Reconfigurable Hardware Implementation of Mesh Routing in the Number Field Sieve Factorization

Sashisu Bajracharya[1], Deapesh Misra[1], Kris Gaj[1], Tarek El-Ghazawi[2]

[1]ECE Department, George Mason University
4400 University Drive, Fairfax, VA 22030, USA
[2]ECE Department, The George Washington University
801 22nd street NW, Washington DC 20052, USA
{sbajrach, dmisra, kgaj}@gmu.edu, tarek@gwu.edu

Abstract

Factorization of large numbers has been a constant source of interest in cryptanalysis. The fastest known algorithm for factoring large numbers is the Number Field Sieve (NFS). The two most time consuming phases of NFS are Sieving and Matrix Step. In this paper, we propose an efficient way of implementing the Matrix step in reconfigurable hardware. Our solution is based on the Mesh-Routing method proposed by Lenstra et al. We determine the practical size of a partial mesh that can fit in one FPGA device, Xilinx Virtex II XC2V8000. We further extrapolate the computation time for the case of a square systolic array of FPGAs for 512-bit and 1024-bit numbers' factorization. We demonstrate that for practical sizes of numbers used in cryptography, 1024 bits, the Matrix Step of factorization can be performed using 1024 Virtex II FPGAs in about 27 days.

1. Introduction

Factoring a large integer into its prime factors is one of the challenging tasks in cryptanalysis both in terms of computational complexity and implementation. The Number Field Sieve (NFS) introduced by Pollard J M in 1988, is the asymptotically fastest known algorithm for the factorization of large numbers.

The NFS algorithm consists of the following four steps:

1. Polynomial Selection
2. Sieving
3. Matrix Step
4. Square Root step

The two most time consuming steps of the NFS algorithm are the *Sieving* and the *Matrix* Step. This paper focuses on the Matrix Step. This step is used to identify a linear dependence between the entries in the sparse matrix obtained as a result of the Sieving Step. For the Matrix Step, two hardware architectures have been proposed in the literature: Mesh Sorting architecture by Bernstein [8] and Mesh Routing architecture by Lenstra et al [2]. Geiselmann and Steinwandt proposed a distributed variant of both aforementioned methods to be implemented using an array of ASIC chips [12]. We propose an implementation of the Mesh Routing architecture in reconfigurable hardware.

We believe that 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, FPGAs are much more flexible, easy to program and experiment with, and reusable compared to specialized hardware based on ASICs. Particularly in the field of factorization, reconfiguration is needed since the best factorization algorithms involve computationally intensive sequentially executed steps, such as Sieving and Matrix Step. In reconfigurable hardware, these steps can be executed using the same hardware, without any additional cost. Additionally, when the new better algorithms for factorization are developed, hardware architecture can be upgraded and reconfigurable devices re-utilized. It can also be expected that once a certain number is factored, the next higher number would be targeted, and in such a scenario it would be easy to adapt the reconfigurable hardware to factor a new larger number.

In this paper, we use the space-sharing time-multiplexing approach by which we are able to reutilize the FPGA devices in subsequent 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, we report all performance measures for a varying number of FPGA devices. Our paper presents the first concrete performance and resource measurements regarding the reconfigurable hardware architecture for the NFS Mesh Routing, as the reports to date were only theoretical in nature.

2. Mesh Routing Algorithm

The matrix step concerns with finding linear dependencies in the matrix A obtained from the Sieving Step. The linear dependencies are found using Block Wiedemann algorithm [7] [10] [9] by doing multiple matrix-by-vector multiplications of the form:

$$A \cdot v_i, A^2 \cdot v_i, \ \ldots\ , A^k \cdot v_i \qquad (1)$$

where v_i is one of the random vectors ($1 \le i \le k$) and $k \approx 2D/K$. D is the number of columns of matrix A, K is the blocking factor where either K=1 or $K \ge 32$ (and K different vectors v_i are handled simultaneously). Another random vectors u_i are selected and the sequences

$$u_i \cdot v_i\ ,\ u_i A v_i,\ \ldots\ldots\ u_i A^k \cdot v_i \qquad (2)$$

($1 \le i \le K$) are used to find the linear dependent vectors in the Block Wiedemann algorithm [9].

Each matrix-by-vector multiplication is done using the Mesh Routing circuit. Referring to Fig. 1, multiplication can be performed very efficiently by considering only the non-zero entries in the columns of the sparse matrix.

Each such column entry of the sparse matrix can be viewed as a packet which needs to be routed to its destination. The accumulation of the results with the same positions will then provide the result to the matrix multiplication. Thus, a mesh of cells is created and these packets are routed in it to their destination cells.

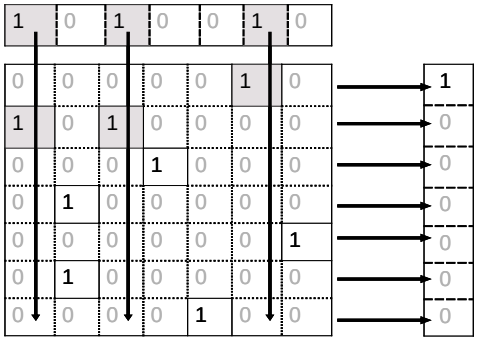

Figure 1. Matrix-vector multiplication operation through routing.

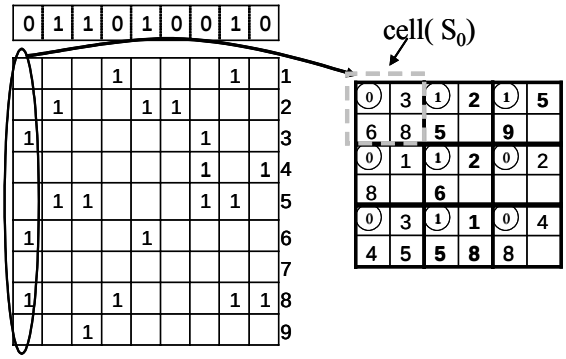

Figure 2. Mesh corresponding to the sparse matrix A.

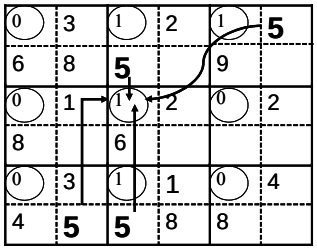

Figure 3. Routing of the packets to the cell in the mesh.

Lenstra et al proposed two versions of the routing based circuit, a simpler version and an improved routing version. The improved version is what we have implemented in hardware.

It is assumed that each of the D columns of the D*D sparse matrix A, has a weight/density of h of ones. The row and the column positions of the 'ones' in the columns are denoted by 'r' and 'c'. The vectors are of length D. The mesh has an equal number m of columns and rows, m. Sj denotes the j-th cell in the row major order, $j \in \{1,2,\ldots, (m*m)\}$. Each cell Sj is the target destination of the packet whose destination row and column indices match with the cell's row and column position. As shown in Fig. 3, all the packets to be routed to the fifth cell are routed to it.

The clockwise transposition algorithm is used for routing the individual packets to their destinations. This algorithm repeats four steps till 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 destinations of the packets in the cells are compared and packets are exchanged only if the exchange leads to the shortening of the distance of the farthest traveling packet. This compare and exchange operation is done till all the packets are routed to their destinations.

3. Implementation

3.1. Loading and Unloading

The row and column indices stored in a packet, correspond to the matrix entries which have non-zero values. Along with this routing address the loading address is also generated. These packets are loaded from the memory to the mesh as shown in Fig. 4. The loading of the vectors is done similarly, entering the mesh through the leftmost cells, and shifting from one cell to another.

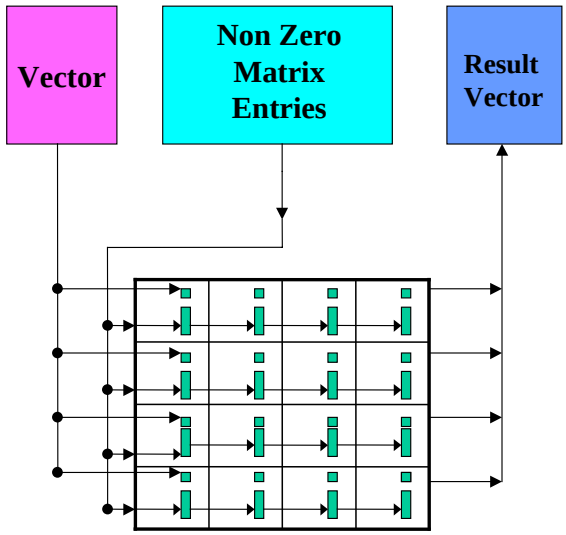

Figure 4. Loading and Unloading.

The result of the matrix-by-vector multiplication is a vector produced after completing Mesh Routing. After the computation is finished, and the result vector stored in each cell, the result vector is unloaded from the rightmost cells.

3.2. Mesh Routing Operation

The matrix-by-vector multiplication operation is done by routing each packet, together with the corresponding vector bits, to the destination cells determined by the *r* and *c* address of the packet. Whenever a packet reaches its destination, the vector bits in the packet are xored with the partial result stored in the destination 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.

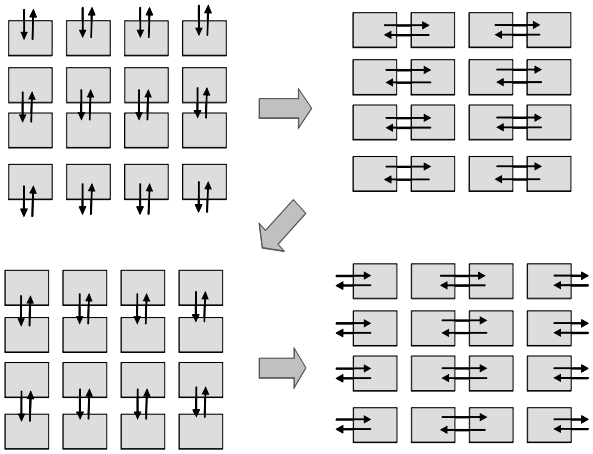

Figure 5. Four iterations of Compare-Exchange.

Clockwise transposition routing repeats four phases of compare-exchange operations. As shown in Fig. 5, in the first phase, the odd row does the compare-exchange operation with the top even row. In the second phase, the odd column does compare-exchange with the right even column. In the third phase, the odd row does compare-exchange with the bottom even row. In the fourth phase, the odd column does compare exchange with the left even column.

It can be observed that the first cell does comparisons in the clockwise order. The second cell does compare-exchange in the anticlockwise fashion. These clockwise and anticlockwise compare and exchange operations are as shown in Fig. 6.

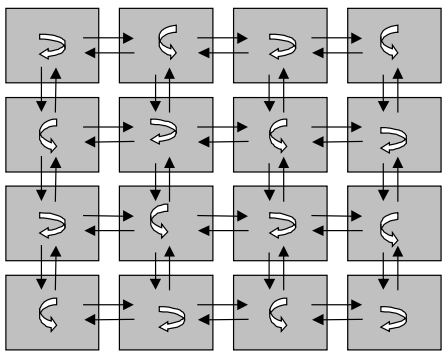

Figure 6. Compare- exchange direction for each cell.

In each compare-exchange the two neighbors send their packet to the each other and each cell independently compares the incoming packet with its packet and decides on whether to exchange by replacing its packet with the incoming packet or not to exchange by discarding the incoming packet. After reaching its destination, a packet becomes invalid. An analysis reveals that there are four cases of compare-exchanges, as follows:

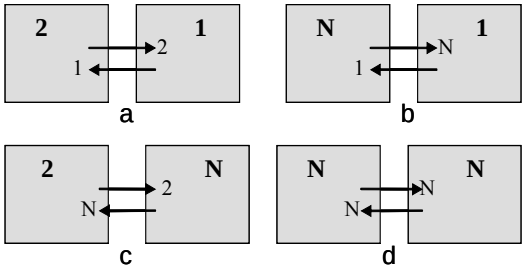

Figure 7. Compare-exchange cases.

a) Both packets are valid (Fig.7a). Thus, each cell may need to exchange the packets. Each cell decides independently by comparing the incoming packet's destination cell with the current packet's destination cell.

b) Current packet in the cell is invalid but the incoming new packet is valid (Fig. 7b). The cell may need to keep the new packet if it is traveling in the right direction.

c) Current packet in the cell is valid and the incoming new packet is invalid (Fig. 7c). 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 (Fig 7d). In this case, nothing needs to be done.

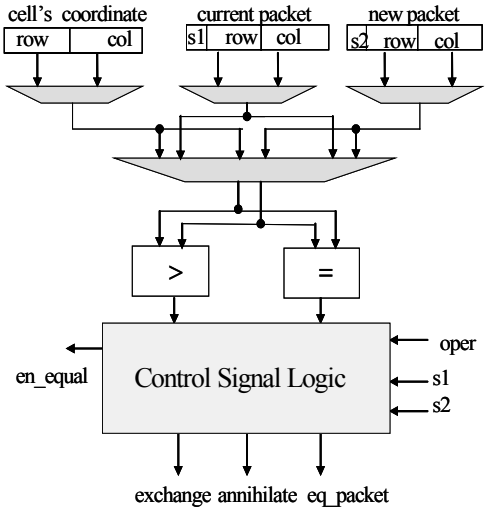

Figure 8. Comparator Unit.

The Comparator Unit is implemented in each cell as shown in Fig. 8. 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. Then the status of the current packet (s1) and the new incoming packet (s2) are used to evaluate between which of the four cases to decide the comparison upon.

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.

The circuit for each cell is shown in Fig. 9. 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 and its role of being preceding or following neighbor changes per a phase of clock. The *oper* control signal signifies 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. The P[i] registers, being a part of the Loading Unit, store 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 (implemented using LUT-RAM) used for the storage of 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 decodes if the address of loading matches the cell's address and enables the write operation to the memory.

The cell stores its coordinates in *r, c* format. The P'[i] registers, in the Result Calculation Unit, store the intermediate result vector bits after each routing. When a packet reaches the destination, the new vector bits are xored with the intermediate result bits stored in P'[i]. 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 Comparator Unit generates three control signals. The *annihilate* signal flips the status bit of the packet if annihilation needs to be done. The *exchange* signal enables loading to the register for the current packet register, CR. The *eq_packet* control signal is utilized when the current packet and the new packet have the same destination to reduce congestion.

Each cell has status bits which are constants set during synthesis based on the cell's coordinates. Some status bits signify odd or even row or column, and others signify whether the cell is at the end of mesh. Also, there are status bits to signify whether the comparison starts from top or bottom and direction of compare-exchange for each cell (clockwise/*anticlockwise*). The action performed by each 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 using lesser than or greater than relation are

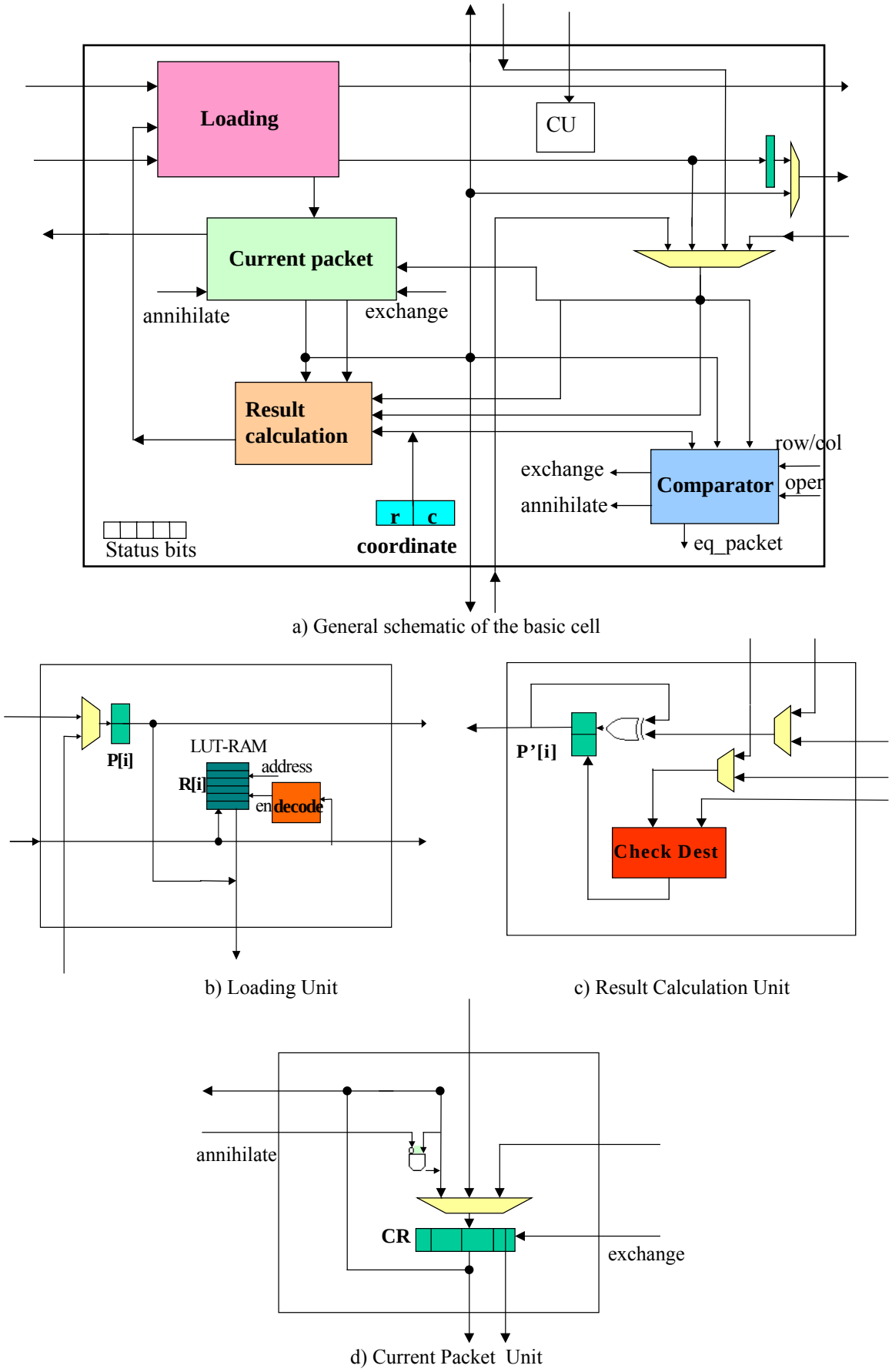

Figure 9. Detailed architecture of the Basic Cell.

determined by these status bits and the phase of iteration. Additionally, there are external control signals distributed to each cell to command on certain operation of loading, computing and unloading.

In the Improved Mesh Routing Design, each cell handles multiple columns of the original matrix. The general schematic of the basic cell, depicted in Fig. 9a remains the same, however the block diagrams of the component units become more complicated. The impact of this change on the circuit area is reduced by moving storage from registers (based on flip-flops) to LUT (look-up-table) RAMs available in Virtex FPGAs.

3.3 Sub-Matrix Computation

Since the particular hardware device of fixed size cannot perform the huge matrix-by-vector multiplication, the computation has to be divided into sub-computations composed of multiplications of smaller sub-matrices with parts of the input vector, as proposed in [12]. This way, the same device can be utilized to do sub-computations one after another, with the number of repetitions dependent on how many devices are available and affordable. The rectangular matrix A from the sieving step is assumed to have been preprocessed to have a uniform distribution of non-zero entries in each column. The matrix A is split into s*s sub matrices $A_{i,j}$ of the same size as shown below.

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} A_{1,1}v_1 + A_{1,2}v_2 + A_{1,3}v_3 \\ A_{2,1}v_1 + A_{2,2}v_2 + A_{2,3}v_3 \\ A_{3,1}v_1 + A_{3,2}v_2 + A_{3,3}v_3 \end{pmatrix}$$

Similarly, the vector v_j is also subdivided into r sub-vectors. Then, the final result A*v can be obtained as shown in equation (3).

$$A \cdot v = \begin{pmatrix} \sum_{j=1}^{s} A_{1,j} \cdot v_j \\ \vdots \\ \sum_{j=1}^{s} A_{s,j} \cdot v_j \end{pmatrix} \quad (3)$$

If only a certain number of FPGAs are available, we need to load the contents of sub-matrices $A_{i,j}$ of the mesh into the chip together with sub-vectors v_j. Maximum number of I/O pins available in the chip is used to load the inputs and unload the outputs for faster processing time. After the computation is over the results are unloaded.

4. Methodology and testing

The design is developed in VHDL code and the testing code is written in C in order to generate test vectors. The design is verified using Aldec Active HDL simulation environment. The synthesis of the circuit is done using Synplicity Synplify Pro, and mapping, placing, and routing using Xilinx ISE. The target FPGA device is Virtex II XC2V8000.

5. Results

Based on the analysis of multiple sets of design parameters, the following values of parameters have been found optimum from the point of view of minimizing the total execution time, using a fixed amount of computational resources available in the Virtex II XC2V8000 FPGA device:

- Mesh size, m x m = 12 x 12 cells
- Number of vectors v_i multiplied simultaneously by the same matrix A, K=50
- Number of matrix columns handled by one mesh cell, p=16.

With these parameters, 93% of the CLB resources of the FPGA device have been utilized. Our design can be used for a multiplication of a 2304 x 2304 sparse matrix, by 50 different 2304 x 1 vectors (please note that $2304 = m \cdot m \cdot p = 12 \cdot 12 \cdot 16$).

The density in each column of the matrix A (which is obtained after the sieving step for a 512-bit factorization) is about 63 when the matrix has $D=6.7 \times 10^6$ columns [2]. This matrix is preprocessed to have uniform distribution of non-zero entries. The matrix is divided into sub-matrices of the size 2304 x 2304. The maximum density per column for each sub-matrix thus turns out to be 1, as 63 ones are uniformly distributed among a very large number of sub-matrices (D/2304). Hence, d (density of the input submatrix) is assumed to be equal to 1.

The resource usage and results of timing analysis after synthesis, placing, and routing, are shown in Table 1 for different values of parameter K. Since for K>1, multiple matrix-by-vector multiplications are performed in parallel, larger values of K correspond to a shorter time per single matrix-by-vector multiplication. K=50 is the largest value of K for which the circuit still fits in the Virtex II XC2V8000 FPGA.

Using distributed approach proposed by Geiselmann and Steinwandt [12], the larger matrix-by-vector multiplication can be broken down into a sequence of smaller matrix-by-vector multiplications, and the partial results can be combined together to get the final result.

Table 1. Results for the Improved Mesh Routing Design in Virtex II 8000 FPGA.

Matrix Size	K	CLB	LUT	FF	Period (ns)	Time for K mult (ns)	Time for 1 mult (ns)
2304 x 2304 (Mesh 12x12, p=16)	1	6,738 (14%)	10,438 (11%)	6,279 (7%)	14.5	11,136	11136
2304 x 2304 (Mesh 12x12, p=16)	32	29,938 (64%)	50,983 (54%)	19,651 (21%)	16.7	12,826	401
2304 x 2304 (Mesh 12x12, p=16)	50	43,402 (93%)	74,030 (89%)	27,406 (29%)	17.7	13,593	271

In practice, only limited amount of FPGA devices is typically available for the implementation of the entire operation. Below, we estimate the total execution time of the Matrix Step under the assumption that the number of available Virtex II 8000 FPGAs is equal to 1, 10^2, 16^2, and 32^2, respectively. We assume that all FPGAs are connected into a generic rectangular array.

We also assume that the matrix-by-vector multiplications dominate the total execution time of the Matrix Step, and the remaining operations of the block Wiedemann algorithm can be performed in either software or hardware in a much shorter amount of time.

We consider two cases corresponding to the size of a factored number equal to 512 bits and 1024 bits respectively.

For the case of factoring a 512-bit number, Table 2 shows the results of our estimations, based on practical implementation results obtained for a single Virtex II 8000 FPGA. D is the number of columns in the matrix obtained after the sieving step. The mesh dimension is m x m. n is the number of sub-matrix by sub-vector multiplications necessary to perform the entire operation, as described in [12].

The matrix A from the sieving step has the size of D x D, where D=6.7 x 10^6. The mesh of the size m x m can handle the sub-matrix of the size $p{\cdot}m^2$ x $p{\cdot}m^2$, where p=16. Thus, the total number of sub-matrix computations required to perform a single matrix-by-vector multiplication is equal to $n = D^2/(p{\cdot}m^2)^2$. The matrix step needs about 3D/K multiplications for the block Wiedemann algorithm [2]. Thus, the total time

Table 2. Time estimates for the Matrix Step of factoring of a 512-bit number 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 the 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)
1	6.7 x 10^6	12	8.4 x 10^6	13593	1568	593
10^2	6.7 x 10^6	120	846	8 x 10^5	2.1 x 10^5	4
16^2	6.7 x 10^6	192	129	1.3 x10^6	3.8 x10^5	0.96
32^2	6.7 x 10^6	384	8	2.6 x10^6	9 x10^5	0.13

for the matrix step is equal to (3D/K) * n * Time for one mesh computation & loading-unloading time.

For comparison, the results reported in [1] for the factorization of a 512 bit number, are 224 CPU hours (9.3 days) of a Cray C916, using the block Lanczos algorithm to achieve the same goal of finding linear dependencies. As shown in Table 2, the same task can be accomplished using only 32^2 = 1024 FPGA devices in 0.13 days = 3.2 hours, which corresponds to the speed-up by a factor of 70.

For doing sub-computations, the contents of the submatrix have to be loaded to the FPGAs together with the sub-vectors. The loading and unloading scheme described in Section 3.1 is used to calculate the loading and unloading time. We also take into account the maximum possible number of input/output pins that can be utilized in the Virtex II FPGAs.

The partial result vectors are unloaded infrequently, since the accumulation of intermediate results involves only an xor operation and in majority of cases can be done inside of the circuit. All loading and unloading operations are taken into account for the calculation of the total time. The loading circuit is assumed to be clocked at 200 MHz.

Let k be the number of bits used for representing row and column coordinates of the packet. The status of each cell can be represented using one bit. Let b be the number of available I/O

pins. Each packet is of the size (1+2*k) bits. Since there are a total of d non-zero entries in the column of sub-matrix and each cell stores d·p non-zero packets, there are a total of $m^2 \cdot d \cdot p$ packets that need to be loaded and K vectors of the size $p \cdot m^2$. Thus, it takes (1+2*k+K) * $p \cdot m^2$ * d / b clock cycles to load all packets and vector bits.

The execution time estimates for factoring 1024 bit numbers using different number of Virtex II FPGAs are shown in Table 3. The most significant result is that the Matrix Step for a 1024-bit number can be performed in 27 days using 1024 Virtex II 8000 FPGAs.

The practical implementation results provide the improved understanding of the amount of FPGA resources required for the Mesh Routing Design, and how these resources are utilized. Apart from the data path, also resources needed for control, data storage, input/output, and routing are taken into account.

Table 3. Time estimates for the Matrix Step of factoring of a 1024-bit number 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 the 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)
1	4×10^7	12	3.0×10^8	13,593	1,568	126,851
10^2	4×10^7	120	3.0×10^4	8×10^5	2.1×10^5	864
16^2	4×10^7	192	4599	1.3×10^6	3.8×10^5	210
32^2	4×10^7	384	287	2.6×10^6	9×10^5	27

6. Using general-purpose reconfigurable supercomputers for factoring of large numbers

During the last few years, a considerable effort has been devoted to the development of general-purpose reconfigurable computers, machines that are based on the close interoperation of traditional microprocessors and FPGAs, and can be programmed using traditional high-level programming languages [15]. Several prototype machines of this kind have been developed including

- SRC 6E from SRC Computers Inc. [16],
- Cray XD1 (formerly OctigaBay 12K) from Cray Inc. [17]
- SGI Altix 3000 from Silicon Graphics, Inc., [18], and
- Star Bridge Hypercomputer from Star Bridge Systems Inc. [19].

These machines have demonstrated speed-ups vs. state-of-the art PCs exceeding 1000 for selected computationally intensive applications, such as DES breaking [20], and Elliptic Curve Cryptography [21].

Reconfigurable Supercomputers support all major features of the specialized ASIC-based hardware, such as parallel processing, distributed memory, specialized functional units (including multiple precision arithmetic units), flexible size and number of registers and buses, high-speed data transfer and embedded memory access. At the same time they eliminate majority of disadvantages of the specialized machines, such as long time to the solution, high non-recurring costs, fixed architecture, and the need for highly trained hardware designers.

Reconfigurable computers are much more flexible, easy to program and experiment with, and reusable compared to specialized ASIC-based hardware. At least in principle, reconfigurable computers can be programmed by mathematicians themselves, assuming that sufficiently versatile library of basic cells has been earlier developed by hardware designers.

Compared to a generic array of FPGAs, the reconfigurable computers offer much greater flexibility, the close integration of microprocessors and FPGAs, ease of software/hardware co-design, and ease of programming resulting from the use of traditional programming languages. In particular, the purchase of general-purpose reconfigurable computing platform can be better justified by its use in the wide spectrum of applications, often not related to cryptography.

Our group has over two years of experience with developing cryptographic libraries and applications for two emerging reconfigurable supercomputers, SRC 6E and Starbridge Hypercomputer.

Most recently, an initial attempt has been made to port our implementation of the Matrix Step of the NFS factoring from the generic array of FPGAs to one of the earliest models of the SRC 6E reconfigurable supercomputer. The results of this

attempt have been documented in [14]. Although the initial results of this investigation were somewhat unimpressive – a slowdown by a factor of 6 vs. a generic array of FPGAs have been observed – the general direction of this research is very promising, and a fast progress can be expected in the near future.

The main reasons for the observed slowdown in the operation of the Mesh Routing circuit after porting it to the SRC 6E machine included:

- the use of a smaller FPGA device XC2V6000 FPGA in SRC 6E vs. XC2V8000 used in the generic array of FPGAs,
- requirement for a fixed clock frequency of 100 MHz in SRC 6E, which led to the need of redesigning the mesh cell, in such a way that each operation of this cell took a larger number of clock cycles,
- relative immaturity of the compiler technology that results in a relatively large area and time overhead of a digital circuit described in C vs. the same circuit described in VHDL or Verilog.

On the other hand, the reasons for optimism, and expected fast progress include

- the emergence of new companies supporting reconfigurable supercomputing, including major players in the area of traditional supercomputing, such as Cray Inc. and SGI [17, 18].
- constant progress in the capabilities, performance, and flexibility of existing reconfigurable computing platforms developed by companies specializing in reconfigurable supercomputing, such as SRC Computers Inc., and Star Bridge Systems [16, 19]
- constant progress in the compiler technology, and logic synthesis of high level programming languages.

Our future work will include the investigation which of the existing and emerging reconfigurable computing platforms and software environments is the most suitable for the implementation of factoring, and other problems related to breaking cryptographic systems. We will also attempt to determine which algorithms and architectures used for codebreaking can be most efficiently implemented using reconfigurable platforms.

7. 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 paper, we move a step closer to a realistic estimate of the difficulty of factoring in hardware for practical sizes of numbers used in cryptography. One of the two most time consuming steps of the factoring algorithm, Matrix Step, has been practically implemented for the first time. A Mesh Routing architecture proposed by Lenstra et al. has been analyzed, designed, and implemented in reconfigurable hardware, using a scalable approach. The area and timing of the implementation has been determined for the state-of-the-art Xilinx Virtex II XC2V8000 FPGA devices. The applicability of the circuit for factoring 512-bit and 1024-bit numbers using an array of FPGA devices has been demonstrated.. With only 1024 Virtex II chips, the Matrix Step of factorization of a 1024-bit number can be performed in 27 days.

Our future work will include the implementation of all remaining steps of the NFS factoring algorithm using a generic array of FPGA devices, and then porting our designs to a selected general-purpose reconfigurable supercomputer.

8. References

[1] 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.

[2] 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.

[3] 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. 55-74.

[4] A.K. Lenstra, H.W. Lenstra, Jr., (eds.), *The development of the number field sieve*, Lecture Notes in Math. 1554, Springer-Verlag, 1993.

[5] A.K. Lenstra, H.W. Lenstra, Jr., *Algorithms in number theory*, chapter 12 in *Handbook of theoretical computer science, Volume A, algorithms and complexity* (J. van Leeuwen, ed.), Elsevier, Amsterdam (1990).

[6] A. Shamir, E. Tromer, "On the cost of factoring RSA-1024", RSA CryptoBytes, vol. 6 no. 2, 2003, pp. 10-19.

[7] D. Coppersmith, "Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm", Math. Comp. bf 62 (1994), pp. 333-350.

[8] D. J. Bernstein, "Circuits for integer factorization: a proposal", http://cr.yp.to/papers/nfscircuit.pdf.

[9] D. Wiedemann, "Solving sparse linear equations over finite fields", *IEEE Transactions on Information Theory*, IT-32 (1986), pp. 54-62 .

[10] G. Villard, “Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems” (extended abstract), Proc. 1997 International Symposium on Symbolic and Algebraic Computation, ACM Press, 1997, pp. 32-39.

[11] H. J. Kim and W. H. Mangione-Smith, *Factoring Large Numbers with Programmable Hardware* UCLA Electrical Engineering Dept. http://klabs.org/richcontent/MAPLDCon99/Presentations/D5A_Kim_S.PDF.

[12] W. Geiselmann, R. Steinwandt, “Hardware to solve sparse systems of linear equations over GF(2)”, Proc. CHES 2003, LNCS 2779, Springer-Verlag, 2003, pp. 51-61.

[13] S. Bajracharya, D. Misra, K. Gaj, T. El-Ghazawi, “Reconfigurable Hardware Implementation of Mesh Routing in the Number Field Sieve Factorization,” Proc. Field Programmable Technology Conf. (FPT’04), Brisbane, Australia, Dec. 2004.

[14] S. Bajracharya, *Reconfigurable Hardware Implementation and Analysis of Mesh Routing for the Matrix Step of the Number Field Sieve Factorization*, MS Thesis, ECE Department, George Mason University, Dec. 2004.

[15] T. El-Ghazawi, D. Buell, M. Gokhale, K. Gaj, *Reconfigurable Supercomputing Systems*, Proc. tutorial presented during Supercomputing 2004 Conf., Pittsburgh, PA, Nov. 2004.

[16] SRC Computers Home Page. Available: http://www.srccomp.com

[17] Cray XD1 Overview. Available: http://www.cray.com/products/xd1

[18] SGI Altix 3000 family of Servers and Supercomputers. Available: http://www.sgi.com/products/servers/altix/

[19] Star Bridge Systems Inc. Home Page. Available: http://www.starbridgesystems.com

[20] O. D. Fidanci, D. Poznanovic, K. Gaj, T. El-Ghazawi, and N. Alexandridis, “Performance and Overhead in a Hybrid Reconfigurable Computer,” Proc. Reconfigurable Architecture Workshop 2003.

[21] S. Bajracharya, C. Shu, K. Gaj, T. El-Ghazawi, “Implementation of Elliptic Curve Cryptosystems over GF(2^n) in Optimal Normal Basis on a Reconfigurable Computer,” 14th International Conference on Field Programmable Logic and Applications, FPL 2004, Antwerp, Belgium, Aug. 30 – Sep. 1, 2004, pp. 1001-1005.