

Accelerating Double Precision Sparse Matrix Vector Multiplication on FPGAs

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Technology

and

Master of Technology

by

Sumedh Attarde
(Roll No. 05D07012)

Under the guidance of
Prof Sachin Patkar



DEPARTMENT OF ELECTRONICS AND ELECTRICAL
ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY–BOMBAY

June, 2010

Dissertation Approval

The dissertation entitled

Accelerating Double Precision Sparse Matrix Vector Multiplication on FPGAs

by

Sumedh Attarde
(Roll No. 05D07012)

is approved for the degree of


Bachelor of Technology

and

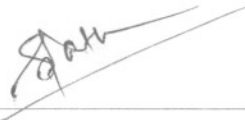
Master of Technology



Examiner



Examiner



Guide



Chairman

Date: 29 - JUN - 2010

Place: IIT BOMBAY

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: 1-JULY-2010

Sumedh
Sumedh Attarde
05D07012

Abstract

Double precision sparse matrix vector multiplication (SpMxV) is a key computational kernel in iterative solvers required in many scientific and engineering applications. General purpose processors achieve only a fraction of their peak performance in sparse matrix vector product operations owing to poor cache behaviour resulting from irregular memory accesses. FPGAs offer more bandwidth and fine-grained parallelism, a combination favourable for acceleration of SpMxV.

In this report, a design of a processing element performing SpMxV on FPGA is described and evaluated. The design is developed with a view to accelerate especially large matrices which have to be stored in off-chip DRAMs. The design improves peak performance by blocking the matrix and pre-fetching data in bursts to hide large DRAM random access latencies. The sparse matrix is decomposed into a relatively ‘dense’ component (still processed as a sparse matrix) and a sparse component. The relatively ‘dense’ component needs to have enough non-zeros to hide substantial portion ($\geq 50\%$) of vector access latencies. A complete description of the input data format along with the required matrix data pre-processing is presented. An optimal scheduling scheme for workload allocation amongst the processing elements without causing RAW hazards is also discussed. The design is targeted for implementation on high-end FPGAs like Xilinx Virtex-5 LX330T which can support a maximum number of five processing elements. The processing element has been prototyped as an accelerator peripheral for the iterative Gauss-Jacobi algorithm in an embedded system on the available Xilinx XUPV5-LX110T development board.

Contents

Abstract	iii
List of Tables	vii
List of Figures	viii
1 Introduction	1
1.1 Organization of the report	3
2 Analysis of SpMxV	4
2.1 Problem Definition	4
2.2 Problem Analysis	4
2.3 Related Work	6
3 Design and Simulation	8
3.1 Overview	8
3.1.1 Matrix Blocking	8
3.1.2 Dataflow	10
3.1.3 Bandwidth Allocation	12
3.2 Data Representation and Storage	13
3.3 Design	15
3.3.1 Local Vector Storage	15
3.3.2 Floating Point Multiplier	18
3.3.3 Isolation Queue	20
3.3.4 Floating Point Adder	20
3.3.5 Partial Sum Storage	21
3.3.6 Pipeline Control	24

3.3.7	Reduction Circuit	27
3.3.8	Xilinx ISE Flow Results	28
3.4	Simulation	29
3.4.1	Functional Simulation	29
3.5	Optimizations	30
4	Application: Iterative Solver	32
4.1	Overview	32
4.2	Building the system	33
4.2.1	Plan A - Using Fast Simplex Link (FSL) Bus	33
4.2.2	Plan B - Using Peripheral Local Bus (PLB)	34
4.3	Adding Processing Element to the System	34
4.4	Designing Interfaces for the Processing Element	36
4.4.1	Co-ordinating with software and hardware	37
4.4.2	Vector Interface Control	38
4.4.3	Matrix Interface Control	38
4.4.4	Partial Sum Interface Control	40
4.5	Software Development	41
4.5.1	Hardware Management	41
4.5.2	Software Application - Gauss Jacobi	43
4.6	Evaluation	44
4.7	Design Flow	46
5	Matrix Pre-Processing	49
5.1	Why Pre-Processing?	49
5.2	Optimal Scheduling Problem	50
5.2.1	Formalization	50
5.2.2	Algorithm	50
5.2.3	Proof of Optimality	53
5.3	Matrix ‘Dense’ Block Identification	54
5.3.1	Formalization	55
5.3.2	Algorithm	55
5.4	Software Implementation	56

6 Conclusion	57
6.1 Future Work	57

List of Tables

3.1	LX330T Device Utilization Summary	28
3.2	Performance Evaluation of Design	29
4.1	Performance comparison for Gauss-Jacobi solver	45
4.2	Accelerator System DMA Transfer Time	45
5.1	Results for Matrix Pre-processing	56

List of Figures

3.1	Matrix Blocking	9
3.2	SpMxV Operation	11
3.3	Data representation	13
3.4	Vector Storage in DDR DRAM memories	14
3.5	Block Diagram of processing element	15
3.6	Vector Storage	16
3.7	Design A: Vector Fetch Control Block	18
3.8	Design B: Vector Fetch Control FSM	19
3.9	Partial Sum Storage	22
3.10	Design A: FSM for Partial Sum Transfer Control	24
3.11	Design B: Partial Sum Transfer Control	25
4.1	Fast Simplex Link (FSL) Bus Block Diagram	34
4.2	System Block Diagram	35
4.3	State machine to co-ordinate with software	37
4.4	FSM for Vector interface for PE	39
4.5	FSM for Matrix interface for PE	40
4.6	FSM for Partial Sum interface for PE	41
4.7	Application Software Flowchart	44
5.1	Occurrence of hazards in adder pipeline	50
5.2	Scheduling in a block:Initial phase	51
5.3	Scheduling in a block:Continuation	52
5.4	Case for k sequences: Distribution of data	53

Chapter 1

Introduction

Sparse Matrix Vector Multiplication (SpMxV) is a key computational kernel in many scientific and engineering applications. Least square problems, eigenvalue problems, FEM, computational fluid dynamics, image reconstruction in medical imaging, circuit analysis, web-connectivity and many more applications need to solve sparse linear systems using iterative methods. It is observed that SpMxV forms the bottleneck in these iterative methods, resulting in only a fraction of the peak performance of the processing cores being utilized.

For SpMxV, unless specific classes of matrices are targeted - like matrices arising from FEM, circuit analysis - distribution of elements from the matrix (A) is unknown and order of access is irregular. As a result there can be no planned reuse of vector elements, unlike in the case of general matrix-matrix multiplication. This calls for higher memory bandwidth requirement. The problem is compounded since fetching of pointers describing location of non-zeros in the matrix also takes up a portion of the memory bandwidth.

Large matrices which have sizes running into millions of rows cannot fit in the caches of general purpose processors. Since memory accesses are irregular, cache hierarchy loses its effectiveness and many random accesses to high latency DRAMs are made. This causes performance of SpMxV to drop to a fraction of peak performance for general purpose processors. It is only by optimization and tuning of SpMxV kernels for the specific matrix structures that these shortcomings can be overcome[1].

Modern FPGAs provide abundant resources for floating point computations. Aside from large logic capabilities, these FPGAs also have sufficient on-chip single-cycle access

blocks of RAM (BRAMs) to provide required on-chip memory bandwidth. On the other hand, if external off-chip memories are to be used, then the high I/O pin count of FPGAs helps to achieve high memory bandwidth and hide access latencies. However, off-chip memories like DRAMs have large access latencies and can considerably slow down the system if used naively.

FPGAs have been used previously to implement SpMxV. The major shortcoming of these implementations is that they use on-chip BRAM or off-chip SRAM to store data. As a result they deal with much reduced problem sizes (few tens of thousand rows) as compared to what are observed in practical applications. In applications like web-connectivity, computational fluid dynamics matrix sizes can run into millions of rows. On-chip memories or SRAMs are simply insufficient to store such large amounts of data. The memory densities required for such problems can be provided only by DRAMs.

We present the analysis and design for an FPGA based architecture that performs double precision floating point Sparse Matrix-Vector Multiplication utilising commodity DRAMs for storage. In typical applications, majority of non-zeroes tend to cluster in relatively dense blocks. The rest of non-zeros scattered sparsely throughout the rest of the matrix form a small percentage of total number of non-zeros. Hence, the problem matrix is decomposed into two component matrices - one component comprising non-zeros in the relatively ‘dense’ regions of the matrix and the other component containing remaining elements. This design aims to accelerate computation over the relatively ‘dense’ regions. This is achieved by arranging data in a burst-friendly manner in DRAMs to hide the access latencies. We analyse the problem from an architectural standpoint, discuss design trade-offs, present the limits on the expected performance and evaluate our design. The design has been targeted at the Virtex-5 LX330T, a high end FPGA from Xilinx device families. By using commodity DRAMs, our design is capable of processing matrices which have multi-million non-zero elements. For most matrices, performance is observed to be very close to the peak performance. Off-the shelf IEEE-754 compliant floating point multipliers and adders are used in the processing cores of our design, ensuring portability to newer device families.

An embedded system comprising of one instance of the processing element was implemented for an application involving iterative matrix vector product - Gauss-Jacobi solver - as a proof of concept. Performance figures for this system were collected and

explained keeping in perspective limited capabilities of an embedded system.

1.1 Organization of the report

The aim of the project is to implement an architecture performing sparse matrix vector multiplication for sparse matrices on the FPGA. The second chapter provides an analysis of the challenges and trade-offs involved in design and also provides a background of existing work.

The third chapter comprises an overview of the computational approach followed by the architecture and then describes two variants of the developed architecture. Finally the chapter provides software implementation flow details and simulation results. The fourth chapter gives a brief overview of the hardware, software (iterative Gauss-Jacobi solver) and hardware software co-design involved in the development of the embedded system built around the SpMxV processing core. The fifth chapter finally discusses the problem of generation of the optimal schedule for SpMxV computation and proposes an algorithm for the same, while providing a proof of optimality.

Chapter 2

Analysis of SpMxV

2.1 Problem Definition

SpMxV is defined as follows: Consider a matrix vector product $y = Ax$. A is a $n \times n$ sparse matrix with N_z non-zero elements. We use A_{ij} ($i = 0, \dots, n - 1; j = 0, \dots, n - 1$) to denote the elements of A . x and y are vectors of length n and their elements are denoted by x_j ($j = 0, \dots, n - 1$) and y_i ($i = 0, \dots, n - 1$). Thus for every A_{ij} , two operations need to be performed - floating point multiplication followed by floating point addition.

$$y_i = y_i + A_{ij} \times x_j \tag{2.1}$$

2.2 Problem Analysis

We observe that x_j is determined only after A_{ij} is fetched along with the corresponding row and column pointers from the memory into the processing element. There is no sure way or predicting j before the nonzero element is fetched. Thus SpMxV suffers from irregular memory accesses, which can slow down the system considerably if high latency memories like DRAMs are used.

If we call the composite operation represented by 2.1 as multiply-accumulate, one such operation requires two double precision (IEEE-754) words and the implicit pointers represented here by i and j subscripts. Unlike the case of dense matrix computations, there can be no planned reuse of the vector entries without knowledge of the matrix structure. The elements are thus used only once after which they are discarded. Only the

result of the multiply-accumulate operation is stored. Since two input words are useful for only two computation operations, the ratio of computation to bandwidth requirement is low compared to other applications (namely general matrix-matrix multiplication). This ratio becomes worse due to overhead of bandwidth requirement for fetching pointers - two per matrix element. Assuming 32-bit pointers and double precision floating point matrix and vector data, 24 bytes are fetched in order to perform 2 operations. Hence, the performance of SpMxV is usually less than a tenth of the bandwidth available to the system.

A deeper analysis of the problem reveals that there is an inherent trade-off involved in allocation of bandwidth for fetching matrix and vector. There are two bottlenecks that need to be overcome. Firstly, in case of pathologically sparse matrix (unstructured), a random access need to be performed for every x_j . The low latency for random accesses offered by SRAM-like memories makes them the natural choice for storing vector elements. Since we would prefer as many processing elements as possible, these memories should be able to handle multiple requests without stalling. However, these memories would be insufficient to store entire vector when the vector size runs into millions of rows. At the other end of the spectrum, in a properly structured matrix, when the matrix elements form clusters, required vector elements can be cached in relatively small size memories. Here, the bottleneck is fetching enough matrix data from the external memories to ensure maximum reuse of the vector elements.

Though modern FPGAs have large amounts of fast access memories, they still fall short of the amount of storage required in case the matrix and/or vector data is to be stored in on-chip memories. The largest Virtex-5 device has less than 24 Mb of storage and the latest Virtex 6 family too has less than 48 Mb of on-chip memory. Assuming 64-bit data, this translates to 0.4M words and 0.8M words in case of Virtex 5 and Virtex 6 respectively. Moreover, as discussed in the above paragraph, if vector elements need to be replicated, then the size of the matrices that can be handled drops far short of the one million mark. Hence, an implementation geared to handle matrices having multi-million rows has to use external DRAMs for storage.

2.3 Related Work

The SpMxV kernel implemented by Zhuo and Prasanna[2] was among the earliest in the field. They use Compressed Row Storage (CRS) format for their input which trims the zeros from the sparse matrix rows. In their architecture, each trimmed row is divided into sub-rows of fixed length equal to the number of processing elements. The dot products in a sub-row are assigned to different processing elements and then a reduction circuit is used to get the final vector element after all sub-rows have been processed. They perform load balancing by merging appropriate sub-rows and padding them with zeros if necessary, which significantly improves performance. However the architecture proposed by them relies on SRAMs for storage of matrix entries which severely limits the matrix size. Moreover the entire vector is replicated in local storage of all processing elements. The sequential nature of the inputs to the already huge reduction circuit results in very high latencies. The largest matrix evaluated had 21200 rows and 1.5 million non-zeros. They reported an average performance of 350 MFLOPS.

Special care has been taken by Gregg et. al to create a DRAM based solution. They use pre-existing SPAR architecture originally developed for ASIC implementation and hence port a deeply pipelined design for FPGA implementation. They use local BRAMs to create a cache for the DRAM data since they consider elimination of cache misses to be of paramount importance. They reported performance of 128 MFLOPS for three parallel SPAR computation units. In case caching is perfect, they achieve performance of 570 MFLOPS for three SPAR units.

The architecture developed by deLorimier[3] uses local FPGA Block RAMs (BRAMs) exclusively to store matrix as well as vector data. While pre-processing the matrix, they exploit the statically available information to schedule computation and communication periods statically. The relevant micro-code is stored in the local storage of each processing element. To maximize operating frequency, the accumulator used has a deep pipeline which could potentially cause RAW (read after write) hazards due to the inherent feedback in the accumulator datapath. During pre-processing, they re-order the matrix and interleave rows to prevent RAW hazards thus circumventing the need for expensive pipeline stalls which might have had to be enforced. However, the architecture has an forced communication stage which is not overlapped with computation stage, decreasing the overall

efficiency of the system. The main limitation of the system is its inability to handle designs larger than what the BRAMs can accommodate. The largest matrix evaluated had 90449 rows and 1.8 million non-zeros. They reported performance of 1.6 GFLOPS for 1 Virtex-II 6000 and about 750 MFLOPS per FPGA with 16 FPGAs.

An architecture has also been developed by Sun, Peterson et. al[4], for a FPGA-accelerator system. Data is transferred from host memory to off-chip memory on the accelerator board. Their architecture relies on dividing a row into blocks of non-zero sections and feeding them to the processing elements. Vector elements are loaded in blocks as required in local on-chip memory. The design does not perform row-interleaving and hence has to rely on other hardware for preventing RAW hazards. On Virtex-2 devices, operation is estimated to be at 165 MHz for double precision data and performance estimates predict up to 96% peak performance.

Our implementation strives to get the best of deLorimier's architecture while using DRAMs as in case of the architecture proposed by Gregg et. al.

Chapter 3

Design and Simulation

3.1 Overview

As shown in previous sections, SpMxV for large matrices cannot be handled exclusively using on-chip memories. Hence we propose a hybrid system wherein the matrix and vector are stored in external off-chip commodity DRAMs and the data is cached in local on-chip memories. Since burst accesses can help to hide DRAM latencies, the following sections explain a processing paradigm which enables data to be stored in a fashion amenable to burst accesses.

From the discussion in previous sections, we know that there are two issues to be addressed in order to accelerate SpMxV. Practically encountered matrices are not pathologically sparse nor are they perfectly clustered, both properties are present in them to varying degrees. We propose that matrices be split accordingly. Clustered non-zeros are handled by one kernel, and the outliers by a different one. In this paper we design an architecture best suited to process the clustered blocks of non-zeros. Since the outliers are few in number and randomly arranged, they should be handled separately.

3.1.1 Matrix Blocking

The operation involving sparse matrix vector multiplication is distributed amongst multiple processing elements. Since maximum throughput will be obtained when all the processing elements are operating in parallel, the vector access should be as fast and conflict-free as possible for each processing element. One solution is to use multi-port

memories - one port for each processing element, but it is not scalable. The other solution, which we have used, is to use separate caches in each processing element for storing vector elements and replicating the vector in all of them.

The local (single-cycle access) BRAM storage in FPGA is far from sufficient to store multiple instances of a multi-million double precision vector. To tackle this problem the relatively ‘dense’ matrix is divided into square blocks of certain predetermined size. The granularity of division is influenced by amount of memory to be dedicated to on-chip caches and definition of ‘dense’ block. A matrix block size of 128×128 is small enough to allow for multiple on-chip caches and also contains sufficient non-zero elements. The block size could be changed with no fundamental changes to the rest of the design. Since we are using external DRAMs for matrix and data storage, we aim to make maximum use of the burst accesses of DRAM to hide the access latencies. As described in previous paragraph, since matrix elements are stored in the order in which they are to be processed, matrix data can be fetched in continuous stream of bursts.

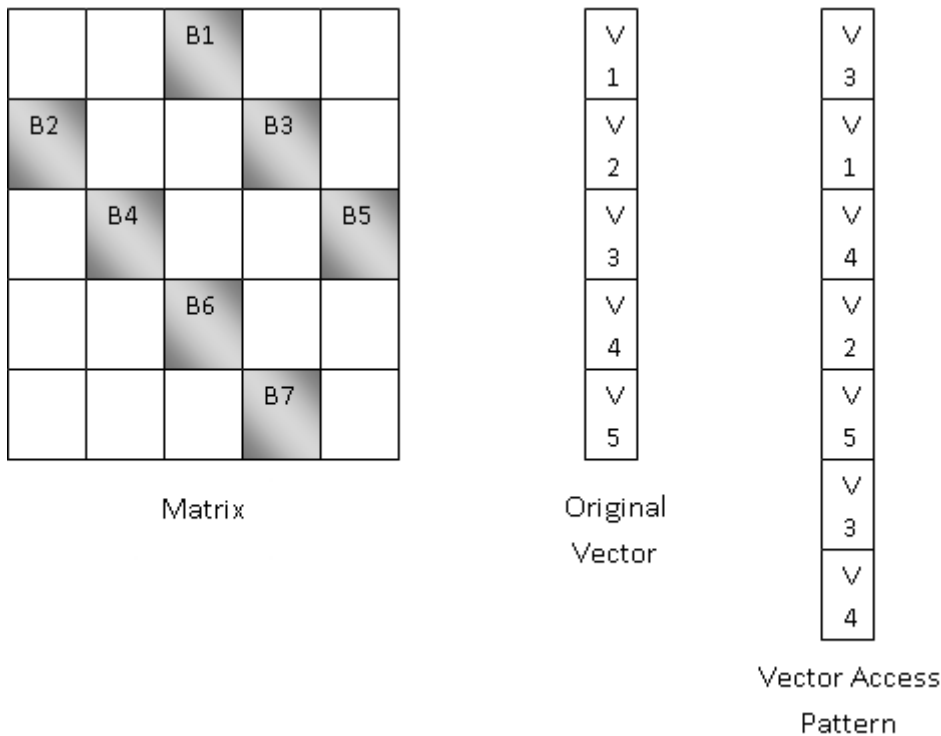


Figure 3.1: Matrix Blocking

Only the relatively ‘dense’ blocks are given as input to the processing elements in a row-major fashion. A rowstrip is defined to be the set of all the blocks ($128n$ row aligned)

having the same set of rows. A direct fallout of such a mechanism is that the vector too can be blocked and thus only elements in required blocks of the vector can be fetched using burst accesses rather than the whole vector being stored. The blocking is described in Figure 3.1. Moreover, we **pre-fetch the next required vector block** to reduce stalls in computation, which is instrumental in hiding the DRAM access latencies. Since the blocking is very fine, the vector block size is small and thus the vector entries can easily be replicated in the local storage of all processing elements, bypassing problems which may arise due to memory contention.

3.1.2 Dataflow

The matrix data is stored arranged in blocked column-row-data format. During operation, relatively ‘dense’ matrix blocks are processed in a row major fashion one after the other. The matrix data in each block is distributed amongst different processing elements in the pre-processing phase. All processing elements operate on the same block at any given time. For each ‘dense’ matrix block, the processing elements compute the product of each matrix element assigned to it and the appropriate vector element using the vector block replicated in each processing element. Each processing element also maintains a 128 element (one for each row in block) partial sum array, which is re-initialized to zeros at the start of processing of a rowstrip. In each processing element, the generated product is accumulated into that element of these partial sum arrays corresponding to the row of the matrix data. These partial sum arrays are transferred out of each processing element at end of processing of a rowstrip, after which they are accumulated in a adder tree. Thus, for each row in the rowstrip, we get the accumulated sum which is the corresponding row element in the result vector. This operation is described in a scaled down example, having two processing elements, in Figure 3.2. In the figure, the variable Y_{ijk} represents that it is a element contributing to the result vector generated by processing element i , for row k of block j of the matrix. The grayed blocks represent ‘dense’ blocks and numbers represent the processing element the matrix element is assigned to.

The processing pattern described above does output the result vector blocks in order but not necessarily in continuation. For instance, if a particular rowstrip in the matrix has no ‘dense’ blocks, there is no input to the processing elements corresponding to that rowstrip. Thus no output partial sum array will be generated for that block of result

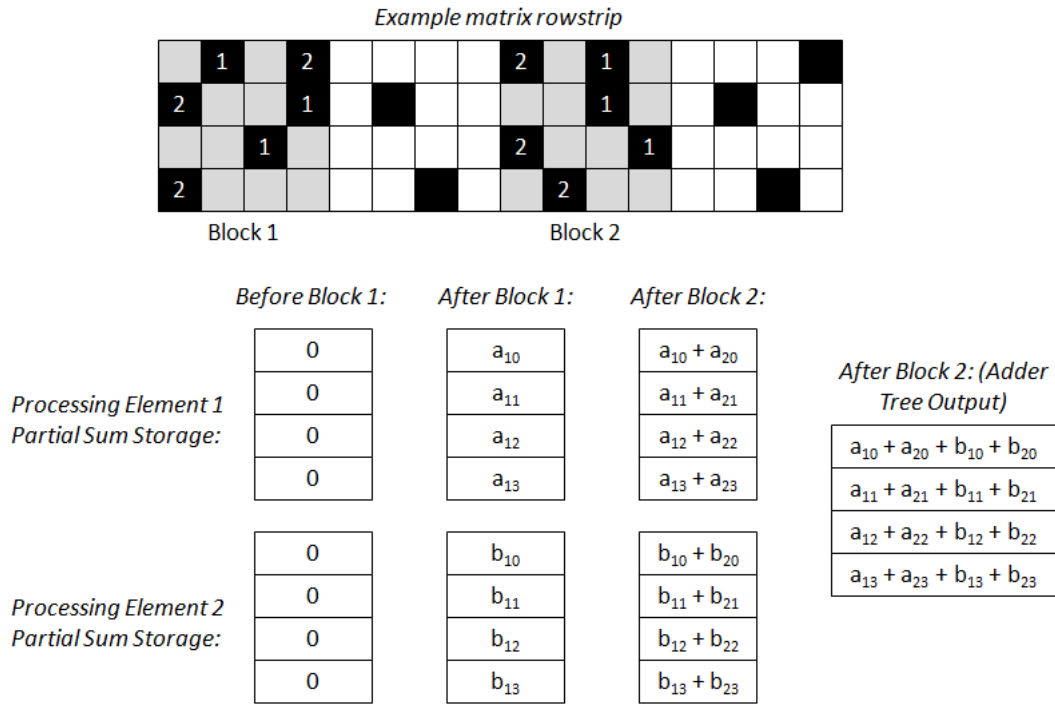


Figure 3.2: SpMxV Operation

vector. This may not seem to be a problem in case matrix-vector product is the only computation to be performed. However this is seldom the case. In most applications, sparse matrix-vector product is used as an intermediate step in an iterative algorithm where the next stage in the application might require knowledge of the entire vector. This would require the design to fill the discontinuities in the result vector with zeros or provide some identification for the output result vector block (for instance, in form of block number). One of the two designs (*Design B*) described in the following sections, provides the complete vector by filling discontinuities with zeros. *Design A* relies on external module to recreate the vector as required by the processing element for the next iteration.

Since we are using external DRAMs for matrix and data storage, we aim to make maximum use of the burst accesses of DRAM to hide the access latencies. As described in previous paragraph, since matrix elements are stored in the order in which they are to be processed, matrix data can be fetched in continuous stream of bursts. Deciding the pattern for storage of vector blocks however requires a bit of thought, which is explained below. Since most applications involving sparse matrix vector product are iterative in nature, the result vector of one iteration will be (after some processing) input vector of

next iteration. If the input vector blocks are stored in the **order of requirement**, then the result vector of this iteration would have to be manipulated to match the format of the input vector, to get the system ready for the next iteration. This may lead to some vector blocks being required to be replicated, while some may have to be dropped. This scenario has the advantage of ease of access but disadvantage of result storage. *Design A* follows the above protocol. On the other hand, if the input vector blocks are stored **in order**, then the result vector needs minimal manipulation. This scenario has advantage of result storage and the disadvantage of random input access. However, the knowledge of the sequence of access of vector blocks is static and can be provided to the vector fetching units at initialization time, reducing the penalty for random block access making the second scenario more feasible than the first for iterative applications. Moreover within a block, the vector data elements are sequentially stored, paving the way for burst accesses. *Design B* follows the above protocol.

3.1.3 Bandwidth Allocation

Since the design processes relatively dense regions of the matrix, more bandwidth has to be allocated for fetching the matrix than for fetching the vector. The most binding constraint restricting the number of processing elements is the IO bandwidth which is just sufficient to support seven DRAM controllers on the largest device LX330T of the Xilinx Virtex-5 family. In the following section, the data representation and storage are explained which determine allocation of bandwidth between matrix and vector and number of processing elements which can be supported.

Fetching a vector block entails fetching 128 64-bit words from DRAMs. The amount of time required to fetch a vector block has a direct impact on the density of the ‘dense’ blocks that can be handled efficiently by the design. This limit can be reduced by allocating more bandwidth for vector fetch however compromising on matrix bandwidth. It was found that putting this limit to around 80 covered significant portions of most matrices, which after simple optimization translates to two memory controllers for fetching vector data. Thus the largest FPGA can support five memory controllers which translates to five processing elements.

3.2 Data Representation and Storage

Since the two designs - *A* and *B* - have slightly differing functionality, their data requirements are also slightly different. First we discuss the similar part of the representation and then discuss the deviations.

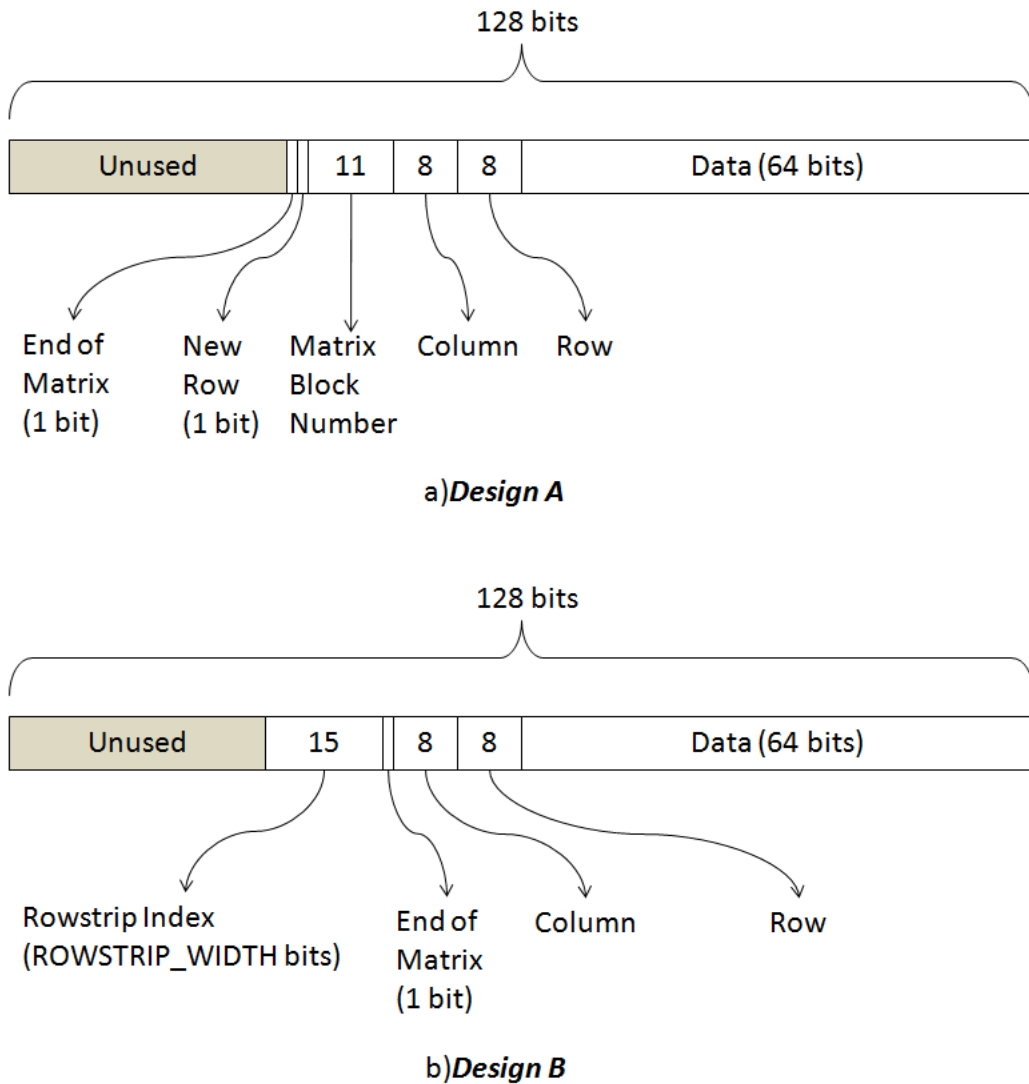


Figure 3.3: Data representation

For both designs, We use the Blocked-Column-Row format to store matrix data in a compressed format. Each matrix element is 128 bits long. The least significant 64 bits represent the matrix data value in double precision. As a result of the matrix and input vector blocking, row and column address need to be provided as offsets from the block boundary only. Thus we need seven bits each to access the proper location in a buffer. However, as two buffers are used in ping-pong fashion for both vector and partial sum

array storage (sections - 3.3.1 and 3.3.5), we need one more bit for each. This bit specifies which of the ping pong buffers is to be utilized. Moreover in both designs we require a bit to signify end of matrix data.

However, in *Design A*, the processing element needs to keep track of the number of ‘dense’ matrix blocks encountered to evaluate if next vector block is required (section 3.3.1). So the matrix data for this design contains an eleven (scalable) bit block number. In *Design B*, this evaluation is performed implicitly by checking the bit which specifies the buffer to be used. This design however does require information about the rowstrip currently in progress, to ensure that result vector is continuous.

Thus for fetching the matrix, we have a memory controller per processing element accessing data stored in an interleaved fashion in two attached 32-bit DRAM banks. Owing to the dual-clock edge operation of DRAMs used, a 128-bit word is accessed per clock cycle in the burst mode. The vector data is stored in an interleaved fashion in each of the pair of DRAMs attached to a memory controller. Thus when accessed in burst, the four dual-clock edged DRAM banks can provide four 64-bit words per clock cycle.

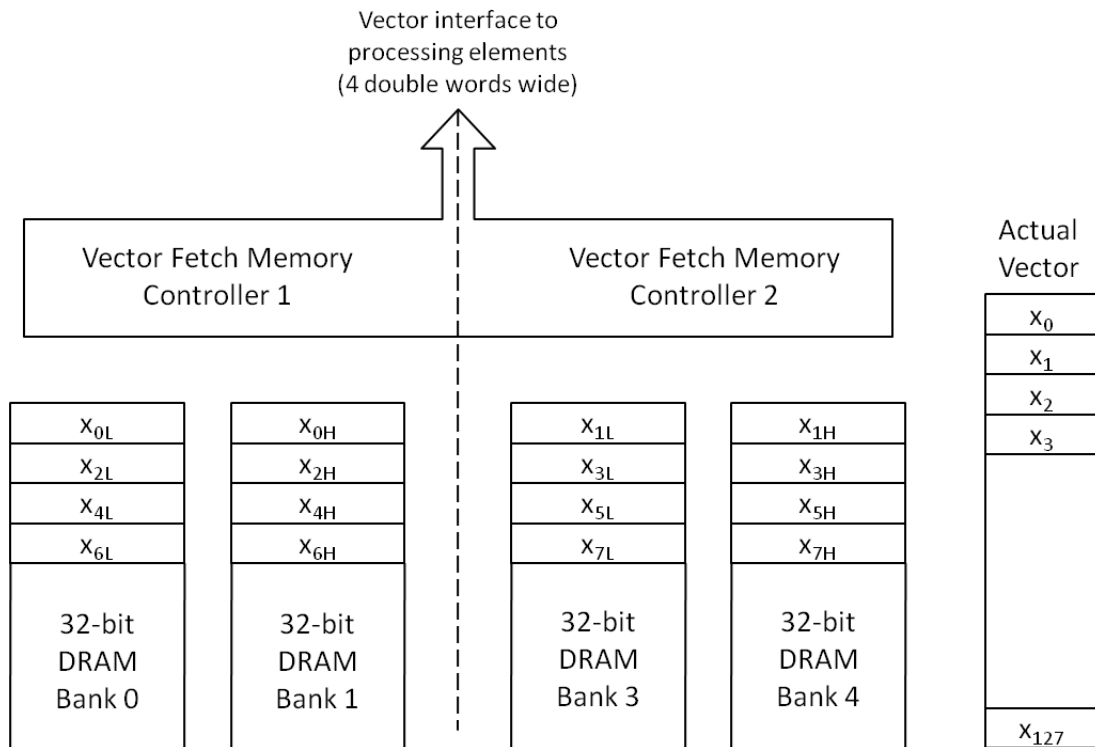


Figure 3.4: Vector Storage in DDR DRAM memories

3.3 Design

A block diagram of the processing element is shown in figure 3.5. Modules in the figure are explained in the following section.

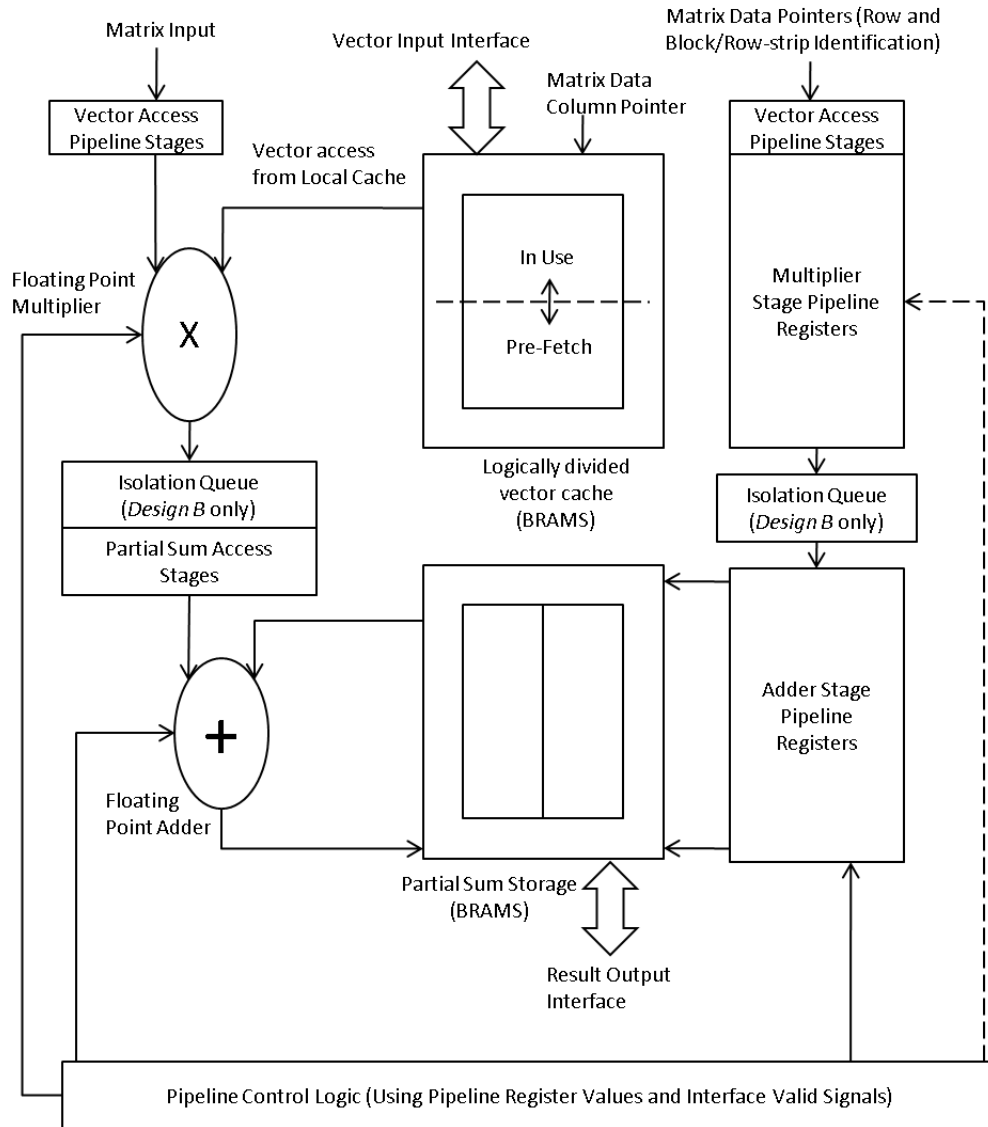


Figure 3.5: Block Diagram of processing element

3.3.1 Local Vector Storage

The vector blocks which are fetched from the DRAMs in burst accesses are replicated in the local vector storage of each processing element. FPGAs have single-cycle Block RAMs (BRAMs) which are used for implementing these memories. Since the matrix is divided into block sizes of 128, we need storage for 128 elements of the block currently

being processed as well as the 128 pre-fetched vector elements corresponding to the next matrix block. By using simple dual port memories, while vector elements are being read from one half of the storage by the processing elements, vector elements corresponding to the next block are pre-fetched in the other half. Thus we effectively utilize ping-pong memories to hide access latencies.

We would like to minimize the number of clock cycles required to replicate the vector elements in all the processing elements. More importantly, this would also ensure that the probability of vector blocks being pre-fetched in time is higher. This was the main motivation for using two DDR DRAM controllers each connected to a DRAM, so that we would get 4 words per clock cycle. Thus one vector block is fetched in 32 clock cycles. Hence, the local storage is organized as a structure 4 double words wide and 64 deep. The vector element address or alternately the *matrix column index* is split into row address and column address internally as shown in the Figure 3.6.

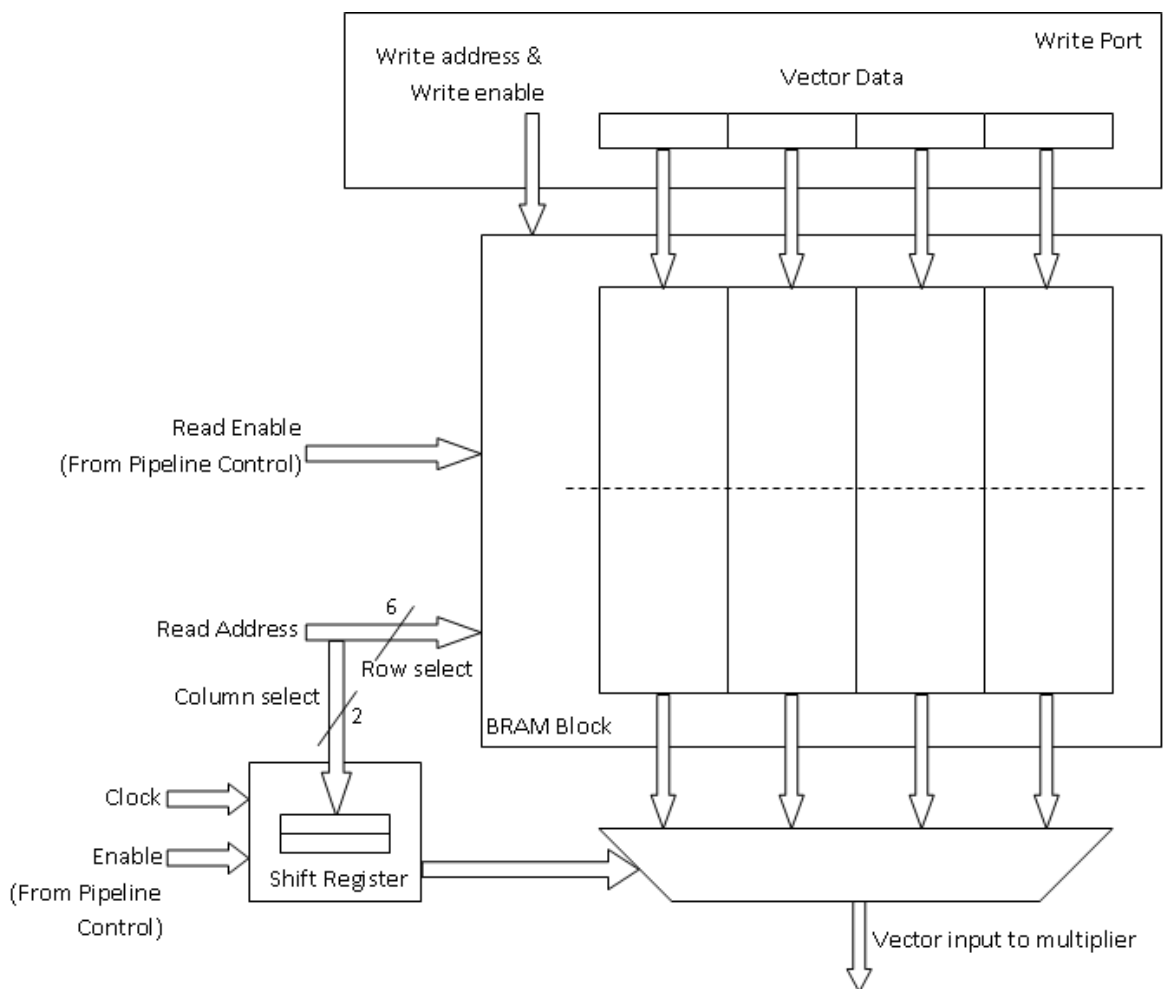


Figure 3.6: Vector Storage

The outputs of the BRAM primitives used for constructing the vector cache are registered. As a result data can be read only at the end of the second clock cycle after the cycle in which read enable is pulled high. This calls for use of a shift register for the column select part of the read address to ensure that the select signal of the multiplexer shown in Figure 3.6 is available at the right time.

Say we have n processing elements. Thus to ensure that the processing elements do not have to wait for the vector-fetch stage to complete, the previous block should have at least $32 \times n$ non-zeros to ensure the communication and computation completely overlap. In case $n = 5$, it imposes a requirement of 160 non-zeros per block of size 128×128 , which can be easily met in relatively dense regions of most sparse matrices. The impact of this value will be discussed in more detail in the section on **Matrix Pre-Processing**.

The interface between the vector fetching memory controllers and the processing element only comprises of valid ($xvalid$), acknowledge ($xdata_ack$), end of block ($xeod$) and the four 64-bit wide vector data input ($xin1, xin2, xin3, xin4$) signals. The address for writing the vector data to the right location in the memory has to be generated internally and the two designs perform this slightly differently.

3.3.1.1 Design A

In this design, a sub-module in this block (*state_machine_shift_load*) houses control logic for the write address generation, vector request generation and handshaking. This is achieved by keeping track of the number of vector blocks brought into the processing element as well as the number of ‘dense’ blocks processed - for which we need the block number explained in section 3.2. By comparing these two values, a decision is made whether or not to accept new vector blocks. This comparison also helps to evaluate whether the matrix input needs to be halted - signal *astop*. A Block diagram of the sub-module is shown in Figure 3.7.

3.3.1.2 Design B

In this design, the module *vector_cache* houses a state machine depicted below in Figure 3.8. This state machine generates the signals which control write address generation, vector request generation and vector input ready indicator (signal *vector_blk_rdy*). The signal is high when FSM is in state *scnt_1* or *scnt_2*, and low otherwise. This implementation

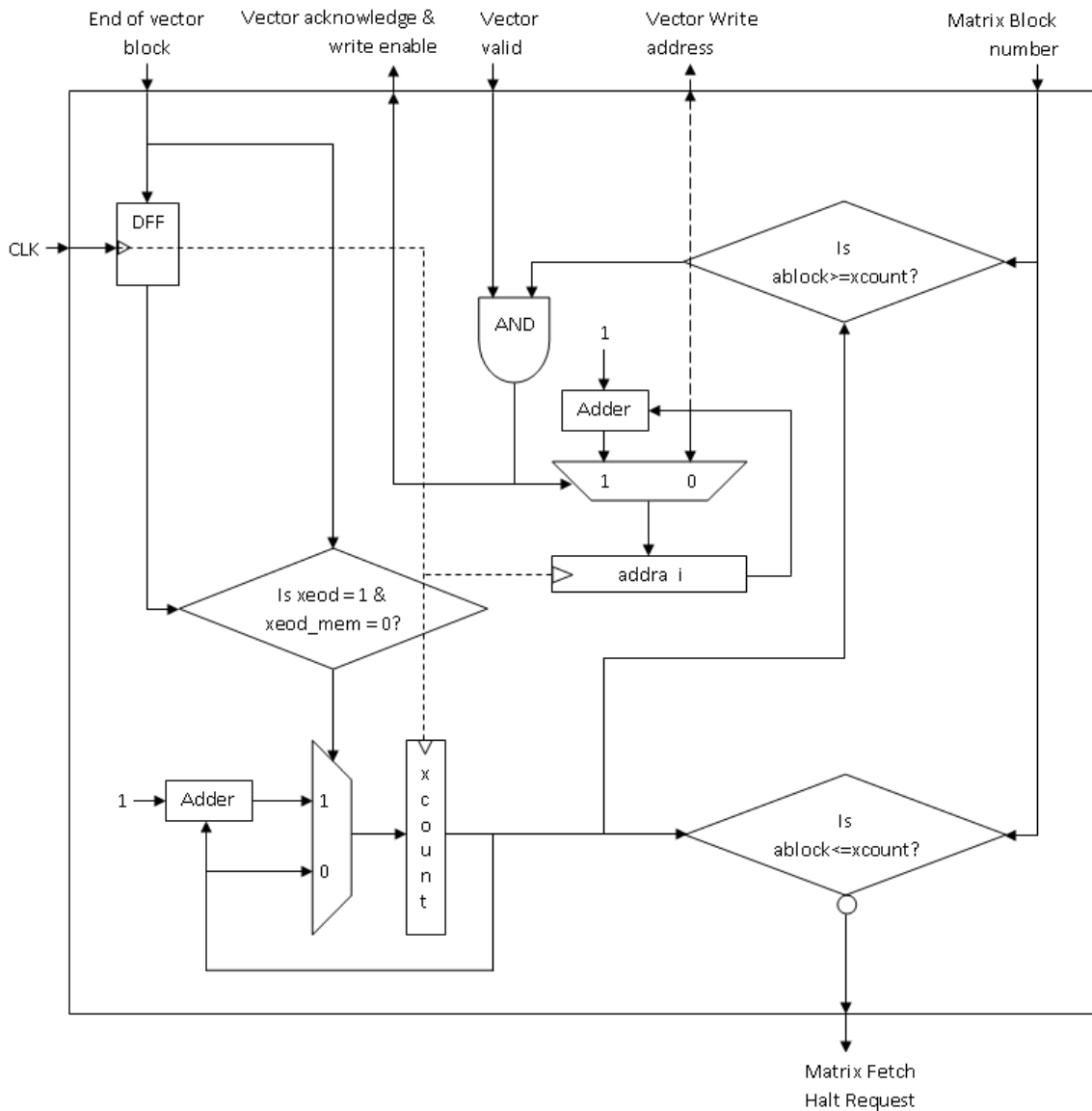


Figure 3.7: Design A: Vector Fetch Control Block

thus does not require any information about the block number of the matrix element. As a result, no comparators are required unlike in *Design A* making this part of the design immune to any increase in delay due to increase in matrix size (which would increase number of bits in *ablock* and *xcount* in *Design A*).

3.3.2 Floating Point Multiplier

A fully IEEE-754 compliant double precision floating point multiplier, generated using Xilinx CoreGenerator was used in the design. The multiplier is deeply pipelined for high throughput. At the input interface of the multiplier, a data valid signal implicitly

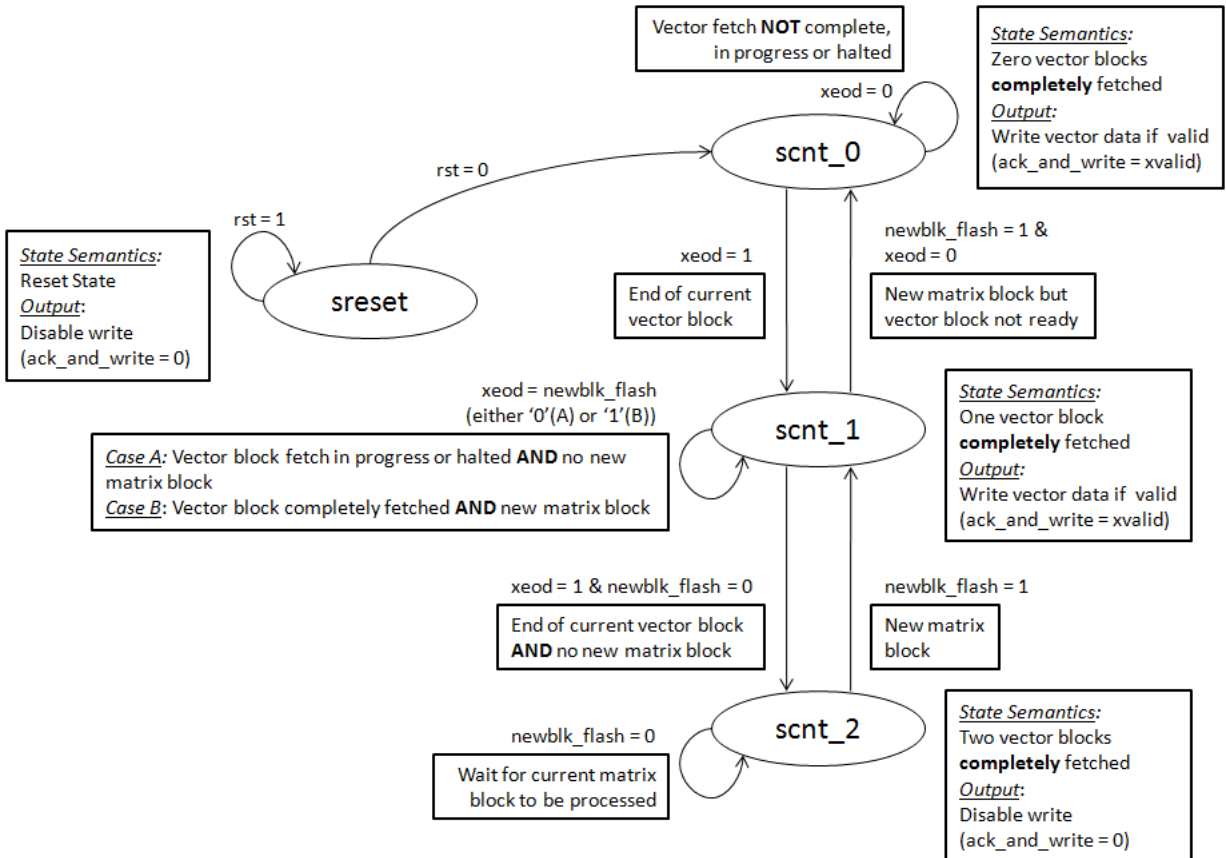


Figure 3.8: Design B: Vector Fetch Control FSM

signifying that both matrix and vector data is ready for multiplication, is provided as *operation_nd*. A synchronous clear (*sclr*), is included to reset the control path of multiplier before start of operation. The multiplier ready indicator on the input interface (*operation_rfd*), is used as one of the inputs in pipeline control. On the output interface, we have only the result and the result valid signal.

Xilinx IPs have a 100 ns initialization period. To account for this, (and for robustness) a counter counting once up to 33 is included which is activated after reset. Multiplier can receive input data valid indication only after the count reaches 33. Since the operating frequency will in all likelihood be less than 330 MHz, this delay will be more than sufficient to cover 100 ns.

In *Design A*, the multiplier (Coregen IP v4.0) is 10-stage pipelined and requires 13 DSP48E slices. Moreover, the clock enable of the multiplier is controlled by the pipeline control block. This however causes the enable signal to be heavily loaded, affecting timing performance. In *Design B*, the multiplier (Coregen IP v5.0) is 15-stage pipelined and requires 11 DSP48E slices. The clock enable of the multiplier is forced to high, and

a different pipeline control mechanism is implemented (section 3.3.6).

3.3.3 Isolation Queue

The isolation queue is used to buffer the output of the multiplier along with some associated meta-data, which together will serve as input to the adder. The queue is required to hold the results of the computation already in the vector access (2 stages deep) and multiplier pipeline, in case the adder is not ready to receive them. This gives us an upper bound on the depth of the queue required - two more than the multiplier latency.

In *Design B*, the modules after the multiplier require the product (64 bits), row offset (8 bits), matrix end indicator (1 bit) and the rowstrip index (configurable, parameterized as ROWSTRIP_WIDTH). Hence the width of the queue has to be $(73 + \text{ROWSTRIP_WIDTH})$. The queue is implemented as two queues having a common control. One queue - data queue - is 64 bits wide, while the other queue is $(9 + \text{ROWSTRIP_WIDTH})$ bits wide (section 3.5).

Design A stalls the entire pipeline - vector access, multiplier and adder - in case adder or partial sum cache is not ready. Hence, no buffering of results is required. In *Design B*, pipeline is not stalled, just null elements are inserted (3.3.6) in the pipeline. As a result, this module is required only in *Design B* and not in *Design A*.

3.3.4 Floating Point Adder

A fully IEEE-754 compliant double precision floating point adder, generated using Xilinx CoreGenerator was used in the design. The adder is pipelined for high throughput. However, since the adder functions as an accumulator, there is a loop from adder output to adder input through local partial sum memory. This has the potential to cause RAW hazards, since the adder might try to read from an address whose value is still in the pipeline and this not been updated. Hence we would not want the pipeline to be very deep, as this would compromise on the efficiency of operation (section 5) and offset the advantage offered by deeper pipelining. Similar to the multiplier, the adder also needs a 100 ns initialization period. The same counter (counting to 33) used for the multiplier is used for the adder.

At the input interface of the adder, a data valid signal implicitly signifying that both

product and partial sum data is ready for accumulation, is provided as *operation_nd*. A synchronous clear (*sclr*), is included to reset the control path of adder before start of operation. The adder ready indicator on the input interface (*operation_rfd*), is used as one of the inputs in pipeline control. On the output interface, we have only the result and the result valid signal.

In *Design A*, the adder (Coregen IP v4.0) is 14-stage pipelined and requires 3 DSP48E slices. Moreover, the clock enable of the adder is controlled by the pipeline control block. This however causes the enable signal to be heavily loaded, affecting timing performance. In *Design B*, the adder (Coregen IP v5.0) is 12-stage pipelined and is chosen to be implemented on logic and not DSP48E slices. The clock enable of the multiplier is forced to high, and a different pipeline control mechanism is implemented (section 3.3.6).

3.3.5 Partial Sum Storage

Since we process the matrix in blocks of 128×128 , we need storage for 128 partial sums in each processing element, each corresponding to one of the 128 rows in the block being processed. Since the accumulation stage involves a pipelined adder, we need one port for reading the existing partial sum corresponding to row in the input stage of the row address pipeline. We need another port for updating the partial sum generated by the adder, corresponding to the row in the final stage of the row address pipeline. Thus we need at least two ports for partial sum storage.

When processing for a row strip is done, the partial sums corresponding to that row strip need to be transferred out of the processing element. This would take as many clock cycles as the depth of the partial sum storage. If we have only one buffer for storing partial sums, the entire processing pipeline will have to halt while this transfer is in progress. To prevent such situations, as in the case of the vector storage, we have a ping pong memory system for storing the partial sums generated by each processing element. This ensures that the accumulator does not have to stall while the partial sums are being read out of the processing element at the end of a row strip. Since each processing element can potentially generate 128 partial sums, the size of the storage buffer is enlarged to 256 elements.

At every stage, care is taken to not consume more clock cycles than necessary, since that may result in the computation pipeline being stalled. To reduce the clock cycles

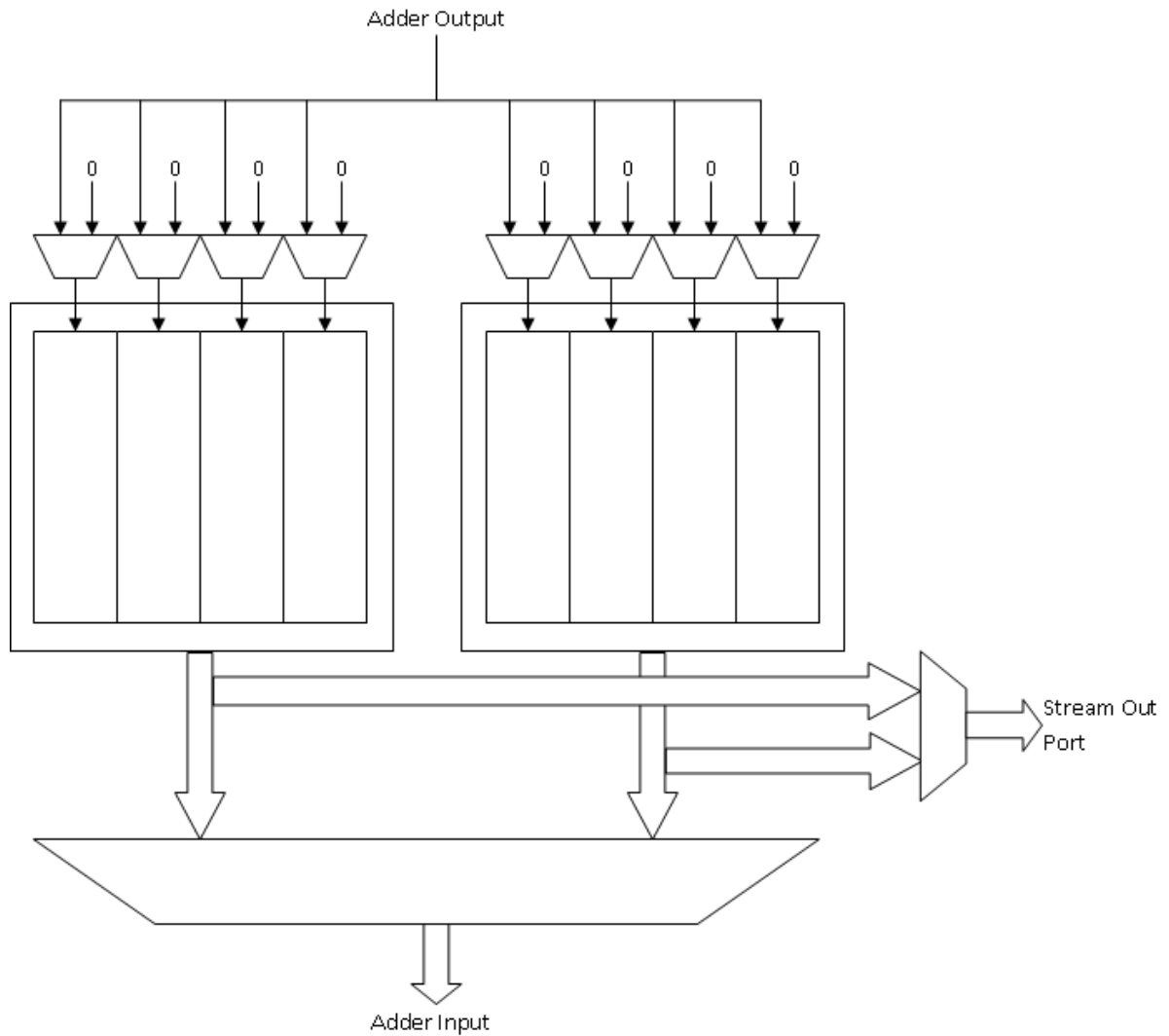


Figure 3.9: Partial Sum Storage

required for transferring partial sums out of the processing element, we use ping-pong buffers to partial sum storage is organized as two buffers each 4 double words wide and 32 words deep. A similar analysis as done for the vector storage yield that a row strip should contain at least $32 \times n$ non-zeros to ensure that computation and IO overlap completely.

As evident from above explanation, the structure of the ping-pong buffer used for storing partial sums is different from the one used for storing vector elements. While the accumulator reads as well as writes to one of the ping-pong memory buffers in the storage, the other buffer is simultaneously being emptied and initialized to zero. Thus during certain periods of operation more than two ports are needed. A simple dual-port memory, as used for vector storage or even a true dual-port memory provides only two ports. As a result, we need two simple dual port memories serving as separate storage

buffers, each of 128 elements as opposed to one big buffer of 256 elements (Figure 3.9).

Owing to the slightly differing functionalities of the two designs, they have different FSMs for controlling the transfer of partial sum arrays out of the processing element.

3.3.5.1 Design A

In *Design A*, the state machine described in figure 3.10 controls the ports of the Block RAMs serving as the partial sum caches. The figure only shows control for the read port. The control for the write port follows exactly the same path except that it is always one step behind. This ensures that the locations in the cache are cleared only after they have been read.

From the state machine, we see that there is no mechanism for identifying the region of the result vector produced. This design relies on external hardware to reproduce the vector in the format required for the next iteration.

3.3.5.2 Design B

This design makes use of the information of the *rowstrip* number to generate the result vector in a continuous fashion. As described in the figure 3.11, when an element of a new rowstrip is encountered, a comparison of the new rowstrip index and old rowstrip index is performed. If the difference is only one, after transmission of the result vector block, the control goes back to idle state unless a new block is required to be transferred immediately. On the other hand, if the difference is more than one, first the result vector block is transmitted after which an appropriate number of zero valued result vector blocks are inserted. As in the case of *Design A*, the state machine also controls address generation and port control of the Block RAMs serving as partial sum caches.

Alternatively, this module could also have been designed to just provide the result vector block along with the *rowstrip* index to the modules performing the writeback to the DRAMs. In this case, if no result block is generated for a portion of the result vector, nothing would be written to the DRAMs, when in fact zeros are supposed to be the result. This would work only if the DRAMs are initialized to zeros before the operation. However, this is rarely done as this operation would require a lot of time. Hence this approach was not used.

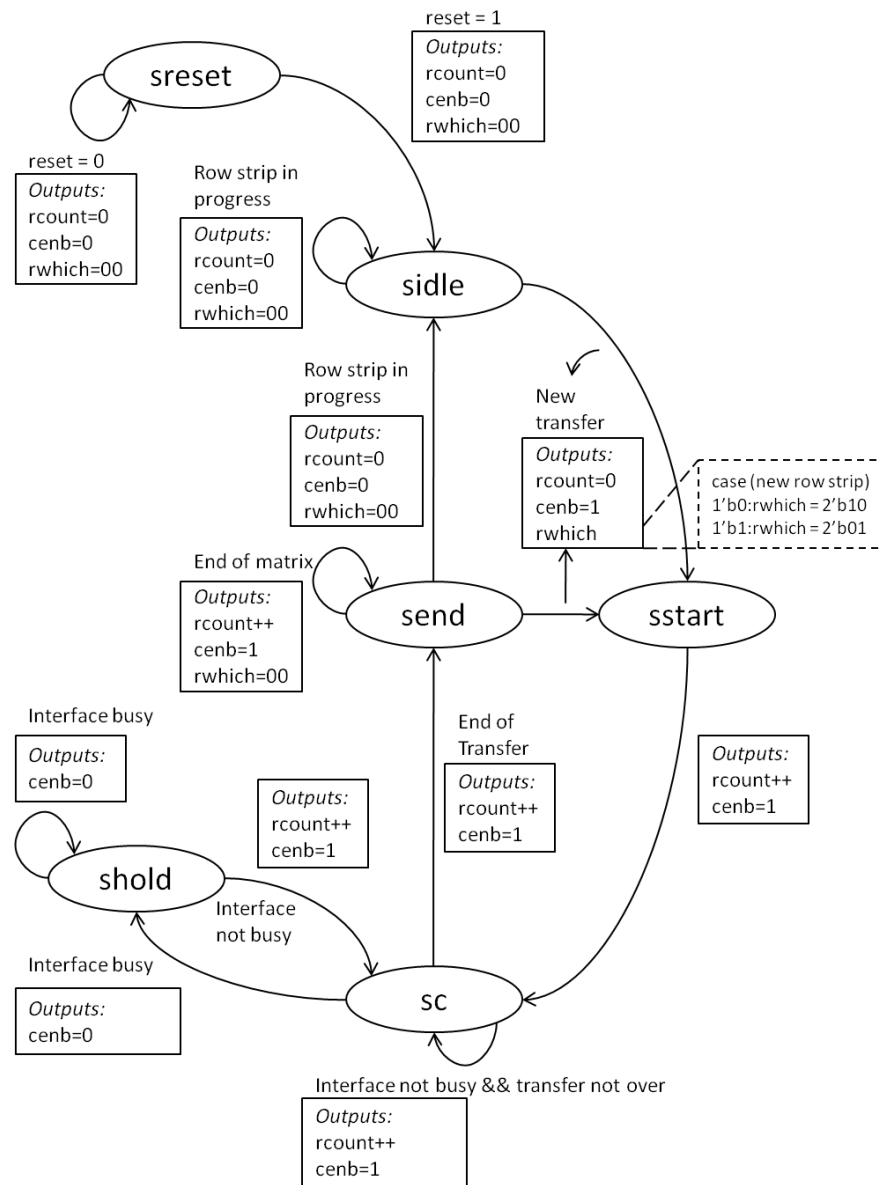


Figure 3.10: Design A:FSM for Partial Sum Transfer Control

3.3.6 Pipeline Control

3.3.6.1 Design A

In *Design A*, we have one whole unified pipeline. The head of the pipeline receives data in the format described in section 3.2. During the following 4 stages, the appropriate vector element is accessed from the local vector cache using the column information. The multiplier pipeline forms the next 10 stages. The next 2 stages in the pipeline take care of the latency of access of partial sum from the local BRAM serving as partial sum array caches. Finally we have the 14-stage adder pipeline.

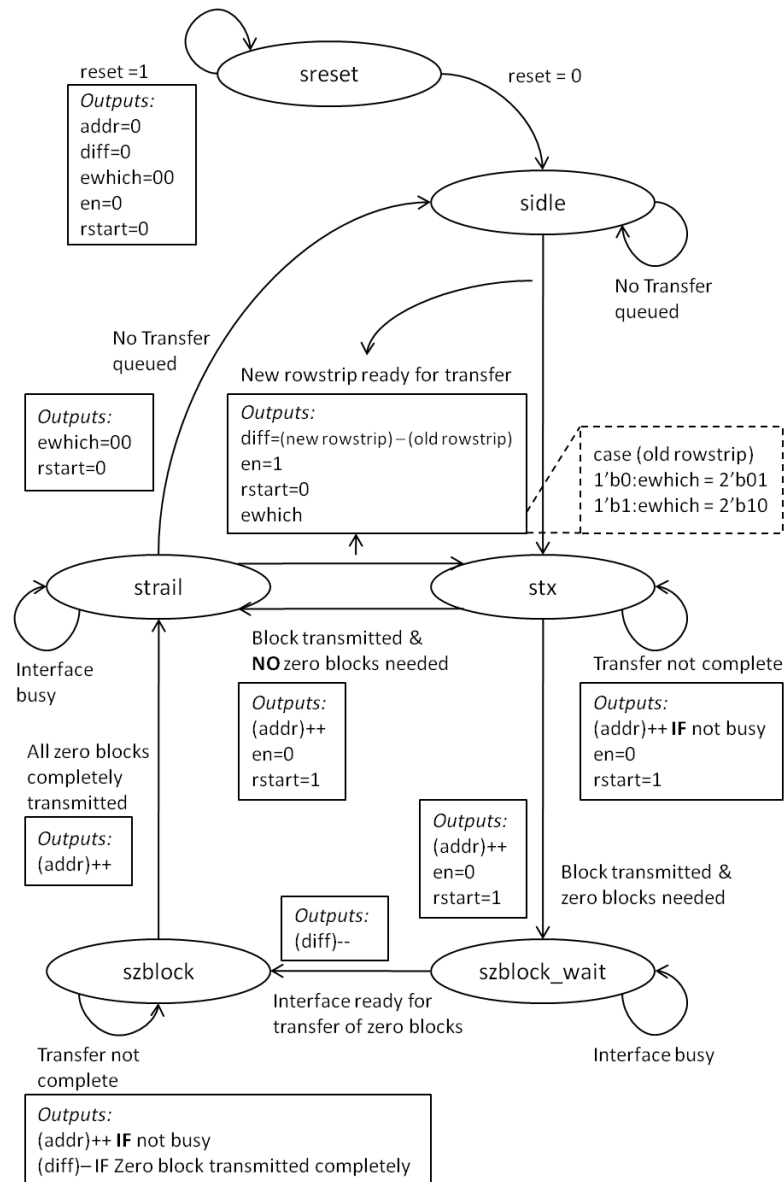


Figure 3.11: **Design B:**Partial Sum Transfer Control

Thus we can see that the entire pipeline needs to be stalled if even one of the following conditions is true:

- Data corresponding to a new matrix block has arrived at input, but the corresponding vector block has not yet been fully fetched. This condition is represented by the *astop* signal being high.
- At the end of the multiplier pipeline, request is made for accessing that partial sum cache which has not yet been emptied fully. This condition is represented by the *clkout_busy* signal being high.

- Matrix data is not valid at input. (*avalid* is low)
- Multiplier or adder are busy - represented by *mult_busy* and *adder_busy* respectively. This would imply that the floating point units are not ready for new data OR their control paths are being synchronously cleared through *sclr*.
- The complete matrix has been processed.

The condition for stalling the pipeline is represented by signal ‘apause’ (active low) as digital ‘OR’ of the above conditions. This signal in conjunction with other signals is used as the ‘clock enable’ for the floating point units. Other pipeline registers decide to retain their value or shift their value to the next stage on the basis of this signal. As a result, this signal was observed to have very high load, which affected routing and thus timing performance.

3.3.6.2 Design B

In *Design B*, we have two pipelines separated by the isolation queue each having separate control. The first pipeline, between the input and the isolation queue contains two stages to match the latency of vector access followed by the multiplier pipeline. The second pipeline, after the isolation queue, contains four stages to match latency of partial sum access followed by the adder pipeline. The latency of the partial sum access is more as compared to that of vector access, because the path from partial sum caches to adder input (comprising of many 64-bit multiplexers) is divided into stages to maximize the throughput.

During operation, any of the following cases can occur with respect to the upper pipeline:

- Data corresponding to a new matrix block has arrived at input, but the corresponding vector block has not yet been fully fetched. This condition is represented by the *astop* signal being high.
- Matrix data is not valid at input. (*avalid* is low)
- Multiplier is busy - either it is not ready for new data or its control path is being synchronously cleared through *sclr*.

- Multiplier has finished computing product for every input matrix data element.
- The isolation queue is full.

In any of these cases, we deassert the valid flag at the head of the pipeline - treating the corresponding elements inserted in the pipeline as null. At the end of the pipeline, the multiplier *rdy* signal helps us identify if the element currently at the output is null or valid. Only if it is valid, it is inserted in the isolation queue. Null elements are discarded. This strategy helps us achieve higher throughput as a pipeline enable signal is not required.

In case of the lower pipeline too, we employ a similar strategy. We insert null elements if any of the following conditions are true:

- At the head of the pipeline, request is made for accessing that partial sum cache which has not yet been emptied fully. This condition is represented by the *clk-out_busy* signal being high.
- The isolation queue is empty.
- Adder is busy - either it is not ready for new data or its control path is being synchronously cleared through *sclr*.
- The matrix has been processed.

The partial sums are updated in the partial sum array only if the corresponding adder output valid signal is high.

3.3.7 Reduction Circuit

Since the non-zeros in a row are distributed amongst the processing elements, partial sums corresponding to the same row may be generated in more than one processing element. It is necessary to reduce them to get the final vector. To achieve this, a reduction circuit ($\log n$ deep) is attached to the outputs of the processing elements. When a row strip is completely processed in all the processing elements, the partial sums clocked out are reduced by the circuit.

3.3.8 Xilinx ISE Flow Results

Both the designs have been synthesized and implemented using Xilinx ISE 11.4 for the Xilinx Virtex 5 device LX330T (package FF1738) of speed grade -2. Table 3.1 below show the device utilization summaries for both the designs.

Table 3.1: LX330T Device Utilization Summary

	Available	Design A		Design B	
		<i>Used</i>	<i>Utilization</i>	<i>Used</i>	<i>Utilization</i>
Number of Occupied Slices	51840	1049	2%	1101	2%
Number of 36k Block RAMs	324	12	3%	12	3%
Number of DSP48Es	192	16	8%	11	5%

The tool provides choices of different strategies for design implementation - timing performance, area optimized design or a balanced approach. For both the designs, the area utilization is not a major concern, but throughput is. Moreover, a complete system will have certain other modules in conjunction with a processing element, namely memory controllers. Thus none of the processing element IOs need to be packed into the IO registers implying no constraints on IO timing. Hence to extract maximum possible speed, “Timing Performance without IOB packing” strategy is selected in ISE for synthesis as well as map operations. To ensure that the design is not bound to any device, minimal timing constraints and no placement constraints were used.

Only a timing constraint of 300 MHz was applied on the period of the system clock. Static timing analysis of both the designs after Place-And-Route revealed that *Design A* can operate at 175.4 MHz while *Design B* can operate at \sim 302 MHz. Moreover, for *Design B*, critical paths involve read/write operation of the BlockRAM resources used for partial sum storage. These critical paths may be broken at the cost of increased accumulator latency which will decrease throughput by imposing stricter conditions to prevent RAW hazards.

3.4 Simulation

3.4.1 Functional Simulation

The simulator used for functional simulation was ModelSim SE 6.2f and the host machine had 4GB RAM (and 5GB swap). Choice of matrix was constrained by the simulator and/or CPU RAM. A large matrix results in large amounts of RAM being consumed by the simulator (in gigabytes), and finally crashes the simulator.

The complete design comprising of five processing elements and seven memory controllers was evaluated using matrices from the University of Florida Sparse Matrix Suite[5] maintained by Tim Davis. The matrices chosen for evaluation represent a range of sparsities as shown in table 3.2.

Simulation was carried out using DDR3 DRAM memory controllers working at 400 MHz. Thus during periods of burst accesses, peak memory bandwidth of 44.8 GB/s can be harnessed from the seven DDR3 memory controllers. Thus as calculated in 2.2, the memory bandwidth allows peak processing capability of 3.73 GFLOPS. However, peak memory bandwidth will rarely be attained. Moreover, the maximum operating frequency of the processing element (PE) designed is 302 MHz which caps peak processing power to 3.02 GFLOPS (604 MFLOPS/PE), and from table 3.2 it is observed that performance of the processing element can reach up to 85% of the peak processing power. This compares favourably to previous work[6].

Table 3.2: Performance Evaluation of Design

	Matrix	Application Area	Non-Zeros (N_z)	Size(n)	$N_z/n^2(\%)$	$MFLOPS/PE$	%Peak
1	<i>gemat11</i>	Power Flow	33185	4929	0.14	520	86.79
2	<i>cryg2500</i>	Material Prob	12349	2500	0.20	512	85.45
3	<i>lns_3937</i>	Fluid Flow	25407	3937	0.16	502	83.2
4	<i>rajat27</i>	Circuit Sim	99777	20460	0.024	333	55.5
5	<i>eurqsa</i>	Economics	46142	7119	0.088	473	78.4

3.5 Optimizations

To improve timing performance, several optimizations were attempted on *Design B* with respect to *Design A*.

- Removed clock enable as pipeline stalling mechanism for the floating point adder and multiplier pipelines.
- Isolated the multiplier and adder pipelines by adding a queue in between.
- Minimized load of pipeline control signals - *astop*, *input_ack* - as much as possible.
- Split the data width of the isolation queue between the multiplier and the adder into data-bits and other bits. The queue has width of $(73 + \text{ROWSTRIP_WIDTH})$. Of these, the least significant 64 bits are for data, the next 8 bits represent row, then end of matrix and then finally rowstrip index. The row bits, after passing through some combinational logic need to be provided as address bits to a Block RAM.

Xilinx offers 512deep \times 72wide as its widest Block RAM primitive. If a unified queue is used, data (64 bits) and row (8 bits) are clubbed together in one Block RAM primitive. This creates a big critical path for the row bits (~ 1.9 ns access, ~ 1.6 ns routing, ~ 0.1 ns combinational logic, again ~ 2 ns routing to the BRAM and ~ 0.27 ns setup time) of ~ 5.8 ns. If instead, the row bits are separated from data, then the queue can be implemented using LUT resources which offer smaller delay path and more placement options for the placement tool in the implementation flow.

(NOTE:Static timing analysis showed the largest critical path for row bits. However it is possible that the other bits (end of matrix and rowstrip index) also suffered from large delays. So all these bits were shifted for implementation on LUTs)

- Broke the adder input selection logic into two stages. In the first stage, two 64-bit 4-to-1 multiplexers bring the number of selections from 8 to 2. In the second stage one 64-bit 2-to-1 multiplexer selects the adder input.

(NOTE:If the design is modified to contain 64deep \times 2-word wide buffers instead of the current 32deep \times 4-word wide buffers, adder input selection can be done in one stage. Making the buffer 128deep \times 1-word wide is not recommended owing to the large number of clock cycles required to transfer partial sums out.

- The state machine *psum_ctrl_fsm_cs* has a state **szblock_wait** which ensures that the signal *rowstrip_diff* is not involved in the multiplexer logic which decides its own value, thus improving timing.
- The *streamout_*i* outputs of the partial sum storage are registered so that the outputs of the partial sum storage Block RAMs are not directly assigned to the output buffers.

Chapter 4

Application: Iterative Solver

4.1 Overview

Sparse matrix vector multiplication is typically used while solving differential or linear systems of equations. In this chapter, we demonstrate building a system to implement the Gauss-Jacobi iterative method for solving linear systems of equations using the sparse matrix vector multiplication unit (*Design A*) described in chapter 3.

Typically, in most iterative methods, and Gauss-Jacobi is no exception, operations other than sparse matrix vector multiplication have structured accesses. These operations can be efficiently performed by a simple off-the-shelf micro-processor in conjunction with a cache. Moreover, for building a complete standalone system, peripheral units like memory controllers, DMA controllers, interrupt controllers, bus controllers will be required which are easily available tailored for the particular processor. This aids in the task of building a system around our sparse matrix vector multiplication unit, by itself useless, which is why we pursued an embedded systems approach for development. Moreover, a proposition demonstrating use of such a peripheral in an embedded system is attractive as it would expand the capabilities of embedded systems. Of course, building a auxiliary co-processor which will do these operations is always possible and may be made even more efficient using parallelizing techniques. However, in such an approach, the auxiliary components too would have to be designed which would involve a considerable investment of time.

Xilinx Platform Studio provides a good platform by providing the proprietary MicroBlaze soft micro-processor core, which can be configured as per system requirements. Using Xilinx peripherals, a complete system was built around our processing element.

Xilinx XUPV5-LX110T development board was used for implementing the system. The board has a single 256MB DDR2 SDRAM (Double Data Rate 2 - Synchronous Dynamic RAM) SODIMM (Small Outline - Dual Inline Memory Module) module. Since only one DRAM interface is available to us, we have to time multiplex our requests. We can do only one of the three DRAM operations - read vector, read matrix or store partial sums - at a time.

4.2 Building the system

The Base System Builder (BSB) Wizard in Xilinx Platform Studio (v11.4) was used to configure the system and the micro-processor. The system was configured to be a single processor system with a 125 MHz floating point unit enabled MicroBlaze processor. The processor was chosen to have 32 KB local memory and no cache. Since our processing element was designed to operate on data stored in external DRAMs, a DDR2 SDRAM controller connected to a DRAM on board was used. Interrupt controller was also included in the system to enable the processor to respond to processing element requests. Finally, a UART following RS232 protocol was included to aid in debugging and result checking.

There were two options for connecting the processor to the processing element using different buses. The first option (4.2.1) used a point to point bus (FSL) while the second (4.2.2) used a shared bus (PLB).

4.2.1 Plan A - Using Fast Simplex Link (FSL) Bus

Initial approaches were aimed at connecting the processing element to the processor through FSL (Fast Simplex Link) Buses since they utilize minimum handshaking and being point to point need no arbitration. The MicroBlaze would get data from the DRAM and place it on the FSLs. The FSL Bus follows FIFO based communication. The block diagram below (Figure 4.1) gives an overview of the control signals involved.

The FSL bus accepts data stable on the lines of FSL_M_Data at a positive clock edge if FSL_M_Write is high (and FSL_M_Full is not asserted). Data can be read off at the slave end if FSL_S_Exists and FSL_S_Read are both stable and high at positive clock edge. Similar protocol is followed if the data word is a control word, except that FSL_M_Control also needs to be stable at the positive clock edge along with data.

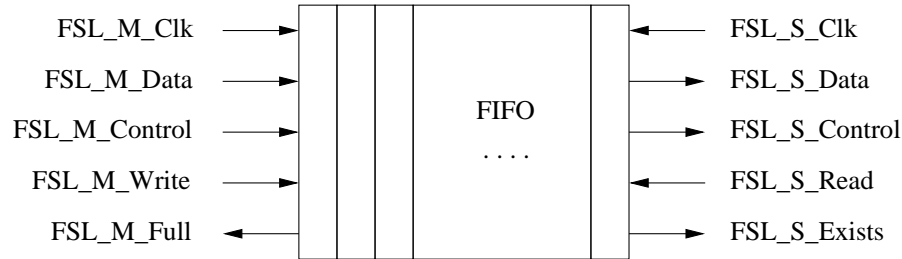


Figure 4.1: Fast Simplex Link (FSL) Bus Block Diagram

However a point to point bus with the other end not connected to a memory was not conducive for our bandwidth hungry peripheral. Every unit of 32-bit data when accesses from DRAM has to pass through the micro-processor local memory before it is placed on the FSL bus. Moreover, data cannot be brought into the processor local memory from the DRAM in burst. The processor can make only random 32-bit data accesses to the DRAM. This approach would have considerably reduced the effective input bandwidth. Hence the use of FSL buses was abandoned.

4.2.2 Plan B - Using Peripheral Local Bus (PLB)

The Peripheral Local Bus (PLB) IP provided by Xilinx is a shared bus which supports burst data transfer. This would provide better data input bandwidth for our processing element. Burst data transfers can be realized by incorporating a DMA on the bus. Other peripherals like interrupt controller, UART are also available as modules compatible with PLB. Moreover, design complexity is reduced if the system has only one bus. Hence, we chose to use PLB over FSLs in our system.

4.3 Adding Processing Element to the System

The processing element is to be added as a peripheral on the PLB bus in our system. The “Create Or Import Peripheral Wizard” in Xilinx Platform Studio creates a wrapper for our peripheral so that it can be easily incorporated in the system. The wrapper incorporates among other things, a PLB IPIF (Interface for PLB) module which on one hand interfaces with the PLB, and on the other provides easy to use interface for the user design. Thus the user design does not need to bother about the intricacies of the PLB handshaking and transfer protocols.

In the system designed, the processing element never needs to initiate any transfers and hence, the wrapper is configured to include only the PLB slave interface and no master interface. The wrapper can also be configured to include programmer visible registers (32-bit), programmer visible memory (64K) spaces (32-bit wide), interrupt control, read FIFO and write FIFO.

The processing element needs three interfaces with the external world - first for accessing vector blocks, second for accessing matrix elements, and third for storing partial sums. Each interface is associated with an active high level interrupt which would signal the processor about the state of the processing element. The interrupt control block included in the wrapper generates a unified interrupt signal from the above three interrupt sources. This interrupt signal is fed to the system interrupt control unit which unifies interrupts of all peripherals in the system (in our case, the DMA and the processing element), and generates the interrupt request signal for the MicroBlaze.

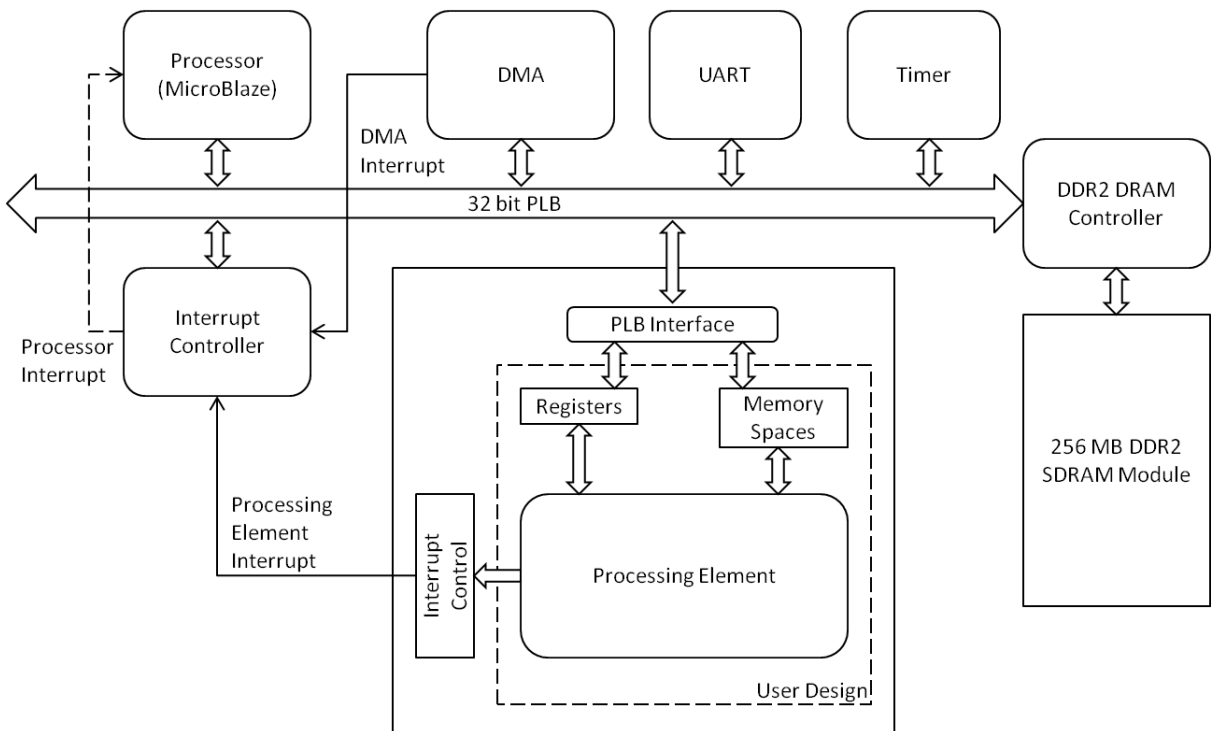


Figure 4.2: System Block Diagram

Since vector and partial sum transfers would be fixed size burst transfers, an interface involving memory spaces is the best choice. Matrix elements too can be brought in fixed size blocks, making the design process simpler by having a similar interface for all three. Aside from the three memory spaces, the wrapper is configured to have three programmer

visible registers. One register is used to communicate interrupt conditions to the software. Another register is used to communicate state of the software to the interface mechanisms. Finally the third register is included as an aid in debugging.

Note that, although we have memories for vector and partial sum at the interface, we cannot do away with the memories used as vector caches or partial sum caches inside our processing element. The underlying reason is that the system is 32-bit while the processing element operates on double precision which is 64-bit data. As a result, inside the memories at the interface, data will be broken into two chunks placed at successive locations, a different data format from the one the processing element was designed to process. Moreover the vector caches inside the processing element are four 64-bit words wide. To emulate such a cache using the 32-bit memory spaces would require data to be brought in or stored in DRAMs in an inconvenient format. Also, we want to demonstrate working of the processing element as a portable complete peripheral independent of the platform, and hence chose not to modify the design solely for a particular embedded implementation.

The resultant system has the configuration as shown in figure 4.2.

4.4 Designing Interfaces for the Processing Element

Transfer of matrix, vector or partial sums between the processing element and DRAM is designed to be a two-stage process. One stage is more closely associated with the particular interface - vector, matrix, partial sum - and hence is different for each, although the skeleton is similar. The other stage co-ordinates with the software and the interface state machines to transfer data between the memory spaces and DRAM. The state machines controlling this stage for matrix, vector and partial sum transfer are hence very similar.

In section 4.4.1, the state machines co-ordinating with software are explained. Since the state machines are designed in conjunction with the software flow, a background of the software involved in the transfer is also provided. The individual FSMs interacting with the processing element are explained in the subsections 4.4.2, 4.4.3 and 4.4.4.

4.4.1 Co-ordinating with software and hardware

Each of the three interfaces has a state machine which generates the interrupt signal requesting transfer of data and then communicates with the software till the end of the transaction. In this section, the control flow in both hardware and software is explained side by side.

In the case of vector and matrix interfaces, interrupt is generated when the memory spaces for vector and matrix storage respectively are empty. In the case of partial sum interface, interrupt is generated when the memory space has the complete final vector block ready for transfer. The hardware state machine, after generating the interrupt, waits for acknowledge, then masks the interrupt signal and waits for the transfer complete notification after which it goes back to its initial state.

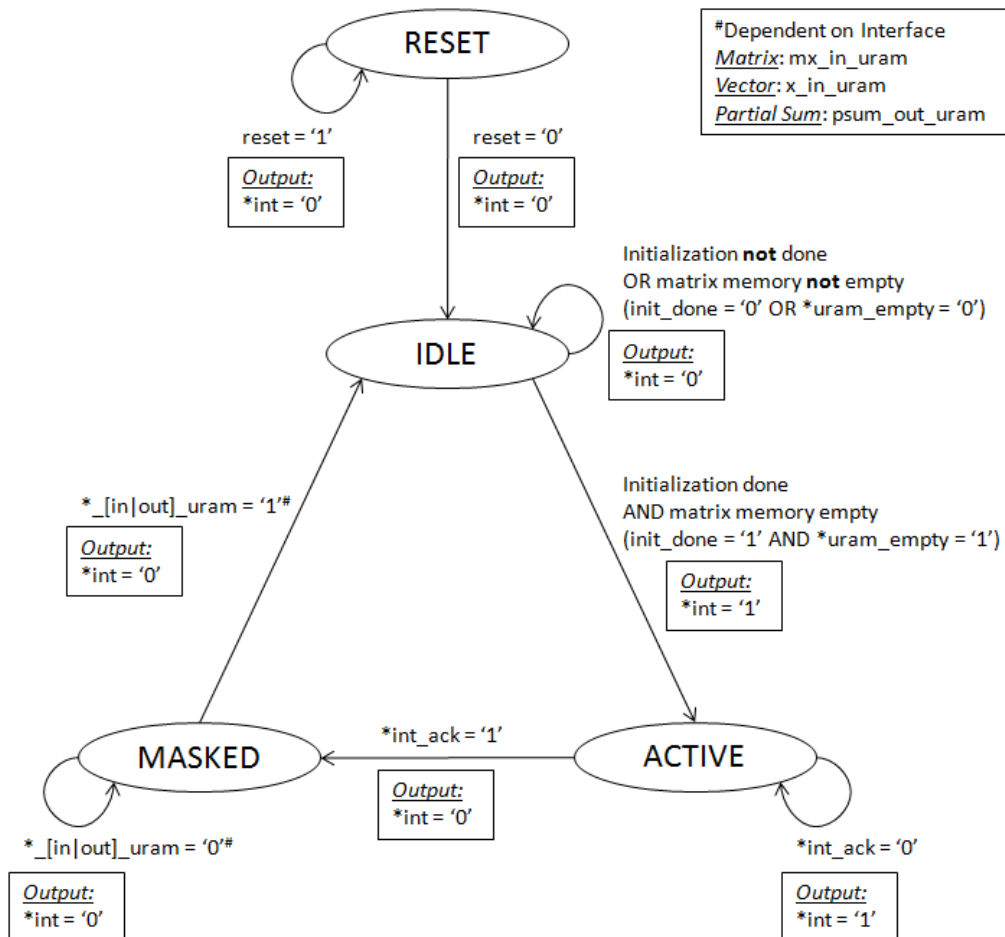


Figure 4.3: State machine to co-ordinate with software

Receipt of an interrupt is the cue for the software to configure the DMA appropriately. Interrupt acknowledge and DMA transfer are multi-cycle processes and thus the

hardware state machine has no indication about their progress. It will keep on interrupting the processor even after interrupt has been acknowledged, if there is no communication between the two.

As depicted in the state diagram (figure 4.3), for each interface we use a pair of signals - *interrupt acknowledge* and *transfer done* - which are predefined to be certain bits in a user register. Initially all signals are cleared to represent *false* condition. When an interrupt is received, software clears the *transfer done* signal and only then sets *interrupt acknowledge*. This ensures that the *transfer done* signal set high for the previous transfer is not interpreted by the hardware state machine for the current transfer. After the DMA signals the end of transaction, first the *interrupt acknowledge* is cleared and only then *transfer done* is set. This is essential because a register write by software is a multi-cycle process. If *transfer done* is set before clearing *interrupt acknowledge*, and an interrupt request is generated in between, then the hardware state machine would interpret the acknowledge for the most recent interrupt.

4.4.2 Vector Interface Control

A fixed number (256) of 32-bit words have to be transferred while transferring a 128-element vector block to the processing element. Hence one user memory instance which implemented as a Block RAM primitive is used to provide storage for the vector data. Once the BRAM has been filled up, the data is transferred to the processing element in 32 sets of 4 double-precision (64-bit) words. The state machine performing the data format conversion is depicted in Figure 4.4 below.

This state machine also generates the transfer control signal following the protocol recognized by the processing element. After successful transfer of vector block to the processing element, the state machine described in section 4.4.1 is notified that the memory is empty.

4.4.3 Matrix Interface Control

As explained in section 3.2, for *Design A*, matrix entries can be treated as a 96-bit wide word with the three most significant bits unused. As in the case of vector interface, the DMA transfers data into a user design accessible 32-bit wide memory block. Since the

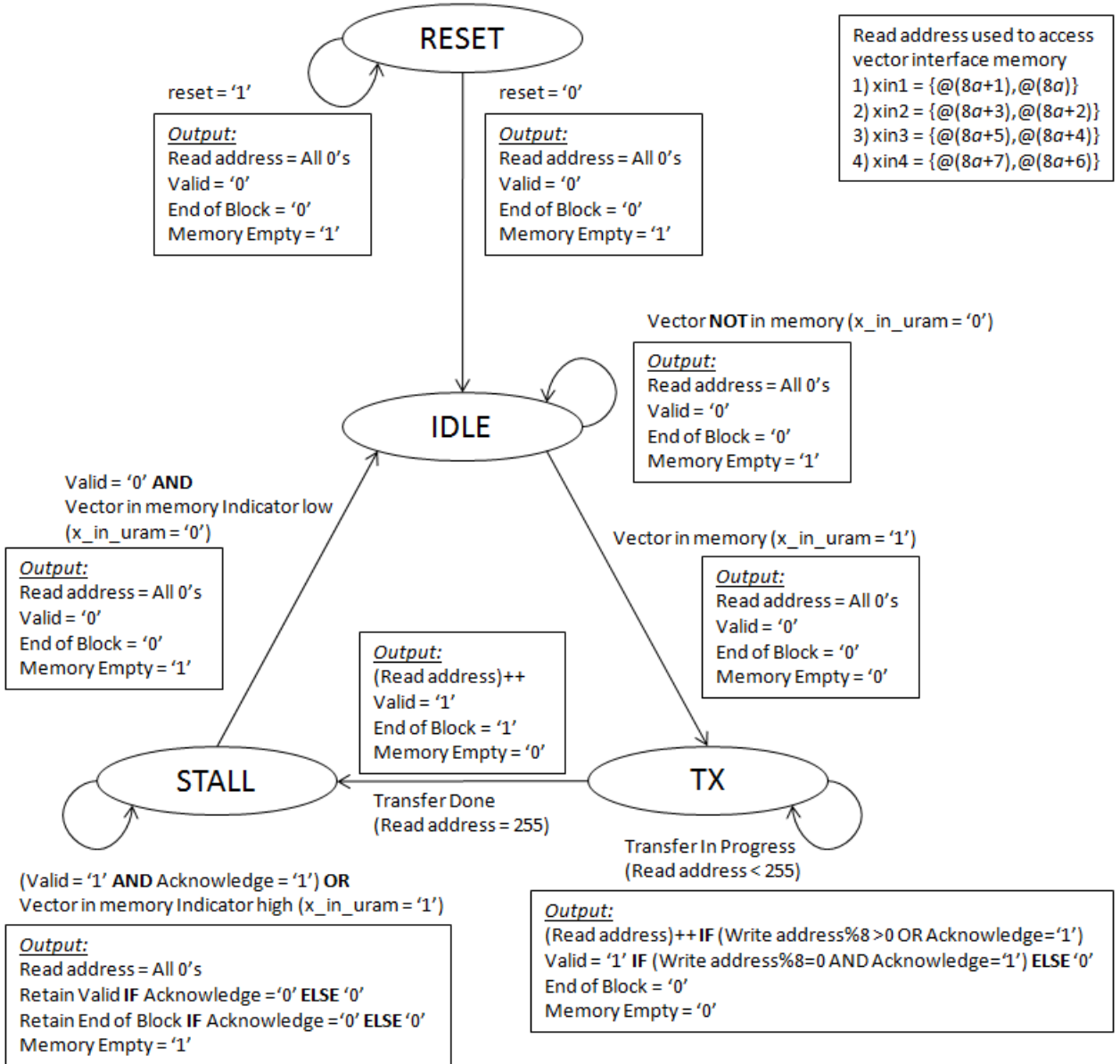


Figure 4.4: FSM for Vector interface for PE

depth of one memory block is 256, and one matrix data word is 96-bits (three 32-bit words), one DMA transfer brings in 85 matrix entries. More memory blocks could have been used to get storage for more than 85 matrix entries.

The state machine for this interface builds the 96-bit word from the 32-bit words at successive memory locations and then sends them to the processing element, following the transfer protocol recognized by the processing element. The state machine for the transfer is depicted in Figure 4.5. After successful transfer of matrix to the processing element, the state machine described in section 4.4.1 is notified that the memory is empty.

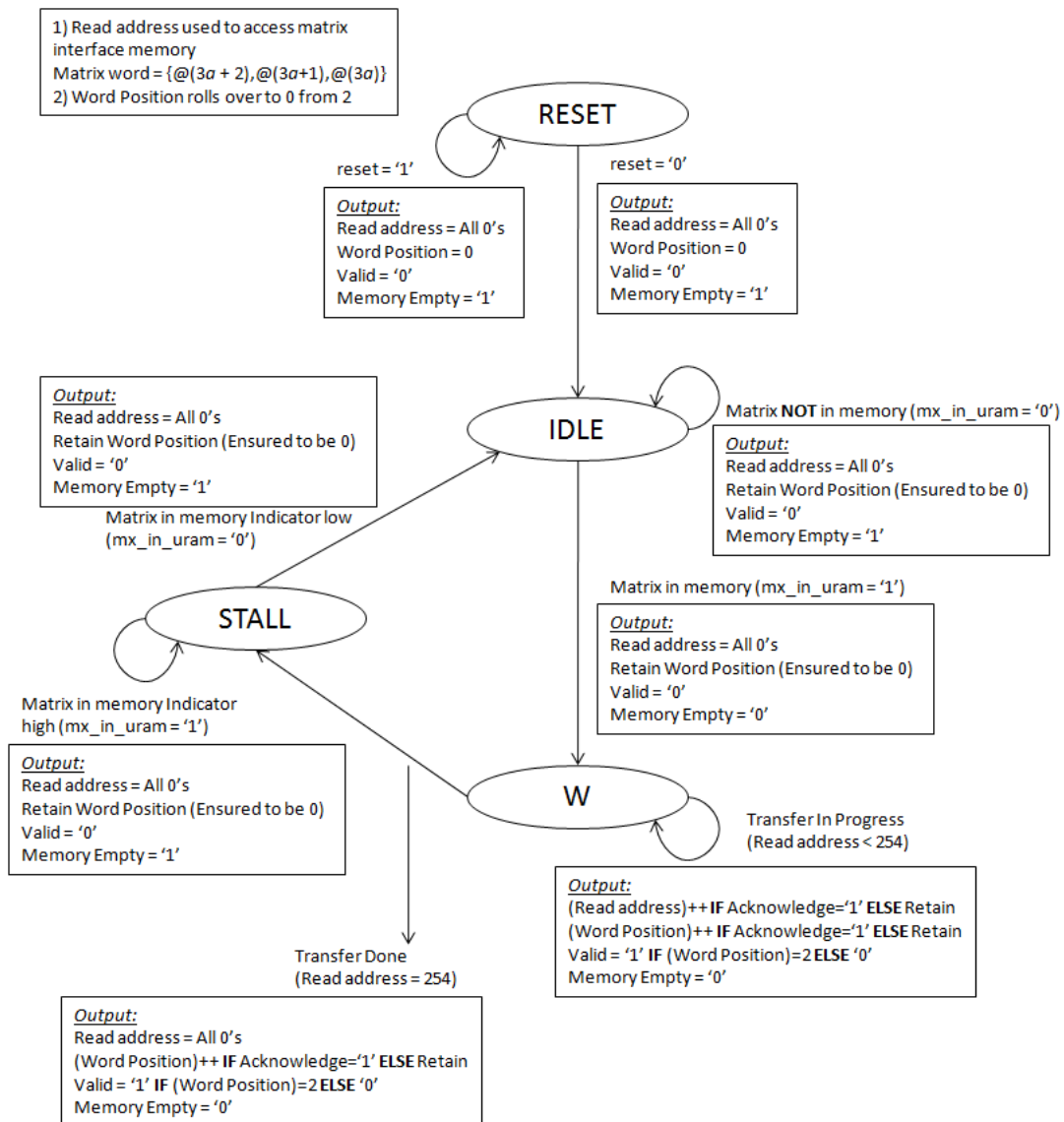


Figure 4.5: FSM for Matrix interface for PE

4.4.4 Partial Sum Interface Control

When the processing element completes processing of a row strip, 128 double precision words (each 64-bit wide) are to be transferred from processing element to the DRAM. The processing element outputs four successive elements of the result vector in one clock cycle. The state machine for this interface formats the data by breaking it into eight chunks each of 32-bits and storing it at successive locations in the memory space. Again as in the case of vector interface, one memory block (256 deep) is used for storing the partial sum data. The state machine for the data format conversion and transfer control is shown in Figure 4.6.

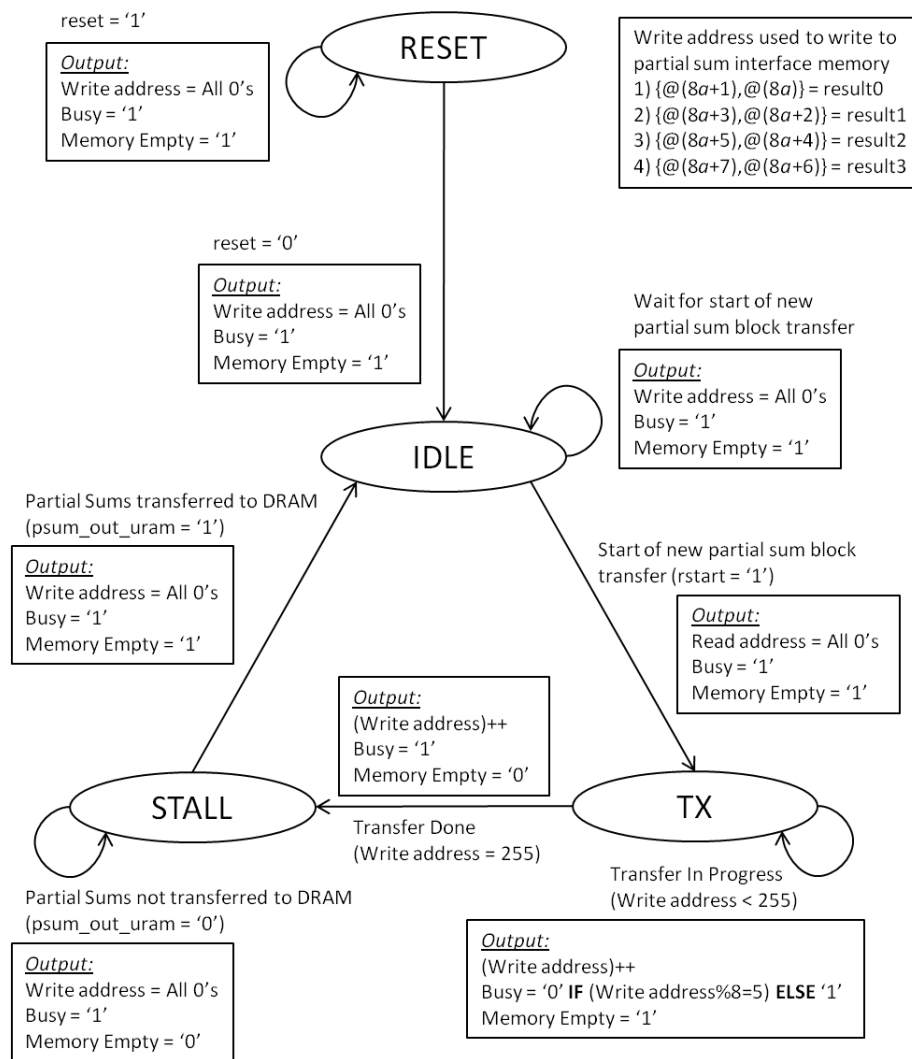


Figure 4.6: FSM for Partial Sum interface for PE

4.5 Software Development

Software for an embedded system can be divided into two classes. The first class includes routines to configure and access the hardware, manage interrupts, essentially create an abstraction for the second class. The second class deals with development of the high level application to implement the desired functionality using the hardware access routines. In the following sections, software developed for the system is described briefly.

4.5.1 Hardware Management

The high level application (in our case - Gauss Jacobi method for iterative solution), usually needs hardware routines to provide data to hardware for processing, accept data

from hardware, finding status of hardware modules and for responding to interrupts. We will describe the routines developed or used to manage different hardware modules like the DMA, interrupt controller and the processing element. The embedded design tool also provides library functions for each device in the system to aid in software development.

4.5.1.1 DMA Controller

The DMA controller needs to be configured and initialized before use. The embedded design tool provides a basic library for the DMA controller which has functions to gather information for configuration (*XDMACentral_LookupConfig*) and apply it (*XDMACentral_CfgInitialize*). Since DMA needs to generate transfer complete interrupt only, the appropriate bit for transfer complete interrupt enable in the interrupt control register of the DMA needs to be set. This can be done by using the library function *XDMACentral_InterruptEnableSet* with the appropriate mask or by explicitly setting the register value using memory mapped IO.

Thereafter, using the DMA involves setting parameters for the transfer, like source address increment and/or destination address increment and length of transfer (number of bytes to be sent). The DMA transfer starts as soon as the *Length* register is updated with a non-zero value. All this can be performed implicitly through the library functions *XDMACentral_SetControl* and *XDMACentral_Transfer*.

In our application, the DMA is used for three types of transfers. Before start of each transfer, a flag is set specifying the type of transfer initiated. The interrupt handler for the DMA complete transfer is a small routine which only clears the interrupt flag and resets the type of transfer flags (*req_str*, *req_vec*, *req_mx*).

4.5.1.2 Interrupt Controller

The processor used (MicroBlaze) has one interrupt line, while there are two interrupt sources in the system. As a result, a module like an interrupt controller is needed, which will collect all interrupts from the system, and create one interrupt request signal. As in the case of DMA controller, library functions provided by the tool can be used, simplifying the design process, and making it more robust with respect to hardware revisions.

Before operation, the interrupt controller has to be initialized using *XIntc_Initialize*. This involves initialization of vectoring tables in the controller and members of the struc-

ture used in software for managing the controller. Following initialization, the interrupt sources from different peripherals are connected to the controller using *XIntc_Connect*. The interrupt controller is then started (*XIntc_Start*) and the interrupt enables in the controller for each of the interrupt sources are set (*XIntc_Enable*). Finally, we have to register the interrupt vector table of the controller with the processor (*microblaze_register_handler*) and enable interrupt on the processor (*microblaze_enable_interrupts*).

4.5.1.3 Processing Element

Low level software for the processing element only comprises of the routine to manage its interrupt. The interrupt handler clears the interrupt flag, disables the interrupt, reads the interrupt source register in the processing element and sets the appropriate transfer request flag.

The interrupt is held active by the processing element till the software acknowledges it by setting bits in the processing element register. The acknowledgement is performed by software outside the interrupt, hence to prevent multiple interrupts for the same event, interrupt needs to be disabled in the handler. It is enabled again after the DMA transfer is successful.

4.5.2 Software Application - Gauss Jacobi

Gauss Jacobi method for iterative solution is implemented to demonstrate proof of working of the processing element. The software first generates a linear system of equations and initial guess and stores them in the DRAM in the appropriate format as required by the processing element. The processing element is then signalled that data initialization is done. Thereafter, the processor performs the rest of the algorithm (besides matrix vector multiplication) as and when data is ready. Occasionally, it needs to configure the DMA, communicate with the processing element and when an iteration is done, reset the processing element for the next iteration.

A flowchart describing the software is depicted in figure 4.7.

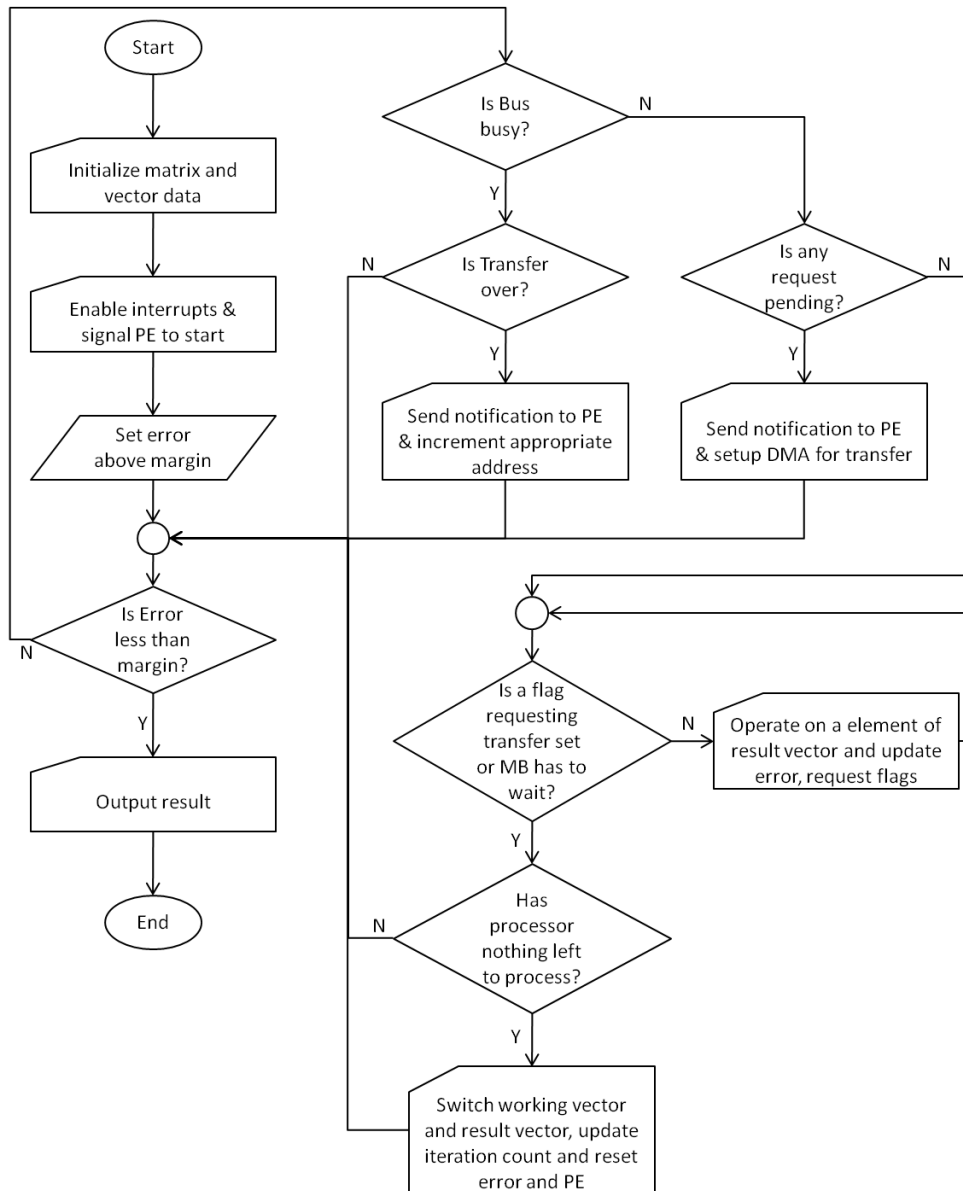


Figure 4.7: Application Software Flowchart

4.6 Evaluation

The system described in the previous sections of this chapter is compared against a pure software Gauss-Jacobi iterative solver. A tri-diagonal matrix was generated by the processor and written to the DRAM before processing. The table 4.1 below shows the comparison of the two systems for different matrix sizes, with a 64KB data cache enabled for both setups. Both the systems provided solutions after 6 iterations in all cases. However, the accelerator system had a 3% inaccuracy.

As shown in the table 4.1 above, for the pure software approach processing time

Table 4.1: Performance comparison for Gauss-Jacobi solver

	Matrix	Storage	Software System			Accelerator System
			Iteration 1	Iteration 2	Iteration 3	
1	640×640	General	0.713 ms	0.709 ms	0.709 ms	0.593 ms
		Compressed	0.251 ms	0.108 ms	0.072ms	
2	3200×3200	General	3.558 ms	3.536 ms	3.536 ms	2.777 ms
		Compressed				
3	12800×12800	General	14.224 ms	14.138 ms	14.137 ms	11.016 ms
		Compressed				

depends on how the data is stored in DRAM. If data is stored closely in a small region of DRAM (referred to as *Compressed* in the table), then for small matrices like the 640×640 matrix which can fit in the data cache, the processing time decreases drastically. This corresponds to almost one operation per cycle throughput. However this advantage vanishes even for small matrices if the data is spread out over the entire address range of DRAM. This may be due to loss of effectiveness of the direct-mapped cache due to absence of spatial locality. The accelerator system remains unaffected by arrangement of data in memory.

From the table 4.1, it is evident that the speedup of the accelerator system is only marginal (about 25%) over the software approach. A closer look at the time spent on one iteration revealed that a large amount of time is spent in transfer of data between accelerator and DRAM via DMA. When the DMA read and write burst sizes are increased from 8 to 16, and the internal DMA FIFO length (provided as a parameter by EDK) is increased from 8 to 48, we get a substantial 1.5X speedup. The table 4.2 provides the time required for transfers for both DMA configurations for the 12800×12800 matrix.

Table 4.2: Accelerator System DMA Transfer Time

	Number of Transfers	DMA Configuration	
		8-8-8	16-16-48
Matrix	300	5.269 ms	3.305 ms
Vector	100	1.749 ms	1.077 ms
Results	100	1.164 ms	0.735 ms

For the 16-16-48 DMA configuration, the time required for just matrix-vector multiplication for the 12800×12800 matrix is 7.55 ms which corresponds to a throughput of 6.73 MFLOPS. There are several reasons for this low throughput, which are enumerated below:

1. The soft processor available is a 32-bit processor. This decreases bus width to 32 bits, in turn decreasing the bandwidth available to the processing element. Multiple clock cycles are required to transfer just one matrix/vector/result word.
2. Embedded system can contain only one bus. Requests to the DRAM have to be made on the same bus on which the DRAM provides data. Firstly, this reduces throughputs since requests cannot be queued. Secondly, this will increase number of bus turnarounds.
3. The board provides only one DRAM interface thus decreasing number of processing elements supported and imposing need to time-multiplex transfers.
4. The DMA controller module makes requests for maximum burst size of 16. Such a small burst is not sufficient to hide DRAM access latency.
5. Data is first transferred from source to the internal DMA FIFO, and then transferred from the FIFO to the destination. Thus data needs to be transferred twice along the same bus. The time required for bus turnaround worsens the situation.

The limitation caused by the last point mentioned above can be eased by including a PLB master attachment to the accelerator peripheral. Such an attachment would enable data to be transferred in burst (of maximum size 16) directly from source to destination.

4.7 Design Flow

- Using the Base System Builder (BSB) Wizard, build the basic system and include the relevant peripherals. In this design the only peripherals selected in the BSB wizard were: Instruction memory controller, Data memory controller, DRAM controller, RS232 UART.
- Modify the User Constraints File (UCF) in \$PROJECT/data folder. The IO Pin locking constraints for the DDR2 SDRAM controller port signal “dq0[63:0]” are

repeated twice in the file, so to take care of MAP stage errors, the constraints which are repeated at the end of the file need to be commented out.

- Using the “Create and Import Peripheral Wizard”, generate a wrapper around the peripheral with the configuration described in Section 4.3 with name \$NAME.
- Generate the interfaces between the PE and PLB slave IPIF and instantiate the processing element in the wrapper file.
- In case design contains many design files, include the VHDL module files in the directory \$PROJECT/pcores/\$NAME_v1_00_a/hdl/vhdl. Include verilog files if any in \$PROJECT/pcores/\$NAME_v1_00_a/hdl/verilog
- In case the design contains netlists, copy them to directory (after creating it if required) in \$PROJECT/pcores/\$NAME_v1_00_a/netlist. However, wrappers for the netlist **MUST** be included in the appropriate folder (\$PROJECT/pcores/\$NAME_v1_00_a/hdl/vhdl/ OR \$PROJECT/pcores/\$NAME_v1_00_a/hdl/verilog). Moreover include the synthesis constraint “BOX_TYPE” with the value “black_box” for each instance of the module/entity for which netlist is to be used instead of the wrapper.
- In case, VHDL and Verilog files are added to the design (of the peripheral), the PAO (Peripheral Analysis Order) file in \$PROJECT/pcores/\$NAME_v1_00_a/data/ should be modified. Include statements of the form:
lib \$NAME_v1_00_a filename vhdl/verilog
- In the MPD file, include the following statements if netlists are included in the design:
 OPTION STYLE=MIX
 OPTION RUN_NGCBUILD=TRUE
- If design is a mixed design (both VHDL/Verilog), include (or modify) the following statement in the MPD file:
 OPTION HDL=MIXED
- If netlists are a part of the design, a BLACK BOX DESCRIPTION file (.bbd) file with the name “\$NAME_v2_1_0.bbd” in the directory “data” within the directory

`$PROJECT/pcores/$NAME_v1.00_a/`. The structure of the data in the file is as follows:

FILES

##IMPORTANT: "FILES" MUST BE THE ONLY WORD ON THE FIRST LINE

`netlist1.ngc, netlist2.ngc, ...`

Chapter 5

Matrix Pre-Processing

5.1 Why Pre-Processing?

- The processing element consumes matrix entries in blocks. Standard matrix representations are in the row-major or column major format. Hence it is necessary to pre-process the matrix to format the data input to the matrix as well as provide block number and row/matrix finish indicators.
- The current architecture only operates on ‘dense’ regions of the matrix and hence pre-processing is needed to identify these regions
- Pre-processing will help in balancing the load between the processing elements, thus maximizing performance
- The computation pipeline in the processing element has a feedback path which can cause RAW hazards. Pre-processing will ensure that such cases are minimized or at least that null computations are inserted to avoid data corruption

The figure 5.1 depicts how RAW hazards can occur in the system owing to the pipelined accumulator.

Pre-processing involves two independent stages. In one stage, ‘dense’ regions in the matrix are identified. In another, the non-zero elements in a given ‘dense’ block are optimally scheduled for the processing elements. In the following sections, first the algorithm for optimal scheduling is provided, followed by identification of the ‘dense’ blocks.

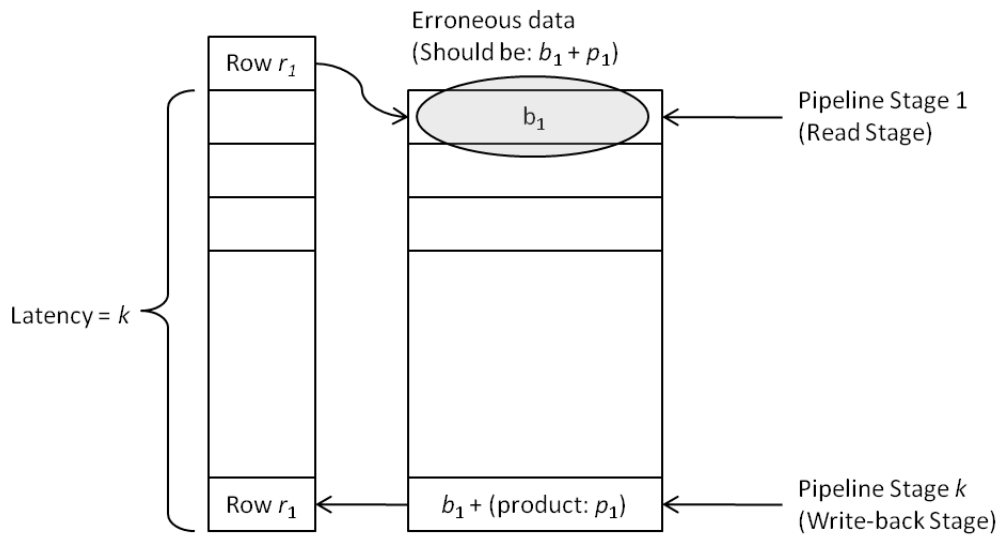


Figure 5.1: Occurrence of hazards in adder pipeline

5.2 Optimal Scheduling Problem

5.2.1 Formalization

Given a value a and a pool of elements each having an attribute (*row number*) having a value bounded on both sides, generate a known number of sequences - k - such that:

1. The sequences are as balanced as possible *w.r.t* number of elements.
2. In each sequence, elements having a particular value of *row number* are separated by at least a elements. If this is not possible, null elements - having special value of *row number* and not bound by this constraint - can be inserted. Null elements count towards the length of the sequence.
3. The length of the largest sequence be as small as possible.

In our problem of matrix pre-processing, k denotes the number of processing elements while a denotes the adder latency. Note that adder latency of a implies that there should be a elements **between** two elements having the same *row number*.

5.2.2 Algorithm

Initially it is assumed that we have to generate only one sequence ($k = 1$). It can be shown that once we have this sequence, generating k sequences is trivial. In such a situation (k

= 1), the optimization constraint is slightly modified to finding the ONE sequence with the least length. Equivalently, the solution sequence should have minimum number of null elements.

All the elements are organized into lists on the basis of their *row number*. These lists are then arranged in increasing order of length. Since, *row number* is bounded from above and below, this sorting can be performed in linear time. To start with, the list with maximum number (say b) of elements is chosen and its elements are arrayed as shown in figure 5.2. If there are two or more lists with the maximum number of elements, then any one can be chosen. The figure 5.2 also shows the traversal pattern which will generate the final sequence. The X 's denote null elements placed so that the minimum distance constraint a is not violated. Thus at this stage, the number of null elements in the sequence is $a(b - 1)$, since trailing X 's in the last row do not count towards length of sequence. They are incorporated in the figure for ease of explanation.

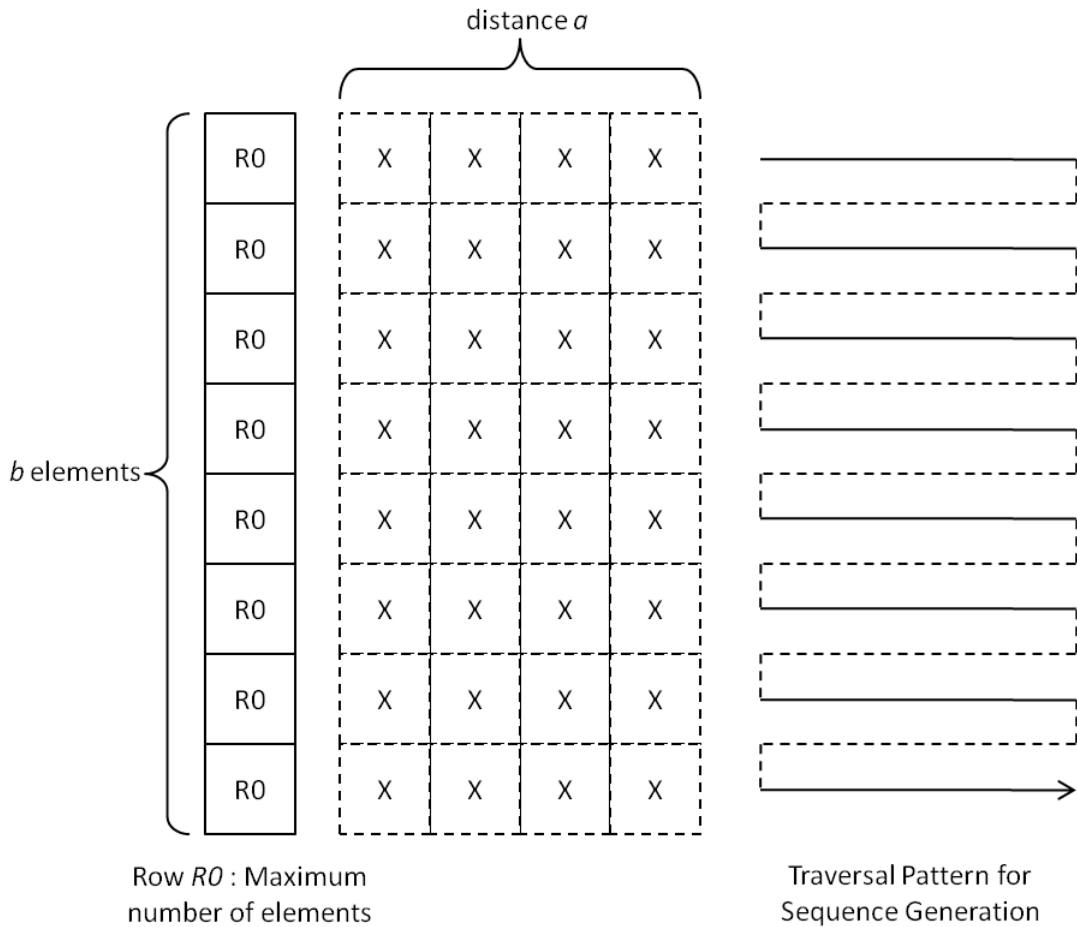
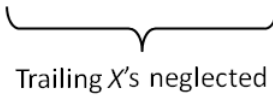


Figure 5.2: Scheduling in a block:Initial phase

R0	R1	R2	R3	R5
R0	R1	R2	R3	R5
R0	R1	R2	R3	R5
R0	R1	R2	R4	X
R0	R1	R2	R4	X
R0	R1	R3	R4	X
R0	R1	R3	R4	X
R0	R1	X	X	X



Trailing X 's neglected

Figure 5.3: Scheduling in a block:Continuation

Thereafter, we check if there are other lists with b elements. For every such list, we are able to replace an entire column of X 's with elements of the list without violating the minimum distance constraint. The remaining lists have a maximum of $(b - 1)$ elements. Only at this stage, we remove the trailing X 's in the last row. In the previous step, all columns of X 's may have been replaced by elements. Even in this case, the rest of the algorithm described below holds. Only difference is that instead of replacing X 's with elements, new columns will be appended.

For the remaining lists, we replace as many X 's in a column as we can and wrap the elements of successive lists to the next column. This step is showed in more detail in the figure 5.3. Since the lists are in decreasing order of length, no list wraps around and reaches even the row just above the one where the list started in the previous column. As a result, minimum distance constraint is never violated. Again, if all X 's are replaced, then new columns whose maximum length can be $(b - 1)$ are appended.

Generating k sequences just requires a slight modification of the above algorithm. After the rows of the block are sorted in decreasing order of number of non-zeros, starting

from the longest row, non-zeros of the rows are distributed amongst the k sequences in a continuous fashion. For each of the k sequences, the non-zeros are grouped according to the row number and again the rows are arranged in decreasing order of number of non-zeros (assigned to the sequence under consideration). These steps are described in figure 5.4. Thereafter the non-zeros in each sequence are arranged as shown in figures 5.2 and 5.3 using exactly the same strategy to get the allocation schedule for each sequence. The grayed elements in figure 5.4 represent the longest rows for that sequence which will be on the leftmost in the structure in the strategy described above.

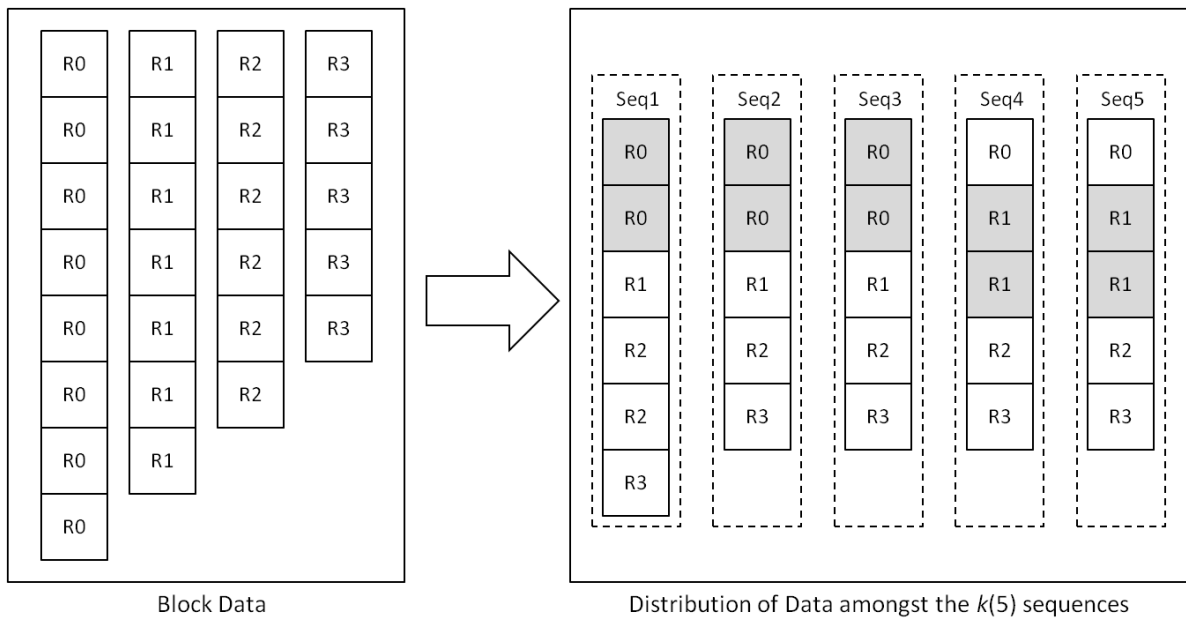


Figure 5.4: Case for k sequences: Distribution of data

5.2.3 Proof of Optimality

From section 5.2.2, it is clear that for the worst case problem, the maximum number of null elements is bound to a constant and is $a(b - 1)$. Now we proceed to show that the scheduling is optimal by considering two mutually complementary cases.

1. **Case A:** Total number of elements in pool less than or equal to $a(b - 1)$

Given that the longest list has b elements, the minimum length of the list is $(a(b - 1) + 1)$. This is the minimum required length enforced by the distance constraint and we cannot do better than that. Also note that, we never add X 's in the entire duration of the algorithm. The null elements represented by X 's are replaced with

the list elements. Since the number of elements is less than or equal to $a(b - 1)$ which is the number of X 's, this operation does not modify the length of the list. Thus our algorithm generates the optimal solution for the specified constraint in this case.

2. **Case B:**Total number of elements in pool greater than $a(b - 1)$

In the algorithm, we started off with $a(b - 1)$ X 's with no X 's added over the course of algorithm. In this scenario, all X 's will be replaced by list elements. In this scenario, the length of the list will be exactly equal to the number of elements in the pool. Again, we cannot do any better than this and thus the algorithm has given us an optimal solution for the given constraints.

For generation of k sequences, only the optimality of distribution of non-zeros amongst the sequences has to be shown.

Proof:If the row with maximum number of non-zeros has b elements, the algorithm ensures that in each of the k sequences, there are no more than $\lceil b/k \rceil$ non-zeros of any row in a sequence. Moreover by the Pigeon-Hole-Principle, at least one sequence should have greater than or equal to $\lceil b/k \rceil$ non-zeros belonging to a row. Since the length of the schedule has a direct correlation to the maximum number of non-zeros belonging to a row assigned to a sequence, the length of the schedule generated is optimal.

5.3 Matrix ‘Dense’ Block Identification

Like the optimal scheduling problem discussed in section 5.2, identification of ‘dense’ blocks in a matrix is constrained by hardware. First we give a background of the constraints imposed by hardware so that the problem can be defined precisely.

In *Design B*, the result vector needs to be recreated fully by the processing element. For regions of result vector, which correspond to those rows of the matrix not in any ‘dense’ block, the processing element has to explicitly insert zeros. Inserting variable number of elements by the granularity of a row would require more information which would translate to more bandwidth requirement. The extra information requirement is manageable if the insertion is performed at granularity of a row-strip. Thus the hardware imposes the constraint that the rowstrips be $128n$ aligned.

In the most general case, identifying regions in the matrix is a two-dimensional problem. In our case, constrained by the hardware, we have a one-dimensional problem wherein we have to find the ‘dense’ regions in a rowstrip.

5.3.1 Formalization

The above problem can be formalized as follows.

- We are given an array a of size n , with $a[i]$ non-negative.
- Find the positions of non-overlapping windows of size w such that the summation of all array values whose indices fall in any window is maximized.
- A window is valid only if summation of array values whose indices fall in the window is greater than a threshold d .

The array $a[i]$ contains number of non-zero elements in the column of index i in the rowstrip. The window size refers to the size of the matrix blocks under consideration while d refers to the threshold which defines a matrix block to be ‘dense’.

5.3.2 Algorithm

We use principle of dynamic programming to find the best set of windows. First we will define some functions which will help in the solution.

$$b[i] = \begin{cases} \sum_{j=i-w+1}^i a[j] & \text{if } i \geq (w - 1) \\ 0 & \text{otherwise} \end{cases} \quad (5.1)$$

$$c[i] = \begin{cases} 0 & \text{if } i = 0 \\ \max\{c[i - w] + b[i], c[i - 1]\} & \text{if } i \geq w \ \& \ b[i] \geq d \\ c[i - 1] & \text{otherwise} \end{cases} \quad (5.2)$$

The function $b[i]$ computes the summation of elements in the window ending at position i . The function $c[i]$ is the actual function which is maximized using dynamic programming.

In each step, represented by consideration of a new index, we have the option of using the window ending at the new index or not. Using the window ending at the new index

implies that as of now, inclusion of the element at the index provides a higher sum and thus a better solution. Of course the rest of the solution will have to change to account for the constraint of non-overlapping windows. In fact the new solution has this new window appended to the best solution at that index which occurs at a distance of *width of a window* before the new index. The combined summation of the elements at new window indices and elements in the solution at indices more than window distance before the current index, was more than the summation of the elements at indices included in the solution at the index immediately to the left. This is represented by the second line of definition of the function *c*.

5.4 Software Implementation

The algorithm described above is implemented in C++ and has been evaluated for matrices from the University of Florida Sparse Matrix Suite for runtime, coverage (*ratio of number of non-zeros selected to total number of non-zeros*), efficiency (*ratio of nonzero elements selected to final number of entries in list*). The runtime category only considers processing time and not the time required for writing schedule to hard disk. The software

Table 5.1: Results for Matrix Pre-processing

Name	Size	Non-zeros	Coverage	Efficiency	Runtime
<i>gemat11</i>	4929	33185	77.43%	99.25%	0.741s
<i>cryg2500</i>	2500	12349	84.42%	99.6%	0.326s
<i>lms_3937</i>	3937	25407	93.29%	99.25%	0.665s
<i>rajat27</i>	20640	99777	73.77%	61.56%	2.954s
<i>webbase</i>	1000005	3105536	65.35%	84.92%	25.204s

implementation takes arguments for defining a ‘dense’ block, adder latency and number of sequences to be generated. Thus lower the threshold for labelling a block as ‘dense’, higher the coverage, but lower the efficiency. For the results above, the block size was set to 128×128 , threshold was set to 83, latency set to 16 and number of processing elements (number of sequences) set to 5. *Design B* actually allows adder latency to be set to 15.

Chapter 6

Conclusion

In this report a modular design for sparse matrix-vector multiplication targeted at the LX-330T FPGA by Xilinx, is presented and discussed. The report discusses the trade-offs for the SpMxV architecture, it aims to provide insight into the design of accelerators. The simplicity of the design and the use of commodity DRAMs makes this design a practical accelerator for the HiPC community. The design can achieve a performance of 600 MFLOPS per PE, which compares favourably with related work.

6.1 Future Work

The section on implementation of the accelerator using embedded development kit suggests using a bus master attachment instead of a DMA to improve performance. Such a modification would most likely cut the data transfer time to half of the present time.

Alternatively, instead of an embedded system on a standalone board, a system can be developed on an accelerator board (Nallatech BenADDA16) connected to a host through a fast interface (PCI Express), which would provide processing results more relevant to the HiPC community and enable testing for large matrices the processing elements were designed to process.

The current design requires vector blocks to be stored in order of requirement in the DRAMs. If the vector fetch units are provided with the information of the order of requirement, vector could be stored in order which is more convenient for iterative algorithms. The vector fetch units can be initialized with this data when the DRAMs are being initialized with data. Moreover, this would also free the matrix blocks from

the requirement of being $128n$ column aligned. This would increase the coverage of the matrix processed by this kernel.

Lastly, this report only discusses design for handling dense regions of the matrix. A processor for the remaining regions of the matrix has to be designed.

Bibliography

- [1] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms,” *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pp. 1–12, 2007.
- [2] L. Zhuo and V.K. Prasanna, “Sparse matrix-vector multiplication on fpgas,” *FPGA 2005: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pp. 63–74, 2005.
- [3] M.deLorimier and A. DeHon, “Floating-point sparse matrix-vector multiplication for fpgas,” *FPGA 2005: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, pp. 75–85, 2005.
- [4] Junqing Sun, Gregory Peterson, and Olaf Storaasli, “Mapping sparse matrix-vector multiplication on fpgas,” 2007.
- [5] Tim Davis, “The university of florida sparse matrix collection,” <http://www.cise.ufl.edu/research/sparse/matrices/>, 2009.
- [6] D.Gregg, C. McSweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, and D. Geraghty, “Fpga based sparse matrix vector multiplication using commodity dram memory,” *FPL*, pp. 786–791, 2007.
- [7] G. R. Morris, V. K. Prasanna, and R. D. Anderson, “A hybrid approach for mapping conjugate gradient onto an fpga-augmented reconfigurable supercomputer,” *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 3–12, 2006.
- [8] G. R Morris and V.K Prasanna, “Sparse matrix computations on reconfigurable hardware,” *Computer*, vol 40, no 3, pp. 58–64, 2007.

Acknowledgments

I am deeply grateful to Prof. Sachin Patkar for his invaluable guidance (and *patience*) which helped in fulfillment of my Dual Degree Project. I am also grateful to Prof. Dinesh Sharma for providing inputs and guiding me during the course of the project.

I thank Mr. Sunil Puranik and Mr. Sanjeev Mangal of Computational Research Laboratories, Pune for their guidance and help provided at various stages in the project. I also thank Yash Deshpande for his contribution in developing the optimal scheduling algorithm for pre-processing. My sincere thanks to Siddharth Joshi for the foundation provided by him and my friends for helping me solve problems during the course of the project.

Lastly, I also thank my parents for their support which helped me to reach where I am.

Date: 1-JULY-2010


Sumedh Attarde