# Unstructured Finite Element Computations on Configurable Computers

by

**Karthik Ramachandran**

Thesis submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements of the degree of

**Master of Science**
in
**Electrical Engineering**

Mark T. Jones, Chairman
Peter M. Athanas
James R. Armstrong

August 3, 1998
Blacksburg, Virginia

Keywords: CCM, FPGA, FEM, BLAS

# Unstructured Finite Element Computations on Configurable Computers

**Karthik Ramachandran**

**Dr. Mark T. Jones, Chairman**

**Bradley Department of Electrical and Computer Engineering**

(Abstract)

Scientific solutions to physical problems are computationally intensive. With the increasing emphasis in the area of Custom Computing Machines, many physical problems are being solved using configurable computers. The Finite Element Method (FEM) is an efficient way of solving physical problems such as heat equations, stress analysis and two- and three-dimensional Poisson's equations. This thesis presents the solution to physical problems using the FEM on a configurable platform. The core computational unit in an iterative solution to the FEM, the matrix-by-vector multiplication, is developed in this thesis along with the framework necessary for implementing the FEM solution. The solutions for 2-D and 3-D Poisson's equations are implemented with the use of an adaptive mesh refinement method. The dominant computation in the method is matrix-by-vector multiplication and is performed on the Wildforce board, a configurable platform. The matrix-by-vector multiplication units developed in this thesis are basic mathematical units implemented on a configurable platform and can be used to accelerate any mathematical solution that involves such an operation.

# ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Mark T. Jones, for giving me this opportunity to work in the area of custom computing machines. He has been a great source of inspiration and has guided me throughout my work. I would also like to thank Dr. Peter Athanas for his valuable suggestions. Thanks are due to Dr. J.R.Armstrong for serving on my committee.

I also thank my family for their constant encouragement and support. I thank all my friends and colleagues who have made my stay at Virginia Tech a memorable one.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# Chapter 1

# Introduction

## 1.1 Motivation

Scientific solutions for many physical problems are computationally intensive. Though conventional processors can be used to solve such, they are not necessarily fast problems due to their generality. The computation time for such problems depends on a number of factors including memory access time, pipelineability and sharing of processor time by other processes. The functionality of a general-purpose processor will be underutilized by a specific computation; the efficiency of the processor is low. Hence a processor or a computing system built for a specific purpose tends to be faster.

*Configurable Computing Machines* (CCMs) are competitive solutions for computationally intensive problems [6]. CCMs are custom made for solving a specific kind of problem. The memory and timing issues pertaining to the specific problem are exploited when building a CCM application. This property makes CCMs more efficient than general-purpose processors. CCMs are thus not versatile, in the sense that they can do only a certain set of functions.

*Application Specific Integrated Circuits* or ASICs, designed from the transistor level, are tailor-made to meet the demands of the problem. Several components can be integrated in a single chip; thus, saving board area and the cost of the CCM itself. The highest achievable efficiency in terms of MFLOPS through the concept of CCM can be obtained through ASICs. Failure of the component, advances in chip processing technology, or changes in the algorithm result in the replacement of the present ASIC chip. Thus ASICs might prove to be costly for applications which change rapidly with time. Also, the time and expense involved in developing an ASIC is quite high. With the increasing emphasis on configurable computing, the ideal CCM should be able to change its configuration easily to adapt to the changes. *Field Programmable Gate Arrays* or FPGAs prove useful in this aspect. FPGAs are analogous to clean slates of logic that can be programmed for a specific task. The application can be enhanced with changes in the program software, because FPGAs are reprogrammable. Development of CCMs using FPGAs is comparatively easier than using ASICs due to factors including ease in debugging and quick implementation. However, efficiency and maximum speed cannot be achieved through FPGAs.

One computationally intensive problem is modeling physical problems through the Finite Element Method (FEM). This method involves solving large systems of simultaneous equations. The elements of the matrices represent physical properties such as temperature and pressure, and are typically represented as floating point numbers. Iterative methods are commonly used to solve these linear systems [5]. The core of most iterative methods is matrix-by-vector multiplication.

In the FEM, the physical domain is approximated as a mesh of elements. Each element is associated with a constant number of unknowns. Matrices for each element represent the physical properties of the domain. This matrix is typically sparse and is used to solve for the unknown values of the domain. This requires many operations on matrices of floating point numbers. Solving this problem using a general-purpose computer can be time consuming because of the large number of memory accesses required. Typically, the matrix does not fit in the cache of the general-purpose processors; the accumulated miss penalties are detrimental to the processing speed. CCMs can increase the processing speed because each PE within a CCM typically has its own local memory. Also, many such machines can be run simultaneously using a central control unit without having to share memory. Hence a set of problems can be solved or the problem may be solved in parts in different CCMs.

### 1.2 Research Contributions

An FPGA-based approach for the above problem is investigated in this thesis. Computational units such as floating point adders and multipliers for matrices have been developed to assist in solving the problem. The core computational unit involved in the problem is the multiplication of a square or a rectangular matrix with a vector and accumulation of the product. This multiplier-accumulator component has been developed with the specific problem in mind, but the unit is a generic one and can be used easily for other applications that call for such an operation. The basic framework for the solution is also designed in this thesis. The Wildforce™ board, a configurable computing platform developed by Annapolis Micro Systems, has been used for this purpose.

### 1.3 Thesis Organization

The second chapter gives the necessary background information required for this thesis. Floating-point arithmetic, the architecture of the Wildforce™ board, and a brief description of the FEM are the topics covered in this chapter. The third chapter discusses the studies carried out by other researchers. Chapter Four explains the methodology involved in developing the computational units. The Saxpy operation, which forms the basic unit for matrix-by-vector multiplication, is explained. The matrix-by-vector multiplication, based on the column-based Saxpy multiplication method, is also explained in Chapter Four. In the fifth chapter, the performance of the computational units, and the Wildforce board are discussed. Useful parameters for these components such as MFLOPS are derived. The Poisson's equation and different methods to solve Poisson's equation are discussed in the sixth chapter along with the results obtained when solving these equations on the Wildforce board. The conclusions and suggestions for future work are also discussed in the seventh chapter.

# Chapter 2

# Background

This chapter presents the topics that form the basis for this thesis. The floating-point units, the reconfigurable platform, and the problem that motivated this thesis, the Finite Element Method, are the topics covered in this chapter.

## 2.1. Floating-Point Arithmetic

The floating-point arithmetic units used in this thesis are the adder and the multiplier. The IEEE 754 standard has been chosen as the floating-point format for this application. A 32-bit single precision format, as shown in Figure 2.1, has one sign bit *s*, 8 bits to represent the exponent *e* with a bias of 127, and 23 bits to represent the mantissa *f*. The mantissa is 24-bits long and the leading one is not included in the 23-bit representation. The floating-point (Fp) value is computed as:

$$\mathbf{Fp = -1^{s}\, 2^{(e-127)}\, (1.f)\ .}$$

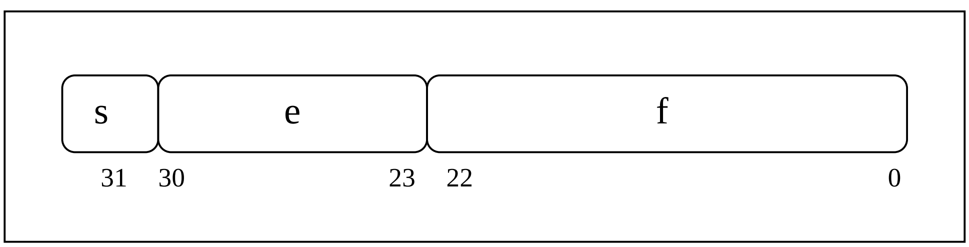


Figure 2.1 32-Bit Floating-Point Format

### 2.1.1. Floating-Point Addition

The algorithm for floating-point addition used in this thesis closely follows the method suggested in [1]. The addition is performed in three stages. The block diagram for the addition algorithm is given in Figure 2.2.

- Stage 1: The two operands are compared and the one with larger magnitude is stored as operand 1. If the two operands have the same exponent, the mantissa fields can simply be added together. To facilitate this, the difference N between the exponents of the two operands is calculated. The sign bits of the two operands are stored in a delay register.
- Stage 2: The mantissa of operand two is shifted N places to the right. The two mantissa fields are added or subtracted depending upon whether both the operands have the same or different sign bits. The exponent and sign bits are stored in delay registers.
- Stage 3: The resultant mantissa is normalized to retain the implicit '1' format and the top 23 bits after the implicit '1' are stored. The exponent is adjusted accordingly. The sign bit of operand one is chosen as the sign bit of the sum. The three separate fields are concatenated to form the sum.

Normalization of the sum of the two mantissa fields is an important step in which the leading '1' is detected and the mantissa field is shifted left until the most significant bit is a '1' [1]. Two methods for normalization have been studied in [1] and the second method, in which the number of places the mantissa is to be shifted left is calculated using a series of *if* statements, is chosen for its smaller size and increased speed of operation. Similarly, *if* statements were employed instead of a *for* loop for shifting the mantissa of operand two N places to the right.

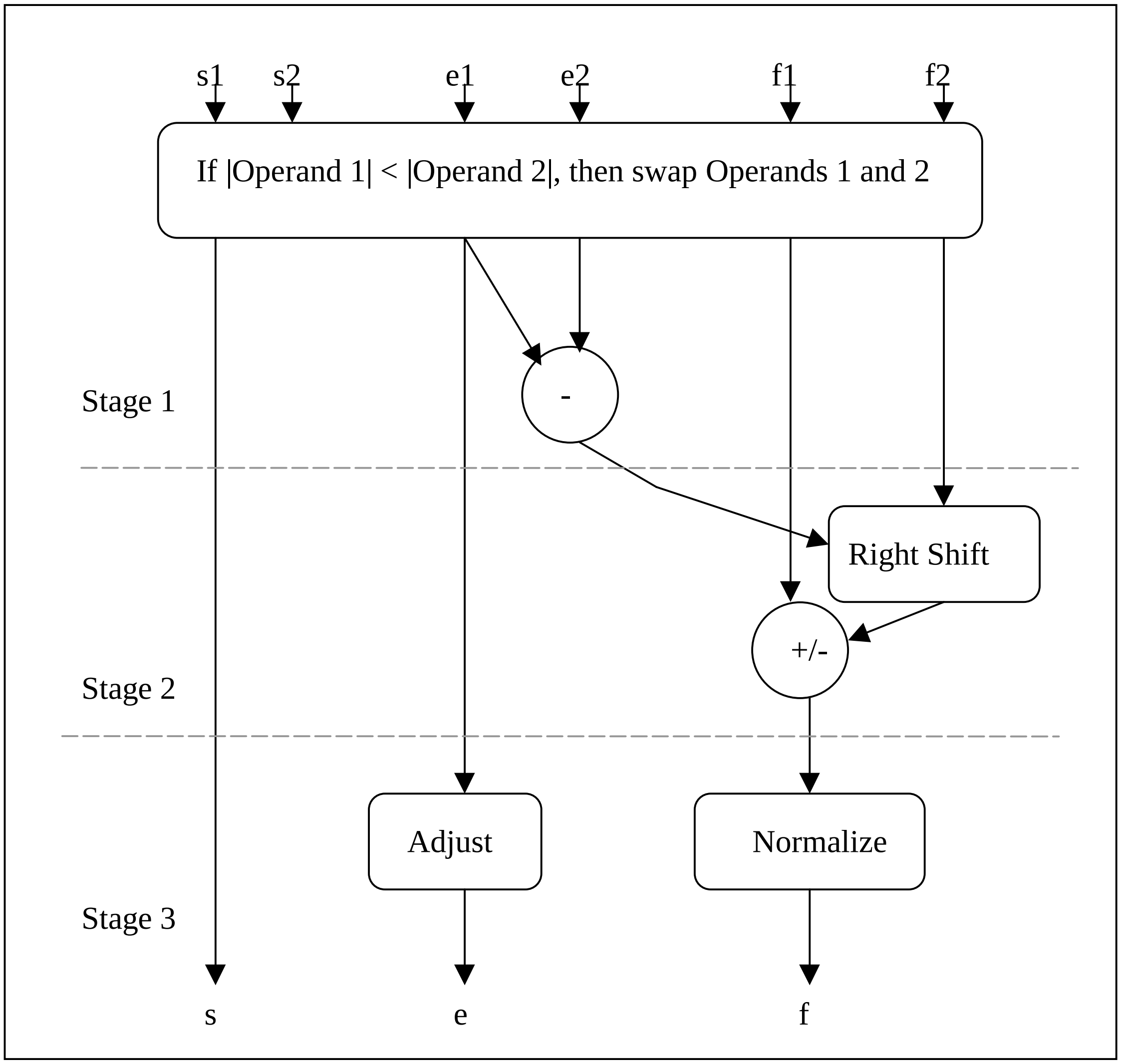


Figure 2.2 32-Bit Floating-Point Addition

### 2.1.2. Floating-point Multiplication

The floating-point multiplier is also a three-stage pipelined design. The algorithm for multiplication is much simpler than that for addition. A block diagram of the algorithm is given in Figure 2.3.

- Stage 1: The three components of each operand are separated and stored into temporary registers. The exponents of the two operands are added.

- Stage 2: Integer multiplication of the two 24-bit mantissas is performed and the top 25 bits are stored. The overflow and underflow conditions are checked. The sign bits are ‘XOR’ed and assigned to the sign bit of the result.
- Stage 3: The resultant mantissa is normalized. The exponent is adjusted depending upon the MSB of the resultant mantissa. The three separate fields are concatenated to form the sum.

Stage 2, where the two 24-bit mantissas are multiplied, is the bottleneck of the algorithm. The integer multiplier available in Synplicity [7] has been used. Though faster implementations suggested in [1] can be used, this was chosen for its acceptable size and speed.

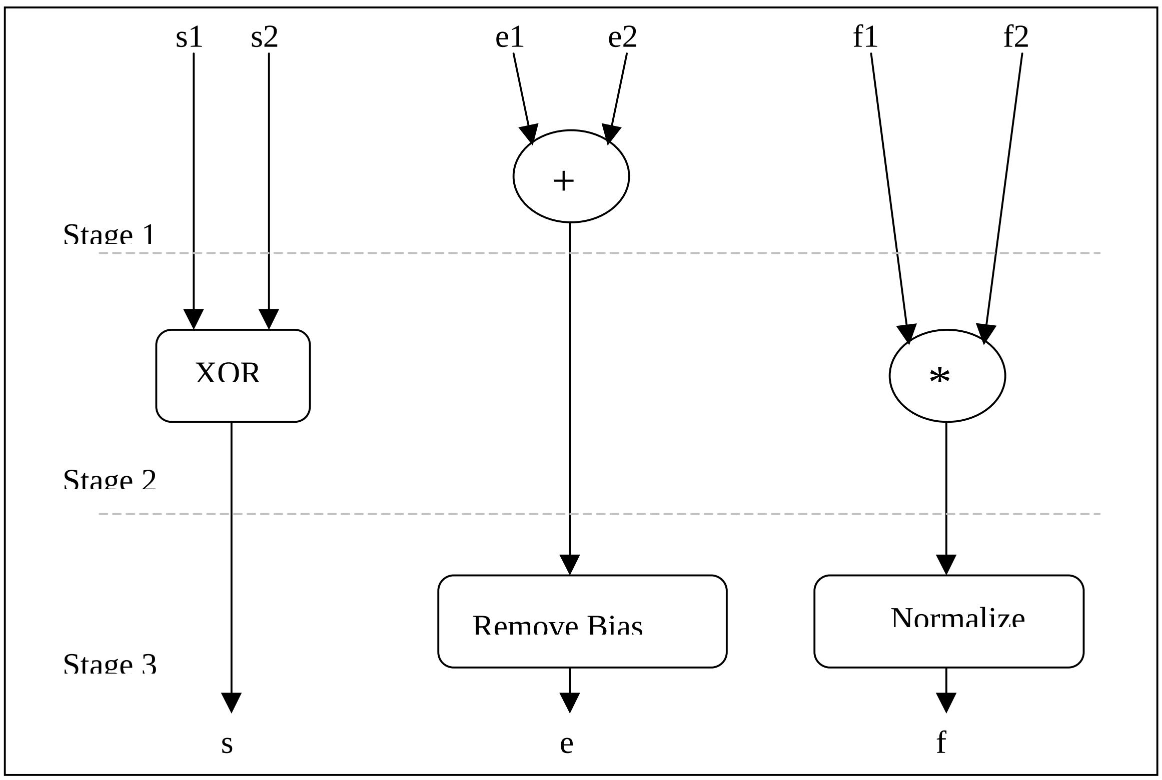


Figure 2.3 32-Bit Floating-Point Multiplication

## 2.2. The Finite Element Method

Two main tasks involved in the study of physical phenomena are the mathematical formulation of the physical process and the numerical analysis of the mathematical model [5]. The mathematical model is developed based on the knowledge of the behavior of the physical model. The mathematical model is often in the form of a set of differential equations relating the various parameters governing the physical system. Mathematical analysis of these equations is accompanied by approximations of the physical system itself. The Finite Element Method (FEM) provides a systematic procedure for approximations over sub-regions of the physical domain [5].

The physical domain is divided into a finite set of sub-domains called *elements*. The elements can be of different shapes such as triangles and rectangles, depending upon the shape of the physical domain and the application. Select points over the element for which the solution is evaluated are called *nodes*. A *finite element*, as defined in [5], is any sub-domain that allows computation of the solution or that provides necessary relations among nodal values. The solution for the physical system is derived from the nodal values and the information on the inter-relation between the nodes. Typically, the number and location of the nodes over the element depend upon the level of approximation desired. The assembly of elements is based on the idea that the solutions are continuous at the inter-element boundaries. The element equations for all the elements in the mesh are derived and are assumed to be of the form

$$\mathbf{[K^e]\{u^e\} = \{F^e\},}$$

where $K^e$ is the element matrix, $u^e$ is the set of dependent variables, and $F^e$ is the set of source variables. The individual element equations are assembled using the inter-element continuity conditions among the primary variables to obtain the equations of the whole problem. The boundary conditions are then imposed on the problem, typically resulting in a large, sparse, nonsingular system of equations to solve. The solution of such large problems can consume significant computational resources.

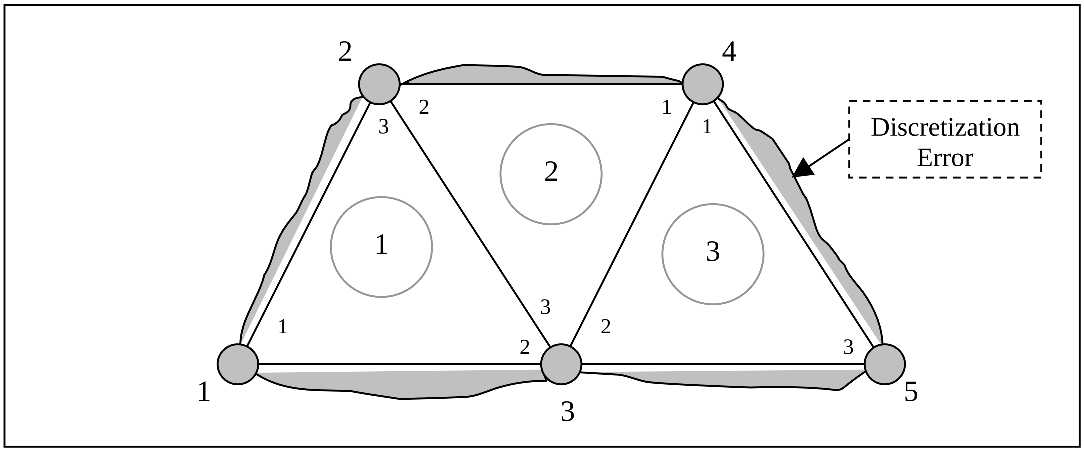


Figure 2.4 Finite Elements of a Physical Domain

To understand the FEM, consider the physical domain in Figure 2.4. A trapezoidal domain is discretized into three triangular finite elements and five global nodes. The global nodes are marked 1 through 5 outside the domain and the element nodes are marked 1 through 3 inside each element. For the sake of simplicity, let each node be associated with one unknown. The coefficient matrix, $K^e$, corresponding to each triangular element is obtained from the physical coordinates and geometric constants associated with its nodes. $K^e$ is a square matrix of size 3x3. Similarly, the vector of unknowns $u^e$ and the vector of source variables $f^e$ are also formed. The global matrix K is a square matrix of dimension 5 corresponding to five global nodes and is formed using the element matrices. The correspondence between some of the elements of the local and the global matrices is given in Table 2.1.

Table 2.1 Correspondence between Global and Local Nodes

| **Global** | **Local** |
|---|---|
| $K_{11}$ | $K^1_{11}$ |
| $K_{23}$ | $K^1_{23} + K^2_{23}$ |
| $K_{25}$ | 0 |
| $K_{33}$ | $K^1_{22} + K^2_{33} + K^3_{22}$ |
| $K_{45}$ | $K^3_{13}$ |

The global vectors *u* and *f* are also generated using the same principle of continuity and equilibrium of variables. The global assembly of equations is then represented in the matrix form $K*u = f$. The vector of unknowns, u, can be solved as $u=K^{-1}*f$.

Typical physical problems that are solved by the FEM include linear elasticity and plasticity problems in 2-dimensional and 3-dimensional domains, heat transfer problems, electrostatics, and magnetostatic applications.

### 2.3. Wildforce Board

The Wildforce board is a configurable computing platform developed by Annapolis Micro Systems Inc. The Wildforce board's architecture and functional capabilities are discussed in this section.

#### 2.3.1 Wildforce Architecture

The Wildforce board has one control-processing element and four array-processing elements (PE). Memory elements can be connected to each PE through a mezzanine connector. The memory elements can be accessed by both the PE and the host CPU through a bus arbitrator called the *Dual Port Memory Controller* (DPMC). The DPMC operates in two modes: Blocked and Arbitrate. In the *Blocked* mode, the PE is blocked from accessing the memory and in the *Arbitrate* mode, the PE is given preference over the host CPU for memory access. All the PEs are connected to an interconnection network, the crossbar, by a 36-bit wide bus. Data from any PE can be broadcast to other PEs through the crossbar. The crossbar can be configured by the control PE to allow broadcast or transfer of data between specific PEs on a nibble basis. Sixteen different configurations for the crossbar can be stored and chosen in real time by the control PE. The host CPU can exchange data between the control PE or with PEs 1 or 4 through FIFO buffers. The control PE and PEs 1 and 4 can communicate with external I/0 as well. Each PE can individually interrupt the host and receive reset signals. Each of the four PEs can also communicate with their neighboring PE through the systolic bus. A block diagram of the Wildforce board is given in Figure 2.5 [9].

### 2.3.2 Interface with the Host CPU

Communication with a Wildforce board can be accomplished via a set of C library routines. These library routines constitute an Application Programming Interface (API) that is available with the board. These API functions are a combination of low-level board communications and reduce the number of Register Transfer Logic (RTL) calls. The kinds of functions available with the API library include programming the PEs, handling interrupts and resets of individual PE, memory and FIFO read and write, and handling the clocks on the board.

### 2.3.3 Programming the Processing Elements

VHSIC Hardware Description Language (VHDL) [8] is used to model the application and the board. VHDL models for the components on the board are available. After simulation, the VHDL models for the processing elements are synthesized to obtain a Xilinx Netlist File, which is a textual format of the hardware design. The Xilinx Netlist File is used to generate a bit stream used to program the FPGA. The host program, written in C language, loads the bitstreams onto the PEs on the Wildforce board and sets up the environment on the board for the VHDL application. The host program handles the interrupts, clocks, and other functions not accessible by the processing elements.

## 2.4 Summary

Floating-point arithmetic, the FEM, and the architecture of the Wildforce board are the three fundamental topics necessary to implement a CCM solution for a FEM solution to a physical problem. The FEM provides an acceptable solution to physical problems and is a computationally intensive method that can be solved efficiently using CCMs. The values in the FEM that represent physical quantities are expressed in a 32-bit floating-point format. A brief study of the three topics was presented in this chapter.

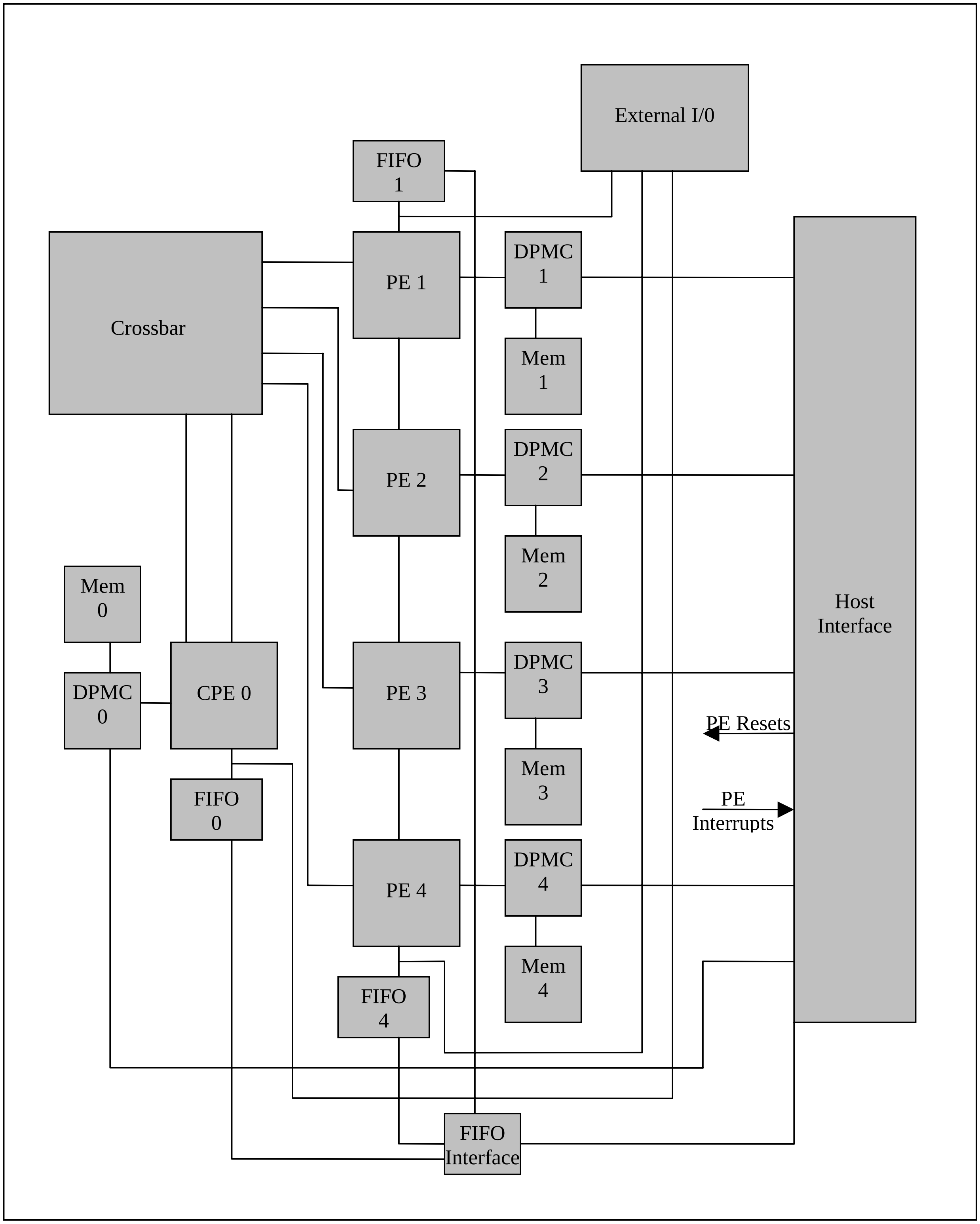


Figure 2.5 Block Diagram of the Wildforce Board

# Chapter 3
# Related Work

This chapter discusses the contribution of other researchers in the field of CCMs. The theory of configurable computing has interested many researchers. A typical area where the theory of configurable computing can be applied is acceleration of computationally intensive mathematical solutions for physical problems. Configurable computing has been identified as a viable solution because of the development of high density FPGAs and related developments in the IC industry. Many components can now be mapped onto a single chip. With the present commercial FPGAs available such as Xilinx 40250, it is now possible to map up to 16 32-bit floating-point operators on a single chip. Reconfigurable platforms have been developed, making design relatively easy and fast. A few such research projects are covered in this chapter. Their methodology and implementation techniques are studied and compared with the work in this thesis.

## 3.1. Floating-Point Operations on FPGAs

Floating-point arithmetic units on FPGAs are often limited by the size of the FPGAs available. Most scientific computing algorithms require a floating-point format for the parameters involved. In their work, Ligon, et al [2] have investigated the practicality of floating-point operations on FPGAs. Floating-point addition and multiplication have been implemented and their performance studied for different FPGAs with varying densities. These single-precision units are implemented as deep pipelines to reduce cycle time. The following sections discuss the various elements of Ligon's findings.

### 3.1.1. Floating-Point Units

In the floating-point multiplier described in [2], the largest component used is the 24-bit unsigned integer multiplier. Multiplication of two 24-bit numbers results in a 48-bit number, which requires a large amount of chip area. In their work, Ligon, et al have used bit serial multiplication and the Booth recoding digit serial approach. In the Booth recoding algorithm, two bits are resolved per clock cycle using a single adder as compared to one bit per clock cycle in the bit serial algorithm. The general algorithm used for the floating-point multiplier is the one suggested by the IEEE standard 754 as explained in Chapter 2. The sign bits of the operands are 'XOR'ed, the exponents are added, and the mantissas are multiplied. The result is then normalized and rounded. The 24-bit mantissa multiplication results in a 48-bit number. The top 25 bits are taken and the exponent is normalized depending on the most significant bit. Of the few rounding techniques proposed by IEEE standard 754 [10], the default method has been chosen for their implementation. Rounding is performed before normalization. Overflow and underflow conditions are handled by setting the result as zero. Setting the result to zero for the overflow condition does not handle the condition accurately.

The floating-point adder follows the algorithm explained in Section 2.1.1. The larger of the two operands is chosen as operand1. The difference N between the two exponents is calculated and the mantissa of operand2 is shifted right N places. The mantissas are added using integer addition units and the sum is rounded, normalized, and the exponent is adjusted. The rounding is done as for the multiplier and the underflow and overflow conditions are handled as for the multiplication.

### 3.1.2. Resource Requirements and Performance

The adder and multiplier units [2] were implemented on different Xilinx devices. The resource requirements are expressed in terms of four input lookup tables (4- LUTs). The Booth-based multiplier needs 283 4-LUTs and 155 flip-flops, and the bit serial algorithm needs 249 4-LUTs and 149 flip-flops. The Booth-recoding-based multipliers required about half as many cycles to produce a result as the bit serial algorithm. This may indicate that the Booth-based algorithm is the preferred one, but the resource requirements may play a role in this choice. The 2-stage pipelined Booth-recoding multiplier has a 9-cycle pipeline stage latency while a 3-stage pipelined bit serial multiplier has 10-stage cycle latency [2]. Any significant advantage to either of the multiplier implementations has not been identified [2]. The implementations were tested at 30 MHz and the peak performance is 5 MFLOPs. The authors have extrapolated from these results to the performance that could be achieved from devices of higher densities. A performance of 140 MFLOPS per FPGA at 40 MHz, for a Xilinx 40250XV that can hold more multipliers interconnected in some fashion is expected.

The fully pipelined implementation of the adder requires about 629 4LUTs and 671 flip-flops. These numbers are more than for the multiplier because the iterative multiplication method repeatedly reuses a single adder, while a large number of operations have to be performed on the operands in the addition [2]. Because of the large number of flip-flops available in the chip, this constitutes only a small portion of the flip-flops available. The performance of the adder has been noted as 30 MFLOPS at 30 MHz. It has been noted that a Xilinx 4020 can accommodate only one adder while the higher end devices like Xilinx 40250XV-09 can support roughly 15 interconnected adders. This may give us a performance of 600 MFLOPs at 40 MHz [2]. However, this is only an extrapolation of the existing results.

### 3.1.3. Matrix Multiplication

The large amount of parallelism in matrix multiplication algorithms can be exploited in their implementation on re-configurable platforms. The key component in the matrix multiplication is the Multiplier-Accumulator or the MAC unit. The speed of operation and efficiency of the system depends on the speed and the number of MACs available on the chip. An example of A*B = C where A, B and C are square matrices of dimension N has been considered in [2]. In their algorithm, a different column of B is sent to each of K different MACs. K is usually a multiple of either the number of modules or the number of chips available. The columns of B are stored in temporary registers. The rows of A are broadcast to all the MACs. This results in K rows of C. The process is repeated for N/K times for a total computation of $(N/K)*T_c + N^2*T_m$, where Tc is the time required to place K columns of B on the board and Tm is the total time to consume an element. The high cost associated with the broadcast of a 32-bit data element suggests the use of a large number of MAC units. But this increase will result in more complicated control logic. Also, the speeds of the units are limited by the I/O and RAM bandwidth.

A single MAC unit occupies most of the chip area in a Xilinx 4020E. The best case scenarios as reported in [3] can be achieved by splitting a single MAC across the 2 chips on a single XMOD of the GigaOps board, giving a total of 4 MAC units. It is expected that using a 6-cycle device at 30 MHz, a 1024x1024 element matrix multiply can be done in about a minute [2]. This performance is still quite competitive compared to 88 seconds on a 150 MHz Pentium processor and 10.6 seconds on a 500 MHz Alpha processor.

The algorithm for matrix multiplication in [2] is different from the approach used in this thesis. The multiplication considered in this thesis is $A*x = y$ where A is a sparse matrix and x and y are vectors. The matrix-by-vector multiplier uses only one MAC unit. The matrix A and vectors x and y are available in the local memory. The vector c is computed using a Saxpy-based algorithm or column based algorithm where the vector y is computed partially every MAC cycle. An element of x is read and multiplied with one column of A and the sum is accumulated in temporary registers. The sum is then written back into the memory. This methodology will be discussed in detail in later chapters. The adders and multipliers developed in this thesis are very similar to the ones described earlier. However, a less stringent method for rounding has been used.

### 3.2. CCM Implementation for a Physical System

In this section, a CCM implementation for simulation of discretized domain physical systems is explained. Simulation of physical systems is a computationally intensive process and is an integral part of research in the area of physical sciences [3]. Experimental measurement of physical parameters is not always feasible. An accurate model of the physical system and its behavior has to be developed. Paar [3], in his research, had developed a CCM solution for heat transfer behavior in physical systems.

The application simulates two dimensional heat transfer problems consisting of conductive, convective, and direct flux modes [3]. The CCM is composed of individual processors controlled by a single control unit.

In most cases, the physical domain is first discretized by subdividing the domain into a finite number of elements forming a mesh. The mesh approximates the continuous nature of the domain as discrete points, which are associated with a set of physical properties. The behavior of the physical system is described by partial differential equations. The physical system can then be modeled accurately by solving these partial differential equations. In his work, Paar has solved heat transfer equations for analysis of conduction and/or convection [11]. The partial differential equations that govern the behavior of heat transfer were first identified and studied. The nodes in the mesh are categorized into four different kinds depending upon their relative positions. Different equations are identified for each of these four kinds of nodes.

### 3.2.1 Implementation and Results

A reconfigurable platform called the Splash-2 was used in the work. The Splash-2 board has a set of seventeen Xilinx 4010 devices attached to their respective local memories. These devices are interconnected through a crossbar; a systolic bus runs through all the devices. A control unit, mapped onto another device, regulates the flow of data and controls the other devices. The problem is partitioned into modules and each module is assigned to a processing element (PE). A control-processing element governs the data flow and instruction issue to all the PEs. Hence an SIMD multiprocessing approach was used. Each PE is responsible for calculating one node at a time. The entire nodal mesh is organized as sub-arrays. Calculations for any node need information from the four surrounding nodes. Hence, the above and below nodes are mapped onto the same PE and the left and right nodes are mapped onto adjacent PEs. The memory operations limit the performance of the computations. The memory operations take 10 cycles and the computations can be completed within these 10 cycles.

The range of numbers required to represent physical properties such as temperature calls for a floating-point format. The IEEE standard 754 needs a 32-bit format for the data while the data bus available on the Splash-2 board is only 16 bits wide. Instead of having two memory operations for a single data read, a custom 16 bit floating point format was used. The temperature, measured in Kelvin, will always be positive, so the sign bit is not necessary. The number of bits for the exponent has been reduced from 8 to 6. The mantissa was represented in 10 bits, which reduces the accuracy of the numbers involved but is traded for simplicity of the entire algorithm. Also, the implied '1' in the IEEE format is actually incorporated in the mantissa itself in the custom format. The floating-point addition and multiplication are similar to the ones presented earlier, but have been simplified due to the simplified floating-point representation. A two-stage pipeline implementation was used instead of the three-stage pipelines for the arithmetic units.

The performance increases with the number of Splash-2 boards and the clock rate. With two Splash-2 boards, the speed-up achieved when compared to the performance of a Sparc-2, is 124 at 1 MHz. The speed-up increases up to 1488 when run at 12 MHz for a 1024x1024-mesh [3]. The performance expressed in MFLOPS is 22 for 2 Splash-2 boards when run at 1 MHz, and 269 when run at 12 MHz. Paar [3] has compared these results with those achievable from Digital Signal Processors (DSP). The Analog Devices SHARC™ processor [12] has a peak performance of 120 MFLOPs [3]. Paar argues that this performance, though comparable to what is achieved through a single Splash-2 board, is actually overrated. He states that this performance can be achieved through MAC instructions. The heat equations use only one such unit and other units are single streamlined operations. Hence, this performance cannot be achieved using the DSP.

This work has clearly demonstrated the use of a CCM for physical system simulations. This thesis is aimed at implementing such a problem on a CCM. The Finite Element Method (FEM) provides accurate models of physical systems similar to the Finite Difference Method used in [3]. The core computational unit in the FEM is the matrix-by-vector multiplication. The matrix-by-vector multiplication is developed in this thesis and used to solve physical problems through the FEM. The reconfigurable platform used in this thesis allows more elaborate design. The bus width on the board is 32 bits and hence the IEEE format 754 can be used for floating point operations; this considerably improves precision with a slight increase in computing complexity. The density of the Xilinx devices on the board is also higher. This aids in mapping more hardware components onto a single chip, thus reducing the overhead required for device communications.

### 3.3. FIR Implementation on the Wildforce Board

The architecture and functional capabilities of the Wildforce board were discussed in Section 2.4. In this section, we discuss the Annapolis Micro Systems Wildforce™ board CCM. The implementation of a linear phase Finite Impulse Response (FIR) filter with complex convolution filters that can support channel impulse response profiles [4] is studied in this section. The FIR is one of the main functions often evaluated in the field of signal processing. Digital filters are necessary in modeling the signal propagation in media. The FIR filter, which is a non-recursive filter, is easily realizable in hardware.

#### 3.3.1. Floating-Point Units

Floating-point arithmetic units, such as the adder and the multiplier, are necessary for a FIR implementation. The arithmetic units have been designed as described in Chapter 2. A 32-bit floating-point format was used in [4]. However, a deep pipeline was used for the adder implementation to improve throughput of the unit. The adder unit has a latency of eight clock cycles. The main difference in the adder unit developed in [4] and the one described in [1] is the use of a different approach for the leading ‘1’ detection. The leading ‘1’ detection step was broken down into two steps. In the first step, the six nibbles of the mantissa are checked for the leading ‘1’. A six-bit word is formed based on the operation. In the second stage, the six-bit word is used for normalizing the result.

The multiplier unit closely follows the method discussed in Chapter 2. The bottleneck of the multiplication, the integer multiplication of the two mantissas, was broken down to a number of simpler steps. Again, this increases the latency in the pipeline, but facilitates throughput. Also, the deep pipeline method has been employed to accommodate a strict resource budget because the multiplier occupies almost half of the

Xilinx 4036EX. The 24x24-integer multiplier in [4] utilizes Booth-recoding and inserts RLOC information and pipeline stages to preserve the routing, timing, and size of the multiplier [4]. Two bits of one multiplicand are issued every cycle starting with the least significant bits. The result is generated in parts where the most significant bits of the result are generated after all the bits of the operand have been issued. Truncation is used instead of rounding to save resources.

### 3.3.2. Implementation on the Wildforce Board and Results

The four PEs on the Wildforce form the computational array. The necessary logic for coordinating the functioning of the PEs is programmed into the control PE. Data flows from the host computer to the input FIFO. The control-processing element feeds this data to the first PE. The data passes through all the PEs through the systolic bus. The results are sent to the host computer through the output FIFO, which then writes them to the data storage device.

The performance of each PE running at 20 MHz is 40 MFLOPS. A maximum of 160 MFLOPs can be achieved if all four PEs in the systolic array are used. Extrapolating these results, a performance of 640 MFLOPs can be achieved when four Wildforce boards are used for a single host [4]. Feeding data back into the system can be used to simulate higher order FIRs but at an increased complexity at the host level.

# Chapter 4

# Development of Matrix-by-Vector Multipliers

The core computation of the iterative solution of the linear system of equations arising in the FEM is sparse matrix-by-vector multiplication. Sparse matrix-by-vector multiplication is computationally intensive and can be performed in a variety of ways. One method is to use the element matrices that form the global sparse matrix to perform matrix-by-vector multiplications. This chapter discusses the methodology involved in the development of a matrix-by-vector multiplier for small matrices. The floating-point arithmetic units are discussed in the following subsection. The basic building block of the matrix-by-vector multiplier, the Saxpy-based Multiplier-Accumulator (MAC), is discussed in Section 4.3.2. Matrix-by-vector multiplication is explained in Section 4.3. The inner product method and saxpy-based methods to perform the matrix-by-vector multiplication are discussed. Two implementations of the Saxpy-based matrix-by-vector multiplier are possible depending upon the source and nature of the vectors. They are discussed in Section 4.3.3. An algorithm for enhancing the saxpy-based matrix-by-vector multiplication is discussed in Section 4.3.5.

## 4.1 Floating-Point Units

The 32-bit floating-point addition and multiplication used in this chapter follow the algorithm described in Chapter 2. The depth of the pipeline has a direct impact on the latency of the computation units. In this thesis, a short pipeline is used because the sizes of the matrices are small and more timely results are desired. A 3-stage pipelined implementation of the algorithm is used.

The approach used for the leading ‘1’ detection in the adder is the exhaustive search method described in [1]. The method was chosen over the method used in [4] because of its simplicity and short latency. In the 32-bit floating-point multiplier, the default integer multiplication available with the synthesis tool has been used for the mantissa multiplication. The rounding technique used in both the adder and multiplier is truncation. Though it introduces error in the final result, truncation is chosen for its simplicity. The implementation is not severely limited by resource constraints because high density FPGAs are used. Hence, simplicity in design has been preferred over hardware simplicity. However, the simplicity in design did not hinder the speed with which the units can be operated. The three-stage multiplier, which occupied about 50% of the Xilinx 4036EX, runs at 30 MHz.

Each stage of the 3-stage pipelined design of the adder and multiplier is modeled using the *process* constructs in VHSIC Hardware Description Language (VHDL) [8]. Each stage is modeled as a process, which executes when its inputs change. The results of each stage computed through combinational logic are latched and fed to the next stage. By latching the results of each stage, the intermediate results of one stage do not falsely trigger the next stage.

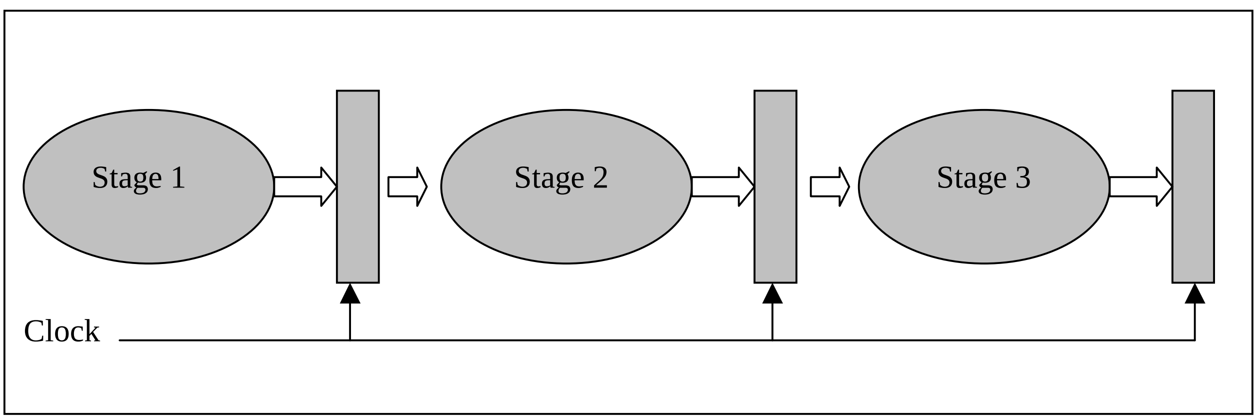


Figure 4.1 Three Stage Pipeline

**4.2 The Saxpy Operation**

The Saxpy operation [13], where a vector $x$ is multiplied by a scalar $\alpha$ and added to another vector $y$, forms the basic building block of the matrix-by-vector multiplication. The block diagram for the Saxpy operation is given in Figure 4.2. One input, the scalar value $\alpha$, of the multiplier remains constant throughout the operation. An element of the vector $x$ is fed into the multiplier and the product is fed as one of the inputs to the adder, the other input being an element of vector $y$.

From Figure 4.2, we find that the *Saxpy* operation can be performed as a series of multiply-and-add operations. A single $(\alpha * x_i + y_i)$ operation is performed after the previous operation is completed and the answer written into the attached PE memory.

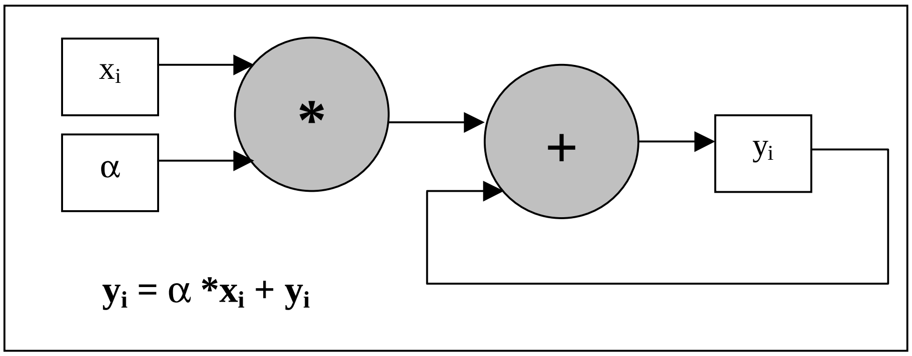


Figure 4.2 The Saxpy Operation

Thus, a single iteration requires two memory reads and one memory write. The final answer $y_i$ is available only six clock cycles after the inputs are available. The timing sequence of the operations is given in Table 4.1. Element $y_i$ is read at time step 3 and should be available to the adder until time step 5 when the addition starts. The next operation can start only at time step 11, after writing the answer of the previous operation in step 10. The next valid product will be available at time step 14. We find that the pipelines are empty for most of the time. One memory read can be performed every clock cycle and the adder and multiplier units are capable of producing a result every clock cycle. The individual multiply-and-add operations are independent of each other and hence can be overlapped in time.

The pipelines can be used more efficiently if the lengths of the vectors are small enough to be stored in temporary registers on the PE. The elements of the vector $x$ can be given serially to the multiplier. While adding the product with the vector $y$, the corresponding $y_i$ should be given as input to the adder. The results are stored in temporary registers and the written into the memory when all the multiply-and-add operations are complete. This method requires a large number of temporary registers that increase the chip area, but has higher throughput than the previous method. The length of the vectors must be known in order to declare the number of temporary registers.

Table 4.1 Timing Sequence for the Saxpy Operation

| Time steps ➔ / Signal name | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $y_i$ | | | 1 | | | | | | | |
| $x_i$ | | 1 | | | | | | | | |
| $\alpha$ | 1 | | | | | | | | | |
| Product | | | | | 1 | | | | | |
| Result | | | | | | | | 1 | | |

### 4.3 Matrix-by-Vector Multiplication

The matrix-by-vector multiplication can be represented as

$$A*x = y, \text{ or}$$

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \ldots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \ldots & a_{1,n-1} \\ & & & \\ a_{m-1,0} & a_{m-1,1} & \ldots & a_{m-1,n-1} \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ . \\ . \\ . \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ . \\ . \\ y_{m-1} \end{pmatrix},$$

where A is a matrix of dimension *mxn* and *x* is a vector of length *n* and *y* is a vector of length *m*.

#### 4.3.1 The Inner Product Method

Matrix-by-vector multiplication can be implemented in a variety of ways [15]. In the inner product method, a row of the matrix is multiplied with the vector *x* and the products are accumulated to generate one element of the resultant vector as

$$y_i = \sum_j A_{ij} * x_j \, .$$

In this method, the vector *x* is read *m* number of times. The number of memory reads involved in this method is *2*m*n* while there are only [*m*n + n* ] elements to be read. To compute one element in the result vector, *n* MAC operations are required. [1] Hence it takes *m*n* MAC operations to complete the matrix-by-vector multiplication. Each MAC operation takes 10 cycles, including 4 clock cycles for 2 memory reads. The timing sequence for the inner-product method closely follows the timing sequence given in Table 4.2. In Table 4.2 the initial memory bus access delay and memory read delay are not shown. Each MAC operation is associated with two memory reads; the multiplier and adder pipelines have to be stalled until the two operands are read. The next set of operands is read only after the final result from the previous MAC operation is available. Hence, the multiplication is performed as a series of non-overlapping MAC operations. The number of floating-point operations performed is *2mn*. The performance that can be achieved for a 3x3 matrix is 2.7 MFLOPS at 15 MHz. A marginal improvement in performance can be obtained if MAC operations are overlapped. The next set of reads can be performed at time step 5 so that the new value of $y_0$ is available for the next *add* operation at time step 9.

Table 4.2. Timing Sequence for the Inner Product Method

| Time step / Signal | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Mem Read | $x_0$ | $A_{00}$ | | | | | | | $x_1$ | $A_{01}$ | | | | |
| Mult_Op1 | $x_0$ | $x_0$ | | | | | | | $x_1$ | $x_1$ | | | | |
| Mult_Op2 | | $A_{00}$ | | | | | | | | $A_{01}$ | | | | |
| Add_Op1 | | | | | p | | | | | | | | p | |
| Add_Op2 | | | | | $y_0$ | | | | | | | | $y_0$ | |
| Result | | | | | | | | $y_0$ | | | | | | |

[1] This operation requires *n* multiplications and *n-1* additions if an initial value for $y_i$ is not given.

### 4.3.2 Saxpy-based Multiplication

A more efficient method for performing matrix-by-vector multiplication is Saxpy-based multiplication. In this method, one column of the matrix is multiplied by an element of the vector *x* and the resultant vector is accumulated. *N* Saxpy operations are performed. The block diagram of this method is given in Figure 4.3.

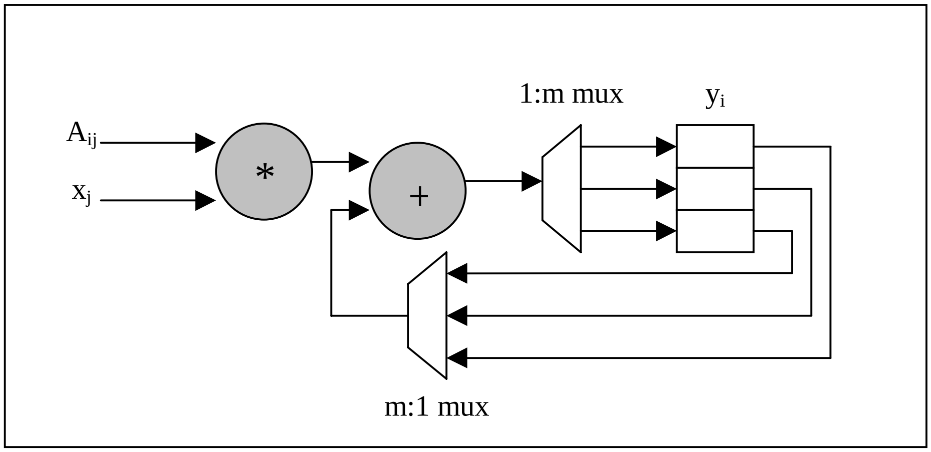


Figure 4.3 Saxpy-Based Matrix-by-Vector Multiplication

An element of the vector *x*, say $x_j$, is read and given as the first operand of the multiplier. The elements of the $j^{th}$ column of matrix *A* are read and serially fed as the second operand. The product is given to the adder. The temporary registers on the PE are multiplexed and the $i^{th}$ register is given as the input to the adder when the $i^{th}$ product is available. The sum is stored in the corresponding temporary register. The timing sequence for a matrix of size 3x2 is given in Table 4.3. In Table 4.3, Add_Op1 is the output of the multiplier and a '?' represents a 'don't-care' or an invalid entry. The initial memory bus access delay and the memory read delay overheads are not shown in Table 4.3. The adder and multiplier pipelines are full most of the time. After the pipeline is filled initially, *m* valid answers are generated for every *m+1* clock cycles because *m+1* operands are required for *m* computations. As the number of rows increases, the throughput for the pipelines improves.

Table 4.3. Timing Sequence for Matrix-by-Vector Multiplication

| Time step / Signal | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Mem Read | $x_0$ | $A_{00}$ | $A_{10}$ | $A_{20}$ | $x_1$ | $A_{01}$ | $A_{11}$ | $A_{21}$ | | | | | | |
| Mult_Op1 | $x_0$ | $x_0$ | $x_0$ | $x_0$ | $x_1$ | $x_1$ | $x_1$ | $x_1$ | | | | | | |
| Mult_Op2 | ? | $A_{00}$ | $A_{10}$ | $A_{20}$ | ? | $A_{01}$ | $A_{11}$ | $A_{21}$ | | | | | | |
| Add_Op1 | | | | ? | $p_0$ | $p_1$ | $p_2$ | ? | $p_0$ | $p_1$ | $p_2$ | | | |
| Add_Op2 | | | | ? | $y_0$ | $y_1$ | $y_2$ | ? | $y_0$ | $y_1$ | $y_2$ | | | |
| Result | | | | | | | ? | $y_0$ | $y_1$ | $y_2$ | ? | $y_0$ | $y_1$ | $y_2$ |

The number of memory reads required by this method is equal to $(m+1)*n$, which is the number of elements involved in the matrix-by-vector multiplication. The number of memory writes is equal to $m$. Some applications that require matrix-by-vector multiplication specify initial values for the vector $y$. The matrix-by-vector multiplication can then be written as

$$y = A*x + y.$$

For such applications, the multiplication process remains the same but the vector $y$ is read and stored in the temporary registers before the process is initiated. The number of memory reads increases to $(m+2)*n$ but is equal to the number of operands required. The chip area required for this method is more than that required in Section 4.3.1 because $m$ 32-bit registers are required to store the result. The increase in chip area is traded for throughput.

### 4.3.3 Variations in Saxpy-Based Multiplication

In certain applications, the vectors $x$ and $y$ are not contiguous and may be a part of a larger vector. In such cases, the addresses of the elements of the $x$ and $y$ vectors have to be read initially. Depending upon the source and nature of the matrices and vectors, two implementations are considered in this thesis.

- *Implementation 1*: The number of columns of the matrix is defined by the user, but the number of rows is constant and defined prior to programming the processing element. The vectors $x$ and $y$ are stored serially in the PE memory. In this implementation a set of such matrix-by-vector multiplications can be performed constrained only by the amount of data that can be written into the PE memory.
- *Implementation 2*: The vectors $x_e$ and $y_e$ are chosen from a master vector x and y, which are stored in the PE memory. The matrix-by-vector multiplication can be represented as

  $$y[index] = A*x[index].$$

  The addresses of the elements to be chosen are stored as another array in the PE memory. The knowledge of the number of columns is required to store the addresses of the elements of $y_e$. A square matrix of known size is assumed for this implementation. This implementation takes more chip area than implementation 1. Again, in this implementation, several such matrix-by-vector multiplications can be performed in series.

#### 4.3.4 Resource Requirements and Performance

The maximum frequency returned by the Xilinx tools for any of the above implementations is approximately 6 MHz. This is a conservative figure because the Xilinx tools assume worst case conditions. In practice, the maximum speed that can be achieved for the matrix-by-vector implementations is 15 MHz.

On the Wildforce board, the memory bus is granted to the processing element three clock cycles after the request is made. Valid data is available on the memory data bus three clock cycles after the address is placed on the address bus. The number of floating-point operations for one matrix-by-vector multiplication in implementations 1 and 2 is $2mn$ where $mxn$ is the size of the matrix. The number of memory accesses in implementation 1 is $[mn + 2m + n]$. The number of memory accesses in implementation 2 is $[mn+3m+2n]$. The ratio of number of floating-point operations to the number of memory accesses for implementations 1 and 2 for different matrix sizes is given in Table 4.4.

Table 4.4 Ratio of Floating-Point Operations to Memory Accesses

| Matrix size | Implementation 1 (Number of columns = 6) | Implementation 2 |
|---|---|---|
| 3 | 1.20 | 0.75 |
| 4 | 1.26 | 0.89 |
| 6 | 1.33 | 1.09 |

The number of clock cycles to complete one matrix-by-vector multiplication through implementation 1 is *[(m+1)*n + (m+1)+ 12]*, where *m* is the number of rows and *n* is the number of columns. The overhead of *[(m+1)+12]* clock cycles comprises 3 cycles for memory bus access, 3 cycles for memory access delay, 6 clock cycles for the pipeline latency, and *[m+1]* cycles for m memory writes. The memory bus access overhead is incurred only once while performing a set of *N* matrix-by-vector multiplications. The number of clock cycles to perform *N* matrix-by-vector multiplications is *[N {(m+1)*n + (m+1) +11} + 3]*. After the memory bus is granted to the PE, it is held until the completion of the matrix-by-vector multiplication. Hence the memory bus access overhead is incurred only once per matrix-by-vector multiplication. An additional 2 clock cycles per matrix-by-vector multiplication is required to compute the remaining number of iterations. The number of floating-point operations performed for a matrix size of *mxn* is *N(2*m*n)*, *N(m*n)* multiplications and *N(m*n)* additions. The performance in terms of MFLOPS increases for larger values of *m* and *n.*

For implementation 2, the number of clock cycles to complete one matrix-by-vector multiplication is an additional *(2m+3)* cycles more than implementation 1. This is because the addresses of the elements of $x_e$ and $y_e$ are read. The number of floating-point operations performed is *(2*m*n*N).* Table 4.5 gives the device utilization summary and expected performance in MFLOPS for Xilinx 4062XL for different matrix sizes. The value of *N* is taken as 1 in Table 4.5 to give the minimum achievable performance.

Table 4.5 Device Utilization and Performance Summary for Xilinx 4062XL at 15 MHz

| **Matrix size** | 3 | | 4 | | 6 | |
|---|---|---|---|---|---|---|
| **Implementation** | 1 | 2 | 1 | 2 | 1 | 2 |
| CLBs | 1325 (57%) | 1414 (61%) | 1367 (59%) | 1417 (61%) | 1485 (64%) | 1686 (73%) |
| Latches | 52 (1%) | 54 (1%) | 52 (1%) | 54 (1%) | 52 (1%) | 54 (1%) |
| CLB Flops | 668 (14%) | 721 (15%) | 703 (15%) | 791 (17%) | 767 (16%) | 938 (20%) |
| 4-LUTs | 2369 (51%) | 2440 (52%) | 2416 (52%) | 2395 (51%) | 2675 (58%) | 2808 ( 60%) |
| 3-LUTs | 216 (9%) | 241 (10%) | 283 (12%) | 317 (13%) | 291 (12%) | 400 (17%) |
| MFLOPS | 9 | 6.92 | 12.3 | 9.6 | 17.14 | 13.85 |

### 4.3.5 Enhancing the Saxpy-Based Multiplication

Table 4.2 suggests that if the memory reads are performed faster than the adder and multiplier pipelines, then valid answers can be generated every clock cycle once the pipeline is filled. In this section, a method is suggested to achieve this task.

The memory reads can be performed at a much faster rate than the arithmetic operations. The arithmetic units are operated at a frequency one-third that of the process clock. The finite state machine, which controls the memory reads and writes, is operated at the frequency of the process clock. Memory reads are spaced in time such that both the operands required by the arithmetic units are available at every rising edge of the arithmetic clock. Table 4.6 gives the timing sequence for matrix-by-vector multiplication for a matrix of size 3x2 using this method. Again, the initial memory bus access delay and memory read delay are not shown and are assumed to have occurred before $T_1$-time step 1. Time steps $T_2$ are the time steps for the arithmetic clock. It takes 12 $T_2$-time steps to complete the multiplication as compared to 14 by the earlier method. Also, the adder and multiplier pipelines generate results every $T_2$ clock cycle once the pipeline is filled. The number of $T_2$-time steps required to complete the multiplication by this method will be *n* less than the number required by the method described in section 4.2.2 where *n* is the number of columns in the matrix. An added advantage of using this method is that another MAC unit can be used to perform another set of matrix-by-vector multiplication; in addition, the time to read the initial address vectors for *x* and *y* will be smaller.

To use a second MAC unit, vector *x* is read in $T_1$-time steps 3 and 12 and the six elements of the matrix *A* are read in $T_1$-time steps 4, 7, 9, 13, 16 and 19. The time steps for the second set of multipliers and adders start at $T_1$-time step 4. Hence a third clock $T_3$ is required which lags the $T_2$ by a phase of $240^o$. The results for the second set are available at $T_1$ time step 37. The timing sequence for using a second MAC unit is shown in Table 4.6. The control logic for this application is complex and the chip area required is also large, but the performance that can be achieved through this application justifies the expense. In the Xilinx 4062XL, two MACs controlled by a single control state machine can be implemented. The arithmetic units are operated at one-third the frequency of the process clock and the board can be expected to operate at three times the clock frequency used in section 4.3.2. The number of floating-point operations that are performed for two matrix-by-vector multiplications is *4mn*. The performance that can be achieved for a matrix size of 3x3 at 45 MHz is 21.6 MFLOPS. When this implementation is mapped onto all the five PEs on the Wildforce board, a performance of 108 MFLOPS can be achieved. For a matrix size of 6x6, a maximum performance of 196 MFLOPS can be achieved when all five PEs are used.

Table 4.6 Timing Sequence for Enhanced Saxpy-based Multiplication

| Time Step $T_1$ / Signal | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Memory Read | | $x_0$ | $A_{00}$ | $x_0$ | $A_{00}$ | $A_{10}$ | $A_{10}$ | | $A_{20}$ | $A_{20}$ | $x_1$ | $A_{01}$ | $x_1$ | $A_{01}$ | $A_{11}$ | $A_{11}$ | | $A_{21}$ | $A_{21}$ | | | | |
| Time step $T_2$ | | | 1 | | | 2 | | | 3 | | | 4 | | | 5 | | | 6 | | | 7 | | |
| Mult_Op1 | **MAC 1** | | $x_0$ | | | $x_0$ | | | $x_0$ | | | $x_1$ | | | $x_1$ | | | $x_1$ | | | | | |
| Mult_Op2 | | | $A_{00}$ | | | $A_{10}$ | | | $A_{20}$ | | | $A_{01}$ | | | $A_{11}$ | | | $A_{21}$ | | | | | |
| Add_Op1 | | | | | | | | | | | | $p_0$ | | | $p_1$ | | | $p_2$ | | | $p_0$ | | |
| Add_Op2 | | | | | | | | | | | | $y_0$ | | | $y_1$ | | | $y_2$ | | | $y_0$ | | |
| Result | | | | | | | | | | | | | | | | | | | | | $y_0$ | | |
| Mult_Op1 | **MAC 2** | | | | $x_0$ | | | $x_0$ | | | $x_0$ | | | $x_1$ | | | $x_1$ | | | $x_1$ | | | |
| Mult_Op2 | | | | | $A_{00}$ | | | $A_{10}$ | | | $A_{20}$ | | | $A_{01}$ | | | $A_{11}$ | | | $A_{21}$ | | | |
| Add_Op1 | | | | | | | | | | | | | | $p_0$ | | | $p_1$ | | | $p_2$ | | | $p_0$ |
| Add_Op2 | | | | | | | | | | | | | | $y_0$ | | | $y_1$ | | | $y_2$ | | | $y_0$ |
| Result | | | | | | | | | | | | | | | | | | | | | | | $y_0$ |

…Contd.,

<table>
<tr><td colspan="2">Time Step $T_1$</td><td rowspan="2">23</td><td rowspan="2">24</td><td rowspan="2">25</td><td rowspan="2">26</td><td rowspan="2">27</td><td rowspan="2">28</td><td rowspan="2">29</td><td rowspan="2">30</td><td rowspan="2">31</td><td rowspan="2">32</td><td rowspan="2">33</td><td rowspan="2">34</td><td rowspan="2">35</td><td rowspan="2">36</td><td rowspan="2">37</td></tr>
<tr><td colspan="2">Signal</td></tr>
<tr><td colspan="2">Memory Read</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="2">Time Step $T_2$</td><td colspan="3">8</td><td colspan="3">9</td><td colspan="3">10</td><td colspan="3">11</td><td colspan="3">12</td></tr>
<tr><td>Mult_Op1</td><td rowspan="5">M A C 1</td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td></tr>
<tr><td>Mult_Op2</td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td></tr>
<tr><td>Add_Op1</td><td colspan="3">$p_1$</td><td colspan="3">$p_2$</td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td></tr>
<tr><td>Add_Op2</td><td colspan="3">$y_1$</td><td colspan="3">$y_2$</td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td></tr>
<tr><td>Result</td><td colspan="3">$y_1$</td><td colspan="3">$y_2$</td><td colspan="3">$y_0$</td><td colspan="3">$y_1$</td><td colspan="3">$y_2$</td></tr>
<tr><td>Mult_Op1</td><td rowspan="5">M A C 2</td><td colspan="2"></td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td><td></td></tr>
<tr><td>Mult_Op2</td><td colspan="2"></td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td><td colspan="3"></td><td></td></tr>
<tr><td>Add_Op1</td><td colspan="2">$p_0$</td><td colspan="3">$p_1$</td><td colspan="3">$p_2$</td><td colspan="3"></td><td colspan="3"></td><td></td></tr>
<tr><td>Add_Op2</td><td colspan="2">$y_0$</td><td colspan="3">$y_1$</td><td colspan="3">$y_2$</td><td colspan="3"></td><td colspan="3"></td><td></td></tr>
<tr><td>Result</td><td colspan="2">$y_0$</td><td colspan="3">$y_1$</td><td colspan="3">$y_2$</td><td colspan="3">$y_0$</td><td colspan="3">$y_1$</td><td>$y_2$</td></tr>
</table>

# Chapter 5

# Performance of Matrix-by-Vector Multipliers

The performance of the matrix-by-vector multiplier is experimentally measured and analyzed in this chapter. Results are obtained for various matrix sizes and implementations.

The matrix-by-vector multipliers were tested on the Wildforce board with Xilinx 4062XL chips. The Xilinx M1 tools returned a conservative figure for the maximum frequency for the applications of 6 MHz. In practice, the maximum frequency of operation for the matrix-by-vector multipliers is 15 MHz. The performance is measured in terms of MFLOPS.

The PE interrupts the host, signaling the start of the operation. The host CPU loads the PE memory with the data and resets the interrupt. The PE performs the matrix-by-vector multiplication and interrupts the host when the multiplication is complete. The host CPU reads the results from the PE memory and resets the interrupts signaling the end of the operation. The sequence of operations is given in Figure 5.1. For implementation 1 discussed in Chapter 4, the time taken to perform the matrix-by-vector multiplication does not include the initial memory setup time. For implementation 2, the time between the arrival of the two interrupts is the time taken to complete the matrix-by-vector multiplication. In implementation 2, the measured time includes the time taken to write vectors *x* and *y* into the PE memory and the time to perform the matrix-by-vector multiplication. The time is measured using the *_ftime* function available in the C *time.h* library. The performance is measured for different matrix sizes for the two implementations. Due to the limits of the Synplicity tool, the maximum size of the matrix that can be used on the Xilinx 4062XL is 6. The performances on Xilinx 4062XL are summarized in Table 5.1.

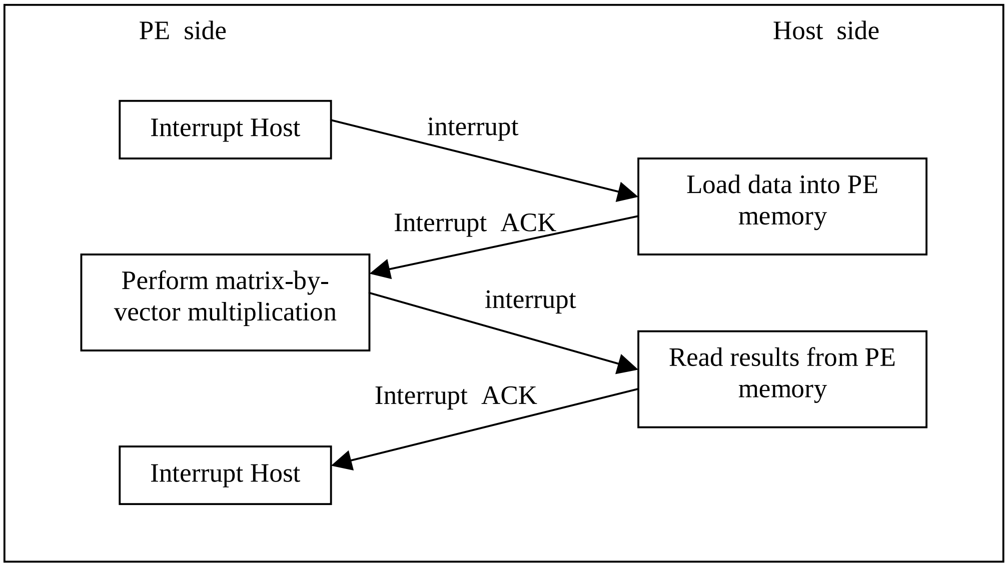


Figure 5.1 Sequence of Events for Matrix-by-Vector Multiplication

Table 5.1 Performance Summary for Xilinx 4062XL

| *Frequency: 15 MHz* | 1 PE Performance in MFLOPS | | | 5 PEs * Performance in MFLOPS | | |
|---|---|---|---|---|---|---|
| Matrix size | 3 | 4 | 6 | 3 | 4 | 6 |
| Implementation 1 No. of Columns = 6 | 9 | 12 | 16 | 45 | 60 | 80 |
| Implementation 2 | 5.4 | 6.4 | 10.8 | 27 | 32 | 54 |

** Values are extrapolated from results from 1 PE*

It is evident from Table 5.1 that the performance increases with matrix size and number of processors. The maximum performance that can be achieved by a single processor when run at *f MHz* is *2f MFLOPS* assuming a matrix of infinite size. Hence, a maximum performance of *150 MFLOPS* can be achieved when the five PEs on the Wildforce board are used and run at 15 MHz.

A significant overhead in the matrix-by-vector multiplications is the time involved in reading from and writing into the PE memory by the host CPU, which can consume upto 50% of the total execution time. The Wildforce API library supports four modes of memory access. The host CPU can access the PE memory using Programmed I/O (PIO). The Dual Port Memory Controller (DMPC) FIFOs can be used or bypassed when using PIO. The most efficient way of accessing the PE memory is by using the Direct Memory Access (DMA) channels; using either contiguous or noncontiguous physical memory. If a contiguous physical memory is allocated in the host CPU, access using DMA channels is faster. Figures 5.2 and 5.3 give the relationship between number of bytes transferred and the time taken for the transfer. The values of the time given in the figures are the average values obtained from several trials. A linear equation relating the number of bytes transferred and the time taken for the transfer can be obtained from these values using the method of least squares. From the equations, we can find the minimum overhead delay involved and the cost per byte for each of the four types of memory accesses. These results are summarized in Tables 5.2 and 5.3. It is inferred from the figures that the DMA can provide the fastest memory transfers and lowest cost per byte.

The theoretical performance values for the matrix-by-vector multipliers given in Chapter 4 are derived by calculating the ratio of number of floating-point operations to the number of clock cycles needed. The time taken for the host CPU to read from and write data into the PE memory is not considered. The actual time taken to complete the matrix-by-vector multiplications will include the memory read and write time also. The memory read and write times are significant, depending upon the mode of memory transfer, when compared to the actual computation time. For example, consider a matrix-by-vector multiplication for a matrix size of 6x6 where 3000 individual multiplications are carried out. It takes 13 milliseconds to write a vector of 32-bit numbers of size 18000 using PIO bypassing the DPMC FIFOs. For the matrix-by-vector multiplication, two such vectors, *x* and *y*, should be written and it takes about 26 milliseconds to write them. The memory write time of 26 milliseconds is more than the actual computation time of 20 milliseconds. The memory read and write times can be considerably reduced if the DMA channels are used, as evident from Figures 5.2 and 5.3. The time to write a vector of 32-bit numbers of size 9000 using the DMA channel using contiguous physical memory is about 0.7 milliseconds. The interrupt handling by the host CPU takes about a few milliseconds. Hence the actual time taken to complete the matrix-by-vector multiplication is the sum of the computation time, memory transfer time, and the interrupt handling time. Tables 5.4 and 5.5 give the read and write times for small vector sizes. Figure 5.4 gives the basic algorithm used to calculate the entries in Tables 5.4 and 5.5. The overhead time is the time involved in allocating memory space and filling the vectors with random values. The overhead time was calculated by bypassing step 3 in Figure 5.4. From Table 5.4 we find that transfer of small vectors results in low memory access rates (in MBytes/Sec). The access rates given in Tables 5.4 and 5.5 are calculated without including the overhead time. The overhead when using the contiguous physical memory is large because the time required to setup a continuous memory space in the physical main memory is large. Hence using DMA channels with contiguous physical memory is advantageous only for large vector sizes.

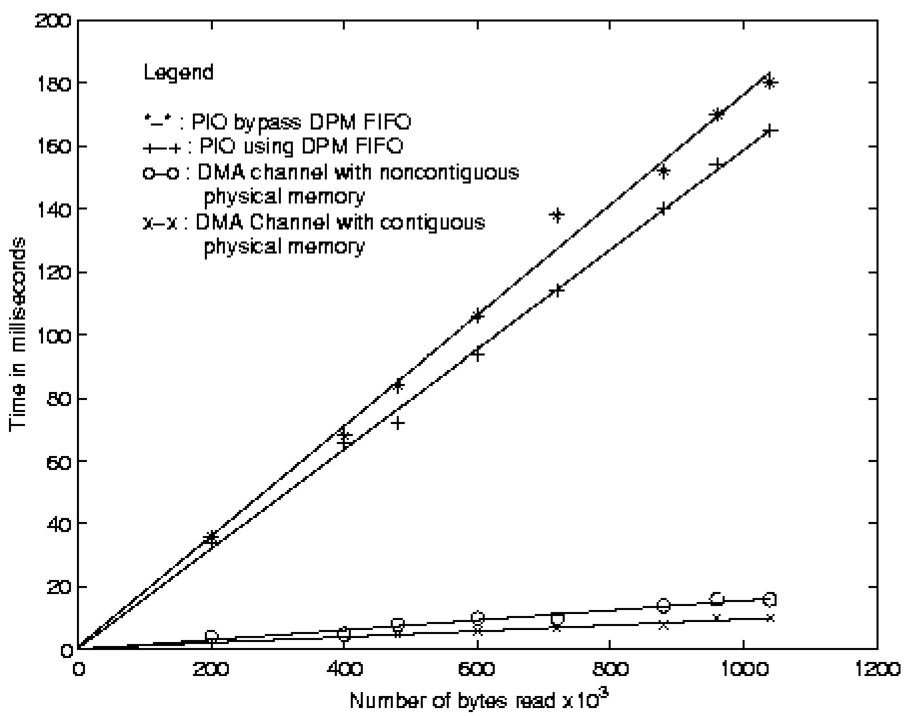


Figure 5.2 Relationship between Memory Read and Access Time

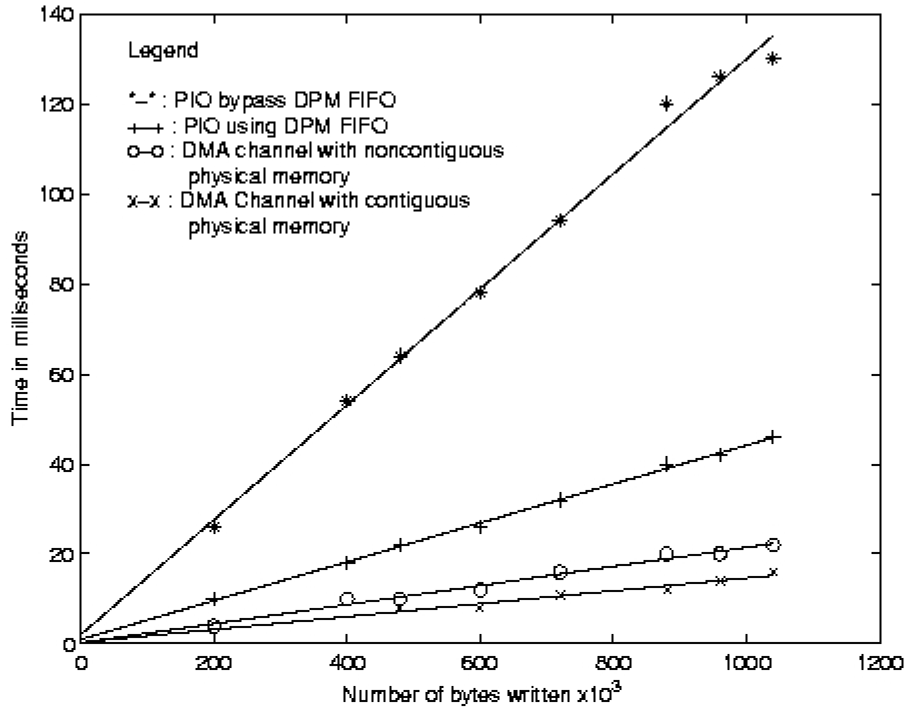


Figure 5.3 Relationship between Memory Write and Access Time

1. **Allocate memory space for *vector***
2. **Fill *vector* with constant value**
3. **Read from / Write into PE memory**
4. **De-allocate memory space for *vector***

Figure 5.4 Algorithm to Compute Read and Write Times

Table 5.2 Cost Analysis of Memory Reads

| | PIO bypass DPMC FIFO | PIO using DPMC FIFO | DMA channel with noncontiguous physical memory | DMA channel with contiguous physical memory |
|---|---|---|---|---|
| Time per Byte (in μSec) | 0.1752 | 0.1581 | 0.0156 | 0.0093 |
| Minimum overhead (in μSec) | 1.1184 | 0.4993 | 0.0844 | 0.0305 |
| MBytes per Sec | 5.7078 | 6.3251 | 64.103 | 107.53 |

Table 5.3 Cost Analysis of Memory Writes

| | PIO bypass DPMC FIFO | PIO using DPMC FIFO | DMA channel with noncontiguous physical memory | DMA channel with contiguous physical memory |
|---|---|---|---|---|
| Time per Byte (in μSec) | 0.1281 | 0.0432 | 0.0212 | 0.0144 |
| Minimum overhead (in μSec) | 1.9814 | 0.9761 | 0.2513 | 0.1243 |
| MBytes per Sec | 7.8064 | 23.148 | 47.17 | 69.44 |

Table 5.4 Read and Write Times for Small Vector Sizes using DMA Channel with Noncontiguous Physical Memory

| Number of Bytes | Time to Read (in µSec) | Time to Write (in µSec) | Overhead time for memory Access (in µSec) | Mbytes/Sec for Read | Mbytes/Sec for Write |
|---|---|---|---|---|---|
| 4000 | 234 | 230 | 37 | 20.30 | 20.73 |
| 8000 | 353 | 359 | 75 | 28.77 | 28.17 |
| 12000 | 480 | 489 | 112 | 32.61 | 31.83 |
| 16000 | 609 | 619 | 148 | 34.71 | 33.97 |
| 20000 | 733 | 741 | 202 | 37.66 | 37.11 |
| 24000 | 856 | 864 | 242 | 39.09 | 38.58 |
| 28000 | 979 | 987 | 283 | 40.23 | 39.77 |
| 32000 | 1105 | 1110 | 324 | 38.98 | 40.30 |
| 36000 | 1226 | 1231 | 363 | 41.71 | 41.47 |
| 40000 | 1350 | 1360 | 405 | 42.32 | 41.88 |

Table 5.5 Read and Write Times for Small Vector Sizes using DMA Channel with Contiguous Physical Memory

| Number of Bytes | Time to Read (in µSec) | Time to Write (in µSec) | Overhead time for memory Access (in µSec) | Mbytes/Sec for Read | Mbytes/Sec for Write |
|---|---|---|---|---|---|
| 4000 | 262 | 282 | 122 | 28.57 | 25.00 |
| 8000 | 425 | 467 | 245 | 44.44 | 36.04 |
| 12000 | 591 | 651 | 367 | 53.57 | 42.25 |
| 16000 | 751 | 831 | 490 | 61.30 | 46.92 |
| 20000 | 914 | 1013 | 612 | 66.22 | 49.87 |
| 24000 | 1078 | 1194 | 734 | 69.76 | 52.17 |
| 28000 | 1234 | 1378 | 855 | 73.87 | 53.53 |
| 32000 | 1402 | 1566 | 979 | 75.65 | 54.51 |
| 36000 | 1566 | 1750 | 1100 | 77.25 | 55.38 |
| 40000 | 1726 | 1935 | 1220 | 79.05 | 55.94 |

# Chapter 6
# Implementation of a FEM Solution

The use of the computational units developed in Chapter 4 to implement a FEM solution to a physical problem is presented in this chapter. A solution to 2-D and 3-D Poisson's equations using the FEM is implemented on the Wildforce board. The solution uses an adaptive mesh refinement strategy to discretize the physical domain to improve accuracy. An iterative method is used to solve the element equations. The core computation of this iterative method is the MAC unit described in Chapter 4.

## 6.1 Solution to the Poisson's equation

The 2-D Poisson's equation is represented as

$$\nabla^2 u(x, y) = f(x, y),$$

subject to the Dirichlet boundary condition $u(x,y) = g(x,y)$ [15]. The solution to the Poisson's equation can be approximated as

$$U(x, y) = \sum_i \alpha_i \phi_i (x, y),$$

where $\phi$ is a set of basis or trial functions, which satisfy the same boundary conditions as $u$, and $\alpha$ is a set of expansion coefficients [15]. When the trial functions are chosen such that their effect is felt or is non-zero only over a small number of neighboring elements, the solution to the above problem is reduced to solving a linear system of equations of the form

$$Ax = b,$$

where A is a sparse matrix and x and b are vectors.

The linear system of equations can be solved in a variety of ways. A widely used technique is the *conjugate gradient* (CG) method [15]. This is an iterative method to which an initial approximation of x is given. The initial negative gradient vector *r* and the search direction vector *p* are calculated using an initial approximation $x^{(1)}$ to x. The initial condition is represented as

$$r^{(1)} = p^{(1)} = b - Ax^{(1)}.$$

The $k^{th}$ iteration in the CG method is given in Figure 6.1.

1. $\mathbf{q^{(k)} = Ap^{(k)}}$
2. $\alpha^{(k)} = (r^{(k)},r^{(k)}) / (q^{(k)},p^{(k)})$
3. $r^{(k+1)} = r^{(k)} - \alpha^{(k)} q^{(k)}$
4. $x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$
5. $\beta^{(k)} = (r^{(k+1)},r^{(k+1)}) / (r^{(k+1)},r^{(k+1)})$
6. $p^{(k+1)} = r^{(k+1)} + \beta^{(k)} p^{(k)}$

Figure 6.1 $K^{th}$ Iteration in the Conjugate Gradient Method

The *(m,n)* denotes the dot product of vectors *m* and *n*. The steps in this method are level-1 and level-2 BLAS operations [13] such as Saxpy, inner product, and matrix-by-vector multiplication. The matrix-by-vector multiplication is the most computationally intensive step. Steps 1 through 6 are repeated until *x* converges to *x**. Another variation of this method is called the *preconditioned conjugate gradient (PCG)* [16]. In this method, a preconditioning matrix *M* is used as an approximation to $A^{-1}$.

Other techniques to solve the system of equations are based on factorization of the matrix A [15]. One method is to factorize the matrix A into a unit diagonal lower triangular matrix *L* and an upper triangular matrix *U*. This method is called the *Doolittle reduction* [15]. The matrix A is factorized as

$$A = LU.$$

If the matrix A is symmetric, the *Cholesky factorization* [15] can be used to factorize the matrix. In the Cholesky method, the sparse matrix is factorized into a lower triangular matrix $L$ and its transpose $L^T$.

$$A = L\,L^T$$

The solution $x$ is computed in two steps. In the first step, an intermediate solution $y$ is calculated from $Ly=b$. This intermediate solution y is used to calculate the final solution from $L^T x = y$.

### 6.2 Adaptive Mesh Refinement Method

In the FEM, the physical domain is discretized as a set of sub-domains of regular geometric shape to form a mesh. A *structured mesh* is one in which all the elements are uniform and all the nodes have the same degree. In an *unstructured mesh* the nodes have different degrees which implies that the elements can be of different shapes and sizes. Typically, an unstructured mesh approximates a complex domain more closely than a structured mesh. An unstructured mesh allows much more flexibility in specifying a geometry and refining select areas of the mesh for more accurate results [14]. In the adaptive mesh strategy, an initial unstructured mesh is further refined in areas where the solution changes more rapidly.

#### 6.2.1 Basic Algorithm

The basic algorithm for the parallel adaptive mesh refinement strategy is presented in [14]. The stiffness matrix, $K_0$ and the load vector $f_0$ for an initial unstructured mesh is assembled and the set of elements is partitioned among the processors. An initial solution $u_0$ is calculated and the error in each element is found. Based on the error estimates of all the elements in the mesh, a set of elements to be refined is determined. The elements in the set are refined to form finer meshes. The set of elements is again distributed among processors. The sparse matrix and load vectors for the refined mesh are assembled and the solution is calculated. The refinement process is repeated until the maximum error for any element is below a certain tolerance value. More information on the adaptive mesh refinement method can be found in [17].

- ***Construct*** **the initial mesh**
- ***Partition*** **the set of elements among the processors**
- ***Assemble*** **the stiffness matrix** $K_0$ **and load vector** $f_0$
- ***Solve*** $K_0\ u_{0} = f_0$
- **Estimate the local error in each element**
- **While ( the maximum error for each element is greater than the given tolerance ) do**
  - **Determine a set of elements to be refined**
  - ***Refine*** **the elements in that set**
  - ***Partition*** **the set of elements among the processors**
  - ***Solve*** $K_i\ u_{i} = f_i$ **for each element**
  - **Estimate local error in each element**
- **End while**

Figure 6.2 Basic Algorithm for Adaptive Mesh Refinement

**6.3 Implementation on the Wildforce Board**

The implementation in this thesis uses the CG iterative method to solve the linear system. The linear system solution is the dominant computation. Hence, this is the computation that is performed on the CCM. The remainder of the code remains on the host CPU.

The host CPU discretizes the domain into a set of finite elements and generates the stiffness matrix and load vector for each element. The set of finite elements is partitioned among the PEs. The matrices and vectors to be operated upon are written into the local memory of each PE. The matrix is written once and is used in all the iterations while new vectors are written for every iteration. The PE signals the start and finish of the multiplication operation using interrupts. The PE is controlled through a set of C host functions, which program the PEs, set up the memories, and control the operation of the

PE. Implementation 2 of the matrix-by-vector multiplier developed in Chapter 4 is used in this application.

### 6.3.1 Memory Organization

The matrix and the vectors are written into the local memory of the PE on the Wildforce board by the host CPU before the multiplication is initiated. Data are written into the memory using the API functions available with the Wildforce software support. The data can be written anywhere in the memory except for the first five locations (0 to 4) where the information about the location of the vectors is stored. The number of matrix-by-vector multiplications, $N$, is written in location 1. The starting addresses of the vectors *Index_x* and *Index_y*, which store the addresses of the vectors $x_e$ and $y_e$, are written in memory locations 2 and 3. The starting address for the matrix $A$ is written in location 4. Location 0 is kept blank and unused as it is altered during the multiplication process. Individual matrices $A_{e,}$ are written in the row major form and in a sequence. The address sets for individual vectors $x_e$ and $y_e$ are also written in sequence in the vectors *Index_x* and *Index_y*. For every set of matrix-by-vector multiplications, the host CPU updates the master vectors $x$ and $y$. Matrix $A$, *Index_x* and *Index_y* remain constant through out the application. The basic algorithm to solve one set of linear equations is given in Figure 6.3. Figure 6.4 gives the basic memory organization in a PE.

***1.*** **Open Wildforce board**

***2.*** **Configure PE on the board**

***3.*** **Set up the PE memory: Write *N, Index_x*, *Index_y*, *A*, Addresses of (*A*, *Index_x*, and *Index_y*)**

**4. While (error in vector *y* is less than the given tolerance) do**

- **Update vectors *x* and *y* on host**
- **Write vectors *x* and *y* to PE**
- **Perform matrix-by-vector multiplication on PE**
- **Read vector *y* from PE**

**end while**

Figure 6.3 Basic Algorithm to Solve Linear Equations on Wildforce Board

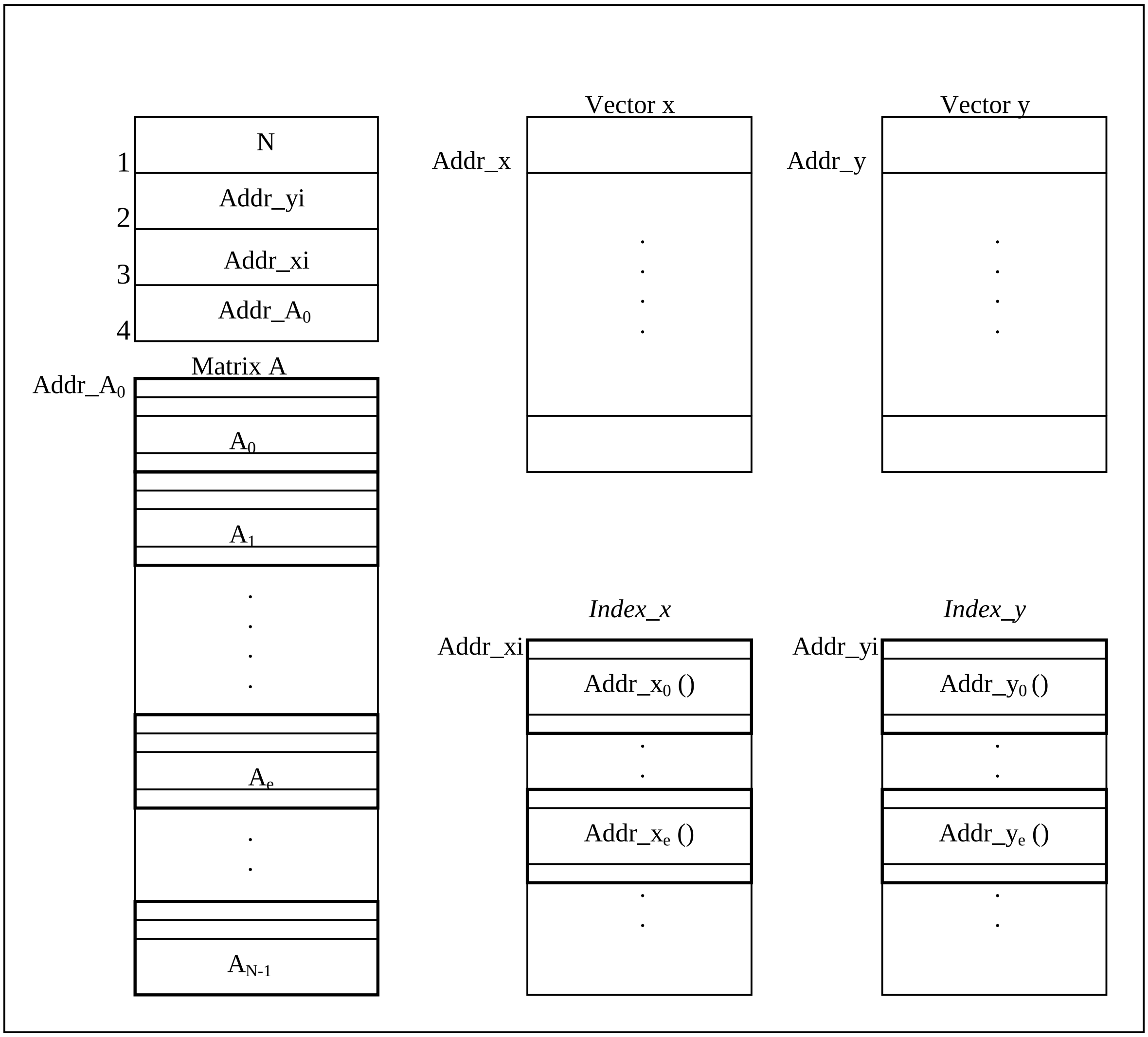


Figure 6.4 Memory Organization in a Processing Element

### 6.4 Results for 2-D and 3-D Poisson's Equations

The results obtained from solving Poisson's equation on the Wildforce board are given in Tables 6.1, 6.2, and 6.3. Linear basis functions are used to form the element matrices for the 2-D Poisson's equation with three unknowns per element and 3-D Poisson's equation with four unknowns per element. Quadratic basis functions are used for the 2-D Poisson's equation with six unknowns per element. The sequence of refined meshes of the domain is given in Figure 6.5. In the Tables 6.1, 6.2, and 6.3, the peak performance is the maximum MFLOPS achieved for any single iteration in the problem. The sustained performance is the overall performance of the Wildforce board for the problem. The two performance measures are compared with the performance achieved by using the BLAS subroutines in the Intel performance library on a Pentium II processor running at 300 MHz [13]. From the tables, we infer that the implementation on the Wildforce board gives better performance in terms of MFLOPS compared to the BLAS subroutines.

As discussed in the earlier sections, the time taken for the memory reads and writes are comparable to the actual computation time depending upon the mode of memory transfer. To reduce the overhead, the DMA channel with noncontiguous physical memory is used for memory transfers. A comparison of the relative times taken by the computations and the memory transfers is presented in Table 6.4. In Table 6.4 the overhead time includes the time to assemble the vectors, to add the result vectors from the PEs to form a single result vector, and the time the PE remains idle due to low parallelism on the host side. A comparison of the number of floating-point operations and the number of main memory accesses is given Table 6.5. From Table 6.5 we find that the ratio of floating-point operations to number of memory accesses decreases as the number of processors increases. This decrease in ratio is due to the increased number of memory writes to each PE. The ratio increases with the matrix size.

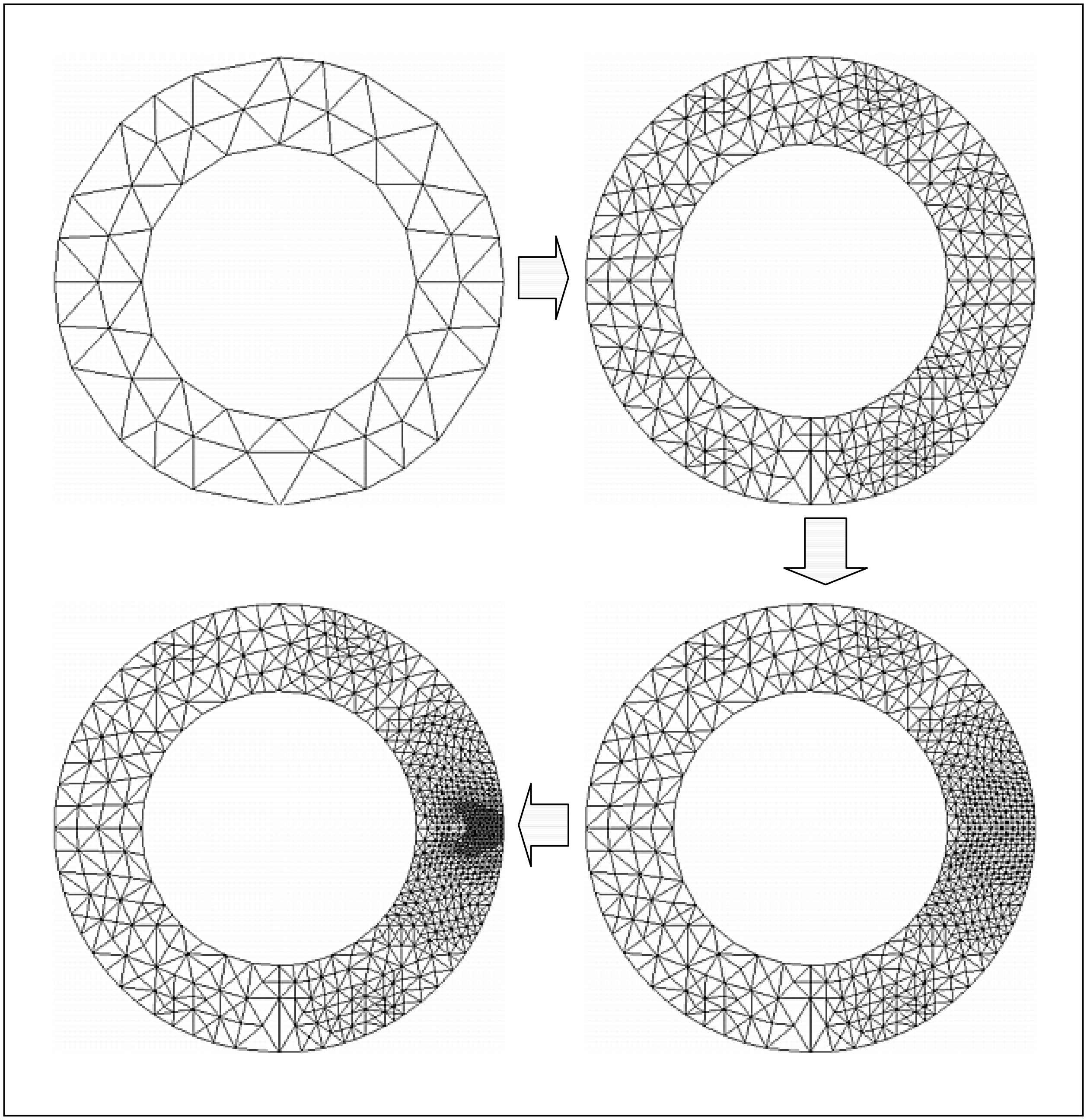

Figure 6.5 Sequence of Refined Meshes

Table 6.1 Performance Characteristics for 2-D Poisson's Equation with 3 Unknowns per Element

| Number of PEs | Peak Performance (in MFLOPS) | Peak Performance from BLAS (in MFLOPS) | Speedup | Sustained Performance | Sustained Performance from BLAS | Number of Matrices | Length of Vectors | Number of Dwords written During Memory setup | Number of Bytes Transferred During Computation |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 5.77 | - | - | 5.32 | - | 13372 | 6796 | 200580 | 20388 |
| 2 | 10.6 | - | - | 9.26 | - | 26777 | 13537 | 401655 | 45717 |
| 3 | 14.9 | - | - | 12.4 | - | 39879 | 20112 | 598185 | 76419 |
| 4 | 18.5 | - | - | 14.6 | - | 53407 | 26919 | 801105 | 103974 |
| 5 | 22.4 | 9.7 | 2.34 | 16.8 | 8.67 | 66240 | 33357 | 993600 | 123537 |

Table 6.2 Performance Characteristics for 2-D Poisson's Equation with 6 Unknowns per Element

| Number of PEs | Peak Performance (in MFLOPS) | Peak Performance from BLAS (in MFLOPS) | Speedup | Sustained Performance | Sustained Performance from BLAS | Number of Matrices | Length of Vectors | Number of Dwords written During Memory setup | Number of Bytes Transferred During Computation |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 10.8 | - | - | 9.31 | - | 4504 | 9153 | 216192 | 27459 |
| 2 | 19.0 | - | - | 14.9 | - | 8996 | 18207 | 431808 | 57012 |
| 3 | 25.7 | - | - | 18.8 | - | 13433 | 27110 | 644784 | 95421 |
| 4 | 30.9 | - | - | 21.3 | - | 17988 | 36273 | 863424 | 130827 |
| 5 | 35.7 | 24.4 | 1.46 | 22.9 | 18.7 | 22730 | 45777 | 1091040 | 161859 |

Table 6.3 Performance Characteristics for 3-D Poisson's Equation with 4 Unknowns per Element

| Number of PEs | Peak Performance (in MFLOPS) | Peak Performance from BLAS (in MFLOPS) | Speedup | Sustained Performance | Sustained Performance from BLAS | Number of Matrices | Length of Vectors | Number of Dwords written During Memory setup | Number of Bytes Transferred During Computation |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 8.37 | - | - | 7.87 | - | 8751 | 1856 | 72432 | 5568 |
| 2 | 16.0 | - | - | 14.6 | - | 20772 | 4187 | 498528 | 22953 |
| 3 | 22.8 | - | - | 20.1 | - | 28659 | 5754 | 687816 | 36570 |
| 4 | 28.7 | - | - | 24.5 | - | 37524 | 7476 | 900576 | 52392 |
| 5 | 33.8 | 16.8 | 2.01 | 28.5 | 15.5 | 50475 | 9871 | 1211400 | 88107 |

Table 6.4 Comparison of Relative times for the Floating-Point Operations and Memory Transfer

| Matrix size | Number of PEs | Percentage Time for Floating-point Operations | Percentage Time for DMA Transfer | Percentage Time for Overhead |
|---|---|---|---|---|
| 3 | 1 | 79.09 | 1.04 | 19.87 |
| 3 | 2 | 73.39 | 2.15 | 24.46 |
| 3 | 3 | 56.64 | 2.80 | 40.56 |
| 3 | 4 | 49.90 | 3.35 | 46.75 |
| 3 | 5 | 46.03 | 3.69 | 50.28 |
| 6 | 1 | 65.57 | 1.67 | 32.66 |
| 6 | 2 | 52.39 | 2.78 | 44.83 |
| 6 | 3 | 38.9 | 3.94 | 57.16 |
| 6 | 4 | 37.38 | 4.55 | 58.07 |
| 6 | 5 | 32.21 | 4.81 | 62.98 |
| 4 | 1 | 58.98 | 0.34 | 40.68 |
| 4 | 2 | 54.95 | 1.07 | 43.98 |
| 4 | 3 | 50.44 | 1.70 | 47.86 |
| 4 | 4 | 46.94 | 2.26 | 50.80 |
| 4 | 5 | 42.32 | 3.29 | 54.39 |

Table 6.5 Comparison of Number of Floating-Point Operations and Main Memory Accesses

| Matrix size | Number of PEs | Number of Floating-Point Operations (Fp) | Number of Bytes Transferred (M) | Ratio Fp/M |
|---|---|---|---|---|
| 3 | 1 | 240696 | 20388 | 11.81 |
| 3 | 2 | 481986 | 45717 | 10.54 |
| 3 | 3 | 717822 | 76419 | 9.39 |
| 3 | 4 | 961326 | 103974 | 9.24 |
| 3 | 5 | 1192320 | 123537 | 9.65 |
| 6 | 1 | 324288 | 27459 | 11.81 |
| 6 | 2 | 647712 | 57012 | 11.36 |
| 6 | 3 | 967176 | 95421 | 10.13 |
| 6 | 4 | 1295136 | 130827 | 9.90 |
| 6 | 5 | 1636560 | 161859 | 10.11 |
| 4 | 1 | 280032 | 5568 | 50.29 |
| 4 | 2 | 664704 | 22953 | 28.96 |
| 4 | 3 | 917088 | 36570 | 25.08 |
| 4 | 4 | 1200768 | 52392 | 22.92 |
| 4 | 5 | 1615200 | 88107 | 18.33 |

# Chapter 7
# Conclusions

## 7.1 Conclusions

The two matrix-by-vector multiplier units developed in this thesis were implemented on the Wildforce board. The maximum frequency of the multiplier units is 15 MHz. The minimum performance of implementation 1 on one PE for a 6x6 matrix is 17.14 MFLOPS, and 13.84 MFLOPS for implementation 2. The performance increases as the number of matrix-by-vector increases as the number of matrix-by-vector multiplications increases. A performance of 35.7 MFLOPS was obtained when implementation 2 was loaded on all the 5 PEs on the Wildforce board and used to solve 2-D Poisson's equation with 6 unknowns per element.

The matrix-by-vector multiplication units developed in this thesis can be used for any mathematical solution which calls for such an operation. The matrix-by-vector multiplication units developed in this thesis and the algorithm for the FEM solution discussed in the earlier chapters are just basic building blocks and can be used for solving a variety of physical problems such as stress analysis and heat transfer. The Poisson's equation solved in this thesis is only a representative of such physical problems.

A CCM solution to a physical problem using the FEM has been successfully implemented in this thesis. The use of CCMs to solve computationally expensive problems with potential parallelism in their algorithms is evident in this thesis. For domains with large number of nodes, a set of independent elements can be assigned to each processor and the speed-up achieved is approximately equal to the number of processors used. The Wildforce board has been identified as a good reconfigurable platform because of its high density FPGAs and efficient inter-PE interface.

## 7.2 Future Work

As discussed in Chapter 4, the performance can be doubled if two multiplier-accumulators (MACs) can be used. This implementation requires a second clock that runs at one-third the frequency of the process clock. The process clock should be used to clock all the memory elements in the application. The delays associated with the combinational logic and state machine introduces undesired changes in logic level for small duration in the second clock. The intermediate logic levels falsely trigger the adders and multipliers, which results in inaccurate answers. The use of a flip-flop with a clock enable pin is a probable solution to the problem. The adders and multipliers used in this thesis can be further optimized to give better performance. For example, the 24-bit integer multiplication can be performed in more than one step to increase speed.

# Bibliography

[1] N. Shirazi, A. Walters, and P. M. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom computing Machines," *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, P. Athanas and K.L. Pocek (eds.), pp. 155-62, April 1995.

[2] W. B. Ligon III, et al, "A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs," *IEEE Symposium on Field Programmable CCMs*, April, Napa, CA., 1998.

[3] K. J. Paar, "A Custom Computing Machine Solution for Simulation of Discretized Domain Physical Systems," Masters Thesis, Virginia Polytechnic Institute and State University, 1996.

[4] A. L. Walters, "A Scaleable FIR Filter Implementation Using 32-bit Floating-Point Complex Arithmetic on a FPGA Based Custom Computing Platform," Masters Thesis, Virginia Polytechnic Institute and State University, 1996.

[5] J. N. Reddy, "An Introduction to the Finite Element Method," Second Ed., McGraw-Hill, 1993.

[6] D.A. Buell and K. L. Pocek, "Custom Computing Machines: An Introduction," *Journal of Supercomputing*, vol. 9, no. 3, pp. 219-229, 1995.

[7] Synplicity, Inc., "Synplify User's Guide," Synplicity Inc., 1994-1996.

[8] J.R. Armstrong and F.G. Gray, "Structured Logic Design with VHDL," Upper Saddle River, NJ: Prentice Hall, 1993.

[9] Annapolis Micro Systems, Inc., Wildforce Reference Manual, 1997.

[10] I.S. Board, “IEEE standard for binary floating-point arithmetic,” Tech. Rep. ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, New York, 1985.

[11] J. N. Reddy and D. K. Gartling, “The Finite Element Method in Heat Transfer and Fluid Dynamics,” Boca Raton, FL: CRC Press, 1994.

[12] Advanced Micro Devices, Inc., ADSP21060/21062, “Super Harvard Architecture Computer,” product literature, 1995.

[13] Intel Corporation, Intel Math Kernal Library Reference Manual, 1994-1998.

[14] M. T. Jones and P. E. Plassman, “Computational Results for Parallel Unstructured Mesh Computations,” *Computing Systems in Engineering*, Vol.5, No.4-6, pp. 297-309, 1994.

[15] T.L. Freeman and C.Phillips, “Parallel Numerical Algorithms,” Prentice Hall, 1992.

[16] J.M. Ortega and R.G. Voigt, “Solution of partial Differential Equations on Vector and Parallel Computers,” *SIAM*, Philadelphia, PA 1985.

[17] M.T. Jones and .P.E. Plassman, “Parallel Algorithms for Adaptive Mesh Refinement,” *SIAM Journal of Scientific Computing*, Vol. 18, pp. 686 – 708, 1997.

# APPENDIX A

## VHDL Code for PE Logic Core for Implementation 1

```
-----------------------------------------------------------------------
--
--  Entity        : PE1_Logic_Core
--
--  Architecture  : rect3
--
--  Filename      : pe1lca.vhd
-- To change size of matrices  change the constant Array_size and
-- the limits of array of the signal Sum_vector
-- Num of Cols is user input. this code is synthesizable
-- introduced more registers which  hold the value (cols(rows+1)) and
counts down
```

```
---------------------------- Library Declarations ---------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
-------------------------- Architecture Declaration ------------------

architecture rect3 of PE1_Logic_Core is

      function INTVAL (VAL: std_logic_vector)
      return INTEGER is
      variable SUM: INTEGER:= 0;
       begin
      for  N in VAL'LOW to VAL'HIGH loop
      assert not(VAL(N) = 'X' or VAL(N) = 'Z')
      report "INTVAL inputs not 0 or 1"
      severity WARNING;
      if  VAL(N) = '1' then
      SUM := SUM + (2**N);
      end if;
      end loop;
      return SUM;
      end INTVAL;

--  State machine states type (enumerated encoding)
      --
      type States is (
            State_Init,
            State_Ready,
            State_WaitAckFinish,
            State_RDSetup,
            State_RdFrstLoc,
            State_RdNxtLoc,
            State_RD_IntVals,
            State_RD_IntVals2,
            State_RD_Cvector,
            State_RD_Compute,
```

```
        State_RD_WR,
        State_WR_Ans,
        State_Decr_nummat,
        State_Chkfinish,
        State_Interrupt,
        State_Idle
);

type array_stdlogicVector is
     array (Natural range<>) of std_logic_vector(31 downto 0);
type Number is range 0 to 31;

constant Num_Rows : integer range 0 to 31 := 6;
--
--  State machine signals
--
signal Present_State    : States;
signal Next_State       : States;

signal S_MemStrobe_n    : std_ulogic;
signal S_MemWriteSel_n  : std_ulogic;
signal S_MemAddr        : std_logic_vector(21 downto 0);
signal S_MemData  : std_logic_vector(31 downto 0);
signal Data_Read  : std_logic_vector(31 downto 0);

signal Addr_Reg   : std_logic_vector(21 downto 0);
signal Addr_RegReset : std_ulogic;
signal Addr_RegEnable : std_ulogic;
signal DAta_Reg   : std_logic_vector(31 downto 0);
signal Data_RegReset : std_ulogic;
signal Data_RegEnable : std_ulogic;

signal s_result : std_logic_vector(31 downto 0);
signal Result_Register : std_logic_vector(31 downto 0);

signal S_s1_R1_mult : std_ulogic;
signal S_S2_R1_mult : std_ulogic;
signal S_e1_R1_mult : std_logic_vector(7 downto 0);
signal S_e2_R1_mult : std_logic_vector(7 downto 0);
signal S_f1_R1_mult : std_logic_vector(23 downto 0);
signal S_f2_R1_mult : std_logic_vector(23 downto 0);
signal s1_R1_mult : std_ulogic;
signal S2_R1_mult : std_ulogic;
signal e1_R1_mult : std_logic_vector(7 downto 0);
signal e2_R1_mult : std_logic_vector(7 downto 0);
signal f1_R1_mult : std_logic_vector(23 downto 0);
signal f2_R1_mult : std_logic_vector(23 downto 0);
signal S_Esum_R1_mult : std_logic_vector(8 downto 0);
signal Esum_R1_mult : std_logic_vector(8 downto 0);
signal S_s_R2_mult : std_ulogic;
signal S_Esum_R2_mult : std_logic_vector(8 downto 0);
signal S_f_R2_mult : std_logic_vector(24 downto 0);
signal S_Underflow_or_Zero_mult : std_ulogic;
signal S_Overflow_mult : std_ulogic;
signal Underflow_or_Zero_mult : std_ulogic;
signal Overflow_mult : std_ulogic;
signal Esum_R2_mult : std_logic_vector(8 downto 0);
```

```
signal s_R2_mult : std_ulogic;
signal f_R2_mult : std_logic_vector(24 downto 0);
signal S_s_R3_mult : std_ulogic;
signal S_e_R3_mult : std_logic_vector(8 downto 0);
signal S_f_R3_mult : std_logic_vector(22 downto 0);

signal s1_R1_add : std_ulogic;
signal s2_R1_add : std_ulogic;
signal E1_R1_add : std_logic_vector(7 downto 0);
signal e2_R1_add : std_logic_vector(7 downto 0);
signal f1_R1_add : std_logic_vector(31 downto 0);
signal f2_R1_add : std_logic_vector(31 downto 0);
signal e_diff_R1_add : std_logic_vector(7 downto 0);

signal s_e_diff_add : std_logic_vector(8 downto 0);
signal s1_add : std_ulogic;
signal s2_add : std_ulogic;
signal E1_add : std_logic_vector(7 downto 0);
signal e2_add : std_logic_vector(7 downto 0);
signal f1_add : std_logic_vector(31 downto 0);
signal f2_add : std_logic_vector(31 downto 0);
signal S_temp_f_R2_add : std_logic_vector(31 downto 0);
signal s1_R2_add : std_ulogic;
signal s2_R2_add : std_ulogic;
signal e1_R2_add : std_logic_vector(7 downto 0);
signal e2_R2_add : std_logic_vector(7 downto 0);
signal temp_f_R2_add : std_logic_vector(24 downto 0);
signal s_s_R3_add: std_ulogic;
signal s_e_R3_add : std_logic_vector(7 downto 0);
signal s_f_R3_add : std_logic_vector(22 downto 0);

signal Addr_x : std_logic_vector(21 downto 0);
signal Addr_Y : std_logic_vector(21 downto 0);
signal Addr_C : std_logic_vector(21 downto 0);
signal C_Addr_Reg : std_logic_vector(21 downto 0);
signal Num_cols : std_logic_vector(15 downto 0);
signal Num_Mat : integer range 0 to 65535;
signal Xi : std_logic_vector(31 downto 0);
signal Yi : std_logic_vector(31 downto 0);
signal Product : std_logic_vector(31 downto 0);

signal SumVector: array_stdlogicvector(0 to 5); -- 0 to Num_Rows-
1

signal RdIntVal_En : std_ulogic;
signal IncrXY_En : std_ulogic;
signal IncrC_En : std_ulogic;
signal RDXi_En : std_ulogic;
signal RDYi_En : std_ulogic;
signal RDCi_En : std_ulogic;
signal Offset : std_logic_vector(15 downto 0);
signal IncrOffset : std_ulogic;

signal Iter_Reset : std_ulogic;
signal Iter_En : std_ulogic;
signal Iter : integer range 0 to 255;
signal Iter2 : integer range 0 to 255;
```

```
	signal SumIndex_Reset : std_ulogic;
	signal SumIndex_En : std_ulogic;
	signal SumIndex : integer range 0 to 31;
	signal Delay1 : integer range 0 to 31;
	signal SumIndex2 : integer range 0 to 31;

	signal Limit : integer range 0 to 255;
	signal Limit_0: integer range 0 to 255;
	signal Limit_1 : integer range 0 to 255;
	signal Limit_2 : integer range 0 to 255;
	signal Limitplus10 : integer range 0 to 255;
	signal Decr : std_ulogic;
	signal Decr3 : std_ulogic;
	signal Decr8 : std_ulogic;
	signal Decr10 : std_ulogic;
	signal Decr_Num_Mat : std_ulogic;


	constant e_zero: std_logic_vector(7 downto 0) := "00000000";
	constant f_zero: std_logic_vector(22 downto 0) :=
"00000000000000000000000";
	constant one_two_seven:    std_logic_vector(8 downto 0) :=
"001111111";
	constant one_two_six:     std_logic_vector(8 downto 0) :=
"001111110";
	constant three_eight_two : std_logic_vector(8 downto 0) :=
"101111110";
	constant zero_7:            std_logic_vector(6 downto 0) :=
"0000000";
	constant zero_8:            std_logic_vector(7 downto 0) :=
"00000000";
	constant zero_25:           std_logic_vector(24 downto 0) :=
"0000000000000000000000000";
	constant zero_31:           std_logic_vector(30 downto 0) :=
"0000000000000000000000000000000";
	constant Address_Zero    : std_logic_vector(21 downto 0) :=
"0000000000000000000000";
	constant Address_One     : std_logic_vector(21 downto 0) :=
"0000000000000000000001";
	constant Address_two     : std_logic_vector(21 downto 0) :=
"0000000000000000000010";
	constant Address_three   : std_logic_vector(21 downto 0) :=
"0000000000000000000011";
	constant Address_four    : std_logic_vector(21 downto 0) :=
"0000000000000000000100";
	constant Address_five    : std_logic_vector(21 downto 0) :=
"0000000000000000000101";


begin
	----------------------------
	--    Signal assignment statements
	----------------------------

	--
	-- Address Register
	--
```

```
process (PE_PClk, PE_Reset)
begin
      if (PE_Reset = '1') then
                  Addr_Reg <= (others => '0');
      elsif ( ( PE_Pclk = '1' ) and ( PE_Pclk'event ) ) then
                  if (Addr_RegReset /= '1') then
                        if (Addr_RegEnable = '1') then

                              Addr_Reg <= S_MemAddr;
                        end if;

                  else

                        Addr_Reg <= (others=>'0');
                  end if;

      end if;

end process;
--
-- Data reg
--
process (PE_PClk, PE_Reset)
      begin
            if (PE_Reset = '1') then
                  Data_Reg <= (others => '0');
            elsif ( ( PE_Pclk = '1' ) and ( PE_Pclk'event ) )
then
                  if (Data_RegReset /= '1') then
                        if (Data_RegEnable = '1') then

                              Data_Reg <= S_MemData;
                        end if;

                  else

                        Data_Reg <= (others=>'0');
                  end if;

      end if;

end process;


P_Iteration_counter: process(PE_Reset,PE_PClk)
begin
      if (PE_Reset = '1') then
            Iter <= 0;
            Iter2 <= 0;
      elsif ( (PE_Pclk = '1' ) and ( PE_Pclk'event ) ) then

            if (Iter_reset /= '1') then
                  if (Iter_En = '1') then
                     Iter <= Iter + 1;

                     If (Iter2 < Num_Rows) then
```

```
                        Iter2 <= Iter2 +1;
                     else
                        Iter2 <= 0;
                     end if;

                  end if;
            else
                  Iter <= 0;
                  Iter2 <= 0;
            end if;
      end if;
end process;

P_SumIndex_counter: process(PE_Reset,PE_PClk)
begin
      if (PE_Reset = '1') then
            SumIndex <= 0;
            Delay1 <= 0;
            SumIndex2 <= 0;

      elsif ( (PE_Pclk = '1' ) and ( PE_Pclk'event ) ) then

            if (SumIndex_Reset /= '1') then
                  if (SumIndex_En = '1') then
                     SumIndex <= SumIndex + 1;
                  end if;
            else
                  SumIndex <= 0;
            end if;

            -- delay by three cycles
            Delay1 <= SumIndex;
            SumIndex2 <= Delay1;

      end if;
end process;

-- Synchronous process that clocks the state machines, and
handles
      --   reset conditions.
      P_Sync: process (PE_Reset, PE_Pclk)

      begin

            if ( PE_Reset = '1' ) then
                  Present_State <= State_Init;

                  s1_R1_mult <= '0';
                  s2_R1_mult <= '0';
                  e1_R1_mult <= (others => '0');
                  e2_R1_mult <= (others => '0');
                  f1_R1_mult <= (others => '0');
                  f2_R1_mult <= (others => '0');
                  ESum_R1_mult <= (others => '0');

                  -- latch stage2
                  s_R2_mult <= '0';
```

```
                        Esum_R2_mult <= (others => ’0’);
                        f_R2_mult <= (others => ’0’);
                        Underflow_or_zero_mult <= ’0’;
                        Overflow_mult <= ’0’;

                        -- latch stage3
                        Product <= (others => ’0’);

                        -- adder latches
                        s1_R1_add <= ’0’;
                        s2_R1_add <= ’0’;
                        e1_R1_add <= (others => ’0’);
                        e2_R1_add <= (others => ’0’);
                         f1_R1_add <= (others => ’0’);
                        f2_R1_add <= (others => ’0’);
                        e_diff_R1_add <= (others => ’0’);

                        -- latch outputs of second stage
                        s1_R2_add <= ’0’;
                        s2_R2_add <= ’0’;
                        e1_R2_add <= (others => ’0’);
                        e2_R2_add <=(others => ’0’);
                        temp_f_R2_add <= (others => ’0’);

                        -- output from third stage
                        SumVector <= (others => (others => ’0’));


                  elsif ( ( PE_Pclk = ’1’ ) and ( PE_Pclk’event ) )
then
                        Present_State <= Next_State;
                        -- latch stage1
                        s1_R1_mult <= S_s1_R1_mult;
                        s2_R1_mult <= S_s2_R1_mult;
                        e1_R1_mult <= S_e1_R1_mult;
                        e2_R1_mult <= S_E2_R1_mult;
                        f1_R1_mult <= S_f1_R1_mult;
                        f2_R1_mult<= S_f2_R1_mult;
                        ESum_R1_mult <= S_Esum_R1_mult;

                        -- latch stage2
                        s_R2_mult <= S_s_R2_mult;
                        Esum_R2_mult <= S_Esum_R2_mult;
                        f_R2_mult<= S_f_R2_mult;
                        Underflow_or_zero_mult <=
S_Underflow_or_Zero_mult;
                        Overflow_mult <= S_Overflow_mult;

                        -- latch stage3
                        Product <= S_s_R3_mult & S_e_R3_mult(7 downto
0) & S_f_R3_mult;



                        -- adder latches

                        s1_R1_add <= s1_add;
```

```
                        s2_R1_add <= s2_add;
                        e1_R1_add <= e1_add;
                        e2_R1_add <= e2_add;
                        f1_R1_add <= f1_add;
                        f2_R1_add <= f2_add;
                        e_diff_R1_add <= s_e_diff_add(7 downto 0);

                        -- latch outputs of second stage
                         s1_R2_add <= s1_R1_add;
                         s2_R2_add <= s2_R1_add;
                        e1_R2_add <= E1_R1_add;
                        e2_R2_add <= e2_R1_add;
                        temp_f_R2_add <= S_temp_f_R2_add(24 downto 0);

                        -- output from third stage

                        if (RDCi_En = '1') then
                            SumVector(Iter-3) <= PE_MemData_Inreg;
                        elsif ((Present_State = State_RD_Compute) and
                               (Iter > 9) and (Limitplus10 > 0)
                               and (SumIndex2 < Num_Rows)) then
                             SumVector(Sumindex2 ) <= s1_R2_add &
s_e_R3_add & s_f_R3_add;
                        end if;

                  end if;

            end process P_Sync;

      P_sync2_Addr : process(PE_Reset, PE_PClk)

      begin
          if (PE_Reset = '1') then

            Limit_0<= 0;
            Limit_1 <= 0;
            Limit_2 <= 0;
            Limitplus10 <= 0;
            Num_mat <= 0;
            Num_cols <= (others => '0');
            C_Addr_Reg <= (others => '0');
            Addr_C <= (others => '0');
            Addr_Y <= (others => '0');
            Addr_X <= (others => '0');
            Offset <= (others => '0');

          elsif ( ( PE_Pclk = '1' ) and ( PE_Pclk'event ) ) then

            if (RDIntVal_En = '1') then
                Num_Mat <= INTVAL(Num_Cols);
                Num_cols <= Addr_C(15 downto 0);
                C_Addr_Reg <= Addr_X; -- copy of Addr_C
                Addr_C <= Addr_X;
                Addr_X <= Addr_Y;
                Addr_Y <= PE_MemData_Inreg(21 downto 0);

            else
```

```
                if (IncrXY_En = ’1’) then

                    Addr_X <= Addr_X + 1;
                  Addr_Y <= Addr_Y + 1;
                elsif (Decr_Num_Mat = ’1’) then
                    Addr_Y <= Addr_Y;
                    Addr_X <= Addr_X + (Num_Rows -
1)*(INTVAL(Num_Cols));
                end if;

               if (IncrC_En = ’1’) then
                    Addr_C <= Addr_C + 1;
               elsif (Present_State = State_RD_WR) then
                    Addr_C <= Addr_C - Num_Rows;
               end if;
               if (Decr_Num_Mat = ’1’) then
                   Num_Mat <= Num_Mat - 1;
               end if;

            end if;

            if (RDIntVal_En = ’1’) then
               Limit <= INTVAL(Addr_C(4 downto 0))*(Num_Rows+1);
               Limit_0<= INTVAL(Addr_C(4 downto 0))*(Num_Rows+1);
               Limit_1 <= (INTVAL(Addr_C(4 downto 0))*(Num_Rows+1)) ;
               Limit_2 <= (INTVAL(Addr_C(4 downto 0))*(Num_Rows+1)) ;
               Limitplus10<= (INTVAL(Addr_C(4 downto 0))*(Num_Rows+1))
+ 10;
            elsif (Decr_Num_Mat = ’1’) then
               Limit_0 <= Limit;
               Limit_1 <= LImit;
               Limit_2 <= Limit;
               Limitplus10 <= Limit + 10;
            else
               if (Decr = ’1’) then
                  Limit_0<= Limit_0- 1;
               end if;
               if (Decr3 = ’1’) then
                  Limit_1 <= Limit_1 -1;
               end if;
               if (Decr8 = ’1’) then
                  Limit_2  <= Limit_2 - 1;
               end if;
               if (Decr10 = ’1’) then
                  Limitplus10 <= Limitplus10 -1;
               end if;

            end if;


            if (IncrXY_En = ’1’) then
               offset <= (others => ’0’);
            elsif (IncrOffset = ’1’) then
               Offset <= Offset + Num_Cols;
            elsif (Decr_Num_Mat = ’0’) then
               Offset <= (others => ’0’);
            end if;
```

```
    end if;

end process;


P_Sync_Data : process (PE_Reset, PE_PClk)
begin
      if (PE_Reset = '1') then

            Xi <= (others => '0');
            Yi <= (others => '0');
      elsif ( ( PE_Pclk = '1' ) and ( PE_Pclk'event ) ) then

            if (RDXi_En = '1') then
                  Xi <= PE_MemData_Inreg;
            end if;
            if (RDYi_En = '1') then
                  Yi <= PE_MemData_Inreg;
            end if;
      end if;
 end process;


-- Asynchronous signal assignment statements

PE_MemStrobe_n <= S_MemStrobe_n;
PE_MemWriteSel_n <= S_MemWriteSel_n;
PE_MemAddr_OutReg <= Addr_Reg(21 downto 0);
PE_MemData_OutReg <= Data_Reg(31 downto 0);


--------------------------------
-- Combinational state machine
--------------------------------
P_ASync: process(Present_State, PE_MemBusGrant_n,
                       PE_InterruptAck_n, iter)

variable limit : integer range 0 to 31;

      begin


            case Present_State is
                  -- Initialization state.
            when State_Init =>
                  PE_InterruptReq_n <= '1';
                  PE_MemBusReq_n <= '1';
                  S_MemWriteSel_n <= '0';
                  S_MemStrobe_n <= '1';
                  Addr_RegReset <= '1';
                  Addr_RegEnable <= '0';
                  Data_RegReset <= '1';
                  Data_RegEnable <= '0';
                  RDXi_En <= '0';
                  RDYi_En <= '0';
```

```
        RDCi_En <= '0';
        RDIntVal_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        IncrC_En <= '0';
        iter_Reset <= '1';
        Iter_En <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '1';
        Decr <= '0';
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_Num_Mat <= '0';
        Next_State <= State_Ready;

-- Interrupt the host; PE ready
when State_Ready =>
        PE_InterruptReq_n <= '0';
        PE_MemBusReq_n <= '1';
        S_MemWriteSel_n <= '0';
        S_MemStrobe_n <= '1';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDIntVal_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        IncrC_En <= '0';
        iter_Reset <= '0';
        Iter_En <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        Decr <= '0';
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_Num_Mat <= '0';
        if (PE_InterruptAck_n = '0')then
              Next_State <= State_WaitAckFinish;
        else Next_State <= State_Ready;
        end if;

-- Wait for ACK to go high again
when State_WaitAckFinish =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '1';
        S_MemWriteSel_n <= '0';
        S_MemStrobe_n <= '1';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '0';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
```

```
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDIntVal_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        IncrC_En <= '0';
        iter_Reset <= '0';
        Iter_En <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        Decr <= '0';
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_Num_Mat <= '0';

        if (PE_InterruptAck_n = '0')then
              Next_State <= State_WaitAckFinish;
        else Next_State <= State_RDSetup;
        end if;

  when State_RDSetup =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '0';
        S_MemStrobe_n <= '0';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '1';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDIntVal_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        IncrC_En <= '0';
        iter_Reset <= '0';
        Iter_En <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        S_MemAddr <= Address_one;
        Decr <= '0';
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_Num_Mat <= '0';
              if (PE_MemBusGrant_n = '0') then
                    Next_State <= State_RDFrstLoc;
              else
                    Next_State <= State_RDSetup;
              end if;

  -- We are reading from the first address location.
  when State_RDFrstLoc =>
        PE_InterruptReq_n <= '1';
```

```
            PE_MemBusReq_n <= '0';
            S_MemStrobe_n <= '0';
            S_MemWriteSel_n <= '1';
            Addr_RegReset <= '0';
            Addr_RegEnable <= '1';
            Data_RegReset <= '0';
            Data_RegEnable <= '0';
            RDXi_En <= '0';
            RDYi_En <= '0';
            RDCi_En <= '0';
            RDIntVal_En <= '0';
            IncrXY_En <= '0';
            IncrOffset <= '0';
            IncrC_En <= '0';
            S_MemAddr <= Address_two;
            SumIndex_En <= '0';
            SumIndex_Reset <= '0';
            Decr <= '0';
            Decr3 <= '0';
            Decr8 <= '0';
            Decr10 <= '0';
            Decr_Num_Mat <= '0';
            Next_State <= State_RDNxtLoc;
            iter_Reset <= '0';
            Iter_En <= '0';

      -- We are reading from the second address location.
      when State_RDNxtLoc =>
            PE_InterruptReq_n <= '1';
            PE_MemBusReq_n <= '0';
            S_MemStrobe_n <= '0';
            S_MemWriteSel_n <= '1';
            Addr_RegReset <= '0';
            Addr_RegEnable <= '1';
            Data_RegReset <= '0';
            Data_RegEnable <= '0';
            RDXi_En <= '0';
            RDYi_En <= '0';
            RDCi_En <= '0';
            RDIntVal_En <= '0';
            IncrXY_En <= '0';
            IncrOffset <= '0';
            IncrC_En <= '0';
            iter_Reset <= '0';
            Iter_En <= '0';
            SumIndex_En <= '0';
            SumIndex_Reset <= '0';
            S_MemAddr <= Address_three;
            Decr <= '0';
            Decr3 <= '0';
            Decr8 <= '0';
            Decr10 <= '0';
            Decr_Num_Mat <= '0';
            Next_State <= State_RD_IntVals;

      -- The data from two reads ago should now be visible
on
```

```
--   the registered data input from the memory.
when State_RD_IntVals =>
      PE_InterruptReq_n <= '1';
      PE_MemBusReq_n <= '0';
      S_MemStrobe_n <= '0';
      S_MemWriteSel_n <= '1';
      Addr_RegReset <= '0';
      Addr_RegEnable <= '1';
      Data_RegReset <= '0';
      Data_RegEnable <= '0';
      RDXi_En <= '0';
      RDYi_En <= '0';
      RDCi_En <= '0';
      RDIntVal_En <= '1';
      IncrXY_En <= '0';
      IncrOffset <= '0';
      IncrC_En <= '0';
      SumIndex_En <= '0';
      SumIndex_Reset <= '0';
      S_MemAddr <= Address_four;
      Decr <= '0';
      Decr3 <= '0';
      Decr8 <= '0';
      Decr10 <= '0';
      Iter_En <= '0';
      Iter_Reset <= '0';
      Decr_Num_Mat <= '0';
      Next_State <= State_RD_Intvals2;

when State_RD_Intvals2 =>
      PE_InterruptReq_n <= '1';
      PE_MemBusReq_n <= '0';
      S_MemStrobe_n <= '0';
      S_MemWriteSel_n <= '1';
      Addr_RegReset <= '0';
      Addr_RegEnable <= '1';
      Data_RegReset <= '0';
      Data_RegEnable <= '0';
      RDXi_En <= '0';
      RDYi_En <= '0';
      RDCi_En <= '0';
      RDIntVal_En <= '1';
      IncrXY_En <= '0';
      IncrOffset <= '0';
      IncrC_En <= '0';
      SumIndex_En <= '0';
      SumIndex_Reset <= '0';
      S_MemAddr <= Address_five;
      Decr <= '0';
      Decr3 <= '0';
      Decr8 <= '0';
      Decr10 <= '0';
      Decr_Num_Mat <= '0';
      if (Iter = 3) then
            Next_State <= State_RD_Cvector;
            Iter_En <= '0';
            Iter_Reset <= '1';
```

```
        else    Next_State <= State_RD_IntVals2;
                Iter_En <= '1';
                Iter_Reset <= '0';
        end if;


  when State_RD_Cvector =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '0';
        S_MemStrobe_n <= '0';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '0';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDIntVal_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        Decr <= '0';
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_Num_Mat <= '0';

        if ((Iter > 2) and (Iter < Num_Rows+3)) then
           RDCi_En <= '1';
        else
           RDCi_En <= '0';
        end if;

        if (Iter < Num_Rows) then
           S_MemAddr <= Addr_C;
           Addr_RegEnable <= '1';
           IncrC_En <= '1';
         else
           IncrC_En <= '0';
           Addr_RegEnable <= '0';
         end if;

        if (Iter < Num_Rows+3) then
           Next_State <= State_RD_Cvector;
           Iter_En <= '1';
           Iter_Reset <= '0';

        else
           Next_State <= State_Rd_Compute;
           Iter_En <= '0';
           Iter_Reset <= '1';
        end if;


  when State_RD_Compute =>
```

```
PE_InterruptReq_n <= ’1’;
PE_MemBusReq_n <= ’0’;
S_MemWriteSel_n <= ’1’;
S_MemStrobe_n <= ’0’;
Addr_RegReset <= ’0’;
Data_RegReset <= ’0’;
Data_RegEnable <= ’0’;
RDIntVal_En <= ’0’;
iter_Reset <= ’0’;
Iter_En <= ’1’;
IncrC_En <= ’0’;
RDCi_en <= ’0’;
-- to place proper addresses on the address bus
Decr_Num_Mat <= ’0’;

if (Limit_0> 0) then
   Decr <= ’1’;
   if(Iter2 = 0) then
       S_MemAddr <= Addr_Y;
       Addr_RegEnable <= ’1’;
       IncrOffset <= ’0’;
   else
       S_MemAddr <= Addr_X + Offset;
       Addr_RegEnable <= ’1’;
       IncrOffset <= ’1’;
   end if;

   if (Iter2 = Num_Rows) then
      IncrXY_En <= ’1’;
   else
     IncrXY_En <= ’0’;
  end if;

else
  Decr <= ’0’;
  IncrOffset <= ’0’;
  Addr_RegEnable <= ’0’;
  IncrXY_En <= ’0’;

end if;



-- Read X and Y values at the correct times
if ( (Iter > 2) and (Limit_1 > 0) )then
   Decr3 <= ’1’;
   if (( iter2 = 3 ) or ((Num_Rows < 3) and
(Iter2 = 0)))then

       RDYi_En <= ’1’;
       RDXi_En <= ’0’;
   else
       RDYi_En <= ’0’;
       RDXi_En <= ’1’;
   end if;

else
   Decr3 <= ’0’;
```

```
                            RDYi_En <= '0';
                            RDXi_En <= '0';

                        end if;

                        -- Start the first counter
                        -- this gives proper sumvector index to the
adder
                        -- It gives index i when product i is available
                        -- sumindex2 is delayed by 3 clock cycles, It
provides
                        -- proper indices to latch the result of the
adder into
                        -- the sumvector registers
                        if ((Iter >= 8) and (Limit_2 > 0) )then
                            Decr8 <= '1';
                            SumIndex_En <= '1';
                            if (SumIndex = Num_Rows) then
                               SumIndex_Reset <= '1';
                            else
                               SumIndex_Reset <= '0';
                            end if;
                          else SumIndex_En <= '0';
                             SumIndex_Reset <= '1';
                             Decr8 <= '0';
                        end if;

                        If (Limitplus10 > 0) then
                           Next_State <= State_RD_Compute;
                           Decr10<= '1';
                        else
                           Next_State <= State_RD_WR;
                           Decr10 <= '0';
                        end if;


                  when State_RD_WR =>
                        PE_InterruptReq_n <= '1';
                        PE_MemBusReq_n <= '0';
                        S_MemStrobe_n <= '1';
                        S_MemWriteSel_n <= '0';
                        Addr_RegReset <= '0';
                        Addr_RegEnable <= '0';
                        Data_RegReset <= '0';
                        Data_RegEnable <= '0';
                        RDXi_En <= '0';
                        RDYi_En <= '0';
                        RDCi_En <= '0';
                        RDIntVal_En <= '0';
                        IncrXY_En <= '0';
                        IncrOffset <= '0';
                        IncrC_En <= '0';
                        iter_Reset <= '1';
                        Iter_En <= '0';
                        SumIndex_En <= '0';
                        SumIndex_Reset <= '0';
                        Decr <= '0';
```

```
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_Num_Mat <= '0';
        Next_State <= State_WR_Ans;

  when State_WR_Ans =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '0';
        S_MemStrobe_n <= '0';
        S_MemWriteSel_n <= '0';
        Addr_RegReset <= '0';
        Data_RegReset <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDIntVal_En <= '0';
        IncrOffset <= '0';
        IncrXY_En <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        Decr <= '0';
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_Num_Mat <= '0';

        if (Iter < Num_Rows) then
            Next_State <= State_WR_Ans;
            IncrC_En <= '1';
            Addr_RegEnable <= '1';
            Data_RegEnable <= '1';
            S_MemAddr <= Addr_C;
            S_MemData <= SumVector(Iter);
            iter_Reset <= '0';
            Iter_En <= '1';
        else
            Next_State <= State_Decr_nummat;
            IncrC_En <= '0';
            iter_Reset <= '1';
            Iter_En <= '0';
            Addr_RegEnable <= '0';
            Data_RegEnable <= '0';

        end if;

  when State_Decr_nummat =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '0';
        S_MemStrobe_n <= '1';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '1';
        Addr_RegEnable <= '0';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
```

```
        RDCi_En <= '0';
        RDIntVal_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        IncrC_En <= '0';
        iter_Reset <= '1';
        Iter_En <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        Decr <= '0';
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_num_mat <= '1';
        Next_State <= State_Chkfinish;

  when State_Chkfinish =>
         PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '0';
        S_MemStrobe_n <= '1';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '0';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDIntVal_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        IncrC_En <= '0';
        iter_Reset <= '1';
        Iter_En <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        Decr <= '0';
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_Num_Mat <= '0';
        if (Num_Mat = 0) then
            Next_State <= State_Interrupt;
        else
            Next_State <= State_RD_Cvector;
        end if;

  when State_Interrupt =>
        PE_InterruptReq_n <= '0';
        PE_MemBusReq_n <= '1';
        S_MemStrobe_n <= '1';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '0';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        RDXi_En <= '0';
```

```
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDIntVal_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        IncrC_En <= '0';
        iter_Reset <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        Iter_En <= '0';
        Decr <= '0';
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_Num_Mat <= '0';
        if (PE_InterruptAck_n = '0') then
              Next_State <= State_Idle;
        else
              Next_State <= State_Interrupt;
        end if;

  when State_Idle =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '1';
        S_MemStrobe_n <= '1';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '0';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDIntVal_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        IncrC_En <= '0';
        iter_Reset <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        Iter_En <= '0';
        Decr <= '0';
        Decr3 <= '0';
        Decr8 <= '0';
        Decr10 <= '0';
        Decr_Num_Mat <= '0';
        Next_State <= State_Idle;

  when others =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '1';
        S_MemStrobe_n <= '1';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '1';
        Addr_RegEnable <= '0';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
```

```
                        RDXi_En <= '0';
                        RDYi_En <= '0';
                        RDCi_En <= '0';
                        RDIntVal_En <= '0';
                        IncrXY_En <= '0';
                        IncrOffset <= '0';
                        IncrC_En <= '0';
                        iter_Reset <= '0';
                        SumIndex_En <= '0';
                        SumIndex_Reset <= '0';
                        Iter_En <= '0';
                        Decr <= '0';
                        Decr3 <= '0';
                        Decr8 <= '0';
                        Decr10 <= '0';
                        Decr_Num_Mat <= '0';
                        Next_state <= State_Init;

            end case;


      end Process;




P_Mult_Stage1 : process(Yi,Xi)
      begin
            S_s1_R1_mult <= Yi(31);
            S_s2_R1_mult <= Xi(31);
            S_e1_R1_mult <= Yi(30 downto 23);
            S_e2_R1_mult <= Xi(30 downto 23);
            S_Esum_R1_mult <= ('1'& Yi(30 downto 23)) + ( '1' & Xi(30
downto 23));
            S_f1_R1_mult <= '1' & Yi(22 downto 0);
            S_f2_R1_mult <= '1' & Xi(22 downto 0);
      end process;
      P_Mult_Stage2 : process(s1_R1_mult, s2_R1_mult, e1_R1_mult,
e2_R1_mult,
                              Esum_R1_mult, f1_R1_mult, f2_R1_mult)
      variable temp_f : std_logic_vector(47 downto 0);
      begin
            -- Check for a = 0 or b = 0 or Underflow

            if (((e1_R1_mult = e_zero) and (f1_R1_mult(22 downto 0) =
f_zero)) or
              ((e2_R1_mult = e_zero) and (f2_R1_mult(22 downto 0) =
f_zero)) or
            (Esum_R1_mult < one_two_seven) ) then

                  S_Underflow_or_Zero_mult <= '1';
                  S_Overflow_mult <= '0';

            -- check for overflow; if yes assign max number

            elsif (Esum_R1_mult > three_eight_two) then
```

```
                        S_Overflow_mult <= '1';
                        S_Underflow_or_Zero_mult <= '0';

                else
                        S_Overflow_mult <= '0';
                        S_Underflow_or_Zero_mult <= '0';


                end if;

                -- integer multiplication
                temp_f := f1_R1_mult * f2_R1_mult;
                S_f_R2_mult <= temp_f(47 downto 23);

                -- sign bit : s1 exor s2
                S_s_R2_mult <= (s1_R1_mult and not(s2_R1_mult)) or
                               (s2_R1_mult and not(s1_R1_mult));

                S_Esum_R2_mult <= Esum_R1_mult;

        end process;
        P_Mult_Stage3: process(s_R2_mult, f_R2_mult,
Underflow_or_Zero_mult,
                             Overflow_mult, ESum_R2_mult)

        begin
                -- Calculate result values

                if (Underflow_or_zero_mult = '1') then
                        -- minumum number 0
                        S_e_R3_mult <= (others => '0');
                        S_f_R3_mult <= (others => '0');

                elsif (Overflow_mult = '1') then
                -- maximum number
                        S_e_R3_mult <= (others => '1');
                        S_f_R3_mult <= (others => '1');

                else
                        -- normalize result
                        if (f_R2_mult(24) = '1') then

                                S_f_R3_mult <= f_R2_mult(23 downto 1);
                                S_e_R3_mult <= Esum_R2_mult - one_two_six;

                        else
                                S_f_R3_mult <= f_R2_mult(22 downto 0);
                                S_e_R3_mult <= Esum_R2_mult - one_two_seven;

                        end if;
                end if;

                S_s_R3_mult <= s_R2_mult;
        end process;


-- adder
```

```
-- stage 1
P_Stage1_add : process (Product,SumVector)
variable f1_var : std_logic_vector(23 downto 0);
variable f2_var : std_logic_vector(23 downto 0);
variable var_e_diff : std_logic_vector(8 downto 0);
variable Index_var : integer range 0 to 31;
begin
            if (SumIndex > (Num_Rows-1)) then
              Index_var := (Num_Rows) - 1;
            else
              Index_var := SumIndex;
            end if;

      -- Change 0.f to 1.f for Product and SumVector(Index_var) if not
equal to zero
            if not(Product(30 downto 0) = zero_31) then
                  f1_var(23) := '1';
            else
                  f1_var(23) := '0';
            end if;

            f1_var(22 downto 0) := Product(22 downto 0);
            if not(SumVector(Index_var)(30 downto 0) = zero_31) then
                  f2_var(23) := '1';
            else
                  f2_var(23) := '0';
            end if;
            f2_var(22 downto 0) := SumVector(Index_var)(22 downto 0);


            -- Check if SumVector(Index_var) > a, if true then swap a
and b to make a > b
            if (( SumVector(Index_var)(30 downto 23)>Product(30 downto
23) ) or
            ((Product(30 downto 23) = SumVector(Index_var)(30 downto
23)) and (SumVector(Index_var)(22 downto 0) > Product(22 downto 0))) )
then
            -- Swap Product and SumVector(Index_var) and separate
fields
            -- Sign
            s1_add <= SumVector(Index_var)(31);
            s2_add <= Product(31);
            -- Exponent
            e1_add(7 downto 0) <= SumVector(Index_var)(30 downto 23);

            e2_add(7 downto 0) <= Product(30 downto 23);

            -- F field
            f1_add(23 downto 0) <= f2_var;
            f2_add(23 downto 0) <= f1_var;
            else
            -- Break Product and SumVector(Index_var) up into separate
fields
            -- Sign
            s1_add <= Product(31);
            s2_add <= SumVector(Index_var)(31);
            -- Exponent
```

```
            e1_add(7 downto 0) <= Product(30 downto 23);
            e2_add(7 downto 0) <= SumVector(Index_var)(30 downto 23);
            -- F field
            f1_add(23 downto 0) <= f1_var;
            f2_add(23 downto 0) <= f2_var;
            end if;
            -- Pad f1 and f2 in order to use 32 bit hard macro
            f1_add(31 downto 24) <= "00000000";
            f2_add(31 downto 24) <= "00000000";

            if ((SumVector(Index_var)(30 downto 23) > Product(30 downto
23)) or
                  ((Product(30 downto 23) = SumVector(Index_var)(30
downto 23)) and (SumVector(Index_var)(22 downto 0) > Product(22 downto
0))) ) then
                  var_e_diff := ("0" & SumVector(Index_var)(30 downto
23)) -  ("0" & Product (30 downto 23)) ;
            else
                  var_e_diff := ("0" & Product(30 downto 23)) - ( "0" &
SumVector(Index_var)(30 downto 23));
            end if;
            S_e_diff_add <= var_e_diff;

end process;

P_Stage2_add :
process(f1_R1_add,f2_R1_add,e_diff_R1_add,s1_R1_add,s2_R1_add)
variable e_diff_var :std_logic_vector(7 downto 0);
variable f2_dly_var :std_logic_vector(23 downto 0);

begin
      if ((e_diff_R1_add(7) = '1') or (e_diff_R1_add(6) = '1') or
          (e_diff_R1_add(5) = '1') ) then
            e_diff_var(4 downto 0) := "11111";
      else
            e_diff_var(4 downto 0) := e_diff_R1_add(4 downto 0);
      end if;
      f2_dly_var := f2_R1_add(23 downto 0);
      if (e_diff_var(0) = '1') then
            f2_dly_var(22 downto 0) := f2_dly_var(23 downto 1);
            f2_dly_var(23) := '0';
      end if;
      if (e_diff_var(1) = '1') then
            f2_dly_var(21 downto 0) := f2_dly_var(23 downto 2);
            f2_dly_var(23 downto 22) := "00";
      end if;
      if (e_diff_var(2) = '1') then
            f2_dly_var(19 downto 0) := f2_dly_var(23 downto 4);
            f2_dly_var(23 downto 20) := "0000";
      end if;
      if (e_diff_var(3) = '1') then
            f2_dly_var(15 downto 0) := f2_dly_var(23 downto 8);
            f2_dly_var(23 downto 16) := "00000000";
      end if;
      if (e_diff_var(4) = '1') then
            f2_dly_var(7 downto 0) := f2_dly_var(23 downto 16);
            f2_dly_var(23 downto 8) := "0000000000000000";
```

```
      end if;

      -- Check sign to see if f_result = f1 + f2 or f1 - f2
      if (s1_R1_add = s2_R1_add) then
            S_temp_f_R2_add(31 downto 0) <= f1_R1_add + ("00000000" &
f2_dly_var );
      else
            S_temp_f_R2_add(31 downto 0) <= f1_R1_add - ("00000000" &
f2_dly_var);
      end if;

end process;

P_Stage3_add: process(temp_f_R2_add,e1_R2_add)
variable carry : std_ulogic;
variable msb   : std_ulogic;
variable temp_f_var : std_logic_vector(24 downto 0);
variable shift_val_var : std_logic_vector (4 downto 0);
variable dec_val_var : std_logic_vector (7 downto 0);
begin

      carry := temp_f_R2_add(24);
      msb := temp_f_R2_add(23);
      temp_f_var := temp_f_R2_add;

      if (carry = ’1’) then
            S_f_R3_add <= temp_f_R2_add(23 downto 1);
            S_e_R3_add <= e1_R2_add + "00000001";
      else
            -- Find position of msb
            if (msb = ’1’) then
                  shift_val_var := "00000";
            else
                  shift_val_var(4) := ( temp_f_var(0) or temp_f_var(1)
or temp_f_var(2) or temp_f_var(3) or temp_f_var(4) or
                                        temp_f_var(5) or temp_f_var(6)
or temp_f_var(7) ) and not(temp_f_var(8) or temp_f_var(9)
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21) or temp_f_var(22) ) ;

                  shift_val_var(3) := ( temp_f_var(8) or temp_f_var(9)
or temp_f_var(10) or temp_f_var(11) or temp_f_var(12)
                                   or temp_f_var(13) or temp_f_var(14)
or temp_f_var(15) ) and not(temp_f_var(16) or
                                   temp_f_var(17) or temp_f_var(18) or
temp_f_var(19) or temp_f_var(20) or temp_f_var(21) or
                                   temp_f_var(22) );

                  shift_val_var(2) := ( ( temp_f_var(0) or
temp_f_var(1) or temp_f_var(2) or temp_f_var(3) ) and not(temp_f_var(4)
or
                                        temp_f_var(5) or temp_f_var(6)
or temp_f_var(7) or temp_f_var(8) or temp_f_var(9)
```

```
                                    or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22) )) or

                                    ((temp_f_var(8) or temp_f_var(9)
or temp_f_var(10) or temp_f_var(11)) and
                                    not(temp_f_var(12) or
temp_f_var(13) or temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22) )) or

                                    ((temp_f_var(16) or temp_f_var(17)
or temp_f_var(18) or temp_f_var(19)) and
                                    not( temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22)) );

                  shift_val_var(1) := ( ( temp_f_var(0) or
temp_f_var(1)) and not(temp_f_var(2) or temp_f_var(3) or temp_f_var(4)
or
                                       temp_f_var(5) or temp_f_var(6)
or temp_f_var(7) or temp_f_var(8) or temp_f_var(9)
                                    or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22) )) or

                                    ((temp_f_var(4) or temp_f_var(5)
)and not( temp_f_var(6) or temp_f_var(7) or temp_f_var(8) or
temp_f_var(9)
                                    or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22) ))  or


                                ((temp_f_var(8) or temp_f_var(9) ) and
not(temp_f_var(10) or temp_f_var(11) or temp_f_var(12) or
temp_f_var(13) or temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22))) or

                                    ((temp_f_var(12) or
temp_f_var(13)) and not(temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
```

```
                                     or temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22))) or

                                     ((temp_f_var(16) or
temp_f_var(17)) and not(temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22))) or

                                     ((temp_f_var(20) or
temp_f_var(21)) and not( temp_f_var(22))) ;

                  shift_val_var(0) := (temp_f_var(0) and
not(temp_f_var(1) or temp_f_var(2) or temp_f_var(3) or temp_f_var(4) or
                                        temp_f_var(5) or temp_f_var(6)
or temp_f_var(7) or temp_f_var(8) or temp_f_var(9)
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22) )) or

                                     ( temp_f_var(2) and not(
temp_f_var(3) or temp_f_var(4) or
                                        temp_f_var(5) or temp_f_var(6)
or temp_f_var(7) or temp_f_var(8) or temp_f_var(9)
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21) or temp_f_var(22) )) or

                                     (temp_f_var(4) and
not(temp_f_var(5) or temp_f_var(6) or temp_f_var(7) or temp_f_var(8) or
temp_f_var(9)
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22) )) or

                                    (temp_f_var(6) and
not(temp_f_var(7) or temp_f_var(8) or temp_f_var(9)
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22))) or

                                     (temp_f_var(8) and
not(temp_f_var(9) or temp_f_var(10) or temp_f_var(11) or temp_f_var(12)
or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
```

```
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22) )) or

                                     (temp_f_var(10) and
not(temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or
temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22) )) or

                                     (temp_f_var(12) and
not(temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22))) or

                                     (temp_f_var(14) and
not(temp_f_var(15) or temp_f_var(16) or temp_f_var(17) or
temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22))) or

                                     (temp_f_var(16) and
not(temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22))) or

                                     (temp_f_var(18) and
not(temp_f_var(19) or temp_f_var(20)  or
                                     temp_f_var(21) or temp_f_var(22)))
or

                                     (temp_f_var(20) and
not(temp_f_var(21)  or temp_f_var(22))) or

                                     ( temp_f_var(22)) ;


      -- Shift f2 left shift_val_var places
            if (shift_val_var(0) = '1') then
                  temp_f_var(23 downto 1) := temp_f_var(22 downto 0);
                  temp_f_var(0) := '0';
            end if;
            if (shift_val_var(1) = '1') then
                  temp_f_var(23 downto 2) := temp_f_var(21 downto 0);
                  temp_f_var(1 downto 0) := "00";
            end if;
            if (shift_val_var(2) = '1') then
                  temp_f_var(23 downto 4) := temp_f_var(19 downto 0);
                  temp_f_var(3 downto 0) := "0000";
            end if;
            if (shift_val_var(3) = '1') then
                  temp_f_var(23 downto 8) := temp_f_var(15 downto 0);
                  temp_f_var(7 downto 0) := "00000000";
            end if;
```

```
            if (shift_val_var(4) = '1') then
                  temp_f_var(23 downto 16) := temp_f_var(7 downto 0);
                  temp_f_var(15 downto 0) := "0000000000000000";
            end if;


            end if;
            s_f_R3_add <= temp_f_var(22 downto 0);

            -- If adding a number plus it's negative the answer should
be zero
            if ( (temp_f_var = zero_25) and (e1_R2_add = e2_R2_add) and
                 (s1_R2_add = not(s2_R2_add)) ) then
                  s_e_R3_add <= zero_8;
            else
                  s_e_R3_add <= e1_R2_add - ("000" & shift_val_var);
            end if;

            s_s_R3_add <= s1_R2_add;


      end if;
end process;

      ----------------------------------------------------------------
      --  "Inactive" output port signal assignments
      ----------------------------------------------------------------
      PE_MemHoldReq_n <= '1';                 -- Disable memory hold
requests

      PE_Left_OE <= ( others => '0' );        -- Disable left port
output

      PE_Right_OE <= ( others => '0' );       -- Disable right port
output

      PE_XbarData_OE <= ( others => '0' );    -- Disable crossbar port
output
      PE_XbarData_WE <= ( others => '0' );    -- Disable crossbar port
writes

      PE_FifoSelect <= "00";                  -- Deselect fifo
                                              --   "00" selects none
                                              --   "01" selects
External I/O Fifo
                                              --   "10" selects On-
Board Fifo
                                              --   "11" selects On-
Board Mailbox
      PE_Fifo_WE_n <= '1';                    -- Disable fifo write
mode
      PE_FifoPtrIncr_EN <= '0';               -- Disable fifo pointer
increment

      PE_CPE_Bus_OE <= ( others => '0' );     -- Disable CPE bus output

end rect3;
```

# APPENDIX B

VHDL Code for PE Logic Core for Implementation 2

```
---------------------------------------------------------------------------
--
--  Entity        : PE1_Logic_Core
--
--  Architecture  : loop
--
--  Filename      : pe1lca.vhd
--  Xij*Yi = Ci
-- In the program, the Yi and ci are a partof a  big vector
-- The number of columns are constant. continous issue of saxpy
operations
-- the matrix-by-vector multiplication is done many times depending
upon the s
-- status word read from a memory location after the host acknowledges
the
-- second interrupt
---------------------------------------------------------------------------

---------------------------- Library Declarations ---------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
-------------------------- Architecture Declaration -------------------

architecture loop_alg of PE1_Logic_Core is


      function INTtoSTDLV(ARG: INTEGER) return Std_Logic_Vector is
       variable result: Std_Logic_Vector(21 downto 0);
            variable temp: integer;
            begin
            temp := ARG;
            for i in 0 to 21 loop
            if (temp mod 2) = 1 then
                  result(i) := ’1’;
            else
                  result(i) := ’0’;
            end if;
            temp := temp / 2;
            end loop;
            return result;
      end;

      function INTVAL (VAL: std_logic_vector)
      return INTEGER is
      variable SUM: INTEGER:= 0;
```

```
begin
      for  N in VAL'LOW to VAL'HIGH loop
        assert not(VAL(N) = 'X' or VAL(N) = 'Z')
        report "INTVAL inputs not 0 or 1"
            severity WARNING;
        if  VAL(N) = '1' then
          SUM := SUM + (2**N);
        end if;
      end loop;
      return SUM;
end INTVAL;

--  State machine states type (enumerated encoding)
--
type States is (
      State_Init,
      State_Ready,
      State_WaitAckFinish,
      State_RDSetup,
      State_RDPtr1,
      State_RDPtr2,
      State_RDPtr3,
      State_RDAddr,
      State_RD_Compute,
      State_RD_WR,
      State_WR_Ans,
      State_DecrLength,
      State_ChkFinish,
      State_Interrupt,
      State_AckFinish,
      State_Idle
);

type array_stdlogicVector32 is
     array (Natural range<>) of std_logic_vector(31 downto 0);
type array_stdlogicVector22 is
     array (Natural range<>) of std_logic_vector(21 downto 0);
type Number is range 0 to 31;

constant Array_Size : integer := 3;
--
--  State machine signals
--
signal Present_State    : States;
signal Next_State       : States;

signal S_MemStrobe_n    : std_ulogic;
signal S_MemWriteSel_n  : std_ulogic;
signal S_MemAddr        : std_logic_vector(21 downto 0);
signal S_MemData  : std_logic_vector(31 downto 0);
signal Data_Read  : std_logic_vector(31 downto 0);

signal Addr_Reg   : std_logic_vector(21 downto 0);
signal Addr_RegReset : std_ulogic;
signal Addr_RegEnable : std_ulogic;
signal DAta_Reg   : std_logic_vector(31 downto 0);
signal Data_RegReset : std_ulogic;
```

```
signal Data_RegEnable : std_ulogic;

signal s_result : std_logic_vector(31 downto 0);
signal Result_Register : std_logic_vector(31 downto 0);

signal S_s1_R1_mult : std_ulogic;
signal S_S2_R1_mult : std_ulogic;
signal S_e1_R1_mult : std_logic_vector(7 downto 0);
signal S_e2_R1_mult : std_logic_vector(7 downto 0);
signal S_f1_R1_mult : std_logic_vector(23 downto 0);
signal S_f2_R1_mult : std_logic_vector(23 downto 0);
signal s1_R1_mult : std_ulogic;
signal S2_R1_mult : std_ulogic;
signal e1_R1_mult : std_logic_vector(7 downto 0);
signal e2_R1_mult : std_logic_vector(7 downto 0);
signal f1_R1_mult : std_logic_vector(23 downto 0);
signal f2_R1_mult : std_logic_vector(23 downto 0);
signal S_Esum_R1_mult : std_logic_vector(8 downto 0);
signal Esum_R1_mult : std_logic_vector(8 downto 0);
signal S_s_R2_mult : std_ulogic;
signal S_Esum_R2_mult : std_logic_vector(8 downto 0);
signal S_f_R2_mult : std_logic_vector(24 downto 0);
signal S_Underflow_or_Zero_mult : std_ulogic;
signal S_Overflow_mult : std_ulogic;
signal Underflow_or_Zero_mult : std_ulogic;
signal Overflow_mult : std_ulogic;
signal Esum_R2_mult : std_logic_vector(8 downto 0);
signal s_R2_mult : std_ulogic;
signal f_R2_mult : std_logic_vector(24 downto 0);
signal S_s_R3_mult : std_ulogic;
signal S_e_R3_mult : std_logic_vector(8 downto 0);
signal S_f_R3_mult : std_logic_vector(22 downto 0);

signal s1_R1_add : std_ulogic;
signal s2_R1_add : std_ulogic;
signal E1_R1_add : std_logic_vector(7 downto 0);
signal e2_R1_add : std_logic_vector(7 downto 0);
signal f1_R1_add : std_logic_vector(31 downto 0);
signal f2_R1_add : std_logic_vector(31 downto 0);
signal e_diff_R1_add : std_logic_vector(7 downto 0);

signal s_e_diff_add : std_logic_vector(8 downto 0);
signal s1_add : std_ulogic;
signal s2_add : std_ulogic;
signal E1_add : std_logic_vector(7 downto 0);
signal e2_add : std_logic_vector(7 downto 0);
signal f1_add : std_logic_vector(31 downto 0);
signal f2_add : std_logic_vector(31 downto 0);
signal S_temp_f_R2_add : std_logic_vector(31 downto 0);
signal s1_R2_add : std_ulogic;
signal s2_R2_add : std_ulogic;
signal e1_R2_add : std_logic_vector(7 downto 0);
signal e2_R2_add : std_logic_vector(7 downto 0);
signal temp_f_R2_add : std_logic_vector(24 downto 0);
signal s_s_R3_add: std_ulogic;
signal s_e_R3_add : std_logic_vector(7 downto 0);
signal s_f_R3_add : std_logic_vector(22 downto 0);
```

```
	signal C_Addr_Ptr : std_logic_vector(21 downto 0);
	signal Y_Addr_Ptr : std_logic_vector(21 downto 0);
	signal Addr_x : std_logic_vector(21 downto 0);
	signal Addr_Y : array_stdlogicvector22(0 to 2);-- 0 to arraysize-
1
	signal Addr_C : array_stdlogicvector22(0 to 2);-- 0 to arraysize-
1

	signal Xi : std_logic_vector(31 downto 0);
	signal Yi : std_logic_vector(31 downto 0);
	signal Product : std_logic_vector(31 downto 0);
	signal SumVector: array_stdlogicvector32(0 to 2); -- 0 to
arraysize-1


	signal Decr_Length :std_ulogic;
	signal Length : integer range 0 to 65535;
	signal Incr_Yptr : std_ulogic;
	signal Incr_Cptr : std_ulogic;
	signal IncrXY_En : std_ulogic;
	signal RDXi_En : std_ulogic;
	signal RDYi_En : std_ulogic;
	signal RDCiAddr_En : std_ulogic;
	signal RDYiAddr_En : std_ulogic;
	signal RDCi_En : std_ulogic;
	signal RDPtr_En : std_ulogic;
	signal Offset : std_logic_vector(4 downto 0);
	signal IncrOffset : std_ulogic;

	signal Iter_Reset : std_ulogic;
	signal Iter_En : std_ulogic;
	signal Iter : integer range 0 to 63;
	signal Iter2 : integer range 0 to 31;
	signal Y_Index : integer range 0 to 31;
	signal SumIndex_Reset : std_ulogic;
	signal SumIndex_En : std_ulogic;
	signal SumIndex : integer range 0 to 31;
	signal Delay1 : integer range 0 to 31;
	signal SumIndex2 : integer range 0 to 31;

	constant e_zero: std_logic_vector(7 downto 0) := "00000000";
	constant f_zero: std_logic_vector(22 downto 0) :=
"00000000000000000000000";
	constant one_two_seven:    std_logic_vector(8 downto 0) :=
"001111111";
	constant one_two_six:     std_logic_vector(8 downto 0) :=
"001111110";
	constant three_eight_two : std_logic_vector(8 downto 0) :=
"101111110";
	constant zero_7:           std_logic_vector(6 downto 0) :=
"0000000";
	constant zero_8:           std_logic_vector(7 downto 0) :=
"00000000";
	constant zero_25:          std_logic_vector(24 downto 0) :=
"0000000000000000000000000";
```

```
      constant zero_31:           std_logic_vector(30 downto 0) :=
"0000000000000000000000000000000";
      constant Address_Zero    : std_logic_vector(21 downto 0) :=
"0000000000000000000000";
      constant Address_One     : std_logic_vector(21 downto 0) :=
"0000000000000000000001";
      constant Address_Two     : std_logic_vector(21 downto 0) :=
"0000000000000000000010";
      constant Address_Three   : std_logic_vector(21 downto 0) :=
"0000000000000000000011";
      constant Address_Four    : std_logic_vector(21 downto 0) :=
"0000000000000000000100";
      constant All_Ones : std_logic_vector(31 downto 0) :=
"11111111111111111111111111111111";


begin
      ----------------------------
      --    Signal assignment statements
      ----------------------------

      --
      -- Address Register
      --

      process (PE_PClk, PE_Reset)
      begin
            if (PE_Reset = '1') then
                        Addr_Reg <= (others => '0');
            elsif ( ( PE_Pclk = '1' ) and ( PE_Pclk'event ) ) then
                        if (Addr_RegReset /= '1') then
                              if (Addr_RegEnable = '1') then

                                    Addr_Reg <= S_MemAddr;
                              end if;

                        else

                              Addr_Reg <= (others=>'0');
                        end if;

            end if;

      end process;
      --
      -- Data reg
      --
      process (PE_PClk, PE_Reset)
            begin
                  if (PE_Reset = '1') then
                        Data_Reg <= (others => '0');
                  elsif ( ( PE_Pclk = '1' ) and ( PE_Pclk'event ) )
then
                        if (Data_RegReset /= '1') then
                              if (Data_RegEnable = '1') then

                                    Data_Reg <= S_MemData;
```

```
                        end if;

                  else

                        Data_Reg <= (others=>'0');
                  end if;

      end if;

end process;


P_Iteration_counter: process(PE_Reset,PE_PClk)
begin
      if (PE_Reset = '1') then
            Iter <= 0;
            Iter2 <= 0;
      elsif ( (PE_Pclk = '1' ) and ( PE_Pclk'event ) ) then

            if (Iter_reset /= '1') then
                  if (Iter_En = '1') then
                     Iter <= Iter + 1;

                     If (Iter2 < Array_Size) then
                        Iter2 <= Iter2 +1;
                     else
                        Iter2 <= 0;
                     end if;

                  end if;
            else
                  Iter <= 0;
                  Iter2 <= 0;
            end if;
      end if;
end process;

P_SumIndex_counter: process(PE_Reset,PE_PClk)
begin
      if (PE_Reset = '1') then
            SumIndex <= 0;
            Delay1 <= 0;
            SumIndex2 <= 0;

      elsif ( (PE_Pclk = '1' ) and ( PE_Pclk'event ) ) then

            if (SumIndex_Reset /= '1') then
                  if (SumIndex_En = '1') then
                     SumIndex <= SumIndex + 1;
                  end if;
            else
                  SumIndex <= 0;
            end if;

            -- delay by three cycles
            Delay1 <= SumIndex;
            SumIndex2 <= Delay1;
```

```
            end if;
      end process;

      -- Synchronous process that clocks the state machines, and
handles
            --   reset conditions.
            P_Sync: process (PE_Reset, PE_Pclk)

            begin

                  if ( PE_Reset = '1' ) then
                        Present_State <= State_Init;

                        s1_R1_mult <= '0';
                        s2_R1_mult <= '0';
                        e1_R1_mult <= (others => '0');
                        e2_R1_mult <= (others => '0');
                        f1_R1_mult <= (others => '0');
                        f2_R1_mult <= (others => '0');
                        ESum_R1_mult <= (others => '0');

                        -- latch stage2
                        s_R2_mult <= '0';
                        Esum_R2_mult <= (others => '0');
                        f_R2_mult <= (others => '0');
                        Underflow_or_zero_mult <= '0';
                        Overflow_mult <= '0';

                        -- latch stage3
                        Product <= (others => '0');

                        -- adder latches
                        s1_R1_add <= '0';
                        s2_R1_add <= '0';
                        e1_R1_add <= (others => '0');
                        e2_R1_add <= (others => '0');
                         f1_R1_add <= (others => '0');
                        f2_R1_add <= (others => '0');
                        e_diff_R1_add <= (others => '0');

                        -- latch outputs of second stage
                        s1_R2_add <= '0';
                        s2_R2_add <= '0';
                        e1_R2_add <= (others => '0');
                        e2_R2_add <=(others => '0');
                        temp_f_R2_add <= (others => '0');

                        -- output from third stage
                        SumVector <= (others => (others => '0'));


                  elsif ( ( PE_Pclk = '1' ) and ( PE_Pclk'event ) )
then
                        Present_State <= Next_State;
                        -- latch stage1
                        s1_R1_mult <= S_s1_R1_mult;
```

```
                        s2_R1_mult <= S_s2_R1_mult;
                        e1_R1_mult <= S_e1_R1_mult;
                        e2_R1_mult <= S_E2_R1_mult;
                        f1_R1_mult <= S_f1_R1_mult;
                        f2_R1_mult<= S_f2_R1_mult;
                        ESum_R1_mult <= S_Esum_R1_mult;

                        -- latch stage2
                        s_R2_mult <= S_s_R2_mult;
                        Esum_R2_mult <= S_Esum_R2_mult;
                        f_R2_mult<= S_f_R2_mult;
                        Underflow_or_zero_mult <=
S_Underflow_or_Zero_mult;
                        Overflow_mult <= S_Overflow_mult;

                        -- latch stage3
                        Product <= S_s_R3_mult & S_e_R3_mult(7 downto
0) & S_f_R3_mult;



                        -- adder latches

                        s1_R1_add <= s1_add;
                        s2_R1_add <= s2_add;
                        e1_R1_add <= e1_add;
                        e2_R1_add <= e2_add;
                        f1_R1_add <= f1_add;
                        f2_R1_add <= f2_add;
                        e_diff_R1_add <= s_e_diff_add(7 downto 0);

                        -- latch outputs of second stage
                         s1_R2_add <= s1_R1_add;
                         s2_R2_add <= s2_R1_add;
                        e1_R2_add <= E1_R1_add;
                        e2_R2_add <= e2_R1_add;
                        temp_f_R2_add <= S_temp_f_R2_add(24 downto 0);

                        -- output from third stage

                        if (RDCi_En = ’1’) then
                            SumVector(Iter - 2*Array_size - 3) <=
PE_MemData_Inreg;
                        elsif ((Present_State = State_RD_Compute) and
                               (Iter > 9) and (Iter <
(Array_Size*(Array_Size+1) + 10))
                               and (SumIndex2 < Array_Size)) then
                             SumVector(Sumindex2 ) <= s1_R2_add &
s_e_R3_add & s_f_R3_add;
                        end if;

                  end if;

            end process P_Sync;

      P_sync2_Addr : process(PE_Reset, PE_PClk)
```

```
      begin
          if (PE_Reset = '1') then
            Y_Index <= 0;
            Addr_C <= (others =>(others => '0'));
            Addr_Y <= (others =>(others => '0'));
            Offset <= (others => '0');

          elsif ( ( PE_Pclk = '1' ) and ( PE_Pclk'event ) ) then

            if (RDCiAddr_En = '1') then
                Addr_C(Iter-3) <= PE_MemData_Inreg(21 downto 0);
                Y_Index <= 0;
            elsif (RDYiAddr_En = '1') then
                Addr_Y(Iter - Array_size - 3) <= PE_MemData_Inreg(21
downto 0);
            elsif (IncrXY_En = '1') then
                Y_Index <= Y_Index + 1;
            end if;

            if (IncrXY_En = '1') then
               offset <= (others => '0');
            elsif (IncrOffset = '1') then
               Offset <= Offset + Array_Size;
            end if;


          end if;

      end process;

      P_Sync_ADDRPtr : process(PE_Reset, PE_PClk)

      begin
          if (PE_Reset = '1') then
            Length <= 0;
              C_Addr_Ptr <= (others => '0');
              Y_Addr_Ptr <= (others => '0');
              Addr_X <= (others => '0');

          elsif ( (PE_PClk'Event) and (PE_PClk = '1') ) then
              if (RDPtr_En = '1') then
                 Length <= INTVAL(C_Addr_Ptr);
                 C_Addr_Ptr <= Y_Addr_Ptr;
                 Y_Addr_Ptr <= Addr_X;
                 Addr_X <= PE_MemData_Inreg(21 downto 0);
              else
                 if (Incr_YPtr = '1') then
                    Y_Addr_Ptr <= Y_Addr_Ptr + 1;
                 end if;
                 if (Incr_CPtr = '1') then
                    C_Addr_Ptr <= C_Addr_Ptr + 1;
                 end if;
                 if (IncrXY_En = '1') then
                    Addr_X <= Addr_X + 1;
                 elsif (Decr_Length = '1') then
                    Addr_X <= Addr_X + (Array_Size)*(Array_Size - 1);
                 end if;
```

```
          if (Decr_Length = '1') then
             Length <= Length - 1;
          end if;
      end if;
   end if;
end process;




P_Sync_Data : process (PE_Reset, PE_PClk)
begin
      if (PE_Reset = '1') then

            Xi <= (others => '0');
            Yi <= (others => '0');
      elsif ( ( PE_Pclk = '1' ) and ( PE_Pclk'event ) ) then

            if (RDXi_En = '1') then
                  Xi <= PE_MemData_Inreg;
            end if;
            if (RDYi_En = '1') then
                  Yi <= PE_MemData_Inreg;
            end if;
      end if;
 end process;


-- Asynchronous signal assignment statements

PE_MemStrobe_n <= S_MemStrobe_n;
PE_MemWriteSel_n <= S_MemWriteSel_n;
PE_MemAddr_OutReg <= Addr_Reg(21 downto 0);
PE_MemData_OutReg <= Data_Reg(31 downto 0);


--------------------------------
-- Combinational state machine
--------------------------------
P_ASync: process(Present_State, PE_MemBusGrant_n,
                       PE_InterruptAck_n, iter)

      begin


            case Present_State is
                  -- Initialization state.
            when State_Init =>
                  PE_InterruptReq_n <= '1';
                  PE_MemBusReq_n <= '1';
                  S_MemWriteSel_n <= '0';
                  S_MemStrobe_n <= '1';
                  Addr_RegReset <= '1';
                  Addr_RegEnable <= '0';
                  Data_RegReset <= '1';
                  Data_RegEnable <= '0';
                  RDXi_En <= '0';
```

```
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDCiAddr_En <= '0';
        RDYIAddr_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        iter_Reset <= '1';
        Iter_En <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '1';
        Decr_Length <= '0';
        RdPtr_En <= '0';
        Incr_Yptr <= '0';
        Incr_Cptr <= '0';
        Next_State <= State_Ready;

  -- Interrupt the host; PE ready
  when State_Ready =>
        PE_InterruptReq_n <= '0';
        PE_MemBusReq_n <= '1';
        S_MemWriteSel_n <= '0';
        S_MemStrobe_n <= '1';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDCiAddr_En <= '0';
        RDYIAddr_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        iter_Reset <= '0';
        Iter_En <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        Decr_Length <= '0';
        RdPtr_En <= '0';
        Incr_Yptr <= '0';
        Incr_Cptr <= '0';
        if (PE_InterruptAck_n = '0')then
              Next_State <= State_WaitAckFinish;
        else Next_State <= State_Ready;
        end if;

  -- Wait for ACK to go high again
  when State_WaitAckFinish =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '1';
        S_MemWriteSel_n <= '0';
        S_MemStrobe_n <= '1';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '0';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        RDXi_En <= '0';
```

```
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDCiAddr_En <= '0';
        RDYIAddr_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        iter_Reset <= '0';
        Iter_En <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        Decr_Length <= '0';
        RdPtr_En <= '0';
        Decr_Length <= '0';
        RdPtr_En <= '0';
        Incr_Yptr <= '0';
        Incr_Cptr <= '0';
        if (PE_InterruptAck_n = '0')then
              Next_State <= State_WaitAckFinish;
        else Next_State <= State_RDSetup;
        end if;

  -- Read the initial values for the addresses
  -- and the scalar A
  when State_RDSetup =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '0';
        S_MemStrobe_n <= '0';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '1';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDCiAddr_En <= '0';
        RDYIAddr_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        Iter_En <= '0';
        iter_Reset <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        S_MemAddr <= Address_one;
        Decr_Length <= '0';
        RdPtr_En <= '0';
        Incr_Yptr <= '0';
        Incr_Cptr <= '0';
        if (PE_MemBusGrant_n = '0') then
          Next_State <= State_RDPtr1;
        else
          Next_State <= State_RDSetup;
        end if;

  when State_RDPtr1 =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '0';
```

```
        S_MemStrobe_n <= '0';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '1';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDCiAddr_En <= '0';
        RDYIAddr_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        Iter_En <= '0';
        iter_Reset <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        S_MemAddr <= Address_two;
        Decr_Length <= '0';
        Incr_Yptr <= '0';
        Incr_Cptr <= '0';
        RdPtr_En <= '0';
        Next_State <= State_RDPtr2;

  when State_RDPtr2 =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '0';
        S_MemStrobe_n <= '0';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '0';
        Addr_RegEnable <= '1';
        Data_RegReset <= '0';
        Data_RegEnable <= '0';
        RDXi_En <= '0';
        RDYi_En <= '0';
        RDCi_En <= '0';
        RDCiAddr_En <= '0';
        RDYIAddr_En <= '0';
        IncrXY_En <= '0';
        IncrOffset <= '0';
        Iter_En <= '0';
        iter_Reset <= '0';
        SumIndex_En <= '0';
        SumIndex_Reset <= '0';
        S_MemAddr <= Address_Three;
        Decr_Length <= '0';
        RdPtr_En <= '0';
        Incr_Yptr <= '0';
        Incr_Cptr <= '0';
        Next_State <= State_RDPtr3;

  when State_RDPtr3 =>
        PE_InterruptReq_n <= '1';
        PE_MemBusReq_n <= '0';
        S_MemStrobe_n <= '0';
        S_MemWriteSel_n <= '1';
        Addr_RegReset <= '0';
```

```
                        Addr_RegEnable <= '1';
                        Data_RegReset <= '0';
                        Data_RegEnable <= '0';
                        RDXi_En <= '0';
                        RDYi_En <= '0';
                        RDCi_En <= '0';
                        RDCiAddr_En <= '0';
                        RDYIAddr_En <= '0';
                        IncrXY_En <= '0';
                        IncrOffset <= '0';
                        SumIndex_En <= '0';
                        SumIndex_Reset <= '0';
                        RDPtr_en <= '1';
                        S_MemAddr <= Address_four;
                        Decr_Length <= '0';
                        Incr_Yptr <= '0';
                        Incr_Cptr <= '0';
                        if (Iter = 3) then
                            Next_State <= State_RDAddr;
                            Iter_Reset <= '1';
                            Iter_en <= '0';
                        else
                            Next_State <= State_RDPtr3;
                            Iter_Reset <= '0';
                            Iter_en <= '1';
                        end if;


                  -- We are reading from the first address location.
                  when State_RDAddr =>
                        PE_InterruptReq_n <= '1';
                        PE_MemBusReq_n <= '0';
                        S_MemStrobe_n <= '0';
                        S_MemWriteSel_n <= '1';
                        Addr_RegReset <= '0';
                        Addr_RegEnable <= '1';
                        Data_RegReset <= '0';
                        Data_RegEnable <= '0';
                        RDXi_En <= '0';
                        RDYi_En <= '0';
                        IncrXY_En <= '0';
                        IncrOffset <= '0';
                        SumIndex_En <= '0';
                        SumIndex_Reset <= '0';
                        Decr_Length <= '0';
                        RdPtr_En <= '0';

                        if ((Iter >= 3) and (Iter < Array_size + 3))
then
                              RDCi_En <= '0';
                              RDCiAddr_En <= '1';
                              RDYIAddr_En <= '0';
                        elsif ( (Iter >= Array_size + 3) and (Iter <
(2*Array_size + 3)) ) then
                                RDCi_En <= '0';
                              RDYiAddr_En <= '1';
                                RDCIAddr_En <= '0';
```

```
                        elsif ( (Iter >= 2*Array_size +3) and (Iter <
3*Array_size + 3)) then
                                RDCi_En <= '1';
                                RDCiAddr_En <= '0';
                                RDYIAddr_En <= '0';
                        else    RDCiAddr_En <= '0';
                                RDYIAddr_En <= '0';
                                RDCi_En <= '0';
                        end if;

                        if (Iter < Array_Size) then
                            S_MemAddr <= C_addr_Ptr;
                            Incr_Cptr <= '1';
                            Incr_Yptr <= '0';
                        elsif (Iter < 2*Array_size) then
                            S_MemAddr <= Y_Addr_Ptr;
                            Incr_Yptr <= '1';
                            Incr_Cptr <= '0';
                        elsif (Iter < 3*Array_Size ) then
                            S_MemAddr <= Addr_C(Iter- 2*Array_size);
                            Incr_Yptr <= '0';
                            Incr_Cptr <= '0';
                        end if;


                        if (Iter < Array_Size*3 + 3) then

                            Next_State <= State_RDAddr;
                            Iter_En <= '1';
                            Iter_Reset <= '0';

                        else
                            Next_State <= State_RD_Compute;
                            Iter_En <= '0';
                            Iter_Reset <= '1';
                        end if;
                  when State_RD_Compute =>
                        PE_InterruptReq_n <= '1';
                        PE_MemBusReq_n <= '0';
                        S_MemWriteSel_n <= '1';
                        Addr_RegReset <= '0';
                        Data_RegReset <= '0';
                        Data_RegEnable <= '0';
                        iter_Reset <= '0';
                        Iter_En <= '1';
                        RDCi_En <= '0';
                        RDCiAddr_En <= '0';
                        RDYIAddr_En <= '0';
                        Decr_Length <= '0';
                        RdPtr_En <= '0';
                        Incr_Yptr <= '0';
                        Incr_Cptr <= '0';


                        -- to place proper addresses on the address bus

                        if (Iter < (Array_Size*(Array_Size+1))) then
```

```
                        if(Iter2 = 0) then
                            S_MemAddr <= Addr_Y(Y_index);
                            Addr_RegEnable <= '1';
                            IncrOffset <= '0';
                        else
                            S_MemAddr <= Addr_X + Offset;
                            Addr_RegEnable <= '1';
                            IncrOffset <= '1';
                        end if;

                        if (Iter2 = Array_size) then
                           IncrXY_En <= '1';
                        else
                          IncrXY_En <= '0';
                      end if;
                    else
                      IncrXY_En <= '0';
                      IncrOffset <= '0';
                      Addr_RegEnable <= '0';
                    end if;
                    if (Iter < (Array_Size*(Array_Size+1) + 1))
then
                        S_MemStrobe_n <= '0';
                    else
                        S_MemStrobe_n <= '1';
                    end if;


                    -- Read X and Y values at the correct times
                    if ( (Iter > 2) and (Iter <
(Array_Size*(Array_Size+1) +3)) )then

                        if ( iter2 = 3 ) then
                            RDYi_En <= '1';
                            RDXi_En <= '0';
                        else
                            RDYi_En <= '0';
                            RDXi_En <= '1';
                        end if;

                    else

                        RDYi_En <= '0';
                        RDXi_En <= '0';

                    end if;

                    -- Start the first counter
                    -- this gives proper sumvector index to the
adder
                    -- It gives index i when product i is available
                    -- sumindex2 is delayed by 3 clock cycles, It
provides
                    -- proper indices to latch the result of the
adder into
                    -- the sumvector registers
```

```
                        if (Iter >= 8) and (Iter <
(Array_size*(Array_Size+1) + 8))then
                            SumIndex_En <= '1';
                            if (SumIndex = Array_size) then
                               SumIndex_Reset <= '1';
                            else
                               SumIndex_Reset <= '0';
                            end if;
                          else SumIndex_En <= '0';
                             SumIndex_Reset <= '1';
                        end if;

                        If (Iter < (Array_Size*(Array_Size+1) + 10))
then
                           Next_State <= State_RD_Compute;
                        else
                           Next_State <= State_RD_WR;
                        end if;


                  when State_RD_WR =>
                        PE_InterruptReq_n <= '1';
                        PE_MemBusReq_n <= '0';
                        S_MemStrobe_n <= '1';
                        S_MemWriteSel_n <= '1';
                        Addr_RegReset <= '1';
                        Addr_RegEnable <= '0';
                        Data_RegReset <= '0';
                        Data_RegEnable <= '0';
                        RDXi_En <= '0';
                        RDYi_En <= '0';
                        RDCi_En <= '0';
                        RDCiAddr_En <= '0';
                        RDYIAddr_En <= '0';
                        IncrXY_En <= '0';
                        IncrOffset <= '0';
                        iter_Reset <= '1';
                        Iter_En <= '0';
                        SumIndex_En <= '0';
                        SumIndex_Reset <= '0';
                        Decr_Length <= '0';
                        RdPtr_En <= '0';
                        Incr_Yptr <= '0';
                        Incr_Cptr <= '0';
                        Next_State <= State_WR_Ans;

                  when State_WR_Ans =>
                        PE_InterruptReq_n <= '1';
                        PE_MemBusReq_n <= '0';
                        S_MemStrobe_n <= '0';
                        S_MemWriteSel_n <= '0';
                        Addr_RegReset <= '0';
                        Data_RegReset <= '0';
                        RDXi_En <= '0';
                        RDYi_En <= '0';
                        RDCi_En <= '0';
                        RDCiAddr_En <= '0';
```

```
            RDYIAddr_En <= '0';
            IncrOffset <= '0';
            IncrXY_En <= '0';
            SumIndex_En <= '0';
            SumIndex_Reset <= '0';
            Decr_Length <= '0';
            RdPtr_En <= '0';
            Incr_Yptr <= '0';
            Incr_Cptr <= '0';

            if (Iter < Array_Size) then
                Next_State <= State_WR_Ans;

                Addr_RegEnable <= '1';
                Data_RegEnable <= '1';
                S_MemAddr <= Addr_C(Iter);
                S_MemData <= SumVector(Iter);
                iter_Reset <= '0';
                Iter_En <= '1';
            else
                Next_State <= State_DecrLength;

                iter_Reset <= '1';
                Iter_En <= '0';
                Addr_RegEnable <= '0';
                Data_RegEnable <= '0';

            end if;

      when State_DecrLength =>
            PE_InterruptReq_n <= '1';
            PE_MemBusReq_n <= '0';
            S_MemStrobe_n <= '1';
            S_MemWriteSel_n <= '1';
            Addr_RegReset <= '0';
            Addr_RegEnable <= '0';
            Data_RegReset <= '0';
            Data_RegEnable <= '0';
            RDXi_En <= '0';
            RDYi_En <= '0';
            RDCi_En <= '0';
            RDCiAddr_En <= '0';
            RDYIAddr_En <= '0';
            IncrXY_En <= '0';
            IncrOffset <= '0';
            iter_Reset <= '1';
            Iter_En <= '0';
            SumIndex_En <= '0';
            SumIndex_Reset <= '0';
            RdPtr_En <= '0';
            Decr_Length <= '1';
            Incr_Yptr <= '0';
            Incr_Cptr <= '0';
            Next_State <= State_Chkfinish;
```

```
when State_Chkfinish =>
      PE_InterruptReq_n <= '1';
      PE_MemBusReq_n <= '0';
      S_MemStrobe_n <= '1';
      S_MemWriteSel_n <= '1';
      Addr_RegReset <= '0';
      Addr_RegEnable <= '0';
      Data_RegReset <= '0';
      Data_RegEnable <= '0';
      RDXi_En <= '0';
      RDYi_En <= '0';
      RDCi_En <= '0';
      RDCiAddr_En <= '0';
      RDYIAddr_En <= '0';
      IncrXY_En <= '0';
      IncrOffset <= '0';
      iter_Reset <= '1';
      Iter_En <= '0';
      SumIndex_En <= '0';
      SumIndex_Reset <= '0';
      Decr_Length <= '0';
      RdPtr_En <= '0';
      Incr_Yptr <= '0';
      Incr_Cptr <= '0';

      if (Length = 0) then
          Next_State <= State_Interrupt;
      else
          Next_State <= State_RDAddr;
      end if;


when State_Interrupt =>
      PE_InterruptReq_n <= '0';
      PE_MemBusReq_n <= '1';
      S_MemStrobe_n <= '1';
      S_MemWriteSel_n <= '1';
      Addr_RegReset <= '0';
      Addr_RegEnable <= '0';
      Data_RegReset <= '0';
      Data_RegEnable <= '0';
      RDXi_En <= '0';
      RDYi_En <= '0';
      RDCi_En <= '0';
      RDCiAddr_En <= '0';
      RDYIAddr_En <= '0';
      IncrXY_En <= '0';
      IncrOffset <= '0';
      iter_Reset <= '0';
      SumIndex_En <= '0';
      SumIndex_Reset <= '0';
      Iter_En <= '0';
      Decr_Length <= '0';
      RdPtr_En <= '0';
      Incr_Yptr <= '0';
      Incr_Cptr <= '0';
```

```
		if (PE_InterruptAck_n = ’0’) then
			Next_State <= State_AckFinish;
		else
			Next_State <= State_Interrupt;
		end if;



when State_AckFinish =>

	PE_InterruptReq_n <= ’1’;
		PE_MemBusReq_n <= ’1’;
		S_MemStrobe_n <= ’1’;
		S_MemWriteSel_n <= ’1’;
		Addr_RegReset <= ’0’;
		Addr_RegEnable <= ’0’;
		Data_RegReset <= ’0’;
		Data_RegEnable <= ’0’;
		RDXi_En <= ’0’;
		RDYi_En <= ’0’;
		RDCi_En <= ’0’;
		RDCiAddr_En <= ’0’;
		RDYIAddr_En <= ’0’;
		IncrXY_En <= ’0’;
		IncrOffset <= ’0’;
		iter_Reset <= ’0’;
		SumIndex_En <= ’0’;
		SumIndex_Reset <= ’0’;
		Iter_En <= ’0’;
		Decr_Length <= ’0’;
		RdPtr_En <= ’0’;
		Incr_Yptr <= ’0’;
		Incr_Cptr <= ’0’;

		if (PE_InterruptAck_n = ’1’) then
			Next_State <= State_Idle;
		else
			Next_State <= State_AckFinish;
		end if;




	when State_Idle =>
		PE_InterruptReq_n <= ’1’;
		PE_MemBusReq_n <= ’1’;
		S_MemStrobe_n <= ’1’;
		S_MemWriteSel_n <= ’1’;
		Addr_RegReset <= ’0’;
		Addr_RegEnable <= ’0’;
		Data_RegReset <= ’0’;
		Data_RegEnable <= ’0’;
		RDXi_En <= ’0’;
		RDYi_En <= ’0’;
		RDCi_En <= ’0’;
		RDCiAddr_En <= ’0’;
		RDYIAddr_En <= ’0’;
```

```
                        IncrXY_En <= '0';
                        IncrOffset <= '0';
                        iter_Reset <= '0';
                        SumIndex_En <= '0';
                        SumIndex_Reset <= '0';
                        Iter_En <= '0';
                        Decr_Length <= '0';
                        RdPtr_En <= '0';
                        Incr_Yptr <= '0';
                        Incr_Cptr <= '0';
                        Next_State <= State_Idle;

                  when others =>
                        PE_InterruptReq_n <= '1';
                        PE_MemBusReq_n <= '1';
                        S_MemStrobe_n <= '1';
                        S_MemWriteSel_n <= '1';
                        Addr_RegReset <= '1';
                        Addr_RegEnable <= '0';
                        Data_RegReset <= '0';
                        Data_RegEnable <= '0';
                        RDXi_En <= '0';
                        RDYi_En <= '0';
                        RDCi_En <= '0';
                        RDCiAddr_En <= '0';
                        RDYIAddr_En <= '0';
                        IncrXY_En <= '0';
                        IncrOffset <= '0';
                        iter_Reset <= '0';
                        SumIndex_En <= '0';
                        SumIndex_Reset <= '0';
                        Iter_En <= '0';
                        Decr_Length <= '0';
                        RdPtr_En <= '0';
                        Incr_Yptr <= '0';
                        Incr_Cptr <= '0';
                        Next_state <= State_Init;

            end case;


      end Process;




P_Mult_Stage1 : process(Yi,Xi)
      begin
            S_s1_R1_mult <= Yi(31);
            S_s2_R1_mult <= Xi(31);
            S_e1_R1_mult <= Yi(30 downto 23);
            S_e2_R1_mult <= Xi(30 downto 23);
            S_Esum_R1_mult <= ('1'& Yi(30 downto 23)) + ( '1' & Xi(30
downto 23));
            S_f1_R1_mult <= '1' & Yi(22 downto 0);
            S_f2_R1_mult <= '1' & Xi(22 downto 0);
      end process;
```

```
      P_Mult_Stage2 : process(s1_R1_mult, s2_R1_mult, e1_R1_mult,
e2_R1_mult,
                              Esum_R1_mult, f1_R1_mult, f2_R1_mult)
      variable temp_f : std_logic_vector(47 downto 0);
      begin
            -- Check for a = 0 or b = 0 or Underflow

            if (((e1_R1_mult = e_zero) and (f1_R1_mult(22 downto 0) =
f_zero)) or
              ((e2_R1_mult = e_zero) and (f2_R1_mult(22 downto 0) =
f_zero)) or
            (Esum_R1_mult < one_two_seven) ) then

                  S_Underflow_or_Zero_mult <= ’1’;
                  S_Overflow_mult <= ’0’;

            -- check for overflow; if yes assign max number

            elsif (Esum_R1_mult > three_eight_two) then

                  S_Overflow_mult <= ’1’;
                  S_Underflow_or_Zero_mult <= ’0’;

            else
                  S_Overflow_mult <= ’0’;
                  S_Underflow_or_Zero_mult <= ’0’;


            end if;

            -- integer multiplication
            temp_f := f1_R1_mult * f2_R1_mult;
            S_f_R2_mult <= temp_f(47 downto 23);

            -- sign bit : s1 exor s2
            S_s_R2_mult <= (s1_R1_mult and not(s2_R1_mult)) or
                           (s2_R1_mult and not(s1_R1_mult));

            S_Esum_R2_mult <= Esum_R1_mult;

      end process;
      P_Mult_Stage3: process(s_R2_mult, f_R2_mult,
Underflow_or_Zero_mult,
                         Overflow_mult, ESum_R2_mult)

      begin
            -- Calculate result values

            if (Underflow_or_zero_mult = ’1’) then
                  -- minumum number 0
                  S_e_R3_mult <= (others => ’0’);
                  S_f_R3_mult <= (others => ’0’);

            elsif (Overflow_mult = ’1’) then
            -- maximum number
                  S_e_R3_mult <= (others => ’1’);
                  S_f_R3_mult <= (others => ’1’);
```

```
            else
                  -- normalize result
                  if (f_R2_mult(24) = '1') then

                        S_f_R3_mult <= f_R2_mult(23 downto 1);
                        S_e_R3_mult <= Esum_R2_mult - one_two_six;

                  else
                        S_f_R3_mult <= f_R2_mult(22 downto 0);
                        S_e_R3_mult <= Esum_R2_mult - one_two_seven;

                  end if;
            end if;

            S_s_R3_mult <= s_R2_mult;
      end process;


-- adder
-- stage 1
P_Stage1_add : process (Product,SumVector)
variable f1_var : std_logic_vector(23 downto 0);
variable f2_var : std_logic_vector(23 downto 0);
variable var_e_diff : std_logic_vector(8 downto 0);
variable Index_var : integer range 0 to 31;
begin
            if (SumIndex > Array_size-1) then
              Index_var := Array_size-1;
            else
              Index_var := SumIndex;
            end if;

      -- Change 0.f to 1.f for Product and SumVector(Index_var) if not
equal to zero
            if not(Product(30 downto 0) = zero_31) then
                  f1_var(23) := '1';
            else
                  f1_var(23) := '0';
            end if;

            f1_var(22 downto 0) := Product(22 downto 0);
            if not(SumVector(Index_var)(30 downto 0) = zero_31) then
                  f2_var(23) := '1';
            else
                  f2_var(23) := '0';
            end if;
            f2_var(22 downto 0) := SumVector(Index_var)(22 downto 0);


            -- Check if SumVector(Index_var) > a, if true then swap a
and b to make a > b
            if (( SumVector(Index_var)(30 downto 23)>Product(30 downto
23) ) or
            ((Product(30 downto 23) = SumVector(Index_var)(30 downto
23)) and (SumVector(Index_var)(22 downto 0) > Product(22 downto 0))) )
then
```

```
            -- Swap Product and SumVector(Index_var) and separate
fields
            -- Sign
            s1_add <= SumVector(Index_var)(31);
            s2_add <= Product(31);
            -- Exponent
            e1_add(7 downto 0) <= SumVector(Index_var)(30 downto 23);

            e2_add(7 downto 0) <= Product(30 downto 23);

            -- F field
            f1_add(23 downto 0) <= f2_var;
            f2_add(23 downto 0) <= f1_var;
            else
            -- Break Product and SumVector(Index_var) up into separate
fields
            -- Sign
            s1_add <= Product(31);
            s2_add <= SumVector(Index_var)(31);
            -- Exponent
            e1_add(7 downto 0) <= Product(30 downto 23);
            e2_add(7 downto 0) <= SumVector(Index_var)(30 downto 23);
            -- F field
            f1_add(23 downto 0) <= f1_var;
            f2_add(23 downto 0) <= f2_var;
            end if;
            -- Pad f1 and f2 in order to use 32 bit hard macro
            f1_add(31 downto 24) <= "00000000";
            f2_add(31 downto 24) <= "00000000";

            if ((SumVector(Index_var)(30 downto 23) > Product(30 downto
23)) or
                  ((Product(30 downto 23) = SumVector(Index_var)(30
downto 23)) and (SumVector(Index_var)(22 downto 0) > Product(22 downto
0))) ) then
                  var_e_diff := ("0" & SumVector(Index_var)(30 downto
23)) -  ("0" & Product (30 downto 23)) ;
            else
                  var_e_diff := ("0" & Product(30 downto 23)) - ( "0" &
SumVector(Index_var)(30 downto 23));
            end if;
            S_e_diff_add <= var_e_diff;

end process;

P_Stage2_add :
process(f1_R1_add,f2_R1_add,e_diff_R1_add,s1_R1_add,s2_R1_add)
variable e_diff_var :std_logic_vector(7 downto 0);
variable f2_dly_var :std_logic_vector(23 downto 0);

begin
      if ((e_diff_R1_add(7) = ’1’) or (e_diff_R1_add(6) = ’1’) or
          (e_diff_R1_add(5) = ’1’) ) then
            e_diff_var(4 downto 0) := "11111";
      else
            e_diff_var(4 downto 0) := e_diff_R1_add(4 downto 0);
      end if;
```

```
	f2_dly_var := f2_R1_add(23 downto 0);
	if (e_diff_var(0) = ’1’) then
		f2_dly_var(22 downto 0) := f2_dly_var(23 downto 1);
		f2_dly_var(23) := ’0’;
	end if;
	if (e_diff_var(1) = ’1’) then
		f2_dly_var(21 downto 0) := f2_dly_var(23 downto 2);
		f2_dly_var(23 downto 22) := "00";
	end if;
	if (e_diff_var(2) = ’1’) then
		f2_dly_var(19 downto 0) := f2_dly_var(23 downto 4);
		f2_dly_var(23 downto 20) := "0000";
	end if;
	if (e_diff_var(3) = ’1’) then
		f2_dly_var(15 downto 0) := f2_dly_var(23 downto 8);
		f2_dly_var(23 downto 16) := "00000000";
	end if;
	if (e_diff_var(4) = ’1’) then
		f2_dly_var(7 downto 0) := f2_dly_var(23 downto 16);
		f2_dly_var(23 downto 8) := "0000000000000000";
	end if;

	-- Check sign to see if f_result = f1 + f2 or f1 - f2
	if (s1_R1_add = s2_R1_add) then
		S_temp_f_R2_add(31 downto 0) <= f1_R1_add + ("00000000" &
f2_dly_var );
	else
		S_temp_f_R2_add(31 downto 0) <= f1_R1_add - ("00000000" &
f2_dly_var);
	end if;

end process;

P_Stage3_add: process(temp_f_R2_add,e1_R2_add)
variable carry : std_ulogic;
variable msb   : std_ulogic;
variable temp_f_var : std_logic_vector(24 downto 0);
variable shift_val_var : std_logic_vector (4 downto 0);
variable dec_val_var : std_logic_vector (7 downto 0);
begin

	carry := temp_f_R2_add(24);
	msb := temp_f_R2_add(23);
	temp_f_var := temp_f_R2_add;

	if (carry = ’1’) then
		S_f_R3_add <= temp_f_R2_add(23 downto 1);
		S_e_R3_add <= e1_R2_add + "00000001";
	else
		-- Find position of msb
		if (msb = ’1’) then
			shift_val_var := "00000";
		else
			shift_val_var(4) := ( temp_f_var(0) or temp_f_var(1)
or temp_f_var(2) or temp_f_var(3) or temp_f_var(4) or
					temp_f_var(5) or temp_f_var(6)
or temp_f_var(7) ) and not(temp_f_var(8) or temp_f_var(9)
```

```
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21) or temp_f_var(22) ) ;

                  shift_val_var(3) := ( temp_f_var(8) or temp_f_var(9)
or temp_f_var(10) or temp_f_var(11) or temp_f_var(12)
                                   or temp_f_var(13) or temp_f_var(14)
or temp_f_var(15) ) and not(temp_f_var(16) or
                                   temp_f_var(17) or temp_f_var(18) or
temp_f_var(19) or temp_f_var(20) or temp_f_var(21) or
                                   temp_f_var(22) );

                  shift_val_var(2) := ( ( temp_f_var(0) or
temp_f_var(1) or temp_f_var(2) or temp_f_var(3) ) and not(temp_f_var(4)
or
                                        temp_f_var(5) or temp_f_var(6)
or temp_f_var(7) or temp_f_var(8) or temp_f_var(9)
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22) )) or

                                     ((temp_f_var(8) or temp_f_var(9)
or temp_f_var(10) or temp_f_var(11)) and
                                     not(temp_f_var(12) or
temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22) )) or

                                     ((temp_f_var(16) or temp_f_var(17)
or temp_f_var(18) or temp_f_var(19)) and
                                     not( temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22)) );

                  shift_val_var(1) := ( ( temp_f_var(0) or
temp_f_var(1)) and not(temp_f_var(2) or temp_f_var(3) or temp_f_var(4)
or
                                        temp_f_var(5) or temp_f_var(6)
or temp_f_var(7) or temp_f_var(8) or temp_f_var(9)
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22) )) or

                                     ((temp_f_var(4) or temp_f_var(5)
)and not( temp_f_var(6) or temp_f_var(7) or temp_f_var(8) or
temp_f_var(9)
```

```
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22) ))  or


                                 ((temp_f_var(8) or temp_f_var(9) ) and
not(temp_f_var(10) or temp_f_var(11) or temp_f_var(12) or
temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22))) or

                                     ((temp_f_var(12) or
temp_f_var(13)) and not(temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22))) or

                                     ((temp_f_var(16) or
temp_f_var(17)) and not(temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20)  or
temp_f_var(21) or temp_f_var(22))) or

                                     ((temp_f_var(20) or
temp_f_var(21)) and not( temp_f_var(22))) ;

                  shift_val_var(0) := (temp_f_var(0) and
not(temp_f_var(1) or temp_f_var(2) or temp_f_var(3) or temp_f_var(4) or
                                        temp_f_var(5) or temp_f_var(6)
or temp_f_var(7) or temp_f_var(8) or temp_f_var(9)
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22) )) or

                                     ( temp_f_var(2) and not(
temp_f_var(3) or temp_f_var(4) or
                                        temp_f_var(5) or temp_f_var(6)
or temp_f_var(7) or temp_f_var(8) or temp_f_var(9)
                                     or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                     or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                     or temp_f_var(20) or
temp_f_var(21) or temp_f_var(22) )) or

                                     (temp_f_var(4) and
not(temp_f_var(5) or temp_f_var(6) or temp_f_var(7) or temp_f_var(8) or
temp_f_var(9)
```

```
                                    or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22) )) or

                                   (temp_f_var(6) and
not(temp_f_var(7) or temp_f_var(8) or temp_f_var(9)
                                    or temp_f_var(10) or
temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22))) or

                                    (temp_f_var(8) and
not(temp_f_var(9) or temp_f_var(10) or temp_f_var(11) or temp_f_var(12)
or temp_f_var(13) or temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22) )) or

                                    (temp_f_var(10) and
not(temp_f_var(11) or temp_f_var(12) or temp_f_var(13) or
temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22) )) or

                                    (temp_f_var(12) and
not(temp_f_var(13) or temp_f_var(14)
                                    or temp_f_var(15) or
temp_f_var(16) or temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22))) or

                                    (temp_f_var(14) and
not(temp_f_var(15) or temp_f_var(16) or temp_f_var(17) or
temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22))) or

                                    (temp_f_var(16) and
not(temp_f_var(17) or temp_f_var(18) or temp_f_var(19)
                                    or temp_f_var(20) or
temp_f_var(21)  or temp_f_var(22))) or

                                    (temp_f_var(18) and
not(temp_f_var(19) or temp_f_var(20)  or
                                    temp_f_var(21) or temp_f_var(22)))
or

                                    (temp_f_var(20) and
not(temp_f_var(21)  or temp_f_var(22))) or
```

```
                                    ( temp_f_var(22)) ;


      -- Shift f2 left shift_val_var places
            if (shift_val_var(0) = '1') then
                  temp_f_var(23 downto 1) := temp_f_var(22 downto 0);
                  temp_f_var(0) := '0';
            end if;
            if (shift_val_var(1) = '1') then
                  temp_f_var(23 downto 2) := temp_f_var(21 downto 0);
                  temp_f_var(1 downto 0) := "00";
            end if;
            if (shift_val_var(2) = '1') then
                  temp_f_var(23 downto 4) := temp_f_var(19 downto 0);
                  temp_f_var(3 downto 0) := "0000";
            end if;
            if (shift_val_var(3) = '1') then
                  temp_f_var(23 downto 8) := temp_f_var(15 downto 0);
                  temp_f_var(7 downto 0) := "00000000";
            end if;
            if (shift_val_var(4) = '1') then
                  temp_f_var(23 downto 16) := temp_f_var(7 downto 0);
                  temp_f_var(15 downto 0) := "0000000000000000";
            end if;


            end if;
            s_f_R3_add <= temp_f_var(22 downto 0);

            -- If adding a number plus it's negative the answer should
be zero
            if ( (temp_f_var = zero_25) and (e1_R2_add = e2_R2_add) and
                 (s1_R2_add = not(s2_R2_add)) ) then
                  s_e_R3_add <= zero_8;
            else
                  s_e_R3_add <= e1_R2_add - ("000" & shift_val_var);
            end if;

            s_s_R3_add <= s1_R2_add;


      end if;
end process;


      ----------------------------------------------------------------------
----------
      --  "Inactive" output port signal assignments
      ----------------------------------------------------------------------
----------
      PE_MemHoldReq_n <= '1';                 -- Disable memory hold
requests

      PE_Left_OE <= ( others => '0' );        -- Disable left port
output
```

```
      PE_Right_OE <= ( others => ’0’ );       -- Disable right port
output

      PE_XbarData_OE <= ( others => ’0’ );    -- Disable crossbar port
output
      PE_XbarData_WE <= ( others => ’0’ );    -- Disable crossbar port
writes

      PE_FifoSelect <= "00";                  -- Deselect fifo
                                              --   "00" selects none
                                              --   "01" selects
External I/O Fifo
                                              --   "10" selects On-
Board Fifo
                                              --   "11" selects On-
Board Mailbox
      PE_Fifo_WE_n <= ’1’;                    -- Disable fifo write
mode
      PE_FifoPtrIncr_EN <= ’0’;               -- Disable fifo pointer
increment

      PE_CPE_Bus_OE <= ( others => ’0’ );     -- Disable CPE bus output

end loop_alg;
```

# VITA

Karthik Ramachandran was born on March 19, 1974 in Madras, India. After his undergraduate studies in Instrumentation and Chemistry at Birla Institute of Technology and Science, Pilani, India, he joined Virginia Tech for graduate studies in Fall 1996. There, he completed a Masters program in Electrical Engineering with an emphasis on Computer Engineering. His interests lie in the field of custom computing machines and VLSI design.

In August 1998, he plans to join the Memory Test Division at Teradyne, California, as a Hardware Design Engineer.