Efficient Hardware Calculation of Inverses in $GF(2^8)$

R. W. Ward, Dr. T. C. A. Molteno[1]

Physics Department
University of Otago
Box 56, Dunedin, New Zealand

[1]Email: tim@physics.otago.ac.nz

Abstract: Galois Fields are an important in many areas including cryptography and error correcting codes. We consider various approaches to calculate multiplicative inverses of elements in the Galois Field $GF(2^8)$ in hardware, as is used in the Rijndael encryption algorithm. We compare speed vs. implementation area for these approaches.

Keywords: Inverse, Galois Field, Rijndael, AES, Encrpytion

Citing this work: Ward R.W., Molteno, T.C.A. "Efficient Hardware calculation of Inverses in $GF(2^8)$", Proceedings of ENZCon 2003. Hamilton New Zealand (2003)

1. INTRODUCTION

Galois Fields are a mathematical structure that have important uses in such fields as cryptography. They are used in the S-box of the Rijndael encryption algorithm [2], which has been adopted as the Advanced Encryption Standard (AES)[1], and have other cryptographic applications, including cryptology using elliptic curves [4], and are also used in error correction, such as the Reed-Solomon codes [5] used for error corrections on Compact Discs.

We consider a specific problem associated with Galois Fields, that of finding the multiplicative inverse. This is an important component of the Rijndael Algorithm as shown in Figure 1 We investigate several algorithms for calculating the inverse and compare the speed/area tradeoff for implementations of these algorithms as implemented on a Complex Programmable Logic Device as a way of estimating their speed/area on silicon. The results are shown in Table 1 and Figure 2. We finally illustrate their use under different constraints.

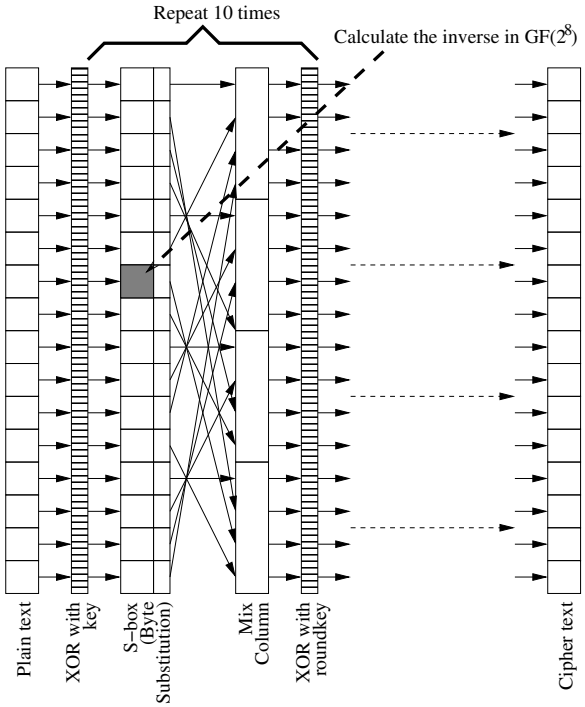

Figure 1. Overview of the Rijndael Encryption Algorithm

2. PROBLEM SPECIFICATION

2.1 Mathematical Background

A finite field, or Galois field is a tuple, $(S, +, \cdot)$ where the finite set S forms an abelian group under the operation $+$ (addition), and the elements of S except 0 form an abelian group under $\cdot$ (multiplication), and the distributive property holds: $a \cdot (b+c) = a \cdot b + a \cdot c$.

Galois fields of size 2^m (denoted $GF(2^m)$) are an attractive structure to use for computing applications because they map well into binary representations. There is also a property that there is only a single Galois field any prime power, so for a particular m, two representations of $GF(2^m)$ are guaranteed to be

[1]http://csrc.nist.gov/CryptoToolkit/aes/

TABLE 1. Inverse in $GF(2^8)$ costs

Method	Representation	Transistor count	Time (ns)	Order
Lookup Table	any	6180	50	$O(2^m m)$
Power of Primitive Element	any	854	5939 †	$O(m^2)$
Extended Euclid	polynomial	10714	755	$O(m^3)$
Exponentiation (looped)	any	1828	708 †	$O(m^2)$
Exponentiation (unrolled)	any	5272	443	$O(m^3)$
Field Factoring	special ‡	1416	164	n/a

† This method clocked
‡ Costs include translation overhead

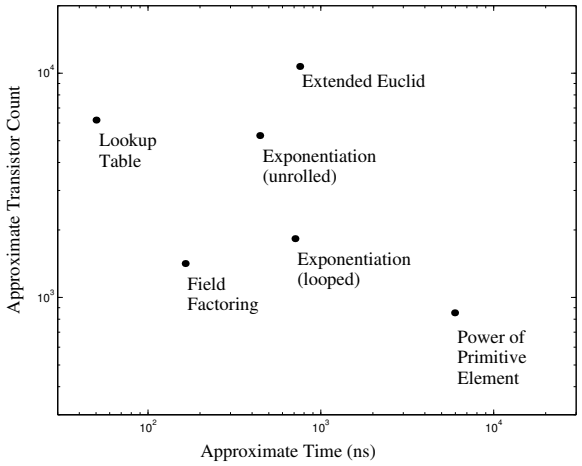

Figure 2. Inverse in $GF(2^8)$ costs

isomorphic.

Each $x \in GF(2^m)$ has a unique inverse element x^{-1} such that $x^{-1} \cdot x = x \cdot x^{-1} = 1$.

There are two main ways of representing elements of $GF(2^m)$ in a binary structure.

2.1.1 Polynomial Representation. A word $b_{m-1}b_{m-2}\ldots b_2b_1b_0$ can be considered as a polynomial $b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \ldots + b_2x^2 + b_1x + b_0$ where the coefficients b_i are elements of $GF(2)$ (i.e. 0 or 1).

Addition is performed by piecewise addition of the terms, and addition in $GF(2)$ is simply the XOR operation, and subtraction is the same as additions, so addition and subtraction can be done by bitwise XOR.

Normal polynomial multiplication cannot be used for multiplication in the Galois field, as it may result in a polynomial of order greater than $m-1$, however we can use multiplication of polynomials modulo an irreducible binary polynomial of degree m. A polynomial is irreducible if it has no divisors other than 1 and itself.

For example, $GF(2^8)$, the polynomial $x^8+x^4+x^3+x+1$ is irreducible.

The Rijndael Encryption algorithm uses a polynomial representation of $GF(2^8)$ with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

2.1.2 Normal Basis Representation. Another way of representing elements of $GF(2^m)$ is to use the property that $GF(2^m)$ has at least one primitive element α such that any element of $GF(2^m)$ can be represented as a power of α. Also, $\alpha^m = \alpha^0 =' 1'$, the identity element, so the elements of $GF(2^m)$ are the set $\{\alpha^{2^m-1}, \alpha^{2^m-2}, \ldots, \alpha^2, \alpha^1, \alpha^0\}$.

A word $b_{m-1}b_{m-2}\ldots b_2b_1b_0$ can be considered as the element $b_{m-1}\alpha^{2^{m-1}} + b_{m-2}\alpha^{2^{m-2}} + \ldots + b_2\alpha^4 + b_1\alpha^2 + b_0\alpha^1$.

Addition is again preformed by piecewise addition of the terms and subtraction is the same as addition, so again addition and subtraction can be done by bitwise XOR.

Multiplication is a little more complicated, and is described in [4]. One property of normal basis representation is that the operation x^2 can be performed very cheaply by a bitwise rotate left.

For some m, $GF(2^m)$ has a characterization known as an *optimal* normal basis, where multiplications have minimum complexity. There is an optimal normal basis for $GF(2^4)$, but not for $GF(2^8)$.

2.2 The problem

We are considering an implementation suitable for a Rijndael S-box (the Byte Substution step in Figure 1), so we look at finding the multiplicative inverse of elements in $GF(2^8)$, where elements are represented in polynomial representation, with an irreducible polynomial of $x^8 + x^4 + x^3 + x + 1$

For the Rijndael S-box, we also requires 0 (which doesn't have an inverse) to map to 0. Where the algorithms described don't already do that, this is noted and checked for.

Also, to avoid any timing attacks, all methods should take the same length of time independent of the input. All our implementations do this.

2.3 Cost calculation

2.3.1 Speed. There are two components to speed: throughput and latency. For this project, we consider that only one inverse calculation is being done at a time so we just consider time T taken to perform the operation. The inverse is then T, the throughput is $1/T$. It is possible that for some solutions, pipelining would be able to improve the throughput (possibly at the cost of some extra latency), but that is beyond the scope if this paper.

Also, there are two types of logic to consider. In *combinatorial* logic, there are no loops or buffers in the circuit - signals are fed in one direction through a structure of gates to arrive at a different set of signals at the other end. The time taken be determined simply by the transition time of the slowest signal through the circuit, which we approximate by gate depth: $T = cd$, where d is the gate depth, and c is some constant. Combinatorial logic has the advantage that it can often be turned into synchronous logic and pipelined if high throughput is desired.

In *synchronous* logic, there is a clock, and some transformations happen every clock tick, with the transformations happening over a number of clock ticks returning our final result. For this type of logic, the time taken will be $T \geq n(cd + v)$, where d is the gate depth for the operations, c is some constant, v is some overhead for the latching of results, and n is the number of clocks. This is a lower bound, because we may not have an available clock at exactly the right speed. If an operation needs several transformations in a loop, it cannot be pipelined.

2.3.2 Area. Area is determined principally by the number of transistors the circuit would take to implement in silicon. Calculating this requires special tools, so we approximate by synthesizing the circuit for a CPLD using Xilinx[2] WebISE tools to synthesize for a CPLD, looking at the cell usage in the synthesis report, and breaking the results down to latches and one or two input gates. An approximation of the area good enough for our needs can then be calculated by summing these, with some weighting constant being multiplied by the count of each gate or latch type.

We use transistor count as an approximation for area, and use estimates for transistor counts of various components [**?**], as described in Table 2.

2.3.3 Scaling to $m \neq 8$. Although this is paper is concerned with calculation of inverses in $GF(2^8)$, some of the techniques used will apply more generally to $GF(2^m)$. We note the order of each approach as an indication indicates how well it scales.

[2]www.xilinx.org

TABLE 2. Appoximate transistor costs for simple logic

Gate or component	Transistor count
INV	2
AND	4
OR	4
NAND	4
NOR	4
XOR	12
Latch	20

2.3.4 Composition considerations. In some calculation approaches, efficiencies can be gained by composing them with preceding or following operations. This will be noted.

2.3.5 Calculation methodology. We calculate cost by using Xilinx WebISE tools to synthesize for a Digilent CoolRunner xcr3152xl-12pq208 CPLD, then looking at the gate count from the low-level synthesis report, expressing the results in terms of two input gates. We can also fit and look at the timing to get some idea of the path length, so timing will be given in the number of nanoseconds the tools report the operation will take.

3. INVERSE CALCULATION APPROACHES

There are many approaches to calculating the inverse of an element in $GF(2^8)$. The implementations of these have different area or gate depth requirements, and some require to be clocked. The particular approach used will depend on the problem.

4. LOOKUP TABLE

This is done using a case statement in the coding language, which gets coded into a small ROM, and is the method often suggested in the literature for such a small field. [9], [1].

TABLE 3. Area and Speed for Lookup Table

Gate	Count
AND	777
NOT	630
OR	450
XOR	1
Transistors	6180
Time	50ns
Order	$O(2^m m)$

This will be the most speed efficient implementation, but is not very area efficient - so choosing any other implementation would have to be a choice based on area considerations.

A lookup table composes well with anything else that can be specified as a lookup table, as the composition

of two lookup tables is a lookup table. In particular if a full Rijndael S-Box is being looked, at, the affine transformation may be included for no extra cost, whereas other methods they require an extra step.

For encoding of the problem in software, lookup tables (at least for the small n, such as 8) are the preferred, as RAM or ROM is a structure that will deal with this very efficiently. Also, for an FPGA design not intended to be transferred to silicon, most FPGAs will have modules built in that synthesise ROMs quite efficiently.

5. POWERS OF A PRIMITIVE ELEMENT

Given every Galois field $GF(2^m)$ has a primitive element (or generator) α, each element a can be represented as α^{i_a}. The inverse of a is then $a^{-1} = \alpha^{2^m-1-i_a}$.

However, since finding i_a is a slow problem (it requires a search), this only has an application (using 256 clock cycles) where gate-count is all important and speed doesn't matter.

The algorithm we use is:

```
Input value x
Set y to the identity 01
repeat 256 times
    if(y=x)
        set y to 01
    else
        set y to y*03
return y
```

y will be set to 1 on the clock cycle $i_x + 1$, hence the final value of y will will be 03^{255-i_x}.

Note that this does not map 00 to 00 - there needs to be additional logic for this if that is required.

Also, most of the gate count is actually in the control logic.

TABLE 4. Area and Speed for Powers of a Primitive Element

Gate	Count
AND	65
NOT	42
OR	2
XOR	25
Latches	18
Transitors	854
Time	5939ns
Order	$O(m)gates, O(2^m)time$

6. EXTENDED EUCLID

For any polynomials with coefficients in $GF(2)$, $b(x)$ and $m(x)$, Extended Euclid's Algorithm can be used to find $a(x)$,$b(x)$ such that $b(x)\cdot a(x)+m(x)\cdot c(x) = 1$ then the inverse $b^{-1}(x) = a(x) mod m(x)$.

```
ExtendedEuclid (a, b) returns
                  values for (d, x, y)
if b = 0
   return (a, 1, 0)
else
  (d', x', y') = ExtendedEuclid (b, a mod b)
  (d, x, y) = (d', y', x'-floor(a/b)*y')
  return (d, x, y)
```

If we evaluate ExtendedEuclid($b(x)$,$m(x)$), the tuple returned is (GCD, $a(x)$, $c(x)$).

6.1 Simple Implementation

There are some things we change to use this. Firstly, for polynomials with coefficients in $GF(2)$, the GCD will always be 1, so we need not keep track of it.

Also, recursion is a little difficult to use in a gate array unless we know the depth in advance, so our implementation expands out the recursion to a sufficient depth to find the GCD in all cases.

We use the following modified algorithm:

```
ModifiedEuclid(level, a, b)
                  returns values for (x, y)
if level = m
  return (a, b)
else
  (x', y') = ModifiedEuclid(level+1,b,a mod b)
  (x, y) = (d', y', x'-floor(a/b)*y')
  return (x, y)
```

Returning (a, b) when the depth is reached is a 'fudge' to make sure that the results get maintained even when m is more than the required number of iterations. This can be expanded into verilog by unrolling, with such optimizations as reducing the bit width by one each time.

If we evaluate ModifiedEuclid($0, b(x)$,$m(x)$), the first element of the pair returned will be the inverse, the other element can be discarded.

TABLE 5. Area and Speed for Extended Euclid

Gate	Count
AND	1511
INV	511
XOR	304
GND	54
VCC	37
Transistors	10714
Time	755ns
Order	$O(m^3)$

6.2 Optimized implementation

It is possible to find several optimizations of this technique. The paper [1] looks into this, and ends up with an efficient algorithm, although their approach is targeted to larger m.

We have not implemented this technique, but working from the information provided, the two input gate count (not including control logic) provided in Table 6.2

TABLE 6. Area and Speed for Optimized Extended Euclid

Gate	Count	Count (m=8)
AND	$38m + log_2 m$	611
OR	$16m$	128
NOT	$16m$	128
XOR	$6m + log_2 m$	51
FlipFlops	$6m + log_2 m$	51
Transistors	-	4844
Time	? $\times 2m$ clocks	? $\times$ 16 clocks
Order	$O(m)$ area $O(m)$ time	

This could possibly be unrolled, but would massively increase the gate count. This technique has got the best order performance, so is most suitable for large m.

We have not implemented this method, so do not have a direct comparison with other methods to put in our results.

7. EXPONENTIATION (LOOPED)

In the field $GF(2^m)$, $x^{2^m} = x$, so $x \cdot x \cdot x^{2^m-2} = x$, so $x \cdot x^{2^m-2} = 1$, hence $x^{-1} = x^{2^m-2}$. Therefore we can calculate the inverse of x in $GF(2^8)$ simply by calculating x^{254}.

TABLE 7. Area and Speed for Exponentiation (looped)

Gate	Count
AND	95
INV	26
OR	8
XOR	82
Latches	20
Transistors	1828
Time	708ns
Order	$O(m^3)$

The standard way [8] of doing this is to calculate x^2, x^4, x^8, x^{16}, x^{32}, x^{64}, x^{128} and multiply to get $x^2 \cdot x^4 \cdot x^8 \cdot x^{16} \cdot x^{32} \cdot x^{64} \cdot x^{128} = x^{254}$. This requires 7 square operations and 6 multiplications, with the critical path being one square operation and 6 multiplications. It could also be done very space-efficiently in a loop using 1 square operation and 1 multiplication, and 2 8-bit latches.

8. EXPONENTIONATION (UNROLLED)

It is possible to use less operations than described in section 7

$$x^{254} = (x^7 \cdot x^{120})^2 = (x^7 \cdot (((x^7 \cdot x^8)^2)^2)^2)^2$$

where $x^8 = (x^4)^2$, $x^7 = x^4 \cdot x^3$, $x^4 = (x^2)^2$ and $x^3 = x^2 \cdot x$.

This requires 7 square operations and 4 multiplications, and has a critical path of 4 multiplications and 5 square operations, but this can only be used in unrolled form, as it is too irregular to use in a loop.

TABLE 8. Area and Speed for Exponentiation (unrolled)

Gate	Count
AND	256
XOR	354
Transistors	5272
Time	443ns
Order	$O(m^3)$

There is another version [3] which is

$$x^{254} = (((((((x^3)^2)^2 \cdot x^3)^2)^2)^2 \cdot x^3)^2 \cdot x)^2$$

where $x^3 = x^2 \cdot x$ which also has 7 square operations and 4 multiplications, but has a critical path of all 11 operations.

This is smaller than lookup table, but the algorithm is considerably deeper, making the throughput considerably lower. Also *xor* gates may be more expensive?

It is worth noting that when multiplication is used instead of squaring, we get the results in Table **??**.

TABLE 9. Area and Speed for Exponentiation (unrolled) with multiplication is used instead of squaring

Gate	Count
AND	704
XOR	781
Transistors	12188

These numbers indicate that there is no optimizations being performed - they are simply the cost for 11 multiplications. This indicates that there is room fore reducing the gate counts still further with appropriate tools.

9. FIELD FACTORING

For any prime power there is a single finite field, hence for a given n, all representations of $GF(2^n)$ are isomorphic. Many operations (such as inverses) in the representation of $GF(2^8)$ are computationally quite expensive. To avoid this, we can change to a composite field representation of $GF((2^4)^2)$. This is described in [7].

Every element in $x \in GF((2^4)^2)$ can be represented as the first degree polynomial $x = a_0 + \beta a_1$, where $\beta^2+\beta+\lambda = 0$, $\beta^2+\beta+\lambda = 0$ for some $\lambda \in GF(2^4)$ and $a_0, a_1 \in GF(2^4)$.

The inverse of an element $x = a_0+\beta a_1$ in $GF((2^4)^2)$ is $x^{-1} = b_0 + \beta b_1$ where $b_0 = (a_0 + a_1)\Delta^{-1}$, $b_1 = a_1\Delta^{-1}$ and $\Delta = a_0^2+a_0a_1+\lambda a_1^2 = a_0(a_0+a_1)+\lambda a_1^2$. Which method of calculating Δ to use depends on the relative costs of the operations in $GF(2^4)$. Note that we still need to calculate an inverse in $GF(2^4)$, but a lookup table for this is only a 16x4 ROM, which is comparatively cheap in gates (much cheaper than the 256x8 ROM required for the full $GF(2^8)$ inverse).

We will have to transform our polynomial representation into the $GF((2^4)^2)$ form and back - these transformations might compose well with other transformations.

We have to make a choice of:

- The representation of $GF(2^4)$
- The mapping between $GF(2^8)$ and $GF((2^4)^2)$

We choose these to minimize the costs of the operations we want to perform, this includes the costs of:

- addition of elements
- multiplication of elements
- squaring of elements (in some cases)
- if we look towards full Rijndael encryption, multiplying by several constants, as happens in the Mix Columns stage.
- if we look towards full Rijndael encryption, performing the affine transformation in the S-Box.
- transforming between representations.

We limit our choices so that the following are satisfied:

- We still want the addition operation in $GF(2^8)$ to be implemented by *xor*.
- We want the mapping between $x \in GF(2^8)$ and $y \in GF((2^4)^2)$ to be the result of applying a matrix transformation, treating x and y as 8-vectors in $GF(2)$.

As we have the freedom to choose the representation of $GF(2^4)$, we investigate two such methods:

9.1 Polynomial Representation

For this approach [7], we represent elements in $GF(2^4)$ as polynomials, just as we did in $GF(2^8)$. We choose $y^4 + y + 1$ as the field polynomial.

Multiplication in this form of $GF(2^4)$ requires 16 two input *and* gates and 15 two input *xor* gates.

The inverse table for $GF(2^4)$ (expressed in hexadecimal) is given in Table 9.1:

TABLE 10. Inverses in $GF(2^4)$

x	1	2	3	4	5	6	7	8
x^{-1}	1	9	E	D	B	7	6	F

x	9	A	B	C	D	E	F
x^{-1}	2	C	5	A	4	3	8

There are four choices for the field polynomial in $GF((2^4)^2)$:

- $x^2 + x + 1001$
- $x^2 + x + 1011$
- $x^2 + x + 1101$
- $x^2 + x + 1110$

We choose $x^2 + x + 1001$ as multiplication by the constant 1001 is easy to compute in $GF(2^4)$.

We then map a primitive element of $GF(2^4)$ to a primitive element of $GF((2^4)^2)$ to create a mapping H, where H is chosen such that $H(x+y) = H(x)+H(y)$ where $+$ is bitwise exclusive-or, so we can still cheaply perform addition in $GF((2^4)^2)$. Such a mapping H will have a corresponding transformation matrix T.

As we can without loss of generality choose the primitive elements in one of the fields (we choose $0001\beta+0000$ represented as 00010000 in $GF((2^4)^2)$), we have less than 256 possibilities to check for primitive elements in $GF(2^8)$ to map to it, so we do this by brute force, and choose the one that is going to have the cheapest transformation matrix (represented by the least number of '1's).

After writing a program to check the possibilities, we choose to map 11111111 in $GF(2^8)$ to map to $0001\beta + 0000$ in $GF((2^4)^2)$). This is different than the solution in [7], as we are not considering the cost of the Rijndael polynomial transformation.

We can calculate $T_{(8-i)(8-j)} = H(2^j)_i$, so we get

giving the matrcies

$$T = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \text{ and}$$

$$T^{-1} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

For the Rijndael S-box, we also have to map zero as the inverse of itself. This can be done by using the above method and defining 0000 to be the inverse of 0000 in $GF(2^4)$.

TABLE 11. Area and Speed for Field Factoring

Gate	Count
AND	80
INV	26
XOR	87
Transistors	1416
Time	164ns
Order	n/a

9.2 Optimal Normal Basis Representation

For this approach [6], we choose β as a primitive element in $GF(2^4)$, and choose the set $\{\beta^{2^3}, \beta^{2^2}, \beta^2, \beta\}$ as a basis. It is optimal in the sense that β is chosen to minimize the number of terms required to perform a multiplication. There is a good discussion of this in [4].

Multiplication in an optimal normal basis requires 16 two input *and* gates and 16 two input *xor* gates, slightly more expensive than the other form. However calculating x^2 becomes a rotate left, costing zero gates.

It is still a matter for investigation whether this gives a smaller solution than the polynomial representation.

10. CONCLUSIONS AND FURTHER WORK

We have various options to implement inverses of $GF(2^8)$ with a wide range of performance/area tradeoffs.

There are two options that stand out - lookup tables for where speed is required and area is not such a strong consideration and field factoring for where area is more of an issue but speed is more important. Powers of a primitive element might be useful where area is critical, but time not important.

It is also likely that a more aggressive gate optimizer might give better, in some cases significantly better gate counts.

11. REFERENCES

[1] H. Brunner, A. Curiger, and M. Hofstetter. On computing multiplicative inverses in $gf(2^m)$. *IEEE Transactions on Computers*, 42(8):1010–1015, August 1993.

[2] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. http://csrc.nist.gov/encryption/aes/rijndael/Rijndaelammended.pdf.

[3] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $gf(2^m)$. *Information and Computation*, 78:171–177, 1988.

[4] Ivan Leung. *A Microcoded Elliptic Curve Cryptographic Processor*. Phd thesis, Department of Computer Science and Engineering, Chinese University of Hong Kong, 2001.

[5] W. W. Peterson and E.J Weldon Jr. *Error-Correcting Codes*. MIT Press, second edition, 1972.

[6] Vincent Rijmen. Efficient implementation of the rijndael s-box, 2000. http://www.esat.kuleuven.ac.be/~rijmen/rijndael/sbox.pdf.

[7] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi. Efficient implementation of rijndael encryption with composite field arithmetic. *Proceedings of Workshop of Cryptographic Hardware and Embedded Systems*, May 2001.

[8] C. Wang, C. Truong, T. K. Shao, H. M. Deutch, L. J. Omura, and I. R. Reed. Vlsi architectures for computing multiplications and inversed in $gf(2^m)$. *IEEE Transactions on Computers*, 34(8):709–716, 1985.

[9] Xinmiao Zhang and Keshab K. Parhi. Implementation approaches for the advanced encryption standard algorithm. *IEEE Circuits and Systems Magazine*, 2(4):24–46, 2002.