Hardware Consolidation of Systolic Architectures on a Coarse Grained Runtime Reconfigurable Architecture

A Thesis

Submitted for the Degree of

Master of Science (Engineering)

in the **Faculty of Engineering**

by

Prasenjit Biswas

Supercomputer Education and Research Centre

INDIAN INSTITUTE OF SCIENCE

BANGALORE – 560 012, INDIA

OCTOBER 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we build the foundation for the work presented in this thesis. Systolic Array Architectures are widely accepted ASIC solutions for Numerical Linear Algebra algorithms. Starting with an overview of this traditional solution, we gradually open the "can of worms" associated with it. We show how Reconfigurable computing platform can serve to contain the "can of worms". In this context we present REDEFINE, a coarse grained runtime reconfigurable architecture for systolic actualization of Numerical Linear Algebra kernels.

1.1 Overview of Systolic Array Solutions

A systolic array is an orchestration of pipelined processors connected in a network topology. In systolic arrays the speciality is the synchronous data flow between the processing elements, usually with particular outputs from a processing element flowing in predefined directions and serve as inputs to other processing elements. According to Kung and Leiserson [**?**], "A systolic system is a network of processors which rythmically compute and pass data through the system". The primary and most important features of a systolic array architecture are modularity, regularity, local interconnection, a high degree of pipelining, and highly synchronised multiprocessing.

The design and organisation of the systolic array architectures differs from that of the

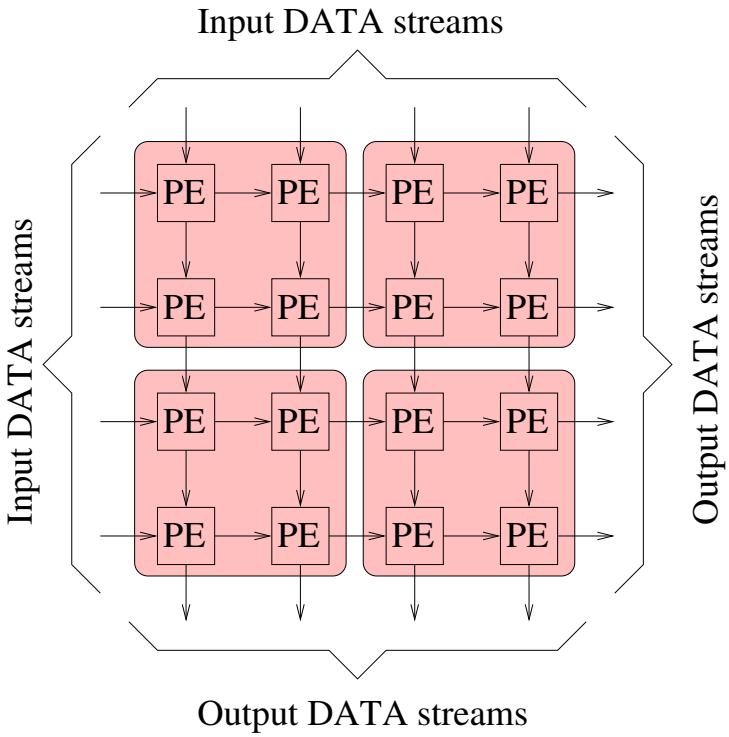


Figure 1.1: A typical systolic array realized on a mesh network. PE stands for "Processing Element". Shaded blocks depict sub-arrays created after the partition of the whole array for optimum array emulation.

conventional Von Neumann Architectures in its highly pipelined and parallel computations distributed over a cluster of processing elements. More precisely, after being recieved from the memory each data-item is used effectively at each processing element it passes while being "pumped" from node to node along the array. There is no global register file arrangement for intermediate data storage. Each processing element maintains an internal register just to store some values to be used as inputs for subsequent computation. Every time one processing element is fired, that stored value takes part in computation, gets modified and stored back for use in next invocation. This avoids the classical memory access bottleneck problem commonly incurred in Von Neumann machines. Figure 1.1 shows a typical example of a systolic array realized on a mesh topology.

So, in essence a systolic array is a computing network possessing the following features:

- **Synchrony**: The presence of a global clock ensures rhythmic computations and the

data produced by those computations proceed through the network.

- **Modularity and regularity**: Modular processing units connected in homogeneous network provide the basic skeleton for any kind of systolic array architecture. Because of structural regularity indefinite extention of the computing network is possible.

- **Spatial locality and temporal locality**: The array manifests a locally-communicative interconnection structure, i.e., spatial locality. There is at-least one cycle delay alloted so that signal transaction from one node to the next can be completed, i.e., temporal locality.

- **Pipelinability**: The array exhibits a linear rate pipelinability.

The major factors favoring systolic arrays for special purpose processing architectures are: simple and regular design, concurrency and communication, and balancing computation and I/O [?].

Simple and regular design: In integrated-circuit technology the cost of design grows with the complexity of the system. By using a regular and simple design and exploiting the VLSI technology, great savings in design cost can be achieved. Furthermore, simple and regular systems are likely to be modular and therefore can be adjusted to meet various performance goals.

Concurrency and Communication: An important factor that contributes to the potential speed of a computing system is the use of concurrency. For special purpose systems, the concurrency depends on the underlying algorithms employed by the system. When a large number of processors work together, communication becomes significant. Concurrent computations should be given more priority over the communication requirements while designing such a system. Systolic arrays flaunt regular and local communication among the nodes which are in concurrent execution. Thus systolic architectures get the certification of performance advantage.

Balancing computations with I/O: A systolic array can be used as a stand-alone ASIC solution as well as a co-processor or as an attached array processor. In both the

cases a proper balance between the computaion rate and I/O rate should be maintained. Generally as a monolithic ASIC the array works at a very high frequency, the operating frequency of the host computer, through which the data is recieved from and the output is sent to, is very less in comparison. Therefore in determing the overall performance the I/O considerations are taken into account. The ultimate performance goal is achieved in systolic array by maintaining a computation rate that balances the available I/O bandwidth with the host. To achieve this proper handshaking signals are used. And this introduces a hint of paradigm shift from synchronus domain to asynchronus domain. The marriage of systolic array philosophy and asynchronus data-flow computing gives birth to wave-front array. We explore the features of wavefront arrays in forthcoming chapters.

1.2 Numerical Linear Algebra (NLA) kernels

Numerical Linear Algebra kernels (NLA) are at the heart of all computational problems. These require hardware acceleration for increased throughput as demanded by the applications like high resolution direction finding, state estimation, adaptive noise cancellation etc.

Algorithm 1.2.1: MATRIX-VECTOR MULTIPLICATION(**c=Ab**)

for $i \leftarrow 1$ **to** N

do $\begin{cases} c[i][j] = 0; \\ \textbf{for } j \leftarrow 1 \textbf{ to } N \\ \quad \textbf{do } \{ c[i][j] = c[i][j] + A[i][j] * b[j]; \end{cases}$

During realization of these NLA kernels on a multiprocessor platform or an array architecture the key aspects that should be considered are:

- **Maximum parallelism**: Two algorithms with equivalent performance in a sequential computer may perform differently in parallel processing environments. An algorithm will be favoured if it expresses a higher parallelism, which is exploitable

by the computing arrays. For example Algorithm 1.2.1 for Matrix-Vector multiplication can be unrolled and realized on an unfolded hardware composed of $N \times N$ multipliers and N number of N input adders. Thus maximum parallelism can be achieved resulting in the best performance.

- **Maximum pipelinability**: Most NLA kernels demand very high throughput and are computationally intensive (as compared with their I/O requirements). The exploitation of pipelining is often very natural in regular and locally connected networks; therefore, a major part of concurrency in systolic array processing will be derived from pipelining. To maximize the throughput, we must select the implementation scheme that ensures optimum performance in the context of REDEFINE. Effective and optimum implementation should be highly pipelined and hence require well structured realization of algorithms with predictable data movements. If due to resource constraint "maximum parallelism" becomes luxury only by dint of pipelining we can still maintain a decent throughput. The presence of both enables us to process multiple kernels with peerless performance. We can visualize these points using Algorithm 1.2.1 as a simple case-instance.

- **Balance between computations and communications and memory**: A good realization should offer a sound balance between different bandwidths incurred in different communication hierarchies to avoid data draining or unnecessary bottlenecks. Balancing the computations and various communication bandwidths is critical to the effectiveness of array computing. In Algorithm 1.2.1 the efforts spent in streaming out the elements of the matrices A and b should be as low as possible. We have to ensure that the time spent in computation should not be overshadowed by the time consumed by memory transaction and transportation. The matchless solution comes with the ability to hide the communication delays in the wrapper of computational delays (i.e overlapping of both the delays) and a smart and rapid memory transaction scheme.

- **Trade off between computation and communication**: To make the interconnection network practical, efficient, and affordable, regular communication should be encouraged. Key issues affecting the communication regularity include local versus global, static versus dynamic, and data-independent versus data-dependent interconnection modules. The criterion should maximize the trade-off between interconnection cost and throughput.

- **Numerical performance and quantization effects**: Numerical behavior depends on many factors, such as the word length of the computation platform, whether it is fixed point or floating point, the nature of the algorithm etc. As an example, a QR decomposition (based on Givens Rotation) is often preferred over an LU decomposition for solving linear systems, since the former has a more stable numerical behavior. The price, however, is that QR takes more computations than LU decomposition. In the context of REDEFINE this led us to decide what kind of number representation (fixed or floating) we will use in the core computational units of the platform and along with that what should be the precision depending upon the application domain. However, the trade off between computation and numerical behavior is very algorithm dependent and there is no general rule to apply.

1.3 Problems of Systolic Array Solutions - Rigid Structure

Systolic arrays provide fast solutions to problems with regular iterative algorithms. Beacause of their regular structure which is algorithm specific, design methodology wise systolic arrays are scalable, though not in actual hardware. We can term these solutions to be ASIC (Application Specific Integrated Circuit) solutions. But the main adversity with application specific systolic arrays is its rigid structure. In spite of the obvious benefits the current technology trend is towards a paradigm shift from ASIC solutions. ASICs are way below to pave the way meeting the issues/challenges to the changing demands. While maintaining space and cost advantages we can make a “just right” Systolic Array/ASIC

with parametric adjustment capability. But commercial ASICs are designed keeping in mind that they should meet their generic feature in their specificity. If the ASIC is optimized for one particular design, it is a custom IC, of little use for other applications. If it is too general, it is likely to be too suboptimal to be feasible. Reduction of package size and cost by reducing pin count of a custom IC results in subsequent reduction in I/O bandwidth and observability. Being very specific ASICs have very short shelf life or are useful for point solutions. One fixed systolic array can be used for the fixed algorithm with the fixed problem size. Though the systolic algorithm ensure scalability but the monolithic ASIC is incapacitated to meet that need. This makes the user very handcuffed. For example in Numerical Linear Algebra application domains like signal processing, Kalman Filtering, computational finance, materials science simulations, structural biology, data mining, bioinformatics, fluid dynamics etc. there is a constant need for computations that deal with matrices of different sizes. In the same applcation domain also the matrices containing the data sets can be of different sizes for different application instances. A systolic array, designed to solve a Numerical Linear Algebra problem of 20×20 matrix size can neither be used for bigger size matrices nor smaller size matrices. In short custom ASIC solutions can not empower us with the license of hardware consolidation i.e a generalized solution in a specialized domain with the added advantage of scalability and a warranty for required throughput. Besides, in VLSI industry there is a cost called Non-recurring engineering cost, that must be paid immaterial of the volume. continually in order to maintain production of a product. Non-recurring engineering (NRE) refers to the one-time cost of researching, developing, designing, and testing a new product. When budgeting for a project, NRE must be considered in order to analyze if a new product will be profitable. The NRE cost associated with VLSI systolic arrays can not be amortized over low volumes.

1.4 Need for Reconfigurable Solutions

In the world of computing two kinds of traditional solutions are very popular. One is computation performed by a General purpose processor (GPP) and the other is application

Architecture	General Purpose Processor	ASIC	Reconfigurable
Resources	Fixed	Fixed	Configware
Algorithms	Software	Fixed	Flowware
Performance	Low	High	Medium
Cost	Low	High	Medium
Power	Medium	Low	Medium
Flexibility	High	Low	High
Computing Model	Mature	Mature	Immature
NRE Cost	Low	High	Medium
Design Cost	High	High	High
Productivity Gap	Low	High	Low
Time to Market(TTM) Cost	Low	High	Low

Table 1.1: Comparison of Representative Computing Architectures

specific computation performed by ASICs as mentioned in the previous section.

Enabled by the powerful tool of programmability any computing task can be solved by a general-purpose processor or GPP. Being a single common piece of silicon platform the applications hosted by GPP are rendered cheaper due to economics of scale for the production of a single integrated circuit. The most prominent feature that favours GPP platforms is their flexibility.

An ASIC, a unique function solution provider delivers high performance and low power but due to its fixed architecture ASICs are anemic in terms of flexibility and lowering NRE cost.

As a trade-off between two extreme characteristics of GPP and ASIC, reconfigurable computing has combined the advantages of both. A comparison of the three different architectures is given in Table 1.1.

From Table 1.1, we observe that reconfigurable computing has the combined advantages of configurable computing resources, called *configware* [], as well as configurable algorithms, called *flowware* []. Further the performance of reconfigurable systems is better than general-purpose systems and the cost is less than that of ASICs. Reconfigurable platforms entrust us with the power of hardware consolidation. Only recently the power consumption of reconfigurable systems has been improved such that it is now either comparable with ASICs or even smaller due to hardware consolidation. The main advantage of the reconfigurable system lies in its high flexibilty, while its main restraint is the lack of a standard computing model. The design effort in terms of NRE cost i.e the chip fabrication cost is in between that of general-purpose processors and ASICs. The other two

axes of direct costs are Design Cost and Productivity Gap. The design cost crops up from the efforts encountered while developing the application and envisioning the architecture. For Reconfigurable platforms the application development cost is same as that of a GPP. Though design of the architecture brings in a high cost for the first time, but it amortizes with multiple applications to be accomodated by the platform. Use of compilers helps in transforming the circuit description from one level of abstraction to a lower level, usually towards physical implementation. Thus, GPPs and Reconfigurable Platforms bridge the Productivity Gap which creates a lacuna between design complexity and design capacity in case of ASICs. Reconfigurable Platforms also can be seen as viable vehicles towards reducing the Time-to-Market costs.

There exists systolic array solutions for NLA kernels. While such custom hardware solutions for NLA Solvers can deliver high performance, they are not scalable. In our work, we show how NLA kernels can be realized on REDEFINE [], a runtime reconfigurable hardware platform. The two kernels we use as running example are Modified Faddeevs Algorithm [] and QR decomposition using Givens Rotation []. REDEFINE is a Coarse Grained Reconfigurable Architecture combining the flexibility of a programmable solution with the execution speed of an ASIC. The solution proposed here is capable of emulating systolic arrays over a wide variety of NLA problem sizes. In REDEFINE Compute Elements are arranged in a honeycomb topology connected via a Network on Chip (NoC) called RECONNECT, to realize the various macro-functional blocks of an equivalent ASIC. Architectural details of REDEFINE are presented in subsequent sections. We propose a few enhancements to improve the performance of REDEFINE in the context of NLA kernels. Along with the actualization details of the afore-mentioned kernels we explore the design space of the proposed solutions. These can be treated as specific examples for the realization of all decomposition type algorithms. We show how REDEFINE meets both the scalability and performance requirements of NLA kernels. We further show the scalability of the architecture by taking increasing problem sizes but keeping performance advantage almost constant.

1.5 Our Contribution

In this thesis we present how the traditional systolic solutions for NLA kernels can be re-targeted for realization on REDEFINE, a runtime reconfigurable platform with appropriate mapping of the nodes of the systolic array. REDEFINE is a coarse grain reconfigurable architecture, where the elementary schedulable unit is HyperOp [**?**]. HyperOps are a subgraph of the application dataflow graph comprising a set of elementary operations that have strong producer-consumer relationship. In REDEFINE, an application specified in a high level language C is compiled into HyperOps. Each HyperOp contains the meta-data that specifies its computation and communication requirements. Configuration information captured in the meta-data is generated statically by the compiler. Hardware resources in the REDEFINE fabric are dynamically provisioned for HyperOps executed at runtime. Application synthesis in REDEFINE follows a compilation process in which an application specified in C is translated into a dataflow graph as an intermediate representation. Subgraphs of this Dataflow graph form HyperOps. HyperOps are coarse grained application substructures that are staged for execution on REDEFINE following a data driven schedule. In order to exploit instruction level parallelism hyperOps are further divided into partitioned hyperOps, pHyperOps in short. pHyperOps contain the compute and transport metadata capturing the computation and communication requirements of the application. Hence, the compilation process [?] is divided into the various phases i.e, **Formation of DFG**, **HyperOp formation**, **Tag generation**, **Mapping HyperOps** and **Formation of Custom Instructions**. Detailed descriptions of the compilation process are available in [?]. From the dataflow graph HyperOps and pHyperOps are created for data driven execution in the CEs [?] maintaining some semantics. But the main problem associated with this is that the HyperOps formations are algorithm agnostic. The same is true when the compiler passes through the mapping phase. Hence, for certain algorithms eg. NLA kernels, this generic approach of HyperOp creation and mapping does not culminate into the achievable optimum outcome. The aim of the work presented here is to obtain a theoretical basis to enable algorithm aware HyperOp creation, and arriving at pHyperOps that can be optimally mapped to CEs. We

take the systolic array solutions mostly realized on mesh topology as our source graph and map them on a target graph of honeycomb topology. We partition the whole array into multiple sub-arrays (refer figure 1.1) and call them HyperOps. Depending upon the size of the sub-arrays computational resources are assigned to them. We determine the right size of the sub-array in accordance with the optimal pipeline depth of the core execution units (CEs) and the number of such units to be used per sub-array. Such a solution will allow emulation of systolic structures on REDEFINE ushering the way for optimal performance.

1.6 Thesis Overview

This thesis has been organised as follows:

Chapter 2 builds the foundation stone of systolic computing paradigm. Then the chapter reviews the specific systolic algorithms that we have realized on REDEFINE. The two algorithms discussed here are Modified Faddeevs Algorithm (Direct Solver) and QR Decomposition (QRD) using Givens Rotation. The benefits of QRD over LU Decomposition is also highlighted here.

Chapter 3 presents the overall architecture of REDEFINE framework.

Chapter 4 advocates QR and other NLA-specific enhancements to REDEFINE in order to meet expected performance goals.

Chapter 5 traces the realization details of Systolic Architectures onto REDEFINE. Here we propose the framework for algorithm aware HyperOp, generation of their partitions into pHyperOps for desired mapping on a set of CEs. We further do design space exploration of the contemplated solution. We present the theoretical results also to make a fair performance comparison of the solution to that of an GPP.

Chapter 6 manifests the detailed hardware architecture of the common core computational units of REDEFINE. The synthesis results are also reported.

Chapter 7 concludes the thesis with avenues for further work.

Chapter 2

Systolic Algorithms

Most of the algorithms used in signal and image processing exhibit features like localized operations, intensive computation and matrix operations. The design approach of special-purpose signal and image processing array processors completely relies on the exploitation of these common features of the algorithms. Expression and transformation of this special class algorithms play an important role in the initial phase of design. For parallel and pipeline processing algorithm expression provides the foundation stone for realization of a more systematic and formal description such as a dependence graph. Among many efforts towards developing a formal description of the space-time activities in array-processors[HTKun78], [Chen83] the most natural approach is to describe the actual space-time activities in terms of snapshots that display data activities at a particular time instant.

In this chapter we talk about the main considerations in providing a formal and powerful description(expression) of any algorithm, the systematic method to transform an algorithm description to an array processor and how to optimize the performance of those parallel algorithms realized on the arrays. Detailed descriptions are given in [Kung]. For reader's convenience the useful stuffs have been reproduced here in a nutshell.

2.1 Parallel algorithm Expression

Parallel algorithm expressions may be derived by two approaches:

- Vectorization of sequential algorithm expressions
- Direct parallel algorithm expressions, such as snapshots, recursive equations, parallel codes, single assignment code, dependence code, dependence graphs and so on.

2.1.1 Vectorization of Sequential Algorithm Expressions:

High level languages like C provide concise algorithm expression and have been used as machine independent programming tools. Programming in these sequential languages requires the decomposition of an algorithm into sequence of steps, each of which performs an operation on a scalar object. For example, consider a mathematical expression of the matrix addition $C = A + B$:

$$C(i,j) = A(i,j) + B(i,j) \ \ \forall \ \ i \ \ and \ \ j \tag{2.1}$$

The corresponding pseudo-code for C code can be written as

Algorithm 2.1.1: MATRIX-MATRIX ADDITION(**C=A+B**)

for $i \leftarrow 1$ **to** N

do { **for** $j \leftarrow 1$ **to** N

do { $C[i][j] = A[i][j] + B[i][j]$;

Here the elements of A and B are accessed in column major order, which by definition, the order in which they are stored. Many computers may not be able to execute the program as efficiently if the order is reversed. In this example, as no ordering is required by the algorithm, it is unwise to encode an ordering in the program.

If no ordering is encoded, the compiler may choose the most efficient ordering for the target computer. Moreover, should the target computer contain parallelism, then some or

all of the operations may be performed concurrently, without analysis or ambiguity. Since ordering is unavoidable when using sequential code, parallel expression of an algorithm is very desirable.

2.1.2 Direct Expressions of Parallel Algorithms:

Extracting the inherent concurrency(parallel and pipeline) of any given program may not be always done effectively by a vectorizing compiler. Hence, it is advantageous that a user/designer use parallel expressions to describe an algorithm in the first place. This is the key step leading to an algorithm-oriented array processor design. Many different expressions may be used to represent a parallel algorithm, including snapshots, recursive algorithms with space time indices, parallel codes, dependence graphs(DGs), or signal flow graphs(SFGs).

Single Assignment Code: A single assignment code is a form where every variable is assigned one value only during the execution of the algorithm.

Recursive Algorithms: A convenient and concise expression for the representation of many algorithms is to use recursive equations. The recursive equation for the matrix-vector multiplication $c = Ab$ is:

$$c_i^{(j+1)} = c_i^{(j)} + a_i^j b_i^{(j)} \ \ \forall \ \ i \ \ and \ \ j \tag{2.2}$$

where j is the recursion index, $j = 1, 2, \cdots, N$, and

$$c_i^{(1)} = 0 \tag{2.3}$$

$$a_i^{(j)} = A(i, j) \tag{2.4}$$

$$b_i^{(j)} = B(j) \tag{2.5}$$

A recursive equation with space-time indices uses one index for time and the other indices for space. By doing so, the activities of a parallel algorithm can be adequately expressed. The preceding equation can be viewed as a recursive equation with the j-index

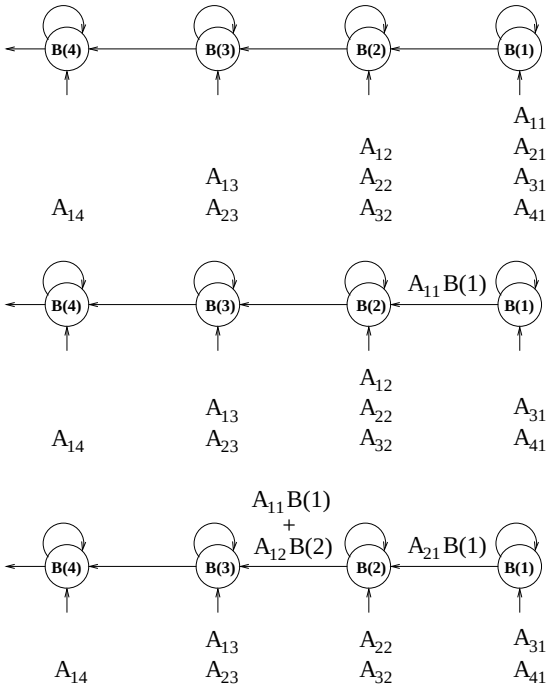


Figure 2.1: Snapshots for a systolic matrix-vector multiplication algorithm

as the time index and the i-index as the space index. A recursive algorithm is inherently given in a single assignment formulation.

Snapshots: A snapshot is a description of the activities at a particular time instant. Snapshots are perhaps the most natural tool an algorithm-array designer can adopt to check or verify a new array algorithm. Sample snapshots for a systolic matrix vector multiplication are depicted in figure 2.1

Dependence Graph: A dependence graph is a graph that shows the dependence of the computations that occur in an algorithm. A DG can be considered as the graphical representation of a single assignment algorithm. In the previously-mentioned algorithm, $C(i, j+1)$ is said to be directly dependent upon $C(i, j)$,$A(i, j)$ and $B(j)$. By viewing each dependence relation as an arc between the corresponding variables located in the index space, a DG as shown in figure 2.2, will be obtained. The operations inside each node are deliberately ignored in the DG, since they will be assigned to the same processing element when the DG is used to map an algorithm to an array-processor. An algorithm

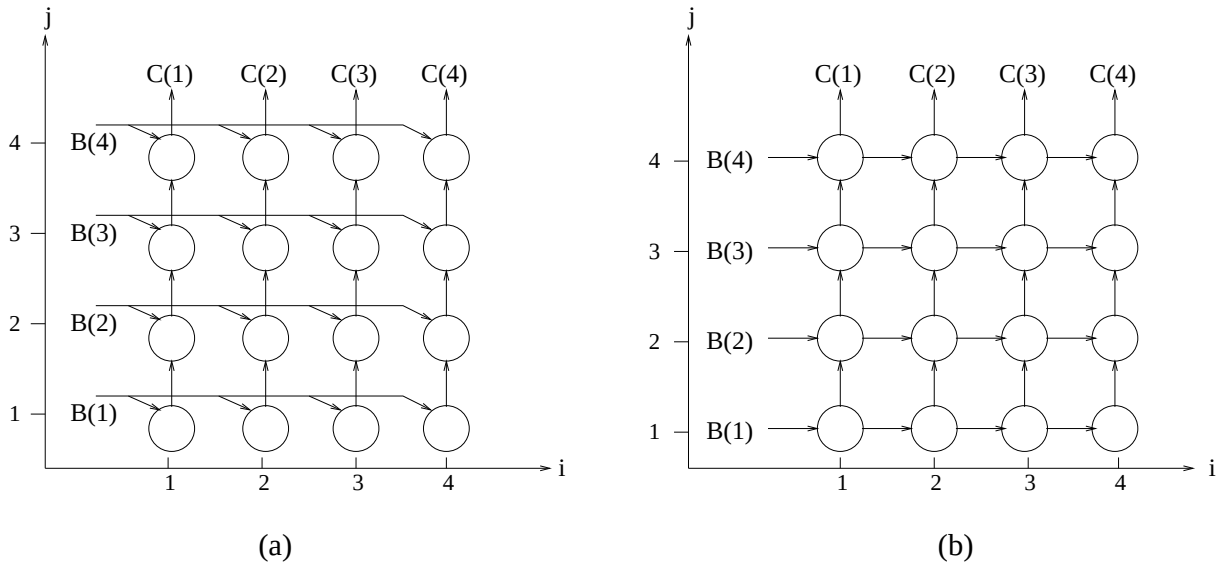


Figure 2.2: DG for matrix-vector multiplication (a) with global communication; (b) with only local communication.

is computable if and only if its complete DG contains no loops or cycles. Since the data dependencies are explicitly expressed in the dependence graph, a systematic approach to derive an array processor implementation by using such regular DGs is possible [Moldo83], [Miran84].

2.1.3 Graph Based Design Methodology

Stage1 - DG Design: After identification of a suitable algorithm for a given problem the user generates a DG for the algorithm expression. Since the structure of the DG greatly affects the final array design, further modification on the DG are often desirable in order to achieve a better design.

Stage2 - SFG Design: Based on different mappings of the DG onto array structure, a number of Signal Flow Graphs can be defined from the DG. The SFG offers a powerful abstraction and graphical representation for problems in scientific and signal processing computations dealing with NLA kernels. The SFG expression, which consists of processing nodes, communicating edges and delays, is shown in figure 2.3. In general, a node is often

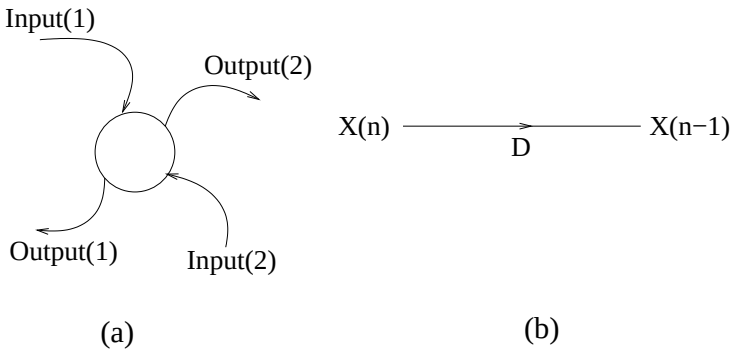


Figure 2.3: SFG Notations: (a) an operation node; (b) an edge as a delay operator.

denoted by a circle representing an arithmetic or logic function performed with zero delay, such as multiply and add. An edge, on the other hand, denotes either a dependence relation or a delay. When an edge is lebeled with a capital letter D, it represents a time delay operator with delay time D. The SFG can be viewed as a simplified graph, a more concise representation than the DG. As SFG is more closer to hardware level design it dictates the type of arrays that will be obtained.

Stage3 - Array Processor Design: The SFG obtained in stage2 can physically be realized in terms of a systolic array. As mentioned earlier a systolic array is a network of processors which rhythmically compute and pass data through the system. A systolic array often represents a direct mapping of computations onto processor array. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network[HTKung78]. For example, it is shown in [HTKung78] that some basic "inner product" Processing Elements(PEs) - each performing the operation $Y \leftarrow Y + A.B$ can be locally connected together to perform digital filtering, matrix multiplication, and other related operations. In general, the data movements in a systolic array are prearranged and are described in terms of the "snapshots" of the activities.

2.1.4 Processor Assignment and Scheduling

There are two basic considerations for mapping from a DG to an SFG:

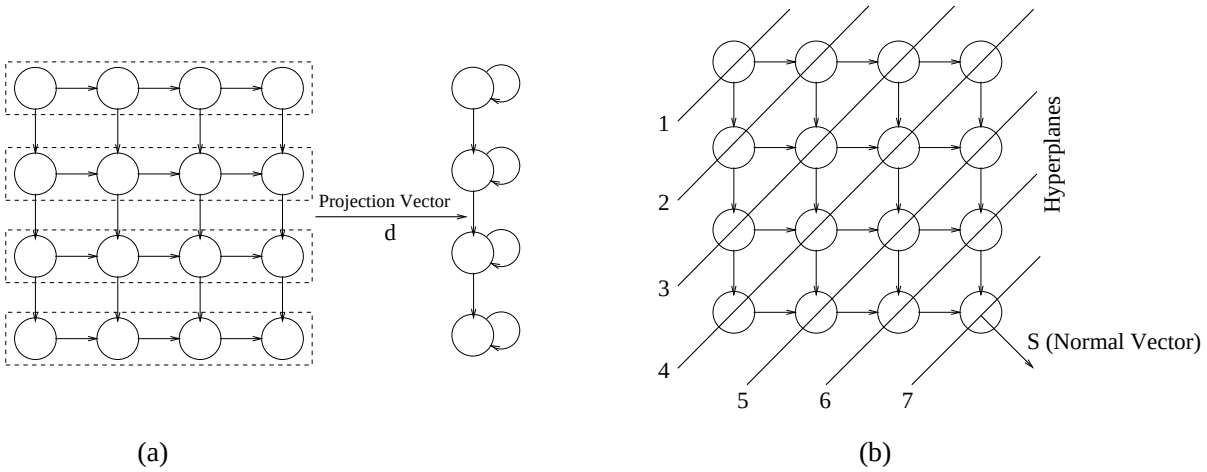


Figure 2.4: Illustration of (a) a linear projection with projection vector d; (b) a linear schedule s and its hyperplanes.

- To which processors should operations be assigned? (A criterion for example might be to minimize communication/exchange of data between processors.)
- In what ordering should the operations be assigned to a processor? (A criterion might be to minimize total computing time.)

It is common to use a linear projection for processor assignment, in which nodes of the DG in a certain straight line are projected(assigned) to a PE in the processor array,(refer figure 2.4), and a linear scheduling, in which nodes in a parallel hyperplane in the DG are scheduled to be processed at the same time step(see figure 2.4).

Processor Assignment: As a simple example, a projection method may be applied, in which nodes of the DG along a straight line are assigned to a common PE. If the DG of an algorithm is very regular, the projection maps the DG onto a lower dimensional lattice of points, known as the processor space. Mathematically, a linear projection is often represented by a projection vector *d*. The results of this projection is represented by the SFG.

Scheduling: Scheduling scheme specifies the sequence of operations in all the PEs. A schedule function represents a mapping from the N-dimensional index space of the DG onto a 1-D schedule(time) space. A linear schedule is based on a set of parallel

and uniformly spaced hyperplanes in the DG. These hyperplanes are called equitemporal hyperplanes, all the nodes on the same hyperpalne must be processed at the same time. Mathematically, the schedule can be represented by a (column) schedule vector s, pointing to the normal direction of the hyperplanes.

Permissible Linear Schedule: Given a DG and a projection direction d, we note that not all the hyperplanes qualify to define a valid schedule for the DG. In order for the given hyperplanes to represent a permissible linear schedule, it is necessary and sufficient that the normal vector s satisfies the following two conditions:

$$\overrightarrow{s}^T \overrightarrow{e} \geq 0, \quad for \ \ any \ \ dependence \ \ arc \ \ \overrightarrow{e}. \tag{2.6}$$

$$\overrightarrow{s}^T \overrightarrow{d} > 0. \tag{2.7}$$

Both the conditions 2.6 and 2.7 can be checked by inspection. In short, the schedule is permissible if and only if

- all the dependency arcs flow in the same direction across the hyperplanes and
- the hyperplanes are not parallel with the projection vector $\overrightarrow{d}$.

The first condition means that a causality should be enforced in a permissible schedule. Namely, if node p depends on node q, then the time step assigned for p can not be less than the time step assigned for q. The second condition implies that nodes on an equitemporal hyperplane should not be projected to the same PE.

2.2 Systolic Solutions for Numerical Linear Algebra kernels

Application domains such as Bio-informatics, DSP, Structural Biology, Fluid Dynamics etc. demand high performance computing solutions for their simulation environments.

The core computations of these applications is in Numerical Linear Algebra (NLA) kernels. These kernels need to be executed taking the nature of the target application into consideration. Direct solvers are predominantly required in the domains like DSP, estimation algorithms like Kalman Filter etc, where the matrices on which operations need to be performed are either small or medium sized, but dense. Here in this section we show how Faddeevs Algorithm can be used as a direct solver. We further talk about QR Decomposition of any matrix, often used to solve the linear least square problem. Systolic realizations of both the kernels are presented.

2.2.1 Faddeeves Algorithm

Faddeev's Algorithm (FA) [**?**] is used for solving dense linear system of equations. Faddeeves Algorithm [**?**] enables us to compute the Schur complement of a compound matrix M (size n×n) composed of four matrices A, B, C, D of sizes (n×n), (n×l), (m×n), (m×l) respectively, provided A is non-singular [**?**]. A variant of this algorithm that is amenable for realization in hardware was proposed by Nash et al. [**?**]. This is referred to as the Modified Faddeev's algorithm (MFA). Calculation of Schur complement $[D + CA^{-1}B]$ using MFA, which in effect, is a two step process i.e trinagularisation of matrix A and nullification of the elements of matrix C [**?**].

$$\text{Let } M = \begin{bmatrix} A & B \\ -C & D \end{bmatrix}$$

The Schur Complement of M is given by,

$$E = D + CA^{-1}B \tag{2.8}$$

The representation of E in matrix form is as follows (for a typical case of 2×2):

$$\begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix} = \begin{bmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{bmatrix} + \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^{-1} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \tag{2.9}$$

Systolic array with their regular lattice structure provides a good parallel platform to realize the calculation of Schur Complement in hardware. For systolic realization of MFA, the desired lattice is a mesh interconnection of CEs. In subsequent sections we will see how REDEFINE can provide a reconfiguarable and scalable solution for the calculation of Schur Complement using MFA.

2.2.2 Algorithm in a nutshell

To illustrate Faddeev's algorithm consider the simple case of computing:

$$C_1X_1 + C_2X_2 + C_3X_3 + \cdots\cdots + C_nX_n + d \tag{2.10}$$

where C_1, C_2, C_3 $\cdots$ C_n are given numbers, and X_1, X_2, X_3 $\cdots$ X_n are the solution to the linear system of equations

$$\begin{aligned}
a_{11}X1 + a_{12}X2 + a_{13}X3 + \cdots\cdots + a_{1n}Xn &= b1 \\
a_{21}X1 + a_{22}X2 + a_{23}X3 + \cdots\cdots + a_{2n}Xn &= b2 \\
a_{31}X1 + a_{32}X2 + a_{33}X3 + \cdots\cdots + a_{3n}Xn &= b3 \\
&\cdots\cdots \\
&\cdots\cdots \\
a_{n1}X1 + a_{n2}X2 + a_{n3}X3 + \cdots\cdots + a_{nn}Xn &= bn
\end{aligned} \tag{2.11}$$

which is not singular. The above equations can be reformulated as in figure 2.5

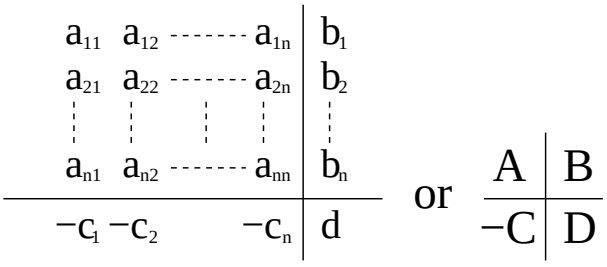


Figure 2.5: Faddevs Algorithm deals with an augmented matrix of four different matrices

Where B is a column vector and C is a row vector. If a suitable linear combination of the rows above the line (from A and B) are added to the rows beneath the line (e.g. $-C+WA$ and $D+WB$ where W specifies appropriate linear combination), so that only zeroes appear in the lower left hand quadrant, then the desired result, $CX+D$ will appear in the lower right quadrant. This follows because the annulment of the lower left hand quadrant requires that

$$W = CA^{-1} \tag{2.12}$$

so that

$$D + WB = D + CA^{-1}B \tag{2.13}$$

Since, $X = A^{-1}B$, we have the final result

$$D + WB = D + CX \tag{2.14}$$

Identification of the multipliers of the rows of A and elements of B is not required; it is only necessary to annul the last row. This can be done by ordinary Gaussian elimination. The triangularization of matrix A is done as traditional LU Decomposition. A brief mathematical insight of LU Decomposition is elucidated in the next section. An important feature of this algorithm is that it avoids the usual back substitution solution to the triangular linear system and obtains the values of the unknowns directly at the end of the forward course of computation, resulting in considerable savings in processing and storage. Statistical studies have shown that the numerical accuracy is comparable to the usual LU decomposition and back substitution. This result can be generalized in case of rectangular matrices C, D and B. After the lower left hand quadrant is annulled, the result $CA^{-1}B + D$ will appear in the lower right hand quadrant. The numerous matrix operations possible by selective entries in the four quadrants are as shown in figure 2.6).

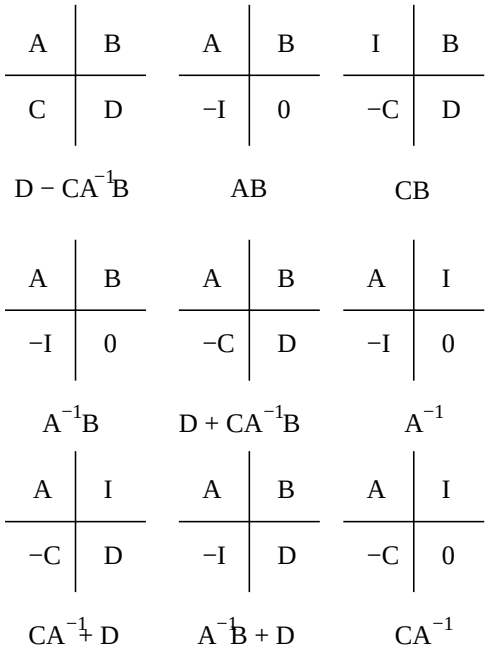


Figure 2.6: Different possible Matrix-Solutions using MFA

Nash and Hassan [?] have modified Faddeev's algorithm (MFA) by introducing orthogonal factorization capability. This leads to more numerical stability. We adopts the MFA algorithm in our work. Different possible results could be obtained by feeding different matrices in place of A, B, C and D. Each result has two or more matrix operations combined together into a single operation. Moreover, matrix inversion is straightforward. These properties can be exploited to reduce he computation involved in the Kalman filter [**?**] equations. Computational steps in these equations [?] (refer figure 2.7) can be decomposed into many sub tasks each of which can be executed in a step using Faddeev's algorithm.

2.2.3 LU Decomposition

Let A be an $n \times n$ square matrix. A can be decomposed into unit lower triangular and upper triangular matrices [] as shown below

$$A = LU \tag{2.15}$$

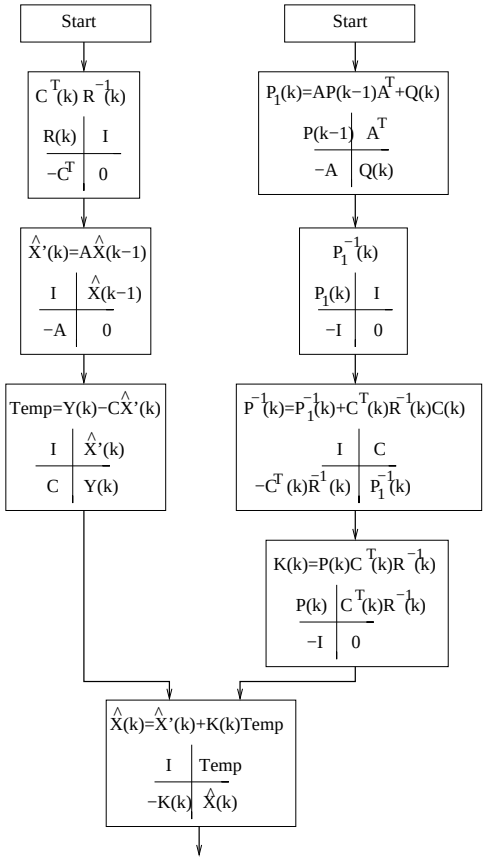


Figure 2.7: Representation of parallel Computational steps in Kalman Filter using Faddeevs Algorithm

where L and U are lower and upper triangular matrices (of the same size i.e $n \times n$) respectively. For a 3×3 matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & 0 \\ u_{31} & 0 & 0 \end{bmatrix} \tag{2.16}$$

Upon multiplying the two matrices L and U get,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} + u_{21} & l_{21}u_{12} + u_{22} & l_{21}u_{13} \\ l_{31}u_{11} + l_{32}u_{21} + u_{31} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} \end{bmatrix} \tag{2.17}$$

Hence by comparing the matrices on element by element basis we get,

$$u_{11} = a_{11}, \;\; u_{12} = a_{12}, \;\; u_{13} = a_{13} \tag{2.18}$$

$$l_{21} = \frac{a_{23}}{u_{13}}, \;\; l_{31} = \frac{a_{33}}{u_{13}} \tag{2.19}$$

$$u_{22} = a_{22} - l_{21}u_{12}, \;\; u_{21} = a_{21} - l_{21}u_{11} \tag{2.20}$$

$$l_{32} = \frac{a_{32} - l_{31}u_{12}}{u_{22}}, \;\; u_{31} = a_{31} - l_{31}u_{11} - l_{32}u_{21} \tag{2.21}$$

If we observe with attention we can form two generalized equations to get the non-zero elements of the lower (L) and upper (U) triangular matrices:

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}}{u_{jj}} \tag{2.22}$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \tag{2.23}$$

The elements of the U and L matrix are uniquely determined on applying the above mentioned equations in the correct order.

2.2.4 Systolic Array realization

The trapezoidal array illustrated in figure 2.8. is the most popular systolic array implementation of the Faddeevs algorithm. If the input matrices are of size $n \times n$, then the Systolic array is made up of a triangular segment i.e sub-array TRIAN and a rectangular segment i.e sub-array RECTAN. These two sub-arrays contain $n(n-1)/2$ and n^2 number of Processing Elements (PEs), respectively. There are two types of PE : Diagonal and Off-diagonal PE. The input-output signatures of the two kinds of PEs are shown in figure 2.8. As shown in the figure 2.8, the elements of matrix A and B are first fed to the sub-arrays TRIAN and RECTAN respectively but in a skewed manner. This skewing are achieved through delay cells. The elements of matrix A are triangularized in the sub-array TRIAN, then are stored in the PEs of that sub-array. At the same time, the factors for elementary row operations are fed to the right-hand sub-array RECTAN, and the same row elements of B encounter the same transformations, and stored back in the internal registers of the PEs of sub-array RECTAN. Continuing the flow elements of matrices C and D are fed to the triangular and rectangular segments of the trapezoidal array respectively. All the processing elements works in dual mode. `Mode 1` is for the sake of trinagularization of matrix A and subsequent operations on the elements of matrix B. In `mode 2` Processing elements are engaged in the job of nullifying the elements of matrix C and promoting the same elementary row operations on the elements of matrix D. the desired result matrix E are out through the bottom of the sub-array RECTAN [4].

2.2.5 QR Decomposition

A matrix, A, can be written as the product of a matrix with orthonormal columns and an invertible upper triangular matrix, that is, $A = QR$, where Q is a matrix with orthonormal

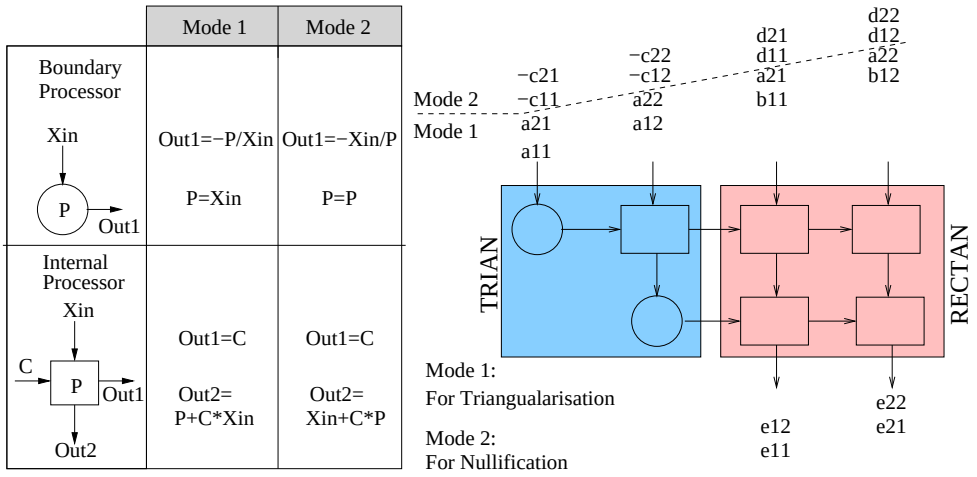


Figure 2.8: Operations of Diagonal processor and off-diagonal processor in a 2×2 systolic array.

columns and R is an upper triangular matrix.

2.2.6 QR Decomposition using Givens Rotation

This decomposition known as QR Decomposition (QRD), can be obtained by a sequence of Givens Rotations [?, ?]. In Givens Algorithm, Givens Rotation provides a numerically stable decomposition solution by plane rotation of the matrix A whose subdiagonal elements of the first column are nullified first, then the elements of the second column, and so forth untill an upper triangular form is eventually reached.

For an invertible matrix A, the upper triangular matrix R is obtained as follows:

$$Q^T A = R \tag{2.24}$$

$$Q^T = Q_{N-1} Q_{N-2} \cdots Q_1 \tag{2.25}$$

and

$$Q_p = Q^{p,p} Q^{p+1,p} \cdots Q^{N-1,p} \tag{2.26}$$

where $Q^{q,p}$ is the Givens Rotation (GR) operator used to annihilate the matrix element located at the $(q+1)^{st}$ row and p^{th} column and has the form as given in figure 2.9.

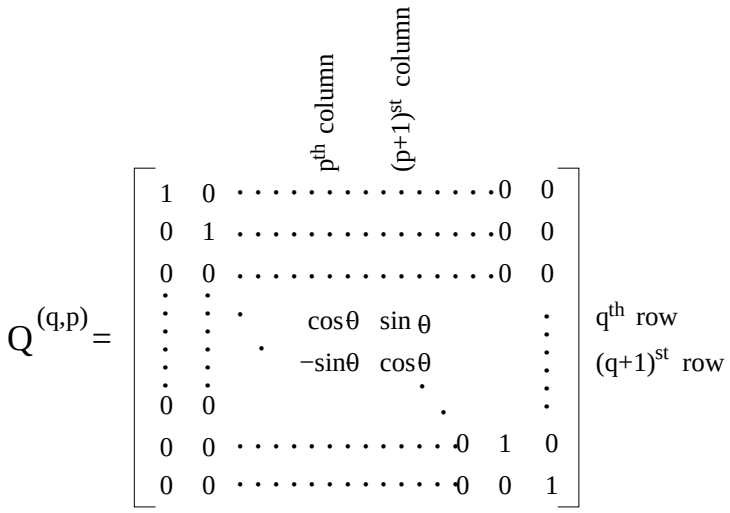


Figure 2.9: GR operations on rows of A

In the figure 2.9, $\theta = tan^{-1}[a_{q+1,p}/a_{q,p}]$ is an abbreviation of the function $\theta(q,p)$. The operation of creating $cos\theta$ and $sin\theta$ is named Givens Generation (GG).
The matrix product $A^{'} = Q^{q,p}A$ can be expressed as:

$$a^{'}_{q,k} = a_{q,k}cos\theta + a_{q+1,k}sin\theta \tag{2.27}$$

$$a^{'}_{q+1,k} = -a_{q,k}sin\theta + a_{q+1,k}cos\theta \tag{2.28}$$

$$a^{'}_{j,k} = a_{j,k} \quad if \quad j \neq q, q+1 \tag{2.29}$$

$$\forall\ k = 1 \cdots N.$$

The effects of GR operations on the q^{th} and $(q+1)^{st}$ rows of A are as follows:

$$\begin{bmatrix} a^{'}_{q,1} & a^{'}_{q,2} & \cdots & a^{'}_{q,N} \\ 0 & a^{'}_{q+1,2} & \cdots & a^{'}_{q+1,N} \end{bmatrix} = \begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} a_{q,1} & a_{q,2} & \cdots & a_{q,N} \\ a_{q+1,1} & a_{q+1,2} & \cdots & a_{q+1,N} \end{bmatrix} \tag{2.30}$$

The $sin\theta$ and $cos\theta$ parameters can be determined from the following equations:

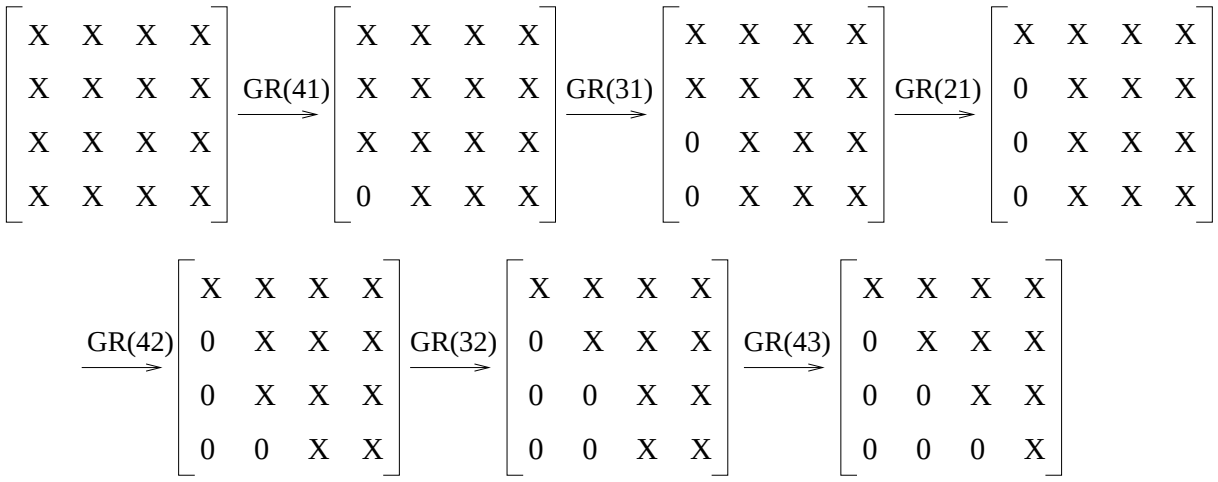


Figure 2.10: Example of Givens Rotation on a 4×4 matrix: Step by step procedure showing the nullification of lower elements and thus forming the right triangular matrix

$$cos\theta = a_{q,k}/\sqrt{a_{q,k}^2 + a_{q+1,k}^2} \quad (2.31)$$

$$sin\theta = a_{q+1,k}/\sqrt{a_{q,k}^2 + a_{q+1,k}^2} \quad (2.32)$$

The nullification of the lower triangular elements of a 4×4 matrix using GR is pictorially represented in figure 2.10.

2.2.7 Systolic array implementation

Triangular array or Gentleman-Kung array [**?**, **?**] is a very popular systolic array solution for QR factorization. Figure 2.11 shows the pictorial representation of the systolic structure where the GG operations are performed in the diagonal Processing Elements (PEs) and the GR operations are in all the other PEs. The diagonal PEs generates the Givens Rotation factors to be used by the elements of a particular row in the input matrix. These rotation angle parameters generated by the diagonal PEs are broadcast to all off-diagonal PEs in the same row. New values are updated and stored in the internal registers of the PEs when the off-diagonal PEs engage themselves in orthonormal transformations in each row using the data recieved from diagonal PEs. In essence, keeping harmony with the

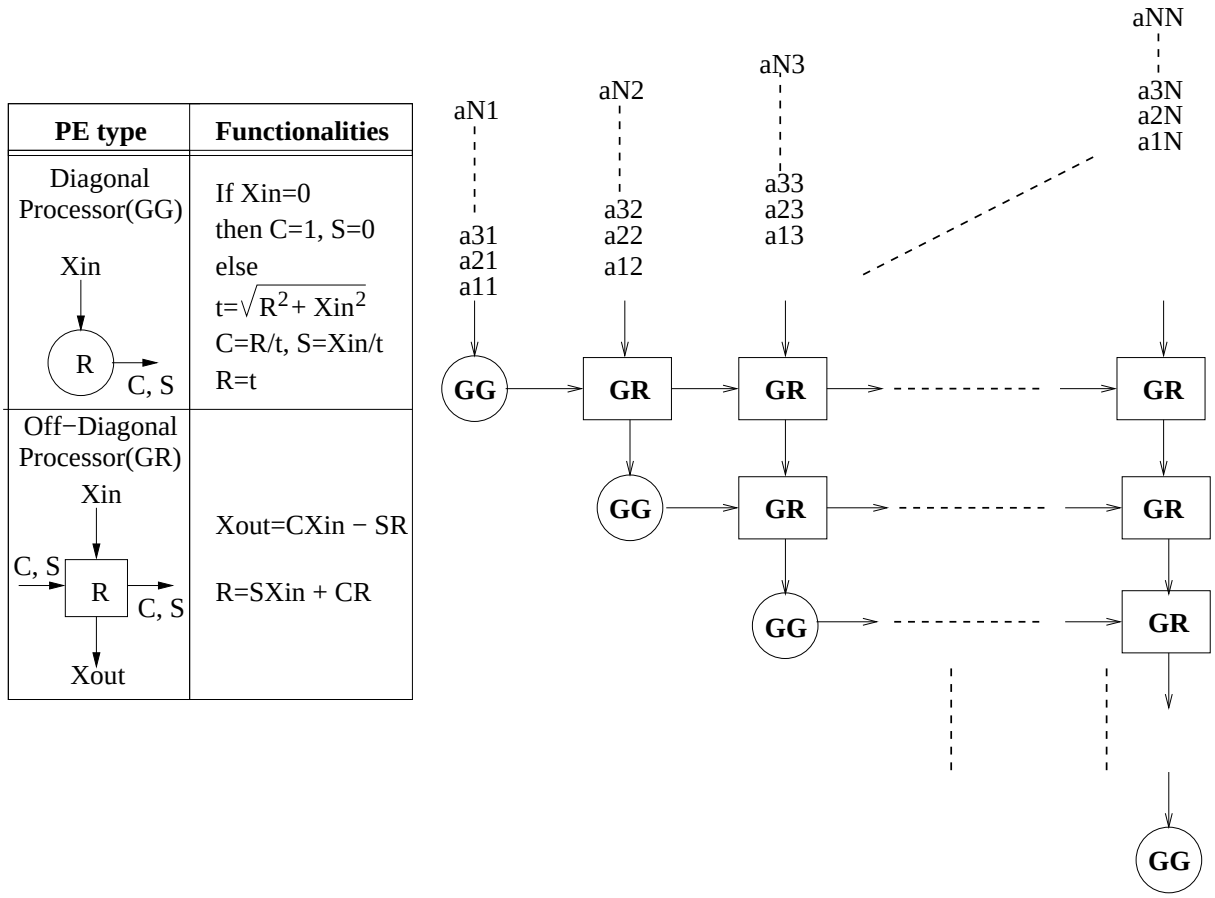


Figure 2.11: Functionalities of the Processing Elements (PEs) of the tri-array used as a basic module for performing the QRD

equations 2.27, 2.28 and 2.29 the rotation angles c and s are generated in the diagonal PEs and the remaining elements at the two rows of the input matrix are updated. These are done on per rotation basis.

The array used for factorization of a an matrix of size $n \times n$ is of a triangular shape with n rows. There is one diagonal element in each row. The array has $n-1$ off-diagonal PEs in the first row, $n-2$ off-diagonal PEs in the second row and so on, so forth. Out and out n diagonal PEs, $n(n-1)/2$ off diagonal PEs and $n(n+1)/2$ local internal memories are required. A typical $n \times n$ triangular systolic structure can be used to factorize any matrix of size $m \times n$ where $m \geq n$. For a $m \times n$ matrix where $n > m$ the array takes a trapezoidal representation with $n - m$ off-diagonal PEs in the last row while keeping the functionalities intact for the two sets of PEs.

Chapter 3

REDEFINE - Revisited

REDEFINE [?, **?**] is a Coarse grained reconfigurable architecture where diverse data-paths are composed of computational structures at runtime. By the term computational structure what we mean is a physical aggregation of hardware resources that can perform coarse grained operation, referred as a Hyper Operation (HyperOp). Here lies the most prominent difference between REDEFINE and FPGAs, where configurable logic blocks (CLBs) are SRAM based memory lookup tables (LUTs) used to define applications specific datapaths. On the contrary in REDEFINE computational structures define the application specific datapaths. As a consequence we get power advantage in case of REDEFINE.

In REDEFINE hardware resources on which the computations are done are organized on a fabric with honeycomb topology. Each computational unit, refferd as Tile is an embodiemnt of Compute Element (CE) with local storage and router. A network-on-chip (NoC) [**?**] called RECONNECT empowers the routers to communicate with each other. By philosophy REDEFINE follows a data-flow execution paradigm. Here the distributed NoC is used to establish the desired interconnections between the CEs on demand at runtime, supported by a dynamic dataflow execution paradigm. Management of the computational resources are done by support logic.

On a FPGA, while loading the configuration information bit level programming of the multiplexers of the interconnect is involved. It is also required to program the truth table

in each logic element i.e LUT/CLB. This type of configuration approach is the main deterrence against dynamic reconfigurability. MathStars Field Programmable Object Array (FPOA) [MathStar 2008] is a solution in which silicon objects can be interconnected in a manner similar to FPGAs. This enables FPOA to be used to support large computationally intensive applications. How- ever, they are not runtime reconfigurable and also share similar limitations as FPGA. In order to reduce the configuration overhead, we choose ALUs/FUs as opposed to Logic El- ements and replace the programmable interconnect with a NoC (refer [Joseph et al. 2008]). Unlike FPGA where applications are specified in RTL, in REDEFINE applications specified in a High Level Language (HLL) are compiled into coarse grained operations containing metadata which captures the computation and communication requirements. This information is used to compose computational structures at runtime. These distinctions of REDEFINE from FPGA solutions provide REDEFINE the application scalability and programmability that in turn reduces application development time significantly. [?] provides a quantitative comparison between REDEFINE and FPGA.

The proposed approach/methodology behind the realization of various applications on REDEFINE relies on a strong interplay between the microarchitecture and the compiler. REDEFINE is an embedded platform where RETARGET provides compiler tool chain support. The input to the compiler is an application developed in some HLL. RETARGET compiles any such application to an intermediate form and convert it into dataflow graphs [**?**]. These dataflow graphs are directed graphs of nodes where each node represents a HyperOp. A hyperOp is a directed acyclic subgraph of the entire application data-flow graph. Each HyperOp comprises multiple fine grained operations. In order to exploit instruction level parallelism that exists within a HyperOp (also due to storage limitation in a CE), each HyperOp is further divided into several partitions (pHyperOp) and each pHyperOp is assigned a CE. RETARGET captures the computation to be performed by each pHyperOp in terms of compute metadata and the inter/intra HyperOp communication in terms of transport metadata.

3.1 Micro-Architecture

In [?, **?**], the micro-architecture of REDEFINE was reported with details of the execution fabric including a high level description of the Support Logic to derive a dynamic dataflow execution schedule of dynamic instances of HyperOps. REDEFINE Figure 3.1 depicts the overall block diagram of REDEFINE architecture.

REDEFINE is a HyperOp execution engine, where HyperOps are atomically scheduled with no rollback. The computation power of the platform comes from the execution fabric that includes tiles connected by a NoC, called RECONNECT. The Support Logic comprises HyperOp Launcher (HL), Load Store Unit (LSU), Inter HyperOp Data Forwarder (IHDF), Hardware Resource Manager (HRM) and Resource Binder (RB). In [?], functional description of these modules is briefly provided. [?] covers the implementation details of the same.

The NoC RECONNECT that is proposed in [?], has a flat honeycomb topology and data can be injected through the tiles located at the boundary of the fabric by Express Lanes connected to the HL by a crossbar. However the Express Lane approach is not scalable due to increased complexity at the crossbar connecting the HL to the fabric and due to increase in wire length.

In the recent version of REDEFINE, the Express Lanes of REDEFINE have been replaced by 12 Access Routers (marked with `A` in figure 3.1) making each row and every alternate column toroidal. Two links are connected to the fabric transforming the flat topology into a toroidal honeycomb with 2 links left for modules of the Support Logic. This extension does not disturb the homogeneity of the fabric, but offers multiple well defined points for injection and ejection of operations and data with short distances to every node. The design of the Access Routers does not differ from the tile routers. In figure 3.1, a tile indicated by `T` comprises a CE and a router [?].

The exact CEs to which HyperOps need to be loaded, is determined by the RB, which maintains a list of idle CEs. The topology suitable for each HyperOp is generated by RETARGET in terms of a configuration matrix and is stored in the memory that is local to the RB. RB finds an appropriate loca- tion on the fabric to launch a HyperOp. This

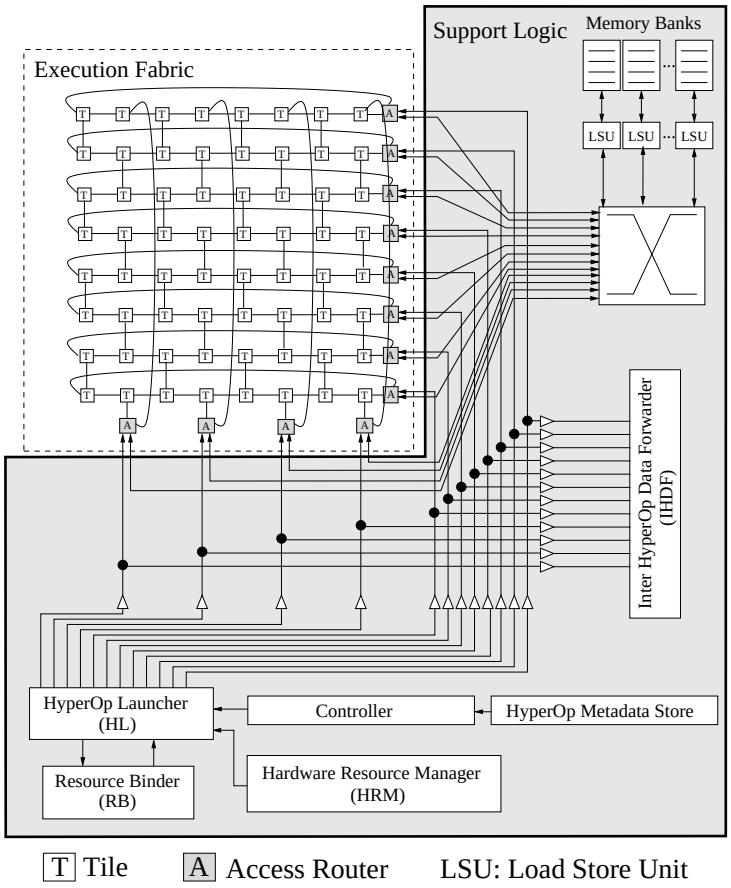


Figure 3.1: Architecture of REDEFINE

location is computed based on the availability of the CE and the topology required.

HyperOps are stored in the HyperOp Metadata Store realized as five different memory banks supporting burst mode read. The HL loads the compute and transport metadata from the HyperOp Metadata Store onto the CEs through the NoC. LSU is the conduit for servicing read/write request of global data to/from Memory Banks.

The compiler generates compute and transport metadata (refer to [**?**]). This metadata contains the compute and transport resource requirements of HyperOps and is used to determine the mapping of HyperOps onto tiles. Compute metadata captures the computational needs of the application and transport metadata makes the fabric aware of the communication requirements i.e internal and external interactions among HyperOps. It is the job of the HRM to identify "ready" HyperOps, arbitrate among them and launch them for execution. If a HyperOp is ready to be launched onto the fabric, they are sent

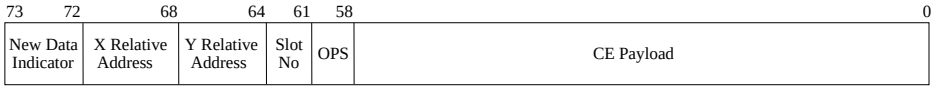
to the HyperOp Launcher. A HyperOp is ready to be launched when all the inputs of a HyperOp are available. HyperOp Selection Logic (HSL) is responsible for choosing one of the ready HyperOps for launching.

While within a HyperOp static dataflow execution paradigm is followed, across HyperOps a dynamic dataflow schedule is used [?], [?]. The Global Wait-Match Unit (GWMU) resident within the HRM, holds the HyperOps waiting for input operands. A result produced (due to computation by a CE) that is destined for a HyperOp which yet to be launched, is routed to IHDF, which in turn sends it to the HRM. Thus the IHDF facilitates communication across HyperOps recieving requests for an inter HyperOp data transfer. The IHDF accepts packets from Access Routers and is responsible for delivering the data to the appropriate dynamic instance of the destination HyperOp.

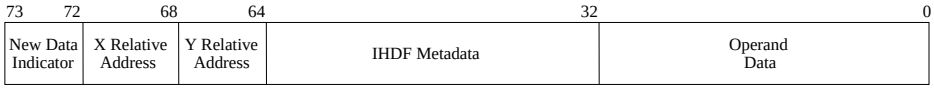
The execution fabric comprises tiles connected in the honeycomb topology. Each tile accomodates a Compute Element (CE) whose task is to execute instruction(s) and a router facilitating communication between tiles over the NoC. All communication between the fabric and modules of the Support Logic is handled by Access Routers. The CE payload packet is of three types i.e instruction packet, operand packet and predicate packet [?]. As shown in figure 3.2 the *OPS* field specifies the type of the payload. Metadata and operands are stored in a local storage referred as Local Wait Match Unit (LWMU). Instructions along with the transport metadata and operands are logically organised as slots in the LWMU. *SlotNo* field specifies the slot of the local storage within the CE. An operation is launched onto the ALU only when all the operands and predicate are available. Detailed architectural description of the CE can be found in [?, ?].

The implementation of router is as described in [?]. Each router in the fabric has four input and four output ports. Three are used to establish a connection to the neighboring routers and one is reserved for the CE itself. Only in case of Access Routers slight modifications are necessary. They have two connections to the neighboring ones, one to communicate to the Load/Store Unit bidirectionally, and one to establish a link to the HL and IHDF. Each router ensures in-order data delivery between source and sink.

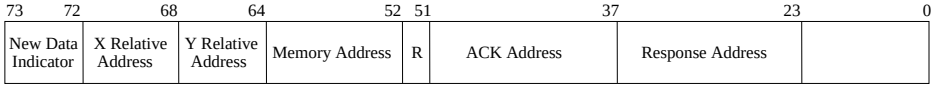
REDEFINE is an architecture in which different modules perform their respective



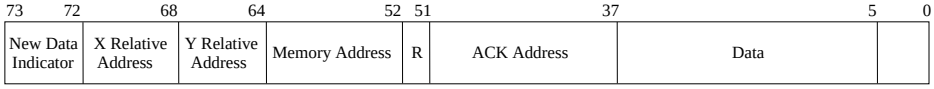
(a) CE Payload Packet



(b) IHDF Packet



(c) Load Request Packet



(d) Store Request Packet

Figure 3.2: Different packet formats handled by the tiles of the fabric

tasks depending on the information/packets they receive from other modules. In the following we take a packet-centric view of the architecture and describe the functionalities of various components. The largest packet determines the overall bus width among the various modules of the architecture. In our approach we align all information to the MSB and leave the unused fields unchanged to conserve switching power.

The Packet-Centric Execution Flow: As depicted in figure 3.2, there are different types of packets that are exchanged over the NoC. When a router receives a new packet, it is indicated by the NPI (New Packet Indicator) bit to distinguish a new incoming packet from a previous one that is still latched. After the packet is received, a simple store and forward routing algorithm decides to which tile/router the packet needs to be forwarded using the fields *X* and *Y Relative Address*. The remaining fields of the packets are ignored by the router. The following are the packets that are exchanged among various modules of REDEFINE-v2.a little mention

- Data and instructions for the CE are transmitted by the CE Payload Packet (figure 3.2(a)). The *Slot No* field defines the slot in the Local Wait Match Unit (see section **??**) to which the packet information is applied to. The *OPS* field distinguishes

among the type of the payload. Hence the CE Payload Packet can further be divided into:

  - The Instruction Packet corresponds to the operations in a HyperOp and the associated metadata. It carries the operation that needs to be executed including up to 3 destinations for the result of one instruction.
  - An Operand Packet provides a 32-bit operand value to an instruction.
  - In some cases operations of a HyperOp need to be terminated due to specific reasons (one of them could be a failed if or else branch for example). A packet in which the CE payload contains a predicate indicates such a packet.

- The IHDF Packet (figure 3.2(b)) is used to deliver results to HyperOps which are currently not mapped on the fabric, but are waiting in the Support Logic to become ready (i. e. all input values have arrived).

- To access the memory through the LSU, the packets shown in figure 3.2(c) and 3.2(d) are used to perform a LOAD or STORE operation respectively indicated by the *R* field (Request type). The packet carries the memory address and coordinates to which CE an acknowledgment is sent to (*ACK Address*). In case of a LOAD the packet contains fields for the coordinates (*Response Address*) of the CE that waits for the response. If a STORE is performed, the packet contains the *Data* to be saved in the memory bank instead.

3.2 Compilation Framework

In this section we describe the process of compiling applications onto REDEFINE. The input to the compiler is an application described in C language. Our compiler is ANSI C complaint. Before we describe the compilation framework used to identify HyperOps, we list below the microarchitectural features of REDEFINE exposed to the compiler.

1. Communication between any two operations in a HyperOp, which are executing on the hardware is accomplished through an interconnect for scalar variables and

through memory for vector variables. (There is no central register file which is seen by the compiler. The use of the interconnect enables direct communication of the result and avoids the overhead of accessing the register file for a read or write.)

2. The interconnect follows a Honeycomb topology. Details of this topology are provided in [?].

3. All CEs are homogeneous. Each CE is capable of executing a set of arithmetic, logic, compare and memory access operations. Apart from these operations, few special operations are used to transfer data directly to other CEs.

4. In order delivery of data is guaranteed between each pair of communicating HyperOps that constitute a Custom Instruction.

The compilation process is divided into various phases:

- **Phase I - Formation of DFG:** Application synthesis in REDEFINE follows a data driven execution paradigm. The first phase transforms the application into a dataflow graph(DFG) and performs several optimizations to reduce the overhead of data transfer.

- **Phase II - HyperOp formation:** The basic entity in our paradigm is a HyperOp. This phase divides the application into several HyperOps.

- **Phase III - Tag generation:** In our execution paradigm multiple HyperOp instances can be active on the fabric simultaneously. To distinguish these HyperOps we generate tags (similar to tags in dynamic dataflow [**?**]) at runtime by the hardware. The necessary information required for the generation of the tags is identified in this phase. To reduce the overhead of tag generation we generate tags only for inputs and outputs of HyperOp. The data tokens within a HyperOp do not contain a tag.

- **Phase IV - Mapping HyperOps:** This phase of compilation is aware of the interconnect topology between the tiles of the reconfigurable fabric. The process

of Metadata generation involves identifying HyperOp partitions called p-HyperOps, such that all operations in a p-HyperOp can be assigned to a single CE. These p-HyperOps are mapped onto multiple CEs in the reconfigurable fabric based on communication patterns between them.

- **Phase V - Formation of Custom Instructions:** This step identifies HyperOps that can be aggregated into Custom Instructions. Custom Instructions are necessary to reduce the overhead of inter HyperOp communication. Unlike HyperOps, Custom Instructions need not be acyclic. We assume special hardware support to execute a Custom Instruction (as explained in section **??**).

Chapter 4

Domain characterization of REDEFINE

An application written in a high level language 'C' is transformed into coarse grain operations called "HyperOps" [?] by RETARGET[1], the compiler for REDEFINE. In order to tailor REDEFINE for a specific application domain, compiler directives may be used to force partitioning and assignment of HyperOps. We need to increase the execution efficiency of parts of applications that are executed multiple number of times. In order to address this, we suggest an improvement in REDEFINE. Computational structures are provisioned once, and repeatedly used for the lifetime of the application. In other words, computational structures are made persistent for the lifetime of HyperOps[2]. We provide an implementation of the suggested improvements in this work. We provide the support needed to efficiently execute core computations of the NLA domains. Core computations are the computations that get statistically often executed in multiple applications of an application domain. We architect hardware to efficiently execute these computations and enhance the CEs with this domain specific hardware. So,further, domain specific Custom Function Units (CFUs), which are micro architectural hardware assists may be

[1]RETARGET uses the LLVM [**?**] front end and generates HyperOps containing basic operations defined by the virtual ISA

[2]By lifetime of a HyperOp, we mean all its dynamic instances.

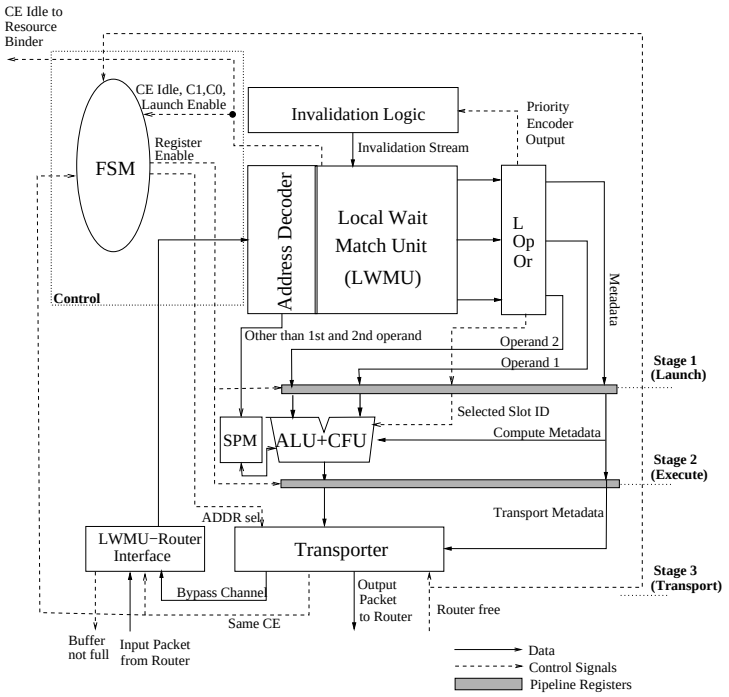


Figure 4.1: Schematic Diagram of Pipelined CE with enhancements over the same that appeared in [?]. The enhancements are the inclusions of CFU and SPM to reduce computation latency and memory latency respectively.

handcrafted to work in tandem with the ALU [?]. In the following sub-sections we elaborate streaming NLA-specific enhancements to REDEFINE in order to meet expected performance goals in a scenario where inputs are streamed, to:

- Making the HyperOps persistent to avoid relaunching overheads
- Reduce delays due to accesses to global memory
- Address rate-mismatch between producer and consumer CEs
- Improve performance by introducing the CFU and logical partitioning of the ALU

4.1 Support for Persistent HyperOps and Custom Instruction Pipeline

In order to meet very high throughput requirements of streaming applications relaunching of HyperOps which get repeatedly executed, must be avoided. We build the capability in the CEs to repeatedly perform the same set of operations. We rely on the support provided by the compiler as reported in [?] for this enhancement.

To make HyperOps persistent, its instructions need to be repeatedly executed several times. Therefore we introduce a new packet type for the CE Payload (refer to figure 3.2(a)). It contains a 16 bit value as counter representing the number of loop iterations for which the instructions of one particular CE are valid. If all instructions of the CE have been launched, the counter is decremented and the launch bits are reset. This process repeats till the counter reaches a value of zero. Then the CE is declared as idle representing that CE is ready to accept a new pHyperOp from the HL.[3] In case of streaming applications, HyperOps are made persistent throughout the lifetime of the application by loading the counter with a value of zero. We further make improvements by delivering loop invariant data only once for the lifetime of a loop.

Overheads incurred due to routing the results (produced by one HyperOp meant for another HyperOp already resident on the fabric) through the Support Logic can be avoided, by supporting channels of communication between the producing and consuming HyperOps.

Due to the Custom Instructions and the necessity of pipelines among them, in-order delivery must be ensured. Although packets are routed by a simple forward-and-store routing algorithm, the order can be changed by the Virtual Channels (VC) [**?**]. Instead of a fairly complex reassembly unit and due to the close neighborhood communication

[3]In case Custom Instruction pipelines are not established, even with persistent HyperOps, inter-HyperOp communications will be routed through the Support Logic. However in our case, RETARGET specifies these inter-HyperOp communications and the necessary enhancements have been made to the NoC. Hence we do not discuss the enhancements needed in the Support Logic for inter-HyperOp communications.

patterns, we use FIFOs at the output ports of the routers instead of VCs.

As mentioned earlier, a Custom Instruction Pipeline establishes communication among different HyperOps resident on the execution fabric, as shown in figure 5.2(a) thus reducing the overhead of data transmission via the Support Logic.

4.2 Reduction of global memory access delays

Each load/store request incurs a long round trip delay time, based on the placement of the CE making the request. Further these latencies are non-deterministic in nature due to the use of NoC. When streaming inputs are needed, if a separate request is to be made for every data element then memory access latencies determine the performance of the kernel. This is called "pull" operation where the CE requiring a global data makes an explicit load request to the global memory. This delay introduced by this pull model gets multiplied in case of streaming data - for every global load operation, the CE has to make an explicit load request and wait for the global data. There are several ways of decreasing this overhead. One mechanism is to enable the CE (to which streaming data is to be loaded) to make one explicit request to the global memory; thereupon the global memory streams the global data (without waiting for further load requests). In other words, a "push" model, would require the global memory to "volunteer" load of global data to CEs. Another enhancement to recede overheads due to global loads is to distribute and pre-load the global data to CEs, provided CEs have local storage. Having local storage will however not overcome the delay associated with indirect references. This delay can be abated partially if the local memory has an associated logic to resolve indirect references as part of the address calculation. The scratch-pad memory (SPM) serves as the local memory within each CE and scratch-pad memory controller (SPMC) has the additional logic for indirect address calculation.

4.3 Flow-Control

In REDEFINE, rate mismatch between a producer and a consumer could arise due to the use of NoC for communication of data. This is addressed by enforcing the consumer to request the producer to for data, once the consumer completes execution of one iteration of operations assigned to it. In other words, intra and inter HyperOp communication for propagating data, results in "chaining" of several producers and consumers. This requires special logic in each CE, so as not to overwrite previously produced data.

4.4 Performance improvement - Introduction of CFU:

Streaming applications require speeding certain critical operations in order to maintain throughput. To speedup these operations we introduce a Custom Functional Unit (CFU) in the CE. Such a CFU is a customized unit for a specific application/domain. For example, most of the NLA applications require multiply accumulate add operations. These applications can execute much faster, if a multiply-accumulate-add CFU is provided in the CE. In this section, we describe the details of the enhancements required to support such CFUs. The process of choosing the CFU, is not discussed in this thesis. We provide flexibility in choosing a CFU by allowing multiple input and multiple output CFUs.

To incorporate this CFU into the existing hardware infrastructure, we have introduced extra operand types. This new operand types specifie that the operands are meant for the CFU. The *SlotNo* field (refer figure 3.2(a)) specifies the input number of the CFU. Hence the number of inputs is limited by the number of bits assigned to the *SlotNo* field. In normal operations same result is delivered to different destinations, as indicated by the destination field. In case of CFU, different results are processed in the Transporter in consecutive clock cycles to form result packets which are then sent to their respective destinations via the NoC. The number of outputs of a CFU is limited by the number of destination fields available in the instruction.

In the context of NLA kernels, the handcrafted CFUs perform core computations required for matrix-vector multiplication i.e MAC, division, prime computations for QRD.

The structure of the CE reported in [?], with the modifications is shown in figure 4.1. The ALU shown in this figure is capable of performing all instructions from the Virtual ISA [?]. In case the ALU is not pipelined then it is obvious that the throughput of this ALU is determined by the highest latency to perform one operation. We go for a pipelined ALU and the operations have been categorized as either unit-cycle or multi-cycle operations. If the CE has to satisfy the throughput requirements in case of streaming inputs, then the ALU has to process both unit-cycle and multi-cycle operations. This would result in pipeline bubbles, reducing the throughput. In order to overcome this we logically partition the ALU into two units - one that performs unit cycle operations and the other that performs multicycle operations. This has the added advantage that both kinds of operations in the ALU can be relinquished, thereby reducing the contribution to area occupied by the ALU. In our work along with direct solver we also realize sparse matrix solver on REDEFINE. It is to be noted that core computations for both the direct and iterative or sparse solvers use the same CFU, but they differ in their NoC useage. Systolic algorithms are realized on REDEFINE by appropriate flow control i.e "chaining" the CEs.

4.5 Need for algorithm-aware compilation

Chapter 5

Realization of Systolic Algorithms on REDEFINE

In this chapter we discuss about the realization details of two kinds of NLA kernels. We target Modified Faddeevs Algorithm (MFA) as a potential direct solver. We bring about a proposition to realize MFA on REDEFINE, a coarse grained reconfiguarable architecture. We compare the performance numbers with that of a GPP solution to show that REDEFINE performs several times faster than traditional GPPs. Further we channelize our interest to QR Decomposition (QRD) to be the next NLA kernel as it ensures better stability than LU and other decompositions. As in the context of MFA we already show the performance enhancement in REDEFINE over GPP, we use QRD as a case study to explore the design space of the solution on the proposed reconfigurable platform i.e REDEFINE. We also investigate the architectural details of the Custom Functional Units (CFU) for these NLA kernels. Further, we report the synthesis results of CEs accomodating those CFUs serving the needs for core computations.

5.1 Realization of Faddeevs algorithm on REDEFINE

This section throws light upon the methodology used to realize Faddeevs Algorithm on REDEFINE. The exhaustive work has been reported in [ISVLSI'2010]. Excerpts of the

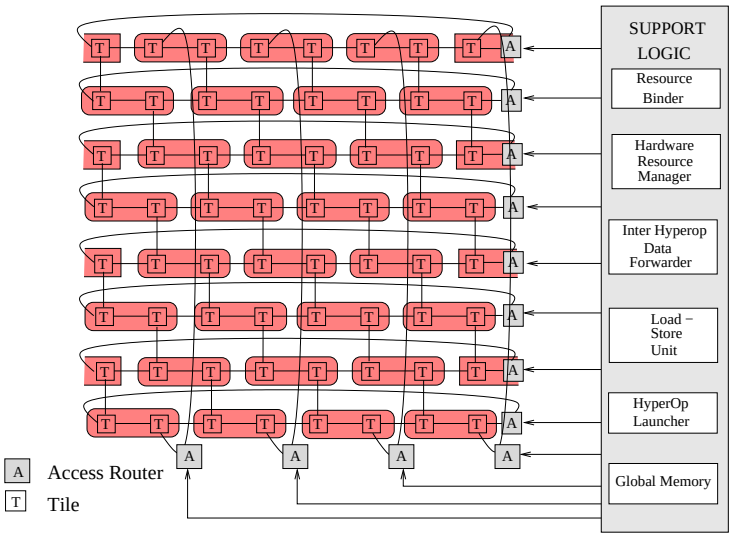
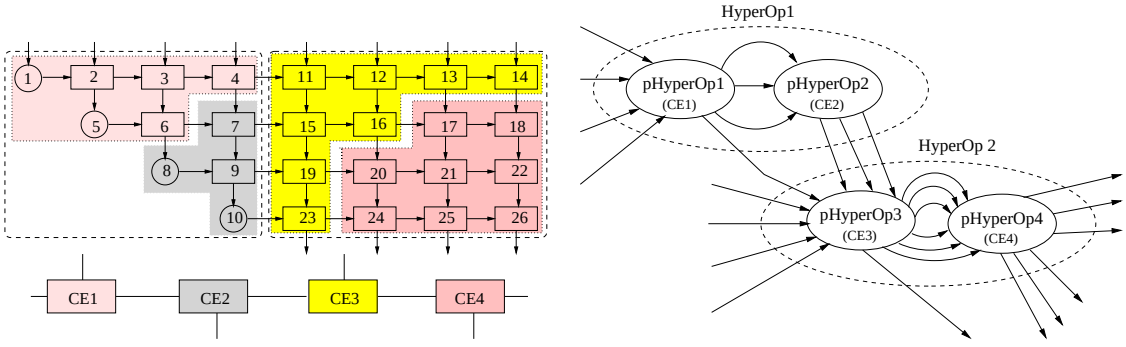


Figure 5.1: Shaded rectangles in the figure show two neighbouring Tiles logically bound together in a mesh interconnection

paper is reproduced here in the following sections.

5.1.1 Partitioning, mapping and other realization details

Systolic array implementations are the most efficient way of realizing MFA in hardware. As indicated previously, this implementation uses a mesh interconnection of processing elements. To emulate this on the REDEFINE, we treat two neighbouring tiles as a single logical entity, as shown in figure 5.1.



(a) Mapping of operations (b) HyperOps and pHyperOps formations

Figure 5.2: Mapping of operations and HyperOps and pHyperOps formations for the 4×4 systolic structure

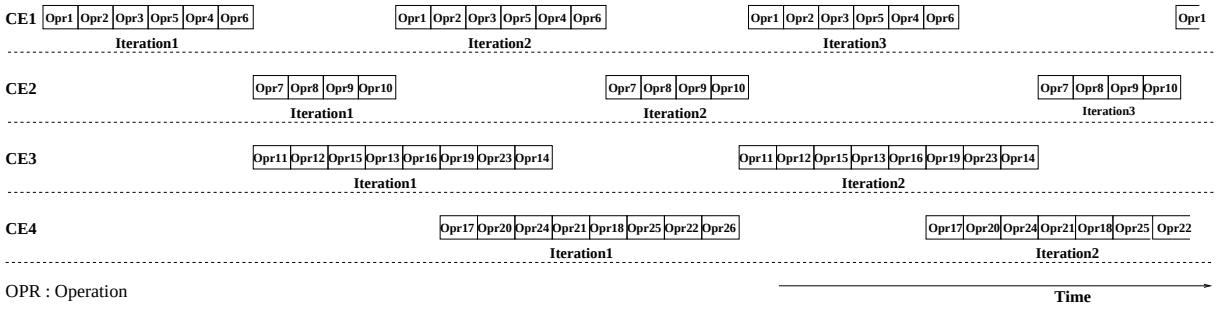


Figure 5.3: Sequence of operations of HyperOps 1 and 2 of the 4×4 systolic structure on REDEFINE

We map a portion of the systolic array i.e. sub-array onto a pair of CEs on REDEFINE. Figure 5.2(a) is the dependence graph for computing Schur complement for a 4×4 matrix. Formation of HyperOps, and assignment of pHyperOps to CEs are shown in figure 5.2(b). It is important to note that such an assignment honors the systolic order of execution. Figure **??** shows the order of execution of operations of the two HyperOps for the 4×4 systolic structure. Figure 5.4 shows the mapping of the systolic sub-array for computing the Schur complement of 8×8 and 16×16 matrices on the REDEFINE fabric. Grey regions in the figure shows the mapping of 8×8 matrix, while the hatched regions depict the mapping of 16×16 matrix. The HyperOp sizes for those two matrix sizes are 4×4 and 8×8 respectively.

Since sub-arrays from the systolic array are HyperOps which are in turn mapped to CEs, REDEFINE can potentially scale to realize large systolic arrays. This is achieved by mapping and scheduling HyperOps on the execution fabric in space and time. It is to be noted that the same fabric can be used as a solution for mapping systolic array of any size (theoretically) at the cost of slow-down. This slow-down is proportional to the number of nodes in a systolic array that are mapped to one CE-pair. As shown in figure 2.8, Division and MAC are the core computations of MFA. A hand-crafted CFU specifically realized to efficiently perform these operations is introduced in a CE appears in figure 5.5 (denoted by FP-CFU). The floating point MAC operation supported by FP-CFU serves the common computational need for both the solvers i.e. MFA and SMVM.

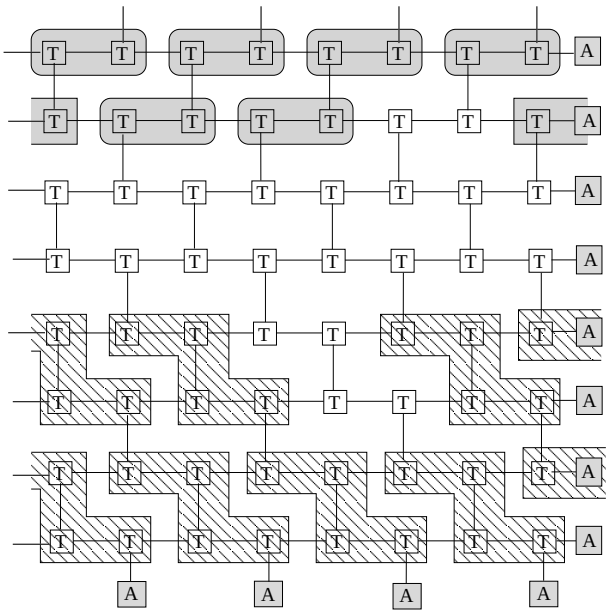


Figure 5.4: Mapping of systolic structures on REDEFINE. Grey regions depict mapping of systolic structure for 8×8 matrix. Hatched regions depict mapping of systolic structure for 16×16 structure. The HyperOp sizes for those two matrix sizes are 4×4 and 8×8 respectively.

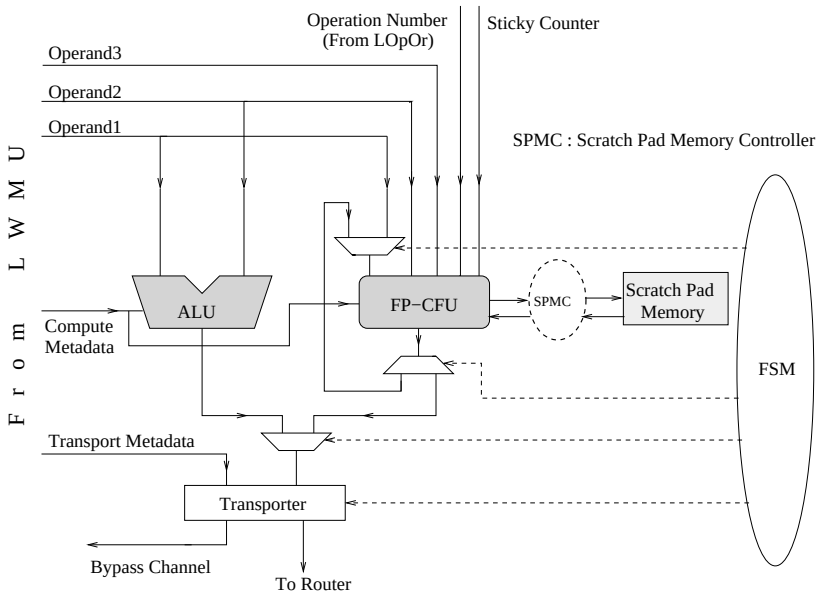


Figure 5.5: Realization of FP-CFU and Memory-CFU in the Compute Element

FP-CFU is a 2-stage pipelined unit that interfaces with the scratch-pad memory (SPM). A register called `Sticky Counter`, loaded with the number of times a HyperOp needs to be executed, is used to make a HyperOp persistent for repeated execution [?]. Further, `Mode Change Register` is used to change the nature of operations executed after certain number of iterations. These registers are initialized with values as indicated by Compute Metadata generated by the compiler. Buffer requirements needed in a systolic solution are realized on the SPM. The FP-CPU shown in figure 5.5 is runtime reconfigurable, in that it can also perform matrix-vector multiplication without any change to the hardware. The datapaths taken within the CE, are however different. Operands for the division and MAC operations required by Faddeev algorithm are supplied as Operand 1 (from Operation Store), Operand 2 (from Operation Store) and Operand 3 (from SPM). The output of the computation is appropriately forwarded to the dependent instructions. If they serve as input operands to operations held by the same CE, the bypass channel delivers them to the same CE. Routers are used to deliver the outputs, if they are destined for operations held by other CEs.

Kalman Filter can be realized as a sequence of MFA stages as described in [?]. For any

k-state Kalman Filter, we need to perform MFA on a compound matrix of size $2k \times 2k$. When $k \leq 16$, this can be realized as two parallel sequences of four MFAs, where each MFA is realized as shown in figure 5.4. For $k > 16$, the MFAs of the Kalman Filter need to be realized sequentially. This is because two instances of the MFA cannot be simultaneously accommodated on REDEFINE.

5.1.2 Results for MFA

The number of CE pairs used to map a given systolic array depends on the throughput requirements. Higher throughput is obtained when more number of CE pairs are assigned for computations. In case the number of CEs is less than this optimum number, this computation can be realized by "folding" multiple sub-arrays to one CE. However this comes at the cost of throughput. Note that, the number of PEs used in systolic array realization is O(n^2), whereas the number of CEs used in REDEFINE is $[3(n/k)^2 + n/k]$ for $k^2 \leq 2s$ and $[(3/2)(n^2/s) + (n/2)(k/s)]$ for $k^2 > 2s$, where $n \times n$ is the application size, $k \times k$ is the substructure size and s is the size of operation store in a CE.

The performance comparison of REDEFINE with respect to a GPP is given in Table 5.1. The compiler performs a semi-automatic partitioning and mapping of the full array into sub-arrays. We obtain the execution latencies of different MFA kernels for different matrix sizes on an Intel Pentium 4 Processor running at 2.2 GHz. The total time taken by the function was determined by Intel VTune Performance analyzer. The execution latency numbers indicate that REDEFINE, running at 50MHz provides several times faster solutions than traditional GPP solutions. Realization of larger size matrices gives more performance enhancement because of higher computation-communication ratio. For comparison with systolic solutions, we define Work Ratio as:

$$Work\,Ratio = \frac{No.of\,CEs \times No.of\,cycles(in\,REDEFINE)}{No.of\,PEs \times No.of\,cycles(in\,Systolic\,array)}$$

As seen in Table 5.1, the low variance in Work Ratio justifies the scalability of the solution.

Output Matrix Size	Systolic-Solution		Realization in REDEFINE		Work Ratio	Time taken[a] by GPP running at 2.2 GHz (in μsec.)	Speed Up in REDEFINE running at 50 MHz
	PEs	Cycles[a]	CEs	Cycles[a]			
2×2	7	6	4	79	7.524	8	5×
4×4	26	14	4	429	4.714	85	10×
			8	241	5.297		17×
6×6	57	22	8	613	3.911	356	29×
8×8	100	30	8	1508	4.021	1278	42×
			14	896	4.181		71×

Table 5.1: Comparison of performance with GPP and Systolic Solutions

[a]The *Cycle count* and *Time taken* reported here are for the computation of one Schur complement

Table 5.2: The area consumed by Floating point CE with and without Custom FU is shown

Number of Slots	CE type	Area in mm^2
16	CE supporting only basic operations	0.140591
16	CE with CFUs	0.166503

5.1.3 Synthesis results

The CE variants have been synthesized using Synopsys Design Vision and Faraday 90nm Standard Performance technology library. The area of a CE comprising 16 slots and supportng only basic floating point two operand operations (i.e, addition,substraction,multiplication,divis is presented in table 5.2. Table 5.2 also shows the area consumed by three operands CE with Floating point Unit that supports Custom Functions like MAC and Spcl Div ($A + BC, A - BC, -A + BC, -A - BC, -X/Y$) along with the before-mentioned basic operations. This enhanced CE also possesses the support that enables the custom operations to be operated in dual mode depending upon the no of iterations in case of persistent pHops. On average the execution time for the enhanced CE improves by 29% in comparision to GPP for a meager 18.43% increase in area.

5.2 Realization of QR Decomposition on REDEFINE

In this section along with the realization details of QR Decomposition (QRD) on REDEFINE we also present a design space exploration of the solution and synthesis reports of a typical CE consisting of QRD specific CFUs. The entire work has been elucidated in [SAMOS'2010]. Subsequent sections are re-duplication of the research-work in a nut-shell.

5.2.1 Actualization Details

The execution core of REDEFINE comprises multiple CEs (refer figure 5.1). Schematic diagram of a CE is shown in figure 4.1. Operations assigned to a CE, are stored in Local Wait Match Unit (LWMU). An operation is ready for execution only when all its input operands are received. It is to be noted that in a honeycomb topology, every node is a degree 3 element.

For systolic realization of QRD, the desired lattice is a mesh interconnection of processing elements. It is well known that systolic arrays are not scalable due to their rigid hard-wired structures. In this chapter we leverage systolic solution for QRD and cast them on REDEFINE. In general, systolic solutions are derived to exploit local communication between nodes in a systolic array. Toroidal honeycomb topology of REDEFINE can be rendered to support a mesh like lattice structure by combining two neighbouring Tiles as a single logical entity as shown in figure 5.1. Each shaded region in the figure depicts a CE-pair.

We map a sub-array of the systolic array onto a pair of CEs on REDEFINE. Each sub-array therefore represents a HyperOp. Depending on the size of the matrix being solved, the systolic array (representing the solution) is divided into multiple HyperOps. In turn each HyperOp is divided into pHyperOps; and each pHyperOp is assigned a CE in the CE-pair. Figure 5.6(a) is the dependence graph for computing QRD of a 8×8 matrix. Formation of HyperOps, and assignment of pHyperOps to CEs are also shown in figure 5.6(a) and figure 5.6(b) respectively. It is important to note that such an assignment honours the systolic order of execution. The dashed lines in figure 5.6(a) represents the

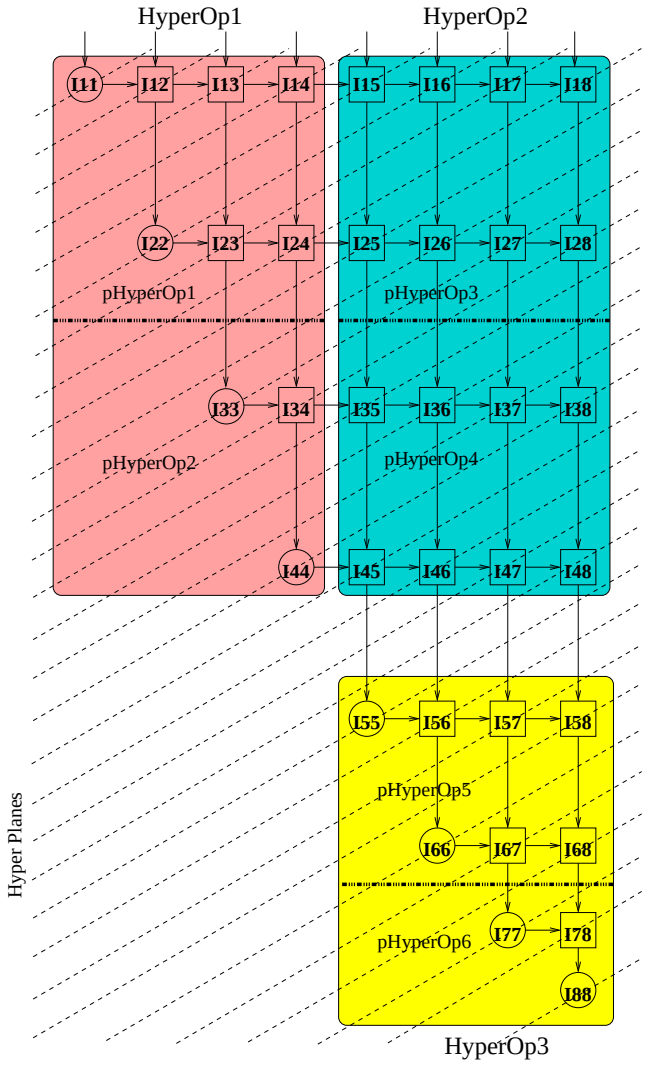
hyperplanes. The flow of the hyperplanes depicts the order of execution of operations of the HyperOps. This order obeys permissible linear schedule conditions [**?**] by ensuring

- All the dependency arcs flow in the same direction across the hyperplanes i.e causality is enforced.
- The hyperplanes[1] are not parallel with the projection vector i.e the nodes on an equitemporal hyperplane are not projected to the same CE.

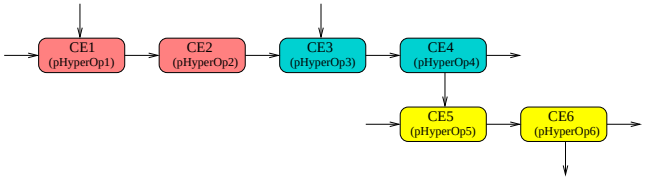
In REDEFINE, all the operations representing the systolic solution are realized in terms of instructions which are executed efficiently in the hand-crafted CFU of the CE. Instructions forming the HyperOp get executed repeatedly on the fabric as persistent HyperOps [?] till the maximum number of iterations needed for the particular output matrix size is reached. A 16 bit register maintains the iteration count [?]. In the systolic realization of QRD, there is a set of diagonal elements that generate factors (C and S as indicated in the figure 2.11) in every iteration and these factors are passed along the row. Once these factors are generated, they can be re-used for evaluation of other instructions of the same row. Storing of these factors in SPM, will reduce overhead compared to the situation when they are stored in global memory. Similarly, the computed values indicated by R in figure 2.11 (corresponding to intermediate values stored in the registers of the systolic array) can also be stored in SPM, thus eliminating the overheads associated with delivering the factor using the bypass channel (refer Figure 4.1) for propagation of R. Use of SPM for locally storing C, S and R potentially reduces communications. For instructions representing diagonal computations intra-CE communication is not required. If elements of same row are realized in different CEs then inter-CE communication is required. In case of off-diagonal computations number of output propagations is reduced from 4 (as shown in figure 2.11) to 1.

Wavefront Array processors [?] are the ASIC realizations of systolic arrays, with data-flow execution semantics. Systolic scheduling in this case propagates as a wave. REDEFINE is akin to realization of wavefront array schedules, since it follows data driven

[1]In systolic realization hyperplanes contain nodes that can be potentially executed in parallel



(a) HyperOps and pHyperOps formations



(b) Assignment of HyperOps and pHyperOps to CEs

Figure 5.6: HyperOps and pHyperOps formation and mapping of operations and for the 8×8 systolic structure for QRD

paradigm both for execution of operations and communication of output data. However rate-mismatches arising in such a situation is overcome by "chaining" the producer-consumer CEs. This mechanism is similar to the modular processing units of a Wavefront Array.

Global memory is used to store the initial matrices. QRD realization to cater to streaming inputs uses the "push" model of accessing global memory to repeatedly load the required data.

5.2.2 Design Space Exploration

REDEFINE is an architecture framework from which domain specific accelerators can be derived. The performance advantage of REDEFINE over FPGAs and General Purpose Processors can be found in [?, **?**]. In this section, we carry out a design space exploration of an n×n systolic array on REDEFINE considering a substructure size k×k to determine the optimal pipeline depth of the CFUs. We first consider each substructure realized on a single CE-pair. Hence each CE computes a substructure of size (k/2)×k.

As mentioned earlier, each CE in REDEFINE is allocated one pHyperOp. Further SPM is used to store the C and S factors, which will be used by all computations of the row assigned to that CE. In figure 5.6(a) C and S factors produced by I11 are stored in SPM, and will be used by I12, I13 and I14. However these factors need to be communicated to other CEs to which computations of the same row are assigned. In figure 5.6(a) C and S factors produced by I11 need to be communicated over the NoC to the CEs assigned I15, I16, I17 and I18. Due to the nature of interconnection of CEs, communication between two CEs directly connected takes 4 cycles [?] and between those connected two hops distance away takes 6 cycles [?].

Figure 5.7 shows the realization of 16×16 systolic structure on REDEFINE, considering a 4×4 substructure. In order to compute the critical path, we introduce dummy computations as shown in figure 5.7. Dashed line in figure 5.7 depicts the critical path, since all computations of a row are dependent on the node generating the C and S factors. pHyperOps on the critical path are realized on CE1, CE2, CE3, CE4, CE9, CE10, CE11,

CE12, CE15, CE16, CE17, CE18, CE19 and CE20 respectively (refer figure 5.7). For a substructure of size (k/2)×k realized on a single CE, k number of GR operations need to be performed in between two consecutive GG operations. Let T_{AB} be the time taken by a CE-pair to compute a part of the critical path between nodes A and B (refer figure 5.7). Note that computations within a CE are sequentially executed. Computations spread across two (or multiple) CEs can take place simultaneously as determined by the data-dependencies. Each CE is assigned one pHyperOp as shown in figure 5.7. Each pHyperOp is composed of k/2 rows, each row comprising (k-1) GR operations and 1 GG operation. A GG operation in a row is data-dependent on a GR operation of a previous row. Note that for the GG operations which are data-dependent on GR operations assigned to different CEs a penalty of 4 cycles (eg. between CE1 and CE2) or 6 cycles (eg. between CE2 and CE4) is experienced. Let $T_{last-substructure}$ be the time taken for the last part of the critical path (refer figure 5.7). The expressions for T_{AB} and $T_{last-substructure}$ are given in equation 5.1 and 5.2.

$$\begin{aligned} T_{AB} &= T^1_{k/2-1} + T^1_{last} + T_{CE1-to-CE2} + T^2_{k/2-1} + \\ &\quad T^2_{last} + T_{CE2-to-CE4} + T^4_{last} + T_{CE4-to-CE9} \\ &= (k/2-1)[T_{GG} + T_L + PB] + T_{GG} + \\ &\quad T_{GR} + 4 + (k/2-1)[T_{GG} + T_L + PB] \\ &\quad + T_{GG} + 6 + T_{GR} + 4 \\ \Rightarrow T_{AB} &= 2(k/2-1)[T_{GG} + T_L + PB] + \\ &\quad 2T_{GG} + 2T_{GR} + 14 \end{aligned} \tag{5.1}$$

$$
\begin{aligned}
T_{last-substructure} &= T^{19}_{k/2-1} + T^{19}_{last} + T_{CE19-to-CE20} + \\
&\quad T^{20}_{last} \\
&= (k/2-1)[T_{GG} + T_L + PB] + \\
&\quad T_{GG} + T_{GR} + 4 + \\
&\quad (k/2-1)[T_{GG} + T_L + PB] + \\
&\quad T_{GG} \\
\Rightarrow T_{last-substructure} &= (k-2)[T_{GG} + T_L + PB] + \\
&\quad 2T_{GG} + T_{GR} + 4
\end{aligned}
\tag{5.2}
$$

where

$$
\begin{aligned}
T^j_{k/2-1} &= Cycles\ taken\ for\ computations\ of \\
&\quad (k/2-1)\ rows\ realized\ in\ CEj \\
T^j_{last} &= Cycles\ taken\ for\ computations\ of\ last \\
&\quad row\ in\ CEj\ before\ the\ consumer\ CE \\
&\quad starts\ its\ computation \\
T_{CEi-to-CEj} &= Cycles\ taken\ for\ data\ delivery \\
&\quad from\ CEi\ to\ CEj \\
PB &= Pipeline\ \ Bubbles \\
T_{GG} &= Cycles\ taken\ for\ one\ GG\ operation \\
T_{GR} &= Cycles\ taken\ for\ one\ GR\ operation \\
T_L &= Cycles\ taken\ for\ launching\ of\ all\ GR \\
&\quad operationsin\ between\ two\ consecutive \\
&\quad GG\ operations.
\end{aligned}
$$

Once the factors C and S (refer figure 2.11) are generated, there is no data dependency

among the instructions of a row. However there is data dependency between an instruction in a row and an instruction in the successor row (for eg.: instruction I12 and I22 in figure 5.6). As depicted in figure 4.1, each CE has three stages, viz., Launch, Execute and Transport. As a general case study if the Execute stages for GG and GR operations are further realized as m1 and m2-stage units respectively, and an instruction which is data dependent on another instruction allocated to the same CE (eg.: I12 and I22 in figure 5.6), then the time difference between the two instructions entering the Execute stage is m2+2. If k<(m2+2), then the number of pipeline bubbles experienced between computations of these two instructions is (m2+2)-k. Pipeline is free of bubbles, if k>(m2+2). The transporter transports only one packet at a time. Hence, an operation, eg. GG operation resides for (m1+1) cycles in the execute stage. The GG operation takes m cycles for execution and it stays for one more cycle till last among the two generated values (C and

S) enters the transport stage. Hence, from equations 5.1 and 5.2 we can say,

$$
\begin{aligned}
For\ \ k < (m2+2),& \\
T_{AB} &= 2(k/2-1)[(m1+1)+k+(m2+2)- \\
&\quad k]+2(m1+1)+2m2+14 \\
\Rightarrow T_{AB} &= k(m1+m2+3)+10 \qquad (5.3) \\
and& \\
T_{last-substructure} &= (k-2)[(m1+1)+k+(m2+2)- \\
&\quad k]+2(m1+1)+(m2+2)+4 \\
\Rightarrow T_{last-subtructure} &= k(m1+m2+3)-m2+2 \qquad (5.4) \\
For\ \ k \geq (m2+2),& \\
T_{AB} &= 2(k/2-1)[(m1+1)+k+0]+ \\
&\quad 2(m1+1)+2m2+14 \\
\Rightarrow T_{AB} &= k(m1+1)+k(k-2)+2m2+14 \\
& \qquad (5.5) \\
and& \\
T_{last-substructure} &= (k-2)[(m1+1)+k+0]+ \\
&\quad 2(m1+1)+(m2+2)+4 \\
\Rightarrow T_{last-subtructure} &= k(m1+1)+k(k-2)+m2+6 \qquad (5.6)
\end{aligned}
$$

For an application size n×n the the path from A to B is repeatedly executed (n/k-1) number of times on the critical path. Since the CEs repeatedly compute the same instructions, we preload Compute and Transport metadata to CEs. Neglecting the time taken for preload (which is expected to be relatively small in comparison to the computation time for reasonable problem sizes), the total number of cycles taken for one iteration is given by the following expressions:

$$T_{single-iteration} = 1 + (n/k - 1)T_{AB} + T_{last-substructure} \tag{5.7}$$

$For \;\; k < (m2 + 2),$

$$\begin{aligned} T_{single-iteration} &= 1 + (n/k - 1)[k(m1 + m2 + 3) + 10] \\ &\quad +k(m1 + m2 + 3) - m2 + 2 \end{aligned} \tag{5.8}$$

$For \;\; k \geq (m2 + 2),$

$$\begin{aligned} T_{single-iteration} &= 1 + (n/k - 1)[k(m1 + 1) + k(k - 2) + \\ &\quad 2m2 + 14] + k(m1 + 1) + k(k - 2) + \\ &\quad m2 + 6 \end{aligned} \tag{5.9}$$

We next consider realization of each substructure on P CE-pairs (refer figure 5.8). In this case, each CE has to perform (k/2P)×k computations. Using the same approach as above the expressions for cycle count for single iteration in this case are:

$For \;\; k < (m2 + 2),$

$$\begin{aligned} T_{single-iteration} &= 1 + (n/k - 1)[k(m1 + 1) + \\ &\quad (m2 + 2)(k - 2P) + (m2 + 4)(2P - 1) + m2 + 10] \\ &\quad +k(m1 + 1) + (m2 + 2)(k - 2P) + (m2 + 4)(2P - 1) \end{aligned} \tag{5.10}$$

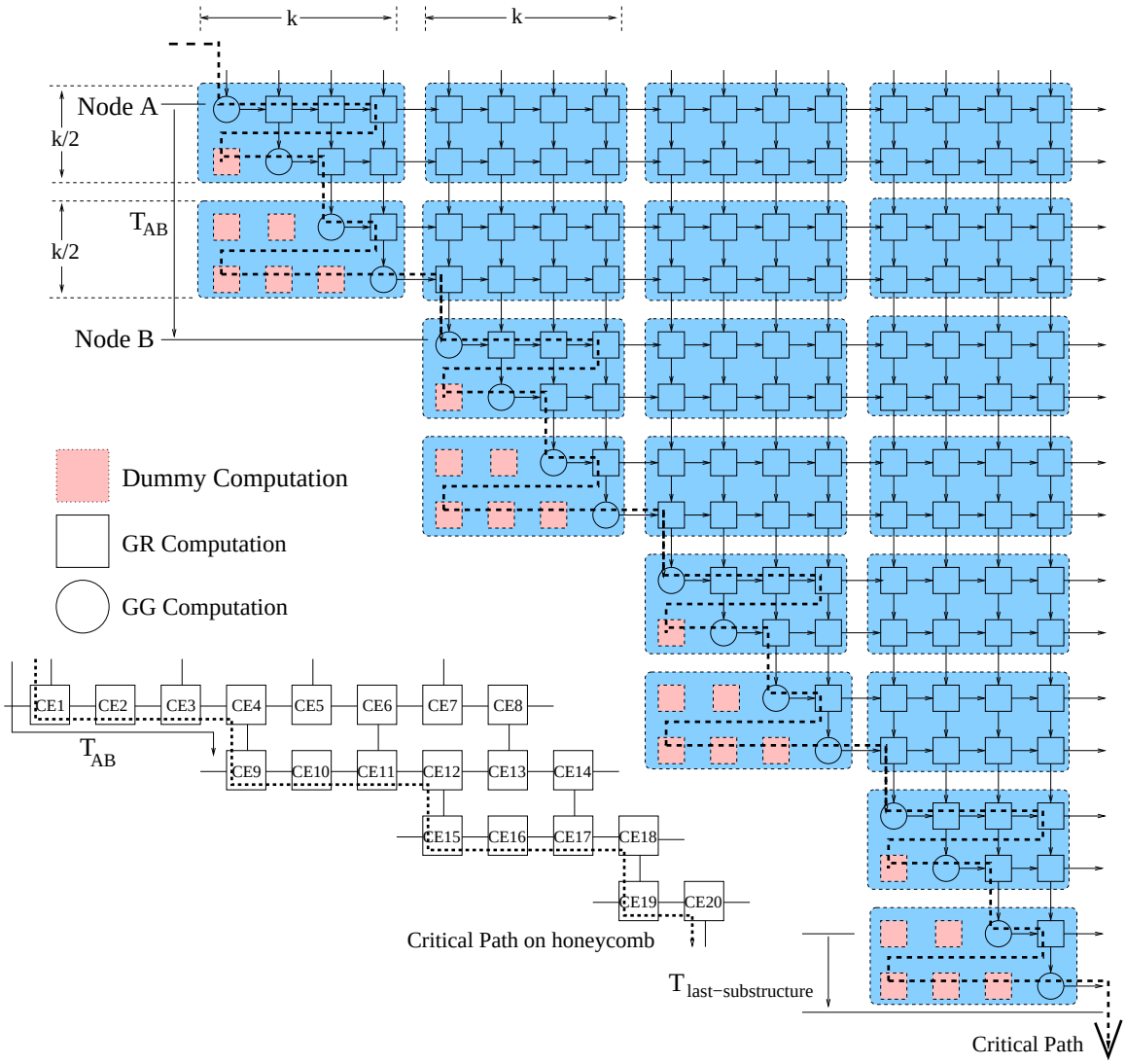


Figure 5.7: Critical path for a typical example of 16×16 systolic structure realization on REDEFINE with a substructure size of 4×4, each substructure is realized on a single CE-pair. Critical path on honeycomb is also shown on one pHyperOp per CE basis.

$$
\begin{aligned}
For\ \ k \geq (m2+2),& \\
T_{single-iteration} &= 1 + (n/k-1)[k(m1+1) + k(k-2P) \\
&\quad +(m2+4)(2P-1) + m2 + 10] + \\
&\quad k(m1+1) + k(k-2P) + (m2+4)(2P-1)
\end{aligned}
\tag{5.11}
$$

In order to complete the factorization of a given n×n matrix, n iterations need to be performed. However, as mentioned earlier, in order to ensure correct execution on REDEFINE, the producer and consumer CEs need to be "chained". This is achieved by the consumer CE sending an "acknowledgment" signal to the producer CE. Acknowledgements are needed to address rate-mismatch between producer and consumer CEs. As a consequence between two consecutive iterations a finite time gap as indicated by $T_{iteration-gap}$ in figure 5.12 is experienced. From figure 5.12 the generic expression for total n iterations of the critical path can be shown as

$$
T_{n-iterations} = T_{single-iteration} + (n-1)[T_{iteration-gap} + T_{last-phOp}] \tag{5.12}
$$

$$
T_{iteration-gap} = T_{non-overlap} + T_{ack} \tag{5.13}
$$

The expression[2] for completely factorizing a n×n matrix is given by

$$
\begin{aligned}
For\ \ k < (m2+2),& \\
T_{n-iterations} &= 1 + (n/k-1)[k(m1+1) + \\
&\quad (m2+2)(k-2P) + (m2+4)(2P-1) + \\
&\quad m2+10] + k(m1+1) + \\
&\quad (m2+2)(k-2P) + (m2+4) \\
&\quad (2P-1) + (n-1)[4 + (m1+1)k/P + \\
&\quad (m2+2)(k/P-2) - k/2P + \\
&\quad +2m2 + T_{ack}] \qquad (5.14)
\end{aligned}
$$

$$
\begin{aligned}
For\ \ k \geq (m+2),& \\
T_{n-iterations} &= 1 + (n/k-1)[k(m1+1) + \\
&\quad k(k-2P) + (m2+4)(2P-1) + \\
&\quad m2+10] + k(m1+1) + k(k-2P) + \\
&\quad (m2+4)(2P-1) + (n-1)[4 + \\
&\quad (m1+1)k/P + k(k/P-2) - \\
&\quad k/2P + 2m2 + T_{ack}] \\
& \qquad (5.15)
\end{aligned}
$$

where T_{ack} is the time taken for the acknowledgment to travel from consumer CE to producer CE.

In the above, it is assumed that both GG ang GR operations are executed in CFUs of pipeline depth m. However in reality, they could be executed in two CFUs of different pipeline depths i.e m1 and m2 respectively. Generally m1 is greater than m2 due to

[2]Due to lack of space, derivations of the expressions for cycle count are not presented here in the thesis

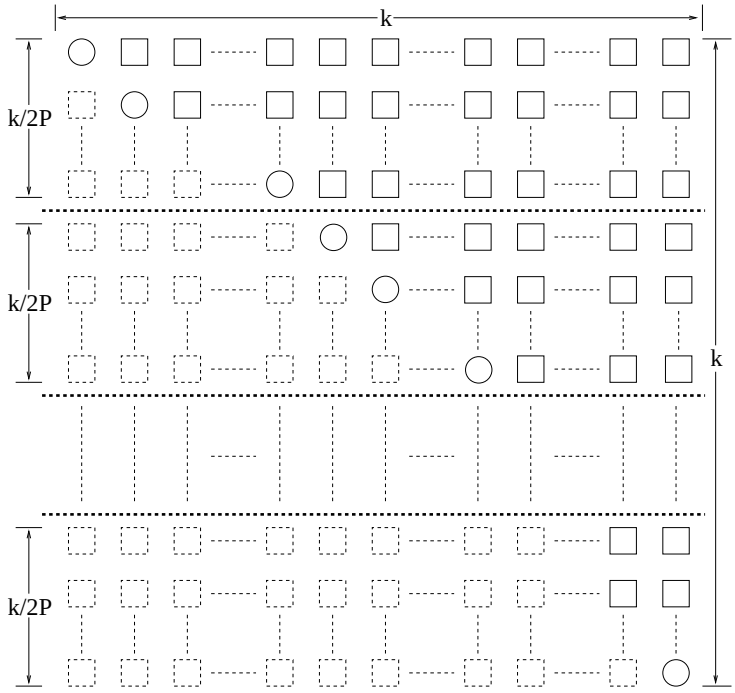


Figure 5.8: Realization of one k×k substructure on P CE pairs

the complexity of GG operations over GR operations. There is only one GG operation per row and once one GG operation is done the next k number of GR operations before the 2^{nd} GG operation are data-independent instructions. They are more prone to be launched in a pipelined fashion. Further when a GR operation is assigned to a CE with a CFU concieved as a MAC unit the GR operation is performed as four interdependent RISC type MAC instructions (partition is done by the CFU internal logic). Among them initially two MAC instructions can be launched in consecutive cycles followed by other two dependent MAC instructions after a latency depending upon the pipeline depth m2 of the CFU responsible to execute the GR operations. The pipeline bubbles introduced by this can be reduced if other GR operations can be launched while one is still on the fly. If 2k≤m2, then the number of pipeline bubbles experienced between computations of the two GR instructions of same column (eg.: I12 and I22 in figure 5.6) is 2(m2-2k)+2. Pipeline is free of bubbles, if 2k>m2. Figure 5.9 depicts the situation assuming the first case. Hence, equation 5.7 takes the following form:

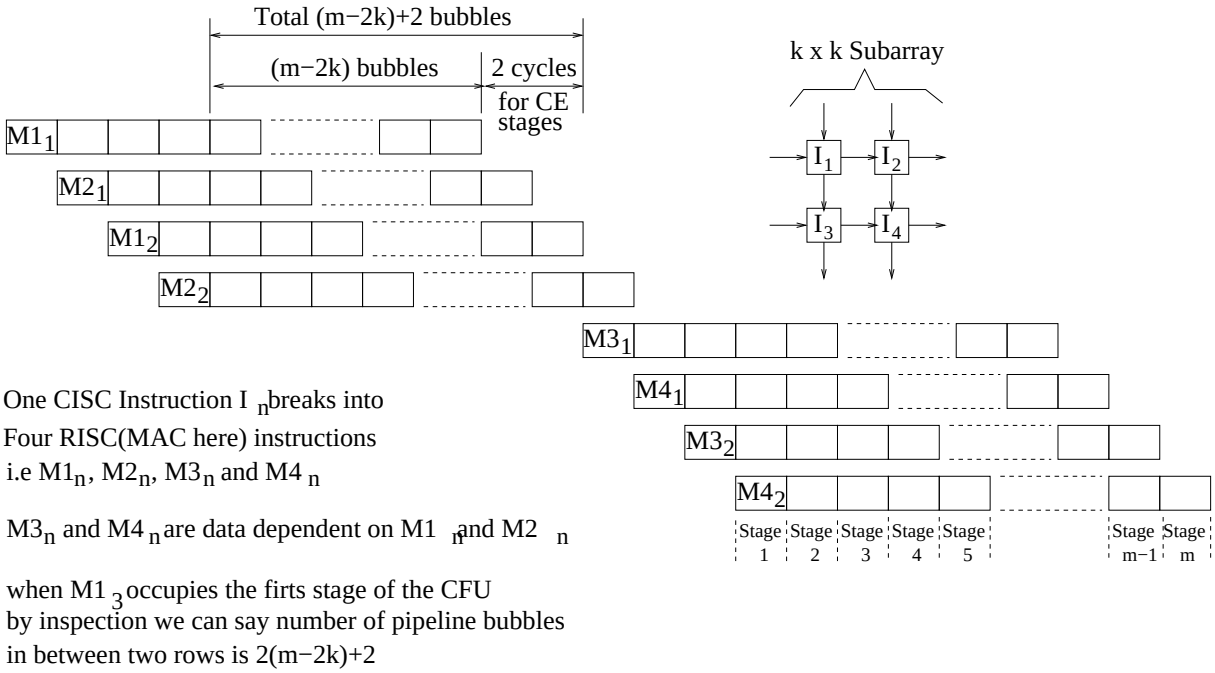


Figure 5.9: For a m stage pipelined CFU, calculation of pipeline bubbles when a CISC instruction breaks into RISC instructions.

$$
\begin{aligned}
&For\ \ 2k \leq m2, \\
T_{single-iteration} &= 1 + (n/k - 1)[2(k/2 - 1)\{m1 + 1 + \\
&\quad 4k + 2(m2 - 2k) + 2\} + 2(m1 + 1) + \\
&\quad 2(2m2 + 1) + 14] + (k - 2)\{m1 + 1 \\
&\quad +4k + 2(m2 - 2k) + 2\} + \\
&\quad 2(m1 + 1) + (2m2 + 1) + 4
\end{aligned}
\tag{5.16}
$$

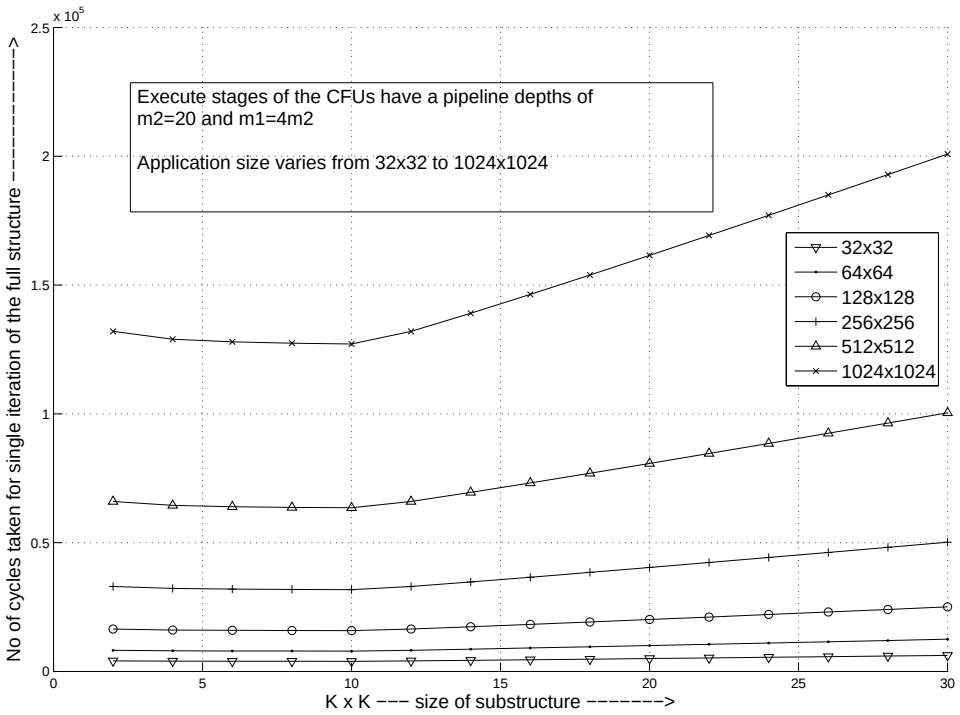


Figure 5.10: Plots indicating the best substructure size for optimal performance in terms of cycle-count

$$
\begin{aligned}
&For\ \ 2k > m2, \\
T_{single-iteration} &= 1 + (n/k - 1)[2(k/2 - 1)(m1 + 1 + \\
&\quad 4k) + 2(m1 + 1) + 2(2m2 + 1) + \\
&\quad 14] + (k - 2)(m1 + 1 + 4k) + \\
&\quad 2(m1 + 1) + (2m2 + 1) + 4
\end{aligned}
\tag{5.17}
$$

For a given m1 and m2 (the pipeline depths of the Execute stages of the CFUs), figure 5.10 shows the plot of number of cycles taken for a single iteration versus varying size of substructures for different problem sizes. As expected, minimum number of cycles is obtained when the substructure size, k is m2/2. Figure 5.11 shows the normalized

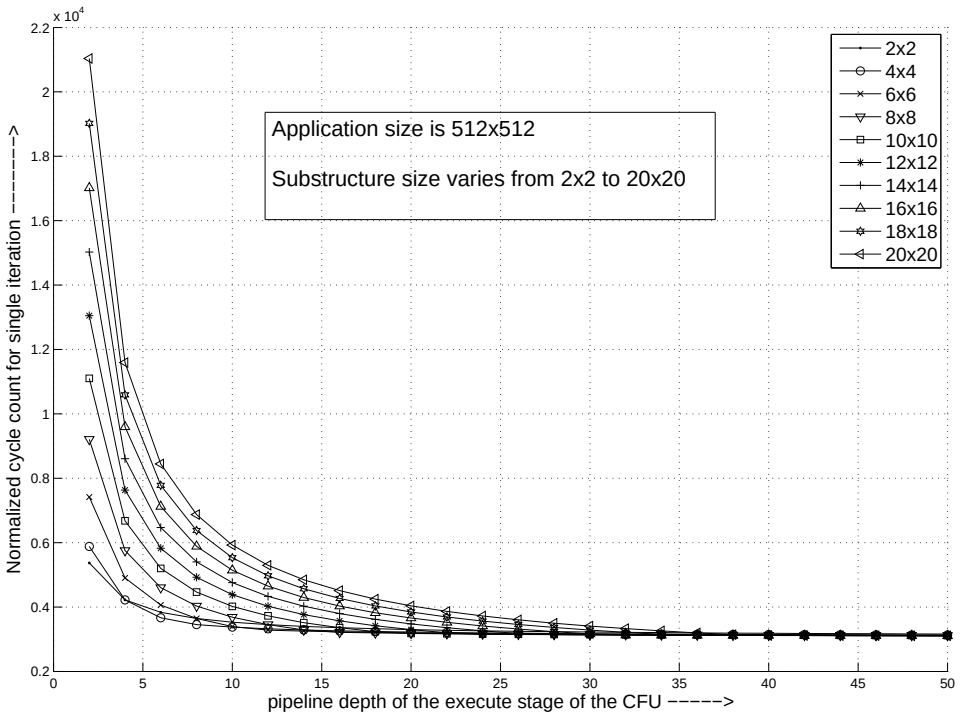


Figure 5.11: Plots showing the normalized cycle-counts with the change in pipe-line depth for different substructure sizes

cycle count (w.r.t. pipeline depth of the Execute stage of a CFU) versus the pipeline width for varying values of substructure of an application size 512×512. From figure 5.11, it is observed that there is negligible performance gain when the pipeline depth (m2) is increased beyond 20–24 for all sizes of substructures. Hence for substantially large problem sizes, the optimal substructure size that can be considered is 10×10 or 12×12.

The modified look of the parameters of equation 5.12 representing time taken for n

iterations become

$$\begin{aligned} T_{single-iterations} &= 1 + (n/k(i) - 1)[2p\{(m1 + 1 + 4k + \\ &\quad PB)(k/2p - 1) + m1 + 1\} + \{(2m2 + 1) + \\ &\quad 4\}(2p - 1) + 6 + (2m2 + 1) + 4] + \\ &\quad 2p\{(m1 + 1 + 4k + PB)(k/2p - 1) + \\ &\quad m1 + 1\} + \{(2m2 + 1) + 4\}(2p - 1) \end{aligned} \tag{5.18}$$

$$\begin{aligned} T_{iteration-gap} &= 4 + (m1 + 1 + 4k + PB)(k/2p - 1) + m1 + 1 \\ &\quad + PB_{sub1} + 4(k - k/p - 1) + m2 - \\ &\quad PB_{sub2} - 4(k - 1 - k/2p) + T_{ack} \end{aligned} \tag{5.19}$$

$$T_{last-phOp} = (m1 + 1 + 4k + PB)(k/2p - 1) + m1 + 1 \tag{5.20}$$

where

$$\begin{aligned} For\ \ 2k \leq m2, & \\ PB &= 2(m2 - 2k) + 2 \\ else\ \ PB &= 0 \\ For\ \ 2(k - k/p) \leq m2, & \\ PB_{sub1} &= m2 - 2(k - k/p) \\ else\ \ PB_{sub1} &= 0 \\ For\ \ 2(k - k/2p) \leq m2, & \\ PB_{sub2} &= m2 - 2(k - k/2p) \\ else\ \ PB_{sub2} &= 0 \end{aligned}$$

For a given pipeline depth (m2) of 20, figure 5.13 shows plots of cycle count versus

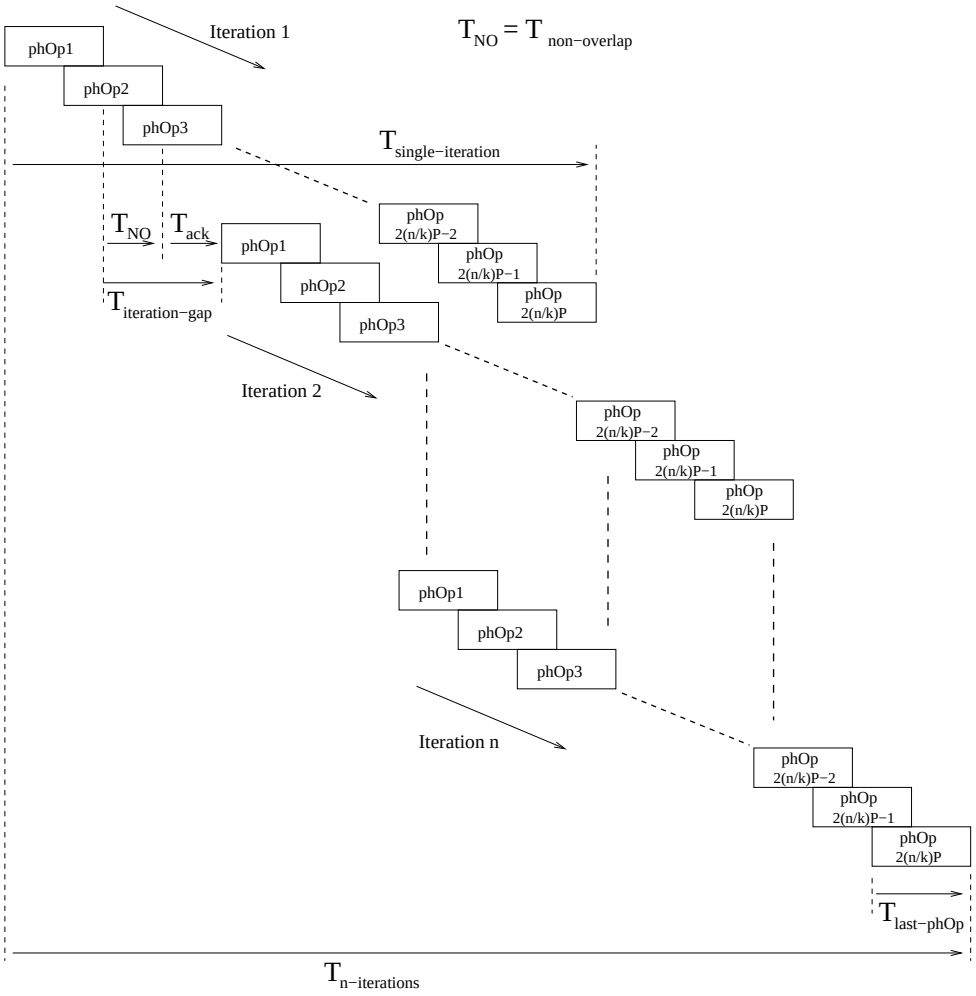


Figure 5.12: Time taken for n iterations of the critical path of problem size n×n

substructure-size for varying values of CE-pairs to be used i.e P, the number of CE-pairs used for mapping one substructure of an application size 512×512. From the plots it is evident that P=k/2 gives the optimal cycle count.

5.2.3 Custom functional Units for QRD realization

In this section we concentrate on the high level implementation details of different CFUs used to realize before-mentioned QRD kernels. For Faddeevs Algorithm the compuation requirements are division and MAC. Realization of QRD needs support for squre-root operation in addition. Every arithmetic unit performs calculation using signed floating-point arithmetic. We further report the synthesis results of a CE comprising those CFUs.

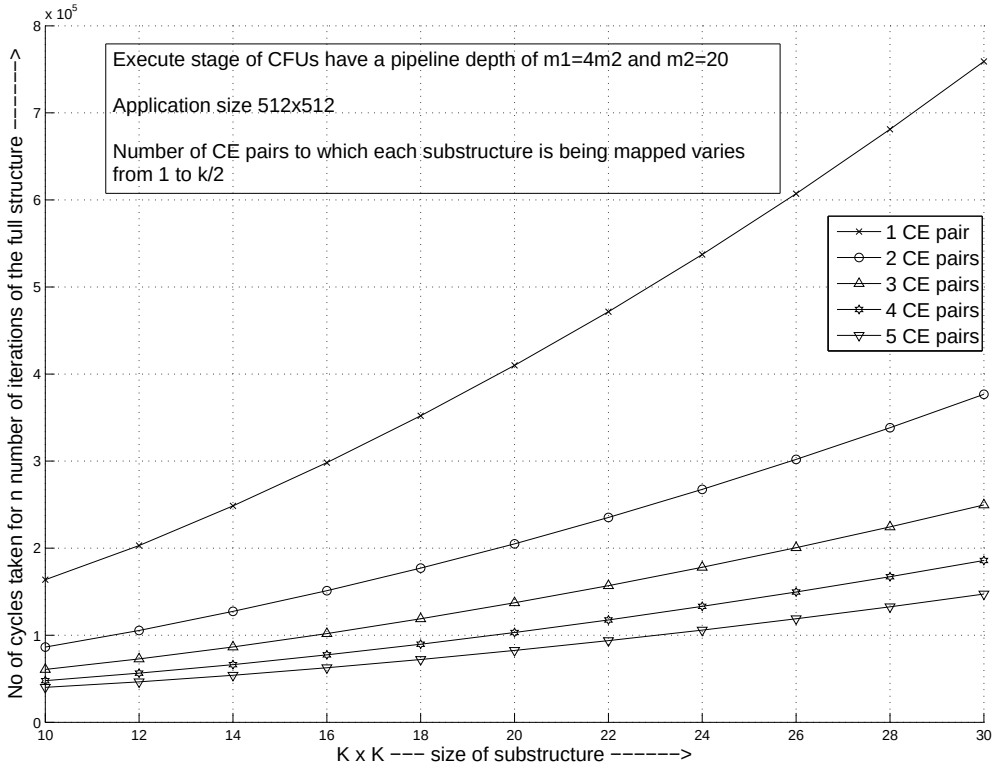


Figure 5.13: Plots indicating the best choice of the number of CE pairs to realize one k×k substructure

From the design space exploration done in the previous section we can come to a conclusion that the CFU providing the support for GR operations should have a pipeline depth of 20 for optimal performance. In accordance with that the optimal substructure size should be 10×10. Ultimate performance comes when each 10×10 substructure gets mapped on 5 CE-pairs. Hence, each CE would accomodate 10 macro-level instructions. A 16 slot CE would suffice for this requirement.

GG operation is a combination of squreroot and division. CFU1 provides the support for that. The operand must be in the square root unit input before the calculation process starts. We used Newtons Iteration Method which is also known as Newton- Raphson Method to find the root of the input data. CFU1 i.e the amalgamation of Square-root and division units consumes two sets of data. X_{in} (refer figure 2.11) comes from the reservation station (Local Operation Orchestrator(LOpOr)) and R gets retrieved from the SPM. First multiplication and then division is done one by one and the C and S factors are generated at consecutive cycles. The division and square root units are pipelined. Internal register is used to hold the intermediate result generated by the square-root unit.

Once C and S factors are generated GR operations can go under execution. GR operations are facilitated in CFU2. GR operations are combined MAC operations. An enhanced MAC unit (shown in figure 5.14) serves the purpose for breaking each complex GR operation into four RISC type MAC instructions and execute them sequentially in a pipelined manner without avoiding the data-dependency constraint. As mentioned previously, number of data-independent GR operations at a time is equal to the number of operations in a row of the substructure. During computation phase the information regarding the number of instructions that can be broken into RISC type MAC operations and launched onto the enhanced MAC unit comes from a register namely `Row_Length_Register` (refer figure 5.14). While loading the configuration data i.e the meta-data into the CE the substructure size value is also written onto that register. One controller unit (partially depicted in figure 5.15) generates the necessary control signals to ensure the correct data movements. Without any alternation in the hardware design the CE with these set of CFUs can be used to realize the core computations of Faddeevs Algorithm mentioned

previously.

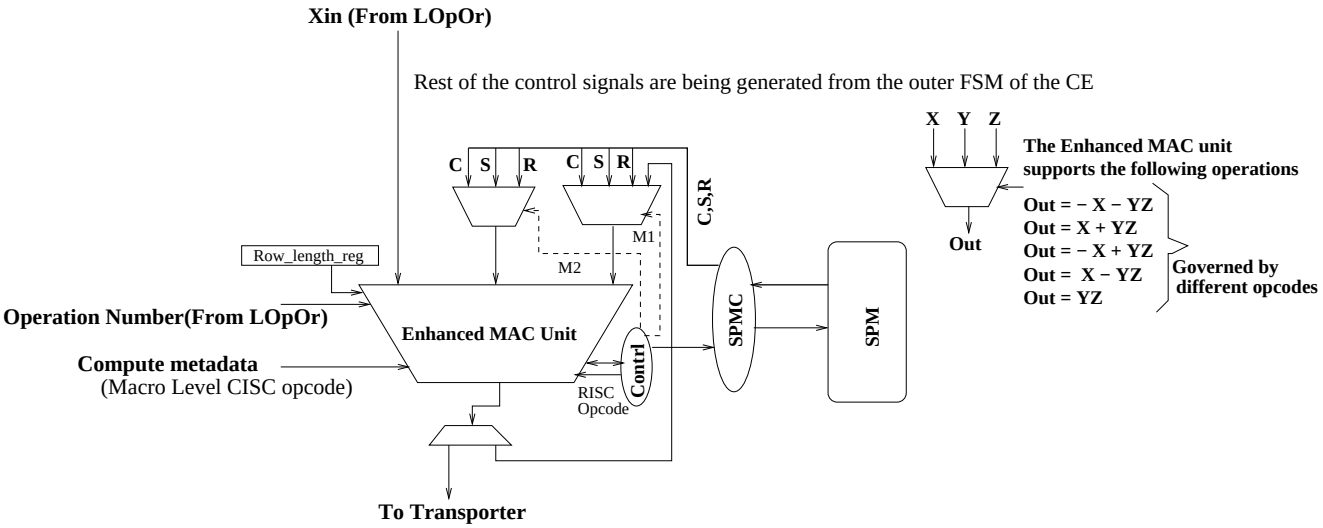


Figure 5.14: Enhancements over FP-CFU and Memory-CFU in the Compute Element to realize QRD kernels

5.2.4 Synthesis results

[h!] A typical CE hosting the CFUs that provide support for GG and GR operations has been synthesized using Synopsys Design Vision and Faraday 90nm Standard Performance technology library. The area and power consumed by a CE comprising 16 slots with a signal activity factor of 50% are reported in table 5.3.

Table 5.3: The power and area consumed by Floating point CE with Custom FUs are reported here

Number of Slots	Power in mw	Area in mm^2	Maximum Operating Frequency in MHz
16	0.165	0.596153	312.5

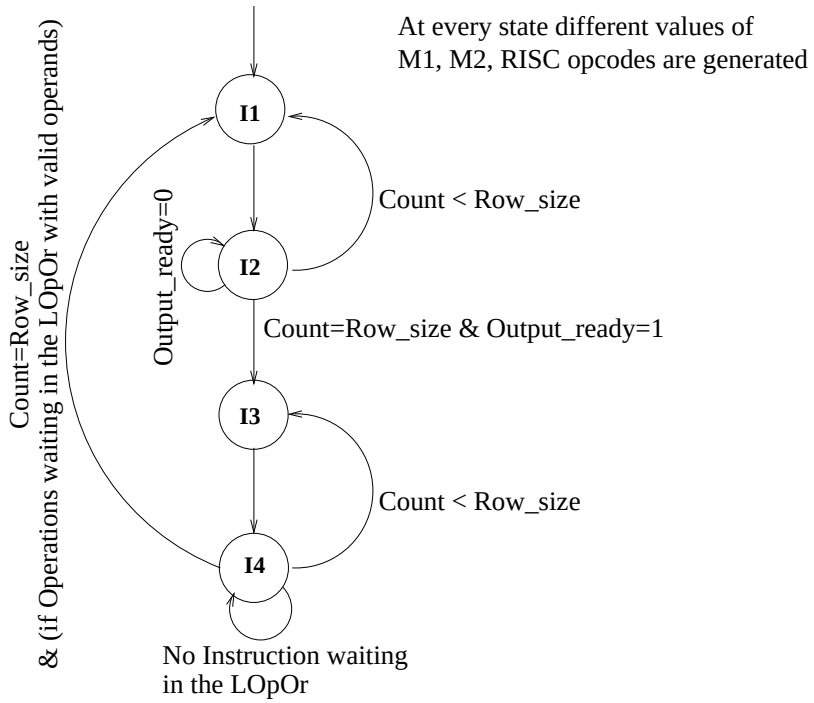


Figure 5.15: Part of the FSM controller that helps to break the macro-level CFU instruction into four RISC type instructions by generating proper control signals for the CE set-up shown in figure 5.14.

Chapter 6

Conclusion and Future work