Scalable and Configurable Array Architectures for Matrix Computation

Luo Jianwen

School of Electrical & Electronic Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirements for the degree of
Doctor of Philosophy

2008

To my family.

Acknowledgements

First and foremost I would like to acknowledge my supervisor Associate Professor Jong Ching Chuen. Without his diligent supervision and helpful discussion, this research work would not be carried out and continue to the end. I am also grateful to Associate Professor Chang Chip Hong, for his tireless discussion, kind help and encouragement throughout the entire days.

It will also be my great pleasure to address my appreciation to my labmates Dr. Xu Fei (Ms.), Mr. Xia Fei, Mr. Wang Chao, Ms. Huang Shaoying, Mr. Chua Egn Kee, Dr. Xu Pengfei, Dr. Gu Jiangmin, Dr. Li Juhui for their helpful support, zealous encouragement and lots of sparkling ideas through the countless discussions during the progress of this research.

I am also appreciating the support and help from my friends and colleagues Mr. Yan Fan, Mr. Lu Dawei, Dr. Zhang Lichen and Dr. Yu Zhuliang. Thanks for their valuable suggestions, comments and criticisms.

Last but not least, I will give my sincere gratitude to my beloved family, my parents and my sister who are always the backbone of my life and are always encouraging me to continue my research work, and my late grandfather who always cared about my study and overseas living in his lifetime. It is truly their deepest love and care that kept me concentrate, perseverant and vigorous on my study and never give up.

Summary

Modern digital signal processing and communications often involve computationally demanding algorithms that deal with large and complex matrix problems such as matrix multiplication and matrix inversion. These computations are commonly demanded in many scientific and engineering applications such as radar signal processing, control algorithms, image and video processing, etc. The matrix algorithms are usually computation and memory intensive. The traditional solutions with digital signal processors or general purpose processors are slow in processing speed and can hardly meet the ever increasing real time processing requirements. An alternative solution to the complex matrix problems is to use hardware accelerators implemented with application specific integrated circuits (ASICs) or reconfigurable hardware such as FPGA devices. The FPGA devices have drawn considerable attention recently due to its reconfigurability feature. With the hardware solutions, data can be processed concurrently in either pipelined or systolic arrays to increase processing speed. The adaptability of the design to different applications or different problem sizes is another important issue to be considered. With the reconfigurability design concept, the same piece of hardware design can be reused for different applications without requiring redesign. To utilize this type of reconfigurable hardware, new design methodology needs to be developed. The new methodology should facilitate short design cycle and promote reusability and reconfigurability.

This thesis focuses on studying and exploring the scalable and reconfigurable ar-

ray architectures for matrix computation, specifically, the highly challenging matrix multiplication and matrix inversion. Several array architectures for matrix multiplication and matrix inversion are proposed, developed and documented here. These architectures have the features of scalable, parameterizable, configurable and efficient in computation speed.

A reconfigurable array processor for matrix multiplication is proposed and developed to save both the hardware cost and processing time. The array processor is very versatile and can be configured as a parallel structure, a systolic array or a uniprocessor. The design and implementation of the proposed matrix multiplier, together with its performance in terms of speed and size, are presented in the thesis.

A reconfigurable array processor for matrix inversion is proposed and developed to tackle the matrix inversion in two steps. In the first step, the matrix inversion algorithm is divided into two separate segments which can be mapped to the architectures with structural similarity. This allows the computation to be decomposed into and carried out by several simple functional units. By studying the computation sequence, a novel parameterizable *Bi-z* CORDIC is proposed specially for the functional processing in the matrix inversion. The *Bi-z* CORDIC takes the advantage of the data processing coherence and eliminates the hardware cost and delay for storing, broadcasting and decomposing the rotation angle values that are usually required. In the second step, two novel mapping methods for mapping efficiently the two-dimensional array onto different linear array architectures are developed. The mapping methods achieve up to 100% processor utilization.

With the scalable and reconfigurable matrix processing arrays in hand, different applications can be swapped in and out of a low cost of dynamically reconfigurable computing platforms. A study is carried out on dynamical and partial reconfiguration (DPR). A DPR platform is proposed and developed for matrix computation acceleration. The DPR platform increases the hardware efficiency by dynamically

altering the circuits according to the execution requirements without interrupting the operation of the application. With the development of these generically predefined reconfigurable matrix processors and the proposed dynamically reconfigurable computing platform, the feasibility and efficiency of the scalable and reconfigurable array architectures for complex data processing is demonstrated in the thesis.

Table of Contents

List of Figures

List of Tables

Symbols and Acronyms

Abbreviations

ADP	Area Delay Product
ASIC	Application Specific Integrated Circuit
BC	Boundary Cell
CFB	Configurable Functional Block
CLB	Configurable Logic Block
CORDIC	Coordinate Rotation Digital Computer
DCM	Digital Clock Manager
DCT	Discrete Cosine Transform
DDR	Double Data Rate
DFT	Discrete Fourier Transform
DG	Dependency Graph
DRLE	Dynamically Reconfigurable Logic Engine
DSP	Digital Signal Processing
EEAS	Extend Elementary Angle Set
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FPL	Field Programmable Logic
GPIO	General Purpose Input Output

GR	Givens' Rotation
HDL	Hardware Description Language
HWICAP	Hardware ICAP
IC	Integrated Circuit; Internal Cell
ICAP	Internal Configuration Access Port
IOB	Input Output Block
IP	Intellectual Property
LFSR	Linear Feedback Shift Registers
LMB	Local Memory Bus
LUT	Look-up Table
OPB	On-chip Peripheral Bus
MAC	Multiplier And Accumulator
PAR	Place And Route
PE	Processing Element
PLD	Programmable Logic Device
PR	Partial Reconfiguration
QRD	QR Decomposition
rDPA	Reconfigurable Data Path Array
SDR	Software Defined Radio
SDRAM	Synchronous Dynamic Random Access Memory
SFG	Signal Flow Graph
SIMD	Single Instruction Multiple Data
SSR	Synchronous Set/Reset
SVD	Singular Value Decomposition
RA	Reconfigurable Array
RC	Reconfigurable Computing
RL	Reconfigurable Logic
RS	Reconfigurable System

RTR	Run-time Reconfiguration
MGS	Modified Gram-Schmidt
MIMD	Multiple Instruction Multiple Data
MIMO	Multi-Input Multi-Output
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
XDL	Xilinx Design Language

Symbols

E	Edge Set
f_r	Rate to Read Bitstream
f_w	Rate to Write Bitstream
G	Dependency Graph
K	Scale Factor
K_i	Scale Factor of The i-th Microrotation
L_c	Cell Latency
M	Period of Scheduling Table
N	Node Set
N_f	Equivalent Number of Partial bitstream
T_d	Dynamical Reconfiguration Time
δ_i	The Sign of The i-th Microrotation Angle
θ	Rotation Angle
θ_i	The i-th Microrotation Angle

Chapter 1

Introduction

1.1 Background

Recent decades have witnessed the fast growth in the silicon industry. According to Moore's Law, the density and functionality of silicon doubled every 18 months, and we are now at the stage that millions of logic gates can be easily incorporated in a fingernail size silicon area. This allows extremely complex functions to be implemented on a single chip with heterogenous structure, different types of memories, control units and processing modules which are previously unfeasible. The ability of integrating different applications at system level increases the robustness and reliability of the devices while reduces the power consumption, the cost of manufacture and the time to market. The availability of low cost, high speed and high density very large scale integration (VLSI) devices facilitates the innovations and breakthroughs in the design and application of massively parallel processors. This has driven a tremendous growth in the ever-increasing and fast changing consumer electronic markets, such as digital TV, mobile and wireless communication, computer game, etc. Consequently, low cost and time to market pressures have been increasingly added onto the designers to meet the deadlines of the new products.

With the advancement of the semiconductor technology, the gap between the number of logic gates that can be designed and verified with the current design methodology and the number of logic gates that can be manufactured by the current technology widens. To try to minimize this gap and catch up with the market demand, instead of implementing all the million-logic chips from scratch, the design methodology intends to move from gate and logic level to higher levels of abstraction. And this comes to the concept of modularity, reusability, parameterisability and reconfigurability.

As the chip size increases, interconnection problems worsen. The power consumption, chip performance and speed are determined primarily by the interconnect delay and area. To cope with this problem, modularity of building blocks and alleviation of the burden of global communication are often essential in VLSI design. By using predesigned functionally verified silicon Intellectual Property (IP) cores, the design complexity will be considerably reduced. The IP cores are modularized circuit description ranging from the low level of hardwired placed and routed layout to the high level description of soft cores such as netlist modules and Hardware Description Language (HDL) cores. These IP cores are functionally verified and are ready to be selected at system level by the designers to meet various functional requirements. For example, Finite Impulse Response (FIR) filter is commonly used in digital signal processing applications. This kind of circuit can be well developed separately by different designers but as long as they provide signal interface description and the way to customize in term of parameterisation, such as the number of taps to be used or the word-length, etc., the designers can choose this FIR IP and "glue" it with their own design. Thus instead of designing everything inside the chip, which is almost impossible for the million-gate chips nowadays, system designers tend to partition the chip into several different functional modules and select different functional IP cores to map to the target functions in term of performance, cost and power consumption and customize the cores according to the application specifications so

that each module can collaborate with other IPs inside the chip, and finally "glue" them together and pass to the test procedure for the system verification. This IP core based design flow can shorten the design time from months to days and greatly improve the design productivity.

To develop this kind of parameterisable circuit architectures, great effort is needed at the system level to deal with issues such as word-length, the number of pipeline stage, hardware mapping, folding techniques, re-timing and I/O interface, etc., before the circuit model is generated and ready to be imported and used in a system. These tasks are tedious and usually designer unfriendly especially when the circuit specifications are constantly changing. To simplify the design procedure and target for circuit reusability, the parameterisability and reconfigurability need to be considered at the IP design stage. This requires a different design approach to develop IPs that are easily parameterised at the modular level to cater for a wide range of applications with qualified solution. By keeping the application-varied parameters parameterisable, integration can be done easily at system level by instantiating these modules with the application specific parameters. Reconfigurability is another way for improving reusability. When the design of IP circuit is done, the modification of the logic part is hard to implement by the system engineers. There are some cases that the inside logic needs to be changed to suit different application requirements. So an effective way to alter the logic at system level is desirable. By designing IP circuit in a reconfigurable fashion, the reusability of the IP products is increasing.

Modern signal processing technology depends critically on the device and architectural innovations of the computing hardware. Traditional sequential processing systems are slow and the way they allocate the computing resources to application tasks limits their ability to be used in the real-time processing systems. In real-time applications, such as video or radar signal processing, the amount of data to be processed are usually very huge, and the time to process these data is always critical

because of the continuity of the data stream for the real time requirements. In such applications, general-purpose parallel computers cannot meet the processing speed due to severe overloads. VLSI array processors become an appealing alternative to gain additional computing power through their parallel processing data paths.

Parallel computers can be divided into three different classes: vector processor, multiprocessor and array processor. The first two belong to the general purpose computers, which require the complicated design of control units and resource optimization scheme at compilation, while the last one belongs to the domain of special purpose computers. The array processor exploits the parallel computing ability and shows the promising potential to meet the real-time system requirements. Digital signal and image processing algorithms are mostly dominated by the transform functions, filtering techniques and linear algebraic methods. Fortunately most of these algorithms share the same properties such as regularity, recursiveness and locality of data access. These properties are very suitable for array signal processing which possesses the structure for uniformly distributed data to be processed locally.

Parallelism, pipelining, folding and multi-array processing are the commonly used methods in the high speed signal processing to reduce the inherent complexity of large scaled array processor. To enhance the data throughput rate, extensive concurrency can be achieved by pipeline and parallel processing. By adopting pipeline techniques at all levels of the module design, the critical path which determines the speed of the data processing can be shortened, thus minimizing the total processing time. Directly mapping the signal processing algorithms to hardware is not cost-efficient especially for those that have constantly regular recurrence structure. By folding the functionally similar arrays in a time sharing fashion, the reusability of the processing arrays is increased and the hardware cost can be dramatically decreased. Multi-array processing exploits the processing parallelism within the constraints of data dependency. By allocating the data independent functions to the different ar-

ray processors in the same processing time slot, and data dependent modules to the adjacent arrays, parallelism will be maximized with no or little expense of increasing the interconnection cost.

Different architectures impose certain constraints on the applications. The ease of increasing certain types of performance will be at the expense of limiting the applicability of a system to a restricted range of applications.

Single instruction multiple data (SIMD) computer is an array processor with local interconnection and local memory access for each of the array. Data are fed into every processing elements (PE) simultaneously and the same instruction is broadcast by the host to every PE at the same time. The parallelism is often explicitly expressed in the user programs. And the system performance relies heavily on the processor compiler which extracts the parallel operations for each of the processing array and generates the code of the control unit for execution.

The very long instruction word (VLIW) computers provide a means to exploit instruction level parallelism. A single long instruction word of a VLIW architecture specifies several operations for concurrent execution on multiple functional units. The advantage is that it is a simplified design as there is no hardware support for dynamic coding. However the operation complexity has been moved to the compiler part, which primarily determines the resulting utilization and the optimal parallelism exploited.

Multiple instruction multiple data (MIMD) computers distribute the processing tasks among the processing elements to increase the processing parallelism. The algorithm mapping is usually carried out at the task level. The potential problems with this structure are the communication bottleneck within PEs as well as the synchronization issue among them. These problems may discourage the popularity of this array structure despite its flexibility in the mapping of irregular structure algorithms.

VLSI array processors such as systolic arrays are meant for data oriented algorithms. The processing arrays inside the structure are driven by the input data synchronously and in a pipelined fashion just as the heart pumping blood to all the vessels inside a body. In a systolic computing system, the function of a processor is analogous to that of a heart. Each processor regularly pumps data in and out by performing some computation inside the array so as to keep up with the pace of the processing data flow throughout the whole network, and hence the name of systolic array. The massive concurrency in systolic arrays is derived from pipeline processing, parallel processing or both. Pipeline processing means that each process is divided into a number of sub-processes, which are chained together. The data, which is to be processed by each of the processors, flows throughout the processing chain step by step. Parallel processing allows all the processes to access the processors simultaneously. This will allow all the processors to be utilized all the time.

The basic design flow for a VLSI array system is depicted in Figure 1.1. This requires a fundamental study of algorithm, architecture and application. Thus, to design a VLSI array processing system, one must acquire a broad range of knowledge including algorithm analysis, numerical analysis, array architectures, structure analysis, programming techniques and many more.

The purpose of this thesis is to study and develop a systematic approach for efficient mapping of computationally intensive linear algebraic algorithms, particularly matrix computation algorithms, onto reconfigurable VLSI array hardware with parameterisable functional modules. The objectives of this study are stated in Section 1.2.

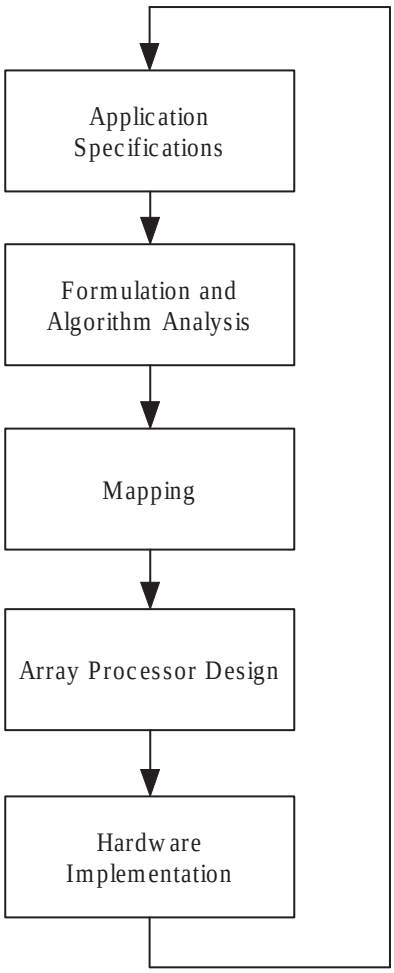


Figure 1.1: VLSI system design flow [1]

1.2 Research Motivations and Objectives

With the ever increasing demand on high speed, high throughput and short design cycle in today's digital electronic market, signal processing algorithms with high flexibility of easy transplanting in terms of reconfiguration and parameterisation to meet the requirements of different applications are highly sought after. Many digital signal processing algorithms can be decomposed into subproblems which are largely related to matrix manipulation. However, due to the storage and execution computation mode of sequential microprocessors, exponentially increasing number of computation cycles are wasted in data fetch and read back for only a single arith-

metic operation. And due to the sequential operation feature of microprocessors, millions of iterations are needed for data and instruction fetch and data storage even for the same operation and it may take several hours or even days to complete a simulation. The situation is worsened for computation intensive applications, such as matrix computation. Reconfigurable computer with its distributed architecture and parallel processing modes thus show great attraction to solve these problems.

Configurable computing is an innovative approach to the computer system design paradigm, which tries to cope with the inefficiency of conventional computing systems, due to their general orientation. It has dissolved the hard borders between hardware and software and joined the potentials of both. In recent years, through a large number of examples, configurable computing has demonstrated its advantages in performance for a range of applications. The first idea of reconfigurable hardware dates back in the nineteen sixties, though the first example of the configurable computer systems appeared in the late eighties. Most of them are based on the FPGA technology because of its great ability for field-programmability.

The dynamically reconfigurable hardware can be changed during run-time and many portions of an application can be mapped to the hardware in a sharing fashion with the potential of parallel processing. This leads to an overall improvement of reconfigurable FPGA fabric when used as hardware accelerator.

Advanced technology in IC and fast growing market in configware industry make possible the advent of million logic gates FPGA with dedicated embedded hardware multiplier and large amount of configurable on-chip RAM as well as many more useful embedded resources, even on-chip RSIC processors. This offers parallel computation and improves the efficiency for data processing.

The main objective of this thesis is to explore and apply the design methodology of reusability and reconfigurability to the development of high performance parameterisable functional modules for fast prototyping of matrix computation algorithms

on reconfigurable hardware with the applicability to cater for application-specific problems in short design time. In particular, the thesis aims to develop a set of scalable functional modules for matrix computation such as matrix multiplication and matrix inversion, as well as efficient array mapping methods to realize the complex computation requirements in matrix inversion.

1.3 Contributions

There are four main areas of contribution, as summarized below.

1. As matrix multiplication is one of the essential kernel processes in many real-time applications, the thesis studies the matrix multiplication techniques. A novel array architecture was proposed and developed for efficient matrix multiplication with versatile configurability. Based on this configuration scheme, several theorems as well as their corollaries were proposed and proved. The proposed matrix multiplication scheme was applied to the DCT algorithm to test the feasibility and the performance of the proposed architecture.

2. The thesis studies the coordinate rotation digital computer (CORDIC) algorithm for digital signal processing applications. A novel Binary-coded Z-path CORDIC (*Bi-z* CORDIC) for continuous data processing was proposed. The *Bi-z* CORDIC is both hardware size and data communication efficient for processing large amount of data, such as matrix inversion. The efficient *Bi-z* CORDIC facilitates the realization of the matrix inversion on array architecture.

3. Two novel mapping methods, namely linear and interlaced, for matrix inversion on array architecture were developed and realized, which featured linear scalable and parameterisable array architectures for hardware efficiency.

The mapping methods take advantage of the properties of the proposed *Bi-z* CORDIC and the numerical coincidence of the operations involved in the algorithm. The control and data scheduling methods for the proposed mapping methods were derived and 100% efficiency of the array processor utilization is achieved.

4. The dynamically reconfigurable platform on run-time configuration scheme was studied in detail. A run-time reconfigurable system specifically designed for applications with large data and configuration swapping was proposed, developed and implemented.

1.4 Organization

The rest of the thesis is organized as follow. Chapter 2 provides an overview of a systematic methodology for algorithm mapping and design of array processor on VLSI architecture. Chapter 3 presents the details of the proposed matrix multiplier with data reconfigurability and linear scalability. The proposed module is compared with other related designs and the 2D-DCT algorithm is presented as an application example for comparison. Chapter 4 studies the existing CORDIC architectures for signal processing algorithms. A novel *Bi-z* CORDIC is proposed and described in detail, including the implementation and comparison results. Chapter 5 presents a matrix inversion architecture with parameterisable linear mapping structure to cater for various matrix sizes, word-lengths, etc. A run-time reconfiguration platform for large data swapping and context exchange is presented in Chapter 6, in which configurable computing architectures and examples of well-known configurable computing platforms are introduced as hardware accelerators. Existing run-time reconfiguration (RTR) schemes are also studied. Chapter 7 concludes the thesis and makes some suggestions for further work.

Chapter 2

Array Architectures for matrix computation

2.1 Introduction

There are many algorithms developed for solving specific classes of problems encountered in digital signal and image processing. Algorithms are sets of well defined rules with certain sequence and correlation to solve certain types of problems in a deterministic number of steps. Most algorithms are related to manipulating some kinds of application data. There are numerous literatures exploring various aspects of computational methods for signal and image processing and scientific computations. The applications cover, to list a few, image and video based vision processing [2], [3], adaptive array processing [4], [5], audio signal processing [6], [7], biomedical applications [8], nuclear and physics studies [9], ecological and geographical applications [10], etc.

Modern digital signal processing (DSP) is categorized into image processing, matrix operation and numerical analysis. Most of these algorithms deal with large amount

of data and typically perform multiple operations in recursive manner. The way that they process the large amount of data which are typically acquired by sensors from real enviroment requires large processing bandwidth to meet the real-time requirements. The task to fulfill the ever-increasing speed and data throughput in the demand of DSP is a big challenge for today's high-speed computing technology.

2.2 Matrix Processing

Most of the DSP algorithms and applications consist of sequences of common operations or basic algorithms, which belong to the following three types:

1. Matrix operations, which include the followings [11], [12], [13].
    - Matrix-vector multiplication, matrix-matrix multiplication, matrix addition and subtraction, matrix transpose.
    - Linear systems, matrix triangularization, back-substitution, QR decomposition, matrix inversion.
    - Eigensytems, singular value decomposition (SVD), eigenvalue decomposition, spectral decomposition.
    - Solution for linear Toeplitz systems.
2. Digital signal processing operations, which cover mainly the following two categories [14], [15], [16], [17], [18], [19].
    - Filtering: FIR and IIR filter, convolution and correlation, interpolation, linear phase filtering, adaptive filtering, Kalman filtering, windowing operation, differential filtering, and so on.
    - Transformation operation: fast Fourier transform (FFT), discrete Fourier transform (DFT), Walsh-Hadamard transform, discrete cosine transform,

Hough transform, etc.

3. Image processing algorithms, which consist of the followings [2] [20].

   - Image restoration, reconstruction, enhancement and smoothing.
   - Edge detection, line detection, texture analysis and feature extraction.
   - Segmentation and geometrical operations.
   - Statistic analysis.

The image algorithms rely more on the first two operations, and to a large extent, on matrix operations, which are very common in signal processing applications. The rest of this section focuses on the matrix operations.

2.2.1 Basic Matrix Operations

Matrix computations featured by having matrix operands and/or matrix results, are frequently used in modern scientific research and engineering applications. Many applications with parallel processing algorithms show that the major computation requirements for many real-time signal processing tasks, such as beam-forming [21], [22], data compression [23], [24], [25], adaptive filtering [4], [26], etc, can be decomposed into basic matrix computations. These computations involve matrix addition, matrix-vector and matrix-matrix multiplication, matrix inversion, eigenvalue solution, linear systems and matrix decomposition.

A matrix of size m by n has $m \times n$ elements on its m rows and n columns. A vector is a special type of matrix with only 1 row or 1 column which is named as row vector or column vector respectively.

2.2.1.1 Inner Product

The inner product of two vectors a and b with the same element number n is represented mathematically as $(a, b) = \sum_{i=1}^{n} a_i b_i$, where $a = [a_1, a_2 \cdots, a_n]$ and $b^T = [b_1, b_2 \cdots, b_n]$, as shown in Equation 2.1.

$$[a_1, a_2 \cdots, a_n] \cdot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \tag{2.1}$$

2.2.1.2 Outer Product

The outer product of m-element column vector a and n-element row vector b is a $m \times n$ matrix as shown in Equation 2.2.

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \cdot \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix} = \begin{bmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & & \vdots \\ a_m b_1 & a_m b_2 & \cdots & a_m b_n \end{bmatrix} \tag{2.2}$$

where $a^T = [a_1, a_2 \cdots, a_m]$ and $b = [b_1, b_2 \cdots, b_n]$.

2.2.1.3 Matrix Multiplication

1. Matrix-Vector Multiplication: The multiplication of $m \times n$ matrix A with n-element column vector B results in a column vector of n-element in Equation

2.3.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{n} a_{1j}b_1 \\ \sum_{j=1}^{n} a_{2j}b_2 \\ \vdots \\ \sum_{j=1}^{n} a_{mj}b_n \end{bmatrix} \tag{2.3}$$

2. Matrix-Matrix Multiplication: The matrix-vector multiplication can be considered as a subset of matrix-matrix multiplication. For more general case, if A is an $m \times p$ matrix and B is a $p \times n$ matrix, the multiplication of A and B produces an $m \times n$ matrix C, which is given in Equation 2.4.

$$C = A \times B = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mp} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & & \vdots \\ b_{p1} & b_{p2} & \cdots & b_{pn} \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{j=1}^{p} a_{1j}b_{j1} & \sum_{j=1}^{p} a_{1j}b_{j2} & \cdots & \sum_{j=1}^{p} a_{1j}b_{jn} \\ \sum_{j=1}^{p} a_{2j}b_{j1} & \sum_{j=1}^{p} a_{2j}b_{j2} & \cdots & \sum_{j=1}^{p} a_{2j}b_{jn} \\ \vdots & \vdots & & \vdots \\ \sum_{j=1}^{p} a_{mj}b_{j1} & \sum_{j=1}^{p} a_{mj}b_{j2} & \cdots & \sum_{j=1}^{p} a_{mj}b_{jn} \end{bmatrix} \tag{2.4}$$

The elements of matrix C are given by Equation 2.5.

$$c_{ij} = \sum_{k=1}^{p} a_{ik}b_{kj} \tag{2.5}$$

Note that for the matrix multiplication in Equation 2.5, if the order of subscript i and j is changed, the final output product is the same, which is to say the calculation sequence, whether it is row order orientated or column

order based, has no effect on the final product. However, for hardware implementation, different data sequence may results in different array structure types, which may directly affect the architecture of the array processor. This is discussed in detail in Chapter 3.

2.2.1.4 Solving Linear Problems

Solving of linear system is of great importance to signal processing, control systems and statistic analysis, where the problem is to find n unknowns from a given set of number p ($p >= n$) equations in Equation 2.6. The problem can be formulated in mathematical notation in Equation 2.6.

$$A \cdot x = y \tag{2.6}$$

where A is a $p \times n$ matrix, x and y are n-element column vectors. The problem is solvable, if the number of unknown is equal to the number of independent equations. But in some scenarios, the problem may be tightly constrained, or theoretically there is no definite solution for the problem. In this case, linear approximation may be the best solution by solving the least squares problem [27].

For a given $p \times n$ observation data matrix, the least square values of model unknowns form an n-element vector x that minimizes the squared deviations, compared with model prediction vector y. The least square error e is defined by Equation 2.7.

$$e = \|y - A \cdot x\|^2 \tag{2.7}$$

Where $\|y - A \cdot x\|$ is the norm of the deviation vector.

The direct solving of linear problems is computation unfriendly and may result in convergence problem, say the inversion of singular matrix for example. The most

popular solution is first to triangularize matrix A, then the problems can be solved by back-substitution, as described next.

2.2.1.5 Matrix Triangularization

Matrix Triangularization is a useful method to simplify the linear system problems. The methods to triangularize a general matrix involve Gaussian elimination, QR decomposition, LU decomposition and many others [13]. This thesis focuses only on the QR decomposition method for matrix triangularization because of its robustness in numerical stability and ease of implementation on CORDIC [14] hardware structure without pivoting process by orthogonal transformation, compared with the other methods.

QR decomposition of matrix A can be written as Equation 2.8.

$$A = QR \tag{2.8}$$

where Q is an orthogonal matrix (meaning that $Q^T Q = I$) and R is an upper triangular matrix. The QR decomposition can be accomplished by a sequence of Givens' Rotation (GR) which performs plane rotation of matrix A and was proved to be unconditionally numerically stable for all nonsingular input matrices [11]. The plane rotations can be used to introduce zeros either in the columns (below the diagonal) or in the rows (above the diagonal). In the QR decomposition, matrix A is converted into upper triangular matrix by introducing zeros in the columns below the diagonal. The goal of Givens' method for matrix triangularization is to progressively eliminate the sub-diagonal elements of matrix A from first column to the last one step by step. Each Givens' rotation is to zero one subdiagonal element of matrix A. For matrix A of nonsingularity, the upper triangular matrix R is

computed by Equation 2.9.

$$Q^T A = R \tag{2.9}$$

The orthogonal transformation matrix Q^T is obtained by Equation 2.10.

$$Q^T = (Q_{n-1,n})(Q_{n-2,n}Q_{n-2,n-1})\dots(Q_{2,n}\dots Q_{2,4}Q_{2,3})(Q_{1,n}\dots Q_{1,3}Q_{1,2}) \tag{2.10}$$

where $Q_{i,j}$ represents the single plane rotation operator with the element at i^{th} row and i^{th} column against the element of j^{th} row and i^{th} column to annihilate the element on the j^{th} row and i^{th} column of the matrix A. For example, the format in Equation 2.11 is the rotation operator of $Q_{2,5}$.

$$Q_{2,5} = \begin{bmatrix} 1 & & & & & \\ & \cos\phi & & \sin\phi & & \\ & & 1 & & & \\ & & & 1 & & \\ & -\sin\phi & & \cos\phi & & \\ & & & & & 1 \end{bmatrix} \tag{2.11}$$

Where ϕ is the rotation angle.

To premultiply a given matrix or vector in the same dimension by this transformation matrix can only affect the elements in the 2nd and 5th rows. Thus these active rows can be extracted to form a temporary matrix with two rows, and the temporary matrix is then multiplied by a 2×2 Givens transformation matrix given in Equation 2.12.

$$\tilde{G} = \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} \tag{2.12}$$

The Givens rotation can be represented as the extracted matrix premultilpied by the Givens transformation matrix $\tilde{G}$. With (x_i, y_i) represents a two-dimensional vector in the x-y plane, the output vector ($x_i^{'}$, $y_i^{'}$) after applying rotation with the angle ϕ can be obtained by Equations 2.13 and 2.14.

$$x_i^{'} = x_i \cos\phi + y_i \sin\phi \tag{2.13}$$

$$y_i^{'} = y_i \cos\phi - x_i \sin\phi \tag{2.14}$$

To force a particular element $y_i^{'}$ to zero, $\phi = \tan^{-1}(\frac{y_i}{x_i})$ is chosen so that the output pair ($x_i^{'}$, $y_i^{'}$) coincides with x-axis.

Viewing the 2-row matrix as a string of pairs, the rotation angle ϕ can be decided by the leading input vector (x_1, y_1) which sets y_1 to zero. The following pairs rotate with the same angle on the same direction. The operations by first discovering the rotation angle followed by the rotation applied on the following series of pairs are used in many signal processing applications as well as in the matrix QR factorization. In such case, the CORDIC algorithm [28] shows great efficiency to perform such rotation as explained in Chapter 4.

2.2.2 The Matrix Decomposition Approaches

Implementation of matrix algorithms onto VLSI hardware requires the transformation of these algorithms into representation of regularized algorithms suitable for hardware structure. Simple matrix computations such as matrix multiplication or regular iterative algorithm is straightforward, so the lack of transformation mechanism is not a problem. However, this type of algorithms is relatively few. For many matrix algorithms the implementation procedure onto array structure requires a matrix decomposition stage. In this way, instead of focusing on each application specific solution, complex matrix computation can be reduced into relatively sim-

ple matrix manipulation steps to ease the algorithm mapping on VLSI array. This step is usually referred to as matrix decomposition. There are several approaches for matrix decomposition, but the most commonly used ones which are often referred to as the "big six" are largely adopted in the algorithm analysis [13]. They are the Cholesky decomposition, LU decomposition, QR decomposition, singular value decomposition (SVD), Schur decomposition and spectral decomposition, as summarized below.

1. The Cholesky decomposition: Given positive defined matrix A, a unique upper triangular matrix L can be found such that $A = L^T L$. The Cholesky decomposition is well derived to solve positive defined linear systems by being substituted with two steps of triangular linear systems. Since the computation of triangular system is more efficient than directly solving the linear equation, the Cholesky decomposition transforms the problem into simpler steps in which the solution is computed.

2. The pivoted LU decomposition: Matrix A of order n can be decomposed into unit lower triangular matrix L (that is matrix L is lower triangular matrix with ones on diagonal) and upper triangular U, with left and right permutation matrices P and Q as shown in $P^T AQ = LU$. Similar to the Cholesky decomposition, the application of the LU decomposition is for solving linear systems. However, LU is more generally applicable than the Cholesky approach since A is a general matrix. But as pivoting is introduced to prevent unstability of the process, it requires extra computation operations.

3. The QR decomposition: Given matrix A, there is an orthogonal matrix Q such that $Q^T A = R$, where R is an upper triangular matrix. The QR decomposition finds its role in a wide range of applications such as recursive least square and eigenvalue algorithm of numerical analysis. Basically, the computation of QR decomposition is classified into two types of method: direct and trans-

formation. Normally in the direct method, Gram-Schmidt algorithm [13] is adopted which proceeds to orthogonalize the kth column of matrix A against the first $k-1$ columns of matrix Q to get the kth column of matrix Q. The Gram-Schmidt algorithm is less computationally stable and thus the modified Gram-Schmidt (MGS) [13] is developed to solve this problem. But MGS is computation bounded.

The orthogonal transformation methods proceed by multiplying A with certain orthogonal matrices until the elements below the diagonal are all zero. And again, there are two kinds of transformation: the Householder transformation and the Given's method [13] [29]. Householder transformation [13] takes the advantage of saving memory cost by gradually replacing the elements of matrix A with the transformation matrix Q in the same place of the memory. The Given's rotation is proven to be numerically stable but has less efficiency in computation.

4. The spectral decomposition: If matrix A is a symmetric matrix of order n, then it can be reduced to diagonal matrix Λ by an orthogonal matrix V in $A = V\Lambda V^T$. The spectral decomposition can be used in applications where the eigensystem of a symmetric matrix is needed. Normally for the computing of the spectral decomposition, there are three classes of algorithms [13], the QR decomposition, the divide-and-conquer algorithm and Jacobi's rotation. The first two reduce the matrix A into tridiagonal matrix with orthogonal similarities, while Jacobi's algorithm invites Jacabi rotation which is slow but may be more accurate for positive defined matrix.

5. The Schur decomposition: Matrix A can be decomposed into upper triangular matrix T with unitary matrix U and its conjugate transpose U^H as $A = UTU^H$. The diagonal elements of T are the eigenvalues of A, which leads the application of this decomposition to eigensystem analysis. The computa-

tion of Schur form can be done by first reducing A into Hessenberg form [13] using householder transformation and Schur form can be calculated with QR algorithm.

6. The singular value decomposition: For matrix A there are orthogonal matrices U and V such that $U^T AV = \Sigma$, where Σ is a diagonal matrix. The diagonal elements of matrix Σ are called the singular values of matrix A. The algorithms computing the singular value decomposition are similar with methods for spectral decomposition. They are QR algorithm, the divide and conquer method and Jacobi-like algorithm.

The matrix decomposition algorithms for matrix processing are not confined to the above-mentioned methods. However, these algorithms are the basic and most commonly used algorithms for matrix transformation and computation which are mostly time consuming and data intensive part for real-time signal processing on VLSI hardware.

2.3 Mapping Algorithms onto VLSI Array

An application is a task to be performed. Many signal processing applications have common computation features such as matrix operations, local and recursive computation, etc. These computations share many common features in their dataflow structure. The exploration of these common features will facilitate the algorithm expression and data transformation modeling for mapping matrix processing algorithms onto array hardware structure.

The algorithm is the computational procedure used to perform the task. Traditional algorithms are written in recursive and sequential structure. For these algorithms to run on application-specific hardware architecture, certain levels of parallelism

in the algorithms must be exploited. Algorithm expression is a method to extract features such as data dependency, processing correlation as well as functional description within each step of the algorithm in such a way that space-time activities in the processing arrays can be easily discovered and processing concurrency can be maximized.

There are many methods for algorithm expression, including recursive algorithm, snapshots, space-time parallel codes, etc. Among them, dependency graph (DG) and signal flow graph (SFG) [13] are widely adopted for algorithm expression due to their visual derivation from algorithms, and structural compatibility with hardware architecture.

A dependency graph is a graph that indicates dependency of computations within an algorithm. Nodes inside a graph are called operation nodes which perform certain types of computation on the input data. Nodes are connected using lines with directional arrow, which are called edges, to indicate the dependency of the two connected operation nodes and the direction of dataflow. A graph G can be represented as $G = (N, E)$ with N for the whole set of nodes and E for the whole set of edges within the graph G. An algorithm is computable if and only if the complete graph contains no loops or cycles.

A signal flow graph (SFG) is a directed graph that shows the processing nodes, communication edges, and delays of communication edges. In a SFG, a node represents arithmetic or logic function with zero delay inside the operation. An edge, on the other hand, indicates either operation dependency or delays. A complete SFG contains both functional and structural description parts. The functional description defines the behavior of the operation nodes, while the structural part describes the interconnection, either computational dependency or delay, between the nodes. Compared with DG, SFG is more concise and specific to hardware level description. Since edges introduce delays of operation, a SFG allows loops within its structure.

The array architecture describes the implementation of the processing elements and their interconnections. Namely, there are two main types of array processors: systolic arrays and wavefront arrays. The determination of array type largely depends on the application specification and the mapping schemes.

The design flow of application-specific array architecture for different applications is depicted in Figure 2.1, of which each step is elaborated in the following subsections with the matrix-matrix multiplication as an array design example.

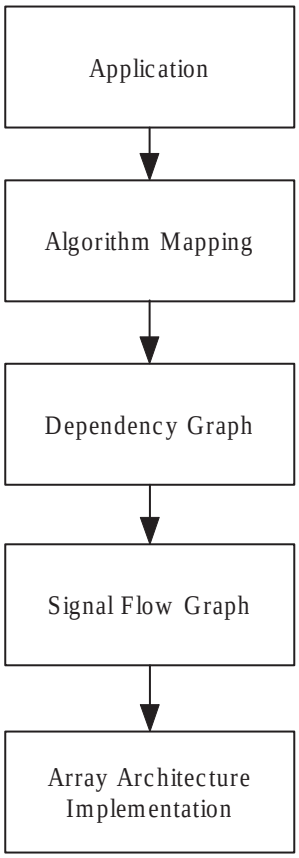


Figure 2.1: Design flow for array processor

2.3.1 Mapping Algorithm to DG

Most signal processing tasks can be formulated as regular, iterative algorithms. Take the $n \times n$ matrix-matrix multiplication of Equation 2.15 as an example.

$$C = A \times B \tag{2.15}$$

Equation 2.15 is computed with Equation 2.16.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{2.16}$$

A sequential C code for this matrix-matrix multiplication is shown in Figure 2.2. In this sequential program, each variable $c[i][j]$ is overwritten n times before the final product is available. This type of code is not orientated to map onto a concurrent array processor. To direct the mapping procedure, the sequential algorithm must first be converted to a recursive algorithm with single assignment, in which each variable is assigned a value only once. Figure 2.3 shows the C code for matrix multiplication in single assignment format.

```
for (i=0; i<n; i++){
   for (j=0; j<n; j++){
      c[i][j] = 0;
      for (k=0; k<n; k++){
         c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
   }
}
```

Figure 2.2: C program for sequential matrix-matrix multiplication

```
for (i=0; i<n; i++){
   for (j=0; j<n; j++){
      c[i][j][0] = 0;
      for (k=0; k<n; k++){
         c[i][j][k+1] = c[i][j][k] + a[i][k] * b[k][j];
      }
      c_out[i][j] = c[i][j][n];
   }
}
```

Figure 2.3: C program for sequential matrix multiplication with single assignment

To map the C code in Figure 2.3 into a DG for matrix multiplication, the concurrent specification is not complete. Array processors should also have temporal locality to store the temporal partial products. Figure 2.4 shows the C code for matrix multiplication with single assignment and temporal locality for partial product.

```
for (i=0; i<n; i++){
   for (j=0; j<n; j++){
      c[i][j][0] = 0;
      for (k=0; k<n; k++){
         if (i == 0)
            b[i][j][k] = b_in[k][j];
         else
            b[i][j][k] = b[i-1][j][k];
         if (j == 0)
            a[i][j][k] = a_in[i][k];
         else
            a[i][j][k] = a[i][j-1][k];
         c[i][j][k+1] = c[i][j][k] + a[i][j][k] * b[i][j][k];
      }
      c_out[i][j] = c[i][j][n];
   }
}
```

Figure 2.4: C program for matrix multiplication with single assignment & temporal locality

The C code in Figure 2.4 shows the elements of matrix A enter the array at $j = 0$, the dataflow of the elements a is along the j axis. The operand matrix elements b enter the array at $i = 0$ and moves along the i axis. The partial product c moves along the axis k and leave the array at $k = n$ with the final products. The formulation of the DG graph for matrix multiplication is given in Equation 2.17.

$$I_{DG} = \{[i, j, k]^T | 1 \leq i, j, k \leq n\} \tag{2.17}$$

Figure 2.5 shows the dependency graph for matrix multiplication with matrix size 4×4, where only the outer nodes and edges are depicted. In this graph, matrix A enters array along the plane $j = 0$, matrix B enters the array along the plane $i = 0$,

and zeros enter the array along the plane $k = 0$. The product matrix C is available at the plane $k = n - 1$. This result is derived directly from the matrix multiplication algorithm in Figure 2.4. Each node of the graph performs a multiplication and addition on the input data, as shown in Figure 2.6.

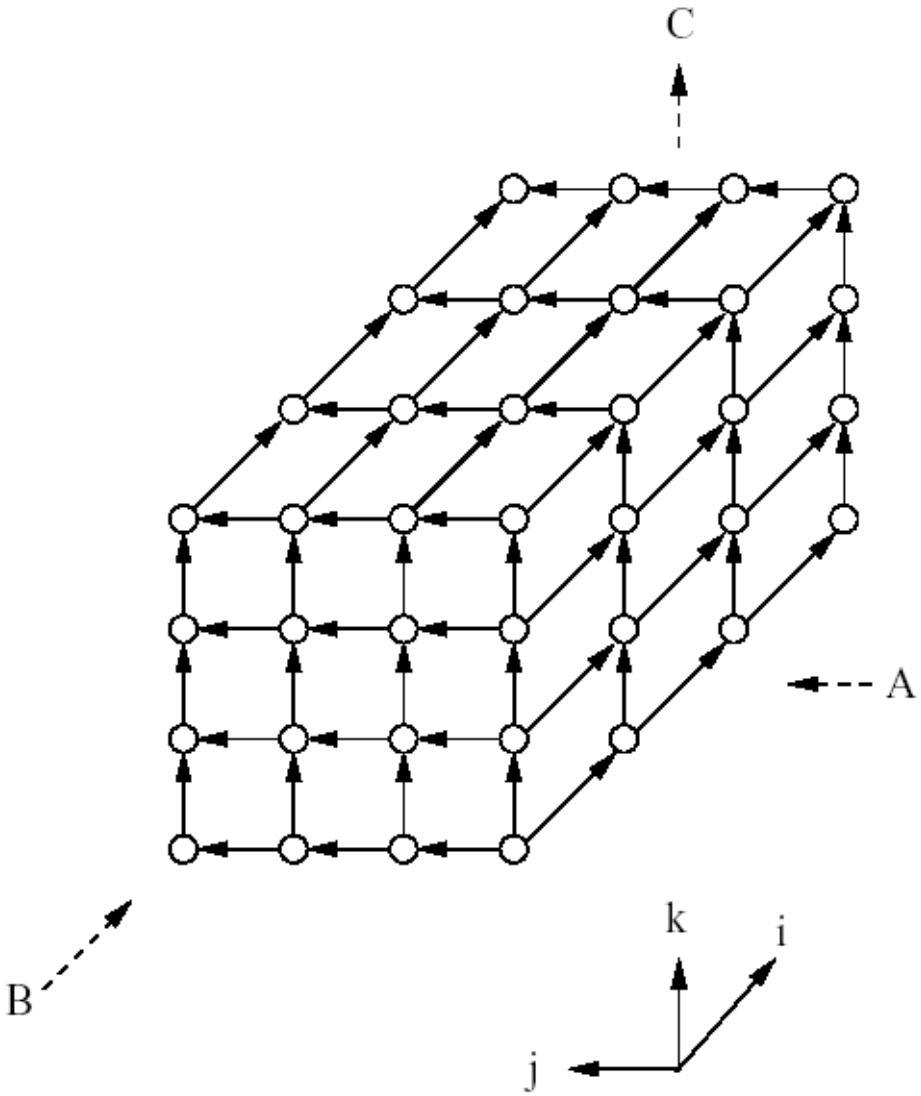


Figure 2.5: Dependency graph for matrix-matrix multiplication

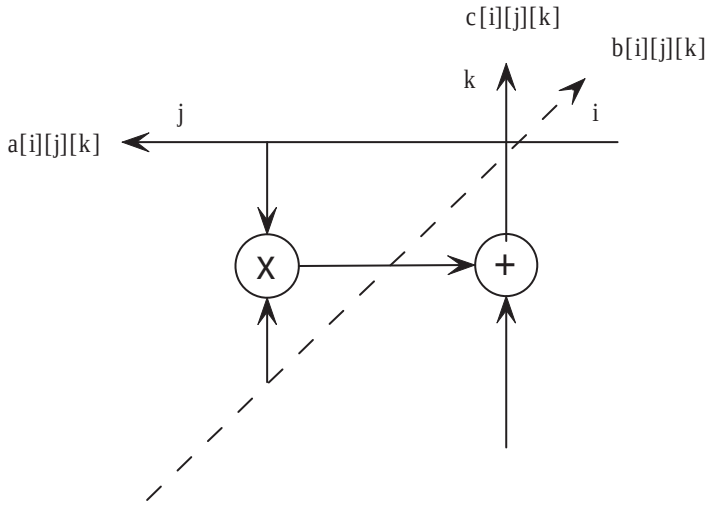


Figure 2.6: Node implementation for dependency graph of matrix multiplication

2.3.2 Mapping DG to SFG

Dependency graph depicts only the relationship of the computational nodes within the array structure. For hardware array design, more specific information needs to be exploited to model the behavior, timing, operation scheduling as well as communication details of the nodes. Basically, signal flow graph shows these kinds of hardware related implementation details in an abstract level.

There are two steps that need to be fulfilled for mapping DG into SFG, the processor assignment and the scheduling. The processor assignment groups nodes along certain partition line into a common processing element. In general, the allowable assignment methods is very broad. However, to simplify scheduling and derive a regular array structure, linear projection and linear scheduling for processor assignment may be adopted. In the linear projection, the nodes along a certain straight line are projected onto a processing element and, in the linear scheduling computation nodes on the same hyperplane of the DG will be scheduled to be executed at the same time step.

For processor assignment, normally, a projection vector **d** is used to indicate the

projection direction. In Figure 2.6, the computational nodes may be projected in either direction i, j or k. Different projection schemes may result in different scheduling and dataflow control. Suppose i is chosen as the projection direction, the resulting SFG is shown in Figure 2.7.

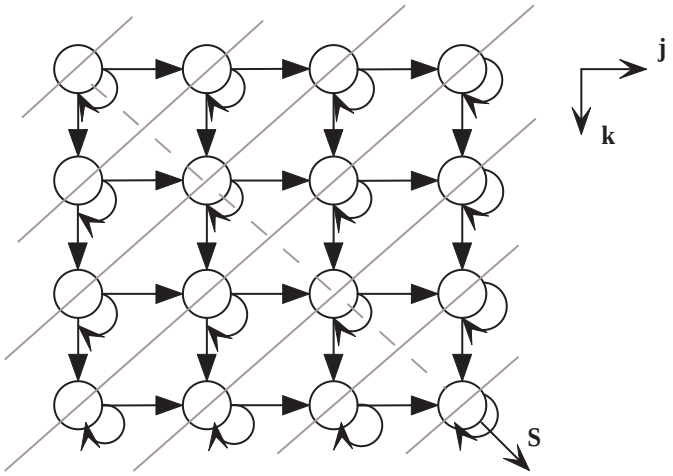


Figure 2.7: Signal flow graph for matrix-matrix multiplication

The scheduling which specifies the operation sequence of each processor is always associated with the processor assignment. Different processor assignment methods will yield different scheduling results. The linear scheduling is based on the hyperplanes, which are uniformly spaced and parallelizing with each other in the SFG as depicted in Figure 2.7 with the shallow diagonal lines. The nodes on the same hyperplane (or schedule line) are performed concurrently for each operation cycle. Mathematically, the scheduling space can be represented by a schedule vector $\mathbf{S}$, which points to the direction parallel to the normal of the hyperplanes. The schedule vector shows the time sequence of operations on the different hyperplanes.

For feasible projection and scheduling schemes, two basic rules are required to be satisfied.

1. The directions of all dependence arcs, which are also the directions of all the dataflow, are in the same direction to cross the hyperplanes.

2. The schedule lines are not parallel with the projection vector **d**.

By projecting the operational nodes in the direction k, each processor in Figure 2.7 is assumed to be preloaded with the corresponding element of matrix B, $b[k][j]$. In the DG, the elements of matrix B is passed along i in Figure 2.5 from one node to another. This transition can be modeled more concisely by the loops depicted in Figure 2.7 on each node. The nodes need to hold the partial product by the loop delay for the next cycle updating. This is called a recursive loop inside the SFG. The SFG simplifies the algorithm analysis by separating the operation and the delay. The operations represented by the functional nodes are assumed to be zero delay arithmetic. All the algorithmic delays are shown with the arcs and loops in Figure 2.7. The result is a more concise algorithmic description than the DG.

2.3.3 Mapping SFG to Array Processor

There are two types of array processors, systolic arrays and wavefront arrays [1]. The systolic arrays have the features of regularity, modularity, scalability and data locality which use synchronous global clocking. The main feature of systolic arrays is that all the data, while transiting regularly and rhythmically throughout the arrays can be efficiently processed in all PEs. A variation of systolic arrays is the semisystolic array, which allows global communication by broadcasting data among the PEs. Another type of array processor is the wavefront array. This type of arrays are often found in the situations when the number of processors or clock rates become large enough to cause troubles because of clock skew issue. The wavefront arrays take advantage of taking both control flow and data flow locally by having special handshaking capabilities, for example request and acknowledgement. But they may require extra FIFO buffers to store the data at the interface with other arrays when the data cannot be immediately processed. The choice of either kind of array

processors dependeds on the application as well as the performance requirement.

The SFG can be mapped directly onto a hardware architecture by physically assigning operations of processing elements (PEs) to each module. The scheduling of the operations should also follow the schedule requirement of the specific SFG graph accordingly. The techniques used to assign operations onto processors are the same as derivation of SFG from DG. For the SFG in Figure 2.7, each node can be assigned to one array processor, and the whole architecture is cascaded as depicted by the arcs in the SFG diagram.

Direct mapping of SFG onto hardware processor arrays may result in a complex architecture thus wasting logic resource. It is possible to perform further optimization on processor assignment and scheduling schemes to reduce the circuit complexity. The advanced techniques that can be adopted include SFG partition and folding to time-share the PEs in the circuit design. Figure 2.8 shows a simple example on how to project a signal flow graph of matrix multiplication onto linear array architecture. Different projection schemes may lead to different processor implementation and operation scheduling. In some scenarios, the processor usage can also be different. This results in efficiency design issues for processor assignment. Further mapping issues are exploited in Chapter 5 for the matrix inversion problem.

2.4 Reconfigurable Computing System

Comparable to the high performance of Application-Specific Integrated Circuits (ASICs) as well as the programmable flexibility of general-purpose processor, reconfigurable computing has become one of the ever-growing hot research topics recently. Reconfigurable systems are computing systems that combine one or more reconfigurable hardware processing units with a software-programmable processor. These systems allow customization of the reconfigurable processing units in order to meet

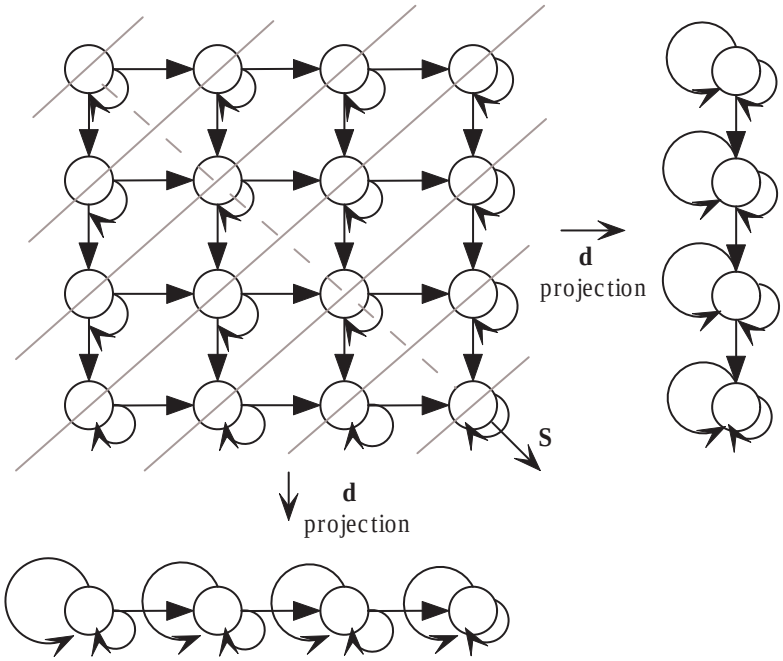


Figure 2.8: Projection of Signal flow graph onto linear arrays

the specific computational requirements of different applications. Reconfigurable computing represents an intermediate approach between the extremes of ASICs and general-purpose processors. A reconfigurable system generally has wider applicability than an ASIC. In addition, the combination of a reconfigurable component with a general-purpose processor results in better performance than a general-purpose processor alone.

Typically, reconfigurable computing machines can be grouped into different types based on different specifications: coarse grained [30], medium grained [31] and fine grained [32] by the granularity of the reconfigurable logic; static and dynamic by the reconfiguration mode (whether at the execution time or not) [33]; loosely or closely coupled with host CPU by integration [34], etc.

In Reconfigurable Systems (RS) it should be clearly distinguished between the areas of Reconfigurable Logic (RL), also called field-programmable logic (FPL), and Reconfigurable Computing (RC). RL, where the typical product name is FPGA (field

programmable gate array) or PLD (programmable logic device), deals with fine-grained reconfigurable circuits and systems. A typical fine-grained reconfigurable circuit consists of an array of CLBs (configurable logic blocks) with a path width of 1 bit, which are embedded in a reconfigurable interconnect fabrics. From an EDA point of view this RL level appears as a methodology of logic design for hardwired logic, but on the high-level design platform it appears as software resources for the custom designs, which are not really hardwired. RC, which is typically called reconfigurable arrays (RA) or reconfigurable data path arrays (rDPA), deals with coarse-grained reconfigurable circuits and systems. A typical coarse-grained reconfigurable circuit consists of an array of CFBs (configurable functional blocks), also called reconfigurable datapath unit (rDPU), with a wide path width, 16 or 32 bits, for instance. Here it can be said that, the RA granularity is 16 or 32 bits. Note that RAs may also be bundled with multiple granularity, say with medium grained rDPU slices of pathwidth W (4 bits, for example).

Another distinguished grouping method is the static and dynamic configuration strings [33], where the later turns out to be one of the favorable research topics today. A static configuration string, aiming at configuring the processor so as to perform a given function, is loaded once at the outset. After which it does not change during execution of the task at hand. Dynamic configurability (also called Run-time reconfiguration, RTR) involves a configuration string that can change during execution of the task at hand with the main advantage of eliminating the configuration latency by overlapping the execution time with configuration stages and can adapt to changing (dynamic) specifications.

Run-time reconfiguration provides a method to accelerate a greater portion of a given application by allowing the configuration of the hardware to change over time. Apart from the benefit of added capacity through the use of virtual hardware [35], [36], run-time reconfiguration also allows circuits to be optimized not only between

applications but also within a single application. However, the benefits of run-time reconfiguration may be decreased due to the heavy penalty of non-productive reconfiguration time. Many studies have carried out to accelerate reconfiguration. These include configuration prefetching [37], [38], configuration compression [39], relocation and defragmentation [37] and configuration caching [40], [41].

The use of FPGAs has been classified broadly into three main categories: rapid prototyping, system implementation and dynamically reconfigurable systems. The last of these areas has been described as being perhaps the most innovative (application) for FPGAs and still in its infancy. Traditionally if one wants to change the design of an FPGA, one would need to halt the device and download a new configuration onto the device, for which the design resided on the system is lost - Static Reconfigurability. Although this is acceptable for most uses of FPGA as glue logic, it has limited use when FPGAs are needed to behave as programmable algorithm accelerators, as the nature of the tasks that they will execute are changing. It also means that if the device configuration in the field is to be updated the whole configuration needs to be changed, which can amount to reload of few megabytes for designs containing several million gates.

Run-time reconfiguration (RTR) allows the circuit configurations to be changed while the application is still running. This provides us the acceleration potential for real-time applications. Currently, there are four typical reconfigurable models for FPGA configuration.

1. Single context devices. Actually, this model lacks of the ability to support the RTR. Any change in configuration requires complete reprogramming and the context switching (reconfiguration) time is in the order of milliseconds (i.e., Xilinx XC4000, Altera FLEX10K). So good partitioning between contexts is essential to minimize the reconfiguration latency.

2. Multicontext devices. This kind of FPGA contains multiple memories for configuration. Since it supports background loading of contexts, context switching time is in nanoseconds (potential to speed up the reconfiguration). The sample device includes Dynamically Reconfigurable Logic Engine (DRLE) from NEC [42] and CS2000 RCP from Chameleon Inc.

3. Partial reconfigurable devices. These devices allow reconfiguration of only a part of the device while the rest of the device is still executing. This gives potential to hide the reconfiguration latency and allows different configurations coexist in the same context. Xilinx XC6200, Virtex and Virtex-II belong to this type.

4. Pipeline reconfigurable devices. This type of device consists of independently configurable pipeline stages and each stage is ready to execute immediately after its programming. PipeRench [43] is the typical representative.

In order to program these FPGA devices, we need to be able to specify the configuration for each resource inside the FPGA. This specification, devised by the FPGA vendor, is called bitstream. The methods of modification of bitstream dynamically can be typically classified into two groups: the differential based and the module based, which are targeting on the designs with "small" change and "large" change at run-time respectively. This will be discussed in detail in Chapter 6.

Chapter 3

Array Processor for Matrix Multiplication

3.1 Introduction

Matrix multiplication can be found in many signal processing applications, such as encryption, computer graphics, video, robotic and control algorithms. Due to computation complexity, the processing of large matrix size could overload the processor, thus degrades the system performance.

Sequential computation is only suitable for small size matrix algorithm. For large size matrix multiplication, special purpose VLSI architectures are desirable to meet the performance requirement, due to its parallel and distributed computation in nature. Different design considerations for matrix multiplication on VLSI may lead to different designs with various hardware and resource utilization as well as performance efficiency in term of speed and data throughput. This results in an open question for further research.

Scalability and reconfigurability is another interesting issue to be considered if the

design specification changes. To meet the requirement of short design cycle to cater for different design parameters and requirements, such as matrix size, operation coefficient, etc., reconfigurable architecture for matrix multiplication with parameterisability is also worthy of consideration.

3.2 Existing Matrix Multipliers

In this section, a literature survey of several existing matrix multipliers is presented. A novel reconfigurable matrix multiplier is then proposed to meet the high performance demand of applications and demonstrates the efficiency of parallel computation for algorithms exploration on VLSI hardware compared with software design.

Figure 3.1 shows the block diagram of a 3x3 matrix multiplier from Xilinx reference design [44].

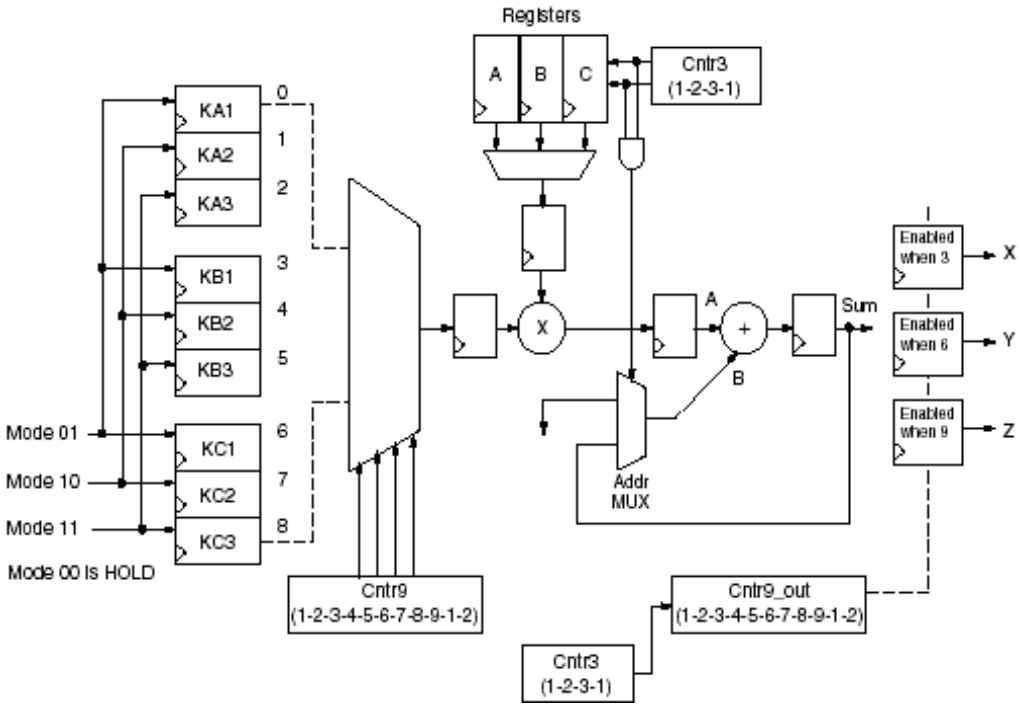


Figure 3.1: Xilinx Reference Design of 3×3 matrix multiplier

The multiplier output is fed into the A input of the adder. It takes 3 clock cycles for the first valid multiplier output to reach the A input. The B input of the adder can be a zero or the feedback value of the accumulating register of the adder. By selecting a zero on the B input, the adder passes the input A through to the accumulating register. By selecting the accumulating register, the contents of the previous operation can be added to the output of the multiplier.

The repeating flow for the accumulating register is such a sequence that in the first clock, the MUX output is '0', the first argument is always passed through to the accumulator register. For the second and third clock, the accumulator register is fed back and added to the output of the multiplication. This is made possible by using the cntr3 outputs as the select lines.

Figure 3.2 shows the multiply accumulator (MAC) module generated by Xilinx Core Generator [45].

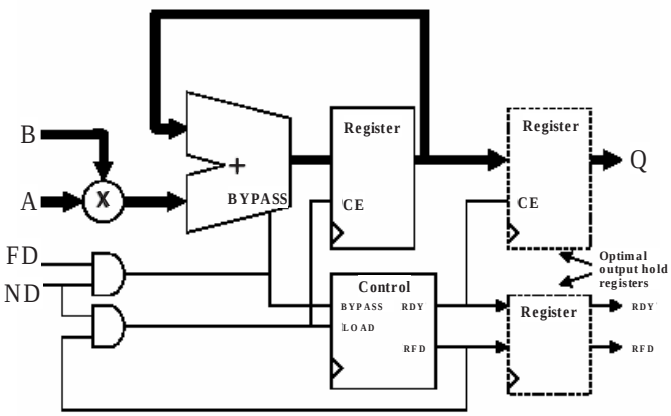


Figure 3.2: MAC by Xilinx Core Generator

The overall function of the Multiply Accumulator (MAC) is given in Equation 3.1.

$$Q = \sum_{n=0}^{count-1} \pm A(n) \times B(n) \tag{3.1}$$

where Q is the primary data output of the core. A and B are multiplied together

and the product added or subtracted from the current result. The Uniprocessor produces one element of the product of the $n \times n$ matrix multiplication in $n + 2$ cycles and it takes $(n + 2) \times n^2 + 1$ clock cycles to obtain the matrix product.

Another matrix multiplication approach is to use Linear Array. Figure 3.3 shows the Linear Array for matrix multiplication proposed in [46].

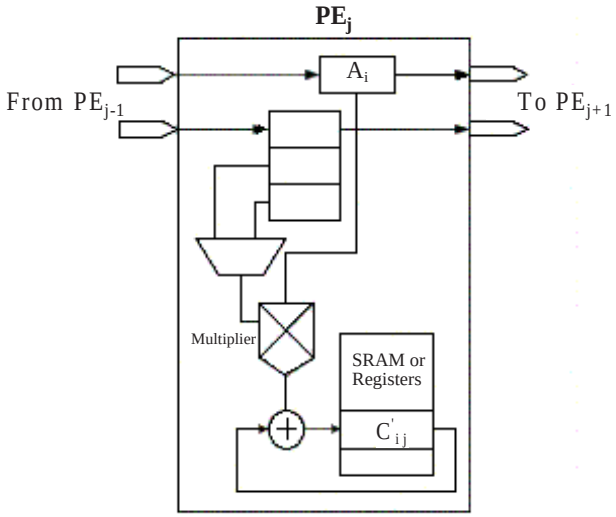


Figure 3.3: Architecture of PE_j Proposed in [46]

The architecture in Figure 3.3 is devised to compute $C_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$ for all i, j. a_{ik}, b_{kj}, and c_{ij} represent the elements of the $n \times n$ matrices A, B and C. The whole architecture for the $n \times n$ matrix multiplication is the cluster of this PE_j with every PE_j produces the j column of the product. Matrix B is fed to the lower I/O port of PE_1 in row major order (b_{11}, b_{12}, b_{13}, ..., b_{1n}, b_{21}, b_{22}, ...). Matrix A is fed to the upper I/O port of PE_1 in column major order (a_{11}, a_{21}, a_{31}, ..., a_{n1}, a_{12}, a_{22}, ...), n cycles behind the matrix B. Once b_{kj} arrives at PE_j, a copy of b_{kj} resides in PE_j until a_{nk} arrives.

3.3 Proposed Matrix Multiplier

Today's FPGA devices provide large amount of memories. For example, Xilinx Virtex-II devices incorporate many 18 Kbit Block SelectRAM modules with versatile configuration options. The memory cells can be used by the matrix operand in the same way as parameterisable registers. Based on the linear array architecture, the proposed matrix multiplier uses two memory blocks. Figure 3.4 shows the architecture of the proposed processing element (PE). Memory B stores the column j of matrix B and memory C is used in the same way as the registers C_{ij} in the Linear Array to store the partial and final products of column j. Compared with the previous techniques, the proposed design significantly reduces the number of registers involved in data movement. $4n$ registers are involved in the data movement in the Linear Array, while only n registers are used in the proposed design. In the Linear Array, $n^2 + 2n$ cycles are needed for computing $n \times n$ matrix multiplication. With run-time configurable parameters and parallel processors, it can save n cycles in the proposed systolic mode design and $2n$ cycles in the parallel mode as explained below.

Based on the proposed architecture, a number of theorems are derived to show the performance of the proposed architecture. Theorem 1 gives the minimum latency requirement of $n \times n$ matrix multiplication with n MACs (multiplier-and-accumulators) and uniprocessor. Theorem 2 improves the Linear Array algorithm for matrix multiplication with respect to both the number of registers and the number of computation cycles. Theorem 2 is extended in Corollary 1 and 2 for demonstration of the ability of the proposed design to meet the latency constraint with n and one MAC hardware resource respectively. Theorem 3 introduces the matrix decomposition for matrices with size lager than the PEs can handle at one time and gives the quantitative analysis of the trade-offs between area and latency.

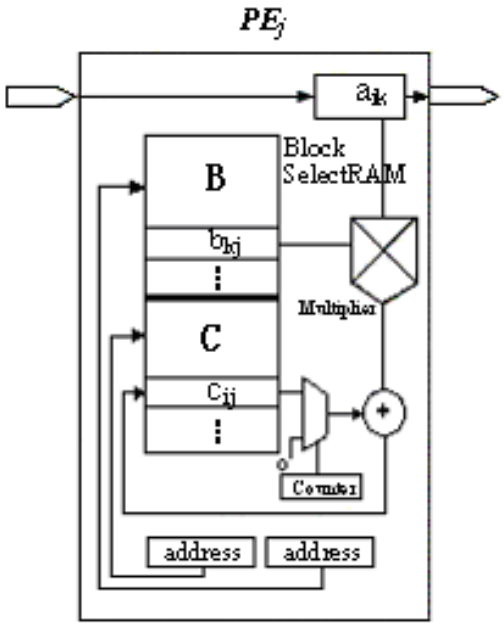


Figure 3.4: Architecture of PE_j in the Proposed Design

Theorem 1 The multiplication of two $n \times n$ matrices can never be performed in less than n^2 cycles with n multipliers or n^3 cycles with one multiplier.

Proof: $O(n^3)$ is defined as the complexity for the multiplication of two $n \times n$ matrices. Equation $C_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$ denotes the calculation of any multiplication of two $n \times n$ matrices $C = A\dot{B}$ with a_{ik}, b_{kj}, and c_{ij} represent the elements of $n \times n$ matrices A, B and C respectively. n times of multiplication are needed to produce each element of product C. Thus, to compute the whole n dimension C, n^3 times of multiplication are needed. If n multipliers working in parallel, n^2 cycles are the latency elapsed in multiplication. Note that the latency is not counted for addition in pipelined processing and those wasted in data movement. So the minimum timing requirement for an matrix multiplication with matrix size $n \times n$ is n^2 cycles with n multipliers, and n^3 cycles with one multiplier for the same reason.

Theorem 2 The multiplication of two $n \times n$ matrices can be performed in $n^2 + n$

cycles using n PEs each with a MAC (multiplier-and-accumulator), a register and a Block SelectRAM of $2n$ words per PE and 1 input and 1 output port.

Proof: Figure 3.4 is devised to compute $c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$ for all i, j. a_{ik}, b_{kj}, and c_{ij} represent the elements of $n \times n$ matrices A, B and C. PE_j denotes the j-th PE in the whole structure. PE_j computes column j of matrix C, c_{1j}, c_{2j}, $\cdots$, c_{nj} stored in Block SelectRAM C part. The input of PE_j connects to the output of PE_{j-1} and the output of PE_j is the input of the next array element PE_{j+1}. In Phase k, row k of matrix A (a_{ik} , $1 \leq i \leq n$) traverse PE_1 , PE_2 , PE_3 ,$\cdots PE_n$ in order. Column j of matrix B resides in the Block SelectRAM of PE_j that can be dynamically reconfigured. This way allows PE_j to update $c_{ij}^{`} = c_{ij}^{`} + a_{ik} \times b_{kj}$ every clock cycle, where $c_{ij}^{`}$ represents the intermediate value of c_{ij}. And it takes n cycles to calculate each element of matrix C. The MAC in PE_j will not start until the first element of matrix A arrives. Thus, PE_j starts computing j cycles after the ready signal activates, and completes in $j + n^2$ cycles. So the result is available after the last element c_{nn} in PE_n is computed which is after the $(n^2 + n)$ th cycle.

Corollary 1 The multiplication of two $n \times n$ matrices can be performed in n^2 cycles using n PEs each with 1 MAC, 1 register and 1 Block SelectRAM of $2n$ words per PE and 1 input and 1 output port.

Proof: No change is carried out in *Theorem 1* except the way matrix A traverses. Instead of through PE_1, PE_2, PE_3, $\cdots\cdots$, PE_n, the elements of matrix A are traveling through data bus and fed into each PE simultaneously. All the PEs are instantiated with the parameters of PE_1 but different column value of matrix B in Block SelectRAM part B - PE_j with j-th column of matrix B. This method allows all the PEs start at the same time and finish with the latency of PE_1 as in the *Theorem 2*.

Corollary 2 The multiplication of two $n \times n$ matrices can be performed in n^3 cycles using 1 PE with 1 MAC, 1 register and a Block SelectRAM of $2n^2$ words per

PE and 1 input and 1 output port.

Proof: Same thing happens to Uniprocessor. The matrix multiplication of two $n \times n$ matrices can also be performed only using PE_1. The value can be parameterized in Block SelectRAM part B with matrix B in column order. Matrix A is fed into PE_1 n times with the production rate of 1 column per time. So n^3 cycles are needed in this case.

Theorem 3 The multiplication of two $n \times n$ matrices can be performed in rn^2 cycles using n/r PEs each with 1 MAC, 1 register and 1 Block SelectRAM of $2n$ words per PE and 1 input and 1 output port where n is divisible by r.

Proof: The multiplication of two $n \times n$ matrices can be decomposed into $r^3 n/r$ matrix multiplications. Using *Corollary 1* with n replaced by n/r, the result follows. The matrix operand management would be like this: Matrix A is fed in with major sequence of the row of sub-matrix, and minor sequence of row order in each sub-matrix; Matrix B resides on the Block SelectRAM, with major order in the column of sub-matrix and minor order of each column within the sub-matrix. For example, if an matrix multiplication of two 8×8 matrices is decomposed with factor of $r = 2$, the matrices can be manipulated in the arrow sequence as shown in Figure 3.5.

Block SelectRAMs of PEs are configured in the order as shown in Figure 3.5 (b). The way that Matrix A is fed is illustrated in the following pseudo code:

for $major\_row\_count = 1$ to r do
  for $major\_row = 1$ to r do
    for $major\_column = 1$ to r do
      for $minor\_row = 1$ to n/r do
        for $minor\_column = 1$ to n/r do
          $a_{ik} = A_{ij}$

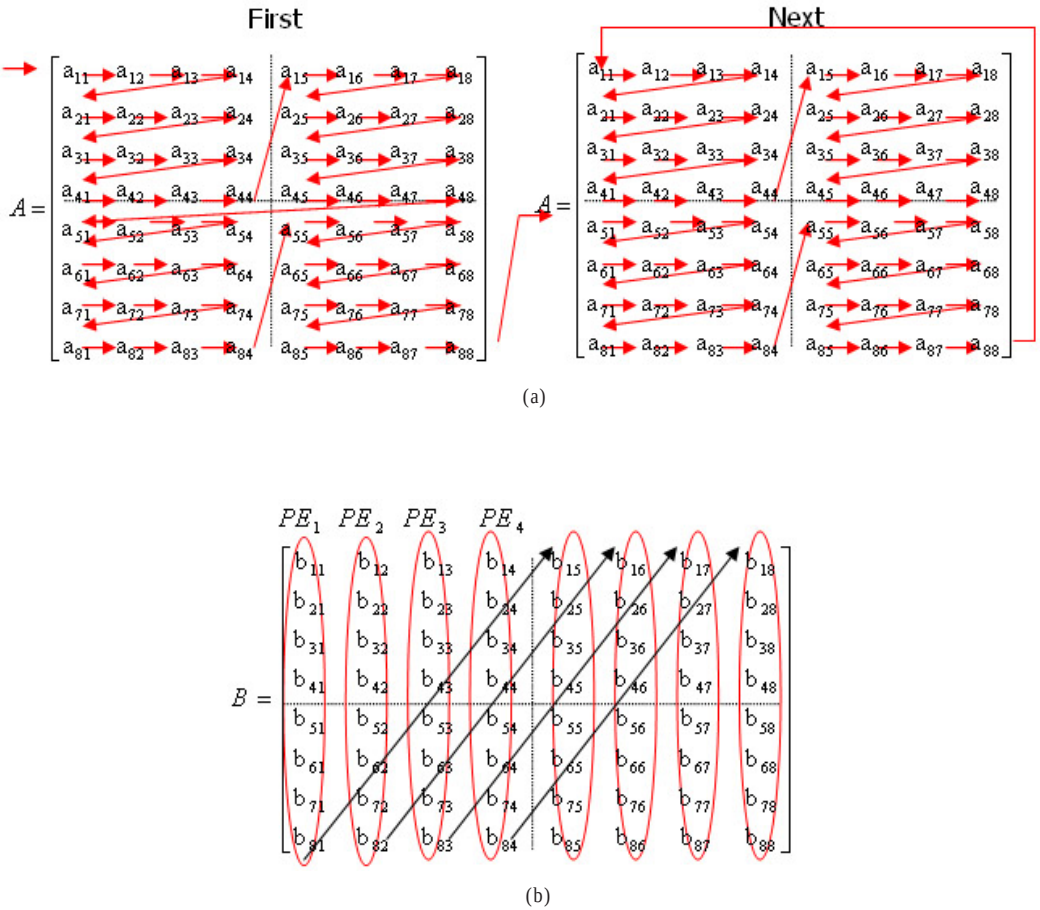


Figure 3.5: Decomposition of Matrix Multiplication in the Proposed Data-Configuration Scheme

Where a_{ik} is the register of a_{ik} in Figure 3.4 and A_{ij} is the current element of matrix A ready for feeding in.

Theorem 3 provides trade-off between area and latency. A smaller value of n/r reduces the number of PEs resulting in lesser area. However, it increases the number of cycles to complete the matrix multiplication.

3.4 FPGA Implementation

The matrix multipliers described above were implemented on Xilinx Virtex-II device and their performance in terms of area and latency metrics was evaluated.

The performance equation is defined as $Perf = n^3/(slices \times Latency)$. By including area in the performance evaluation, it takes into account the effect of increased numbers of processing elements and the area differences for various types of memory. This is especially relevant in an era of deep pipelines and huge caches where small performance improvements are bought at the cost of dramatic increases in area. The operation per area-second performance measurement provides a better indication for selection of design styles.

Table 3.1 shows the different matrix multiplication modules with various area and latency tradeoff. Note that the module by Xilinx Core Generator [45] and by Xilinx reference design [47] use single multiplier so the area is the same for all matrix sizes.

Figure 3.6 shows the performance evaluation of the 3 existing designs against the proposed one for various size of matrix multiplication. The performance equation shows the significant improvement over the existing modules when both area and latency are taking into consideration.

Since the initial design step and simulation test of the proposed design is using MATLAB, the comparison of matrix multiplication with MATLAB is necessary. This time, only the latency is considered. Table 3.2 shows the significant speedup of FPGA implementation of matrix multiplication when compared with MATLAB. It also shows the better speed improvement when compared with Linear Array design.

The hardware setups to run the matrix multiplication are as follows:

MATLAB host : Intel Pentium(r) 4 2.4GHz, 512MB DDR, 40 HDD

FPGA : Xilinx Virtex-II xc2v4000-ff1152-4, 300MHz

Table 3.1: Comparison of 3 existing designs with the proposed design for various size of matrix multiplication

(a) Area Comparison

Matrix Size	Xilinx (slices)	CoreGen (slices)	LinearArray (slices)	Proposed (slices)
3 × 3	207	158	393	123
6 × 6	207	158	786	246
9 × 9	207	158	1179	369
12 × 12	207	158	1572	492
15 × 15	207	158	1965	615
24 × 24	207	158	3912	984
48 × 48	207	158	9360	1968

(b) Latency Comparison

Matrix Size	Xilinx (cycles)	CoreGen (cycles)	LinearArray (cycles)	Proposed (cycles)
3 × 3	45	45	16	13
6 × 6	360	288	49	43
9 × 9	1215	891	100	91
12 × 12	2280	2016	169	157
15 × 15	5625	3825	256	241
24 × 24	23040	14976	625	601
48 × 48	184320	115200	2401	2353

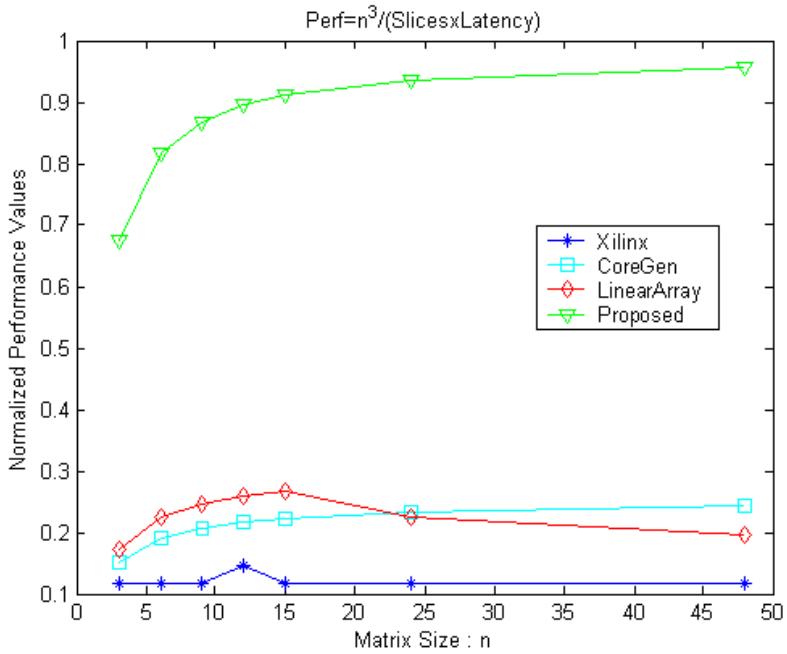


Figure 3.6: Performance Evaluation of Matrix Multiplication with Various Size

The comparison results of the matrix multiplication are also shown graphically in Figure 3.7 and Figure 3.8. The great speedup compared with Linear Array design is largely due to the reason that the frequency achieved is much higher than Linear Array design given Xilinx Virtex-II FPGA as the same targeting platform.

Theorem 2 and its corollaries show the ability of the proposed design to be configured as different types of processor according to different specific requirements. By instantiating the column parameter all with the first column number, a cluster of vector multipliers can be obtained that compute in parallel and can achieve a maximum speedup in any kind of n-processor mode with computation cycles of n^2. Each vector multiplier can also be used individually as a uniprocessor for matrix-vector multiplication. Table 3.3 shows the latency of the proposed module when configured as Uniprocessor, Systolic Array and the Optimal Parallel module.

Figure 3.9 shows the Area-Latency tradeoffs as mentioned in Theorem 3 with n=3, 24 and 48 respectively. For matrix of other sizes, the trend remains the same. It

Table 3.2: Matrix Multiplication Latency and Speedup Compared with MATLAB

Matrix size	MATLAB (cpu time/ s)	LinearArray(o)/Speedup (μs/%) 55MHz		Proposed/Speedup (μs/%) 166MHz	
3 × 3	2.059	0.291	707.6%	0.060	3432%
6 × 6	3.112	0.892	348.9%	0.223	1396%
9 × 9	4.378	1.818	240.8%	0.494	886.2%
12 × 12	7.167	3.076	233.0%	0.873	821.0%
15 × 15	10.32	4.659	221.5%	1.361	758.3%
24 × 24	26.75	11.38	235.1%	3.476	769.6%
48 × 48	202.1	43.70	462.5%	13.89	1455%
120 × 120	2258	266.5	847.3%	86.75	2603%
240 × 240	16450	1057	1556.3%	347.0	4741%
360 × 360	57770	2371	2436.5%	780.7	7400%
480 × 480	134100	4211	3184.5%	1387	9668%

Table 3.3: Latency for Various Size of Matrix Multiplication in Versatility of the Proposed Module

Matrix size	Proposed module configured as:		
	Uniprocessor (cycles)	Systolic Array (cycles)	Parallel (cycles)
3 × 3	28	13	10
6 × 6	217	43	37
9 × 9	730	91	82
12 × 12	1729	157	145
15 × 15	3376	241	226
24 × 24	13825	601	577
48 × 48	110593	2353	2305

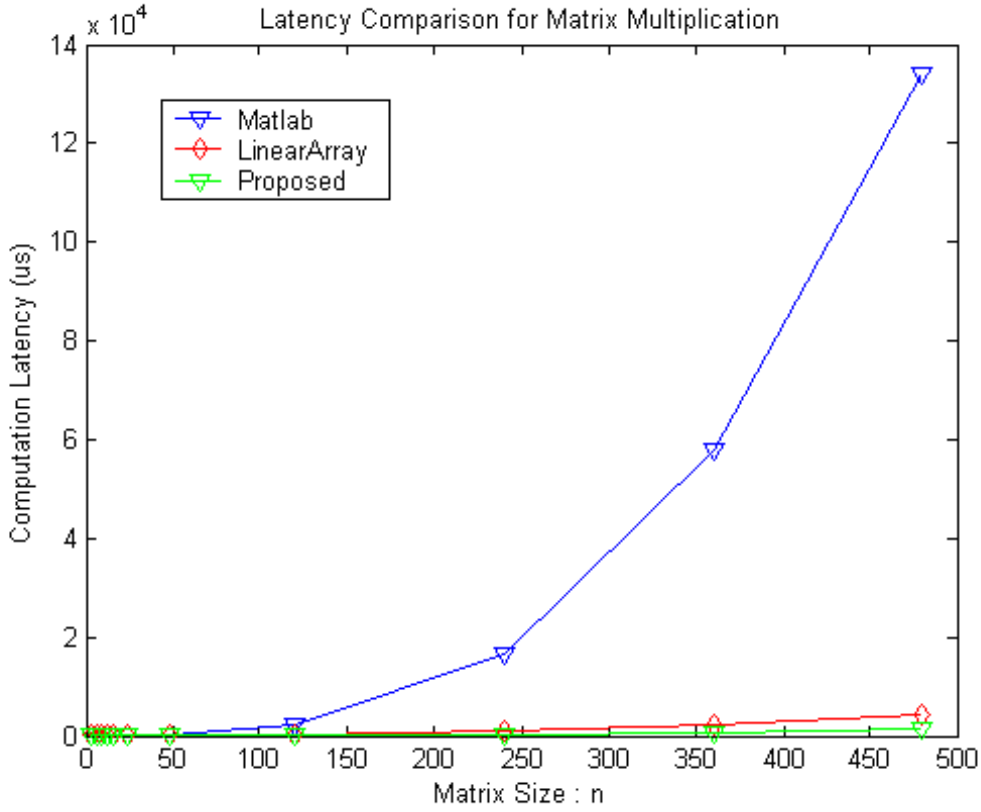


Figure 3.7: Latency Comparison of Matrix Multiplication

can be seen from the curve of Figure 3.9 that the metrics Area-Latency are actually inversely proportional.

3.5 An Application Example

Matrix multiplication is a kernel operation in many signal processing algorithms, such as DCT, IDCT, FFT, etc.

The following example is used to demonstrate the efficiency of the proposed module to accelerate the DCT algorithm. Let's take a brief look at the DCT algorithm first. The two-dimensional DCT of an M-by-N matrix A is defined in Equation 3.2.

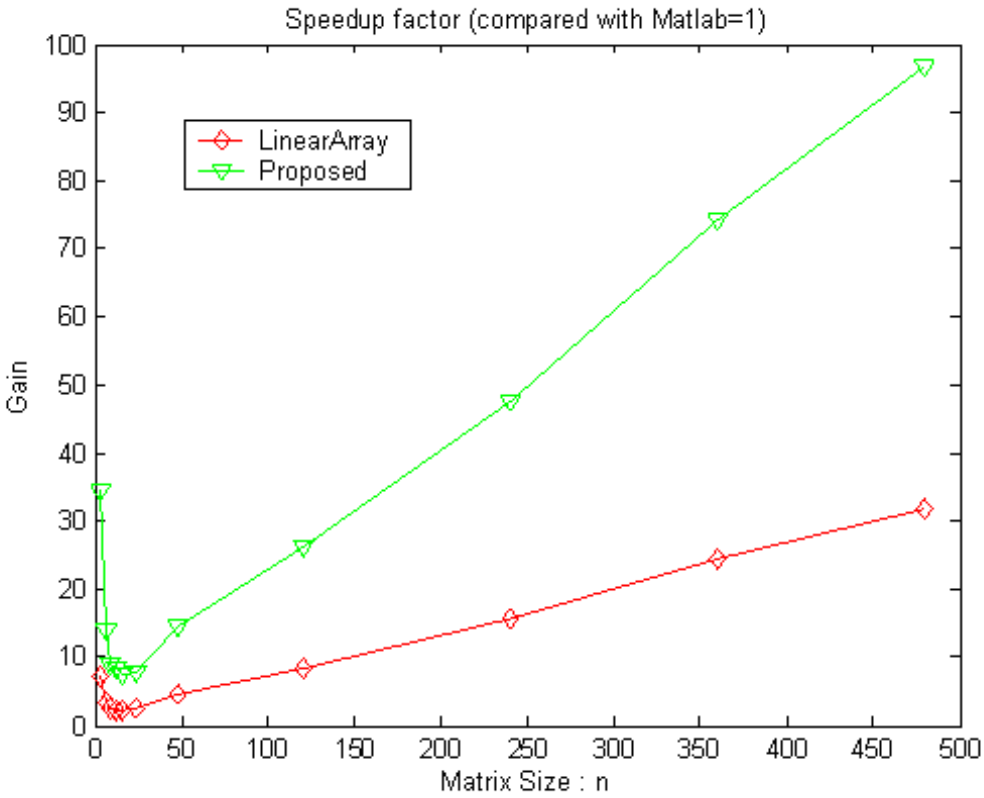


Figure 3.8: Speedup Comparison

$$XC_{pq} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} XN_{mn} \cdot \cos \frac{\pi(2m+1)p}{2M} \cdot \cos \frac{\pi(2n+1)q}{2N} \tag{3.2}$$

First, the 1D DCT of the rows is calculated and then the 1D DCT of the columns is calculated. The 1D DCT coefficients for the rows and columns can be calculated by separating Equation 3.2 into the row part and the column part, as given in Equation 3.3 and 3.4 respectively.

$$C = K \bullet \cos \frac{(2 \cdot col\ number + 1) \bullet row\ number \bullet \pi}{2 \bullet M} \tag{3.3}$$

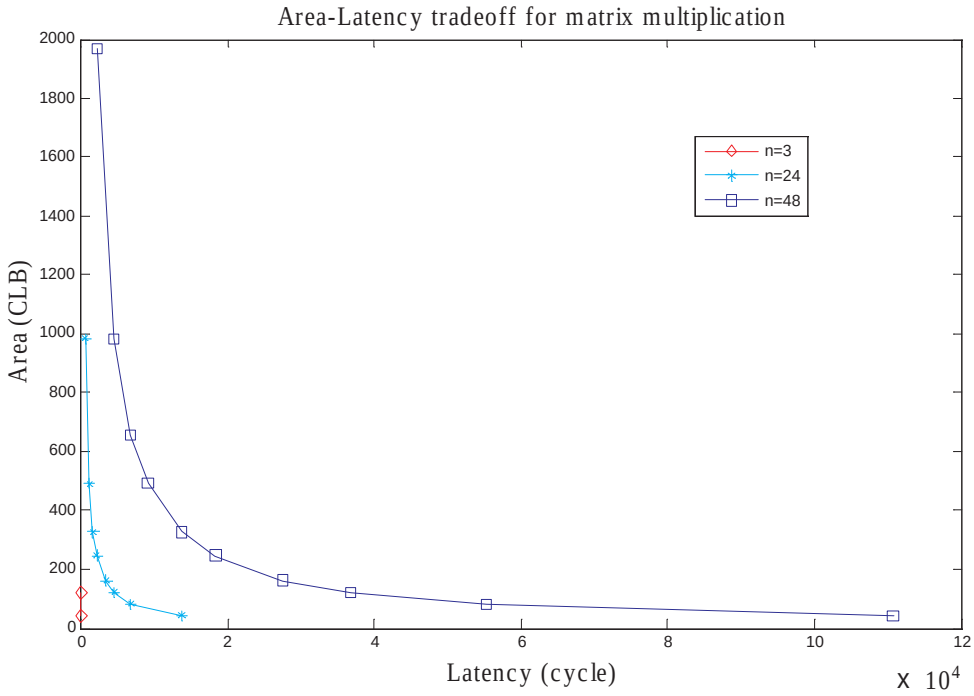


Figure 3.9: Area-Latency Tradeoffs for Matrix Multiplication

where

$$K = \sqrt{\frac{2}{N}} \quad \text{for row} \neq 0$$

$$K = \sqrt{\frac{2}{N}} \quad \text{for row} \neq 0$$

$$C^t = K \bullet \cos \frac{(2 \cdot row\ number + 1) \bullet col\ number \bullet \pi}{2 \bullet N} \tag{3.4}$$

where

$$K = \sqrt{\frac{2}{M}} \quad \text{for col} \neq 0$$

$$K = \sqrt{\frac{2}{M}} \quad \text{for col} \neq 0$$

and M = total number of columns, N = total number of rows.

The constant values for C and C^t calculated from Equations 3.3 and 3.4 are as

follows:

$$C = \begin{bmatrix} 23170 & 23170 & 23170 & 23170 & 23170 & 23170 & 23170 & 23170 \\ 32138 & 27246 & 18205 & 6393 & -6393 & -18205 & -27246 & -32138 \\ 30274 & 12540 & -12540 & -30274 & -30274 & -12540 & 12540 & 30274 \\ 27246 & -6393 & -32138 & -18205 & 18205 & 32138 & 6393 & -27246 \\ 23170 & -23170 & -23170 & 23170 & 23170 & -23170 & -23170 & 23170 \\ 18205 & -32138 & 6393 & 27246 & -27246 & -6393 & 32138 & -18205 \\ 12540 & -30274 & 30274 & -12540 & -12540 & 30274 & -30274 & 12540 \\ 6393 & -18205 & 27246 & -32138 & 32138 & -27246 & 18205 & -6393 \end{bmatrix}$$

$$\mathrm{C^t} = \begin{bmatrix} 23170 & 32138 & 30274 & 27246 & 23170 & 18205 & 12540 & 6393 \\ 23170 & 27246 & 12540 & -6393 & -23170 & -32138 & -30274 & -18205 \\ 23170 & 18205 & -12540 & -32138 & -23170 & 6393 & 30274 & 27246 \\ 23170 & 6393 & -30274 & -18205 & 23170 & 27246 & -12540 & -32138 \\ 23170 & -6393 & -30274 & 18205 & 23170 & -27246 & -12540 & 32138 \\ 23170 & -18205 & -12540 & 32138 & -23170 & -6393 & 30274 & -27246 \\ 23170 & -27246 & 12540 & 6393 & -23170 & 32138 & -30274 & 18205 \\ 23170 & -32138 & 30274 & -27246 & 23170 & -18205 & 12540 & -6393 \end{bmatrix}$$

Using vector processing, the output Y of an 8×8 DCT for input X is given by $Y = \mathrm{C} \cdot X \cdot \mathrm{C^t}$, where C is the cosine coefficients and $\mathrm{C^t}$ is the transpose coefficients. This equation can also be written as $Y = \mathrm{C} \cdot Z$ for 2-DCT, where $Z = X \cdot \mathrm{C^t}$ for 1-DCT. Using the cosine C and inverse cosine $\mathrm{C^t}$ numbers in Equations 3.3 and 3.4, the intermediate value $Z = X \cdot \mathrm{C^t}$ can be calculated, which gives the potential for execution overlapping.

Table 3.4 shows the speed comparison of the matrix multiplication in DCT when the DCT algorithm is implemented both in the proposed module, the Linear Array and Xilinx reference design [47]. The test input is an 8×8-pixel picture and the

Table 3.4: Matrix Multiplication Comparison in 2D-DCT

2D-DCT (8×8)		MATLAB (cpu time/μs)	Xilinx reference design (μs/cycle)	Linear Array (μs/cycle)	Proposed (Parallel) (μs/cycle)
DCT	1-DCT	3.406e+3	0.789 / 142	1.47 / 81	0.392 / 65
	2-DCT		0.789 / 142	1.47 / 81	0.392 / 65
Total time		3.406e+3	0.872 / 157	4.15 / 228	0.741 / 123

proposed module runs at 166MHz. It demonstrates the potential advantages to use this data-configured module for parallel computation with the speedup factor of about 4,600 times as compared with MATLAB. By instantiating the block RAM as parameterisable register, the latencies for read and write of intermediate data produced are eliminated. Further speedup can be achieved by execution overlapping of the consequential temporal configurations that use the dual-port RAM for data access.

Chapter 4

Bi-z CORDIC for Signal Processing

4.1 CORDIC Background

CORDIC (COordinate Rotation DIgital Computer) was first introduced by Jack Volder [28] as a highly efficient, low complexity and robust technique to solve real-time navigational problems. CORDIC is designed for computation of elementary functions, especially trigonometric functions, multiplication, division and datatype conversion. It was further expanded to deal with hyperbolic functions by Walther [48]. It is a class of shift-and-add algorithms to rotate vectors in a planar domain. By incrementally rotating the input vectors with specific angles selected for each step, elementary functions can be calculated after convergence of the rotations. Initially designed for military use, CORDIC has found its way in a wide range of applications including pocket calculators, digital coprocessors and high performance radar signal processing. With today's fast-increasing semiconductor industry, the CORDIC algorithm with its performance advantage and hardware efficiency, has become an appealing VLSI arithmetic units for application-specific hardware design.

Digital signal processing algorithms such as FFT, filtering and numerical conversion require high data throughput as well as hardware efficiency. The CORDIC algorithm is an attractive solution.

4.2 Basic CORDIC Algorithms

Given a vector $v(x, y)$, the resulting vector $v^{'}(x^{'}, y^{'})$ after rotating the vector v with an angle θ in counter-clockwise direction can be expressed in Equations 4.1 and 4.2.

$$x^{'} = x\cos\theta - y\sin\theta \tag{4.1}$$

$$y^{'} = y\cos\theta + x\sin\theta \tag{4.2}$$

By taking out the common coefficient $\cos\theta$, Equations 4.1 and 4.2 can be rewritten as Equations 4.3 and 4.4.

$$x^{'} = \cos\theta(x - y\tan\theta) \tag{4.3}$$

$$y^{'} = \cos\theta(y + x\tan\theta) \tag{4.4}$$

Considering a successive vector rotation with a rotation angle θ_i for the i-th rotation step. The updated vector after the i-th iteration can be calculated with Equations 4.5 and 4.6.

$$x_{i+1}^{'} = \cos\theta_i(x_i^{'} - \delta_i \cdot y_i^{'}\tan\theta_i) \tag{4.5}$$

$$y_{i+1}^{'} = \cos\theta_i(y_i^{'} + \delta_i \cdot x_i^{'}\tan\theta_i) \tag{4.6}$$

Where $\delta_i = \{1, -1\}$. δ_i is the sign of the rotation angle, indicating the rotation direction, with positive for counter-clockwise and negative for clockwise rotation.

The total accumulated rotation angle θ after the i-th rotation iteration is $\Sigma\delta_i\theta_i$.

From the hardware point of view, by choosing the rotation angle $\theta_i = \tan^{-1}(2^{-i})$, the rotation Equations 4.5 and 4.6 are expressed as Equations 4.7 and 4.8.

$$x_{i+1}^{'} = \frac{1}{k_i}(x_i^{'} - \delta_i \cdot y_i^{'} 2^{-i}) \tag{4.7}$$

$$y_{i+1}^{'} = \frac{1}{k_i}(y_i^{'} + \delta_i \cdot x_i^{'} 2^{-i}) \tag{4.8}$$

where $k_i = \frac{1}{\cos\theta_i}$. If the coefficient k_i is omitted and the computation of the angle residue is taken into consideration, Equations 4.9 to 4.11 can be obtained.

$$x_{i+1}^{'} = x_i^{'} - \delta_i \cdot y_i^{'} 2^{-i} \tag{4.9}$$

$$y_{i+1}^{'} = y_i^{'} + \delta_i \cdot x_i^{'} 2^{-i} \tag{4.10}$$

$$z_{i+1}^{'} = z_i - \delta_i \cdot \theta_i \tag{4.11}$$

From Equations 4.9 to 4.11, it can be seen that the computation only involves shift and addition/subtraction operations that are friendly for hardware implementation. By comparing Equations 4.7 and 4.8 with Equations 4.9 and 4.10, it is noted that the computation in the later parts introduces scaling effect, which is represented as $K = \prod_{i=0}^{n-1} k_i$ after the n-th rotation iteration. The magnitude factor K is calculated in Equation 4.12, and is defined as the scale factor of the CORDIC rotation.

$$\begin{aligned} K &= \prod_{i=0}^{n-1} k_i \\ &= \prod_{i=0}^{n-1} \frac{1}{\cos\theta_i} \\ &= \prod_{i=0}^{n-1} \sqrt{1 + \tan^2\theta_i} \end{aligned} \tag{4.12}$$

The basic CORDIC algorithm with an accuracy of n fractional bits on a radix-2

based system is defined in the unified Equations 4.13, 4.14 and 4.15. There are two basic CORDIC modes, namely the rotation mode and the vectoring mode. The rotation mode is to rotate the input vector by a given angle, and the vectoring mode is used to determine the angle between the given vector and the positive x-axis. Both of the modes are based on the same primitive operations and can be used to perform different computations.

$$x_{i+1} = x_i - m \cdot \delta_i 2^{-i} y_i \tag{4.13}$$

$$y_{i+1} = y_i + \delta_i 2^{-i} x_i \tag{4.14}$$

$$z_{i+1} = z_i - \delta_i \theta_i \tag{4.15}$$

where $\delta_i \in \{1, -1\}, i = 0, 1, 2, \ldots, n-1$. The variable $m \in \{1, 0, -1\}$ defines different operations as listed in Equation 4.16.

$$m = \begin{cases} 1, & \theta_i = \tan^{-1} 2^{-i} & circular \\ 0, & \theta_i = 2^{-i} & linear \\ -1, & \theta_i = \tanh^{-1} 2^{-i} & hyperbolic \end{cases} \tag{4.16}$$

In the rotation mode the desired rotation angle ϕ is given for an input vector $(x, y)^T$. The initial values are set to x_0=x, y_0=y and $z_0 = \phi$. The addition/subtraction operation of the i-th micro-rotation is selected as $\delta_i = sign(z_i)$, where θ_i is defined in Equation 4.16 i.e, $\delta_i = +1$ if $z_i \geq 0$ and $\delta_i = -1$ if $z_i < 0$.

In the vectoring mode, the rotation direction δ_i is determined by the sign of x_i and y_i, $\delta_i = -sign(x_i) \cdot sign(y_i)$, which intends to drive the input vector (x, y) towards x-axis with a magnitude $\sqrt{x^2 + y^2}$. The initial z_0 is set to 0, so after n iterations z_n contains the total accumulated rotation angle in negative value given by Equation 4.17.

$$z_n = -\sum_{i=0}^{n-1} \delta_i \theta_i \tag{4.17}$$

Table 4.1: Functions computed by CORDIC algorithm

m	mode	initial	output	function
1	rotation	$x_0 = x$ $y_0 = y$ $z_0 = \theta$	$x^{'} = K \cdot (x\cos\theta - y\sin\theta)$ $y^{'} = K \cdot (y\cos\theta + x\sin\theta)$ $z^{'} = 0$	sin cos
1	vectoring	$x_0 = x$ $y_0 = y$ $z_0 = \theta$	$x^{'} = K \cdot sign(x) \cdot \sqrt{x^2 + y^2}$ $y^{'} = 0$ $z^{'} = \theta + \tan^{-1}(\frac{y}{x})$	$\sqrt{\ }$ $\tan^{-1}$
0	rotation	$x_0 = x$ $y_0 = y$ $z_0 = z$	$x^{'} = x$ $y^{'} = y + x \cdot z$ $z^{'} = 0$	$\times$
0	vectoring	$x_0 = x$ $y_0 = y$ $z_0 = z$	$x^{'} = x$ $y^{'} = 0$ $z^{'} = z + \frac{y}{x}$	$\div$
-1	rotation	$x_0 = x$ $y_0 = y$ $z_0 = \theta$	$x^{'} = K \cdot (x\cosh\theta - y\sinh\theta)$ $y^{'} = K \cdot (y\cosh\theta + x\sinh\theta)$ $z^{'} = 0$	sinh cosh
-1	vectoring	$x_0 = x$ $y_0 = y$ $z_0 = \theta$	$x^{'} = K \cdot sign(x) \cdot \sqrt{x^2 - y^2}$ $y^{'} = 0$ $z^{'} = \theta + \tanh^{-1}(\frac{y}{x})$	$\sqrt{\ }$ $\tanh^{-1}$

The beauty of the CORDIC algorithm lies in the ability to compute up to two elementary functions at the same time. The summary of the function pairs performed by the CORDIC algorithm is depicted in Table 4.1, given arguments x, y, z, and minor modification to the CORDIC architecutre. These functions are commonly used in many signal processing applications, including discrete transform [49] [50], filtering [51] [52] and direct digital synthesis for communication systems [53].

4.3 Existing CORDIC Architectures

Extensive research work has been carried out on the CORDIC algorithm, its applications and its hardware implementations over the past decades. The conventional CORDIC algorithm is represented in sequential shift-and-add operations which are simple for hardware implementation. However, the main drawback with this structure is that it has a large number of iterations which are relatively slow and impede the high performance of the computation.

Many works on the CORDIC algorithm have been carried out to reduce the computation latency of the iterations. One approach is to introduce the redundant number representation, carry-save or signed-digit for example, to save the carry propagation delay. Ercegovac [54] employs the radix-2 CORDIC with redundant signed-digit adders to alleviate the carry propagation delay. This method greatly speeds up the computation but also increases the complexity of the scaling calculation. To eliminate the overhead of the scaling operations that are inevitable in the conventional CORDIC algorithm, several approaches have been developed. They include the constant scale factor CORDIC [55], [56] and the mixed-scaling-rotation CORDIC [57].

The redundant CORDIC [54] is efficient to save carry propagation delay in the rotation mode, but it suffers from extra logic and delay for the conversion between conventional number and redundant number representations. Moreover, for the vectoring mode angle calculation, it has to detect the sign of the redundant representation in every iteration. The detection process is slow for the redundant CORDIC which yields high latency on the overall processing time. Several approaches have been developed to accelerate the decision of the rotation directions. P-CORDIC [58] computes the direction of the required micro-rotation before the start of CORDIC computation and stores those rotation angle of each iteration inside a ROM, which

yields large memory consumption. In the hybrid CORDIC algorithm [59], the angle prediction is divided into two parts: the conventional process and the ROM-based angle prediction. The execution time for the angle prediction is reduced to only one third of the conventional CORDIC at the expense of extra ROM logic. The hybrid CORDIC is a compromise between P-CORDIC and the conventional CORDIC algorithm. The Double Step Branching CORDIC [60] executes two rotations in a single step at the price of a much complicated angle computation data-path where several significant digits are examined at each iteration to determine the sign of the residual angle. Juang proposes Para-CORDIC [61] to reduce the critical path delay on the determination of the rotation directions of the conventional CORDIC. The binary-to-bipolar recoding and micro-rotation angle recoding techniques are adopted which parallelize the prediction process for all the iterations. Para-CORDIC achieves both the lowest computation delay as well as the structural simplicity among those reported in the paper, but it is only dedicated to CORDIC in the rotation mode providing the rotation angle is known in advance.

Another solution to minimize the computation latency is to reduce the number of iterations. A table-lookup based approach is proposed in [62] to accelerate the convergence rate of the angle rotation process. Wu [63] proposes the design of extended elementary angle set (EEAS) CORDIC algorithm, in which an extended elementary angle set is defined and Trellis-based searching schemes are used to determine the optimal iteration number and the number of iteration is reduced significantly. CORDIC with high radix number representation, radix-4 for example or even higher radix arithmetic, is also proposed to reduce the number of iterations [64], [65], [66]. The trade-off is that the complexity of the selection function and the compensation of the scale factor is also increasing.

The CORDIC architectures mentioned above are for the CORDIC in either the rotation mode or the vectoring mode for angle computation. But there are some sit-

uations that the rotation and vectoring operations need to be carried out in sequence. For example, for many signal processing algorithms, one of the key properties is to first derive the parameters with the leading input variables and then to apply the results to the following input signals, such as adaptive filtering algorithms [4], and matrix computations [13]. In these scenarios, the explicit calculation of the rotation angle can be exempted if the CORDIC algorithm can be operated in both the vectoring and the rotation mode. This is where the *Bi-z* CORDIC which is developed in this work becomes useful.

4.4 *Bi-z* CORDIC Architecture

The *Bi-z* CORDIC is operated in the situations where the parameters of the rotation angle is first calculated based on the leading input vector and the derived rotation directions are then applied to the following input vectors. The concept is to eliminate the arithmetic complexity of the architecture as well as to save the interconnection cost between adjacent processors by taking advantage of the sequential coherence properties in signal processing algorithms.

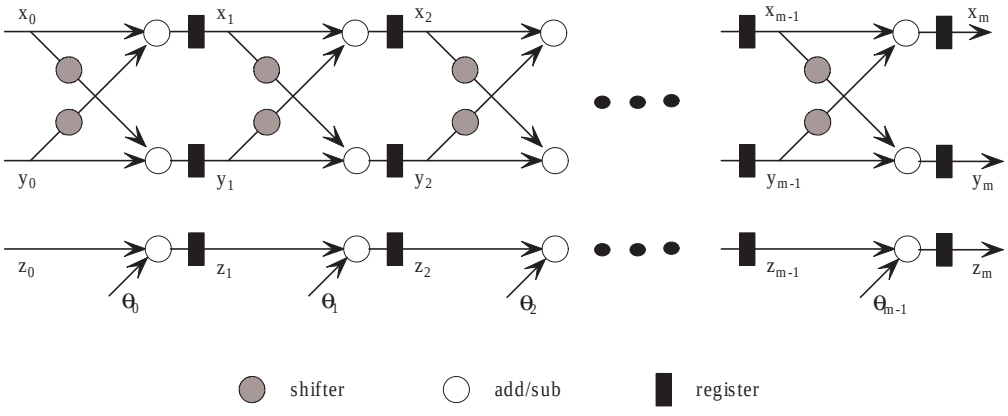


Figure 4.1: The architecture diagram of the conventional CORDIC

Figure 4.1 shows the simplified conventional CORDIC architecture. There are three addition/subtraction operators within each micro-rotation step as depicted with the

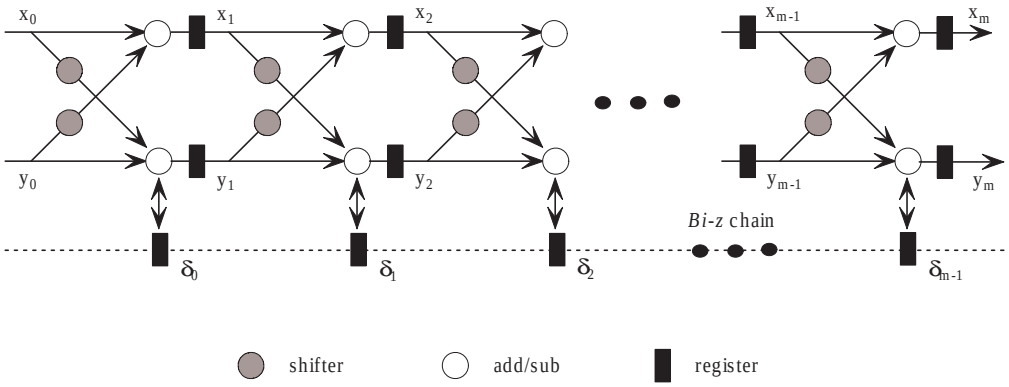


Figure 4.2: The architecture diagram of the *Bi-z* CORDIC

white circles. For the conventional CORDIC in the rotation mode, the Z-path chain performs angle quantization based on the residual angle from the previous micro-rotation step, and the rotation direction of the current micro-rotation is determined after the elementary angle calculation of the previous step. That is to say, the vector computation in the X/Y datapath can only be carried out after the angle quantization of the previous micro-rotation step. If the conventional CORDIC is in the vectoring mode, the angle formation is performed according to Equation 4.17 which is performed after the vector computation in the X/Y datapath of the previous micro-rotation step. The key point of these two operation modes is to develop the rotation direction of each step, either clockwise or counter-clockwise. The rotation direction δ_i is determined either by the sign of x and y in X/Y datapath of each step for CORDIC in the vectoring mode, or the sign of z in the Z-path for the rotation mode. Both of these modes need angle calculation, either quantization or formation. In terms of hardware utilization, the Z-path in each micro-rotation step contains an adder/subtractor. Moreover, the latency of the conventional CORDIC is relatively large, since there exists a computation dependency from X/Y datapath to Z-path within each micro-rotation step, or vice versa.

For the conventional CORDIC in signal processing algorithms, such as matrix decomposition discussed in Section 2.2, where the parameters derived by the leading

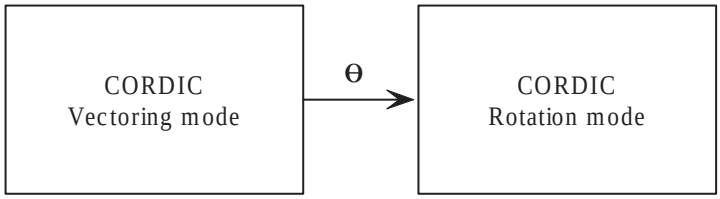


Figure 4.3: Signal processing with angle passthrough in conventional CORDIC

input vector are applied to the following vector signal, separate CORDIC processors need to be used. As depicted in Figure 4.3, the first processor is used to derive the rotation angle from the input signal and the rotation information θ is passed to the second CORDIC processor where the angle is factorized and the rotation direction is extracted for the following vectors to be processed. If the CORDIC engine can perform in both the vectoring mode and the rotation mode, these two separate CORDIC modules can be combined together. In this case, the two hardware modules can share one CORDIC engine, thus the hardware resources are saved, the interconnection for passing rotation angle is skipped, and the angle computation and factorization circuitry inside these two CORDIC modules are also eliminated, which are not possible to achieve in the conventional CORDIC algorithms.

The architecture of the *Bi-z* CORDIC is illustrated in Figure 4.2. In this structure, there is no adder/subtractor in each micro-rotation of the Z-path. The conventional Z-path chain is replaced with a 1-bit register in each step. In the vectoring mode, the input vector is driven in line with the x-axis. Instead of computing the rotation angle with the rotation direction of each micro-rotation step, the summation of each elementary rotation angle is replaced by a series of the binary rotation direction δ_i and stored inside the register of each micro-rotation step. Thus the formation of the explicit rotation angle can be completely exempted. For the *Bi-z* CORDIC in the rotation mode, which is using the rotation angle immediately derived from the previous rotation mode, the rotation angle is not necessary to go through the angle quantization process and extract the rotation direction from it. The rotation

direction is already stored inside the 1-bit register of each step and ready to be used. The computation latency introduced by the angle quantization can also be eliminated in the *Bi-z* CORDIC.

4.4.1 *Bi-z* CORDIC in Circular Coordinate

The *Bi-z* CORDIC can perform both vectoring and rotation operations. For the *Bi-z* CORDIC operating in the circular coordinate, the rotation angle is first binarized with the leading input vector $V(x, y)$ in the vectoring mode. The rotation direction for each micro-rotation is stored in the 1-bit register of each step. For the CORDIC algorithm in the circular coordinate, the input vector is transformed through a series of elementary rotations by the micro-rotation angles. The total rotation angle accumulated is only determined by a series of rotation direction δ_i in Equation 4.17. For the vector $V^{'}(x^{'}, y')$ to rotate the same angle through the CORDIC architecture, the only thing is to keep the rotation direction δ_i the same for each step, because the absolute rotation angle $\tan^{-1} 2^{-i}$ of i-th micro-rotation is fixed. After the elementary rotation sequence with the same rotation direction δ_i, the input vector $V^{'}$ is transformed to a vector with angle increased by θ and norm scaled by factor K, where

$$\theta = tan^{-1}(\frac{y}{x}) - \varepsilon \tag{4.18}$$

with $\varepsilon < tan^{-1}2^{-n}$ for input data with precision n, and

$$K = \prod_{i=0}^{n-1} \sqrt{1 + \tan^2 2^{-i}} \tag{4.19}$$

For most of the cases $\theta \approx tan^{-1}(\frac{y}{x})$, and ε is called the angle quantization error because of data representation precision limit.

Figure 4.4 shows the rotation trajectory for the *Bi-z* CORDIC in the circular co-

ordinate. The elementary rotation directions are derived in Figure 4.4 (a) when performing the vectoring operation. Figure 4.4 (b) shows the following input vector $V_0^{'}$ to perform exactly the same rotation by adopting the same rotation directions δ_i derived in Figure 4.4 (a) as the rotation guides. As can be seen that the *Bi-z* CORDIC seamlessly combines the two separate sequential operations into one unified structure that would be carried out in different hardware modules for all the previous known CORDIC architectures.

4.4.2 *Bi-z* CORDIC in Linear Coordinate

For the *Bi-z* CORDIC operating in the linear coordinate, $m = 0$, $\theta = 2^{-i}$, and x is kept constant $x_{i+1} = x$ throughout the rotations. Thus for the vectoring mode, the objective is to drive the input vector $V(x_0, y_0)$ in line with the positive half x-axis. From Equations 4.14 and 4.15, Equations 4.20 to 4.22 can be obtained.

$$0 = y_0 + \sum_{i=0}^{n-1} \delta_i 2^{-i} x_i \tag{4.20}$$

$$y_0 = -\sum_{i=0}^{n-1} \delta_i 2^{-i} x_0 \tag{4.21}$$

$$z_n = -\sum_{i=0}^{n-1} \delta_i 2^{-i} \tag{4.22}$$

So the division of y_0/x_0 is equal to the total accumulated rotation angle after n iterations, which is given in Equation 4.23.

$$y_0/x_0 = -\sum_{i=0}^{n-1} \delta_i 2^{-i} = z_n \tag{4.23}$$

If the *Bi-z* CORDIC is operating in the rotation mode, where the rotation angle is defined from angle quantization in the previous vectoring mode. The rotated angle is the same as in the vectoring mode for the following input vector $V_0^{'}(x_0^{'}, y_0^{'})$ as

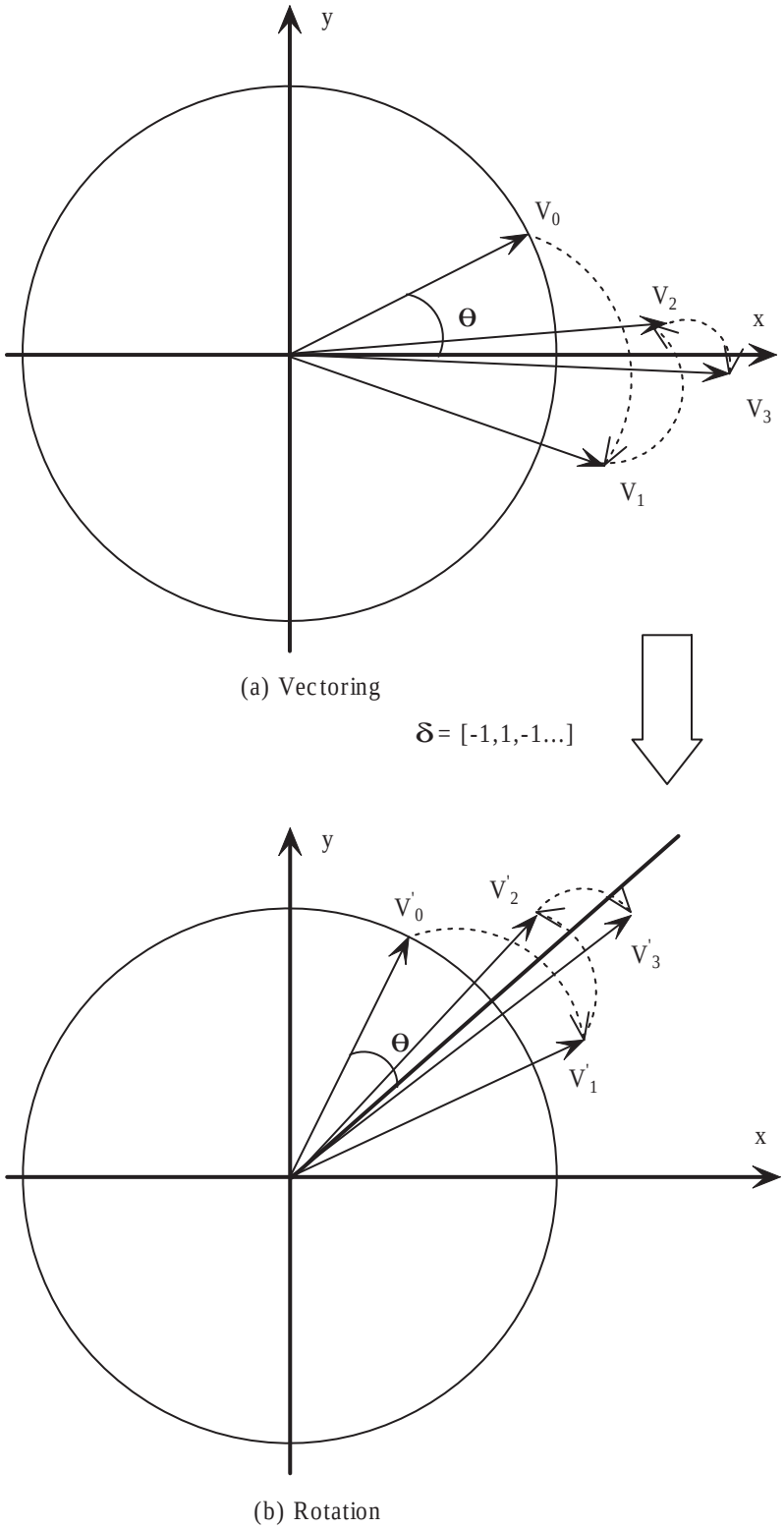


Figure 4.4: *Bi-z* CORDIC in circular coordinate

shown in Figure 4.5. The output $V_n^{'}(x_n^{'}, y_n^{'})$ is represented in Equation 4.24.

$$y_n^{'} = y_0^{'} + x_0^{'} \times \theta \tag{4.24}$$

Where θ is the total rotation angle and is equal to y_0/x_0 in Equation 4.23.

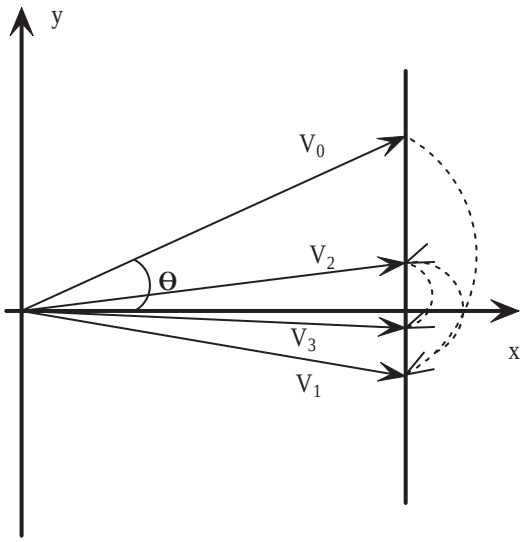
4.4.3 Scaling Factor

The scaling factor introduced in the CORDIC algorithm is the result of unbalanced weighting on the x and y coordinates to facilitate the implementation on the binary computer system. This is a tradeoff for the linear vector rotation on hardware. The compensation of the possible variable scale factor needs multiplications on the X/Y datapath which generally introduces extra latency for the CORDIC computation. As shown in Equation 4.12, the value of the scale factor K is determined by the total number of micro-rotations carried out, which is equal to the data precision n. Generally the data representation is known in advance, which defines the scale factor. So the compensation value is fixed for all CORDIC processors with the same data precision. The compensation multipliers can be deployed either prior to the start of rotation process or after the process.

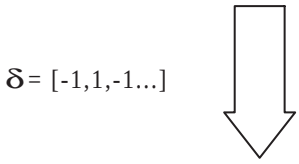
4.5 Implementation Results

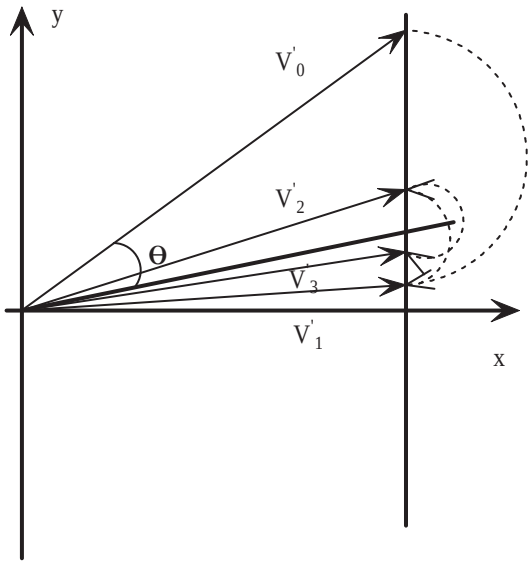
The proposed *Bi-z* CORDIC was implemented in Xilinx Virtex-II FPGA in different configurations: word-parallel and bit-serial, pipelined and recursive architectures, for different specification requirements.

Among these configurations, the bit-serial recursive structure consumes the least hardware resource. The proposed *Bi-z* CORDIC engine, implemented in the bit-serial mode, contains two dual-port RAMs for X/Y datapath and one bit-serial



(a) Vectoring





(b) Rotation

Figure 4.5: *Bi-z* CORDIC in linear coordinate

single port RAM to store the rotation direction δ, two bit-serial adders, and 3 linear feedback shift registers (LFSR) for RAM address generation. The function generator inside the Xilinx FPGA slice can be programmed as the dual-port RAM, which is efficient in hardware resource utilization. By properly sequencing the second address using LFSR, the dual-port RAM emulates a shift register and increments the address by 1 after every micro-rotation cycle. Since for the CORDIC algorithm, it is the rotation direction for each step that matters, there is no need to know and calculate the explicit rotation angle value. Thus one adder can be saved by using only one bit-serial distributed shift register for the storage of binary rotation direction for each step. The bit-serial *Bi-z* CORDIC engine shown in Figure 4.6 uses only 37 slices and can run at a bit rate of up to 170MHz, when implemented in a Xilinx XC2V4000-4 device.

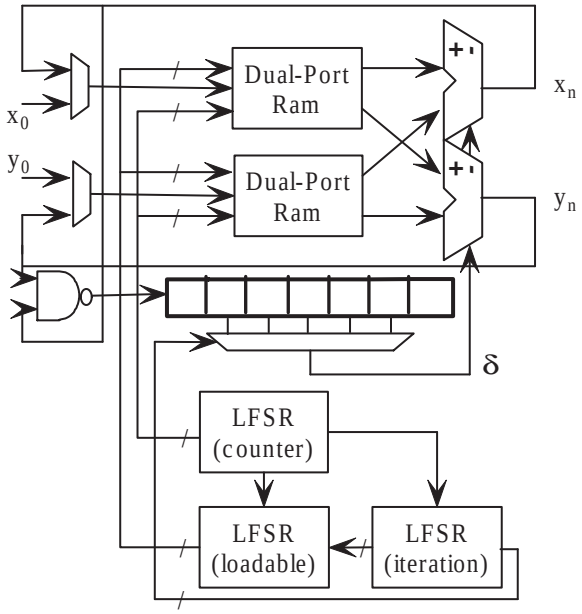


Figure 4.6: *Bi-z* CORDIC configured as recursive bit-serial engine

A comparison has been made for 16-bit implementation in Table 4.2 with Xilinx LogiCORE IP design [67] with two different configurations: parallel and recursive, which mean the operations of shift-addsub for each bit of accuracy can be either

parallel or serial. The Xilinx CORDIC design can also be configured in either rotation mode or vectoring mode, but not both at the same time. However, the *Bi-z* CORDIC can take care of both operation modes in a single design module. The serial-pipelined design achieves the best operation speed but the computation latency for serial arithmetic is long in nature, thus impedes the data throughput significantly, usually scaled down by the word precision number. The parallel-pipelined design takes up the largest logic resource yet achieves the highest throughput. As a highlight of hardware efficiency, recursive architectures in both bit-serial and parallel are also presented. All the *Bi-z* CORDIC designs run faster than the Xilinx counterparts except the parallel-recursive *Bi-z* CORDIC, of which the speed limitation is mainly due to the parallel data recurrence. To make a fair comparison, the proposed *Bi-z* CORDIC is also compared with the Xilinx design in terms of area-delay-product (ADP) $= T \times A$, where T is the total latency for completing one CORDIC operation and A is the hardware cost in terms of the number of Virtex-II FPGA slices. All the ADP of the *Bi-z* CORDIC designs are less or comparable to the Xilinx designs except for the serial-pipelined design, which has a significantly higher ADP mainly due to the massive memory consumption for each rotation step as temporary serial data storage. All the *Bi-z* CORDIC designs also outperform the Xilinx reference designs in term of hardware consumption, which is due to the elimination of Z-path for angle formation and quantization operations that are required in the conventional CORDIC.

Table 4.2: CORDIC characterization data comparison on FPGA, for wordlength=16

		Size ($slice$)	Speed (MHz)	Clock Cycle	Latency (μs)	ADP ($T \times A$)
Xilinx LogiCORE	Parallel Rotate	680	170	20	0.118	80.2
	Parallel Translate	613	164	20	0.122	74.8
	Recursive Rotate	366	133	20	0.150	54.9
	Recursive Translate	300	134	20	0.149	44.7
Bi-z CORDIC	Parallel Recursive	291	87	19	0.218	63.4
	Parallel Pipelined	331	170	19	0.112	37.1
	Serial Recursive	37	170	256	1.51	55.9
	Serial Pipelined	315	178	256	1.44	454

Chapter 5

Parameterisable Array Architectures for Matrix Inversion

5.1 Introduction

Many well-known algorithms for modern digital signal processing, communications and control applications involve manipulation of large matrices, which is one of the most computation overload contributors, especially matrix inversion [68]. Many large-scale applications such as image processing, video processing and multimedia communication require the computation of matrix inversion. Traditional acceleration algorithms based on microprocessor run in sequence and the performance is limited especially for large matrices. Hardware implementation of matrix inversion can accelerate the computation greatly due to its parallel and distributed structure.

Matrix computation, especially matrix inversion, involves many complicated arithmetic operations such as multiplication, division, square root, etc., which require massive computing capabilities. The situation worsens when the problem size becomes larger or the size varies to cater for different applications, especially in real

time processing. For matrix inversion, iterative methods are shown to have better numerical stability than the direct inversion methods [69]. Thus, most of the work for matrix inversion on VLSI architectures are based on Cholesky Factorization [70], LU decomposition [71], Gram-Schmidt method [72], Givens Rotation (GR) based QR decomposition [73], or Gauss-Jordan Elimination [74], as discussed in Chapter 2. Sherman-Morrison method [75] is another algorithm used for updating inversion of successive matrices with small difference of perturbation. Among these, the GR-based QR decomposition is particularly suitable for hardware implementation. It avoids the square root operation and involves pure orthogonal transformation which is proved to be numerically stable for nonsingular input matrix [11]. To the best knowledge of the author, till now the structures for matrix inversion are not for practical applications - the matrix size is limited by the resource of the hardware since it involves huge consumption of logic resources even for implementation of problems with small matrix size. So a special VLSI processor targeting matrix inversion with resource efficiency, high data throughput and parameterisability is highly desirable.

In this chapter the proposed binary-coded Z-path CORDIC (*Bi-z* CORDIC) described in Chapter 4 is used to solve the matrix inversion problem. The aim is to accelerate the computation of data intensive matrix inversion for time critical applications. The inversion algorithm consists of four parts. First, a matrix A is factorized using the QR decomposition [76]. This can be carried out by the CORDIC algorithm [14], which involves only shift and add operations that are ideal for hardware implementation. In the second phase, the transformation matrix Q is produced by passing the identity matrix through the QR processors that calculate and store the rotation parameters in the first phase. And at the same time the triangularized matrix R is inverted by the recurrence algorithm [73] using the proposed *Bi-z* CORDIC structure. In the last step, A^{-1} is obtained with the multiplication of R^{-1} by Q.

Several functional modules with efficient hardware resources were designed to implement the matrix inversion. The proposed *Bi-z* CORDIC structure was designed for the QR decomposition and the inversion of triangularized matrix. The proposed structure avoids all the complicated operations and saves logic resources, which makes it possible to accommodate large matrix size by adopting the binary-coded Z-path method. In this chapter, the existing array structures for matrix decomposition and matrix inversion are described first, followed by the descriptions of different linear mapping methods and scheduling schemes for matrix inversion problem. The later part describes the properties of the *Bi-z* CORDIC structure and by targeting on the proposed mapping methods for matrix inversion, the hardware utilization is maximized to 100% or nearly 100% for each processing cell. The structure is fully scalable and parameterisable for different word-length and matrix size. Thus, the design time is shortened, the hardware efficiency is improved and the data throughput is increased, providing a flexible and scalable solution for different applications involving matrix inversion.

5.2 Existing Array Structures

5.2.1 Triangular Array Structure

Matrix inversion architectures for VLSI implementation are usually based on systolic array because of its regular and parallel processing structure. Figure 5.1 shows the inversion of 4×4 matrix A by mapping it to two cascaded triangular array architectures [73]. The shadowed triangular array on the left performs the QR decomposition by the GR which is an orthogonal transformation for the input matrix A, $QA = R$, as described in Section 2.2.1. Once Matrix A is decomposed into the orthogonal matrix Q and the upper triangular matrix R, the inversion can be computed by $A^{-1} = R^{-1}Q$ in the following cascaded triangular array as shown in the

right part of Figure 5.1. The inversion of a general matrix in Figure 5.1 consists of following steps: first, Matrix A is fed into the array architecture in a slightly skewed way where it is factorized into an upper triangular matrix R using QR decomposition method [76] by a series of Givens transformations. This can be carried out by the CORDIC algorthim [14]. Each PE Q with the subscript index generates the corresponding element in the triangularized matrix R. For example, PE Q_{12} produces the element R_{12} of Matrix R, and followed by the formation of element Q_{12} of Matrix Q using the same rotation parameters developed in the triangularizing step. The triangularization and the formation steps are fully overlapped and the matrices R and Q continue flowing into the triangular structure W seamlessly. Similarly, in the second step, transformation Q is formed by passing identity matrix of the same size through the QR triangular array in which exactly the same transformation parameters will be applied and the orthogonal transformation matrix Q is generated. At the same time the upper triangular matrix R flows seamlessly into the inversion arrays W where it is inverted and multiplied with the orthogonal transformation matrix Q immediately followed behind. Thus the inversion A^{-1} is obtained.

Although the mapping method with this architecture is intuitive in nature, the main problem is its huge hardware consumption. For an $n \times n$ matrix, it requires $n \times (n+1)$ array cells. This is not realistic for hardware implementation especially when the number of array cells used grows exponentially with the matrix size. Another problem with this architecture is the lack of scalability. Each time when the matrix size changes, a new structure must be created accordingly.

The data inputs for the processors of the QR arrays can be derived from Figure 5.1, as shown in Table 5.1, where a matrix with size of 5 is used for illustration. The control signals for each QR array can be drawn out as shown in Table 5.2. Note the letters 'I', 'V' and 'R' are used to represent the idle, vectoring and rotation operations respectively as discussed in the next section. Here it is assumed that

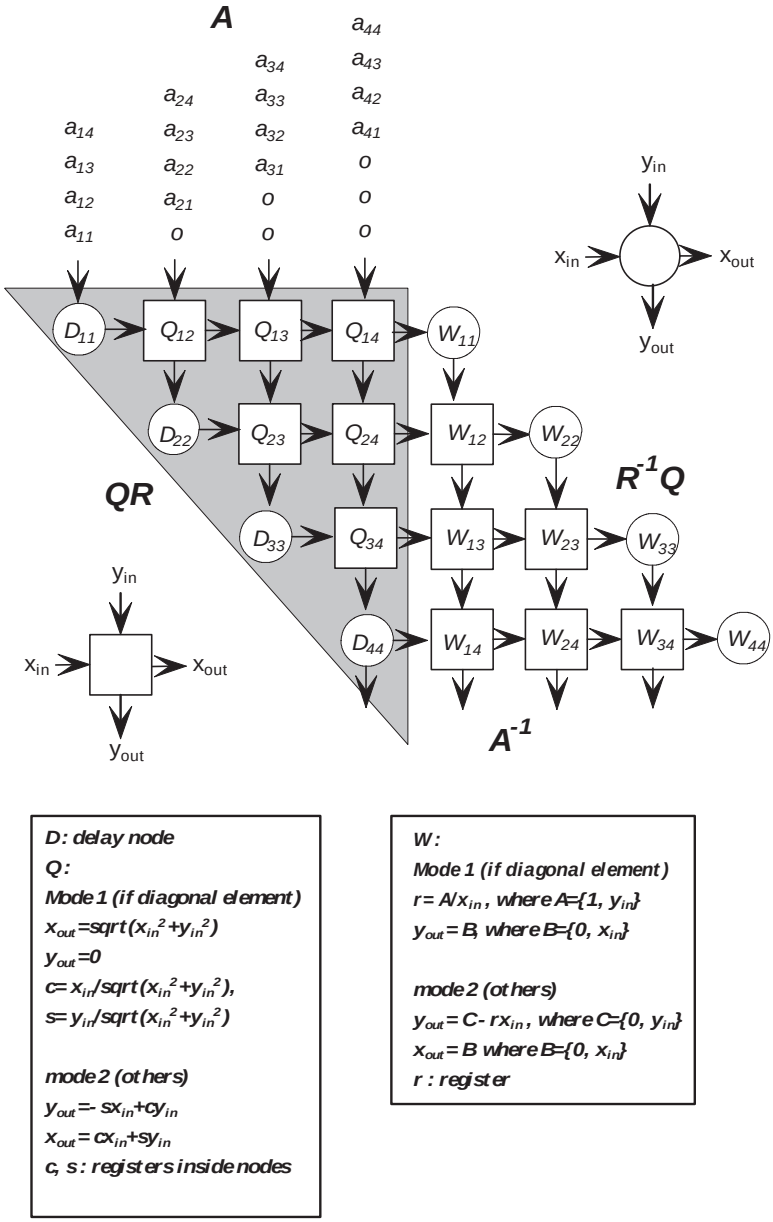


Figure 5.1: Cascaded triangular array structures for 4×4 matrix inversion

the processing time for each PE is 1 clock cycle. For an $n \times n$ matrix, the first output of the QR arrays becomes available after n clock cycles. The identity matrix immediately follows the input matrix so the total QR arrays are updated at time $3n - 1$. From the data and mode control signal tables, it can be seen that the array processors are in idle state for most of the processing time.

Table 5.1: Data Inputs Control For The QR Array Processor

Cycle	Linear Array									
	Q_{12}	Q_{13}	Q_{14}	Q_{15}	Q_{23}	Q_{24}	Q_{25}	Q_{34}	Q_{35}	Q_{45}
1	I	I	I	I	I	I	I	I	I	I
2	$a_{11}a_{21}$	I	I	I	I	I	I	I	I	I
3	$a_{12}a_{22}$	$a_{11}a_{31}$	I	I	I	I	I	I	I	I
4	$a_{13}a_{23}$	$a_{12}a_{32}$	$a_{11}a_{41}$	I	I	I	I	I	I	I
5	$a_{14}a_{24}$	$a_{13}a_{33}$	$a_{12}a_{42}$	$a_{11}a_{51}$	$a_{22}a_{32}$	I	I	I	I	I
6	$a_{15}a_{25}$	$a_{14}a_{34}$	$a_{13}a_{43}$	$a_{12}a_{52}$	$a_{23}a_{33}$	$a_{22}a_{42}$	I	I	I	I
7	I	$a_{15}a_{35}$	$a_{14}a_{44}$	$a_{13}a_{53}$	$a_{24}a_{34}$	$a_{23}a_{43}$	$a_{22}a_{52}$	I	I	I
8	I	I	$a_{15}a_{45}$	$a_{14}a_{54}$	$a_{25}a_{35}$	$a_{24}a_{44}$	$a_{23}a_{53}$	$a_{33}a_{43}$	I	I
9	I	I	I	$a_{15}a_{55}$	I	$a_{25}a_{45}$	$a_{24}a_{54}$	$a_{34}a_{44}$	$a_{33}a_{53}$	I
10	I	I	I	I	I	I	$a_{25}a_{55}$	$a_{35}a_{45}$	$a_{34}a_{54}$	$a_{44}a_{54}$
11	I	I	I	I	I	I	I	I	$a_{35}a_{45}$	$a_{45}a_{55}$
12	I	I	I	I	I	I	I	I	I	I

Table 5.2: Mode Control Signals For The Array Processor

Cycle	Input	Linear Array									
		Q_{12}	Q_{13}	Q_{14}	Q_{15}	Q_{23}	Q_{24}	Q_{25}	Q_{34}	Q_{35}	Q_{45}
1	a_{11}	I	I	I	I	I	I	I	I	I	I
2	a_{21}	V	I	I	I	I	I	I	I	I	I
3	a_{31}	R	V	I	I	I	I	I	I	I	I
4	a_{41}	R	R	V	I	I	I	I	I	I	I
5	a_{51}	R	R	R	V	V	I	I	I	I	I
6		R	R	R	R	R	V	I	I	I	I
7		I	R	R	R	R	R	V	I	I	I
8		I	I	R	R	R	R	R	V	I	I
9		I	I	I	R	I	R	R	R	V	I
10		I	I	I	I	I	I	R	R	R	V
11		I	I	I	I	I	I	I	I	R	R
12		I	I	I	I	I	I	I	I	I	I

5.2.2 Linear Array Structures

An alternative approach is to partition and map the two dimensional arrays onto linear array structures [77, 78], as depicted in Figure 5.2. By projecting the two dimensional structure onto different hyperplanes, it may result in different interconnection and scheduling schemes for different linear array mapping methods [78, 79]. The linear array structures can save the number of processing cells substantially. For an $n \times n$ matrix, in this case, only $2 \times n$ processors are needed. The scaling can also be done easily in 1-dimensional direction when the matrix size changes. But these methods suffer from drawbacks such as inefficiency of cell utilization, thus waste hardware resource, and they are not resource efficient for hardware utilization. A more resource efficient architecture can be derived by folding and interlacing the processor structure to achieve 100% processor utilization as discussed in Section 5.5.

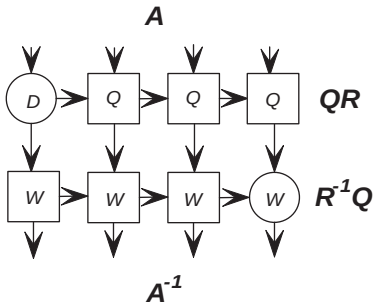


Figure 5.2: Cascaded linear array structures for 4×4 matrix inversion

By mapping the two-dimensional array onto linear array architectures, different mapping methods may result in different data scheduling, processor structure as well as hardware utilization efficiency. Rader [80] proposed the mixed mapping methods for the QR decomposition in Figure 5.3. The operations of the triangular array are mapped onto the dual function processors which are fully utilized. By moving the bottom part of the array and filling up the position as depicted in Figure 5.3, it has equal operations for each row of the array. The operations are then mapped to

the linear array processors, in this case, two rows mapping onto one processor. In this mapping method, two distinct types of operation, boundary and internal, are allocated to the one processor, which yields complicated architecture design of the functional processor.

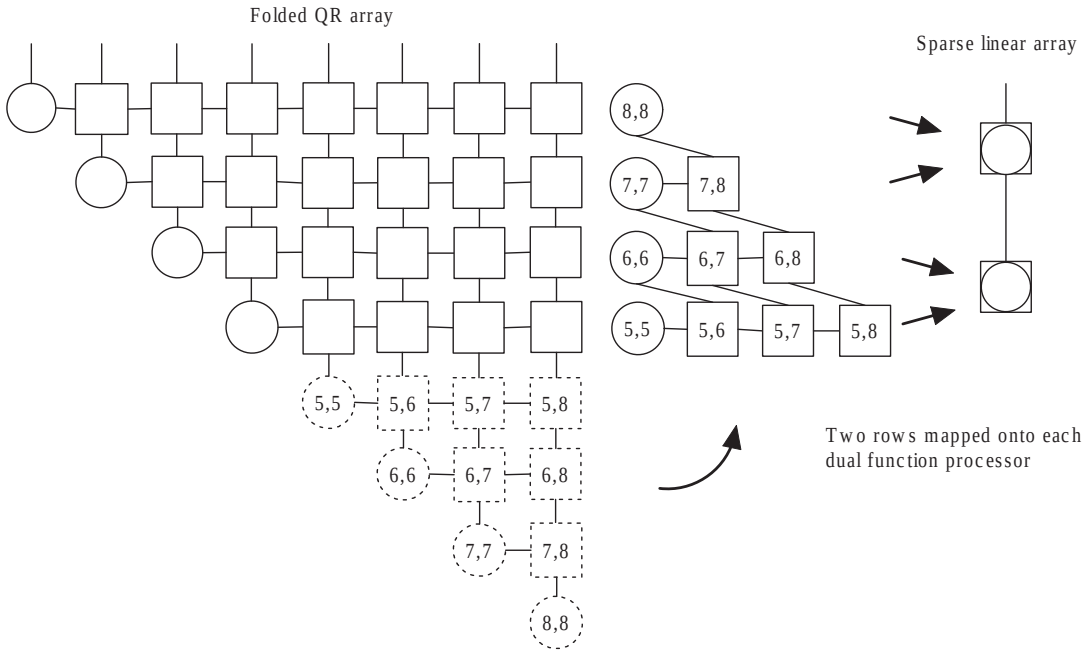


Figure 5.3: Mixed mapping method for QR decomposition

The discrete mapping technique [81,82] shown in Figure 5.4 distinguishes these two types of operation and has the processor fully utilized. This is done by first moving the lower part of array architecture to the top of the design and then folding from middle to interleave the operation. The resulting structure is then mapped onto a linear array processor with the boundary and internal operations separated. The operation scheduling for the discrete mapping method is a bit complicated since it involves irregular data input.

Another mapping method to map the triangular architecture onto linear array processors is shown in Figure 5.5 [83]. The figure only shows the mapping method for the QR decomposition array. The overall architecture for the matrix inversion is by cascading two of this linear array together as shown in Figure 5.2. This linear

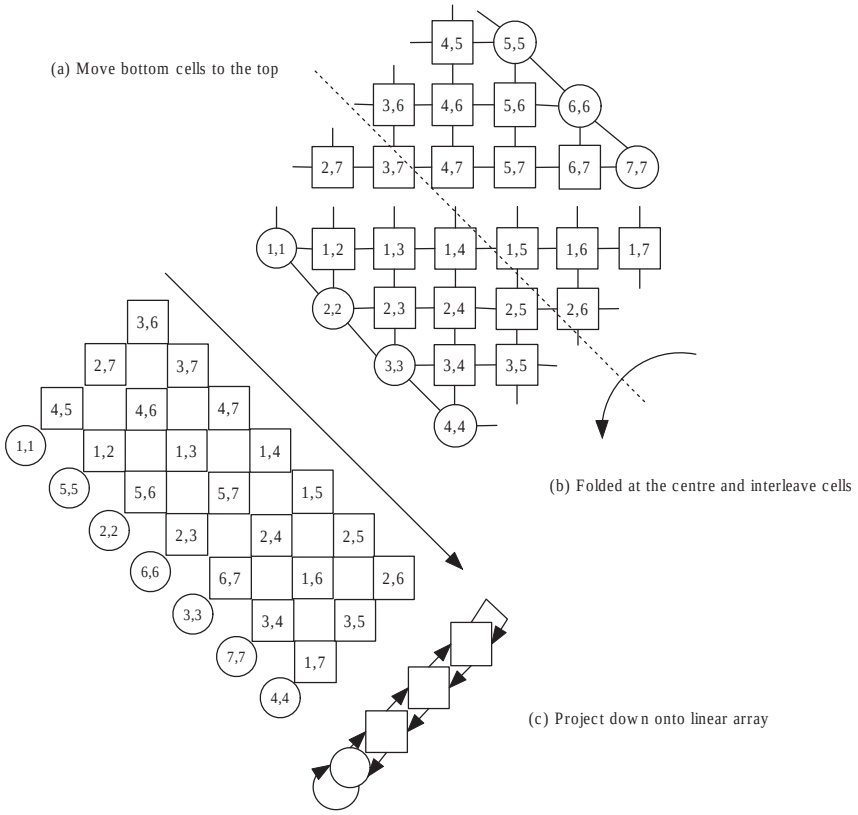


Figure 5.4: Discrete mapping method for QR decomposition

mapping method is straightforward and intuitive in nature by grouping the same types of operations together. But the hardware utilization is less than the previous two methods. The average processor utilization is around 62.5%.

5.3 The *Bi-z* CORDIC in Matrix Inversion

The proposed *Bi-z* CORDIC in Chapter 4 is adopted for the matrix inversion architectures. The *Bi-z* CORDIC can be used in both the QR decomposition arrays and the triangular matrix inversion arrays of Figure 5.1, as discussed in this section.

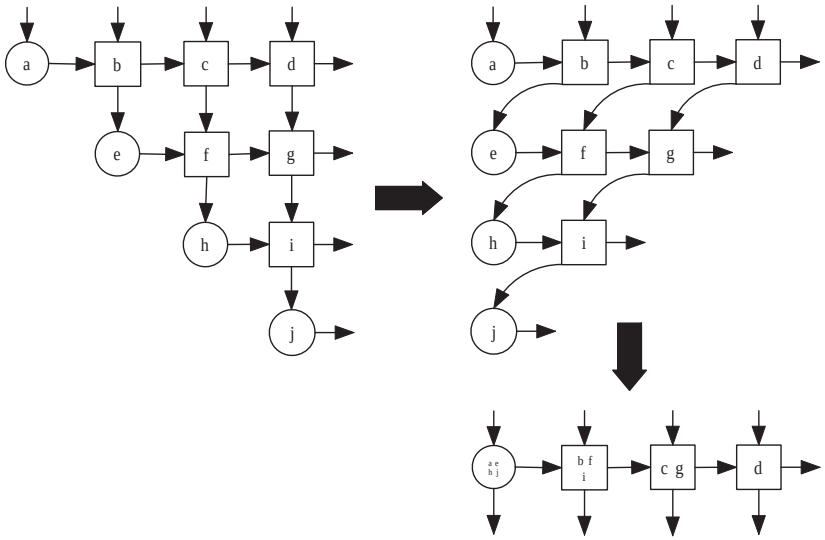


Figure 5.5: Linear mapping method

5.3.1 The Proposed *Bi-z* CORDIC

It is noticed that the processing methods of the two triangular array structures in Figure 5.1 for matrix inversion are similar, i.e. the rotation parameters are first generated by the leading vector, followed by the transformation of the following vectors using the same parameters. The QR decomposition arrays can use Givens rotation to generate the parameters which are suitable for implementation with the CORDIC in the circular operation mode, while the inversion arrays for the triangularized matrix inversion, which involves the calculation of division and reciprocation, can use the CORDIC in the linear operation mode. Since the parameters are used immediately after being required, there is no need to compute the explicit rotation angle z_i as in the conventional CORDIC Z-path chain. The only issue that matters is to generate the rotation parameters represented in a series of binary rotation directions δ_i, and to apply the binary rotation directions on the following input vectors. Thus tremendous logic resources can be saved by adopting the *Bi-z* CORDIC, which is free of square root, multiplication and direct division arithmetic.

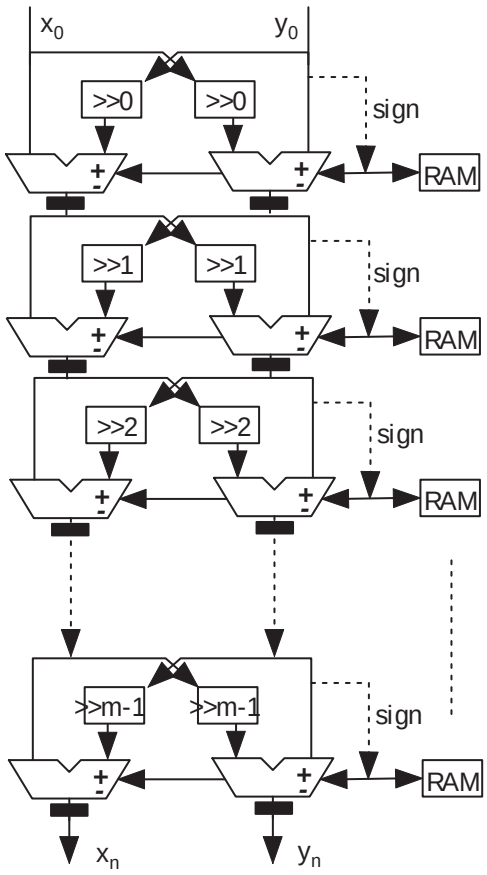


Figure 5.6: Pipelined m-tap *Bi-z* CORDIC circular structure

5.3.2 *Bi-z* CORDIC in Decomposition Arrays

There are two modes involved in the PEs of the decomposition part: vectoring and rotation. The rotation directions are first generated by the leading vector with y input elements on the diagonal of the matrix in the vectoring mode. The value of the rotation angle is calculated theoretically with Equation 4.15. The actual accumulated rotation angle is represented by Equation 4.17. It can be seen that if this rotation angle is applied as the input angle value z_0 for the CORDIC in the rotation mode, the exact same rotation direction δ_i in each micro-rotation will be generated. Thus in the *Bi-z* CORDIC, only the 1-bit rotation direction δ_i will be generated and stored in the register of each step. There is no actual angle value calculation in the Z-path chain. For the following vectors to be rotated by the

same angle, only the rotation direction bits generated in the previous rotation stage will be applied in each of the micro-rotation steps. Figure 5.6 shows the pipelined *Bi-z* CORDIC structure with rotation steps equal to m (m-tap) in the circular operation mode. The Z-path in each step of the micro-rotation needs only 1-bit register to store the rotation direction. Thus an adder-subtractor chain (with m adder-subtractors plus control logic) is saved. Figure 5.8 (a) shows the processor symbols of the decomposition modules. The round node performs the delay function and the square node is for the decomposition operation.

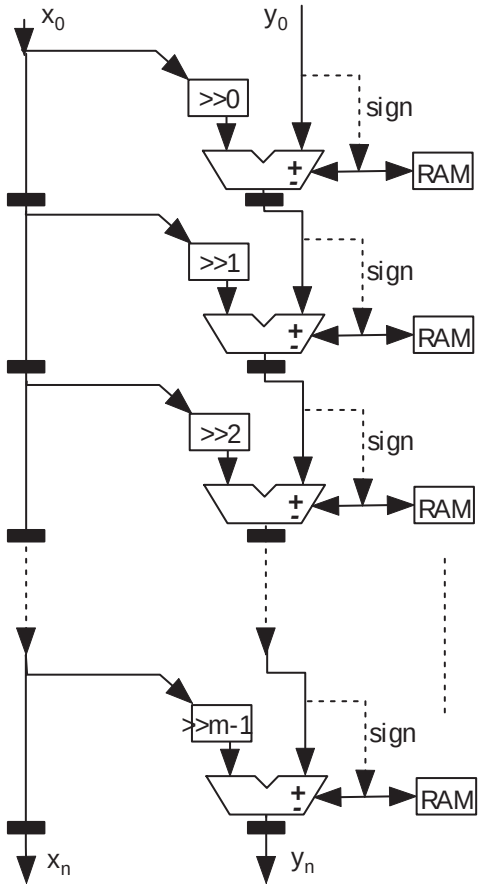


Figure 5.7: Pipelined m-tap *Bi-z* CORDIC linear structure

5.3.3 *Bi-z* CORDIC in Inversion Arrays

The PEs perform the recurrence algorithm as shown in Figure 5.1 and 5.2 with nodes marked W. Similar to the decomposition part, there are two operation modes: for the diagonal input elements the round nodes perform reciprocal operation while the square ones perform division. The result r is stored in the internal register and multiplied with the following vectors as illustrated in Figure 5.1. The *Bi-z* CORDIC in the linear rotation mode shown in Figure 5.7 can be used to calculate the division and reciprocation, and again since these values are immediately applied to the following vectors, there is no need to calculate the explicit values, as explained in Section 4.4.2.

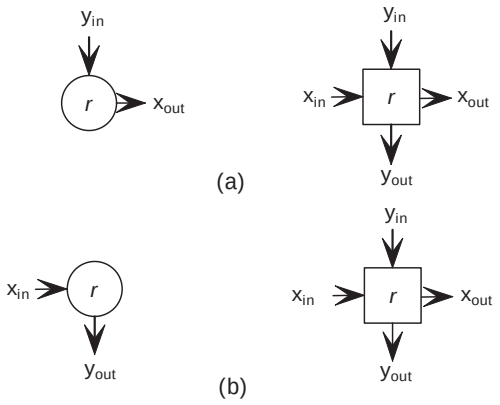


Figure 5.8: Processor symbols for matrix inversion

Figure 5.8 (b) shows the processors for calculating the reciprocal and division. The round node is to calculate the reciprocal and the square node is for the division. The series of binary rotation direction values, which are δ_i in Equation 4.23, that represent the reciprocal or division are generated and stored in the 1-bit registers of each micro-rotation step. The operation followed is the multiplication of input vector with the stored value of division or reciprocation represented in the binary coded rotation directions δ_i, which only involves shift and add operation in the proposed structure as shown in Equations 5.1 and 5.2. For reciprocal processor in

the left part of Figure 5.8 (b), two types of operation mode are involved, Mode 1 and Mode 2. Mode 1 is used to derive the division value while Mode 2 is to multiply the derived value with the following input data.

Mode 1 (if x_{in} is on diagonal)

$$\begin{cases} x_{out} = x_{in} \qquad \text{(omitted)} \\ y_{in} = 1 \\ y_{out} = 0 \\ r = 1/x_{in} = -\sum_{i=0}^{n-1} \delta_i 2^{-i} \end{cases} \tag{5.1}$$

Mode 2 (if x_{in} is off-diagonal element)

$$\begin{cases} x_{out} = x_{in} \qquad \text{(omitted)} \\ y_{in} = 0 \\ y_{out} = -r \cdot x_{in} = \sum_{i=0}^{n-1} \delta_i 2^{-i} \cdot x_{in} = \sum_{i=0}^{n-1} \delta_i \cdot x_{in}^{\{-i\}} \end{cases} \tag{5.2}$$

As can be seen from Equation 5.1, the reciprocal value r is factorized on the base-2 progression, which is represented as the summation of $\delta_i 2^{-i}$. When applying this reciprocal value to the subsequent input vectors as shown in Equation 5.2, the multiplication is equalized with a series of shift and add/sub operations. The binary rotation direction δ_i acts as an addition or substraction operator.

Similarly, for the division processor in the right part of Figure 5.8 (b), there are two types of operation mode as described in Equations 5.3 and 5.4.

Mode 1 (if x_{in} is on diagonal)

$$\begin{cases} x_{out} = x_{in} \\ y_{out} = 0 \\ r = y_{in}/x_{in} = -\sum_{i=0}^{n-1} \delta_i 2^{-i} \end{cases} \tag{5.3}$$

Mode 2 (if x_{in} is off-diagonal element)

$$\begin{cases} x_{out} = x_{in} \\ y_{out} = y_{in} - r \cdot x_{in} = y_{in} + \sum_{i=0}^{n-1} \delta_i 2^{-i} \cdot x_{in} \\ \qquad = y_{in} + \sum_{i=0}^{n-1} \delta_i \cdot x_{in}^{\{-i\}} \end{cases} \tag{5.4}$$

The value of division is in the form of a base-2 series as indicated in Equation 5.3. The following input vectors are transformed according to Equation 5.4 where only shift and addition are involved. The binary rotation direction δ_i performs the same operator function as in the reciprocal PEs in Equation 5.2.

5.4 Linear Mapping method

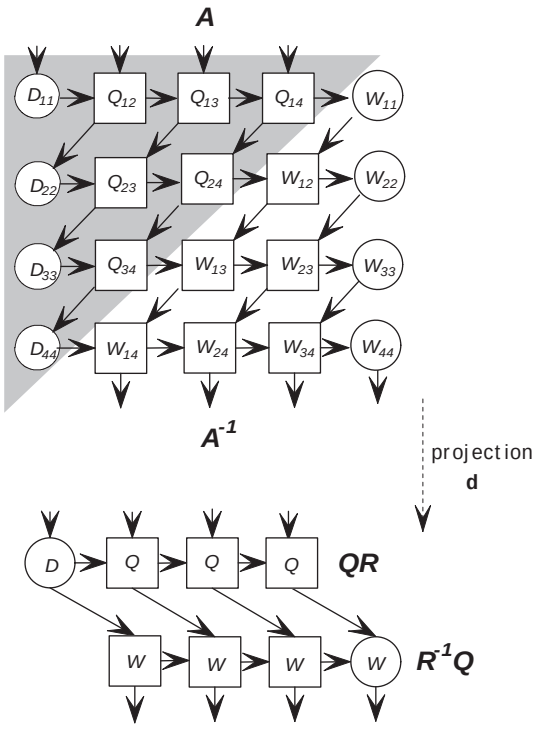


Figure 5.9: Linear mapping of matrix inversion onto array structure

For mapping the 2-D triangular structure onto the linear array, there exist many par-

tition methods. Different methods may lead to different timing and data scheduling as well as processor utilization [81]. In the linear mapping method, the projection direction **d** is shown in Figure 5.9, where the decomposition arrays and inversion arrays are collapsed and mapped onto two linear arrays separately. Since the array structures of the two triangular arrays in Figure 5.1 are the same, the dataflows and control signals are actually similar. The schedule problems can be derived from either of these two triangular array structures. The QR decomposition array is chosen for illustration, and the same derivative method can be applied to both the QR array and the inversion array.

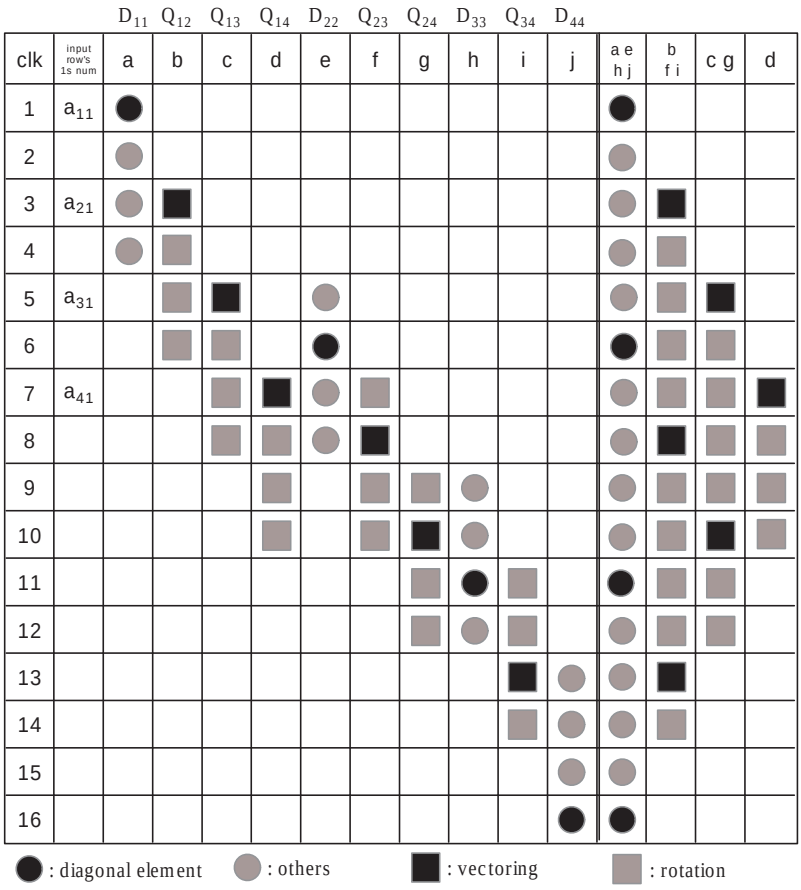


Figure 5.10: Scheduling table for the linear array structure

The left part of Figure 5.10 shows the scheduling scheme of the triangular array for the QR decomposition in Figure 5.1, with a representing the scheduling of node D_{11}

and so on. The symbol name of each schedule is used to represent the corresponding node. In the right part of Figure 5.10, the nodes *a, e, h, j* are grouped together and mapped to a single array, with *b, f, i* mapped to one linear cell, *c, g* to another and *d* is left alone. The PEs in this mapping and scheduling scheme are not fully utilized, but with an average utilization of 62.5%. However, this method is intuitive and has steady input and output samples. There are other possible scheduling schemes that result in 100% efficiency [84], but these designs suffer from eccentric data flows and communication and control problems among the PEs. It is assumed that the processing time of each PE is 1 clock cycle. It takes $2n^2$ cycles for each linear array to process the data. As soon as the first available data appears on the input of the inversion array, the inversion array can start processing without finish of the decomposition array. This counts for $2n-1$ clock cycles overlapping. The first result comes out 1 clock cycle after the total processing time, which is $2n^2-2n+1$ clock cycles. So the total number of clock cycles to invert an $n \times n$ matrix is $O(2\times(n^2-n+1))$ in this case. The actual processing time in each PE usually takes P-stage pipelined cycles. Thus the actual processing time is $O(2P(n^2-n+1))$ clock cycles instead.

5.4.1 Scheduling

The mapped decomposition and inversion processors are cascaded and interconnected by multiplexers as shown in Figure 5.11. For the QR decomposition processors *PE*, the y input, y_{in}, is either from the external input or the processor to the right of it and the x input, x_{in}, is taken from the left processor. Similarly, the x input of the inversion processors $PE^{'}$ is either from the output of the QR decomposition processors or from the output of the $PE^{'}$ on the left hand side, and the y input is taken from the right processor. The input data are connected and scheduled by the input multiplexers as shown in Figure 5.11. The multiplexers are controlled

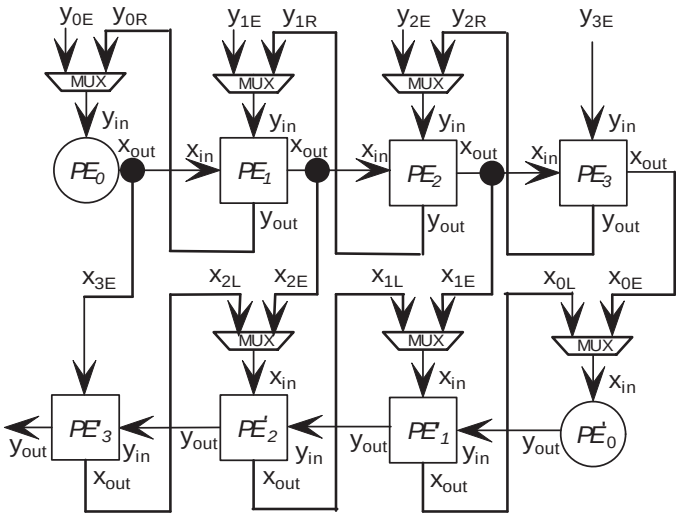


Figure 5.11: Linear architecture interconnection

by data control signals. Figure 5.12 depicts the scheduling of the control signals for linear mapped processors in Figure 5.9. Figure 5.12 (a) indicates the input data control signals for the QR decomposition processors, while Figure 5.12 (b) shows the controlling signals for the inversion processors.

The letters 'E', 'R', 'L' in the table stand for data flow from either external (or output of QR processors in (b) for inversion array), right or left processor respectively. The asterisks indicate the elements on the matrix diagonal are fed in as input. For example, '*ER' in (a) at clock row 1 under PE_0 means the processor PE_0 takes the input matrix on the diagonal element at time unit 1 from the external input and outputs the data to the right processor PE_1. While 'LR' in (b) in clock row 17 and column labeled PE_1' means the processor PE_1' takes the output of left processor PE_2' as x input and the output of right processor PE_0' as y input for its input vector (x, y) at time unit 17.

The processors for the matrix inversion have pipelined structure in systolic data flow. So each processor unit must have the same processing cycle. This is to guarantee that the data flow can be properly aligned. The latencies of the critical functional

clk	PE_0	PE_1	PE_2	PE_3
1	*ER			
2	ER			
3	ER	*EL		
4	ER	EL		
5	RR	EL	*EL	
6	*RR	EL	EL	
7	RR	RL	EL	*EL
8	RR	*RL	EL	EL
9	RR	RL	RL	EL
10	RR	RL	*RL	EL
11	*RR	RL	RL	
12	RR	RL	RL	
13	RR	*RL		
14	RR	RL		
15	RR			
16	*RR			

E: external, R: right, L: left
*: diagonal elements

(a) Control signal for QR PEs

clk	PE_3'	PE_2'	PE_1'	PE_0'
11				*EL
12				EL
13			*ER	EL
14			ER	EL
15		*ER	ER	LL
16		ER	ER	*LL
17	*ER	ER	LR	LL
18	ER	ER	*LR	LL
19	ER	LR	LR	LL
20	ER	*LR	LR	LL
21		LR	LR	*LL
22		LR	LR	LL
23			*LR	LL
24			LR	LL
25				LL
26				*LL

(b) Control signal for inversion PEs

Figure 5.12: Input data control signal of the linear array structure

modules and the processor before retiming are depicted in Table 5.3. Each type of the processors must have the same timing $L_c = max\{C_B + C_M, C_B + C_S + C_A\}$, so delay units are inserted into the necessary processors to synchronize the data flow. The data throughput is expressed as $\frac{f_{CLK}}{2L_c \cdot (n^2 - n + 1)}$ with matrix size n and system clock f_{CLK}.

5.4.2 Architecture Design

As discussed in Section 5.3, both the decomposition and the inversion processors can be implemented by using the *Bi-z* CORDIC. But there are still some hardware design issues need to be considered. The decomposition processor adopts the Givens rotation method which is a constant rotation. The vector length remains unchanged throughout the processing. As can be seen from Equations 4.13 to 4.15, in each micro-rotation step the CORDIC introduces a scaling factor $k_i = \sqrt{1 + 2^{-2i}}$, thus the total accumulated scaling value after n-th rotation is defined as the scaling factor $K = \prod_{i=0}^{n-1} k_i$. If the number of rotations is known in advance, the scaling factor introduced is a constant value and can be compensated by a constant multiplication either at the post- or pre-processing stage. Figure 5.13 depicts the functional structure for the processors in the decomposition arrays. The cell performing the delay functions is shown in (a) which is implemented with shift registers. The lower part diagram (b) is the architecture of the cell to perform the circular rotation both in the vectoring and rotation mode. The figures show the cell architectures with input data word-length of 16 bits. In the proposed design, the constant scaling multiplier is put in front of the *Bi-z* CORDIC processor.

In the inversion arrays, the reciprocal and division are realized by the proposed *Bi-z* CORDIC. The value of the quotient is limited by the expression of Equation 4.23. The result of using the direct CORDIC method to compute division cannot exceed $\lim_{n\to\infty} \sum_{i=0}^{n} 2^{-i} = 2$ by nature. To conquer this limitation, the input vectors are pre-processed before being fed into the *Bi-z* CORDIC, which is shown in Figure 5.14. The cell in Figure 5.14 (a) is for the reciprocal while (b) is for the division. For the reciprocal cell, the first input variable x is compared with 1 and shifted if necessary to ensure the quotient is within the range (-2, 2). Since only the quotient value is needed (the output of x is omitted in the reciprocal operation), there is no need to recover the denominator, so the post-process shift back register is omitted.

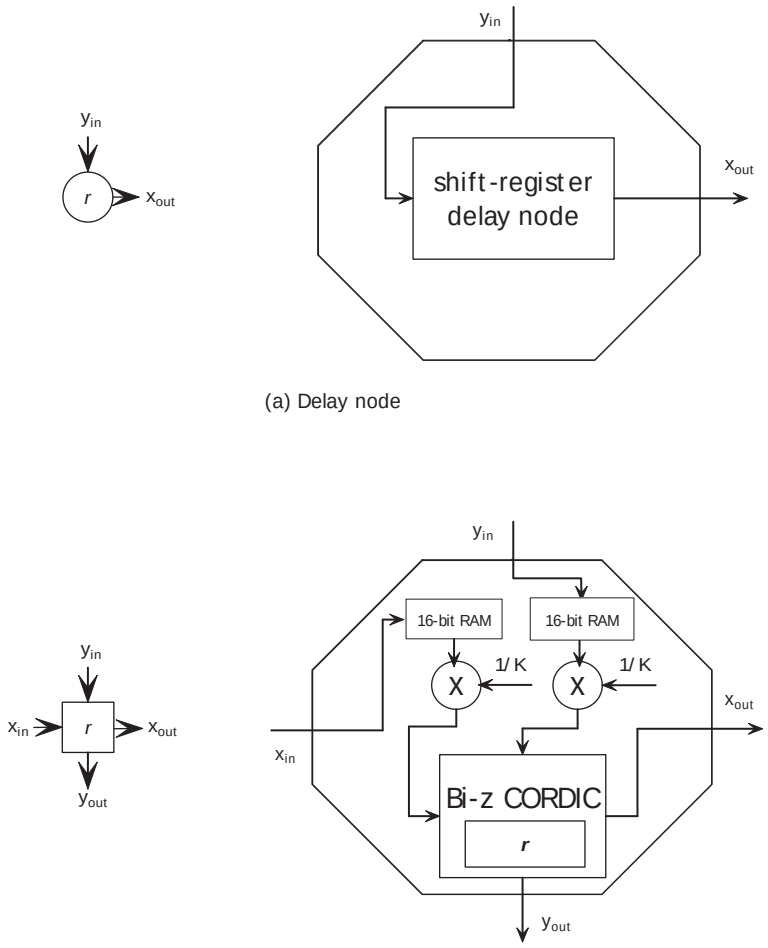
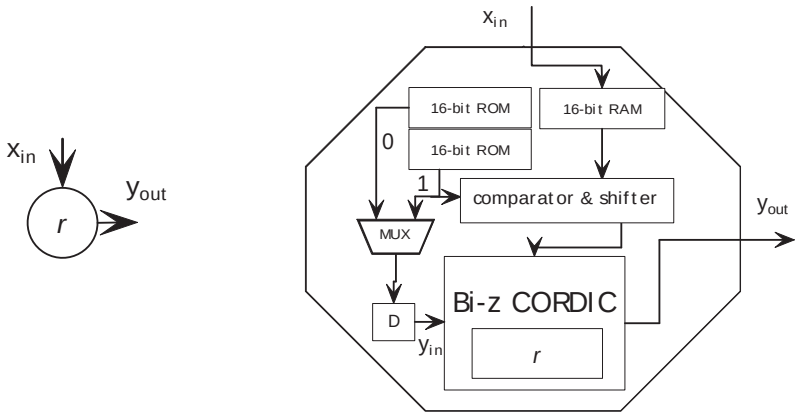


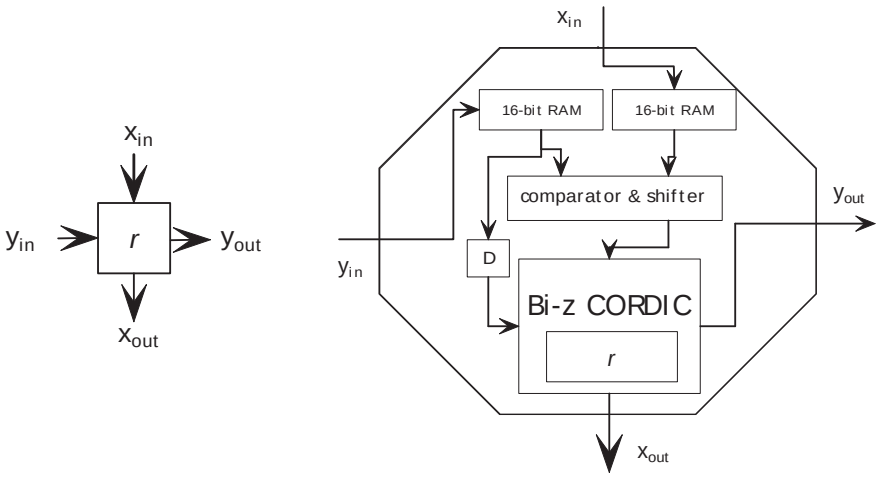
Figure 5.13: Processor architecture for decomposition arrays

The number of shift bits plus the rotation direction δ_i are stored and applied to the following input variables. This is similar for the division cell. This time instead of being compared with the pre-stored constant value 1, the two variables of the first input vector are compared and x input is shifted and scaled to make the quotient within the range (-2, 2). After the process of the *Bi-z* CORDIC, the post-processing shift register is used to recover the denominator x_{out}. The derived shift bits and rotation directions in terms of binary value are then applied to the following input vectors. The operation is similar to that in the reciprocal cell.

The hardware cost of each operation element inside the array processors in terms of



(a) Architecture of PE for reciprocal



(b) Architecture of PE for division

Figure 5.14: Processor architecture for inversion arrays

Xilinx FPGA slice is shown in Table 5.3 (a). Table 5.3 (b) unveils the area cost of each type of the processor elements in the linear mapped structure for performing different operations.

Table 5.3: Latency & Area Of Functional Modules And Processors

(a)

Module	*Bi-z* CORDIC	Constant multiplier	Scaling shifter	Adder
Latency	C_B	C_M	C_S	C_A
Area (slice)	341	28	32	1

(b)

Processor	Delay	Decomposition	Reciprocal	Division
Latency	C_B+C_M	C_B+C_M	C_B+C_S	C_B+C_S+C_A
Area (slice)	36	414	373	387

5.4.3 Hardware Implementation

The matrix inversion architecture presented here was developed using Verilog HDL, simulated in Modelsim environment [85] and MATLAB environment and synthesized with Xilinx XST synthesizer. It takes $O(2P(n^2 - n + 1))$ clock cycles to have one $n \times n$ matrix inversion update, i.e. loading the next input data to the cells, for the linear mapping arrays, where P is the processing latency of the array cell. The *Bi-z* CORDIC for the linear mapping arrays was implemented in the 16-bit pipelined engine and Table 5.4 shows the resource utilization for different matrix sizes and computation throughput with processing speed up to 170 MHz. The implementation was targeted on Xilinx Virtex-II xc2v4000-4 FPGA.

The 4×4 matrix inversion using the linear mapping method was implemented also in 16-bit bit-serial architecture. Both the shift-adder registers and the delay nodes are emulated by using a dual-port RAM inside the chip. The bit-serial implementation is chosen since it can be better mapped to the fine-grained FPGA resources and it consumes much less silicon area compared with the parallel solution, as can be seen from Table 5.3 where the area cost in term of slices for different functional modules and processors inside the architecture is listed. The proposed bit-serial

Table 5.4: Resource Usage And Throughput For Different Matrix Sizes

Matrix Size	Resource usage			Latency	
	Slice	FF	Mult	Cycle	Delay(μs)
4	1,353	2,421	6	352	2.05
7	2,424	4,371	12	1,376	8.00
11	3,851	6,971	20	3,552	20.65
16	5,637	10,221	30	7,712	44.84
21	7,422	13,471	40	13,472	78.33
26	9,207	16,721	50	20,832	121.12
31	10,992	19,971	60	29,792	173.21
36	12,777	23,221	70	40,352	234.61
41	14,562	26,471	80	52,512	305.30
46	16,347	29,721	90	66,272	385.30
51	18,133	32,971	100	81,632	474.61
56	19,918	36,221	110	98,592	573.21
61	20,598	37,548	120	117,152	681.12

CORDIC engine is adopted, so the actual unit time is 16 clock cycles instead. The implemented design can run at 140 MHz, and it takes 47.54 μs to process a single 4×4 matrix inversion.

5.5 Interlaced mapping method

There have been existing works to map the 2-dimensional triangular array structures into linear arrays. Most of the works for matrix inversion on VLSI implementation contain idle time slots to align the data sequence among processing arrays. This will result in inefficient usage of array cells, for example with average utilization around 65% [78, 79]. There have been some other possible scheduling schemes which lead to 100% utilization [86], but these designs are not targeting at matrix inversion problem. To the author's best knowledge, there is still no design for matrix inversion on

VLSI architecture that yields 100% processing cell utilization. This section presents a mapping method that achieves 100% processor utilization. By properly projecting and folding the 2-dimensional array architecture, the 100% or near 100% processing cell utilization can be achieved.

Figure 5.15 and 5.16 depict the proposed interlaced mapping methods to map a 2-dimensional matrix inversion structure onto a linear array. Figure 5.15 is for even size matrix inversion while Figure 5.16 is for odd size matrix inversion. For input matrix with even size, dummy cells are first introduced at the connection between triangular arrays and inversion arrays as shown with the empty boxes in Figure 5.15. The shadowed cells in the lower part of the structure are cut and moved to the top of the input cells. The dotted lines with number labeled are the schedule planes for the data processing, which depict the clock schedules for each of the cell to start processing. The cells that process the data at the same time will be aligned together with the same schedule line. To simplify the analysis, here it is assumed that the processing time for each of the array cells is 1 clock cycle. For real case, the processing time of the cells will be multiplied by the maximum number of pipelined stages within each cell. Then the array structure is folded at the center and all the processing cells (including the dummy cells) will be fully interlaced with each other on each schedule line. The array cells will then be projected down to form a linear array in the direction perpendicular to the schedule plane as depicted in Figure 5.15. For an $n \times n$ matrix, where n is even number, the average cell utilization of the proposed interlaced mapping method is $(n-1)/n$.

If the input matrix is of odd size, no dummy cell is inserted as shown in Figure 5.16. The mapping technique is the same except that one more schedule line is moved to the top of the array structure. In this case, 100% cell utilization is achieved. The BC and IC represent the boundary cell and the internal cell respectively. In the interlaced mapping method, there are two operation modes in the BC, namely the

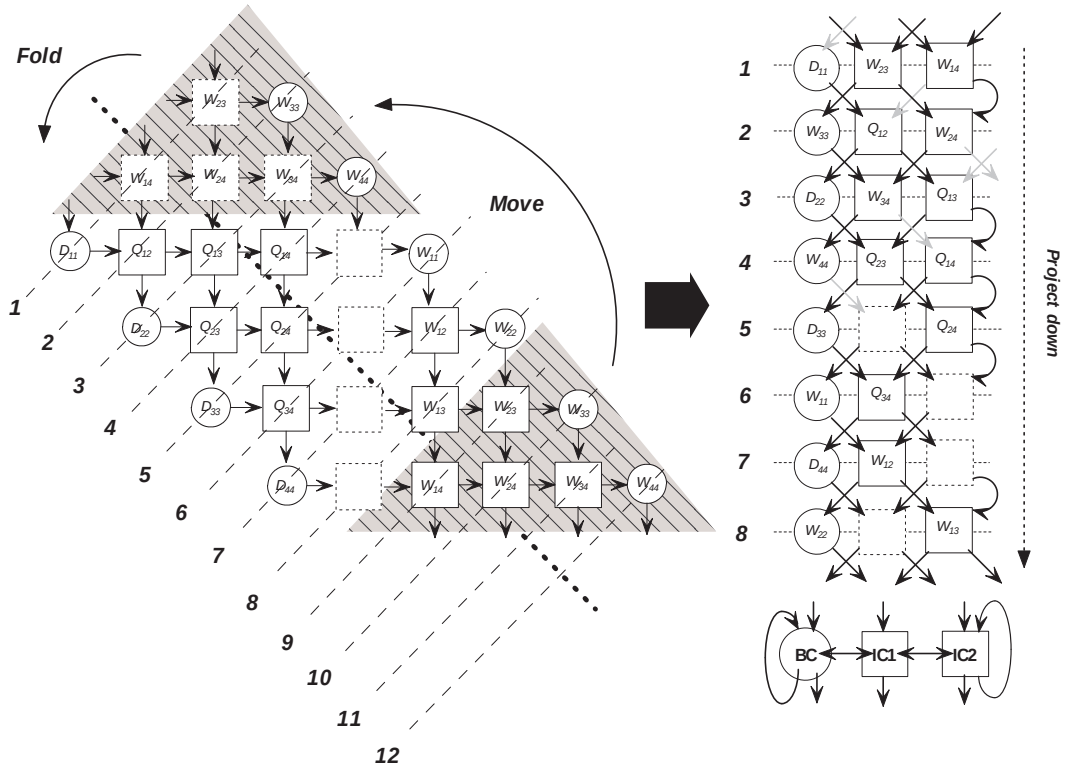


Figure 5.15: Interlaced mapping method for matrix inversion of even size

idle mode, which represents the delay operation, and the reciprocal operation mode, which is the result of interlaced mapping with the inversion array cells. Likewise, the IC handles the QR decomposition as well as the division operation of the inversion array in Figure 5.1.

In general, for $n \times n$ matrix inversion, the interlaced mapping method requires 1 BC and N IC cells, where N is the integer part of $\frac{n}{2}$, that is $N = [\frac{n}{2}]$. The following derivation can be made directly from this mapping method.

- The data scheduling is periodic and repeats every M cycles, where $M = 2 \times n$;
- The new data is input every M cycles, and the output results are available in every M cycles. This allows the linear arrays to be updated every M cycles.
- The source of the input data for all the processors is either from external input, or the outputs of the adjacent processors, or the outputs of themselves

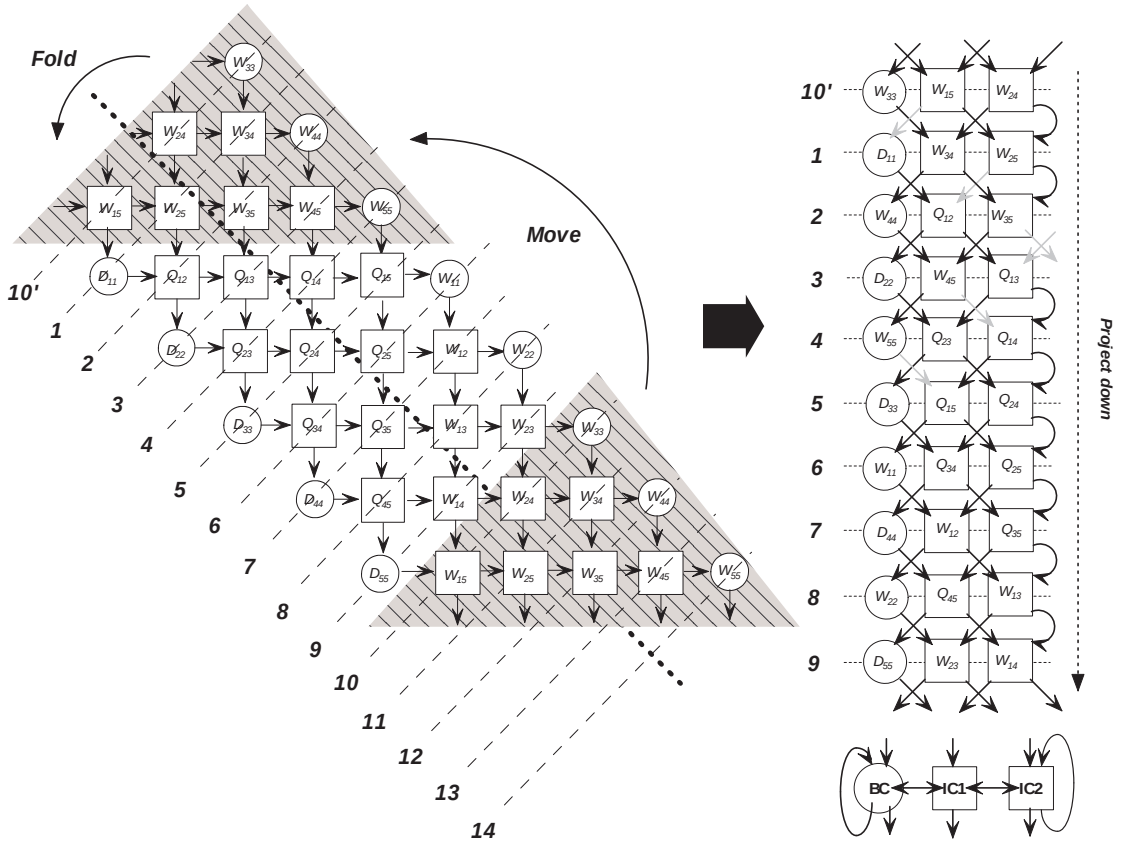


Figure 5.16: Interlaced mapping method for matrix inversion of odd size

for boundary cells.

Given the above conditions, the scheduling scheme is first derived for the interlaced mapping method, assuming the processing cell latency equal to 1 clock cycle. Then, considering the influence of the latency of each cell, a general scheduling technique can be applied to this architecture where the cell latencies are involved. This is achieved by the processor re-timing and the proper scaling of the scheduling sequence accordingly.

5.5.1 Scheduling Challenge

As can be seen from Figure 5.15 and Figure 5.16, by interlacing and collapsing the QR arrays and the inversion arrays together, the operation scheduling and the

control signals for the IO data input is much more complicated compared with the linear scheduling method, which only involves changing the modes within the *Bi-z* CORDIC engine. By mapping two types of array structure together into the same cell, different operations need to be performed on the same array processor at different time slots, and each type of operations has different modes in the *Bi-z* CORDIC engine.

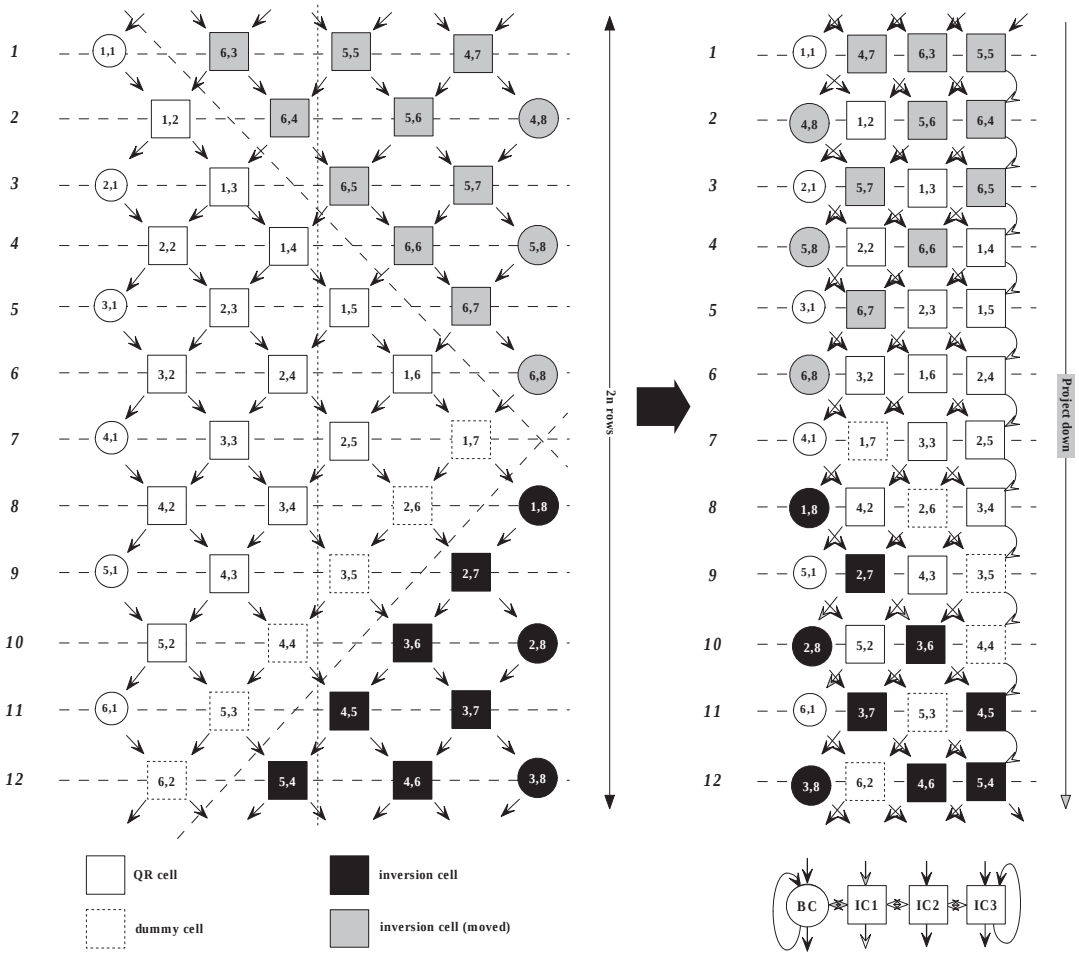


Figure 5.17: Processor array for even number matrix inversion (n=6)

Figure 5.17 shows the processor array for even number matrix inversion. The left part is the array structure after cutting and moving of the cells. The array cells with white box are the QR array. The dark color cells are for inversion and the gray ones are the cells in the inversion array after cutting and moving. The diagonal

lines indicate the boundary planes of the processor arrays, i.e., the IO plane and the boundary between the QR and the inversion arrays. The numbers inside the cells are the coordinate of the array elements to illustrate the cell sharing after folding as depicted in the left part of the figure. After folding, the cell consumption is split by half. When projected down in the normal direction of the schedule plane, the boundary cells (BC) perform both QR and inversion operations for these two types of arrays, and the cell is 100% utilized. The internal cells are switching among the QR, inversion and dummy operations in different time slots. The dummy cells are introduced to align the BCs and ICs within different schedule planes. The total number of schedule line is $2 \times n$, where n is the matrix size. The internal cell (IC) utilization is $(n-1)/n$, while before folding the cells of the array have half of the operation time idle.

The processor array for the matrix inversion with odd number is shown in Figure 5.18. To align the BCs into the same column, instead of introducing the dummy cells, the last schedule line is put on the top of the structure. After projecting down onto the linear array, the processors will start execution from the second scheduling line which is labeled with '1'. After folding at the center line and interlacing the two types of arrays cells, the schedule time is fully utilized for both the BCs and the ICs. The projected down linear array takes $2 \times n$ schedule time for one matrix inversion update, where n is the matrix size.

The result array structure is different for even and odd number matrices after folding and mapping onto linear arrays. The operations in the BC include dummy operation to propagate input data for the QR array, and the reciprocal operation in the inversion array. The Givens rotation (GR) of the QR array and the division operation of the inversion array are the operations performed by the linear mapped IC. The dummy operation is also included for the even size input matrix inversion architecture. The sequence to perform these operations within the cells for even and

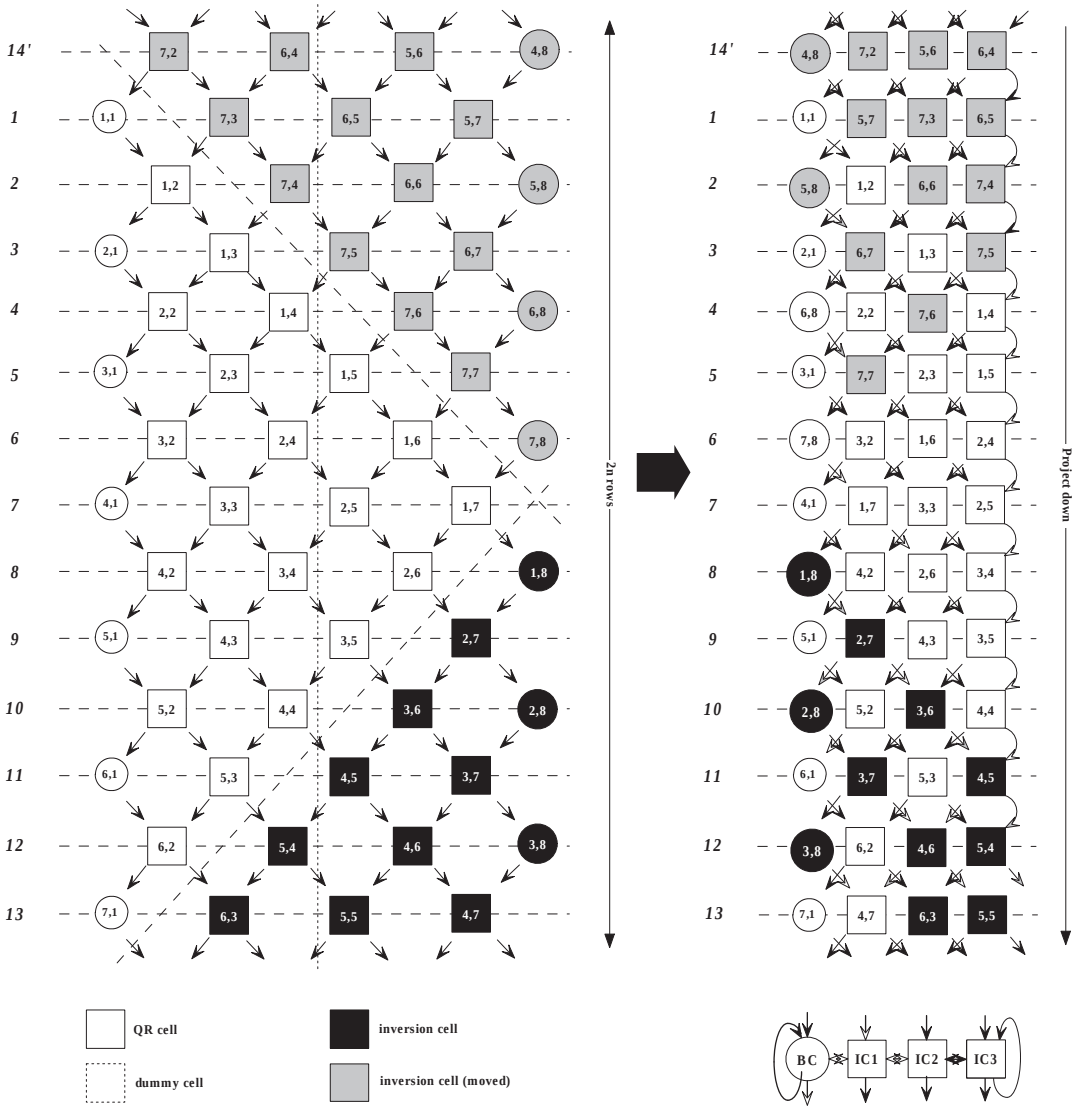


Figure 5.18: Processor array for odd number matrix inversion (n=7)

odd matrix is different. It is controlled by their scheduling timetable. The schedule sequence can be derived directly from Figure 5.17 and 5.18.

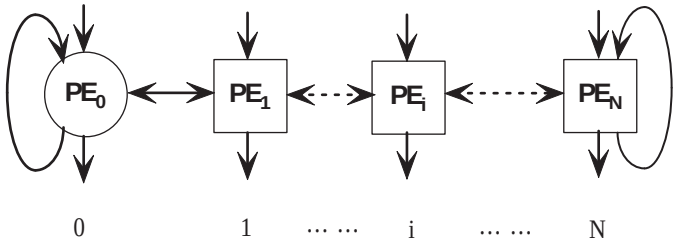


Figure 5.19: Mapped linear structure for $n \times n$ matrix inversion, where $N = [\frac{n}{2}]$

Generally speaking, for $n \times n$ matrix inversion, the scheduling timetable for the mapped linear array in Figure 5.19 is described below for different operation sequences, given i a value between 0 and N ($i = 0$ for BC) as the processor number shown in Figure 5.19, and j as the scheduling time index.

1. Even size matrix inversion

   - Internal Cell (IC):
     - Dummy: $j = n \times 2 - i + 1$ or $j = n + i$;
     - GR: $j \bmod 2 = (i + 1) \bmod 2$ and $j < n \times 2 - i$;
     - Division: $j \bmod 2 = i \bmod 2$, where

$$\begin{cases} j \geq n \times 2 - i : division \quad (uncut) \\ j < n - i + 1 : division \quad (cut\ and\ moved) \end{cases}$$

   - Boundary Cell (BC):
     - Dummy: $j \bmod 2 = 1$;
     - Reciprocal: $j \bmod 2 = 0$, where

$$\begin{cases} j > n : reciprocal \quad (uncut) \\ j \leq n : reciprocal \quad (cut\ and\ moved) \end{cases}$$

2. Odd size matrix inversion

   - Internal Cell (IC):
     - GR: $j \bmod 2 = i \bmod 2$ and $j < n \times 2 - i + 1$;
     - Division: $j \bmod 2 = (i + 1) \bmod 2$, where

$$\begin{cases} j \geq n \times 2 - i + 1 : division \quad (uncut) \\ j < n - i + 1 : division \quad (cut\ and\ moved) \end{cases}$$

- Boundary Cell (BC):
  - Dummy: $j \bmod 2 = 0$;
  - Reciprocal: $j \bmod 2 = 1$, where

$$\begin{cases} j > n : reciprocal \quad (uncut) \\ j \leq n : reciprocal \quad (cut\ and\ moved) \end{cases}$$

Note that the operations in the IC and the BC have two types of operation mode each, except for the dummy operation. The operation modes are the same as in the normal linear mapping method, which are depicted in Equations 5.1 to 5.4. And the condition for the mode switching is also the same as the previous linear mapping method, which is actually the switching control for the *Bi-z* CORDIC operation mode switching. The scheduling timetable can be either hard-coded inside control logic for each of the array processor or manipulated by the control signals from outside. The processors are designed with autonomous control logic inside each cell.

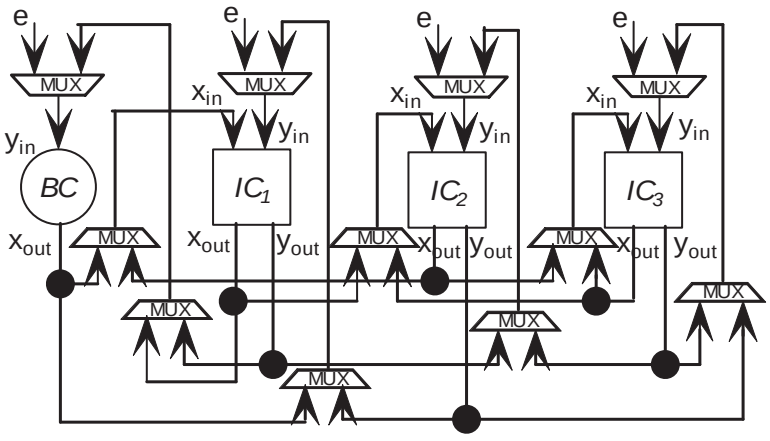


Figure 5.20: Linear array interconnection for interlaced mapping method

The IO control signals control the directions of the dataflow, which must be synchronized with the scheduling timetable accordingly. The processors in the mapped linear arrays are only connected with their neighboring processors. So the source data inputs for the internal cells are either from the left processor, the right proces-

sor (including dataflow feedback from itself) or the external data input. This is the same for the boundary cells. But as the boundary cells are only connected with the processor left to it, the data input source is either from the external input source or the output of the right processor. This output includes x_{out} and y_{out} of the right processor. Figure 5.20 demonstrates the processor interconnection via multiplexers. The control data for IO dataflow are coded as follows.

1. For boundary cell:
   - 0 - data input is from the external IO;
   - 2 - data input is from y_{out} output of the right processor;
   - 3 - data input is from x_{out} output of the right processor.
2. For internal cell:
   - 0 - data input for x_{in} is from the left processor, and data input for y_{in} is from the external IO;
   - 1 - data input for x_{in} is from the right processor, and data input for y_{in} is from the external IO;
   - 2 - data input for x_{in} is from the left processor, and data input for y_{in} is from the right processor;
   - 3 - data input for x_{in} is from the right processor, and data input for y_{in} is from the left processor.

The control data index j is defined as

$$j = c \mod M$$

where c is the cth clock cycle, and M is the period of scheduling table which is $M = 2 \times n$ for $n \times n$ matrix inversion.

Table 5.5 shows the control data scheduling tables for the IO multiplexers of matrix size n=4, 5, 6 and 7 respectively, with the control data specified above. The superscript 'o' means the PEs have the output result at that particular time slot. The superscript '*' shows the PEs are put into dummy logic (idle) at that time instance. A more general control data schedule formula can be derived based on Table 5.5 for any $n \times n$ matrix inversion as below.

Give PE_i, and control data index j, the control data of the IO multiplexers for PE_i is defined as:

1) for i=0 (BC):

$$c_{0,j} = \begin{cases} 0 & j = 1 \\ 2 & j \in odd,\ and\ \ j \neq 1 \\ 3 & j \in even \end{cases} \tag{5.5}$$

2) for $i \in [1, [\frac{n}{2}]]$ (IC):

$$c_{i,j} = \begin{cases} 1 & j = n - i + (n\ mod\ 2) \\ c_{0,j-i} & else\ if\ j > i \\ c_{0,j-i+2\times n} & else,\ (j \leq i) \end{cases} \tag{5.6}$$

The above scheduling of the control data can be directly mapped onto the IO multiplexers assuming the processor latencies are equal and to be unity. But in real scenario, most of the processing time of the mapped processors are greater than 1 clock cycle, and may not be equal to each other. To properly schedule the control and dataflow in this case, the processor re-timing needs to be considered, which is discussed in Section 5.5.3.

Table 5.5: Control Data Scheduling Table for $n = 4, 5, 6$ and 7

(a) n=4

CLK	PE_0	PE_1	PE_2
1	0	3	2^o
2	3	0	3^o
3	2	3^o	0
4	3^o	2	1
5	2	3^*	2
6	3	2	3^*
7	2	3	2^*
8	3	2	3

(b) n=5

CLK	PE_0	PE_1	PE_2
1	0	3	2^o
2	3	0	3^o
3	2	3	0
4	3^o	2	1
5	2	1	2
6	3	0	3
7	2	3	2
8	3	2	3
9	2	3	2
10	3	2^o	3

(c) n=6

CLK	PE_0	PE_1	PE_2	PE_3
1	0	3	2^o	3
2	3	0	3	2^o
3	2	3	0	3^o
4	3	2	3^o	0
5	2	3^o	2	1
6	3^o	2	1	2
7	2	3^*	2	3
8	3	2	3^*	2
9	2	3	2	3^*
10	3	2	3	2^*
11	2	3	2^*	3
12	3	2^*	3	2

(d) n=7

CLK	PE_0	PE_1	PE_2	PE_3
1	0	3	2^o	3
2	3	0	3	2^o
3	2	3	0	3
4	3	2	3	0
5	2	3	2	1
6	3^*	2	1	2
7	2	1	2	3
8	3	2	3	2
9	2	3	2	3
10	3	2	3	2
11	2	3	2	3
12	3	2	3	2
13	2	3	2	3
14	3	2^*	3	2

o: output

*: idle

5.5.2 Processor Architectures

By folding the two types of the processors in Figure 5.13 and Figure 5.14 into the boundary cell and the internal cell respectively, the same *Bi-z* CORDIC hardware engine can be time-shared to perform different operations, either for QR updating or inversion task. Figure 5.21 and Figure 5.22 show the internal architectures of the boundary cell and the internal cell respectively. The boundary cell is derived from mapping the processors in Figure 5.13 (a) and Figure 5.14 (a) together. Likewise, folding the processor for QR decomposition in Figure 5.13 (b) and the processor for division of Figure 5.13 (b) together, the internal cell structure is obtained. Note that each folded cell performs multiple functions in a time sharing fashion, so a scheduling table based on those discussed in Section 5.5.1 is hard coded as the control logic inside each of the processing cell in parameterisable Verilog code during implementation.

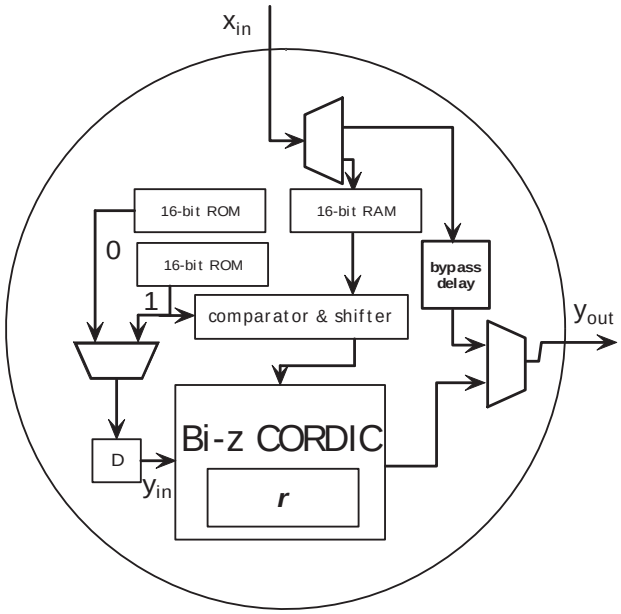


Figure 5.21: Processor architecture for boundary cell

For the *Bi-z* CORDIC in the interlaced mapped processors of the internal cell and the boundary cell, the operation tasks involved include dummy logic and reciprocal

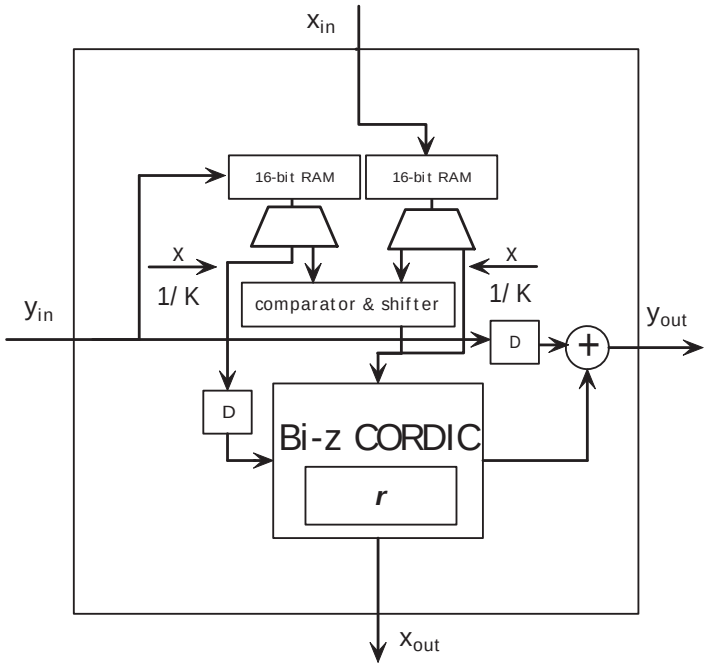


Figure 5.22: Processor architecture for internal cell

in the boundary cell, and GR and division in the internal cell. Each of the operations requires a series of *Bi-z* chain registers to store and reuse the rotation directions. So instead of 1-bit register for each micro-rotation step, a scalable memory is implemented for each of the micro-rotation step which forms a memory chain for all the rotation directions of every operation on the Z-path of the *Bi-z* CORDIC. The depth of each individual memory is equal to the period of the control scheduling cycle. Figure 5.23 shows the *Bi-z* CORDIC architecture with the register chains to perform different operation switching. The *Bi-z* CORDICs in Section 5.3 for the QR decomposition and the inversion have been merged together to form a single architecture. The selection of the QR or the inversion computing mode is determined by the scheduling table and controlled by the multiplexers inside the micro-rotation steps.

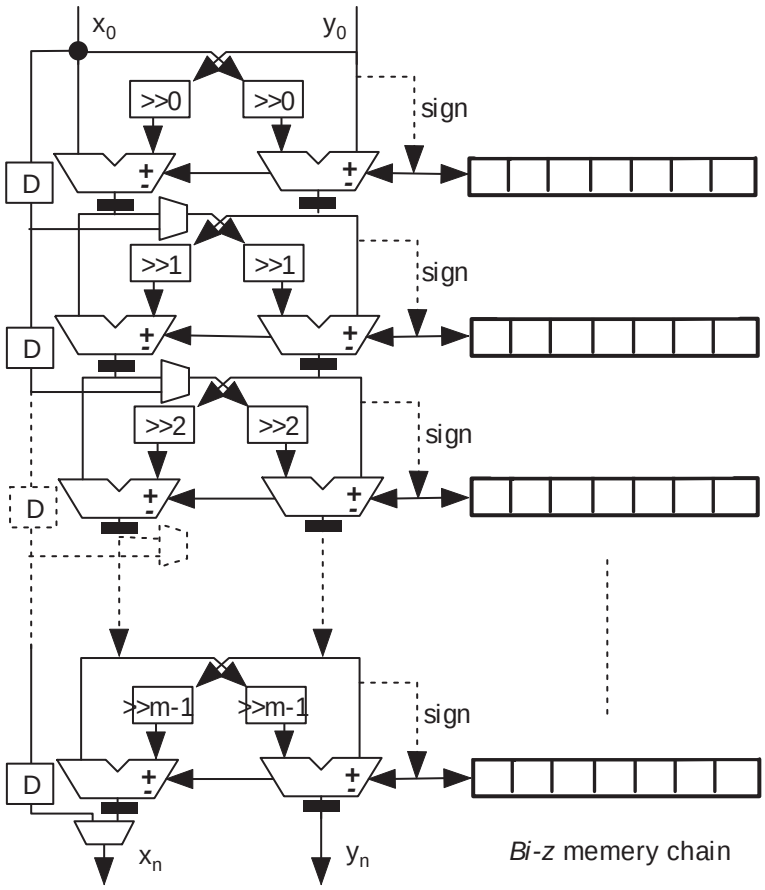


Figure 5.23: Pipelined m-tap *Bi-z* CORDIC structure for linear mapping arrays

5.5.3 Retiming

The matrix inversion mapping methods discussed so far assume that the processing cells for the QR and the inversion have 1 clock delay, which is ideal for problem analysis. For real case, due to the major latency contributor of the *Bi-z* CORDIC processing, the latency issue is far from ideal. The involving of non-unitary processor latency can affect the scheduling of the architecture greatly. Considering the array architecture in Figure 5.24 after introducing a latency equal to L_c between the interconnected cells, the total processing time will be stretched by the factor of L_c. That is to say, the next updating for the matrix inversion will begin after $M \times L_c$ clock cycles, where M is the number of the row in the array architecture. Due to the processor latency involved, the processor utilization is also decreased to $1/L_c$.

Thus, this is obviously not an optimal scheduling solution.

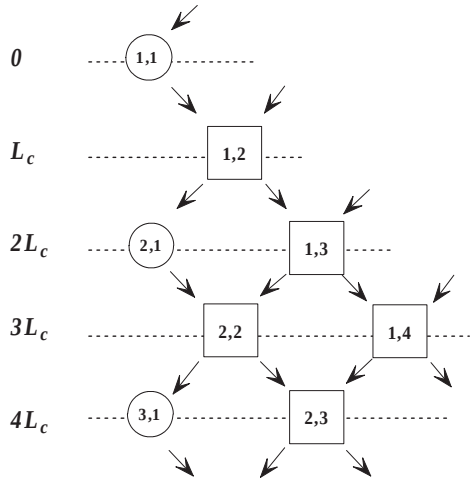


Figure 5.24: Array structure with stretched latency

To rearrange the scheduling sequence, the data input need to be fed in at a much faster rate. For example, instead of update the input data after $M \times L_c$ clock cycles, the processor will be scheduled immediately at the next available time slot for the next matrix inversion update, which might be the $L_c + 1$ clock cycles after the start of the first update. So within $M \times L_c$ clock cycles time, there will be an average number of L_c matrix inversion updates co-exist in the same array structure. Figure 5.25 shows the schedule of interlaced array processor with a cell latency L_c equal to 3 for a 6×6 matrix inversion. The sequence number -1 and 0 are the previous two updates which are the result of cut and folding for this mapping method, while some parts of the cells for update 3 will be extended to the next $M \times L_c$ clock cycles iteration. The period of scheduling is expanded by the factor of L_c for 100% cell utilization. The first number of each line is the clock cycle after rescheduling. The numbers inside the brackets are the row numbers for each matrix inversion update as in Figure 5.17, with superscript indicating the order of the update. The scheduling of interlaced mapping method for linear array with latency can be described more precisely in Table 5.6, where $L_c = 3$. t is an arbitrary time for general scheduling analysis. Only updates 1, 2 and 3 are depicted, which are shown in the schedule

Table 5.6: Schedule For 6×6 Matrix Inversion On Interlaced Array With Latency $L_c = 3$

Clock cycle	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8	t+9
Row schedule	1^1			2^1	1^2		3^1	2^2	1^3
Clock cycle	t+10	t+11	t+12	t+13	t+14	t+15	t+16	t+17	t+18
Row schedule	4^1	3^2	2^3	5^1	4^2	3^3	6^1	5^2	4^3
Clock cycle	t+19	t+20	t+21	t+22	t+23	t+24	t+25	t+26	t+27
Row schedule	7^1	6^2	5^3	8^1	7^2	6^3	9^1	8^2	7^3
Clock cycle	t+28	t+29	t+30	t+31	t+32	t+33	t+34	t+35	t+36
Row schedule	10^1	9^2	8^3	11^1	10^2	9^3	12^1	11^2	10^3
Clock cycle	t+37	t+38	t+39	t+40	t+41	t+42			
Row schedule		12^2	11^3			12^3			

timetable. The rest of the updates are simply the iteration of it. The left scheduling slots t+2, t+3 and t+6 are actually for the previous updates, while the slots t+37, t+40, t+41 are for the following updates which are left blank intentionally in this table for clarity.

5.5.4 Hardware Implementation

The implementation of the boundary cell and the internal cell is first analyzed. Each of the cells contains 1 *Bi-z* CORDIC. The latency of the cells is dominated by the latency of the CORDIC processor, which is determined by the input word-length. Each cell contains a certain number of multiplexers to select the input dataflow. The implementation cost of the boundary cell and the internal cell is shown in Table 5.7 for input data of 16-bit word-length.

A single $n \times n$ matrix inversion using the interlaced mapping method requires n processing cells with $6Pn^2$ clock cycles for non-retiming and $4n^2 + Pn$ time units after retiming, where P is the number of pipelined stage within each cell. With

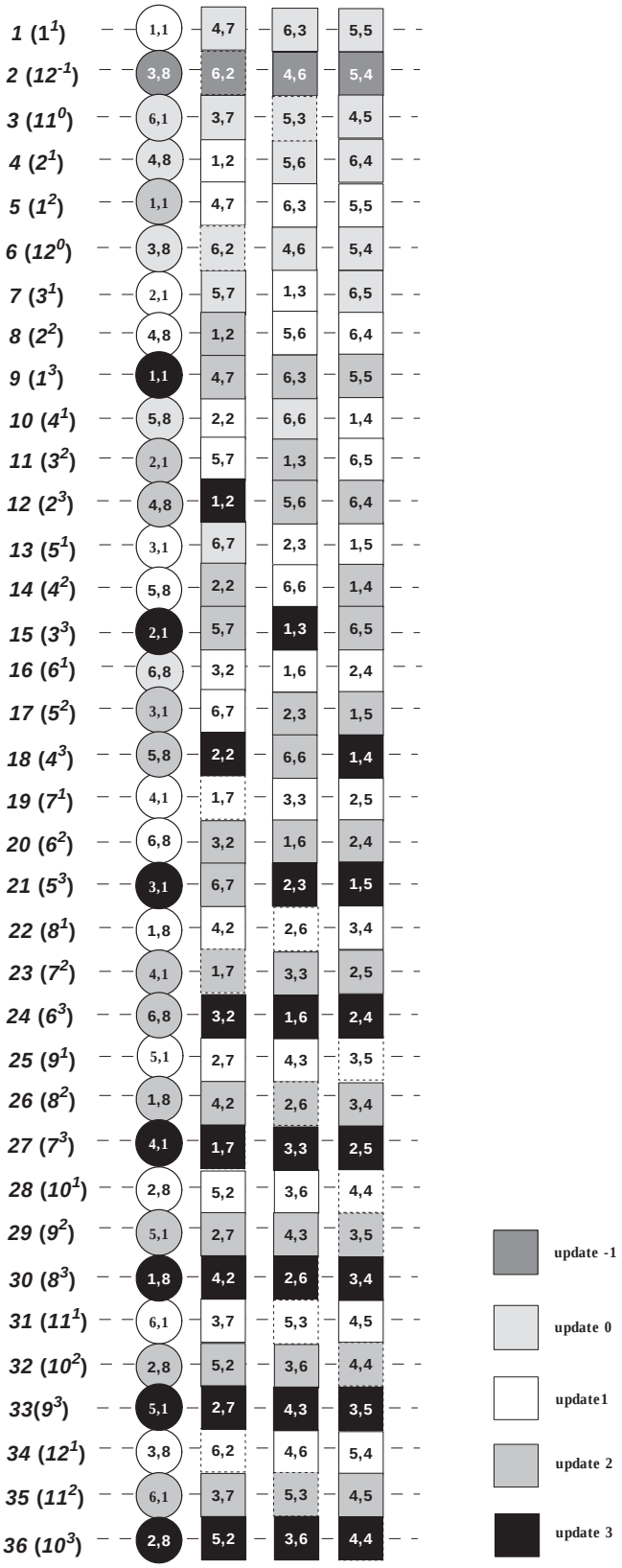


Figure 5.25: Schedule of interlaced array processor with latency ($L_c = 3$)

Table 5.7: Implementation Result Of Boundary Cell And Internal Cell

	Bi-z CORDIC	Mult	MUX	Slice	LUT	L_c
BC	1	0	2	534	996	19
IC	1	4	3	981	1858	19

Table 5.8: Comparison Of Matrix Inversion Methods

	No. of PE	Computation cycle	Processor utilization
El-Amawy [73]	n^2+n	$5Pn$	35%
LaRoche [75]	n^2+n	$4Pn^2$	-
Edman [78]	$2n$	$2P(n^2+n-1)$	62.5%
Linear	$2n$	$2P(n^2-n+1)$	62.5%
Interlaced	n	$6Pn^2$	100%
Interlaced(retiming)	n	$4n^2+Pn$	100%

comparison, the architecture with the maximum data throughput in [73] requires n^2+n processors with an average of 35% processor utilization, which has the lowest efficiency and the largest hardware consumption. The proposed linear mapping method has the same number of processor as Edman's method [78], but has a shorter computation latency. In LaRoche's design [75], the complexity of processor and the computation cycle are both in the order of square of matrix size which is the least efficient among the designs. The comparisons are summarized in Table 5.8. From the comparison, the proposed methods achieve 100% processor efficiency as well as the least hardware consumption. The linear array of the proposed methods also guarantees the number of processor increasing linearly with the size of the matrix, which is good for large matrices.

Another important feature of the proposed designs is the scalability and parameterisability for the change of different specifications, such as the matrix size and the word-length. Table 5.9 shows the implementation results for the matrix inversion

Table 5.9: Implementation Of Matrix Inversion With Different Specification Requirements For Interlaced Mapping Method

Matrix size	16-bit			32-bit		
	Slice	FF	Mult	Slice	FF	Mult
4	1,052	673	4	2,194	1,138	8
7	1,456	932	6	2,956	1,679	12
11	2,381	1,515	10	4,854	2,577	20
16	3,901	2,438	16	7,907	4,079	32
21	4,582	3,123	20	9,632	5,157	40
26	6,565	4,225	26	13,239	7,008	52
31	7,375	4,961	30	14,828	8,063	60
36	9,043	6,149	36	19,167	10,215	72
41	9,747	6,980	40	20,869	11,692	80
46	12,123	8,421	46	23,438	10,867	92
51	12,416	9,264	50	25,783	15,279	100
56	14,901	1,0696	56	30,429	17,743	112
61	15,601	11,954	60	31,428	18,509	120
66	18,958	13,458	66	37,231	21,428	132
71	19,535	14,361	70	39,418	23,548	140
76	22,630	16,117	76	44,475	24,951	152
81	22,776	17,446	80	45,649	20,722	160

with different matrix sizes and word-lengths. The hardware cost is in terms of slice, flip-flop and 18×18 -bit multiplier of Xilinx Virtex-II FPGA. The data representation can also be parameterized. The input data format can be adjusted to satisfy different input data range and the computation precision. For 16-bit input data implementation, Table 5.9 uses 12 bits for fractional number while for 32-bit design, 24-bit fractional bits with 7-bit integer are used as the data representation, with the highest bit for sign bit.

For the interlaced mapping method, the computation latency for a single matrix

Table 5.10: Latency Comparison Of Interlaced Mapping Methods

Matrix size	Interlaced $\mu m/cycle$		Interlaced(retiming) $\mu m/cycle$	
	16-bit	32-bit	16-bit	32-bit
4	21.93/1,824	54.54/3,360	1.66/140	3.31/204
7	67.13/5,586	168.73/10,290	3.95/329	7.23/441
11	166.79/13,794	399.48/25,410	8.38/693	13.66/869
16	352.05/29,184	876.28/53,760	16.02/1,328	26.08/1,600
21	629.00/50,274	1,493.97/92,610	27.06/2,163	40.31/2,499
26	946.15/77,064	2,278.69/141,960	39.26/3,198	58.01/3,614
31	1,334.20/109,554	3,273.48/201,810	53.98/4,433	79.95/4,929
36	1,848.95/147,744	4,182.57/27,2160	73.43/5,868	99.03/6,444
41	2,389.24/191,634	5,476.93/353,010	93.54/7,503	126.58/8,156
46	2,963.04/241,224	7,040.70/444,360	114.70/9,338	159.61/10,074
51	3,809.96/296,514	8,798.91/546,210	146.13/11,373	196.35/12,189
56	4,563.31/357,504	10,303.52/658,560	173.69/13,608	226.92/14,504
61	5,226.70/424,194	13,293.35/781,410	197.69/16,043	289.52/17,019
66	6,317.78/496,584	15,213.29/914,760	237.63/18,678	328.19/19,734
71	7,434.23/574,674	16,840.76/1,058,610	278.30/21,513	360.30/22,649
76	8,501.79/658,464	19,556.93/1,212,960	316.95/24,548	415.40/25,764
81	9,960.90/747,954	23,542.65/1,377,810	370.00/27,783	496.87/29,079

inversion is depicted in Table 5.10 with different matrix sizes and word-lengths. As can be seen, the computation cycle is increasing roughly in the ratio of square of input matrix size. The latency is also proportional to the input data word-length for the same mapping method. Retiming can greatly save the computation cycles since each of the matrix inversion updates will be pipelined inside the processors for every clock cycle. Thus tremendous reduction of computation time is achieved at the cost of only slight increasing of the scheduling complexity, which can be predetermined.

A comparison has been made in Table 5.11 for the 4×4 matrix inversion. All the

Table 5.11: 4×4 Matrix Inversion Comparison

	Latency (μs)	Cycle	f_{clk} (MHz)	Resource	ADP ($T \times A$)
LaRoche [75]	0.59	64	108.3	4,446	2,627
Edman [78]	1.75	175	100	2,948	5,159
Singh [72]	0.24	67	277	72K	-
Linear	2.05	352	172	1,353	2,773
Interlaced	21.93	1,824	83	1,052	23,070
Interlaced(retiming)	1.66	140	83	1,052	1,746

implementations have been targeting on Xilinx Virtex-II FPGA except for Singh's design [72] which uses 0.18 μm CMOS technology and achieves the best performance both in terms of speed and computation cycle. The resource usage for the rest of the designs is measured by the CLB consumption. The low latency and low ADP in LaRoche's design [75] are achieved by the high resource consumption. As reported, additional 101 18×18 embedded multipliers are used besides 4,446 CLB consumption for a single 4×4 matrix inversion. Edman's design [78] is comparable with the proposed designs both in terms of latency and speed, but costs more hardware resource thus high ADP. The proposed linear mapping method achieves a highest speed among all the FPGA designs while still maintains the low hardware cost. The optimal design of the lowest ADP is the interlaced mapping method after retiming, which reaches the maximum hardware efficiency at the relatively low latency cost as a result of rescheduling and computation folding. The high ADP in the interlaced mapping method without retiming is mainly due to the large delay between adjacent scheduled computing nodes which use the relatively slow CORDIC method for data processing before retiming.

Chapter 6

Dynamically Reconfigurable Computing Platform

6.1 Introduction

Traditional algorithms implemented on VLSI chips are static and fixed once the chips are fabricated. The functions might not be robust and flexible for some circumstances. For example, in the autonomous robotic systems [87], the control algorithm circuitry might need to be adjusted for different conditions. For this kind of systems to function correctly, several chips or several hardware modules inside the chip need to be implemented separately in advance, with each for every different control algorithms. This results in large circuit and board design and waste of hardware resources.

Reconfigurable hardware chips such as FPGA and CPLD, offer circuit and system designers more flexible hardware platform for algorithm prototyping and final products. The on-chip hardware resources can be reconfigured and thus the functionalities of the chips can be changed as many times as the applications require.

Reconfigurable hardware is working in a way much like the context swapping in a microprocessor chip as depicted in Figure 6.1. In such a system, different functions can time-share the same piece of hardware resource without interrupting the execution of other functions in the same chip. This makes it possible for several algorithms to be implemented in the same chip and swapped to the frontend when required.

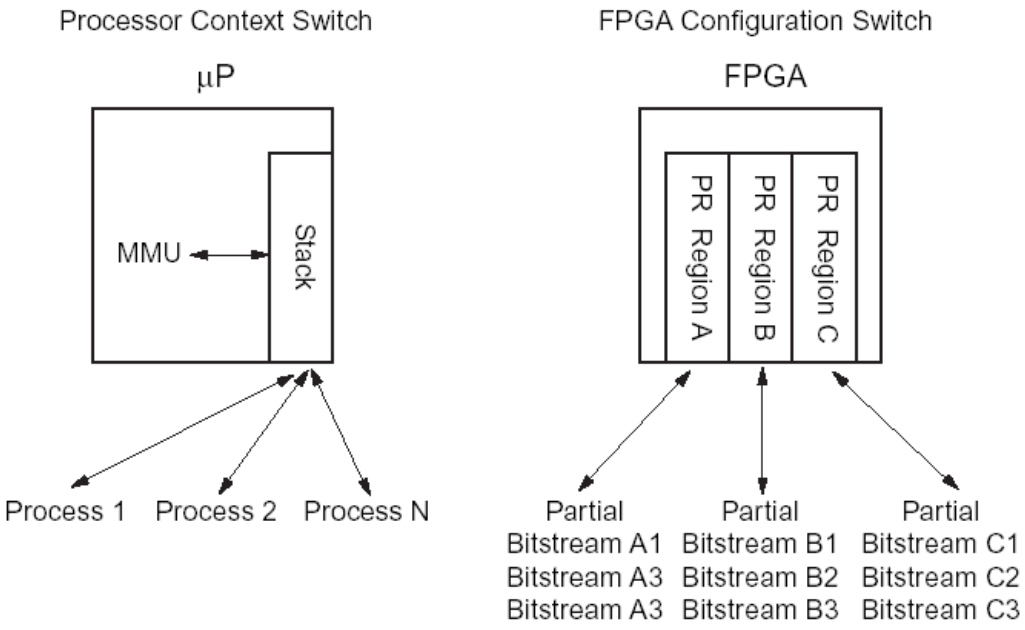


Figure 6.1: Analogy between microprocessor context switching and FPGA partial reconfiguration [88]

Another advantage of using partial and dynamical reconfiguration for system implementation is to maintain the functional continuity of the device. For instance, Figure 6.2 shows a reconfigurable system with multiple interfaces to the outside world, such as video link, radio and bus connection. These kinds of task-critical functions are required throughout the time of operation, and are not intended to be interrupted under any circumstances. With partial reconfigurability, the functions of all these real-time links are maintained while the utilization of the hardware resource is maximized by timely-swapping of different applications onto the hardware.

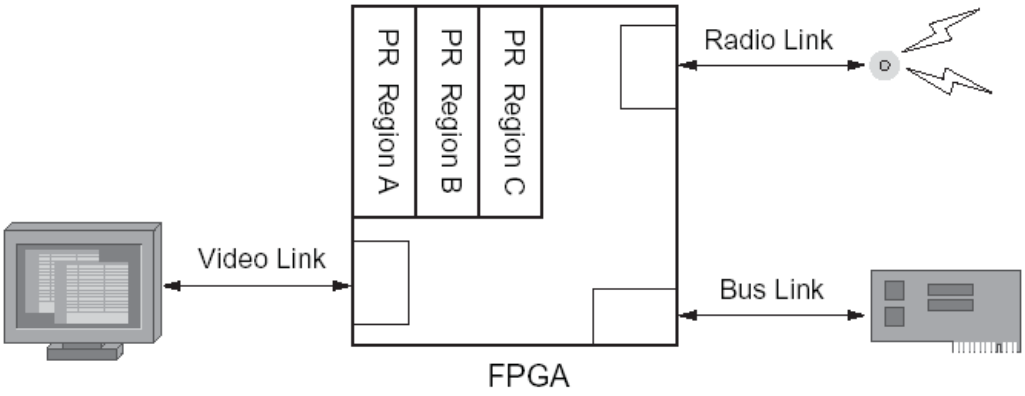


Figure 6.2: Maintaining real-time link during partial reconfiguration [88]

6.2 Background of Reconfigurable Systems

Although the proposed dynamically reconfigurable computing platform in this chapter is targeted at the Xilinx Virtex serial FPGAs, several well-known reconfigurable computing platforms are studied first. The architecture of the Virtex-II FPGA is introduced in the second part of this section, which unveils the resources that can be utilized for dynamical configuration in the proposed design.

6.2.1 Existing Reconfigurable Computing Platforms

Many projects were carried out to design various FPGA computing platforms catering for different computation intensive applications. Some well-known FPGA systems are summarized below. SPLASH-2 [32], [34] is a fine-grained loosely coupled system with potential partial reconfigurability for each of the 16 FPGAs on board, which can be conceptually viewed as a linear array of processing elements. This architecture makes SPLASH-2 a good candidate for systolic applications. Each FPGA is coupled with a dedicated memory that guarantees high memory bandwidth for SIMD algorithms. Though the application domain is usually restricted to systolic array style of computation, SPLASH-2 has been successfully used for image pro-

cessing, motion detection and DNA sequence matching. Since it is an FPGA based system, it inherits the slow speed of reconfiguration. Another burden is the low speed of integration. The system has a high capacity that is spread over several FPGAs and results in large dimensions and low working frequency, up to 40MHz.

RAW [34], [89] has multiple RISC processors, each having fine-grained logic as the reconfigurable components designed in a MIMD manner with multiple instruction streams for general-purpose computing. There are two reconfigurable parts. One is at the lower layer with a fine grained configurable logic in each tile. The other is the network that interconnects the tiles. The reconfigurable bottleneck has been greatly reduced by letting each tile having its own memory that supplies configuration for both fine grained configurable logic and network switch. The whole system is distributed without the need for a single control unit. Each tile is therefore independent, with its own data, instruction and switch memory, and without potential memory access bottleneck. Most of the resources are under the control of a compiler. Consequently, the compiler is more complex and tends to eliminate user interventions during compile time.

Xputer [31], [34] is a medium-grained closely coupled reconfigurable computer, with an array of reconfigurable ALUs in a mesh-type interconnection network controlled by a control unit and an address generation unit. Configuration data itself is supplied in the same way as the process data, with the only difference in the state of configuration bit. In this way, the overall configuration speed is increased. Another speedup is achieved through partial reconfigurability: each programmable unit can be configured separately, at any time, depending on the state of the configuration bit. However, its memory architecture represents a potential bottleneck of this system. Data supplied through a single I/O bus and cached in a FIFO or a register file. This is sufficient for data-like algorithms where it does not slow down the system. But some SIMD algorithms, dealing with large amount of data (such as image

processing), will suffer from the low memory bandwidth.

Matrix [34], [90] approach proposes the design of a basic unit interconnection network which can be configured for operation in conventional approaches, such as systolic arrays, VLIW or SIMD fashion. The base of the unit is an 8-bit ALU coupled with a 256-byte memory. The unit is supplied with data through network ports, which have three levels of network interconnections: neighbor, medium and global. Configuration of a basic unit is obtained from other units, through the network ports, as ordinary data. Thus it greatly speeds up the reconfiguration process. Furthermore, the integration of certain amount of memory onto the same die with a configurable logic alleviates the potential memory bottleneck. Although the Matrix system offers the possibilities for implementation of a large variety of programming approaches, none of the compiling tools for generating this code has been presented in [90].

RaPiD [34], [91] system is a representative of a medium-grained, closed-coupled architecture. This design is organized as a linear array of reconfigurable processing units with datapath parallelism in the temporal domain. Each unit has several different elements: ALU, register, multiplier and memory. All the elements are connected to the reconfigurable datapath bus. There are two kinds of control signal: static and dynamic. The static control signals are used to load the configuration on to the array. The dynamic signals, which are generated by the integrated RISC core, control loops and dynamic events. Configuration lines form a pipeline, similar to the data one, thus enabling fast per-cycle reconfiguration. A limited amount of local memory is provided, which leads to potential bottleneck.

Morphosys system-on-chip [30] is a coarse-grained closely coupled run-time reconfigurable system. It has a matrix of 2-D reconfigurable cells (RC) that support multi-context dynamic reconfiguration. The use of the on-chip multi-context memory and two-set frame buffer offers potential overlapping of the reconfiguration and

data transfer and high parallelism is achieved. Many research works targeted on applications and kernel scheduling of Morphosys have been carried out [92], [93], [94]. And there are still many research studies on the configware industry [95].

6.2.2 Xilinx Virtex-II Architecture

This project uses Xilinx Virtex-II FPGA device as the target device. The main reason to choose this target device is its ability to support partial configuration. And it is the first Xilinx's FPGA with embedded multipliers for algorithm acceleration.

Each Virtex-II device contains configurable logic blocks (CLBs), input-output blocks (IOBs), block RAMs, multipliers, digital clock managers (DCM) and global clock multiplexer buffers for clock resources, programmable routing, and configuration circuitry as depicted in Figure 6.3 [96]. These logic functions are configurable through the configuration bitstreams. The configuration bitstreams contain a combination of commands and data. The configuration bitstreams can be read and written through one of the configuration interfaces on the device - SelectMAP^{TM} interface, master/slave serial interfaces, or the Boundary-Scan interface.

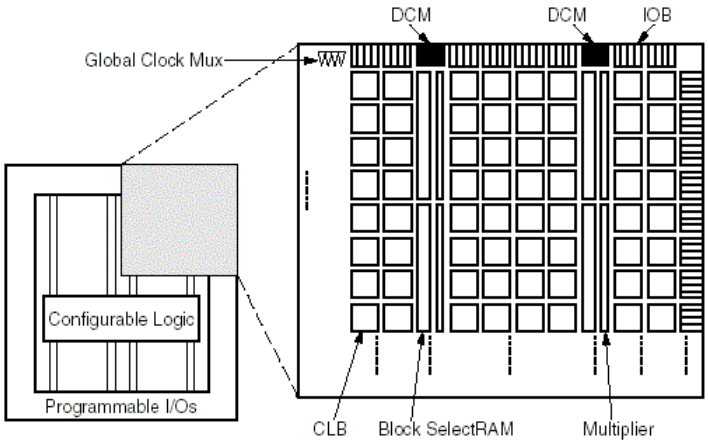


Figure 6.3: Virtex-II Architecture Overview [97]

The configuration bitstream is a collection of configuration bits. The bitstream is

a series of configuration commands and configuration data. Configuration is done by writing some or all the configuration commands into the configuration memory through the desired interface followed by the configuration data.

The Virtex-II configuration memory can be visualized as a rectangular array of bits. The bits are grouped into vertical frames that are one-bit wide and extend from the top of the array to the bottom. A frame is the atomic unit of configuration - it is the smallest portion of the configuration memory that can be written to or read from. Frames are grouped together into larger units called columns. Each device contains one center column that includes configuration for the four global clock pins. Two IOB columns represent configuration for all of the IOBs on the left and right edges of the device. The majority of columns are CLB columns each of which contains one column of CLBs and the two IOBs above and below those CLBs. The remaining two column types contain the block RAM: one for content and the other for interconnect.

The block RAM is true dual-read/write port synchronous RAM, with 18K memory cells. Each port of the block SelectRAM^{TM} memory can be independently configured as a read/write port, a read port, or a write port, and each port can be configured to a specific data width. Both ports are functionally identical. CLK, EN, WE, and SSR (for Synchronous Set/Reset) polarities are defined through configuration. The Virtex-II block SelectRAM supports various configurations, including single-port and dual-port RAM and various data/address aspect ratios. The memory configuration varies from 512 $\times$ 36-bit to 16K$\times$1-bit.

A Virtex-II multiplier block is an 18-bit by 18-bit 2's complement signed multiplier. Virtex-II devices incorporate many embedded multiplier blocks. These multipliers can be associated with an 18-Kbit block SelectRAM resource or can be used independently. They are optimized for high-speed operations and have a lower power consumption compared to an 18-bit x 18-bit multiplier in slices.

Configurable logic blocks (CLB) are organized in an array and are used to build combinatorial and synchronous logic designs. Each CLB element is tied to a switch matrix to access the general routing matrix, as shown in Figure 6.4. A CLB element comprises 4 similar slices, with fast local feedback within the CLB. The four slices are split in two columns of two slices with two independent carry logic chains and one common shift chain. Each slice includes two 4-input function generators, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements. Each 4-input function generator is programmable as a 4-input Look-up Tables (LUT), 16 bits of distributed SelectRAM memory, or a 16-bit shift register element. The output from the function generator in each slice drives both the slice output and the D input of the storage element. Figure 6.5 shows a more detailed view of a single slice.

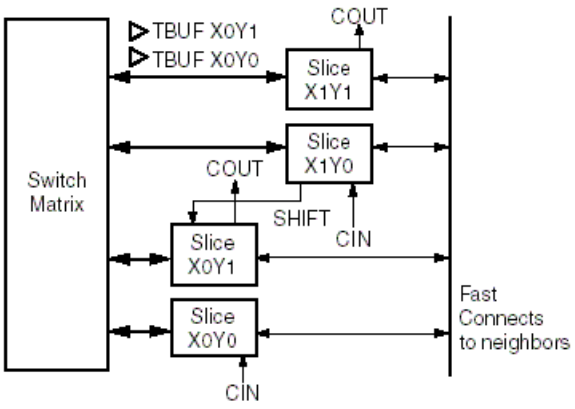


Figure 6.4: Virtex-II CLB element [97]

6.3 Dynamically and Partially Reconfigurable Platform

There are two main different approaches for reconfiguring a partial reconfiguration (PR) system, namely module based and difference based. The module based is

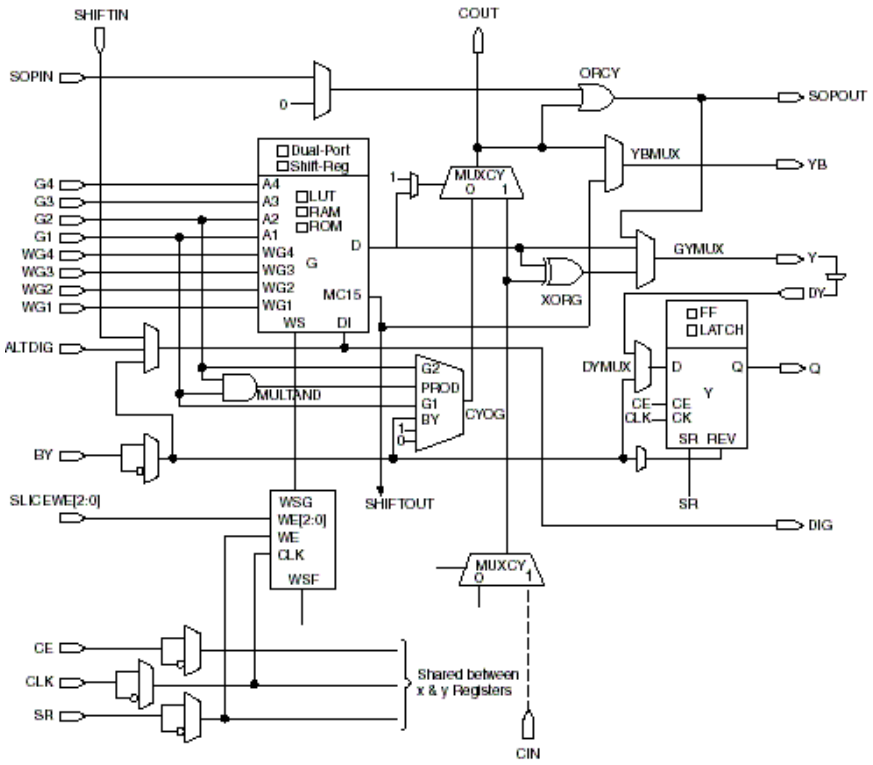


Figure 6.5: Top Half of the Virtex-II Slice Internal logic [97]

for large configuration change and the difference based is for manipulation of small number of configuration bits.

6.3.1 PR on FPGA

FPGA is a kind of generic programmable logic device made up of memory-based lookup tables, logic cells, registers and interconnection lines to implement logic functions with reconfigurability. Today's FPGA contains large amount of dedicated functional blocks such as memory and multiplier. The functions implemented can be altered by changing the configuration bitstream. The bitstream is technology dependent and generated by an implementation software tool offered by a specific device vendor. This bitstream determines the logic functionality, interconnection and physical routing information inside the device. Some FPGA devices provide

partial reconfigurability. The bitstream is composed of only parts of the device component information. Therefore, it is now possible to change a subset of the FPGA resources while maintaining the other parts intact. This allows the unaffected components to perform their own executions without interruption. Xilinx FPGAs such as Virtex, Virtex-II, Virtex-II Pro, Virtex4 and Virtex5 offer the capability of partial reconfiguration. Both styles of partial reconfiguration, module based and differential based, are possible [98].

In the difference based partial reconfiguration, the designer makes small changes to the low level design implementation such as lookup table equations, block memory contents, I/O standards, etc., normally done manually in the *FPGA_Editor* [98]. After the changes have been edited and saved, the partial bitstream generation function is called and the bitstream containing only the difference between the two designs is produced. This partial bitstream is usually much smaller compared to the full bitstream for the whole device. And the configuration switching between these two implementations is very fast. The difference based partial reconfiguration flow is for design switching with small changes. The components and designs that are allowed to be altered are very limited.

In the module based partial reconfiguration, the whole design implementation can be split into smaller modules based on the Xilinx Modular Design methodology [99]. Each module must go through the entire implementation flow separately from the HDL or EDIF netlist input, area and timing constraint specifications, synthesis, place and route and implementation to generate an independent partial bitstream file. The whole system design may contain fixed logic and reconfigurable parts as shown in Figure 6.6. The life time of the fixed part spans the entire processing period while the reconfigurable logic is time variable and can be swapped according to the requirement of the application. The modules are linked together by a special communication interface called bus macro. This bus macro is predefined data

exchange channel for adjacent modules, both fixed and reconfigurable parts, and is placed on the fixed predefined place. For each module, the I/O blocks for the signals going through nearby modules are defined to be connected to the bus macro only. A complete initial configuration bitstream for both fixed and initialization of reconfigurable parts is generated and used to program the whole device first, then the partial reconfiguration for each part can be loaded on demands.

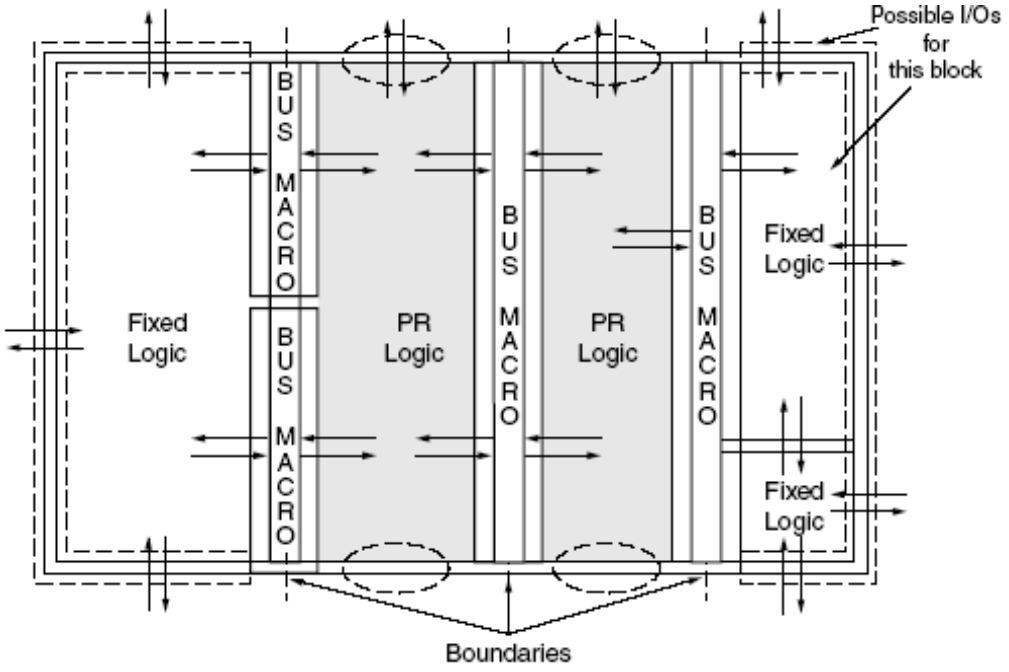


Figure 6.6: Design Layout with Two Reconfigurable Modules [98]

The module based partial reconfiguration has some placement constraints: (1) The size and the position of each module are fixed. (2) All logic resources encompassed by the width of reconfigurable module are parts of reconfiguration components in the corresponding partial bitstream. This includes block memories, IO blocks (IOBs), TBUFs, multipliers and routing resource. (3) The reconfigurable modules can only communicate, except for communication through IOBs to the off-chip components, with neighboring modules and it must be done through the bus macro (see Figure 6.6). (4) Xilinx states that no global signal is allowed to cross modules. This can be a problem if multiple modules or modules in alternate positions or cross context

layers need to communicate. In this work, this problem is solved by a proposed context-based partial reconfiguration flow in the following section for the proposed PR system platform.

6.3.2 The Proposed DPR Platform

The proposed dynamically and partially reconfigurable (DPR) platform consists of the system architecture and the hardware substrate mapping. The basic system architecture contains three modules, as shown in Figure 6.7, the fixed module, the reconfigurable module and the bus macro control system for context based partial reconfiguration. The fixed module contains a Microblaze 32-bit soft RISC processor with separate data and instruction access bus for fast data access and instruction execution on full speed. This processor serves as a coordinator between the host PC and the reconfigurable coprocessor, which can be used in the reconfiguration process, including supervising on-chip self-reconfiguration of the reconfigurable part, accessing data in both the on-chip and external memory and communication with the PC through RS232 and PCI bus. The reconfigurable part is allowed to implement user-defined functions. In this work, it is configured as a data-intensive matrix computation engine. The proposed bus macro system is used in the context-based reconfiguration design flow. The context is the implementation of sub-tasks in the bitstream format and stored in the layer of configuration memory and used for one-time continuous reconfiguration process at a given time slot.

6.3.3 System Architecture

DN3000k10 FPGA board from the Dini Group [100] is used for this work. The board can be stuffed with up to five Xilinx Virtex-II FPGAs, four 512K x 36-bit Synchronous Pipeline Burst SRAMs and many header pins and clock networks that

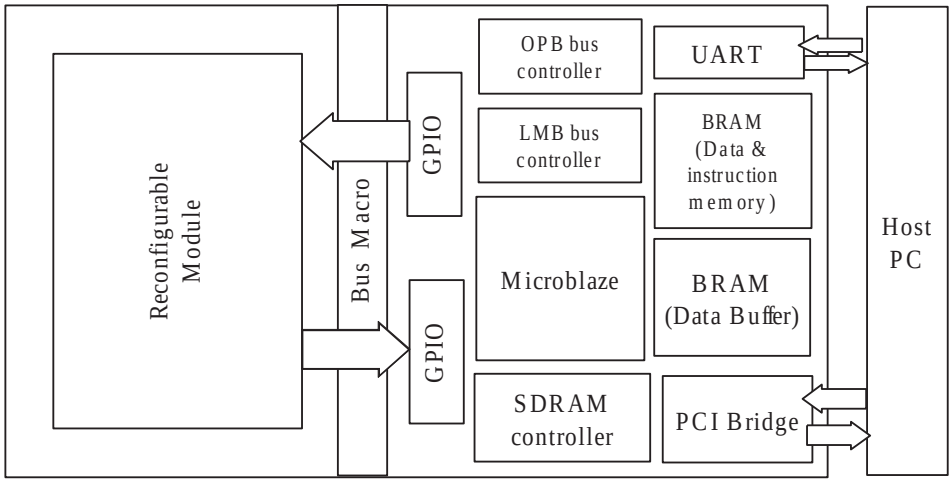


Figure 6.7: Proposed self reconfigurable platform

the user can choose for board configuration. The board has its own configuration channel implemented by an on-board μP and a Xilinx CPLD. The configuration file is stored inside a SD flash card that can be used to copy the program bitstream file from PC and plug into the on-board card reader. The μP reads the configuration file from the SD card and configures the FPGAs using SelectMAP configuration mode with the CPLD as the configuration I/O interface. The configuration file has to be copied and pasted manually using a SD card. Since the configuration method that this board offers does not meet the dynamic partial reconfiguration requirement in which the reconfiguration file can be loaded on application demand, a new configuration channel is implemented in this work. As the goal is to accelerate the matrix computation with parallel hardware, minimizing the configuration time is one of the critical issues to be considered in the proposed design. Xilinx offers three kinds of configuration methods: JTAG, Serial and SelectMAP, among which SelectMAP has the highest configuration bandwidth of up to 50 MByte/sec running at 50 MHz. Self-configuration offers the on-chip configuration solution that enhances the configuration speed. Xilinx ICAP [101] uses the SelectMAP bus protocol as an on-chip access port to the device configuration memory. This ICAP is used to perform the self-configuration. The three modules of the proposed platform are

detailed as follows.

Fixed module: The fixed module is composed of a Microblaze processor and its peripherals: a UART for serial communication with the PC for debugging and monitoring purpose and a PCI bridge core for transporting data and configuration file from the host PCI protocol to the Microblaze On-chip Peripheral Bus (OPB) processor system in both directions. The Local Memory Bus (LMB) controller fetches the instruction and data stored in the block RAM using a separate bus channel. The FPGA on board has a direct connection with the PCI interface on the host PC, so a PCI core is adopted to bridge the interface between the PCI and the local Microblaze bus system. The PCI interface is used for both data communication and device configuration. The high throughput of the PCI bus, 132MBtye/sec for the 33MHz 32-bit standard, allows fast transfer of reconfiguration bitstream from the PC to the FPGA board. It can satisfy the SelectMAP configuration speed of up to 50 MByte/sec. The on-chip block RAM with 2 separate access ports is used to act as a configuration and data buffer for the on-chip PCI interface. This shields the FPGA system from the host PC and allows them to work at different speed rates. The configuration bit size for Virtex-II devices XC2V4000 on the board is 15,659,936 and this far exceeds the total memory storage of the on-chip block RAM. The memory space on the host PC is used as a virtual memory pool of the reconfigurable platform. The memory on the host PC is mapped to the internal memory address space of the Microblaze system through the PCI bridge. The Microblaze thread is used to monitor and control the configuration status. If the buffer is about to be filled up, the PCI transfer is halted by keeping the TRDY# signal de-asserted until the buffer is cleaned up. If there is no bitstream inside the buffer when the configuration process is in progress, the configuration clock CCLK is turned off and the process is suspended until the bitstream data is available again.

Reconfigurable module: The reconfigurable module is used for implementing

the computing engine in a time-shared swapping fashion. The partitioned module contexts are loaded by the context scheduler in the host PC through the on-chip self-reconfiguration channel, which is controlled by the Microblaze SelectMAP configuration thread. The reconfigurable part is connected to the Microblaze General Purpose IO (GPIO) block through the bus macro and makes itself a peripheral to the Microblaze on the proposed Bus macro peripheral system.

Bus macro peripheral system: The Xilinx bus macro provides some connections between the reconfigurable modules in the module-based partial reconfiguration flow. This bus macro is implemented with internal 3-state buffers to ensure the fixed connectivity in the reconfiguration process. The bus macro is one directional to guarantee interface insulation for regions under reconfiguration. There are some constraints that limit the use of the Xilinx bus macro. The reconfigurable modules can only communicate with their neighboring parts, left or right, through a bus macro. No global signal is allowed throughout the reconfigurable system (e.g. Reset) except for the system clock which uses a separate configuration network and frame column. Data communication will have the continuity and caching problem in the case when there is data flow among the adjacent swapping contexts. Data must be cached before the context changes status if there is any connection between adjacent reconfigurable context modules. To eliminate these problems and enhance the data communication among the reconfigurable peripheral modules and the contextual related modules, a customized bus macro was developed with dedicated long line that traverses throughout the whole reconfigurable part and with caching capability.

6.3.4 Substrate Mapping

The initial design substrate consists of the fixed module, the bus macro system and some dummy reconfigurable modules. This top level full bitstream is to initialize the whole device before the reconfiguring process begins. The initial substrate as well as

the reconfigurable parts were developed based on the Xilinx module-based partial reconfiguration flow [98]. The proposed bus macro system consists of the bus macro connection, the control logic and the storage memories. The Xilinx bus macro allows reconfigurable modules connected with only the neighboring modules. A customized bus macro with arbitary width and signals proposed by Jens Thorvinger [102] is adopted as the bus macro module file for the initial top level design with some modifications as described below. The description of the bus macro is generated by *Perl* script in the XDL (Xilinx Design Language) format [103], which is a text-based layout description language. It is then converted back to the hard macro format.

The data manipulation during both the input stage and the context swapping stage is a time consuming process especially for large data size. It can result in bus congestion and memory access bottleneck. This is alleviated by using block RAM dispersing throughout the whole device. The block RAM has two sets of completely independent input output ports with clock control interface. Since the matrix computation that this project is focusing on has the parallel and distributed data manipulation fashion in nature, these distributed memory blocks are used as the local data storage as well as the context caching device for the reconfigurable swapping operation. The memory columns in the Xilinx Virtex-II device can be considered as static modules in the initial design plane. These static memory modules are embedded inside each reconfigurable module encompassing them. Thorvinger assumes that all the reconfigurable modules connect to the master-slave memory bus with uniform interface. This may not be the case for variety of a reconfigurable module. To alleviate this problem, the bus macro port for each reconfigurable module in the proposed design is connected to the embedded memory column by the vertical lines. The memories are addressed by the static control logic block in the master-slave memory protocol bus. The reconfigurable modules will only need to exchange and update data with the local storage.

The control logic block can be directly mapped into FPGA with dedicated hardware modules or using a microprocessor. In this work, the Xilinx Microblaze is used as the partial configuration supervisor. The soft μP is idle for the rest of the processing time. Therefore, the control logic for the inter-module communication is implemented also in the Microblaze. The bus macro arbiter decides who has the right to talk to the bus and the interrupt handler deals with the data requests from the reconfigurable modules as well as the reconfiguration request from the user application running on the Microblaze.

The position of the static module on FPGA is chosen to facilitate the external connections. The physical connection of the PCI interface in the FPGA of the board is at the bottom right part of the IO blocks. The internal configuration access port (ICAP) of VirtexII FPGA for the self-configuration access is also located at the bottom right part of the chip. Therefore the FPGA is split into two parts with the right part for the static control logic and the left for the reconfigurable modules.

Each reconfigurable module as well as the initial top-level configuration plane is implemented as a separate project in the Xilinx ISE. A correct design is the key element to a successful partial reconfiguration. Therefore, the functions for each module must be fully verified in the modular design phase. To implement the initial design, the Xilinx ISE project includes a bus macro file with all the reconfigurable blocks and the static module declared as black boxes. After synthesis, each project generates its own module netlist file. The initial budgeting phase is to assign the top level constraints to the design. The physical constraint includes the area constraints of each module, the bus macro position and the memory blocks for each reconfigurable module. Each active module is annotated with the top-level constraint file produced by the initial budgeting phase. After the map process and the place and route (PAR) process in the Xilinx ISE, physically implemented reconfigurable modules are generated. The static and initial reconfigurable modules are assembled in

the assembling phase using the initial budgeting constraints and the corresponding physically implemented modules from the previous design flow steps. The initial full bitstream and partial configurable bitstream are generated by using the Bitgen function in ISE.

6.4 Experimental Setup

The experimental setup consists of two parts: the implementation of the dynamically self-reconfigurable platform on FPGA, and the case study of the dynamical and partial reconfiguration with the applications swapping between different matrix computations such as matrix multiplication and/or matrix inversion.

6.4.1 Hardware Platform on FPGA

The fixed module and the bus macro were implemented. Figure 6.8 depicts the layout of the fixed module in the proposed self-reconfiguration platform. No signals are allowed to cross the boundary between the fixed and the reconfigurable modules except through the bus macro. The proposed Microblaze embedded system with the PCI and the other peripherals shown in Figure 6.7 was implemented as the fixed module on the hardware platform. This Microblaze system including peripherals like PCI bridge, UART, SDRAM controller, etc., occupies 3,850 slices. It consumes about 16% of the overall resouces. The signal direction of the bus macro is unidirectional, either left-to-right or right-to-left. The bus macro is two-CLB wide and can route up to 8-bit signals each. The bus macros after initial budgeting are indicated in Figure 6.9 with the box. Figure 6.10 shows the detailed left-to-right bus macro implementation straddling cross the boundary between the partial reconfiguration module and the static module with the center line as the boundary line.

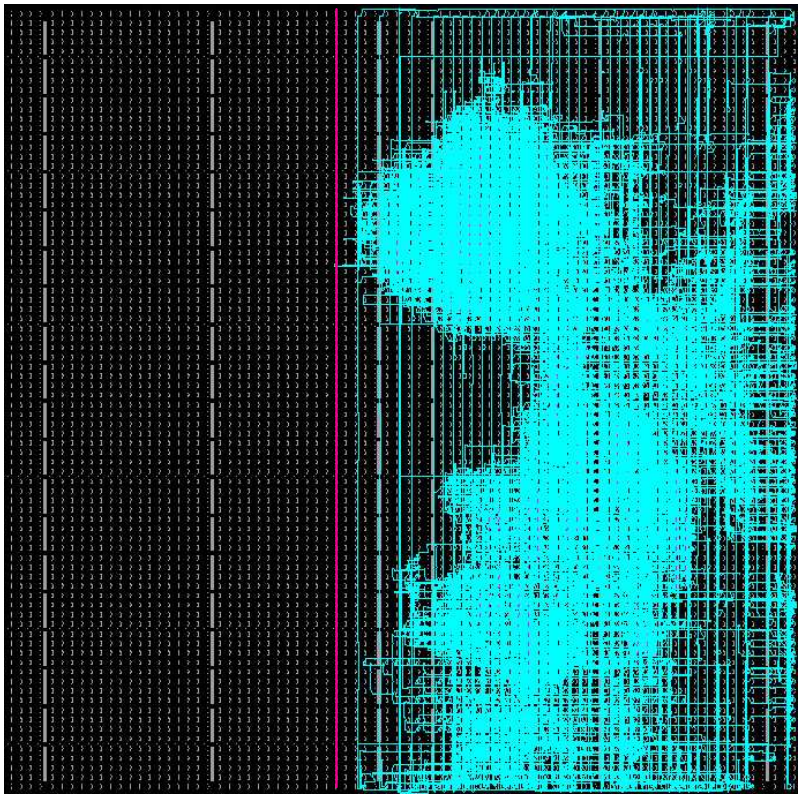


Figure 6.8: Routed FPGA layout of the fixed module in the proposed self reconfiguration platform in an Xilinx XC2V4000 device

6.4.2 A Case Study

Assuming there is an application requiring either matrix multiplication and/or matrix inversion operation where the selection of the operation and the size of the matrix is dependent on the input requirements. Normally for implementation of this application on hardware, both the matrix multiplier and the matrix inversion modules are necessary to be coexistent on the chip at the same time. The functions are switched between these two blocks. But in any time instance, only one module is in use. This leads to the waste of hardware resources and limits the size of the matrix to be tackled. With reconfigurable hardware, two hardware modules implemented for matrix inversion and matrix multiplication can time-share the same piece of hardware resource and swap "on line" when required by the application, without occupying the silicon area at the same time.

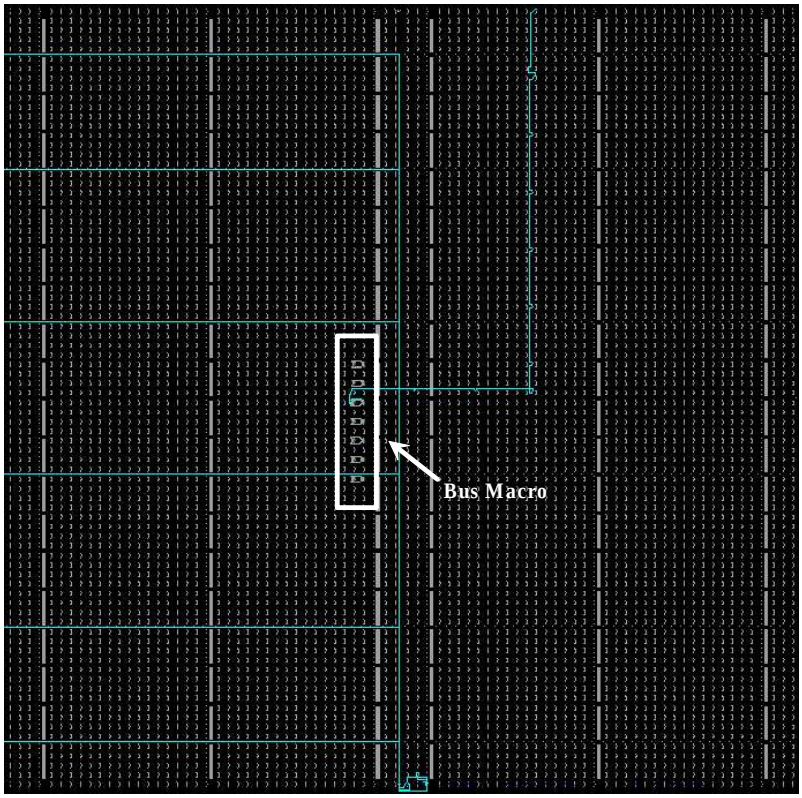


Figure 6.9: The bus macros in the proposed self reconfiguration platform

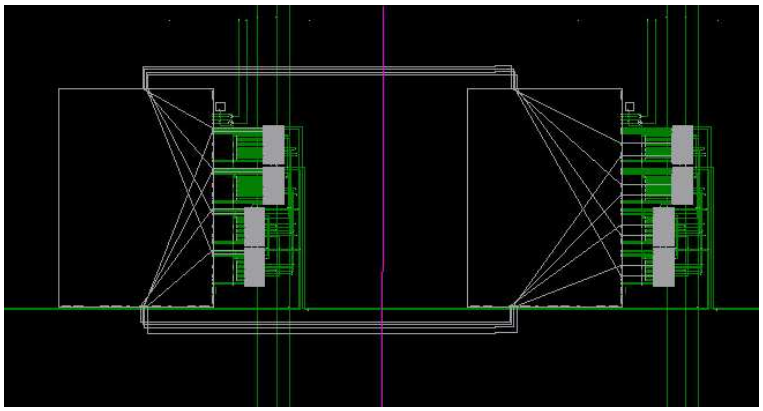


Figure 6.10: Bus macro cell

Suppose on part of hardware accelerator for a software defined radio (SDR) system [104] requires matrix multiplication and/or matrix inversion for different acceleration scenarios. SDR allows one radio platform to service multiple cellular standards, and must meet today's reconfigurability requirements to adapt to emerging wireless communication standards. However, availability, cost and the diverse market have

limited its deployment. The use of partial reconfiguration, multiple applications or waveforms can be supported on a single set of processing resources, allowing one or more waveforms to be dynamically changed within the device while other waveforms continue to be processed without interruption. This reduces the cost of the hardware platform by supporting reconfigurable (software-defined) architecture at low cost.

Multi-Input Multi-Output (MIMO) systems have drawn increasing attention in the research and communication industries due to the high channel capacity and high reliability on transmission. The high performance gained by MIMO systems made them widely adopted in the next generation wireless communication systems. To fully exploit the potential performance that MIMO systems have, the baseband wireless communication MIMO receiver must meet the real-time challenge of communication requirements, fast processing speed, low hardware cost and low latency. Dedicated circuits for MIMO receiver have to be developed for high data throughput, low latency and high computation accuracy.

One of the critical components in a MIMO receiver is the matrix inversion, which involves many complicated arithmetical operations such as multiplication, division and square root, etc. Matrix transformation and inversion require massive computing capabilities. The situation is worsened when the problem size is large and varies to cater for different applications when the number of transmitters and receiving antennaes changes.

A MIMO system with M transmitters and N receivers is shown in Fig. 6.11. In the flat fading channel condition which can be modeled as linear and time-invariant, the received data can be given in Equation 6.1 where $\mathbf{x}$ is the N-dimensional received signal for the M-dimensional transmitted symbol $\mathbf{s}$ transmitting through $M \times N$ complex valued flat fading channel $\mathbf{H}$, and corrupted by a M-dimensional additive

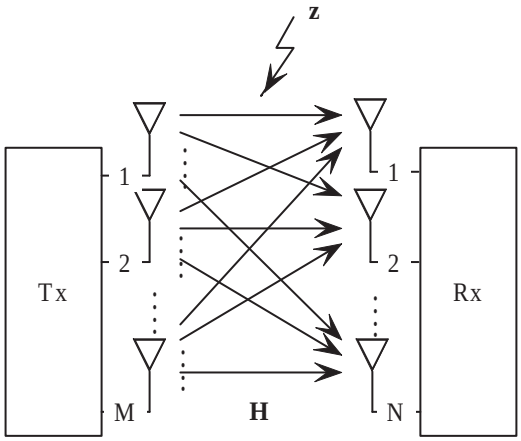


Figure 6.11: MIMO communication system

Gaussian white noise matrix **z**.

$$\mathbf{x} = \mathbf{H}\mathbf{s} + \mathbf{z} \tag{6.1}$$

The complex format of the MIMO system in Equation 6.1 can be replaced with its real value counterpart in Equation 6.2, in which the matrices are augmented with their real and imaginary parts respectively.

$$x = Hs + z \tag{6.2}$$

where $x = [\Re(\mathbf{x})^T \ \Im(\mathbf{x})^T]^T$, $s = [\Re(\mathbf{s})^T \ \Im(\mathbf{s})^T]^T$ and $z = [\Re(\mathbf{z})^T \ \Im(\mathbf{z})^T]^T$. The channel matrix **H** is decomposed into a $m \times n$ real value matrix in Equation 6.3, with $m = 2M$, $n = 2N$.

$$H = \begin{bmatrix} \Re(\mathbf{H}) & -\Im(\mathbf{H}) \\ \Im(\mathbf{H}) & \Re(\mathbf{H}) \end{bmatrix} \tag{6.3}$$

To estimate the transmitted symbol **s** based on the received signal, least squares method can be adopted, in which the receiver forms an estimated transmitted symbol. The optimal solution is the one with the minimum probability of error. The

solutions to the least squares problem include the zero-forcing equalization [105], the minimum mean-square error [106] or the sphere decoding techniques [107]. The first two involve the matrix inversion while the last needs the QR factorization of matrix **H**. To meet the requirement of the MIMO receiver, the hardware specific circuit architectures for the QR decomposition and the matrix inversion is highly desirable, which are the key issues of SDR embedded system platform for flexibility, expandability, scalability, reconfigurability, and reprogrammability.

The dynamical reconfiguration design flow is adopted for the design of the data intensive and timing critical tasks, which are the matrix multiplication and the matrix inversion computation modules in this case. Figure 6.12 and 6.13 show the implementation layout of partial bitstreams for the multiplication of two 4×4 matrices proposed in Chapter 3 and 4×4 matrix inversion in Chapter 5.

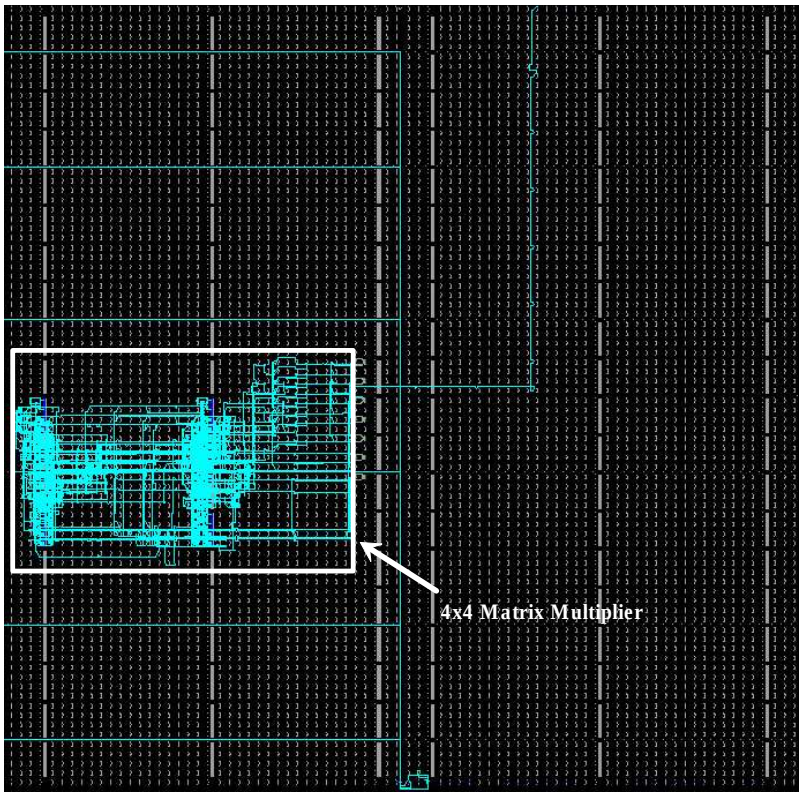


Figure 6.12: Routed FPGA layout of the partial reconfiguration for the multiplication of two 4×4 matrices

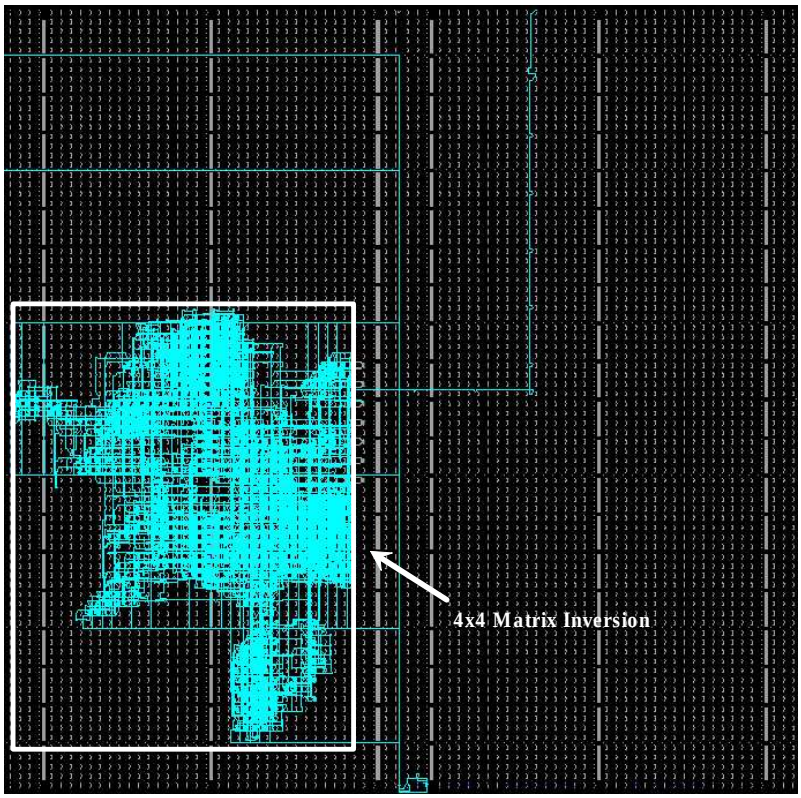


Figure 6.13: Routed FPGA layout of the partial reconfiguration for 4×4 matrix inversion

The dynamically reconfigurable procedure adopted in the proposed partially reconfigurable platform is depicted in Figure 6.14. The partial bitstream is stored in the on-board DDR memory. When the dynamical reconfiguration begins, it is segmented and packed in 32-bit package and read into the FPGA under the control of on-chip SDRAM contrller. The partial bitstream is then transferred through the OPB bus to the BRAM inside HWICAP and written to the ICAP port. The on-chip SDRAM controller and HWICAP are all the peripherals of the Microblaze processor and share the same OPB bus. Thus the read and write operations on partial bitstream are mutually exclusive. The dynamical reconfiguration overhead can be formulated in Equation 6.4.

$$T_d = N_f \times \left(\frac{1}{f_r} + \frac{1}{f_w} \right) \tag{6.4}$$

where T_d is the dynamical reconfiguration time, N_f is the equivalent number of frames of the partial bitstream, f_r and f_w are the rate for read and write operations of bitstream. Additional information such as command and pad frames are taking into account to calculate the equivalent number of partial bitstream frame.

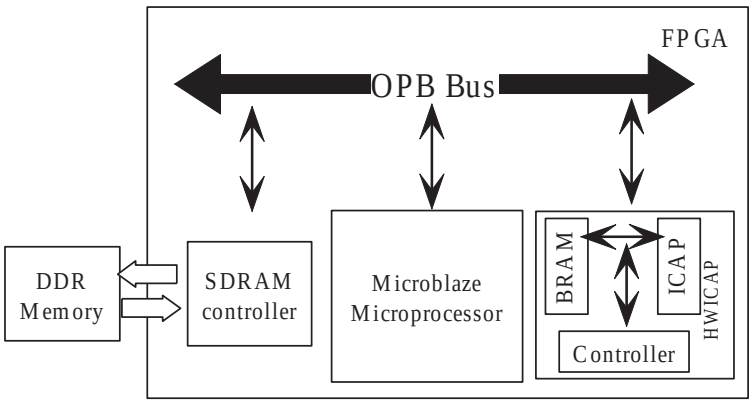


Figure 6.14: Self configuration scheme

The proposed dynamically reconfigurable platform has the ability to change the silicon functions in the run-time at the reconfiguration bandwidth of up to 50 MByte/sec. The size of the partial bitstream for the reconfiguration module is 627,888 bytes. Simulation results show the time to read this partial bitstream into the OPB peripheral parts takes 0.003 second under 50 MHz OPB bus frequency. The reconfiguration time for the change of reconfigurable module is less than 0.29 second while the fixed module initialization takes 0.39 second to complete. The configuration overhead is not a critical factor in SDR as the change of communication standard is not frequently happening. Thus, this reconfigurable computing platform can easily meet the SDR requirements for matrix computation acceleration on different MIMO communication systems. With scalable and parameterisable matrix computation modules in hand, as discussed in Chapters 3 to 5, the reconfiguration of this embedded computing platform allows the SDR system to support multiple communication standards with dynamical features of parameterisability, scalability and reconfigurability.

6.5 Summary

The preliminary platform for the complete implementation and practical experiment for the matrix computation on dynamically reconfigurable FPGA was developed. The proposed platform for performing matrix computation especially matrix inversion and multiplication on parallel FPGA implementation with dynamical and partial reconfiguration gives the possibility to deal efficiently with large matrix problems and different applications for multi-purpose standards support, such as application on SDR, thanks to the partial reconfiguration that allows context swapping when execution is in progress. The chip level self-configuration platform offers process autonomy and even faster configuration speed compared to the in-system programming on board level. By adopting embedded distributed memory block for data caching when context swapping progress occurs in the module base partial reconfiguration design flow, a more robust communication system for dynamical self-reconfiguration platform is also achieved.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis describes the development and implementation of matrix algorithms, such as matrix multiplication and inversion, onto reconfigurable VLSI hardware. The key benefits of prototyping such matrix algorithms on VLSI hardware are the scalability, parameterisability and reconfigurability, which shorten the design cycles to cater for different circumstances and promote the design robustness for reusability and reconfigurability.

Matrix multiplication and matrix inversion are taken as design examples in this thesis, which require large amount of resources and high data throughput. Chapter 3 focuses on the array architecture design for matrix multiplication. The proposed design takes the advantage of the properties of matrix multiplication and re-sequences the data flow, which results in the short computation time. The configurable Block RAM on the reconfigurable hardware makes it possible for parameterisable circuitry to be changed on the fly. Thus, dynamical reconfiguration can be realized without the interruption of the functionality of the whole chip.

A *Bi-z* CORDIC for signal processing algorithm is proposed in Chapter 4. The traditional CORDIC algorithms are relatively slow since they involve progressive iterations. The determination of the rotation angle on the Z-path introduces large latency. Each step of iteration is based on the calculation result of the previous micro-rotation. To speed up the data throughput, the angle decision Z-path in the *Bi-z* CORDIC is redesigned in such a way that the explicit computation of the residual rotation angle is eliminated. The data dependency between the Z-path and the X/Y-path is minimized. The simplified Z-path also reduces the hardware cost, which is the result of the exemption of the circuitry for the angle calculation. The interconnection IO is also reduced in the signal processing modules adopting the *Bi-z* CORDIC engine. The complexity of the routing and circuit interconnections is reduced. The necessity of passing the rotation angle between different CORDIC processors in the traditional CORDIC is eliminated in the *Bi-z* CORDIC, which also increases the overall speed.

Matrix inversion architecture is widely used in many signal processing algorithms, such as MIMO communication systems and adaptive beamforming algorithms in radar processing. Traditional realization is on general purposed processors or DSP processors, and the speed and implementable problem size are limited by the hardware resources. Several special purpose hardware architectures are proposed in this work in Chapter 5, which utilize the hardware architecture properties such as mapping with folding and interlacing on the structural level, and pipelining and parallelism on the dataflow path. The matrix inversion architecture is realized with two steps. The Matrix is first decomposed into an upper triangular matrix in the QR decomposition array, which is realized by the *Bi-z* CORDIC based on the GR algorithm. After the QR decomposition, the triangularized matrix is inverted in the inversion array and multiplied with the QR transformation matrix Q to obtain the matrix inversion.

The introductory of the *Bi-z* CORDIC to solve both the QR decomposition and the triangular matrix inversion problem simplifies the hardware architecture design. The *Bi-z* CORDIC makes it possible to fold together functionally distinguished processors inside the matrix inversion arrays. Time-sharing of the same piece of processor module increases the hardware utilization. The linear mapping method achieves 62.5% processor utilization, while the improved interlaced mapping method improves the utilization up to 100% by interlacing the QR and the inversion arrays and folding them onto the same processor, thanks to the functional flexibility of the *Bi-z* CORDIC. Retiming of the matrix inversion processor is a kind of interlacing operations down to the scale of dataflow circuitry level. By interlacing different matrix inversion updates within the pipelined stage, the maximum data throughput can be achieved.

The flexibility of the module design flow for matrix computation such as multiplication and inversion shows its advantage in the reconfigurable computing platform addressed in Chapter 6. Dynamically reconfigurable hardware chips such as Xilinx FPGA chips allow part of the on-chip hardware to be addressed and reconfigured while leaving other parts of the circuitry intact and functionally operational. This ability allows the time-sharing of the same hardware resource for multi-functional applications, which is equivalent to context switching between different processes in a microprocessor chip. A dynamically reconfigurable computing platform proposed in Chapter 6 demonstrates the efficiency of the chip utilization while still allowing the continuity of the functionalities of the base region modules, which are called 'static' and reside on the chip throughout the processing time.

With the fast ever-growing technologies on digital circuit design, new methodologies must keep up the pace with the trend of fast prototyping, robustness in terms of scalability, modularity and parameterisability on the fly to meet the requirements of data intensive real-time applications, which is the key aspect that this thesis is

addressing.

7.2 Suggestions for Future Works

The thesis casts light on the design methodology with reconfigurable and parameterisable concept for matrix computation VLSI systems. There are more works need to be done further. Some further works are suggested below.

1. The matrix multiplication design can be further optimized and tradeoff can be made if the arithmetic cells such as multiplier and adder are implemented in more efficient designs, for example, redundant [108], carry-free or signed-digit recoding techniques [109]. And comparisons on implementation merits and performance factors can be made.

2. The *Bi-z* CORDIC proposed in Chapter 4 is implemented in a parameterisable fashion with different data precision, word parallel or serial dataflow, etc. Combined with the generic matrix inversion architecture mapping techniques, a more comprehensive design pool can be constructed and more comparisons can be made between the parameterisation at the cell and the array level, and the tradeoff between the hardware cost and the data throughput as well as the power consumption can be further investigated.

3. The development of a more generic matrix inversion architecture may be continued. Chapter 5 proposes two mapping methods for the matrix inversion. Yet for the generic linear mapping methods of the matrix inversion, more techniques can be considered. For example, a rectangular linear array, which takes up of two or more rows with different groups of scheduling lines mapped onto different linear arrays, or sparse linear array in which two or more columns of processing cells may be collapsed into one processor, can be developed. Thus

a complete mapping techniques for the matrix inversion architecture may be formed to cater for different application scenarios. The software interface from which all the control and retiming signals and parameters can be generated automatically and embedded onto the hardware processor IP core for instantiation on real-time processing. The idea of control and parameter generation can be extended to all the generic architecture designs as well.

4. The proposed dynamically reconfigurable computing system is implemented in Xilinx Virtex-II FPGA. The drawback of using the off-the-shelf FPGA devices lies in the lack of customization. The reconfigurable modules must be designed and PARed along the full column of the device, since for Xilinx FPGA the minimum configurable unit is addressed from the top to the bottom of the device. A customized reconfigurable VLSI chip for the matrix computation with dynamical reconfiguration, fast context swapping and random configuration memory access may be worthy of further investigation.

5. Recent advancement of FPGA technology enables more powerful and complex hardware resources to be embedded in the FPGA devices. For example, Xilinx Virtex-4 includes DSP modules and Virtex-5 includes LUTs with more than 4 inputs. These dedicated hardware can be exploited for matrix processing. Further exploration on how to maximize their usages requires further research effort.

Author's Publications

Journal Paper

1. Luo Jianwen and Jong Ching Chuen, "A Scalable Matrix Multiplication Architecture for Reconfigurable Hardware," submitted to *International Journal of High Performance Systems Architecture*.

2. Luo Jianwen and Jong Ching Chuen, "Array Architectures for Matrix Inversion," Submitted to *IEEE Transactions on Computers*.

Conference Paper

1. Luo Jianwen and Jong Ching Chuen, "Partially reconfigurable matrix multiplication for area and time efficiency on FPGAs," 30th Euromicro Symposium on Digital System Design(DSD'04), 31 Aug.- 3 Sept. 2004, Rennes, France.

2. Luo Jianwen and Jong Ching Chuen, "Matrix Inversion on Reconfigurable Hardware using Binary-coded z-path CORDIC," IEEE Asia Pacific Conference on Circuits and Systems(APCCAS'06), 4 - 7 Dec. 2006, Singapore.

3. Yi Wang, Jussipekka Leiwo, Thambipillai Srikanthan and Luo Jianwen "An Efficient Algorithm for DPA-resistent RSA," IEEE Asia Pacific Conference on Circuits and Systems(APCCAS'06), 4 - 7 Dec. 2006, Singapore.

4. Luo Jianwen and Jong Ching Chuen, "A System-on-Chip Dynamically Reconfigurable FPGA Platform for Matrix Inversion," International Symposium on Integrated Circuits (ISIC 2007), 26 - 28 Sept. 2007, Singapore.

5. Wang Chao, Gan Woon-Seng, Jong Ching Chuen and Luo Jianwen, "A Low-Cost 256-Point FFT Processor for Portable Speech and Audio Applications,"

International Symposium on Integrated Circuits (ISIC 2007), 26 - 28 Sept. 2007, Singapore.

Bibliography

[1] S. Y. Kung. *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[2] Anil K. Jain. *Fundamentals of digital image processing*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[3] K. Jack. *Video demystified : a handbook for the digital engineer*. Newnes, 2005.

[4] Simon Haykin. *Adaptive Filter Theory*. Prentice Hall, 4th edition, 2002.

[5] B. Widrow and SD Stearns. *Adaptive Signal Processing*. Prentice Hall, Englewood Cliffs, NJ., 1985.

[6] Dimitris G. Manolakis. *Statistical and adaptive signal processing : spectral estimation, signal modeling, adaptive filtering and array processing*. McGraw-Hill, 2000.

[7] Steven L. Gay. *Acoustic signal processing for telecommunication*. Kluwer Academic, 2000.

[8] Kayvan Najarian. *Biomedical Signal and Image Processing*. CRC, 2006.

[9] R. A. Dunlap. *Experimental physics : modern methods*. Oxford University Press, USA, 1988.

[10] G. Christakos. *Temporal Geographical Information Systems: Advanced Functions for Field-Based Applications*. Springer, 1st edition, 2002.

[11] Alan Jennings. *Matrix Computation for Engineers and Scientists*. John Wiley, Feb. 1977.

[12] Jagdish J. Modi. *Parallel algorithms and matrix computation*. Oxford University Press, USA, 1989.

[13] G. H. Golub and C. F. V. Loan. *Matrix Computations*. John Hopkins University Press, Baltimore and London, 3rd edition, 1996.

[14] K. K. Parhi and T. Nishitani. *Digital Signal Processing for Multimedia Systems*. Marcel Dekker, 1996.

[15] John G. Proakis. *Digital Signal Processing : Principles, Algorithms, and Applications*. Prentice Hall, New Jersey, 3rd edition, 1995.

[16] Hari Krishna Garg, Hari Krishna, and Bal Krishna. *Digital Signal Processing Algorithms: Number Theory, Convolution, Fast Fourier Transforms, and Applications*. CRC, 1998.

[17] Keshab K.Parhi. *Vlsi Digital Signal Processing Systems: Design and Implementation*. John Wiley, 1999.

[18] Stephen W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 2nd edition, Oct. 2003.

[19] Sanjit K. Mitra. *Digital Signal Processing : a Computer-Based Approach*. McGraw-Hill, 2nd edition, 2006.

[20] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice Hall, 2nd edition, 2002.

[21] John Litva. *Digital Beamforming in Wireless Communications*. Artech House, 1st edition, 1996.

[22] J. Li. *Robust adaptive beamforming*. Wiley-Interscience, 2006.

[23] A. Gersho. *Vector quantization and signal compression*. Kluwer Academic Publishers, 1992.

[24] N. Jayant, J. Johnston, and R. Safranek. Signal compression based on models of human perception. In *Proc. of the IEEE*, volume 81 of *10*, pages 1385–1422, Oct 1993.

[25] N. Jayant. *Signal Compression : Coding of Speech, Audio, Text, Image and Video*. World Scientific, 1997.

[26] B. Farhang-Boroujeny. *Adaptive Filters Theory and Applications*. Wiley, 1999.

[27] Simon Haykin. *Least-Mean-Square Adaptive Filters*. Wiley-Interscience, 2003.

[28] J. E. Volder. The CORDIC trigonometric computing technique. *IRE Trans. Electronic Computers*, EC-8(3), Sep 1959.

[29] W. Givens. Computation of plane unitary rotations transforming a general matrix to triangular form. *J. Soc. Indust. Appl. Math*, 6(1), 1958.

[30] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. Morphosys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *Computers, IEEE Transactions on*, 49(5), May 2000.

[31] R.W. Hartenstein, R. Kress, and H. Reinig. A reconfigurable data-driven ALU for Xputers. In *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, 10-13, pages 139 –146, April 1994.

[32] Duncan A. Buell, Jeffrey M. Arnold, and Walter J. Kleinfelder. Splash-2, FPGAs in a Custom Computing Machine. IEEE Computer Society Press, 1996.

[33] E. Sanchez, M. Sipper, J.-O. Haenni, J.-L. Beuchat, A. Stauffer, and A. Perez-Uribe. Static and Dynamic Configurable Systems. *Computers, IEEE Transactions on*, 48(6), Jun 1999.

[34] B. Radunovic. An Overview of advances in Reconfigurable Computing Systems. In *System Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on*, 1999.

[35] Burns, Donlin, Hogg, Singh, and Watt. A dynamic Reconfiguration Run-Time System. In *Proc. 5th Annual IEEE Symposium on Custom Computing Machines*. IEEE Computer Society Press, 1997.

[36] Satnam Singh and Pierre Bellec. Virtual Hardware for Graphics Applications using FPGAs. In *FCCM'94*. IEEE Computer Society, 1994.

[37] K. Compton, Zhiyuan Li, J. Cooley, S. Knol, and S. Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 10(3), Jun 2002.

[38] Zhiyuan Li and Scott Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *Tenth ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, California, USA, Feb 2002.

[39] Andreas Dandalis and Viktor K. Prasanna. Configuration compression for FPGA-based embedded systems. In *Ninth international symposium on Field programmable gate arrays*, pages 173–182, Monterey, California, United States, Feb 2001.

[40] Zhiyuan Li, K. Compton, and Scott Hauck. Configuration Caching Management Techniques for reconfigurable Computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, 2000.

[41] Deepali Deshpande, Arun K. Somani, and Akhilish Tyagi. Configuration caching vs data caching for striped FPGAs. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, 1999.

[42] T. Fujii, K.-i. Furuta, M. Motomura, M. Nomura, M. Mizuno, K.-i. Anjo, K. Wakabayashi, Y. Hirota, Y.-e. Nakazawa, H. Ito, and M. Yamashina. A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture. In *IEEE International Solid-State Circuits Conference(ISSCC), Digest of Technical Papers*, 1999.

[43] S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, , and D.E. Thomas. Managing PipelineReconfigurable FPGAs. In *In 6th International Symposium on Field Programmable Gate Arrays*, pages 55–64, California, USA, 1998.

[44] Xilinx Corp. Application notes xapp284 (v1.1), October 15 2001.

[45] Xilinx Corp. Multiply accumulator (v4.0), ds336, April 28 2005.

[46] Ju-Wook Jang, Seonil Choi, and Viktor K. Prasanna. Area and Time Efficient Implementation of Matrix Multiplication on FPGAs. In *The First IEEE International Conference on Field Programmable Technology (FPT)*, December 2002.

[47] Xilinx Corp. Application notes xapp610 (v1.2), April 24 2002.

[48] J. S. Walther. A unified algorithm for elementary functions. In *AFIPS Spring Joint Conference*, pages 379–385, 1971.

[49] B. Das and S. Banerjee. Unified cordic-based chip to realize dft/dht/dct/dst. *Computers and digital techniques, IEE Proceedings*, 149(4), 2002.

[50] Sungwook Yu and J. Swartzlander. A scaled dct architecture with the cordic algorithm. *IEEE Trans. signal processing*, 50(1), 2002.

[51] J. Ma, K. Parhi, and E. F. Deprettere. Pipelined cordic-based cascade orthogonal iir digital filters. *Circuits and systems II: analog and digital signal processing*, 47(11), 2000.

[52] S. Shiraishi, M. Haseyama, and H. Kitajima. A cost-effective and high-precision architecture for cordic-based adaptive lattice filters. In *IEEE international symposium on circuits and systems, ISCAS 2002*, volume 5, May 2002.

[53] Chang Yong Kong and J. Swartzlander. An analysis of the cordic algorithm for direct digital frequency synthesis. In *IEEE international conference on application-specific systems, architecture and processors*, July 2002.

[54] M. D. Ercegovac and T. Lang. Redundant and on-line cordic: Application to matrix triangularization and svd. *IEEE Trans. Comput.*, 39(6):725–740, Jun 1990.

[55] J.-A. Lee and T. Lang. Constant-factor redundant cordic for angle calculation and rotation. *IEEE Trans. Comput.*, 41(8):1016 – 1025, Aug. 1992.

[56] N. Takagi, T. Asada, and S. Yajima. Redundant cordic methods with a constant scale factor for sine and cosine computation. *IEEE Trans. Comput.*, 40(9):989 – 995, Sept. 1991.

[57] C.-H. Lin and A.-Y. Wu. Mixed-scaling-rotation cordic (msr-cordic) algorithm and architecture for high-performance vector rotational dsp applications. *IEEE Trans. Circuits and Syst. I*, 52(11):2385 – 2396, Nov 2005.

[58] M. Kuhlmann and K. K. Parhi. P-cordic: a precomputation based rotation cordic algorithm. *EURASIP Journal on Applied Signal Processing*, 2002(9):936–943, 2002.

[59] Shaoyun Wang, Vincenzo Piuri, and Jr. Earl E. Swartzlander. Hybrid cordic algorithms. *IEEE Trans. Comput.*, 46(11), 1997.

[60] D. S. Phatak. Double step branching cordic: a new algorithm for fast sine and cosine generation. *IEEE Trans. Comput.*, 47(5), May 1998.

[61] T.-B. Juang, S.-F. Hsiao, and M.-Y. Tsai. Para-cordic: Parallel cordic rotation algorithm. *IEEE Trans. Circuits and Syst. I*, 51(8), Aug. 2004.

[62] J. C. Chih and S. G. Chen. A fast cordic algorithm based on a novel angle recoding scheme. In *IEEE Int. Symp. Circuits Syst.*, volume 4, pages 621–624, 2000.

[63] C.-S. Wu, A.-Y. Wu, and C.-H. Lin. A high-performance/low-latency vector rotational cordic architecture based on extended elementary angle set and trellis-based searching schemes. *IEEE Trans. Circuits Syst. II*, 50(9), Sept. 2003.

[64] J. M. Muller. *Elementary Functions: Algorithms and Implementations*. Cambridge, MA: Birkhauser, 1997.

[65] E. Antelo, T. Lang, and J. D. Bruguera. Very-high radix circular cordic: Vectoring and unified rotation/vectoring. *IEEE Trans. Comput.*, 49(7):727–739, Jul. 2000.

[66] E. Antelo, J. Villalba, D. Bruguera, and E. Zapata. High performance rotation architectures based on the radix cordic algorithm. *IEEE Trans. Comput.*, 46(8):855–870, Aug. 1997.

[67] Xilinx Corp. Product specification cordic (v3.0), ds249, May 2004.

[68] A. Bojańczyk. Complexity of solving linear systems in differenct models of computation. *SIAM Journal on Numerical Analysis*, 21(3), Jun. 1984.

[69] T. Shepherd, J. McWhirter, S. Haykin, J. Litva, and T. Shepherd. Systolic adaptive beamforming. *in Radar Array Processing, Eds. Springer-Verlag*, 1993.

[70] M. Ylinen, A. Burian, and J. Takala. Direct versus iterative methods for fixed-point implementation of matrix inversion. In *Proceedings of the 2004 International Symposium on Circuits and Systems*, volume 3, pages III – 225–8, May 2004.

[71] A. El-Amawy. A systolic architecture for fast dense matrix inversion. *IEEE Trans. Comput.*, 38(3):449 – 455, 1989.

[72] C.K. Singh, S.H. Prasad, and P.T. Balsara. Vlsi architecture for matrix inversion using modified gram-schmidt based qr decomposition. In *20th International Conference on VLSI Design*, pages 836 – 841, Jan 2007.

[73] A. El-Amawy and K.R Dharmarajan. Parallel VLSI algorithm for stable inversion of dense matrices. In *Computers and Digital Techniques, IEE Proceedings*, pages 575 – 580, 1989.

[74] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 74-76, Wellesley, Massachusetts, 3rd edition, 2003.

[75] I. LaRoche and S. Roy. An efficient regular matrix inversion circuit architecture for mimo processing. In *Proceedings of IEEE International Symposium on Circuits and Systems, ISCAS 2006.*, May 2006.

[76] G. W. Stewart. The decompositional approach to matrix computation. *Computing in Science and Engineering, IEEE Journal*, 2(1):50 – 59, 2000.

[77] R. L. Walke. *High Sample-Rate Givens Rotations for Recursive Least Squares*. PhD thesis, Warwick University, 1997.

[78] F. Edman and V. Owall. A scalable pipelined complex valued matrix inversion architecture. In *IEEE International Symposium on Circuits and Systems*, volume 5, pages 4489 – 4492, May 2005.

[79] Luo Jianwen and Jong Ching Chuen. Matrix inversion on reconfigurable hardware using binary-coded z-path cordic. In *IEEE Asia Pacific Conference on Circuits and Systems*, Singapore, Dec 2006.

[80] C. M. Rader. Vlsi systolic arrays for adaptive nulling. *IEEE Signal Processing Magazine*, 13(4):29–49, Jul 1996.

[81] Walke R.L., Smith R.W.M., and Lightbody G. Architectures for adaptive weight calculation on asic and fpga. In *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, volume 2, pages 1375–1380, 1999.

[82] G. Lightbody. *High Performance VLSI Architecture for Recursive Least Squares Adaptive Filtering*. PhD thesis, Queen's University of Belfast, Sept 1999.

[83] F. Edman and V. Owall. Implementation of a scalable matrix inversion architecture for triangular matrices. In *14th IEEE Proceedings on Personal, Indoor and Mobile Radio Communications, (PIMRC 2003)*, volume 3, pages 2558 – 2562, Sept 2003.

[84] Zhaohui Liu, McCanny J.V., Lightbody G., and Walke R. Generic SoC QR array processor for adaptive beamforming. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 50:169–175, April 2003.

[85] Modelsim Mentor Graphics. http://www.model.com/.

[86] G. Lightbody, R. Woods, and R. Walke. Design of a parameterizable silicon intellectual property core for qr-based rls filtering. *IEEE Transactions on Very Large Scale Integration Systems*, 11(4):659 – 678, 2003.

[87] Andres Upegui, Rico Moeckel, Elmar Dittrich, Auke Ijspeert, and Eduardo Sanchez. An fpga dynamically reconfigurable framework for modular robotics. In *Workshop on Dynamical Reconfigurable Systems at the18th International Conference on Architecture of Computing Systems (ARCS 05)*, Innsbruck, Austria, 2005.

[88] Xilinx Corp. user guide, ug208 (v1.1), March 6 2006.

[89] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *Computer*, 30(9):86–93, 1997.

[90] E. Mirsky and A. DeHon. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pages 157 –166, Apr 1996.

[91] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *6th International Workshop on Field-Programmable Logic and Applications*, 1996.

[92] Guangming Lu, H. Singh, Ming-Hau Lee, N. Bagherzadeh, F.J. Kurdahi, E.M.C. Filho, and V. Castro-Alves. The MorphoSys Dynamically Reconfigurable System-On-Chip. In *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, 1999.

[93] H. Diab, E. Abdennour, and F. Kurdahi. FIR Filter Mapping and Performance Analysis on Morphosys. In *The 7th IEEE International Conference on Electronics, Circuits and Systems ICECS*, 2000.

[94] H. Diab, E. Abdennour, and N. Mansour. Spreadsheet Model for Morphosys RC Array. In *The 7th IEEE International Conference on Electronics, Circuits and Systems ICECS*, 2000.

[95] R. Hartenstein. Trends in Reconfigurable Logic and Reconfigurable Computing. In *9th International Conference on Electronics, Circuits and Systems*, volume 2, Sept 2002.

[96] Xilinx Corp. Application notes xapp151 (v1.6), March 24 2003.

[97] Xilinx Corp. Data sheets ds031-1 (v1.9), September 26 2002.

[98] Xilinx Corp. Xapp290(v1.2): Two flows for partial reconfiguration: module based or difference based.

[99] Xilinx Corp. Xilinx development systems reference guide (chapter 4): modular design.

[100] The Dini Group. http://www.dinigroup.com/.

[101] R. Hartenstein. A lightweight approach for embedded reconfiguration of FPGAs. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, pages 801 – 808, Volume 2, 15–18, Sept. 2002.

[102] J. Thorvinger. Dynamic partial reconfiguration of an FPGA for computational hardware support. Master's thesis, Master's theis, Lund Institute of Technology, 2004.

[103] Xilinx Corporation. Xilinx design language version 1.4, 1998.

[104] J. Mitola. The software radio architecture. *IEEE Communications Magazine*, 33(5):26 – 38, May 1995.

[105] H. Bölcskei (Editor), D. Gesbert (Editor), C. B. Papadias (Editor), and A.-J. van der Veen (Editor). *Space-Time Wireless Systems: From Array Processing to MIMO Communications*. Cambridge University Press, 2006.

[106] Ake Bjórck. *Numerical Methods for Least Squares Problems*. SIAM, 1996.

[107] H. Vilkalo. *Sphere Decoding Algorithms for Digital Communications*. PhD thesis, Stanford University, 2003.

[108] Alejandro F. González and Pinaki Mazumder. Redundant arithmetic, algorithms and implementations. *Integration, the VLSI Journal*, 30(1):13 – 53, 2000.

[109] Milo D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

Appendix A

Selected Verilog Codes for the Array Multiplier

A.1 Top Module of 4×4 matrix multiplier

```
‘define DATA_WIDTHR 35:0
‘define DATA_WIDTHW 8:0
‘define ADDR_WIDTH 8:0
‘define msize 8
‘define j 1

module matrixmult4x4(clk_in, rst_in, A_in, read_data_out, read_enable1, write_enable1,
read_enable2, write_enable2, read_enable3, write_enable3, read_enable4, write_enable4,
);

input clk_in, rst_in;
input[7:0] A_in;
input read_enable1, write_enable1;
input read_enable2, write_enable2;
input read_enable3, write_enable3;
input read_enable4, write_enable4;
output[31:0] read_data_out;

wire clk; wire rst = rst_in;
wire[7:0] A = A_in;
wire[31:0] read_data;
wire read_enable1, write_enable1;
wire read_enable2, write_enable2;
wire read_enable3, write_enable3;
wire read_enable4, write_enable4;

assign read_data_out = read_data;
```

```
BUFGP gclk1 (.I(clk_in), .O(clk));

PE_CELL pe1( .clk(clk), .rst_in(rst), .read_enable_in(read_enable1),
.write_enable_in(write_enable1), .A_in(A), .read_data_out(read_data));

PE_CELL pe2(.clk(clk), .rst_in(rst), .read_enable_in(read_enable2),
.write_enable_in(write_enable2), .A_in(A), .read_data_out(read_data));

PE_CELL pe3( .clk(clk), .rst_in(rst), .read_enable_in(read_enable3),
.write_enable_in(write_enable3), .A_in(A), .read_data_out(read_data));

PE_CELL pe4( .clk(clk), .rst_in(rst), .read_enable_in(read_enable4),
.write_enable_in(write_enable4), .A_in(A), .read_data_out(read_data));

endmodule
```

A.2 PE of Matrix Multiplier

```
module PE_CELL(clk, rst_in, read_enable_in, write_enable_in, A_in, read_data_out);

input clk, rst_in, read_enable_in, write_enable_in;
input[7:0] A_in; output[31:0] read_data_out;

reg[7:0] A, A_in_wire, B_out;
wire[31:0] C_out, dob, C_in;
wire clk, rst;
wire read_enable, write_enable;
wire[‘ADDR_WIDTH] gnd_bus;
wire gnd, vcc;

assign read_data_out = (read_enable==1)?dob:’hz;
assign A_in_wire = A_in;
assign rst = rst_in;
assign read_enable = read_enable_in;
assign write_enable = write_enable_in; assign gnd_bus = ‘ADDR_WIDTH’h0;
assign gnd = 0;
assign vcc = 1;

always @ (posedge clk or posedge rst)
begin if (rst)
A<=8’b0;
else
A<= A_in_wire;
end

always @ (posedge clk)
begin
```

```
if (rst == 1'b1)
cntr_j <= `msize - `j;
else
begin
if(cntr_j == `msize-1) cntr_j <= 0;
else cntr_j <= cntr_j + 1;
end
end

always @ ( posedge clk or posedge rst )
begin if (rst)
addr_memC <= 1'b1,7'b0 + `msize - 1 ;
else if(cntr_j == `msize-1) begin
if (addr_memC == 1'b1,7'b0 + `msize - 1)
addr_memC <= 1'b1,7'b0;
else
addr_memC <= addr_memC + 1;
end
end

assign C_in = A * B_out + C_out;
assign C_out = (cntr_j == 1)?0:dob;

RAMB16_S9_S36 BRAM1 (
.ADDRB(1'b0,addr_memC),
.ADDRA(1'b0,cntr_j),
.DIB(C_in),
.DIPB(gnd_bus[3:0]),
.DIA(gnd_bus[8:1]),
.DIPA(gnd_bus[0:0]),
.WEB(write_enable),
.WEA(gnd),
.CLKB(clk),
.CLKA(clk),
.SSRB(gnd),
.SSRA(gnd),
.ENB(vcc),
.ENA(vcc),
.DOB(dob),
.DOPB(),
.DOA(B_out),
.DOPA()
);

endmodule
```

Appendix B

Selected Verilog Codes for the *Bi-z* CORDIC

B.1 Top Module of Parallel Pipelined *Bi-z* CORDIC

```
‘define WDLENGTH 16
‘define MATRIXSIZE 76


module para_cordic_pipelined(clk,mctr,xin,yin,xout,yout);
input clk, mctr;
input [‘WDLENGTH-1:0] xin, yin;
output[‘WDLENGTH-1:0] xout, yout;

wire[‘WDLENGTH*‘WDLENGTH-1:0] xin_wire;
wire[‘WDLENGTH*‘WDLENGTH-1:0] yin_wire;
wire[‘WDLENGTH*‘WDLENGTH-1:0] xout_wire;
wire[‘WDLENGTH*‘WDLENGTH-1:0] yout_wire;


generate
genvar i;
for (i = 1; i <= ‘WDLENGTH; i = i + 1)
begin : cordic_pipelined // block label
cordic_pipelined_pe PE_inst (
.clk(clk),
.mctr(mctr),
.xin(xin_wire[(i*‘WDLENGTH-1)-:‘WDLENGTH]),
.yin(yin_wire[(i*‘WDLENGTH-1)-:‘WDLENGTH]),
.xout(xout_wire[(i*‘WDLENGTH-1)-:‘WDLENGTH]),
.yout(yout_wire[(i*‘WDLENGTH-1)-:‘WDLENGTH])
```

```
);
defparam cordic_pipelined[i].PE_inst.stage = i-1;
end
endgenerate

assign xin_wire[‘WDLENGTH-1:0]=xin;
assign yin_wire[‘WDLENGTH-1:0]=yin;
assign xout=xout_wire[(‘WDLENGTH*‘WDLENGTH-1)-:‘WDLENGTH];
assign yout=yout_wire[(‘WDLENGTH*‘WDLENGTH-1)-:‘WDLENGTH];
generate
genvar j;
for (j = 1; j <= ‘WDLENGTH-1; j = j + 1)
begin : cordic_interconnection
assign xin_wire[((j+1)*‘WDLENGTH-1)-:‘WDLENGTH]=xout_wire[(j*‘WDLENGTH-
1)-:‘WDLENGTH];
assign yin_wire[((j+1)*‘WDLENGTH-1)-:‘WDLENGTH]=yout_wire[(j*‘WDLENGTH-
1)-:‘WDLENGTH];
end
endgenerate


endmodule
```

B.2 PE of Parallel *Bi-z* CORDIC

```
module cordic_pipelined_pe(clk,mctr,xin,yin,xout,yout);
input clk, mctr;
input [‘WDLENGTH-1:0] xin, yin;
output[‘WDLENGTH-1:0] xout, yout;

reg[‘WDLENGTH-1:0] xout, yout;
reg sign_bit;

wire[‘WDLENGTH-1:0] xin, yin;
wire[‘WDLENGTH-1:0] xinshift, yinshift;

parameter stage = 0;
parameter type = 0;

always @ (posedge clk) begin
if(mctr)
sign_bit <= xin[‘WDLENGTH-1]∧yin[‘WDLENGTH-1];
end

assign signbit=mctr?xin[‘WDLENGTH-1]∧yin[‘WDLENGTH-1]:sign_bit;
```

```
always @ (posedge clk) begin
if(yin[‘WDLENGTH-1])
xout <= type? xin : (sign_bit? xin-yinshift : xin+yinshift);
else
xout <= type? xin : (sign_bit? xin-(yin>>stage) : xin+(yin>>stage));
if(xin[‘WDLENGTH-1])
yout <= sign_bit? yin+xinshift : yin-xinshift;
else
yout <= sign_bit? yin+(xin>>stage) : yin-(xin>>stage);
end

assign xinshift=‘WDLENGTH1’b1,xin>>stage;
assign yinshift=‘WDLENGTH1’b1,yin>>stage;

endmodule
```

Appendix C

Selected Verilog Codes for the Matrix Inversion

C.1 Top Module of Matrix Inversion

```
‘define WDLENGTH 16
‘define MATRIXSIZE 76
‘define KCOEFF 16’b0010011011011101
‘define ODD (‘MATRIXSIZE%2)
‘define ARRAYSIZE (‘MATRIXSIZE/2)

module MI(clk,rst,dclk,s,yin,mctr,yout);
input[(‘ARRAYSIZE+1)*‘WDLENGTH-1:0] yin;
input[‘ARRAYSIZE*3+1:0] s;
input[‘ARRAYSIZE:0] dclk;
input[‘ARRAYSIZE:0] mctr;
input clk,rst;
output[(‘ARRAYSIZE+1)*‘WDLENGTH-1:0] yout;

wire[(‘ARRAYSIZE*3+2)*(‘WDLENGTH+1)-1:0] mux_x,mux_y,mux_out;
wire[‘ARRAYSIZE*(‘WDLENGTH+1)-1:0] xincomb;
wire[(‘ARRAYSIZE+1)*(‘WDLENGTH+1)-1:0] xoutcomb;
wire[(‘ARRAYSIZE+1)*(‘WDLENGTH+1)-1:0] yincomb;
wire[‘ARRAYSIZE*(‘WDLENGTH+1)-1:0] youtcomb;

// Boundary Cell
bizcordic2 PE0(
.clk(clk),
.rst(rst),
.dclk(dclk[0]),
.xincomb(yincomb[‘WDLENGTH:0]),
```

```
.youtcomb(xoutcomb[‘WDLENGTH:0]));

// Internal Cell
generate
genvar i;
for (i = 1; i <= ‘ARRAYSIZE; i = i + 1)
begin : PE
bizcordic1 PE_inst(
.clk(clk),
.rst(rst),
.dclk(dclk[i]),
.xincomb(xincomb[(i*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]),
.yincomb(yincomb[((i+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]),
.xoutcomb(xoutcomb[((i+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]),
.youtcomb(youtcomb[(i*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]));
end
endgenerate

// Generate all MUXs
generate
genvar j;
for (j = 0; j <= ‘ARRAYSIZE*3+1; j = j + 1)
begin : MUX
mux mux_inst(
.x(mux_x[((j+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]),
.y(mux_y[((j+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]),
.s(s[j]),
.out(mux_out[((j+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]));
end
endgenerate

//*** MUX interconnection ***//
// external input to MUXs
generate
genvar k;
for (k = 0; k <= ‘ARRAYSIZE; k = k + 1)
begin : MUX_E
assign mux_x[((k+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]
={mctr[k],yin[((k+1)*‘WDLENGTH-1)-:‘WDLENGTH]};
end
endgenerate

// y input of the top MUXs, connect to the output of bottom MUXs
generate
genvar l;
for (l = 0; l <= ‘ARRAYSIZE; l = l + 1)
```

```
begin : MUX_YIN
assign mux_y[((l+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]=mux_out[((l+1)*
(‘WDLENGTH+1)+(‘ARRAYSIZE+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)];
end
endgenerate

// yin input of cells
generate
genvar m;
for (m = 0; m <= ‘ARRAYSIZE; m = m + 1)
begin : MUX_YIN_MUX
assign yincomb[((m+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]=
mux_out[((m+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)];
end
endgenerate

// Bottom MUXs interconnection
generate
genvar n;
for (n = 2; n <= ‘ARRAYSIZE-1; n = n + 1)
begin : MUX_YOUT_MUXIN
assign mux_x[((n+1)*(‘WDLENGTH+1)+(‘ARRAYSIZE+1)*(‘WDLENGTH+1)-
1)-:(‘WDLENGTH+1)]=youtcomb[((n-1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)];
assign mux_y[((n+1)*(‘WDLENGTH+1)+(‘ARRAYSIZE+1)*(‘WDLENGTH+1)-
1)-:(‘WDLENGTH+1)]=youtcomb[((n+1)*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)];
end
endgenerate

assign mux_x[(1*(‘WDLENGTH+1)+(‘ARRAYSIZE+1)*(‘WDLENGTH+1)-1)
-:(‘WDLENGTH+1)]=xoutcomb[(2*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)];
assign mux_y[(1*(‘WDLENGTH+1)+(‘ARRAYSIZE+1)*(‘WDLENGTH+1)-1)
-:(‘WDLENGTH+1)]=youtcomb[(1*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)];
assign mux_x[(2*(‘WDLENGTH+1)+(‘ARRAYSIZE+1)*(‘WDLENGTH+1)-1)
-:(‘WDLENGTH+1)]=xoutcomb[(1*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)];
assign mux_y[(2*(‘WDLENGTH+1)+(‘ARRAYSIZE+1)*(‘WDLENGTH+1)-1)
-:(‘WDLENGTH+1)]=youtcomb[(2*(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)];
assign mux_x[((‘ARRAYSIZE+1)*(‘WDLENGTH+1)+(‘ARRAYSIZE+1)
(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]=youtcomb[(‘ARRAYSIZE*
(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)];
assign mux_y[((‘ARRAYSIZE+1)*(‘WDLENGTH+1)+(‘ARRAYSIZE+1)*
(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)]=youtcomb[((‘ARRAYSIZE-1)*
(‘WDLENGTH+1)-1)-:(‘WDLENGTH+1)];

// Middle MUXs for xin of cells
generate
genvar p;
```

```
for (p = 1; p <= `ARRAYSIZE-1; p = p + 1)
begin : MUX_XIN
assign mux_x[(p*(`WDLENGTH+1)+(`ARRAYSIZE*2+2)*(`WDLENGTH+1)-1)-
:(`WDLENGTH+1)]=xoutcomb[(p*(`WDLENGTH+1)-1)-:(`WDLENGTH+1)];
assign mux_y[(p*(`WDLENGTH+1)+(`ARRAYSIZE*2+2)*(`WDLENGTH+1)-1)-
:(`WDLENGTH+1)]=xoutcomb[((p+2)*(`WDLENGTH+1)-1)-:(`WDLENGTH+1)];
end
endgenerate

assign mux_x[(`ARRAYSIZE*(`WDLENGTH+1)+(`ARRAYSIZE*2+2)*
(`WDLENGTH+1)-1)-:(`WDLENGTH+1)]=xoutcomb[(`ARRAYSIZE*
(`WDLENGTH+1)-1)-:(`WDLENGTH +1)];
assign mux_y[(`ARRAYSIZE*(`WDLENGTH+1)+(`ARRAYSIZE*2+2)*
(`WDLENGTH+1)-1)-:(`WDLENGTH+1)]=xoutcomb[((`ARRAYSIZE+1)*
(`WDLENGTH+1)-1)-:(`WDLENGTH +1)];

// xin input of cells
generate
genvar q;
for (q = 1; q <= `ARRAYSIZE; q = q + 1)
begin : MUX_XIN_MUX
assign xincomb[(q*(`WDLENGTH+1)-1)-:(`WDLENGTH+1)]=mux_out[(q*
(`WDLENGTH
+1)+(`ARRAYSIZE*2+2)*(`WDLENGTH+1)-1)-:(`WDLENGTH+1)];
end
endgenerate

// yout output assignment
assign yout[`WDLENGTH-1:0]=xoutcomb[`WDLENGTH-1:0];

generate
genvar r;
for (r = 1; r <= `ARRAYSIZE; r = r + 1)
begin : YOUT
assign yout[((r+1)*`WDLENGTH-1)-:`WDLENGTH]=youtcomb[(r*(`WDLENGTH+1)-
2)-:`WDLENGTH];
end
endgenerate

endmodule
```

C.2 BC of Matrix Inversion

```
module bizcordic2(clk,rst,dclk,xincomb,youtcomb);
input clk, dclk, rst;
```

```
input[‘WDLENGTH:0] xincomb;
output[‘WDLENGTH:0] youtcomb;

integer i,j;

reg[‘WDLENGTH/2:0] stage,stage_1,stage_2;
reg[‘WDLENGTH-1:0] xintemp, yintemp;
reg[‘WDLENGTH-1:0] shift_bit,xinbit,yinbit;
reg xin_sign;
reg type;
reg[‘MATRIXSIZE:0] tcounter;
reg[‘WDLENGTH-1:0] xinmux, yinmux;
reg[‘WDLENGTH-1:0] xinreg, yinreg;
reg[‘MATRIXSIZE*2*‘WDLENGTH-1:0] delt;
reg[‘WDLENGTH+6:0] dclk_shift,mctr_shift;
reg[‘WDLENGTH-1:0] yout;
reg[‘MATRIXSIZE*‘WDLENGTH-1:0] deltadd;
reg[‘WDLENGTH-2:0] xinshift_1;

wire[‘WDLENGTH-1:0] xin;
wire[‘WDLENGTH-1:0] xinopt, yinopt;
wire[‘WDLENGTH-1:0] xinoptshift;
wire sign_bit;
wire[‘WDLENGTH-1:0] xinshift_2;
wire mctr, mctrout;

parameter number=1;

assign xin = xincomb[‘WDLENGTH-1:0];
assign mctr = xincomb[‘WDLENGTH];

always @ (posedge clk or posedge rst)
begin
if(rst)
begin
xintemp <= 0;
yintemp <= 0;
end
else
begin
xintemp <= xin;
yintemp <= mctr? (‘WDLENGTH’b1<<(‘WDLENGTH-2)) : ‘WDLENGTH’b0;
end
end

always @ (posedge dclk or posedge rst)
begin
```

```
if(rst)
xin_sign <= 0;
else
xin_sign <= xin[‘WDLENGTH-1];
end

always @ (posedge dclk or posedge rst)
begin
if(rst)
xinbit <= 0;
else
for(i=0;i<‘WDLENGTH;i=i+1)
begin
if(xinbit==0&&xin[i]==~xin[‘WDLENGTH-1])
xinbit <= i;
end
end

always @ (posedge dclk or posedge rst)
begin
if(rst)
yinbit <= 0;
else
yinbit <= ‘WDLENGTH-2;
end

always @ (posedge clk or posedge rst)
begin
if(rst)
shift_bit <= 0;
else if(mctr)
shift_bit[tcounter-1] <= ((yinbit-xinbit-2)>0)?(yinbit-xinbit-2):0;
end

always @ (posedge clk)
begin
xinshift_1 = xintemp[‘WDLENGTH-2:0]<<shift_bit[tcounter-1];
end

assign xinshift_2 = {xin_sign,xinshift_1};

always @ (posedge clk)
begin
xinmux <= xinshift_2;
yinmux <= yintemp;
end
```

```
always @ (posedge dclk or posedge rst)
begin
if(rst)
tcounter <= 0;
else if(tcounter == `MATRIXSIZE*2)
tcounter <= 1;
else
tcounter <= tcounter + 1;
end

always @ (posedge clk or posedge rst)
begin
if(rst)
type <= 0;
else if(`ODD==0)
begin
if(tcounter%2==1)
type <= 0;
else
type <= 1;
end
else
begin
if(tcounter%2==0)
type <= 0;
else
type <= 1;
end
end

always @ (posedge clk)
begin
dclk_shift<={dclk_shift[`WDLENGTH+5:0],dclk};
end

always @ (posedge clk)
begin
mctr_shift<={mctr_shift[`WDLENGTH+5:0],mctr};
end

always @ (posedge clk)
begin
if((dclk_shift[3] | dclk_shift[3+`WDLENGTH/2]))
stage <= 0;
else if(stage==`WDLENGTH-1)
stage <= `WDLENGTH-1;
```

```
else
stage <= stage+1;
end

always @ (posedge clk) begin
stage_1 <= stage;
stage_2 <= stage_1;
end

always @ (posedge clk)
begin
if(mctr)
begin
delt[deltadd] <= xinopt[‘WDLENGTH-1]∧yinopt[‘WDLENGTH-1];
end
end

always @ (posedge clk)
begin
if(rst)
deltadd <= 0;
else
deltadd <= (tcounter-1)*‘WDLENGTH+stage;
end

always @ (posedge clk or posedge rst)
begin
if(rst)
begin
xinreg <= 0;
yinreg <= 0;
end
else
begin
if(yinopt[‘WDLENGTH-1])
xinreg <= xinopt;
else
xinreg <= xinopt;
if(xinopt[‘WDLENGTH-1])
yinreg <= sign_bit? yinopt+xinoptshift : yinopt-xinoptshift;
else
yinreg <= sign_bit? yinopt+(xinopt>>stage_2) : yinopt-(xinopt>>stage_2);
end
end

assign sign_bit = delt[deltadd];
assign xinopt = (stage_2==0)? xinmux : xinreg;
```

```
assign yinopt = (stage_2==0)? yinmux : yinreg;
assign xinoptshift={{`WDLENGTH{1'b1}},xinopt}>>stage_2;

always @ (posedge dclk_shift[`WDLENGTH+6] or posedge rst)
begin
if(rst)
yout <= 0;
else if(type == 1'b1)
yout <= mctr?0:yinreg;
else if(type == 1'b0)
yout <= xintemp;
end

assign mctrout = mctr_shift[`WDLENGTH+6];
assign youtcomb = {mctrout,yout};

endmodule
```

C.3 IC of Matrix Inversion

```
module bizcordic1(clk,rst,dclk,xincomb,yincomb,xoutcomb,youtcomb);
input clk, dclk, rst;
input [`WDLENGTH:0] xincomb, yincomb;
output[`WDLENGTH:0] xoutcomb, youtcomb;

integer i,j; integer xinbit,yinbit;

reg[`WDLENGTH/2:0] stage,stage_1,stage_2;
reg[`WDLENGTH*2-1:0] xintemp, yintemp;
reg[`WDLENGTH-1:0] shift_bit;
reg xin_sign, yin_sign;
reg[1:0] type;
reg[`MATRIXSIZE:0] tcounter;
reg[`WDLENGTH-1:0] xinmux, yinmux;
reg[`WDLENGTH-1:0] xinreg, yinreg;
reg [`MATRIXSIZE*2*`WDLENGTH-1:0] delt;
reg[`WDLENGTH+6:0] dclk_shift,xmctr_shift,ymctr_shift;
reg[`WDLENGTH-1:0] xout, yout;
reg[`MATRIXSIZE*`WDLENGTH-1:0] deltadd;

wire[`WDLENGTH-1:0] xin, yin;
wire[`WDLENGTH-1:0] xinopt, yinopt;
wire[`WDLENGTH-1:0] xinoptshift, yinoptshift;
wire sign_bit;
reg[`WDLENGTH-2:0] xinshift_1;
```

```
wire[‘WDLENGTH-1:0] xinshift_2;
wire xmctr,ymctr,xmctrout,ymctrout;

parameter number=1;

assign xin = xincomb[‘WDLENGTH-1:0];
assign xmctr = xincomb[‘WDLENGTH];
assign yin = yincomb[‘WDLENGTH-1:0];
assign ymctr = yincomb[‘WDLENGTH];

always @ (posedge clk or posedge rst)
begin
if(rst)
begin
xintemp <= 0;
yintemp <= 0;
end
else if(type==2’b00)
begin
xintemp <= xin*‘KCOEFF;
yintemp <= yin*‘KCOEFF;
end
else if(type==2’b11)
begin
if(xin[‘WDLENGTH-1])
xintemp <= -xin;
else
xintemp <= xin;
if(yin[‘WDLENGTH-1])
yintemp <= yin;
else
yintemp <= yin;
end
else
begin
xintemp <= xin;
yintemp <= yin;
end
end

always @ (posedge dclk or posedge rst)
begin
if(rst)
begin
xin_sign <= 0;
yin_sign <= 0;
```

```
end
else
begin
xin_sign <= xin[‘WDLENGTH-1];
yin_sign <= yin[‘WDLENGTH-1];
end
end

always @ (posedge dclk or posedge rst)
begin
if(rst)
xinbit <= 0;
else
for(i=0;i<‘WDLENGTH;i=i+1)
begin
if(xinbit==0&&xin[i]==~xin[‘WDLENGTH-1])
xinbit <= i;
end
end

always @ (posedge dclk or posedge rst)
begin
if(rst)
yinbit <= 0;
else
for(j=0;j<‘WDLENGTH;j=j+1)
begin
if(yinbit==0&&yin[j]==~yin[‘WDLENGTH-1])
yinbit <= j;
end
end

always @ (posedge clk or posedge rst)
begin
if(rst)
shift_bit <= 0;
else if(xmctr)
shift_bit[tcounter-1] <= ((yinbit-xinbit-2)>0)?(yinbit-xinbit-2):0;
end

always @ (posedge clk)
begin
xinshift_1<=xin_sign?-(xintemp[‘WDLENGTH-2:0]<<shift_bit[tcounter-1]):(xintemp
[‘WDLENGTH-2:0]<<shift_bit[tcounter-1]);
end

assign xinshift_2 = {xin_sign,xinshift_1};
```

```
always @ (posedge clk)
begin
xinmux <= ~type[0]?xintemp[`WDLENGTH*2-1:`WDLENGTH]:xinshift_2;
yinmux <= ~type[0]?yintemp[`WDLENGTH*2-1:`WDLENGTH]:yintemp[`WDLENGTH-
1:0];
end

always @ (posedge dclk or posedge rst)
begin
if(rst)
tcounter <= 0;
else if(tcounter == `MATRIXSIZE*2)
tcounter <= 1;
else
tcounter <= tcounter + 1;
end

always @ (posedge clk or posedge rst)
begin
if(rst)
type <= 0;
else if(`ODD==0)
begin
if(tcounter==`MATRIXSIZE*2-number+1 || tcounter==`MATRIXSIZE+number)
type <= 2'b01;
else if(tcounter%2==(number+1)%2 && tcounter<`MATRIXSIZE*2-number)
type <= 2'b00;
else
type <= 2'b11;
end
else
begin
if(tcounter%2==number%2 && tcounter<`MATRIXSIZE*2-number+1)
type <= 2'b00;
else
type <= 2'b11;
end
end

always @ (posedge clk)
begin
dclk_shift<={dclk_shift[`WDLENGTH+5:0],dclk};
end

always @ (posedge clk)
begin
```

```
xmctr_shift<={xmctr_shift[‘WDLENGTH+5:0],xmctr};
end

always @ (posedge clk)
begin
ymctr_shift<={ymctr_shift[‘WDLENGTH+5:0],ymctr};
end

always @ (posedge clk)
begin
if(∼(dclk_shift[3] | dclk_shift[3+‘WDLENGTH/2]))
stage <= 0;
else if(stage==‘WDLENGTH-1)
stage <= ‘WDLENGTH-1;
else
stage <= stage+1;
end

always @ (posedge clk)
begin
stage_1 <= stage;
stage_2 <= stage_1;
end

always @ (posedge clk)
begin
if(xmctr)
begin
delt[deltadd] <= xinopt[‘WDLENGTH-1]∧yinopt[‘WDLENGTH-1];
end
end

always @ (posedge clk)
begin
if(rst)
deltadd <= 0;
else
deltadd <= (tcounter-1)*‘WDLENGTH+stage;
end

always @ (posedge clk or posedge rst)
begin
if(rst)
begin
xinreg <= 0;
yinreg <= 0;
end
```

```
else
begin
if(yinopt[`WDLENGTH-1])
xinreg<=type[0]?xinopt:(sign_bit?xinopt-yinoptshift:xinopt+yinoptshift);
else
xinreg<=type[0]?xinopt:(sign_bit?xinopt-(yinopt>>stage_2):xinopt+(yinopt>>stage_2));
if(xinopt[`WDLENGTH-1])
yinreg<=sign_bit?yinopt+xinoptshift:yinopt-xinoptshift;
else
yinreg<=sign_bit?yinopt+(xinopt>>stage_2):yinopt-(xinopt>>stage_2);
end
end

assign sign_bit = delt[deltadd];
assign xinopt = (stage_2==0)? xinmux : xinreg;
assign yinopt = (stage_2==0)? yinmux : yinreg;
assign xinoptshift={{`WDLENGTH{1'b1}},xinopt}>>stage_2;
assign yinoptshift={{`WDLENGTH{1'b1}},yinopt}>>stage_2;

always @ (posedge dclk_shift[`WDLENGTH+6] or posedge rst)
begin
if(rst)
begin
xout <= 0;
yout <= 0;
end
else if(type == 2'b00)
begin
xout <= xinreg;
yout <= xmctr?0:yinreg;
end
else if(type == 2'b01)
begin
xout <= xinreg;
yout <= xmctr?0:yinreg;
end
else if(type == 2'b10)
begin
xout <= xintemp;
yout <= yintemp;
end
end

assign xmctrout = xmctr_shift[`WDLENGTH+6];
assign ymctrout = ymctr_shift[`WDLENGTH+6];
assign xoutcomb = {xmctrout,xout};
```

```
assign youtcomb = {ymctrout,yout};
endmodule
```