

Reconfigurable Hardware Implementation and Analysis of Mesh Routing for the Matrix
Step of the Number Field Sieve Factorization

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science at George Mason University

By

Sashisu M. Bajracharya
Bachelor of Science
George Mason University, 2002

Director: Dr. Kris M. Gaj, Associate Professor
Department of Electrical and Computer Engineering

Fall Semester 2004
George Mason University
Fairfax, VA

Copyright 2004. Sashisu M. Bajracharya
All Rights Reserved

ACKNOWLEDGEMENTS

I would like to thank Dr. Kris Gaj for the support and the development of the topic. I would like to thank SRC Computers Inc. for their help in implementing my designs using the SRC-6e reconfigurable computer. I would like to thank Deapesh Misra for his support in the generation of test vectors and performance comparison with software. I would like to thank the Committee for taking the time to work with me on this endeavor.

TABLE OF CONTENTS

| | Page |
|---|------|
| Abstract..... | viii |
| 1. Introduction..... | 1 |
| 2. Number Field Sieve Factorization and the Matrix Step..... | 5 |
| 2.1 NFS..... | 5 |
| 2.2 Matrix Step in NFS..... | 8 |
| 3. Mesh Circuit for the Matrix Step..... | 10 |
| 3.1 Bernstein's Mesh Based Approach..... | 11 |
| 3.2 Mesh Sorting..... | 11 |
| 3.3 Mesh Routing..... | 11 |
| 3.4 Mesh Sorting vs. Mesh Routing..... | 12 |
| 4. Distributed Matrix Computation..... | 13 |
| 5. Mesh Routing Design..... | 15 |
| 5.1 Sparse Matrix and Vector..... | 17 |
| 5.2 Mesh of Cells..... | 18 |
| 5.3 Clockwise Transposition Routing Algorithm..... | 20 |
| 5.4 Improved Mesh Routing Algorithm..... | 21 |
| 6. FPGA Hardware Platform..... | 22 |
| 7. Hardware Architectures of Mesh Routing Designs..... | 25 |
| 7.1 Hardware Architecture of Basic Mesh Routing Design..... | 25 |
| 7.1.1 Loading and Unloading..... | 25 |
| 7.1.2 Routing Operation..... | 28 |
| 7.1.3 Compare-Exchange Operation..... | 30 |
| 7.1.4 Comparator in Cell..... | 32 |
| 7.1.5 Basic Cell Architecture..... | 35 |
| 7.2 Hardware Architecture of Improved Mesh Routing Design..... | 38 |
| 7.2.1 Loading and Unloading..... | 38 |
| 7.2.2 Routing Operation..... | 39 |
| 7.2.3 Improved Cell Architecture..... | 39 |
| 8. Results and Analysis | 43 |
| 8.1 Design Process, Tools and Testing..... | 43 |
| 8.2 Results for Basic Mesh Routing Design and Analysis..... | 44 |
| 8.2.1 Area and Latency..... | 44 |
| 8.2.2 512-bit and 1024-bit Performance Estimation and Cost..... | 47 |
| 8.3 Results for Improved Mesh Routing Design and Analysis..... | 53 |
| 8.3.1 Area and Latency..... | 53 |

| | |
|--|----|
| 8.3.2 512-bit and 1024-bit Performance Estimation and Cost..... | 54 |
| 8.4 Comparison of Basic Mesh Routing and Improved Mesh Routing Implementations..... | 57 |
| 9. SRC Computing Platform..... | 61 |
| 9.1 Hardware Architecture..... | 61 |
| 9.2 Programming Model..... | 62 |
| 9.3 SRC Restrictions..... | 65 |
| 10. Implementation on SRC..... | 66 |
| 10.1 Design Scheme..... | 66 |
| 10.2 SRC-Mesh Design..... | 68 |
| 10.3 SRC-Cells Design..... | 70 |
| 11. Results on SRC and Analysis..... | 73 |
| 11.1 Design Process, Tools and Testing..... | 73 |
| 11.2 Results for Basic Mesh Routing Design..... | 73 |
| 11.3 Results for Improved Mesh Routing Design..... | 80 |
| 12. Comparison of the SRC Designs versus Standalone FPGA Designs..... | 84 |
| 13. Conclusions..... | 85 |
| List of References..... | 88 |

LIST OF TABLES

| Table | Page |
|--|------|
| 1. Synthesis results for the Basic Mesh Routing design in Virtex II FPGA, XC2V8000..... | 45 |
| 2. Time comparison for sparse matrix multiplication of size 144x144 in optimized software and in Virtex II FPGA | 47 |
| 3. Time estimates for the Matrix step of factoring 512 bit numbers with one Virtex II chip, XC2V8000 and multiple Virtex II chips in Basic Mesh Routing..... | 49 |
| 4. Time estimates for the Matrix step of factoring 1024 bit numbers with one Virtex II chip and multiple Virtex II chips in Basic Mesh Routing..... | 52 |
| 5. Synthesis results for the Improved Mesh Routing design in Virtex II FPGA..... | 54 |
| 6. Time estimates for the Matrix step of factoring 512 bit numbers with one Virtex II chip and multiple Virtex II chips in Improved Mesh Routing..... | 55 |
| 7. Time estimates for the Matrix step of factoring 1024 bit numbers with one Virtex II chip and multiple Virtex II chips in Improved Mesh Routing..... | 56 |
| 8. Area for SRC Basic Mesh Routing designs..... | 74 |
| 9. Performance for SRC Basic Mesh Routing designs..... | 75 |
| 10. Comparison of SRC-mesh and SRC-Cells for the same mesh parameters..... | 79 |
| 11. Area resources for SRC Improved Mesh Routing designs..... | 81 |
| 12. Performance for SRC Improved Mesh Routing designs..... | 82 |

LIST OF FIGURES

| Figure | Page |
|---|------|
| 1. Distributing computation of a large matrix using sub-matrix computations..... | 14 |
| 2. Matrix-by-vector multiplication operation..... | 16 |
| 3. Mesh corresponding to the sparse matrix A..... | 18 |
| 4. Routing of packets to the cell in the mesh..... | 19 |
| 5. Virtex II FPGA Architecture..... | 23 |
| 6. CLB slice structure..... | 23 |
| 7. Basic loading and unloading..... | 26 |
| 8. Parallel loading and unloading on multiple rows..... | 28 |
| 9. Four iterations of compare-exchange operation..... | 29 |
| 10. Compare-exchange direction for each cell..... | 30 |
| 11. Compare-exchange cases..... | 31 |
| 12. Comparator logic..... | 33 |
| 13. Detailed architecture of each Basic Cell..... | 36 |
| 14. Detailed architecture of each Improved Cell..... | 41 |
| 15. Speedup for multiple Virtex II chips connected in a mesh | 51 |
| 16. Comparison of Basic and Improved Mesh Routing for 512-bit factorization..... | 57 |
| 17. Comparison of Basic and Improved Mesh Routing for 1024-bit factorization..... | 58 |
| 18. Speedup of Improved to Basic Mesh Routing vs. Number of Virtex II FPGAs..... | 60 |
| 19. Hardware architecture of SRC-6e..... | 61 |
| 20. Compilation process of SRC-6e..... | 63 |
| 21. Programming model of SRC-6e..... | 64 |
| 22. SRC-Mesh Design..... | 67 |
| 23. SRC-Cells Design..... | 68 |
| 24. Architecture of a basic cell in the SRC-Mesh design..... | 69 |
| 25. Cell instantiation with circulation of output to input..... | 70 |
| 26. Cell structure of the SRC-Cells design..... | 71 |
| 27. Comparison of performance of different mesh sizes and K for a 512-bit matrix..... | 78 |
| 28. Comparison of area of the SRC-Mesh and the SRC-Cells of 10x10..... | 80 |

ABSTRACT

RECONFIGURABLE HARDWARE IMPLEMENTATION AND ANALYSIS OF MESH
ROUTING FOR THE MATRIX STEP OF NUMBER FIELD SIEVE FACTORIZATION

Sashisu M. Bajracharya, M.S.

George Mason University, 2004

Thesis Director: Dr. Kris Gaj

Factorization of large numbers has been a constant source of interest as it is the basis of security for the well-known RSA cryptosystem. The fastest known algorithm for factoring large numbers is the Number Field Sieve (NFS). The most time consuming phases of NFS are Sieving and Matrix Step. This thesis is concentrated on the Matrix Step, and an efficient way of implementing this step in reconfigurable hardware is proposed. This solution is based on the Mesh-Routing method, proposed by Lenstra et al., for which only theoretical estimates have been reported. The Mesh-Routing method has been implemented in the FPGA devices in order to come up with the concrete performance measures. The two types of Mesh Routing method, basic and improved, have been implemented and compared. Based on the experimental results for a partial mesh implemented on a single FPGA, the execution times of the Matrix Step for the case of factoring 512-bit and 1024-bit numbers have been calculated. The computation time for the case of a square systolic array of FPGAs interconnected among each other has

been extrapolated. For practical sizes of numbers used in cryptography, 1024 bits, the Matrix Step of factorization can be performed using 1024 Virtex II FPGAs in 27 days.

The design has been further implemented using SRC-6e Reconfigurable Computer, which is a hybrid computer consisting of microprocessors and FPGAs. Different approaches to partitioning the design description between VHDL and C have been investigated. The size of the mesh that can be implemented using the SRC-6e computer has been determined, and the execution time estimated for the case of factoring of large numbers. Furthermore, the influence of mesh parameters on the execution time and utilization of FPGA resources has been explored.

1. Introduction

RSA, developed by Ron Rivest, Adi Shamir and Leonard Adleman in 1977, is the primary public key cryptosystem used for providing the security of electronic information over the Internet. RSA is used in a variety of products and applications around the world. It is estimated that it protects around 95% of the electronic commerce [7]. RSA is used in many network security protocols, such as SSL, S/MIME and OpenPGP. It is integrated into many current operating systems including Microsoft Windows and Sun Solaris. It is widely used by corporations, laboratories and universities.

The security of RSA is based on the difficulty of factoring a large integer N into its prime factors P and Q . Factoring large integers is one of the most challenging tasks in cryptanalysis. The factorization of a 512-bit number required about 8400 MIPS years, and the complete process took about seven calendar months with 300 fast PCs, workstations and Cray C916 supercomputer spread over twelve sites in six countries [7]. According to the estimate from RSA Security Inc. regarding the number of memory and PCs needed to break the 1024-bit number, it would take 342,000,000 PCs with 500 MHz speed and 170 GB RAM to work for one year to factor a 1024-bit number.

The Number Field Sieve (NFS) is the asymptotically fastest known algorithm for the factorization of large numbers (110 digits or more) [8]. This method has recently been used very effectively to factorize the RSA numbers, the latest being the RSA-576 number

with 576 bits (174 decimal digits) [6]. These experiments used distributed PCs and supercomputers, which are general purpose computers, and are hard to scale for larger sizes of numbers to factorize. Recently, there have been proposals for custom-built hardware circuits that can reduce the time and cost of factorization for large number compared to software implementations.

The two most time consuming steps of the NFS algorithm are the Sieving and Matrix steps. My thesis focuses on the Matrix step in the NFS. This step involves multiplications of a large sparse matrix with vectors. This result is then used to identify linear dependencies between the entries in the sparse matrix. For the Matrix step, there are two proposed solutions. Mesh Sorting approach was proposed by Bernstein [2] while the Mesh Routing method was proposed by Lenstra et al. [10]. Architectures proposed by Bernstein [2] and Lenstra et al. [10], are estimated to bring significant improvement in the computing cost and time for factoring very large numbers like 1024-bit numbers. These architectures scale very effectively over the conventional approach of NFS.

I have implemented the Mesh Routing algorithm in reconfigurable hardware for the matrix step to provide the concrete performance and resource measures in the case of FPGA (Field Programmable Gate Arrays) technology. Previously, only theoretical estimations have been shown for the proposed mesh algorithms.

It is important to apply hardware solutions to the NFS, as when large computational power is needed for factoring large numbers, hardware implementations have the distinct advantage of inherent parallelism.

For a computationally intensive problem, such as factoring, reconfigurable hardware offers inherently better performance, scalability, and the price-to-performance ratio than conventional computers based on microprocessors. At the same time, reconfigurable hardware is much more flexible, easy to program and experiment with, and reusable compared to specialized hardware based on ASICs. Field Programmable Gate Arrays (FPGAs) are widely used to implement reconfigurable hardware. In the field of factorization, reconfiguration is needed since the best factorization algorithms involve computationally intensive sequentially executed steps, such as the Sieving and Matrix steps. In reconfigurable hardware, these steps can be executed using the same hardware, without any additional cost. Additionally, when new better algorithms for factorization are developed, hardware architecture can be upgraded and reconfigurable devices re-utilized.

In this study, I use the space-sharing time-multiplexing approach by which we are able to reutilize the FPGA devices in subsequent internal stages of the computations. This overcomes the problem of the need for a large number of FPGA devices, and the need for a large budget. In order to evaluate trade-offs between cost and performance, I report all performance measures for a varying number of FPGA devices. My implementation and study presents the first concrete performance and resource measurements regarding the reconfigurable hardware architectures for the NFS Mesh Routing method.

Reconfigurable Computers are general-purpose high-end computers based on a hybrid architecture and close system-level integration of traditional microprocessors and

FPGAs. One such computer is the SRC-6e reconfigurable computer. Reconfigurable computers constitute a perfect tool for cryptographers and cryptanalysts since they combine the inherent parallelism of hardware designs in the FPGAs, with the ease of programming in high-level languages for rapid application development. They have distributed memory, specialized functional units, flexible size, high speed data transfer and embedded memory access. Further, they provide the reconfigurability of the hardware to implement different architectures at different times. I have implemented my designs using the SRC-6e reconfigurable computer and further analyzed different design entry approaches for this distributed application.

2. Number Field Sieve Factorization and the Matrix Step

2.1 NFS

The Number Field Sieve (NFS) is the fastest known algorithm for factoring large integers [8]. Number Field Sieve has a sub-exponential time and space complexity with respect to the size of the number being factored. Sub-exponential functions rise faster than polynomial functions and slower than exponential functions. Let N be the number to factor. The sub-exponential function of the Number Field Sieve is defined as:

$$L_N(a) = e^{(a+o(1))} (\log N)^{1/3} (\log \log N)^{2/3} \quad (1)$$

This function applies to both the time and the space complexity of the Number Field Sieve. The $o(1)$ is a function which approaches 0 as N gets large and a is the positive real parameter which affects the growth rate of the function. For General Number Field Sieve, the value of a is 1.923.

The four steps of the Number Field Sieve algorithm are:

1. Polynomial Selection
2. Sieving
3. Matrix (Linear Algebra)
4. Square Root

Out of these four steps, the Sieving and Matrix steps are the most expensive steps. There are two fields involved in NFS, the rational field and the algebraic field. The rational

field is formed by rational numbers. The algebraic field is formed by algebraic numbers where an algebraic number is the root of the monic irreducible polynomial. In the Polynomial Selection step, one chooses a positive degree d ,

$$d \approx \left(\frac{3 \log N}{\log \log N} \right)^{1/3} \quad (2)$$

Then, a number m is chosen such that,

$$m \approx N^{1/(d+1)} \quad (3)$$

Two polynomials are constructed. The first polynomial, of degree d ,

$$f_1(x) = \sum_{i=0}^d a_i x^i \quad (4)$$

is constructed such that $f_1(m) \equiv 0 \pmod{N}$. The coefficients of the polynomial are obtained by representing N in base m .

The second polynomial used is

$$f_2(x) = x - m \quad (5)$$

These two polynomials are converted to the corresponding polynomials in two variables as:

$$F_1(x, y) = y^d f_1(x/y) \quad (6)$$

$$F_2(x, y) = y \cdot f_2(x/y) = x - ym \quad (7)$$

Before we proceed to Sieving step, we choose a set of prime numbers starting from the smallest prime number 2 consecutively to a chosen maximum prime number. This set is called a *factor base*. An integer is called *B-smooth* if all of its prime factors are less than the number B . Synonymously, an integer is called to be *smooth* within a *factor*

base with the maximum bound B , if all of its factors are the primes contained inside the *factor base*. The B_r and B_a are chosen as the maximum numbers of the *factor bases* for the rational and algebraic fields.

In the Sieving step, we proceed to find many (a,b) pairs such that $F_1(a,b)$ is B_a -smooth and $F_2(a,b)$ is B_r -smooth in the respective algebraic and rational factor bases. This means that all of the prime factors of $F_1(a,b)$ are less than B_a and all of the prime factors of $F_2(a,b)$ are less than B_r . Each such (a,b) pair is called a *relation*. Each *relation* will yield a sparse D -dimensional bit vector. The contents of D -dimensional bit vector is the exponents of prime factors of $F_1(a,b)$ or $F_2(a,b)$ modulo 2.

$$D \approx \pi(B_r) + \pi(B_a), \quad \pi(y) = \text{number of primes less than } y \quad (8)$$

D is the sum of the size of the two factor bases. *Relations* more than D are searched in the Sieving step to form the square matrix of D rows and D columns. The columns represent the *relations* and the rows represent the primes.

The collection of sparse D -dimensional vectors obtained after the Sieving step forms a sparse matrix to be processed in the Matrix step. In the Matrix step, one or more linear dependencies modulo 2 among the corresponding D dimensional bit vectors are searched by doing matrix operations. The product taken over these dependencies is used to build the congruence of squares,

$$X^2 \equiv Y^2 \pmod{N} \quad (9)$$

or, $(X-Y)(X+Y) \equiv 0 \pmod{N} \quad (10)$

Then, $\gcd (X-Y, N)$ will give one of the factors of N . The other factor can be found by dividing N by this factor. The numbers X and Y are obtained after the Square Root step.

There is a tradeoff between the Sieving step and the Matrix step. If we choose large prime factor bounds B_a and B_r , it is easy and less time consuming to find (a,b) pairs for which $F_1(a,b)$ and $F_2(a,b)$ are *smooth* (have all the prime factors less than B_a and B_r respectively). This is because numbers having large primes are now found to be *smooth* and quickly detected. However, the Matrix step gets more time consuming due to the large matrix size formed by large bounds of B_r and B_a . The tradeoff also occurs in the reverse way correspondingly.

2.2 Matrix Step in NFS

The Matrix step is concerned with finding linear dependencies in the sparse matrix ‘ A ’ obtained from the Sieving step. There are different ways of finding linear dependencies in a sparse matrix of large size. The Gaussian Elimination method is ineffective due to the very large size of the matrix and the property of the matrix being sparse. More effective methods are Block Lanczos algorithm and Block Wiedemann algorithm. The more efficient method for hardware implementation is the Block Wiedemann algorithm. The linear dependencies are found using the Block Wiedemann algorithm [4][16] by doing multiple matrix-by-vector multiplications of the form

$$A \cdot v_i, A^2 \cdot v_i, \dots, A^k \cdot v_i \quad (11)$$

where v_i is one of the random vectors ($1 \leq i \leq K$) and $k \approx 2D/K$. These vectors are selected randomly and are not sparse. D is the number of columns of matrix A , K is the blocking factor where either $K=1$ or $K \geq 32$ (where different vectors v_i are handled simultaneously). Another set of vectors $\{u_i\}$ is selected randomly and the sequences

$$u_i \cdot v_i, \quad u_i \cdot A \cdot v_i, \quad \dots, \quad u_i \cdot A^k \cdot v_i \quad (12)$$

($1 \leq i \leq K$) are used to find the linear dependent vectors [4] [16] with additional D/K multiplications performed at the end. The total number of multiplications required is $3D/K$. These matrix-by-vector multiplications dominate the storage cost and the time complexity of the Matrix step.

3. Mesh Circuit for the Matrix Step

3.1 Bernstein's Mesh Based Approach

Daniel Bernstein has proposed a mesh based approach for the Matrix step of the Number Field Sieve [2]. Bernstein proposed the distributed algorithm for the Matrix step, which reduces the asymptotic running time of the Number Field Sieve algorithm for large numbers. The computation is done in a mesh-connected array of processors with local memory. This utilizes memory efficiently compared to a PC-based implementation, where a huge memory is waiting for just one processor to be accessed. The distributed approach of mesh performs multiple numbers of operations in parallel. This reduces the asymptotic runtime by $O(\sqrt{D})$, where D is the number of columns or rows in the matrix. This has also brought down the cost of a Matrix step from trillions of dollars to millions of dollars [13].

Bernstein introduced the measure of throughput cost which is the multiplication of memory cost and the running time of the factorization algorithms. Since most of the processor and hardware cost is dominated by the memory, this is a reasonable measure of the product of the cost and time.

3.2 Mesh Sorting

Bernstein has proposed the Mesh Sorting algorithm for doing matrix-by-vector multiplications [2]. This uses Schimmler's sorting method [2]. Schimmler's algorithm sorts m^2 numbers in $8m-8$ compare-exchange steps in a two-dimensional mesh of size m^2 , where m is a power of 2. In each step, simultaneous operations are done among the cells. It uses a recursive algorithm to sort inner quadrant first before doing row and column sorting at the end. Schimmler's sorting can be built with a cost proportional to m^2 . The computational time is proportional to m . One matrix-by-vector multiplication in Mesh Sorting, requires three Schimmler's sorting operations with a total of $3 \cdot 8 \cdot m = 24 \cdot m$ compare-exchange steps. The throughput cost of the computation (product of cost and time) is in the order of m^3 .

If the computation is done on the processor, the processor on the other hand will also have the cost proportional to m^2 words of memory. It can sort the numbers in time in the order of m^2 . The throughput cost is $m^2 \cdot m^2 = m^4$. Thus, the throughput cost has decreased from m^4 to m^3 when going to the custom mesh machine. Bernstein's approach has produced improvement in the throughput cost asymptotically on large numbers.

3.3 Mesh Routing

Lenstra et al. built upon Bernstein's idea of doing the mesh computations with distributed cells and active local memory. Lenstra et al. proposed the Mesh Routing method to do matrix-by-vector multiplication in the Matrix step which uses the Block Wiedemann algorithm[4][16]. The blocking factor $K=1$ or $K \geq 32$ is used. There is a

single routing operation per multiplication, which has an average number of $2 \cdot m$ to $4 \cdot m$ steps, where $m \times m$ is the size of the mesh. The routing algorithm used is the clockwise transposition routing algorithm, which repeats four steps of compare-exchange operations in four directions.

3.4 Mesh Sorting vs. Mesh Routing

In Mesh Sorting, the recursive sorting is used. In Mesh Routing, the routing is done by repeated compare-exchange operations in four directions. In Mesh Sorting, only one multiplication of a matrix and a vector occurs at a time, whereas in Mesh Routing, K multiplications of a matrix and K vectors can be handled at the same time using blocking factor $K \geq 32$. This reduces the total time. The device cost is also reduced in Mesh Routing as each cell in the mesh can handle multiple matrix columns. Mesh Routing can handle large matrix size for a fixed size chip. Thus, it does more computations and provides improved performance. Accordingly, I have chosen Mesh Routing for my design.

4. Distributed Matrix Computation

When factoring numbers of the size of 512-bits and 1024-bits, used in cryptography, the matrix generated after Sieving step is huge, and the number of columns exceeds a million. For a matrix of this size, the complete mesh takes a large amount of area exceeding the normal die size of the chip, or the size of a single FPGA chip. To account for this problem, Geiselmann and Steinwandt have proposed the distributed variant of the Matrix Step, breaking down the large matrix-by-vector multiplication into smaller matrix-by-vector multiplications [5].

The same device can be utilized to do sub-computations one after another depending on how many devices are available and affordable. The rectangular matrix A obtained from the Sieving step is preprocessed to have a uniform distribution of non-zero entries in each column. The matrix A can then be broken down into $s \times s$ sub-matrices of the size D/s by D/s , where D is the column size of the original matrix A . In Figure 1, the matrix is broken down into $3 \times 3 = 9$ sub-matrices $A_{i,j}$, $1 \leq i, j \leq s$. The vector v is broken down into 3 sub-vectors v_j such that the multiplication $A \cdot v$ can be realized as shown in Figure 1.

$$\begin{array}{|c|c|c|} \hline \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} \\ \hline \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} \\ \hline \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} \\ \hline \end{array}
 \begin{array}{|c|} \hline \mathbf{v}_1 \\ \hline \mathbf{v}_2 \\ \hline \mathbf{v}_3 \\ \hline \end{array}
 =
 \begin{pmatrix} \mathbf{A}_{1,1}\mathbf{v}_1 + \mathbf{A}_{1,2}\mathbf{v}_2 + \mathbf{A}_{1,3}\mathbf{v}_3 \\ \mathbf{A}_{2,1}\mathbf{v}_1 + \mathbf{A}_{2,2}\mathbf{v}_2 + \mathbf{A}_{2,3}\mathbf{v}_3 \\ \mathbf{A}_{3,1}\mathbf{v}_1 + \mathbf{A}_{3,2}\mathbf{v}_2 + \mathbf{A}_{3,3}\mathbf{v}_3 \end{pmatrix}$$

Figure 1. Distributing computation of a large matrix using sub-matrix computations

The final result $A \cdot v$ can be obtained as shown in equation (13) .

$$\mathbf{A} \cdot \mathbf{v} = \begin{pmatrix} \sum_{j=1}^s \mathbf{A}_{1,j} \cdot \mathbf{v}_j \\ \vdots \\ \sum_{j=1}^s \mathbf{A}_{s,j} \cdot \mathbf{v}_j \end{pmatrix} \quad (13)$$

If only a certain number of chips is available, we need to load the contents of sub-matrices A_{ij} to the mesh in the chip together with sub-vectors v_j . Maximum number of I/O pins available in the chip are used to load the inputs and unload the outputs for faster processing time. At the end, the results are xored with the results of the other matrix-by-vector multiplications at the end.

For limited resources, this gives us the opportunity to reuse the device with a tradeoff being the running time. The circuit I developed is based on this principle, and performs a large matrix-by-vector multiplication through a sequence of smaller sub-matrix by sub-vector multiplications.

5. Mesh Routing Design

Matrix-by-vector multiplication is done using the Mesh Routing circuit proposed by Lenstra et al. [10]. When the matrix is sparse, multiplication can be performed efficiently by considering only the non-zero entries in the columns of the sparse matrix. In Figure 2, the matrix-by-vector multiplication is shown for matrix A and vector v . The column vector v is horizontally positioned to show the multiplication of bits at the same positions of the vector v and the row of the matrix A . For efficiency, only the non-zero bits of the matrix A can be multiplied with the bits of vector v to compute the bits of the result vector $A \cdot v$.

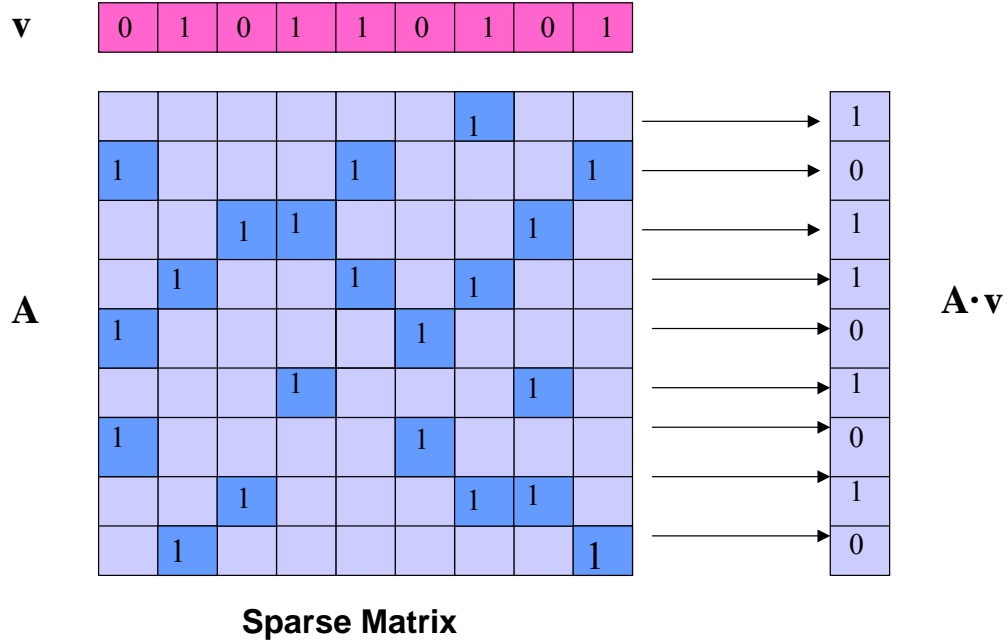


Figure 2. Matrix-by-vector multiplication operation

Each non-zero entry of the sparse matrix A can be viewed as a packet together with the vector bit of v that should be routed to the row position of the destination result vector. Accumulating the xor of all the packet's vector bit with the destination position's result bit will give the final result bit at that position. Thus, the multiplication can be performed by routing each such packet to the destination row-position of the packet, and accumulating the xor of the packet bits. The routing is performed in a square mesh of cells of two dimensions. There is a single routing operation per multiplication.

Lenstra et al. proposed two versions of the routing based circuit, a 'Simpler' version and an 'Improved Routing' version. The basic routing design I developed is a slight variant of improved version, where one cell handles one column of the matrix. The Improved Routing design I developed is the full improved routing version proposed by

Lenstra et al. [10], where each cell handles multiple columns of the matrix. In the Basic Design, one cell holds the non-zero row indices of one column of the sparse matrix.

5.1 Sparse Matrix and Vector

The sparse matrix is generated from the Sieving step of the factorization. We have a sparse matrix A , whose columns represent the entry for each sieving pair of (a,b) found in the Sieving step. The column values represent the exponents of the primes modulo 2 in the prime *factor base*, where function of (a,b) is the product of the primes with those exponents.

Let the number of columns of the sparse matrix be D and the density be d . A density of d means the maximum number of ones in any given column never exceeds d . The vector is a randomly chosen vector with length D . This is to be multiplied by matrix A . In the mesh, multiple vectors can be loaded into each mesh cell, thus achieving concurrent multiplications of these vectors with the matrix.

5.2 Mesh of Cells

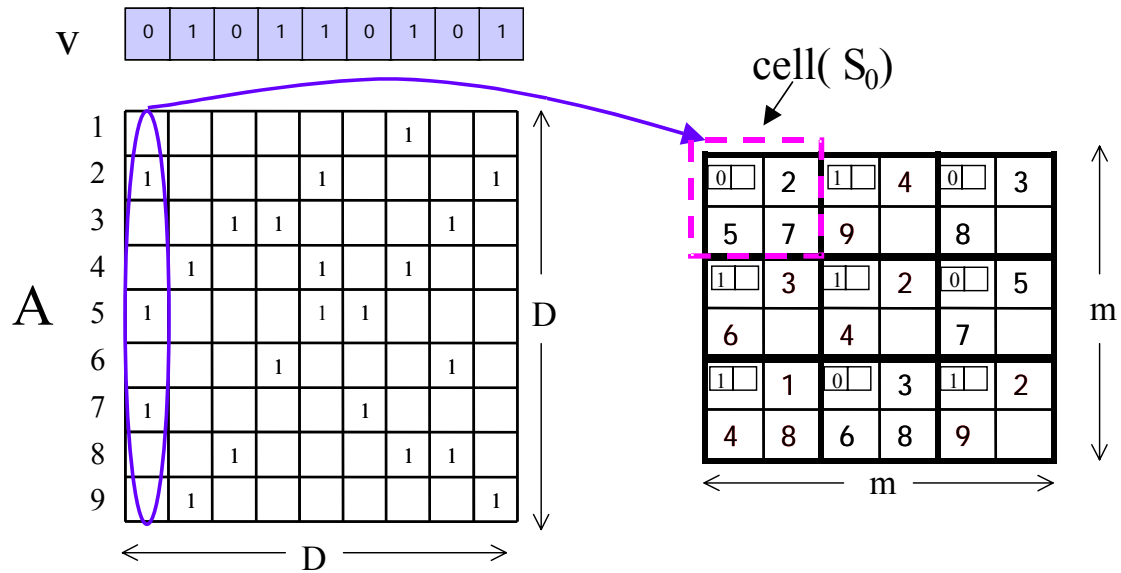


Figure 3. Mesh corresponding to the sparse matrix A

The mesh of cells is generated as shown in Figure 3 where the mesh has equal number m of rows and columns where $m = \sqrt{D}$. The coordinates of each non-zero entry in the column are stored in each cell. These are utilized in the routing operation. S_i denotes the i -th cell in the row major order, $i \in \{1, 2, \dots, (m \cdot m)\}$. Each cell S_i is the target destination of the packet whose destination row and column indices match with the cell's row and column position. At the end of the routing algorithm, all the packets stored initially are routed to their destinations. This can be seen in Figure 4, where the packets with destination of four are routed to the fourth cell.

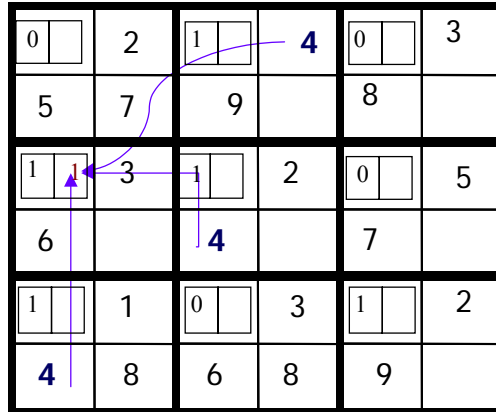


Figure 4. Routing of packets to the cell in the mesh

Each cell has registers $P[i]$, which store the vector bits, registers $P'[i]$ are used to store the intermediate results. Local memory $R[i]$ is used to store the row and the column indices of the packets of all the 'ones' in one column C_i of the matrix A . These destination indices are obtained from the position indices of the ones in the sparse matrix for a given column.

Multiplication Steps:

- 1 Load the cells with $R[i]$ values of the matrix A , and $P[i]$ with the vector bits of the multiplier vector v
2. Set $P'[i]=0$ for all i in S_i
3. Invoke clockwise transposition packet routing algorithm of the each non zero $R[i]$ values to the target cell $S_{R[i]}$

4. Each time a value arrives at the target, the bit of $P'[i]$ is xored with the incoming packet vector bit
5. Copy bit results from $P'[i]$ to $P[i]$
6. $P[i]$ contains the result of the multiplication in S_i . Unload $P[i]$ and the mesh is ready for the next multiplication

Blocking factor $K \geq 32$ can be used to handle K multiplications in parallel in the same circuit. K bits of information are transferred to the target cell in this case related to K different vectors at the same time of one Mesh Routing.

5.3 Clockwise Transposition Routing Algorithm

This algorithm is used for routing the individual packets to their destinations. This algorithm repeats the four steps until all the packets are routed to their destination cells. In each step of this algorithm the compare and exchange operation is done between two neighboring row or column cells. The destination of the packets in the cells are compared and packets are exchanged only if the exchange reduces the distance to target of the farthest traveling packet.

The four steps involved in the compare and exchange operation are done in the following manner:

- 1: compare-exchange between each cell in the odd row and its neighboring cell in the even row above it
- 2: compare-exchange between each cell in the odd column and its neighboring cell in the right even column

3: compare-exchange between each cell in the odd row and its neighboring cell in the even row below it

4: compare-exchange between each cell in the odd column and its neighboring cell in the left even column

This compare and exchange operation is done until all the packets are routed to their destinations. The time taken by routing operation is $4 \cdot m$ compare-exchange operations.

5.4 Improved Mesh Routing Algorithm

In the improved routing circuit, multiple column entries of A are handled in one cell. Each cell has non-zero row coordinates of $p > 1$ columns of the original matrix A in $R[i]$ storage which has a size of $d \cdot p$ entries. Each cell also has p bits of each vector v in $P[i]$. The mesh needs D/p cells(processors). Here, $m = \sqrt{(D/p)}$. The time taken by clockwise transposition routing for a single value is about $4 \cdot m = 4 \cdot \sqrt{(D/p)}$ compare-exchange operations. Since there are $p \cdot d$ matrix entries in each cell, the routing operation is iterated for $p \cdot d$ times with the total of $p \cdot d \cdot 4 \sqrt{(D/p)}$ compare-exchange operations. $P[i]$ containing the result of multiplication will be two dimensional with K rows and p columns. The result will be contained in $P[i]$. Blocking factor $K= 1$ or $K \geq 32$ is used to handle K multiplications in parallel in the same circuit as similar to the Basic Mesh Routing algorithm.

6. FPGA Hardware Platform

Hardware implementation has the benefit of performance over software implementation. There are two dominant hardware platforms: Application Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs). ASICs are the custom built hardware circuits that must be designed all the way from specification to physical layout. They have high performance but have disadvantages of long design cycles and large development costs. The circuit is fixed after it is fabricated. FPGAs are off-the-shelf devices that can be reprogrammed by the designers themselves. FPGAs can be reconfigured for different circuits, providing different functionality at different times. Reconfiguration takes less than a tenth of a second. FPGAs provide performance improvement over microprocessor designs.

Most of the dominant FPGAs on the market are produced by Xilinx and Altera companies. Xilinx has produced the advanced FPGAs in the family of Virtex FPGAs. The most recent on the market are Virtex II FPGAs, which can hold the largest number of logic blocks. A structure of Virtex II FPGA is shown in Figure 5. FPGA consists of many Configurable Logic Block slices (CLB slices), which are connected through programmable interconnects.

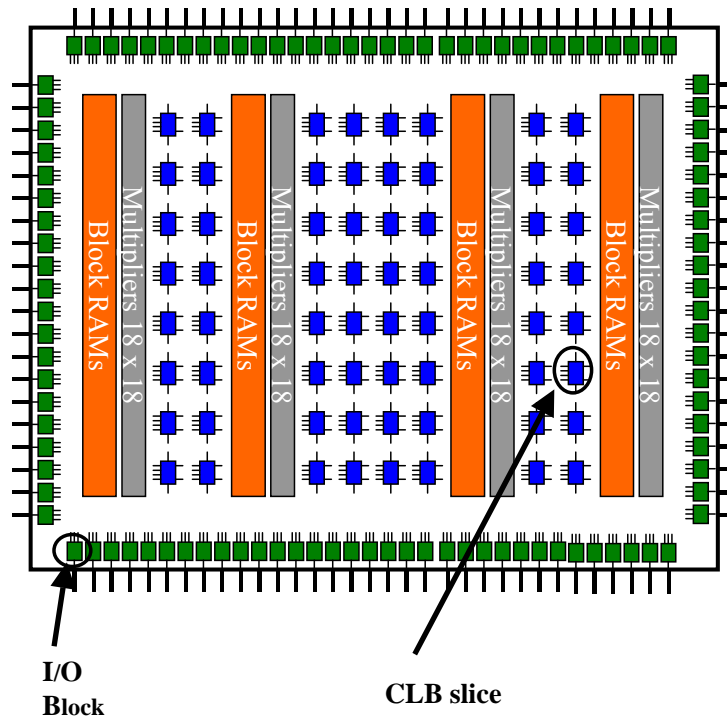


Figure 5 Virtex II FPGA Architecture

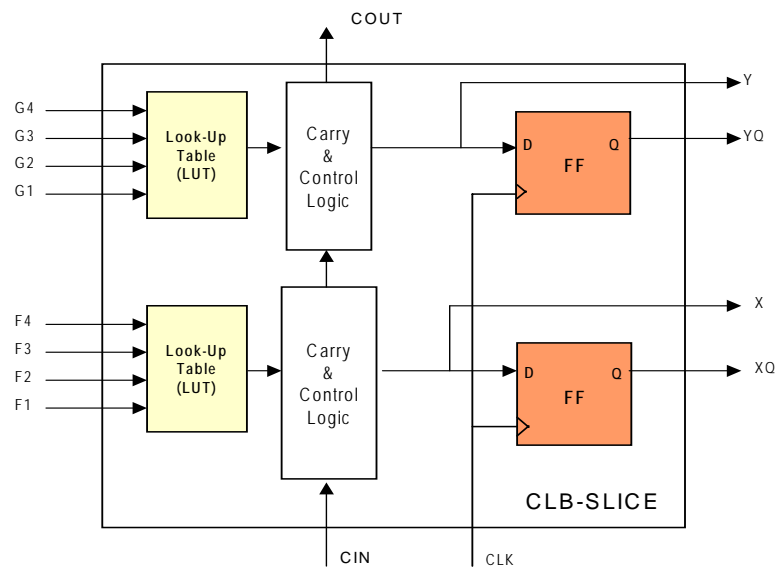


Figure 6. CLB slice structure

Each CLB slice contains two Look-Up-Tables (LUT) and two Flip-Flops (FF), as shown in Figure 6. LUT is used to implement combinational logic, and FF is used for register storage. Each LUT can handle any logic function of 4 inputs and produce one output. LUT consists of a 16x1-bit memory. LUT can be configured to perform the function of ROM, RAM, and shift register. The delays in the FPGAs consist of LUT delays, called logic delays, and the delays of interconnects between LUTs, referred to as routing delays.

FPGAs also include I/O Blocks, which are used for input and output interface and buffering of I/O signals, as shown in Figure 5. In addition, Virtex II FPGA has dedicated multipliers and Block-RAM storage. Area in FPGA is counted in terms of the number of CLB slices used. This measure can be further decomposed into the number of LUTs and the number of FFs used.

7. Hardware Architectures of Mesh Routing Designs

7.1 Hardware Architecture of Basic Mesh Routing Design

In the Basic Mesh Routing design, each cell holds the non-zero entries of one column of the original matrix A . The hardware architecture for performing different operations of Basic Mesh Routing design are illustrated next.

7.1.1 Loading and Unloading

The row value of non-zero entries in the original matrix A is the routing address, which is converted to row and column indices (r, c) . The column value of non-zero matrix entries is the loading address, which is also converted to the coordinates (ri, ci) which tell which cell should keep this packet on loading. Routed together with this value is the status bit, showing the validity of the packet.

Packets are loaded one after another from the memory to the leftmost top cell of a mesh, one per clock cycle, as shown in Figure 7 for the case of a 4x4 mesh. Each cell will shift this packet to its right neighbor. So each packet comes one after another in the pipelined fashion. The rightmost cell of each row forwards the packet to the leftmost cell of the next row. In this way the packet shifts through the cells. Each cell decodes the loading address of the packet. If the address matches its coordinates and if the packet is

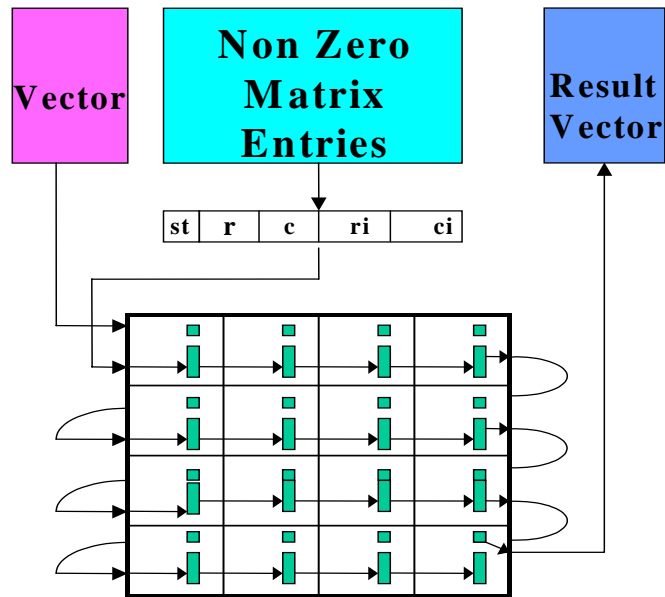


Figure 7. Basic loading and unloading

valid, then it stores the packet by writing it in the next available address in the LUT RAM storage.

In order to minimize the total number of clock cycles for loading, the initial packets to be shifted to the mesh should be the packets that correspond to the last cell at the rightmost bottom end. The next packet should be of second last cell and so on. In this way in $d \cdot m \cdot m$ clock cycles (where d =maximum number of packets per cell, m = number of mesh columns or rows), all the packets are guaranteed to reach the corresponding cells in this phase. The loading of the vectors is done similarly, entering the mesh through top-leftmost cell, shifting from one cell to another and from one row to the next row. Here, $m \cdot m$ clock cycles are needed to load the vectors.

The result of the matrix and the vector multiplication is the result vector. After the computation is finished, and the result vector stored in each cell, the result vector is unloaded from the rightmost bottom cell. The vectors from each cell are shifted out in the same direction as in the loading phase to minimize the interface resources of the cells. Finally, each vector's members are stored in the memory serially. This is the basic approach of the design.

Another approach with loading to multiple rows in parallel and loading out from multiple rows in parallel is also developed in the design as shown in Figure 8. This reduces loading and unloading time but is restricted by the IO pins available in the chip or the maximum bit width of the interface to the memory. So some hybrid approach is also possible in the design where data is loaded to some k rows in parallel. Maximum IO pins in the chip are considered for calculating the loading and unloading time in estimating for 512-bit and 1024-bit factorization.

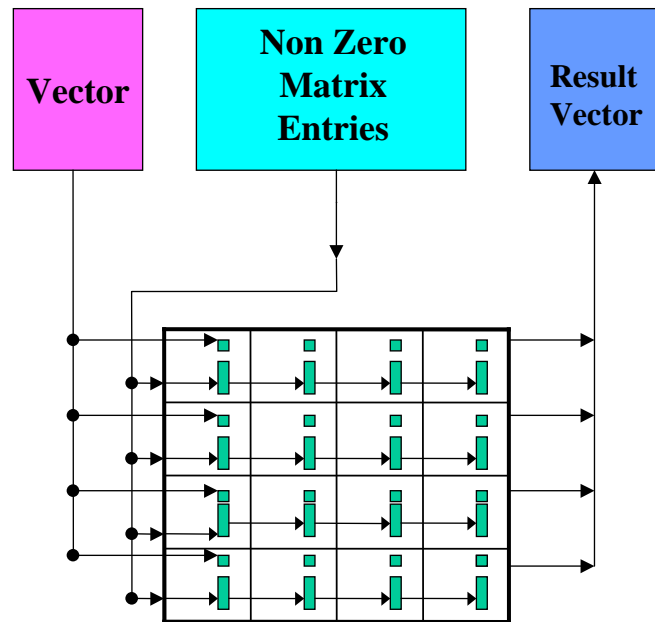


Figure 8. Parallel loading and unloading on multiple rows

7.1.2 Routing Operation

After loading, the mesh is ready to do the computations for matrix-by-vector multiplication operation. The matrix-by-vector multiplication operation is done by routing each packet with the corresponding vector bits of that packet (vector elements of v at the corresponding column position of original matrix A) to the destination cells determined by the routing address (r,c) in the packet. Whenever a packet reaches its destination, the vector bits in the packet are xored to the accumulating partial result in that destination cell. After all packets are routed, the accumulating result vector registers will have the final result in each cell.

The maximum number of non-zero entries in each column of the original matrix A determines the maximum number of packets each cell is holding at the beginning. This

determines the number of iterations for which the routing operation has to be repeated. In each iteration, the next packet stored in local memory (RAM) in each cell is loaded to the current packet holding register in each cell. Then, these current packets are routed to the destination by the use of clockwise transposition routing algorithm as mentioned before.

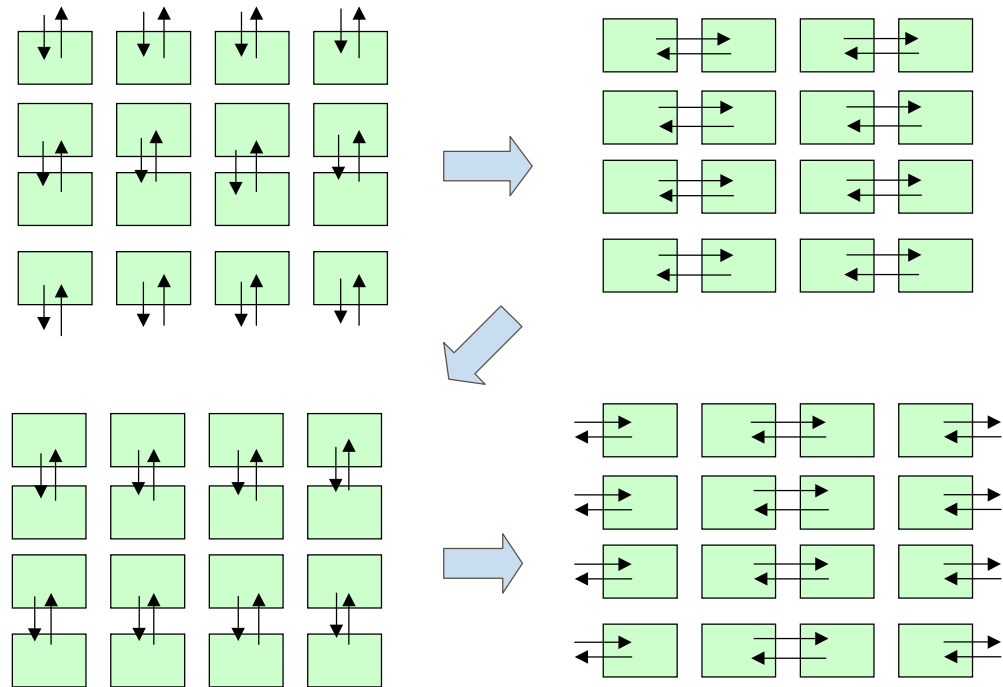


Figure 9. Four iterations of compare-exchange operation

Clockwise transposition routing repeats four phases of compare-exchange operations as reported before. Figure 9 shows the four steps of compare-exchange operations for the case of mesh of 4x4. On careful examination, I found that the first cell starts doing compare-exchange with the top neighbor and then right neighbor and bottom neighbor and then left neighbor. So it does comparisons in the clockwise order.

Observing the second cell, it does compare-exchange in the anticlockwise fashion. These clockwise and anticlockwise compare and exchange operations are as shown in Figure 10 for the case of mesh of 4x4. Actually, I found that the direction for compare and exchange depends on the property of sum of coordinates of the cell being either odd or even.

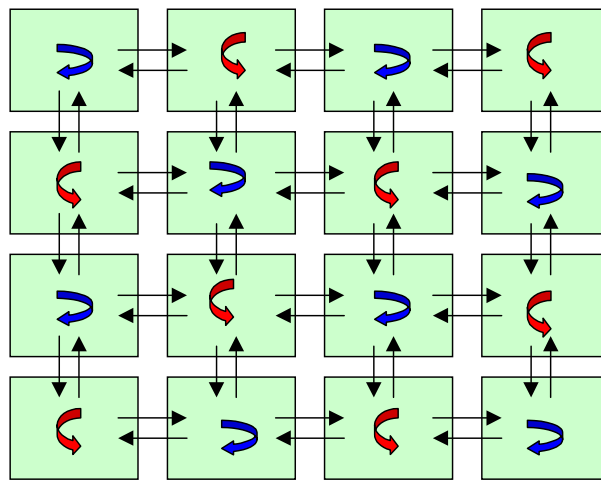


Figure 10. Compare-exchange direction for each cell

7.1.3 Compare-Exchange Operation

In each compare-exchange, the two neighbors send their packets to each other. Then, each cell independently compares the incoming packet with its packet and decides on whether to exchange (replacing its packet with the incoming packet) or not to exchange (discarding the incoming packet). There are two types of packets. One is valid and the other is invalid. Packets become invalid when they reach the destination. On

analysis, I found that there are four cases of compare-exchange operations for these different types of packets:

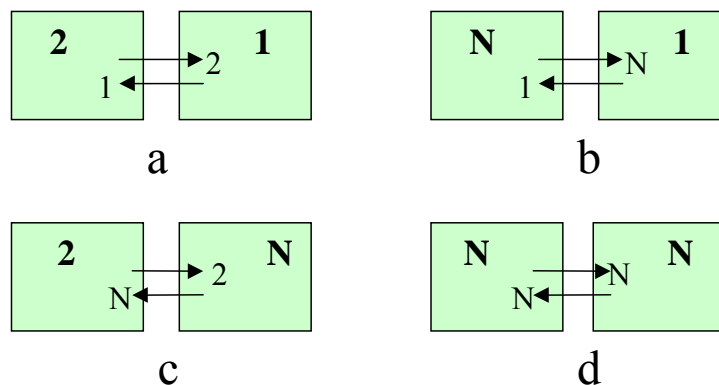


Figure 11. Compare-exchange cases

- a) Both packets are valid (Figure 11a). Thus, each cell may need to exchange the packets. Each cell decides independently (which is synchronized by the logic being implemented in each cell) by comparing the incoming packet's destination address with the current packet's destination address.
- b) Current packet in the cell is invalid but the incoming new packet is valid (Figure. 11 b, left cell, N represents the invalid packet). The cell may need to keep the new packet if it is traveling in the right direction or reaches the destination.
- c) Current packet in the cell is valid and the incoming new packet is invalid (Figure. 11c, left cell). The cell may need to destroy (annihilate) its packet if the other neighbor keeps its packet.

- d) Current packet in the cell is invalid and the incoming new packet is also invalid
(Figure 11d). In this case, no action taken.

7.1.4 Comparator in Cell

I implemented this logic in each cell in the comparator to account for all of the four cases as shown in Figure 12. As shown in Figure 12, the comparator takes in three values, the current packet, the new packet, and the cell's coordinates. Based on the phase of iteration, either row or column values have to be compared which is selected in the first level of multiplexors. Then the status of the current packet ($s1$) and the new incoming packet ($s2$) are compared. If the status bits are both one meaning both packets are valid, then the current packet and the new packet are compared. One cell compares greater than relation and the other cell compares less than relation. If the comparison returns true, then both cells replace their packets with new packets by enabling the 'exchange' control signal. Otherwise, the 'exchange' goes low signifying no exchange is needed.

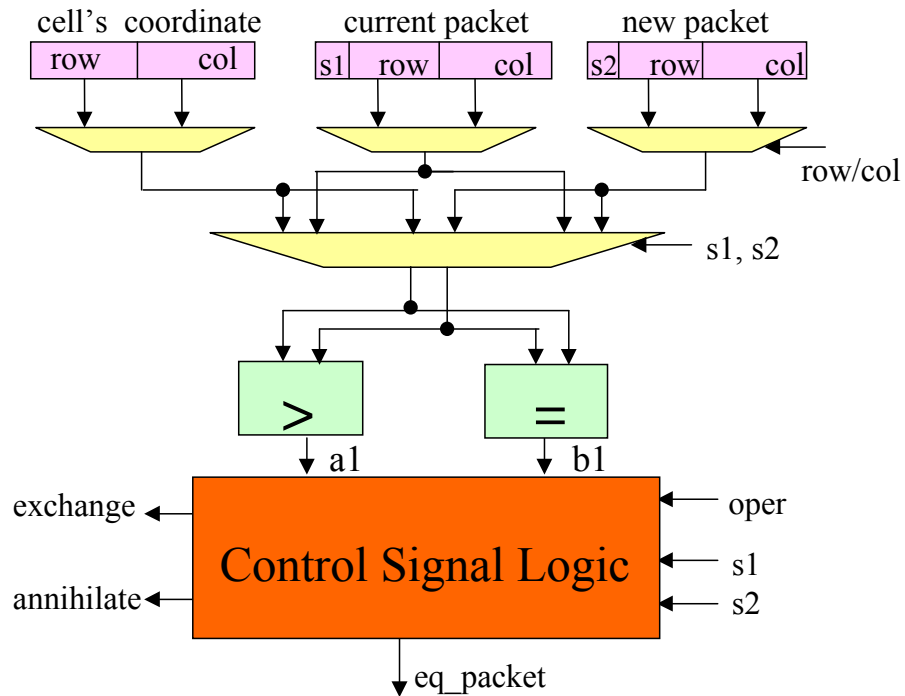


Figure 12. Comparator logic

When the status bits are $s1=0$ and $s2=1$, the current packet is invalid and new packet is valid. So the cell decides whether to keep the new packet by comparing cell's coordinate with the new packet coordinate by doing either greater or equal to comparison or less or equal to comparison and enables '*exchange*' control signal if it needs to keep the new packet.

When the status bits are $s1=1$ and $s2=0$, the current packet is valid and new packet is invalid. So the cell decides whether to destroy (annihilate) its current packet by comparing cell's coordinate with its current packet's coordinate. It does this by doing greater than or less than comparison and enables '*annihilate*' control signal if it needs to destroy its packet.

Even though each cell is doing independent comparisons, the same logic of compare-exchange in each cell ensures that both cells' decisions match with each other. So if for both valid packets, if one cell exchanges, the other one also exchanges or none of them exchange. In the case when one is valid and the other is invalid, if one keeps the new packet, the other destroys its packet. If one does not keep the new packet, the other keeps its old packet. This ensures that there is no packet duplication and no packet loss.

When current packet and new packet have the same destination, the '*eq_packet*' signal is asserted. This leads to packet annihilation in one of the cells and the other cell xors the current packet's vector bits with the new packet's vector bits. This operation reduces the number of packet's being routed and reduces the congestion in routing. However, the practical experiment has shown that this does not improve the total routing operations on average.

7.1.5 Basic Cell Architecture

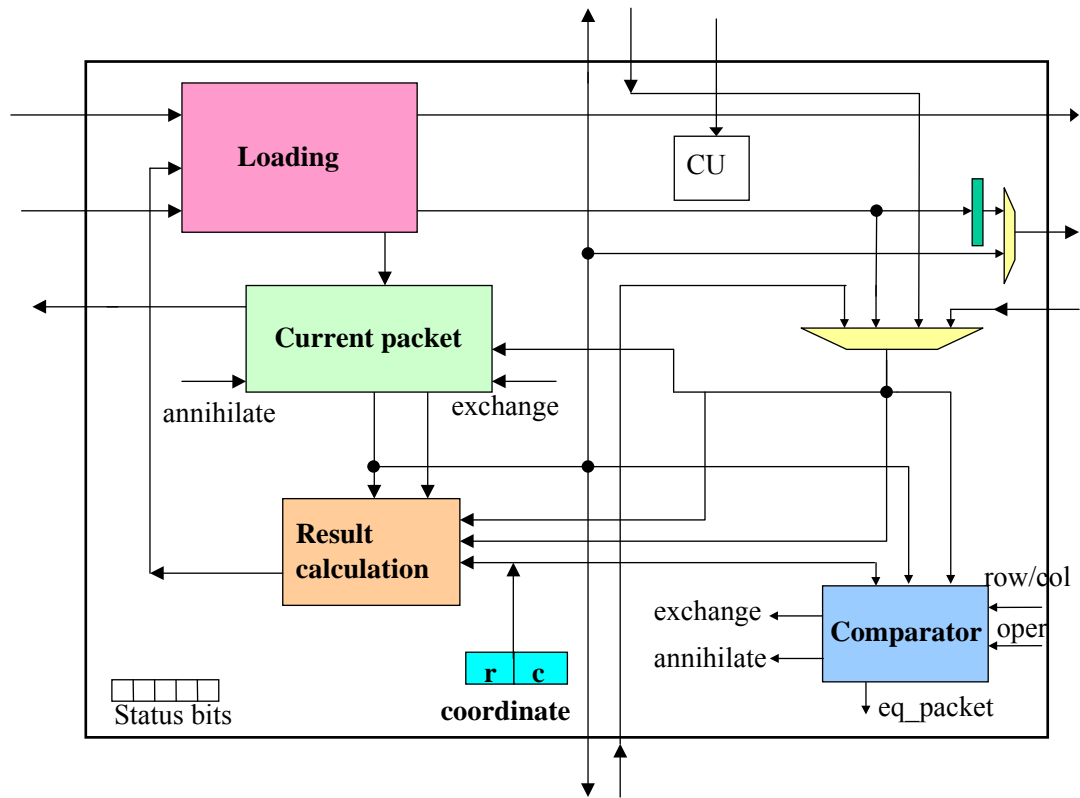


Figure 13a. Each Basic Cell

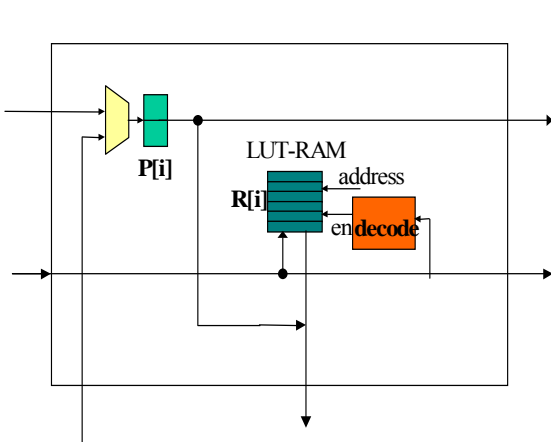


Figure 13b. Loading Unit

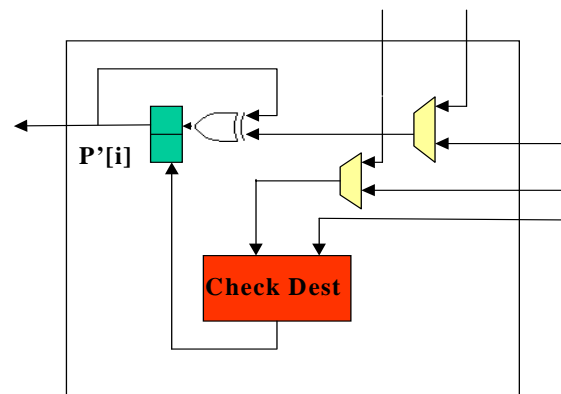


Figure 13c. Result Calculation Unit

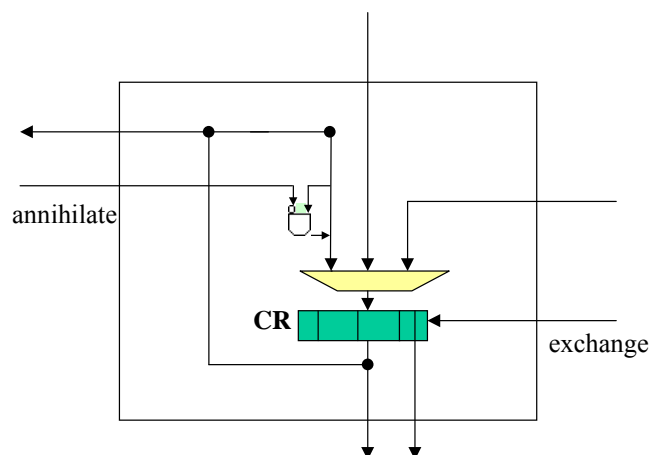


Figure 13d. Current Packet Unit

Figure 13. Detailed architecture of each Basic Cell

I designed the basic cell architecture as shown in Figure 13 where Figure 13a shows the high-level diagram of the cell's structure containing sub-blocks. The sub-blocks are shown in Figure 13 b, 13c, 13d. The Comparator resides in each cell and does comparison operation as described previously. The comparison operation is dynamic as the cell compares in clockwise or anticlockwise direction. Its role of being preceding or following neighbor changes per phase of clock cycle. The '*oper*' control signal signifies what direction of comparison to do, whether to decide on less than comparison or greater than comparison.

Each cell is connected to its four neighbors. So each cell gets input from its four neighbors and sends its current packet value to its four neighbors. We consider the Loading Unit (Figure 13b). The $P[i]$ registers store the input vector bits. The design is scalable to handle any number of vector bits with a corresponding change in the area.

The $R[i]$ is the local memory storage stored in Look-Up-Table RAM(LUT-RAM) for the packets in each cell. Each cell keeps the packets corresponding to the non-zero entries of one column in the original matrix A . The ‘*decode*’ unit decides if the loading address of the packet matches the cell’s address and enables the write operation to the memory on loading phase.

The cell stores its coordinates in (r,c) format. The $P[i]$ registers in Result Calculation Unit (Figure 13c) store the intermediate result vector bits after each routing and when the packet reaches the destination, the new vector bits are xored with the intermediate result bits in it. The ‘*Check_Dest*’ unit checks if the packet has reached its destination by comparing the cell’s coordinates with the new packet’s coordinates or its current packet coordinates.

The ‘*annihilate*’ signal in Current Packet Unit (Figure 13d) resets the status bit of the packet which changes it to ‘0’ if annihilation needs to be done. The ‘*exchange*’ signal in Current Packet Unit (Figure 13d) enables loading to the register for the current packet register, CR. The ‘*eq_packet*’ (Figure 13a) control signal is utilized when the current packet and the new packet have the same destination.

Each cell has status bits which are constants set during synthesis based on the cell’s coordinates. Some status bits are ‘*odd/even row*’, ‘*odd/even column*’ to signify whether it is in the odd row or column or not. Others are ‘*top-end*’, ‘*bottom-end*’, ‘*left-end*’ and ‘*right-end*’ to signify whether the cell is at the edges and at which edge. Also, there are status bits to signify whether the comparison starts from top or bottom (*top_start*) and direction of compare-exchange for each cell (*clockwise/anticlockwise*).

The action performed by the cell depends on these status values of the cell and the particular phase of iteration. So, the determination of which neighbor to compare, and to compare lesser than or greater than relation are determined by these status bits and the phase of iteration. There are external control signals of ‘*state*’ from the top unit to each cell to command on certain operation of loading, computing and unloading.

7.2 Hardware Architecture of Improved Mesh Routing Design

The Improved Mesh Routing design is the same as Improved Mesh Routing algorithm of Lenstra et al. [10]. This design has the property that the entries of multiple columns of the original matrix A are stored in each cell. Multiple entries share the computational logic making it possible to handle the larger matrix size within one FPGA chip, lowering the cost of computations.

The cell architecture for the Improved Mesh Routing is shown in Figure 14 where Figure 14a shows the high-level diagram of the Improved cell’s structure containing sub-blocks. The sub-blocks are shown in Figure 14 b, 14c, 14d.

7.2.1 Loading and Unloading

The loading and unloading is similar to the Basic Mesh Routing Design, with the difference that each cell stores entries from the multiple (p) columns of the matrix A . The storage also includes the indices saying which of the p columns the entry corresponds to. The storage is done in the LUT RAM, $R[i]$ (Figure 14b). When loading input vectors, the first vector bits are sent first. On loading, the first cell stores the incoming vector bits for

p clock cycles to the $P[i]$ storage (Figure 14b). Then, the first cell sends a one-bit valid signal together with the next incoming vector bits to shift to the second cell. Thus, the value of a valid signal is rippled through, together with the vector bits, to let each cell know when to store the vector bits available at the loading line.

7.2.2 Routing Operation

The routing algorithm is the same as for the Basic Mesh Routing Circuit as described in Chapter 7.1.2. Thus the Comparator (Figure 14a) performs the same function. However, routing has to be iterated $p \cdot d$ times, since each cell is handling more entries than the Basic Mesh Routing design. The mesh size has now decreased, but the number of iterations in routing is increased. The upper bits of a routing address of the packet are used for the comparison in the Comparator. The lower bits of the address of the packet will be used to determine column position of the result vector storage in the destination cell. When the packet reaches the destination, the intermediate result vector bits on these positions will be xored with the packet vector bits.

7.2.3 Improved Cell Architecture

In the Improved Mesh Routing design, each cell handles multiple columns of the original matrix. Each cell becomes more complex than the basic Mesh Routing Cell depicted previously in Chapter 7.1.5. p is the number of columns each cell handles. p is designed to be the power of 2, in order to efficiently handle the addressing in the computation. The design has $p=16$ to efficiently store the values in LUT RAM. Any

number smaller than 16 would still require the same storage space. The total number of memory words needed for the packets is $p \cdot d$, where d is the maximum number of non-zero entries per column of the matrix.

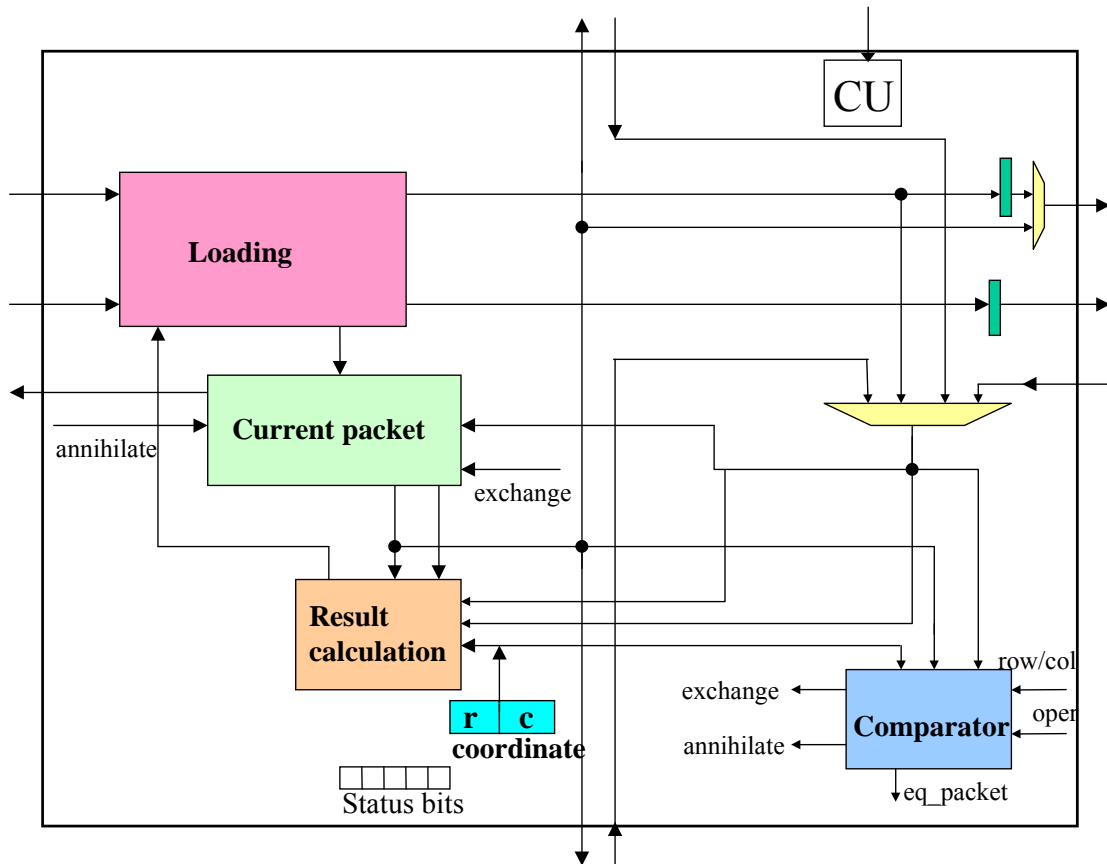


Figure 14a. Each Improved Cell

of K bits of $P[i]$ and $P'[i]$ need to be stored. Thus, it would consume a large number of flip-flops if they are used. Thus it would considerably increase the amount of resources in FPGA for the design. Instead, storage of $P[i]$ and $P'[i]$ are moved to LUT-RAMs in Loading Unit (Figure 14b) and Result Calculation Unit (Figure 14c). Each LUT-RAM can store 16 bits of data. Each CLB slice has 2 LUTs. Thus, LUT-RAMs provide compact storage for $P[i]$ and $P'[i]$.

In Loading Unit (Figure 14b), when the data packets stored in $R[i]$ and vector bits stored in $P[i]$ are loaded to the current packet register (CR) in Current Packet Unit (Figure 14d) for routing, last few bits stored in $R[i]$ are used to select vector bits stored in $P[i]$, shown as '*addr*' signal in Figure 14b.

When the packet reaches the destination, similarly the lower bits of address in the packet are used to select the position for result vectors in $P'[i]$, as shown in Figure 14c with '*addr2*' signal. In this position, the vector bits of incoming packet are xored with the intermediate result vector bits on the cell.

8. Results and Analysis

8.1 Design Process, Tools and Testing

The implemented circuits have been modeled in VHDL code. Reference implementation used to generate test vectors has been developed in C. Both codes are generic in nature. In VHDL, a mesh is created by the instantiation of the same cell multiple number of times. The design has generic parameter for mesh size and the number of vectors. The number of iterations of routing to be performed is also a parameter. Moreover, the generic capability of the VHDL language has been utilized to set up the status bits of each cell after the compilation, such that the cells are instantiated with different status values. Hence, after the compilation, each cell behaves differently according to its positions even though only one generic code has been developed for all cells.

The design is verified in Aldec-Active HDL by functional and timing simulation using test vectors generated by software written in C. The synthesis and implementation of the circuit is done using Synplicity Synplify Pro and Xilinx ISE. The target device is the Virtex II XC2V8000 and Virtex II XC2V6000 devices. The Virtex II XC2V8000 is chosen because of the largest area of the Virtex II family of FPGAs. XC2V6000 is chosen because of its use in the SRC-6e reconfigurable computer as described in Chapter 9. Test vectors have been generated in software and stored in a file. These files are used

by the VHDL testbench. The input entries for the matrix are formatted to have row and column coordinates stored in an ASCII file. These values are read by the VHDL testbench and fed into the mesh circuit. Similarly, the random input vectors are stored in a file. The VHDL testbench reads this input file and loads the vectors into the mesh. The expected result of the multiplication, generated in software, is also present in a file. Finally, when the VHDL code generates the output, the VHDL testbench compares the actual output generated by the circuit with the expected output stored in a file, and displays the result of comparison.

8.2 Results for Basic Mesh Routing Design and Analysis

8.2.1 Area and Latency

For Basic Mesh Routing Design, the row and column indices for the packets in the mesh are four bits long. This is because the implemented mesh size is 12x12, which is the maximum size of the mesh that can fit on the Virtex II FPGA(XC2V8000) device with $K=70$. K represents the number of vectors multiplied simultaneously. Thus, this mesh can perform matrix-by-vector multiplication of a sparse matrix of the size 144x144 with 70 vectors, each of the size 144x1 in one iteration of the Mesh Routing algorithm. Each packet has one status bit, four bits for representing the row coordinate and 4 bits for representing the column coordinate. These packets together with the vector bits need to be downloaded to the circuit for each sub-computation.

The density of ones in each column of the matrix A (which is obtained after the Sieving step for 512-bit factorization) is about 63 when the matrix has 6.7×10^6 columns

[10]. This matrix is preprocessed to have uniform distribution of ones. The matrix is divided into s^2 sub-matrices. The maximum density (d) per column for each sub-matrix is calculated to be one.

Table 1: Synthesis results for the Basic Mesh Routing design in Virtex II FPGA, XC2V8000

| Matrix Size | K | CLB slices | LUTs | FFs | Clock Period (ns) | Time for K mult (ns) | Time per 1 mult (ns) |
|-------------------------|----|-----------------|-----------------|-----------------|-------------------|----------------------|----------------------|
| 144x144 (Mesh 12x12) | 1 | 8123 (17%) | 15,495 (16%) | 5,318 (5%) | 14 | 672 | 672 |
| 144x144 (Mesh 12x12) | 32 | 23,949 (51%) | 46,944 (50%) | 23,419 (25%) | 16.6 | 797 | 25 |
| 144x144 (Mesh 12x12) | 70 | 43,065 (92%) | 84,836 (91%) | 45,378 (48%) | 17.8 | 854 | 12 |

The circuit was first synthesized for $K=1$, which is the basic case for doing one matrix-by-vector multiplication, to find out the resource usage. The circuit was slightly optimized in order to reduce the high fan-out of control signals by replicating control signals. The synthesis results for different K values are shown in Table 1. Since for $K>1$, K matrix-by-vector multiplications are occurring at the same time, larger K corresponds to a shorter total execution time, provided that the circuit can fit within one FPGA device. This limit was reached for $K=70$, using Virtex II XC2V8000 FPGA device, for the mesh size of 12x12.

Increase in K increases the area usage. Particularly comparing for the cases $K=1$ and $K=70$, it can be noted that there is an increase of more than five times for CLB slices and LUTs and a nine times increase in FF resource usage. In this case, the limiting factor is the amount of LUTs, which implement the combinational logic of the circuit.

Each routing needs $4 \cdot m$ clock cycles to ensure that all the values are routed completely. Hence, for doing one round of matrix-by-vector multiplication of 144×144 matrix (with maximum non-zero entries being 1 and mesh dimension $m=12$) with the 70 vectors of 144×1 , it takes about $1 \cdot 4 \cdot 12$ clock cycles. This translates to 854 ns. Since multiple matrix-by-vector multiplications are done at the same time, the time per multiplication becomes 12 ns.

The practical implementation results provide the understanding of how the circuit resources are utilized. The theoretical calculations do not provide the complete picture for the final results. The logic needed for control and the complete circuit functioning has to be taken into account. Only then, the total resource usage and timing of the circuit are estimated accurately.

The reference software results are obtained on a WinXP platform, Pentium IV, 2.768 GHz machine with 1GB memory. The time for one matrix-by-vector multiplication (one vector being multiplied) is obtained to be 3440 ns. The speedup of the hardware implementation vs. software implementation is about 282 as shown in Table 2.

Table 2: Time comparison for sparse matrix multiplication of size 144x144 in optimized software and in Virtex II FPGA

| Matrix Size | Multiplication Time in SW (ns) | Multiplication Time in HW (ns) | Speedup |
|----------------------------|--------------------------------|--------------------------------|---------|
| 144x144 (Mesh 12x12) | 3440 | 12 | 282 |

8.2.2 512-bit and 1024-bit Performance Estimation and Cost

Using distributed approach as proposed by Geiselmann and Steinwandt [5], the larger matrix-by-vector multiplication can be broken down into smaller matrix-by-vector multiplications and the results can be combined together to get the final result. However, instead of using all the chips and doing all the computations at once, I obtain the performance measures for limited resources of FPGA chips particularly for the case of 1 chip , 10^2 chips, 16^2 chips, 32^2 chips connected in parallel. Multiple chips are connected together in two dimensions with IO pins running at high frequency. Virtex II chip has a total of 1104 IO pins with 276 pins on each of the four sides. Since the connection bus size between cells becomes more than the number of pins available, time multiplexing is performed for each data exchange. Thus, each compare-exchange operation and loading between cells takes multiple clock cycles when considering multiple chips with one mesh instantiation. I am particularly interested in knowing how the speedup behaves for multiple chips as opposed to having multiple CPUs do the multiple computations in parallel.

I take the case for 512-bit factorization. Table 3 shows the result of this calculation based on the practical implementation results obtained for one Virtex II chip. D is the number of columns in the matrix obtained after the Sieving step for the 512-bit factorization. The mesh dimension is $m \times m$. Since multiple sub-multiplications have to be done for one large matrix-by-vector multiplication, n represents the number of such multiplications that needs to be done. The original matrix A from the Sieving step has a size of $D \times D$. The mesh of size $m \times m$ will handle the sub-matrix of size $m^2 \times m^2$. So, the total number of sub-matrix computations needed is calculated as $n = D^2/(m^2)^2$. The Matrix step needs about $3D/K$ multiplications for the block Wiedemann algorithm [16]. Thus the total time (T_{Total}) for the Matrix step is $3 \cdot (D/K) \cdot n \cdot (\text{Time for one mesh computation \& loading-unloading time})$.

Table 3: Time estimates for the Matrix step of factoring 512 bit number with one Virtex II chip, XC2V8000 and multiple Virtex II chips in Basic Mesh Routing

K = number of parallel matrix vector multiplications = 70
 D = number of columns in matrix A
 m = mesh dimension
 n = number of times to repeat sub-multiplications
 T_K = time for K multiplications in the mesh
 T_{Load} = time for loading and unloading for K multiplications
 T_{Total} = total time for a Matrix step = $3 \cdot (D/K) \cdot n \cdot (T_K + T_{Load})$

| Virtex II chips | D | m | n | T_K (ns) | T_{Load} (ns) | T_{Total} (days) | Speed up vs. 1 chip |
|------------------------|-------------------|----------|-------------------|---------------------------|------------------------------|---------------------------------|----------------------------|
| 1 | 6.7×10^6 | 12 | 2.1×10^9 | 854 | 113 | 6971 | 1 |
| 10^2 | 6.7×10^6 | 120 | 2.1×10^5 | 68352 | 18136 | 61.6 | 113 |
| 16^2 | 6.7×10^6 | 192 | 33,032 | 109560 | 29036 | 15.2 | 459 |
| 32^2 | 6.7×10^6 | 384 | 2,064 | 219120 | 58904 | 1.92 | 3630 |

For multiple Virtex chips, the chips are assumed to be connected in two dimensions. For instance, for the case of 10^2 Virtex II chips, there is a 10×10 array of chips and the single mesh of size 120×120 is spread over these 100 Virtex II chips. The time estimation for this case is extrapolated from the basic time for one Virtex II chip. Since there are 276 pins on one side of FPGA, and there are 12 cells connections in both directions needing size of about 87 bits per cell, it takes 8 clock cycles for each compare-exchange operation. The limited IO pins has increased the computation and loading and unloading time in multiple chips. The slowdown factor (h) is 8.

For doing sub-computations, the contents of the sub-matrix have to be loaded on to the chip together with the sub-vectors. The maximum possible loading and unloading

bus sizes are taken into consideration to calculate the loading and unloading time for the maximum number of IO pins that can be utilized in the Virtex II chips. The loading and unloading time has to be taken into account for the calculation of the total time. The frequency of loading circuit is assumed to be clocked at 200 MHz, taking into account the loading shift circuit has very few logic gates involved in the critical path. The maximum number of I/O pins available at one side, out of four sides of Virtex II, is taken into account to calculate the bus size used for loading and unloading for the case of multiple chips connected in two dimensions. For one chip, the number of IO pins available can be obtained from four sides as it is not connected to other chips. For one chip, half of the total IO pins, is used for input and half for output.

Let t be the number of bits representing row and column coordinates of the packet. The status bit requires one bit. Let b be the available I/O pin size. K is the number of multiple vectors handled. Each packet is of the size $(1+4\cdot t)$ bits. Since there are a total of d non-zero entries in a column of sub-matrix and each cell stores d non-zero packets, there are a total of $d\cdot m^2$ packets that need to be loaded and K vectors of size m^2 . Thus it takes $((d\cdot(1+4\cdot t)+K) \cdot m^2) / b$ clock cycles to load the packets and vector bits simultaneously. T_{Load} represents the time for loading and unloading for one mesh computation. T_K is the time for one mesh computation. From this, the total time for the matrix step is calculated.

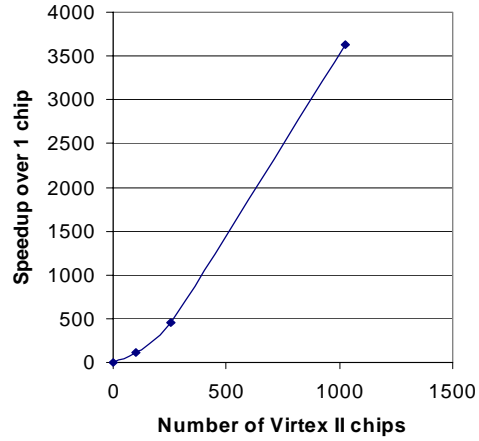


Figure 15: Speedup for multiple Virtex II chips connected in a mesh

The speedup increases in more than linear fashion as shown in Figure 15. Actually, the speedup increases by about $(\text{number of chips})^{3/2} / h$ (where h is a slowdown factor). This can be seen from equation 14 for the total time for multiple chips. m_1 is the mesh size for one chip. The mesh dimension is increased by a factor of $\sqrt{\#\text{chip}}$. Thus, the routing time is $d \cdot 4 \cdot m_1 \cdot \sqrt{\#\text{chip}}$ and number of sub-computations (n) is $D^2 / (m_1 \cdot \sqrt{\#\text{chip}})^4$.

$$T_{\text{Total}} = 3 \frac{D}{K} \frac{D^2}{(m_1 \cdot \sqrt{\#\text{chip}})^4} \cdot (d \cdot 4 \cdot m_1 \cdot \sqrt{\#\text{chip}} + T_{\text{load}}) \quad (14)$$

If the separate chips are used to do separate sub-matrix multiplications, then the time only decreases linearly with the number of chips.

The same combination of Virtex II devices with the same basic synthesis results for one Virtex II chip is now used to estimate the performance for 1024-bit Matrix step as shown in Table 4. For 1024-bit, using the 'small matrix' from [10] after Sieving step, the

number of columns of matrix is about $D = 4 \times 10^7$. The density in each column of the matrix A of 1024 bit is estimated to be about 100 [10]. The maximum density per column for each sub-matrix is calculated to be one using preprocessing of the matrix .

Due to the larger matrix size, the number of sub-multiplications, n , increases now in comparison to the 512-bit case. The time for multiplication and time to load are calculated in the same manner as 512-bit. As seen with 512 bit case, the speedup rises with the rate of about $(\text{number of chips})^{3/2} / h$ in also this case. $h = 8$ for this design. With $32^2 = 1024$ Virtex II chips, the Matrix step for 1024 bit numbers can be done in 408 days.

Table 4: Time estimates for the Matrix step of factoring 1024 bit numbers with one Virtex II chip and multiple Virtex II chips in Basic Mesh Routing

K = number of parallel matrix vector multiplications = 70

D = number of columns in matrix A

m = mesh dimension

n = number of times to repeat sub-multiplications

T_K = time for K multiplications in the mesh

T_{Load} = time for loading and unloading for K multiplications

T_{Total} = total time for a Matrix step = $3 \cdot (D/K) \cdot n \cdot (T_K + T_{Load})$

| Virtex II chips | D | m | n | T_K (ns) | T_{Load} (ns) | T_{Total} (days) | Speedup vs. 1 chip |
|------------------------|-----------------|----------|----------------------|------------------------------|-----------------------------------|--------------------------------------|---------------------------|
| 1 | 4×10^7 | 12 | 7.7×10^{10} | 854 | 113 | 1.4×10^6 | 1 |
| 10^2 | 4×10^7 | 120 | 7.7×10^6 | 68352 | 18136 | 13224 | 105.8 |
| 16^2 | 4×10^7 | 192 | 1.2×10^6 | 109560 | 29036 | 3208 | 436 |
| 32^2 | 4×10^7 | 384 | 73,586 | 219120 | 58904 | 408 | 3431 |

8.3 Results for Improved Mesh Routing Design and Analysis

8.3.1 Area and Latency

For Improved Mesh Routing, each cell handles the non-zero entries from p columns of the original matrix A . For the same FPGA chip, Virtex II XC2V8000, using the maximum area, it is now possible to fit a larger sub-matrix. Using $p=16$ and the mesh size of 12×12 , we can fit a mesh with $K=50$ as shown on Table 5. The area used for the CLB slices is 93%. Here, because of the additional storage for p vectors in each cell, K cannot become as large as the value obtained for the Basic Mesh Routing Circuit of $K=70$. But the sub-matrix now being multiplied is of size 2304×2304 , since the mesh handles $12 \cdot 12 \cdot 16 = 2304$ columns. This sub-matrix is multiplied with 50 vectors of size 2304×1 in one iteration of Mesh Routing. After synthesis, the clock period for this circuit is 17.7 ns. Now, for doing one round of matrix-by-vector multiplication of 2304×2304 matrix with the 50 vectors of 2304×1 , it takes about $16 \cdot 1 \cdot 4 \cdot 12$ clock cycles, which translates to about 13593 ns. Time per multiplication becomes 271 ns.

Table 5 : Synthesis results for the Improved Mesh Routing design in Virtex II FPGA

| Matrix Size | K | CLB slices | LUTs | FFs | Period (ns) | Time for K mult (ns) | Time per 1 mult (ns) |
|------------------------------------|----|-----------------|-----------------|-----------------|-------------|----------------------|----------------------|
| 2304x2304 (Mesh 12x12, p=16) | 1 | 6738 (14%) | 10,438 (11%) | 6,279 (7%) | 14.5 | 11136 | 11136 |
| 2304x2304 (Mesh 12x12, p=16) | 32 | 29,938 (64%) | 50,983 (54%) | 19,651 (21%) | 16.7 | 12826 | 401 |
| 2304x2304 (Mesh 12x12, p=16) | 50 | 43,402 (93%) | 74,030 (89%) | 27,406 (29%) | 17.7 | 13593 | 271 |

8.3.2 512-bit and 1024-bit Performance Estimation and Cost

Each packet has one status bit, $4t-2r$ bits for loading address and routing address where t is the minimum bit number representation for column number and r is $\log p$. These packets together with the vector bits need to be downloaded to the circuit for each sub-computation.

The time estimates for 512-bit is calculated as previously for the case of Basic Mesh Routing circuit and shown in Table 6. Now, each Mesh Routing circuit is handling $m^2 \cdot p$ columns of the sub-matrix, which is different than m^2 columns in Basic Mesh Routing Circuit. Thus, the number of sub-computations needed for one large matrix-vector multiplication is $n = D^2 / (m^4 \cdot p^2)$.

Table 6: Time estimates for the Matrix step of factoring 512 bit numbers with one Virtex II chip and multiple Virtex II chips in Improved Mesh Routing

K = number of concurrent multiplications = 50
 p = number of columns handled in one cell = 16
 D = number of columns in matrix A
 m = mesh dimension
 n = number of times to repeat sub-multiplications
 T_K = time for K multiplications in the mesh
 T_{Load} = time for loading and unloading for K multiplications
 T_{Total} = total time for a Matrix step = $3 \cdot (D/K) \cdot n \cdot (T_K + T_{Load})$

| Virtex II chips | D | m | n | T_K (ns) | T_{Load} (ns) | T_{Total} (days) | Speedup vs. 1 chip |
|-----------------|-------------------|-----|-------------------|-------------------|-------------------|--------------------|--------------------|
| 1 | 6.7×10^6 | 12 | 8.4×10^6 | 13593 | 1568 | 593 | 1 |
| 10^2 | 6.7×10^6 | 120 | 846 | 8×10^5 | 2.1×10^5 | 4 | 148 |
| 16^2 | 6.7×10^6 | 192 | 129 | 1.3×10^6 | 3.8×10^5 | 0.96 | 617 |
| 32^2 | 6.7×10^6 | 384 | 8 | 2.6×10^6 | 9×10^5 | 0.132 | 4492 |

The time necessary for K sub-multiplications is obtained as $T_K = p \cdot d \cdot 4 \cdot m \cdot \text{clock period}$, where d = density of non-zero entry in each column, p = number of columns of A handled in one cell of mesh, m = mesh size in one dimension. The loading and unloading time is obtained as previously for Basic Mesh Routing design accounting more entries now of $d \cdot m^2 \cdot p$ packet entries and $m^2 \cdot p \cdot K$ vector bits for the mesh. For multiple chips circuit, the slowdown due to limited IO pins is 6 times on routing, and loading and unloading. Thus, the slowdown factor $h = 6$.

The total time is calculated as $3 \cdot (D/K) \cdot (D^2 / (m^4 \cdot p^2)) \cdot (d \cdot p \cdot 4 \cdot m \cdot \text{Clock period} + T_{Load})$. The result reported in [7] for the Matrix step of the factorization of a 512 bit number, is 224 CPU hours (9.3 days) of a Cray C916, using the block Lanczos algorithm

to achieve the same goal of finding linear dependencies. It can be seen that for only 32^2 (1024) FPGA chips, this step can be done in 0.132 days (3.2 hours) from Table 6 by the Improved Mesh Routing design. The time estimate for 1024-bit is shown in Table 7. With only 1024 Virtex II chips, the Matrix step for 1024-bit can be done in 27 days.

Table 7 : Time estimates for the Matrix step of factoring 1024 bit numbers with one Virtex II chip and multiple Virtex II chips in Improved Mesh Routing

K= number of concurrent multiplications=50
 p=number of columns handled in one cell=16
 D = number of columns in matrix A
 m = mesh dimension
 n = number of times to repeat sub-multiplications
 T_K = time for K multiplications in the mesh
 T_{Load} = time for loading and unloading for K multiplications
 T_{Total} = total time for a Matrix step = $3 \cdot D / K \cdot n \cdot (T_K + T_{Load})$

| Virtex II chips | D | m | n | T_K (ns) | T_{Load} (ns) | T_{Total} (days) | Speed up vs. 1 chip |
|------------------------|-----------------|----------|-------------------|------------------------------|-----------------------------------|--------------------------------------|----------------------------|
| 1 | 4×10^7 | 12 | 3.0×10^8 | 13593 | 1568 | 126851 | 1 |
| 10^2 | 4×10^7 | 120 | 3.0×10^4 | 8×10^5 | 2.1×10^5 | 864 | 147 |
| 16^2 | 4×10^7 | 192 | 4599 | 1.3×10^6 | 3.8×10^5 | 210 | 604 |
| 32^2 | 4×10^7 | 384 | 287 | 2.6×10^6 | 9×10^5 | 27 | 4698 |

8.4 Comparison of Basic Mesh Routing and Improved Mesh Routing Implementations

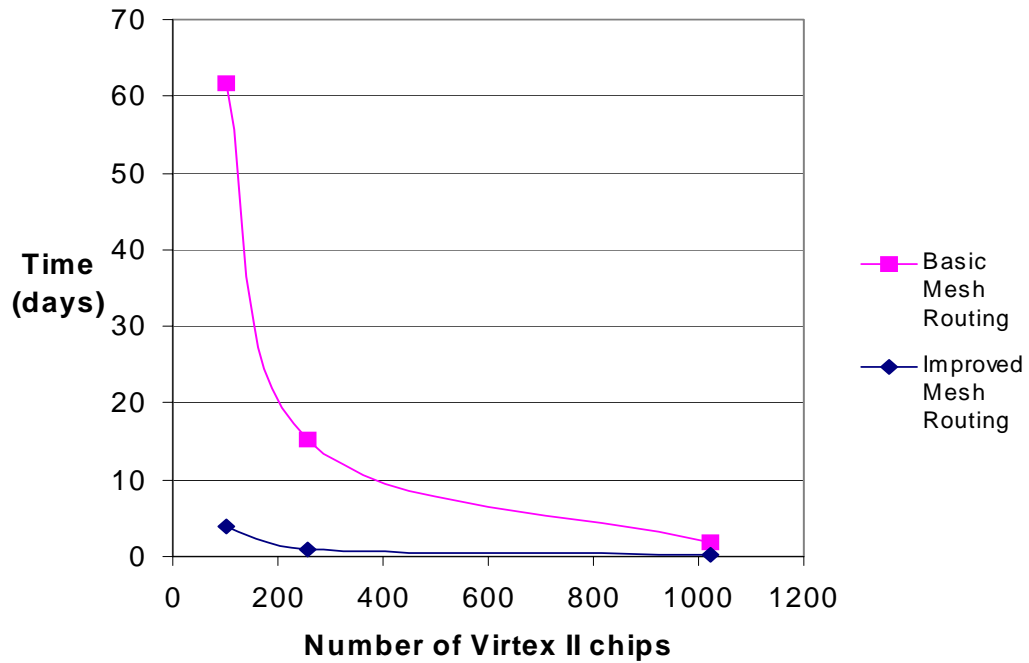


Figure 16. Comparison of Basic and Improved Mesh Routing for 512-bit factorization

Figure 16 shows the comparison for the time estimates for Basic Mesh Routing and Improved Mesh Routing designs for the 512-bit Matrix step. Figure 17 shows the comparison for 1024-bit Matrix step.

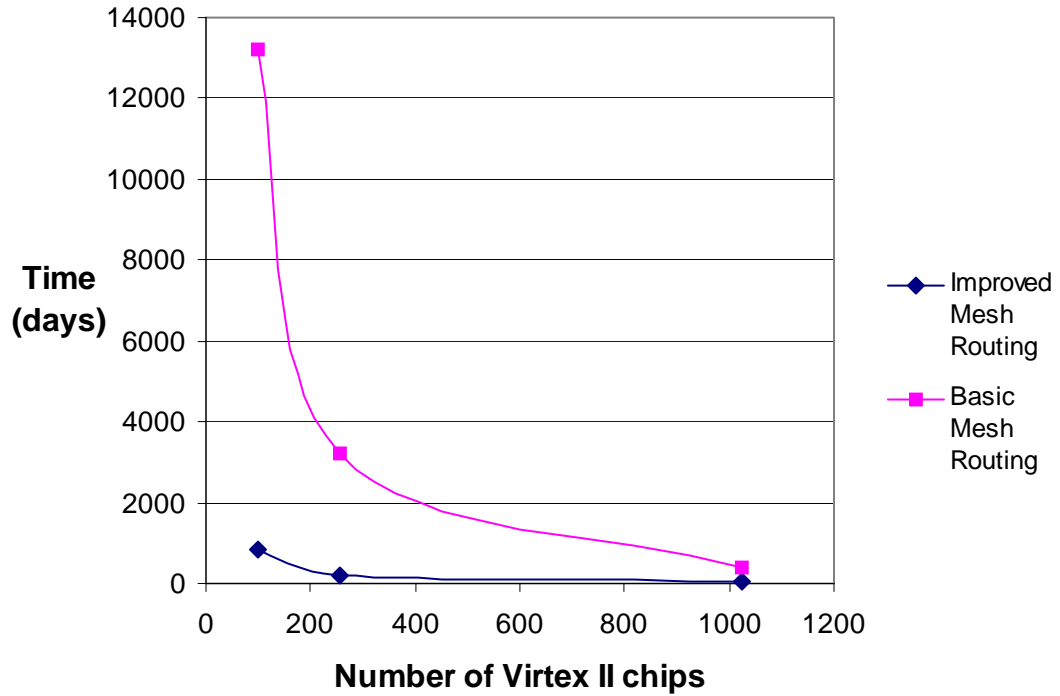


Figure 17. Comparison of Basic and Improved Mesh Routing for 1024-bit factorization

The formula for the total time for Improved Mesh Routing is,

$$3 \frac{D}{K} \frac{D^2}{(m^4 \cdot p^2)} \cdot (d \cdot 4 \cdot m \cdot p + T_{load}) \quad (15)$$

where T_{load} represents both loading and unloading time. In Improved Mesh Routing, as multiple column entries of the original matrix can be stored in one cell, one fixed size chip can handle larger matrix size with only slight increase in area. This is because other combinational and control units of each cell do not need to be duplicated for each column of the original matrix A . Since the sub-matrix handled in the Improved Mesh Routing circuit has become larger than in the Basic Mesh Routing circuit, the number of sub-

multiplications ($n = D^2/(m^4 \cdot p^2)$), that has to be done for one complete matrix-by-vector multiplication of 512-bit or 1024-bit size, becomes much smaller. The decrease in the time for one matrix-by-vector multiplication is by about p , besides some other effect of loading and unloading time when m is same for both designs as seen from equation 15. Since $p=16$, this is faster than the increase in the time due to smaller K value, where K changes from 70 to 50 from Basic to Improved Mesh Routing design. Hence, the Improved Mesh Routing design, which contains entries of multiple columns in one cell, performs better than the Basic Mesh Routing design. This can be seen on comparing Basic Mesh Routing with Improved Mesh Routing results in Tables 3,4 and Tables 6,7.

Figure 18 shows the speedup ratio of Improved Mesh Routing design to Basic Mesh Routing design for different number of FPGAs obtained from the Tables 3,4 and Tables 6,7. The speedup is seen to be around a factor of 11 to 15. For one Virtex II XC2V8000 chip, the speedup of Improved Mesh Routing design to Basic Mesh Routing design is about 11. For multiple Virtex II XC2V8000 chips, the speedup of Improved Mesh Routing design to Basic Mesh Routing design is about 15. The speedups are different because of the different slowdown factors on multiple chips for Improved Mesh Routing design and Basic Mesh Routing design. This is due to the fact that the two designs have different number of IO connections required between cells of the mesh in separate FPGAs.

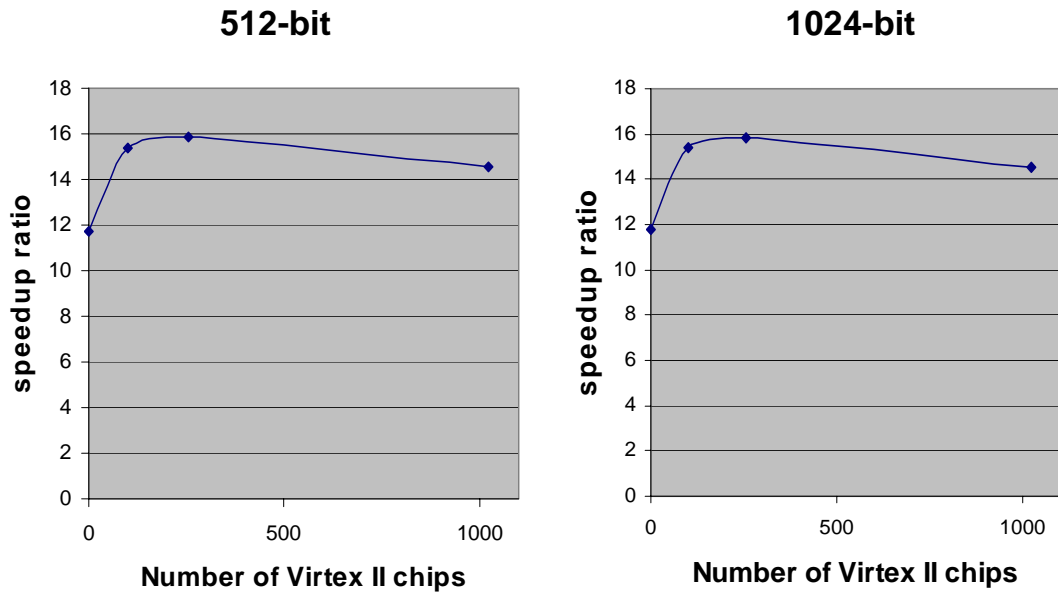


Figure 18. Speedup of Improved to Basic Mesh Routing vs. Number of Virtex II FPGAs

9. SRC Computing Platform

9.1 Hardware Architecture

P3 – Intel Pentium 3 Microprocessor
 L2 – Level 2 Cache
 MIOC – Memory and I/O Bridge Controller
 PCI – Peripheral Component Interconnect Interface
 DDR Interface – Double Data Rate Memory Interface
 SNAP – SRC-developed Memory Interconnect
 MAP - Reconfigurable Processor

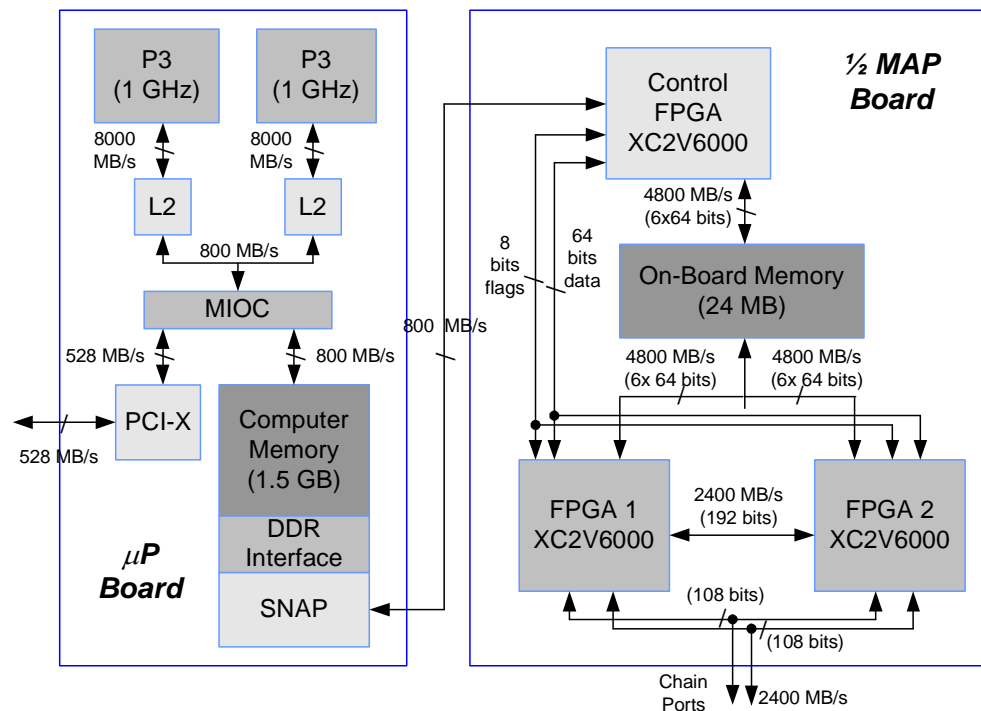


Figure 19 . Hardware architecture of SRC-6e

SRC-6e is a hybrid-architecture platform, which consists of two double-processor boards and one Multi-Adaptive Processor (MAPTM) board [14]. A block diagram depicting a half of the SRC-6e machine is shown in Figure 19. The MAP board includes two User FPGAs, On-Board Memory, and one Control FPGA. All FPGAs included on the MAP board are Xilinx Virtex II XC2V6000 FPGA devices. Each processor board is connected to the MAP board through the so-called SNAP card. A SNAP card plugs into two DIMM slots on the microprocessor motherboard and can support a peak bandwidth of 800 MB/s.

9.2 Programming Model

The SRC-6e has a similar compilation process as a conventional microprocessor-based computing system, but needs to support additional tasks in order to produce logic for the MAP reconfigurable processor, as shown in Figure 20. There are two types of the application source files to be compiled. Source files of the first type are compiled targeting execution on the Intel platform. Source files of the second type are

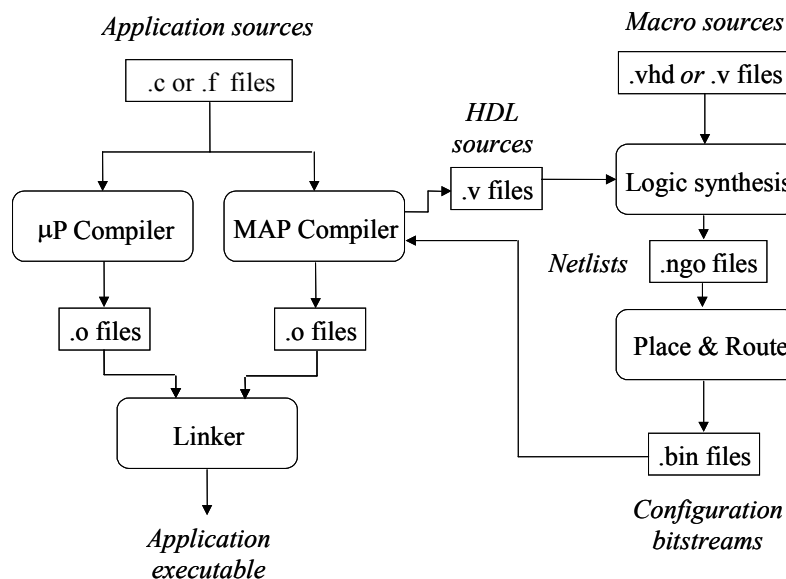


Figure 20 . Compilation process of SRC-6e

compiled targeting execution on the MAP board. A file that contains a program to be executed on the Intel processor is compiled using the microprocessor compiler to produce an object (.o) file. All files containing functions that call hardware macros and thus execute on the MAP are compiled by the MAP C compiler or MAP FORTRAN compiler. These compilers produce several relocatable object files (.o), corresponding to respective subroutines. Object files resulting from both the Intel and MAP compilation steps are then linked with the MAP libraries into a single executable file. The resulting binary file may then be executed on the SRC-6E Intel and MAP hardware.

MAP source files contain MAP functions composed of macro calls. Here, macro is defined as a piece of hardware logic designed to implement a certain function. Macros are created in VHDL or Verilog and are integrated into the compilation process of SRC.

SRC-compiler invokes third party tools to compile the hardware logic by using Synplify & Xilinx ISE for synthesis and place and route. The macro is invoked from within the C or FORTRAN subroutine by means of a subroutine call.

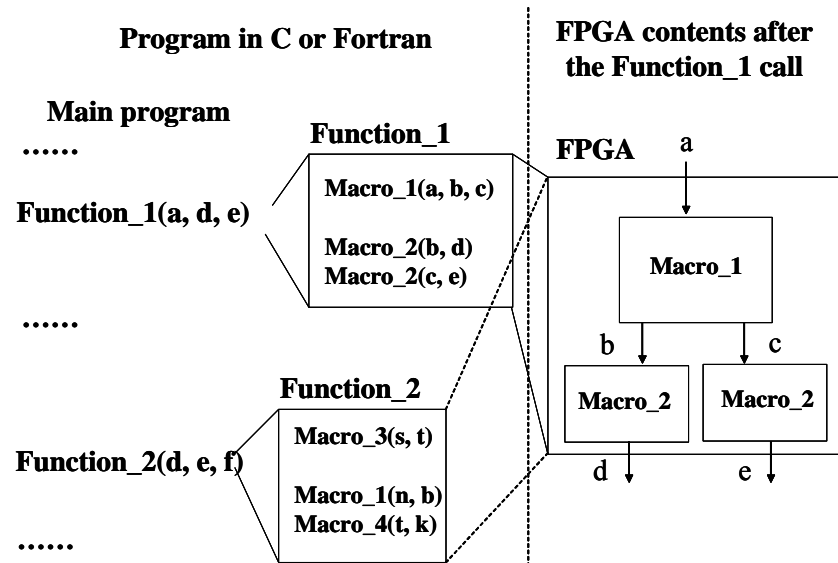


Figure 21. Programming model of SRC-6e

Figure 21 shows the mapping between macro calls and the corresponding contents of a MAP FPGA. Note that Macro 2, called twice in Subroutine 1, results in two instantiations of the logic block representing Macro 2. Values of arguments in the macro calls determine interconnects between macro instantiations in hardware. The contents of each MAP function in software determine the configuration of the entire FPGA device in hardware. Each time a new MAP function is called, the contents of the entire FPGA changes.

9.3 SRC Restrictions

The only clock frequency supported by the SRC system for the hardware macros is 100 MHz. The memory interface and control timings of Control FPGA work at 100 MHz, so user macro should have the same frequency. Henceforth, the previous designs had to be modified, since they had a frequency below 100 MHz. The necessary modifications include adding registers in the critical paths and improving fanouts of logic components. The control circuit needed to be modified to account for these changes. After such modifications, the VHDL code could be executed on the SRC machine.

Currently, the SRC compiler optimizes speed over the area for the user designs. As a result, any C-compiled hardware has a relatively large area compared to the design described entirely in VHDL.

10. Implementation on SRC

10.1 Design scheme

Any implementation running on the SRC-6e machine can be described using three different design entries as a

- a) C function running on the microprocessor
- b) MAP-C code compiled by the SRC-compiler to the hardware code running on the User FPGAs
- c) VHDL macro compiled to the hardware code running on the User FPGAs.

Different parts of the design can be partitioned to run on the microprocessor or FPGA based on the performance needs. For the design to run on the FPGA, the design entry can be done in C or as a macro in VHDL. The description in C is similar to an algorithmic model of the problem and does not require the knowledge of hardware. A user does not need to devote much effort to the design of the control circuit since the compiler handles most of the control. However, using C, it is harder to express the parallelism inherent in many algorithms. C leads also to a less efficient hardware code compared to pure VHDL code. Moreover, hardware described in C takes more area due to the addition of multiple registers and the delay chains by compiler needed to synchronize macros.

The Mesh Routing design involves a lot of parallelism in the communication among distributed cells. The compare-exchange operations between all adjacent cells happen at the same time. Considering this nature of my design, I have developed a SRC macro corresponding to the entire mesh as shown in Figure 22. I call this design the SRC-Mesh design. I also implemented the second design in which only a single cell is described in VHDL code as a macro and the connection of cells and the control are done by the MAP C, as shown in Figure 23. I call this design the SRC-Cells design. In this design, extra care needs to be taken in order to not imply sequential execution of cells. The mesh is described in MAP C, which is then compiled by the SRC compiler to the hardware running on the User FPGA.

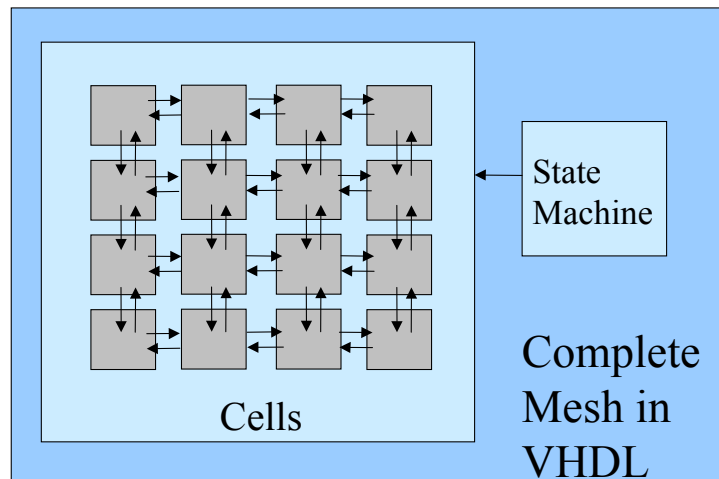


Figure 22. SRC-Mesh Design

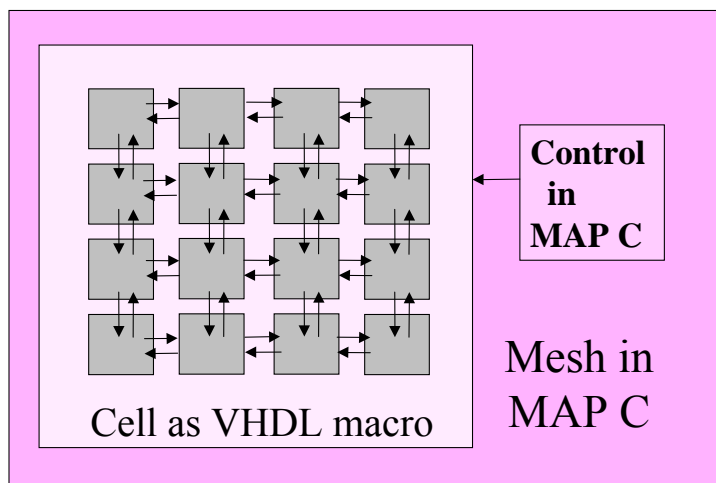


Figure 23. SRC-Cells Design

10.2 SRC-Mesh Design

In SRC-Mesh, the complete mesh is in VHDL. This macro is called in the MAP C routine, in a for loop, with transfer of data to the mesh from the OBM memory, then computation, and the transfer of the data out from the mesh to the OBM memory. The microprocessor side calls this MAP C routine, for the execution in FPGA, and transfers the input and output data by DMA.

The original Mesh Routing design in VHDL had to be modified since it did not satisfy the 10 ns clock period limitation. In order to make these modifications, the previous designs had to be analyzed for all critical paths and high fanout nets. I reduced the high fanouts by adding registers to replicate the control signals. Also, in the data path inside the cell, I added registers to increase the frequency. So now each compare-exchange operation takes two clock cycles of 10 ns, instead of one clock cycle as in the

original design described in Chapter 7. The modified cell design is shown in Figure 24 which shows addition of register R1. The enable signals to registers and select signals to multiplexers are also registered.

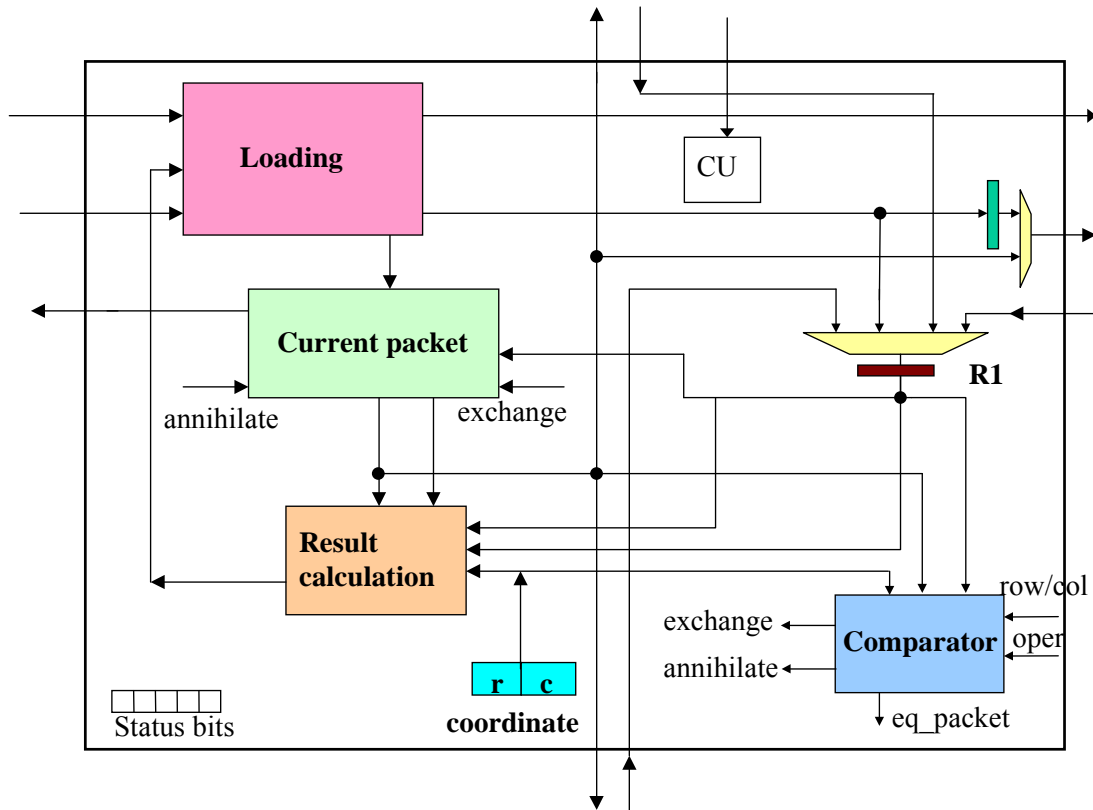


Figure 24. Architecture of a basic cell in the SRC-Mesh design

The similar modifications are done for the Improved Mesh Routing SRC-Mesh design. For some designs with larger K , I had to add more registers in the datapath inside the Comparator and other parallel paths in order to meet the target frequency. In such design, each compare-exchange operation takes three clock cycles of 10ns, increasing the total latency of the design.

10.3 SRC-Cells Design

In the SRC-Cells design, only the basic cells are described in VHDL, and the mesh is described in MAP C. The m^2 cells are instantiated with the appropriate input and output parameters. The exchange of the data between cells is done by variable assignments. Due to a large number of cells and data parameters, I developed a C program to generate this MAP C code.

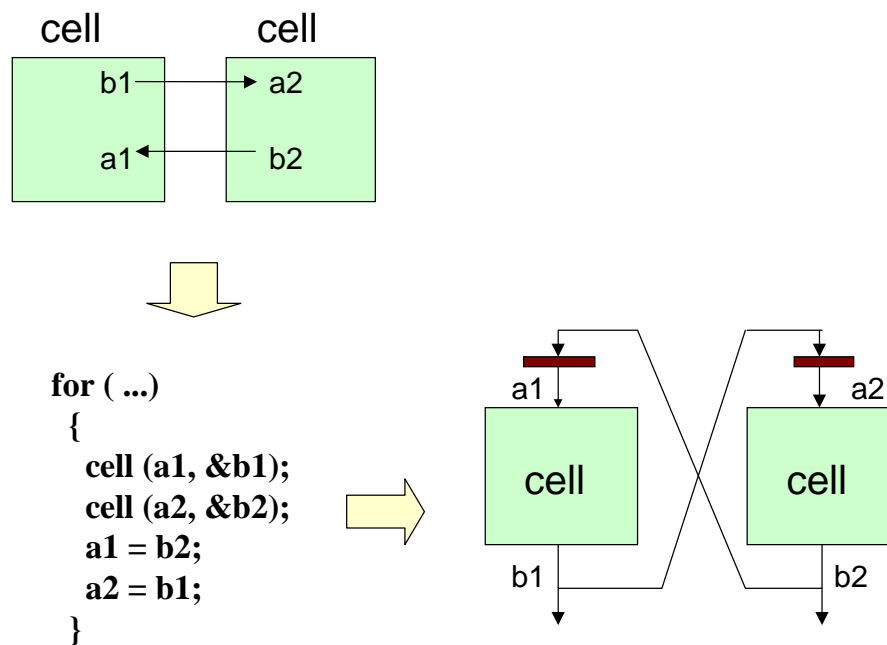


Figure 25. Cell instantiation with circulation of output to input

Certain consideration has to be made for the parameters of the cells in the macro calls in order not to imply sequential execution in the SRC compilation. The sample C code format needed to be written in C is shown in Figure 25 to perform the function of

compare-exchange operation for two adjacent cells in one direction. For a mesh of 10x10, 100 cells with 8 parameters each are instantiated and about 400 variable assignments are done. As shown in Figure 25, for the connection between cells, there is one extra register added by the SRC compiler before the macros input when circulating output to input.

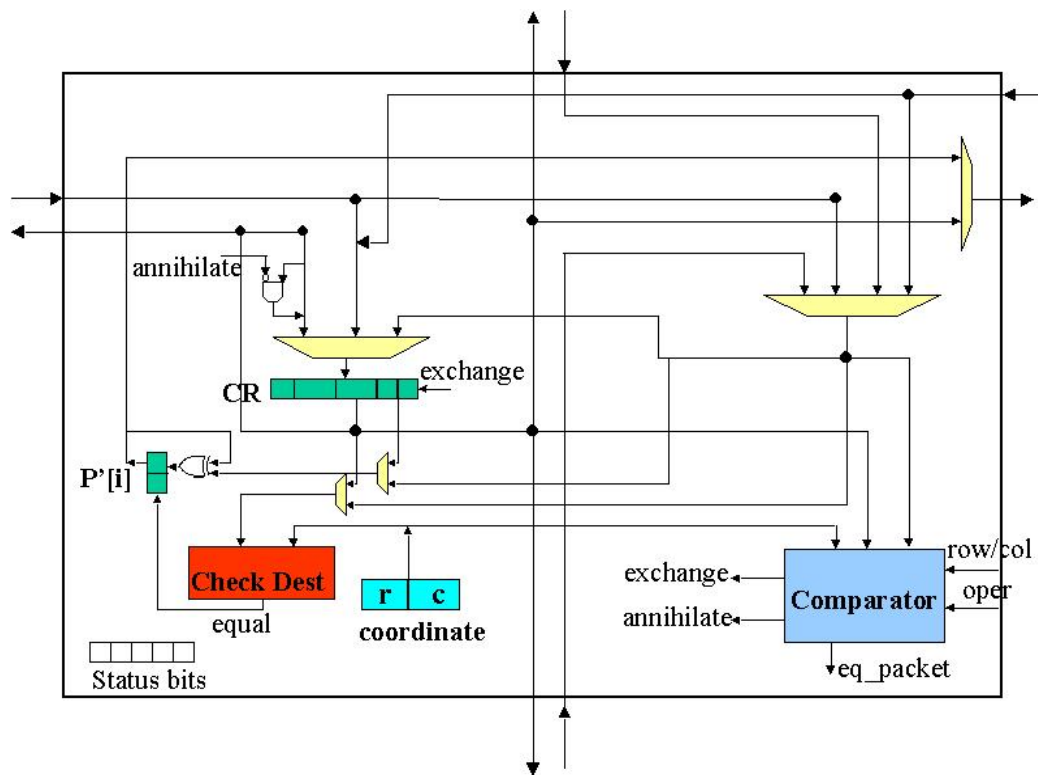


Figure 26. Cell structure of the SRC-Cells design

The cell for the SRC-Cells design is shown in Figure 26. In SRC-Cells the loading of initial values to external registers is done outside the cells. All the cells are loaded with values in external registers at the same clock cycle. The cell structure is

modified accordingly by modifying control signals inside the cells. This design also needs to be modified for improving the frequency by adding registers inside the Comparator. The input for the vectors is done through the right input, and the input for packets through the left input. Similarly, the output result vector is multiplexed to the right output.

11. Results on SRC and Analysis

SRC-6e includes Virtex II XC2V6000 FPGAs which are smaller than Virtex II XC2V8000 FPGAs used in the previous study described in Chapter 8. Virtex II XC2V6000 has a capacity of 33,792 CLB slices, which include 67,584 LUTs and 67,584 FFs. Virtex II XC2V8000 has a capacity of 46,592 CLB slices, which include 93,184 LUTs and 93,184 FFs.

11.1 Design Process, Tools and Testing

All implemented circuits have been redesigned for the execution on the SRC-6e machine. All designs for the SRC-6e are optimized for 100 MHz frequency. The codes to be executed on SRC are partially written in C. The C code is compiled by the SRC compiler to the corresponding hardware code. The execution time in FPGAs has been measured with the hardware timer macro provided as a function of the standard SRC library. The designs are tested on the platform with the reference software implementation test vectors.

11.2 Results for Basic Mesh Routing Design

As before, m is the mesh size, K is the number of vectors being multiplied concurrently. Let t be the number of bits to represent the address in the mesh. Each

packet has one status bit, t bits for a row address and t bits for a column address. As before, the density of non-zero entries, d , is one. The areas for different SRC Basic Mesh Routing designs are shown in Table 8. The results for performance are shown in Table 9.

Table 8. Area for SRC Basic Mesh Routing designs

| Design Type | Mesh Size | K | CLB slices | LUTs | FFs |
|--------------------|------------------------------|----------|-------------------|-----------------|-----------------|
| SRC-Mesh | 12x12 (Matrix 144x144) | 1 | 11,981 (35%) | 18,364 (27%) | 12,619 (18%) |
| SRC Mesh | 12x12 (Matrix 144x144) | 42 | 30,743 (90%) | 54,166 (80%) | 43,545 (64%) |
| SRC-Mesh | 20x20 (Matrix 400x400) | 1 | 31,533 (93%) | 54,691 (80%) | 28,636 (42%) |
| SRC-Mesh | 10x10 (Matrix 100x100) | 70 | 31,566 (93%) | 55,528 (82%) | 46,647 (69%) |
| SRC-Cells | 11x11 (Matrix 121x121) | 1 | 32,814 (97%) | 29,959 (44%) | 47,759 (70%) |

Table 9. Performance for SRC Basic Mesh Routing designs

D = number of columns in matrix A

m = mesh dimension

K = number of vectors being multiplied concurrently

n = number of times to repeat sub-multiplications = $D^2/(m^4)$

x = clock cycles per compare-exchange

T_{Kroute} = time for K multiplications in the mesh = $d \cdot 4 \cdot m \cdot x \cdot \text{period}$

T_{KTot} = time for K multiplications, including loading, unloading in SRC 6e

$T_{512 \text{ Compute}}$ = total computational time for a 512-bit Matrix step

$T_{512 \text{ Total}}$ = total time for a 512-bit Matrix step = $3 \cdot (D/K) \cdot n \cdot (T_{KTot})$

| Design Type | Mesh Size | K | Period (ns) | x | T_{Kroute} | T_{KTot} | $T_{512 \text{ Compute}}$ (days) | $T_{512 \text{ Total}}$ (days) |
|-------------|------------------------------|----|-------------|---|--------------|------------|-----------------------------------|---------------------------------|
| SRC-Mesh | 12x12 (Matrix 144x144) | 1 | 10 | 2 | 960 ns | 1270 ns | 483,516 | 639,410 |
| SRC-Mesh | 12x12 (Matrix 144x144) | 42 | 10 | 2 | 960 ns | 1870 ns | 11,512 | 22,416 |
| SRC-Mesh | 20x20 (Matrix 400x400) | 1 | 10 | 2 | 1600 ns | 2270 ns | 104,222 | 147,865 |
| SRC-Mesh | 10x10 (Matrix 100x100) | 70 | 10 | 2 | 800 ns | 1870 ns | 11,938 | 27, 898 |
| SRC-Cells | 11x11 (Matrix 121x121) | 1 | 10 | 3 | 1320 ns | 1610 ns | 939,676 | 1,146,120 |

x is the number of clock cycles needed for one compare-exchange operation.

T_{Kroute} is the time required for the routing phase of K multiplications in the mesh. This is equal to $4 \cdot m \cdot x \cdot \text{clock period}$. T_{KTot} is the total time for K multiplications, including loading, unloading, routing and control time.

A mesh of the size $m \times m$ will handle the sub-matrix of the size $m^2 \times m^2$. Thus, the total number of sub-matrix computations needed is $n = D^2/(m^2)^2$. The Matrix step requires $3D/K$ multiplications for the block Wiedemann algorithm [16]. Thus, the total time for the Matrix step is $3 \cdot (D/K) \cdot n \cdot T_{Ktot}$. $T_{512} Total$ denotes the total time for the Matrix step of a 512-bit number. $T_{512} Compute$ denotes the total computational time for a 512-bit Matrix step. It is obtained by $3 \cdot (D/K) \cdot n \cdot T_{Kroute}$.

SRC has six OBM banks with 64-bits width each for either input or output from the FPGA. Thus, there is a total of 384-bits for input and output that can be done for loading and unloading. The parallel loading and unloading of rows has been used, utilizing the IO bandwidth allowed by the OBM banks. For large values of K , the input packet size becomes large so the packets cannot be sent to all of the rows in parallel. Thus, limited parallel loading and unloading, for instance loading to every two rows or every three rows at a time is performed.

For the SRC-Mesh design, the 12x12 mesh with $K=1$ takes 35% of the CLB slices. Increasing K from 1 to 42 increases the number of CLB slices to about 90%. Both cases need two clock cycles for a compare-exchange operation. For the $K=1$ case, the total time for K multiplications is 1270 ns, which is a slight rise from the routing time of 960 ns. For the $K=42$ case, the total time rises by large amount to 1870 ns. It can be seen that designs with large K have a longer loading and unloading time. Since the packet size increases and there is a fixed IO bandwidth, it takes more clock cycles to load more bits into and out of the mesh.

In order to find the optimum values of parameters m and K , I have experimented with different values of these two parameters leading to a circuit that can fit in one chip using about 90% of CLB slices. Increasing to full load will adversely affect the clock timing due to the routing bottleneck. Large m will have the advantage that the number of sub-multiplications, n , will be reduced. Large K will reduce the total computational time by a factor of K , as K multiplications are done in parallel.

For $m=12$ and $K=42$, 22,416 days are required for a 512 bit Matrix step. The number of CLB slices used is 90%. For $m=20$ and $K=1$, the circuit takes 93% of the CLB resources. Its total time for a 512 bit Matrix step is equal to 147,865 days, which is larger than for the mesh of 12x12 with $K=42$. Increasing K increases the area of the mesh slowly in a linear fashion, since the combinational logic for comparator and storage of the data packets are not affected. As a result, large values of K can be used. However, when m is increased, the area increases proportionally to m^2 and as a result, large values of m cannot be used. Comparing the mesh of 12x12 with $K=42$ with the mesh of 20x20 with $K=1$, the effect of performance gain due to the increase in K is larger than the performance loss due to the smaller value of m . Therefore, the mesh of 12x12 with $K=42$ outperforms the mesh of 20x20 with $K=1$. Further, I have decreased the value of m and increase the value of K to get the mesh of 10x10 with $K=70$ with around same area. The total end-to-end time is comparable with the mesh of 12x12 with $K=42$. The mesh of 12x12 with $K=42$ is slightly faster.

The optimum choice of m and K is influenced by the dependence of the circuit area on m and K . Area grows proportional to m^2 and linear to K . The graphical

comparison of the mesh performance for the three different mesh sizes and values of K is shown in Figure 27. From Figure 27, for implementation on FPGAs, the mesh size of 12x12 with $K=42$ is the optimum choice for the basic Mesh Routing design. This choice is particularly dependent on the architecture implementation and the platform.

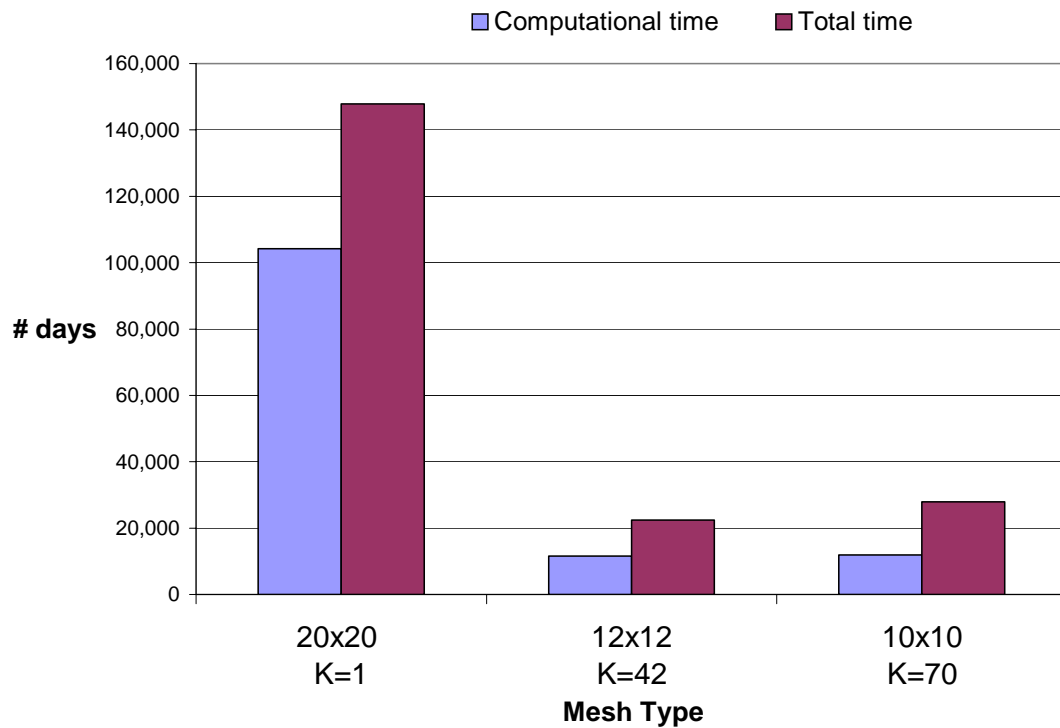


Figure 27. Comparison of performance of different mesh sizes and K for a 512-bit matrix

SRC-Cells design, where only a cell macro is defined in VHDL and the entire mesh in MAP C code, takes a lot of registers and LUTs. Even the mesh of 12x12 with $K=1$ does not fit in a single Virtex II FPGA. SRC-Cells design of the mesh of 11x11 with $K=1$ takes about 97% of the CLB slices. Its performance is worse than SRC-Mesh design

because it takes 3 clocks to do one compare-exchange and the size of K can only be up to one due to large area requirements. Moreover, m is also smaller. The benefit is that much of the design is in high level language to provide the ease of programming.

Table 10. Comparison of SRC-Mesh and SRC-Cells for the same mesh parameters

| Design Type | Mesh Size | K | CLB slices | LUTs | FFs | Period (ns) | x | T _{Kroute} |
|-------------|------------------------------|---|----------------|-----------------|----------------|-------------|---|---------------------|
| SRC-Cells | 10x10 (Matrix 100x100) | 1 | 25325 (74%) | 22056 (33%) | 36042 (53%) | 10 | 3 | 1200 ns |
| SRC-Mesh | 10x10 (Matrix 100x100) | 1 | 9,347 (27%) | 13,427 (19%) | 10439 (15%) | 10 | 2 | 800 ns |

To compare the SRC-Cells and SRC-Mesh designs with the same values of mesh parameters, I implemented the mesh of 10x10 cells with $K=1$ for both designs as shown in Table 10. The design for SRC-Cells gives much larger area requirements than SRC-Mesh as shown in Table 10 and Figure 28. The increase in the total number of CLB slices is 2.7 times. Much of the rise is in the Flip-Flop (FF) resources due to many instantiations of registers from the variables used in MAP C code. Currently, SRC compiler optimizes performance over the area. Area optimizations will improve as the compiler matures.

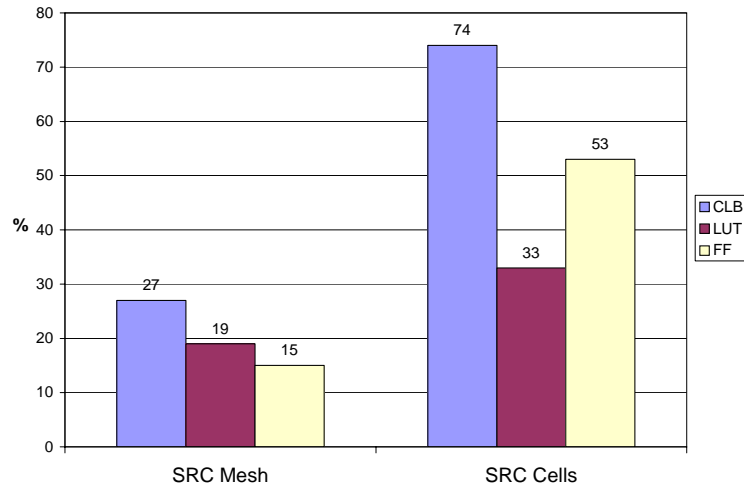


Figure 28. Comparison of area of the SRC-Mesh and the SRC-Cells of 10x10

11.3 Results for Improved Mesh Routing Design

The SRC-Mesh Improved Mesh Routing design has also been redesigned in order to meet the frequency requirement of the SRC-6e machine. In Improved Mesh Routing, each cell is handling p multiple columns. The Improved Mesh Routing Design handles $m^2 \cdot p$ columns of the sub-matrix. Hence, it does matrix-by-vector multiplications for the matrix size of $m^2 \cdot p \times m^2 \cdot p$. As a result, it effectively handles larger matrix size than the basic design. The area resources for different Improved SRC-Mesh designs are shown in Table 11. The area for the SRC-Mesh of 10x10 with $p=16$ and $K=1$ is about 40% of CLB slices. Increasing K to 32 takes about 91%. Again, smaller mesh size with large K is explored with equivalent amount of area resources. For this, I can get the mesh of 8x8 with increasing K up to 64.

Table 11 . Area resources for SRC Improved Mesh Routing designs

| Design Type | Mesh Size /w p =16 | K | CLB slices | LUTs | FFs |
|-------------------|--------------------------|----|--------------|--------------|--------------|
| Improved SRC-Mesh | 10x10 (matrix 1600x1600) | 1 | 13,577 (40%) | 20,929 (30%) | 13,513 (19%) |
| Improved SRC-Mesh | 10x10 (matrix 1600x1600) | 32 | 31,002 (91%) | 51,905 (76%) | 29,954 (44%) |
| Improved SRC-Mesh | 8x8 (matrix 1024x1024) | 64 | 31,456 (93%) | 53,060 (78%) | 31,082 (45%) |

The performance measures for the Improved SRC-Mesh designs are shown in Table 12. The time for routing is $T_{Kroute} = p \cdot d \cdot 4 \cdot m \cdot x \cdot \text{clock period}$, where d = density of non-zero entries in each column, p =number of columns of the matrix A handled in one cell of mesh, m = mesh size in one dimension and x = number of clock cycles per compare-exchange operation. T_{Ktot} is the total time for K multiplications including loading, unloading, routing and control time. The loading and unloading time is performed using maximum possible IO bandwidth to six banks of OBM in the SRC-6e machine. $T_{512} \text{ Compute}$ is a total computational time for a 512-bit Matrix step. $T_{512} \text{ Total}$ is the total end-to-end time for a 512-bit Matrix step including, computation, loading, unloading and control. The total time is calculated as $3 \cdot (D/K) \cdot (D^2/(m^4 \cdot p^2)) \cdot T_{KTot}$.

Table 12. Performance for SRC Improved Mesh Routing designs

D = number of columns in matrix A

m = mesh dimension

p = number of columns of A handled in one cell

K = number of vectors being multiplied concurrently

n = number of times to repeat multiplications = $D^2/(m^2p)^2$

x = clock cycles per exchange

T_{Kroute} = routing time for K multiplications in the mesh = $d \cdot 4 \cdot m \cdot x \cdot \text{clock period}$

T_{KTot} = total time for K multiplications including loading and unloading in SRC-6e

$T_{512 \text{ Compute}}$ = total computational time for a 512-bit Matrix step

$T_{512 \text{ Total}}$ = total end-to-end time for a 512-bit Matrix step = $3 \cdot (D/K) \cdot n \cdot (T_{KTot})$

| Design Type | Mesh Size | K | Period (ns) | x | T_{Kroute} (ns) | T_{Ktot} (ns) | $T_{512 \text{ Compute}}$ (days) | $T_{512 \text{ Total}}$ (days) |
|-------------------|-----------------------------|-----|-------------|-----|-------------------|-----------------|----------------------------------|--------------------------------|
| Improved SRC-Mesh | 10x10 (matrix 1600x1600) | 1 | 10 | 2 | 12,800 | 16,580 | 52,111 | 67,500 |
| Improved SRC-Mesh | 10x10 (Matrix 1600x1600) | 32 | 10 | 3 | 19,200 | 31,480 | 2444 | 4,013 |
| Improved SRC-Mesh | 8x8 (matrix 1024x1024) | 64 | 10 | 3 | 15,360 | 25,260 | 2414 | 3,930 |

For the Improved SRC-Mesh design with $m=10$ and $K=1$, only one level of registers needs to be inserted so that each compare-exchange operation takes two clock cycles. However, for the mesh of 10x10 with $K=32$, two register levels need to be added so that each compare-exchange operation takes 3 clock cycles. The same holds for the case of mesh of 8x8 with $K=64$. The time estimated for a 512-bit Matrix step for the Improved SRC-Mesh of 10x10 with $K=32$ is obtained to be 4013 days. For the Improved SRC-Mesh of 8x8 with $K=64$, it is 3930 days. This is faster than the best Basic SRC-

Mesh design of mesh of 12x12 with $K=42$, which is 22,416 days. The Improved SRC-Mesh design is faster than the Basic SRC-Mesh design by a factor of 5.7 .

The Improved SRC-Mesh design handles large matrix size. Each sub-multiplication takes slightly longer time proportional to $m \cdot p$, but the number of multiplications needed decreases by an order of $m^4 \cdot p^2$. Therefore, the total time reduces by a large factor overcoming the effect of a decrease in the value of K . As a result, the Improved SRC-Mesh performs better than the Basic SRC-Mesh. This improvement would even be higher if each compare-exchange operation took two clock cycles for the Improved SRC-Mesh.

12. Comparison of the SRC Designs versus Standalone FPGA Designs

SRC designs have been implemented in a smaller FPGA than standalone FPGA designs, so the designs obtained using standalone FPGAs have the advantage of fitting larger mesh size. This brings improved performance. The implementations on SRC have increased the latency of the designs because of the need to meet the frequency requirements. Thus, the standalone implementations have an advantage in computational latency over the SRC implementations. Also, SRC designs have limited input/output bandwidth, but in standalone computation, the maximum IO bandwidth of the Virtex II chip is used. For one standalone Virtex II XC2V8000, a 512-bit Matrix step can be performed in 593 days, compared to 3930 days using the SRC machine with one Virtex II XC2V6000 FPGA.

13. Conclusions

Factoring of large numbers is a problem of great practical importance. The difficulty of this problem determines the security of common public key cryptosystems (such as RSA), which are used as a basis for electronic commerce. Users of these cryptosystems need accurate assessments of the cost of integer factorization in order to select minimum secure key sizes that guarantee computational resistance against even the most powerful adversaries. Since such powerful adversaries are likely to employ hardware in their attacks, it is misleading to merely assess the cost of factorization in software using conventional general-purpose computers. On the other hand, building specialized hardware for the purpose of cost assessment is too expensive and inflexible.

In this thesis, I have moved a step closer to a realistic estimate of the difficulty of factoring in reconfigurable hardware for practical sizes of numbers used in cryptography. The Mesh Routing architecture has been selected as the best architecture for the Matrix Step. This architecture has been practically implemented for the first time. The architecture has been analyzed, designed, and implemented in reconfigurable hardware, using a scalable approach. The generic architecture is developed to handle any mesh size. The constraints come from the size of the chip that can fit a particular mesh size and certain mesh parameters. The area and timing of the implementation have been determined for the state-of-the-art Xilinx Virtex II XC2V8000 and XC2V6000 FPGA

devices. The initial results of this investigation have been reported in my publication at the FPT conference [1].

I have developed two versions of the Mesh Routing design, Basic and Improved. The implementation results for both of the designs have been analyzed. It has been shown that for the implementation on the fixed number of chips of Virtex II XC2V8000, the Improved Mesh Routing Design performs better than the Basic Mesh Routing Design by a factor of 11-15 because of the large sub-matrix computation and its significant influence on the performance.

The benefit of distributed hardware computations over highly optimized software computations has been confirmed by a speedup of about 280. The applicability of the circuit for factoring 512-bit and 1024-bit numbers using an array of FPGA devices has been demonstrated. The mesh in the square grid of FPGA chips has the advantage in regards that the speedup rises by polynomial time compared to the number of chips. With only 1024 (32^2) Virtex II FPGA chips using the Improved Mesh Routing design, the Matrix Step of factorization of 512-bit number can be performed in 3.2 hours. Furthermore, with only 1024 (32^2) Virtex II FPGA chips using the Improved Mesh Routing design, the Matrix Step of factorization of 1024-bit number can be performed in 27 days.

Furthermore, I have practically implemented the Mesh Routing design using the SRC-6e reconfigurable computer. The high level design capabilities of reconfigurable computer have been explored for the case of distributed computations of factoring. As a result, the SRC-Mesh and the SRC-Cells designs have been developed, analyzed and

compared. Based on the results obtained using the current compiler version, I have found that the SRC compiler does not currently provide area optimizations necessary to fit a large mesh size on a chip for improved performance. Hence, the optimum architecture is obtained by describing the complete mesh design in VHDL instead of C. Different mesh sizes and different values of the architecture parameters have been explored. Both Basic and Improved Mesh Routing designs have been compared using the SRC-6e reconfigurable computer. Similar to before, the Improved Mesh Routing performs faster than the Basic Mesh Routing design by a factor of 5.7 in the SRC-6e implementations. The influence of the input-output bandwidth on the distributed mesh computations of the Matrix step has been explored by comparing I/O data transfer time with the FPGA computation time.

My implementations and investigations may provide a guidance to evaluate the viability of hardware attack against RSA for practical key sizes. Further, through my analysis, the concrete area and performance comparisons for two approaches in Mesh Routing design have been obtained. This may guide the future designers who want to implement the Matrix step of factoring in a large scale FPGA or ASIC. Moreover, the investigations of the implementations on a reconfigurable computer may help researchers using reconfigurable computers in evaluating the minimum amount of effort required for the Matrix step of factoring.

List of References

List of References

- [1] S. Bajracharya, D. Misra, K. Gaj, T. El-Ghazawi, "Reconfigurable Hardware Implementation of Mesh Routing in Number Field Sieve Factorization" Field Programmable Technology, Proc. FPT Conference 2004, Brisbane, Australia, Dec. 2004.
- [2] D. J. Bernstein. *Circuits for integer factorization: a proposal*.
<http://cr.yp.to/papers/nfscircuit.pdf>.
- [3] D. Coppersmith, *Modifications to the number field sieve*, Journal of Cryptology, vol 6 (1993), pp.169-180.
- [4] D. Coppersmith, *Solving homogeneous linear equations over $GF(2)$ via block Wiedemann algorithm*, Math. Comp. bf 62 (1994), pp.333-350.
- [5] W. Geiselmann, R. Steinwandt, *Hardware to solve sparse systems of linear equations over $GF(2)$* , Proc. CHES 2003, LNCS, Springer-Verlag, pp. 51-61.
- [6] <http://wissrech.iam.uni-bonn.de/main/news/RSA576.html>
- [7] A. K. Lenstra et al., *Factorization of a 512-bit RSA Modulus*, Advances in Cryptology, EUROCRYPT 2000 LNCS 1807, Springer-Verlag 2000, pp. 1-17.
- [8] A.K. Lenstra, H.W. Lenstra, Jr., (eds.), *The development of the number field sieve*, Lecture Notes in Math. 1554, Springer-Verlag 1993.
- [9] A.K. Lenstra, H.W. Lenstra, Jr., *Algorithms in number theory*, Chapter 12, *Handbook of theoretical computer science, Volume A, algorithms and complexity* (J. van Leeuwen, ed.), Elsevier, Amsterdam (1990).
- [10] A. K. Lenstra, A. Shamir, J. Tomlinson, E. Tromer, *Analysis of Bernstein's Factorization Circuit*, Proc. Asiacrypt 2002, LNCS 2501, Springer-Verlag, 2002, pp. 1-26.

- [11] A. K. Lenstra, E. Tromer, A. Shamir, W. Kortsmit, B. Dodson, J. Hughes, P. Leyland, *Factoring estimates for a 1024-bit RSA modulus*, Proc. Asiacrypt 2003, LNCS 2894, Springer-Verlag 2003, pp. 331-346.
- [12] A. Shamir, E. Tromer, *On the cost of factoring RSA-1024*, RSA CryptoBytes, vol. 6 no. 2, 2003, pp.10-19.
- [13] Robert D. Silverman, *A cost-based security analysis of symmetric and asymmetric key lengths*, Bulletin 13, RSA Security, 2000, <http://www.rsasecurity.com/rsalabs/bulletins/bulletin13.html>
- [14] SRC Inc. <http://www.srccomp.com/>
- [15] G. Villard, *Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems (extended abstract)*, International Symposium on Symbolic and Algebraic Computation, ACM Press, 1997, pp. 32-39.
- [16] D. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Transactions on Information Theory, IT-32 (1986), pp. 54-62.

CURRICULUM VITAE

Sashisu M. Bajracharya was born on October 19, 1978 in Kathmandu, Nepal. He graduated from Martyrs' Memorial High School, Kathmandu, Nepal, in 1995. His biography was published in The National Dean's List 1998-99. He received his Bachelor of Science degree from George Mason University in 2002. He received an Outstanding GMU Undergraduate Student Award in 2002. He was employed as a research assistant in Cryptography and Network Security Implementation Lab for two years.

He has published the conference paper "*Implementation of Elliptic Curve Cryptosystems over $GF(2^n)$ in Optimal Normal Basis on a Reconfigurable Computer*" in Proc. of FPL 2004. He has also published the conference paper on "*Reconfigurable Hardware Implementation of Mesh Routing in Number Field Sieve Factorization*" in Proc. of FPT 2004.