Spring 5-31-2003

Design of an FPGA-based parallel SIMD machine for power flow analysis

Tirupathi Rao Kunta
New Jersey Institute of Technology

ABSTRACT

DESIGN OF AN FPGA-BASED PARALLEL SIMD MACHINE FOR POWER FLOW ANALYSIS

by
Tirupathi Rao Kunta

Power flow analysis consists of computationally intensive calculations on large matrices, consumes several hours of computational time, and has shown the need for the implementation of application-specific parallel machines. The potential of Single-Instruction stream Multiple-Data stream (SIMD) parallel architectures for efficient operations on large matrices has been demonstrated as seen in the case of many existing supercomputers. The unsuitability of existing parallel machines for low-cost power system applications, their long design cycles, and the difficulty in using them show the need for application-specific SIMD machines. Advances in VLSI technology and Field-Programmable Gate-Arrays (FPGAs) enable the implementation of Custom Computing Machines (CCMs) which can yield better performance for specific applications. The advent of Soft-Core processors made it possible to integrate reconfigurable logic as a slave to a peripheral bus and has demonstrated the ability in the rapid prototyping of complete systems on programmable chips. This thesis aims at designing and implementing an FPGA-based SIMD machine for power flow analysis. It presents the architecture of an SIMD machine that consists of an array of processing elements with mesh interconnection and a Soft-Core processor; the latter is used as the host. The FPGA-based SIMD machine is implemented on the Annapolis Microsystems Wildstar-II board that contains multiple Virtex-II FPGAs. The Soft-Core processor used is the Xilinx Microblaze and the application targeted is matrix multiplication.

DESIGN OF AN FPGA-BASED PARALLEL SIMD MACHINE FOR POWER FLOW ANALYSIS

by
Tirupathi Rao Kunta

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

May 2003

APPROVAL PAGE

DESIGN OF AN FPGA-BASED PARALLEL SIMD MACHINE FOR POWER FLOW ANALYSIS

Tirupathi Rao Kunta

Dr. Sotirios Ziavras, Thesis Advisor — Date
Professor of Electrical and Computer Engineering, and
Computer and Information Science, Associate Chair for Graduate Studies, NJIT

Dr.Roberto Rojas-cessa, CommitteeMember — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Alexandros Gerbessiotis, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Tirupathi Rao Kunta

Degree: Master of Science

Date: May 2003

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering
  New Jersey Institute of Technology, Newark, NJ, 2003

- Bachelor of Science in Electrical and Electronics Engineering
  University of Madras, Chennai, India, 2001

Major: Electrical Engineering

To My Parents and the Almighty

ACKNOWLEDGEMENT

I would like to express my deepest appreciation to Dr. Sotirios Ziavras, who not only served as my research advisor, providing valuable and countless resources, insight, intuition, but also constantly gave me support, encouragement, and reassurance. My special thanks are given to Dr. Alexandros Gerbessiotis and Dr. Roberto Rojas-cessa for actively participating in the committee.

I also would like to thank my team mates Amit, Satchit, Sunil, Prabhakar and Abhishek in the Power Grid Laboratory, for their constant encouragement and support. I also wish to give my special thanks to Xizhen for his valuable support and encouragement through out my research. Last, but not the least, I wish to thank my close friends Harsha, Pavan and Ramesh for their understanding and moral support through out my masters.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Power System Problems

Most power system problems such as power flow, transient stability, etc., require the repetitive solution of linear equations represented by large matrices, on the order of 3000X3000. Power flow analysis is done in real time to monitor the performance of the network continuously in order to determine disturbances, such as power station failures, broken lines, and line overcharge [1]. This real-time analysis is also used to speed up the process of deciding to purchase power from neighboring utilities according to expected customer needs and available prices of power. The heavy computational load associated with repetitive solutions of large matrices involved in the above real time analysis poses a big challenge to computational times for conventional machines; they take several hours to complete the analysis [9]. Many parallel techniques have been developed to reduce the number of floating-point operations and, therefore, the total computation time. However, these techniques do not reduce, to a great extent, the computational times which are several hours for these computation intensive power flow problems. Recent developments in computer technology suggest that computational times can be drastically reduced by developing parallel architectures specific to the applications.

Parallel processing hardware and software provide an opportunity and challenge to apply this new computing technology to solve power system problems. There is a recent shift towards this technology as it has the potential to be cost effectively used to solve computationally intensive problems. The three main issues in parallel processing

are parallel architecture development, software development, and algorithm development. Much progress has been reported in these areas from past research and still a lot of research is expected to solve specific problems.

The major computational part of the power system applications consists of algorithms such as LU decomposition, matrix multiplication, and iterative techniques for matrix inversion. These operations require floating-point operations with large matrices. These floating-point operations appear in the most critical computational kernels of the algorithm. Solving a system of N linear equations, such as A*x=b, is at the core of power flow analysis. To solve A*x=b, we can use direct methods or iterative methods. The most widely used direct method is LU decomposition, which is a decomposition of A into a product of upper and lower triangular matrices (U and L, respectively). Once the elements in L and U are determined, then the unknown vector x can be determined by forward reduction and backward substitution, respectively, using the equations L*y=b and U=x*y. LU decomposition has drastically reduced costs compared to matrix inversion, especially for large matrices.

LU decomposition with forward reduction and backward substitution is a more numerically stable technique when compared to matrix inversion because every non-singular matrix can be factored with LU decomposition [9]. The other alternative to LU decomposition is to solve N Gaussian elimination problems followed by N back substitution problems, so the order of the algorithm is $N*\mathbf{O}(N^3) + N * \mathbf{O}(N^2) = \mathbf{O}(N^4)$. In contrast, LU decomposition is applied once in time $\mathbf{O}(N^3)$ and can be used to solve the N systems using N additional $\mathbf{O}(N^2)$ algorithms for forward and backward substitutions. Therefore, the complexity of LU decomposition is $\mathbf{O}(N^3) + N * \mathbf{O}(N^2) =$

$\mathbf{O}(N^3)$; the storage space required is minimal as the original space storing A is used to store L and U. If we assume that L has all 1's on the diagonal, then A=L*U in matrix form is

$$\begin{pmatrix} A_{11} & A_{12} & \dots & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & \dots & A_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ A_{N1} & A_{N2} & \dots & \dots & A_{NN} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & \dots & 0 \\ L_{21} & 1 & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ L_{N1} & L_{N2} & \dots & \dots & 1 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & \dots & \dots & U_{1N} \\ 0 & U_{22} & \dots & \dots & U_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \dots & U_{NN} \end{pmatrix}$$

We can derive the following equations for the LU decomposition algorithm:

$$L_{ij} = (A_{ij} - \sum_{k=1}^{j-1} L_{ik} * U_{kj}) * \frac{1}{U_{jj}} \quad for\ j \in [1, i-1]$$

$$U_{ij} = A_{ij} - \sum_{k=1}^{i-1} L_{ik} * U_{kj} \quad for\ j \in [i, N]$$

From observing the structure of the L and U matrices, it is clear that there is no need to store L, U, and A separately. During decomposition, the modified elements of A are replaced with new elements of L and U. The diagonal of L always contains 1's and does not require storage. Matrix multiplication also requires large computational time when it is applied to large matrices. The complexity of matrix multiplication is O (N^3). These two applications are most widely used to benchmark parallel computers.

1.2 Overview of Parallel Architectures

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. This definition is broad enough to include parallel supercomputers that have hundreds or thousands of processors, networks of workstations,

multiple-processor workstations, and embedded systems. Parallel computers offer the potential to appropriately allocate computational resources on important computational problems; whether they are processors, memory, or I/O bandwidth.

According to Flynn's classification, the two important parallel processing architecture classes are Single-Instruction stream Multiple-Data stream (SIMD) and Multiple-Instruction stream Multiple-Data stream (MIMD). In MIMD, each processor runs its own instruction sequence and works on a different part of the problem; each processor can communicate data to other processors. In MIMD, processors may have to wait for other processors to send data or to access global data. In SIMD, all processors are given the same instruction each time and processors operate on different data. Typically, MIMD processors have powerful processors, distributed shared-memory architecture, and are suitable for control level parallelism and coarse-grain parallelism. In contrast, SIMD processors typically have many elementary processors and distributed memory, and are suitable for data-level parallelism and fine-grain parallelism. Figures 1.1 and 1.2 show block diagrams for SIMD and MIMD machines, respectively, where CU is the control unit.

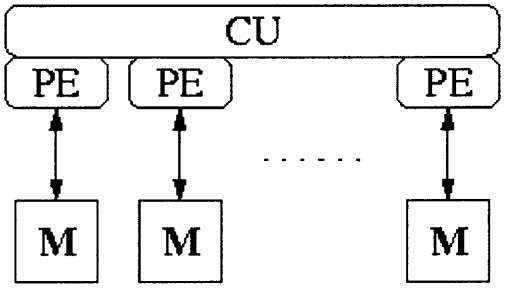

Figure 1.1 SIMD.

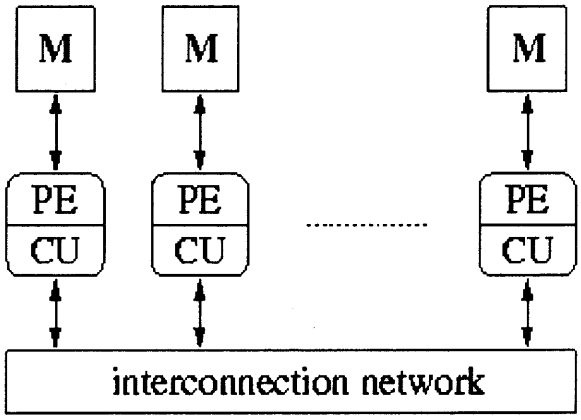

Figure 1.2 MIMD.

Another way to classify parallel computers is based on memory-processor connectivity and the interconnection network.

1.2.1 Memory-Processor Connectivity

We can distinguish parallel computers by the way processors are connected to shared memory. We have shared memory, distributed memory and distributed-shared memory systems.

1.2.1.1 Shared Memory. The main feature of shared-memory multiprocessors is that each processor has access to a global address space, which is shared by all processors in the system. Communication and synchronization are implicitly done by shared variables. Problems can arise if two processors want to write the same variable simultaneously. To prohibit this situation, the user can use synchronization mechanisms like semaphores and memory location locks. Generally, writing programs for shared-memory multiprocessors is said to be easier than for distributed because the address space is global and can be used conveniently. However, a major drawback is related to the scalability of the system.

Adding more processors to the system leads to substantial performance degradation due to frequent message collisions. In fact, most of such systems do not have more than 64 processors, because the centralized memory and the interconnection network are both difficult to scale once built.

1.2.1.2 Distributed Memory. In a distributed-memory multiprocessor, each processor has its own address space. The information exchange has to be done over the point-to-point interconnection network by message-passing. Every processing unit with its memory block can be seen as a node within the multiprocessor. If a processor needs information from another processor, it has to make a request over the communication network. For message-passing systems, different kinds of messages exist.

1.2.1.3 Distributed Shared-Memory. Distributed shared-memory multiprocessors combine features of distributed as well as shared-memory machines. The memory is physically distributed among the nodes. However, the user has the illusion of one big single address space. An example of a distributed shared-memory machine is the SGI Origin 2000, a cache-coherent non-uniform memory access (ccNUMA) multiprocessor designed and manufactured by Silicon Graphics, Inc and Cray. It consists of up to 512 nodes interconnected by a Cray link network. The distributed memory blocks can transparently be accessed like for shared-memory architecture.

1.2.2 Interconnection Networks

One of the most important components of a multiprocessor machine is the interconnection network. In order to solve a problem, the processing elements have to cooperate and exchange their computed data over the network. We can distinguish between two different classes of inter connection topologies:

- Static and
- Dynamic.

1.2.2.1 Static Connection Topologies. In static networks, the nodes are connected to each other directly by wires. Once the connection between nodes has been established, they cannot be changed. In a ring, for example, every processing element (PE) is wired to just two neighbors, whereas in a 4-D hypercube every PE is connected to four neighbors. If a node wants to communicate with any non-neighboring node, all nodes between the two communicating processors have to be accessed.

(a) Mesh and Ring: The simplest connection topology is the n-dimensional mesh. In the 1-D mesh, all nodes are arranged in a line as an array structure, where the interior nodes have two and the boundary nodes have one neighbor(s). In the two-dimensional mesh, each node in the two-dimensional array is connected to four neighbors. If the free ends of the mesh are connected to the opposite sides, the network is then called torus.

(b) Binary Tree: The basic (binary) tree has two children for the root node. The interior nodes have three connections (two children, and one parent), whereas all leaves are connected to just one parent.

(c) Star: A star has one central node, which is connected to all other nodes. The disadvantage of this topology is obvious. Higher-level nodes can easily become bottlenecks.

(d) Hypercube: Several companies, like Intel and NCUBE, have built hypercube-based multicomputers. In the n-dimensional hypercube, the number of nodes is $N = 2^n$. The numbering of the nodes in a hypercube is done so that the addresses of neighbors differ in

just one bit position if n-bit distinct numbers are assigned as node addresses. For example, in the 4-D hypercube node 0 (0000) has the following neighbors:

Node 1 (0001)

Node 2 (0010)

Node 4 (0100)

Node 8 (1000)

1.2.2.2 Dynamic Connection Topologies. Contrary to static networks, dynamic network topologies can vary at runtime. Generally, dynamic networks can be distinguished by their switching strategy and their number of switching stages (single-stage or multi-stage networks). Many computationally intensive problems have a very high degree of structured, fine-grain parallelism and can benefit from highly parallel execution. Power system problems come under the above categories and map well onto parallel SIMD processor arrays [**3**].

1.2.3 Overview of SIMD Computers

SIMD architectures are common in parallel processing. The ability of SIMD machines to manipulate large vectors efficiently has created a phenomenal demand in such areas as weather modeling and cancer radiation research. The power of this kind of architecture is seen when the number of processors is equal to the size of the vectors. Even when the number of processors available is less than the size of the vectors, the speedup, compared to a sequential algorithm, can be immense. In SIMD architectures, a single control unit (CU) fetches and decodes instructions. Then, each instruction is executed either in the

CU itself or is broadcast to a collection of processing elements (PEs). These PEs operate synchronously but their local memories have different contents.

Illiac IV was the fist large SIMD machine built in the early 1970s. It had 32 PEs, each of which was a processor with its own local memory; the PEs were connected in a ring. The ICL Distributed Array Processor (DAP) and the Goodyear (Massively Parallel Processing) MPP SIMD machines came in the late 1970s.The ICL had either 1K or 4K one-bit processors arranged in a square plane, each connected in a rectangular fashion to its nearest neighbors. The MPP had 16K one-bit processors, each with local memory, and was programmed in Pascal and assembly language. The DAP and the MPP were fine-grain systems based on single-bit processors, whereas the Illiac IV was a large-grain SIMD system.

In the late 1980s, three new SIMD fine-grain machines were developed: CM-2, MasPar and a new version of DAP. These SIMD machines are generally used in conjunction with conventional computers. CM-2 uses a VAX machine or a Silicon Graphics workstation as a host computer. CM-2 has upto 64K single-bit processors, 2048 64-bit floating-point processors and 8 Gbytes of memory. An application running on the host downloads data into the processor array of the CM-2. Each PE in the CM-2 acts as a single memory unit. The host then issues instructions to each PE of the CM-2 simultaneously. After the computations are complete, the host then reads back the result from CM-2 as though it were conventional memory. The CM-2 has been measured at 5.2 GFLOPS running the unlimited Linpack benchmark solving a linear system of order 26,624. Even higher performance was achieved on some applications, e.g., seismic data processing and QCD. The new DAP is front ended by sun workstations or VAXes. The

MasPar architecture consists of fine-grain SIMD units combined with enhanced floating-point performance; it uses 4-bit (Maspar MP-1) or 32-bit (Maspar MP-2) basic SIMD units. By the 1990s, parallel computers reached substantial peak performance with the Intel Paragon and the CM-5 yielding more than 100GFLOPS. However, their high price, long development cycles, the difficulty of sometimes programming them, the high cost of maintaining them, and their unsuitability for all applications limit their usage in diverse computing fields.

1.3 Overview of Reconfigurable Computing Systems

Field-Programmable Gate-Arrays (FPGAs) have been used in systems spanning a broad range of applications ever since their introduction in 1985 [**4**]. Most of the systems used FPGAs as a glue logic providing the advantages of high integration levels without the expense and risk of custom ASIC devices. However, as FPGAs have increased in capability, their use as in-system configurable computing elements is receiving considerable attention. The use of FPGAs as reconfigurable computing elements is poised to expand rapidly in the commercial market, where FPGA–based parallel processors will compete with parallel computers and even some supercomputers in computationally intensive applications. Many research projects were done over the past few years in developing these FPGA-based high-performance machines. Reconfigurable FPGA technology holds the potential for reshaping the future of computing by providing the capability to dynamically alter hardware resources to optimally serve immediate computational needs [**2**].

FPGA-based reconfigurable systems can be used as specialized co-processors, processor-attached functional units, attached message routers in parallel machines, and specialized systems for parallel processing. This was made possible with the advent of multi-million gate FPGAs. In the past decade, FPGA-based configurable computing machines have acquired significant attention for improving the performance of algorithms in several fields, such as DSP, data communications, genetics, image processing, pattern recognition, etc. FPGA-based co-processors are implemented as attached co-processors dedicated to off-loading computationally-intensive tasks from host-processors in PCs and workstations. Reconfigurable co-processors are viable platforms for a wide-range of computationally-intensive applications. FPGA-based configurable computing systems have garnered support from the scientific and academic communities. Many research projects have demonstrated the viability of configurable computing systems that can deliver the performance of supercomputers for specific applications. Most of the FPGA–based parallel machines currently reside in multi-FPGA systems interconnected via a specific network **[6]**. Some of the configurable computing systems are:

1. The Ganglion Project at IBM Almaden Research Centre used XC3090 and XC3042 FPGA devices to implement a feed-forward, fully interconnected neural network on a single VME board.

2. DEC's Paris Research Lab has designed and implemented four generations of FPGA-based configurable co-processors called Programmable Active Memories (PAMs).

3. SPLASH-1 includes a 32-stage linear-logic array with a VME-interface to a SUN workstation. Each stage consists of an XC3090 FPGA and a 128Kbyte static memory buffer. SPLASH-1 outperformed Cray-2 by a factor of 325 in specific applications and a custom built NMOS device by a factor of 45. SPLASH-2 uses 17 XC4010 FPGA devices arranged in a linear array and also interconnected via a 16x16 crossbar.

4. PRISM-1 from Brown University coupled XC3090 with the Motorola M68010 microprocessor and PRISM-11 coupled XC4010 FPGA devices as co-processors to an AMD29050 RISC processor.

Advances in VLSI technology not only brought about multi-million gate FPGAs, but also facilitated the integration of numerous functions onto a single FPGA chip. Peripherals formerly attached to the FPGA at the board level now can be embedded into the same chip with the configurable logic. According to Xilinx predictions, the count of FPGA system gates will exceed 50 million and FPGA chips will operate at more than 500 MHz [**9**].Thus, the availability of multi-million system gates in FPGAs introduced a new design paradign, System-On-a-Chip (SOC), with which entire systems can be implemented on a single FPGA chip without the need for expensive non-recurring engineering charges or costly software tools.

1.4 Overview of Soft-Core Processors

Today's highly competitive embedded system market requires embedded system designers to reduce both the time to market, and the non-returnable engineering and manufacturing costs. In order to achieve this, embedded system designers need to rapidly prototype new designs and easily integrate different digital components into one design, while still meeting the design's functionality and performance requirements. Designers also have to handle other issues such as the use of reusable intellectual property software or hardware modules. The ability to use existing code and migrate code to and from different embedded system platforms are also important design issues for embedded system designers.

To the embedded system designer, FPGAs represent the ability to reconfigure digital designs and also to integrate different digital designs together. The advantages of microprocessors over FPGAs are ease of use and good performance. Recently, designers have sought to combine the advantages of custom logic devices with that of microprocessors. One of the more popular solutions is the use of soft-core microprocessors. A soft-core processor is a reconfigurable microprocessor combining the advantages of programmable logic devices and microprocessors. The reconfigurable architecture of a soft-core processor is used to modify hardware and software designs, which makes a soft core processor ideal for rapid prototyping. The main difference between hardcore-processors (ASIC devices) and soft-core processors is that soft-core processors have reconfigurable architectures. Soft-core processors are more flexible than hard-core processors since custom logic can be easily interfaced to the processor. This gives the designer more flexibility when interfacing soft-core processors with the rest of the embedded system. They are also scalable, which allows more than one processor to be used in a particular design. The scalable, flexible and rapid prototyping factors make the soft-core processor suitable to use in a variety of applications such as communications or digital signal processing.

1.5 Motivation

Power system applications require complex iterative operations on huge matrices, often on the order of 3000x3000 operations. Conventional machines consume 8-10 hours in computing several power-flow analysis problems. These complex iterative operations are often carried out using conventional software or hardware approaches. Software

approaches apply a set of instructions to map an application algorithm to a general-purpose computer. Since general-purpose computers are not implemented for special algorithms, they do not often yield the best possible performance. Hardware approaches are specific to the application and produce outstanding results. Recent studies in power engineering suggest that parallel architectures specific to the application reduce the computational times drastically. This led us to the development of an application-specific parallel architectural solution for power-flow analysis problems.

The SIMD mode of computation can speed up the implementation of algorithms for solving linear systems of equations, like LU decomposition and iterative techniques for matrix inversion. The ability of SIMD architectures to manipulate large vectors in minimal time has been demonstrated on existing SIMD machines. Most algorithms consist of code that is rarely executed, so attempting to map the entire program into the hardware logic will give us an inefficient solution. As power system applications have a combination of rarely executed code as well as code to manipulate large matrices in real time, we have chosen an SIMD architecture where an array of processing elements that carries out the most critical computations are connected to a host.

FPGAs have provided an alternative method to computing by supporting the fine-tuning of hardware to match software requirements. The fact that the number of system gates in FPGAs has been increasing rapidly in recent years encourages the development of large–scale application-specific custom computing machines on FPGAs for better hardware performance. While these FPGA-based Custom Computing Machines (CCMs) may not challenge the performance of microprocessors for all applications, for specific

applications an FPGA-based system can offer extremely high performance. This led us to develop an FPGA-based SIMD machine for power system analysis computations.

Nevertheless, CCMs that integrate processors and FPGAs suffer from slow interfaces between the general-purpose processor and the FPGAs, demonstrating the need for higher bandwidth communications between these resources. The advent of soft- core processors made it possible to integrate reconfigurable logic as a slave to its peripheral bus. Integrating a soft core processor, reconfigurable logic and memory into a single system allows us to achieve high performance. This led us to develop an on-chip FPGA-based SIMD machine that uses a soft-core processor as the host and an SIMD array of processing elements as a slave attached to its peripheral bus.

1.6 Objectives

The main objective of this thesis is to design an application-specific SIMD architecture for power-flow analysis problems and implement it on an FPGA. This thesis also aims at implementing a single-processor with a floating-point co-processor on an FPGA using the Xilinx Microblaze soft-core processor, and later comparing the performance of the SIMD machine with the sequential implementation.

These objectives will be addressed in this thesis by proposing an SIMD architecture. Implementation details are reported and the matrix multiplication application is mapped on to both the single-processor system and the SIMD machine to evaluate their performance. The target system is the Annapolis Micro systems (AMS) Wildstar-II development board that contains two FPGAs.

CHAPTER 2

DESIGN OF AN FPGA-BASED SIMD MACHINE

2.1 Overview of the Wildstar-II Board

The Annapolis Microsystems high-performance Wildstar-II Board combines the high density of reconfigurable system gates from Xilinxs Virtex-II FPGAs with very large memory and high I/O bandwidth. We chose the PCI-based Wildstar-II board as its two XC2V600 Virtex-II FPGAs can deliver great levels of processing power, and its substantial on-board DDR or DDR-II SRAM and DDR DRAM memories make it an ideal choice for building custom computing machines.

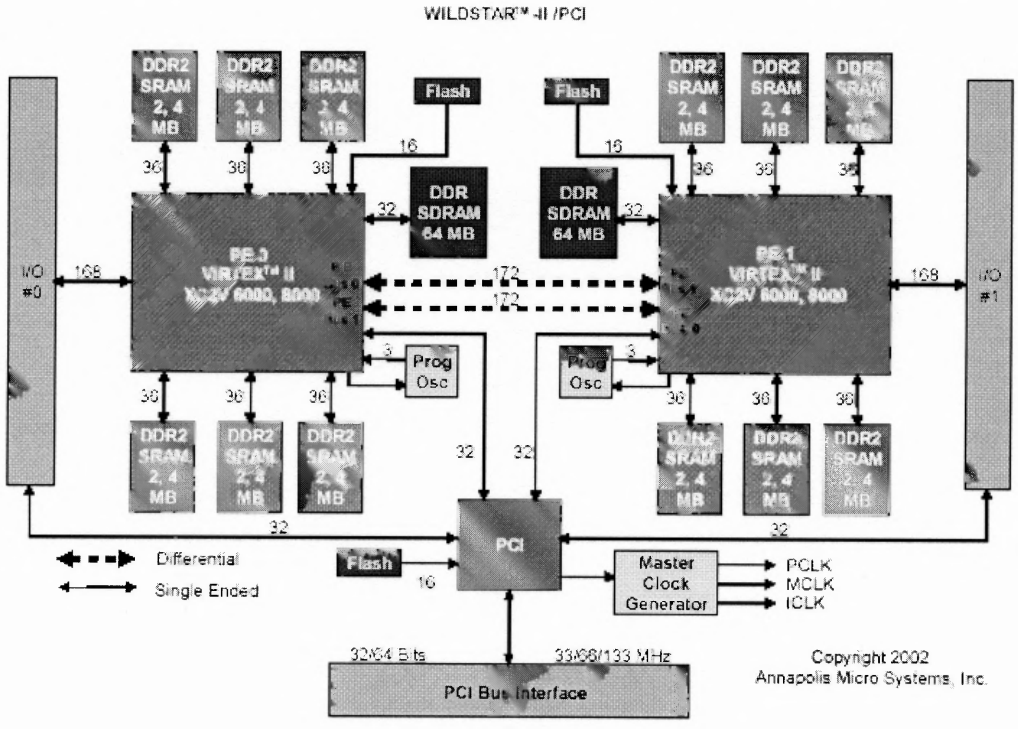

Figure 2.1 Wildstar-II/PCI Block Diagram [**14**].

Figure 2.1 shows the block diagram of the Annapolis Microsystems's Wildstar-II /PCI board. It uses two Xilinx XC2V6000 FPGAs, with up to 16 million system gates each. A host computer can communicate with the board via the PCI interface. The PCI bus interface communicates with the Wildstar-II board's PCI controller. The PCI controller has access to the FPGAs and Euro I/O cards using the Local Address Data (LAD) bus. The host has direct register access and communicates with the FPGAs and the I/O cards over the LAD bus. It has 12Mbytes of DDR II SRAM and 128MB of DDR SDRAM on the board. It has programmable Flash bank per FPGA for image storage.

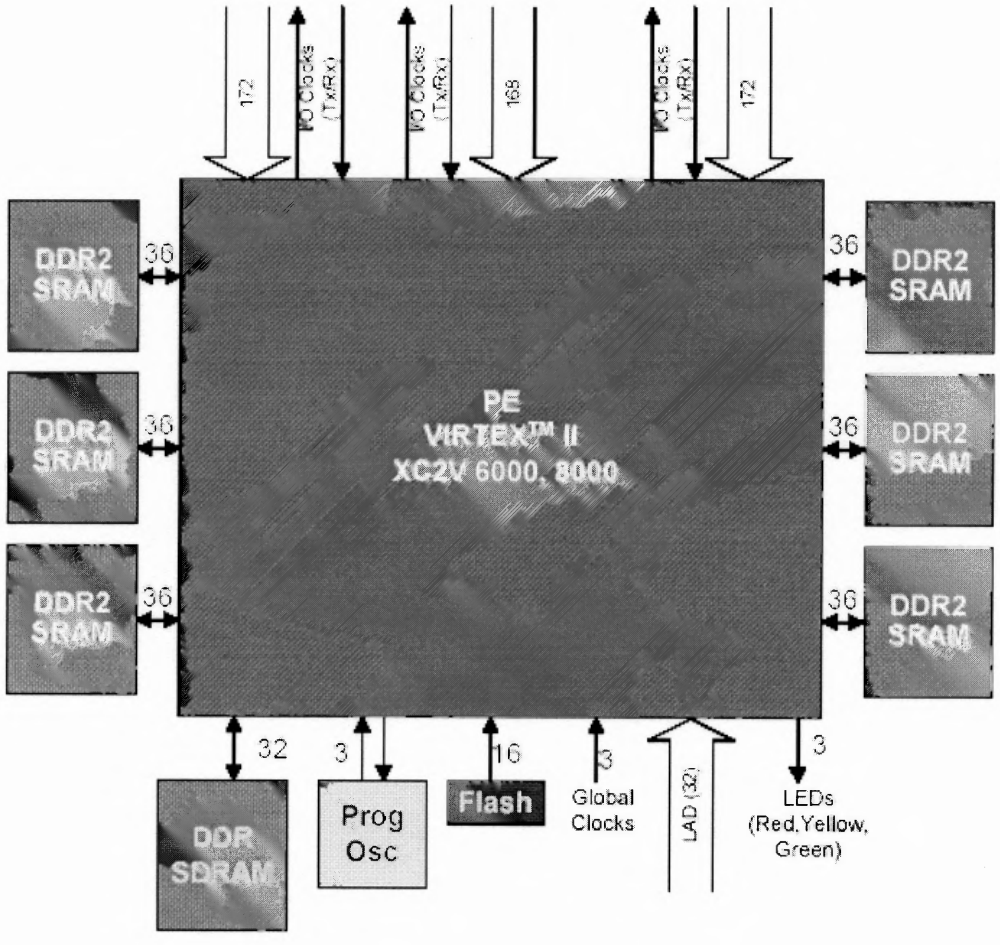

Figure 2.2 Wildstar-II Processing Module [**14**].

Each processing module consists of a Xilinx Virtex-II FPGA, six independent DDR2 SRAM ports, one bulk DDR DRAM port, three input/output Transmit(Tx) And Receive(Rx) clocks, a 32-bit LAD bus, Flash storage for multiple FPGA images, three global clocks, three user clocks and three user LEDs. The Wildstar-II board has two types of clocks: the global board clocks MCLK, PCLK and ICLK and the local clocks for each FPGA consisting of ACLK, BCLK and CCLK. MCLK is differential and asynchronous to PCLK. It is configurable through the Wildstar-II host software. PCLK is differential and asynchronous to MCLK, and is configurable through the Wildstar-II host software. ICLK is the Local Address Data Bus clock. It is fixed at 132MHz and the FPGA uses this clock to interface to the PCI controller for host access via the LAD bus.

The host communicates with the board using Wildstar-II Application Programming Interfaces (APIs). The host software includes Wildstar-II APIs, device drivers, a run time library and utilities which enable efficient communication between the host and the board through the PCI bus. The Application Programming Interface is a set of functions coded in the C language allowing communication between an application and the Wildstar-II run-time library. Many APIs are provided to open the board, close the board, program the FPGAs, deprogram the FPGAs, write onto them and read from them.

2.2 Overview of the Microblaze Soft-Core Processor

Microblaze embedded systems comprise the Microblaze soft-core processor, on-chip block RAM, standard bus interconnects and on-chip peripheral bus peripherals. Microblaze systems can range from a simple processor core with a minimum local memory to a large system with many Microblaze processors or many OPB peripherals.

The Microblaze embedded soft-core was chosen as it is a 32-bit Reduced Instruction Set Computer (RISC) processor optimized for implementation in Xilinx FPGAs. The Microblaze soft core processor has the following features:

1. Thirty-two 32-bit general purpose registers.
2. 32-bit instruction words with three operands and two addressing modes.
3. Separate 32-bit instruction and data buses (Harvard Architecture) that conform to IBM's OPB (On-chip Peripheral Bus) specifications.
4. Separate 32-bit instruction and data buses with direct connections to on-chip block RAM through a Local Memory Bus (LMB).
5. 32-bit address bus.
6. Single issue pipeline.
7. Hardware multiplier in Virtex-II.

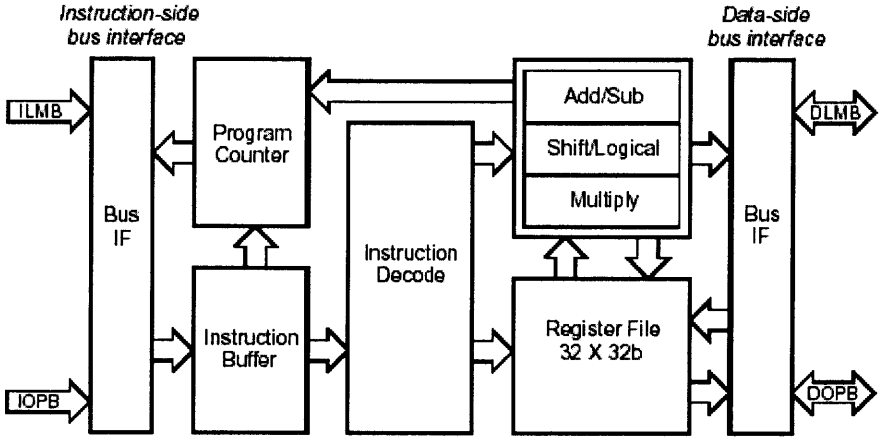

Figure 2.3 Microblaze Core Block Diagram [13].

Microblaze is a big-endian processor. The Microblaze core is organized as a Harvard architecture with separate bus interface units for data and instruction accesses. Each bus interface unit is further split into a Local Memory Bus (LMB) and IBM's On-

chip Peripheral Bus (OPB). The LMB provides single cycle accesses to on-chip dual-port block RAM. The OPB interface provides a connection to both on-chip and off-chip peripherals and memory. Microblaze bus interfaces are available in six different configurations as shown below:

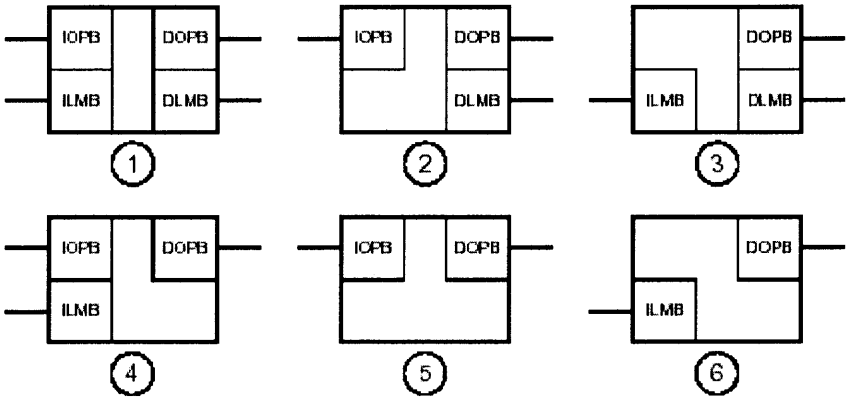

Figure 2.4 Microblaze Bus Configurations [**13**].

The optimal configurations are chosen from the above set of configurations depending on the code spaces and data spaces, and if fast accesses to both are required.

2.2.1 Overview of the OPB Bus

The OPB is one element of IBM's Core Connect architecture, and is a general-purpose synchronous bus designed for easy connection of on-chip peripherals. OPB has the following features:

1. 32-bit or 64-bit data bus.
2. Up to 64-bit address.
3. Supports 8-bit, 16-bit, 32-bit and 64-bit slaves.
4. Supports 32-bit and 64-bit masters.
5. Dynamic bus sizing with byte, halfword, fullword, and double word transfers.

6. Optional Byte Enable support.
7. Distributed multiplexer bus instead of 3-state drivers.
8. Single cycle transfers between OPT masters and OPB slaves.
9. Support for sequential address protocol.
10. 16-cycle bus-time out.
11. Slave time out suppress capability
12. Support for multiple OPB masters

Most of these features are well mapped onto FPGAs; however, some features can result in inefficient use of FPGA resources or can lower the system clock rates. Xilinx uses an efficient subset of OPB for Xilinx developed OPB devices.

2.2.2 Microblaze Software Support

The Microblaze software support consists of a complete GNU C compiler and binary utilities of tool suites. These suites allow users to compile, assemble and link their C code or Microblaze assembly programs. The compiler's code optimizer and code generator have been customized to achieve the best possible performance for applications with the Microblaze ISA.

2.3 Sequential Architecture

The sequential architecture consists of a Microblaze embedded processor with local memory, where the instruction bus is connected to only the Local Memory Bus (LMB) while the data bus is connected to both the LMB and OPB buses. This sequential architecture is designed for implementation on the Wildstar II board's Xilinx XC2V6000 FPGA. The sequential architecture also consists of a global memory as a custom

peripheral to the OPB bus of Microblaze. This global memory enables the host to communicate with the Microblaze embedded system on the Wildstar-II board. This global memory consists of a dual-port RAM (DPM) to store the results produced from

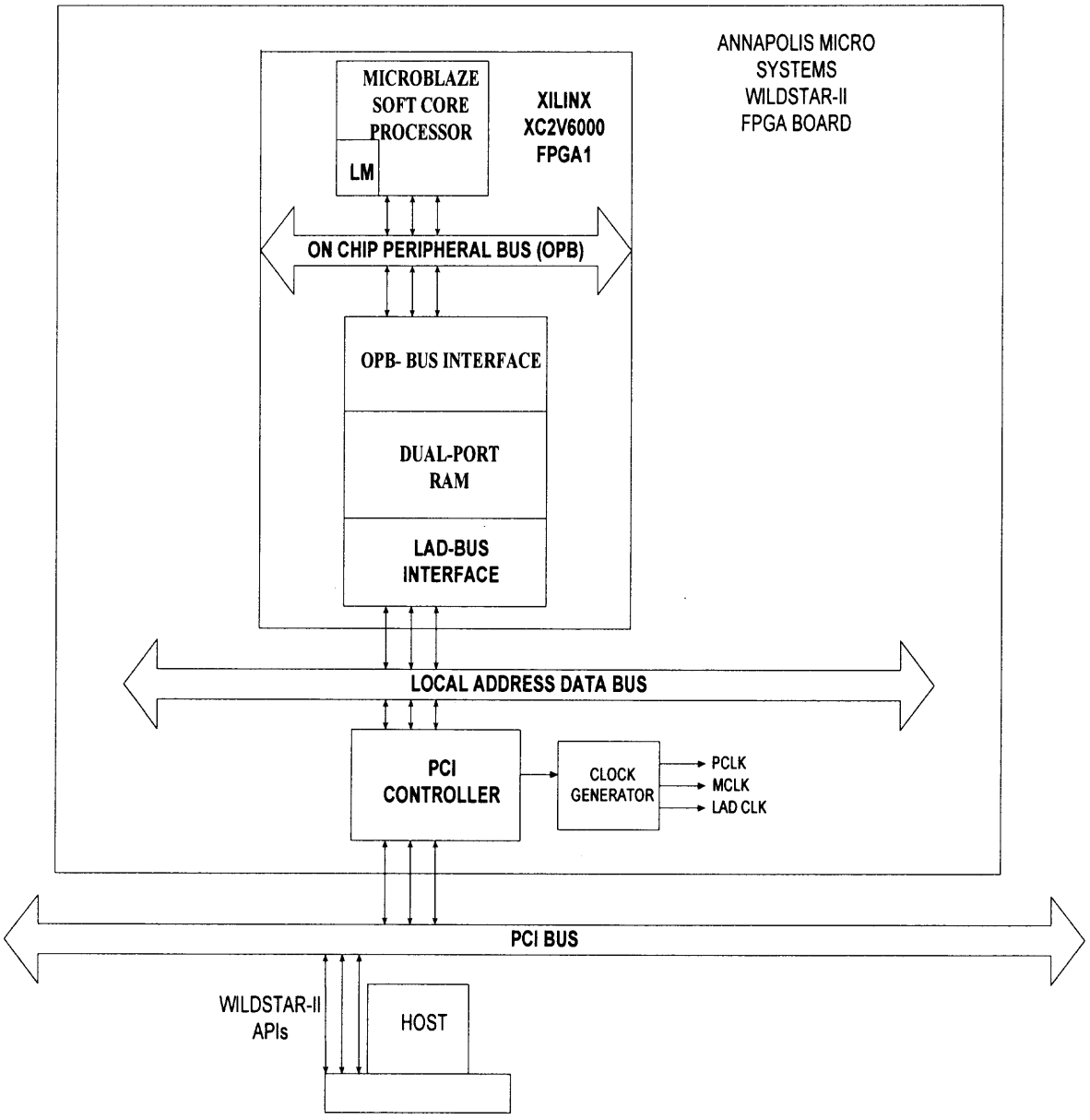

Figure 2.5 Sequential Architecture of Microblaze.

Microblaze, glue logic to interface the ports on one side of DPM with the OPB bus, enabling the OPB bus to read and write, and glue logic to interface the ports on the other side of DPM with the LAD bus, enabling the host to read the results stored in DPM through the PCI bus controller using the Local Address Data Bus. One side of the global memory is connected to the data side OPB of the Microblaze embedded processor as seen in Figure 2.6.

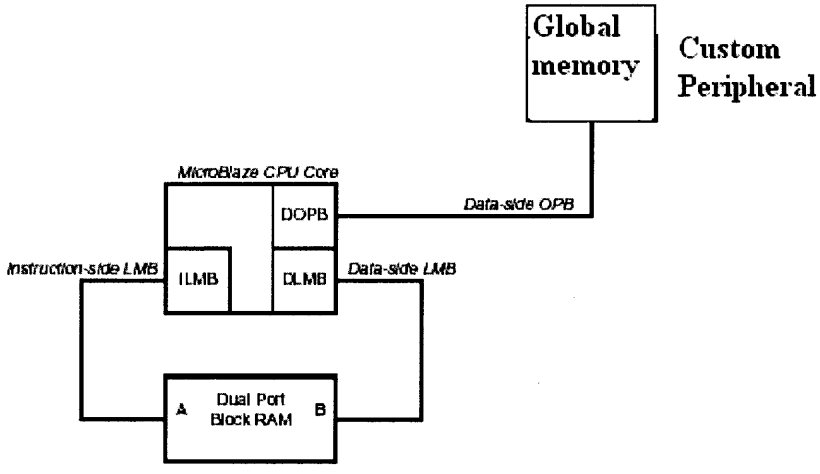

Figure 2.6 Global Memory as Custom Peripheral.

The sequential algorithms were written in C and built into the local memory while the Microblaze embedded system was built using the Microblaze Development Kit tools (MDK). The final X86 or bit file for the Microblaze system containing the sequential code is generated using the MDK tools. An application code is written using the Wildstar-II APIs to program the FPGA with the X86 file, to program the clocks and to read the results from the board into the host. These APIs are compiled on the host using Wildstar-II host software and the executable is run to program the FPGA and to read back the results into the host. The sequential architecture can run floating-point

operations using software support as Microblaze does not have a floating-point unit. The results obtained from the sequential implementation of LU decomposition and matrix multiplication applications for different matrices on the sequential architecture are discussed in the Implementation Chapter.

2.4 SIMD Architecture

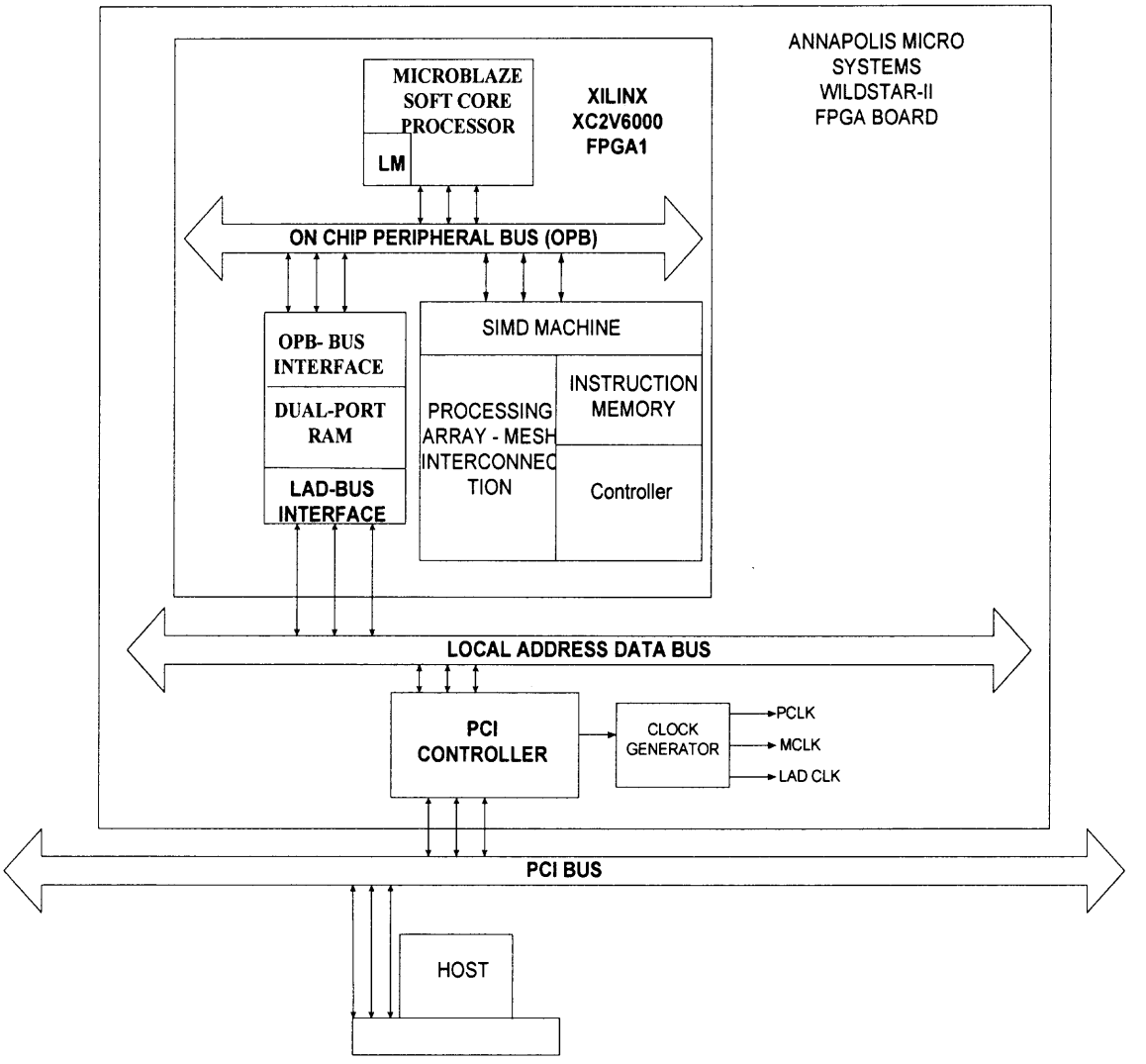

Figure 2.7 SIMD Architecture.

The SIMD architecture was developed around the Microblaze embedded system using the OPB bus on the Wildstar-II board with Xilinx XC2V6000 FPGAs. Figure 2.7 shows the proposed architecture for the FPGA-based SIMD implementation using a soft-core Microblaze processor as the host. The FPGA-based SIMD machine comprises an instruction memory connected to the OPB bus, the controller, the mesh-connected processing array, the global memory with an interface to both the OPB and LAD buses, and a Microblaze as the host processor. The processing array consists of a 3X3 mesh of

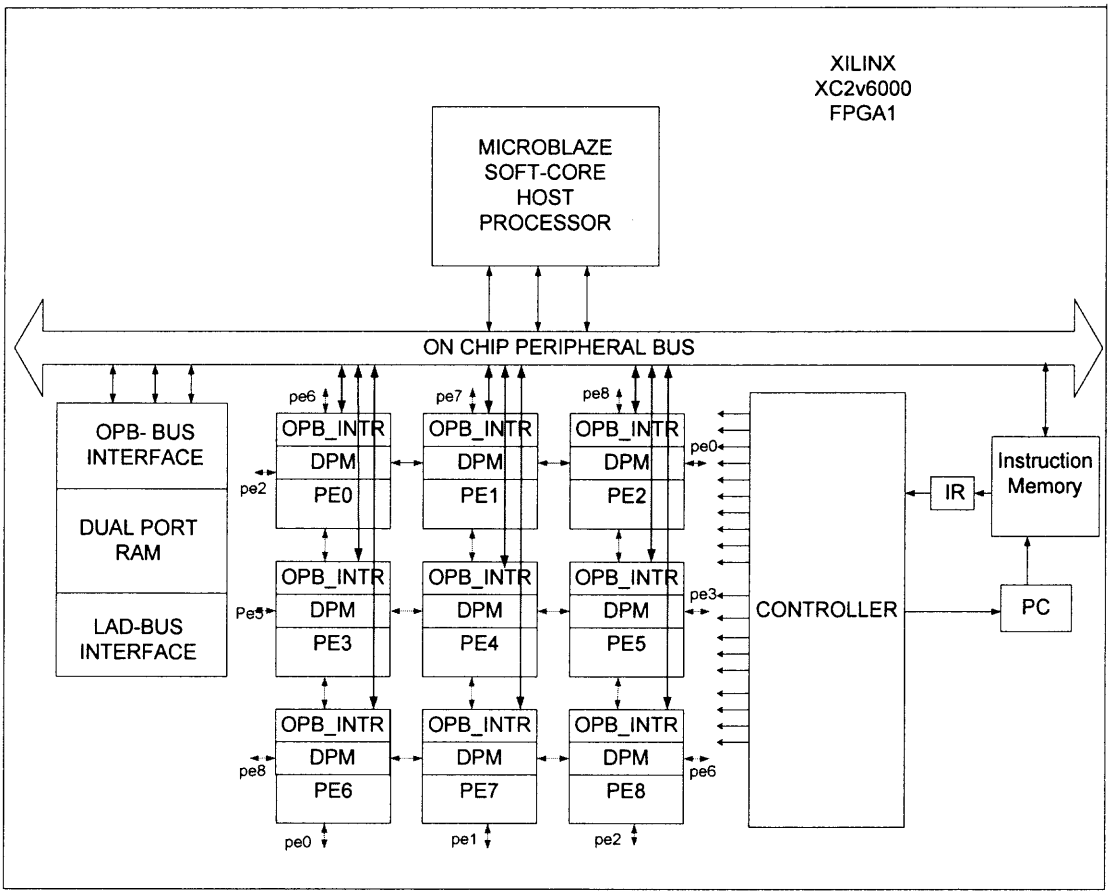

Figure 2.8 Architecture of the SIMD Machine.

processing elements (PEs) with each processing element attached to one side of a dual-ported RAM that serves as local memory; it also contains glue logic to interface the other side of the dual-ported RAM with the OPB bus, enabling the OPB bus to read and write. Finally, mesh connections enable communications with neighboring PEs in the North, South, East and West directions. The instruction memory is a dual-ported RAM; one of the ports is connected to the OPB bus enabling OPB read and write operations, and the other ports are connected to the controller. The SIMD instructions are stored in the instruction memory. The controller fetches each instruction from the instruction memory, decodes it and broadcasts the control signals to all the PEs. All the control signals reach the PEs using hardwired connection as the fanout of each FPGA is big enough. The Microblaze soft-core processor acts as a host to the SIMD machine. The SIMD processing array and controller are connected to Microblaze through its Peripheral bus OPB. The Microblaze can perform read and write operations on the PEs through the OPB bus as though each PE is a single memory unit. The Microblaze loads the data into each of the PE's data memory, loads the SIMD instructions into the instruction memory and reads back the results from the processing array through the OPB bus. The global memory is a dual-ported memory with one end interfacing the OPB bus and other end of the dual-ported RAM interfacing the LAD bus enabling the host PC to read the results from the board through the PCI controller.

Microblaze reads back the results from the PEs and stores them in the global memory that has the LAD bus interface, so that the host PC can read the results from the board. For exploiting parallelism, along with the parallel architecture we should also have a parallel compiler for efficiently mapping the algorithms onto the parallel architecture.

But developing a compiler is a Herculean task. So, for this machine the parallel pieces in the algorithms are found and the corresponding instructions are stored in the instruction memory by the Microblaze. So, the Microblaze C code consists of a sequential part that runs on the host Microblaze and a parallel part written as store operations into the instruction memory of the SIMD machine. Once the instructions are stored in the instruction memory, the controller fetches the instructions, decodes them and broadcasts the control signals to the PEs. All PEs execute the same instruction simultaneously and the results are read back by the Microblaze. The Microblaze stores those results in the global memory, so that the host PC reads the results from the board using the Wildstar-II APIs. The important property of this architecture is its scalability which lies primarily in the architecture of the PE.

2.4.1 Architecture of the Processing Element

The processing array consists of a 3X3 mesh of PEs attached as a slave to the peripheral bus of the Microblaze processor. Each PE is a processor without a control unit and also contains local memory. Each PE contains a dual-ported memory connected both to the internal data bus of the PE and to the global bus OPB. It allows the Microblaze to load the initial data into the PE's local memory from the global memory and to read the results from the PE's memory after all computations have been implemented. The PEs are connected to neighboring PEs in the north, South, East and West directions using mesh interconnections. All main operations in power flow analysis are floating-point operations, so the PE should have an efficient floating -point unit. PEs should have the capability to execute load, store, floating- point addition, floating-point subtraction, floating-point multiplication and floating-point division and routing operations. The

interconnection between the PEs is register based using full- duplex connections. The main advantage of the PE architecture is its scalability that allows to implement as many PEs as needed.

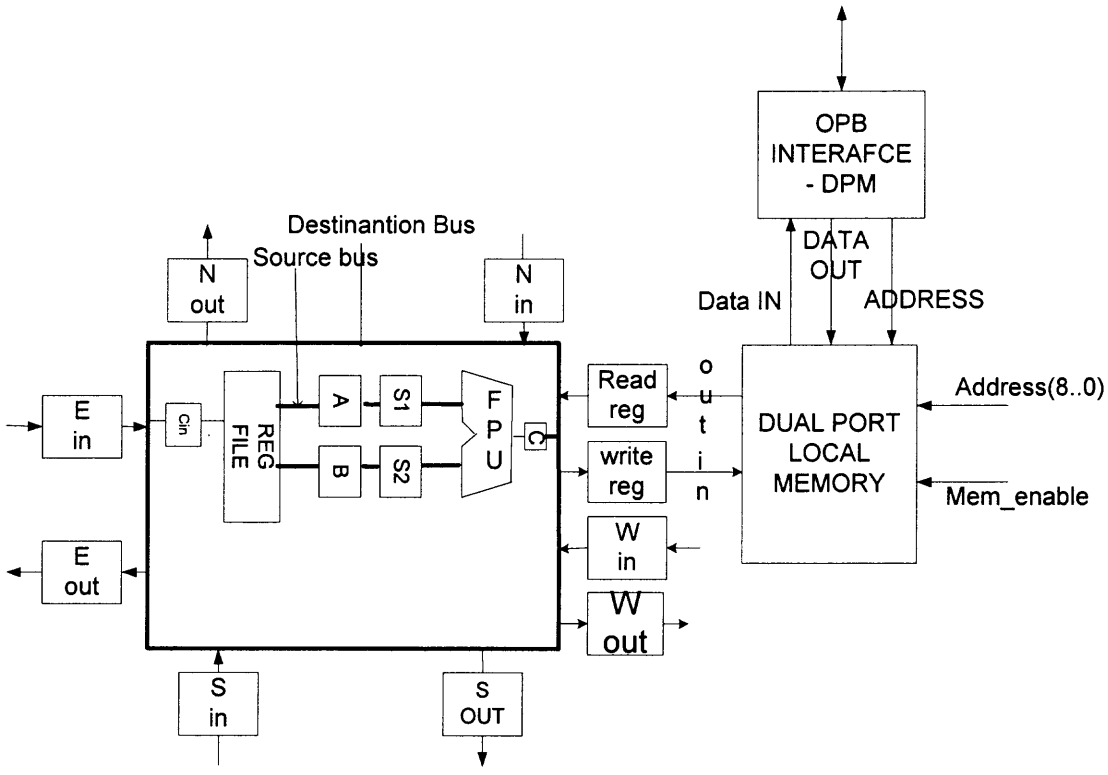

Figure 2.9 Architecture of the PE.

The PE uses a register file with 16 32-bit registers. It has two local buses, one for the source operands and the other for the destination bus. Internal PE communications use 32-bit local buses. The data on the destination bus can be transferred to memory, register file or can be routed to neighboring PEs. All of these destinations may be specified in the instruction. It also has eight registers connected to the destination bus for the interconnection of PEs. This register-based interconnection enables the data on the destination bus to be routed to any of the neighboring PEs in the North, South, East and

West directions, as specified by the instruction. The main computation part of the PE is the floating-point unit (FPU), which turns out to be the most critical path in the SIMD design. The PE has a FPU which is intended primarily to implement floating-point operations or pass data onto the destination bus for storing back into the memory or for data routing. The FPU conforms to the IEEE 745 single-precision floating-point format.

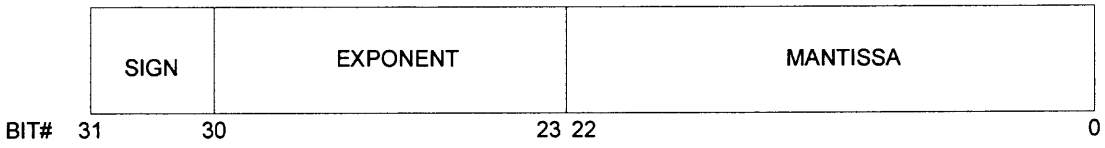

Figure 2.10 Floating-Point Data Format.

According to the IEEE-754 floating-point format, data is represented with 32 bits, where bit# 31 is used as the sign bit, bit#30 to bit#23 (8 bits) are used as the exponent, and bit #23 to bit#0 and an implied '1' (24 bits) are used as the mantissa. The FPU employs a three-stage pipelined floating-Point adder, subtractor and multiplier, and a 28-stage pipelined divider. The sequential implementations of these algorithms give very poor performance on FPGAs. So, a pipelined architecture was developed for the floating-point adder, subtractor, multiplier and divider to improve the performance on the FPGAs.

2.4.1.1 Pipelined Floating-Point Adder/Subtractor. The floating-point addition and subtraction algorithms [7] are similar and are performed in three stages.

Stage 1:

- The sign, exponent and mantissa with their implied bits are extracted from the floating-point numbers in the input.

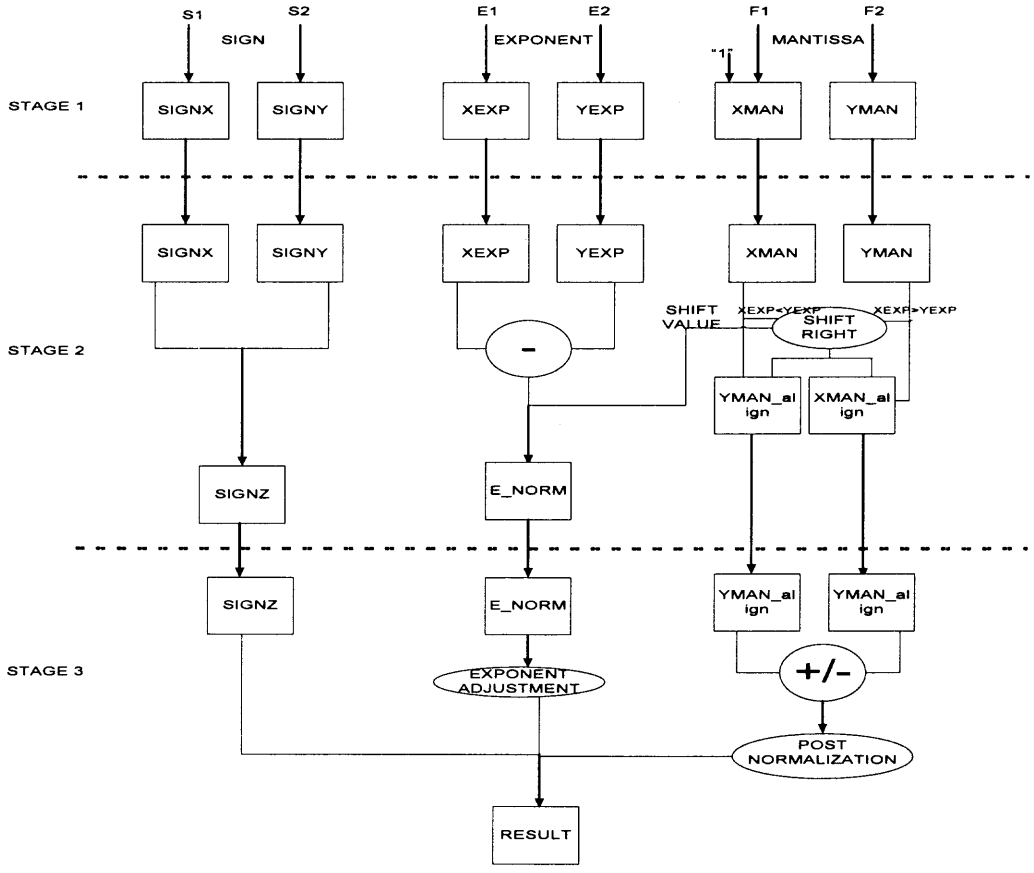

Figure 2.11 3-stage Pipelined Floating-Point Adder/Subtractor.

Stage 2:

- The exponents are compared and checked to identify the larger value, and an appropriate shift value is calculated.
- If the exponent of data x is less than the exponent of data y, then the mantissa of x is shifted right by the shift value, or else the mantissa of data y is shifted right by the shift value.
- The exponents are adjusted according to the shift value.

Stage 3:

- The aligned mantissa is added or subtracted and then post normalized.
- The exponent is adjusted to perform post normalization.
- The result is obtained by combining the sign bit, the mantissa and the exponent.

2.4.1.2 Pipelined Floating-Point Multiplier. The floating-point multiplier [7] is a three stage pipelined multiplier.

Stage 1:

- The sigh, exponent and mantissa with the implied '1' bit of the two numbers are extracted.

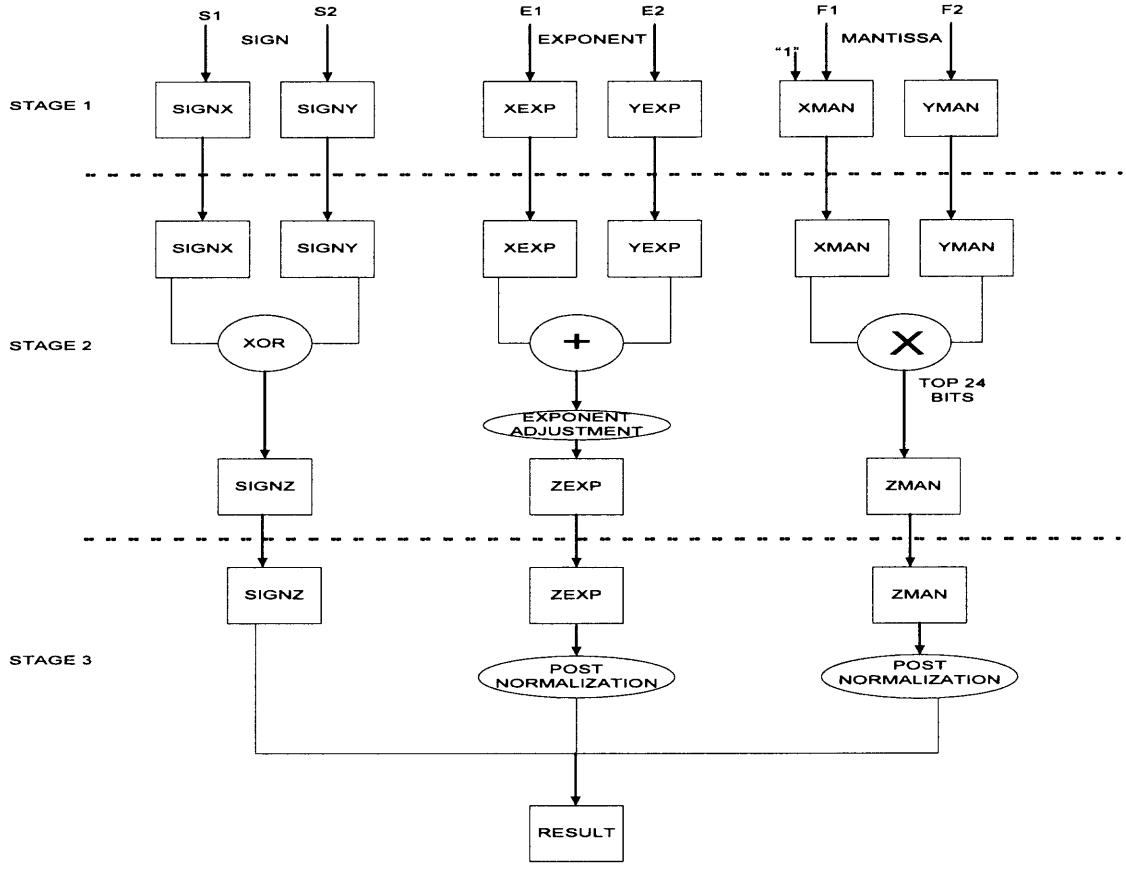

Figure 2.12 3-stage Pipelined Floating-Point Multiplier.

Stage 2:

- The resultant sign bit is calculated by doing a xor of the sign bit in the data x and the sign bit in the data y.
- The exponent of x and y are added and adjusted.
- The mantissas of X and Y are multiplied and the upper 24 bits are taken.

Stage 3:

- The post normalization for the exponent and mantissa is done.
- The result is obtained by combining the sign bit, the exponent and the mantissa.

2.4.1.3 Pipelined Floating-Point Divider. The floating-point divider uses the SRT non restoring division algorithm [**5**]. The sequential implementation of this algorithm gives a very poor performance on FPGAs. A VHDL technique called "unrolling of the for loops" was used in developing a 28-stage pipelined architecture. The iterations of a loop are unrolled into 24 stages and they are replaced using simple if statements.

The PE is connected to a 2KB dual-ported RAM using read and write registers as local memory. These dual-ported RAMs are connected both to the local data bus (destination bus in the PE) and global data bus (OPB bus). This enables the host to load the data into the local memory and to read the data from the local memory. This also gets the addresses from the host as well as from the sequencer of the SIMD machine.

2.4.2 Instruction Set

Two types of instruction formats were developed using the register-register instruction and immediate-type instruction types. The register-register instruction format uses 6 bits for the opcode, 5 bits each for the address of the source registers, 5 bits for the destination register and 11 bits for masking. The immediate-type instruction format uses 6 bits for

the opcode, 10 bits for the immediate address, 5 bits for the destination register, and 11 bits for masking.

Opcode (6bit)	Source1 (5bit)	Source2 (5bit)	Destination (5bit)	Mask bits (11bit)

Figure 2.13 Register-register type instruction format.

Opcode (6bit)	Immediate address (10bit)	Register (5bit)	Mask bits (11bit)

Figure 2.14 Immediate-type instruction format.

The opcode field, which is 6 bits long, specifies the operation for the PE, and the 5 bit addresses of the source and destination registers specify the addresses of the source and destination operands, respectively. The memory address field specifies the address of the memory to load from or store into the memory. The 11 bits for enabling the PEs are used to mask a subset of the PEs when all of the PEs are not needed to be active for some of the instructions.

FPU Operations:

1. Instruction: add/sub/mul/div R_d, R_{s1}, R_{s2}

   This instruction performs addition, subtraction, multiplication or division of two floating-point numbers. The source registers are R_{s1}, R_{s2} and the result is stored into the destination register.

2. Instruction: nr/er/wr/sr R_d

This instruction performs routing, where the contents received from the north/east/west/south is stored into the destination register R_d.

3. Instruction: ns/es/ws/ss R_{s1}

This instruction performs routing, where the contents of the source register R_{s1} is sent to the north/east/west/south processing element via the north/east/west/south out registers.

4. Instruction: LW Rd, mem(x)

This instruction loads the contents of the memory location with address x into the destination register R_d.

5. Instruction: sw mem(x), R_d

This instruction stores the contents of the register R_d into the memory address x.

2.4.3 Controller

The controller serves as the sequencer of the SIMD machine. The host issues instructions to the controller by storing them in the instruction buffer. The sequencer fetches instructions from the instruction buffer and decodes them using its state machine. There are six states in the sequencer: instruction fetch, instruction decode, instruction execute, load memory, store memory and writeback. The state machine is shown in Figure 2.16. This state machine shows the different states that the sequencer goes through for fetching, decoding and executing the instruction. The execution state has many sub-states in it. Whenever the machine is reset, the sequencer fetches the instruction from the instruction buffer, increments the program counter, decodes the instruction and sends the control signals to all the PEs to execute the instruction. The connection between the PEs and the

sequencer is hardwired. All the control signals are connected to the PEs and, according to the instruction, either all PEs or some of them execute the instruction simultaneously.

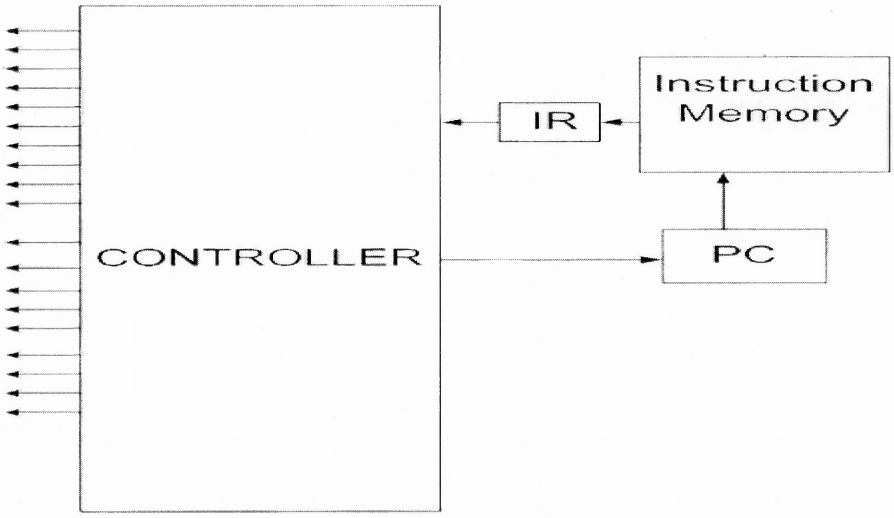

Figure 2.15 Controller of the SIMD machine.

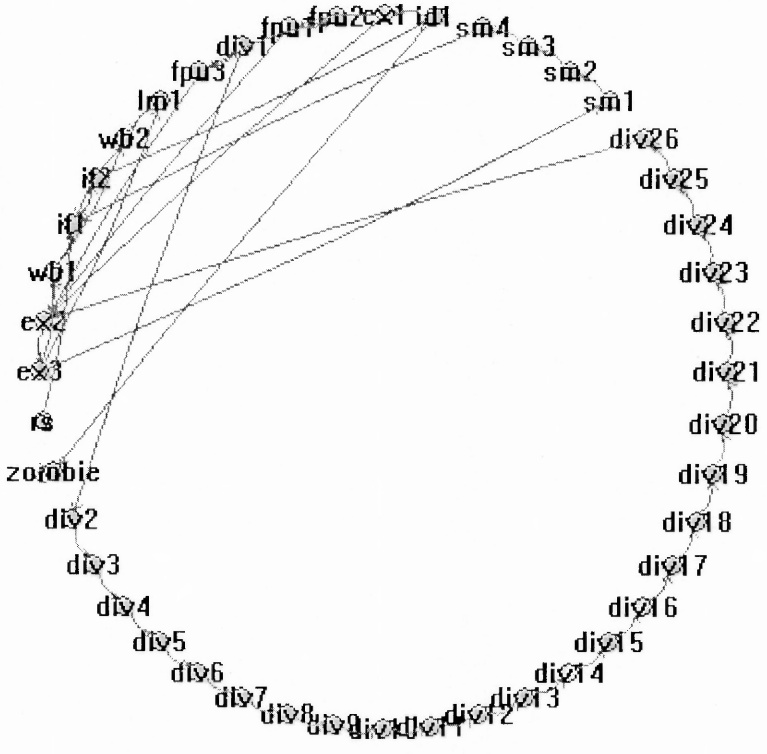

Figure 2.16 State machine of the controller.

2.4.4 Global Memory

This is a dual-ported memory that has one end connected to the glue logic that interfaces the OPB bus and the other end connected to the glue logic that interfaces the LAD bus. Microblaze reads the results from the processing array and the final results after computations are stored in the global memory. This enables the host PC to read the results from the board, using the Wildstar-II APIs, through the PCI controller that interfaces the LAD bus.

The algorithms used for sequential applications should be modified so that they are efficiently mapped on to the SIMD machine. Once the algorithms for the SIMD machine are developed, then the code is written in C for Microblaze; it contains instructions to store the SIMD instructions in to the instruction memory. The C code is built into the Microblaze system using the Microblaze Development Tools. An application code is written for the host PC using the Wildstar-II APIs to program the FPGA, program the clock, and read back the results. The implementation details for the SIMD machine are discussed in the Implementation Chapter.

2.5 Sequential Architecture with the Floating-Point Co-Processor

Microblaze works on integer numbers only since it does not have a hardware FPU.The floating-point co-processor is designed to enable floating-point operations with Microblaze. The floating point co-processor consists of a single PE with an FPU, and a controller and instruction memory interfaced to Microblaze through the OPB bus. The design of the PE is similar to the design of a PE in the SIMD machine. This PE uses a FPU, a register file with 16 32-bit registers and a dual-ported local memory. The

controller design is also similar to the design of the controller in the SIMD machine. The global memory is a dual-ported memory used to interface both the OPB bus and LAD bus. The results are stored by the Microblaze after the computations in the global memory, and are read from the host. The architecture of the sequential Microblaze system with a floating-point co-processor is shown in Figure 2.17.

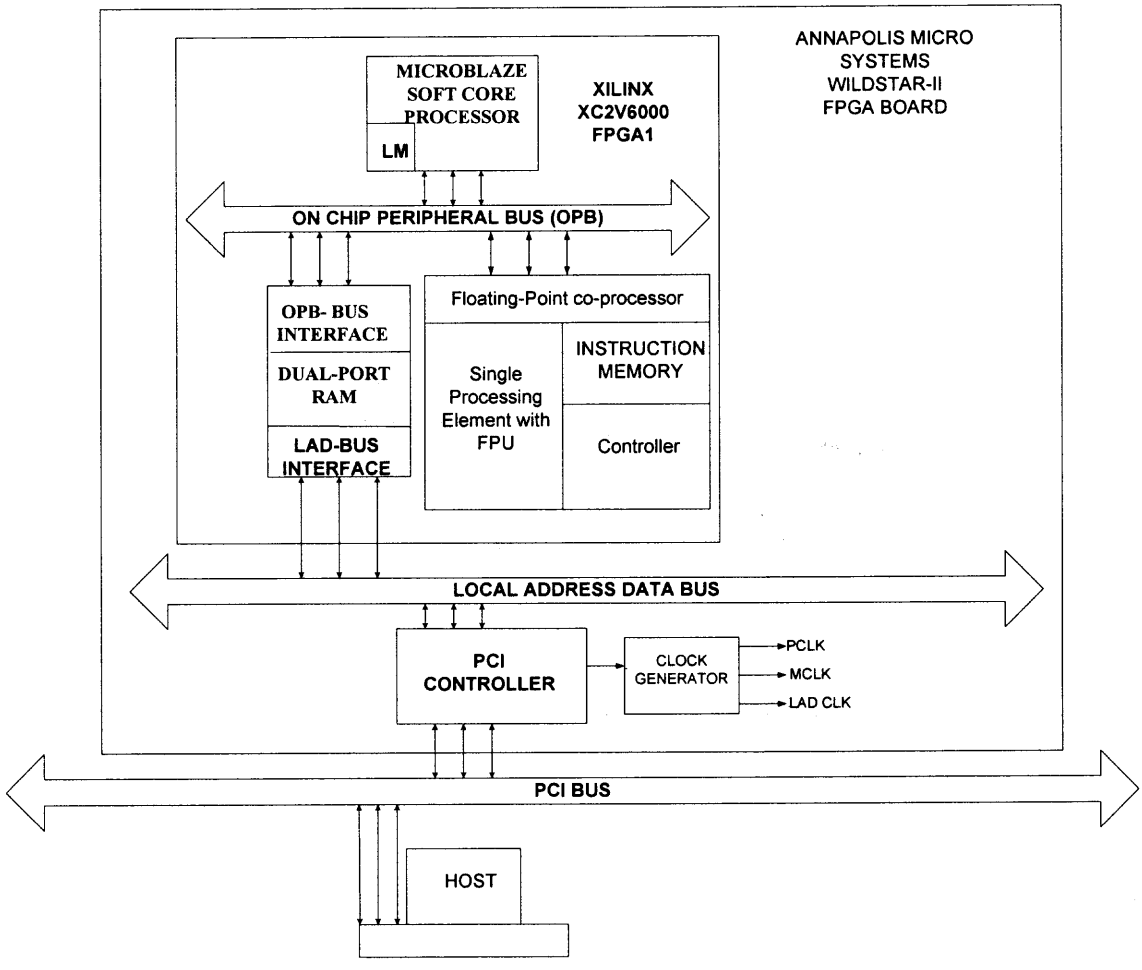

Figure 2.17 Floating-Point Co-Processor.

The applications are written in assembly language and the assembly instructions are stored in the instruction memory of the co-processor to test the floating point co-processor. The results are read from the host using Wildstar-II APIs. The performance results and comparisons with the SIMD machine are discussed in Chapter 3.

CHAPTER 3

IMPLEMENTATION AND RESULTS

The SIMD machine is designed using the hardware description language VHDL. The design of the processing array, controller, global memory and glue logic to interface the OPB and LAD buses are done in VHDL and integrated with the soft-core processor Microblaze as a custom peripheral. During this design, different tools at various levels of integration are used. We have followed two design flows in generating the complete system. Before integrating the SIMD machine or any custom peripheral with the Microblaze system, the FPGA design flow is followed and then the Microblaze design flow is used to generate the complete system.

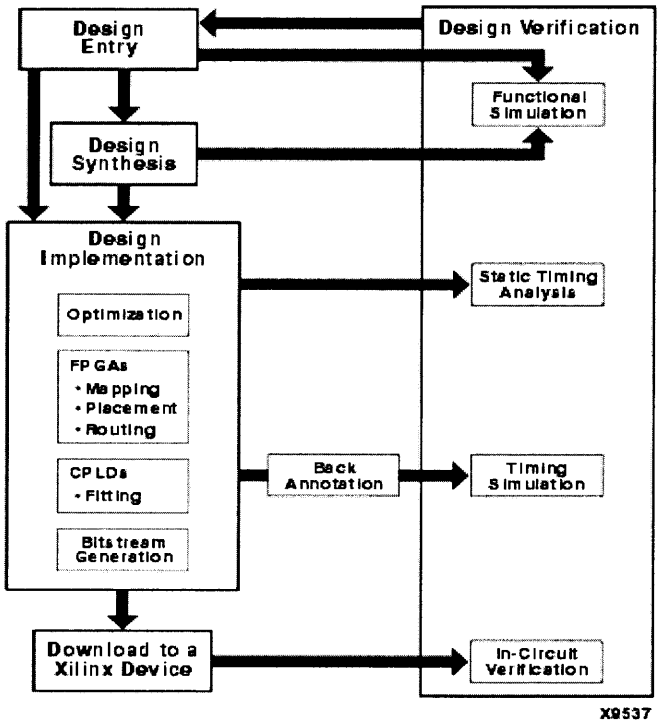

Figure 3.1 FPGA design flow [**12**].

The following are the steps followed in the FPGA design flow:

1. The design of all modules required for both SIMD and sequential implementation is done using a synthesizable subset of the VHDL language. The coding and compilation is done using the Mentor Graphics Modelsim simulator.

2. The functional simulation is performed using the Modelsim simulator. Many test benches were developed to test the SIMD machine using simulation. All the instructions for the SIMD machine were tested using test benches.

3. These VHDL files are given as input to the synthesis tool Synplify. During Synthesis the behavioral description in the HDL file is translated into a structural netlist and the design is optimized for the Xilinx device XC2V6000. This generates a netlist in the EDIF and VHDL formats.

4. The output VHDL file from the synthesis tool is used to verify the functionality by doing post synthesis simulation using the Modelsim simulator.

5. The netlist EDIF file is given to the implementation tools of the Xilinx ISE 4.1i. This step consists of translation, mapping, placing and routing, and bit stream generation. The design implementation begins with the mapping or fitting of the logical design file to a specific device, and is complete when the physical design is completely routed and a bitstream is generated. Timing and static simulations are done to verify the functionality. This tool generates an X86 file which is used to program the FPGA.

6. Finally, the results are obtained from the FPGA board and are checked for functionality.

All these steps are followed in a general design methodology to program the FPGA. In our design, the implementation steps in the FPGA design flow are carried out after the Microblaze development flow. All iterative steps till synthesis are done for verifying the SIMD machine. The EDIFs or HDLs generated after synthesis are used as inputs for the Microblaze design flow. Finally, implementation steps are followed to generate the final bitstream for the complete SIMD machine.

3.2 Microblaze Development Methodology

The sequential and parallel implementations are implemented using Microblaze and its development tools. Both implementations use the available netlists for Microblaze and the OPB bus, and the MDK tool flow to configure them according to the requirements of the parallel and sequential systems, respectively. This tool allows custom designed peripherals, both in the sequential and parallel implementations, to integrate and generate a complete Microblaze system by modifying the netlists accordingly. The Microblaze development tool involves the following steps:

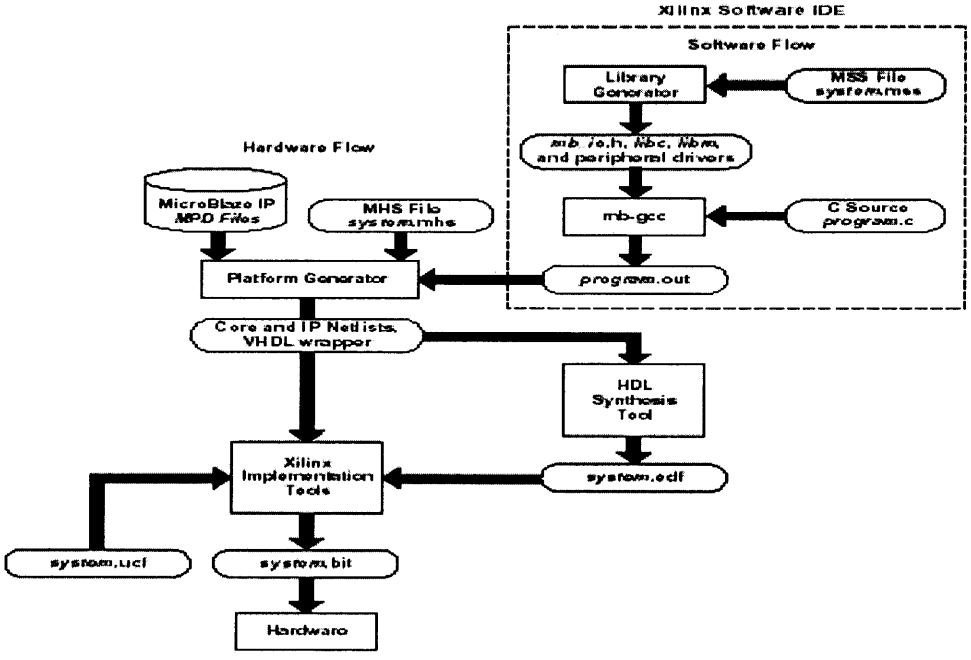

Figure 3.2 Microblaze design flow [13].

1. Define a Microblaze hardware system in a text based Microprocessor Hardware Specification (MHS) file. This file consists of peripherals, one of the six configurations of Microblaze, system connectivity and memory spacing for each of the peripherals. The MHS file for the sequential implementation consists of a Microblaze processor with configuration-3, global memory as peripheral and global memory address space as 8000-

81fff. The MHS file for the parallel implementation consists of the sequential components, the SIMD components and their memory address space. Also, we define the software specifications using a Microprocessor Software Specification File (MSS).

2. The hardware and software specification files are used to build the Microblaze system automatically involving the Library and Platform generators.

3. The Library Generator is used to build as system specific library of C functions that map basic C functions to peripherals; it also configures the custom c libraries. Then, the mb-gcc compiler is used to compile the source code. The C code is written for matrix multiplication and LU decomposition assuming different sizes of matrices; it is compiled using the mb-gcc compiler.

4. The executable from the mb-gcc compiler is given as input to the Platform Generator. It builds this executable into the Block RAM of Microblaze.

5. The Microblaze peripheral definition file, which contains the description of the custom peripheral, how it is connected to the OPB bus and the address space of the peripherals is also used as input to the Platform Generator to generate the complete system.

6. The Platform Generator uses the MHS and MPD files, and looks for all the IP core netlists, VHDL code for the custom peripherals and other libraries required for generating the system as specified in the Peripheral Analyze order file.

7. Finally, using this entire information the Platform Generator generates a top level HDL file that has all the information about the complete system.

8. This HDL file is synthesized in Synplify and a final netlist Electronic Data Interchange Information (EDIF) is formed.

9. After this step, the Xilinx implementation tools are used. The resulting EDIF file is used as input to the Xilinx Place and Route tool. This merges all the required netlists and produces an X86 file with which we can program the FPGA to realize the hardware.

Thus, the generated X86 file is used to program the FPGA present on the Wildstar-II board. An application code is written using the Wildstar-II APIs to program the FPGA with the X86 file, program the clock based on the timing reports and read back the results from the board. This application code is compiled and executed to read back the results into the host PC. The synthesis and performance results after running the applications are discussed below.

3.3 Sequential Implementation

3.3.1 Implementation

The sequential implementation uses the above discussed methodologies for implementations on the board. The custom peripheral with the global memory and the glue logic for interfacing the OPB and LAD buses are designed in VHDL. They are synthesized using Synplify and a netlist EDIF is generated. The Microblaze development tools are used as discussed above and a system top level HDL file is generated. This is synthesized using Synplify and implementation tools are used to generate an X86 file for the complete Microblaze system. The synthesis results are discussed below.

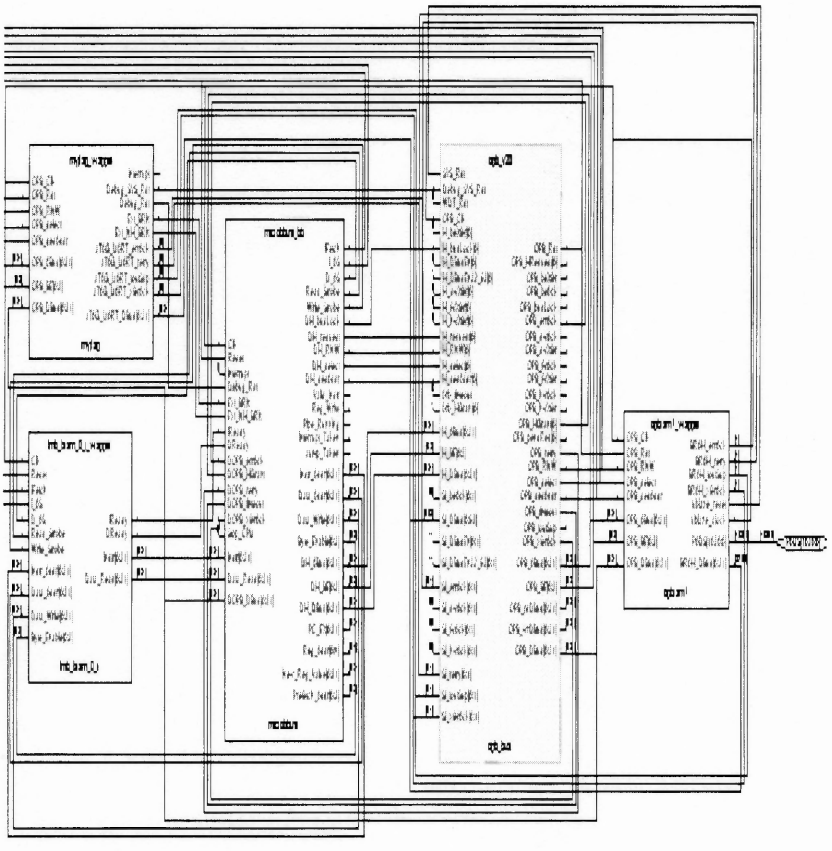

Figure 3.3 Synthesis RTL view of sequential implementation.

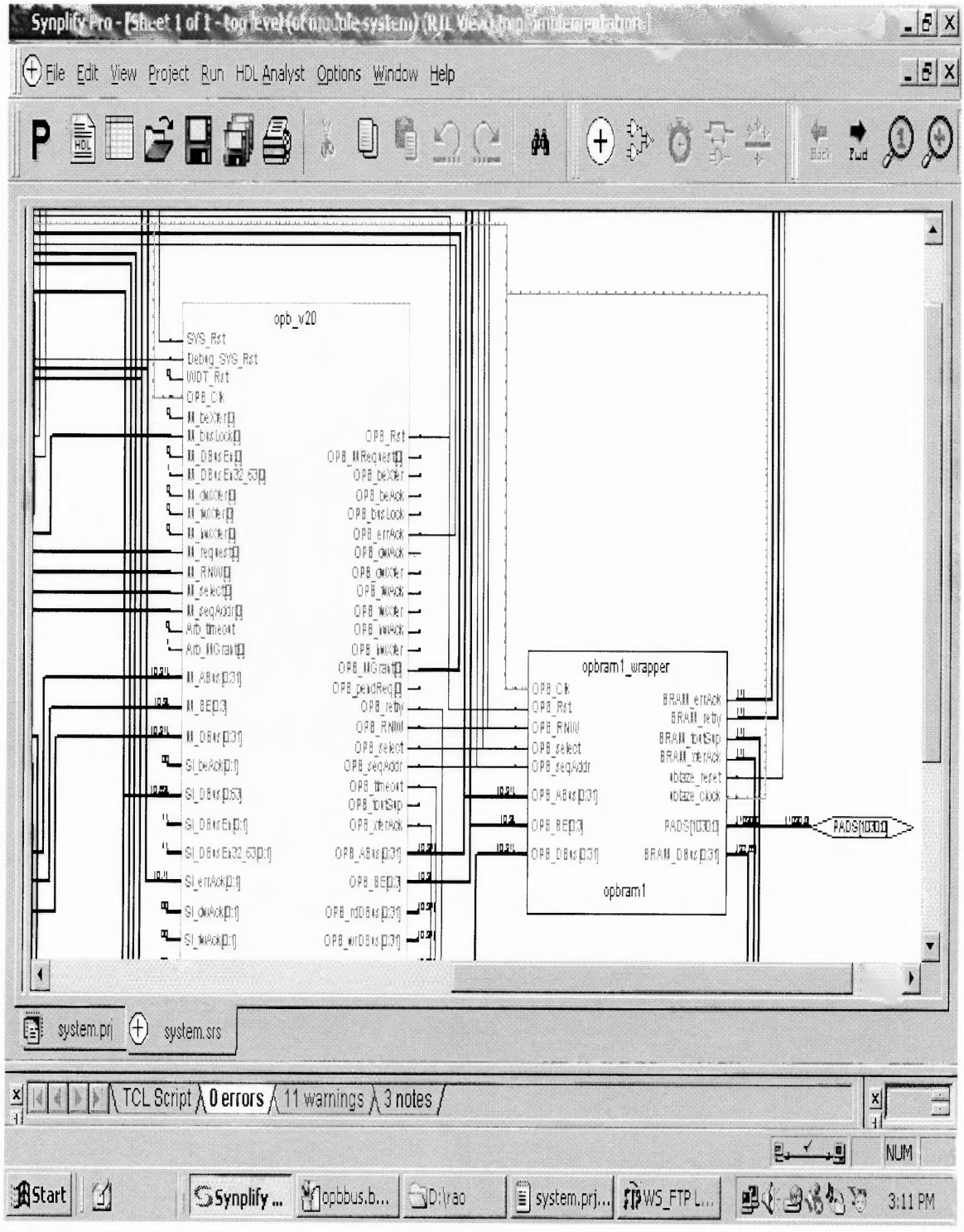

Figure 3.4 Custom peripheral connected to the OPB bus.

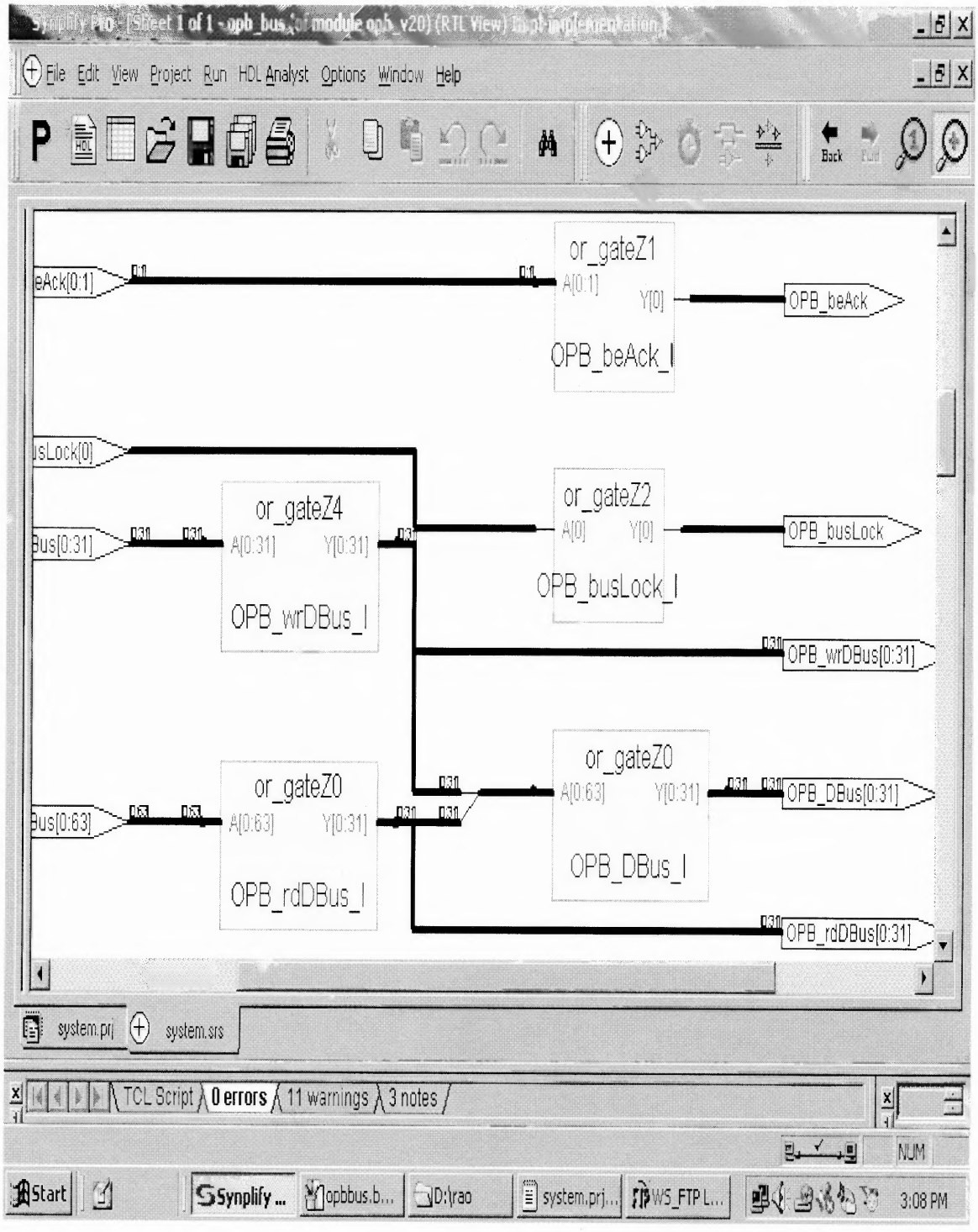

Figure 3.5 Hierarchical view of the OPB bus.

The above Figures 3.3, 3.4, 3.5 show the RTL view of the sequential implementation after synthesis. The maximum frequency at which the sequential design operates is 639 MHz. This occupies just 6% of the FPGA resources.

3.3.2 Performance Results on the Sequential Machine

The two applications, matrix multiplication and LU decomposition, which are major computational parts of power flow analysis, were written in C code for matrices of different sizes. The data are integer numbers, as Microblaze can work on only integer numbers and are initialized in C code. C codes were stored into the Microblaze Block RAM while the Microblaze system was generated. The Wildstar-II APIs were used to program the FPGA using the generated X86 file and to read back the results. APIs were also used to program the clock to be 33 MHz for the sequential implementation. The execution times are calculated by multiplying the number of clock cycles by the clock period. The execution times are as given below:

Matrix Multiplication:

1. For a 3X3 matrix, C code was written and pointers were used to store the results in the global memory that interfaces the host PC. The host reads these results using the APIs; they are verified with the results on the PC.

   The number of integer operations for a 3X3 matrix is 45. Also,

   Clock speed = 33 MHz

   Clock period =0.303 ns

   Number of cycles per instruction= 3

   Total number of clock cycles =45*3 = 135

   Total execution time = 135*0.303 =40.905 ns

2. For a 5x5 matrix, C code was written and pointers were used to store the results in the global memory after computations were carried out. The timing results are as shown below:

   The number of integer operations for the 5X5 matrix is 225. Also,

   Clock period =0.303 ns

   Total number of clock cycles = 225*3 =675

   Total execution time= 375*0.303 = 204.525 ns

LU Decomposition:

1. For a 3X3 matrix, C code was written for LU decomposition and the results were stored in the global memory after computations were carried out. The timing results are as shown below:

   The number of integer operations for a 3X3 matrix =13

   Clock period =0.303 ns

   Total number of clock cycles =13x3 =39

   Total execution time =39*0.303 =11.817 ns

2. For a 5X5 matrix, C code was written and the timing results are as shown below:

   The number of integer operations for a 5X5 matrix is 70

   Clock period =0.303 ns

   Total number of clock cycles =70*3=210

   Total execution time=210*0.303=63.63 ns

Table 3.1 Matrix multiplication on Microblaze

Size of Matrix	**Number of operations (I)**	**Total number of cycles C= I*3**	**Total execution time T= C*0.303 ns**
3x3	45	45*3=135	135*0.3033= 40.905 ns
5x5	225	225*3=675	675*.303= 204.525 ns

Table 3.2 LU decomposition on Microblaze

Size of Matrix	**Number of operations (I)**	**Total number of cycles C= I*3**	**Total execution time T= C*0.303 ns**
3x3	13	13*3=39	39*0.303= 11.817 ns
5x5	70	70*3=210	210*0.303= 63.63 ns

The above Tables 3.1 and 3.2 show the execution times for matrix multiplication and LU decomposition for 3x3 and 5x5 matrices on Microblaze running at 33 MHz. From the

above tables it can be observed that as the size of the matrix increases the execution times are increase drastically.

3.3.3 Performance Results on Microblaze with Floating-Point Co-Processor

Matrix Multiplication:

Matrix multiplication for 3x3 matrices was tested on Microblaze with a floating-point co-processor. The code for matrix multiplication of 3x3 matrices was written using assembly language. The instructions are stored in the instruction memory of the floating-point co-processor and the data are initialized in the data memory of the co-processor. Once the instructions are executed the results are read from the host using Wildstar-II APIs. The assembly language code consists of 90 instructions, of which 36 are load instructions, 9 are store instructions and 45 are floating-point operations. The floating point co-processor consumes 8 cycles for load operations, 11 cycles for floating-point operations and 10 cycles for store operations. The timing results are as shown below:

The number of load operations is 36

The number of floating-point operations is 45

The number of store operations is 9

Total number of cycles = (36*8) + (9*10) + (45*11) = 873

Total execution time = 873*0.303 = 264.519 ns

3.4 SIMD Implementation

3.4.1 Implementation

The SIMD design also uses the above discussed methodologies in the implementation. The modules of the SIMD machine are designed using VHDL code, and are simulated and synthesized. After synthesis is done, the Microblaze development flow is followed where the complete system is generated. This Microblaze development tool configures the soft- core process and its peripheral bus according to the requirements of the system as specified in the MHS file. The Platform Generator counts the number of peripherals as specified in the MHS file, and modifies the OPB bus so that it generates the logic that enables the data, address and control signals connected to all the peripherals. The glue logic written in VHDL for all the peripherals takes care of OPB bus read and write operations by decoding the address specified for that peripheral and then enabling that peripheral. The Platform Generator generates the top level HDL file that integrates the SIMD machine, and glue logic for the OPB and LAD buses. This file is synthesized using Synplify and the final EDIF file is generated. This EDIF file is used by the implementation tools and the final X86 file is generated. This X86 file is used to program the FPGA and realize the SIMD machine on that FPGA. Wildstar-II APIs were used to program the FPGA using this X86, program the clock frequency and read back the results into the host. The synthesis results and the implementation of matrix multiplication on the SIMD machine are discussed below.

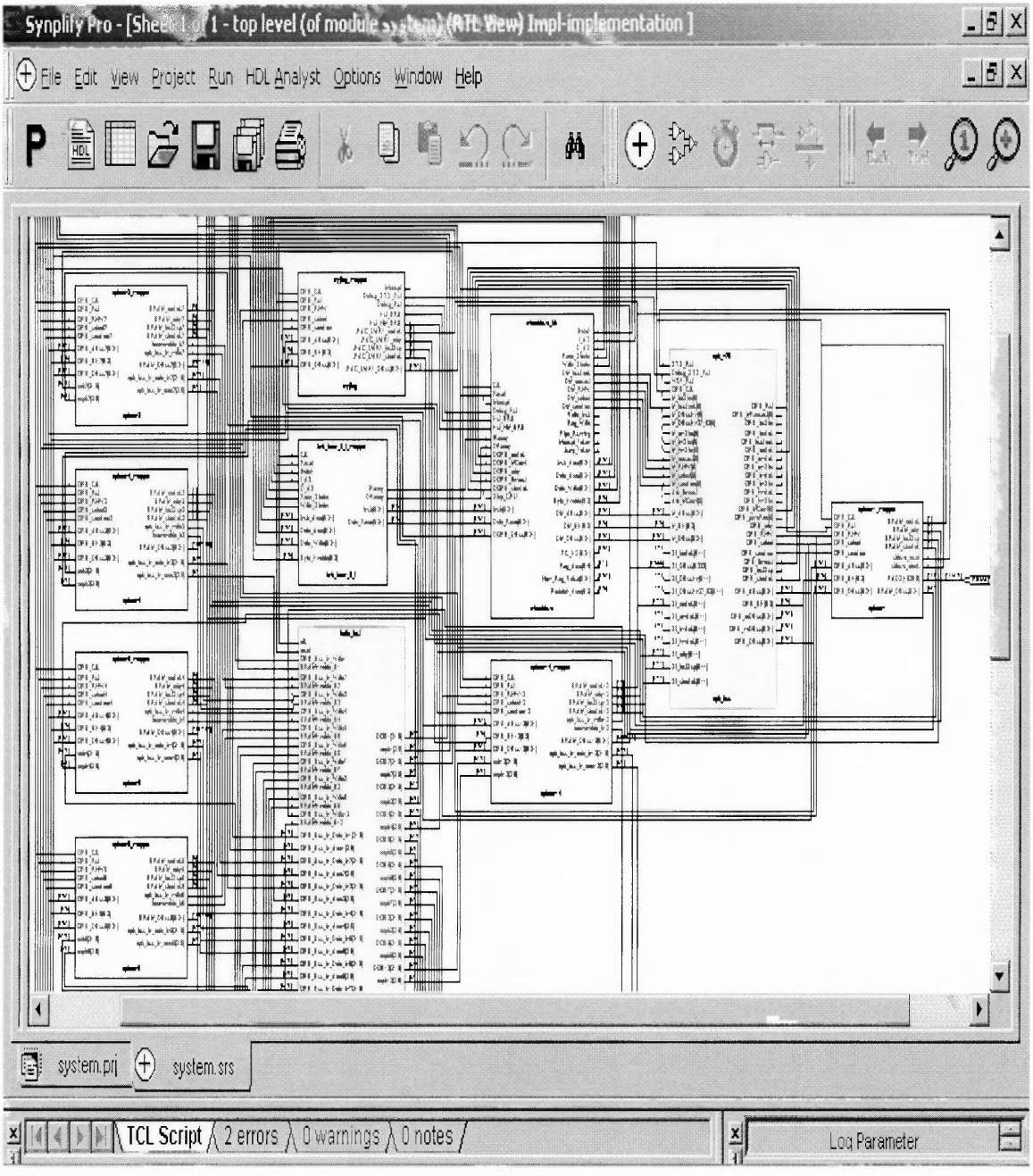

Figure 3.6 RTL view of the SIMD machine after synthesis.

The above Figure 3.6 shows the complete RTL view of the SIMD machine for array processing, the glue logic and the integration of the OPB bus of Microblaze. The following figure shows the RTL view of each module in a high-level block diagram. The processing array is seen clearly in the RTL view shown in Figure 3.7.

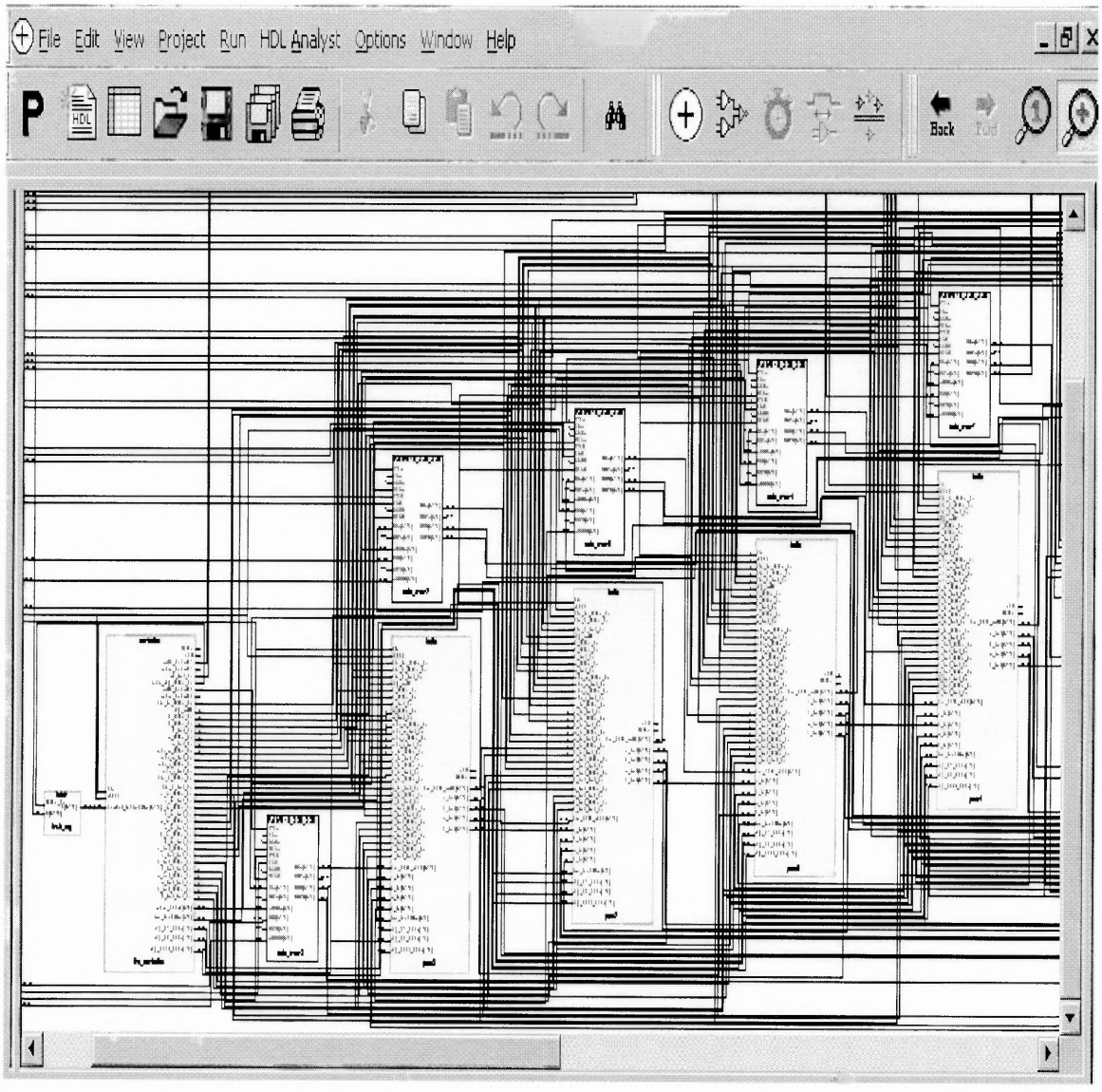

Figure 3.7 RTL view of the processing elements after synthesis.

From this figure, it can be observed that all PEs are connected using the mesh and local memories connected to the PEs. The left-most module is the controller whose RTL view is seen clearly in the diagram of shown in Figure 3.8.

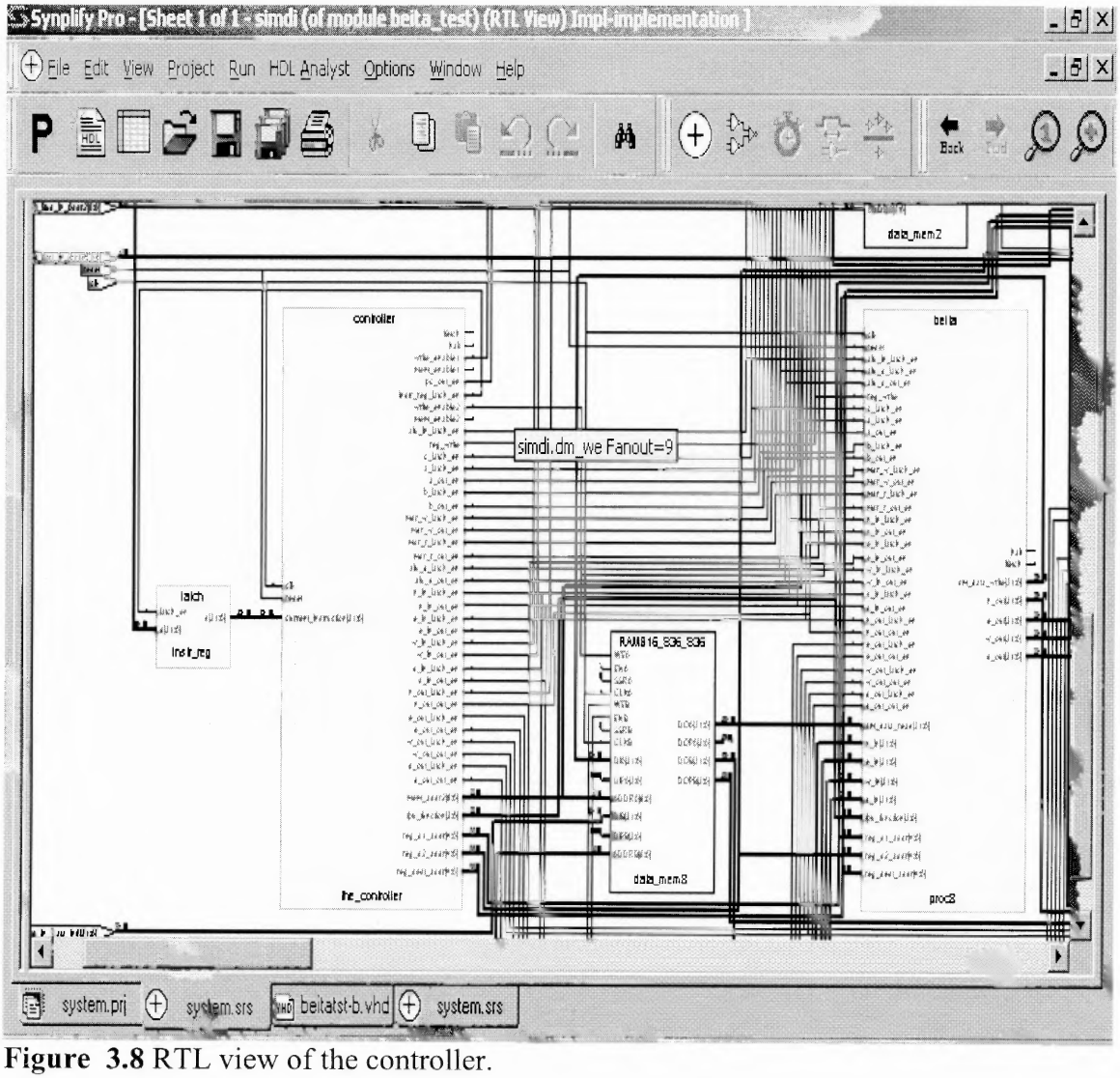

Figure 3.8 RTL view of the controller.

Figure 3.8 shows the RTL view of the controller; the control signals are hardwired to all nine PEs. All the control signals are hardwired because the FPGA supports a fan-out more than 100; as seen in the above figure, the fan-out used for the control signals is only nine which is supported by the Vertex-II FPGA. Figure 3.9 shows a clear view of a PE.

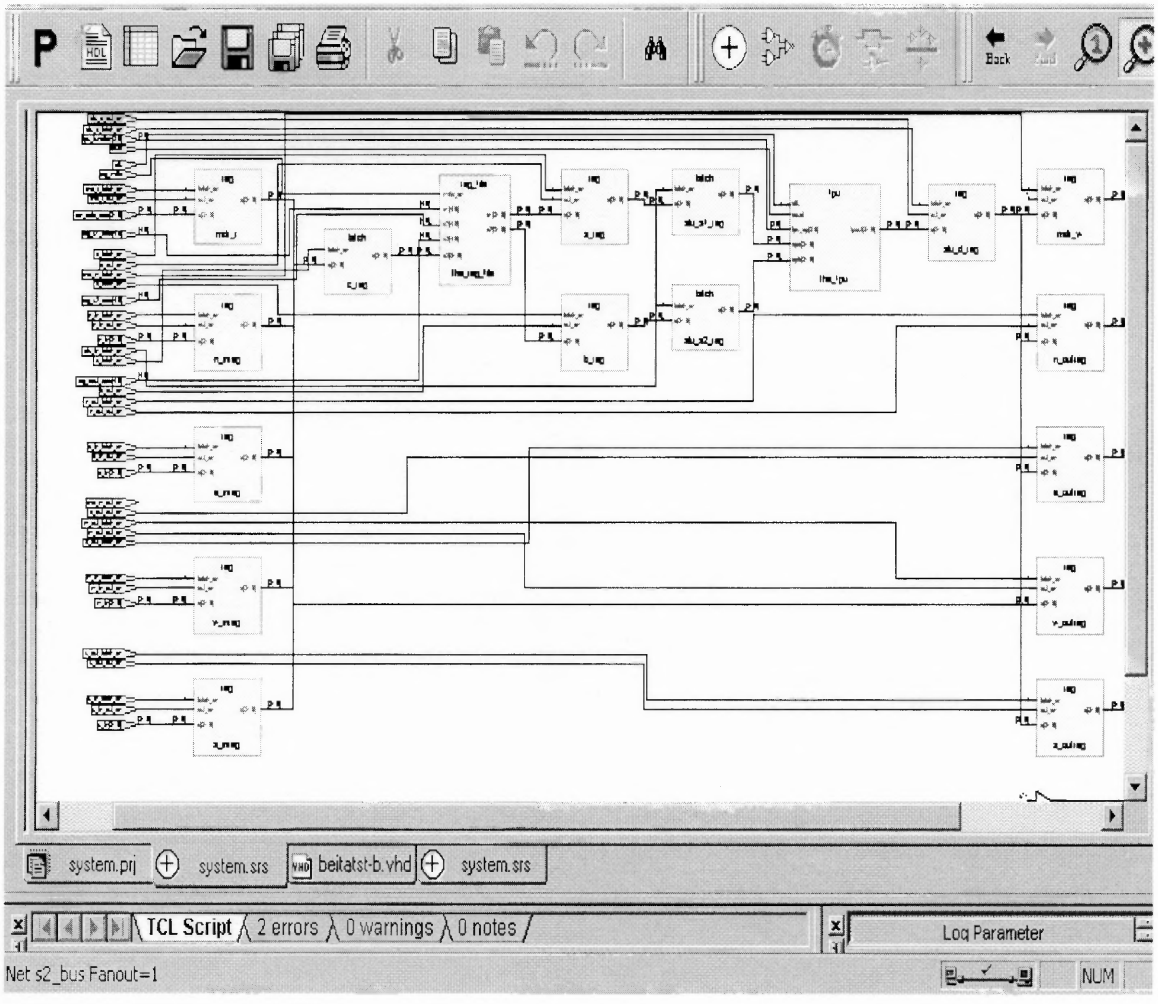

Figure 3.9 RTL view of a Processing Element.

The RTL view of a PE shows its complete architecture. The register interface used for the mesh interconnection is seen in the above RTL view. There are four register inputs and four register outputs for the communication of neighboring PEs in the north,

east, south and west directions. The pipelined functional unit is seen in the above figure connected to the source and destination buses. The pipelined functional unit is seen in Figure 3.10.

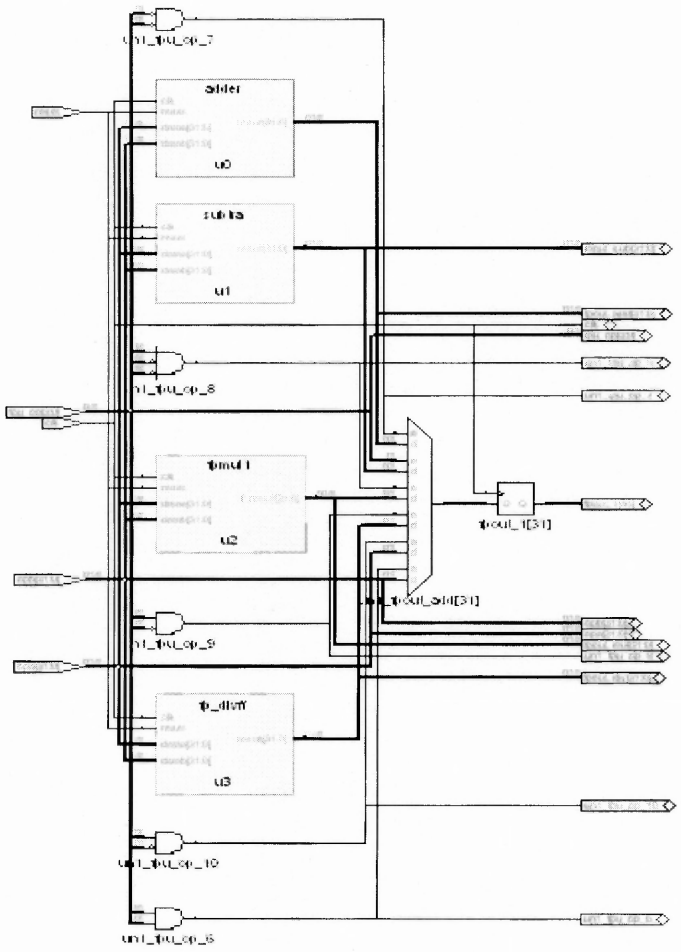

Figure 3.10 RTL view of the pipelined floating-point unit.

The floating-point unit consists of a three-stage adder, a subtractor, and a multiplier and a 28-stage divider. The FPU contains the critical path of the SIMD design, so the FPU is pipelined to improve the clock speed. The maximum frequency at which the FPU can work is 50 MHz. This increases the overall system frequency of the SIMD

design. More specifically, the divider is the slowest functional unit and the critical path lies in the divider, so deep level pipelining is used within the divider. Figure 3.11 shows the 28-stage pipelined divider.

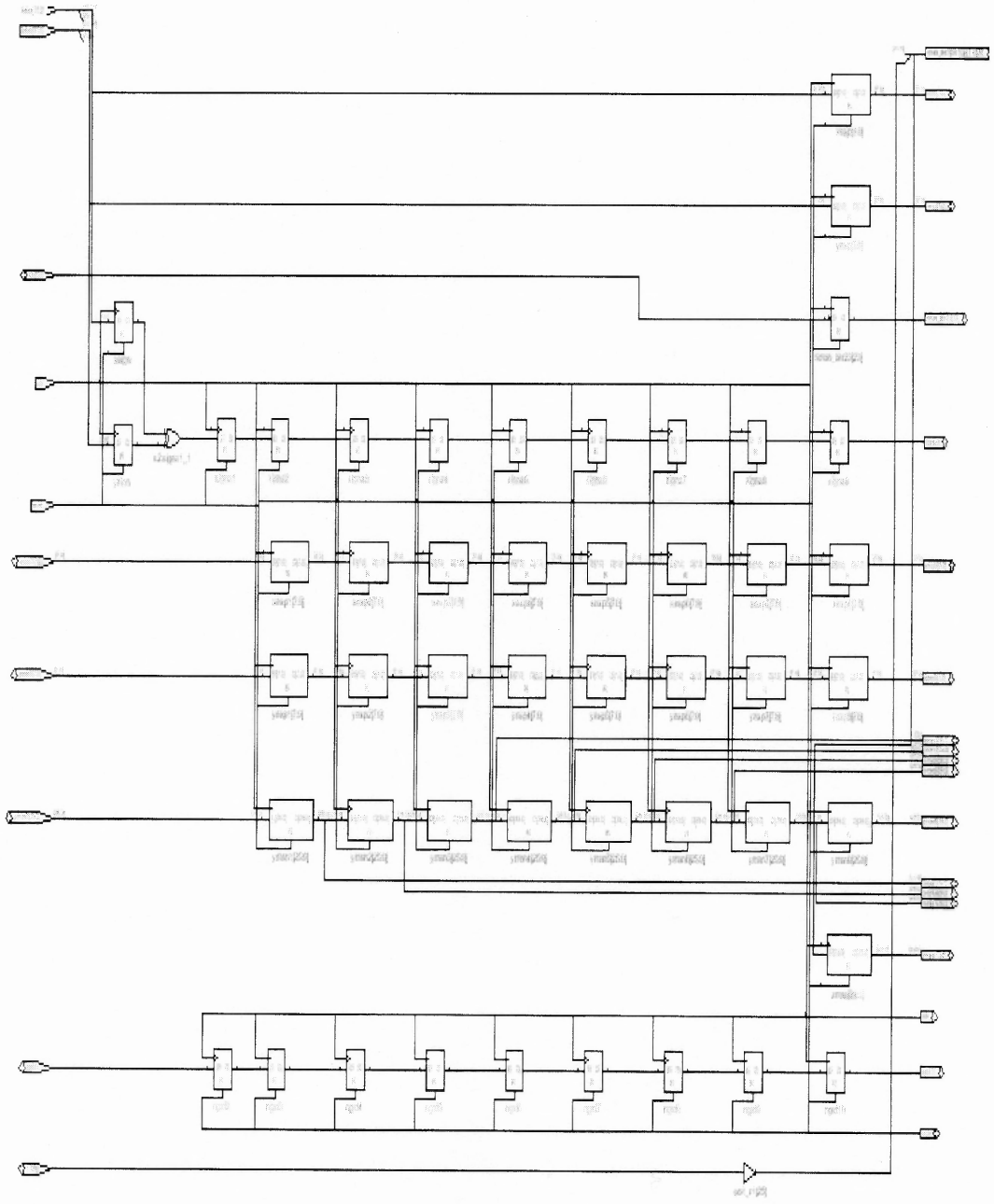

Figure 3.11 RTL view of the pipelined divider.

This is a major functional unit that brought down the system frequency, so a deep pipelined structure was developed to increase the system frequency. The 28-stage pipelined divider works at maximum frequency of 100 MHz. The following figure 3.12 shows the timing report for the SIMD machine.

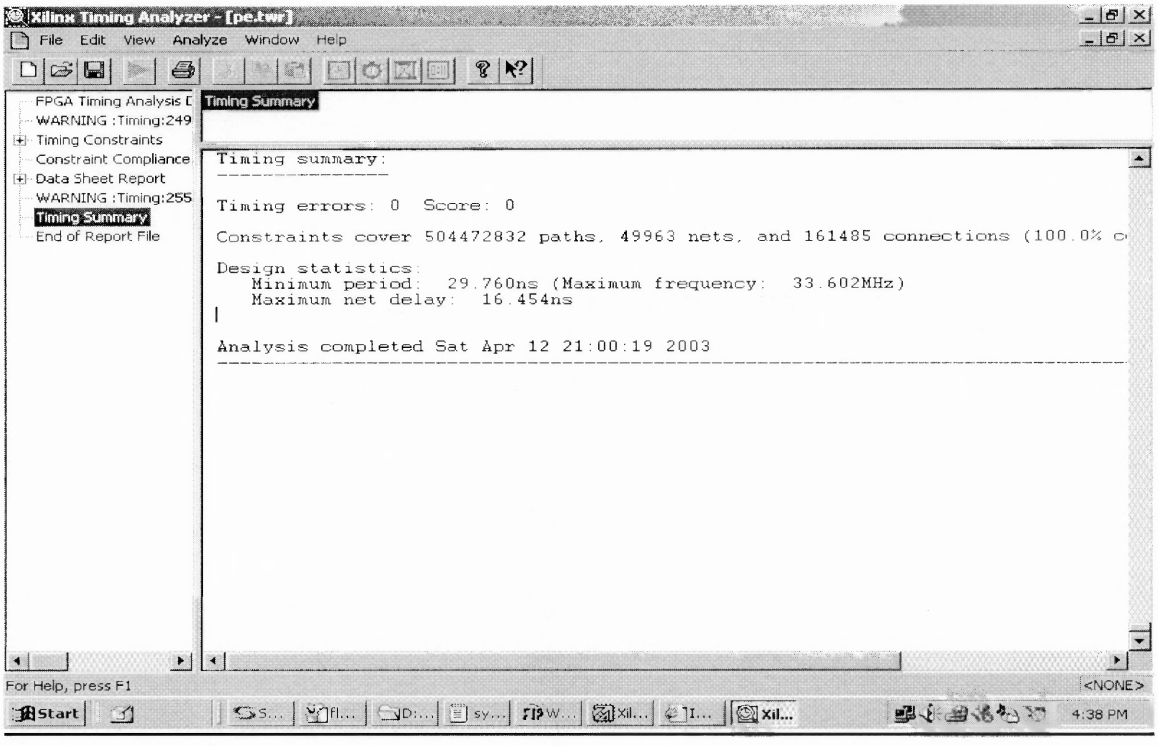

Figure 3.12 Timing report for the SIMD machine.

This report shows that the maximum frequency at which the SIMD machine actually works is 33.602 MHz. Based on this value, the clock generator on the Wildstar board is programmed to be 33 MHz using the Wildstar-II APIs.

The following Figures 3.13 and 3.14, show the device utilization summary showing that about 78% of the "slices" are used. Each slice is a combination of two function generators and two storage elements. This can also be represented in terms of

the Combinational Logical Blocks (CLBs), where each CLB includes four slices and two 3-state buffers. It also shows that 19 block RAMs have been used out of the 144 which are available. This shows that we can still increase the local memory capacity, which is now 2KB. The total space of available block select RAMs is 3MB, so we can increase the local memory size significantly. The total number of gates consumed is 1, 781,590 as seen in Figure 3.14.

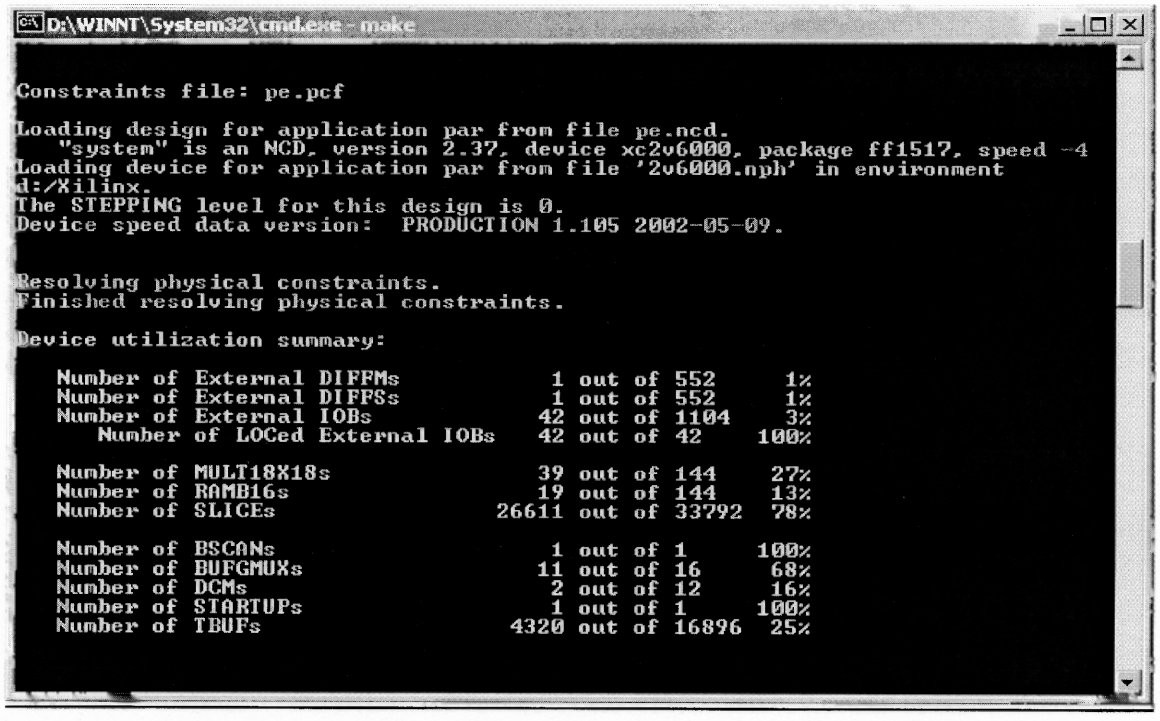

Figure 3.13 Device utilization summary.

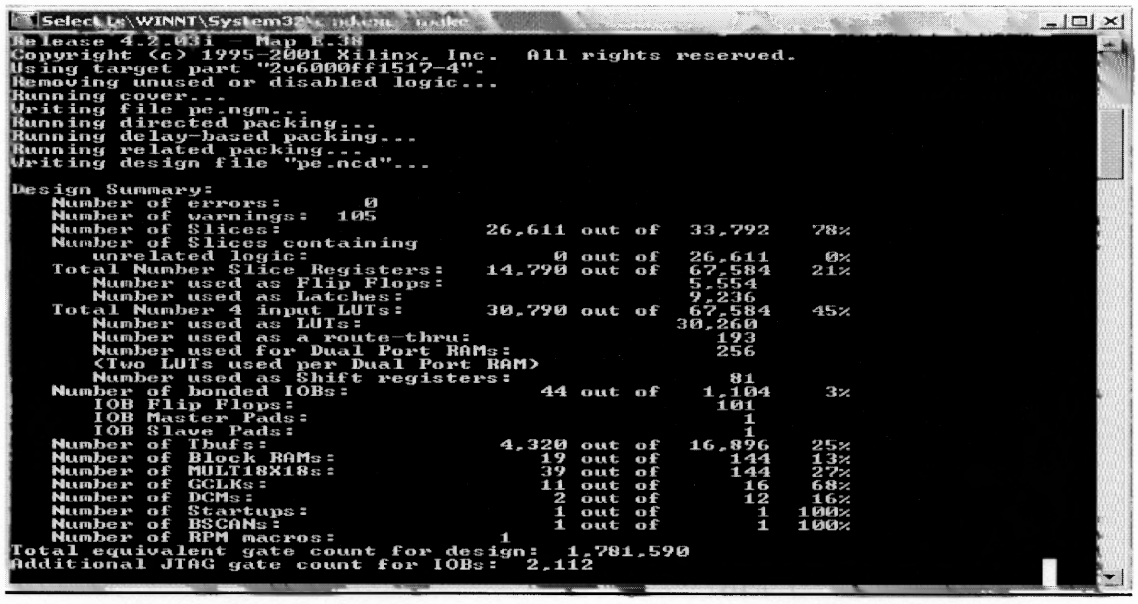

Figure 3.14 Design summary.

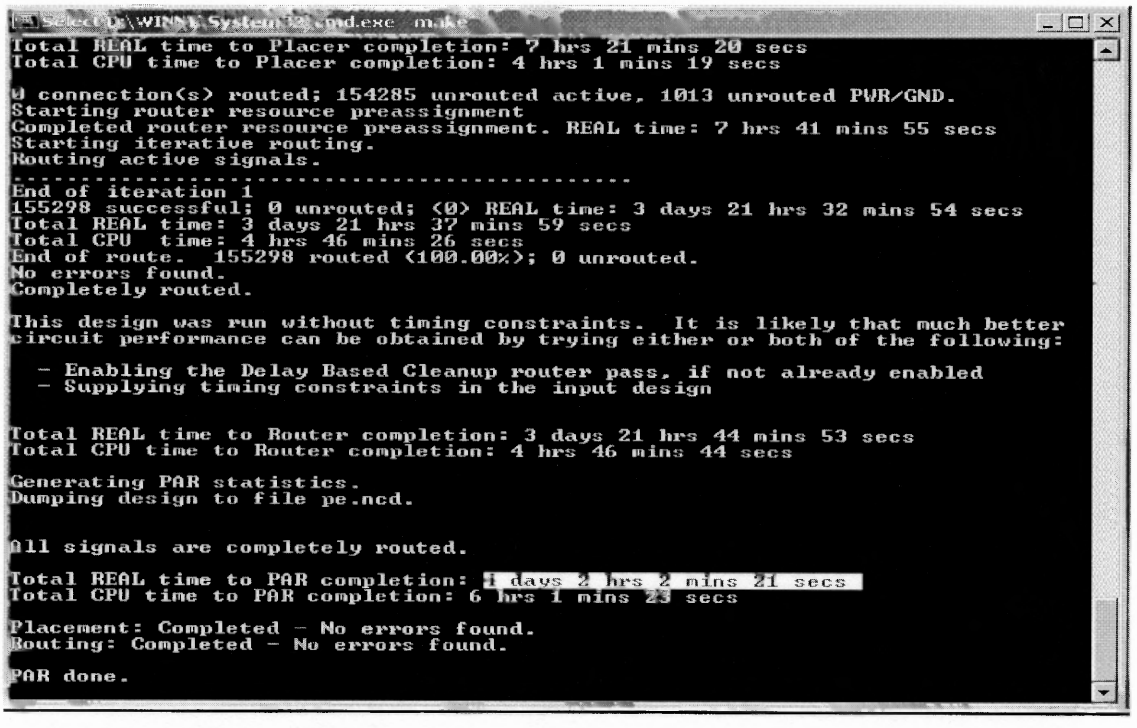

Figure 3.15 Real time for the completion of the "place" and "route" functions.

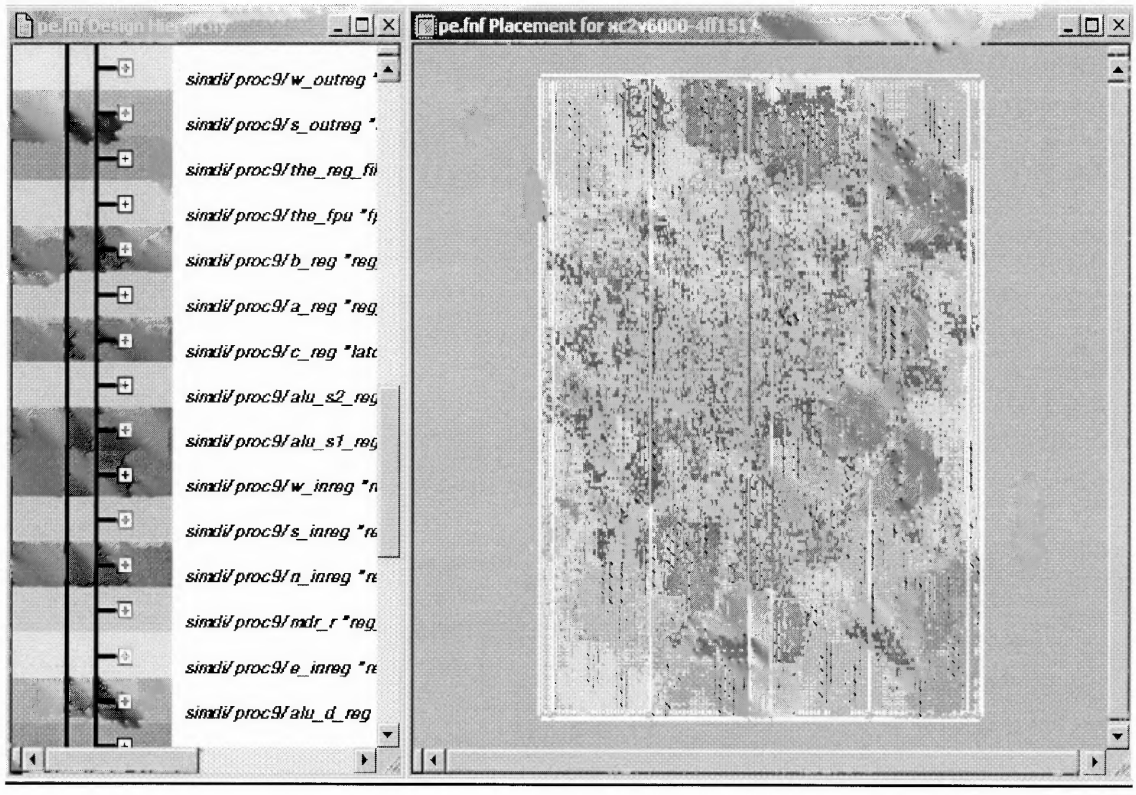

Figure 3.16 Floor plan of the FPGA after "place" and "route".

Figure 3.13 shows that the real time taken to generate the X86 file for the SIMD machine is approximately four days. Figure 3.14 shows the floor plan of the FPGA after "place" and "route".

The X86 file generated after "place" and "route" is used to program the FPGA .Wildstar-II APIs are used to program the clock to be 33 MHz, to program the FPGA, to realize the SIMD on the FPGA and to read back the results from the board. The instruction memory is initialized with different instructions and is realized on the FPGA; all instructions work properly. To port applications, like matrix multiplication and LU decomposition, we should come up with parallel algorithms that map efficiently onto the SIMD machine. The matrix multiplication algorithm was modified for the SIMD

machine, results were obtained, and execution times were calculated by multiplying the number of clock cycles with the clock period. The execution time for matrix multiplication on SIMD machine is discussed below.

3.4.2 Performance Results on the SIMD Machine

Matrix Multiplication:

Matrix multiplication for a 3x3 matrix was tested on the SIMD machine with nine PEs. The 3X3 matrix multiplication algorithm was modified for the SIMD machine with nine PEs. The code was written using the assembly language and consists of twelve instructions; of which six of them are for load operations; five for floating-point operations and one for a store operation. The instructions are stored in the instruction memory and the data are stored in the local memories of the PEs. This reduces the number of instructions, from ninety in the sequential implementation with a floating-point co-processor to twelve in the SIMD implementation. The local memory of PE0 contains the first row of matrix A and the second column of matrix B, PE1 stores the first row of matrix A and the second column of matrix B, and so on. The following assembly-language instructions are converted into binary form and stored into the instruction memory of the SIMD machine to perform 3X3 matrix multiplication simultaneously on the nine PEs:

	Contents of the instruction memory
1. Load r0, mem (0000000000) 000110 0000000000 00000 00000000000 =	18000000
2. Load r1, mem (0000000001) 000110 0000000001 00001 00000000000 =	18010800
3. Load r2, mem (00000000010) 000110 0000000010 00010 0000000000 =	18021000

4. Load r3, mem (0000000011)
000110 0000000011 00011 0000000000 = 18031800

5. Load r4, mem (0000000100)
000110 0000000100 00100 0000000000 = 18042000

6. Load r5, mem (0000000101)
000110 0000000101 00101 0000000000 = 18052800

7. Mul r0, r3, r6
100010 00000 00011 00110 0000000000 = 88033000

8. Mul r1, r4, r7
100010 00001 00100 00111 0000000000 = 88243800

9. Mul r2, r5, r8
100010 00010 00101 01000 0000000000 = 88454000

10. Add r6, r7, r9
000010 00110 00111 01001 0000000000 = 08c74800

11. Add r9, r8,r10
000010 01001 01000 01010 0000000000 = 09285000

12. Store r11, mem (000000111)
000111 0000000111 01010 0000000000 = 1c075000

The input matrices are mapped onto the data memories of the SIMD machine as shown below:

Inputs:

$$\text{Matrix A} = \begin{pmatrix} 3 & 2 & 1 \\ 4 & 5 & 6 \\ 2 & 1 & 3 \end{pmatrix}$$

$$\text{Matrix B} = \begin{pmatrix} 1 & 2 & 4 \\ 7 & 8 & 9 \\ 3 & 5 & 6 \end{pmatrix}$$

1. Data_mem1 ="4040000040e000003f8000003f800000400000040400000"

--- (3, 2, 1, 1, 7, 3) first row first column

2. Data_mem2 = "40a0000041000000400000003f800000400000040400000"

--- (3, 2, 1, 2, 8, 5) first row second column

3. Data_mem3 = "40c0000041100000408000003f800000400000040400000"

-- (3, 2, 1, 4, 9, 6) first row third column

4. Data_mem4 = "4040000040e000003f80000040c0000040a0000040800000"

-- (4, 5, 6, 1, 7, 3) second row first column

5. Data_mem5 = "40a0000041000000400000040c0000040a0000040800000"

-- (4, 5, 6, 2, 8, 5) second row second col

6. Data_mem6 = "40c0000041100000408000040c0000040a0000040800000"

-- (4, 5, 6, 4, 9, 6) second row third column

7. Data_mem7 = "4040000040e000003f800000404000003f80000040000000"

--(2,1,3,1,7,3) third row first column

8. Data_mem8 = "40a0000041000000400000004040000003f80000040000000"

-- (2, 1, 3, 2, 8, 5) third row second column

9. Data_mem9 = "40c0000041100000408000004040000003f80000040000000"

-- (2, 1, 3, 4, 9, 6) third row third column

The data and instructions are initialized in the data and instruction memories, respectively, before generating the X86 file. APIs are used to program the FPGA using the generated X86 file, to program the clock to be 33MHz and to read back the results. The results obtained for this 3x3 matrix multiplication for the given matrix are found to be correct. The SIMD machine consumes eight cycles for a load operation, ten cycles for

a store operation and eleven cycles for a floating-point operation. The execution times are calculated as shown below:

Number of load operations is 6

Number of floating-point operations is 5

Number of store operations is 1

Total number of cycles = (6*8) + (5*11) + (1*10) = 113

Clock period =0.303 ns

Total execution time = 113*0.303 = 34.239 ns

3.5 Analysis

The execution time on the SIMD machine was compared with the execution time of Microblaze with a floating-point co-processor for 3x3 matrix multiplication. The SIMD machine uses nine PEs, each with a FPU and local memory, while the sequential implementation uses a Microblaze with a floating-point co-processor. These execution times do not include the time taken for input/output operations like loading the local memories of PEs, storing the results from local memory to global memory and reading the results from the host. The speed-up factor and efficiency of the SIMD machine are calculated for 3x3 matrix multiplication from the execution times, as shown in the Table 3.3.

Table 3.3 Execution Times for 3x3 Matrix Multiplication

Architecture	Number of Load operations (L)	Number of Store operations (S)	Number of FP operations (FP)	Total number of cycles (c) = L*8+S*10 +FP*11	Total execution time (t) = C*0.303 ns
Microblaze with a FP co-processor	36	9	45	(36*8)+ (9*10)+ (45*11)= 873	873*0.303= 264.519 ns
SIMD Machine	6	1	5	(6*8)+ (5*11)+ (1*10)= 113	113*0.303= 34.239 ns

The execution time for 3x3 matrix multiplication on a single Microblaze with floating-point co-processor running at 33 MHz is 264.519 ns, while the execution on the SIMD machine running at 33 MHz is 34.239 ns. The calculation of the speed up and efficiency are as shown below:

Speedup factor S (n) = T (1)/ T (n)

= 264.519/34.239

= 7.725

Efficiency: E (n) = S (n)/n

= 7.725/9

= 0.8584

This shows the efficiency of the SIMD machine for floating-point operations which are critical operations in power flow applications. The speed-up of 7.725 for nine processors is achieved as the matrix size is equal to the number of processors; for larger matrices comparably better performance is possible. The main advantage of the SIMD

architecture is its scalability that enables it to scale to any number of PEs, limited only by the available FPGA resources. Thus, the efficiency of this FPGA-based SIMD machine is proved over the sequential machine. This performance of the SIMD machine can still be improved by implementing instruction pipelining, or improving the floating-point unit or replacing most of the components by Virtex-II optimized primitives.

CHAPTER 4

CONCLUSIONS

The architecture of an FPGA-based SIMD machine that takes advantage of scalability to produce good performance for power flow analysis problems has been presented. This SIMD machine was implemented on the Annapolis Microsystems Wildstar-II board. It was tested for matrix multiplication and its performance was compared with the sequential implementation using a single Microblaze processor with a floating-point co-processor. The performance of the FPGA-based SIMD architecture is better than that of the sequential processor. With future enhancements, like implementing instruction pipelining, improving the performance of the FPU, or replacing the modules with Xilinx primitives, we could produce even better results. Scaling this machine onto a multi-FPGA system is going to improve the performance even further.

APPENDIX A

CODE ON SIMD MACHINE

The code for matrix multiplication on SIMD machine having nine processing elements is written in assembly language. The following is the code in assembly language.

3x3 matrix multiplication on 9 PEs

1. load r1, mem(0000000000)
000110 0000000000 00000 00000000000 = 18000000

2. load r2, mem(0000000001)
000110 0000000001 00001 00000000000 = 18010800

3. load r3,mem(00000000010)
000110 0000000010 00010 0000000000 = 18021000

4. load r4,mem(0000000011)
000110 0000000011 00011 0000000000 = 18031800

5. load r5,mem(0000000100)
000110 0000000100 00100 0000000000 = 18042000

6. load r6,mem(0000000101)
000110 0000000101 00101 0000000000 = 18052800

7. mul r1,r4,r7
100010 00000 00011 00110 0000000000 = 88033000

8. mul r2,r5,r8
100010 00001 00100 00111 0000000000 = 88243800

9. mul r3,r6,r9
100010 00010 00101 01000 0000000000 = 88454000

10. add r7,r8,r10
000010 00110 00111 01001 0000000000 = 08c74800

11. add r10,r9,r11
000010 01001 01000 01010 0000000000 = 09285000

12. store r11,mem(000000111)
000111 0000000111 01010 0000000000 = 1c075000

Inputs:
matrix a={3 2 1
4 5 6
2 1 3}
matrix b={1 2 4
7 8 9
3 5 6}

data_mem1 ="4040000040e000003f8000003f800000400000040400000"---
(3,2,1,1,7,3)firstrow first col
data_mem2 = "40a0000041000000400000003f800000400000040400000"---
(3,2,1,2,8,5)first row sec col
data_mem3 = "40c0000041100000408000003f800000400000040400000"--
(3,2,1,4,9,6)first row thrd col
data_mem4 = "4040000040e000003f80000040c0000040a0000040800000"--
(4,5,6,1,7,3)secondrow firstcol
data_mem5 = "40a0000041000000400000040c0000040a0000040800000"--
(4,5,6,2,8,5)secondrow sec col
data_mem6 = "40c0000041100000408000040c0000040a0000040800000"--
(4,5,6,4,9,6)secondrow thrd col
data_mem7 = "4040000040e000003f800000404000003f80000040000000"--
(2,1,3,1,7,3)thrd row first col
data_mem8 = "40a0000041000000400000004040000003f80000040000000"--
(2,1,3,2,8,5)thrd row sec col
data_mem9 = "40c0000041100000408000004040000003f80000040000000"--
(2,1,3,4,9,6)thrd row thrd col

result={41a00000 41d80000 42100000

42640000 429c0000 42c20000

41900000 41d80000 420c0000}

APPENDIX B

CODE ON SEQUENTIAL MACHINE

The code for matrix multiplication on sequential machine with co-processor is written in assembly language. The following is the code in assembly language.

3x3 matrix multiplication on 1 PEs

1. load r0,mem(1)	18010000
2. load r1,mem(2)	18020800
3. load r2, mem(3)	18031000
4. load r3,mem(10)	180a1800
5. load r4,mem(11)	180b2000
6. load r5,mem(12)	180c2800
7. mul ro,r3,r6	88033000
8. mul r1,r4,r7	88243800
9. mul r2,r5,r8	88454000
10. add r6,r7,r9	08c74800
11. add r9,r8,r10	09285000
12. store r10,mem(19)	1c135000
13. load r3,mem(13)	180d1800
14. load r4,mem(14)	180e2000
15. load r5,mem(15)	180f2800
16. mul ro,r3,r6	88033000
17. mul r1,r4,r7	88243800

18. mul r2,r5,r8	88454000
19. add r6,r7,r9	08c74800
20. add r9,r8,r10	09285000
21. store r10,mem(20)	1c145000
22. load r3,mem(16)	18101800
23. load r4,mem(17)	18112000
24. load r5,mem(18)	18122800
25. mul ro,r3,r6	88033000
26. mul r1,r4,r7	88243800
27. mul r2,r5,r8	88454000
28. add r6,r7,r9	08c74800
29. add r9,r8,r10	09285000
30. store r10,mem(21)	1c155000
31. load r0,mem(4)	18040000
32. load r1,mem(5)	18050800
33. load r2, mem(6)	18061000
34. load r3,mem(10)	180a1800
35. load r4,mem(11)	180b2000
36. load r5,mem(12)	180c2800
37. mul ro,r3,r6	88033000
38. mul r1,r4,r7	88243800
39. mul r2,r5,r8	88454000
40. add r6,r7,r9	08c74800

41. add r9,r8,r10	09285000
42. store r10,mem(22)	1c165000
43. load r3,mem(13)	180d1800
44. load r4,mem(14)	180e2000
45. load r5,mem(15)	180f2800
46. mul ro,r3,r6	88033000
47. mul r1,r4,r7	88243800
48. mul r2,r5,r8	88454000
49. add r6,r7,r9	08c74800
50. add r9,r8,r10	09285000
51. store r10,mem(23)	1c175000
52. load r3,mem(16)	18101800
53. load r4,mem(17)	18112000
54. load r5,mem(18)	18122800
55. mul ro,r3,r6	88033000
56. mul r1,r4,r7	88243800
57. mul r2,r5,r8	88454000
58. add r6,r7,r9	08c74000
59. add r9,r8,r10	09285000
60. store r10,mem(24)	1c185000
61. load r0,mem(7)	18070000
62. load r1,mem(8)	18080800
63. load r2, mem(9)	18091000

64. load r3,mem(10) 180a1800

65. load r4,mem(11) 180b2000

66. load r5,mem(12) 180c2800

67. mul ro,r3,r6 88033000

68. mul r1,r4,r7 88243800

69. mul r2,r5,r8 88454000

70. add r6,r7,r9 08c74800

71. add r9,r8,r10 09285000

72. store r10,mem(25) 1c195000

73 load r3,mem(13) 180d1800

74. load r4,mem(14) 180e2000

75. load r5,mem(15) 180f2800

76. mul ro,r3,r6 88033000

77. mul r1,r4,r7 88243800

78. mul r2,r5,r8 88454000

79. add r6,r7,r9 08c74800

80. add r9,r8,r10 09285000

81. store r10,mem(26) 1c1a5000

82. load r3,mem(16) 18101800

83. load r4,mem(17) 18112000

84. load r5,mem(18) 18122800

85. mul ro,r3,r6 88033000

86. mul r1,r4,r7 88243800

87. mul r2,r5,r8 88454000

88. add r6,r7,r9 08c74800

89. add r9,r8,r10 09285000

90. store r10,mem(27) 1c1b5000

Inputs:
matrix a={3 2 1
4 5 6
2 1 3}
matrix b={1 2 4
7 8 9
3 5 6}

data_mem1 ="
----------start of matrix a----------
40400000 ---- memoryt location 1 mem(1)
40000000
3f800000
40800000
40a00000
40c00000
40000000
3f800000
40400000 -----mem(9)
----------end of matrix a----------
----------start of matrix b----------
3f800000 ------mem(10)
40e00000
40400000
40000000
41000000
40a00000
40800000
41100000
40c00000 ------mem(18)
----------end of matrix b----------

----------results stored from mem(19) to mem(27)----------

```
result=41a00000     41d80000     42100000
       42640000     429c0000     42c20000
       41900000     41d80000     420c0000
```

LU decomposition on Sequential Machine without co-processor

```
#include<stdio.h>
#define max 5

main()
{
        //float *c=(float*)8001;
        int A[max][max]={6,9,3,-2,-1,7,0,1,5,1,2,3,4,5,6,7,8,9,0,2,3,4,5,4,5};
        int i,j,k;
    int cntdiv =0;
         int cntmul =0;
         int cntsub =0;


        for(k=0;k<max-1;k++)
        {
                if(A[k][k]==0)
                {
                        printf("This Matrix's def==0,no result!\n");

                }
                for(i=k+1;i<max;i++)
                {
                        A[i][k]=A[i][k]/A[k][k];
                        cntdiv=cntdiv+1;
                }

                for(i=k+1;i<max;i++)
                {
                        for(j=k+1;j<max;j++)
                        {
                                A[i][j]=A[i][j]-A[i][k]*A[k][j];
                                cntsub=cntsub+1;
                                cntmul=cntmul+1;

                        }


                }

        }
        printf("no.of operations");
```

```
        printf("%d %d %d \n",cntdiv,cntmul,cntsub);
        for(i=0;i<max;i++)
        {
                for(j=0;j<max;j++)
                {
                        //*c++=A[i][j];
                        printf("%d \n",A[i][j]);


                }
        }
return 0;
}
```

APPENDIX C

CODE ON HOST MACHINE

The code on host machine is written using Wildstar-II APIs. The following is the code on the host machine using Wildstar-II APIs.

```
#define WSII_PCI_WIN32
#include "wsii.h"
#include <stdio.h>
#include <conio.h>


#define M_CLOCK_FREQ 30.0

#define PE_FILENAME "h:\\rao\\simd9pe_matmul backup\\simd1pe\\pe.x86"


#define RESET_BASE 0x7FFFF
#define CTRL_REG_BASE 0x100
#define BLOCK_RAM_BASE 0x200
#define BLOCK_RAM_DWORDS 0x200

DWORD ProgramPE(WSII_BOARD hBoard, WSII_PE_NUM PENum, char *
pFilename);
DWORD ProgramBPE(WSII_BOARD hBoard, WSII_PE_NUM PENum, char *
pFilename);


//#define PE_FILENAME
"c:\\annapolis2\\host\\tools\\XC2V6000\\FF1517\\wsii_pci_pe.x86"
//#define PE_FILENAME
"c:\\prabhu\\wildstar\\mb_ex\\implementation\\configfpga\\wsii_pci_pe.x86"


int main (int argc, char* argv[])

{

        int j;
        DWORD dBoardNum = 0;
        WSII_BOARD hBoard;
```

```
DWORD *pReadBuffer;
DWORD *pWriteBuffer;


DWORD dUserInput;

DWORD error,i;

BOOLEAN bError;

DWORD dPEMask;

WSII_PE_NUM dPENum = WSII_PE0;

double fClockFreq;


int argi;

for ( argi = 1; argi < argc; argi++ )
{
 if ( argv [ argi ][ 0 ] == '-' )
 {
  switch ( toupper( argv [ argi ] [ 1 ]) )
  {

  case 'B':
   argi++;
   if (argi < argc)
   {
      dBoardNum = strtoul( argv [ argi ], NULL, 0 );
      printf("Setting Board Number to %x\n", dBoardNum );
   }
   else
   {
    printf( "  Warning:  Invalid Board Number!\n");
    return(0);
   }

    break;

  case 'P':
   argi++;
   if (argi < argc)
   {
    dUserInput = strtoul( argv [ argi ], NULL, 0 );
```

```
        if (dUserInput == 0)
        {
         dPENum = WSII_PE0;
         printf("Setting PE Number to PE%d\n", dUserInput );
        }
        else if (dUserInput == 1)
        {
         dPENum = WSII_PE1;
         printf("Setting PE Number to PE%d\n", dUserInput );
        }
        else
        {
         printf( "  Warning: Invalid PE Number PE%d!\n", dUserInput);
         return(0);
        }
       }
       else
       {
        printf( "  Warning: Invalid PE Number!\n");
        return(0);
       }

       break;


       default:
        printf ( "  Unknown option: \"%s\"\n", argv [ argi ] );
        return(0);
     }
    }
}

/*********************************************************
 *  The first step for all programs must be to open the  *
 *  board(s), using WSII_Open.  If successful, this call *
 *  returns a board handle.  This handle must be used in *
 *  all subsequent API calls that targer this board.     *
 *********************************************************/

        printf("Opening Board ... ");
        hBoard = WSII_Open (dBoardNum, &error, WSII_FLAGS_RESERVED );

/*  An error in opening the board has occured if WSII_Open *
 *  returns NULL, or if the error parameter is not         *
 *  WSII_SUCCESS.  If an error occured, we print the error *
```

```
 * and exit.                                    */
 if ((hBoard == NULL) || (error!=WSII_SUCCESS))
 {
  printf( "WSII API was unable to open board.\nrc=%d: %s\n", error,
WSII_GetErrorString(error) );
  return 0;
 }

 /************************************************************
  * The next step is to programm all PE Clocks to the      *
  * desired frequencies for the test.  Clocks should be    *
  * set BEFORE the PE is programmed so that they are stable*
  * when the PE comes out of reset.  This allows any DCMs  *
  * in the PE to lock on startup.                          *
  ************************************************************/

 printf("Setting M Clock to %f.\n", M_CLOCK_FREQ);

 fClockFreq = WSII_SetClockFrequency (hBoard, WSII_CLK_GLOBAL_M,
M_CLOCK_FREQ);
 printf ("M Clock Set to %f\n", fClockFreq);


 printf("Programming PE with %s ... ", PE_FILENAME);

 if (!ProgramPE(hBoard, dPENum, PE_FILENAME)) return 0;
 else printf("DONE\n");

 /************************************************************
  * Initialize Variables :                                 *
  * This example uses two buffers to communicate with the PE, *
  * a read buffer and a write buffer.  The write buffer is    *
  * used to store data written to the PE, and the read buffer *
  * for data read from the PE.  At the end of the example     *
  * these buffers are compared and the differences printed.   *
  ************************************************************/
 pReadBuffer  = (DWORD*) malloc(BLOCK_RAM_DWORDS*sizeof(DWORD));
 pWriteBuffer = (DWORD*) malloc(BLOCK_RAM_DWORDS*sizeof(DWORD));

 if (!pReadBuffer || !pWriteBuffer)
 {
  printf("ERROR Allocating Buffers\n");
  return 0;
 }

 srand(time(NULL));
```

```
 for (i = 0; i< BLOCK_RAM_DWORDS;i++ )
 {
  /*********************************************************
   * Clear the read buffer and init the write buffer with  *
   * a random data pattern.                                *
   *********************************************************/
 // pReadBuffer[i] = 0;
  pWriteBuffer[i] = rand() | (rand() << 14) | (rand() << 28);
 }

 printf("\nResetting PE0          ... ");
 if (dPENum == WSII_PE0) dPEMask = WSII_MASK_PE0;
 else if (dPENum == WSII_PE1) dPEMask = WSII_MASK_PE1;

 WSII_PeReset(hBoard, dPEMask, WSII_RESET_PULSE);
 if (WSII_GetLastError (hBoard) != WSII_SUCCESS)
 {
                printf("Error %d Resetting Board\n", WSII_GetLastError (hBoard) );
  return 0;
        }
 printf("DONE\n");

 for(j=0;j<100000000;j++)
          ;



printf("\n\n*****************************************************************
******\n"
        "* This Example tests the LAD and LEDs on the WILDSTARII(tm) board. *\n"

"*****************************************************************\n\n
"
      " STEP 1 Test the LAD by Writing and Reading a Block RAM\n");

 /*************************************************************
  * In this PE Image there is a block RAM mapped to LAD addresses *
  * 0x200 to 0x400.  Here we read and write to the Block RAM to   *
  * test the LAD bus.                                     *
  *************************************************************/
 // WSII_WriteReg_32(hBoard, dPENum, BLOCK_RAM_BASE,
BLOCK_RAM_DWORDS, pWriteBuffer);
  if (WSII_GetLastError (hBoard) != WSII_SUCCESS)
  {
                printf("Error %d Writing to Board\n", WSII_GetLastError (hBoard) );
   return 0;
```

```
        }

 //wait until SIMD instructions have been finished
 printf("press any key when SIMD finished all the instructions\n");
 getchar();
 WSII_ReadReg_32(hBoard, dPENum, BLOCK_RAM_BASE,
BLOCK_RAM_DWORDS, pReadBuffer);
 if (WSII_GetLastError (hBoard) != WSII_SUCCESS)
 {
                printf("Error %d Writing to Board\n", WSII_GetLastError (hBoard) );
  return 0;
        }

 bError = FALSE;
 for (i=0; i< BLOCK_RAM_DWORDS; i++)
 {
 //  if (pReadBuffer[i] != pWriteBuffer[i])
 //  {
    printf("  BLOCK RAM value at offset %d.  Found 0x%08x \n", i, pReadBuffer[i]);
 //   bError = TRUE;
 //  }
 }

 if (!bError) printf("    Block RAM LAD test SUCCESSFUL\n\n");


 printf("  STEP2: Test the LEDs\n\n");

 printf("Press Any Key to Turn Off all LEDs\n");
 getch();
 pWriteBuffer[0] = 0x0;

 WSII_WriteReg_32(hBoard, dPENum, CTRL_REG_BASE, 1, pWriteBuffer);
 if (WSII_GetLastError (hBoard) != WSII_SUCCESS)
 {
                printf("Error %d Writing to Board\n", WSII_GetLastError (hBoard) );
  return 0;
        }


 printf("Press Any Key to Turn On the GREEN LED\n");
 getch();
 pWriteBuffer[0] = 0x1;

 WSII_WriteReg_32(hBoard, dPENum, CTRL_REG_BASE, 1, pWriteBuffer);
 if (WSII_GetLastError (hBoard) != WSII_SUCCESS)
```

```
{
		printf("Error %d Writing to Board\n", WSII_GetLastError (hBoard) );
 return 0;
	}

printf("Press Any Key to Turn On the YELLOW LED\n");
getch();
pWriteBuffer[0] = 0x2;

WSII_WriteReg_32(hBoard, dPENum, CTRL_REG_BASE, 1, pWriteBuffer);
if (WSII_GetLastError (hBoard) != WSII_SUCCESS)
{
		printf("Error %d Writing to Board\n", WSII_GetLastError (hBoard) );
 return 0;
	}


printf("Press Any Key to Turn On the RED LED\n");
getch();
pWriteBuffer[0] = 0x4;

WSII_WriteReg_32(hBoard, dPENum, CTRL_REG_BASE, 1, pWriteBuffer);
if (WSII_GetLastError (hBoard) != WSII_SUCCESS)
{
		printf("Error %d Writing to Board\n", WSII_GetLastError (hBoard) );
 return 0;
	}


printf("Press Any Key to Turn On ALL LEDs\n");
getch();
pWriteBuffer[0] = 0x7;

WSII_WriteReg_32(hBoard, dPENum, CTRL_REG_BASE, 1, pWriteBuffer);
if (WSII_GetLastError (hBoard) != WSII_SUCCESS)
{
		printf("Error %d Writing to Board\n", WSII_GetLastError (hBoard) );
 return 0;
	}


printf("Press Any Key to Turn off ALL LEDs\n");
getch();
pWriteBuffer[0] = 0x0;

WSII_WriteReg_32(hBoard, dPENum, CTRL_REG_BASE, 1, pWriteBuffer);
```

```
 if (WSII_GetLastError (hBoard) != WSII_SUCCESS)
 {
                printf("Error %d Writing to Board\n", WSII_GetLastError (hBoard) );
  return 0;
        }

 printf("Press Any Key to Exit\n");
 getch();
 WSII_PeDeprogram(hBoard, dPENum );

 return 0;
}


DWORD ProgramPE(WSII_BOARD hBoard, WSII_PE_NUM PENum, char *
pFilename)
{
        FILE *f;
        long filesize = 0;

        DWORD *pBuffer;

        f = fopen(pFilename, "rb");
        if (!f)
        {
                printf("Cannot Open File : %s\n", pFilename);
                return 0;
        }

        fseek(f, 0, SEEK_END);
        filesize =ftell(f);
        fseek(f, 0, SEEK_SET);

        pBuffer = (DWORD*) malloc(filesize);

        fread(pBuffer, 1,filesize, f);

 WSII_PeProgram (hBoard, PENum,  pBuffer, filesize>>2);

 if (WSII_GetLastError (hBoard) != WSII_SUCCESS)
 {
                printf("Error %d programming PE\n", WSII_GetLastError (hBoard) );
  return 0;
        }

 return 1;}
```

REFERENCES

1. Anjan Bose and Jun Qiang Wu, "Parallel Solution of Large Sparse Matrix Equations and Parallel Flow," IEEE Transactions on Power Systems, Vol. 10, No. 3, August 1995.

2. Bozidar Radanovich, "An Overview of Advances in Reconfigurable Computing Systems," Proceedings, Conference on System Sciences, 1999.

3. Daniel J.Tylavsky and Anjan Bose, "Parallel Processing in Power Systems Computation," Transactions on Power Systems, Vol. 7, No. 2, May 1992.

4. Reiner.Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective," IEEE Proc. Int. Conf. Exhib. Design Automation, Testing Europe, Munich, Germany, 2001, pp. 135-143.

5. Harris, D.L.; Oberman, S.F.; Horowitz, M.A.; Computer Arithmetic, 1997. Proceedings, 13th IEEE symposium on, 6-9 Jul 1997.

6. Scott Hauck, Gaetano Borriello, Carl Ebeling, "Mesh Routing Topologies for Multi-FPGA Systems," International Conference on Computer Design, pp. 170-177, 1994.

7. Nabeel. Shirazi, Al. Walters, and Peter. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines," IEEE Proceedings Symposium, FPGAs Custom Computing Machines, Napa Valley, California, April 1995, pp. 155-162.

8. Sotirios. G. Ziavras, "Processor Design Based on Dataflow Concurrency," Microprocessors and Microsystems, Vol. 27, No. 4, May 2003, pp. 199-220.

9. Xiaofang.Wang and Sotirios.G. Ziavras, "Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines," Concurrency and Computation, 2003.

10. Sotirios.G. Ziavras, "Investigation of Various Mesh Architectures with Broadcast Buses for High Perform. Bus-Based Arch., R.Lin and S.Olariu, Vol. 9, No. 1, Jan 1999, pp. 29-54.

11. E.V.Krishnamurthy and S.G. Ziavras, "Complexity of Matrix Partitioning Schemes for g-inversion on the Connection Machine," Centre for Automation Research and Computer Science Department, University of Maryland, CAR-TR-400 and CS-TR-216, Oct.1998.

12. http://toolbox.xilinx.com/docsan/xilinx4/data/docs/lib/dsgnelpr32.html,(retrieved on April 2003).

13. Microblaze Hardware Reference Guide, version 2.1, March 2003.

14. Widlstar-II Hardware Reference manual, Annapolis Microsystems, revision 2.4, 2002.