phases. The first phase is the dynamic compilation of MATLAB code to VHDL, which can then be synthesized and executed on FPGAs. Directives set by the user in the high-level language, such as type, shape and precision of arrays that cannot be inferred, can be used to dynamically modify the configuration and architecture of the VHDL code during compilation. Of course, this process cannot be used to produce VHDL code for intrinsic MATLAB functions that do not have code descriptions.

The second phase for the utilization of FPGAs is the development of hand-coded MATLAB library functions written in VHDL. These library functions have been optimized for performance, and may be called by the
compiler during the translation and mapping phases of the compilation from MATLAB to VHDL. This process is analogous to MATLAB itself in which their exists an extensive library of functions ranging from simple matrix operations to highly complex signal processing functions. Externally, the complexities of the low-level computations in these library functions are not apparent. By developing these hand-coded library functions for the MATCH compiler, it provides the user with the flexibility to use a large selection of these MATLAB library functions in the high-level applications. Additionally, it provides significant improvement in the overall performance of the MATCH compiler system.

The two phases of the MATCH compiler mentioned above are the focus of this research and they are discussed in more detail in this work. A model of the compiler is shown in Figure 1.3.

![Figure 1.3: The MATCH Compiler.](image)

### 1.5 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 [18,19]. Some of these developments have been in software engineering in the form of tools that enable easy mapping of applications on reconfigurable systems and the development of such applications themselves. Some of these developments are explained below.

The Cameron project [3] at 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. It is specifically focused on Image Processing (IP) applications. The Cameron 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. 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 [4] combines the performance advantages of an FPGA-based computer to the floating-point arithmetic and complex algorithm execution capabilities of a DSP. The goal of the 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. 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 [5] at the University of Tennessee, Knoxville, also focuses on providing tools to automate the process of mapping image processing applications in Khoros onto reconfigurable systems. This project's approach is to build a library of pre-compiled primitives that can be used as building blocks of image processing applications.

The MATCH compiler project [1] differs from all of the above in that its goal is to develop an integrated compilation environment for generating code for FPGAs, DSPs and embedded processors, using both a library-based approach and automated generation of C code for the DSP and RTL VHDL code for the FPGAs [14].

1.6 Thesis Outline

The following chapters discuss the design and evaluation of MATLAB functions on FPGAs. Chapter 2 describes the hardware and software architecture of the Annapolis WILDCCHILD and WILDSTAR FPGA systems, which are part of the MATCH testbed. Furthermore, it discusses the design process and certain issues related to application development for these two systems.
Chapter 3 describes the implementation of a very useful image-processing algorithm, namely the Wavelet Transform on general purpose and FPGA based hardware. We compare implementations of the same algorithm using MATLAB (an interpreted language), C (a compiled language) and VHDL (a hardware description language) and observe the performance improvements that can be obtained through implementations in hardware.

Chapter 4 discusses two implementations of Matrix Multiplication, on a single processor and on parallel processors. Their performance on the WILDCHILD and WILDSTAR systems on are evaluated and compared.

Chapter 5 discusses the MATCH Compiler for the WILDSTAR multi-FPGA system. The compiler-generated MATLAB code for the Matrix Multiplication function is evaluated in comparison to the hand-coded designs. Several benchmarks were also tested on the WILDCHILD and WILDSTAR compilers to evaluate and compare the efficiency of the two systems.

Chapter 6 draws some conclusions from this work, and discusses possible improvement in the development, as well as future work.
THE WILDCHILD AND WILDSTAR SYSTEMS

This chapter describes the hardware and software architectures of the WILDCHILD and WILDSTAR FPGA systems, developed by Annapolis Micro Systems. This chapter also considers the design process, implementation and execution of reconfigurable computing applications on these systems. Details are provided for the design methodology in developing an application for the system using VHDL, as well as C code and the host application program interface (API). Certain design issues related to application development on these two FPGA systems are mentioned. While both systems are included in the MATCH testbed, each system has advantages over the other, which are discussed here as well.

2.1 The WILDCHILD Multi-FPGA System

The WILDCHILD custom computing engine is an integrated system consisting of multi-FPGA reconfigurable hardware units, 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.
2.1.1 Hardware Architecture

The WILDCHILD multi-FPGA board consists of nine FPGAs of the popular Xilinx 4000 family. Eight of these are 4010 FPGAs, referred to as Processing Elements 1 through 8 (PE1-PE8) or generically PEX, each with 400 CLBs (approximately 10,000 gates) and all are identical. The ninth is a 4028 FPGA referred to as Processing Element 0 (PE0) with 1024 CLBs (approximately 30,000 gates). The nine FPGAs are arranged in a master-slave configuration as shown in Figure 2.1.

![Figure 2.1: The WILDCHILD System Architecture.](image_url)

Each FPGA on the board is connected to a pseudo dual-ported memory, which can be accessed by both the FPGA and the host. Each 4010 FPGA is connected to a 512KB RAM, containing $2^{18}$ addressable locations of 16-bits wide. The PE0 contains a 1MB RAM, containing $2^{18}$ addressable locations of 32-bits wide.

The architecture of the system enables high-throughput systolic computation using the 36-bit bus that connects each PEX to its neighbors and also to the on-board FIFOs. These connections allow the board to be used as a ring topology. There is also a crossbar network that is capable of connecting arbitrary FPGAs to each other for communication in up to sixteen different pre-programmed configurations, thus enabling irregular computations between...
different processors. The capability of programming the crossbar configuration lies with PE0, the master FPGA, by use of a 4-bit control register. There is also a separate broadcast mode in which PE0 can transmit data to each PEX simultaneously via their crossbar ports regardless 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 2.2.

![Figure 2.2: Crossbar Configuration Specification.](image)

As shown in Figure 2.1, PE0, PE1 and PE8 each have FIFO connections to the host processor via the systolic bus, which allow data streaming between the FPGAs and the host processor, the microSparc II. In addition to the crossbar network, PE0 can communicate control and status information to each PEX processor using a set of global handshake signals connected to each FPGA. The system also features a global reset signal, however, the reset of each FPGA does not necessarily occur simultaneously. When a design application requires all FPGAs to be synchronized at reset, it may be useful to take advantage of the handshaking mechanism. The system clock is synchronous across all the FPGAs, and is completely configurable to operate at any rate up to a maximum frequency of 40 MHz.

Applications for the FPGAs are designed using register transfer logic (RTL) VHDL. The code is then synthesized into a bitmap image using Xilinx tools and programmed onto the FPGA via the host interface. Since the architecture of all PEXs is similar, they may share the same FPGA design. However, the architecture of PE0 is unique, and therefore requires a separate design. This process is discussed in more detail in section 2.1.2.2.
2.1.2 Software

The software design for the WILDCHILD system is divided into two sections. The first section deals with the host interface, which is responsible for the system configuration and used as an interface between the user and the WILDCHILD system. The second section discusses the implementation of the FPGA designs using RTL VHDL.

2.1.2.1 Host Interface

The host interface to the WILDCHILD 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. The software interface to the WILDCHILD system consists of a set of custom headers and application program interface (API) library functions provided by Annapolis Micro Systems. The API functions are utilized for application development in C using the gcc compiler. They allow the host to control the FPGAs in performing tasks such as loading bitmap designs into the FPGAs, reading and writing data to the onboard RAMs, resetting the board, and receiving interrupt signals from the FPGAs. A description of the most important WILDCHILD API functions that are used to perform the basic tasks involved in running any application is listed in Appendix A. For a more comprehensive list and detailed description of each function, please refer to the WILDCHILD user manual [21].

When creating a host interface for an application on the WILDCHILD system, it is useful to follow a basic uniform algorithm. However, the host program may be modified to suit the desired application. The standard algorithm used for MATCH applications is as follows: The host begins by configuring the system. This includes opening the board, setting the desired clock frequency, masking interrupts, etc. Once the board has been configured, the PE bitmap image files are loaded onto the FPGAs and the crossbar network configuration is set. Data may then be written to any of the memory locations on the board. The entire process is started by resetting all the PEs. The host must then wait for the necessary interrupt signals from the PEs to recognize when the application has completed, after which it may read back any new data from the memories. Finally, the host completes the process by reprogramming the PEs with a safe image, followed by shutting down the system.
2.1.2.2 VHDL modeling for WILDCHILD FPGAs

The VHSIC Hardware Description Language (VHDL) is one of the most popular languages used in modeling digital systems due to its varying levels of abstraction. Systems can be modeled as structural components, behavioral models, or a combination of both. The latter, a more popular approach in VHDL design, is the use of register transfer logic (RTL) description of digital systems in which the system progresses from one state to another, usually on a global clock signal. The state machine methodology was adopted for all MATCH application designs on the FPGAs due to its simplicity in design and automatic synthesis. Refer to [8] for a better understanding of the RTL methodology for FPGA synthesis.

Annapolis Micro Systems provides a complete library and description of the entire WILDCHILD board in VHDL down to the level of each individual FPGA architecture. 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 such as Vsim to verify the functionality of the design. All PEX architectures have the same configuration and structure, and therefore share the same signals and libraries. However, the architecture of PE0 is unique and therefore requires a separate design and libraries. Figure 2.3 shows the VHDL entity model commonly used in MATCH applications for simple designs on PE0 that only require memory access.

entity WF_PE0 is
  port (     
    clock : in  std_logic;
    reset : in  std_logic;
    mem_grant_n : in  std_logic;
    mem_data_in : in  std_logic_vector(31 downto 0);
    mem_data_out : out std_logic_vector(31 downto 0);
    mem_address : out std_logic_vector(17 downto 0);
    mem_request_n : out std_logic;
    mem_write_n : out std_logic;
    done : out std_logic
  );
end entity;

Figure 2.3: A sample PE0 entity commonly used for FPGA designs on the WILDCHILD system.

In addition to the entity, a corresponding architecture is created using a state machine, as described above. This method is especially useful when accessing data
from the memory locations, since timing plays an important role in the deliverance and acquisition of data from memory. Figure 2.4 shows an example of state machine implementing a for-loop in which data is written to memory. It is significant to mention here that in association with the timing diagrams and information provided by Annapolis Micro Systems, the use of a state machine also eases the ability to pipeline data to and from the memory locations at each clock cycle. This can surely improve overall performance in any application.

if ( reset = '1' ) then
  i := ( others => '0' );
  mem_request_n <= '1';
  mem_write_n <= '1';
  mem_address <= ( others => '0' );
  mem_data_out <= ( others => '0' );
  done <= '0';
  state <= s1;
elsif rising_edge( clock ) then
  case state is
    when s1 =>
      if ( i > "1000000000") then
        state <= s4;
      else
        state <= s2;
      end if;
    when s2 =>
      mem_request_n <= '0';
      mem_write_n <= '0';
      mem_address <= i;
      mem_data_out <= VALUE * i;
      state <= s3;
    when s3 =>
      if ( mem_grant_n = '0' ) then
        i := i + '1';
        state <= s1;
      else
        state <= s3;
      end if;
    when s4 =>
      mem_request_n <= '1';
      mem_write_n <= '1';
      done <= '1';
      state <= s4;
  end if;
end if;

Figure 2.4: A state machine implementation of a for-loop in which data is read from the local memory.
Once the architecture is complete, the entity may be inserted into the core architecture of the PE as a component. This component is also instantiated as a process in which the signals of the system are connected to the interfaces on the FPGA provided by the manufacturer. Figure 2.5 shows an example of the core architecture for PE0, which incorporates the design entity of Figure 2.3. For a more detailed description of the architectures, signal mapping, and timing diagrams, please refer to the WILDCHILD user manual [21].

```vhdl
architecture Behavior of PE0_Core is

component WF_PE0
port (  
clock : in  std_logic;
reset : in  std_logic;
mem_grant_n : in  std_logic;
mem_data_in : in  std_logic_vector(31 downto 0);
mem_data_out : out std_logic_vector(31 downto 0);
mem_address : out std_logic_vector(17 downto 0);
mem_request_n : out std_logic;
done : out std_logic
);
end component;

begin

master_process : WF_PE0
port map (  
clock  => Pclk,
reset   => Reset,
done     => Intr,
mem_write_n => MwriteEN_n,
mem_request_n => Mreq_n,
mem_grant_n  => Mgnt_n,
mem_address => Maddr_OutFF,
mem_data_in  => Mdata_InFF,
mem_data_out => Mdata_OutFF
);

end Behavior;
```

Figure 2.5: A sample core architecture for PE0 that incorporates a user’s FPGA design as a component, and then connects the signals to the FPGA interfaces in a process named “master_process”.
2.2 The WILDSTAR Multi-FPGA System

Similar to the WILDCHILD, the WILDSTAR system is an integrated system consisting of multi-FPGA reconfigurable hardware units, and a host that controls it and provides a conventional software interface to the complete system. The WILDSTAR FPGA unit is also a VME-compatible board that is installed alongside the WILDCHILD system 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 WILDSTAR unit communicate through the VME bus, which is also shared by other components of the MATCH adaptive computing system.

2.2.1 Hardware Architecture

The WILDSTAR multi-FPGA board consists of three FPGAs of the Virtex Xilinx 400 series, each containing 2400 CLBs (over 465,000 gates). The two outer FPGAs, referred to as Processing Elements 1 and 2 (PE1 and PE2) or generically PEX, have identical architectures. The middle FPGA, Processing Element 0 (PE0), has a unique architecture. The three FPGAs are arranged in a master-slave configuration as shown in Figure 2.6.

![Figure 2.6: The WILDSTAR System Architecture.](image)
Each PEX on the board is connected to two pseudo dual-ported local memories, which can be accessed by both the FPGA and the host. Each left and right local memory is a 1MB RAM, containing $2^{18}$ addressable locations of 32-bits wide. The PE0 does not contain any local memories of its own.

In addition to the local memories, there are also two sets of left and right Mezzanine memory cards, which are shared between PE0 and each PEX. Two individual 2MB RAMs, containing $2^{18}$ addressable locations of 64-bits wide, occupy each Mezzanine Card. There is also a crossbar network in each Mezzanine Card that is capable of connecting the FPGAs to either of the two RAMs, or to each other to enable communication between PEX and PE0. As depicted in Figure 2.7, the four possible crossbar configurations only allow an FPGA access a single memory at a time, or the ability to send and receive data from its neighboring FPGA. When accessing data from memory, PEX has a 64-bit port, allowing access to the full 64 bits of each address in memory. However, PE0 is limited to a 32-bit port, allowing access to only the lower 32 bits of each register in the memories. When using the crossbars for data communication between PEX and PE0, two modes are available for selecting either the upper or lower 32 bits of PEX to shunt to PE0. Control of the crossbar configurations for each Mezzanine Card may be assigned to either PEX or PE0, however, they can not share crossbar control simultaneously. This control option is set in the core architectures of the PEs and may be modified dynamically in an application design. Please refer to the WILDSTAR user manual [22] for more details on the Mezzanine Cards.

![Figure 2.7: Four Crossbar configurations for the Mezzanine Cards.](image)

- “00” PEX accesses MEM 0, PE0 accesses MEM 1; “01” PEX accesses MEM 1, PE0 accesses MEM 0; “10” PEX addr/ctrl & low data shunted to PE0; “11” PEX addr/ctrl & high data shunted to PE0.
The architecture of the system also enables high-throughput systolic computation between the two PEXs using the 36-bit top and bottom systolic bus. These connections allow the board to be used as a ring topology and permits data to be shared between all PEs. In addition to the crossbar network and systolic buses, PE0 can communicate control and status information to each PEX processor using a set of global handshake signals connected to each FPGA. This mechanism is highly useful when a design application requires the synchronization of multiple FPGAs, such as during the reset stage.

As shown in Figure 2.6 above, all three PEs have FIFO connections to the host processor via the Local Addressable Data (LAD) bus, which allows data streaming between the FPGAs and the host processor, the microSpare II. The host interface may use the LAD bus to read and write data to any of the $2^{15}$ addressable registers in each FPGA. This functionality may be used in many applications, such as writing to a Reset register in order to reset a process or application, or reading from an interrupt register to assert whether an application has completed. The interrupt process to the host for designs on the WILDSTAR board is shown in Figure 2.8. The host continuously reads from the register at address INTERRUPT_BASE, and waits for the value ‘0x1’ to appear, which identifies that the design has completed. The interrupt signal is set in the final state of the main architecture of a design application.

```vhdl
P_Interrupt : process ( reset, Kclk, LAD, interrupt )
bEGIN
  if ( reset = '1' ) then
    LAD.Akk    <= '0';
    LAD.Data_Out <= ( others => '0' );
  elsif ( rising_edge (Kclk) ) then
    if ( LAD.Strobe = '1' ) then
      if ( LAD.Write = '0' ) then
        if ( ( LAD.Addr (INTERRUPT_BASE'range) and BASE_MASK ) = INTERRUPT_BASE ) then
          LAD.Akk      <= '1';
          LAD.Data_Out  <= interrupt;
        END if;
      end if;
    elsif ( LAD.Strobe = '0' ) then
      LAD.Akk            <= '0';
    end if;
  end if;
end process;
```

**Figure 2.8: Interrupt process used in WILDSTAR designs, where the host reads the interrupt from a register at address INTERRUPT_BASE.**
The WILDSTAR system contains four system clocks that are synchronous across all the FPGAs. The two most commonly utilized in applications is the K_CLK, which is the LAD bus clock, and the M_CLK, which is used for any memory access by the FPGAs. The former is always set to 33 MHz, while the latter may be configured via the host program to operate at any rate between 25 to 100 MHz.

Applications for the FPGAs are designed using register transfer logic (RTL) VHDL. The code is then synthesized into a bitmap image using Xilinx tools and programmed onto the FPGA via the host interface. Since the architecture of all PEXs is similar, they may share the same FPGA design. However, the architecture of PE0 is unique, and therefore requires a separate design. This process is discussed in more detail in section 2.2.2.2.

### 2.2.2 Software

The software design for the WILDSTAR system is divided into two sections. The first section deals with the host interface, which is responsible for the system configuration and used as an interface between the user and the WILDSTAR system. The second section discusses the implementation of the FPGA designs using RTL VHDL.

#### 2.2.2.1 Host Interface

Similar to the WILDCHILD system, the host interface to the WILDSTAR on the FORCE 5V consists of a set of drivers that communicate with the WILDSTAR board using other drivers that talk to the VME back-plane. The software interface to the WILDSTAR system consists of a set of custom headers and application program interface (API) library functions provided by Annapolis Micro Systems. The API functions are utilized for application development in C using the gcc compiler. They allow the host to control the FPGAs in performing tasks such as loading bitmap designs into the FPGAs, reading and writing data to the onboard memories and registers, and setting the clock frequency. A description of the most important WILDSTAR API functions that are used to perform the basic tasks involved in running any application is listed in Appendix A. For a more comprehensive list and detailed description of each function, please refer to the WILDSTAR user manual [22].
When creating a host interface for an application on the WILDSTAR system, it is useful to follow a basic uniform algorithm. However, the host program may be modified to suit the desired application. The standard algorithm used for MATCH applications is as follows: The host begins by configuring the system. This includes tasks such as opening the board and setting the desired clock frequency. Once the board has been configured, the PE bitmap image files are loaded onto the FPGAs. Data may then be written to any of the memory locations on the board. The entire process is started when the PEs are reset by writing to the Reset register on each FPGA. The host must then continuously read an interrupt vector from the Interrupt register on each FPGA to recognize when the application has completed, after which it may read back any new data from the memories. Finally, the host completes the process by reprogramming the PEs with a safe image, followed by shutting down the system.

2.2.2.2 VHDL modeling for WILDSTAR FPGAs

As discussed in section 2.1.2.2 above, the state machine methodology of RTL VHDL was adopted for all MATCH application designs on the FPGAs due to its simplicity in design and automatic synthesis. Annapolis Micro Systems provides a complete library and description of the entire WILDSTAR board in VHDL down to the level of each individual FPGA architecture. 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 such as Vsim to verify the functionality of the design.

All PEX architectures have the same configuration and structure, and therefore share the same signals and libraries. However, the architecture of PE0 is unique and therefore requires a separate design and libraries. Figure 2.9 shows the VHDL entity model, which is commonly used in MATCH applications for simple designs on PEX that only require memory access. The WILDSTAR architecture makes use of MUXs for each component to efficiently appropriate control of the memories, LAD bus, and other components between multiple processes in the application.
entity WS_PEX is
        port (  
        Mclk            : in  std_logic;  
        Kclk            : in  std_logic;  
        reset           : in      std_logic;  
        Left_Local_Mux  : inout Mem32_Mux;  
        Right_Local_Mux : inout  Mem32_Mux;  
        Left_Mezz_Mux   : inout  Mem64_Mux;  
        Right_Mezz_Mux  : inout Mem64_Mux;  
        LAD             : inout  LAD_MUX;  
        LEDS_Out        : out  LED_Std_IF_Out_Type  
        );
    end entity;

    Figure 2.9: A sample PEX entity commonly used for FPGA designs on the WILDSTAR system.

In addition to the entity, a corresponding architecture is created using a state machine, as described above. This method is especially useful when accessing data from the memory locations, since timing plays an important role in the deliverance and acquisition of data from the different memory locations on the board. Figure 2.10 shows an example of state machine implementing a for-loop in which data is read from the PEX’s left local memory.

It is important to mention here that when accessing data from any of the WILDSTAR’s memories using the MUX components, one may not make any assumptions about the behavior of the \( a_kk \) and \( data\_valid \) signals in terms of when they will be asserted or deasserted. This undeterministic behavior prevents the pipelining of data to and from the memory locations at each clock cycle. Consequently, performance and efficiency in applications may suffer, if not realize its full potential.

One way of improving the performance of data acquisition in designs on the WILDSTAR board is by taking advantage of the fact that data in different RAMs can be accessed in parallel rather than sequentially. This is in contrast to the WILDCHILD system in which each FPGA only has access to a single local memory. However, the WILDSTAR system provides an abundance of memory components, and by distributing data sets among the different memories, one may access several sets of data simultaneously, while increasing the overall performance of the design.
if ( reset = '1' ) then
    i            := (others => '0') ;
    Left_Local_Mux.Addr <= (others => '0') ;
    Left_Local_Mux.Data_Out <= (others => '0') ;
    Left_Local_Mux.Req <= '0' ;
    Left_Local_Mux.Write <= '0' ;
    interrupt(0)             <= (others => '0') ;
    state  <= state_0 ;
elsif ( rising_edge (Mclk) ) then
    case state is
        when state_0 =>
            if  ( i > "1000000000") then
                state <= state_4 ;
            else
                state <= state_1 ;
            end if  ;
        when state_1 =>
            Left_Local_Mux.Req     <= '1' ;
            Left_Local_Mux.Addr    <= Left_Local_Addr ;
            if (Left_Local_Mux.Akk = '1') then
                state            <= state_2 ;
            else
                state            <= state_1 ;
            end if ;
        when state_2 =>
            Left_Local_Mux.Req     <= '1' ;
            if (Left_Local_Mux.Data_Valid = '1') then
                VALUE <= Left_Local_Mux.Data_In ;
                state   <= state_3 ;
            else
                state   <= state_2 ;
            end if ;
        when state_3 =>
            Left_Local_Mux.Write     <= '0' ;
            if (Left_Local_Mux.Data_Valid = '0') then
                i       := i + '1' ;
                state  <= state_0 ;
            else
                state  <= state_3 ;
            end if ;
        when state_4 =>
            interrupt(0) <= '1' ;
            state       <= state_4 ;
    end case ;
end if ;

Figure 2.10: A state machine implementation of a for-loop in which
data is read from the PEX’s left local memory.
The entity of the design is inserted into the core architecture of the PE as a component. This component is also instantiated as a process in which the signals of the system are connected to the interfaces on the FPGA provided by the manufacturer. Figure 2.11 shows an example of the core architecture for PEX, incorporating the design entity of Figure 2.9. For a more detailed description of the architectures, signal mapping, and timing diagrams, please refer to the WILDSTAR user manual [22].

architecture Behavior of PEX_Core is
  
signal LAD_Mux Bus : LAD_Mux_vector ( 0 to 5 );
signal Left_Local_Mux : Mem32_Mux_vector ( 0 to 1 );
signal Right_Local_Mux : Mem32_Mux_vector ( 0 to 1 );
signal Left_Mezz_Mux : Mem64_Mux_vector ( 0 to 1 );
signal Right_Mezz_Mux : Mem64_Mux_vector ( 0 to 1 );

component WS_PEX is
  port ( 
    Mclk : in      std_logic;
    Kclk : in      std_logic;
    Left_Local_Mux : inout   Mem32_Mux;
    Right_Local_Mux : inout   Mem32_Mux;
    Left_Mezz_Mux : inout   Mem64_Mux;
    Right_Mezz_Mux : inout   Mem64_Mux;
    LAD : inout   LAD_Mux;
    LEDs_Out : out     LED_Std_IF_Out_Type;
    reset : in      std_logic
  );
end component;

begin

  PEX_Instance : WS_PEX
  port map ( 
    Mclk => Clocks_In.M_Clk,
    Kclk => Clocks_In.K_Clk,
    Left_Local_Mux => Left_Local_Mux(1),
    Right_Local_Mux => Right_Local_Mux(1),
    Left_Mezz_Mux => Left_Mezz_Mux(1),
    Right_Mezz_Mux => Right_Mezz_Mux(1),
    LAD => LAD_Mux_Bus(0),
    LEDs_Out => LEDs_Out,
    reset => Global_Reset
  );

end Behavior;

Figure 2.11: A sample core architecture for PEX that incorporates a user’s FPGA design as a component, and then connects the signals to the FPGA interfaces in a process named “PEX_Instance”.

22
2.3 WILDCHILD vs. WILDSTAR

When designing an application for an FPGA system, it is essential to determine the requirements of the application and ascertain which system may provide the overall best results. Issues such as system architecture, speed, parallelism, size, communication, memory requirements, and the design complexity must be taken into consideration when creating any application. The Annapolis WILDCHILD and WILDSTAR systems both have advantages and disadvantages that effect design applications. This section will address some of these differences and issues.

When comparing the architecture of the two systems, one considerable difference is this size of the FPGAs. The WILDCHILD system is made of eight Xilinx 4010 FPGAs, each with 400 CLBs, and one 4028 FPGA with 1024 CLBs. The WILDSTAR system contains three Xilinx Virtex XCV400 series FPGAs, each with 2400 CLBs. Clearly, the size of the FPGAs in the WILDSTAR system far surpasses that of the WILDCHILD system, allowing larger application designs. However, for smaller designs that require more parallelism, the WILDCHILD system may have a greater advantage due to the accessibility of nine FPGAs, as compared to the WILDSTAR system, which only allows a maximum of three FPGAs to process in parallel. In this case, the WILDCHILD system may complete a design in only a fraction of the time required by the WILDSTAR system. Data communication between FPGAs in both systems is relatively comparable.

With regard to the speed of an application design, the WILDSTAR’s FPGAs can run designs at a maximum frequency of 100 MHz, while the WILDCHILD’s FPGAs are limited to a maximum frequency of 40 MHz. Clearly, the former can potentially run designs at least twice as fast as the latter, which may conceivably make up for the system’s limited parallelism.

Memory size and accessibility is another significant issue when selecting a system for an application design. Each FPGA in the WILDCHILD system contains its own local RAM and is not directly accessible by any other FPGA on the board. The largest memory component is a 1MB RAM that is connected to PE0, containing $2^{18}$ addressable locations of 32-bits wide; the remaining eight PEXs are connected to a 512KB RAM, containing $2^{18}$ addressable locations of 16-bits wide. Data sharing and memory storage is obviously limited on this board. The only way an FPGA can access the data from another local memory is if it is sent via the systolic bus, crossbar network, or if the data is copied to the FPGA’s local RAM by
the host interface. In either case, data access time becomes a large factor and may dramatically slow down the computation time.

The memory architecture of the WILDSTAR is abundant and adequately sized for most applications, and accessibility is not a consequential issue. Each PEX is connected to two local 1MB RAMs, containing $2^{18}$ addressable locations of 32-bits wide. Furthermore, the four Mezzanine Cards allow each PEX to access up to four additional 2MB RAMs, containing $2^{18}$ addressable locations of 64-bits wide. PE0 has access to all eight Mezzanine memories, allowing data sharing between PEX and PE0. However, data sharing between the two PEX components is limited and requires assistance of the top and bottom systolic buses to communicate information across the board.

Data pipelining is another large factor in FPGA designs. As mentioned in sections 2.1.2.2 and 2.2.2.2, the architecture of the WILDCHILD system permits data pipelining to and from memory in each clock cycle based on the timing diagrams and information provided by Annapolis Microsystems. This ability increases efficiency in designs and computation time. In contrast, the use of the MUX components in the WILDSTAR architecture restricts the ability to pipeline data to and from memory since one may make no assumptions when the $akk$ and $data\_valid$ signals may be asserted or deasserted. Consequently, data access time is increased and larger designs may even see enormous increases in computation time. This constraint may be avoided or perhaps minimized with the use of block RAMs inside a design, created with arrays or matrices of standard logic vectors. Data is then stored inside the FPGA rather than in memory, and access to this data is reduced to constant time. Alternatively, since each FPGA on the WILDSTAR board is connected to several memories, it can acquire data from different memories simultaneously, decreasing the memory read times. Although it does not make up for the speed increase due to pipelining, it does however greatly effect the total computation time of an application.

2.4 Design Process

The process involved in designing applications on the FPGA is both complicated and arduous, as depicted graphically in Figure 2.12. The design code for the FPGAs of the WILDCHILD and WILDSTAR systems is written in VHDL. The main reason for this choice is that Annapolis Microsystems supplies extensive
VHDL libraries for use with simulation of FPGA designs, including representations of all the signals available on the FPGA. During this phase, designs can be simulated as if they were actually running on the board using tools such as Vsim by ModelTech. The simulation allows the user to see values associated with signals as the design proceeds so that it may be debugged.

Figure 2.12: FPGA design process.

Once the design’s correctness has been verified in simulation, the logic corresponding to the design needs to be synthesized using the Synplicity Synplify tool from Synplicity. This tool is designed to synthesize netlists that can be placed and routed for FPGAs and ASICs. The tool performs optimizations on the design, and is ideal for behavioral code in the form of finite state machines. The synthesis tool reports the resource requirements for the design in terms of Configurable Logic Block (CLB) or Look Up Table (LUT) count on the target FPGA. Timing results are also provided in terms of the delay of the critical path. The designer may then determine whether or not the size and speed of the generated logic fits the target FPGA, and if need be, alter the design to optimize its resource utilization.

Once synthesized, Xilinx tools are used to place and route the design and optimizations are once again performed during this phase. Xilinx converts the synthesized netlists it into a bitmap file (.bit), which in turn is converted into a PROM file (.mcs). Finally, a small program included from Annapolis Microsystems converts this file into a special format (.m68) for use with
WildChild and WildStar systems. The design is now ready to be loaded onto the FPGA.

Using the C API from Annapolis Microsystems, a host interface is created to program the FPGAs and with the bitmap designs, load the appropriate data into the memories, and start and end the computation, as described in sections 2.1.2.1 and 2.2.2.1. The designer must then check the output of the system to assert whether or not the design works properly. Further debugging is then required at the VHDL simulation level, and the entire design process must be repeated.

2.5 Summary

This chapter discussed the hardware and software architecture of the WildChild and WildStar reconfigurable computing systems. A description was provided for the design methodologies used in creating applications for the two FPGA systems, as well as a comparison of their architectures and other important related issues for determining which system is more suitable for an application. Finally, a description was provided of the entire design cycle involved in the development of an application on the WildChild and WildStar boards. The following chapters describe and evaluate the design implementation of various designs and library functions on the two systems.
This chapter discusses the implementation of the Wavelet Transform as a benchmarking tool for WILDCHILD system as part of the MATCH testbed. The compression algorithm is based on the biorthogonal Cohen-Daubechies-Feauveau wavelet, and was provided by Honeywell, Inc. as part of their Versatility Benchmarking Tool for Configurable Computing [17]. An overview of the Wavelet Transform is provided, followed by a comparison of the algorithm written in MATLAB, C, and VHDL. The function was run on the WILDCHILD multi-FPGA system and tested on three distinct images. The performance and timing results were compared to that of MATLAB and C.

3.1 Overview of the Wavelet Transform

The Wavelet Transform is a compression algorithm that performs a high-pass and low-pass filter on the pixels of the image in two different stages; the first is a row operation and the second is a column operation. The image is then divided into four sub-bands, as depicted graphically in Figure 3.1. The algorithm is performed on the image two more times on the top-left-most sub-band.

![Figure 3.1: The filter operations of the Wavelet Transform.](image-url)
Figure 3.2 is a graphic representation of a Wavelet Transform performed on a 512x512 image. The first time will divide the image into four sub-bands of size 256x256, each subjected to a combination of high-pass (H) and/or low-pass (L) filtering. The second time will divide the top-left 256x256 sub-band into four 128x128 sub-bands; the third time will divide the top-left sub-band into four 64x64 sub-bands. The top-left-most sub-band (LL$_3$) contains the most important data, while the bottom-right-most sub-band (HH$_1$) contains the least important data, mostly values close to zero which range from –2 to +2. These small values will eventually be rounded off to zero when using lossy compression since they will not have a great effect on the image restoration during the decompression of the image.

3.2 Wavelet Transform Algorithm in C Language

The Wavelet Transform algorithm was provided in C language by Honeywell, Inc. as part of their Versatility Benchmarking Tool for Configurable Computing [17]. The algorithm used is based on a modified version of the biorthogonal Cohen-Daubechies-Feauveau wavelet. The C program was designed specifically to perform a wavelet transform on a 512x512 grayscale image. The main function, `forward_wavelet`, performs the wavelet transform three times on the image by calling the `fcdf22` function to perform the row and column operations. The `forward_wavelet` function in C Language is shown in Figure 3.3.
forward_wavelet() {
    int I, nt;
    int ROW = 1, COL = 512;
    for (nt=512; nt>=128; nt>>=1) {   // perform 3 times
        for (i=0; i<nt*512; i+=512)
            fcdf22 (&int_data[i], nt, ROW);  // row operation
        for (i=0; i<nt; i++)
            fcdf22 (&int_data[i], nt, COL);  // column operation
    }
}

Figure 3.3: The main function of the Wavelet Transform in C Language.

The fcdf22 function, shown in Figure 3.4, performs all of the computations for the high-pass and low-pass filters of the Wavelet Transform. The algorithm is made up of three for-loops. The first for-loop loads all the pixel values for the low-pass into the array, s[i], and the values for the high-pass into the array, d[i]. The second for-loop performs the high and low pass filters and calculates new values for s[i] and d[i]. During these computations, only four pixel values from the arrays are required: s[i], s[i+1], d[i], and d[i-1]. Once the new values of all the pixels are computed, the third for-loop restores the new values back into the image. The same function is performed for both row and column operations.

Figure 3.4: The fcdf22 function of the Wavelet Transform which performs the high-pass and low-pass filters.
It was realized in the early development of this benchmark that the WILDCHILD’s largest local memory, which is connected to PE0, has a limited RAM size of 512x512 addressable spaces, and it was only large enough to hold the individual 512x512 image matrix. Consequently, the system was unable to sustain enough resources to hold the entire image in RAM and perform the wavelet function without affecting the data in memory. It was therefore necessary to make modifications to the algorithm in order to successfully implement this benchmarking tool on the WILDCHILD system.

In order to run the function properly on the WILDCHILD system, it was necessary to redesign the \textit{fcdf22} function to avoid the computation of all the pixels in RAM at one time. As shown in Figure 3.5, the redesigned \textit{fcdf22} function uses loop-fusion to perform the algorithm in two loops rather than in three. The first loop only loads the four necessary pixels for the immediate calculations, and then performs the high-pass and low-pass filter operations. The new pixel values are stored in a temporary array, which the second loop restores back into the image.

```c
void fcdf22(int x[], int n, int st) {
    int mid, i, si, si_next, di, di_prev;
    int temp[1000];
    si=0; si_next=0; di=0; di_prev=0;
    mid=(n/2)-1;
    for (i=0;i<=mid;i++) {
        if (i==0) { si = x[2*i*st]; } // special case when i=0
        else { si = si_next;} // load s[i]=s[i+1]
        di_prev = di; // load d[i-1]=d[i]
        di = x[(2*i*st)+st]; // load d[i]
        if (i==mid) { si_next = x[2*i*st]; } // special case when i=mid
        else { si_next = x[2*(i+1)*st]; } // load s[i+1]
        di = di + di - si - si_next; // high-pass filter
        if (i==0) { di_prev = di; } // special case when i=0
        si = si + ( (di_prev + di) >> 3 ); // low-pass filter
        temp[i] = si;
        temp[(i+mid+1)] = di;
    }
    for (i=0;i<=mid;i++) { // store pixel back into image
        x[i*st] = temp[i];
        x[(i+mid+1)*st] = temp[(i+mid+1)];
    }
}
```

Figure 3.5: The redesigned \textit{fcdf22} function by means of loop-fusion.
3.3 Wavelet Transform Algorithm in MATLAB

Currently, the MATLAB function library does not include the Wavelet Transform. As a result, a Wavelet function had to be created. The algorithm used for the function was a direct interpretation of the C language provided by Honeywell using the redesigned $fcdf22$ function as described above in section 3.2. When coding in MATLAB, it always best to avoid function calls within loops since they tend to negatively affect the timing performance. The MATLAB interpretation of the Wavelet Transform therefore incorporates $fcdf22$ inside forward_wavelet as a single function. The portion of the MATLAB code which implements the $fcdf22$ function is shown in Figure 3.6. The entire MATLAB code for the Wavelet Transform function may be found in Appendix C.

![C code for the fcdf22 function](a)

![Corresponding MATLAB code](b)

Figure 3.6: (a) C code for the $fcdf22$ function. (b) Corresponding MATLAB code.
3.4 Implementation of the Wavelet Transform on the WILDCHILD FPGA System

The implementation of the Wavelet Transform on the WILDCHILD system requires the algorithm to be rewritten in VHDL, a hardware description programming language. The translation of the Wavelet Transform algorithm from C language to Register Transfer Level (RTL) VHDL was implemented in the form of a state machine. Figure 3.7 shows a pseudo-VHDL model for the state machine implementation of the forward_wavelet function. The VHDL code for the Wavelet Transform on the WILDCHILD system may be found in Appendix C.

```
when fwd_wavelet_st0 =>
  if nt >= 128 then continue to state fwd_wavelet_st0a
  else goto state fwd_wavelet_st7
when fwd_wavelet_st0a =>
  set Tor-loop counter and counter-limit for Row operation
  set inputs to fcdf22 function
when fwd_wavelet_st1 =>
  set Return state for fcdf22 function to fwd_wavelet_st2
  if (i<nt*512), call fcdf22 function to perform the ROW operations
  goto state fcdf22_st0 [returns to fwd_wavelet_st2]
  else if fcdf22 ROW operation is complete
  continue to state fwd_wavelet3
when fwd_wavelet_st2 =>
  increment counter (i+=512) for row operation
  goto state fwd_wavelet_st1
when fwd_wavelet_st3 =>
  set Tor-loop counter and counter-limit for Column operation
  set inputs to fcdf22 function
when fwd_wavelet_st4 =>
  set Return state for fcdf22 function to fwd_wavelet_st5
  if (i<nt), call fcdf22 function to perform the COL operations
  goto state fcdf22_st0 [returns to fwd_wavelet_st5]
  else if fcdf22 COL operation is complete,
  continue to state fwd_wavelet6
when fwd_wavelet_st5 =>
  increment counter (i++) for COL operation
  goto state fwd_wavelet_st4
when fwd_wavelet_st6 =>
  divide by 2 (nt>>=1)
  loop back to state fwd_wavelet_st0 to check if (nt >= 128)
when fwd_wavelet_st7 =>
  set Interrupt signal high for 4 clock cycles
end of program
```

Figure 3.7: Pseudo VHDL model of the fwd_wavelet function.
In the main loop of the _fwd_wavelet_ state machine, it branches off to the _fcdf22_ state machine, where the pixel values are retrieved from memory and the filter operations are performed. For each set of computations, two pixel values are loaded in from memory, \( d[i] \) and \( s[i+1] \). There are two additional values used in the computations, \( d[i-1] \) and \( s[i] \), which passed down from the previous loop cycle to minimize the memory accessing, thus increasing the timing efficiency.

The _fcdf22_ function contains three main procedures. It computes the memory locations to read the \( s[i+1] \) and \( d[i] \) pixel values. Next, it calculates the new \( s[i] \) and \( d[i] \) values. Finally, it computes the memory locations to write the new values. Row and column operations have different memory read and write locations. Figures 3.8 and 3.9 describe the memory read and write cycles for the row operations and column operations respectively.

**Figure 3.8: Memory read and write for row operations (ST = 1).**
When the VHDL program is synthesized to be placed and routed onto the FPGA, the software algorithm is translated to hardware components. As an example, the FPGA will use a combination of adders/subtractors and shift registers to compute the new \(s[i]\) and \(d[i]\) values in the \(fcdf22\) function. Registers may be used to hold the \(s[i+1]\) and \(d[i]\) values until the next cycle of the main loop, where these values are passed down to \(s[i]\) and \(d[i-1]\) respectively. A graphic representation of the Wavelet architecture is shown in Figure 3.10.

\[
\begin{align*}
\text{Read Memory Location} & \quad \text{Write Memory Location} \\
\hline
i & s[i] & d[i] & s[i] & d[i] \\
0 & 0 & 512 & 0 & 131072 \\
1 & 1024 & 1536 & 512 & 131584 \\
2 & 2048 & 2560 & 1024 & 132096 \\
3 & 3072 & 3584 & 1536 & 132608 \\
4 & 4096 & 4608 & 2048 & 133120 \\
5 & 5120 & 5632 & 2560 & 133632 \\
6 & 6144 & 6656 & 3072 & 134144 \\
7 & 7168 & 7680 & 3584 & 134656 \\
8 & 8192 & 8704 & 4096 & 135168 \\
9 & 9216 & 9728 & 4608 & 135680 \\
10 & 10240 & 10752 & 5120 & 136192 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
254 & 260096 & 260608 & 130048 & 261120 \\
255 & 261120 & 261632 & 130560 & 261632 \\
\end{align*}
\]

Figure 3.9: Memory read and write for column operations (ST = 512).

Figure 3.10: One possible FPGA implementation of the \(fcdf22\) function.
The VHDL model of the Wavelet Transform was synthesized using the Synplify synthesis tool. The result produced a Xilinx netlist that required at least 480 CLBs (47%), and could potentially be clocked at 29.2 MHz. The netlist was placed and routed using the Xilinx Foundation Series 2.11 - Design Manager tools. The final result uses 612 CLBs (59%), and could potentially be clocked at 7.739 MHz.

The Wavelet function was loaded onto Processing Element 0 (PE0) of the WILDCHILD system, which has a RAM containing 512x512 addressable locations, each 32 bits wide. Although the pixel values of grayscale images are limited to 8 bits, nevertheless the output of the Wavelet Transform consists of negative values, and therefore requires a width greater than 8 bits. The upper 16 of the 32 bits were used for temporarily packing data due to insufficient RAM space. The Wavelet Transform was clocked on PE0 at 21.0 MHz.

### 3.5 Results

The performance of the Wavelet Transform was evaluated and compared between MATLAB, C, and RTL VHDL. The algorithm was performed on three distinct grayscale images, shown in Figure 3.11. Each image was of size 512x512, and the pixel values ranged from 0-255 (8 bits wide). An individual image took up the entire RAM space of PE0 on the WILDCHILD board. The average configuration and computation times for the software implementations are listed in Tables 3.1, while the times for the hardware implementation on the WILDCHILD system is listed in Table 3.2.

![Figure 3.11: Three 512x512 grayscale images - Barbara (left), Goldhill (middle), Lena (right).](image)
Table 3.1: Computation results for the Wavelet Transform implemented in C and MATLAB. Times are given in seconds.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Lena Time (s)</th>
<th>Goldhill Time (s)</th>
<th>Barbara Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Language (VMECAGE host)</td>
<td>1.170</td>
<td>1.170</td>
<td>1.170</td>
</tr>
<tr>
<td>C Language (Solaris UltraSPARC III 269 MHz Processor)</td>
<td>0.450</td>
<td>0.470</td>
<td>0.470</td>
</tr>
<tr>
<td>MATLAB (Solaris UltraSPARC III 269 MHz Processor)</td>
<td>818.129</td>
<td>833.247</td>
<td>827.350</td>
</tr>
</tbody>
</table>

Table 3.2: Hardware implementation of the Wavelet Transform performed on the WILDCHILD multi-FPGA System at 21 MHz. Times are given in seconds.

<table>
<thead>
<tr>
<th>WILDCHILD multi-FPGA System</th>
<th>Lena Time (s)</th>
<th>Goldhill Time (s)</th>
<th>Barbara Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA Configuration Time</td>
<td>0.1650</td>
<td>0.1650</td>
<td>0.1850</td>
</tr>
<tr>
<td>Memory Loading Time</td>
<td>0.0350</td>
<td>0.0700</td>
<td>0.0700</td>
</tr>
<tr>
<td>Computation Time</td>
<td>0.0022</td>
<td>0.0022</td>
<td>0.0022</td>
</tr>
<tr>
<td>Memory Read-back Time</td>
<td>0.0850</td>
<td>0.0700</td>
<td>0.0750</td>
</tr>
<tr>
<td><strong>Total Execution Time</strong></td>
<td><strong>0.2872</strong></td>
<td><strong>0.3072</strong></td>
<td><strong>0.3322</strong></td>
</tr>
</tbody>
</table>

It is clear that the hardware computation times on the FPGA are negligible compared to its configuration, memory load and read-back times. Moreover, it is not surprising that the total hardware execution times showed performance improvements in orders of magnitude higher than the computation times of MATLAB and C on the host processor and the Solaris machine. This clearly displays the efficiency in the performance of Wavelet Transform through a hardware emulation over that of software.

3.6 Summary

This chapter described the implementation of the Wavelet Transform, a compression algorithm widely used in image processing applications. A comparison in the performance of the Wavelet Transform was evaluated on three different platforms: (1) implementation in MATLAB on a SUN workstation (2) implementation in C on a SUN workstation (3) implementation using VHDL on Xilinx FPGAs on the WILDCHILD reconfigurable computing platform. It was demonstrated that there exists the possibility to obtain performance improvements between a general-purpose processor and special purpose hardware.
This chapter describes the effort in the development of the Matrix Multiplication function for the MATLAB libraries on the WILDCHILD and WILDSTAR multi-FPGA systems. An overview of the function is provided, followed by a description of the architecture design of the function for a single and parallel processor implementation on the WILDCHILD and WILDSTAR boards. Results are provided for the single and parallel implementations on both boards. A comparison of results of the algorithm on different platforms is also provided.

4.1 Function Description

Matrix Multiplication is a function widely used in many applications, ranging from simple matrix computations to complex signal processing algorithms. The function is defined as \( C = A \times B \) with complexity \( O(n^3) \), as illustrated in Figure 4.1. Point-wise multiplication is performed on each row \( (i) \) of Matrix A and column \( (j) \) of Matrix B. The result is then stored in position \( (i,j) \) of Matrix C. When performing Matrix Multiplication, matrix dimensions must be compatible. Hence, the row size of Matrix A must be the same as the column size of Matrix B.

![Figure 4.1: (a) Matrix Multiplication. (b) Row-column operation for Matrix Multiplication.](image-url)
4.2 Matrix Multiplication on the WILDCHILD System

This section discusses the implementation of the Matrix Multiplication function using single and parallel processors on the WILDCHILD system. It has been included in the work as a benchmark for comparison with the WILDSTAR system. All VHDL designs, figures, tables and information regarding Matrix Multiplication on the WILDCHILD board were provided by Alex Jones [20]. This section will summarize his work, providing an architectural description of the two implementations, performance results, and a comparison in of the two processes.

4.2.1 Matrix Multiplication Using a Single Processor

The FPGA on the WILDCHILD system that was chosen to run the Matrix Multiplication design was PE0. There were several reasons for this choice. Assuming 16-bit input data and 16-bit output data, PE0’s 32-bit memory pipe, allows for access to two 16-bit data points per cycle and potential speed optimizations for the design. Additionally, the computation requires a RAM large enough to hold all three matrices. Consequently, PE0’s RAM is twice as large as that of PEX, maximizing the size of computation possible for the single processor design. Finally, it was determined that the design would not fit on a PEX slave. However, the size of the master FPGA chip is more that 2½ times larger than that of a slave FPGA, making it the most optimal choice for this design. The single-processor architecture for this function is displayed in Figure 4.2.

![Matrix Multiplication Diagram](image)

Figure 4.2: Single processor Matrix Multiplication implemented on PE0 of the WILDCHILD board.
4.2.1.1 Architecture for the Single Processor

The PE0 memory is divided into four different sections. The first section is comprised of the first four memory addresses 0-3, and contains the number of rows and columns of the two matrices. The next three sections are of equal size, and each holds one of the matrices in the computation, as shown in Figure 4.2 above. The first one contains Matrix A and uses addresses 0x4-0x15557; the second one contains Matrix B and uses addresses 0x15558-0x2AAAA; the final section contains the result, Matrix C, and uses addresses 0x2AAAB-0x3FFFF.

As mentioned earlier, each data point is 16-bits wide, while the pipe to the memory is 32-bits wide. To maximize the utilization of the RAM, each memory address will store two data points in the 32-bit space. This entitles two data points to be read in a single clock cycle. Consequently, each time an address is read, one data point is used and the other is discarded.

In order improve the performance, Matrix A is stored in row-major order while Matrix B is stored in column-major order. This allows both matrices to be read in sequential order when performing a row-column operation so all data points are used in each clock cycle. During clock cycle one, elements $a_{11}$ and $a_{12}$ are read and stored in registers. During clock cycle two, elements $b_{11}$ and $b_{21}$ are read. Element $b_{11}$ is sent directly to the multiply accumulator along with $a_{11}$ a register, while element $b_{21}$ is stored in a register. In the following cycle $a_{12}$ and $b_{21}$ are now sent to the multiply accumulator, while elements $a_{13}$ and $a_{14}$ are read and stored in registers. In the fourth clock cycle, elements $b_{31}$ and $b_{41}$ are read; the former is sent directly to the multiply accumulator along with $a_{13}$, while the latter is stored in a register. This process continues until an entire row has been multiplied by an entire column, after which the value is written to memory. It is now possible to see that with the exception of the first clock cycle, the multiply accumulator has been in use every subsequent cycle, increasing the efficiency by a factor of two.

The design begins by reading the row and column sizes from addresses 0-3 in memory. It then proceeds to alternately read data from the first row of Matrix A and the first column of Matrix B. Each cycle, with the exception of the first, values are multiplied and accumulated in a result register. When all values of the row and column have been used, the total value is written back to memory, and the design increments the rows of Matrix A while continuing with the first column of Matrix B. This allows the result data to be created and easily stored in column major order. The computation continues cycling through all the rows of Matrix A, and then
returns to the first row and begins working on the second column of Matrix B. The design continues until all the rows and columns have been multiplied through.

In order to properly store the 16-bit result back into the 32-bit memory space while preserving the other 16-bit value that shares the same memory address, it was necessary to first read in the data from the address. The 16-bits to be preserved and the new result would be joined and written back to memory as a 32-bit value. The extra read is in actuality a small performance cost compared to the already heavy performance penalty. Once the entire computation has completed, PE0 raises its interrupt to signal to the host that the process has concluded.

4.2.1.2 Performance

The design was tested with several different matrix sizes, as listed in Table 4.1, and it was verified to run correctly at 29 MHz for data sizes up to 420x420 elements. Timing results were broken down. Load Design refers the time required to load the Matrix Multiplication design onto the WILDCHILD board. Prepare Data refers to the time to convert the input data into a form usable for the application. Load Data is the time required to load the data into the FPGA’s local RAM. Compute Time is the time taken from start to finish of the computation on WILDCHILD. Retrieve Data is the time taken to retrieve the data from the master FPGA’s local RAM. It is important to note that the execution time is dominated by the computation time for large datasets. For small datasets, the execution time is dominated by the design loading time and data preparation time.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Load Design</th>
<th>Prepare Data</th>
<th>Load Data</th>
<th>Compute Time</th>
<th>Retrieve Data</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.170</td>
<td>0.033</td>
<td>0.010</td>
<td>0.011</td>
<td>0.003</td>
<td>0.227</td>
</tr>
<tr>
<td>128x128</td>
<td>0.169</td>
<td>0.146</td>
<td>0.034</td>
<td>0.077</td>
<td>0.013</td>
<td>0.439</td>
</tr>
<tr>
<td>256x256</td>
<td>0.172</td>
<td>0.615</td>
<td>0.126</td>
<td>0.597</td>
<td>0.059</td>
<td>1.569</td>
</tr>
<tr>
<td>416x416</td>
<td>0.165</td>
<td>1.655</td>
<td>0.324</td>
<td>2.524</td>
<td>0.155</td>
<td>4.823</td>
</tr>
</tbody>
</table>
The data in Table 4.1 and Figure 4.3 suggests that the design loading time is constant regardless of the matrix sizes used in the computation, while the other values increase linearly with matrix size. This result could be predicted since each operation is $O(n)$ where $n$ is the number of elements in the matrix.

### 4.2.2 Matrix Multiplication Using Parallel Processors

This section describes the implementation of a 16-bit fixed-point Matrix Multiplication function on the WILDCHILD board using all nine processors. There are two variations on the design with regards to the resulting Matrix C. The first design outputs 32-bit fixed-point values for Matrix C, and the second outputs 16-bit fixed-point values.

In both implementations, Matrix A may contain up to 524,256 elements; Matrix B may contain up to 1048512, assuming row numbers divisible by 8 and 16-bit output precision; Matrix C may contain 1048576 elements with similar restrictions as Matrix B, or 524,288 elements assuming 32-bit output precision. If we assume square matrices, this results in a maximum computation size of 724x724 for all three matrices.
The matrices are distributed in the memories as follows: Assuming the computation written in the form $C = A \times B$, the entire Matrix A is stored in row-major order in the memory of the master FPGA, PE0. Matrices B and C are distributed in column-major order among the eight slave processors such that each processor contains $c/N$ columns, where $c$ is the total number of columns and $N$ is the number of slave processors. The data is distributed in block fashion, such that PE1 contains the first $c/N$ columns, PE2 contains the next $c/N$ columns, and so on. To perform the computation, the master must broadcast a single data point to each slave every cycle that can be used by all the slaves. Figure 4.4 illustrates the architecture of the parallel Matrix Multiplication function on the WILDCHILD board.

4.2.2.1 Architecture for Parallel Processors

As the data layout suggests, the Matrix Multiplication design on multiple processors uses a master/slave configuration. Thus, only two designs needed to be developed, one for the PE0 and one for all of the PEXs. The crossbar is set to continual broadcast mode, while the handshaking lines are used for processor synchronization. The systolic connections are not used.

The PE0 design begins by reading from its local memory several initialization parameters about the computation, such as the number of columns and
rows of the matrix stored in memory and the maximum number of columns each particular slave might have. It then waits to receive a signal from each PEX slave via the handshaking lines to assert that they are all ready to begin computations. Once all the slaves are accounted for, the master sends a signal back to all of the slaves simultaneously to begin the computation. The processors are now synchronized.

The master proceeds to read each element in the rows of Matrix A from its memory and sends this data to the crossbar, which is set to continuous broadcast mode. This mode allows each of the slave processors to simultaneously acquire the data within one clock cycle from the master. The master continues to send each row element on the crossbar until it has sent an entire row. At this point, the slaves have the opportunity to operate an entire row of Matrix A with an entire row of Matrix B and write the result to their local memory. The write-result procedure requires an explicit number of clock cycles, so the master is able to stall for that time frame until the write cycle is completed. This allows the FPGAs to remain synchronized without the need for handshaking signals. Once the results of Matrix C have been written to memory, PE0 begins sending the next row of Matrix A on to the crossbar. This process continues until the master has sent all of the rows in the matrix to slaves. The master then repeats the process of sending the entire matrix in the same fashion, according to the number of Matrix B columns distributed among the slaves. When the entire process is finished, PE0 asserts its interrupt to the host to signal that the computation is complete. The host may now read the resulting Matrix C back from the WILDCHILD board.

The slaves operate similar to PE0 above. Each PEX begins by reading several initialization parameters from its local memory. These parameters include the number of columns in Matrix B stored in memory, the length of the columns, and the number of rows in Matrix A. Each slave then sends a signal to the master on its handshaking line that it is ready and waits for the response from the master to proceed with the computation.

In each cycle the slave reads a Matrix B value from its local memory and receives a value from Matrix A over the crossbar. It multiplies these values together and accumulates them in a register. Once an entire row-column operation is complete, the slave writes back the resulting value to memory. It then returns to the beginning of the same column in Matrix B and begins operating with the next row from Matrix A. Once all the rows of Matrix A have been broadcasted, Matrix B is advanced to the next column, while the master begins sending each row in
Matrix A from the beginning. The entire process is completed once all columns of Matrix B located in each PEX slave has been multiplied by each row of Matrix A.

### 4.2.2.2 Performance

Two distinct designs have been created to handle different output precisions. The synthesized design for 32-bit output precision resulted in a Xilinx netlist that required 315 CLBs or 79% of the total chip area. The synthesized design for 16-bit output precision resulted in a Xilinx netlist that required 303 CLBs or 76% of the total chip area. Both versions of the design were tested with several different matrix sizes. They were verified to run correctly at 26 MHz for data sizes up to 720x720 elements. Timing results were broken down and are listed in Tables 4.2 and 4.3.

**Table 4.2:** Performance of Matrix Multiplication using parallel processors on the WILDCHILD system. Data uses 16-bit fixed-point input and 32-bit fixed-point output. Times are listed in seconds.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Load Design</th>
<th>Prepare Data</th>
<th>Load Data</th>
<th>Compute</th>
<th>Retrieve Data</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.1710</td>
<td>0.0350</td>
<td>0.0150</td>
<td>0.0000</td>
<td>0.0110</td>
<td>0.2320</td>
</tr>
<tr>
<td>128x128</td>
<td>0.1710</td>
<td>0.1710</td>
<td>0.0450</td>
<td>0.0100</td>
<td>0.0300</td>
<td>0.4270</td>
</tr>
<tr>
<td>256x256</td>
<td>0.1730</td>
<td>0.7170</td>
<td>0.1610</td>
<td>0.0820</td>
<td>0.1230</td>
<td>1.2560</td>
</tr>
<tr>
<td>512x512</td>
<td>0.1730</td>
<td>2.9130</td>
<td>0.6270</td>
<td>0.6520</td>
<td>0.4960</td>
<td>4.8610</td>
</tr>
<tr>
<td>720x720</td>
<td>0.1709</td>
<td>5.8091</td>
<td>1.2300</td>
<td>1.8155</td>
<td>0.9782</td>
<td>10.0037</td>
</tr>
</tbody>
</table>

**Table 4.3:** Performance of Matrix Multiplication using parallel processors on the WILDCHILD system. Data uses 16-bit fixed-point input and 16-bit fixed-point output. Times are listed in seconds.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Load Design</th>
<th>Prepare Data</th>
<th>Load Data</th>
<th>Compute</th>
<th>Retrieve Data</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.1680</td>
<td>0.0350</td>
<td>0.0140</td>
<td>0.0020</td>
<td>0.0090</td>
<td>0.2280</td>
</tr>
<tr>
<td>128x128</td>
<td>0.1670</td>
<td>0.1530</td>
<td>0.0430</td>
<td>0.0120</td>
<td>0.0160</td>
<td>0.3910</td>
</tr>
<tr>
<td>256x256</td>
<td>0.1630</td>
<td>0.6410</td>
<td>0.1560</td>
<td>0.0840</td>
<td>0.0640</td>
<td>1.1080</td>
</tr>
<tr>
<td>512x512</td>
<td>0.1640</td>
<td>2.7040</td>
<td>0.6230</td>
<td>0.6530</td>
<td>0.2450</td>
<td>4.3890</td>
</tr>
<tr>
<td>720x720</td>
<td>0.1718</td>
<td>5.1964</td>
<td>1.2218</td>
<td>1.8082</td>
<td>0.4864</td>
<td>8.8846</td>
</tr>
</tbody>
</table>

Figures 4.5 and 4.6 show how different portions of the computation time of the design are related to matrix size. All of the computation times other than design loading time increase linearly with matrix size. This result could be predicted since each operation is O(n) where n is the number of elements in the matrix. Both designs produce similar performance except that the retrieve data portion of the 16-
bit output is half the time as the 32-bit output version, most likely due to the fact that the amount of data points to be transfer is doubled for the 32-bit version. It is important to note that the total execution is not dominated by the computation time. For small datasets, the execution time is dominated by the design loading time; for larger datasets, the execution time is dominated by the data preparation time.

Figure 4.5: Performance versus time for parallel Matrix Multiplication on WILDCHILD with 16-bit fixed-point input and 32-bit fixed point output.

Figure 4.6: Performance versus time for parallel Matrix Multiplication on WILDCHILD with 16-bit fixed-point input and 16-bit fixed point output.
4.2.3 Single Processor vs. Parallel Processors on WILDCHILD

As apparent in Table 4.4, there is a consistent improvement of the Matrix Multiplication implementation on the WILDCHILD board with the single processor design to the multi-processor design. The first result is discarded since the timing precision was too small to obtain an accurate timing. However, the other three data sizes show an average speed factor of 7.4, which is very close to the linear speed-up from one to eight processors.

Table 4.4: Performance comparison between single and multiple processor designs on the WILDCHILD board. Times are given in seconds.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Single FPGA</th>
<th>Multiple FPGAs</th>
<th>Speed Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.011</td>
<td>0.0000</td>
<td>∞</td>
</tr>
<tr>
<td>128x128</td>
<td>0.077</td>
<td>0.0100</td>
<td>7.700</td>
</tr>
<tr>
<td>256x256</td>
<td>0.597</td>
<td>0.0820</td>
<td>7.280</td>
</tr>
<tr>
<td>416x416</td>
<td>2.524</td>
<td>0.3510</td>
<td>7.191</td>
</tr>
</tbody>
</table>

These results can be understood from the structure of the two designs. In the single processor design, PE0 is able to read from memory two values of Matrix A or Matrix B each clock cycle with its 32-bit wide RAM bandwidth. Similarly, in the parallel processor design, each PEX is able to read a Matrix B value from its local memory every clock cycle, while simultaneously receiving a Matrix A value from PE0. In both instances, data pipelining allows the processors to perform a multiply-accumulate every clock cycle, while maximize its resources and obtaining close to linear speed-ups.

4.3 Matrix Multiplication on the WILDSTAR System

This section discusses the implementation of the Matrix Multiplication function using single and parallel processors on the WILDSTAR system. Architectural descriptions of the two designs are provided, as well as timing results and a comparison in performance of the two processes. The VHDL and Host code for the single and parallel processor Matrix Multiplication designs may be found in Appendix D. The core architectures for PEX and PE0 are located in Appendix B.
4.3.1 Matrix Multiplication Using a Single Processor

Since all three FPGAs on the WILDSTAR board are of the same architecture, any one of them could have been used for the single processor Matrix Multiplication design. However, PEX was chosen for were several reasons. Both FPGAs (PE1 and PE2) are connected to two local 1MB RAMs containing $2^{18}$ addressable spaces of size 32-bits wide. These provide two optimal locations to store Matrices A and B with 32-bit precision input data. PEX is also connected to two Mezzanine memory cards, each containing two 2MB RAMs with $2^{18}$ addressable spaces of size 64-bits wide. The output may be stored in one of these Mezzanine memories, taking advantage of the 64-bit precision for the output data. If PE0 were chosen, it would be required to store all data in the Mezzanine memories using 64-bit precision. For most applications, 32-bit precision is quite sufficient, and any unnecessary increase in the data width may result in larger designs and slower computations. The single-processor design was implemented on PE1, though it may have just as well been implemented on PE2. The crossbar mode is set so that PE1 has access to MEM 1 of its right Mezzanine memory card. The architecture for this design is displayed in Figure 4.7.

![Diagram of single processor Matrix Multiplication](image)

**Figure 4.7**: Single processor Matrix Multiplication implemented on PE1 of the WILDSTAR board.
4.3.1.1 Architecture for the Single Processor

The WILDSTAR system has an abundance of memory storage space, which allows a matrix to occupy an entire RAM. As shown above in Figure 4.7, Matrix A is stored in the left local memory, Matrix B is stored in the right local memory, and the resulting Matrix C is stored in MEM 1 of the right Mezzanine memory card. Assuming square matrices, each RAM can hold a maximum size matrix of 512x512, with 32-bits wide for input data and 64-bits wide for output data. The first two addresses in each RAM, 0-1, contain the row and column sizes for the matrix stored in that RAM.

An advantage of each FPGA having access to several memories is that they are able to acquire data elements from different memories simultaneously. This advantage surely decreases the memory read time, and in effect decreases the total computation time of an application. To simplify this task, a process was created for each local and Mezzanine RAM to automate the reading and writing to the memories. These concurrent processes can all run in parallel, thus allowing simultaneous reads and writes from the different memories. The VHDL code for these processes may be found in Appendix D.

The Matrix Multiplication design begins by simultaneously reading in the row sizes of Matrix A and Matrix B in address 0 of the left and right local memories respectively. It then proceeds to read in the column sizes for the two matrices in address 1 of the two local memories. Immediately, it determines the dimensions of the resulting Matrix C, and writes the row and column values to addresses 0 and 1 of the right Mezzanine memory.

Once the matrix dimensions are known, PEX proceeds to simultaneously read data from the first row of Matrix A and the first column of Matrix B. Each set of values are multiplied and accumulated in a result register of 64-bits wide. When all values of the row and column have been used, the total value is written to the corresponding address in the Mezzanine memory. The design increments the row of Matrix A while continuing with the first column of Matrix B. The computation continues cycling through all the rows of Matrix A, and then returns to the first row and begins working on the second column of Matrix B. This procedure continues until all the rows and columns have been multiplied through. Once the entire computation has completed, PEX asserts its interrupt register high to signal the host that the process has concluded.
4.3.1.2 Performance

The synthesized design resulted in a Xilinx netlist that required 3435 LUTs (4 LUTs per CLB), or 35% of the total chip area. The design was tested with the matrix sizes listed in Table 4.5, and it was verified to run correctly at 50 MHz for data sizes up to 510x510 elements. Timing results were broken down and plotted in Figure 4.8. *Configure System* refers to the time required to configure the WILDSTAR board and load the design onto the FPGA. *Prepare Data* is the time required to prepare the data to be written to memory. *Load Data* is the loading time of the data into the FPGA’s local RAMs. *Compute Time* is the time taken from start to finish of the computation on WILDSTAR board. *Retrieve Data* is the time taken to retrieve the result data from the right Mezzanine memory.

Table 4.5: Performance results of the Matrix Multiplication function using a single processor on the WILDSTAR system. Times are listed in seconds.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Configure System</th>
<th>Prepare Data</th>
<th>Load Data</th>
<th>Compute Time</th>
<th>Retrieve Data</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.480</td>
<td>0.180</td>
<td>0.010</td>
<td>0.079</td>
<td>0.020</td>
<td>0.769</td>
</tr>
<tr>
<td>128x128</td>
<td>0.470</td>
<td>0.750</td>
<td>0.040</td>
<td>0.630</td>
<td>0.090</td>
<td>1.980</td>
</tr>
<tr>
<td>256x256</td>
<td>0.450</td>
<td>2.940</td>
<td>0.150</td>
<td>5.030</td>
<td>0.350</td>
<td>8.920</td>
</tr>
<tr>
<td>416x416</td>
<td>0.460</td>
<td>7.780</td>
<td>0.390</td>
<td>21.550</td>
<td>0.930</td>
<td>31.110</td>
</tr>
<tr>
<td>510x510</td>
<td>0.480</td>
<td>12.270</td>
<td>0.580</td>
<td>39.730</td>
<td>1.390</td>
<td>54.450</td>
</tr>
</tbody>
</table>

Figure 4.8: Performance results versus time for the Matrix Multiplication function using a single processor on the WILDSTAR system.
The data in Table 4.5 and Figure 4.8 suggests that all the timing values increase linearly with matrix size, except the configuration time, which is relatively constant since it is independent of the data size. This result could be predicted since each operation is $O(n)$ where $n$ is the number of elements in the matrix. It is important to note that the execution time is dominated by the computation time for large datasets. However, for small datasets the execution time is dominated by the system configuration and data preparation time.

4.3.2 Matrix Multiplication Using Parallel Processors

This section describes the implementation of a 32-bit fixed-point matrix multiplication function on the WILDSTAR system using two processors, PEX and PE0. All data is stored in the left and right Mezzanine memories, and PEX’s local memories are not used in this design. The crossbar mode is set such that PEX has access to MEM 1 of the left and right Mezzanine memories, while PE0 has access to MEM 0 of both memories. Figure 4.9 illustrates the architecture of the parallel Matrix Multiplication function on the WILDSTAR board.

![Figure 4.9: Parallel Matrix Multiplication implemented on PE1 and PE0 of the WILDSTAR board.](image)
4.3.2.1 Architecture for Parallel Processors

In this implementation, the entire Matrix B is stored in both MEM 0 and MEM 1 of the PEX’s right Mezzanine memory, allowing simultaneous access for both PE0 and PEX respectively. Matrix A is distributed row-wise, half is stored in MEM 0 and half is stored in MEM 1 of PEX’s left Mezzanine memory in addresses 0x00000 to 0x1FFFF. This provides PEX and PE0 access to only the necessary portion of Matrix A for their computation. The two FPGAs will each perform half of the computation $C = A \times B$, and each half of the resulting Matrix C is stored in the corresponding RAM of PEX’s left Mezzanine memory in addresses 0x20000 to 0x3FFFF. The first two addresses in each RAM where a matrix is stored contains the row and column sizes for that matrix. Since each RAM in the left Mezzanine memories contain only half of Matrix A, the column size value stored in address 1 will be half of the actual size of the column.

Assuming square matrices, Matrix A, B and C may individually contain up to 512x512 elements with 32-bit precision. Although the memory architecture allows for 64-bit precision for both input and output data, nevertheless 32-bit precision was chosen since PE0’s access port to the Mezzanine memories is limited to 32-bits wide. Alternatively, the design could have been implemented on both PEXs rather than on PE0. This would eliminate the port size constraint of PE0 and allow the design to take advantage of the full 64-bit bandwidth for input and output data.

As described in section 4.3.1.1, several concurrent processes were created for the PEX architecture to assist in accessing the different memories in parallel. These processes were extended for the architecture of PE0 to allow its data access of the Mezzanine memories in parallel. The VHDL code for these processes may be found in Appendix D.

The architecture of the parallel Matrix Multiplication design on the WILDSTAR board allows both processing elements to work independently and without the need for synchronization. The reason this is possible is because each FPGA’s memories hold their own copy of Matrix B, and each half of Matrix A can actually be regarded as an entire matrix. Data sharing is thus unnecessary.

While PEX and PE0 work independently, they share the same algorithm, similar to that of the single processor design. The parallel Matrix Multiplication design begins by each of the two processors simultaneously reading in the row sizes of Matrix A and Matrix B in address 0 of the left and right Mezzanine memories.
respectively. Next, they proceed to read in the column sizes for the two matrices in address 1 of the two Mezzanine memories. Immediately, they determine the dimensions of the resulting Matrix C, and write the row and column values to addresses 0x20000 and 0x20001 of the right Mezzanine memories.

Once the matrix dimensions are known, PEX and PE0 proceed to simultaneously read data from the first row of Matrix A and the first column of Matrix B in their corresponding memories. Each set of values are multiplied and accumulated in a result register of 64-bits wide. When all values of the row and column have been used, the bottom 32-bits of the total value is written to the corresponding Matrix C address in the Mezzanine memories. Each PE increments the row of Matrix A while continuing with the first column of Matrix B. The computation continues cycling through all the rows of Matrix A, and then returns to the first row and begins working on the second column of Matrix B. This procedure continues until all the rows and columns have been multiplied through. Once the entire computation has completed, each processor asserts its interrupt register high to signal the host that its process has concluded.

4.3.2.2 Performance

Two individual designs were implemented, one for PEX and one for PE0. The synthesized architecture on PE0 resulted in a Xilinx netlist that required 2696 LUTs (4 LUTs per CLB), or 28% of the total chip area; the architecture on PEX resulted in a Xilinx netlist that required 2455 LUTs, or 25% of the total chip area. The design was tested with several different matrix sizes, as listed in Table 4.6, and it was verified to run correctly at 50 MHz for data sizes up to 510x510 elements. Timing results were broken down and plotted in Figure 4.10. Configure System refers to the time required to configure the WILDSTAR board and load the design onto the FPGA. Prepare Data is the time required to prepare the data to be written to memory. Load Data is the loading time of the data into the FPGA’s local RAMs. Compute Time is the time taken from start to finish of the computation on WILDSTAR board. Retrieve Data is the time taken to retrieve the result data from the right Mezzanine memory.

The data in Table 4.6 and Figure 4.10 suggests that all the timing values increase linearly with matrix size, except the configuration time, which is relatively constant since it is independent of the data size. This result could be predicted since each operation is O(n) where n is the number of elements in the matrix. It is important to note that the execution time is dominated by the computation time for
large datasets. However, for small datasets the execution time is dominated by the system configuration and data preparation time. These results are consistent with that of the single processor design.

Table 4.6: Performance results of the Matrix Multiplication function using parallel processors on the WILDSTAR system. Times are listed in seconds.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Configure System</th>
<th>Prepare Data</th>
<th>Load Data</th>
<th>Compute Time</th>
<th>Retrieve Data</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.490</td>
<td>0.210</td>
<td>0.020</td>
<td>0.040</td>
<td>0.010</td>
<td>0.770</td>
</tr>
<tr>
<td>128x128</td>
<td>0.500</td>
<td>0.810</td>
<td>0.080</td>
<td>0.316</td>
<td>0.060</td>
<td>1.766</td>
</tr>
<tr>
<td>256x256</td>
<td>0.500</td>
<td>3.040</td>
<td>0.300</td>
<td>2.520</td>
<td>0.270</td>
<td>6.630</td>
</tr>
<tr>
<td>416x416</td>
<td>0.500</td>
<td>8.380</td>
<td>0.790</td>
<td>10.790</td>
<td>0.700</td>
<td>21.160</td>
</tr>
<tr>
<td>510x510</td>
<td>0.510</td>
<td>12.130</td>
<td>1.200</td>
<td>19.850</td>
<td>1.060</td>
<td>34.750</td>
</tr>
</tbody>
</table>

Figure 4.10: Performance results versus time for the Matrix Multiplication function using parallel processors on the WILDSTAR system.

4.3.3 Single Processor vs. Parallel Processors on WILDSTAR

As apparent in Table 4.7, there is a consistent improvement of the Matrix Multiplication implementation on the WILDSTAR board with the single processor design to the multi-processor design. All four data sizes show an average speed
factor close to 2.0, which is very close to the linear speed-up from one to two processors.

These results can be understood from the structure of the two designs. In the single processor design, PEX reads from memory a Matrix A and Matrix B value simultaneously, performs the multiply-accumulate operation, and then writes the data to memory. Similarly, in the parallel processor design, both PEX and PE0 perform the same operation as in the single processor design on only half the matrix. This cuts each processor’s amount of work in half, while allowing them to work in parallel. Thus, the design is able to obtain close to linear speed-ups.

Table 4.7: Performance comparison between single and multiple processor designs on the WILDSTAR board. Times are given in seconds.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Single FPGA</th>
<th>Multiple FPGAs</th>
<th>Speed Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.079</td>
<td>0.040</td>
<td>1.975</td>
</tr>
<tr>
<td>128x128</td>
<td>0.630</td>
<td>0.316</td>
<td>1.994</td>
</tr>
<tr>
<td>256x256</td>
<td>5.030</td>
<td>2.520</td>
<td>1.996</td>
</tr>
<tr>
<td>416x416</td>
<td>21.550</td>
<td>10.790</td>
<td>1.997</td>
</tr>
<tr>
<td>510x510</td>
<td>39.730</td>
<td>19.850</td>
<td>2.000</td>
</tr>
</tbody>
</table>

4.4 Matrix Multiplication Performance Comparisons

Several different implementations of the Matrix Multiplication function were tested for comparison in efficiency. These implementations include the host processor using RT Express from ISI [23], a single DSP and four DSPs in parallel by Nayak [24], a single processor and eight processors of an SGI Origin, a single FPGA and eight FPGAs in parallel on the WILDCHILD system by Jones [20], and finally a single FPGA and two FPGAs in parallel on the WILDSTAR system. Performance results for these implementations are listed in Table 4.8. The two unavailable data sets on the DSPs for matrix sizes of 416x416 occur because those data sizes were too large for that platform. Figure 4.11 and 4.12 shows graphically the performance differences between the different implementations using single and parallel processors respectively.

As displayed in Figures 4.11 and 4.12, the WILDSTAR designs are surprisingly the slowest of the single and parallel processor implementations. It is highly likely that these results are due to the fact that data pipelining cannot be
implemented on this board, as explained earlier in section 2.2.2.2. The cost of the extra clock cycles for each read and write has created a dramatic effect on the performance of the system, even while using parallel implementation. One can see the difference in performance between the WILDCHILD and WILDSTAR multi-FPGA systems. Obviously, the FPGAs of the latter system are far more superior as compared to that of the former with regard to speed and size. Nevertheless, the WILDCHILD designs showed far better results than the WILDSTAR designs due to these pipelining issues. However, it is quite possible that with the development of data pipelining for the WILDSTAR memories, it may successfully produce the most efficient results as compared to all the other platforms.

Table 4.8: Performance Comparison of Matrix Multiplication Algorithms [20].

<table>
<thead>
<tr>
<th>Platform</th>
<th>64x64</th>
<th>128x128</th>
<th>256x256</th>
<th>416x416</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT Express 85 MHz</td>
<td>0.360</td>
<td>2.350</td>
<td>16.480</td>
<td>21.290</td>
</tr>
<tr>
<td>Single DSP 60 MHz</td>
<td>0.090</td>
<td>0.640</td>
<td>3.660</td>
<td>X</td>
</tr>
<tr>
<td>Multiple DSPs 60 MHz</td>
<td>0.051</td>
<td>0.242</td>
<td>1.440</td>
<td>X</td>
</tr>
<tr>
<td>SGI Origin Single Node</td>
<td>0.010</td>
<td>0.133</td>
<td>1.001</td>
<td>5.228</td>
</tr>
<tr>
<td>SGI Origin Eight Node</td>
<td>0.011</td>
<td>0.036</td>
<td>0.177</td>
<td>0.666</td>
</tr>
<tr>
<td>WILDCHILD Single 29 MHz</td>
<td>0.011</td>
<td>0.077</td>
<td>0.597</td>
<td>2.524</td>
</tr>
<tr>
<td>WILDCHILD Parallel 26 MHz</td>
<td>0.000</td>
<td>0.010</td>
<td>0.082</td>
<td>0.351</td>
</tr>
<tr>
<td>WILDSTAR Single 50 MHz</td>
<td>0.079</td>
<td>0.630</td>
<td>5.030</td>
<td>21.550</td>
</tr>
<tr>
<td>WILDSTAR Parallel 50 MHz</td>
<td>0.040</td>
<td>0.316</td>
<td>2.520</td>
<td>10.790</td>
</tr>
</tbody>
</table>

Figure 4.11: Performance Comparison of Matrix Multiplication Algorithms on a single processor.
RT Express has an interesting behavior. There is a distinct bend in the trend at the 416x416 matrix size. It appears that at this point the processor may have started to have to use virtual memory slowing the computation down significantly. Otherwise the performance might have started to over take the WILDSTAR single processor results. With regards to the DSP results, the hardware and memory should have been able to support much higher matrix sizes than are shown. Unfortunately, the results could not be obtained. The WILDCHILD single and parallel implementations produced the best results compared to all the platforms in the MATCH testbed [20].

4.5 Summary

This chapter described the efforts in the development of the Matrix Multiplication function for the MATLAB libraries on the WILDCHILD and WILDSTAR multi-FPGA systems. An overview of the function was provided, followed by architectural descriptions, performance results and comparisons of the design implemented on single and parallel processors on both systems. A performance comparison of the Matrix Multiplication algorithm on different platforms of the MATCH testbed was also provided. The WILDSTAR system exhibited the worst results most likely due to its lack in ability to pipeline data.
This chapter describes the adaptation of the MATCH compiler to target the WILDSTAR board based on the work of Haldar and Nayak for the WILDCHILD compiler [1,14,25]. A brief overview of the compiler and its basic functionality is provided. The modifications and additions made to the WILDCHILD compiler to conform to the WILDSTAR system is discussed, followed by an evaluation of the compiler’s MATLAB generated Matrix Multiplication design in comparison with the hand-coded designs. Finally, the performance results of several benchmark designs in MATLAB that were compiled to VHDL for the WILDCHILD and WILDSTAR boards are compared and evaluated.

5.1 An Overview of the MATCH Compiler

It seems apparent that the development of efficient code for high-speed heterogeneous computing systems can be a long and arduous task, often followed by the frustration of debugging the designs. These restrictions make the systems difficult to use for scientists and engineers. However, with the assistance of the MATCH compiler, which automatically translates high-level applications written in MATLAB to distributed heterogeneous computing systems, such as FPGAs, DSPs and embedded processors, much of the burden is alleviated.

There are several reasons why MATLAB was chosen as the source language for the compiler. MATLAB is more intuitive and easier to learn to operate than C language, which reduces development time by orders of magnitude. Since MATLAB is already a popular language for image and signal processing applications and contains a rich set of library functions for these purposes, it makes this language the optimal tool for such designs. Furthermore, since most MATLAB
computations are expressed as matrix operations, which are very capable of parallelization, it presents the potential for parallelism in the compiled VHDL model [25].

Figure 5.1: Synthesis flow of the MATCH compiler [25].

Figure 5.1 shows the overview of the synthesis process from MATLAB to RTL VHDL. The front end parses the input MATLAB program and builds a MATLAB Abstract Syntax Tree (AST). The input code may contain directives set by the user, such as type, shape and precision of arrays that cannot be inferred. They begin with a `%%match` parameter, which is considered a comment in MATLAB, but in actuality hides important data for the compiler. The directives are attached to the AST nodes as annotations, which allows the user to directly modify and optimize the configuration and architecture of the VHDL code during compilation. This is followed by the type-shape analysis phase, which infers the type and shape of variables in the program that are not set by user directives. The scalarization phase expands operations on matrices into loops. If optimized library functions are available for these operations, they are used in place of scalarization. Levelization breaks down complex statements into simpler ones. This allows the VHDL AST to be constructed, as well as hardware related optimizations. Precision analysis is an inference scheme used to minimize the number of bits required in representing each variable in the AST. Transformations are then performed on the AST to optimize according to the memory accesses present in the program and characteristics of the external memories. Additional optimizations are performed in the scheduling phase related to resources present and the opportunities of parallel execution and pipelining available. In the final phase of the compiler, the optimized VHDL AST produces the output code. [25]
5.2 Adaptations for the WILDSTAR System

The MATCH compiler for the WILDSTAR is currently targeted only for PEX, and is based off an earlier version of the compiler designed for the WILDCHILD board by Haldar and Nayak [1,14,25]. Therefore, several functions available on the most recent version of the WILDCHILD compiler are currently unavailable for the WILDSTAR. This includes parallelization as well as other new user directives, which may be added later to increase the design flexibility.

Most of the synthesis techniques used in the MATCH compiler for generating VHDL from MATLAB code are very general and therefore easily integrated with other FPGA systems. The only functions of the compiler which require modification are those related to the memory read and write algorithms, such as signal names and width, the actual state machines for the read and write operations, and the reset stage.

On the WILDCHILD board, only one memory component is available for each FPGA. Therefore, the compiler was designed to store all data in a single local RAM, dividing the address space into 256x256 data blocks for matrices. However, with the abundance of memory components on the WILDSTAR board, each FPGA has the advantage of using an entire RAM to store a matrix. It was therefore necessary to add additional functionality to the WILDSTAR compiler to provide users with the ability to dynamically allocate different memories for data sets.

To serve this purpose, a new directive, memloc, was created to allow the user to specify a particular memory location for each matrix. The directive is used by the compiler to generate the corresponding memory read and write algorithms to access the data set in memory. Figure 5.2 above is a simple example of a MATLAB program that uses the memloc directive. The PEX architecture has four memory location options available for the memloc directive: left (left local memory), right...
(right local memory), \textit{mezzleft} (left Mezzanine memory), \textit{mezzright} (right Mezzanine memory). Obviously, the latter two are also dependent on the crossbar setting of the Mezzanine card. If a memory location directive is not specified, the compiler assumes all data to be stored in the PEX’s left local memory. If two data sets are designated to be stored in the same memory location, the compiler creates a 256x256 block of address space in the memory for each data set. Each RAM can hold a maximum of four matrix blocks. Figure 5.3 is a portion of the VHDL finite state machine, synthesized from the code in Figure 5.2 by the MATCH compiler.

```vhdl
case state is
  when s0 ->
    i:= "0001" - '1';
    state <= s1;
  when s1 ->
    if ( i > "1111" - '1' ) then
      state <= s7;
    else
      state <= s2;
    end if;
  when s2 ->
    array_address_9bb38(31 downto 0) := BaseAddress_DATA + i;
    state <= s3;
  when s3 ->
    Right_Local_Mux.Req <= '1';
    Right_Local_Mux.Write <= '1';
    Right_Local_Mux.Addr <= array_address_9bb38;
    if ( 'Right_Local_Mux.Akk = '1' ) then
      Right_Local_Mux.Data_Out( 3 downto 0 ) <= i;
      Right_Local_Mux.Data_Out( 31 downto 4 ) <= (others => '0');
      state <= s4;
    end if;
  when s4 ->
    Right_Local_Mux.Req <= '0';
    Right_Local_Mux.Write <= '0';
    state <= s5;
  when s5 ->
    i:=i + "0001";
    state <= s6;
  when s6 ->
    state <= s1;
  when s7 ->
    interrupt(0) <= '1';
    state <= s7;
end case;
```

\textbf{Figure 5.3:} Synthesized RTL VHDL of the MATLAB code in Figure 5.2 by the MATCH compiler.
In the MATLAB code in Figure 5.2, two directives are used to guide the compiler in generating code for the WILDSTAR FPGA. The first is the type directive, which sets \( i \) and \( DATA \) to be integer values. Likewise, since \( DATA \) is an array of values that we wish to store in memory, it was necessary to declare our intent to the compiler by specifying the number of elements (16) to be stored in memory. Finally, the \textit{memloc} directive is used to specify that the values in \( DATA \) should be stored in PEX’s right local memory.

In state \( s0 \) in Figure 5.3, the counter \( i \) is initialized to zero by subtracting 1 from 1. The reason it is initialized in this manner is because all vectors and matrices in MATLAB begin with address 1, following in the footsteps of FORTRAN. However, in VHDL and most standard languages, addresses always begin with 0. Therefore, it was necessary to decrease the counters by one to allow access to address 0 of data elements. There are complications involved in this transformation which must be taken into account. For instance, when using the counter in a computation or as an argument in an if-statement in MATLAB, it is difficult to determine which value, \( i \) or \( i-1 \), should now be used in the synthesized VHDL model of the code. The wrong value may have radical effects on the outcome of the application.

States \( s1 \), \( s5 \) and \( s6 \) setup the constructs for the for-loop. While the counter has not reached its limit, the path continues on to state \( s2 \), where the memory address is computed. In this state, \textit{BaseAddress\_DATA} is a constant created by the compiler for 256x256 matrix blocks, and is used as the starting address of the matrix in memory. In this case, there is only one array of values stored in memory, so its base address starts at zero. Once the counter has reached its limit it branches to state \( s7 \), where the process terminates by asserting the interrupt vector high to notify the host that the process has completed.

State \( s3 \) contains the portion of the state machine that writes the values of \( DATA \) to the right local memory. Note that during the Precision Analysis phase of the compiler, it was determined that \( i \) required only four bits to hold its values, since its maximum value is 15, or “1111” in binary. This is why \( i \) is only written to the first four bits in the \textit{Data\_Out} signal of the left local memory MUX. These optimizations are highly useful for the \textit{Synplify} tool, which will optimize the hardware design during synthesis by setting the top 27 bits of the \textit{Data\_Out} signal to constant zero and removing unnecessary registers to save space on the chip.
5.3 Compiler-Generated Matrix Multiplication

This section discusses the synthesis of a Matrix Multiplication design from MATLAB to VHDL for the WILDSTAR board. Several designs with different matrix sizes were tested using the compiler, and the performance results were compared to that of the hand-coded designs in chapter 4.

5.3.1 Matrix Multiplication in MATLAB

Figure 5.4 shows the MATLAB code for an explicit 64x64 Matrix Multiplication function that was synthesized for the WILDSTAR board. In this design, user directives are used to inform the MATCH compiler that a, b and c are 64x64 matrices of type integer. Furthermore, a is designated to be stored in the left local memory of PEX, b is stored in right local memory, and c is stored in PEX’s right Mezzanine memory. These parameters are equivalent to the structure of the hand-coded single processor design.

```matlab
%!match TYPE integer a(64,64)
%!match MEMLOC left a(64,64)
a = [];  

%!match TYPE integer b(64,64)
%!match MEMLOC right b(64,64)
b = [];  

%!match TYPE integer i
for i = 1:1:64

%!match TYPE integer j
for j = 1:1:64

%!match TYPE integer temp
temp = 0;  

%!match TYPE integer k
for k = 1:1:64

temp = temp + a(i,k)*b(k,j);  
end  

%!match TYPE integer c(64,64)
%!match MEMLOC mezzright c
c(i,j) = temp;  
end

end
```

Figure 5.4: MATLAB code for 64x64 Matrix Multiplication.
5.3.2 Performance Evaluation

The code for the other Matrix Multiplication designs are identical to the one above in Figure 5.4, however the parameter sizes and counter limits are modified to fit the desired matrix dimensions. Each design was compiled to VHDL and targeted for PEX on the WILDSTAR board. They were then compiled and simulated using *Vsim* to prove the design’s correctness. Finally, the designs were synthesized using *Synplify* and *Xilinx* tools.

While the compiler for the WILDSTAR board currently does not support simultaneous data access from different memories, it is expected that the computation results will be slightly higher than then hand-coded designs since all data must be accessed sequentially rather than in parallel. The performance results of the compiler-generated Matrix Multiplication designs are listed in Table 5.1, and it was verified to run correctly at 50 MHz for data sizes up to 510x510 elements. Timing results were broken down and plotted in Figure 5.5. *Configure System* refers to the time required to configure the WILDSTAR board and load the design onto the FPGA. *Prepare Data* is the time required to prepare the data to be written to memory. *Load Data* is the loading time of the data into the FPGA’s local RAMs. *Compute Time* is the time taken from start to finish of the computation on WILDSTAR board. *Retrieve Data* is the time taken to retrieve the result data from the right Mezzanine memory. The Xilinx netlist count is also provided in terms of LUTs (4 LUTs per CLB).

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>LUTs</th>
<th>Configure System</th>
<th>Prepare Data</th>
<th>Load Data</th>
<th>Compute Time</th>
<th>Retrieve Data</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>2013 (20%)</td>
<td>0.480</td>
<td>0.002</td>
<td>0.010</td>
<td>0.121</td>
<td>0.020</td>
<td>0.633</td>
</tr>
<tr>
<td>128x128</td>
<td>2020 (21%)</td>
<td>0.500</td>
<td>0.067</td>
<td>0.040</td>
<td>0.967</td>
<td>0.090</td>
<td>1.664</td>
</tr>
<tr>
<td>256x256</td>
<td>2033 (21%)</td>
<td>0.550</td>
<td>2.670</td>
<td>0.140</td>
<td>7.710</td>
<td>0.350</td>
<td>11.420</td>
</tr>
<tr>
<td>416x416</td>
<td>2103 (21%)</td>
<td>0.490</td>
<td>6.970</td>
<td>0.380</td>
<td>33.080</td>
<td>0.920</td>
<td>41.840</td>
</tr>
<tr>
<td>510x510</td>
<td>2080 (21%)</td>
<td>0.540</td>
<td>10.310</td>
<td>0.560</td>
<td>60.900</td>
<td>1.390</td>
<td>73.700</td>
</tr>
</tbody>
</table>

Similar to the hand-coded Matrix Multiplication designs, the data in Table 5.1 and Figure 5.5 suggests that all the timing values increase linearly with matrix sizes. This result could be predicted since each operation is O(n) where n is the number of elements in the matrix. The execution time is dominated by the
computation time for large datasets. However, for small datasets the execution time is dominated by the system configuration time.

The performance of the compiler-generated designs did not quite measure up to the performance of the manual Matrix Multiplication designs. However, the MATCH compiler did produce significant results. In terms of design size, the synthesized compiler-generated code produced average netlists consisting of 21% of the chip size, while the synthesized hand-coded designs produced netlists ranging from 25-28% for the parallel implementation and as much as 35% for the single processor design. Clearly, the hardware and software optimizations performed by the compiler had a remarkable effect on the size of the design.

The configuration, data load and data retrieve times of the compiler-generated designs are comparable to that of the single processor design. This is understandable since the requirements of these three parameters are the same. Table 5.2 is a comparison of the computation results for the three Matrix Multiplication designs using different matrix sizes. Speed factors of the compiler and the single and parallel processor designs are also shown. It is apparent that the MATCH compiler produced results that are comparable with the hand implementations. The
speed factor of the single processor design is only 1.5 times faster than the compiler-generated designs, while the parallel processor design is only 3 times faster. Figure 5.6 shows the performance comparison of the three implementations graphically.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>MATCH Compiler</th>
<th>Single FPGA</th>
<th>Multiple FPGAs</th>
<th>Speed Factor (Single)</th>
<th>Speed Factor (Multiple)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64x64</td>
<td>0.121</td>
<td>0.079</td>
<td>0.040</td>
<td>1.532</td>
<td>3.025</td>
</tr>
<tr>
<td>128x128</td>
<td>0.967</td>
<td>0.630</td>
<td>0.316</td>
<td>1.535</td>
<td>3.060</td>
</tr>
<tr>
<td>256x256</td>
<td>7.710</td>
<td>5.030</td>
<td>2.520</td>
<td>1.533</td>
<td>3.060</td>
</tr>
<tr>
<td>510x510</td>
<td>60.900</td>
<td>39.730</td>
<td>19.850</td>
<td>1.533</td>
<td>3.068</td>
</tr>
</tbody>
</table>

5.4 Benchmarks for the MATCH Compilers

In order to evaluate and compare the performance of the MATCH compilers for the WILDCHILD and WILDSTAR systems, benchmarks were tested on both
The benchmarks consist of seven MATLAB programs that were compiled and synthesized for the two systems. The source code for the seven benchmark designs in MATLAB are located in Appendix E. The results of the benchmarks are shown in Table 5.3.

<table>
<thead>
<tr>
<th>Benchmark Design</th>
<th>WILDCHILD</th>
<th></th>
<th></th>
<th>WILDSTAR</th>
<th></th>
<th></th>
<th>Speed Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CLBs</td>
<td>MHz</td>
<td>Time</td>
<td>CLBs</td>
<td>MHz</td>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>VECTORSUM</td>
<td>80 (8%)</td>
<td>28.1</td>
<td>0.040</td>
<td>175 (7%)</td>
<td>73.2</td>
<td>0.017</td>
<td>2.353</td>
</tr>
<tr>
<td>MULT_128</td>
<td>146 (15%)</td>
<td>28.4</td>
<td>1.420</td>
<td>217 (9%)</td>
<td>80.0</td>
<td>0.608</td>
<td>2.336</td>
</tr>
<tr>
<td>MULT_256</td>
<td>149 (15%)</td>
<td>27.9</td>
<td>11.480</td>
<td>221 (9%)</td>
<td>71.5</td>
<td>5.672</td>
<td>2.024</td>
</tr>
<tr>
<td>ITHRESHOLD</td>
<td>83 (9%)</td>
<td>29.9</td>
<td>0.050</td>
<td>176 (7%)</td>
<td>87.2</td>
<td>0.018</td>
<td>2.778</td>
</tr>
<tr>
<td>HOMOGENEOUS</td>
<td>118 (12%)</td>
<td>29.7</td>
<td>0.060</td>
<td>195 (8%)</td>
<td>90.5</td>
<td>0.016</td>
<td>3.750</td>
</tr>
<tr>
<td>MOTION_EST</td>
<td>250 (25%)</td>
<td>28.2</td>
<td>0.010</td>
<td>241 (10%)</td>
<td>69.5</td>
<td>0.002</td>
<td>5.000</td>
</tr>
<tr>
<td>SOBEL</td>
<td>488 (48%)</td>
<td>28.7</td>
<td>0.330</td>
<td>317 (13%)</td>
<td>53.4</td>
<td>0.205</td>
<td>1.610</td>
</tr>
</tbody>
</table>

When comparing the two compilers, in which both generate completely sequential codes, it is apparent that the WILDSTAR system demonstrated better performance. Nevertheless, this is not surprising, since it is capable of running at clock speeds two or three times faster than the WILDCHILD system, producing speed factors ranging from 1.6 to 5.0 faster, depending on the complexity of the design. Furthermore, the advanced architecture of the Virtex FPGAs in the WILDSTAR system provides larger chip space, so designs implemented on these FPGAs have a greater number of CLBs to implement the functions than on the WILDCHILD system.

In the latest version of the WILDCHILD compiler, parallelism is available as a directive option, which would definitely improve the performance of the benchmarks. This function has not yet been implement for the WILDSTAR compiler, though its addition would surely increase the compilers potential.

### 5.5 Summary

This chapter described the adaptation of the MATCH compiler for the WILDSTAR multi-FPGA system. A brief overview of the compiler and its basic functionality was provided. The modifications and additions made to the
WILDCHILD compiler to conform to the WILDSTAR system is discussed. The WILDSTAR compiler was evaluated by comparing compiler-generated Matrix Multiplication designs in MATLAB with the hand-coded designs as discussed in chapter 4. Though the single and parallel processor designs had superior performance over the compiler-generated designs, they only produced speed factors on the order of 1.5 and 3 times faster respectively. This is most likely due to the fact that the hand-coded designs are able to access data from different memories in parallel, reducing the computation time. However the compiler optimizations did produce better results with respect to design size.

Several benchmark designs in MATLAB were compiled to VHDL and tested on the WILDCHILD and WILDSTAR boards. The performance results were compared and evaluated; the WILDSTAR designs showed better overall performance due to its advanced FPGA architecture. The benchmark performances may be enhanced with the addition of automated parallelism construction in the compiler-generated code.
CONCLUSIONS

It seems apparent that developing high-speed heterogeneous computing engines is of great benefit to computationally intensive scientific applications. One major disadvantage of such an engine is the difficulty in programming the engine for the computation. The objective of MATCH is to build a compiler to alleviate the arduous task of developing efficient code for systems such as FPGAs, DSPs and embedded processors, by automatically translating high-level applications written in MATLAB and mapping them to these systems.

The MATCH testbed consists of several COTS components, including the WILDCHILD and WILDSTAR reconfigurable computing systems by Annapolis Micro Systems. Several benchmarks were performed on the WILDCHILD and WILDSTAR systems to evaluate performances of the systems with hand-implemented and compiler-generated designs, as well as to determine new optimization strategies for the MATCH compiler.

The Wavelet Transform algorithm was used as benchmark tool for performance comparison of a design implementation in MATLAB, C and VHDL for the WILDCHILD reconfigurable computing platform. It was demonstrated that there exists the possibility to obtain performance improvements in orders of magnitude between a general-purpose processor and special purpose hardware.

Single processor and parallel processor implementations of the Matrix Multiplication algorithm were designed for the WILDSTAR board and compared to that of the WILDCHILD and several other platforms that are part of the MATCH testbed. Although the WILDSTAR hand-coded designs produced the worst results overall, it is hypothesized that this is due to the fact that the data pipelining is not an available option for the system. The cost of wasted clock cycles is apparent in the results, even at higher clock speeds. However, if pipelining ability is added to the system, performance increases may even push the system to top.
To complete the MATCH compiler, an interface was necessary to generate code for the WILDSTAR multi-FPGA system. Modifications and additions were made to the WILDCHILD compiler to conform to the WILDSTAR system, specifically the signals and algorithms dealing with memory access. Additionally, a new user directive was added to the MATCH compiler to provide users with the ability to dynamically target data sets in MATLAB to any memory component on the WILDSTAR board.

The WILDSTAR compiler was evaluated by comparing compiler-generated Matrix Multiplication designs in MATLAB with the hand-coded designs. The single and parallel processor designs had superior performance over the compiler-generated designs, however, they only produced speed factors on the order of 1.5 and 3 times faster respectively. This is due to the fact that the hand-coded designs are able to access data from different memories in parallel, which reduces the computation time. Optimizations made by the compiler produced better results with respect to design size on the FPGAs.

Several benchmark designs in MATLAB were compiled to VHDL and tested on the WILDCHILD and WILDSTAR boards. The performance results were compared, and the WILDSTAR designs showed better overall performance due to the advanced FPGA architecture.

While the WILDSTAR system has a more advanced architecture than the WILDCHILD system, optimizations in the design algorithms of WILDSTAR programming are required to improve its performance. The system has the ability to access data in parallel from different memories, though performances still suffer greatly due to the lack in ability to pipeline data. In addition, the current WILDSTAR compiler is based on an older version of the WILDCHILD compiler, and thus lacks some of its latest advances, such as parallelism. The addition of this functionality would surely enhance the performance of the compiler-generated code for the WILDSTAR system. These are in part issues that need to be addressed in future work on the MATCH compiler.
HOST API FOR WILDCCHILD AND WILDSTAR

A.1  WILDCCHILD Host API
A.2  WILDSTAR Host API
A.1 WILDCHILD Host API

- **WFireOpen** \(\text{Opens an interface to the WILDCHILD board.}\)
- **WFireResetBoard** \(\text{Resets the specified board.}\)
- **WFireMaskPEInt** \(\text{Masks the interrupt from a specified FPGA.}\)
- **WFireProgramPE** \(\text{Programs the specified PE with a circuit.}\)
- **WFireXBarConfig** \(\text{Loads 16 crossbar configurations from a text file.}\)
- **WFireClearPEInts** \(\text{Clears any pending interrupts on the host from the PE.}\)
- **WFireUnMaskPEInts** \(\text{Unmasks PE interrupts to enable them to alert the host.}\)
- **WFireSetMemoryMode** \(\text{Sets mode of the dual-port memory controller for a PE.}\)
- **WFireClearMem** \(\text{Clears the specified PE’s memory.}\)
- **WFireWrite** \(\text{Loads the specified PE’s local memory with data.}\)
- **WFireRead** \(\text{Reads the specified PE’s local memory.}\)
- **WFireClkSetFreq** \(\text{Configures the system clock frequency.}\)
- **WFireResetPE** \(\text{Resets the specified PE or set of PEs.}\)
- **WFireGetPEIntEvents** \(\text{Reads the interrupt status register on the board.}\)
- **WFireCloseBoard** \(\text{Closes the WILDCHILD board.}\)

A.2 WILDSTAR Host API

**General Functions:**
- **WS_Open** \(\text{Open WILDSTAR Board}\)
- **WS_Close** \(\text{Close WILDSTAR Board}\)
- **WS_ErrorString** \(\text{Return error string from error code}\)
- **WS_GetVersions** \(\text{Get Hardware/Software versions of board}\)
- **WS_GetPhysicalConfig** \(\text{Get board physical configuration information}\)
- **WS_ResetBoard** \(\text{Reset the WILDSTAR VME board}\)
- **WS_QueryLadBusSpeed** \(\text{Get clock speed of LAD bus}\)
- **WS_SetLadBusSpeed** \(\text{Set clock speed of LAD bus}\)

**PE Functions:**
- **WS_ProgramPe** \(\text{Program a PE with a design}\)
- **WS_DeProgramPE** \(\text{Deprogram the specified PE}\)

**Register Functions:**
- **WS_ReadPeReg** \(\text{Read from PE register space}\)
- **WS_WritePeReg** \(\text{Write to PE register space}\)
Memory Interface Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS_Mem_Create</td>
<td>Allocates Host memory for Memory Instance</td>
</tr>
<tr>
<td>WS_Mem_Release</td>
<td>Free memory allocated by WS_Mem_Create call</td>
</tr>
<tr>
<td>WS_Mem_IsIdle</td>
<td>Check if Memory Interface is active or idle</td>
</tr>
<tr>
<td>WS_Mem_Read</td>
<td>Copy data from the WILDSTAR to the Host</td>
</tr>
<tr>
<td>WS_Mem_Write</td>
<td>Copy data from the Host to WILDSTAR</td>
</tr>
<tr>
<td>WS_Mem_Copy</td>
<td>Copy data from one set of addresses to another</td>
</tr>
<tr>
<td>WS_Mem_Set</td>
<td>Write a value to a range of address locations</td>
</tr>
</tbody>
</table>

Display Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS_UpdateDisplay</td>
<td>Send characters to the board’s display</td>
</tr>
<tr>
<td>WS_DisplayIntensity</td>
<td>Set the board’s display intensity</td>
</tr>
</tbody>
</table>

Clock Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS_MClkSetConfig</td>
<td>Set the memory clock frequency</td>
</tr>
<tr>
<td>WS_UClkSetConfig</td>
<td>Set the user clock frequency</td>
</tr>
</tbody>
</table>
WILDSTAR CORE ARCHITECTURES

B.1 WS PEX Core Interface
B.2 WS PE0 Core Interface
B.1 WS PEX Core Interface

-- Copyright (C) 1998-2000, Annapolis Micro Systems, Inc.
-- All Rights Reserved.
--

-- Entity        : PEX
-- Architecture  : PEX_Behavior
-- Filename      : pex_arch.vhd
-- Created by    : David Zaretsky (dcz@nwu.edu)
-- Date          : 4/1/01
-- Description   : Architecture for PEX functions
--

-- ------------------------ Glossary ------------------------
--
-- Name Key:
-- =========
-- _AS       : Address Strobe
-- _CE       : Clock Enable
-- _CS       : Chip Select
-- _DS       : Data Strobe
-- _EN       : Enable
-- _OE       : Output Enable
-- _RD       : Read Select
-- _WE       : Write Enable
-- _WR       : Write Select
-- _d[d...]  : Delayed (registered) signal (each 'd' denotes one
--             level of delay)
-- _n        : Active low signals (must be last part of name)
--
-- Port Name                      Dir  Description
-- ============================   ===  ================================
-- Pads.Clock.M_Clk               I   Memory clock
-- Pads.Clock.P_Clk               I   Processor clock
-- Pads.Clock.K_Clk               I   LAD-bus clock
-- Pads.Clock.U_Clk               I   User clock
-- Pads.LAD_Bus.Addr_Data          B   LAD-bus shared address/data bus
-- Pads.LAD_Bus.AS_n               I   LAD-bus address strobe
-- Pads.LAD_Bus.DS_n               I   LAD-bus data strobe
-- Pads.LAD_Bus.Ack_n              O   LAD-bus acknowledge strobe
-- Pads.LAD_Bus.Reg_n              I   LAD-bus register select
-- Pads.LAD_Bus.WR_n               I   LAD-bus write select
-- Pads.LAD_Bus.CS_n               I   LAD-bus chip select
-- Pads.LAD_Bus.Int_Req_n          O   LAD-bus interrupt request
-- Pads.LAD_Bus.FIFO_Stat          O   FIFO status flags
-- Pads.Left_Mem.Addr              O   Left memory address bus
-- Pads.Left_Mem.Data              B   Left memory data bus
-- Pads.Left_Mem.Addr_CS_n         O   Left memory address chip select
-- Pads.Left_Mem.CS_n              O   Left memory chip select
-- Pads.Left_Mem.WE_n              O   Left memory write enable
-- Pads.Right_Mem.Addr             O   Left memory address bus
-- Pads.Right_Mem.Data             B   Left memory data bus
-- Pads.Right_Mem.Addr_CS_n        O   Left memory address chip select
-- Pads.Right_Mem.CS_n             O   Left memory chip select
-- Pads.Right_Mem.WE_n             O   Left memory write enable
-- Pads.Left_Sys                   B   Left systolic bus
-- Pads.Right_Sys                  B   Right systolic bus
-- Pads.Mezz.Left                  B   Left mezzanine connector
-- Pads.Mezz.Right                  B   Right mezzanine connector
-- Pads.LEDs_n.Red_n               O   Red light emitting diode
-- Pads.LEDs_n.Green_n             O   Green light emitting diode
-- Pads.PE0_Bus                    B   PE0 bus
--

------------------------------------------------------------------------------
-------------------------- Library Declarations ------------------------
--
-- IEEE Libraries --
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;

-- Wildstar System Libraries -
library SYSTEM;
use SYSTEM.Xilinx_Package.all;
use SYSTEM.AMS_package.all;

-- Wildstar PEx Libraries --
library PEX_Lib;
use PEX_Lib.PE_Package.all;
use PEX_Lib.PE_Mezz_Mem_package.all;
--use PEX_Lib.PE_Mezz_Jumper_Package.all;
use PEX_Lib.Mezz_Mem_Mux_pkg.all;
use PEX_Lib.Mezz_Mem64_Mux_pkg.all;
use PEX_Lib.PEX_Mem32_Mux_pkg.all;
use PEX_Lib.PE_LAD_Mux_pkg.all;
use PEX_Lib.LAD_Mem32_Mux_pkg.all;
use PEX_Lib.LAD_Mem64_Mux_pkg.all;

-- LAD Mux Libraries --
library LAD_Mux_Lib;
use LAD_Mux_Lib.LAD_Mux_pkg.all;

-- Mem Mux Libraries --
library Mem32_Mux_Lib, Mem64_Mux_Lib;
use Mem32_Mux_Lib.Mem32_Mux_pkg.all;
use Mem64_Mux_Lib.Mem64_Mux_pkg.all;

-- DMA Mux Libraries --
--library DMA_Mux_Lib;
--use DMA_Mux_Lib.DMA_Mux_Pkg.all;

------------------------ Architecture Declaration ----------------------

architecture PEX_Behavior of PEX is

------------------------------- Glossary -----------------------------

--
-- Name Key:
--
-- _AS       : Address Strobe
-- _CB       : CardBus
-- _CE       : Clock Enable
-- _CS       : Chip Select
-- _DS       : Data Strobe
-- _EN       : Enable
-- _OE       : Output Enable
-- _PE       : Processing Element
-- _RD       : Read Select
-- _WE       : Write Enable
-- _WR       : Write Select
-- _d[d...]  : Delayed (registered) signal (each 'd' denotes one
--              level of delay)
-- _n        : Active low signals (must be last part of name)
--

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Width</th>
<th>Dir</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>1</td>
<td>-</td>
<td>Signal ground (logical '0')</td>
</tr>
<tr>
<td>Global_Reset</td>
<td>1</td>
<td>-</td>
<td>Global reset (or set) signal</td>
</tr>
<tr>
<td>Reset_Register</td>
<td>1</td>
<td>-</td>
<td>LAD bus accessible reset</td>
</tr>
<tr>
<td>Clocks_In.M_Clk</td>
<td>1</td>
<td>I</td>
<td>Memory clock</td>
</tr>
<tr>
<td>Clocks_In.P_Clk</td>
<td>1</td>
<td>I</td>
<td>Processor clock</td>
</tr>
<tr>
<td>Clocks_In.K_Clk</td>
<td>1</td>
<td>I</td>
<td>LAD-bus clock</td>
</tr>
<tr>
<td>Clocks_In.U_Clk</td>
<td>1</td>
<td>I</td>
<td>User clock</td>
</tr>
<tr>
<td>Clocks_In.M_Clk_2x</td>
<td>1</td>
<td>I</td>
<td>2x Memory clock</td>
</tr>
<tr>
<td>Clocks_In.P_Clk_2x</td>
<td>1</td>
<td>I</td>
<td>2x Processor clock</td>
</tr>
<tr>
<td>Clocks_In.K_Clk_2x</td>
<td>1</td>
<td>I</td>
<td>2x LAD-bus clock</td>
</tr>
<tr>
<td>Clocks_In.U_Clk_2x</td>
<td>1</td>
<td>I</td>
<td>2x User clock</td>
</tr>
<tr>
<td>Clocks_In.M_Clk_Locked</td>
<td>1</td>
<td>I</td>
<td>M_Clk CLKDLL locked flag</td>
</tr>
<tr>
<td>Clocks_In.P_Clk_Locked</td>
<td>1</td>
<td>I</td>
<td>P_Clk CLKDLL locked flag</td>
</tr>
<tr>
<td>Clocks_In.K_Clk_Locked</td>
<td>1</td>
<td>I</td>
<td>K_Clk CLKDLL locked flag</td>
</tr>
<tr>
<td>Clocks_In.U_Clk_Locked</td>
<td>1</td>
<td>I</td>
<td>U_Clk CLKDLL locked flag</td>
</tr>
<tr>
<td>Clocks_Out.M_Clk_DllRst</td>
<td>1</td>
<td>O</td>
<td>M_Clk CLKDLL reset signal</td>
</tr>
<tr>
<td>Clocks_Out.P_Clk_DllRst</td>
<td>1</td>
<td>O</td>
<td>P_Clk CLKDLL reset signal</td>
</tr>
<tr>
<td>Clocks_Out.K_Clk_DllRst</td>
<td>1</td>
<td>O</td>
<td>K_Clk CLKDLL reset signal</td>
</tr>
</tbody>
</table>
--  Clocks_Out.U_Clk_DllRst      1   O  U_Clk CLKDLL reset signal
--  LAD_Bus_In.Addr             24   I  LAD bus DWORD address bus input
--  LAD_Bus_In.Data_In          32   I  LAD bus data bus input
--  LAD_Bus_In.Reg_Strobe_n     1   I  LAD bus register access strobe
--  LAD_Bus_In.Mem_Strobe_n     1   I  LAD bus memory access strobe
--  LAD_Bus_In.Write_Sel_n      1   I  LAD bus write select
--  LAD_Bus_In.DMA_Chan         2   I  LAD bus DMA channel number
--                                      indicator
--  LAD_Bus_Out.Data_Out        32   O  LAD bus data bus output
--  LAD_Bus_Out.Int_Req_n       1   O  LAD bus interrupt request
--  LAD_Bus_Out.DMA_Stat         2   O  LAD bus DMA channel status
--                                      flags
--  LEDs_Out.Red_n              1   O  Red LED output
--  LEDs_Out.Green_n            1   O  Green LED output
--  Left_Mem_In.Data_In         32   I  Left on-board memory input
--                                      data bus
--  Left_Mem_In.Data_Valid_n    1   I  Left on-board memory valid
--                                      read flag
--  Left_Mem_Out.Addr           20   O  Left on-board memory
--                                      address bus
--  Left_Mem_Out.Data_Out       32   O  Left on-board memory output
--  Left_Mem_Out.Strobe_n       1   O  Left on-board memory access
--                                      strobe
--  Left_Mem_Out.Write_Sel_n    1   O  Left on-board memory write
--                                      select
--  Right_Mem_In.Data_In        32   I  Right on-board memory input
--  Right_Mem_In.Data_Valid_n   1   I  Right on-board memory valid
--                                      read flag
--  Right_Mem_Out.Addr          20   O  Right on-board memory address
--                                      bus
--  Right_Mem_Out.Data_Out      32   O  Right on-board memory output
--  Right_Mem_Out.Strobe_n      1   O  Right on-board memory access
--  Right_Mem_Out.Write_Sel_n   1   O  Right on-board memory write
--                                      select
--  Left_Mezz_In.Data_In        64   I  Left mezz mem data input
--  Left_Mezz_In.Low_Valid_n    1   I  Left mezz mem low data valid
--  Left_Mezz_Out.Addr          20   O  Left mezz mem address
--  Left_Mezz_Out.Data_Out      32   O  Left mezz mem data output
--  Left_Mezz_Out.Low_Strobe_n  1   O  Left mezz mem low Dword access
--  Left_Mezz_Out.Xbar_Mode_Out  2   O  Left mezz mem xbar mode output
--  Left_Mezz_Out.Xbar_Mode_OE_n 1   O  Left mezz mem xbar mode output

77
-- Below are all of the standard PE pad interface signals. Simply
-- uncomment the signal(s) that are needed by the PE design. All
-- other unused signals may remain commented out. Be sure to
-- uncomment any component instances used by the interface.
--
signal GND            : std_logic := '0';
signal Global_Reset   : std_logic := '0';
signal Reset_Register : std_logic := '0';
signal Clocks_In      : Clock_Std_IF_In_Type;
signal Clocks_Out     : Clock_Std_IF_Out_Type;
signal LEDs_Out : LED_Std_IF_Out_Type;

-- signal LAD_Bus_In       : LAD_Bus_Std_IF_In_Type;
-- signal LAD_Bus_Out      : LAD_Bus_Std_IF_Out_Type;
-- signal Left_Mem_In      : Mem_Std_IF_In_Type;
-- signal Left_Mem_Out     : Mem_Std_IF_Out_Type;
-- signal Right_Mem_In     : Mem_Std_IF_In_Type;
-- signal Right_Mem_Out    : Mem_Std_IF_Out_Type;
-- signal Left_Mezz_In     : Mezz_Mem_Std_IF_In_Type;
-- signal Left_Mezz_Out    : Mezz_Mem_Std_IF_Out_Type;
-- signal Right_Mezz_In    : Mezz_Mem_Std_IF_In_Type;
-- signal Right_Mezz_Out   : Mezz_Mem_Std_IF_Out_Type;
-- signal PE0_Bus_In       : PE0_Bus_Std_IF_In_Type;
-- signal PE0_Bus_Out      : PE0_Bus_Std_IF_Out_Type;
-- signal Top_Sys_In       : Systolic_Std_IF_In_Type;
-- signal Top_Sys_Out      : Systolic_Std_IF_Out_Type;
-- signal Bot_Sys_In       : Systolic_Std_IF_In_Type;
-- signal Bot_Sys_Out      : Systolic_Std_IF_Out_Type;

-- Below are signals which can be used with the Mem_Mux and
-- LAD_Mux interfaces. Modify each signal to match the number of
-- clients for each mux interface.

signal LAD_Mux_Bus           : LAD_Mux_vector   ( 0 to 5 );
signal Left_Local_Mux        : Mem32_Mux_vector ( 0 to 1 );
signal Right_Local_Mux       : Mem32_Mux_vector ( 0 to 1 );
signal Left_Mezz_Mux         : Mem64_Mux_vector ( 0 to 1 );
signal Right_Mezz_Mux        : Mem64_Mux_vector ( 0 to 1 );
signal Left_Mezz_XBar_Mode   : std_logic_vector (1 downto 0);
signal Right_Mezz_XBar_Mode  : std_logic_vector (1 downto 0);

-- signal Ver_Register        : LAD_Mux_register_vector(0 to 3);
-- signal XBar_Register       : LAD_Mux_register_vector(0 to 1);

constant KCLK_DLL_TYPE  : Clk_DLL_Out_Type := USE_1x;
constant KCLK_2x        : boolean          := FALSE;

component WS_PEX is
  port
  (
    Mclk                 : in      std_logic;
    Kclk                 : in      std_logic;
    Left_Local_Mux       : inout   Mem32_Mux;
    Right_Local_Mux      : inout   Mem32_Mux;
    Left_Mezz_Mux        : inout   Mem64_Mux;
    Right_Mezz_Mux       : inout   Mem64_Mux;
    Left_Mezz_XBar_Mode  : out     std_logic_vector (1 downto 0);
    Right_Mezz_XBar_Mode : out     std_logic_vector (1 downto 0);
  );
LAD : inout LAD_Mux;
LEDs_Out : out LED_Std_IF_Out_Type;
reset : in std_logic
);
end component;

begin

PEX_Instance : WS_PEX
port map
(
  Mclk => Clocks_In.M_Clk,
  Kclk => Clocks_In.K_Clk,
  Left_Local_Mux => Left_Local_Mux(1),
  Right_Local_Mux => Right_Local_Mux(1),
  Left_Mezz_Mux => Left_Mezz_Mux(1),
  Right_Mezz_Mux => Right_Mezz_Mux(1),
  Left_Mezz_XBar_Mode => Left_Mezz_XBar_Mode,
  Right_Mezz_XBar_Mode => Right_Mezz_XBar_Mode,
  LAD => LAD_Mux_Bus(0),
  LEDs_Out => LEDs_Out,
  reset => Global_Reset
);

-- Below are all of the standard PE pad interface components. Simply
-- uncomment the interface(s) that are needed by the PE design. All
-- other unused interfaces may remain commented out. Be sure to
-- uncomment any signal declarations used by the interface.
--
-- Version register lad mux component
--
-- U_REGFILE_0 : LAD_Mux_RegFile
-- generic map
-- (
--   Mask => x"7FFC",
--   Base => x"7FF0",
--   L2Num => 2
-- )
-- port map
-- (  
  Kclk => Clocks_In.K_Clk,  
  LAD => LAD_Mux_Bus(1),  
  Regs => Ver_Register  
);  
  
  -- Ver_Register(0).Data_Out <= x"020A0000"; -- VHDL  
  -- Ver_Register(1).Data_Out <= x"0005040E"; -- MODELTECH  
  -- Ver_Register(2).Data_Out <= x"00060103"; -- SYNPLICITY  
  -- Ver_Register(3).Data_Out <= x"00030104"; -- XILINX  
--  
-- u_Regfile: LAD_Mux_RegFile  
--  generic map  
-- (  
--    Mask => x"7E00",  
--    Base => x"0000",  
--    L2Num => 0  
-- )  
--  port map  
-- (  
--    Kclk => Clocks_In.K_Clk,  
--    LAD => LAD_Mux_Bus(2),  
--    Regs => XBar_Register  
-- );  

-- The following four bridge components provide LAD-based access to  
PEX's memories.  

U_LEFT_LOCAL_BRIDGE : LAD_Mem32_Bridge  
generic map  
(    
    Mask => x"7E00",  
    Base => x"1000"  
)  
port map  
(    
    Kclk => Clocks_In.K_Clk,  
    Mclk => Clocks_In.M_Clk,  
    LAD => LAD_Mux_Bus(1),  
    Mem => Left_Local_Mux(0)  
);  

U_RIGHT_LOCAL_BRIDGE : LAD_Mem32_Bridge  
generic map  
(    
    Mask => x"7E00",  
    Base => x"1200"  
)  
port map  
(    
    Kclk => Clocks_In.K_Clk,  
    Mclk => Clocks_In.M_Clk,  
    LAD => LAD_Mux_Bus(2),  
    Mem => Right_Local_Mux(1)  
);
Mem => Right_Local_Mux(0);

U_LEFT_MEZZ_BRIDGE : LAD_Mem64_Bridge
generic map
{
    Mask => x"7C00",
    Base => x"1400"
}
port map
{
    Kclk => Clocks_In.K_Clk,
    Mclk => Clocks_In.M_Clk,
    LAD => LAD_Mux_Bus(3),
    Mem => Left_Mezz_Mux(0)
};

U_RIGHT_MEZZ_BRIDGE : LAD_Mem64_Bridge
generic map
{
    Mask => x"7C00",
    Base => x"1800"
}
port map
{
    Kclk => Clocks_In.K_Clk,
    Mclk => Clocks_In.M_Clk,
    LAD => LAD_Mux_Bus(4),
    Mem => Right_Mezz_Mux(0)
};

-- XBar_Register(0).Data_Out <= XBar_Register(0).Data_In;
-- XBar_Register(1).Data_Out <= XBar_Register(1).Data_In;

-- Below are all of the standard PE pad interface components. Simply
-- uncomment the interface(s) that are needed by the PE design. All
-- other unused interfaces may remain commented out. Be sure to
-- uncomment any signal declarations used by the interface.
--
-- U_Clocks : Clock_Std_IF
-- generic map
-- {
--     K_CLK_DLL_OUT => KCLK_DLL_TYPE,
--     M_CLK_DLL_TYPE => LOW_FREQ,
--     F_CLK_DLL_TYPE => LOW_FREQ,
--     U_CLK_DLL_TYPE => LOW_FREQ
-- }
-- port map
-- {
--     Pads => Pads.Clocks,
--     
-- }
User_In      => Clocks_In,
User_Out     => Clocks_Out
);
--
-- --@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
-- OBUF_Drive => SLOW_12mA
-- port map
--
-- M_Clk => Clocks_In.M_Clk,
-- Global_Reset => Global_Reset,
-- Pads => Pads.Left_Mem,
-- User_In => Left_Mem_In,
-- User_Out => Left_Mem_Out
-- );
--
-- --@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
-- @@
-- @@ NOTE: Use either the U_Right_Local_Mem_Mux or the
-- @@ U_Right_Mem, but not simultaneously.
-- @@
-- --@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

U_Right_Local_Mem_Mux : Mem32_Mux_Priority_IF
port map

U_Right_Mem : Mem_Std_IF
-- generic map
--
-- INFF_Delay => NODELAY,
-- OBUF_Drive => SLOW_12mA
--
-- port map
--
-- M_Clk => Clocks_In.M_Clk,
-- Global_Reset => Global_Reset,
-- Pads => Pads.Right_Mem,
-- Clients => Right_Local_Mux
-- );
--
-- --@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
-- @@
-- @@ NOTE: Use either the U_Mezz_Mem_Mux_Left or the
-- @@ U_Left_Mezz_Mem, but not simultaneously.
-- @@
-- --@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

U_Mezz_Mem_Mux_Left : Mezz_Mem64_Mux_Priority_IF
port map

--
-- --@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

Pads            => Pads.Mezz.Left,
Clients         => Left_Mezz_Mux,
Mode            => "00",
                  --      Mode            => Left_Mezz_XBar_Mode,
                  Mode_En         => '1'
                  );

--  U_Left_Mezz_Mem : Mezz_Mem_Std_IF
--    generic map
--    (     
--      INFF_Delay      => NODELAY,
--      OBUF_Drive      => SLOW_12mA
--    )
--    port map
--    (     
--      M_Clk           => Clocks_In.M_Clk,
--      Global_Reset    => Global_Reset,
--      Pads            => Pads.Mezz.Left,
--      User_In         => Left_Mezz_In,
--      User_Out        => Left_Mezz_Out
--    );

--  --@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
--  --@@
--  --@@  NOTE: Use either the U_Mezz_Mem_Mux_Right or the
--  --@@        U_Right_Mezz_Mem, but not simultaneously.
--  --@@
--@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

U_Mezz_Mem_Mux_Right : Mezz_Mem64_Mux_Priority_IF
port map
(     
  M_Clk            => Clocks_In.M_Clk,
  Reset            => Global_Reset,
  Pads             => Pads.Mezz.Right,
  Clients          => Right_Mezz_Mux,
  Mode             => "00",
  Mode            => Right_Mezz_XBar_Mode,
  Mode_En          => '1'
);
--
-- U_PE0_Bus : PE0_Bus_Std_IF
--
--   generic map
--   {
--     INFF_Delay      => NODELAY,
--     OBUF_Drive      => FAST_8mA
--   }
--   port map
--   {
--     Clk             => Clocks_In.M_Clk,
--     Global_Reset    => Global_Reset,
--     Pads            => Pads.PE0_Bus,
--     User_In         => PE0_Bus_In,
--     User_Out        => PE0_Bus_Out
--   };

-- U_Top_Sys : Systolic_Std_IF
--
--   generic map
--   {
--     INFF_Delay      => NODELAY,
--     OBUF_Drive      => FAST_8mA
--   }
--   port map
--   {
--     Clk             => Clocks_In.M_Clk,
--     Global_Reset    => Global_Reset,
--     Pads            => Pads.Top_Sys,
--     User_In         => Top_Sys_In,
--     User_Out        => Top_Sys_Out
--   };

-- U_Bot_Sys : Systolic_Std_IF
--
--   generic map
--   {
--     INFF_Delay      => NODELAY,
--     OBUF_Drive      => FAST_8mA
--   }
--   port map
--   {
--     Clk             => Clocks_In.M_Clk,
--     Global_Reset    => Global_Reset,
--     Pads            => Pads.Bot_Sys,
--     User_In         => Bot_Sys_In,
--     User_Out        => Bot_Sys_Out
--   };

U_LEDS : LED_Std_IF
port map
{ Pads             => Pads.LEDs,
  User_Out         => LEDs_Out
};
-- Readback interface : Use the CAPTURE_VIRTEX block for readback.
--
-- U_Capture_Readback : CAPTURE_VIRTEX
--
-- port map
-- {
-- CAP         => Capture
-- CLK         => Clocks_In.K_Clk
-- }
--
-- --@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
--
-- The following LAD_Mux component provides a global reset signal to
-- the PE. It is mapped to LAD address 0x7fff. To change it's
-- mapping alter the constant passed to the Base generic.
--
U_LAD_Mux_Reset: LAD_Mux_Reset
  generic map
  (    Mask            => x"7F00",
       Base            => x"7F00"
  )
  port map
  (    Rclk            => Clocks_In.M_Clk,
       Kclk            => Clocks_In.K_Clk,
       LAD             => LAD_Mux_Bus(5),
       Reset           => Global_Reset,
       DLL_Reset_0     => Clocks_Out.M_Clk_DllRst,
       DLL_Reset_1     => Clocks_Out.P_Clk_DllRst,
       DLL_Reset_2     => Clocks_Out.K_Clk_DllRst,
       DLL_Reset_3     => Clocks_Out.U_Clk_DllRst
  );
--
-- Global reset interface : Attach the Reset_Register signal to a
-- register bit of a LAD bus accessible register. This reset
-- mechanism generates a one K_Clk cycle long pulse to the GSR line
-- of the STARTUP block. The STARTUP block is also synchronous to
-- K_Clk.
--
U_Reset_Pulse_Gen : One_Shot
  port map
--    ( 
--      Clk             => Clocks_In.K_Clk, 
--      I               => Reset_Register, 
--      O               => Global_Reset 
--    );
--
--  U_Startup : STARTUP_VIRTEX_GSR
--    port map
--    ( 
--      GSR             => Global_Reset 
--    );

-- NOTE :  The following line must remain in all designs
--          to ensure that all of the PE pads are driven.
--  Init_PEX_Pads ( Pads );

end PEX_Behavior ;
B.2 WS PE0 Core Interface

-- Copyright (C) 1998-2000, Annapolis Micro Systems, Inc.
-- All Rights Reserved.

-- Entity : PE0
-- Architecture : PE0_Behavior
-- Filename : pe0_arch.vhd
-- Created by : David Zaretsky (dcz@nwu.edu)
-- Date : 4/1/01
-- Description : Architecture for PE0 functions

------------------------------- Glossary -------------------------------
-- Name Key:
-- =========
-- _AS : Address Strobe
-- _CE : Clock Enable
-- _CS : Chip Select
-- _DS : Data Strobe
-- _EN : Enable
-- _OE : Output Enable
-- _RD : Read Select
-- _WE : Write Enable
-- _WR : Write Select
-- _d[d...] : Delayed (registered) signal (each 'd' denotes one
-- level of delay)
-- _n : Active low signals (must be last part of name)

-- Port Name  Dir  Description
-- =========================  ===  ================================
-- Pads.Clocks.M_Clk     I  Memory clock
-- Pads.Clocks.P_Clk     I  Processor clock
-- Pads.Clocks.K_Clk     I  LAD-bus clock
-- Pads.Clocks.U_Clk     I  User clock
-- Pads.Reset           I  Global reset
-- Pads.LAD_Bus.Addr_Data B  LAD-bus shared address/data bus
-- Pads.LAD_Bus.DS_n     I  LAD-bus data strobe
-- Pads.LAD_Bus.Reg_n    I  LAD-bus register select
-- Pads.LAD_Bus.WR_n     I  LAD-bus write select
-- Pads.LAD_Bus.CS_n               I   LAD-bus chip select
-- Pads.LAD_Bus.Int_Req_n          O   LAD-bus interrupt request
-- Pads.LAD_Bus.DMA_Chan           I   DMA channel number indicator
-- Pads.LAD_Bus.DMA_Stat           O   DMA channel status flags
-- Pads.PE1_Mezz.Left              B   Left PE1 mezzanine connector
-- Pads.PE1_Mezz.Right             B   Right PE1 mezzanine connector
-- Pads.PE2_Mezz.Left              B   Left PE2 mezzanine connector
-- Pads.PE2_Mezz.Right             B   Right PE2 mezzanine connector
-- Pads.PE1_Bus                    B   Communication bus to PE1
-- Pads.PE2_Bus                    B   Communication bus to PE2
-- Pads.IO_Conn_0                  B   I/O connector 0
-- Pads.IO_Conn_1                  B   I/O connector 1
-- Pads.LEDs.Red_n                 O   Red light emitting diode
-- Pads.LEDs.Green_n               O   Green light emitting diode

-- -------------------------- Library Declarations ------------------------
-- IEEE Libraries --
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;

-- Wildstar System Libraries --
library SYSTEM;
use SYSTEM.Xilinx_Package.all;
use SYSTEM.AMS_package.all;

-- Wildstar PE0 Libraries --
library PE0_Lib;
use PE0_Lib.PE_Package.all;
use PE0_Lib.PE_Mezz_Mem_package.all;
use PE0_Lib.Mezz_Mem_Mux_pkg.all;
use PE0_Lib.Mezz_Mem32_Mux_pkg.all;
use PE0_Lib.PE_LAD_Mux_pkg.all;
use PE0_Lib.LAD_Mem32_Mux_pkg.all;

-- LAD Mux Libraries --
library LAD_Mux_Lib;
use LAD_Mux_Lib.LAD_Mux_pkg.all;

-- Mem Mux Libraries --
library Mem32_Mux_Lib;
use Mem32_Mux_Lib.Mem32_Mux_pkg.all;

-- DMA Mux Libraries --
library DMA_Mux_Lib;
--use DMA_Mux_Lib.DMA_Mux_Pkg.all;

-------------------------------- Architecture Declaration ----------------------
architecture PE0_Behavior of PE0 is

-- ----------------------------- Glossary -----------------------------
--
-- Name Key:
-- =========
-- _AS       : Address Strobe
-- _CB       : CardBus
-- _CE       : Clock Enable
-- _CS       : Chip Select
-- _DS       : Data Strobe
-- _EN       : Enable
-- _OE       : Output Enable
-- _PE       : Processing Element
-- _RD       : Read Select
-- _WE       : Write Enable
-- _WR       : Write Select
-- _d[d...]  : Delayed (registered) signal (each 'd' denotes one
--              level of delay)
-- _n        : Active low signals (must be last part of name)
--
-- Port Name                     Width Dir Description
-- ============================= ===== ===

--  GND                              1   -  Signal ground (logical '0')
--  Global_Reset                     1   -  Global reset (or set)
--  Signal Reset_Register            1   O  LAD bus accessible reset
--  Clocks_In.M_Clk                  1   I  Memory clock
--  Clocks_In.P_Clk                  1   I  Processor clock
--  Clocks_In.K_Clk                  1   I  LAD-bus clock
--  Clocks_In.U_Clk                  1   I  User clock
--  Clocks_In.M_Clk_2x               1   I  Doubled Memory clock
--  Clocks_In.P_Clk_2x               1   I  Doubled Processor clock
--  Clocks_In.K_Clk_2x               1   I  Doubled LAD-bus clock
--  Clocks_In.U_Clk_2x               1   I  Doubled User clock
--  Clocks_In.M_Clk_Locked           1   I  M_Clk CLKDLL locked flag
--  Clocks_In.P_Clk_Locked           1   I  P_Clk CLKDLL locked flag
--  Clocks_In.K_Clk_Locked           1   I  K_Clk CLKDLL locked flag
--  Clocks_In.U_Clk_Locked           1   I  U_Clk CLKDLL locked flag
--  Clocks_Out.M_Clk_D11Rst           1   O  M_Clk DLL Reset
--  Clocks_Out.P_Clk_D11Rst           1   O  P_Clk DLL Reset
--  Clocks_Out.K_Clk_D11Rst           1   O  K_Clk DLL Reset
--  Clocks_Out.U_Clk_D11Rst           1   O  U_Clk DLL Reset
--  LAD_Bus_In.Addr                   24  I  LAD bus DWORD input
--  LAD_Bus_In.Data_In                32  I  LAD bus data bus input
--  LAD_Bus_In.Reg_Strobe_n           1   I  LAD bus access strobe
--  LAD_Bus_In.Mem_Strobe_n           1   I  LAD bus mem access strobe
--  LAD_Bus_In.Write_SEL_n            1   I  LAD bus write select
--  LAD_Bus_In.DMA Chan               2   I  LAD bus DMA channel number
--                                          indicator
--  LAD_Bus_Out.Data_Out              32  O  LAD bus data bus output
-- LAD_Bus_Out.Int_Req_n  1  O  LAD bus interrupt request
-- LAD_Bus_Out.DMA_Stat   2  O  LAD bus DMA channel status flags
--
-- PE1_Left_Mezz_In.Data_In  32  I  PE1 left mezz mem data input
-- PE1_Left_Mezz_In.Valid_n   1  I  PE1 left mezz mem data valid flag
--
-- PE1_Left_Mezz_In.Xbar_Mode_In  2  I  PE1 left mezz mem xbar mode input
--
-- PE1_Left_Mezz_In.Shunt_In  54  I  PE1 left mezz mem shunt input
--
-- PE1_Left_Mezz_Out.Addr   20  O  PE1 left mezz mem address output
--
-- PE1_Left_Mezz_Out.Data_Out  32  O  PE1 left mezz mem data output
--
-- PE1_Left_Mezz_Out.Strobe_n  1  O  PE1 left mezz mem access strobe
--
-- PE1_Left_Mezz_Out.Write_Sel_n  1  O  PE1 left mezz mem write select
--
-- PE1_Left_Mezz_Out.Xbar_Mode_Out  2  O  PE1 left mezz mem xbar mode output
--
-- PE1_Left_Mezz_Out.Xbar_Mode_OE_n  1  O  PE1 left mezz mem xbar mode output enable
--
-- PE1_Left_Mezz_Out.Shunt_Out  54  O  PE1 left mezz mem shunt output
--
-- PE1_Left_Mezz_Out.Shunt_OE_n  54  O  PE1 left mezz mem shunt output enable
--
-- PE1_Right_Mezz_In.Data_In   32  I  PE1 right mezz mem data input enable
--
-- PE1_Right_Mezz_In.Valid_n   1  I  PE1 right mezz mem data valid flag
--
-- PE1_Right_Mezz_In.Xbar_Mode_In  2  I  PE1 right mezz mem xbar mode input
--
-- PE1_Right_Mezz_In.Shunt_In  54  I  PE1 right mezz mem shunt input
--
-- PE1_Right_Mezz_Out.Addr   20  O  PE1 right mezz mem address output
--
-- PE1_Right_Mezz_Out.Data_Out  32  O  PE1 right mezz mem data output
--
-- PE1_Right_Mezz_Out.Strobe_n  1  O  PE1 right mezz mem access strobe
--
-- PE1_Right_Mezz_Out.Write_Sel_n  1  O  PE1 right mezz mem write select
--
-- PE1_Right_Mezz_Out.Xbar_Mode_Out  2  O  PE1 right mezz mem xbar mode output
--
-- PE1_Right_Mezz_Out.Xbar_Mode_OE_n  1  O  PE1 right mezz mem xbar mode output enable
--
-- PE1_Right_Mezz_Out.Shunt_Out  54  O  PE1 right mezz mem shunt output
--
-- PE1_Right_Mezz_Out.Shunt_OE_n  54  O  PE1 right mezz mem shunt output enable
--
-- PE2_Left_Mezz_In.Data_In   32  I  PE2 left mezz mem data input
--
-- PE2_Left_Mezz_In.Valid_n   1  I  PE2 left mezz mem data valid flag
--
-- PE2_Left_Mezz_In.Xbar_Mode_In  2  I  PE2 left mezz mem xbar mode
<table>
<thead>
<tr>
<th>Port</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE2_Left_Mezz_In.Shunt_In</td>
<td>I</td>
<td>PE2 left mezz mem shunt input</td>
</tr>
<tr>
<td>PE2_Left_Mezz_Out.Addr</td>
<td>O</td>
<td>PE2 left mezz mem address output</td>
</tr>
<tr>
<td>PE2_Left_Mezz_Out.Data_Out</td>
<td>O</td>
<td>PE2 left mezz mem data output</td>
</tr>
<tr>
<td>PE2_Left_Mezz_Out.Strobe_n</td>
<td>O</td>
<td>PE2 left mezz mem access strobe</td>
</tr>
<tr>
<td>PE2_Left_Mezz_Out.Write_Sel_n</td>
<td>O</td>
<td>PE2 left mezz mem write select</td>
</tr>
<tr>
<td>PE2_Left_Mezz_Out.Xbar_Mode_Out</td>
<td>O</td>
<td>PE2 left mezz mem xbar mode output</td>
</tr>
<tr>
<td>PE2_Left_Mezz_Out.Xbar_Mode_OE_n</td>
<td>O</td>
<td>PE2 left mezz mem xbar mode output enable</td>
</tr>
<tr>
<td>PE2_Left_Mezz_Out.Shunt_Out</td>
<td>O</td>
<td>PE2 left mezz mem shunt output</td>
</tr>
<tr>
<td>PE2_Left_Mezz_Out.Shunt_OE_n</td>
<td>O</td>
<td>PE2 left mezz mem shunt output enable</td>
</tr>
<tr>
<td>PE2_Right_Mezz_In.Data_In</td>
<td>I</td>
<td>PE2 right mezz mem data input</td>
</tr>
<tr>
<td>PE2_Right_Mezz_In.Valid_n</td>
<td>I</td>
<td>PE2 right mezz mem data valid flag</td>
</tr>
<tr>
<td>PE2_Right_Mezz_In.Xbar_Mode_In</td>
<td>I</td>
<td>PE2 right mezz mem xbar mode input</td>
</tr>
<tr>
<td>PE2_Right_Mezz_In.Shunt_In</td>
<td>O</td>
<td>PE2 right mezz mem shunt input</td>
</tr>
<tr>
<td>PE2_Right_Mezz_Out.Addr</td>
<td>O</td>
<td>PE2 right mezz mem address output</td>
</tr>
<tr>
<td>PE2_Right_Mezz_Out.Data_Out</td>
<td>O</td>
<td>PE2 right mezz mem data output</td>
</tr>
<tr>
<td>PE2_Right_Mezz_Out.Strobe_n</td>
<td>O</td>
<td>PE2 right mezz mem access strobe</td>
</tr>
<tr>
<td>PE2_Right_Mezz_Out.Write_Sel_n</td>
<td>O</td>
<td>PE2 right mezz mem write select</td>
</tr>
<tr>
<td>PE2_Right_Mezz_Out.Xbar_Mode_Out</td>
<td>O</td>
<td>PE2 right mezz mem xbar mode output</td>
</tr>
<tr>
<td>PE2_Right_Mezz_Out.Xbar_Mode_OE_n</td>
<td>O</td>
<td>PE2 right mezz mem xbar mode output enable</td>
</tr>
<tr>
<td>PE2_Right_Mezz_Out.Shunt_Out</td>
<td>O</td>
<td>PE2 right mezz mem shunt output</td>
</tr>
<tr>
<td>PE2_Right_Mezz_Out.Shunt_OE_n</td>
<td>O</td>
<td>PE2 right mezz mem shunt output enable</td>
</tr>
<tr>
<td>PE1_Left_Mezz_Jumper_In.Data_In</td>
<td>I</td>
<td>PE1 left mezz jumper data input</td>
</tr>
<tr>
<td>PE1_Left_Mezz_Jumper_Out.Data_Out</td>
<td>O</td>
<td>PE1 left mezz jumper data output</td>
</tr>
<tr>
<td>PE1_Left_Mezz_Jumper_Out.Data_OE</td>
<td>O</td>
<td>PE1 left mezz jumper data output enable</td>
</tr>
<tr>
<td>PE1_Right_Mezz_Jumper_In.Data_In</td>
<td>I</td>
<td>PE1 right mezz jumper data input</td>
</tr>
<tr>
<td>PE1_Right_Mezz_Jumper_Out.Data_Out</td>
<td>O</td>
<td>PE1 right mezz jumper data output</td>
</tr>
</tbody>
</table>
--  PE1_Right_Mezz_Jumper_Out.Data_OE 54 O  PE1 right mezz jumper data output enable
--
--  PE2_Left_Mezz_Jumper_In.Data_In 54 I  PE2 left mezz jumper data input
--  PE2_Left_Mezz_Jumper_Out.Data_Out 54 O  PE2 left mezz jumper data output
--  PE2_Left_Mezz_Jumper_Out.Data_OE 54 O  PE2 left mezz jumper data output enable
--
--  PE2_Right_Mezz_Jumper_In.Data_In 54 I  PE2 right mezz jumper data input
--  PE2_Right_Mezz_Jumper_Out.Data_Out 54 O PE2 right mezz jumper data output
--  PE2_Right_Mezz_Jumper_Out.Data_OE 54 O  PE2 right mezz jumper data output enable
--
--  PE1_Bus_In.Data_In               2   I  PE1 bus data input
--  PE1_Bus_Out.Data_Out             2   O  PE1 bus data output
--  PE1_Bus_Out.Data_OE_n            2   O  PE1 bus data output enable
--  PE2_Bus_In.Data_In               2   I  PE2 bus data input
--  PE2_Bus_Out.Data_Out             2   O  PE2 bus data output
--  PE2_Bus_Out.Data_OE_n            2   O  PE2 bus data output enable
--  IO_Conn_0_In.Data_In             66  I  I/O connector 0 data input
--  IO_Conn_0_Out.Data_Out           66  O  I/O connector 0 data output
--  IO_Conn_0_Out.Data_OE_n          66  O  I/O connector 0 data output enable
--  IO_Conn_1_In.Data_In             66  I  I/O connector 1 data input
--  IO_Conn_1_Out.Data_Out           66  O  I/O connector 1 data output
--  IO_Conn_1_Out.Data_OE_n          66  O  I/O connector 1 data output enable
--  LEDs_Out.Red_n                   1   O  Red LED output
--  LEDs_Out.Green_n                 1   O  Green LED output

--
-- Below are all of the standard PE pad interface signals. Simply
-- uncomment the signal(s) that are needed by the PE design. All
-- other unused signals may remain commented out. Be sure to
-- uncomment any component instances used by the interface.

signal GND                : std_logic := '0';
signal Global_Reset       : std_logic := '0';
signal Reset_Register     : std_logic := '0';
signal Clocks_In          : Clock_Std_IF_In_Type;
signal Clocks_Out         : Clock_Std_IF_Out_Type;
signal LEDs_Out : LED_Std_IF_Out_Type;
-- signal LAD_Bus_In : LAD_Bus_Std_IF_In_Type;
-- signal LAD_Bus_Out : LAD_Bus_Std_IF_Out_Type;
-- signal PE1_Left_Mezz_In : Mezz_Mem_Std_IF_In_Type;
-- signal PE1_Left_Mezz_Out : Mezz_Mem_Std_IF_Out_Type;
-- signal PE1_Right_Mezz_In : Mezz_Mem_Std_IF_In_Type;
-- signal PE1_Right_Mezz_Out : Mezz_Mem_Std_IF_Out_Type;
-- signal PE2_Left_Mezz_In : Mezz_Mem_Std_IF_In_Type;
-- signal PE2_Left_Mezz_Out : Mezz_Mem_Std_IF_Out_Type;
-- signal PE2_Right_Mezz_In : Mezz_Mem_Std_IF_In_Type;
-- signal PE2_Right_Mezz_Out : Mezz_Mem_Std_IF_Out_Type;
-- signal PE1_Bus_In : PEX_Bus_Std_IF_In_Type;
-- signal PE1_Bus_Out : PEX_Bus_Std_IF_Out_Type;
-- signal PE2_Bus_In : PEX_Bus_Std_IF_In_Type;
-- signal PE2_Bus_Out : PEX_Bus_Std_IF_Out_Type;
-- signal IO_Conn_0_In : IO_Conn_Std_IF_In_Type;
-- signal IO_Conn_0_Out : IO_Conn_Std_IF_Out_Type;
-- signal IO_Conn_1_In : IO_Conn_Std_IF_In_Type;
-- signal IO_Conn_1_Out : IO_Conn_Std_IF_Out_Type;
-- signal PE1_Left_Mezz_Jumper_In : Mezz_Jumper_Std_IF_In_Type;
-- signal PE1_Left_Mezz_Jumper_Out : Mezz_Jumper_Std_IF_Out_Type;
-- signal PE1_Right_Mezz_Jumper_In : Mezz_Jumper_Std_IF_In_Type;
-- signal PE1_Right_Mezz_Jumper_Out : Mezz_Jumper_Std_IF_Out_Type;
-- signal PE2_Left_Mezz_Jumper_In : Mezz_Jumper_Std_IF_In_Type;
-- signal PE2_Left_Mezz_Jumper_Out : Mezz_Jumper_Std_IF_Out_Type;
-- signal PE2_Right_Mezz_Jumper_In : Mezz_Jumper_Std_IF_In_Type;
-- signal PE2_Right_Mezz_Jumper_Out : Mezz_Jumper_Std_IF_Out_Type;

-- Below are signals which can be used with the Mem_Mux and LAD_Mux interfaces. Modify each signal to match the number of clients for each mux interface.

//
signal LAD_Mux_Bus : LAD_Mux_vector ( 0 to 5 );
signal PE1_Left_Mezz_Mux : Mem32_Mux_vector ( 0 to 1 );
signal PE1_Right_Mezz_Mux : Mem32_Mux_vector ( 0 to 1 );
signal PE2_Left_Mezz_Mux : Mem32_Mux_vector ( 0 to 1 );
signal PE2_Right_Mezz_Mux : Mem32_Mux_vector ( 0 to 1 );
signal PE1_Left_Mezz_XBar_Mode : std_logic_vector (1 downto 0);
signal PE1_Right_Mezz_XBar_Mode : std_logic_vector (1 downto 0);
signal PE2_Left_Mezz_XBar_Mode : std_logic_vector (1 downto 0);
signal PE2_Right_Mezz_XBar_Mode : std_logic_vector (1 downto 0);

constant KCLK_DLL_TYPE : Clk_DLL_Out_Type := USE_1x;
constant KCLK_2x : boolean := FALSE;

component WS_PE0 is
port (

begin

PE0_Instance: WS_PE0
port map
(
    Mclk => Clocks_In.M_Clk,
    Kclk => Clocks_In.K_Clk,
    PE1_Left_Mezz_Mux => PE1_Left_Mezz_Mux(1),
    PE1_Right_Mezz_Mux => PE1_Right_Mezz_Mux(1),
    PE2_Left_Mezz_Mux => PE2_Left_Mezz_Mux(1),
    PE2_Right_Mezz_Mux => PE2_Right_Mezz_Mux(1),
    PE1_Left_Mezz_XBar_Mode => PE1_Left_Mezz_XBar_Mode(1),
    PE1_Right_Mezz_XBar_Mode => PE1_Right_Mezz_XBar_Mode(1),
    PE2_Left_Mezz_XBar_Mode => PE2_Left_Mezz_XBar_Mode(1),
    PE2_Right_Mezz_XBar_Mode => PE2_Right_Mezz_XBar_Mode(1),
    LAD => LAD_Mux_Bus(0),
    LEDs_Out => LEDs_Out,
    reset => Global_Reset
);

-- The following four bridge components provide LAD-based access to
-- PE0's memories.
U_PE1_LEFT_MEZZ_BRIDGE : LAD_Mem32_Bridge
generic map
(
    Mask => x"7E00",
    Base => x"1000"
)
port map
(
    Kclk => Clocks_In.K_Clk,
    Mclk => Clocks_In.M_Clk,
    LAD => LAD_Mux_Bus(1),
    Mem => PE1_Left_Mezz_Mux(0)
);
U_PE1_RIGHT_MEZZ_BRIDGE : LAD_Mem32_Bridge
  generic map
  (  
    Mask => x"7E00",
    Base => x"1200"
  )
  port map
  (  
    Kclk => Clocks_In.K_Clk,
    Mclk => Clocks_In.M_Clk,
    LAD => LAD_Mux_Bus(2),
    Mem => PE1_Right_Mezz_Mux(0)
  );

U_PE2_LEFT_MEZZ_BRIDGE : LAD_Mem32_Bridge
  generic map
  (  
    Mask => x"7C00",
    Base => x"1400"
  )
  port map
  (  
    Kclk => Clocks_In.K_Clk,
    Mclk => Clocks_In.M_Clk,
    LAD => LAD_Mux_Bus(3),
    Mem => PE2_Left_Mezz_Mux(0)
  );

U_PE2_RIGHT_MEZZ_BRIDGE : LAD_Mem32_Bridge
  generic map
  (  
    Mask => x"7C00",
    Base => x"1600"
  )
  port map
  (  
    Kclk => Clocks_In.K_Clk,
    Mclk => Clocks_In.M_Clk,
    LAD => LAD_Mux_Bus(4),
    Mem => PE2_Right_Mezz_Mux(0)
  );

-- Below are all of the standard PE pad interface components. Simply
-- uncomment the interface(s) that are needed by the PE design. All
-- other unused interfaces may remain commented out. Be sure to
-- uncomment any signal declarations used by the interface.


U_Clocks : Clock_Std_IF
  generic map
  (
    K_CLK_DLL_OUT => KCLK_DLL_TYPE,
    M_CLK_DLL_TYPE => LOW_FREQ,
    P_CLK_DLL_TYPE => LOW_FREQ,
    U_CLK_DLL_TYPE => LOW_FREQ
  )
  port map
  (
    Pads          => Pads.Clocks,
    User_In       => Clocks_In,
    User_Out      => Clocks_Out
  );

-- --'-----------------------------------'
-- --@@
-- --@@  NOTE: Use either the U_LAD_MUX_IF or the U_LAD.Bus,
-- --@@        but not simultaneously.
-- --@@
-- --'-----------------------------------'

U_LAD_MUX_IF : LAD_Mux_IF
  generic map
  (  
    K_Clk_x2   => KCLK_2x
  )
  port map
  (   
    Kclk        => Clocks_In.K_Clk,
    Reset       => Global_Reset,
    pads        => Pads.LAD_Bus,
    Clients     => LAD_Mux_Bus
  );

-- U_LAD_Bus : LAD_Bus_Std_IF
  port map
  (   
    K_Clk                   => Clocks_In.K_Clk,
    Global_Reset            => Global_Reset,
    Pads                    => Pads.LAD_Bus,
    User_In                 => LAD_Bus_In,
    User_Out                => LAD_Bus_Out
  );

-- --'-----------------------------------'
-- --@@
-- --@@  NOTE: Use either the U_Mezz_Mem_Mux_PE1_Left or the
-- --@@        U_PE1_Left_Mezz_Mem, but not simultaneously.
-- --@@
-- --'-----------------------------------'

U_Mezz_Mem_Mux_PE1_Left : Mezz_Mem32_Mux_Priority_IF
port map
(
  Mclk        => Clocks_In.M_Clk,
  Reset       => Global_Reset,
  Pads        => Pads.PE1_Mezz.Left,
  Clients     => PE1_Left_Mezz_Mux,
  Mode        => "00",
  Mode_En     => '0'
);

-- U_PE1_Left_Mezz_Mem : Mezz_Mem_Std_IF
-- generic map
-- (
--   INFF_Delay => NODELAY,
--   OBUF_Drive => SLOW_12mA
-- )
-- port map
-- (
--   M_Clk      => Clocks_In.M_Clk,
--   Global_Reset => Global_Reset,
--   Pads       => Pads.PE1_Mezz.Left,
--   User_In    => PE1_Left_Mezz_In,
--   User_Out   => PE1_Left_Mezz_Out
-- );

-- @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

-- NOTE: Use either the U_Mezz_Mem_Mux_PE1_Right or the
-- U_PE1_Right_Mezz_Mem, but not simultaneously.
-- @@@
-- @
-- @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

U_Mezz_Mem_Mux_PE1_Right : Mezz_Mem32_Mux_Priority_IF
port map
(
  Mclk       => Clocks_In.M_Clk,
  Reset      => Global_Reset,
  Pads       => Pads.PE1_Mezz.Right,
  Clients    => PE1_Right_Mezz_Mux,
  Mode       => "00",
  Mode_En    => '0'
);

-- U_PE1_Right_Mezz_Mem : Mezz_Mem_Std_IF
-- generic map
-- (
--   INFF_Delay => NODELAY,
--   OBUF_Drive => SLOW_12mA
-- )
-- port map
-- (
--   M_Clk      => Clocks_In.M_Clk,
--   Global_Reset => Global_Reset,
--   Pads       => Pads.PE1_Mezz.Right,
--   User_In    => PE1_Right_Mezz_In,
--   User_Out   => PE1_Right_Mezz_Out
-- );
-- User_In => PE1_Right_Mezz_In,
-- User_Out => PE1_Right_Mezz_Out
-- );
--
-- NOTE: Use either the U_Mezz_Mem_Mux_PE2_Left or the
-- U_PE2_Left_Mezz_Mem, but not simultaneously.
--
-- U_Mezz_Mem_Mux_PE2_Left : Mezz_Mem32_Mux_Priority_IF
port map
  (  
    Mclk => Clocks_In.M_Clk,  
    Reset => Global_Reset,  
    Pads => Pads.PE2_Mezz.Left,  
    Clients => PE2_Left_Mezz_Mux,  
    Mode => "00",  
    Mode_En => '0'
  );

-- U_PE2_Left_Mezz_Mem : Mezz_Mem_Std_IF
--
-- port map
--  (  
--    M_Clk => Clocks_In.M_Clk,  
--    Global_Reset => Global_Reset,  
--    Pads => Pads.PE2_Mezz.Left,  
--    User_In => PE2_Left_Mezz_In,  
--    User_Out => PE2_Left_Mezz_Out
--  );
--
-- NOTE: Use either the U_Mezz_Mem_Mux_PE2_Right or the
-- U_PE2_Right_Mezz_Mem, but not simultaneously.
--
-- U_Mezz_Mem_Mux_PE2_Right : Mezz_Mem32_Mux_Priority_IF
port map
(  
  Mclk => Clocks_In.M_Clk,  
  Reset => Global_Reset,  
  Pads => Pads.PE2_Mezz.Right,  
  Clients => PE2_Right_Mezz_Mux,  
  Mode => "00",  
  Mode_En => '0'
);

-- U_PE2_Right_Mezz_Mem : Mezz_Mem_Std_IF
-- generic map
-- (  
--   INFF_Delay => NODELAY,  
--   OBUF_Drive => SLOW_12mA  
-- )  
-- port map  
-- (  
--   M_Clk => Clocks_In.M_Clk,  
--   Global_Reset => Global_Reset,  
--   Pads => Pads.PE2_Mezz.Right,  
--   User_In => PE2_Right_Mezz_In,  
--   User_Out => PE2_Right_Mezz_Out  
-- );  
--  
-- U_PE1_Bus : PEX_Bus_Std_IF  
-- generic map  
-- (  
--   INFF_Delay => NODELAY,  
--   OBUF_Drive => FAST_8mA  
-- )  
-- port map  
-- (  
--   Clk => Clocks_In.M_Clk,  
--   Global_Reset => Global_Reset,  
--   Pads => Pads.PE1_Bus,  
--   User_In => PE1_Bus_In,  
--   User_Out => PE1_Bus_Out  
-- );  
--  
-- U_PE1_Left_Mezz_Jumper : Mezz_Jumper_Std_IF  
-- generic map  
-- (  
--   INFF_Delay => NODELAY,  
--   OBUF_Drive => FAST_8mA  
-- )  
-- port map  
-- (  
--   Clk => Clocks_In.K_Clk,  
--   Global_Reset => Global_Reset,  
--   Pads => Pads.PE1_Mezz.Left,  
--   User_In => PE1_Left_Mezz_Jumper_In,  
--   User_Out => PE1_Left_Mezz_Jumper_Out  
-- );  
--  
-- U_PE1_Right_Mezz_Jumper : Mezz_Jumper_Std_IF  
-- generic map  
-- (  
--   INFF_Delay => NODELAY,  
--   OBUF_Drive => FAST_8mA  
-- )  
-- port map  
-- (  
--   Clk => Clocks_In.K_Clk,  
--   Global_Reset => Global_Reset,
-- Pads          => Pads.PE1_Mezz.Right,
-- User_In       => PE1_Right_Mezz_Jumper_In,
-- User_Out      => PE1_Right_Mezz_Jumper_Out
-- );
--
-- U_PE2_Left_Mezz_Jumper : Mezz_Jumper_Std_IF
-- generic map
-- {
--  INFF_Delay    => NODELAY,
--  OBUF_Drive    => FAST_8mA
-- }
-- port map
-- {
--  Clk           => Clocks_In.K_Clk,
--  Global_Reset  => Global_Reset,
--  Pads          => Pads.PE2_Mezz.Left,
--  User_In       => PE2_Left_Mezz_Jumper_In,
--  User_Out      => PE2_Left_Mezz_Jumper_Out
-- );
--
-- U_PE2_Right_Mezz_Jumper : Mezz_Jumper_Std_IF
-- generic map
-- {
--  INFF_Delay    => NODELAY,
--  OBUF_Drive    => FAST_8mA
-- }
-- port map
-- {
--  Clk           => Clocks_In.K_Clk,
--  Global_Reset  => Global_Reset,
--  Pads          => Pads.PE2_Mezz.Right,
--  User_In       => PE2_Right_Mezz_Jumper_In,
--  User_Out      => PE2_Right_Mezz_Jumper_Out
-- );
--
-- U_PE2_Bus : PEX_Bus_Std_IF
-- generic map
-- {
--  INFF_Delay    => NODELAY,
--  OBUF_Drive    => FAST_8mA
-- }
-- port map
-- {
--  Clk           => Clocks_In.M_Clk,
--  Global_Reset  => Global_Reset,
--  Pads          => Pads.PE2_Bus,
--  User_In       => PE2_Bus_In,
--  User_Out      => PE2_Bus_Out
-- );
--
-- U_IO_Conn_0 : IO_Conn_Std_IF
-- generic map
-- {
--  INFF_Delay    => NODELAY,
```
-- OBUF_Drive => FAST_8mA
-- port map
-- (  
--   Clk => Clocks_In.M_Clk,
--   Global_Reset => Global_Reset,
--   Pads => Pads.IO_Conn_0,
--   User_In => IO_Conn_0_In,
--   User_Out => IO_Conn_0_Out
-- );

-- U_IO_Conn_1 : IO_Conn_STD_IF
-- generic map
-- (  
--   INFF_Delay => NODELAY,
--   OBUF_Drive => FAST_8mA
-- )
-- port map
-- (  
--   Clk => Clocks_In.M_Clk,
--   Global_Reset => Global_Reset,
--   Pads => Pads.IO_Conn_1,
--   User_In => IO_Conn_1_In,
--   User_Out => IO_Conn_1_Out
-- );

U_LEDs : LED_STD_IF
  port map
  (  
    Pads => Pads.LEDs,
    User_Out => LEDs_Out
  );

-- Readback interface : Use the CAPTURE_VIRTEX block for readback.

-- U_Capture_Readback : CAPTURE_VIRTEX
  port map
  (  
    CAP => Capture
    CLK => Clocks_In.K_Clk
  );
```

-- The following LAD_Mux component provides a global reset signal to
-- the PE. It is mapped to LAD address 0x7fff. To change it’s
-- mapping alter the constant passed to the Base generic.

U_LAD_Mux_Reset: LAD_Mux_Reset
    generic map
    (
        Mask        => x"7F00",
        Base        => x"7F00"
    )
    port map
    (
        Rclk        => Clocks_In.M_Clk,
        Kclk        => Clocks_In.K_Clk,
        LAD         => LAD_Mux_Bus(5),
        Reset       => Global_Reset,
        DLL_Reset_0 => Clocks_Out.M_Clk_DllRst,
        DLL_Reset_1 => Clocks_Out.P_Clk_DllRst,
        DLL_Reset_2 => Clocks_Out.K_Clk_DllRst,
        DLL_Reset_3 => Clocks_Out.U_Clk_DllRst
    );

-- Global reset interface : Attach the Reset_Register signal to a
-- register bit of a LAD bus accessible register. This reset
-- mechanism generates a one K_Clk cycle long pulse to the GSR line
-- of the STARTUP block. The STARTUP block is also synchronous to
-- K_Clk.

U_Reset_Pulse_Gen : One_Shot
    port map
    (  
        Clk           => Clocks_In.K_Clk,
        I             => Reset_Register,
        O             => Global_Reset
    );

U_Startup : STARTUP_VIRTEX_GSR
    port map
    (  
        GSR           => Global_Reset
    );

-- NOTE : The following line must remain in all designs
-- to ensure that all of the PE pads are driven.

Init_PE0_Pads ( Pads );

end PEO_Behavior;
WAVELET TRANSFORM CODE FOR WILDCHILD

C.1 WILDCHILD Interface for Wavelet Transform
C.2 Wavelet Transform Architecture
C.3 Wavelet Transform Host Software
C.4 Wavelet Transform in MATLAB
C.1 WILDCHILD Interface for Wavelet Transform

---
-- Copyright (C) 1995, 1996 Annapolis Micro Systems, Inc.
--
--- (C) 1992 United States Government as represented
-- by the Secretary of the Department of Defense.
-- All Rights reserved.
---

-- Entity : PE0_Core
--
-- Architecture : wavelet
--
-- Filename : pe0_wavelet.vhd
--
-- Description : This code reads an image (512x512) from the memory and
--               : perform a wavelet on the image. The code perform a row
--               : operation followed by a column operation on each block
--               : of the image.
--
-- Date : 9/2/99
---

-------------------------------------------------------------

---------------------- Library Declarations ----------------------
library IEEE;
use IEEE.std_logic_1164.all;
use WORK.WF_Package.all;

architecture pe0_wavelet of PE0_Core is

component wavelet -- use the Entity "wavelet" in this model
    PORT {
        PE_Pclk             : IN  std_logic;
        PE_Reset            : IN  std_logic;
        PE_MemBusGrant_n    : IN  std_logic;
        PE_MemData_InReg    : IN  std_logic_vector(31 DOWNTO 0);
        PE_MemData_OutReg   : OUT std_logic_vector(31 DOWNTO 0);
        PE_MemAddr_OutReg   : OUT std_logic_vector(17 DOWNTO 0);
        DONE_wavelet        : OUT std_logic;
        PE_MemWriteSel_n    : OUT std_logic;
        PE_MemStrobe_n      : OUT std_logic
    };
end component;

begin

106
-- "Inactive" output port signal assignments

F0_Data_Out <= ( others => '0' );  -- Set FIFO '0' data to zero
F0_Data_OE <= '0';                -- Disable data to FIFO '0'
F0_EN     <= '0';                -- Disable FIFO '0' read/write
F0_CS_n   <= '1';                -- Deselect FIFO '0'
F0_WE_n   <= '1';                -- Disable FIFO '0' write mode
F1_EN     <= '0';                -- Disable FIFO '1' read/write
F1_CS_n   <= '1';                -- Deselect FIFO '1'
F1_WE_n   <= '1';                -- Disable FIFO '1' write mode
Fn_EN     <= '0';                -- Disable FIFO 'n' read/write
Fn_CS_n   <= '1';                -- Deselect FIFO 'n'
Fn_WE_n   <= '1';                -- Disable FIFO 'n' write mode
XB0_Out   <= ( others => '0' );  -- Set crossbar port '0' data to zero
XB0_OE    <= ( others => '0' );  -- Disable data to crossbar port '0'
XB0_En_n  <= ( others => '1' );  -- Disable data from crossbar port '0'
XB0_SEL   <= ( others => '1' );  -- Select crossbar Config 0
XB0_BC_n  <= '1';                -- Disable crossbar bcast mode
XBm_Out   <= ( others => '0' );  -- Set crossbar port 'm' data to zero
XBm_OE    <= ( others => '0' );  -- Disable data to crossbar port 'm'
XBm_En_n  <= '1';                -- Disable data from crossbar port 'm'
-- Maddr_OutFF  <= ( others => '0' );  -- Set memory address to zero
-- Mdata_OutFF  <= ( others => '0' );  -- Set memory output data to zero
-- Mreq_n      <= '1';                -- Disable request for memory access
-- MwriteEn_n  <= '1';                -- Disable memory write mode
SW0_EN_n   <= '1';                -- Disable switch '0'
SW0_SEL   <= '0';                -- Select XB0 to connect to SIMD
SW1_EN_n   <= '1';                -- Disable switch '1'
SW1_SEL   <= '0';                -- Select PE1 to connect to bottom systolic
SWn_EN_n   <= '1';                -- Disable switch 'n'
SWn_SEL   <= '0';                -- Select PEn to connect to bottom systolic
PclkStop_n <= '1';                -- Enable Pclk to run
-- Intr      <= '0';                -- Disable interrupt to host
BC_DPMC_Out <= '0';                -- Set broadcast DPMC to '0'
BC_DPMC_OE <= '0';                -- Disable broadcast to DPMC
HS0_Out   <= '0';                -- Set handshake '0' to host to a '0'
HS0_OE   <= '0';                -- Disable handshake '0' to host
HS1_Out   <= '0';                -- Set handshake '1' to host to a '0'
HS1_OE   <= '0';                -- Disable handshake '1' to
-- host
GORR0_Out <= '0'; -- Set GOR result '0' to host to a '0'
GORV0_Out <= '0'; -- Set GOR valid '0' to host to a '0'
GORRi_Out <= (others => '0'); -- Set GOR result 'i' to PE to a '0'
GORRi_OE <= (others => '0'); -- Disable GOR result 'i' to PE
GORVi_Out <= (others => '0'); -- Disable GOR valid 'i' to PE
GORVi_OE <= (others => '0'); -- Disable GOR valid 'i' to PE
VME_A_Out <= '0'; -- Set VME backplane signal -- 'A' to a '0'
VME_A_OE_n <= '1'; -- Disable VME backplane signal 'A'
VME_B_Out <= '0'; -- Set VME backplane signal -- 'B' to a '0'
VME_B_OE_n <= '1'; -- Disable VME backplane signal 'B'
LED <= '0'; -- Set LED signal to a '0'

wavelet_inst : wavelet
port map (
  PE_Pclk => Pclk,
  PE_Reset => reset,
  DONE_wavelet => Intr,
  PE_MemData_InReg => Mdata_InFF(31 downto 0),
  PE_MemData_OutReg => Mdata_OutFF(31 downto 0),
  PE_MemAddr_OutReg => Maddr_OutFF(17 downto 0),
  PE_MemBusGrant_n => Mgnt_n,
  PE_MemStrobe_n => Mreq_n,
  PE_MemWriteSel_n => MwriteEn_n
);

end pe0_wavelet;
C.2 Wavelet Transform Architecture

---
-- Entity        : wavelet
-- Filename      : wavelet.vhd
-- Description   : Performs a "Wavelet Transform" on a 512 x 512 image
-- Date          : 2/3/00
-- Author        : David C. Zaretsky, Northwestern University
---

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_unsigned.ALL;
use ieee.numeric_std.all;

ENTITY  wavelet IS
  PORT ( PE_Pclk             : IN  std_logic;
         PE_Reset            : IN  std_logic;
         PE_MemBusGrant_n    : IN  std_logic;
         PE_MemData_InReg    : IN  std_logic_vector(31 DOWNTO 0);
         PE_MemData_OutReg   : OUT std_logic_vector(31 DOWNTO 0);
         DONE_wavelet        : OUT std_logic;
         PE_MemWriteSel_n    : OUT std_logic;
         PE_MemStrobe_n      : OUT std_logic;
         PE_MemAddr_OutReg   : OUT std_logic_vector(17 DOWNTO 0)
  );
END wavelet;

architecture behavior of wavelet is

  type state_values is (init_state, fwd_wavelet_st0, fwd_wavelet_st0a,
                       fwd_wavelet_st1, fwd_wavelet_st2, fwd_wavelet_st3,
                       fwd_wavelet_st4, fwd_wavelet_st5, fwd_wavelet_st6,
                       fwd_wavelet_st7, fwd_wavelet_st8, fwd_wavelet_st9,
                       fwd_wavelet_st10, fwd_wavelet_st11, fwd_wavelet_st12, fcdf22_st0,
                       fcdf22_st1, fcdf22_st2, fcdf22_st2a, fcdf22_st3, fcdf22_st4,
                       fcdf22_st5, fcdf22_st6, fcdf22_st7, fcdf22_st7a, fcdf22_st8,
                       fcdf22_st8a, fcdf22_st8b, fcdf22_st8c, fcdf22_st8d, fcdf22_st9,
                       fcdf22_st10, fcdf22_st10a, fcdf22_st10b, fcdf22_st11,
                       fcdf22_st11a, fcdf22_st11b, fcdf22_st11c, fcdf22_st11d,
                       fcdf22_st12, fcdf22_st12a, fcdf22_st12b, fcdf22_st13,
                       fcdf22_st13a, fcdf22_st14, fcdf22_st15, fcdf22_st16,
                       fcdf22_st16a, fcdf22_st17, fcdf22_st17a, fcdf22_st18,
                       fcdf22_st18a, fcdf22_st18b, fcdf22_st19, fcdf22_st19a,
                       fcdf22_st19b, fcdf22_st19c, fcdf22_st20, end_state);
signal state           : state_values;
signal ret_state      : state_values;
signal mem_req_n      : std_logic;
signal rw_n           : std_logic;
signal addr_sig       : std_logic_vector(17 downto 0);
signal st             : std_logic_vector(9 downto 0);
signal nt             : std_logic_vector(9 downto 0);
signal addr_offset    : std_logic_vector(17 downto 0);
signal si             : std_logic_vector(31 downto 0);
signal di             : std_logic_vector(31 downto 0);
signal si_next        : std_logic_vector(31 downto 0);
signal di_prev        : std_logic_vector(31 downto 0);
signal di_and_si      : std_logic_vector(31 downto 0);
signal di_and_si_new  : std_logic_vector(31 downto 0);

BEGIN  -- behavior

CONTROLER : process (PE_Pclk, PE_Reset)

variable s_addr_in           : std_logic_vector(19 downto 0);
variable d_addr_in           : std_logic_vector(19 downto 0);
variable s_addr_temp         : std_logic_vector(19 downto 0);
variable d_addr_out          : std_logic_vector(19 downto 0);
variable d_addr_out_temp     : std_logic_vector(19 downto 0);
variable di_and_si_addr      : std_logic_vector(19 downto 0);
variable di_and_si_addr_temp : std_logic_vector(19 downto 0);
variable di_temp             : std_logic_vector(31 downto 0);
variable si_temp0            : std_logic_vector(31 downto 0);
variable si_temp1            : std_logic_vector(31 downto 0);
variable si_temp2            : std_logic_vector(31 downto 0);
variable nt_temp             : std_logic_vector(9 downto 0);
variable mid                 : std_logic_vector(9 downto 0);
variable count1              : std_logic_vector(9 downto 0);
variable count2              : std_logic_vector(9 downto 0);
variable count3              : std_logic_vector(9 downto 0);
variable ROW_count           : std_logic_vector(18 downto 0);
variable COL_count           : std_logic_vector(18 downto 0);
variable count_limit         : std_logic_vector(18 downto 0);
variable ROW                 : std_logic_vector(9 downto 0);
variable COL                 : std_logic_vector(9 downto 0);
variable BLOCK_SIZE          : std_logic_vector(9 downto 0);
variable done_counter        : std_logic_vector(4 downto 0);

BEGIN   -- process
if (PE_Reset = '1') then
  addr_sig <= (others => '0');
  DONE_wavelet <= '0';
  mem_req_n <= '1';
  rw_n <= '1';
  state <= init_state;
  nt <= (others => '0');
  n <= (others => '0');
  si <= (others => '0');
  di <= (others => '0');
  di_and_si <= (others => '0');
  st <= (others => '0');
  di_and_si_new <= (others => '0');
  si_next <= (others => '0');
  di_prev <= (others => '0');
  addr_offset <= (others => '0');
  s_addr_temp := (others => '0');
  done_counter := (others => '0');
  count1 := (others => '0');
  count2 := (others => '0');
  count3 := (others => '0');
  s_addr_in := (others => '0');
  d_addr_in := (others => '0');
  s_addr_out := (others => '0');
  d_addr_out := (others => '0');
  d_addr_out_temp := (others => '0');
  di_and_si_addr := (others => '0');
  di_and_si_addr_temp := (others => '0');
  di_temp := (others => '0');
  si_temp0 := (others => '0');
  si_temp1 := (others => '0');
  si_temp2 := (others => '0');
  nt_temp := (others => '0');
  mid := (others => '0');
  ROW_count := (others => '0');
  COL_count := (others => '0');
  count_limit := (others => '0');
  ROW := "000000001";
  COL := "1000000000"; -- '512'
  BLOCK_SIZE := "0010000000"; -- '128'
elsif rising_edge(PE_Pclk) then
  -- reset = '0'
  DONE_wavelet <= '0';
  rw_n <= '1';
  mem_req_n <= '0';

case state is

when init_state =>
  nt <= "1000000000";
  state <= fwd_wavelet_st0;
when fwd_wavelet_st0 =>  
  if (nt >= BLOCK_SIZE) then  
    state <= fwd_wavelet_st0a;  
  else  
    state <= fwd_wavelet_st7;  
  end if;  

when fwd_wavelet_st0a =>  
  nt_temp := nt;  
  count_limit(18 downto 9) := nt(9 downto 0);  
  count_limit(8 downto 0) := "000000000";  
  ROW_count := (others => '0');  
  st <= ROW;  
  n <= nt;  
  state <= fwd_wavelet_st1;  

when fwd_wavelet_st1 =>  
  if (ROW_count < count_limit) then  
    addr_offset <= ROW_count(17 downto 0);  
    ret_state <= fwd_wavelet_st2;  
    state <= fcdf22_st0;  
  else  
    state <= fwd_wavelet_st3;  
  end if;  

when fwd_wavelet_st2 =>  
  ROW_count := ROW_count + "1000000000";  
  state <= fwd_wavelet_st1;  

when fwd_wavelet_st3 =>  
  -- for (i=0; i<nt;  
  count_limit(9 downto 0) := nt(9 downto 0);  
  count_limit(18 downto 10) := "000000000";  
  COL_count := (others => '0');  
  st <= COL;  
  n <= nt;  
  state <= fwd_wavelet_st4;  

when fwd_wavelet_st4 =>  
  if (COL_count < count_limit) then  
    addr_offset <= COL_count(17 downto 0);  
    ret_state <= fwd_wavelet_st5;  
    state <= fcdf22_st0;  
    -- return state  
  else  
    state <= fwd_wavelet_st6;  
    -- call fcdf22  
  end if;
when fwd_wavelet_st5 =>
    COL_count := COL_count + '1';               -- increment (i++)
    state <= fwd_wavelet_st4;
when fwd_wavelet_st6 =>
    nt(8 downto 0) <= nt_temp(9 downto 1);     -- divide by 2
    nt(9)          <= '0';
    state          <= fwd_wavelet_st0;
when fcdf22_st0 =>
    if (PE_MemBusGrant_n = '0') then
        state <= fcdf22_st1;
    else
        state <= fcdf22_st0;
    end if;
when fcdf22_st1 =>                                 -- mid = (n/2)-1
    mid(8 downto 0)   := n(9 downto 1);             -- divide (n/2)
    mid(9)            := '0';
    mid               := mid - '1';
    count1            := (others => '0');
    count2            := (others => '0');
    count3            := (others => '0');
    state             <= fcdf22_st2;
when fcdf22_st2 =>
    if (st = COL) then
        s_addr_temp(18 downto 9)  := count1(9 downto 0);
        s_addr_temp(8 downto 0)   := "000000000";
        s_addr_temp(19)           := '0';
    else
        s_addr_temp(9 downto 0)   := count1(9 downto 0);
        s_addr_temp(19 downto 10) := (others => '0');
    end if;
    state <= fcdf22_st2a;
when fcdf22_st2a =>
    s_addr_in(19 downto 1) := s_addr_temp(18 downto 0);
    s_addr_in(0)         := '0';
    s_addr_in            := s_addr_in + addr_offset;
    addr_sig            <= s_addr_in(17 downto 0);
    state <= fcdf22_st3;
when fcdf22_st3 =>
  d_addr_in := s_addr_in + st;
  addr_sig  <= d_addr_in(17 downto 0);
  state     <= fcdf22_st4;

when fcdf22_st4 =>
  count2 := count1;
  count1 := count1 + '1';
  if (st = COL) then
    s_addr_temp(18 downto 9) := count1(9 downto 0);
    s_addr_temp(8 downto 0) := "000000000";
    s_addr_temp(19) := '0';
  else
    s_addr_temp(9 downto 0) := count1(9 downto 0);
    s_addr_temp(19 downto 10) := (others => '0');
  end if;
  state <= fcdf22_st5;

when fcdf22_st5 =>
  s_addr_in(19 downto 1) := s_addr_temp(18 downto 0);
  s_addr_in(0) := '0';
  s_addr_in := s_addr_in + addr_offset;
  addr_sig <= s_addr_in(17 downto 0);
  di_prev <= di;
  if (count1 = "000000001") then
    si <= PE_MemData_InReg;
  else
    si <= si_next;
  end if;
  state <= fcdf22_st6;

when fcdf22_st6 =>
  di <= PE_MemData_InReg;
  state <= fcdf22_st7;

when fcdf22_st7 =>
  state <= fcdf22_st7a;

when fcdf22_st7a =>
  if (count1 = (mid + '1')) then
    si_next <= si;
  else
    si_next <= PE_MemData_InReg;
  end if;
  state <= fcdf22_st8;
when fcdf22_st8 =>
  di_temp(31 downto 1) := di(30 downto 0);
  di_temp(0) := '0';
  state <= fcdf22_st8a;

when fcdf22_st8a =>
  di_temp := di_temp - si;
  di <= di_temp - si_next;
  if (count1 = "000000001") then
    di_prev <= di_temp - si_next;
  end if;
  state <= fcdf22_st8b;

when fcdf22_st8b =>
  si_temp0 := si;
  si_temp1 := di_prev + di;
  state <= fcdf22_st8c;

when fcdf22_st8c =>
  si_temp2(28 downto 0) := si_temp1(31 downto 3);
  if (si_temp1(31) = '1') then
    si_temp2(31 downto 29) := "111";
  else
    si_temp2(31 downto 29) := "000";
  end if;
  state <= fcdf22_st8d;

when fcdf22_st8d =>
  si <= si_temp0 + si_temp2;
  state <= fcdf22_st9;

when fcdf22_st9 =>
  di_and_si(31 downto 0) <= di(15 downto 0) & si(15 downto 0);
  state <= fcdf22_st10;

when fcdf22_st10 =>
  if (st = COL) then
    di_and_si_addr(18 downto 9) := count2(9 downto 0);
    di_and_si_addr(8 downto 0) := "000000000";
    di_and_si_addr(19) := '0';
  else
    di_and_si_addr(9 downto 0) := count2(9 downto 0);
    di_and_si_addr(19 downto 10) := (others => '0');
  end if;
  state <= fcdf22_st10a;

when fcdf22_st10a =>
di_and_si_addr := di_and_si_addr + addr_offset;
state <= fcdf22_st10b;

when fcdf22_st10b =>
  rw_n <= '0';
  state <= fcdf22_st11;

when fcdf22_st11 =>
  rw_n <= '0';
  addr_sig (17 downto 0) <= di_and_si_addr(17 downto 0);
  PE_MemData_OutReg(31 downto 0) <= di_and_si(31 downto 0);
  if (count1 <= mid) then
    state <= fcdf22_st11a;
  else
    state <= fcdf22_st11c;
  end if;

when fcdf22_st11a =>
  state <= fcdf22_st11b;

when fcdf22_st11b =>
  state <= fcdf22_st3;

when fcdf22_st11c =>
  state <= fcdf22_st11d;

when fcdf22_st11d =>
  state <= fcdf22_st12;

when fcdf22_st12 =>
  if (st = COL) then
    di_and_si_addr_temp(18 downto 9) := count3(9 downto 0);
    di_and_si_addr_temp(8 downto 0) := "000000000";
    di_and_si_addr_temp(19) := '0';
  else
    di_and_si_addr_temp(9 downto 0) := count3(9 downto 0);
    di_and_si_addr_temp(19 downto 10) := (others => '0');
  end if;
  state <= fcdf22_st12a;

when fcdf22_st12a =>
  di_and_si_addr := di_and_si_addr_temp + addr_offset;
  state <= fcdf22_st12b;
when fcdf22_st12b =>
    addr_sig(17 downto 0)   <= di_and_si_addr(17 downto 0);
    di_and_si(31 downto 0)  <= (others => '0');
    state                   <= fcdf22_st13;

when fcdf22_st13 =>
    state           <= fcdf22_st14;

when fcdf22_st14 =>
    state           <= fcdf22_st15;

when fcdf22_st15 =>
    di_and_si_new(31 downto 16)<= PE_MemData_InReg(31 downto 16);
    di_and_si_new(15 downto 0) <= PE_MemData_InReg(15 downto 0);
    state                   <= fcdf22_st16;

when fcdf22_st16 =>
    di (15 downto 0)     <= di_and_si_new (31 downto 16);
    state                <= fcdf22_st16a;

when fcdf22_st16a =>
    if ( di_and_si_new(31) = '1' ) then
        di (31 downto 16) <= "1111111111111111";
    end if;
    if ( di_and_si_new(31) = '0' ) then
        di (31 downto 16) <= "0000000000000000";
    end if;
    state                <= fcdf22_st17;

when fcdf22_st17 =>
    si (15 downto 0)      <= di_and_si_new (15 downto 0);
    state                 <= fcdf22_st17a;

when fcdf22_st17a =>
    if ( di_and_si_new(15) = '1' ) then
        si (31 downto 16) <= "1111111111111111";
    end if;
    if ( di_and_si_new(15) = '0' ) then
        si (31 downto 16) <= "0000000000000000";
    end if;
    state                 <= fcdf22_st18;

when fcdf22_st18 =>

117
rw_n <= '0';
s_addr_out := di_and_si_addr;
state <= fcdf22_st18a;

when fcdf22_st18a =>
    rw_n <= '0';
    addr_sig <= s_addr_out(17 downto 0);
    PE_MemData_OutReg <= si;
    count3 := count3 + '1';
    state <= fcdf22_st18b;

    when fcdf22_st18b =>
        d_addr_out_temp(9 downto 0) := count3;
        d_addr_out_temp(19 downto 10) := (others => '0');
        state <= fcdf22_st19;

    when fcdf22_st19 =>
        d_addr_out_temp := d_addr_out_temp + mid
        state <= fcdf22_st19a;

    when fcdf22_st19a =>
        if (st = COL) then
            d_addr_out(18 downto 9) := d_addr_out_temp(9 downto 0);
            d_addr_out(8 downto 0) := "0000000000";
            d_addr_out(19) := '0';
        else
            d_addr_out(9 downto 0) := d_addr_out_temp(9 downto 0);
            d_addr_out(19 downto 10) := (others => '0');
        end if;
        state <= fcdf22_st19b;

    when fcdf22_st19b =>
        rw_n <= '0';
        d_addr_out := d_addr_out + addr_offset
        state <= fcdf22_st19c;

    when fcdf22_st19c =>
        rw_n <= '0';
        addr_sig <= d_addr_out(17 downto 0);
        PE_MemData_OutReg <= di;
        state <= fcdf22_st20;

    when fcdf22_st20 =>
        if (count3 <= (mid)) then
            state <= fcdf22_st11c;
        else
            state <= ret_state;
        end if;
end if;

-----------------------------------------
when fwd_wavelet_st7 =>
  DONE_wavelet <= '1';
  if (done_counter = 5) then
    state <= fwd_wavelet_st8;
  else
    done_counter := done_counter + '1';
    state <= fwd_wavelet_st7;
  end if;

when fwd_wavelet_st8 =>
  DONE_wavelet <= '1';
  state <= fwd_wavelet_st9;

when fwd_wavelet_st9 =>
  DONE_wavelet <= '1';
  state <= fwd_wavelet_st10;

when fwd_wavelet_st10 =>
  DONE_wavelet <= '1';
  state <= fwd_wavelet_st11;

when fwd_wavelet_st11 =>
  DONE_wavelet <= '1';
  state <= fwd_wavelet_st12;

when fwd_wavelet_st12 =>
  DONE_wavelet <= '1';
  state <= end_state;

when end_state =>
  mem_req_n <= '1';
  state <= end_state;

when others =>
  null;
end case;
end if;
end process;

PE_MemAddr_OutReg <= addr_sig;
PE_MemWriteSel_n  <= rw_n;
PE_MemStrobe_n    <= mem_req_n;
end behavior;
C.3 Wavelet Transform Host Software

/*********************** Includes *******************************/
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <ctype.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/times.h>

/* Wildfire specific */
#include <wildfire.h>

/*********************** Constants ******************************/
#define TRUE 1
#define FALSE 0
#define RETURN_FAILURE( x )
    if ( x != WF_SUCCESS ) { fprintf ( stdout, "command failed.\n" );
    abort (); return x; }

/*********************** Globals *******************************/
WF_BOARD hBoard;

/* Systolic PE Xilinx type */
DWORD dBoardType;
DWORD dBoardVersion;

/* Systolic PE Xilinx type */
DWORD pex_type;
DWORD pe0_type;

/*********************** Prototypes ******************************/
WF_ERROR WF_ConfigureWildfire ( WF_BOARD hBoard, float *clk_freq );
WF_ERROR WF_ShutdownWildfire ( WORD unit, WF_BOARD hBoard );
WF_ERROR read_and_output_PE_memory (WF_BOARD hBoard);

/*********************** Local Function **************************/
* Function:  main
* Parameters: argc, argv - Command line parameters
* Returns:   void
* Description: Wavelet Transform

120
DWORD interrupt_vect;
struct tms ustart, uend;

main () {
  int tries = 0, t, iterations=100;
  char filename[20];

  /* Interrupt vector */
  DWORD pe_interrupt_vector = 0;

  /* WF_ERROR return value */
  WF_ERROR ret_val = WF_SUCCESS;

  /* Unit and slot variables */
  WORD unit=0;
  WORD slot=1;

  /* Pclk frequency variable */
  float clk_freq;

  /* Initialize variables */
  ret_val = WF_SUCCESS;
  clk_freq = FLOAT(21.0);

  /* Open the WILDFIRE system specified by unit and slot */
  ret_val = WFireOpen ( unit, slot, WF_OPEN_SHARED_BIT, &hBoard, 0L );
  if ( ret_val == WF_SUCCESS ) {
    /* Configure the WILDFIRE system specified by unit and slot */
    ret_val = WF_ConfigureWildfire ( hBoard, &clk_freq );
    if ( ret_val == WF_SUCCESS ) {
      fprintf(stdout,"Disabling reset for all PEs...");
      ret_val = WFireResetPE( hBoard, WF_ALL_PEs, WFDISABLE);
      RETURN_FAILURE(ret_val );
      fprintf(stdout, " DONE.
" );
      times(&ustart);
      for (t=0; t<iterations; t++) {
        pe_interrupt_vector = 0;
        /* wait for the interrupt */
        /* printf("\nWaiting for interrupt from PE0...\n");
        while( ! (pe_interrupt_vector & WF_PE0_BIT) ) { */
        /* tries++;
        sleep(1);
        pe_interrupt_vector = WFireGetPEIntEvents (hBoard);*/
      }
    } else {
      fprintf(stderr,"Configuration failed:"
    }
  } else {
    fprintf(stderr,"Opening WILDFIRE failed:"
  }
} else {
  fprintf(stderr,"Opening WILDFIRE failed:"
}
}

121
WF_ERROR WF_ConfigureWildfire ( WF_BOARD hBoard, float *clk_freq )
{
  /* Function return value */
  WF_ERROR ret_val;
  int i;
  char c;
  DWORD *buf;
  DWORD upper, lower;
  DWORD bpc_mode = WF_BPC0;
  FILE *upper_file, *lower_file;
  int max = 512*512;

  buf = ( DWORD * ) WildfireAlloc ( 262144 * sizeof ( DWORD ) );

  /* Reset the Wildfire board (this may take about 10-15 seconds) */
  fprintf(stdout,"\nReset Board? (y/n) ");
  c=getchar();
  if (c=='y') {
    fprintf(stdout,"Resetting Wildfire, please wait...");
    ret_val = WFResetBoard ( hBoard );
  }
}

}
RETURN_FAILURE(ret_val);
    fprintf(stdout, " DONE.\n");
}

fprintf(stdout,"Enabling PE reset for all PEs...\n");
ret_val = WFireResetPE(hBoard, WF_ALL_PEs, WF_ENABLE);
RETURN_FAILURE(ret_val);
fprintf(stdout, " DONE.\n");

// Mask all PE interrupts so that no spurious asynchronous events are
// scheduled during PE programming.

fprintf(stdout,"Masking all PE interrupts...\n");
ret_val = WFireMaskPEInts(hBoard, WF_ALL_PEs);
RETURN_FAILURE(ret_val);
fprintf(stdout, " DONE.\n");

times(&ustart);

/*
fprintf(stdout,"Programming PE0 with blank design (pe0blank.m68)...\n");
ret_val = WFireProgramPE(hBoard, WF_PE0_BIT, "/export/match/david/versatility/xilinx/pe0blank.m68\n");
RETURN_FAILURE(ret_val);
fprintf(stdout, " DONE.\n");
*/

fprintf(stdout,"Programming PE0 with Wavelet Transform
(wavelet.m68)...\n");
ret_val = WFireProgramPE(hBoard, WF_PE0_BIT, "wavelet.m68");
RETURN_FAILURE(ret_val);
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Programming all PEs with blank design
(pexblank.m68)...\n");
ret_val = WFireProgramPE(hBoard, WF_ALL_PE1_16, "pexblank.m68");
RETURN_FAILURE(ret_val);
fprintf(stdout, " DONE.\n");

fprintf ( stdout, "Loading XBar configuration (xbar_wavelet.cfg)...\n");
ret_val = WFireXBarConfig(hBoard, "xbar_wavelet.cfg");
RETURN_FAILURE(ret_val);
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Clearing PE interrupts...\n");
ret_val = WFireClearPEInts(hBoard, WF_ALL_PEs);
RETURN_FAILURE(ret_val);
fprintf(stdout, " DONE.\n");
fprintf(stdout,"Flushing PE0 asynchronous interrupt events...\n");
ret_val = WFireWaitForPEInt( hBoard, WF_PE0_BIT, WF_FLUSH, 0 );
if ( ret_val == WF_SUCCESS ) {
  fprintf(stdout, "command failed.\n");
} else {
  fprintf(stdout, " DONE.\n");
}

fprintf(stdout,"Unmasking PE0 interrupts...\n");
ret_val = WFireUnMaskPEInts( hBoard, WF_PE0_BIT );
RETURN_FAILURE( ret_val );
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Blocking memory access to PE0...\n");
ret_val = WFireSetMemoryMode( hBoard, WF_PE0_BIT, WF_BLOCKED);
RETURN_FAILURE ( ret_val );
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Blocking memory access to PEXs...\n");
ret_val = WFireSetMemoryMode( hBoard, WF_ALL_PE1_16, WF_BLOCKED);
RETURN_FAILURE ( ret_val );
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Clearing PE0 memory...\n");
ret_val = WFireClearMem( hBoard, WF_PE0_NUM);
RETURN_FAILURE ( ret_val );
fprintf(stdout, " DONE.\n");

times(&uend);
printf("\n\nFPGA Configuration Time: %f
\n\n",((float)(uend.tms_utime+uend.tms_stime -ustart.tms_utime -ustart.tms_stime)/(float)CLK_TCK));
times(&ustart);

fprintf(stdout,"Opening upper file...\n");
upper_file = fopen("pe0upper.mem", "r");
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Opening lower file...\n");
lower_file = fopen("pe0lower.mem", "r");
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Reading data from files...\n");
for (i=0; i<max; i++) {
  fscanf(upper_file, "%d", &upper);
  fscanf(lower_file, "%d", &lower);
  buf[i] = upper<<16 | lower;
}
fprintf(stdout, " DONE.\n");
fclose(upper_file);
fclose(lower_file);

times(&uend);

printf("\nData Prepare Time: %f
\n",((float)(uend.tms_utime+uend.tms_stime
-ustart.tms_utime -ustart.tms_stime)/(float)CLK_TCK));

times(&ustart);

fprintf(stdout,"Writing data to PE0 memory...");
ret_val = WFireWrite ( hBoard, WF_PE0_NUM, 0, 512*512, buf );
fprintf(stdout, " DONE.\n");

times(&uend);

printf("\nTotal FPGA Memory Loading Time : %f
\n",((float)(uend.tms_utime+uend.tms_stime
-ustart.tms_utime -ustart.tms_stime)/(float)CLK_TCK )));

fprintf(stdout,"Setting memory access mode for PE0...");
ret_val = WFireSetMemoryMode( hBoard, WF_PE0_BIT, WF_HW_ARBITRATE );
RETURN_FAILURE ( ret_val );
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Configuring Pclk for local mode...");
ret_val = WFireClkConfig( hBoard, WF_LOC, bpc_mode );
RETURN_FAILURE( ret_val );
fprintf(stdout, " DONE.\n");

fprintf ( stdout, "Setting the frequency to %f MHz...", *clk_freq );
ret_val = WFireClkSetFreq( hBoard, clk_freq );
RETURN_FAILURE( ret_val );
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Free running Pclk...");
ret_val = WFireClkFreeRun( hBoard );
RETURN_FAILURE( ret_val );
fprintf(stdout, " DONE.\n");

return ret_val;
}

/**************************
read_memory
**************************/

WF_ERROR read_and_output_PE_memory (WF_BOARD hBoard)
FILE *wave_mem;
DWORD *output_buf;
int i;
WF_ERROR ret_val;
output_buf = ( DWORD * ) WildfireAlloc ( 512*512 * sizeof ( DWORD ) );

if ( output_buf == ( DWORD * ) NULL ) {
    fprintf(stdout,"Allocating output buffer failed : ");
    abort();
}

fprintf(stdout,"Setting memory access mode for PE0...");
ret_val = WFireSetMemoryMode( hBoard, WF_PE0_BIT, WF_BLOCKED );
fprintf(stdout, " DONE.
");
times(&ustart);

fprintf(stdout,"Reading data from PE0 memory...");
ret_val = WFireRead (hBoard, WF_PE0_NUM, 0, 512*512, output_buf);
RETURN_FAILURE ( ret_val );
fprintf(stdout, " DONE.
");
times(&uend);

printf("\nTotal Time Reading FPGA Memory: %f
\n",((float)(uend.tms_utime+uend.tms_stime -ustart.tms_utime -ustart.tms_stime)/(float)CLK_TCK ));

fprintf ( stdout, "Writing PE0 memory to file (wave.mem)..." );
wave_mem = fopen("wave.mem", "wb");
for (i=0; i<(512*512); i++) {
    fprintf (wave_mem, "%d
", output_buf[i] );
}
fclose(wave_mem);
fprintf(stdout, " DONE.
");

return ret_val;
}

/*******************************************************************************
 * Function    : WF_ShutdownWildfire
 * Parameters  : unit     : Bit3 unit (including VME backplane)
 *               hBoard     : Slot in the unit
 * Returns     : WF_ERROR
 * Description : Shuts down a WILDFIRE system specified by unit and slot.
 * *******************************************************************************/
WF_ERROR WF_ShutdownWildfire ( WORD unit, WF_BOARD hBoard )
{
    WF_ERROR ret_val = WF_SUCCESS;
}
WF_CLOCK_STATUS pclk_status;

/* Mask and clear any pending PE interrupts */

fprintf(stdout,"Masking all PE interrupts...");
ret_val = WFireMaskPEInts( hBoard, WF_ALL_PEs );
RETURN_FAILURE( ret_val );
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Clearing all PE interrupts...");
ret_val = WFireClearPEInts( hBoard, WF_ALL_PEs );
RETURN_FAILURE( ret_val );
fprintf(stdout, " DONE.\n");

fprintf(stdout,"Get clock status...");
ret_val = WFireClkGetStatus ( hBoard, &pclk_status, 5 );
RETURN_FAILURE( ret_val );
fprintf(stdout, " DONE.\n");

if ( pclk_status.run_state == WF_RUNNING )
{
    fprintf(stdout,"Clock is free-running. Stopping Pclk...");
    ret_val = WFireClkSuspend( hBoard );
    RETURN_FAILURE( ret_val );
    fprintf(stdout, " DONE.\n");
}

/* Close the WILDFIRE unit */
fprintf(stdout,"Closing WILDFIRE...");
ret_val = WFireCloseBoard( hBoard );
ret_val = WFireClose( unit );
RETURN_FAILURE( ret_val );
fprintf(stdout, " DONE.\n");

return ret_val;
C.4 Wavelet Transform in MATLAB

function wavelet(input_file)
    int_data = load (input_file);

    clock_start = clock

    block_size = 512*512;
    ROW = 1;
    COL = 512;
    nt = 512;
    times = 0;

    while (nt>=128)
        nt
        max = 512*nt;
        for j = 0:512:(max-1)
            x=int_data(j+1:block_size);
            n=nt;
            st=ROW;

            si=0;
            si_next=0;
            di=0;
            di_prev=0;

            mid=(n/2)-1;

            for i = 0:1:mid
                if (i==0)
                    si = x( (2*i*st)+1);
                else
                    si = si_next;
                end

                di_prev = di;
                di = x( (2*i*st)+st+1 );

                if (i==mid)
                    si_next = x(2*i*st+1);
                else
                    si_next = x(2*(i+1)*st+1);
                end
        end
    end

end
\[ di = di + di - si - si_{next}; \]

\[
\text{if } (i==0)
\quad di_{prev} = di;
\quad \text{end}
\]

\[ si = si + \left( \frac{di_{prev} + di}{8} \right); \]

\[ temp(i+1) = si; \]
\[ temp(i+\text{mid}+1+1) = di; \]
\[ \text{end} \]

\[ \text{for } k = 0:1:\text{mid} \]
\[ x(k*st+1) = temp(k+1); \]
\[ x( (k+\text{mid}+1)*st+1 ) = temp(k+\text{mid}+1+1); \]
\[ \text{end} \]

\[ \text{int\_data}(j+1:\text{block\_size})= x(1:(\text{block\_size}-j)); \]
\[ \text{end}; \]

\%---------- FCDF22 -----------------

\[ x = \text{int\_data}(l+1:\text{block\_size}); \]
\[ n=nt; \]
\[ st=\text{COL}; \]

\[ si=0; \]
\[ si_{next}=0; \]
\[ di=0; \]
\[ di_{prev}=0; \]

\[ \text{mid}=(n/2)-1; \]

\[ \text{for } m = 0:1:\text{mid} \]
\[ \text{if } (m==0) \]
\[ \quad si = x((2*m*st)+1); \]
\[ \quad \text{else} \]
\[ \quad si = si_{next}; \]
\[ \quad \text{end} \]

\[ di_{prev} = di; \]
\[ di = x( (2*m*st)+st+1 ); \]

\[ \text{if } (m==\text{mid}) \]
\[ \quad si_{next} = x(2*m*st+1); \]
\[ \text{else} \]
\[ \quad si_{next} = x(2*m*st+1); \]
\[ \quad \text{end} \]
si_next = x( 2*(m+1)*st+1 );
end

di = di + di - si - si_next;

if (m==0)
di_prev = di;
end

si = si + ( (di_prev + di)/8 );

temp(m+1) = si;
temp(m+mid+1+1) = di;
end

for c = 0:1:mid
  x(c*st+1) = temp(c+1);
  x( (c+mid+1)*st +1) = temp(c+mid+1+1);
end

int_data(l+1:block_size)= x(l:(block_size-l));
end

nt = nt/2;
times = times + 1
end

clock_end = clock
computation_time = clock_end - clock_start

save int_data.dat int_data /ascii
MATRIX MULTIPLICATION
CODE FOR WILDSTAR

D.1 WS Single Processor Matrix Multiplication on PEX
D.2 WS Parallel Processor Matrix Multiplication on PEX
D.3 WS Parallel Processor Matrix Multiplication on PE0
D.4 WS Host Code for Parallel Matrix Multiplication
D.1 WS Single Processor Matrix Multiplication on PEX

---
-- Copyright (C) 1998-2000, Annapolis Micro Systems, Inc.
-- All Rights Reserved.
--

-- Entity        : PEX
--
-- Architecture  : WS_Matrix_Multiply
--
-- Filename      : pex_matrix_multiply.vhd
--
-- Created by    : David Zaretsky (dcz@nwu.edu)
--
-- Date          : 11/28/00
--
-- Description   : Reads in matrix A from Left Mezz Memory and matrix B
-- from Right Mezz Memory of PEX, computes the matrix
-- multiplication (C=A*B) and stores the resulting
-- matrix, C, into the Right Local Memory of PEX.
--

---

-- IEEE Libraries --
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;

-- Wildstar System Libraries -
library SYSTEM;
use SYSTEM.Xilinx_Package.all;
use SYSTEM.AMS_package.all;

-- Wildstar PEx Libraries --
library PEX_Lib;
use PEX_Lib.PE_Package.all;
use PEX_Lib.PE_Mezz_Mem_package.all;
use PEX_Lib.PE_Mezz_Jumper_Package.all;
use PEX_Lib.PEX_Mem32_Mux_pkg.all;
use PEX_Lib.PEX_Mem64_Mux_pkg.all;
use PEX_Lib.PE_LAD_Mux_pkg.all;
use PEX_Lib.LAD_Mem32_Mux_pkg.all;
use PEX_Lib.LAD_Mem64_Mux_pkg.all;

-- LAD Mux Libraries --
library LAD_Mux_Lib;
use LAD_Mux_Lib.LAD_Mux_pkg.all;

-- Mem Mux Libraries --
library Mem32_Mux_Lib, Mem64_Mux_Lib;
use Mem32_Mux_Lib.Mem32_Mux_pkg.all;
use Mem64_Mux_Lib.Mem64_Mux_pkg.all;

-- DMA Mux Libraries --
library DMA_Mux_Lib;
use DMA_Mux_Lib.DMA_Mux_Pkg.all;

------------------------ Entity Declaration ----------------------
entity WS_PEX is
port
(  
  Mclk                 : in      std_logic;
  Kclk                 : in      std_logic;
  Left_Local_Mux       : inout   Mem32_Mux;
  Right_Local_Mux      : inout   Mem32_Mux;
  Left_Mezz_Mux        : inout   Mem64_Mux;
  Right_Mezz_Mux       : inout   Mem64_Mux;
  Left_Mezz_XBar_Mode  : out     std_logic_vector (1 downto 0);
  Right_Mezz_XBar_Mode : out     std_logic_vector (1 downto 0);
  LAD                  : inout   LAD_Mux;
  LEDs_Out             : out     LED_Std_IF_Out_Type;
  reset                : in      std_logic
);
end entity;

------------------------ Architecture Declaration ----------------------
architecture WS_Matrix_Multiply of WS_PEX is

-------------------------------------------------------------------------
--
-- Signal declarations for the base address of each register used
--
-------------------------------------------------------------------------
constant BASE_MASK       : std_logic_vector ( 15 downto 0 ) := x"7FFF";
constant INTERRUPT_BASE  : std_logic_vector ( 15 downto 0 ) := x"0300";
constant addr_offset     : std_logic_vector ( 5 downto 0 ) := "000010";
constant Matrix_C_Addr_offset  : std_logic_vector ( 31 downto 0 ) := x"00000000";
-- constant Matrix_C_Addr_offset  : std_logic_vector ( 31 downto 0 ) := x"00020000";
signal Matrix_A_Rows         : std_logic_vector (31 downto 0);
signal Matrix_A_Cols         : std_logic_vector (31 downto 0);
signal Matrix_A_Addr         : std_logic_vector (31 downto 0);
signal Matrix_B_Rows         : std_logic_vector (31 downto 0);
signal Matrix_B_Cols         : std_logic_vector (31 downto 0);
signal Matrix_B_Addr         : std_logic_vector (31 downto 0);
signal Matrix_C_Rows         : std_logic_vector (31 downto 0);
signal Matrix_C_Cols         : std_logic_vector (31 downto 0);
signal Matrix_C_Addr         : std_logic_vector (31 downto 0);
signal A               : std_logic_vector (31 downto 0);
signal B               : std_logic_vector (31 downto 0);
signal C               : std_logic_vector (63 downto 0);
signal AB               : std_logic_vector (63 downto 0);
signal i               : std_logic_vector ( 9 downto 0);
signal j               : std_logic_vector ( 9 downto 0);
signal k               : std_logic_vector ( 9 downto 0);
signal Interrupt             : std_logic_vector (31 downto 0);
signal Left_Local_Data_In    : std_logic_vector (31 downto 0);
signal Left_Local_Data_Out   : std_logic_vector (31 downto 0);
signal Left_Local_Addr       : std_logic_vector (31 downto 0);
signal Left_Local_ready      : std_logic;
signal Left_Local_read       : std_logic;
signal Left_Local_write      : std_logic;
signal Right_Local_Data_In   : std_logic_vector (31 downto 0);
signal Right_Local_Data_Out  : std_logic_vector (31 downto 0);
signal Right_Local_Addr      : std_logic_vector (31 downto 0);
signal Right_Local_ready     : std_logic;
signal Right_Local_read      : std_logic;
signal Right_Local_write     : std_logic;
signal Left_Mezz_Data_In     : std_logic_vector (63 downto 0);
signal Left_Mezz_Data_Out    : std_logic_vector (63 downto 0);
signal Left_Mezz_Addr        : std_logic_vector (31 downto 0);
signal Left_Mezz_ready       : std_logic;
signal Left_Mezz_read        : std_logic;
signal Left_Mezz_write       : std_logic;
signal Right_Mezz_Data_In    : std_logic_vector (63 downto 0);
signal Right_Mezz_Data_Out   : std_logic_vector (63 downto 0);
signal Right_Mezz_Addr       : std_logic_vector (31 downto 0);
signal Right_Mezz_ready      : std_logic;
signal Right_Mezz_read       : std_logic;
signal Right_Mezz_write      : std_logic;

type state_values is (init_state, state_1, state_2, state_2a, state_3,
state_3a, state_3b, state_3c, state_3d, state_4, state_5,
state_6, state_6a, state_6b, state_7, state_7a, state_7b,
state_8, state_9, state_9a, state_9b, state_9c, state_9d,
state_10, state_10a, state_10b, state_10c, state_10d,
state_10e, state_10f, state_10g, state_10h, state_11, state_12, state_13, state_14, state_15, end_state);

signal state              : state_values;
signal Left_Local_state   : state_values;
signal Right_Local_state  : state_values;
signal Left_Mezz_state    : state_values;
signal Right_Mezz_state   : state_values;

begin

-- The P Interrupt process is used to send an Interrupt signal to the host, via the LAD Bus.

P Interrupt : process ( reset, Kclk, LAD, Interrupt )
begin

if ( reset = '1' ) then                     -- reset
  LAD.Akk                    <= '0';
  LAD.Data_Out               <= ( others => '0' );
elsif ( rising_edge (Kclk) ) then
  if ( LAD.Strobe = '1' ) then             -- read from Interrupt register
    if ( LAD.Write = '0' ) then
      if ( ( LAD.Addr ( INTERRUPT_BASE'range ) and BASE_MASK ) = INTERRUPT_BASE ) then
        LAD.Akk                 <= '1';
        LAD.Data_Out            <= Interrupt;
      end if;
    end if;
  elsif ( LAD.Strobe = '0' ) then
    LAD.Akk                 <= '0';
  end if;
end if;

end process;
The P_Left_Local_Mem process is for access to Left Local Memory.

begin
if ( reset = '1' ) then  -- Initializations
    Left_Local_Mux.Addr    <= (others => '0');
    Left_Local_Mux.Data_Out <= (others => '0');
    Left_Local_Mux.Req     <= '0';
    Left_Local_Mux.Write   <= '0';
    Left_Local_Data_In     <= (others => '0');
    Left_Local_ready       <= '0';
    Left_Local_state       <= init_state;
elsif ( rising_edge (Mclk) ) then
    Left_Local_Mux.Write   <= '0';
    Left_Local_Mux.Req     <= '0';
    
    case Left_Local_state is
    when init_state =>
        if (Left_Local_read = '1') then
            Left_Local_state <= state_1;
        elsif (Left_Local_write = '1') then
            Left_Local_state <= state_4;
        else
            Left_Local_state <= init_state;
        end if;
    when state_1 =>
        Left_Local_Mux.Req  <= '1';
        Left_Local_Mux.Addr <= Left_Local.Addr;
        if (Left_Local_Mux.Akk = '1') then
            Left_Local_state  <= state_2;
        else
            Left_Local_state  <= state_1;
        end if;
    when state_2 =>
        Left_Local_Mux.Req  <= '1';
if (Left_Local_Mux.Data_Valid = '1') then
  Left_Local_Data_In          <= Left_Local_Mux.Data_In;
  Left_Local_ready            <= '1';
  Left_Local_state            <= state_3;
else
  Left_Local_state            <= state_2;
end if;

when state_3 =>
  if (Left_Local_Mux.Data_Valid = '0') then
    Left_Local_ready            <= '0';
    Left_Local_state            <= init_state;
  else
    Left_Local_state            <= state_3;
  end if;

when state_4 =>
  Left_Local_Mux.Req     <= '1';
  Left_Local_Mux.Write   <= '1';
  Left_Local_Mux.Addr    <= Left_Local_Addr;
  if (Left_Local_Mux.Akk = '1') then
    Left_Local_Mux.Data_Out     <= Left_Local_Data_Out;
    Left_Local_ready            <= '1';
    Left_Local_state            <= state_5;
  else
    Left_Local_state            <= state_4;
  end if;

when state_5 =>
  Left_Local_Mux.Req     <= '0';
  Left_Local_Mux.Write   <= '0';
  Left_Local_ready       <= '0';
  Left_Local_state       <= init_state;

when others =>
  null;
end case;
end if;
end process;

-- The P_Right_Local_Mem process is for access to Right Local Memory.

P_Right_Local_Mem : process ( reset, Mclk, Right_Local_Mux, Right_Local_read, Right_Local_write )
begin
if ( reset = '1' ) then -- Initializations
Right_Local_Mux.Addr <= (others => '0');
Right_Local_Mux.Data_Out <= (others => '0');
Right_Local_Mux.Req <= '0';
Right_Local_Mux.Write <= '0';
Right_Local_Data_In <= (others => '0');
Right_Local_ready <= '0';
Right_Local_state <= init_state;
elsif ( rising_edge (Mclk) ) then
Right_Local_Mux.Write <= '0';
Right_Local_Mux.Req <= '0';
case Right_Local_state is
when init_state =>
Right_Local_ready <= '0';
if (Right_Local_read = '1') then
Right_Local_state <= state_1;
elsif (Right_Local_write = '1') then
Right_Local_state <= state_4;
else
Right_Local_state <= init_state;
end if;
when state_1 =>
Right_Local_Mux.Req <= '1';
Right_Local_Mux.Addr <= Right_Local_Addr;
if (Right_Local_Mux.Akk = '1') then
Right_Local_state <= state_2;
else
Right_Local_state <= state_1;
end if;
when state_2 =>
Right_Local_Mux.Req <= '1';
if (Right_Local_Mux.Data_Valid = '1') then
Right_Local_Data_In <= Right_Local_Mux.Data_In;
Right_Local_ready <= '1';
Right_Local_state <= state_3;
else
Right_Local_state <= state_2;
end when;
end case;
end if;
end process;
end if;

when state_3 =>
  if (Right_Local_Mux.Data_Valid = '0') then
    Right_Local_ready <= '0';
    Right_Local_state <= init_state;
  else
    Right_Local_state <= state_3;
  end if;

when state_4 =>
  Right_Local_Mux.Req <= '1';
  Right_Local_Mux.Write <= '1';
  Right_Local_Mux.Addr <= Right_Local_Addr;
  if (Right_Local_Mux.Akk = '1') then
    Right_Local_Mux.Data_Out <= Right_Local_Data_Out;
    Right_Local_ready <= '1';
    Right_Local_state <= state_5;
  else
    Right_Local_state <= state_4;
  end if;

when state_5 =>
  Right_Local_Mux.Req <= '0';
  Right_Local_Mux.Write <= '0';
  Right_Local_ready <= '0';
  Right_Local_state <= init_state;

when others =>
  null;
end case;
end if;
end process;

------------------------------------------------------------------
-- The P_Left_Mezz_Mem process is for access to Left Mezz Memory.
--------------------------------------------------------------------

P_Left_Mezz_Mem : process ( reset, Mclk, Left_Mezz_Mux, Left_Mezz_read, Left_Mezz_write )
begin
if ( reset = '1' ) then -- Initializations
  Left_Mezz_Mux.Addr <= (others => '0');
  Left_Mezz_Mux.Data_Out <= (others => '0');
  Left_Mezz_Mux.Req <= '0';
  Left_Mezz_Mux.Write <= '0';
  Left_Mezz_Mux.Low_Enable <= '1';
  Left_Mezz_Mux.High_Enable <= '1';

  Left_Mezz_Data_In <= (others => '0');
  Left_Mezz_ready <= '0';
  Left_Mezz_state <= init_state;

elsif ( rising_edge (Mclk) ) then
  Left_Mezz_Mux.Write <= '0';
  Left_Mezz_Mux.Req <= '0';
  Left_Mezz_Mux.Low_Enable <= '1';
  Left_Mezz_Mux.High_Enable <= '1';

  case Left_Mezz_state is
  when init_state =>
    Left_Mezz_ready <= '0';
    if (Left_Mezz_read = '1') then
      Left_Mezz_state <= state_1;
    elsif (Left_Mezz_write = '1') then
      Left_Mezz_state <= state_4;
    else
      Left_Mezz_state <= init_state;
    end if;

  when state_1 =>
    Left_Mezz_Mux.Req <= '1';
    Left_Mezz_Mux.Addr <= Left_Mezz_Addr;
    if (Left_Mezz_Mux.Akk = '1') then
      Left_Mezz_state <= state_2;
    else
      Left_Mezz_state <= state_1;
    end if;

  when state_2 =>
    Left_Mezz_Mux.Req <= '1';
    if (Left_Mezz_Mux.Low_Data_Valid = '1') and
        (Left_Mezz_Mux.High_Data_Valid = '1') then
      Left_Mezz_Data_In <= Left_Mezz_Mux.Data_In;
      Left_Mezz_ready <= '1';
      Left_Mezz_state <= state_3;
    else

      140
when state_3 =>
  if (Left_Mezz_Mux.Low_Data_Valid = '0') and
    (Left_Mezz_Mux.High_Data_Valid = '0') then
    Left_Mezz_ready <= '0';
    Left_Mezz_state <= init_state;
  else
    Left_Mezz_state <= state_3;
  end if;

when state_4 =>
  Left_Mezz_Mux.Req  <= '1';
  Left_Mezz_Mux.Write <= '1';
  Left_Mezz_Mux.Addr  <= Left_Mezz_Addr;
  if (Left_Mezz_Mux.Akk = '1') then
      Left_Mezz_Mux.Data_Out <= Left_Mezz_Data_Out;
      Left_Mezz_ready <= '1';
      Left_Mezz_state <= state_5;
  else
      Left_Mezz_state <= state_4;
  end if;

when state_5 =>
  Left_Mezz_Mux.Req  <= '0';
  Left_Mezz_Mux.Write <= '0';
  Left_Mezz_ready  <= '0';
  Left_Mezz_state  <= init_state;

when others =>
  null;
end case;
end if;
end process;

-- The P_Right_Mezz_Mem process is for access to Right Mezz Memory.
--
begin
if ( reset = '1' ) then  -- Initializations
    Right_Mezz_Mux.Addr       <= (others => '0');
    Right_Mezz_Mux.Data_Out    <= (others => '0');
    Right_Mezz_Mux.Req         <= '0';
    Right_Mezz_Mux.Write       <= '0';
    Right_Mezz_Mux.Low_Enable  <= '1';
    Right_Mezz_Mux.High_Enable <= '1';
    Right_Mezz_Data_In         <= (others => '0');
    Right_Mezz_ready           <= '0';
    Right_Mezz_state           <= init_state;
elsif ( rising_edge (Mclk) ) then
    Right_Mezz_Mux.Write       <= '0';
    Right_Mezz_Mux.Req         <= '0';
    Right_Mezz_Mux.Low_Enable  <= '1';
    Right_Mezz_Mux.High_Enable <= '1';
case Right_Mezz_state is
when init_state =>
    Right_Mezz_ready           <= '0';
    if (Right_Mezz_read = '1') then
        Right_Mezz_state <= state_1;
    elsif (Right_Mezz_write = '1') then
        Right_Mezz_state <= state_4;
    else
        Right_Mezz_state <= init_state;
    end if;
when state_1 =>
    Right_Mezz_Mux.Req       <= '1';
    Right_Mezz_Mux.Addr       <= Right_Mezz_Addr;
    if (Right_Mezz_Mux.Akk = '1') then
        Right_Mezz_state <= state_2;
    else
        Right_Mezz_state <= state_1;
    end if;
when state_2 =>
    Right_Mezz_Mux.Req       <= '1';
    if (Right_Mezz_Mux.Low_Data_Valid = '1') and
        (Right_Mezz_Mux.High_Data_Valid = '1') then
        Right_Mezz_Data_In       <= Right_Mezz_Mux.Data_In;
        Right_Mezz_ready         <= '1';
        Right_Mezz_state         <= state_3;
end case;
end if;
else
    Right_Mezz_state <= state_2;
end if;

when state_3 =>
  if (Right_Mezz_Mux.Low_Data_Valid = '0') and
    (Right_Mezz_Mux.High_Data_Valid = '0') then
    Right_Mezz_ready <= '0';
    Right_Mezz_state <= init_state;
  else
    Right_Mezz_state <= state_3;
  end if;
when state_4 =>
  Right_Mezz_Mux.Req <= '1';
  Right_Mezz_Mux.Write <= '1';
  Right_Mezz_Mux.Addr <= Right_Mezz_Addr;
  if (Right_Mezz_Mux.Akk = '1') then
    Right_Mezz_Mux.Data_Out <= Right_Mezz_Data_Out;
    Right_Mezz_ready <= '1';
    Right_Mezz_state <= state_5;
  else
    Right_Mezz_state <= state_4;
  end if;
when state_5 =>
  Right_Mezz_Mux.Req <= '0';
  Right_Mezz_Mux.Write <= '0';
  Right_Mezz_ready <= '0';
  Right_Mezz_state <= init_state;
when others =>
  null;
end case;
end if;
end process;

---
-- The P_Matrix_Multiply process computes C=A*B, where matrix A is
-- stored in the Left Mezz Memory and matrix B is located in the Right
-- Mezz Memory. The output, matrix C, is written to the Right Local
-- Memory.
--
P_Matrix_Multiply : process ( reset, Mclk, LAD,
    Left_Local_ready, Right_Local_ready,
    Left_Mezz_ready, Right_Mezz_ready) begin
    if ( reset = '1' ) then -- Initializations
        LEDs_Out.Red_n         <= '1';
        LEDs_Out.Green_n       <= '0';
        Matrix_A_Rows          <= (others => '0');
        Matrix_A_Cols          <= (others => '0');
        Matrix_A_Addr          <= (others => '0');
        Matrix_B_Rows          <= (others => '0');
        Matrix_B_Cols          <= (others => '0');
        Matrix_B_Addr          <= (others => '0');
        Matrix_C_Rows          <= (others => '0');
        Matrix_C_Cols          <= (others => '0');
        Matrix_C_Addr          <= (others => '0');
        A                      <= (others => '0');
        B                      <= (others => '0');
        C                      <= (others => '0');
        AB                     <= (others => '0');
        i                      <= (others => '0');
        j                      <= (others => '0');
        k                      <= (others => '0');
        Left_Local_Data_Out    <= (others => '0');
        Left_Local_Addr        <= (others => '0');
        Left_Local_read        <= '0';
        Left_Local_write       <= '0';
        Right_Local_Data_Out   <= (others => '0');
        Right_Local_Addr       <= (others => '0');
        Right_Local_read       <= '0';
        Right_Local_write      <= '0';
        Left_Mezz_Data_Out     <= (others => '0');
        Left_Mezz_Addr         <= (others => '0');
        Left_Mezz_read         <= '0';
        Left_Mezz_write        <= '0';
        Right_Mezz_Data_Out    <= (others => '0');
        Right_Mezz_Addr        <= (others => '0');
        Right_Mezz_read        <= '0';
        Right_Mezz_write       <= '0';
        Left_Mezz_XBar_Mode    <= "00"; -- PEX -> MEM0, PE0 -> MEM1
        Right_Mezz_XBar_Mode   <= "00"; -- PEX -> MEM0, PE0 -> MEM1
    end if;
Interrupt <= (others => '0');
state <= init_state;

elsif ( rising_edge (Mclk) ) then

  Left_Mezz_XBar_Mode <= "00";  -- PEX -> MEM0, PE0 -> MEM1
  Right_Mezz_XBar_Mode <= "00";  -- PEX -> MEM0, PE0 -> MEM1

  case state is
  when init_state =>
    Matrix_C_Addr <= Matrix_C_Addr_offset;
    if (Left_Local_ready = '0') and (Right_Local_ready = '0') then
      Left_Local_read <= '1';
      Right_Local_read <= '1';
      state <= state_1;
    else
      Left_Local_read <= '0';
      Right_Local_read <= '0';
      state <= init_state;
    end if;

  when state_1 =>
    if (Left_Local_ready = '1') and (Right_Local_ready = '1') then
      Matrix_A_Rows <= Left_Local_Data_In;
      Matrix_B_Rows <= Right_Local_Data_In;
      Left_Local_Addr <= Left_Local_Addr + '1';
      Right_Local_Addr <= Right_Local_Addr + '1';
      Left_Local_read <= '0';
      Right_Local_read <= '0';
      state <= state_2;
    else
      state <= state_1;
    end if;

  when state_2 =>
    if (Left_Local_ready = '0') and (Right_Local_ready = '0') then
      Left_Local_read <= '1';
      Right_Local_read <= '1';
      state <= state_2a;
    else
      Left_Local_read <= '0';
      Right_Local_read <= '0';
      state <= state_2;
    end if;

  when state_2a =>
if (Left_Local_ready = '1') and (Right_Local_ready = '1') then
  Matrix_A_Cols         <= Left_Local_Data_In;
  Matrix_B_Cols         <= Right_Local_Data_In;
  Left_Local_read       <= '0';
  Right_Local_read      <= '0';
  state                 <= state_3;
else
  state                 <= state_2a;
end if;

when state_3 =>
  Matrix_C_Rows         <= Matrix_A_Rows;
  Matrix_C_Cols         <= Matrix_B_Cols;
  state                 <= state_3a;

when state_3a =>
  Right_Mezz_Addr                    <= Matrix_C_Addr_offset;
  Right_Mezz_Data_Out(31 downto 0)   <= Matrix_C_Rows;
  if (Right_Mezz_ready = '0') then
    Right_Mezz_write      <= '1';
    state                 <= state_3b;
  else
    Right_Mezz_write      <= '0';
    state                 <= state_3a;
  end if;

when state_3b =>
  if (Right_Mezz_ready = '1') then
    Right_Mezz_Addr                  <= Right_Mezz_Addr + '1';
    Right_Mezz_Data_Out(31 downto 0) <= Matrix_C_Cols;
    Right_Mezz_write      <= '0';
    state                 <= state_3c;
  else
    state                 <= state_3b;
  end if;

when state_3c =>
  if (Right_Mezz_ready = '0') then
    Right_Mezz_write      <= '1';
    state                 <= state_3d;
  else
    Right_Mezz_write      <= '0';
    state                 <= state_3c;
  end if;

when state_3d =>
  if (Right_Mezz_ready = '1') then
    Right_Mezz_write      <= '0';
state <= state_4;
else
state <= state_3;
end if;

when state_4 =>
if (i < Matrix_A_Rows) then
state <= state_5;
else
i <= (others => '0');
state <= end_state;
end if;

when state_5 =>
if (j < Matrix_B_Cols) then
state <= state_6;
else
i <= i + '1';
j <= (others => '0');
state <= state_4;
end if;

when state_6 =>
if (k < Matrix_A_Cols) then
state <= state_9;
else
Matrix_C_Addr(20 downto 0) <= (i * Matrix_C_Cols(10 downto
0)) + j;
state <= state_7;
end if;

when state_7 =>
Right_Mezz_Addr <= Matrix_C_Addr + Matrix_C_Addr_offset +
addr_offset;
Right_Mezz_Data_Out <= C;
if (Right_Mezz_ready = '0') then
Right_Mezz_write <= '1';
state <= state_8;
else
Right_Mezz_write <= '0';
state <= state_7;
end if;

when state_8 =>
if (Right_Mezz_ready = '1') then
Right_Mezz_write <= '0';
j <= j + '1';
k <= (others => '0');
when state_9 =>
  Matrix_A.Addr(20 downto 0) <= (i * Matrix_A.Cols(10 downto 0)) + k;
  state <= state_10a;

when state_10a =>
  Matrix_B.Addr(20 downto 0) <= (k * Matrix_B.Cols(10 downto 0)) + j;
  state <= state_10b;

when state_10b =>
  Left_Local.Addr <= Matrix_A.Addr + addr_offset;
  Right_Local.Addr <= Matrix_B.Addr + addr_offset;
  state <= state_10c;

when state_10c =>
  if (Left_Local.ready = '0') and (Right_Local.ready = '0') then
    Left_Local.read <= '1';
    Right_Local.read <= '1';
    state <= state_10d;
  else
    Left_Local.read <= '0';
    Right_Local.read <= '0';
    state <= state_10c;
  end if;

when state_10d =>
  if (Left_Local.ready = '1') and (Right_Local.ready = '1') then
    A <= Left_Local.Data_In;
    B <= Right_Local.Data_In;
    Left_Local.read <= '0';
    Right_Local.read <= '0';
    state <= state_11;
  else
    state <= state_10d;
  end if;

when state_11 =>
  AB <= (A * B);
  state <= state_12;

when state_12 =>
C <= C + AB;
k <= k + '1';
state <= state_6;

when end_state =>
  LEDs_Out.Red_n <= '0';
  LEDs_Out.Green_n <= '1';
  Interrupt(0)    <= '1';
  state           <= end_state;

when others =>
  null;
end case;
ed if;
ed process;

del WS_Matrix_Multiply ;
D.2 WS Parallel Processor Matrix Multiplication on PEX

-- Copyright (C) 1998-2000, Annapolis Micro Systems, Inc.
-- All Rights Reserved.
--
-- Entity : PEX
-- Architecture : WS_Matrix_Multiply_Parallel
-- Filename : pex_matrix_multiply_parallel.vhd
-- Created by : David Zaretsky (dcz@nwu.edu)
-- Date : 11/28/00
-- Description : Reads in matrix A from Left Mezz Memory and matrix B
-- from Right Mezz Memory of PEX, computes the matrix
-- multiplication (C=A*B) and stores the result matrix, C, into the Right Local Memory of PEX.
--
--
-- Library Declarations
-- IEEE Libraries
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
-- Wildstar System Libraries
library SYSTEM;
use SYSTEM.Xilinx_Package.all;
use SYSTEM.AMS_package.all;

-- Wildstar PEx Libraries
library PEX_Lib;
use PEX_Lib.PE_Package.all;
use PEX_Lib.PE_Mezz_Mem_package.all;
--use PEX_Lib.PE_Mezz_Jumper_Package.all;
use PEX_Lib.Mezz_Mem_Mux_pkg.all;
use PEX_Lib.Mezz_Mem64_Mux_pkg.all;
use PEX_Lib.PEX_Mem32_Mux_pkg.all;
use PEX_Lib.PE_LAD_Mux_pkg.all;
use PEX_Lib.LAD_Mem32_Mux_pkg.all;
use PEX_Lib.LAD_Mem64_Mux_pkg.all;
-- LAD Mux Libraries --
library LAD_Mux_Lib;
use LAD_Mux_Lib.LAD_Mux_pkg.all;

-- Mem Mux Libraries --
library Mem32_Mux_Lib, Mem64_Mux_Lib;
use Mem32_Mux_Lib.Mem32_Mux_pkg.all;
use Mem64_Mux_Lib.Mem64_Mux_pkg.all;

-- DMA Mux Libraries --
--library DMA_Mux_Lib;
--use DMA_Mux_Lib.DMA_Mux_Pkg.all;

entity WS_PEX is
  port
  (
    Mclk             : in      std_logic;
    Kclk             : in      std_logic;
    Left_Local_Mux   : inout   Mem32_Mux;
    Right_Local_Mux  : inout   Mem32_Mux;
    Left_Mezz_Mux    : inout   Mem64_Mux;
    Right_Mezz_Mux   : inout   Mem64_Mux;
    Left_Mezz_XBar_Mode : out     std_logic_vector (1 downto 0);
    Right_Mezz_XBar_Mode : out     std_logic_vector (1 downto 0);
    LAD               : inout   LAD_Mux;
    LEDs_Out         : out     LED_Std_IF_Out_Type;
    reset            : in      std_logic
  );
end entity;

architecture WS_Matrix_Multiply_Parallel of WS_PEX is

-- Signal declarations for the base address of each register used
--
classification
--constant BASE_MASK      : std_logic_vector ( 15 downto 0 ) := x"7FFF";
classification
constant INTERRUPT_BASE : std_logic_vector ( 15 downto 0 ) := x"0300";
classification
constant addr_offset    : std_logic_vector ( 5 downto 0 ) := "000010";
classification
constant Matrix_C_Addr_offset  : std_logic_vector ( 31 downto 0 ) :=
x"00020000";

signal Matrix_A_Rows         : std_logic_vector (31 downto 0);
signal Matrix_A_Cols         : std_logic_vector (31 downto 0);
signal Matrix_A_Addr : std_logic_vector (31 downto 0);
signal Matrix_B_Rows : std_logic_vector (31 downto 0);
signal Matrix_B_Cols : std_logic_vector (31 downto 0);
signal Matrix_B_Addr : std_logic_vector (31 downto 0);
signal Matrix_C_Rows : std_logic_vector (31 downto 0);
signal Matrix_C_Cols : std_logic_vector (31 downto 0);
signal Matrix_C_Addr : std_logic_vector (31 downto 0);
signal A : std_logic_vector (31 downto 0);
signal B : std_logic_vector (31 downto 0);
signal C : std_logic_vector (31 downto 0);
signal AB : std_logic_vector (63 downto 0);
signal i : std_logic_vector (9 downto 0);
signal j : std_logic_vector (9 downto 0);
signal k : std_logic_vector (9 downto 0);
signal Interrupt : std_logic_vector (31 downto 0);
signal Left_Local_Data_In : std_logic_vector (31 downto 0);
signal Left_Local_Data_Out : std_logic_vector (31 downto 0);
signal Left_Local_Addr : std_logic_vector (31 downto 0);
signal Left_Local_ready : std_logic;
signal Left_Local_read : std_logic;
signal Left_Local_write : std_logic;
signal Right_Local_Data_In : std_logic_vector (31 downto 0);
signal Right_Local_Data_Out : std_logic_vector (31 downto 0);
signal Right_Local_Addr : std_logic_vector (31 downto 0);
signal Right_Local_ready : std_logic;
signal Right_Local_read : std_logic;
signal Right_Local_write : std_logic;
signal Left_Mezz_Data_In : std_logic_vector (63 downto 0);
signal Left_Mezz_Data_Out : std_logic_vector (63 downto 0);
signal Left_Mezz_Addr : std_logic_vector (31 downto 0);
signal Left_Mezz_ready : std_logic;
signal Left_Mezz_read : std_logic;
signal Left_Mezz_write : std_logic;
signal Right_Mezz_Data_In : std_logic_vector (63 downto 0);
signal Right_Mezz_Data_Out : std_logic_vector (63 downto 0);
signal Right_Mezz_Addr : std_logic_vector (31 downto 0);
signal Right_Mezz_ready : std_logic;
signal Right_Mezz_read : std_logic;
signal Right_Mezz_write : std_logic;

type state_values is (init_state, state_1, state_2, state_2a, state_3,
state_3a, state_3b, state_3c, state_3d, state_4, state_5,
state_6, state_6a, state_6b, state_7, state_7a, state_7b,
state_8, state_8a, state_9, state_9a, state_9b, state_9c, state_9d,
state_10, state_10a, state_10b, state_10c, state_10d,
state_10e, state_10f, state_10g, state_10h, state_11,
state_12, state_12a, state_13, state_14, state_15, end_state);
signal state : state_values;
signal Left_Local_state : state_values;
signal Right_Local_state : state_values;
signal Left_Mezz_state    : state_values;
signal Right_Mezz_state   : state_values;

begin

-- The P_Interrupt process is used to send an Interrupt signal to the
-- host, via the LAD Bus.
--
P_Interrupt : process ( reset, Kclk, LAD, Interrupt )
begin

if ( reset = '1' ) then -- reset
  LAD.Akk <= '0';
  LAD.Data_Out <= ( others => '0' );
elsif ( rising_edge (Kclk) ) then
  if ( LAD.Strobe = '1' ) then
    if ( LAD.Write = '0' ) then
      if ( ( LAD.Addr ( INTERRUPT_BASE'range ) and BASE_MASK ) = INTERRUPT_BASE ) then
        LAD.Akk <= '1';
        LAD.Data_Out <= Interrupt;
      end if;
    end if;
  elsif ( LAD.Strobe = '0' ) then
    LAD.Akk <= '0';
  end if;
end if;
end process;

-- The P_Left_Local_Mem process is for access to Left Local Memory.
--
P_Left_Local_Mem : process ( reset, Mclk, Left_Local_Mux, Left_Local_read, Left_Local_write )
begin
  if ( reset = '1' ) then                     -- Initializations
    Left_Local_Mux.Addr        <= (others => '0');
    Left_Local_Mux.Data_Out    <= (others => '0');
    Left_Local_Mux.Req         <= '0';
    Left_Local_Mux.Write       <= '0';
    Left_Local_Data_In         <= (others => '0');
    Left_Local_ready           <= '0';
    Left_Local_state           <= init_state;
  end if;
elsif ( rising_edge (Mclk) ) then
  Left_Local_Mux.Write     <= '0';
  Left_Local_Mux.Req       <= '0';
  case Left_Local_state is
    when init_state =>
      if (Left_Local_read = '1') then
        Left_Local_state            <= state_1;
      elsif (Left_Local_write = '1') then
        Left_Local_state            <= state_4;
      else
        Left_Local_state            <= init_state;
      end if;
    when state_1 =>
      Left_Local_Mux.Req     <= '1';
      if (Left_Local_Mux.Akk = '1') then
        Left_Local_state            <= state_2;
      else
        Left_Local_state            <= state_1;
      end if;
    when state_2 =>
      if (Left_Local_Mux.Data_Valid = '1') then
        Left_Local_Data_In          <= Left_Local_Mux.Data_In;
        Left_Local_ready            <= '1';
        Left_Local_state            <= state_3;
      else
        Left_Local_state           <= state_2;
      end if;
  end case;
end if;
end
Left_Local_state <= state_2;
end if;

when state_3 =>
  if (Left_Local_Mux.Data_Valid = '0') then
    Left_Local_ready <= '0';
    Left_Local_state <= init_state;
  else
    Left_Local_state <= state_3;
  end if;

when state_4 =>
  Left_Local_Mux.Req <= '1';
  Left_Local_Mux.Write <= '1';
  Left_Local_Mux.Addr <= Left_Local_Addr;

  if (Left_Local_Mux.Akk = '1') then
    Left_Local_Mux.Data_Out <= Left_Local_Data_Out;
    Left_Local_ready <= '1';
    Left_Local_state <= state_5;
  else
    Left_Local_state <= state_4;
  end if;

when state_5 =>
  Left_Local_Mux.Req <= '0';
  Left_Local_Mux.Write <= '0';
  Left_Local_ready <= '0';
  Left_Local_state <= init_state;

when others =>
  null;
end case;
end if;
end process;

-- The P_Right_Local_Mem process is for access to Right Local Memory.
--
P_Right_Local_Mem : process ( reset, Mclk, Right_Local_Mux,
  Right_Local_read, Right_Local_write )
begin
    if ( reset = '1' ) then                     -- Initializations
        Right_Local_Mux.Addr        <= (others => '0');
        Right_Local_Mux.Data_Out    <= (others => '0');
        Right_Local_Mux.Req         <= '0';
        Right_Local_Mux.Write       <= '0';
        Right_Local_Data_In         <= (others => '0');
        Right_Local_ready           <= '0';
        Right_Local_state           <= init_state;
    elsif ( rising_edge (Mclk) ) then
        Right_Local_Mux.Write     <= '0';
        Right_Local_Mux.Req       <= '0';

    case Right_Local_state is
        when init_state =>
            Right_Local_ready            <= '0';
            if (Right_Local_read = '1') then
                Right_Local_state            <= state_1;
            elsif (Right_Local_write = '1') then
                Right_Local_state            <= state_4;
            else
                Right_Local_state            <= init_state;
            end if;
        when state_1 =>
            Right_Local_Mux.Req     <= '1';
            Right_Local_Mux.Addr    <= Right_Local_Addr;
            if (Right_Local_Mux.Akk = '1') then
                Right_Local_state            <= state_2;
            else
                Right_Local_state            <= state_1;
            end if;
        when state_2 =>
            Right_Local_Mux.Req     <= '1';
            if (Right_Local_Mux.Data_Valid = '1') then
                Right_Local_Data_In          <= Right_Local_Mux.Data_In;
                Right_Local_ready            <= '1';
                Right_Local_state            <= state_3;
            else
                Right_Local_state            <= state_2;
            end if;
        when state_3 =>

    end case;
end;
if (Right_Local_Mux.Data_Valid = '0') then
    Right_Local_ready <= '0';
    Right_Local_state <= init_state;
else
    Right_Local_state <= state_3;
end if;

when state_4 =>
    Right_Local_Mux.Req <= '1';
    Right_Local_Mux.Write <= '1';
    Right_Local_Mux.Addr <= Right_Local_Addr;
    if (Right_Local_Mux.Akk = '1') then
        Right_Local_Mux.Data_Out <= Right_Local_Data_Out;
        Right_Local_ready <= '1';
        Right_Local_state <= state_5;
    else
        Right_Local_state <= state_4;
    end if;

when state_5 =>
    Right_Local_Mux.Req <= '0';
    Right_Local_Mux.Write <= '0';
    Right_Local_ready <= '0';
    Right_Local_state <= init_state;

when others =>
    null;
end case;
end if;
end process;

-- The P_Left_Mezz_Mem process is for access to Left Mezz Memory.

P_Left_Mezz_Mem : process ( reset, Mclk, Left_Mezz_Mux, Left_Mezz_read, Left_Mezz_write )
begin
    if ( reset = '1' ) then -- Initializations
        Left_Mezz_Mux.Addr <= (others => '0');
        Left_Mezz_Mux.Data_Out <= (others => '0');
        Left_Mezz_Mux.Req <= '0';
    end if;
end process;
Left_Mezz_Mux.Write <= '0';
Left_Mezz_Mux.Low_Enable <= '1';
Left_Mezz_Mux.High_Enable <= '1';
Left_Mezz_Data_In <= (others => '0');
Left_Mezz_ready <= '0';
Left_Mezz_state <= init_state;

elsif ( rising_edge (Mclk) ) then
  Left_Mezz_Mux.Write <= '0';
  Left_Mezz_Mux.Req <= '0';
  Left_Mezz_Mux.Low_Enable <= '1';
  Left_Mezz_Mux.High_Enable <= '1';

case Left_Mezz_state is
  when init_state =>
    Left_Mezz_ready <= '0';
    if (Left_Mezz_read = '1') then
      Left_Mezz_state <= state_1;
    elsif (Left_Mezz_write = '1') then
      Left_Mezz_state <= state_4;
    else
      Left_Mezz_state <= init_state;
    end if;

  when state_1 =>
    Left_Mezz_Mux.Req <= '1';
    Left_Mezz_Mux.Addr <= Left_Mezz_Addr;
    if (Left_Mezz_Mux.Akk = '1') then
      Left_Mezz_state <= state_2;
    else
      Left_Mezz_state <= state_1;
    end if;

  when state_2 =>
    Left_Mezz_Mux.Req <= '1';
    if (Left_Mezz_Mux.Low_Data_Valid = '1') and (Left_Mezz_Mux.High_Data_Valid = '1') then
      Left_Mezz_Data_In <= Left_Mezz_Mux.Data_In;
      Left_Mezz_ready <= '1';
      Left_Mezz_state <= state_3;
    else
      Left_Mezz_state <= state_2;
    end if;
when state_3 =>
  if (Left_Mezz_Mux.Low_Data_Valid = '0') and
  (Left_Mezz_Mux.High_Data_Valid = '0') then
    Left_Mezz_ready <= '0';
    Left_Mezz_state <= init_state;
  else
    Left_Mezz_state <= state_3;
  end if;

when state_4 =>
  Left_Mezz_Mux.Req <= '1';
  Left_Mezz_Mux.Write <= '1';
  Left_Mezz_Mux.Addr <= Left_Mezz_Addr;
  if (Left_Mezz_Mux.Akk = '1') then
    Left_Mezz_Mux.Data_Out <= Left_Mezz_Data_Out;
    Left_Mezz_ready <= '1';
    Left_Mezz_state <= state_5;
  else
    Left_Mezz_state <= state_4;
  end if;

when state_5 =>
  Left_Mezz_Mux.Req <= '0';
  Left_Mezz_Mux.Write <= '0';
  Left_Mezz_ready <= '0';
  Left_Mezz_state <= init_state;

when others =>
  null;
end case;
end if;
end process;

-- The P_Right_Mezz_Mem process is for access to Right Mezz Memory.

P_Right_Mezz_Mem : process ( reset, Mclk, Right_Mezz_Mux,
                              Right_Mezz_read, Right_Mezz_write )
begin
  if ( reset = '1' ) then -- Initializations
    Right_Mezz_Mux.Addr <= (others => '0');
Right_Mezz_Mux.Data_Out <= (others => '0');
Right_Mezz_Mux.Req <= '0';
Right_Mezz_Mux.Write <= '0';
Right_Mezz_Mux.Low_Enable <= '1';
Right_Mezz_Mux.High_Enable <= '1';
Right_Mezz_Data_In <= (others => '0');
Right_Mezz_ready <= '0';
Right_Mezz_state <= init_state;

elsif ( rising_edge (Mclk) ) then
    Right_Mezz_Mux.Write <= '0';
    Right_Mezz_Mux.Req <= '0';
    Right_Mezz_Mux.Low_Enable <= '1';
    Right_Mezz_Mux.High_Enable <= '1';

    case Right_Mezz_state is
        when init_state =>
            Right_Mezz_ready <= '0';
            if (Right_Mezz_read = '1') then
                Right_Mezz_state <= state_1;
            elsif (Right_Mezz_write = '1') then
                Right_Mezz_state <= state_4;
            else
                Right_Mezz_state <= init_state;
            end if;

        when state_1 =>
            Right_Mezz_Mux.Req <= '1';
            Right_Mezz_Mux.Addr <= Right_Mezz_Addr;
            if (Right_Mezz_Mux.Akk = '1') then
                Right_Mezz_state <= state_2;
            else
                Right_Mezz_state <= state_1;
            end if;

        when state_2 =>
            Right_Mezz_Mux.Req <= '1';
            if (Right_Mezz_Mux.Low_Data_Valid = '1') and
               (Right_Mezz_Mux.High_Data_Valid = '1') then
                Right_Mezz_Data_In <= Right_Mezz_Mux.Data_In;
                Right_Mezz_ready <= '1';
                Right_Mezz_state <= state_3;
            else
                Right_Mezz_state <= state_2;
            end if;
    end case;
end if;
when state_3 =>
    if (Right_Mezz_Mux.Low_Data_Valid = '0') and (Right_Mezz_Mux.High_Data_Valid = '0') then
        Right_Mezz_ready <= '0';
        Right_Mezz_state <= init_state;
    else
        Right_Mezz_state <= state_3;
    end if;
when state_4 =>
    Right_Mezz_Mux.Req <= '1';
    Right_Mezz_Mux.Write <= '1';
    Right_Mezz_Mux.Addr <= Right_Mezz_Addr;
    if (Right_Mezz_Mux.Akk = '1') then
        Right_Mezz_Mux.Data_Out <= Right_Mezz_Data_Out;
        Right_Mezz_ready <= '1';
        Right_Mezz_state <= state_5;
    else
        Right_Mezz_state <= state_4;
    end if;
when state_5 =>
    Right_Mezz_Mux.Req <= '0';
    Right_Mezz_Mux.Write <= '0';
    Right_Mezz_ready <= '0';
    Right_Mezz_state <= init_state;
when others =>
    null;
end case;
end if;
end process;

-- The P_Matrix_Multiply process computes C=A*B, where matrix A is stored in the Left Mezz Memory and matrix B is located in the Right Mezz Memory. The output, matrix C, is written to the Right Local Memory.
--
P_Matrix_Multiply : process ( reset, Mclk, LAD,                 
                      Left_Local_ready, Right_Local_ready,   
                      Left_Mezz_ready, Right_Mezz_ready)

begin

  if ( reset = '1' ) then                     -- Initializations
    LEDs_Out.Red_n <= '1';
    LEDs_Out.Green_n <= '0';
    Matrix_A_Rows  <= (others => '0');
    Matrix_A_Cols  <= (others => '0');
    Matrix_A.Addr  <= (others => '0');
    Matrix_B_Rows  <= (others => '0');
    Matrix_B_Cols  <= (others => '0');
    Matrix_B.Addr  <= (others => '0');
    Matrix_C_Rows  <= (others => '0');
    Matrix_C_Cols  <= (others => '0');
    Matrix_C.Addr  <= (others => '0');
    A              <= (others => '0');
    B              <= (others => '0');
    C              <= (others => '0');
    AB             <= (others => '0');
    i              <= (others => '0');
    j              <= (others => '0');
    k              <= (others => '0');
    Left_Local_Data_Out  <= (others => '0');
    Left_Local_Addr    <= (others => '0');
    Left_Local_read    <= '0';
    Left_Local_write   <= '0';
    Right_Local_Data_Out <= (others => '0');
    Right_Local_Addr   <= (others => '0');
    Right_Local_read   <= '0';
    Right_Local_write  <= '0';
    Left_Mezz_Data_Out  <= (others => '0');
    Left_Mezz_Addr     <= (others => '0');
    Left_Mezz_read     <= '0';
    Left_Mezz_write    <= '0';
    Right_Mezz_Data_Out <= (others => '0');
    Right_Mezz_Addr    <= (others => '0');
    Right_Mezz_read    <= '0';
    Right_Mezz_write   <= '0';
    Left_Mezz_XBar_Mode <= "00";   -- PEX->MEM0, PE0->MEM1
    Right_Mezz_XBar_Mode <= "00";   -- PEX->MEM0, PE0->MEM1
    Interrupt        <= (others => '0');
state <= init_state;

elsif (rising_edge(Mclk)) then

    Left_Mezz_XBar_Mode <= "00";  -- PEX->MEM0, PE0->MEM1
    Right_Mezz_XBar_Mode <= "00"; -- PEX->MEM0, PE0->MEM1

    case state is
    when init_state =>
        Matrix_C_Addr <= Matrix_C_Addr_offset;
        if (Left_Mezz_ready = '0') and (Right_Mezz_ready = '0') then
            Left_Mezz_read <= '1';
            Right_Mezz_read <= '1';
            state <= state_1;
        else
            Left_Mezz_read <= '0';
            Right_Mezz_read <= '0';
            state <= init_state;
        end if;

    when state_1 =>
        if (Left_Mezz_ready = '1') and (Right_Mezz_ready = '1') then
            Matrix_A_Rows <= Left_Mezz_Data_In (31 downto 0);
            Matrix_B_Rows <= Right_Mezz_Data_In (31 downto 0);
            Left_Mezz_Addr <= Left_Mezz_Addr + '1';
            Right_Mezz_Addr <= Right_Mezz_Addr + '1';
            Left_Mezz_read <= '0';
            Right_Mezz_read <= '0';
            state <= state_2;
        else
            state <= state_1;
        end if;

    when state_2 =>
        if (Left_Mezz_ready = '0') and (Right_Mezz_ready = '0') then
            Left_Mezz_read <= '1';
            Right_Mezz_read <= '1';
            state <= state_2a;
        else
            Left_Mezz_read <= '0';
            Right_Mezz_read <= '0';
            state <= state_2;
        end if;

    when state_2a =>
        if (Left_Mezz_ready = '1') and (Right_Mezz_ready = '1') then

    end case;

end elsif;
Matrix_A_Cols <= Left_Mezz_Data_In (31 downto 0);
Matrix_B_Cols <= Right_Mezz_Data_In (31 downto 0);
Left_Mezz_read <= '0';
Right_Mezz_read <= '0';
state <= state_3;
else
  state <= state_2a;
end if;

when state_3 =>
  Matrix_C_Rows <= Matrix_A_Rows;
  Matrix_C_Cols <= Matrix_B_Cols;
  state <= state_3a;

when state_3a =>
  Right_Mezz_Addr <= Matrix_C_Addr_offset;
  Right_Mezz_Data_Out(31 downto 0) <= Matrix_C_Rows;
  if (Right_Mezz_ready = '0') then
    Right_Mezz_write <= '1';
    state <= state_3b;
  else
    Right_Mezz_write <= '0';
    state <= state_3a;
  end if;

when state_3b =>
  if (Right_Mezz_ready = '1') then
    Right_Mezz_addr <= Right_Mezz_Addr + '1';
    Right_Mezz_Data_Out(31 downto 0) <= Matrix_C_Cols;
    Right_Mezz_write <= '0';
    state <= state_3c;
  else
    state <= state_3b;
  end if;

when state_3c =>
  if (Right_Mezz_ready = '0') then
    Right_Mezz_write <= '1';
    state <= state_3d;
  else
    Right_Mezz_write <= '0';
    state <= state_3c;
  end if;

when state_3d =>
  if (Right_Mezz_ready = '1') then
    Right_Mezz_write <= '0';
    state <= state_4;
else
  state <= state_3d;
end if;

when state_4 =>
  if (i < Matrix_A_Rows) then
    state <= state_5;
  else
    i <= (others => '0');
    state <= end_state;
  end if;

when state_5 =>
  if (j < Matrix_B_Cols) then
    state <= state_6;
  else
    i <= i + '1';
    j <= (others => '0');
    state <= state_4;
  end if;
when state_6 =>
  if (k < Matrix_A_Cols) then
    state <= state_9;
  else
    Matrix_C_Addr(20 downto 0) <= (i * Matrix_C_Cols(10 downto 0)) + j;
    state <= state_7;
  end if;

when state_7 =>
  Right_Mezz_Addr <= Matrix_C_Addr + Matrix_C_Addr_offset + addr_offset;
  Right_Mezz_Data_Out(31 downto 0) <= C;
  if (Right_Mezz_ready = '0') then
    Right_Mezz_write <= '1';
    state <= state_8;
  else
    Right_Mezz_write <= '0';
    state <= state_7;
  end if;
when state_8 =>
  if (Right_Mezz_ready = '1') then
    Right_Mezz_write <= '0';
    j <= j + '1';
    k <= (others => '0');
    C <= (others => '0');
when state_9 =>
  Matrix_A_Addr(20 downto 0) <= (i * Matrix_A_Cols(10 downto 0)) + k;
  state <= state_10a;
when state_10a =>
  Matrix_B_Addr(20 downto 0) <= (k * Matrix_B_Cols(10 downto 0)) + j;
  state <= state_10b;
when state_10b =>
  Left_Mezz_Addr <= Matrix_A_Addr + addr_offset;
  Right_Mezz_Addr <= Matrix_B_Addr + addr_offset;
  state <= state_10c;
when state_10c =>
  if (Left_Mezz_ready = '0') and (Right_Mezz_ready = '0') then
    Left_Mezz_read <= '1';
    Right_Mezz_read <= '1';
    state <= state_10d;
  else
    Left_Mezz_read <= '0';
    Right_Mezz_read <= '0';
    state <= state_10c;
  end if;
when state_10d =>
  if (Left_Mezz_ready = '1') and (Right_Mezz_ready = '1') then
    A <= Left_Mezz_Data_In (31 downto 0);
    B <= Right_Mezz_Data_In (31 downto 0);
    Left_Mezz_read <= '0';
    Right_Mezz_read <= '0';
    state <= state_11;
  else
    state <= state_10d;
  end if;
when state_11 =>
  AB <= (A * B);
  state <= state_12;
when state_12 =>
  C <= C + AB(31 downto 0);
k <= k + '1';
state <= state_6;

when end_state =>
    LEDs_Out.Red_n <= '0';
    LEDs_Out.Green_n <= '1';
    Interrupt(0) <= '1';
    state <= end_state;

when others =>
    null;
end case;
end if;
end process;

end WS_Matrix_Multiply_Parallel;
D.3 WS Parallel Processor Matrix Multiplication on PE0

-- Entity : WS_PE0
-- Architecture : WS_Matrix_Multiply_Parallel
-- Filename : pe0_matrix_multiply_parallel.vhd
-- Created by : David Zaretsky (dcz@nwu.edu)
-- Date : 4/23/01
--
-- Library Declarations

-- IEEE Libraries
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;

-- Wildstar System Libraries
library SYSTEM;
use SYSTEM.Xilinx_Package.all;
use SYSTEM.AMS_package.all;

-- Wildstar PE0 Libraries
library PE0_Lib;
use PE0_Lib.PE_Package.all;
use PE0_Lib.PE_Mezz_Mem_package.all;
use PE0_Lib.Mezz_Mem_Mux_pkg.all;
use PE0_Lib.Mezz_Mem32_Mux_pkg.all;
use PE0_Lib.PE_LAD_Mux_pkg.all;
use PE0_Lib.LAD_Mem32_Mux_pkg.all;

-- LAD Mux Libraries
library LAD_Mux_Lib;
use LAD_Mux_Lib.LAD_Mux_pkg.all;

-- Mem Mux Libraries
library Mem32_Mux_Lib;
use Mem32_Mux_Lib.Mem32_Mux_pkg.all;

-- DMA Mux Libraries
library DMA_Mux_Lib;
use DMA_Mux_Lib.DMA_Mux_Pkg.all;
entity WS_PE0 is
  port
  (
    Mclk : in std_logic;
    Kclk : in std_logic;
    PE1_Left_Mezz_Mux : inout Mem32_Mux;
    PE1_Right_Mezz_Mux : inout Mem32_Mux;
    PE2_Left_Mezz_Mux : inout Mem32_Mux;
    PE2_Right_Mezz_Mux : inout Mem32_Mux;
    PE1_Left_Mezz_XBar_Mode : out std_logic_vector(1 downto 0);
    PE1_Right_Mezz_XBar_Mode : out std_logic_vector(1 downto 0);
    PE2_Left_Mezz_XBar_Mode : out std_logic_vector(1 downto 0);
    PE2_Right_Mezz_XBar_Mode : out std_logic_vector(1 downto 0);
    LAD : inout LAD_Mux;
    LEDs_Out : out LED_Std_IF_Out_Type;
    reset : in std_logic
  );
end entity;

architecture WS_Matrix_Multiply_Parallel of WS_PE0 is

constant BASE_MASK : std_logic_vector ( 15 downto 0 ) := x"7FFF";
constant INTERRUPT_BASE : std_logic_vector ( 15 downto 0 ) := x"0300";
constant addr_offset : std_logic_vector ( 5 downto 0 ) := "000010";
constant Matrix_C_Addr_offset : std_logic_vector ( 31 downto 0 ) := x"00020000";

signal Matrix_A_Rows : std_logic_vector (31 downto 0);
signal Matrix_A_Cols : std_logic_vector (31 downto 0);
signal Matrix_A_Addr : std_logic_vector (31 downto 0);
signal Matrix_B_Rows : std_logic_vector (31 downto 0);
signal Matrix_B_Cols : std_logic_vector (31 downto 0);
signal Matrix_B_Addr : std_logic_vector (31 downto 0);
signal Matrix_C_Rows : std_logic_vector (31 downto 0);
signal Matrix_C_Cols : std_logic_vector (31 downto 0);
signal Matrix_C_Addr : std_logic_vector (31 downto 0);

signal A : std_logic_vector (31 downto 0);
signal B : std_logic_vector (31 downto 0);
signal C : std_logic_vector (31 downto 0);
signal AB : std_logic_vector (63 downto 0);
signal i : std_logic_vector (9 downto 0);
signal j                   : std_logic_vector ( 9 downto 0);
signal k                   : std_logic_vector ( 9 downto 0);
signal Interrupt             : std_logic_vector (31 downto 0);
signal PE1_Left_Mezz_Data_In     : std_logic_vector (31 downto 0);
signal PE1_Left_Mezz_Data_Out    : std_logic_vector (31 downto 0);
signal PE1_Left_Mezz_Addr        : std_logic_vector (31 downto 0);
signal PE1_Left_Mezz_ready       : std_logic;
signal PE1_Left_Mezz_read        : std_logic;
signal PE1_Left_Mezz_write       : std_logic;
signal PE1_Right_Mezz_Data_In    : std_logic_vector (31 downto 0);
signal PE1_Right_Mezz_Data_Out   : std_logic_vector (31 downto 0);
signal PE1_Right_Mezz_Addr       : std_logic_vector (31 downto 0);
signal PE1_Right_Mezz_ready      : std_logic;
signal PE1_Right_Mezz_read       : std_logic;
signal PE1_Right_Mezz_write      : std_logic;
signal PE2_Left_Mezz_Data_In     : std_logic_vector (31 downto 0);
signal PE2_Left_Mezz_Data_Out    : std_logic_vector (31 downto 0);
signal PE2_Left_Mezz_Addr        : std_logic_vector (31 downto 0);
signal PE2_Left_Mezz_ready       : std_logic;
signal PE2_Left_Mezz_read        : std_logic;
signal PE2_Left_Mezz_write       : std_logic;
signal PE2_Right_Mezz_Data_In    : std_logic_vector (31 downto 0);
signal PE2_Right_Mezz_Data_Out   : std_logic_vector (31 downto 0);
signal PE2_Right_Mezz_Addr       : std_logic_vector (31 downto 0);
signal PE2_Right_Mezz_ready      : std_logic;
signal PE2_Right_Mezz_read       : std_logic;
signal PE2_Right_Mezz_write      : std_logic;

type state_values is (init_state, state_1, state_2, state_2a,
state_3, state_3a, state_3b, state_3c, state_3d,
state_4, state_5, state_6, state_6a, state_6b, state_7,
state_7a, state_7b, state_8, state_9, state_9a, state_9b,
state_9c, state_9d, state_10, state_10a, state_10b,
state_10c, state_10d, state_10e, state_10f, state_10g,
state_10h, state_11, state_12, state_13, state_14,
state_15, end_state);
signal state                      : state_values;
signal PE1_Left_Mezz_state        : state_values;
signal PE1_Right_Mezz_state       : state_values;
signal PE2_Left_Mezz_state        : state_values;
signal PE2_Right_Mezz_state       : state_values;

begin


-- The P_Interrupt process is used to send an Interrupt signal to the
-- host, via the LAD Bus.

P_Interrupt : process ( reset, Kclk, LAD, Interrupt )
begin
if ( reset = '1' ) then  -- reset
  LAD.Akk <= '0';
  LAD.Data_Out <= ( others => '0' );
elsif ( rising_edge (Kclk) ) then
  if ( LAD.Strobe = '1' ) then
    if ( ( LAD.Addr ( INTERRUPT_BASE'range ) and BASE_MASK ) =
        INTERRUPT_BASE ) then
      LAD.Akk <= '1';
      LAD.Data_Out <= Interrupt;
    end if;
  end if;
elsif ( LAD.Strobe = '0' ) then
  LAD.Akk <= '0';
end if;
end if;
end process;

-- The P_PE1_Left_Mezz_Mem process is for access to Left Mezz Memory.

P_PE1_Left_Mezz_Mem : process ( reset, Mclk, PE1_Left_Mezz_Mux,
P_E1_Left_Mezz_read, PE1_Left_Mezz_write )
begin
if ( reset = '1' ) then  -- Initializations
  PE1_Left_Mezz_Mux.Addr <= (others => '0');
  PE1_Left_Mezz_Mux.Data_Out <= (others => '0');
  PE1_Left_Mezz_Mux.Req <= '0';
  PE1_Left_Mezz_Mux.Write <= '0';
elseif ( LAD.Strobe = '0' ) then
  LAD.Akk <= '0';
end if;
end if;
end process;
PE1_Left_Mezz_Data_In <= (others => '0');
PE1_Left_Mezz_ready <= '0';
PE1_Left_Mezz_state <= init_state;

elsif ( rising_edge (Mclk) ) then

PE1_Left_Mezz_Mux.Write <= '0';
PE1_Left_Mezz_Mux.Req <= '0';

case PE1_Left_Mezz_state is
when init_state =>
    PE1_Left_Mezz_ready <= '0';
    if (PE1_Left_Mezz_read = '1') then
        PE1_Left_Mezz_state <= state_1;
    elsif (PE1_Left_Mezz_write = '1') then
        PE1_Left_Mezz_state <= state_4;
    else
        PE1_Left_Mezz_state <= init_state;
    end if;
when state_1 =>
    PE1_Left_Mezz_Mux.Req <= '1';
    PE1_Left_Mezz_Mux.Addr <= PE1_Left_Mezz_Addr;
    if (PE1_Left_Mezz_Mux.Akk = '1') then
        PE1_Left_Mezz_state <= state_2;
    else
        PE1_Left_Mezz_state <= state_1;
    end if;
when state_2 =>
    PE1_Left_Mezz_Mux.Req <= '1';
    if (PE1_Left_Mezz_Mux.Data_Valid = '1') then
        PE1_Left_Mezz_Data_In <= PE1_Left_Mezz_Mux.Data_In;
        PE1_Left_Mezz_ready <= '1';
        PE1_Left_Mezz_state <= state_3;
    else
        PE1_Left_Mezz_state <= state_2;
    end if;
when state_3 =>
    if (PE1_Left_Mezz_Mux.Data_Valid = '0') then
        PE1_Left_Mezz_ready <= '0';
        PE1_Left_Mezz_state <= init_state;
    else
        PE1_Left_Mezz_state <= state_3;
    end if;
end case;
when state_4 =>
    PE1_Left_Mezz_Mux.Req <= '1';
    PE1_Left_Mezz_Mux.Write <= '1';
    PE1_Left_Mezz_Mux.Addr <= PE1_Left_Mezz_Addr;

    if (PE1_Left_Mezz_Mux.Akk = '1') then
        PE1_Left_Mezz_Mux.Data_Out <= PE1_Left_Mezz_Data_Out;
        PE1_Left_Mezz_ready <= '1';
        PE1_Left_Mezz_state <= state_5;
    else
        PE1_Left_Mezz_state <= state_4;
    end if;

when state_5 =>
    PE1_Left_Mezz_Mux.Req <= '0';
    PE1_Left_Mezz_Mux.Write <= '0';
    PE1_Left_Mezz_ready <= '0';
    PE1_Left_Mezz_state <= init_state;

when others =>
    null;
end case;
end if;
end process;

-------------------------------------------------------------------------------
--
--  The P_PE1_Right_Mezz_Mem process is for access to Right Mezz Mem.
--
-------------------------------------------------------------------------------

P_PE1_Right_Mezz_Mem : process ( reset, Mclk, PE1_Right_Mezz_Mux, 
                                  PE1_Right_Mezz_read, PE1_Right_Mezz_write )
begin
    if ( reset = '1' ) then  -- Initializations
        PE1_Right_Mezz_Mux.Addr <= (others => '0');
        PE1_Right_Mezz_Mux.Data_Out <= (others => '0');
        PE1_Right_Mezz_Mux.Req <= '0';
        PE1_Right_Mezz_Mux.Write <= '0';
        PE1_Right_Mezz_Data_In <= (others => '0');
        PE1_Right_Mezz_read <= '0';
        PE1_Right_Mezz_state <= init_state;
    end if;
end process;
elsif (rising_edge(Mclk)) then

    PE1_Right_Mezz_Mux.Write <= '0';
    PE1_Right_Mezz_Mux.Req  <= '0';

case PE1_Right_Mezz_state is
when init_state =>
    PE1_Right_Mezz_ready <= '0';
    if (PE1_Right_Mezz_read = '1') then
        PE1_Right_Mezz_state <= state_1;
    elsif (PE1_Right_Mezz_write = '1') then
        PE1_Right_Mezz_state <= state_4;
    else
        PE1_Right_Mezz_state <= init_state;
    end if;
when state_1 =>
    PE1_Right_Mezz_Mux.Req        <= '1';
    PE1_Right_Mezz_Mux.Addr       <= PE1_Right_Mezz_Addr;
    if (PE1_Right_Mezz_Mux.Akk = '1') then
        PE1_Right_Mezz_state            <= state_2;
    else
        PE1_Right_Mezz_state            <= state_1;
    end if;
when state_2 =>
    PE1_Right_Mezz_Mux.Req        <= '1';
    if (PE1_Right_Mezz_Mux.Data_Valid = '1') then
        PE1_Right_Mezz_Data_In         <= PE1_Right_Mezz_Mux.Data_In;
        PE1_Right_Mezz_ready            <= '1';
        PE1_Right_Mezz_state            <= state_3;
    else
        PE1_Right_Mezz_state            <= state_2;
    end if;
when state_3 =>
    if (PE1_Right_Mezz_Mux.Data_Valid = '0') then
        PE1_Right_Mezz_ready            <= '0';
        PE1_Right_Mezz_state            <= init_state;
    else
        PE1_Right_Mezz_state            <= state_3;
    end if;
when state_4 =>
PE1_Right_Mezz_Mux.Req  <= '1';
PE1_Right_Mezz_Mux.Write <= '1';
PE1_Right_Mezz_Mux.Addr  <= PE1_Right_Mezz_Addr;
if (PE1_Right_Mezz_Mux.Akk = '1') then
  PE1_Right_Mezz_Mux.Data_Out  <= PE1_Right_Mezz_Data_Out;
  PE1_Right_Mezz_ready        <= '1';
  PE1_Right_Mezz_state        <= state_5;
else
  PE1_Right_Mezz_state        <= state_4;
end if;
when state_5 =>
  PE1_Right_Mezz_Mux.Req  <= '0';
  PE1_Right_Mezz_Mux.Write <= '0';
  PE1_Right_Mezz_ready    <= '0';
  PE1_Right_Mezz_state    <= init_state;
when others =>
  null;
end case;
end if;
end process;

-----------------------------------------------------------------------
-- The P_PE2_Left_Mezz_Mem process is for access to Left Mezz Memory.
-----------------------------------------------------------------------
P_PE2_Left_Mezz_Mem : process ( reset, Mclk, PE2_Left_Mezz_Mux,    
                              PE2_Left_Mezz_read , PE2_Left_Mezz_write )
begin
  if ( reset = '1' ) then -- Initializations
    PE2_Left_Mezz_Mux.Addr          <= (others => '0');
    PE2_Left_Mezz_Mux.Data_Out      <= (others => '0');
    PE2_Left_Mezz_Mux.Req           <= '0';
    PE2_Left_Mezz_Mux.Write         <= '0';
    PE2_Left_Mezz_Data_In           <= (others => '0');
    PE2_Left_Mezz_ready             <= '0';
    PE2_Left_Mezz_state             <= init_state;
  elsif ( rising_edge (Mclk) ) then

  end if;
end process;
case PE2_Left_Mezz_state is
  when init_state =>
    PE2_Left_Mezz_ready <= '0';
    if (PE2_Left_Mezz_read = '1') then
      PE2_Left_Mezz_state <= state_1;
      elsif (PE2_Left_Mezz_write = '1') then
        PE2_Left_Mezz_state <= state_4;
      else
        PE2_Left_Mezz_state <= init_state;
    end if;

  when state_1 =>
    PE2_Left_Mezz_Mux.Req     <= '1';
    PE2_Left_Mezz_Mux.Addr    <= PE2_Left_Mezz_Addr;
    if (PE2_Left_Mezz_Mux.Akk = '1') then
      PE2_Left_Mezz_state       <= state_2;
    else
      PE2_Left_Mezz_state       <= state_1;
    end if;

  when state_2 =>
    PE2_Left_Mezz_Mux.Req     <= '1';
    if (PE2_Left_Mezz_Mux.Data_Valid = '1') then
      PE2_Left_Mezz_Data_In    <= PE2_Left_Mezz_Mux.Data_In;
      PE2_Left_Mezz_ready      <= '1';
      PE2_Left_Mezz_state       <= state_3;
    else
      PE2_Left_Mezz_state       <= state_2;
    end if;

  when state_3 =>
    if (PE2_Left_Mezz_Mux.Data_Valid = '0') then
      PE2_Left_Mezz_ready      <= '0';
      PE2_Left_Mezz_state       <= init_state;
    else
      PE2_Left_Mezz_state       <= state_3;
    end if;

  when state_4 =>
    PE2_Left_Mezz_Mux.Req     <= '1';
    PE2_Left_Mezz_Mux.Write   <= '1';
    PE2_Left_Mezz_Mux.Addr    <= PE2_Left_Mezz_Addr;
if (PE2_Left_Mezz_Mux.Akk = '1') then
    PE2_Left_Mezz_Mux.Data_Out <= PE2_Left_Mezz_Data_Out;
    PE2_Left_Mezz_ready      <= '1';
    PE2_Left_Mezz_state      <= state_5;
else
    PE2_Left_Mezz_state      <= state_4;
end if;

when state_5 =>
    PE2_Left_Mezz_Mux.Req    <= '0';
    PE2_Left_Mezz_Mux.Write  <= '0';
    PE2_Left_Mezz_ready      <= '0';
    PE2_Left_Mezz_state      <= init_state;

when others =>
    null;
end case;
end if;
end process;

-----------------------------------------------------------------------
--
--  The P_PE2_Right_Mezz_Mem process is for access to Right Mezz Mem.
--
-----------------------------------------------------------------------

P_PE2_Right_Mezz_Mem : process ( reset, Mclk, PE2_Right_Mezz_Mux, 
   PE2_Right_Mezz_read, PE2_Right_Mezz_write )
begin
    if ( reset = '1' ) then                    -- Initializations
        PE2_Right_Mezz_Mux.Addr           <= (others => '0');
        PE2_Right_Mezz_Mux.Data_Out      <= (others => '0');
        PE2_Right_Mezz_Mux.Req           <= '0';
        PE2_Right_Mezz_Mux.Write         <= '0';
        PE2_Right_Mezz_Data_In           <= (others => '0');
        PE2_Right_Mezz_ready             <= '0';
        PE2_Right_Mezz_state             <= init_state;
    elsif ( rising_edge (Mclk) ) then
        PE2_Right_Mezz_Mux.Write         <= '0';
        PE2_Right_Mezz_Mux.Req           <= '0';
case PE2_Right_Mezz_state is
  when init_state =>
    PE2_Right_Mezz_ready     <= '0';
    if (PE2_Right_Mezz_read = '1') then
      PE2_Right_Mezz_state <= state_1;
    elsif (PE2_Right_Mezz_write = '1') then
      PE2_Right_Mezz_state <= state_4;
    else
      PE2_Right_Mezz_state <= init_state;
    end if;
  when state_1 =>
    PE2_Right_Mezz_Mux.Req        <= '1';
    PE2_Right_Mezz_Mux.Addr       <= PE2_Right_Mezz_Addr;
    if (PE2_Right_Mezz_Mux.Akk = '1') then
      PE2_Right_Mezz_state <= state_2;
    else
      PE2_Right_Mezz_state <= state_1;
    end if;
  when state_2 =>
    PE2_Right_Mezz_Mux.Req        <= '1';
    if (PE2_Right_Mezz_Mux.Data_Valid = '1') then
      PE2_Right_Mezz_Data_In        <= PE2_Right_Mezz_Mux.Data_In;
      PE2_Right_Mezz_ready          <= '1';
      PE2_Right_Mezz_state          <= state_3;
    else
      PE2_Right_Mezz_state          <= state_2;
    end if;
  when state_3 =>
    if (PE2_Right_Mezz_Mux.Data_Valid = '0') then
      PE2_Right_Mezz_ready          <= '0';
      PE2_Right_Mezz_state          <= init_state;
    else
      PE2_Right_Mezz_state          <= state_3;
    end if;
  when state_4 =>
    PE2_Right_Mezz_Mux.Req        <= '1';
    PE2_Right_Mezz_Mux.Write      <= '1';
    PE2_Right_Mezz_Mux.Addr       <= PE2_Right_Mezz_Addr;
    if (PE2_Right_Mezz_Mux.Akk = '1') then
      PE2_Right_Mezz_Mux.Data_Out    <= PE2_Right_Mezz_Data_Out;
      PE2_Right_Mezz_ready          <= '1';
      PE2_Right_Mezz_state          <= state_5;
else
    PE2_Right_Mezz_state <= state_4;
end if;

when state_5 =>
    PE2_Right_Mezz_Mux.Req <= '0';
    PE2_Right_Mezz_Mux.Write <= '0';
    PE2_Right_Mezz_ready <= '0';
    PE2_Right_Mezz_state <= init_state;

when others =>
    null;
end case;
end if;
end process;

-- The P_Matrix_Multiply process computes C=A*B, where matrix A is
-- stored in the PE1 Left Mezz Memory and matrix B is located in the
-- PE1 Right Mezz Memory. The output, matrix C, is written to the PE1
-- Right Mezz Memory.
-----------------------------------------------------------------------

P_Matrix_Multiply : process ( reset, Mclk, LAD,
    PE1_Left_Mezz_ready, PE1_Right_Mezz_ready,
    PE2_Left_Mezz_ready, PE2_Right_Mezz_ready )
begin
    if ( reset = '1' ) then
        -- Initializations
        LEDs_Out.Red_n <= '1';
        LEDs_Out.Green_n <= '0';
        Matrix_A_Rows <= (others => '0');
        Matrix_A_Cols <= (others => '0');
        Matrix_A_Addr <= (others => '0');
        Matrix_B_Rows <= (others => '0');
        Matrix_B_Cols <= (others => '0');
        Matrix_B_Addr <= (others => '0');
        Matrix_C_Rows <= (others => '0');
        Matrix_C_Cols <= (others => '0');
        Matrix_C_Addr <= (others => '0');
A <= (others => '0');
B <= (others => '0');
C <= (others => '0');
AB <= (others => '0');
i <= (others => '0');
j <= (others => '0');
k <= (others => '0');

PE2_Left_Mezz_Data_Out <= (others => '0');
PE2_Left_Mezz_Addr <= (others => '0');
PE2_Left_Mezz_read <= '0';
PE2_Left_Mezz_write <= '0';
PE2_Right_Mezz_Data_Out <= (others => '0');
PE2_Right_Mezz_Addr <= (others => '0');
PE2_Right_Mezz_read <= '0';
PE2_Right_Mezz_write <= '0';

PE1_Left_Mezz_Data_Out <= (others => '0');
PE1_Left_Mezz_Addr <= (others => '0');
PE1_Left_Mezz_read <= '0';
PE1_Left_Mezz_write <= '0';
PE1_Right_Mezz_Data_Out <= (others => '0');
PE1_Right_Mezz_Addr <= (others => '0');
PE1_Right_Mezz_read <= '0';
PE1_Right_Mezz_write <= '0';

PE1_Left_Mezz_XBar_Mode <= "00";   -- PE1 -> MEM0, PE0 -> MEM1
PE1_Right_Mezz_XBar_Mode <= "00";   -- PE1 -> MEM0, PE0 -> MEM1
PE2_Left_Mezz_XBar_Mode <= "00";   -- PE2 -> MEM0, PE0 -> MEM1
PE2_Right_Mezz_XBar_Mode <= "00";   -- PE2 -> MEM0, PE0 -> MEM1

Interrupt <= (others => '0');
state <= init_state;

elsif ( rising_edge (Mclk) ) then

case state is
when init_state =>
    Matrix_C_Addr <= Matrix_C_Addr_offset;
    if (PE1_Left_Mezz_ready='0') and (PE1_Right_Mezz_ready='0') then
        PE1_Left_Mezz_read <= '1';
        PE1_Right_Mezz_read <= '1';
        state <= state_1;
    else
        PE1_Left_Mezz_read <= '0';
        PE1_Right_Mezz_read <= '0';
        state <= init_state;
    end if;
else
    Interrupt <= (others => '0');
end case;

state <= init_state;
when state_1 =>
  if (PE1_Left_Mezz_ready='1') and (PE1_Right_Mezz_ready='1') then
    Matrix_A_Rows <= PE1_Left_Mezz_Data_In;
    Matrix_B_Rows <= PE1_Right_Mezz_Data_In;
    PE1_Left_Mezz_Addr <= PE1_Left_Mezz_Addr + '1';
    PE1_Right_Mezz_Addr <= PE1_Right_Mezz_Addr + '1';
    PE1_Left_Mezz_read <= '0';
    PE1_Right_Mezz_read <= '0';
    state <= state_2;
  else
    state <= state_1;
  end if;
when state_2 =>
  if (PE1_Left_Mezz_ready='0') and (PE1_Right_Mezz_ready='0') then
    PE1_Left_Mezz_read <= '1';
    PE1_Right_Mezz_read <= '1';
    state <= state_2a;
  else
    PE1_Left_Mezz_read <= '0';
    PE1_Right_Mezz_read <= '0';
    state <= state_2;
  end if;
when state_2a =>
  if (PE1_Left_Mezz_ready='1') and (PE1_Right_Mezz_ready='1') then
    Matrix_A_Cols <= PE1_Left_Mezz_Data_In;
    Matrix_B_Cols <= PE1_Right_Mezz_Data_In;
    PE1_Left_Mezz_read <= '0';
    PE1_Right_Mezz_read <= '0';
    state <= state_3;
  else
    state <= state_2a;
  end if;
when state_3 =>
  Matrix_C_Rows <= Matrix_A_Rows;
  Matrix_C_Cols <= Matrix_B_Cols;
  state <= state_3a;
when state_3a =>
  PE1_Left_Mezz_Addr <= Matrix_C_Addr_offset;
  PE1_Left_Mezz_Data_Out <= Matrix_C_Rows;
  if (PE1_Left_Mezz_ready = '0') then
    PE1_Left_Mezz_write <= '1';
    state <= state_3b;
else
    PE1_Left_Mezz_write <= '0';
    state <= state_3a;
end if;

when state_3b =>
    if (PE1_Left_Mezz_ready = '1') then
        PE1_Left_Mezz_Addr <= PE1_Left_Mezz_Addr + '1';
        PE1_Left_Mezz_Data_Out <= Matrix_C_Cols;
        PE1_Left_Mezz_write <= '0';
        state <= state_3c;
    else
        state <= state_3b;
    end if;

when state_3c =>
    if (PE1_Left_Mezz_ready = '0') then
        PE1_Left_Mezz_write <= '1';
        state <= state_3d;
    else
        PE1_Left_Mezz_write <= '0';
        state <= state_3c;
    end if;

when state_3d =>
    if (PE1_Left_Mezz_ready = '1') then
        PE1_Left_Mezz_write <= '0';
        state <= state_4;
    else
        state <= state_3d;
    end if;

when state_4 =>
    if (i < Matrix_A_Rows) then
        state <= state_5;
    else
        i <= (others => '0');
        state <= end_state;
    end if;

when state_5 =>
    if (j < Matrix_B_Cols) then
        state <= state_6;
    else
        i <= i + '1';
        j <= (others => '0');
        state <= state_4;
    end if;
when state_6 =>
  if (k < Matrix_A_Cols) then
    state <= state_9;
  else
    Matrix_C_Addr(20 downto 0) <= (i * Matrix_C_Cols(10 downto 0)) + j;
    state <= state_7;
  end if;
when state_7 =>
  PE1_Left_Mezz_Addr       <= Matrix_C_Addr + Matrix_C_Addr_offset + addr_offset;
  PE1_Left_Mezz_Data_Out   <= C;
  if (PE1_Left_Mezz_ready = '0') then
    PE1_Left_Mezz_write      <= '1';
    state                   <= state_8;
  else
    PE1_Left_Mezz_write      <= '0';
    state                   <= state_7;
  end if;
when state_8 =>
  if (PE1_Left_Mezz_ready = '1') then
    PE1_Left_Mezz_write       <= '0';
    j                         <= j + '1';
    k                         <= (others => '0');
    C                         <= (others => '0');
    state                     <= state_5;
  else
    state                     <= state_8;
  end if;
when state_9 =>
  Matrix_A_Addr(20 downto 0) <= (i * Matrix_A_Cols(10 downto 0)) + k;
  state                      <= state_10a;
when state_10a =>
  Matrix_B_Addr(20 downto 0) <= (k * Matrix_B_Cols(10 downto 0)) + j;
  state                      <= state_10b;
when state_10b =>
  PE1_Left_Mezz_Addr        <= Matrix_A_Addr + addr_offset;
  PE1_Right_Mezz_Addr       <= Matrix_B_Addr + addr_offset;
  state                     <= state_10c;
when state_10c =>
    if (PE1_Left_Mezz_ready='0') and (PE1_Right_Mezz_ready='0') then
        PE1_Left_Mezz_read <= '1';
        PE1_Right_Mezz_read <= '1';
        state <= state_10d;
    else
        PE1_Left_Mezz_read <= '0';
        PE1_Right_Mezz_read <= '0';
        state <= state_10c;
    end if;
when state_10d =>
    if (PE1_Left_Mezz_ready='1') and (PE1_Right_Mezz_ready='1') then
        A <= PE1_Left_Mezz_Data_In;
        B <= PE1_Right_Mezz_Data_In;
        PE1_Left_Mezz_read <= '0';
        PE1_Right_Mezz_read <= '0';
        state <= state_11;
    else
        state <= state_10d;
    end if;
when state_11 =>
    AB <= (A * B);
    state <= state_12;
when state_12 =>
    C <= C + AB(31 downto 0);
    k <= k + '1';
    state <= state_6;
when end_state =>
    LEDs_Out.Red_n <= '0';
    LEDs_Out.Green_n <= '1';
    Interrupt(0) <= '1';
    state <= end_state;
when others =>
    null;
end case;
end if;
end process;
end WS_Matrix_Multiply_Parallel;
D.4 WS Host Code for Parallel Matrix Multiplication

/***********************************************
*    Annapolis Micro Systems Inc.   *
*    190 Admiral Cochrane   *
*    Suite 130   *
*    Annapolis, MD  21401   *
***********************************************/

/***********************************************
*    File        :  ws_matrix_multiply_parallel.c  *
*    Project     :  WILDSTAR(tm) Memory Example   *
*    Copyright   :  Annapolis Micro Systems Inc., 2000   *
***********************************************

/***********************************************
*    Includes                                     *
***********************************************

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <ctype.h>
#include <sys/times.h>
#if defined(WSCPU_MVME2604) || defined(WSCPU_I960)
#include "tasklib.h"
#include "sysLib.h"
#endif
#include "includes/ws.h"
#include "includes/ws_shared.h"
#include "includes/lad_mem_bridge.h"

/***********************************************
*    Defines                                     *
***********************************************

#if defined (WS_VME)
#define DEFAULT_SLOT  (0xF)
#else
#define DEFAULT_SLOT  (0)
#endif
#if !defined WIN32

185

185

185
#define MAX_PATH                  (255) 
#endif
#define MAX_ERR_COUNT             (32) 
#define BITS_IN_DWORD             (32) 
#define MAX_PE_MASK               (0x7) 

/* Register definitions for Memory Access */
#define PEx_LEFT_LOCAL_MEM        (0x1000) 
#define PEx_RIGHT_LOCAL_MEM       (0x1200) 
#define PEx_LEFT_MEZZ_MEM         (0x1400) 
#define PEx_RIGHT_MEZZ_MEM        (0x1800) 

#define PEO_LEFT_MEZZ1            (0x1000) 
#define PEO_RIGHT_MEZZ1           (0x1200) 
#define PEO_LEFT_MEZZ2            (0x1400) 
#define PEO_RIGHT_MEZZ2           (0x1600) 

/* Other Registers in PE's Register Space */
#define RESET_REG_OFFSET          (0x7F00)   /* Reset register */ 
#define INTERRUPT_REG_OFFSET      (0x0300)   /* Interrupt Register */ 

/* Define Crossbar Modes */
#define XBAR_MODE_0             (0x004) /* Both Xbar's Enabled mode 00 */
#define XBAR_MODE_1             (0x005) /* Both Xbar's Enabled mode 01 */
#define XBAR_MODE_X             (0x000) /* Both Xbar's Disabled */

#define WS_32_BIT_PORT            ( 0x1 )
#define WS_64_BIT_PORT            ( 0x0 )

/* Address/Data Test */
#define CK_ADDRESS                (0)
#define CK_DATA                   (1)

/* Default test settings */
#define DEFAULT_ITERATIONS        (1)
#define DEFAULY_FREQUENCY         (50.0)
#define PE_DEFAULT_MASK           (7)
#define MAX_OUTPUT                (256)

/***********************************************************************
*                              Prototypes                             *
***********************************************************************/

static WS_RetCode
WS_Matrix_Multiply ( DWORD    WS_SlotNumber,
DWORD    dPeNum,
float    clk_freq,
DWORD    PeMask,
DWORD    dBusFreq,
BOOLEAN  bIs32Bit );

static WS_RetCode
VerifyData( DWORD ref[],
    DWORD test[],
    DWORD size,
    char *errstr );

WS_RetCode
Mem_Startup( DWORD                   WS_SlotNumber,
WS_PhysicalBoardConfig *WildstarConfiguration,
DWORD                   *rPeMask,
float                   clk_freq,
DWORD                   dBusFreq );

WS_RetCode
Mem_Shutdown( DWORD                   WS_SlotNumber,
WS_PhysicalBoardConfig *WildstarConfiguration,
DWORD                   PeMask );

WS_RetCode
ProgramPE( DWORD                      WS_SlotNumber,
DWORD                      PeNum,
char                       *filename);

/*******************************************************************************/
/*                                   Main                              */
*******************************************************************************/
struct tms ustart, uend;
double config_time, prepare_time, load_time, read_time, compute_time;
DWORD iterations = DEFAULT_ITERATIONS;    /* # iterations to run */;

WS_RetCode
main( int argc, char *argv[] )
{
    WS_RetCode
rc = WS_SUCCESS;

    int
argi;

    float
    clk_freq = DEFAULT_FREQUENCY;

    DWORD
    PeMask=PE_DEFAULT_MASK;    /* Mask of PEs to be tested */

    DWORD
    WS_SlotNumber = DEFAULT_SLOT;  /* Default Slot number for the
Wildstar WS_SlotNumber */

    BOOLEAN
    bBoard = FALSE;
DWORD
dBusFreq = 33;

const char * help_string =
"Usage: ws_vectorsum <list of options>\n"
" Options:\n"
"   -f <frequency>   Set the clock Freqency (default = 50 MHz )\n"
"   -h                Show this help.\n"
"   -l <local bus freq>  Set the Wildstar local bus frequency: \n"
"       33 - for 33 MHz bus frequency\n"
"       66 - for 66 MHz bus frequency\n"
"   -i <iterations>   Number of iterations for each test.\n" (default = 1)\n"
"   -m <PE mask>     Mask of PEs to test. ( PE0= 0x1, PE1= 0x2, PE2= 0x4 )\n"
"   -s or -b <slot ID>   Set the slot number. Default slot number is:\n"
"       0xF (15) for VME boards\n"
"       0 for PCI boards\n"
" Precede all hex entries with 0x... (Ex: 0xF not F)\n";

/***************************************************************
**                                                           **
**        Parse command line parameters                      **
**                                                           **
/***************************************************************
for ( argi = 1; argi < argc; argi++ )
{
    if ( argv[ argi ][ 0 ] == '-' )
    {
        switch ( toupper( argv[ argi ][ 1 ] ) )
        {
            case 'S': case 'B':  /* Set the slot number */
                /* Check for multiple entries */
                if ( bBoard == TRUE )
                {
                    printf ( "Error: Multiple attempts to assign slot/board number!!!\n"");
                    printf ( "%s\n", help_string );
                    return(rc);
                    break;
                }
                bBoard = TRUE;
            }
        }
    /* Do the actual parsing */
    argi++;
    if ( argi < argc )
    {
        WS_SlotNumber = strtoul( argv[ argi ], NULL, 0 );
        printf ( "Setting Slot Number to %x\n", WS_SlotNumber );
    }
} else {
    printf ( " Warning: Invalid Slot Number!\n" );
    return(rc );
}

if ( (WS_SlotNumber < 0) || (WS_SlotNumber > WS_MAX_BOARDS) ) {
    printf ( " Warning: Invalid Slot Number!\n" );
    return(rc );
}

break;

case 'L': /* Set Wildstar local bus frequency */
    /* Do the actual parseing */
    argi++;
    if (argi < argc) {
        dBUsFreq = strtol( argv [ argi ], NULL, 10 );
        printf ("Setting local bus frequency to %u\n", dBUsFreq );
    } else {
        printf ( " Warning: Invalid local bus frequency!\n" );
        return(rc );
    }

if ( (dBUsFreq != 33) && (dBUsFreq != 66) ) {
    printf ( " Warning: Invalid local bus frequency!\n" );
    return(rc );
}

break;

    case 'F': /* Set the clock frequency */
    argi++;
    if (argi < argc) {
        clk_freq = (float) atof ( argv [ argi ] );
        printf ( "Setting the Mclk to [%2.1f] MHz.\n", clk_freq );
    } else {
        printf ( "Warning: Invalid Clock Frequency (Max Value is [%2.1f])!\n", (float)WS_MAX_MCLK_FREQ );
        printf ( "                                             Min Value is [%2.1f])!\n", (float)WS_MIN_MCLK_FREQ );
        return(rc );
    }

    if ( (clk_freq < WS_MIN_MCLK_FREQ) || (clk_freq > WS_MAX_MCLK_FREQ) ) {
        printf ( "Warning: Invalid Clock Frequency (Max Value is [%2.1f])!\n", (float)WS_MAX_MCLK_FREQ );
        }
printf ( "Min Value is\n", (float)WS_MIN_MCLK_FREQ);
    return(rc);
}
break;

case 'H': /* Print the help message */
    printf ( "%s\n\n", help_string );
    return(rc);
    break;

case 'I': /* Set the number of iterations */
    argi++;
    if (argi < argc)
    {
        iterations = atoi( argv [ argi ] );
        printf ("Setting number of iterations to %u\n", iterations );
    }
    else
    {
        printf("Warning: Invalid iteration value!\n");
        return(rc);
    }
    break;

case 'M': /* Set the PE mask */
    argi++;
    if (argi < argc)
    {
        PeMask = strtoul( argv [ argi ], NULL, 0 );
        printf ( "Setting the PE Mask to %d.\n", PeMask);
    }
    else
    {
        PeMask = PE_DEFAULT_MASK;
        printf ( "Warning: PE Mask ( Setting PEMask to %X )\n", PeMask );
    }
    if (PeMask > MAX_PE_MASK)
    {
        printf ( "Warning: PE Mask (%X) invalid.", PeMask );
        PeMask = PE_DEFAULT_MASK;
        printf ( " Setting PE Mask to %X.\n", PeMask );
    }
    break;

default:
    printf ( "Unknown option: \"%s\"\n", argv [ argi ] );
    printf ( "%s\n\n", help_string );
    return(-1);
}
/* ******************************************************
** **
** ** Open the board for testing **
** **
******************************************************/
if ( dBusFreq == 33 )
{
    rc = WS_Open( WS_SlotNumber, WS_SHARED_FLAG | WS_33MHZ_LADBUS_FLAG );
    DISPLAY_OPEN_ERROR(rc);
}
else if ( dBusFreq == 66 )
{
    rc = WS_Open( WS_SlotNumber, WS_SHARED_FLAG | WS_66MHZ_LADBUS_FLAG );
    DISPLAY_OPEN_ERROR(rc);
}
else
{
    printf ( " Warning:  Invalid local bus frequency!\n" );
    return(-1);
}

/****************** ******************************************
** **
** ** Display the Test Name on the LED display. **
** **
******************************************************/
rc = WS_DisplayIntensity( WS_SlotNumber, Level_2 );
DISPLAY_ERROR(rc);
rc = WS_UpdateDisplay( WS_SlotNumber, "MULT" );
DISPLAY_ERROR(rc);

/****************** ******************************************
** **
** ** Run Matrix Multiply Function **
** **
******************************************************/
rc = WS_Matrix_Multiply( WS_SlotNumber, 1, clk_freq, PeMask, dBusFreq,
WS_32_BIT_PORT);
DISPLAY_ERROR(rc);

/****************** ******************************************
** **
** ** Remove the Test Name on the LED display. **
** **
******************************************************/
rc = WS_UpdateDisplay( WS_SlotNumber, "    ");
DISPLAY_ERROR(rc);
Close the board

```c
rc = WS_Close( WS_SlotNumber );
DISPLAY_ERROR(rc);

return(rc);
}
```

---

Function:  WS_Matrix_Multiply

Description: Computes the multiplication of two Matrices, \( C = A \times B \)

Arguments:
- `WS_SlotNumber`: Slot or board number to access
- `clk_freq`: frequency with which to program the clock

Returns:
- `WS_SUCCESS` upon successful completion

```c
WS_RetCode
WS_Matrix_Multiply ( DWORD    WS_SlotNumber,
                     DWORD    dPeNum,
                     float    clk_freq,
                     DWORD    PeMask,
                     DWORD    dBusFreq,
                     BOOLEAN  bIs32Bit )
{
    int i, j, k, l, m, n, p, t, w, x, y, z;

    DWORD
      *Matrix_A_Buffer_64bit,
      *Matrix_B_Buffer_64bit,
      *Matrix_C_Buffer_64bit,
      *Matrix_A_Buffer_32bit,
      *Matrix_B_Buffer_32bit,
      *Matrix_C_Buffer_32bit,
      *A_ReadBuffer,
      *B_ReadBuffer,

    Matrix_A_Offset = 0x0;
    Matrix_B_Offset = 0x0;
    Matrix_C_Offset = 0x020000;
```
Matrix_A_NumRows,
Matrix_A_NumCols,
Matrix_A_NumDwords,

Matrix_B_NumRows,
Matrix_B_NumCols,
Matrix_B_NumDwords,

Matrix_C_NumRows,
Matrix_C_NumCols,
Matrix_C_NumDwords,

Matrix_NumDwords,
pData = 0x0,
Reset0 = 0x01,
Reset1 = 0x01,
PEX_Interrupt = 0,
PE0_Interrupt = 0;

WS_Mem_Object
 *PE1_Left_Mem_obj,
 *PE1_Right_Mem_obj,
 *PE1_Left_Mezz_obj,
 *PE1_Right_Mezz_obj,
 *PE0_Left_Mezz1_obj,
 *PE0_Right_Mezz1_obj,
 *PE0_Left_Mezz2_obj,
 *PE0_Right_Mezz2_obj;

FILE
 *Matrix_A_File,
 *Matrix_B_File,
 *Matrix_C_File;

WS_RetCode
 rc = WS_SUCCESS;

static WS_PhysicalBoardConfig
 WildstarConfiguration;

YSQL_LOAD_MATRIX_A_B

 Matrix_A_File = fopen ("Matrix_A.dat", "r");
 fscanf (Matrix_A_File, "%d\n", &Matrix_A_NumRows);
 fscanf (Matrix_A_File, "%d\n", &Matrix_A_NumCols);

Matrix_A_NumRows = Matrix_A_NumRows / 2;  // split matrix A into two
Matrix_A_NumDwords = Matrix_A_NumRows * Matrix_A_NumCols;

Matrix_B_File = fopen("Matrix_B.dat", "r");
fscanf(Matrix_B_File, "%d\n", &Matrix_B_NumRows);
fscanf(Matrix_B_File, "%d\n", &Matrix_B_NumCols);
Matrix_B_NumDwords = Matrix_B_NumRows * Matrix_B_NumCols;

Matrix_C_NumRows = Matrix_A_NumRows;
Matrix_C_NumCols = Matrix_B_NumCols;
Matrix_C_NumDwords = Matrix_C_NumRows * Matrix_C_NumCols;

/***************************************************************
**                                                           **
**                   Allocate buffers                        **
**                                                           **
/***************************************************************

A_ReadBuffer = malloc(sizeof(DWORD)*(Matrix_A_NumDwords+2)*2);
if ( A_ReadBuffer == NULL )
{
    rc = ERROR_MEMORY_ALLOC;
    CHECK_RC(rc);
}

B_ReadBuffer = malloc(sizeof(DWORD)*(Matrix_B_NumDwords+2)*2);
if ( B_ReadBuffer == NULL )
{
    rc = ERROR_MEMORY_ALLOC;
    CHECK_RC(rc);
}

Matrix_A_Buffer_64bit = malloc(sizeof(DWORD)*(Matrix_A_NumDwords+2)*2);
if ( Matrix_A_Buffer_64bit == NULL )
{
    rc = ERROR_MEMORY_ALLOC;
    CHECK_RC(rc);
}

Matrix_B_Buffer_64bit = malloc(sizeof(DWORD)*(Matrix_B_NumDwords+2)*2);
if ( Matrix_B_Buffer_64bit == NULL )
{
    rc = ERROR_MEMORY_ALLOC;
    CHECK_RC(rc);
}

Matrix_C_Buffer_64bit = malloc(sizeof(DWORD)*(Matrix_C_NumDwords+2)*2);
if ( Matrix_C_Buffer_64bit == NULL )
{
    rc = ERROR_MEMORY_ALLOC;
    CHECK_RC(rc);
}
Matrix_A_Buffer_32bit = malloc(sizeof(DWORD)*(Matrix_A_NumDwords+2));
if ( Matrix_A_Buffer_64bit == NULL )
{
    rc = ERROR_MEMORY_ALLOC;
    CHECK_RC(rc);
}

Matrix_B_Buffer_32bit = malloc(sizeof(DWORD)*(Matrix_B_NumDwords+2));
if ( Matrix_B_Buffer_64bit == NULL )
{
    rc = ERROR_MEMORY_ALLOC;
    CHECK_RC(rc);
}

Matrix_C_Buffer_32bit = malloc(sizeof(DWORD)*(Matrix_C_NumDwords+2));
if ( Matrix_C_Buffer_64bit == NULL )
{
    rc = ERROR_MEMORY_ALLOC;
    CHECK_RC(rc);
}

/***************************************************************
**                                                           **
**                       Fill buffers                        **
**                                                           **
/***************************************************************

printf ("-nCreating MATRIX A buffer for PE0 Left Mezz1 Memory
(Matrix_A.dat): \n");

Matrix_A_Buffer_32bit[0] = Matrix_A_NumRows;
Matrix_A_Buffer_32bit[1] = Matrix_A_NumCols;

printf ("Number of Rows: %d\n", Matrix_A_Buffer_32bit[0]);
printf ("Number of Cols: %d\n\t\t", Matrix_A_Buffer_32bit[1]);

for (w=2; w<(Matrix_A_NumDwords+2); w++) {
    fscanf (Matrix_A_File, "%d\n", &Matrix_A_Buffer_32bit[w]);
    if (Matrix_A_NumDwords < MAX_OUTPUT) {
        printf ("\t%d", Matrix_A_Buffer_32bit[w]);
        if ( (w-1) % Matrix_A_NumCols == 0 )
            printf ("\n\t\t");
    }
}

printf ("-nCreating MATRIX B buffer for PE0 Right Mezz1 Memory
(Matrix_B.dat): \n");

Matrix_B_Buffer_32bit[0] = Matrix_B_NumRows;
Matrix_B_Buffer_32bit[1] = Matrix_B_NumCols;

printf ("Number of Rows: \%d\n", Matrix_B_Buffer_32bit[0]);
printf ("Number of Cols: \%d\t\t", Matrix_B_Buffer_32bit[1]);

for (x=2; x<(Matrix_B_NumDwords+2); x++) {
    fscanf (Matrix_B_File, "\%d\n", &Matrix_B_Buffer_32bit[x]);
    
    if (Matrix_B_NumDwords < MAX_OUTPUT) {
        printf ("\t\%d", Matrix_B_Buffer_32bit[x]);
        if ( (x-1) % Matrix_B_NumCols == 0 )
            printf ("\n\t\t");
    }
}

printf ("\n\nCreating MATRIX A buffer for PE1 Left Mezz Memory (Matrix_A.dat): \n");
Matrix_A_Buffer_64bit[0] = Matrix_A_NumRows;
Matrix_A_Buffer_64bit[1] = 0;
Matrix_A_Buffer_64bit[2] = Matrix_A_NumCols;
Matrix_A_Buffer_64bit[3] = 0;

printf ("Number of Rows: \%d\n", Matrix_A_Buffer_64bit[0]);
printf ("Number of Cols: \%d\t\t", Matrix_A_Buffer_64bit[2]);

for (i=4; i<(Matrix_A_NumDwords+2)*2; i=i+2) {
    fscanf (Matrix_A_File, "\%d\n", &Matrix_A_Buffer_64bit[i]);
    //Matrix_A_Buffer_64bit[i] = Matrix_A_Buffer_32bit[i/2];
    Matrix_A_Buffer_64bit[i+1] = 0;
    
    if (Matrix_A_NumDwords < MAX_OUTPUT) {
        printf ("\t\%d", Matrix_A_Buffer_64bit[i]);
        if ( ((i-1)/2) % Matrix_A_NumCols == 0 )
            printf ("\n\t\t");
    }
}

printf ("\n\nCreating MATRIX B buffer for PE1 Right Mezz Memory (Matrix_B.dat): \n");
Matrix_B_Buffer_64bit[0] = Matrix_B_NumRows;
Matrix_B_Buffer_64bit[1] = 0;
Matrix_B_Buffer_64bit[2] = Matrix_B_NumCols;
Matrix_B_Buffer_64bit[3] = 0;

printf ("Number of Rows: \%d\n", Matrix_B_Buffer_64bit[0]);
printf ("Number of Cols: \%d\n\t\t", Matrix_B_Buffer_64bit[2]);
for (j=4; j<(Matrix_B_NumDwords+2)*2; j=j+2) {
    // fscanf (Matrix_B_File, "%d\n", &Matrix_B_Buffer_64bit[j]);
    Matrix_B_Buffer_64bit[j] = Matrix_B_Buffer_32bit[j/2];
    Matrix_B_Buffer_64bit[j+1] = 0;
    if (Matrix_B_NumDwords < MAX_OUTPUT) {
        printf ("\t%d", Matrix_B_Buffer_64bit[j]);
        if ( ((j-1)/2) % Matrix_B_NumCols == 0 )
            printf ("\n\t\t");
    }
}

fclose(Matrix_A_File);
fclose(Matrix_B_File);

// end prepare_time
times(&uend);
prepare_time = ( (float)(uend.tms_utime + uend.tms_stime
            - ustart.tms_utime - ustart.tms_stime) / (float)CLK_TCK
        );

// start config_time
times(&ustart);

/******************************************************************************
*                                                                            *
*              Get board configuration information                           *
*                                                                            *
*******************************************************************************/
rc = WS_GetPhysicalConfig( WS_SlotNumber, &WildstarConfiguration );
CHECK_RC(rc);

/******************************************************************************
*                                                                            *
*              Display configuration information                              *
*                                                                            *
*******************************************************************************/
rc = DisplayConfiguration( WS_SlotNumber, &WildstarConfiguration );
CHECK_RC(rc);

printf ( "\n\n******************************************************");
printf ( "\tWILDSTAR(tm) FUNCTION \n");
printf ( "**************************************************************************");

/******************************************************************************
*                                                                            *
*              Setup clocks and program pe(s)                                *
*                                                                            *
*******************************************************************************/
rc = Mem_Startup( WS_SlotNumber, &WildstarConfiguration, &PeMask, 
clk_freq, dBusFreq);
CHECK_RC(rc);

printf ( "\nExample setup successful!\n\n" );

/*********************************************************
**                                                     **
**           Create the Memory Access for PE1          **
**                                                     **
*********************************************************/

printf ("\n\nCreating PE1 Left Local Memory Object... ");
PE1_Left_Mem_obj = WS_Mem_Create(WS_SlotNumber, WS_PE1, 
PEx_LEFT_LOCAL_MEM, WS_32_BIT_PORT);
if( PE1_Left_Mem_obj==NULL )
{
    rc = WS_ERR_MEM_ALLOC;
    return(rc);
}
printf ("Done.\n");

printf ("Creating PE1 Right Local Memory Object... ");
PE1_Right_Mem_obj = WS_Mem_Create(WS_SlotNumber, WS_PE1, 
PEx_RIGHT_LOCAL_MEM, WS_32_BIT_PORT);
if( PE1_Right_Mem_obj==NULL )
{
    rc = WS_ERR_MEM_ALLOC;
    return(rc);
}
printf ("Done.\n");

printf ("Creating PE1 Left Mezz Memory Object... ");
PE1_Left_Mezz_obj = WS_Mem_Create(WS_SlotNumber, WS_PE1, 
PEx_LEFT_MEZZ_MEM, WS_64_BIT_PORT);
if( PE1_Left_Mezz_obj==NULL )
{
    rc = WS_ERR_MEM_ALLOC;
    return(rc);
}
printf ("Done.\n");

printf ("Creating PE1 Right Mezz Memory Object... ");
PE1_Right_Mezz_obj = WS_Mem_Create(WS_SlotNumber, WS_PE1, 
PEx_RIGHT_MEZZ_MEM, WS_64_BIT_PORT);
if( PE1_Right_Mezz_obj==NULL )
{
    rc = WS_ERR_MEM_ALLOC;
return(rc);
}
printf ("Done.\n");

/*********************************************************
**                                                     **
**           Create the Memory Access for PE0           **
**                                                     **
/*********************************************************/

printf ("Creating PE0->PE1 Left Mezz Memory Object... ");
PE0_Left_Mezz1_obj = WS_Mem_Create(WS_SlotNumber, WS_PE0,
PE0_LEFT_MEZZ1, WS_32_BIT_PORT);
if( PE0_Left_Mezz1_obj==NULL )
{
    rc = WS_ERR_MEM_ALLOC;
    return(rc);
}
printf ("Done.\n");

printf ("Creating PE0->PE1 Right Mezz Memory Object... ");
PE0_Right_Mezz1_obj = WS_Mem_Create(WS_SlotNumber, WS_PE0,
PE0_RIGHT_MEZZ1, WS_32_BIT_PORT);
if( PE0_Right_Mezz1_obj==NULL )
{
    rc = WS_ERR_MEM_ALLOC;
    return(rc);
}
printf ("Done.\n");

printf ("Creating PE0->PE2 Left Mezz Memory Object... ");
PE0_Left_Mezz2_obj = WS_Mem_Create(WS_SlotNumber, WS_PE0,
PE0_LEFT_MEZZ2, WS_32_BIT_PORT);
if( PE0_Left_Mezz2_obj==NULL )
{
    rc = WS_ERR_MEM_ALLOC;
    return(rc);
}
printf ("Done.\n");

printf ("Creating PE0->PE2 Right Mezz Memory Object... ");
PE0_Right_Mezz2_obj = WS_Mem_Create(WS_SlotNumber, WS_PE0,
PE0_RIGHT_MEZZ2, WS_32_BIT_PORT);
if( PE0_Right_Mezz2_obj==NULL )
{
    rc = WS_ERR_MEM_ALLOC;
    return(rc);
}
printf ("Done.\n");
/*********************************************************
**                                                     **
**        Clear PE1 Memories                           **
**                                                     **
*********************************************************/

printf ("\n\nClearing PE1 Left Local Memory... ");
WS_Mem_Set(PE1_Left_Mem_obj, 0, 0, 262144);
if( PE1_Left_Mem_obj->rc!=WS_SUCCESS )
{
    return(PE1_Left_Mem_obj->rc);
}
printf ("Done.\n");

printf ("Clearing PE1 Right Local Memory... ");
WS_Mem_Set(PE1_Right_Mem_obj, 0, 0, 262144);
if( PE1_Right_Mem_obj->rc!=WS_SUCCESS )
{
    return(PE1_Right_Mem_obj->rc);
}
printf ("Done.\n");

printf ("Clearing PE1 Left Mezz Memory... ");
WS_Mem_Set(PE1_Left_Mezz_obj, 0, 0, 262144);
if( PE1_Left_Mezz_obj->rc!=WS_SUCCESS )
{
    return(PE1_Left_Mezz_obj->rc);
}
printf ("Done.\n");

printf ("Clearing PE1 Right Mezz Memory... ");
WS_Mem_Set(PE1_Right_Mezz_obj, 0, 0, 262144);
if( PE1_Right_Mezz_obj->rc!=WS_SUCCESS )
{
    return(PE1_Right_Mezz_obj->rc);
}
printf ("Done.\n");

/*********************************************************
**                                                     **
**        Clear PE0 Memories                            **
**                                                     **
*********************************************************/

printf ("Clearing PE0->PE1 Left Mezz Memory... ");
WS_Mem_Set(PE0_Left_Mezz1_obj, 0, 0, 262144);
if( PE0_Left_Mezz1_obj->rc!=WS_SUCCESS )
{

200
return(P0_LEFT_Mezz1_obj->rc);
}
printf ("Done.
");

printf ("Clearing P0->PE1 Right Mezz Memory... ");
WS_Mem_Set(P0_Right_Mezz1_obj, 0, 0, 262144);
if( P0_Right_Mezz1_obj->rc!=WS_SUCCESS )
{
    return(P0_Right_Mezz1_obj->rc);
}
printf ("Done.
");

printf ("Clearing P0->PE2 Left Mezz Memory... ");
WS_Mem_Set(P0_Left_Mezz2_obj, 0, 0, 262144);
if( P0_Left_Mezz2_obj->rc!=WS_SUCCESS )
{
    return(P0_Left_Mezz2_obj->rc);
}
printf ("Done.
");

printf ("Clearing P0->PE2 Right Mezz Memory... ");
WS_Mem_Set(P0_Right_Mezz2_obj, 0, 0, 262144);
if( P0_Right_Mezz2_obj->rc!=WS_SUCCESS )
{
    return(P0_Right_Mezz2_obj->rc);
}
printf ("Done.
");

// end config_time
times(&uend);
config_time = ( (float)(uend.tms_utime + uend.tms_stime
    - ustart.tms_utime - ustart.tms_stime) /
    ((float)CLK_TCK) );

/*********************************************************
**                                                     **
**       Write Matrix A & B data to PE1 Memories       **
**                                                     **
*********************************************************/
// start load_time
times(&ustart);

printf ("\nWriting Matrix B to PE1 Right Mezz Memory.... ");
WS_Mem_Write(PE1_Right_Mezz_obj, Matrix_B_Offset,
    (Matrix_B_NumDwords+2)*2, Matrix_B_Buffer_64bit );
if( PE1_Right_Mezz_obj->rc!=WS_SUCCESS )
{

return(PE1_Right_Mezz_obj->rc);
}
printf("Done.\n");

printf("Writing Matrix A to PE1 Left Mezz Memory.... ");
WS_Mem_Write(PE1_Left_Mezz_obj, Matrix_A_Offset,
(Matrix_A_NumDwords+2)*2, Matrix_A_Buffer_64bit );
if( PE1_Left_Mezz_obj->rc!=WS_SUCCESS )
{
    return(PE1_Left_Mezz_obj->rc);
}
printf("Done.\n");

/*********************************************************
**                                                     **
**       Write Matrix A & B data to PE0 Memories        **
**                                                     **
**********************************************************/
printf("Writing Matrix B to PE0->PE1 Right Mezz Memory.... ");
WS_Mem_Write(PE0_Right_Mezz1_obj, Matrix_B_Offset,
(Matrix_B_NumDwords+2), Matrix_B_Buffer_32bit );
if( PE0_Right_Mezz1_obj->rc!=WS_SUCCESS )
{
    return(PE0_Right_Mezz1_obj->rc);
}
printf("Done.\n");

printf("Writing Matrix A to PE0->PE1 Left Mezz Memory.... ");
WS_Mem_Write(PE0_Left_Mezz1_obj, Matrix_A_Offset,
(Matrix_A_NumDwords+2), Matrix_A_Buffer_32bit );
if( PE0_Left_Mezz1_obj->rc!=WS_SUCCESS )
{
    return(PE0_Left_Mezz1_obj->rc);
}
printf("Done.\n");

// end load_time
times(&uend);
load_time = ( (float)(uend.tms_utime + uend.tms_stime
    - ustart.tms_utime - ustart.tms_stime) / (float)CLK_TCK
    );

/**********************************************************
**                                                     **
**              Read Matrix A data                      **
**                                                     **
**********************************************************/
202
/*
   printf ("\n\n\n\nReading MATRIX A Data back from PE0 Left Mezz1
Memory: \n")

   WS_Mem_Read(PE0_Left_Mezz1_obj, Matrix_A_Offset,
   (Matrix_A_NumDwords+2), A_ReadBuffer );
   if( PE0_Left_Mezz1_obj->rc!=WS_SUCCESS )
   {
      return(PE0_Left_Mezz1_obj->rc);
   }

   // for (k=0; k<(Matrix_A_NumDwords+2); k++) {
   //    printf ("\n%d", A_ReadBuffer[k]);
   // }

   // printf ("\n");
*/

/*********************************************************
**                                                     **
**              Verify Matrix A data                   **
**                                                     **
/*********************************************************/

/*
rc = VerifyData( Matrix_A_Buffer_32bit, A_ReadBuffer,
   (Matrix_A_NumDwords+2) , "\nVerify failed\n" );
CHECK_RC(rc);

   printf ("\n\tMatrix A Data Written to PE0 Left Mezz Memory1 and
Verified!\n\n");
*/

/*********************************************************
**                                                     **
**              Read Matrix B data                     **
**                                                     **
/*********************************************************/

/*
   printf ("\n\n\n\nReading MATRIX B Data back from PE0 Right Mezz1
Memory: \n")

   WS_Mem_Read(PE0_Right_Mezz1_obj, Matrix_B_Offset,
   (Matrix_B_NumDwords+2), B_ReadBuffer );
   if( PE0_Right_Mezz1_obj->rc!=WS_SUCCESS )
   {
      return(PE0_Right_Mezz1_obj->rc);
   }
*/
// for (k=0; k<(Matrix_B_NumDwords+2); k++) {
//    printf ("\n%d", B_ReadBuffer[k]);
// }

// printf ("\n");

/************************************************************
** ** Verify Matrix B data **
** ************************************************************/
/*
   rc = VerifyData( Matrix_B_Buffer_32bit, B_ReadBuffer,
   (Matrix_B_NumDwords+2) , "\nVerify failed\n" );
   CHECK_RC(rc);
   printf ("\n\tMatrix B Data Written to PE0 Right Mezz1 Memory and
Verified!\n\n");
*/

/******************************************************************************
** Reset PE1 **
******************************************************************************

printf ("\nResetting PE1... ");
printf ("\nWaiting for Interrupt signal... ");

/* start computation time */
times(&ustart);

for (t=0; t<iterations; t++) {
    PEX_Interrupt = 0;
    PE0_Interrupt = 0;

    rc = WS_WritePeReg( WS_SlotNumber, WS_PE0, RESET_REG_OFFSET, 1,
    &Reset0 );
    CHECK_RC(rc);

    rc = WS_WritePeReg( WS_SlotNumber, WS_PE1, RESET_REG_OFFSET, 1,
    &Reset1 );
    CHECK_RC(rc);
    // printf ("done.\n");

    /******************************************************************************
** Wait for Interrupt from the LAD Bus **
******************************************************************************/
// printf ("\nWaiting for Interrupt signal... ");
do {
    rc = WS_ReadPeReg( WS_SlotNumber, WS_PE0,
    INTERRUPT_REG_OFFSET, 1, &PE0_Interrupt );
    rc = WS_ReadPeReg( WS_SlotNumber, WS_PE1,
    INTERRUPT_REG_OFFSET, 1, &PEX_Interrupt );
    CHECK_RC(rc);
} while ( (PEX_Interrupt != 1)&(PE0_Interrupt != 1) );

*/ end computation time */
times(&uend);
compute_time = ( (float)(uend.tms_utime + uend.tms_stime
    - ustart.tms_utime - ustart.tms_stime) /
((float)CLK_TCK * iterations) );

printf ("done.\n");

/***************************************************************
**                                                           **
**         Read back Matrix C from PE0 Left Mezz Memory      **
**                                                           **
/***************************************************************
// start read_time
times(&ustart);

printf ("\n\nReading Matrix C Data from PE0 Left Mezz1 Memory:\n");

WS_Mem_Read(PE0_Left_Mezz1_obj, Matrix_C_Offset,
(Matrix_C_NumDwords+2), Matrix_C_Buffer_32bit );
if( PE0_Left_Mezz1_obj->rc!=WS_SUCCESS )
    return(PE0_Left_Mezz1_obj->rc);

printf ("\nPE0:\n");
printf ("Number of Rows: %d\n", Matrix_C_Buffer_32bit[0]);
printf ("Number of Cols: %d\n\t", Matrix_C_Buffer_32bit[1]);

if (Matrix_C_NumDwords < MAX_OUTPUT) {
    for (m=2; m<(Matrix_C_NumDwords+2); m++) {
        printf ("\t%d", Matrix_C_Buffer_32bit[m]);
        if ( (m-1) % Matrix_B_NumCols == 0 )
            printf ("\n\t");
    }
}

/***************************************************************
**                                                           **
**         Read back Matrix C from PE1 Left Mezz Memory      **
**                                                           **
/***************************************************************
printf ("\nReading Matrix Data from PE1 Memory:\n");

WS_Mem_Read(PE1_Left_Mezz_obj, Matrix_C_Offset,
(Matrix_C_NumDwords+2)*2, Matrix_C_Buffer_64bit );
if( PE1_Left_Mezz_obj->rc!=WS_SUCCESS )
{
    return(PE1_Left_Mezz_obj->rc);
}

// end read_time

read_time = ( (float)(uend.tms_utime + uend.tms_stime
- ustart.tms_utime - ustart.tms_stime) / (float)CLK_TCK
);

WS_Mem_Read(PE1_Left_Mezz_obj, Matrix_A_Offset,
(Matrix_A_NumDwords)*2, Matrix_A_Buffer_64bit );
if( PE1_Left_Mezz_obj->rc!=WS_SUCCESS )
{
    return(PE1_Left_Mezz_obj->rc);
}

WS_Mem_Read(PE1_Right_Mezz_obj, Matrix_B_Offset,
(Matrix_B_NumDwords)*2, Matrix_B_Buffer_64bit );
if( PE1_Right_Mezz_obj->rc!=WS_SUCCESS )
{
    return(PE1_Right_Mezz_obj->rc);
}

PEX:

printf ("Number of Rows: %d\n", Matrix_C_Buffer_64bit[0]);
printf ("Number of Cols: %d\n\t", Matrix_C_Buffer_64bit[2]);

if (Matrix_C_NumDwords < MAX_OUTPUT) {
    for (m=2; m<Matrix_C_NumDwords+2; m++) {
        printf ("\t%d", Matrix_C_Buffer_64bit[m*2]);
        if ( (m-1) % Matrix_B_NumCols == 0 )
            printf ("\n\t");
    }
}

/*
 printf ( "\n\nPEX \n" );
printf ( "-----\n" );

for (m=0; m<256; m++) {
    printf ("\na[%3d]: %d", m, Matrix_A_Buffer_64bit[m*2]);
    printf ("\tb[%3d]: %d", m, Matrix_B_Buffer_64bit[m*2]);
    printf ("\tc[%3d]: %d", m, Matrix_C_Buffer_64bit[m*2]);
}*/

206
if (Matrix_C_NumDwords < MAX_OUTPUT) {
    for (m=0; m<(Matrix_C_NumDwords+2); m++) {
        printf ("[%3d] =	%d", m, Matrix_C_Buffer_32bit[m]);
        printf ("\___/ <    ´--------'       
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
"
printf ("FPGA Memory Read Time : \n", read_time);
printf ("Total Run Time : %f\n",
config_time+prepare_time+load_time+compute_time+read_time);
/*
 printf ("start time = \t%f\t%f\n", (float)ustart.tms_utime,
 (float)ustart.tms_stime);
 printf ("end time = \t%f\t%f\n", (float)uend.tms_utime,
 (float)uend.tms_stime);
 printf ("Loops Performed = \t%d\n", iterations);
 */

/************************************************************
 **                                                     **
 **                  Clear PE1 Memories                  **
 **                                                     **
 ************************************************************/

printf ("
\nClearing PE1 Left Local Memory... ");
WS_Mem_Set(PE1_Left_Mem_obj, 0, 0, 262144);
if( PE1_Left_Mem_obj->rc!=WS_SUCCESS )
{
    return(PE1_Left_Mem_obj->rc);
}
printf ("Done.\n");

printf ("Clearing PE1 Right Local Memory... ");
WS_Mem_Set(PE1_Right_Mem_obj, 0, 0, 262144);
if( PE1_Right_Mem_obj->rc!=WS_SUCCESS )
{
    return(PE1_Right_Mem_obj->rc);
}
printf ("Done.\n");

printf ("Clearing PE1 Left Mezz Memory... ");
WS_Mem_Set(PE1_Left_Mezz_obj, 0, 0, 262144);
if( PE1_Left_Mezz_obj->rc!=WS_SUCCESS )
{
    return(PE1_Left_Mezz_obj->rc);
}
printf ("Done.\n");

printf ("Clearing PE1 Right Mezz Memory... ");
WS_Mem_Set(PE1_Right_Mezz_obj, 0, 0, 262144);
if( PE1_Right_Mezz_obj->rc!=WS_SUCCESS )
{
    return(PE1_Right_Mezz_obj->rc);
}
printf ("Done.\n");
printf ("Clearing PE0->PE1 Left Mezz Memory... ");
WS_Mem_Set(PE0_Left_Mezz1_obj, 0, 0, 262144);
if( PE0_Left_Mezz1_obj->rc!=WS_SUCCESS )
{
    return(PE0_Left_Mezz1_obj->rc);
}
printf ("Done.
");

printf ("Clearing PE0->PE1 Right Mezz Memory... ");
WS_Mem_Set(PE0_Right_Mezz1_obj, 0, 0, 262144);
if( PE0_Right_Mezz1_obj->rc!=WS_SUCCESS )
{
    return(PE0_Right_Mezz1_obj->rc);
}
printf ("Done.
");

printf ("Clearing PE0->PE2 Left Mezz Memory... ");
WS_Mem_Set(PE0_Left_Mezz2_obj, 0, 0, 262144);
if( PE0_Left_Mezz2_obj->rc!=WS_SUCCESS )
{
    return(PE0_Left_Mezz2_obj->rc);
}
printf ("Done.
");

printf ("Clearing PE0->PE2 Right Mezz Memory... ");
WS_Mem_Set(PE0_Right_Mezz2_obj, 0, 0, 262144);
if( PE0_Right_Mezz2_obj->rc!=WS_SUCCESS )
{
    return(PE0_Right_Mezz2_obj->rc);
}
printf ("Done.
");

/**
 ** Free the PE1 Left Memory
 **
***************************************************************************/
 printf("\n\nReleasing Memory...");
WS_Mem_Release( PE1_Right_Mem_obj );

209
WS_Mem_Release( PE1_Left_Mem_obj );
WS_Mem_Release( PE1_Right_Mezz_obj );
WS_Mem_Release( PE1_Left_Mezz_obj );
WS_Mem_Release( PE0_Right_Mezz1_obj );
WS_Mem_Release( PE0_Left_Mezz1_obj );
WS_Mem_Release( PE0_Right_Mezz2_obj );
WS_Mem_Release( PE0_Left_Mezz2_obj );

free (Matrix_A_Buffer_64bit);
free (Matrix_B_Buffer_64bit);
free (Matrix_C_Buffer_64bit);
free (Matrix_A_Buffer_32bit);
free (Matrix_B_Buffer_32bit);
free (Matrix_C_Buffer_32bit);
free (A_ReadBuffer);
free (B_ReadBuffer);

printf ( "done.\n\n" );

/***************************************************************
**                                                          **
**             Leave the board in a known state              **
**                                                          **
/***************************************************************/
rc = Mem_Shutdown( WS_SlotNumber, &WildstarConfiguration, PeMask);
CHECK_RC(rc);

printf ( "\nMatrix Mulitply shutdown successful.\n\n\n" );

return (rc);
}

/***************************************************************
**          Exported Function                                **
**          ***************************************************************/
static WS_RetCode
VerifyData( DWORD    ref[],
DWORD    test[],
DWORD    size,
char    *errstr )
{

DWORD
memCntr,
errCount;

WS_RetCode
rc = WS_SUCCESS;

return (rc);
}
errCount=0;

/*********************************************************
**                                                     **
** Loop counts off in DWORDS. Mismatches will stop **
** being counted after MAX_ERR_COUNT errors. **
**                                                     **
*********************************************************/
for (memCntr=0; (memCntr < size) && (errCount < MAX_ERR_COUNT); memCntr++)
{
    if (ref[memCntr] != test[memCntr])
    {
        rc=ERROR_MEMORY_COMPARE FAILED;
        if (errCount==0)
        {
            printf ("\n\nERROR: %s\n", errstr);
            printf ("t Failure offset:\tDWORD:\tExpected Data:\tActual Data:\n\n");
            printf ("\t 0x%08lx\t\t%i\t0x%08x\t0x%08x\t",
                    memCntr*sizeof(DWORD), memCntr, ref[memCntr], test[memCntr]);
            printf ("\n\n");
            errCount++;
        }
    }
}
else if (errCount!=0)
{
    printf ("Number of memory errors found: %d.\n\n", errCount);
}
else if (errCount!=0)
{
    printf ("Memory comparison stopped after %d errors.\n\n", MAX_ERR_COUNT);
}
return(rc);

/************************************************************************
* Function:        Mem_Startup
* Description:    Initializes the WILDSTAR board and checks the PE mask.
* Arguments:      
*   WS_SlotNumber          Slot or board number to access.
*   *WildstarConfiguration  Pointer to Wildstar physical configuration
*   *rPeMask                Pointer to PE mask associated with target
*   * clk_freq               Frequency to set the clock to
*   
*/
WS_RetCode
Mem_Startup( DWORD                   WS_SlotNumber,
            WS_PhysicalBoardConfig *WildstarConfiguration,
            DWORD                  *rPeMask,
            float                   clk_freq,
            DWORD                   dBusFreq )
{
    WS_RetCode
    rc = WS_SUCCESS;
    DWORD
    PeMask = *rPeMask;
    char
    PathName[MAX_PATH],
    FileName[MAX_PATH];
    /**********************************************************************************
    **
    **                  Check the Pe mask
    **
   **********************************************************************************/
    assert ( *rPeMask <= MAX_PE_MASK );
    if ( WildstarConfiguration->BaseInfo.BaseBoardType == WS_STARFIRE )
    {
        if ( (*rPeMask & (WS_PE1_MASK ^ MAX_PE_MASK)) != 0 )
        {
            PeMask = WS_PE1_MASK;
        }
    }
    else if ( WildstarConfiguration->BaseInfo.BaseBoardType ==
              WS_STARFIRE_II )
    {
        if ( (*rPeMask & ((WS_PE1_MASK | WS_PE2_MASK) ^ MAX_PE_MASK)) != 0 )
        {
            PeMask = WS_PE1_MASK | WS_PE2_MASK;
        }
    }
    *rPeMask = PeMask;
    /**********************************************************************************
    **
    **                  Set the MClock Frequency
    **
   **********************************************************************************/
    rc = WS_MClkSetConfig ( WS_SlotNumber, PROG_OSCILLATOR, clk_freq, 1 );
    CHECK_RC(rc);
    printf ( "Successfully Set MCLK to [%5.2f] MHz\n\n", clk_freq );

212
printf ( "Programming PEs...
" );

/***************************************************************
**                                                           **
**                         Program PE0                       **
**                                                           **
/***************************************************************/

sprintf (FileName , "pe0_matrix_multiply_parallel.m68");

rc = ProgramPE(WS_SlotNumber, WS_PE0, FileName );
if ( rc != WS_SUCCESS )
{
    CHECK_RC (rc);
}

printf ( "Successfully programmed PE0 with %s
", FileName );

/***************************************************************
**                                                           **
**                         Program PE1                       **
**                                                           **
/***************************************************************

sprintf (FileName , "pex_matrix_multiply_parallel.m68");

rc = ProgramPE(WS_SlotNumber, WS_PE1, FileName );
if ( rc != WS_SUCCESS )
{
    CHECK_RC(rc);
}

printf ( "Successfully programmed PE1 with %s
", FileName );

/***************************************************************
**                                                           **
**                         Program PE2                       **
**                                                           **
/***************************************************************

sprintf (FileName , "safe.m68");

rc = ProgramPE(WS_SlotNumber, WS_PE2, FileName );
if ( rc != WS_SUCCESS )
{
    CHECK_RC(rc);
}

printf ( "Successfully programmed PE2 with %s
", FileName );

return(rc);
/**
 * Function:        Mem_Shutdown
 * 
 * Description:     Loads "safe" PE images to put the board in a known state.
 * 
 * Arguments:
 *   WS_SlotNumber          Slot or board number to access.
 *   *WildstarConfiguration Pointer to Wildstar physical configuration structure.
 *   *PeMask                 PE mask associated with target board.
 * 
 * Returns:         DWORD - return code
 * 
 */

WS_RetCode Mem_Shutdown( DWORD                   WS_SlotNumber,
                         WS_PhysicalBoardConfig *WildstarConfiguration,
                         DWORD                   PeMask )
{

    WS_RetCode rc = WS_SUCCESS;

    char PathName[MAX_PATH],
         FileName[MAX_PATH];

    printf ( "Programming PEs with safe images...
" );

    /**
     **
     **     Program PE0
     **
     **
     */

    sprintf (FileName , "safe.m68");

    rc = ProgramPE(WS_SlotNumber, WS_PE0, FileName );
    if ( rc != WS_SUCCESS )
    {
        CHECK_RC (rc);
    }
    printf ( "Successfully programmed PE0 with %s\n", FileName );

    /**
     **
     **     Program PE1
     **
     **
     */

    sprintf (FileName , "safe.m68");

    rc = ProgramPE(WS_SlotNumber, WS_PE1, FileName );
    if ( rc != WS_SUCCESS )
{ CHECK_RC(rc); }
printf ( "\tSuccessfully programmed PE1 with %s\n", FileName );

/****************************************************************************
**                        Program PE2                              **
****************************************************************************/
sprintf (FileName , "safe.m68");
rc = ProgramPE(WS_SlotNumber, WS_PE2, FileName );
if ( rc != WS_SUCCESS )
{
  CHECK_RC(rc);
}
printf ( "\tSuccessfully programmed PE2 with %s\n\n", FileName );

return (rc);
}

/**************************************************************************** Exported Function ****************************************************************************
* Function:   ProgramPE
* Returns:    DWORD - return code
* Description:   programs PE from given filename
***************************************************************************/
WS_RetCode
ProgramPE( DWORD WS_SlotNumber, DWORD PeNum, char * filename)
{
#define WS_MAX_PGM_DWORDS ( 400000 )

FILE*
pInFile;

DWORD
*progBuffer,
numDwordsRead;

WS_RetCode
rc = WS_SUCCESS;

if ( (pInFile = fopen(filename,"rb")) != NULL )
{
  progBuffer = (DWORD *)malloc( WS_MAX_PGM_DWORDS * sizeof(DWORD) );
  if( progBuffer != NULL )
  {
    numDwordsRead = fread( progBuffer,
                           sizeof(DWORD),
                           WS_MAX_PGM_DWORDS,
pInFile);

rc = WS_ProgramPe( WS_SlotNumber,
    PeNum,
    progBuffer,
    numDwordsRead,
    NULL);

    free(progBuffer);
} else
{
    printf ("Memory allocation error\n");
    rc = -1;
} fclose( pInFile );
} else
{
    printf ("Unable to open file '%s'\n", filename );
    rc = -1;
}

return(rc);
BENCHMARK CODES IN MATLAB

E.1 Homogeneous
E.2 Ithresh
E.3 Motion Estimation
E.4 128x128 Matrix Multiplication
E.5 256x256 Matrix Multiplication
E.6 Sobel Transform
E.7 Vectorsum
E.1 Homogeneous

%!match TYPE integer min
min = 255;

%!match TYPE integer T
T = 256;

%!match TYPE integer max
max = 0;

for i = 1:1:256
    for j = 1:1:256
        a(256)
        a(i,j) = i - 1;
    end
end

for i = 1:1:256
    for j = 1:1:256
        a(256)
        if(a(i,j) < min)
            min = a(i,j);
        end
        a(256)
        if(a(i,j) > max)
            max = a(i,j);
        end
    end
end

%!match TYPE integer min
%!match TYPE integer max
%!match TYPE integer T
if((max - min) < T)
    homoge = 1;
else
    homoge = 0;
end

%!match TYPE integer homoge_val(1)
homoge_val(1) = homoge;
E.2 Ithresh

```plaintext
%!match TYPE integer t
t = 100;

for i = 1:1:256
    for j = 1:1:256
        %!match TYPE integer a(256)
        a(i,j) = i - 1;
    end
end

for i = 1:1:256
    for j = 1:1:256
        %!match TYPE integer a(256)
        %!match TYPE integer t
        if(a(i,j) < t)
            %!match TYPE integer b(256)
            b(i,j) = 0;
        else
            %!match TYPE integer b(256)
            b(i,j) = 1;
        end
    end
end
```

E.3 Motion Estimation

```plaintext
%!match TYPE integer x
for x = 1:1:40
  %!match TYPE integer y
  for y = 1:1:40
    %!match TYPE integer new(40,40)
    new(x,y) = 255;
    %!match TYPE integer old(40,40)
    old(x,y) = 240;
  end
end

%!match TYPE integer delta
delta = 0;

%!match TYPE integer t
t = 0;
%!match TYPE integer i
i = 1;

%!match TYPE integer g
for g = 1:1:4
```
%!match TYPE integer h
for h = 1:1:4
%!match TYPE integer x
 for x = 1:1:4
%!match TYPE integer y
 for y = 1:1:4
%!match TYPE integer delta
     delta = 0;
%!match TYPE integer k
     for k= 1:1:4
%!match TYPE integer l
     for l= 1:1:4
%!match TYPE integer t
%!match TYPE integer new(40,40)
%!match TYPE integer old(40,40)
     t = new(g*8+k,h*8+l)-old(g*8+x+k,h*8+y+l);
%!match TYPE integer t
     if(t < 0)
%!match TYPE integer delta
         delta = delta - t;
     else
%!match TYPE integer delta
     delta = delta + t;
     end
%!match TYPE integer i
%!match TYPE integer delta_out(4500)
%!match TYPE integer delta
     delta_out(i) = delta;
%!match TYPE integer i
     i = i+1;
end
end
end
end
end
E.4 128x128 Matrix Multiplication

```plaintext
%!match TYPE integer i
for i = 1:1:128
%!match TYPE integer j
  for j = 1:1:128
%!match TYPE integer a(128,128)
%!match MEMLOC right a
  a(i,j) = 10;
%!match TYPE integer b(128,128)
%!match MEMLOC right b
  b(i,j) = 10;
end
end
```

```plaintext
%!match TYPE integer i
for i = 1:1:128
%!match TYPE integer j
  for j = 1:1:128
%!match TYPE integer temp
%!match TYPE integer k
  for k = 1:1:128
%!match TYPE integer a(128,128)
%!match TYPE integer b(128,128)
    temp = temp + a(i,k)*b(k,j);
  end
%!match TYPE integer c(128,128)
%!match MEMLOC right c
  c(i,j) = temp;
end
end
```

E.5 256x256 Matrix Multiplication

```plaintext
%!match TYPE integer i
for i = 1:1:256
%!match TYPE integer j
  for j = 1:1:256
%!match TYPE integer a(256,256)
  a(i,j) = 10;
%!match TYPE integer b(256,256)
  b(i,j) = 10;
end
end
%!match TYPE integer i
```
for i = 1:1:256
    %!match TYPE integer j
    for j = 1:1:256
        %!match TYPE integer temp
        temp = 0;
        %!match TYPE integer k
        for k = 1:1:256
            %!match TYPE integer a(256,256)
            %!match TYPE integer b(256,256)
            temp = temp + a(i,k)*b(k,j);
        end
        %!match TYPE integer c(256,256)
        c(i,j) = temp;
    end
end

E.6 Sobel Transform

%!match TYPE integer c
%  c  = 0;
%!match TYPE integer t
for  t=1 : 1 : 256
    %!match TYPE integer v
    for  v=1 : 1 : 256
        %!match TYPE integer a(256,256)
        a(t,v)  = t;
    end
end

%!match TYPE integer x
for  x=2 : 1 : 255
    %!match TYPE integer y
    for  y=2 : 1 : 255
        %!match TYPE integer cc
        cc = 0 -a(x-1,y-1) -2*a(x-1,y) - a(x-1,y+1);
        %!match TYPE integer cc
        %!match PARALLEL BEGIN
        cc = cc + a(x+1,y-1) + 2*a(x+1,y) + a(x+1,y+1);
        %!match TYPE integer cc
        if( cc < 0 )
            %!match TYPE integer c
            c = 0 - cc;
        else
            %!match TYPE integer c
            c = cc;
        end
        %!match TYPE integer ccl
        ccl = 0 -a(x-1,y-1) -2*a(x,y-1) - a(x+1,y-1);
        %!match TYPE integer ccl
        %!match PARALLEL BEGIN

222
\[ cc1 = cc1 + a(x-1,y+1) + 2*a(x,y+1) + a(x+1,y+1); \]

```
cc1 = cc1 + a(x-1,y+1) + 2*a(x,y+1) + a(x+1,y+1);

%!match TYPE integer cc1
  if( cc1 < 0)
    %!match TYPE integer c
    %!match PARALLEL BEGIN
      c = c - cc1;
    end
  else
    %!match TYPE integer c
    %!match PARALLEL BEGIN
      c = c + cc1;
    end
  end

%!match TYPE integer c
  if(c > 255)
    %!match TYPE integer c
    c = 255;
  end
%!match TYPE integer b(256,256)
%!match BITS(8) b
  b(x,y) = c;
end
end
```

E.7 Vectorsum

```
%!match TYPE integer i
  for i = 1:1:5
    %!match TYPE integer a(65000)
    %!match MEMLOC mezzleft a
    a(i) = 5;
  end

%!match TYPE integer temp
temp = 0;

%!match TYPE integer i
  for i = 1:1:5
    %!match TYPE integer a(65000)
    temp = temp + a(i);
  end

%!match TYPE integer vsun(1)
%!match MEMLOC mezzright vsun
  vsun(1) = temp;
```
REFERENCES


224


