SMITH - A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2)

A. Bogdanov, M.C. Mertens, C. Paar, J. Pelzl, A. Rupp
Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
{bogdanov,mertens,cpaar,pelzl,arupp}@crypto.rub.de

Abstract

This paper presents a hardware-optimized variant of the well-known Gaussian elimination over GF(2) and its highly efficient implementation. The proposed hardware architecture, we call SMITH[1], can solve any regular and (uniquely solvable) overdetermined linear system of equations (LSE) and is not limited to matrices of a certain structure. Besides solving LSEs, the architecture at hand can also accomplish the related problem of matrix inversion extremely fast. Its average running time for $n \times n$ binary matrices with uniformly distributed entries equals $2n$ (clock cycles) as opposed to about $\frac{1}{4}n^3$ in software. The average running time remains very close to $2n$ for random matrices with densities much greater or lower than 0.5. The architecture has a worst-case time complexity of $O(n^2)$ and also a space complexity of $O(n^2)$. With these characteristics the architecture is particularly suited to efficiently solve medium-sized LSEs as they for example appear in the cryptanalysis of certain stream cipher classes.

Moreover, we propose a hardware-optimized algorithm for matrix-by-matrix multiplication over GF(2) which runs in linear time and quadratic space on a similar architecture. This opens up the possibility of building a more complex architecture for efficiently solving larger LSEs by means of Strassen's algorithm. This architecture could significantly improve the time complexity of algebraic attacks on various ciphers.

As proof-of-concept we realized SMITH on a contemporary low-cost FPGA. The implementation for a 50×50 LSE can be clocked with a frequency of up to 300 MHz and computes the solution in 0.33μs on average.

1 Introduction

Solving linear systems of equations is a very common computational problem appearing in numerous research disciplines. From a complexity theoretical point of view, the solution of an LSE is efficiently computable, i.e., using for example the well-known Gaussian elimination algorithm any LSE can be solved in at most cubic time. However, for some areas current algorithms and their software implementations are insufficient or unsatisfying in practice. This is often due to the large dimension or number of LSEs that must be solved in order to accomplish a specific task.

For instance, this is the case in the cryptanalysis of asymmetric encryption schemes like RSA or ElGamal, where very large but sparse LSEs over GF(2) or GF(p) (p prime) need to be solved to obtain a secret key. Since these LSEs must be solved accurately, approximative solutions using fast iterative algorithms are simply useless.

Another kind of problems - which we primarily address in this paper - appears in the cryptanalysis of symmetric ciphers. The overwhelming majority of deterministic symmetric cipher can be represented as finite state machines with the state transition and output functions acting over some binary fields.

[1]Scalable Matrix Inversion on Time-Area optimized Hardware.

The output of such a finite machine depends on a number of input bits (which are often known) and the unknown bits of the internal state (key bits) which are to be determined. These internal dependencies can be reformulated in terms of linear equations over an expanded set of unknown values (variables). This gives rise to the problem of solving the appearing systems of linear equations with respect to the unknown bits of the internal state. More concretely, this leads to the following problems: solving individual large dense LSEs over GF(2) and solving a large number of medium-sized[2] dense LSEs over GF(2) (for some special classes of stream ciphers). Efficiently accomplishing these tasks would potentially mean the reassessment of the security level of almost all symmetric ciphers, since the existing methods for solving the LSEs in software are rather inefficient with respect to the brute force attacks.

However, to the best of our knowledge the alternative of an efficient implementation in hardware has not been analyzed thoroughly yet. In this paper we show that there is still room for drastic efficiency improvements of current algorithms when targeting special purpose hardware such as re-programmable logic.

In order to develop such a special purpose hardware architecture we analyzed the suitability of various algorithms while taking possible simplifications over GF(2) into account. Gaussian elimination turned out to be basically best suited for hardware implementation due to its simplicity. Moreover, the coefficient matrices of LSEs are not required to have any special properties like being symmetric positive definite[3] as this is the case for example for the conjugate gradient methods [16]. However, compared to software, a direct implementation of Gaussian elimination in hardware does not yield the desired advantage in efficiency. By slightly modifying the logic of the algorithm and parallelizing element and row operations, we were able to develop a highly efficient architecture for solving medium-sized LSEs and computing inverse matrices. In a similar way, an architecture for fast matrix multiplication can be obtained and possibly be used in conjunction with the other architecture to solve large LSEs by means of Strassen's algorithm [20].

This paper is roughly structured as follows. We start with a brief discussion of previous work in this area. After reviewing standard Gaussian elimination with backward substitution, we present our optimized version of the algorithm for hardware implementation and analyze its performance. After that, further applications of the hardware architecture are discussed including its particular suitability in the area of cryptanalysis. We then present the actual design of the hardware architecture for solving medium-sized LSEs. Finally, we describe our proof-of-concept implementation on a contemporary low-cost FPGA and provide estimates for a possible ASIC implementation.

2 Previous Work

The special case of efficiently using Gaussian elimination for LSEs with binary coefficients is not well documented yet. To the best of our knowledge, no previous work covers the particular topic of improving time efficiency by implementing an optimized, highly parallelized variant of Gaussian elimination in hardware.

All research done until now deals with parallelized implementations of the standard algorithm for Gaussian elimination without altering the algorithm itself. Furthermore, there is almost no performance data available.

In [8], Gaussian elimination for binary coefficient matrices is implemented as part of the DARPA HPCS Discrete Mathematics Benchmark. However, since the author's focus was on a different part (pattern matching) of the benchmark, there is no explicit performance data for solving LSEs given.

[2]E.g., 2^{16} LSEs with 656 variables for the A5/2 cipher used in GSM.

[3]I.e., $A^T = A$ and $x^T A x > 0$ for all non-zero vectors x.

In [19], an implementation of Gaussian elimination over GF(2) is described for the ICL-DAP[4]. The limit is a matrix containing 4096×4096 elements according to the 4096 processors of the ICL-DAP. Therefore, all element operations (componentwise XOR) within one row operation have been parallelized, permitting execution of a row operation within one clock cycle which results in an overall quadratic time complexity. The need for a supercomputer makes this solution quite expensive, while the achieved grade of parallelization with special purpose hardware is still not as high as possible.
The most recent work describes an implementation of an adapted algorithm for LU-decomposition on an NVidia GeForce 7800 GPU which is able to solve a 3500×3500 matrix with 32bit real number coeffi cients in approximately 5 seconds, where the asymptotic time complexity is still cubic [14]. This impressively shows the power of non-x86 hardware, the cost (around $500 for a GeForce 7800 GTX based graphics card at the time of writing) and time complexity are not yet optimal, though.

Further work in the fi eld of solving LSEs is done for very large, but sparse matrices with implementations of the Wiedemann or Lanczos algorithm which both exploit the sparsity of the matrix. The problem of solving an LSE is reduced to the problem of computing matrix-vector products [7, 18, 15]. For matrices of dimensions up to a magnitude of 10^3 the overhead introduced by these advanced algorithms more than compensates their better time complexity Since our focus is on medium sized and preferably not sparse matrices, the given references follow a different approach which is not considered in this work.

3 Basic Notation

In order to maintain simplicity and consistency throughout this paper, we will use the following notation in all algorithms and explanations:

- Matrices are denoted by capital letters, e.g. A.
- The i-th row vector of a matrix A is denoted by $\vec{a}_i$.
- The j-th column vector of a matrix A is denoted by a_j.
- An element in the i-th row and j-th column of a matrix A is denoted by a_{ij}.
- Column vectors are denoted by lowercase letters, e.g. v, while a row vector is denoted by $\vec{w} = w^T$.

4 Mathematical Background

There are basically two flavors of the Gauß algorithm (both having cubic complexity in time) which are usually used synonymously in literature. In this paper, we will differentiate between Gaussian elimination and LU-decomposition, though.

For a given LSE of the form $A \cdot x = b$, LU-decomposition decomposes the coeffi cient matrix A into the product of a lower triangular matrix L and an upper triangular matrix U s.t. $L \cdot U \cdot x = b$. The solution is obtained by fi rst solving the problem $L \cdot y = b$ for y using forward substitution, where $y = U \cdot x$ and then fi nally solving $U \cdot x = y$ for x using backward substitution.

The relevant approach for the proposed architecture is Gaussian elimination, carried out as follows: A given LSE of the form $A \cdot x = b$ is transformed to the equivalent system $U \cdot x = b'$, where U is an upper triangular matrix, by applying elementary *row operations* (i.e., swapping rows and adding a multiple of one row to another). The system $U \cdot x = b'$ can then trivially be solved by using backward substitution.

[4]Distributed Array Processor made by ICL.

For implementation, the right hand side b is treated as an additional, *passive column* to the coefficient matrix A. A passive column in this context means: The algorithm is carried out as if that column was not present (its presence does not influence the execution of the algorithm in any way), but every executed row operation also affects the passive column. Therefore we do not have to take care of the right hand side while discussing the algorithm; considering just the operations on the coefficient matrix is sufficient. The final algorithm actually used for solving LSEs is then implemented using passive columns.
For a more complete introduction to Gaussian elimination, the interested reader is referred to, e.g. [17].

4.1 Gaussian Elimination over GF(2)

For matrices over GF(2), Gaussian elimination becomes quite simple: The required row operations consist of the conditional XOR of two rows and the swapping of two rows.

Given a binary regular matrix A, Algorithm 1 performs Gaussian elimination and backward substitution in parallel, i.e., when the algorithm terminates the result matrix is not just an upper triangular matrix U, but the identity matrix I, so the corresponding LSE is already solved. Algorithm 1 works as follows: The outer for-loop of the algorithm is executed for each column of A. Within the for-loop, two things happen: First lines 2 to 6 take care of the necessary pivoting by choosing a row below the diagonal with a nonzero element (pivot element) in the currently examined column (pivot column) as the pivot row. Next, lines 7 to 13 execute a *column elimination*: The pivot row is XORed (*row operation*, lines 9 to 11) to each row having a nonzero element in the pivot column, effectively zeroing all elements but one (the pivot element) in the pivot column. Since the pivot element is always moved to the diagonal, the identity matrix remains after the outer for-loop has been executed for every column of A.

Algorithm 1 Binary Gaussian Elimination with Pivoting

```
Require: Regular matrix $A \in \{0,1\}^{n \times n}$
 1: for each column $k = 1 : n$ do
 2:     $s := k$
 3:     while $a_{sk} = 0$ do
 4:         $s := s + 1$
 5:     end while
 6:     exchange $\vec{a}_k$ with $\vec{a}_s$
 7:     for each row $i = 1 : n$ do
 8:         if $(i \neq k) \wedge (a_{ik} = 1)$ then
 9:             for each element j = k : n do
10:                 $a_{ij} := a_{ij} \oplus a_{kj}$
11:             end for
12:         end if
13:     end for
14: end for
```

Let us consider the worst and average case running time of Algorithm 1. The worst case complexity for the pivoting step is quadratic. Furthermore, n column eliminations are necessary, each consisting of n row operations. A row operation during the k-th column elimination consists itself of $n - k + 1$ XOR operations. Hence, in total about n^3 XOR operations are required yielding a worst case running time of $O(n^3)$.

For the average case let us assume a matrix $A = (a_{ij})$ with uniformly distributed entries, i.e., $Pr(a_{ij} = 1) = 0.5$, as input to the algorithm. In this case the pivoting step has linear complexity. Moreover, during a column elimination a row operation will only be executed with probability 0.5 (due to the condition in line 8). Thus, we obtain the following (expected) number of required XOR operations

$$\sum_{k=1}^{n}\sum_{i=1}^{n}\left(\frac{1}{2}\cdot\sum_{j=k}^{n}1\right)=\frac{n^3+n^2}{4}\approx\frac{1}{4}\cdot n^3.$$

Hence, the running time of Algorithm 1 in the average case as defined above is still cubic.

5 Proposed Algorithm

5.1 Gaussian Elimination in Hardware

Since a direct hardware implementation of Algorithm 1 in hardware would not yield the desired speed-up compared to a software implementation, we perform several slight modifications.

Clearly, our goal is to make use of parallelization in hardware. Obviously, the following parts of Algorithm 1, responsible for the actual elimination, are basically well suited for this purpose:

- The per-element operation within the row operations (Lines 9-11).
- The row operations as a whole (Lines 7-13).

Moreover, we change the logic of the algorithm in order to simplify an implementation in hardware: It is equivalent to have a fixed matrix and change the row under examination or to always examine the first row and cyclic shift the whole matrix accordingly. Obviously, this is also true for columns. The shifting approach has the following "advantages": We always need to consider only the first column in order to find a pivot element. Furthermore, always the first row can be used for elimination. Hence, we choose this approach for our implementation since it is better suited for hardware.

Applying the changes described above, Gaussian elimination over GF(2) is transformed to Algorithm 2 which works as described in the following. As before, we iterate over all columns of the matrix

Algorithm 2 Parallelized Binary Gaussian Elimination with Pivoting in Hardware

```
Require: Regular matrix $A \in \{0,1\}^{n\times n}$
1: for each column $k = 1 : n$ do
2:    while $a_{11} = 0$ do
3:       $A := shiftup(n-k+1, A)$;
4:    end while
5:    $A := eliminate(A)$;
6: end for
```

A. In the k-th iteration we perform the following steps to obtain a pivot element: We do a cyclic shift-up of all rows not yet used for elimination (due to the shifting approach these are the first $n-k+1$ rows) until the element at the fixed pivot position (a_{11}) equals 1. More precisely, the mapping $shiftup$ on $(n-k+1, A)$ is computed[5] (in one step) until $a_{11} = 1$, where $shiftup$ is defined as:

$$shiftup : \{1,\dots,n\}\times\{0,1\}^{n\times n}\to\{0,1\}^{n\times n}$$

$$(i,(\vec{a}_1,\dots,\vec{a}_n)^T)\mapsto(\vec{a}_2,\dots,\vec{a}_i,\vec{a}_1,\vec{a}_{i+1},\dots\vec{a}_n)^T\ ,$$

After a pivot element has been found in the k-th iteration, we add the first row $\vec{a}_1$ to all other rows $\vec{a}_i$ where $i \neq 1$ and $a_{i1} = 1$ to eliminate these elements. In addition, we do a cyclic shift-up of all rows

[5]In the actual hardware implementation we keep track of the used rows by means of a $used$-flag instead of using a counter.

and a cyclic shift-left of all columns. By doing the cyclic shift-up operations after an elimination, rows already used for elimination are ”collected” at the bottom of the matrix, which ensures that these rows are not involved in the pivoting step anymore. The cyclic shift-left ensures that the elements which should be eliminated in the next iteration are located in the first column of the matrix. More precisely, the following (partial) mapping is computed which combines the actual elimination, the cyclic shift-up and the cyclic shift-left:

$$eliminate : \{0,1\}^{n\times n} \to \{0,1\}^{n\times n}$$

$$\begin{pmatrix} 1 & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \mapsto \begin{pmatrix} a_{22} \oplus (a_{12} \wedge a_{21}) & \dots & a_{2n} \oplus (a_{1n} \wedge a_{21}) & 0 \\ \vdots & & \vdots & \vdots \\ a_{n2} \oplus (a_{12} \wedge a_{n1}) & \dots & a_{nn} \oplus (a_{1n} \wedge a_{n1}) & 0 \\ a_{12} & \dots & a_{1n} & 1 \end{pmatrix}$$

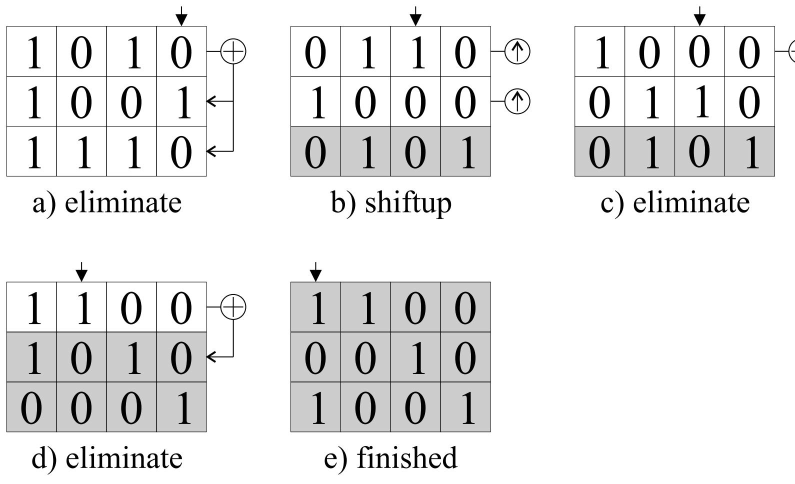
Indeed, the mapping $eliminate$ can be computed within a single clock cycle using our hardware architecture SMITH described in Section 7.

In order to illustrate the procedure above, an example calculation is given in Figure 1. Though explicit b-vectors are generally not considered throughout our paper, for the sake of concreteness the LSE shown below is explicitly solved for x.

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

As outlined in Section 4, the right hand side is implemented as a passive column to the right of A. Figure 1 depicts the change of the matrix elements, while Algorithm 2 is executed. In each step, the corresponding mapping is executed as defined above. Rows already used for elimination are visualized by shading the corresponding elements. For convenience, a small arrow indicates the position of the passive column.[6]

Figure 1: Example matrix solution.



When the algorithm terminates, the coefficient matrix has been transformed into the unit matrix I. The passive column is the leftmost column and contains the solution

$$x = (1,0,1)^T .$$

[6]However, such a pointer to the passive column is not needed in the actual implementation of the algorithm.

5.2 Performance

In the following we analyze the running time of Algorithm 2 on input $A \in \{0,1\}^{n \times n}$. Since we assume that an application of $shiftup$ and $eliminate$ can be computed in a single clock cycle, we simply have to count the required total number of these primitive operations. As one can easily see from Algorithm 2, always exactly n applications of $eliminate$ are performed and only the number of $shiftup$ applications varies depending on the concrete matrix.

Best and Worst Case Running Time. In the best case we have no application of $shiftup$ at all and in the worst we have exactly $n-k$ applications of $shiftup$ during the k-th iteration (assuming A is uniquely solvable). Hence, we obtain the following bounds on the running time:

Proposition 1 (Bounds on Running Time) *The time complexity $T(n)$ of Algorithm 2 is bounded by*

$$n \leq T(n) \leq \frac{n^2+n}{2} = O(n^2).$$

Average Running Time for Random Matrices. Let us consider a random matrix A as input to Algorithm 2 where each entry a_{ij} equals 1 with probability $\alpha \in (0,1)$, i.e., $Pr(a_{ij}=1)=\alpha$ for all $1 \leq i,j \leq n$. Furthermore, for $0 \leq k \leq n$ let $A^{(k)}$ denote the matrix A after the k-th iteration of the for-loop and let $H^{(k)} = (h_{ij}^{(k)})$ denote the $(n-k) \times (n-k)$ matrix which is obtained by deleting the last k rows and columns of $A^{(k)}$. Now, it is important to observe that during the k-th iteration only the elements of the first column of $H^{(k-1)}$ are considered in the pivoting step. If $\alpha = 0.5$ then it is easy to see that the probability $Pr(h_{ij}^{(k)}=1)=0.5$ stays the same for all $0 \leq k < n$.[7] Thus, the expected number of required $shiftup$ operations equals 1 in each iteration. Hence, for $\alpha = 0.5$ we obtain the following average running time:

Proposition 2 (Average Running Time for $\alpha = 0.5$) *The average running time of Algorithm 2 on a binary $n \times n$ matrix A with $Pr(a_{ij}=1)=0.5$ is $2n$.*

Doing similar considerations (and some simplifying assumptions), the following estimate of the running time can be established for the general case $\alpha \in (0,1)$:

Proposition 3 (Average Running Time for $\alpha \in (0,1)$) *Let $A = (a_{ij}) \in \{0,1\}^{n \times n}$ be a random matrix with $Pr(a_{ij}=1)=\alpha \in (0,1)$. Then the average running time $T(n)$ of Algorithm 2 on input A can be estimated by*

$$T(n) \approx \sum_{k=1}^{n} \frac{1}{P_k},$$

where $P_1 = \alpha$ and $P_k = -2P_{k-1}^3 + P_{k-1}^2 + P_{k-1}$, $k > 1$.

It is easy to see that the sequence P_k converges to 0.5 for all $\alpha \in (0,1)$. Since this sequence converges quite fast for α belonging to a large interval around 0.5, the running time is very close to $2n$ for almost all matrices except for very sparse ($\alpha < 0.05$) or very dense ($\alpha > 0.95$) matrices. The accuracy of our estimations has been approved by a real-life benchmark of an emulation of the proposed hardware architecture, where the number of required clock cycles to solve matrices of different dimensions and densities has been counted. The results of this benchmark are depicted in Figure 3 and Figure 2.

[7]This is due to the fact that the XOR sum of two uniformly distributed bits is uniformly distributed.

Figure 2: Size-dependency of running time (for $\alpha = 0.5$).

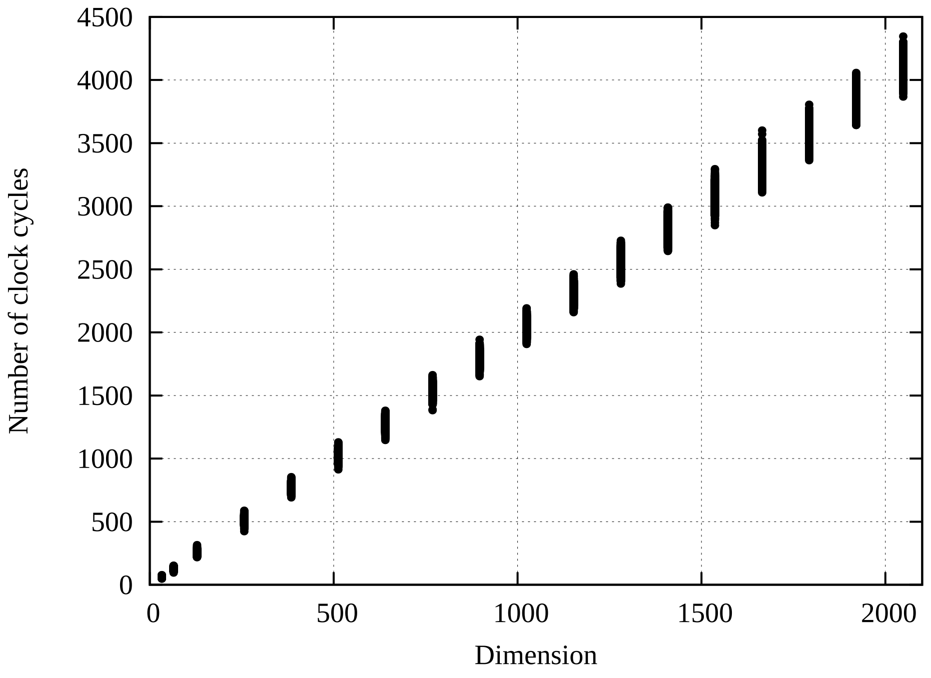
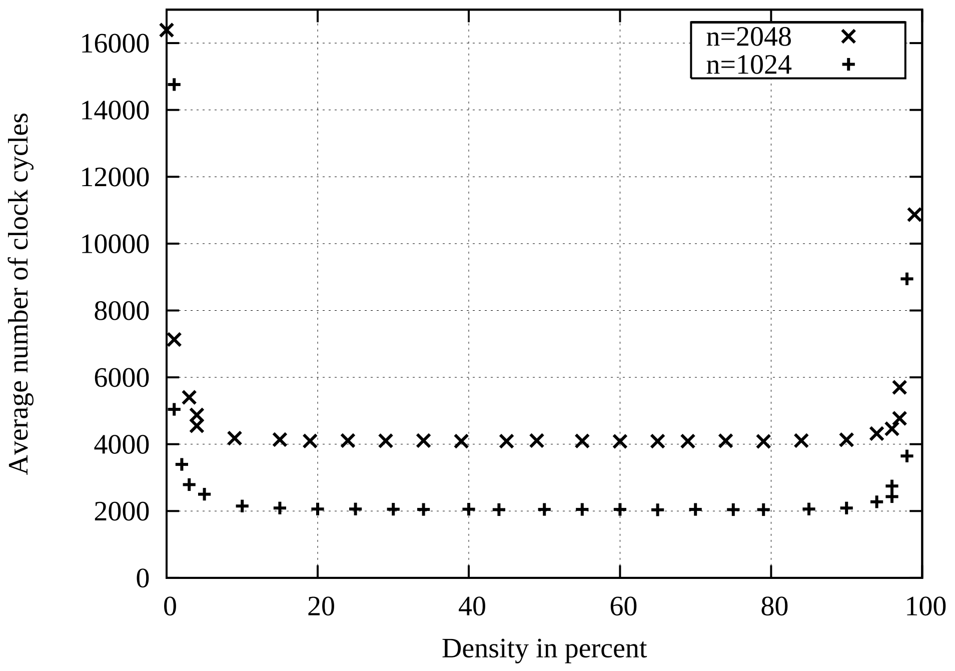


Figure 3: Density-dependency of running time.



5.3 Solving overdetermined LSEs

In practice, solutions to overdetermined LSEs must be computed quite often. An LSE $A \cdot x = b$, where $A \in \{0,1\}^{m \times n}$ and $b \in \{0,1\}^m$, is called overdetermined if $m > n$ (i.e., more equations than unknowns). To compute a solution to an uniquely solvable[8] overdetermined LSE using Algorithm 2, simply the additional rows must be taken into account in the definition (and actual implementation) of $shiftup$ and $eliminate$ by replacing n with m every time it refers to the row count. More precisely, this means that in the k-th iteration the first $m - k + 1$ rows must be shifted up and each application of the elimination operation must also involve the additional rows. Then, by applying Algorithm 2, the matrix A and the passive column b are transformed to

$$A \rightarrow \left(\frac{0}{I}\right), \, b \rightarrow \left(\frac{0}{x}\right),$$

where I is the $n \times n$ identity matrix and the n-bit vector x is the solution to the LSE.

5.4 Multiple b-Vectors and Matrix Inversion

Sometimes, solutions to LSEs of the form $A \cdot x_i = b_i$ ($1 \leq i \leq k$) are required for multiple different right hand sides b_i and the same $m \times n$ coefficient matrix A. Using Algorithm 2 (without modifications), all solutions[9] x_i can be computed in parallel by simply maintaining k passive columns (see Section 4) instead of 1. Clearly, the running time of the algorithm remains the same as in the case of a single passive column. In other words, a solution to $A \cdot X = B$ is computed where $X = (x_1, \ldots, x_k)$ is an $n \times k$ and $B = (b_1, \ldots, b_k)$ is an $m \times k$ matrix. Hence, as a special case we can compute the inverse of a regular $n \times n$ matrix A by simply setting $B = I$ where I is the $n \times n$ identity matrix.

6 Further Applications

Besides solving possibly overdefined medium-sized binary LSEs and computing matrix inverses, the presented architecture can be used for a whole suite of other useful applications. Moreover, the idea behind the architecture can lead to other very efficient matrix algorithms. In Subsection 6.1, we show how to use the approach to derive a very efficient matrix-by-matrix multiplication algorithm. Furthermore, we describe how to solve some other large matrix problems with our design and with the help of Strassen's algorithm. Finally, Subsection 6.3 shows how cryptanalysis of symmetric ciphers is affected by the design at hand.

6.1 Matrix Multiplication

Matrix multiplication is computationally equivalent to matrix inversion. To compute the product of two binary $n \times n$ matrices $C = A \cdot B$ over GF(2) using matrix inversion, and thus by exploiting the SMITH architecture, the following idea can be used. Let the $3n \times 3n$ binary matrix D be defined as follows:

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}$$

Then it is easy to see that the inverse matrix of D has the following form:

$$D^{-1} = \begin{pmatrix} I_n & A & A \cdot B \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}$$

[8] $A \cdot x = b$ is uniquely solvable iff $rank(A) = rank(A|b) = n$, where $A|b$ denotes the matrix A extended by the column vector b.

[9] The algorithm still works if $A \cdot x_i = b_i$ is not uniquely solvable for some b_i.

Thus, by applying Algorithm 2 as outlined in Subsection 5.4 for the inversion of D one obtains a working and quite efficient algorithm for matrix multiplication.

However, further improvements are possible by observing that actually one does not need to run the whole inversion algorithm: To get the result it suffices to perform elimination for the submatrix A only using the elements of the I_n in the middle of D for pivoting. Then the result of multiplication could be directly read out of the upper right $n \times n$ submatrix of D. Exploiting this observation leads to the following optimized algorithm for matrix multiplication which can be implemented in hardware in a similar way as Algorithm 2.

Algorithm 3 Parallelized Binary Matrix Multiplication

```
Require: $A, B, C \in \{0,1\}^{n\times n}$
1: $C := 0$
2: for $k = 1 : n$ do
3:     $C := C \oplus \left(a_{1k} \wedge \vec{b}_k, \ldots, a_{nk} \wedge \vec{b}_k\right)^T$
4: end for
```

Note that the correctness of Algorithm 3 immediately follows from the standard definition of a matrix product. Its running time is n which includes loading the row vector $\vec{b}_k$ and the column vector a_k in each iteration. The space complexity of the algorithm is $O(n^2)$.

6.2 Solving large matrix problems

Due to constraints on the fabrication process of custom designed hardware, Algorithms 2 and 3 can only be implemented for some limited value of n. This circumstance motivates the study of (recursively) splitting large matrix problems into smaller tasks which could in turn be solved using the above described algorithms directly in hardware.

Strassen [20] suggested a number of matrix algorithms breaking down the complexity of the basic matrix operations. Strassen's algorithms provide a way of splitting large matrix problems into smaller ones. Here, only Strassen's multiplication and inversion algorithms for matrices over GF(2) are discussed.

The idea behind Strassen's algorithm for matrix multiplication is that of partitioning matrices and of exploiting specific block representations of matrices to reduce the overall number of required bit operations. For two binary $n \times n$ matrices A and B ,n being even, the product

$$\begin{pmatrix} C_{11}C_{12} \\ C_{21}C_{22} \end{pmatrix} = C = A \times B = \begin{pmatrix} A_{11}A_{12} \\ A_{21}A_{22} \end{pmatrix} \begin{pmatrix} B_{11}B_{12} \\ B_{21}B_{22} \end{pmatrix}$$

can be computed in the following way as expressed using $\frac{n}{2} \times \frac{n}{2}$ matrix subblocks A_{ij} and B_{ij}:

$$\begin{array}{rcl}
P_1 &=& (A_{11}+A_{22})(B_{11}+B_{22}) \\
P_2 &=& (A_{21}+A_{22})B_{11} \\
P_3 &=& A_{11}(B_{12}+B_{22}) \\
P_4 &=& A_{22}(B_{21}+B_{11}) \\
P_5 &=& (A_{11}+A_{12})B_{22} \\
P_6 &=& (A_{21}+A_{11})(B_{11}+B_{22}) \\
P_7 &=& (A_{12}+A_{22})(B_{21}+B_{22}) \\
C_{11} &=& P_1+P_4+P_5+P_7 \\
C_{12} &=& P_3+P_5 \\
C_{21} &=& P_2+P_4 \\
C_{22} &=& P_1+P_3+P_2+P_6
\end{array} \tag{1}$$

This matrix multiplication algorithm requires asymptotically $O(n^{\log_2 7})$ bit operations with a small constant compared to about $O(n^3)$ for the classical multiplication. Strassen's algorithm for matrix multiplication requires 7 multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices instead of 8 ones for the classical algorithm. This is achieved at the cost of several auxiliary matrix additions. The crossover point for Strassen's multiplication algorithm lies as a rule reportedly [4], [5], [9] in the range $n = 16 - 128$ depending on the concrete implementation platform and optimization techniques used. The algorithm is perfectly parallelizable: 7 submatrix multiplications can be performed in parallel, the same holding for the subsequent 4 addition steps.

Strassen's algorithm for matrix inversion is related to the algorithm for multiplication. Let the $n \times n$ matrix C represent the result of inversion of A:

$$\begin{pmatrix} A_{11}A_{12} \\ A_{21}A_{22} \end{pmatrix}^{-1} = \begin{pmatrix} C_{11}C_{12} \\ C_{21}C_{22} \end{pmatrix}$$

In terms of half-sized matrix blocks the inverse matrix C can be calculated as follows:

$$\begin{aligned} P_1 &= A_{11}^{-1} \\ P_2 &= A_{21} \times P_1 \\ P_3 &= P_1 \times A_{12} \\ P_4 &= A_{21} \times P_3 \\ P_5 &= P_4 + A_{22} \\ C_{22} &= P_5^{-1} \\ C_{12} &= P_3 \times C_{22} \\ C_{21} &= C_{22} \times P_2 \\ C_{11} &= P_1 + P_3 \times C_{21} \end{aligned} \tag{2}$$

Applying Strassen's matrix multiplication algorithm for half-sized multiplications and Strassen's inversion algorithm for half-sized inversions recursively we obtain a complexity of $O(n^{\log_2 7})$ operations. The inversion algorithm needs 2 inversions and 6 multiplications of half-sized submatrices. These operations are highly dependent. However, the computation of P_2, P_3 and of C_{12}, C_{21} can be performed in parallel. So the running time of the algorithm is defined by the performance of 2 inversions, 4 multiplications and 2 additions of $\frac{n}{2} \times \frac{n}{2}$ matrices.

Large systems of linear equations, large matrix inversion problems and large matrix multiplication problems can be split into smaller subproblems using Strassen's algorithms for matrix inversion and matrix multiplication recursively. As the optimal minimum splitting size is achieved, the hardware algorithms for inversion (Algorithm 2 used as outlined in Subsection 5.4) and multiplication (Algorithm 3) are to be applied.

6.3 Large matrix problems in cryptanalysis

Large matrix problems over GF(2) are in general strongly related to cryptanalysis and in particular to algebraic attacks on symmetric ciphers which are generically applicable to any block or stream cipher. This is due to the fact that any output of a deterministic binary finite state machine can be represented as a (sometimes rather complicated) boolean function of the initial internal state and input values (if any) giving rise to LSEs over GF(2).

Typical algebraic attacks include the following stages:

- *Generating a system of non-linear equations over GF(2):* In this step some weakness of the analysed cryptographical algorithm is used to derive advantageous binary non-linear equations of some preferably low degree.

- *Linearizing the non-linear equations:* Since no efficient generic algorithm for solving systems of arbitrary non-linear equations is known (moreover, since this problem is exponential in the number of variables), the attacker hopes to get a system of linear equations with a relatively low number of variables in order to be able to solve it using one of known polynomial algorithms. Some linearization algorithm is applied to convert the system of non-linear equations into a system of linear equations with a greater number of unknowns.

- *Solving the system of linear equations:* This step is as a rule the most difficult one and influences the overall cryptanalytical efficiency drastically.

Due to these facts the boolean functions representing symmetric ciphers are always required to be highly non-linear. However, in practice it often turns out to be possible to mount an efficient algebraic attack. This is typical for hardware oriented stream ciphers. Such ciphers can be efficiently implemented in hardware, where non-linear operations over large numbers of variables are relatively expensive. As a rule the designers of hardware stream ciphers combine several linear or poorly non-linear mappings. This makes the ciphers vulnerable to algebraic attacks. Below some instances are briefly discussed in the context of the proposed algorithms for solving basic matrix problems over GF(2).

In [1], an efficient algebraic attack on the keystream generator underlying the encryption scheme E_0 used in the Bluetooth specification is proposed. The key length of the E_0 keystream generator is 128 bit. The equation generation and linearization methods succeed to output an LSE with at most $2^{24.056}$ unknowns[10]. The system of linear equations can be constructed offline once and implemented in one or several ASICs. For each attack (for each new key) the only input data are the elements of the concrete output stream. Using our algorithms for binary matrix inversion and multiplication in a linear number of clock cycles and following the proposed approach for splitting large systems of linear equations into smaller matrix problems, it seems possible to build a hardware system which could significantly reduce the actual running time of the attack and to achieve a good time/money ratio value.

Other examples are algebraic attacks on the A5/2 algorithm used in GSM. Here the attack from [6] is mentioned only. A5/2 is based on 4 maximal-length linear feedback shift registers clocked irregularly. The register R4 controls the clocking process for the registers R1, R2 and R3. The key is the initial filling of these 4 registers and is of length 64 bit. The output is computed using a non-linear function of some positions in R1, R2 and R3. It proves to be possible to construct a system of 656 linearly independent equations by trying all the possible different 2^{16} initial values of R4. The solution of a system which agrees with the output stream is the session key. That is, one has to solve max. 2^{16} systems of 656 linear equations over GF(2) to restore the initial filling of the 4 registers completely. Our algorithm for solving LSEs can be directly applied in this attack breaking down the overall running time significantly. If no data transmission and equation system construction overhead is considered, then the attack can be completed in $2^{16} \cdot 2 \cdot 656 \approx 2^{26.35}$ clock cycles on a single ASIC using the proposed hardware architecture. If the hardware runs with 1 GHz, the key could be found within about 86 ms. The attack can be parallelized by using several parallel ASICs. In this case the running time of the attack can be significantly reduced (max. by factor 2^{16}). This running time can be compared to 40 minutes on a Linux 800 MHz PIII PC [6], the solution of systems of linear equations having the major contribution to the overall complexity.

Moreover, all stream ciphers (e.g. LILI-128, Toyocrypt, Phelix, Snow, Turing, etc.) based on simple combiners and on combiners with memory [11] are potentially vulnerable to algebraic attacks [12], [11], [13] , [10]. In this context the hardware approach to solving large systems of linear equations in hardware may lead to a revaluation of the security of a numerous well-known symmetric ciphers.

[10]Since then some improvements of this attack have been proposed [2], [3].

[11]These are several parallel linear feedback registers filtered by a specially selected output boolean function with or without memory.

7 The Proposed Hardware Architecture SMITH

We will now describe the basic functionality and the operating modes of the hardware architecture of Algorithm 2. First, we discuss the improvement in area-time (AT) complexity resulting from the new design.

7.1 Improvement of AT Complexity

The dramatic improvement in complexity of the average running time from a software implementation of binary Gauß (Algorithm 1) of $O(\frac{1}{4}n^3)$ to the architecture at hand of $O(2n)$ is achieved. We simply compute on all elements in parallel, yielding an improvement of $O(n^2)$ in the running time. However, we do *not* simply trade this improvement of $O(n^2)$ in running time with an equivalent increase in area. In fact, the design scales with the same asymptotic complexity as the standard Gauß in software.

In both algorithms, the memory requirement scales with $O(n^2)$, i.e., with the number of elements of the matrix. The only difference is that the type of memory cells in our design is a bit more complex (smart memory). Thus, the algorithmic improvements do not only yield a much better running time, but also lower the overall AT product compared to the standard algorithm.

7.2 Functional Description

In order to implement Algorithm 2 in hardware, we use a mesh structure of "smart memory" cells: The whole device consists of memory cells, which are able to execute the two basic operations defined in Subsection 5.1 in a single clock cycle. The design at hand comprises a parallel implementation of both operations where the pivot element is used as multiplexing signal for the appropriate result. This way, we save a clock cycle for deciding between the two operations. Besides performing the actual computation, both operations can also be used for initially loading the values into the design. Below, we will describe the basic constituents of the design and its interrelation. The terms in `typewriter` describe the name of the signals used in the figures. If appropriate, the corresponding variable of Algorithm 2 is provided additionally.

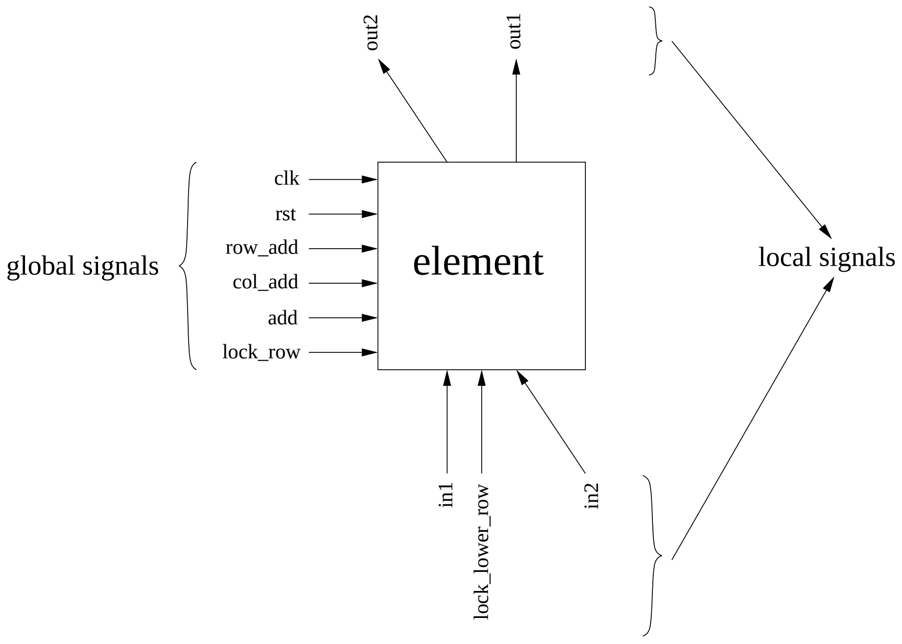
7.2.1 The Basic Cell and its Interconnection

Each cell stores a single bit of the matrix (a_{ij}) and has local connections to 4 direct neighbors: above (`out1`, `lock_row`), below (`in1`, `lock_lower_row`), left above (`out2`), and right below (`in2`). Furthermore, it is connected to a global network: The global network comprises the signals from the pivot element (`add`=a_{11}), the first element in the actual row (`row_add`=a_{i1}), and the first element in the actual column (`col_add`=a_{1j}). Remark, that the $used$-flags for the actual row and the row below are provided by the signals `lock_row` and `lock_lower_row`, respectively. Additionally, every element is connected to the global clock (`clk`) and reset (`rst`) network. Figure 4 depicts a single matrix element with all required local and global connections. All cells are aligned in a rectangular array as shown in Figure 5. The upmost left element is the pivot element of the matrix (a_{11}). Its `out1`-signal is used as `add`-signal for the whole design. The `out1`-signals of all other elements in the first column are used as `row_add`-signals for the actual row (a_{i1}). Similarly, the `out1`-signals of all other elements in the first row are used as `col_add`-signals for the actual column (a_{1j}).

7.2.2 Operation 1: Shift-Up

In case of the actual pivot element being '0', the architecture performs a simple shift-up operation. All elements in the matrix with the $used$-flag set to '0' (`lock_row=0`) simply shift their value one row

Figure 4: Signals from and to a basic matrix element.



up. Values in the upmost row will get shifted to the lowest unused row, resulting in a cyclic shift of all unused rows. All rows with the $used$-flag set to '1' (`lock_row=1`) are not affected by the shift and stay constant. For the implementation, we will use the `col_add`-signals to realize the cyclic shift from the first row to the last unused row.

In the very beginning, values have to be loaded into the design. Since a reset of the design sets all elements to '0', a shift-up is applied n times until the matrix is completely loaded. In order not to load more than n values into the design, the loading phase is coordinated by an input multiplexer, switching off the input after n steps.

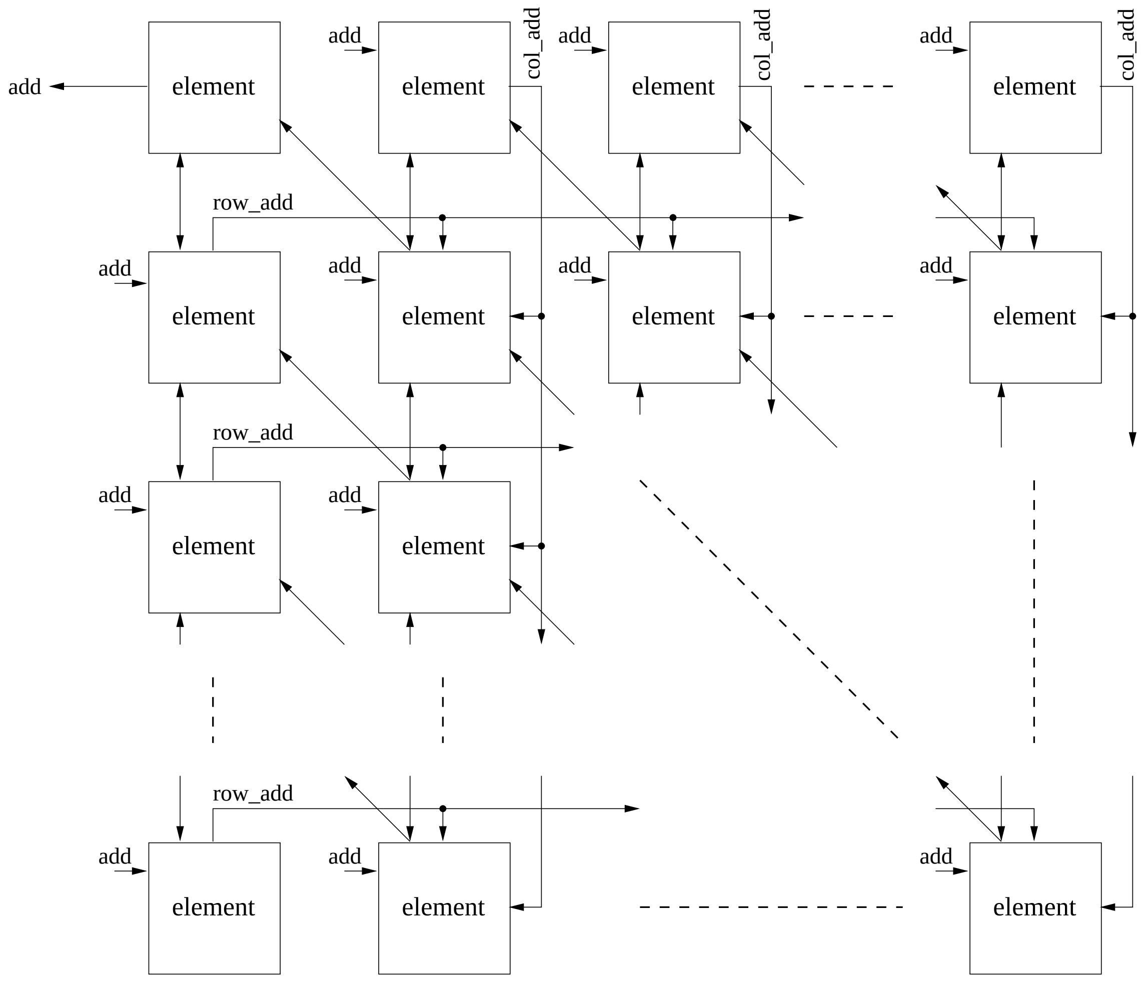
7.2.3 Operation 2: Elimination

Operation 2 is invoked when the pivot element is '1' (`add`=1=a_{11}). Every element except for those in the first row will compute an XOR with the upmost column entry (`col_add`=1=a_{1j}) if the first entry in the row is '1' (`row_add`=1=a_{i1}). In the same step, the result is shifted to the element on the left in the upper row. The leftmost elements need no computation, since this step always yields a column consisting of zeroes and a single '1'.
The first row (with the pivot element) is unaffected and will be shifted to the last row and its $used$-flag (`lock_row`) will be set to '1'. The latter procedure prevents this row from being used again for pivoting.

7.2.4 End of Computation

When all rows have actively been used for pivoting, the $used$-flag reaches the topmost row and the architecture stops. In our design, this signal is used for indicating the end of computation. E.g., in the case of a single solution vector b, the result can simply be read from the first column by reading the `column_add`-signals, which are wired out explicitly. Depending on the outer control logic, the topmost $used$-flag can be used to indicate the finished state and eventually stop the computation until the result has been read.

Figure 5: Interconnection of elements in the architecture.



7.3 Solving Regular and Overdetermined LSEs

For solving a regular LSE of rank n, we need n linearly independent equations. Hence, the design has to implement n rows of elements. After executing Operation 2 (see 7.2.3) n times, the solution is available. For the sake of simplicity, we have always assumed uniquely solvable LSEs in this paper. However, the design can easily be extended to handle overdefi ned LSEs and detect unsolvable LSEs, which we discuss now.

7.3.1 Handling Overdefined LSEs

In the same number of elementary operations, we can solve overdetermined LSEs, i.e., LSEs with rank n and $m > n$ equations. In this case we only need to implement more rows of elements. The solution is obtained after n-times execution of Operation 2.

7.3.2 Solvability of LSEs

In some applications, the solvability of an LSE might not be known prior computation and we have to introduce additional logic for the detection of a valid solution. The architecture presented can be extended in order to distinguish between uniquely solvable and unsolvable LSEs. In such a case, we have to observe following facts:

- A uniquely solvable LSE of rank n is *always* solved after exactly n-times Operation 2. Hence, the $used$-flag is the stop criterion.
- An LSE with infinitely many solutions will never reach the stop criterion since infinitely many 'shift-up' operations will occur at a certain point in time. To avoid such a *dead-lock*, we have to initialize a counter after every elimination step to limit the number of the subsequent 'shift up'-steps. In fact, this number must be less than the number of rows minus the number of locked rows. If not, we know that the LSE has no unique solution.
- Invalid solution might appear in overdetermined LSEs when the b-vector has a '1' at a position corresponding to a zero row in the matrix. Thus, all such positions and have to be checked for '0'.

7.4 Pipelining

In order to achieve improved performance when solving many LSEs on a single “smart memory” chip, one can exploit the fact, that once a column has undergone column elimination, it does not have to be kept in memory anymore. Instead of keeping these columns, it is possible to preload a new column of the matrix which will be processed next, each time a column elimination is executed on the current matrix. With this approach, solving the current matrix and loading the next matrix could be done simultaneously. However, there are some potential caveats to take care of. It has to be ensured, that neither row permutations (caused by “pivoting” step) nor XOR operations (caused by “eliminate step”) can corrupt the columns of the partially loaded new matrix. This can for example be achieved with a flag similar to the already implemented $used$-flag, protecting the rightmost columns. In fact, the already implemented $used$-flag can easily be reused for this purpose.

8 Implementation

This section explains the methodology used to implement the previously presented architecture SMITH on FPGAs. Along with implementational results such as running time and area consumption, we provide estimates about a possible ASIC implementation.

8.1 Methodology

The architecture has been programmed in VHDL and is kept generic to allow for easily changing to arbitrary matrix sizes and different dimensions of the b-vector. The presented results originate from the running implementation. For synthesis, map, and place and route, Xilinx ISE 7.1 was used. Both behavioral and post-place-and-route simulation were done using Mentor Graphics ModelSimXE 6.0a. As device, we picked a contemporary low-cost FPGA, namely the Xilinx Spartan-3 XC3S1000 device.

For loading data from and storing data into the FPGA, we additionally implemented a simple interface to a host PC. It facilitates automated tests with several test values for architectures of different size. Remark, that the speed of the data in- and output is critical in case of an ultimate implementation. Hence, interfaces should be capable of running at high-speed (e.g., by using PCI-express or proprietary systems such as those on a Cray-XD1).

8.2 Results

Table 1 presents the results of the implementation in terms of occupied FPGA resources depending on the size of the matrix. For the implementation, we have chosen a system of linearly independent equations of dimension n for $n = \{5, 10, 20, 50\}$. All values represent the results for the running and verifi ed design on the actual FPGA.

A basic element can be implemented in only ≈ 1.5 slices on average, which limits implementations on FPGAs to a maximum possible matrix size (see Table 1). With the current low-cost FPGA (Xilinx XC3S1000), matrix sizes of up to 70x70 are feasible. Note that we did not optimize the code for a certain FPGA. I.e., FPGA-specifi c optimizations might yield better area usage. The implementation at hand should be seen as proof-of-concept and not as ultimate design. However, the frequency of the matrix elements can be as high as 300 MHz which is close to the maximum possible frequency of the FPGA at hand. All designs have been tested at that frequency and displayed the correct results. Due to the restrictive nature of conventional FPGAs concerning the matrix size, an ASIC implementation should be considered for larger matrices and is discussed in the following section.

Table 1: Area in slices for different matrix sizes on a Spartan-3 device (XC3S1000, 300 MHz).

Matrix dimension	5	10	20	50
Area / slices	54	187	656	4004

8.3 Estimates for an ASIC Design

Conventional FPGAs dispose of several features which can not be used for our design and, hence, the FPGA can not be utilized completely for the algorithm presented. Furthermore, the FPGA's inherent overhead to maintain its programmability prevents from an optimal and AT effi cient design. When targeting a high-performant matrix solver for medium-sized matrices, a realization as Application Specifi c Integrated Circuit (ASIC) which can be tweaked completely to fi t the requirements should be considered.

Since SMITH has been successfully implemented on an FPGA, we now analyze its area consumption as ASIC for different matrix sizes and conclude with an estimate of the expected performance. For the estimate, we will not take drivers and buffers for the high-fanout wires into account. Furthermore, the maximum chip size will be limited due to the long data paths and expect to be able to built such a device for matrices no larger than 1000 rows.

8.3.1 Equivalent Gate Count of Basic Building Blocks

As previously shown, the presented design is extremely simple and does not require complicated control logic. The matrix cell can be implemented with a few basic gates, as shown in Figure 6.

Thus, the area consumption in terms of transistors or equivalent gates (standard CMOS logic) can be evaluated pretty accurate. As essential building blocks, we need XORs, ORs, ANDs, NOTs, and memory cells. The essential building blocks of the design are listed in Table 2, accompanied by the required number of gates for the implementation in standard CMOS logic. For a single element of the matrix, we estimate an equivalent of 54 gates.

Obviously, the number of gates scales For comparison, a conventional Pentium 4 "Prescott" processor accumulates approximately 125,000,000 transistors (0.09 microns). Hence, with current technology such an architecture for matrices up to a dimension of 1000 could be implemented on a die size of less than that for a conventional desktop CPU. Figure 7 shows the number of equivalent gates depending on the

Figure 6: Schematic of the basic matrix cell.

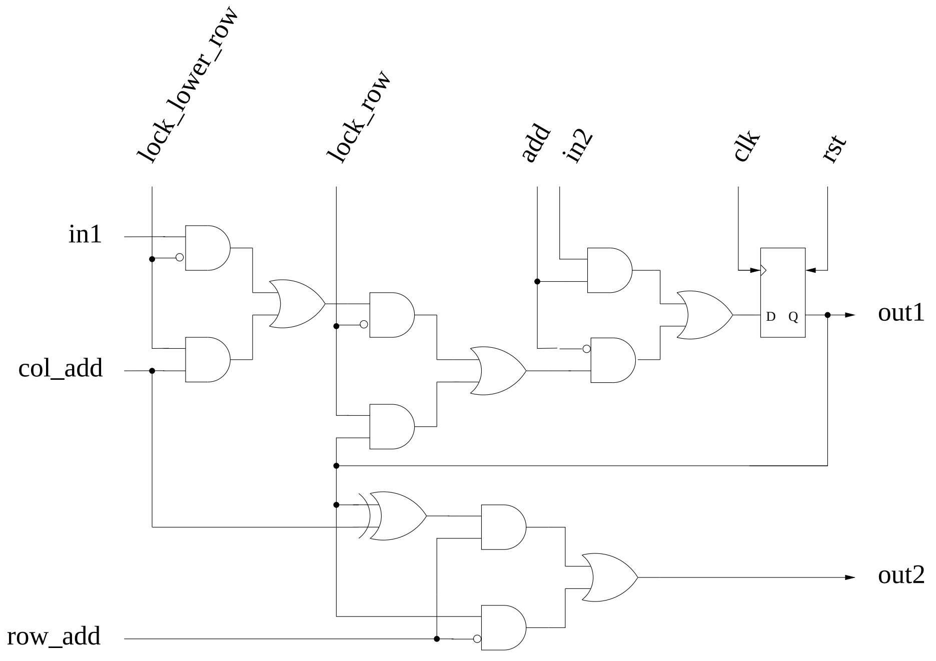


Table 2: Equivalent gate-count for different CMOS logic cells (all quantities per bit).

Logic type	XOR	OR	AND	NAND	SRAM
Equivalent Gates	3	3	4	1	3

matrix size. Note that power consumption and cooling might become an issue since, in contrast to a conventional CPU with cache, the whole circuit is active during a computation.

Remark: The number of estimated gates is based upon simple 2-input logic gates. Hence, further optimization is possible and will result in a lower transistor count. Due to the regularity and relatively low latency of the design, clock frequencies in the range of 500 MHz up to a few GHz should be feasible.

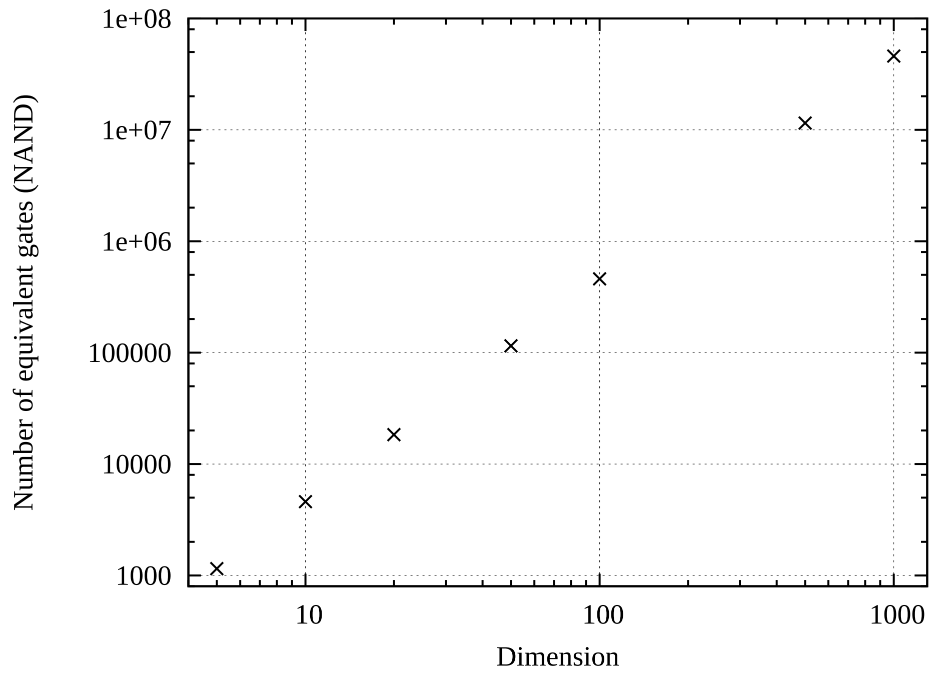
8.3.2 Expected Performance

Depending on the matrix-size, the number of required building blocks together with Table 2 yield an approximation of the necessary number of equivalent gates, which are displayed in Figure 7. The fi gure includes an estimated overhead for the control logic.

Loading an LSE of dimension 1000 into such a design, solving it, and reading out the solution could be done in 4 μs on average, assuming a clock rate of 1 GHz. Hence, up to 250 000 LSEs could be solved per second, assuming an adequate data input and output rate.

For applications, an integrated pre- and post processing might be of interest in order to avoid the architecture from running idle. Only a high-speed stream of in- and outputs can bring out the best performance.

Figure 7: Estimated number of equivalent gates depending on the dimension of the LSE



9 Conclusion

The paper at hand seems to be the first contribution towards a highly efficient architecture for solving linear systems of binary equations. We formulate an algorithmic improvement for the well-known Gaussian elimination for the binary case, focussing hardware implementations. The new algorithm has been thoroughly analyzed and we provide lower and upper bounds on the running time as well as the average running time. Results from software experiments confirmed the correctness of our theoretical findings. As a result, an architecture implementing the proposed algorithm can solve an LSE over GF(2) of rank n in $2n$ steps on average, compared to $\frac{1}{4}n^3$ steps for the standard Gaussian algorithm implemented in software.

Besides the derivation and analysis of the new algorithm, a hardware implementation for LSEs up to a rank of 50 has been realized using VHDL. The resulting implementation can be operated at high speed (up to 300MHz) and was verified on a contemporary low-cost FPGA (Xilinx Spartan3-1000). In addition, we provide a first estimate of the area complexity of an ASIC design: With current technology, the design can be implemented for LSEs up to a rank of 1000 yielding conventional chip sizes. As a remarkable feature, the area complexity of the architecture scales with $O(n^2)$, which is similar to the standard Gaussian approach, though the complexity of the running time is $O(n^2)$ lower. Thus, the overall improvement regarding the AT-product is of order $O(n^2)$ compared to the standard algorithm in software.

Due to technological constraints, we always face a maximum possible matrix size which can be realized in practice. Consequently, we describe how to solve larger LSEs by breaking down the initial matrix size with Strassen's Algorithm. As showed within the paper, further applications include fast matrix-by-matrix multiplication and utilization in cryptanalysis. In the latter case, time consuming parts of actual cryptanalytical algorithms can be accelerated by order of several magnitudes with the architecture presented.

References

[1] F. Armknecht. A Linearization Attack on the Bluetooth Key Stream Generator. Cryptology ePrint Archive: Report 2002/191 Available from `http://eprint.iacr.org/2002/191`, December 2002.

[2] F. Armknecht. Improving Fast Algebraic Attacks. In *FSE 2004*, volume 3017 of *LNCS*. Springer Verlag, 2004.

[3] F. Armknecht and G. Ars. Introducing a New Variant of Fast Algebraic Attacks and Minimizing Their Successive Data Complexity. In *Mycrypt 2005*, volume 3715 of *LNCS*, 2005.

[4] D. Bailey and H. Ferguson. A Strassen-Newton Algorithm for High-Speed Parallelizable Matrix Inversion. In *the 1988 ACM/IEEE conference on Supercomputing*, pages 419 – 424, 1988.

[5] D. Bailey, K. Lee, and H. Simon. Using Strassen's Algorithm to Accelerate the Solution of Linear Systems. *Journal of Supercomputing*, 4:357–371, 1990.

[6] E. Barkan, E. Biham, and N. Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. In *CRYPTO 2003*, volume 2729 of *LNCS*, pages 600–616. Springer Verlag, 2003.

[7] D.J. Bernstein. Circuits for Integer Factorization: A Proposal. Manuscript. Available at `http://cr.yp.to/papers.html#nfscircuit`, 2001.

[8] D. A. Buell, S. Akella, J. P. Davis, G. Quan, and D. Caliga. The DARPA boolean equation benchmark on a reconfigurable computer. Technical report, Department of Computer Science and Engineering, University of South Carolina, 2004.

[9] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2nd edition, 2001.

[10] N. Courtois. Cryptanalysis of Sfinks. Cryptology ePrint Archive: Report 2005/243. Available from `http://eprint.iacr.org/2005/243`.

[11] N. Courtois. Fast Algebraic Attacks on Stream Ciphers with Linear Feedback. In *Crypto 2003*, volume 2729 of *LNCS*, pages 177–194. Springer Verlag, 2003.

[12] N. Courtois. Algebraic Attacks on Combiners with Memory and Several Outputs. In *ICISC 2004*, LNCS, 2004.

[13] N. Courtois and W. Meier. Algebraic Attacks on Stream Ciphers with Linear Feedback. In *Eurocrypt 2003*, volume 2656 of *LNCS*, pages 345–359, 2003.

[14] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In *ACM/IEEE SC—05 Conference, November 12-18*, 2005.

[15] W. Geiselmann, A. Shamir, R. Steinwandt, and E. Tromer. Scalable Hardware for Sparse Systems of Linear Equations, with Applications to Integer Factorization. In *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2005*, pages 131–146. Springer, 2005.

[16] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal Res. Nat. Bur. Standards*, 49:409–436, 1952.

[17] C. Kanzow. *Numerik linearer Gleichungssysteme*. Springer Verlag, 2005. ISBN 3-540-20654-X.

[18] M. LaMacchia and A. Odlyzko. Solving large sparse linear systems over finite fields. In A.J. Menezes and S.A. Vanstone, editors, *Advances in Cryptology - Crypto '90*, volume 537 of *LNCS*, pages 109–133, 1991.

[19] D. Parkinson. A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers. *Parallel Computing*, 1:65–73, 1984.

[20] V. Strassen. Gaussian Elimination Is Not Optimal. *Numerical Mathematics*, 13:354–356, 1969.