# Reconfigurable Hardware Implementation of Mesh Routing in 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 XC2V6000. 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 less than 40 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 involves the multiplication of a large sparse matrix with vectors. The result is then used to identify a linear dependence between the entries in the sparse matrix. 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].

Matrix 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.

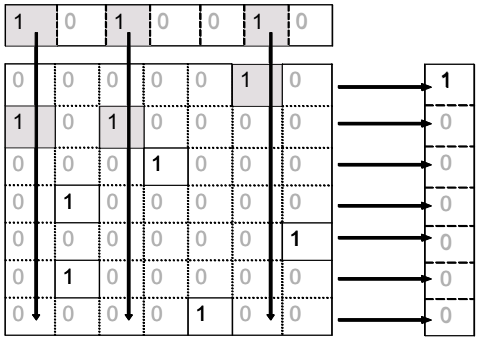

**Figure 1. Matrix-vector multiplication operation through routing.**

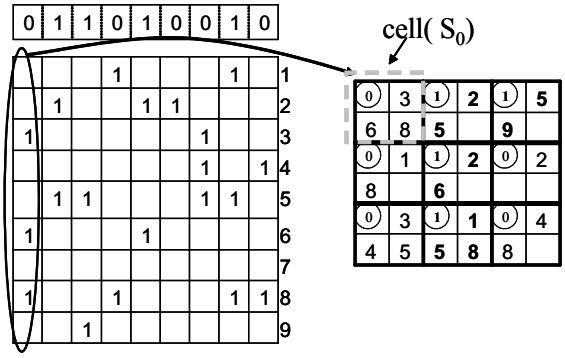


**Figure 2. Mesh corresponding to the sparse matrix A.**

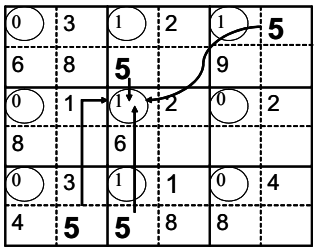

**Figure 3. Routing of the packets to the cell in the mesh.**

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, with a difference that each mesh cell holds the row indices of the non zero values in one column of the sparse matrix.

It is assumed that each of the D columns of the D*D sparse matrix A, has a weight/density '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, where $m = \sqrt{D}$. 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 the circuit, correspond to the matrix entries which are 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 top-leftmost cell, shifting from one cell to another.

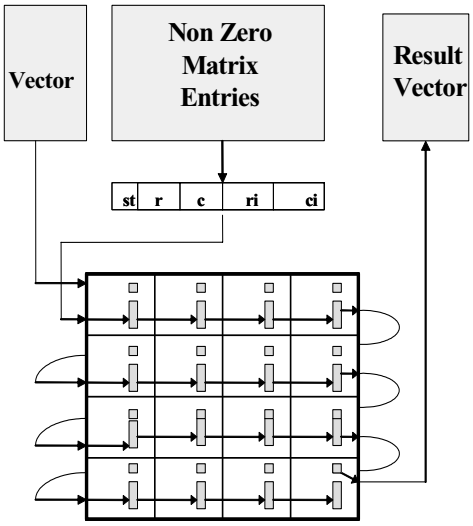


**Figure 4. Loading and Unloading.**

The result of the matrix and the vector multiplication is the 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 bottom cell. This is the basic approach of the design. Another approach with maximum IO pins utilization is considered for calculating the loading and unloading time in Section 5.

### 3.2. Mesh Routing Operation

The matrix-vector multiplication operation is done by routing each packet with the corresponding vector bits of that packet to the destination cells determined by the *r* and *c* address 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.

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.

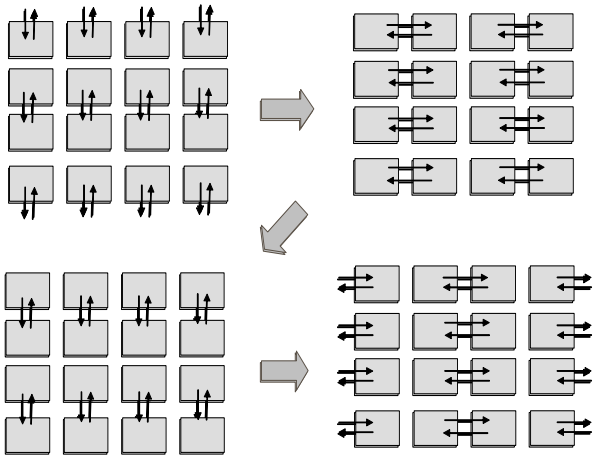

**Figure 5. Four iterations of Compare-Exchange.**

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 is 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.

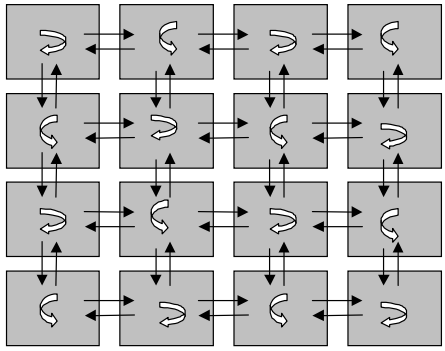

**Figure 6. Compare- exchange direction for each cell.**

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. An analysis reveals that there are four cases of compare-exchanges, as follows:

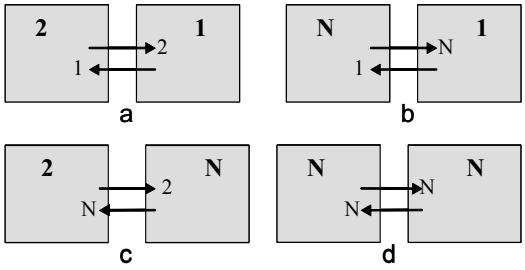


**Figure 7. Compare-exchange cases.**

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.

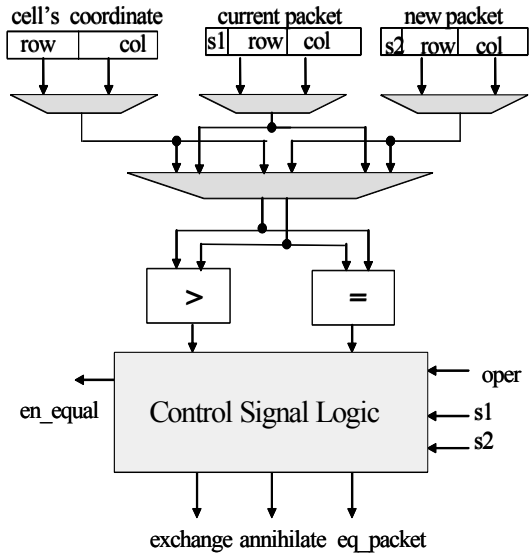


**Figure 8. Comparator Logic.**

The comparison logic is implemented in each cell in the comparator to account for all of these cases as shown in Fig. 8. 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 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 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 (LUT-RAM) storage 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 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 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 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. 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 lesser than or greater than relation are determined by these status bits and the phase of iteration. There are external control signals to each cell to command on certain operation of loading, computing and unloading.

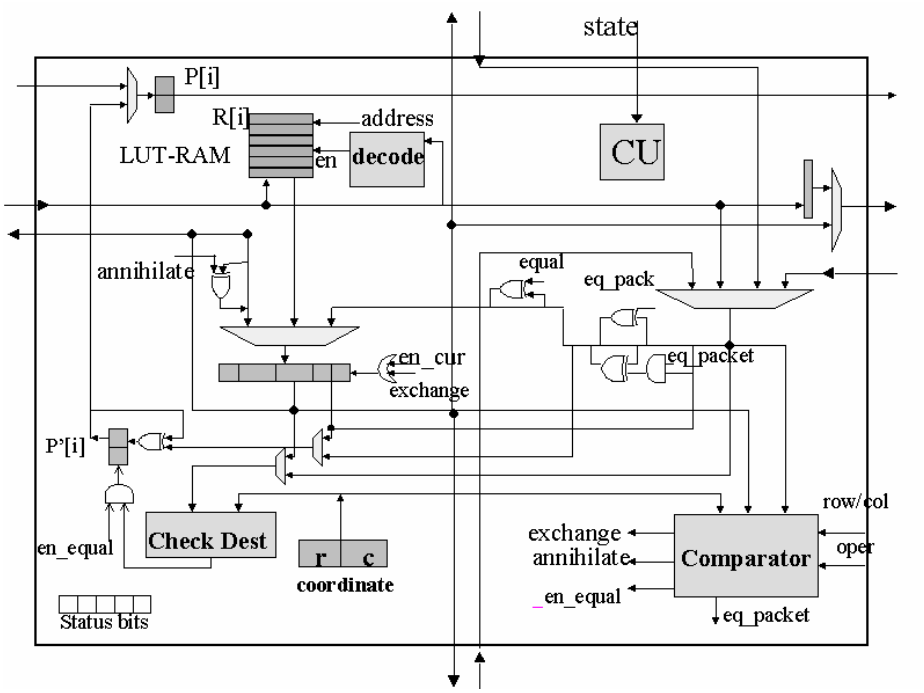


**Figure 9. Detailed architecture of each cell.**

### 3.3. Sub-Matrix Computation

Since the particular hardware device of fixed size cannot perform the huge matrix-vector multiplication, the computation has to be divided into sub-computations of multiplication of smaller sub-matrices with a part of the vector as proposed in [12]. This way 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 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).

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

If only a certain number of chips 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 developed in C for the test vectors. The design is verified in Aldec-Active HDL platform through simulation with the test vectors from the software written in C. The synthesis of the circuit is done through Synplicity Synplify Pro and Xilinx ISE Series to target the Virtex II XC2V6000 device. The test vectors and the inputs required by the hardware circuit are generated in C.

## 5. Results

The row and column indices for the mesh implemented are 4 bits long. This is because the implemented mesh size is 12x12 which is the maximum size of any mesh which can fit on the Virtex II FPGA device. The maximum size of K

(number of vectors multiplied simultaneously) for which the mesh of 12x12 can fit on the Virtex II FPGA device is obtained to be K=50. Thus this mesh can perform matrix-vector multiplication of a sparse matrix of size 144x144 with 50 vectors, each of size 144x1 in one iteration of the Mesh Routing algorithm. Each packet has 1 status bit, 4 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 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 x $10^6$ columns [2]. This matrix is preprocessed to have uniform distribution of non-zero entries. The matrix is divided into $m^2$ sub-matrices. The maximum density per column for each sub-matrix thus turns out to be 1 as 63 ones need to be uniformly distributed in $m^2$ sub-matrices. Hence, d (density) is equal to 1.

**Table 1. Synthesis Results for Mesh Routing Circuit in Virtex II FPGA.**

| Matrix Size | K | CLB | LUT | FF | Period (ns) | Time for K mult (ns) | Time for 1 mult (ns) |
|---|---|---|---|---|---|---|---|
| 144x144 (Mesh 12x12) | 1 | 7,989 (23%) | 15,330 (23%) | 5,255 (7%) | 17.3 | 415 | 415.2 |
| 144x144 (Mesh 12x12) | 42 | 29,325 (86%) | 57,282 (84%) | 29,417 (43%) | 16.9 | 406 | 9.6 |
| 144x144 (Mesh 12x12) | 50 | 33,280 (98%) | 65,119 (96%) | 33,280 (50%) | 17.7 | 425 | 8.5 |

The circuit was first synthesized for K=1 which is the basic case for doing one vector multiplication with the matrix, to find out the resource usage. The circuit was slightly optimized for high fan-out of control signals to all the mesh cells by replicating control signals. After synthesis, the result for maximum clock periods for different K values is as shown in Table 1. Since for K>1, K multiple matrix-vector multiplications are occurring at the same time, larger K corresponds to more speedup provided it can fit on one FPGA device. This limit was obtained to be K=50 for Virtex II FPGA device for the mesh size of 12x12. The area resources used are shown in Table 1. Increase in K significantly increases the area usage. Particularly comparing for the cases K=1 and K=42, it can be noted that there is an increase of less than 4 times for CLB and LUT and a 6 times increase in FF resource usage. In this case, the limiting factor is LUT which implements the combinational logic of the circuit. With K=50, the LUT consumption goes to about 96%.

Each routing needs an average of 2*m clock cycles. Hence, for doing one round of matrix-vector multiplication of 144x144 matrix (with maximum non-zero entries being 1 and mesh dimension m=12) with the 50 vectors of 144x1 takes about 1*2*12 clock cycles, which translates to about 425 ns. Since multiple matrix-vector multiplications are done at the same time, time per multiplication becomes 8.5 ns.

The practical implementation results provide the understanding of how the circuit resource will be utilized. The logic needed for control and the complete circuit functioning has to be taken into account. Hence, these parts also contribute to the area resource usage and timing of the circuit as seen in the implementation results.

Using distributed approach as proposed by Geiselmann and Steinwandt [12], the larger matrix-vector multiplication can be broken down into smaller matrix-vector multiplications and the results can be combined together to get the final results. But instead of using all the chips and doing all the computations at once, we obtain the performance measures for limited resources of FPGA chips particularly for the case of one 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 such that the connection between mesh cells in two chips can be carried out through fewer pins. Particularly for Virtex II chip, there are 1104 IO pins in total. We are particularly interested in knowing how the speedup behaves for multiple chips as opposed to having multiple CPUs do the multiple computations in parallel.

We take the case for 512-bit factorization. Table 2 shows the result of this calculation based on the practical implementation results obtained for 1 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 x m. Since multiple multiplications have to be done serially, n represents the number of such multiplications that needs to be done. Thus n is the number of sub-computations of multiplications of sub-matrices with sub-vectors. The original matrix A from the sieving step has size of D x D. The mesh of size mxm will handle the sub-matrix of size $m^2$ x $m^2$. 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 [2]. Thus the total time for the matrix step is 3*D/K* n * Time for one mesh computation & loading-unloading time.

The results reported in [1] for the factorization of a 512 bit number, are 224 CPU hours (9.3 days)

**Table 2. Time estimates for matrix step for factoring 512 bit numbers with one Virtex II chip and multiple Virtex II chips connected in a mesh with K= number of vectors=50.**

D = number of columns in matrix A
m = mesh dimension
n = number of times to repeat 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 Matrix step = $3*D/K*n*(T_K + T_{Load})$

| Virtex II chips | D | m | n | $T_K$ (ns) | $T_{Load}$ (ns) | $T_{Total}$ (days) |
|---|---|---|---|---|---|---|
| 1 | $6.7 \times 10^6$ | 12 | $2.1 \times 10^9$ | 425 | 64 | 4928 |
| $10^2$ | $6.7 \times 10^6$ | 120 | $2.1 \times 10^5$ | 4250 | 1815 | 6.1 |
| $16^2$ | $6.7 \times 10^6$ | 192 | 33032 | 6797 | 2892 | 1.49 |
| $32^2$ | $6.7 \times 10^6$ | 384 | 2064 | 13593 | 5773 | 0.19 |

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 $32^2$ FPGA chips, this step can be done in 0.2 days from Table 2 .

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 10x10 array of chips and the single mesh of size 120x120 is spread over these 100 Virtex chips. The time estimation for this case is extrapolated from the basic time for 1 Virtex II chip.

For doing sub-computations, the contents of the submatrix have to be loaded on the chip together with the sub-vectors. The modified approach for loading and unloading bus sizes is taken into consideration to calculate the loading and unloading time with maximum possible IO pins that can be utilized in the Virtex II chips. This analytical variant to the original design is considered for analytical extrapolation in calculating the loading and unloading time. The partial result vectors are unloaded infrequently, since the xor operations of intermediate results can be done inside the circuit. The loading and unloading time has to be taken into account for the calculation of total time. The frequency of loading circuit is assumed to be clocked at 200 MHz since the loading shift circuit has very few logic gates involved in the critical path. The maximum pins of the Virtex II I/O pins available at one side is taken into account to calculate the total number of clock cycles to load the matrix-bits and vector bits into the circuit and out of the circuit for the case of multiple chips connected in two dimensions. Let n be the bits for 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 size (1+2*n) stored in memory. Since there are a total of d non-zero entries in the column of sub-matrix and each cell stores d non-zero packets, there are total of $d*m^2$ packets that needs to be loaded and K vectors of size $m^2$. Thus it takes $(1+2*n+K) * m^2 * d / b$ 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. For one chip the number of IO pins available can be obtained from four sides as it is not connected to other chips. Thus the loading time is very less compared to the computing time. The loading time increases in the case of multiple FPGAs being used.

The speedup seen for 100 Virtex chips compared to one Virtex chip is about 1000. This speedup increases in more than linear fashion than the speedup expected with multiple devices. Actually, the speedup increases by about (number of chips)$^{3/2}$ because of the utilization of distributed mesh computation. The execution time estimates for factoring 1024 bit numbers using different number of Virtex II chips are shown in Table 3.

**Table 3. Time estimates for matrix step of factoring 1024 bit numbers with one Virtex II chip and multiple Virtex II chips connected in a mesh with K=number of vectors=50.**

D = number of columns in matrix A
m = mesh dimension
n = number of times to repeat multiplications
$T_K$ = time for K multiplications in the mesh
$T_{Load}$ = time for loading and unloading for K multiplication
$T_{Total}$ = total time for Matrix step = $3*D/K* n *( T_K + T_{Load})$

| Virtex II chips | D | m | n | $T_K$ (ns) | $T_{Load}$ (ns) | $T_{Total}$ (days) |
|---|---|---|---|---|---|---|
| 1 | $4 \times 10^7$ | 12 | $7.7 \times 10^{10}$ | 425 | 64 | $1.05 \times 10^6$ |
| $10^2$ | $4 \times 10^7$ | 120 | $7.7 \times 10^6$ | 4250 | 1815 | 1297 |
| $16^2$ | $4 \times 10^7$ | 192 | $1.17 \times 10^6$ | 6797 | 2892 | 316 |
| $32^2$ | $4 \times 10^7$ | 384 | $7.35 \times 10^4$ | 13593 | 5773 | 40 |

## 6. 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 XC2V6000 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 $32^2$ (1024) Virtex II chips, the Matrix Step of factorization of 1024 bit numbers can be performed in 40 days.

## 7. 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.