# Efficient Implementation of Rijndael Encryption with Composite Field Arithmetic


Atri Rudra[1], Pradeep K. Dubey[1], Charanjit S. Jutla[2], Vijay Kumar[⋆1], Josyula R. Rao[2], and Pankaj Rohatgi[2]

[1] IBM India Research Lab,
Block I, Indian Institue of Technology,
Hauz Khas, New Delhi, 110016, India
{ratri,pkdubey,vijayk}@in.ibm.com

[2] IBM Thomas J. Watson Research Center,
P.O.Box 704,
Yorktown Heights, NY 10598, U.S.A.
{csjutla,jrrao,rohatgi}@watson.ibm.com



**Abstract.** We explore the use of subfield arithmetic for efficient implementations Galois Field arithmetic in the context of Rijndael cipher. Our technique involves mapping field elements to a composite field representation. We describe how to select a representation which minimizes the computation cost of the relevant arithmetic, taking into account the overhead such as the cost of the mapping. Our method results in a very compact and fast gate circuit for Rijndael encryption.

In conjunction with bit-slicing techniques applied to parallelizable modes of operation ( such as IAPM [7] ) , our circuit leads to a high-performance software implementation for Rijndael encryption which offers significant speedup compared to previously reported implementations.


## 1 Introduction

In October 2000, the US National Institute of Standards and Technology (NIST) announced that it had selected the Rijndael Block Cipher [3] as the proposed new Advanced Encryption Standard (AES). In addition to being the proposed standard, Rijndael is a cipher which offers a good "combination of security, performance, efficiency, implementability and flexibility" [20]. It has already attained considerable popularity and acceptance. Rijndael is a block cipher with a block size of 16 bytes, each of which represents an element in the Galois Field $GF(2^8)$. All operations in Rijndael are defined in terms of arithmetic in this field.

Apart from Rijndael, there are numerous other instances of the use of Galois Field arithmetic in cryptography and coding theory [10]. The efficiency and performance of such applications is dependent upon the representation of field


⋆ As of April 2001, the author can be reached at Amazon.com, 605 $5^{th}$ Ave South, Seattle, WA 98104, U.S.A.

elements and the implementation of field arithmetic, and it is a common practice to seek efficiency through this route [9–11]. In particular, it is well-known that the computational cost of certain Galois Field operations is lower when field elements are mapped to some isomorphic composite field, in which these operations are implemented using lower-cost subfield arithmetic operations as primitives [11]. Depending upon the computation involved and the choice of representation, there are costs and overhead related to the mapping and conversion, and trade-offs between such overhead and the savings obtained. The design task is to carefully evaluate these trade-offs to minimize the computational cost.

In addition to an efficient hardware implementation, a good circuit design is also useful in obtaining fast software implementations. Using the technique of bit-slicing [2] a circuit with a small number of gates can be simulated using a wide-word processor. Multiple instances of the underlying computation are thus performed in parallel to exploit the parallelism implicit in a wide-word computer. This technique is used in [2] to obtain a fast DES implementation.

In this paper, we study the use of composite field techniques for Galois Field arithmetic in the context of the Rijndael cipher. We show that very substantial gains in performance can be obtained through such an approach. We obtain a very compact gate circuit for Rijndael and use its design to illustrate the trade-offs associated with design choices such as field polynomials and representations. We use our circuit design to obtain a fast and light software implementation of Rijndael for wide-word architectures. The performances of both hardware as well as software implementations show large gains in comparison with previously reported performance figures.

The rest of this paper is organised as follows. In Section 2 we detail the mapping of Galois Field operations to composite field arithmetic. Section 3 outlines the data-slicing technique for realising a highly parallel software implementation from a circuit design. In Section 4 we describe the mapping of Rijndael operations into a particular class of composite fields. The selection of field polynomials and representations and the associated optimizations are discussed in Sections 5 and 6, and in Section 7 we present our results and a comparison with previously reported performance figures for Rijndael. Drawings of our Rijndael encryption circuit are included in the Appendix.

## 2 GF Arithmetic and Composite Fields

Composite fields are frequently used in implementations of Galois Field arithmetic [9–11]. In cases where arithmetic operations rely on table lookups, subfield arithmetic is used to reduce lookup-related costs. This technique has been used to obtain relatively efficient implementations for specific operations such as multiplication, inversion and exponentiation. Much of this work has been aimed at implementation of channel codes. The object has usually been to obtain better software implementations by using smaller tables through subfield arithmetic. Applications to hardware design (such as [10]) have been relatively infrequent.

Our techniques are directed at both hardware and software implementations. We take advantage of the efficiency obtainable by the use of subfield arithmetic, not merely in the matter of smaller tables but the overall low-level (gate count) complexity of various arithmetic operations. The computation and comparison of such gains and cost is dependent upon across several parameters – the overhead of mapping between the original and the composite field representations, the nature of the underlying computation and its composition in terms of the relative frequency of various arithmetic operations, and in case of software implementations the constraints imposed by the target architecture and its instruction set. Based on these parameters we select the appropriate field and representation to optimize a hardware circuit design. As we shall see, there can be several objectives for this optimization, such as critical path lengths and gate counts, depending upon the overall design goals. The circuit design obtained can then be used to obtain parallelism in a software implementation by means of slicing techniques.

After [11], the two pairs $\{GF(2^n), Q(y)\}$ and $\{GF((2^n)^m), P(x)\}$ constitute a *composite field* if $GF(2^n)$ is constructed from $GF(2)$ by $Q(y)$ and $GF((2^n)^m)$ is constructed from $GF(2^n)$ by $P(x)$, where $Q(y)$ and $P(x)$ are polynomials of degree $n$ and $m$ respectively. The fields $GF((2^n)^m)$ and $GF(2^k)$, $k = nm$, are isomorphic to each other. Since the complexity of various arithmetic operations differs from one of these fields to the other, we can take advantage of the isomorphism to map a computation from one to the other in search of efficiency. For a given underlying field $GF(2^k)$, our gains depend on the choice of $n$ and $m$ as well as of the polynomials $Q(y)$ and $P(x)$.

While we restrict our description to composite fields of the type $GF((2^n)^m)$, it is easy to see that the underlying techniques are fully general and can be used be any composite field.

## 3 Slicing Techniques

*Bit-slicing* is a popular technique [2] to make use of the inbuilt parallel-processing capability of a wide-word processor. Bit-slicing regards a $W$-bit processor as a SIMD parallel computer capable of performing $W$ parallel 1-bit operations simultaneously. In this mode, an operand word contains $W$ bits from $W$ different instances of the computation. Initially, $W$ different inputs are taken and arranged so that the first word of the re-arranged input contains the first bit from each of the $W$ inputs, the second word contains the second bit from each input, and so forth. The resulting bit-sliced computation can be regarded as simulating $W$ instances of the hardware circuit for the original computation. Indeed, a bit-sliced computation is designed by first designing a hardware circuit and then simulating it using $W$-bit registers on the rearranged input described above.

A bit-sliced implementation corresponding to an $N$-gate circuit requires $N$ instructions to carry out $W$ instances of the underlying computation, or $N/W$ instructions per instance. This can be particularly efficient for computations which are not efficiently supported by the target architecture. Consider for instance $GF(2^8)$ multiplication on AltiVec [4]. The straightforward implementation uses three table-lookups and one addition modulo 255. 16-parallel lookups in a 256-entry table can be performed on AltiVec in 20 instructions. Thus, a set of 128 multiplications would require 488 instructions. In comparison, our 137-gate multiplication circuit translates into a bit-sliced implementation that can perform 128 multiplications in 137 instructions!

The above computation ignores the cost of ordering the input in bit-sliced fashion and doing the reverse for the output. To evaluate the trade-off correctly that cost must be taken into account. Clearly, that cost will depend on the target instruction set.

However, it is possible to think of scenarios in which a particular operation may be efficiently supported in an architecture. For example, if the AltiVec architecture were to provide an instruction for 16 parallel $GF(2^8)$ multiplications which use the underlying field polynomial of interest to us (a hypothetical but nonetheless technically feasible scenario since the critical path of the multiplication circuit is only six gates deep), then a direct computation would require only eight instructions, compared to the 137 required by the bit-sliced version.

Now consider $GF(2^{16})$ multiplications on this hypothetical version of the AltiVec architecture. It is easy to see that the most efficient computation is neither a direct one, nor a bit-sliced version, but a *byte-sliced* computation, in which each $GF(2^{16})$ multiplication is mapped to a small number of $GF(2^8)$ operations, which are efficiently supported by the architecture in question. In general, the right "slice" to use would depend on the target architecture.

### 3.1 Encrypting Without Chaining

Our Rijndael implementation processes 128 blocks of data in parallel. Traditionally, such a scheme would be regarded as more useful for decryption than for encryption, since encryption is usually performed in inherently sequential modes such as Cipher Block Chaining or CBC [17–19]. The well-known CBC [17–19] is used as a defence against replay attacks [12]. In the CBC mode of encryption parallel blocks would not be available for encryption except where data from many streams is sought to be encrypted in parallel.

However, a new parallelizable variant of CBC [7] removes this limitation and makes it possible to use CBC encryption without the usual sequentiality. This makes it possible to utilize the high throughput rates of our implementation in conjunction with the popular CBC mode.

## 4 Rijndael in a Composite Field

Rijndael involves arithmetic on $GF(2^8)$ elements. In a straightforward implementation, inverse, multiplication and substitution are likely to be the operations that determine the overall complexity of the implementation. The most common approach is to use table lookups for these operations. By mapping the operations into a composite field we are able to obtain both a small circuit in case of a hardware implementation as well as smaller instruction counts and table sizes in case of software implementations.

For our Rijndael implementation, we work in the composite field $GF((2^4)^2)$. We selected the field polynomial $Q(y) = y^4 + y + 1$ for $GF(2^4)$. For $P(x)$ we considered the choice of all primitive polynomials of the form $P(x) = x^2 + x + \lambda$ where $\lambda$ is an element of $GF(2^4)$. There are four such polynomials, for each of which there are seven different transformation matrices to consider, one corresponding to each possible choice of basis. The criterion used by us to compare various choices is the gate count of the resulting Rijndael circuit implementation.

Rijndael operations translate to the composite field representation as follows. $\mathbf{H}$ denotes the mapping from $GF(2^8)$ to $GF((2^4)^2)$, and $\mathbf{T}$ the corresponding transformation matrix — that is, $\mathbf{H}(x) = \mathbf{T}x$. $S$ is a $4 \times 4$ matrix (the *state*) on which all operations are performed.

- ByteSub transformation : This has essentially two sub-steps:
  1. $P_{ij} = (S_{ij})^{-1}$. In the composite field, $\mathbf{H}(P_{ij}) = (\mathbf{H}(S_{ij}))^{-1}$.
     The calculation of an inverse is as follows. Every $A \in GF((2^4)^2)$ can be represented as $A = a_0 + \beta a_1$ where $\beta^2 + \beta + \lambda = 0$, and $a_0, a_1 \in GF(2^4)$. The inverse is $B = A^{-1} = b_0 + \beta b_1$, $b_0, b_1 \in GF(2^4)$ , such that $b_0 = (a_0 + a_1)\Delta^{-1}$ and $b_1 = a_1\Delta^{-1}$, where $\Delta = a_0(a_0 + a_1) + \lambda a_1^2$.
  2. $Q_{ij} = \mathbf{A}P_{ij} + \mathbf{c}$ where $\mathbf{A}$ is a fixed $8 \times 8$ matrix and $\mathbf{c} \in GF(2^8)$. In the composite field, $\mathbf{H}(Q_{ij}) = \mathbf{H}(\mathbf{A}P_{ij}) + \mathbf{H}(\mathbf{c}) = \mathbf{TA}P_{ij} + \mathbf{H}(c)$ $= \mathbf{TAT}^{-1}\mathbf{H}(P_{ij}) + \mathbf{H}(\mathbf{c})$.
- ShiftRow transformation : This step is independent of representation.
- MixColumn transformation : This involves essentially the computation $P_{ij} = a_1S_{1j} + a_2S_{2j} + a_3S_{3j} + a_4S_{4j}$, where $(a_1, a_2, a_3, a_4)$ is a permutation of $(01, 01, 02, 03)$. In the composite field,
  $\mathbf{H}(P_{ij}) = \mathbf{H}(a_1)\mathbf{H}(S_{1j}) + \mathbf{H}(a_2)\mathbf{H}(S_{2j}) + \mathbf{H}(a_3)\mathbf{H}(S_{3j}) + \mathbf{H}(a_4)\mathbf{H}(S_{4j})$.
  The following observations are useful in the implementation -
  - If $x \in GF((2^4)^2)$ then $\mathbf{H}(01) \times x = x$ as the identity element is mapped to the identity element in a homomorphism.
  - $\mathbf{H}(03) = \mathbf{H}(02) + \mathbf{H}(01)$.
- Round Key addition : The operation is $P = S + K$ where $K$ is the round key. In the composite field, $\mathbf{H}(P) = \mathbf{H}(S) + \mathbf{H}(K)$. Addition is simply an EXOR in either representation.

The mapping of the arithmetic to the composite field together with judicious choice of the field polynomial gives us a substantially smaller circuit, as we shall see in Section 7.

## 5 Optimizations

All operations in the Rijndael block cipher are in $GF(2^8)$. As outlined in section 4, some of these $GF(2^8)$ operations have relatively inefficient gate circuit implementations and can be implemented more efficiently in some isomorphic composite field. One overhead in the using subfield arithmetic is the cost of the conversion from the original to the composite field and vice-versa. To illustrate, consider our Rijndael implementation, which uses subfield arithmetic. The cost of the transformations is dependent on the choice of the composite field. We describe below a method by which an *efficient*[1] transformation matrix from the set $\mathcal{T} = \{\mathbf{T}_0, \mathbf{T}_1, ....\}$ of valid transformation matrices can be chosen.

Let $\mathcal{C}(\theta)$ denote the cost of the operation $\theta$, which in the present case is taken to be the gate count of the circuit implementation of $\theta$. Depending upon design objectives and application, there can be alternative cost measures, such as the depth of the critical path, for instance. Let $\mathcal{W}(x)$ denote the *hamming weight* of $x$, i.e., the number of 1s in the polynomial representation of $x$.

The aim is to find $\mathbf{T}^*$, the most *efficient* tranformation, and the corresponding choice of composite field. This is the composite field which minimizes the gate count of the Rijndael circuit implementation. Note that while comparing the cost for different transformations, we need to consider only those Rijndael operations whose costs are dependent upon the choice of composite field. The relevant operations are those which involve $\lambda$ or the conversion matrices ($\mathbf{T}$ and $\mathbf{T}^{-1}$).

The costs of different operations are:

- Transform : This step involves computing $\mathbf{H}(\mathrm{S})$.
  Thus, $\mathcal{C}(\text{Transform}) = 16 \times \mathcal{C}(\mathbf{T}.x)$.
- ByteSub transformation : As noted earlier, this step consists of an inverse calculation and an affine transform –
  1. $P_{ij} = (S_{ij})^{-1}$. The only operation who cost depends on the choice of field is the calculation of $\lambda a_1^2$. So $\mathcal{C}(inverse) = 16 \times \mathcal{C}(\lambda.x)$.
  2. $Q_{ij} = \mathbf{A}P_{ij} + \mathbf{c}$, or, in the composite field, $\mathbf{H}(Q_{ij}) = \mathbf{H}(\mathbf{A}P_{ij}) + \mathbf{H}(\mathbf{c})$
     $= \mathbf{TA}P_{ij} + \mathbf{H}(c)$
     $= \mathbf{TAT}^{-1}\mathbf{H}(P_{ij}) + \mathbf{H}(\mathbf{c})$.
     Thus $\mathcal{C}(affine) = 16 \times (\mathcal{C}(\mathbf{B}.x) + \mathcal{W}(\mathbf{H}(c))\ )$, where $\mathbf{B} = \mathbf{TAT}^{-1}$. [2].
- ShiftRow transformation : This step does not require any computation.
- MixColumn transformation :
  As note earlier, this step is the computation
  $\mathbf{H}(P_{ij}) = \mathbf{H}(a_1)\mathbf{H}(S_{1j}) + \mathbf{H}(a_2)\mathbf{H}(S_{2j}) + \mathbf{H}(a_3)\mathbf{H}(S_{3j}) + \mathbf{H}(a_4)\mathbf{H}(S_{4j})$.

  Since $\mathbf{H}(01).x = x$, $\mathcal{C}(mixClm) = 16 \times (\mathcal{C}(\mathbf{H}(02).x) + \mathcal{C}(\mathbf{H}(03).x))$.
- Round Key addition : The computation is $\mathbf{H}(P) = \mathbf{H}(S) + \mathbf{H}(K)$, so $\mathcal{C}(addKey)$ $= 16 \times \mathcal{C}(\mathbf{T}.x)$.
- Inverse Transform : $\mathcal{C}(invTranform) = 16 \times \mathcal{C}(\mathbf{T}^{-1}.x)$.

$\mathbf{T}^*$ depends upon whether a pipelined (unrolled loop) or iterative (loop not unrolled) Rijndael circuit is to be obtained. The former offers superior performance compared to the latter at the cost of a larger gate count.

[1] In terms of the Rijndael gate-circuit implementation.
[2] Note that $\mathcal{W}(\ \mathbf{H}(c)\ )$ is the number of *not* gates required to implement $\mathbf{H}(\mathbf{c})$+x , where $x \in GF((2^4)^2)$.

The criterion for the best transformation can be represented as follows:

$$\mathbf{T}^* = \arg\min_{\mathbf{T}_i \in \mathcal{T}} ( \quad \mathcal{C}(transform) + n \times \mathcal{C}(inverse) + n \times \mathcal{C}(affine)$$
$$+ m \times \mathcal{C}(mixClm) + (n+1) \times \mathcal{C}(addKey) + \mathcal{C}(invTransform)).$$

i.e.

$$\mathbf{T}^* = \arg\min_{\mathbf{T}_i \in \mathcal{T}} ( \ (n+2) \times \mathcal{C}(\mathbf{T}.x) + n \times \mathcal{C}(\lambda.x) + n \times (\mathcal{C}(\mathbf{B}.x) + \mathcal{W}(\mathbf{H}(c)) \ )$$
$$+ m \times ( \ \mathcal{C}(\mathbf{H}(02).x) \ + \ \mathcal{C}(\mathbf{H}(03).x) \ ) + \mathcal{C}(\mathbf{T}^{-1}.x) \ \ )$$

where $m$ and $n$ are both 1 for an iterative circuit, and $\mathcal{R}$ and $\mathcal{R}-1$ respectively for a pipelined circuit, where $\mathcal{R}$ is the number of *rounds* as specified in the Rijndael cipher.

Based on these considerations, we selected the polynomial $P(x) = x^2 + x + \omega^{14}$. That is, we chose $\lambda$ to be $\omega^{14}$ where $\omega$ is the primitive element of $GF(2^4)$. The following transformation matrix maps an element from $GF(2^8)$ to the corresponding element in the chosen composite field:

$$\begin{pmatrix} 1\,0\,1\,0\,0\,0\,0\,0 \\ 1\,0\,1\,0\,1\,1\,0\,0 \\ 1\,1\,0\,1\,0\,0\,1\,0 \\ 0\,1\,1\,1\,0\,0\,0\,0 \\ 1\,1\,0\,0\,0\,1\,1\,0 \\ 0\,1\,0\,1\,0\,0\,1\,0 \\ 0\,0\,0\,0\,1\,0\,1\,0 \\ 1\,1\,0\,1\,1\,1\,0\,1 \end{pmatrix}$$

## 6 Finding a Transform

A method for generating a transformation matrix to map elements of $GF(2^k)$ to $GF((2^n)^m)$ can be found in literature, for instance in [11] , for the case where all the field polynomials involved are primitive polynomials. However, in the case of Rijndael the field polynomial is $R(z) \ = \ z^8 + z^4 + z^3 + z + 1$ which is an irreducible polynomial but not primitive. Since the fields involved are small, we use an exhaustive search method that can find the transformation in question in case $R(z)$ is irreducible but not primitive. The basic idea is to map $\alpha$, the primitive element of $GF((2^n)^m)$ to $\gamma$, a primitive element of $GF(2^n)$, such that field homomorphism holds.

The algorithm is composed of the following three steps —

1. Get a primitive element $\gamma$ of $GF(2^k)$ and map $\alpha^i$ to $\gamma^i$ for $i \in [0..(2^k-1)]$. Note that this step preserves the multiplicative group homomorphism — for any $i, j \in [0..(2^k-1)]$, $\alpha^i \times \alpha^j = \alpha^{i+j}$ maps to $\gamma^i \times \gamma^j = \gamma^{i+j}$.
2. Perform the following check — $\forall i \in [0..(2^k-1)]$, if $\alpha^r = \alpha^i + 1$ then $\gamma^r = \gamma^i + 1$. If so then we have the required mapping; else repeat this step for the next primitive element.

This is to verify additive group homomorphism, which requires that $\forall i,j \in [0..(2^k-1)]\ \alpha^t = \alpha^i + \alpha^j \Rightarrow \gamma^t = \gamma^i + \gamma^j$. That is, $\alpha^t = \alpha^i \times (1+\alpha^{j-i}) \Rightarrow \gamma^t = \gamma^i \times (1+\gamma^{j-i})$.

Multiplicative group homomorphism implies that it is sufficient to verify whether $\forall i,j \in [0..(2^k-1)],\ \alpha^{t-i} = 1+\alpha^{j-i} \Rightarrow \gamma^{t-i} = 1+\gamma^{j-i}$.

3. The matrix,$\mathbf{T^{-1}}$ is obtained by placing in the $i^{th}$ column the element $\mathbf{H}(2^i)$ in the standard basis representation[3] for all $i$.

## 7 Performance

Our performance figures reported below are for Rijndael encryption circuit and software, which assume key size of 128 bits.

Our core circuit for Rijndael encryption contains less than four thousand gates. For the purpose of comparison, we report numbers based upon a circuit with 520 I/O pins that uses multiple cores in parallel.

**Table 1.** Circuit Performance Figures

| | Transistor/Gate count | Cycles/block | Throughput |
|---|---|---|---|
| Ichikawa[4]et al.[6] | 518K gates | ? | 1.95 Gbps |
| Weeks et al.[13] | 642K transistors | ? | 606 Mbps |
| Elbirt et al.[5] | ? | 6 | 300Mbps@14MHz |
| (256-pin I/O) | ? | 2.1 | 1.938 Gbps@32 MHz |
| Our hardware circuit | 256K gates | 0.5 | 7.5 Gbps@32 MHz |
| | using 32 parallel cores (iterated) of 4k gates each and 252 gate levels | | |

Table 2 lists cycle counts and target architectures for various reported implementations. In our case, the numbers apply to any architecture that can support bitwise AND and EXOR in addition to LOAD and STORE operations. The three numbers we report correspond to architectures with effective datapath widths (number in parenthesis) of 256 bits, 384 bits and 512 bits respectively (this is perhaps the interesting range of architectures today). The cycle count goes down with increasing datapath width.

It may be mentioned that no minimization or sysnthesis tools were used for our ciruit — the only minimization used is in the sense of section 5. The only gates in our circuit are XOR, AND and NOT gates.

---

[3] Here $2^i$ denotes the element whose bit representation contains all 0s except a 1 in the $i$th place. For example for $n=4, m=2$, $2^4$ is the element 00010000, i.e., $\alpha$.

[4] This circuit performs encryption as well as decryption.

**Table 2.** Cycle counts per block for software implementations

| | | | |
|---|---|---|---|
| Worley et al[16] | 284 (Pentium) | 176 (PA-RISC) | 124 (IA-64) |
| | Requires an 8KB table | | |
| Weiss et al.[14] | 210 (Alpha 21264) | | |
| Wollinger et al.[15] | 228 (TMS320C6x) | | |
| Aoki et al.[1] | 237 (Pentium II) | | |
| Our bit-sliced software[5] | 170 (256b) | 119 (384b) | 100 (512b) |
| | Requires only EXOR, AND, L/S, and 2KB table | | |


## 8 Acknowledgements

The authors would like to thank Christof Paar and Gaurav Aggarwal for helpful discussions.


## References


1. Kazumaro Aoki and Helger Lipmaa, "Fast Implementations of AES candidates". In *Proc. Third AES Candidate Conference*, April 13-14, 2000. `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html`
2. Eli Biham, "A Fast New DES Implementation in Software". In *Proc. Fast Software Encryption 4,1997.* `http://www.cs.technion.ac.il/~biham/publications.html`
3. Joan Daemen and Vincent Rijmen, "AES Proposal: Rijndael". `http://www.esat.kuleuven.ac.be/~rijmen/rijndael`.
4. Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung and Hunter Scales, "AltiVec Extension to PowerPC Accelerates Media Processing". In *IEEE Micro*, March-April 2000, pp85-95.
5. AJ Elbirt, W Yip, B Chetwynd and C Paar, "An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists". In *Proc. Third AES Candidate Conference*, April 13-14, 2000. `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html`
6. Tetsuya Ichikawa, Tomomi Kasuya and Mitsuru Matsui, "Hardware Evaluation of the AES Finalists". In *Proc. Third AES Candidate Conference*, April 13-14, 2000. `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html`
7. Charanjit S. Jutla, "Encryption Modes with Almost Free Message Integrity". Manuscript.
8. Rudolf Lidl and Harald Niederreiter, *Introduction to finite fields and their applications.* Cambridge University Press, Cambridge, Ma., 1986.
9. Edoardo D. Mastrovito, *VLSI Architectures for Computations in Galois Fields.* PhD Thesis, Dept. of EE, Linköping University, Linköping, Sweden 1991.
10. Christof Paar and Pedro Soria-Rodriguez, "Fast Arithmetic Architectures for Public-Key Algorithms over Galois Fields $GF((2^n)^m)$". In *Proc. EUROCRYPT '97.*


[5] The performance numbers include the cost of ordering the input in bit-sliced fashion and the reverse for the output.

11. Chirstof Paar, *Efficient VLSI Architectures for Bit-Parallel Computations in Galois Fields.* PhD Thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994. `http://www.ece.wpi.edu/Research/crypt/theses/paar_thesispage.html`.
12. Bruce Schneier, *Applied Cryptography*, John Wiley and Sons,1996.
13. Bryan Weeks, Mark Bean, Tom Rozylowicz and Chris Ficke, " Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithm". In *Proc. Third AES Candidate Conference*, April 13-14, 2000. `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html`
14. Richard Weiss and Nathan Binkert "A comparison of AES candidates on the Alpha 21264". In *Proc. Third AES Candidate Conference*, April 13-14, 2000. `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html`
15. Thomas J. Wollinger, Min Wang, Jorge Guajardo and Christof Paar, "How Well Are High-End DSPs suited for AES Algorithms ?" In *Proc. Third AES Candidate Conference*, April 13-14, 2000. `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html`
16. John Worley, Bill Worley, Tom Christian and Christopher Worley, "AES Finalists on PA-RISC and IA-64: Implementations & Performance". In *Proc. Third AES Candidate Conference*, April 13-14, 2000. `http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html`
17. "American National Standard for Information Systems — Data Encryption Algorithm — Modes of Operation". ANSI X3.106, American National Standards Institute, 1983.
18. "Information processing — Modes of operation for a 64-bit block cipher algorithm". ISO 8372, International Organisation for Standardisation, Geneva, Switzerland, 1987.
19. "DES modes of operation". NBS FIPS PUB 81, National Bureau of Standards, U.S. Department of Commerce, 1980.
20. `http://www.nist.gov/public_affairs/releases/g00-176.htm`, US Commerce Department Press Release.

## Appendix: Our Rijndael Ciruit

Presented below are drawings of our gate circuit for Rijndael encryption.The figures appear in the order of the level of detail in them – Figure 1 showing the high level view of our circuit.

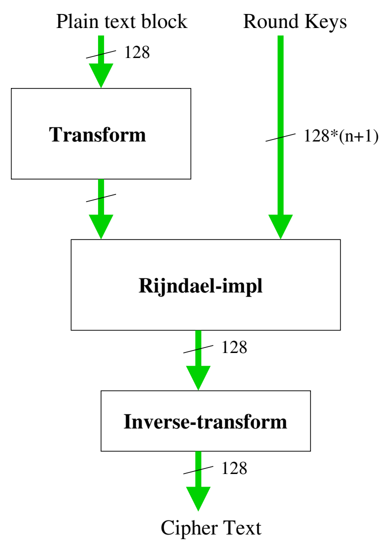


**Fig. 1.** This figure contains the high level view of the Rijndael encryption circuit. The *transform* function consists of 16 parallel ciruits for $\mathbf{T}^{*}.x$, where $x \in GF(2^8)$ and $\mathbf{T}^{*}$ is the matrix to convert elements from $GF(2^8)$ to elements of composite field as decided by section 5. Similarly, Inverse-transform consists of 16 parallel circuits for $(\mathbf{T}^{*})^{-1}.x$. Circuits for the multiplication of a constant matrix with a vector are obtained from the method given in [11]

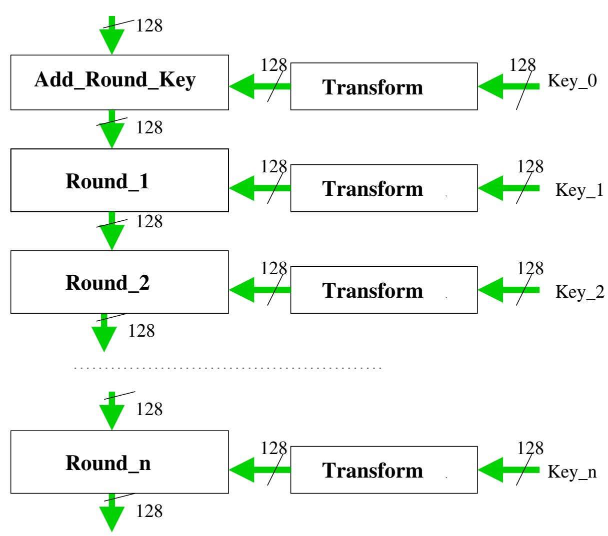


**Fig. 2.** This figure describes the *rijndael-impl* block in Figure 1

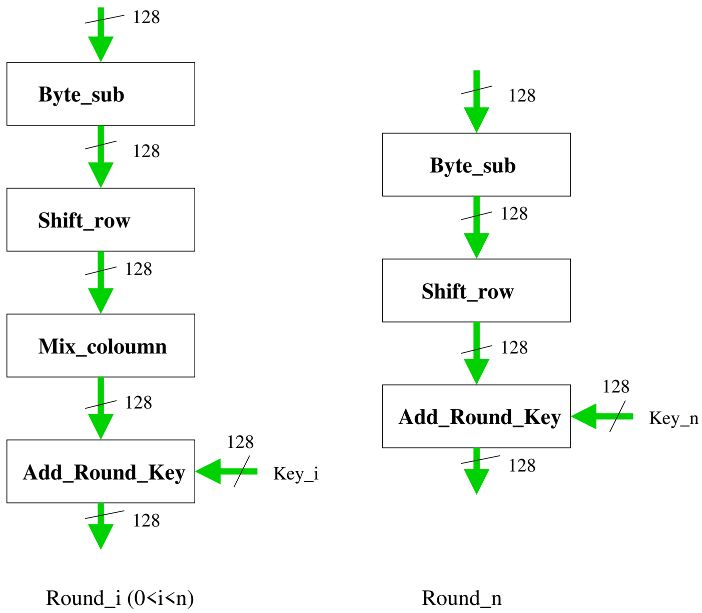


**Fig. 3.** This figure shows the composition of each round. Note that in our implementation, $n$=10. *Shift_row* does not require any gate. *Add_Round_Key* is simply the EXOR of the corresponsing bits of the two inputs

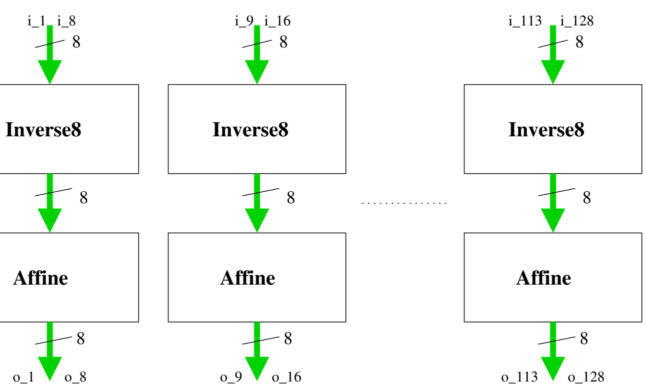


**Fig. 4.** This figure shows the implementation of the *Byte_sub* operation. *Affine* has 16 parallel ciruits for calculating $\mathbf{T}^{*}A(\mathbf{T}^{*})^{-1}.x+\mathbf{H}(\mathbf{c})$

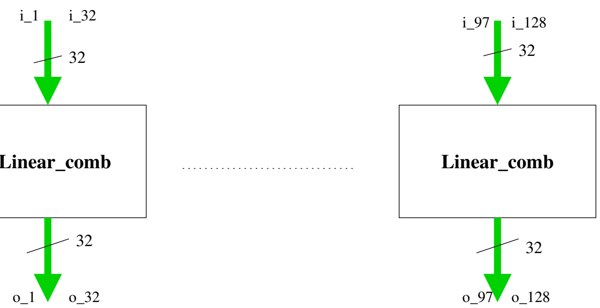


**Fig. 5.** *mix_column*

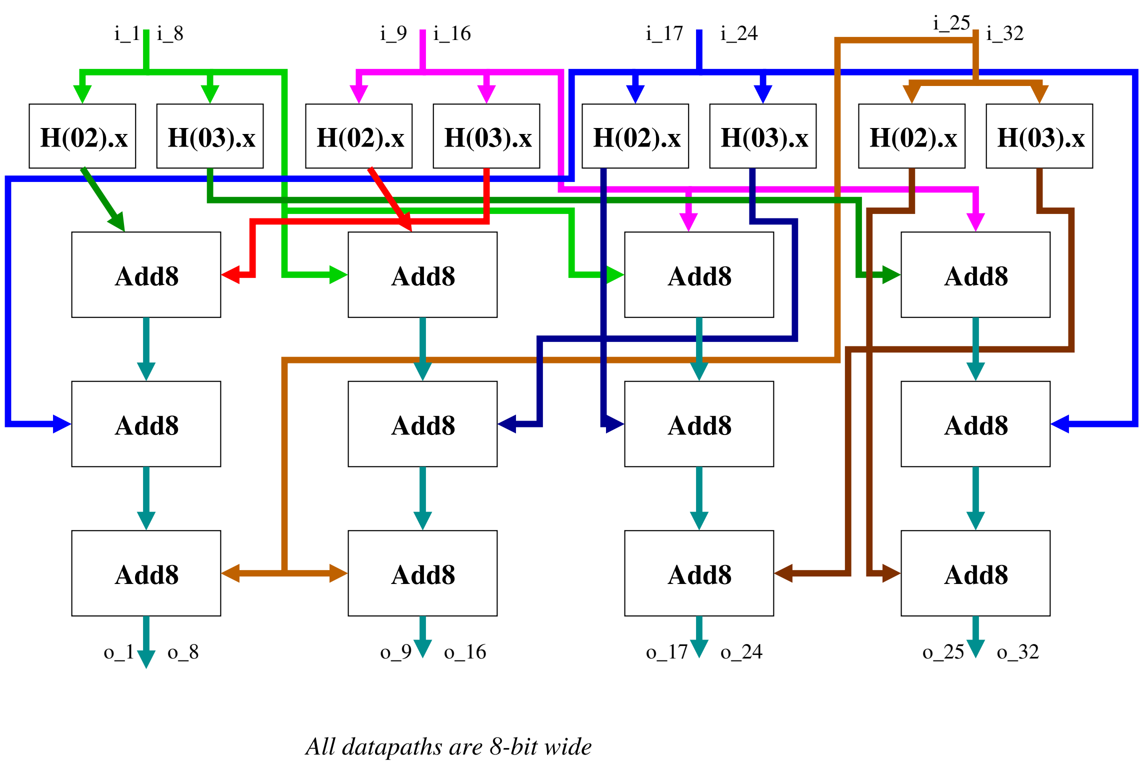


**Fig. 6.** This figure describes the *linear_comb* operation from Figure 5. *Add8* is simply the EXOR of the corresponding bits of the two inputs

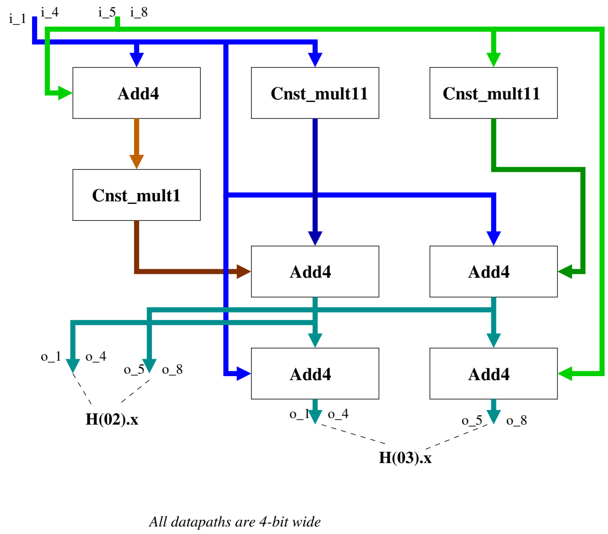


**Fig. 7.** This figure shows the ciruits for calculating $\mathbf{H(02)}.x$ and $\mathbf{H(03)}.x$; *Add4* is simply the EXOR of the corresponding bits of the two inputs. *Const_multi* evaluates the constant multiplication $\omega^i.x$, where $\omega$ is the primitive element of $GF(2^4)$ and $x \in GF(2^4)$. These ciruits have been obtained from [11]

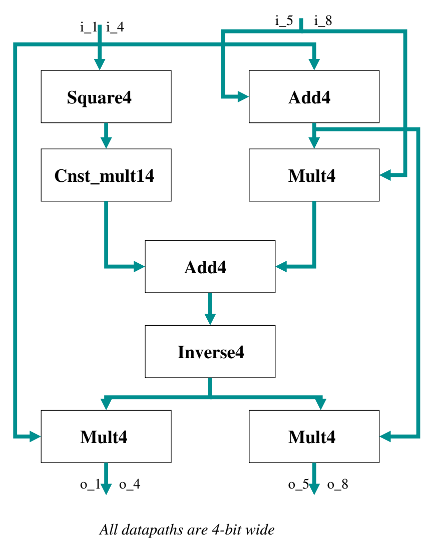


**Fig. 8.** This figure 8 shows the *Inverse8* operation in Figure 4. *Square4* is from [11]; *Mult4* and *Inverse4* are from [9]