

On the Implementation of Finite Field Operations

Somsak Choomchuay

Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang

Abstract

This paper describes operations performed in finite field, in particular Galois field $GF[2^m]$. The main concerns are the hardware implementations of such operations which are ranging from simple addition to inversion. Several techniques are described in the aspects of different field element representations i.e., different basis. Likewise, both bit-parallel and bit-serial approaches are explained in the hardware realisation aspects. Pros and cons as well as the computational workload of each technique are discussed and compared.

1. Finite Field Mathematics

Finite field arithmetic is enormously used in several arena of digital signal processing. It is, in fact, a special case of abstract algebra. Data encryption and error control coding are some good examples of their applications. We assume that readers are familiar with mathematical backgrounds that concern signal processing applications. However, to ease the understanding of subsequent sections, element representations i.e., standard basis, normal basis and dual basis are brought into focus in this section.

Let α be a generator of the field $GF[2^m]$ which has the order of N where $N = 2^m$, i.e., all the field elements (except 0) can be represented by the power of α . Any element can be expressed either as $\beta = \alpha^i$, for $0 \leq i \leq N-1$, or a linear combination of a basis set $\{\gamma_i\}$.

Let Z be an element in $GF[2^m]$. In the *standard basis* representation it can be expressed as :

$$Z = z_0 + z_1\alpha + z_2\alpha^2 + \dots + z_{m-1}\alpha^{m-1}, \quad (1)$$

where $z_i \in \{0,1\}$.

This representation is a conventional way of representing a field element. It is easy to understand since coefficients z_i given in eqn (1) are coincident with the value given by an m -bit binary symbol.

Alternatively, an element Z can be uniquely expressed using the *normal basis* as :

$$Z = z_0\alpha + z_1\alpha^2 + z_2\alpha^4 + \dots + z_{m-1}\alpha^{2^{(m-1)}}, \quad (2)$$

where $z_i \in \{0,1\}$.

The main advantage of using the normal basis can be illustrated by the squaring property exploited in the Massey-Omura multiplier [1, 2].

Suppose that $\{\alpha, \alpha^2, \dots, \alpha^{2^{(m-1)}}\}$ is a normal basis of $GF[2^m]$. With the finite field property that $\alpha^{2^m} = \alpha$, the square of Z can be written as :

$$\begin{aligned} Z^2 &= z_0\alpha^2 + z_1\alpha^4 + z_2\alpha^8 + \dots + z_{m-1}\alpha^{2^m}, \\ &= z_{m-1}\alpha + z_0\alpha^2 + z_1\alpha^4 + \dots + z_{m-2}\alpha^{2^{m-1}}. \end{aligned} \quad (3)$$

By comparing eqn (2) and eqn (3), it can be seen that with normal basis representation, Z^2 is a cyclic shift of Z . This notice leads to simpler hardware implementation of squaring, inversion and multiplication detailed in the following sections.

A field element Z can also be expressed using the *dual basis* representation as :

$$Z = z'_0\lambda_0 + z'_1\lambda_1 + z'_2\lambda_2 + \dots + z'_{m-1}\lambda_{m-1}, \quad (4)$$

where $z'_k \in \{0,1\}$ is the trace of Z , i.e. $z'_k = Tr(Z\alpha^k)$. The definition of trace is given below.

Let β an element in $GF[2^m]$. The trace of β is defined as :

$$Tr(\beta) = \sum_{k=0}^{m-1} \beta^{2^k} \quad (5)$$

With the dual basis field element representation, a relatively simple and efficient multiplier can be implemented [3]. Such a multiplier involves only addition and shift operations.

2. Operations and Implementations

The following section deals with finite filed operations, implementations and their workload contributions. Notation T, S and τ with subscript operation refer to time (in clock cycles), area (in term of gate count) and delay time (in nanosecond) respectively. Details of number of gates and their associated parameters are given in Appendix at the end of this paper. Operands and result holding registers are not counted in the required hardware complexity.

2.1 Addition

Addition in the finite field of $GF[2^m]$ is relatively simple since there is neither carry nor borrow. This operation can be realised using bit-wise XOR, i.e. $C = A \oplus B$. However for simplicity the plus sign (+) instead of \oplus is used throughout this paper. Shown in Figure 1, addition can be implemented using both bit-serial and bit-parallel methods. In a bit-serial technique, data are shifted in bit by bit serially in to an adder. Obviously, only a 2-input XOR gate is required. The operation is completed in m clock cycles. In a bit-parallel technique, m -bit data are parallelly fed into adders and the result is obtained simultaneously.

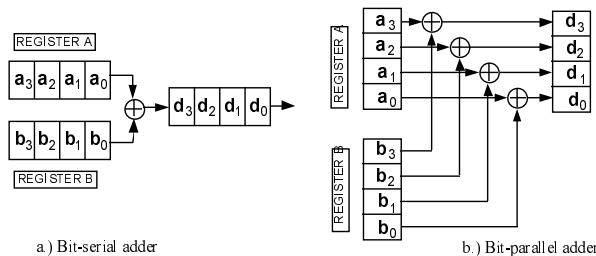


Figure 1 A bit-serial and a bit-parallel finite field adders

Therefore, a bit-parallel adder required only one clock cycle but at the complexity of m times greater than that required by a bit-serial circuit.

The property of both types of finite field adders are summarised as follows :

$$T_{ser_add} = m \quad (6a)$$

$$S_{ser_add} = S_{XOR} \quad (6b)$$

$$\tau_{ser_add} = m\tau_{XOR} \quad (6c)$$

and

$$T_{para_add} = 1 \quad (7a)$$

$$S_{para_add} = mS_{XOR} \quad (7b)$$

$$\tau_{para_add} = \tau_{XOR} \quad (7c)$$

2.2 Multiplications

Although multiplications in the finite field is rather simple compared to those in the ordinary field, they are still complicate and take considerable chip area. Several techniques used to implement finite field multiplications are detailed in the following subsections. In many cases, it should be noted that a field element is defined using its polynomial notation.

2.2.1 Standard Basis Multipliers

A standard basis multiplier is generally more preferable for most applications. In many cases no basis conversion is necessary when the system has to interface to other sub-systems since the standard basis field element representation is conventional used.

A.) A General Purpose Bit-Serial Multiplier

In the standard basis, let the polynomial $A(x)$ and $B(x)$ be defined as :

$$A(x) = a_{m-1}x^{m-1} + \dots + a_2x^2 + a_1x + a_0, \quad (8a)$$

and

$$B(x) = b_{m-1}x^{m-1} + \dots + b_2x^2 + b_1x + b_0, \quad (8b)$$

where a_i and b_i are defined similar to z_i given in eqn (1). Then, the polynomial

$$D(x) = d_{m-1}x^{m-1} + \dots + d_2x^2 + d_1x + d_0 \quad (8c)$$

denotes the product $A(x)B(x) \bmod p(x)$ where $p(x)$ is the primitive polynomial in that field. The operation is explained in more details as below.

$$\begin{aligned} D(x) &= A(x) \cdot B(x) \bmod p(x) \\ &= A(x)\{b_{m-1}x^{m-1} + \dots + b_2x^2 + b_1x + b_0\} \bmod p(x) \\ &= \{A(x)b_{m-1}x^{m-1} \bmod p(x) + \dots + A(x)b_2x^2 \bmod p(x) \\ &\quad + A(x)b_1x \bmod p(x) + A(x)b_0 \bmod p(x)\} \bmod p(x) \end{aligned} \quad (9)$$

The multiplication is performed recursively for m steps since the element is an m -bit signal. Each step gives an intermediate product. Results in m computation blocks are cascaded to each other as shown in Figure 2.

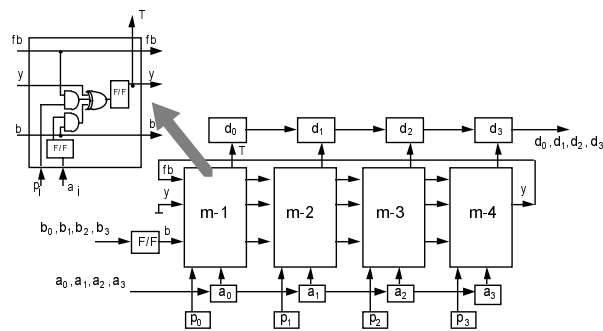


Figure 2 A general purpose $GF[2^4]$ standard basis bit-serial multiplier

For the first m clock cycles, a_i are shifted in serially. Then they loaded into latches located in the computing cells and remain unchanged for the following m clock cycles. In second m -clock cycles coefficients b_i are shifted in, with the most significant bit first. A partial multiplication is performed after each shift and the result is stored in an internal flip flop. After m clock cycles the multiplication result is obtained. The output register, D is parallelly load with the content of internal flip flop and started to shift out in the serial manner subsequently.

It should be noted that to obtain the smooth input and output data flow i.e. 1 bit/clock cycle, $A(x)$ should lead $B(x)$ by m clock cycles. As shown in Figure 2, p_i denote the coefficients of the primitive polynomial. For example, in $GF[2^4]$ where $p(x) = x^4 + x + 1$, then $p_0 = p_1 = 1$, and $p_2 = p_3 = 0$. For the fix polynomial $p(x)$ an m -bit storage register can be eliminated and inputs of AND gates are solidly tie to “1” or “0” logic level. The last state partial product is broadcast to every states include itself providing the modulo operation effect.

A basic computation cell comprises of 2 flip flops, 2 AND gates and a 3-input XOR gate. There are m cells which all are identical. Each is allocated for each coefficient. The delay time in each computing cell can be approximated to be the time that signal propagates through an AND gate, an XOR gate and a flip flop. Hence,

$$T_{ser_std_mul} = m \quad (10a)$$

$$S_{ser_std_mul} = m(2S_{DFF} + 2S_{AND} + S_{XOR}) \quad (10b)$$

$$\tau_{ser_std_mul} = m(\tau_{DFF} + \tau_{AND} + \tau_{XOR}) \quad (10c)$$

B.) A Fixed-Coefficient Bit-Serial Multiplier

A fixed-coefficient bit-serial multiplier is much simpler than a general purpose one since unnecessary terms can be removed. Consider an example given in a field $GF[2^4]$ where $p(x) = x^4 + x + 1$. Let $A(x)$ is a constant, $A(x) = x^3 + x^2$ and let $B(x)$ be any field element that expressed as $B(x) = \sum_{k=0}^{k=3} b_k x^k$. Then $D(x) = A(x)B(x) \bmod p(x)$ can be implemented using the circuit shown in Figure 3 below.

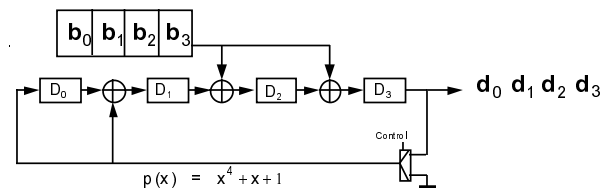


Figure 3 A fixed-coefficient bit-serial multiplier

An m bit register is used to store the immediate product. This register is first initialised with 0. The component b_i of $B(x)$ is shifted in serially with the most significant bit first. This component is added to the previous result at the bit positions defined by $A(x)$. The most significant product bit of the register is fed back and added to the lower significant bit according to the primitive polynomial. Hence, there are two operations involved. First, $A(x)$ is multiplied by a component of $B(x)$. Second, the intermediate product is taken modulo $p(x)$. After m clock cycles, the last bit of $B(x)$ has shifted in, the register contains the product $A(x)B(x) \bmod p(x)$. This can be shifted out serially as the feedback path is disable.

C.) A General Purpose Bit-Parallel Multiplier

Let the field element $A(x)$ and $B(x)$ be expressed as :

$$A(x) = \sum_{i=0}^{m-1} a_i x^i \text{ and } B(x) = \sum_{i=0}^{m-1} b_i x^i$$

Then, for $D(x) = A(x)B(x) \text{ mod } p(x)$, each component of $D(x)$ can be written as follows:

$$d_k = \sum_{i=0}^{m-1} a_i b_{k-i} \text{ for } 0 \leq k \leq m-1,$$

where the multiplication is taken modulo $p(x)$. For example, in $GF(2^4)$, let $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ and $B(x) = b_0 + b_1x + b_2x^2 + b_3x^3$. The product $D(x)$ can be expressed as :

$$\begin{aligned} D(x) &= A(x)B(x) \text{ mod } p(x) \\ &= (a_1b_3 + a_2b_2 + a_3b_1 + a_0b_0) \\ &+ (a_1b_3 + a_2b_2 + a_3b_1 + a_0b_1 + a_1b_0 + a_2b_3 + a_3b_2)x \\ &+ (a_2b_3 + a_3b_2 + a_0b_2 + a_1b_1 + a_2b_0 + a_3b_3)x^2 \\ &+ (a_3b_3 + a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)x^3 \end{aligned} \quad (11)$$

Equation (11) can be implemented using regular array architecture shown in Figure 4. Each computing cell comprises a 2-input AND and a 2-input XOR gates. A and B registers are parallel loaded with $A(x)$ and $B(x)$ respectively. Most basic cells are identical which can be slightly different for the top most and the bottom most cell arrays. The m output bits are delayed at most $(5\tau_{\text{cell}} + \tau_{\text{XOR}})$, where τ_{cell} denotes the delay of a single cell and τ_{XOR} denotes the delay of a 2-input XOR gate.

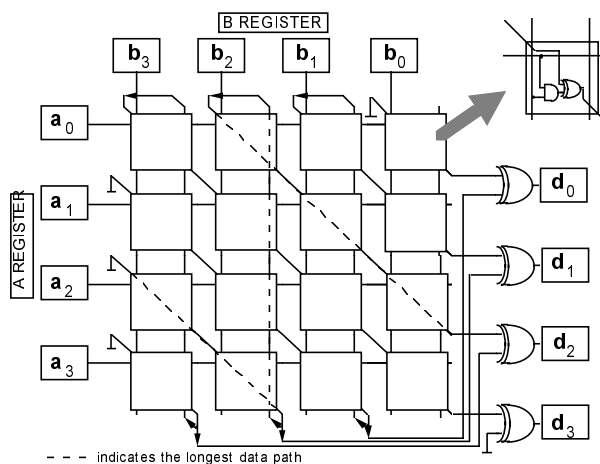


Figure 4 A bit-parallel standard basis multiplier

A general purpose bit-parallel multiplier can be made slightly faster at the cost of the basic cell irregularity. Consider a diagram in Figure 5,

all multiplications are computed in parallel and followed by additions which are again computed in parallel. Now the delay of this circuit can be computed as $\tau = \tau_{\text{AND}} + \tau_{\text{XOR}_7}$, where τ_{XOR_7} refers to the delay time of the 7-input XOR array. Such an array can be implemented using seven of 2-input XOR gates, cascaded in three stages. The array gives the delay of $3\tau_{\text{XOR}_2}$. This is twice faster than the circuit shown in Figure 4.

A general purpose bit-parallel multiplier (Figure 4) comprises of m^2 2-input AND gates and m^2+m 2-input XOR gates. The longest data delay is $6\tau_{\text{XOR}}$ (approx. for $GF[2^4]$). The multiplication result can be obtained in one clock cycle.

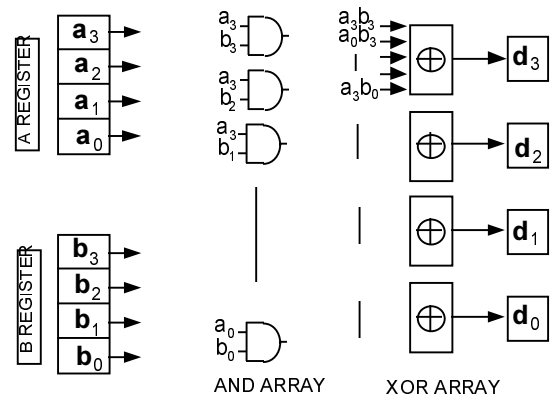


Figure 5 An alternative realisation of a bit-parallel multiplier

The multiplier shown in Figure 5 also comprises m^2 2-input AND gates and 22 XOR operations which are divided into 4 groups each for an output coefficient. Each group can be implemented with 2-input XOR gates. The biggest group involves 7 XOR operations which can be implemented using three stages 2-input XOR gates. For m which is not very large (*i.e.* $m = 8$) the number of XOR stages can be approximated to $3m/4$. The latter multiplier can be twice faster than the first one while both have about the same number of gates. In both multipliers, there appears to be 3 more m -bit registers to store the input words and output word. However, the first multiplier has the advantage of its design regularity. We, hence, exploit its complexity as detailed below.

$$T_{\text{para_std_mul}} = 1 \quad (12a)$$

$$S_{\text{para_std_mul}} = m^2(S_{\text{AND}} + S_{\text{XOR}}) + mS_{\text{XOR}} \quad (12b)$$

$$\tau_{\text{para_std_mul}} \approx m(\tau_{\text{XOR}}) \quad (12c)$$

D.) A Fixed-Coefficient Bit-Parallel Multiplier

Similar to the bit-serial multiplier, a fixed coefficient bit-parallel multiplier is also simpler than a general purpose one. Modulo operation can be pre-computed and hard-wired. Consider an example given in $GF[2^4]$, let $B(x)$ be an arbitrary field element and $A(x)$ is a constant, $A(x) = x^3 + x^2$. Then $A(x)B(x) \bmod p(x)$ can be derived as :

$$\begin{aligned}
 D(x) &= A(x)B(x) \bmod p(x) \\
 &= b_0A(x) + b_1A(x) + b_2A(x) + b_3A(x) \\
 &= b_0(x^3 + x^2) + b_1x(x^3 + x^2) \\
 &\quad + b_2x^2(x^3 + x^2) + b_3x^3(x^3 + x^2) \quad (13) \\
 &= (b_1 + b_2) + (b_1 + b_3)x + (b_0 + b_2)x^2 \\
 &\quad + (b_0 + b_1 + b_3)x^3,
 \end{aligned}$$

which can be re-written in the matrix form as $[D] = [T][B]$, where $[T]$ is the transformation matrix given by the following equation.

$$[d_0, d_1, d_2, d_3] = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} [b_0, b_1, b_2, b_3]. \quad (14)$$

The computation of above equations can be arranged as shown in Figure 6. XOR gates are combined to form the transformation matrix $[T]$ given in eqn (14). The number of 2-input XOR gates can be computed as half of the number of non-zero entries of above matrix $[T]$.

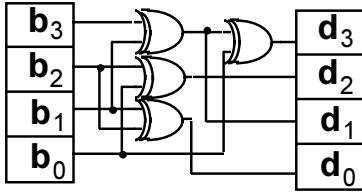


Figure 6 A fixed-coefficient bit-parallel multiplier ($D(x) = \alpha^6 B(x)$, in $GF[2^4]$)

2.2.2 Normal Basis Multiplications

Normal basis multiplication is based on Massey-Omura's technique. The implementation is relies upon the normal basis representation of the field elements. This section elaborates more details of the circuit complexity and delay time introduced by different approaches. In $GF[2^m]$, let $A = [a_0, a_1, \dots, a_{m-1}]$ and $B = [b_0, b_1, \dots, b_{m-1}]$ be two field elements. Also let $D = [d_0, d_1, \dots, d_{m-1}]$

be the product of those two elements. Then the last term of such a product can be represented as some binary function, f . That is,

$$d_{m-1} = f(a_0, a_1, \dots, a_{m-1}, b_0, b_1, \dots, b_{m-1}).$$

With the normal basis representation, the square of the product is the cyclic shift form as remarked in the previous section, i.e.,

$$\begin{aligned}
 D^2 &= A^2 B^2 \\
 &= [a_{m-1}, a_0, \dots, a_{m-2}] \cdot [b_{m-1}, b_0, \dots, b_{m-2}] \quad (15) \\
 &= [d_{m-1}, d_0, \dots, d_{m-2}].
 \end{aligned}$$

Similarly, the component d_{m-2} can be obtained by the same function f that applied to the components of A^2 and B^2 . Thus,

$$d_{m-2} = f(a_{m-1}, a_0, \dots, a_{m-2}, b_{m-1}, b_0, \dots, b_{m-2}).$$

Keeping squaring A and B repeatedly, one obtains d_{m-3} through d_0 .

Consider an example given in $GF[2^4]$ where $p(x) = x^4 + x + 1$. The product D can be written as :

$$\begin{aligned}
 D &= A \cdot B \\
 &= (a_0\alpha + a_1\alpha^2 + a_2\alpha^4 + a_3\alpha^8) \cdot (b_0\alpha + b_1\alpha^2 + b_2\alpha^4 + b_3\alpha^8). \quad (16)
 \end{aligned}$$

The term d_3 of $D = (d_0\alpha + d_1\alpha^2 + d_2\alpha^4 + d_3\alpha^8)$ is therefore,

$$\begin{aligned}
 d_3 &= a_0b_1 + a_1b_0 + a_0b_3 + a_3b_0 + a_3b_1 \\
 &\quad + a_3b_2 + a_2b_3, \quad (17)
 \end{aligned}$$

which is the function f . It is important to note that the number of product terms that formulates d_{m-1} depends crucially on the selection of primitive polynomial. For the Galois field $GF[2^m]$ where $m \geq 4$, there appear to be more than one generator polynomial that can have the normal basis representations [5]. With a clever choice of $p(x)$, the hardware complexity can be reduced since the number of term required in d_{m-1} can be made small [2]. For example, in $GF[2^8]$, the primitive polynomial $p(x) = x^8 + x^5 + x^3 + x + 1$ gives the minimum number of terms.

A normal basis multiplier can be implemented using both bit-serial and bit-parallel techniques as depicted in Figure 7 and 8.

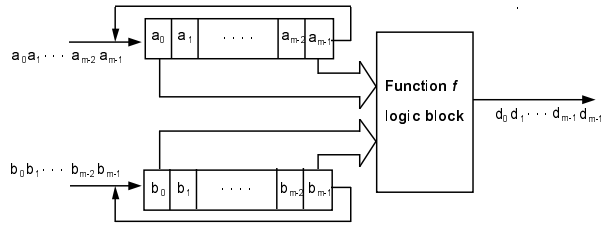


Figure 7 A bit-serial Messay-Omura multiplier

The same function f logic block can be used to implement all output product coefficients in the above implementation. When the first component is obtained, input registers are cyclically shifted by one position in order to perform squaring operation. Hence, it requires m clock cycles to perform each element multiplication. The circuit complexity is considered to be minimum since only one logic block is recursively used for m times. The number of terms in the function f logic block are found to be between $2m-3m$. In the worst case, $3m$ AND and $3m$ XOR gated are required. These XOR gates can be cascaded into $3m/4$ levels. Hence,

$$T_{ser_nor_mul} = m \tag{18a}$$

$$S_{ser_nor_mul} = (2m+1)S_{DFE} + 3m(S_{AND} + S_{XOR}) \tag{18b}$$

$$\tau_{ser_nor_mul} = \tau_{DFE} + \tau_{AND} + 0.75m\tau_{XOR} \tag{18c}$$

A bit-parallel scheme can be implemented in a slight different way. There are number of function f logic blocks. Each comprises of AND and XOR arrays. The internal connection is vary from block to block. However, only slight changes. Gate array is the most suitable technology for implementing this multiplier.

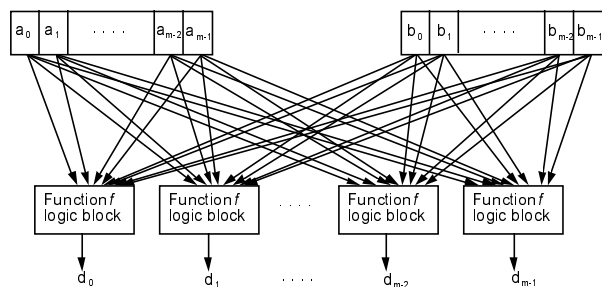


Figure 8 A bit-parallel Messay-Omura multiplier

Compared to a bit-serial implementation, the bit-parallel multiplier requires m times as much as the area required by a bit-serial type. However, the operation is as fast as a single clock cycle and the operation delay time is the delay time of a function f logic block.

2.2.3 Dual basis multipliers

A dual basis multiplication technique described in the following section operates on general polynomial coefficients rather than a fixed-coefficient as detailed in [3].

Let Z be element expressed in the dual basis representation as shown in eqn (4) and let G be an element represented in standard basis as:

$$G = g_0\alpha^0 + g_1\alpha^1 + g_2\alpha^2 + \dots + g_{m-1}\alpha^{m-1}, \tag{19}$$

for $0 \leq i \leq m-1$, where $g_i \in \{0,1\}$. The product W is expressed in the dual basis as :

$$W = \sum_{k=0}^{k=m-1} w'_k \lambda_k, \tag{20}$$

where $w'_k = Tr(ZG\alpha^k)$ and $G = \sum_{j=0}^{j=m-1} g_j\alpha^j$.

Let $T^{(k)}(Z) = Tr(ZG\alpha^k)$, then the Berlekamp's technique is used to performed $T^{(k)}(Z)$ recursively on k for $0 \leq k \leq m-1$. The procedure, hence, computes the coefficient of λ_0 first, and then that of λ_1 and so on.

Consider the following example given in $GF[2^4]$ where $\{\alpha^k\}$ and $\{\lambda_k\}$ are complement basis and $p(x) = x^4 + x + 1$ is the primitive polynomial. Initially, for $k = 0$, one obtains

$$\begin{aligned} T^{(0)}(Z) &= Tr((\alpha Z)G\alpha^0) \\ &= g_0z'_0 + g_1z'_1 + g_2z'_2 + g_3z'_3. \end{aligned} \tag{21}$$

To compute $T^{(k)}(Z)$ for $k > 0$, one may notice that $T^{(k)}(Z) = Tr((\alpha Z)G\alpha^{k-1}) = T^{(k-1)}(Z)$. Therefore $T^{(k)}(Z)$ is obtained from $T^{(k-1)}(Z)$ by replacing Z by αZ . This is accomplished by shifting up Z register and fill z'_3 with the feed back term $z'_0 + z'_1$. Thus, w'_1 can be computed as:

$$\begin{aligned} T^{(1)}(Z) &= Tr((\alpha Z)G\alpha^0) \\ &= g_0z'_1 + g_1z'_2 + g_2z'_3 + g_3(z'_0 + z'_1). \end{aligned} \tag{22}$$

With similar procedures, w'_2 and w'_3 can be obtained. The circuit used to implemented the above discussed technique can be drawn as shown in Figure 9.

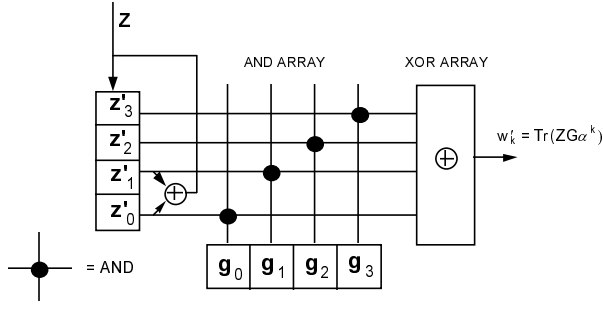


Figure 9 A dual basis multiplier

Many cases of the finite field operation, for example, the encoding of many classes of codes [6, 7], require polynomial multiplication. A code word is defined such that it is divisible by a generator polynomial $G(x)$, i.e., $C(x) = M(x)G(x)$, where $C(x)$ is a code word polynomial and $M(x)$ is a message word which also defined in a polynomial form. The encoder that designed for any generator polynomial can be rather complicated since the polynomial $G(x)$ cannot be made fixed. A simpler multiplier circuit is then necessary. The following paragraph illustrates the application of trace and dual basis techniques in implementing polynomial multiplication.

Given $G(x)$ as a degree t polynomial,

$$G(x) = \sum_{i=0}^{t-1} G_i x^i, \text{ where } G_i = \sum_{j=0}^{m-1} g_{ij} \alpha^j. \quad (23)$$

Thus,

$$ZG(x) = \sum_{i=0}^{t-1} ZG_i. \quad (24)$$

Now let $T_i^{(k)} = \text{Tr}(ZG_i \alpha^k)$. For $k = 0$, the value of trace can be written in the matrix form as :

$$\begin{bmatrix} T_0^{(0)} \\ T_1^{(0)} \\ - \\ T_t^{(0)} \end{bmatrix} = \begin{bmatrix} \text{Tr}(ZG_0) \\ \text{Tr}(ZG_1) \\ - \\ \text{Tr}(ZG_t) \end{bmatrix} = \begin{bmatrix} g_{00} & g_{01} & - & g_{0(m-1)} \\ g_{10} & g_{11} & - & g_{1(m-1)} \\ - & - & - & - \\ g_{t0} & g_{t1} & - & g_{t(m-1)} \end{bmatrix} \begin{bmatrix} z'_0 \\ z'_1 \\ - \\ z'_{(m-1)} \end{bmatrix} \quad (25)$$

The following steps are to compute $T_i^{(k)}$ for $0 \leq i \leq t$ and $0 \leq k \leq m-1$ by using feed back and shift as

described earlier. The product term ZG_i is obtained from

$$ZG_i = \sum_{k=0}^{m-1} \text{Tr}(ZG_i \alpha^k) \lambda_k = \sum_{k=0}^{m-1} T_i^{(k)}(z) \lambda_k. \quad (26)$$

Hence,

$$ZG(x) = \sum_{i=0}^{t-1} \sum_{k=0}^{m-1} T_i^{(k)}(Z) \lambda_k. \quad (27)$$

In fact, the circuit used to implement eqn (27) as shown in Figure 10 is t parallel sets of the circuit shown previously in Figure 9. Each set computes one coefficient of the polynomial $ZG(x)$. For any fixed G_i , the circuit can be further simplified since the AND array can be omitted. A multiplier then requires only an XOR array because element g_{ij} is 0 or 1.

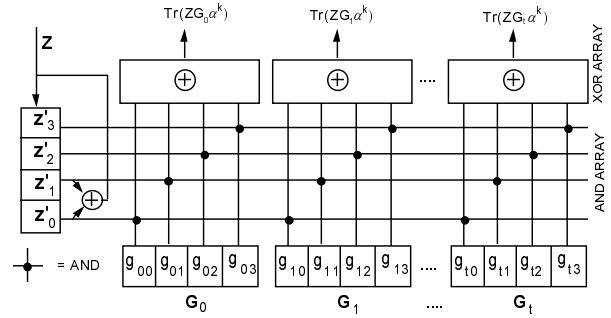


Figure 10 A multiplier that uses dual basis field element representation.

It should be noted that a dual basis multiplier operates on two components defined in different basis, i.e., Z is defined in the dual basis whilst G is defined in the standard basis. The obtained result is defined in the dual basis. Regardless of the basis conversion, the dual basis multiplier comprises of an AND array, an XOR array, an m -bit register and a delay flip flop. AND and XOR arrays comprises of m 2-input AND gates and m 2-input XOR gates respectively. XOR gates can be arranged into $m/2$ cascaded stages. Signal delay time of the multiplier can be those of 2 DFFs, AND gate and $m/2$ stages XOR gates. Hence,

$$T_{dual_mu} = m \quad (28a)$$

$$S_{dual_mul} = (m+1)S_{DFF} + m(S_{AND} + S_{XOR}) \quad (28b)$$

$$\tau_{dual_mul} = m(2\tau_{DFF} + \tau_{AND} + 0.5m\tau_{XOR}) \quad (28c)$$

2.3 Squares & Square roots

In the finite field, squaring is the special case of multiplication, that is, the field element is multiplied by itself. Square operation is also linear since $(A+B)^2 = A^2+B^2$. A squaring circuit can be implemented similar to a fixed-coefficient bit-parallel multiplier. As an example, let $B(x)$ be the polynomial representation of any field element defined in $GF[2^4]$. The transformation matrix can be derived as follows :

$$D(x) = \{B(x)\}^2 \text{ mod } p(x), \quad (29a)$$

$$[D] = [T][B], \quad (29b)$$

$$\text{where } [T] = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (29c)$$

The inversion of transform matrix given in eqn (29c) defines the square root transform matrix of $B(x)$, i.e.,

$$[T]^{-1} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (29d)$$

The logic circuit used to implement square and square root corresponding to eqns (29c) and (29d) are shown in Figure 11a and 11b respectively.

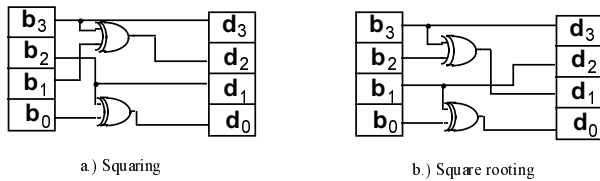


Figure 11 Square and square root circuits

Also be noted that the standard basis field element representation is assumed in the above discussion. In the normal basis representation, squaring is relatively simple as the cyclic shifting of the field element register.

2.4 Inversions

There are number of methods employed in computing the finite field element inversion. Each has different speed, complexity, and hence, suits each particular application. A conventional bit-parallel sequential technique is outlined first.

A new developed bit-serial technique is discussed subsequently.

The Galois field $GF[2^m]$ comprises of a cyclic multiplicative group of 2^m-1 elements, exclude zero. For any field element β , $\beta^{2^m-1} = 1$ or $\beta^{-1} = \beta^{2^m-2}$. An efficient way to compute β^{-1} is to form a sequence $\beta, \beta^2, \beta^3, \beta^4, \dots, \beta^{2^m-2}$ using successive multiplying and squaring as :

$$\beta^{-1} = [\{(\beta^2)\beta\}^2 \beta \dots \beta]^2. \quad (30)$$

Alternatively, $2^m - 2$ can be factorised as ;

$$2^m - 2 = 2 + 2^2 + 2^3 + \dots + 2^{m-1}.$$

Then β^{-1} can be computed from :

$$\beta^{-1} = \beta^{2^1} \cdot \beta^{2^2} \cdot \beta^{2^3} \dots \beta^{2^{(m-1)}}. \quad (31)$$

Equations (30) and (31) given above can be implemented using the circuits shown in Figure 12a and 12b respectively.

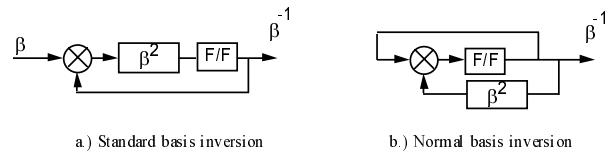


Figure 12 Sequential circuit for computing finite field inversion (data paths are m -bit wide)

The above circuits require $m-1$ clock cycles to perform any inversion which include multiplying and squaring. If the element is represented in the normal basis a squaring can be simple as described earlier. Even so, the major chip area consumption in the circuit is still a multiplier. This cannot be avoid even in the standard basis representation of which squaring is also relatively simple. Assume that the squaring circuit has approximately $m/2$ adders. The complexity of an inversion circuit can be then concluded as :

$$T_{seq_inv} = m-1 \quad (32a)$$

$$S_{seq_inv} = mS_{DFE} + S_{multiplier} + 0.5S_{XOR} \quad (32b)$$

$$\tau_{seq_inv} = (m-1)(2\tau_{DFE} + \tau_{multiplier}) \quad (32c)$$

An inversion can also be computed using Euclid's algorithm [8]. The hardware implementation requires 3 of $(m+2)$ -bit registers and some comparing logics. Such a circuit takes

about $4m$ clock cycles to complete an element inversion.

A bit-serial technique detailed in the following section can be very efficient in term of hardware since no multiplier or even a complicate comparing circuit is required. There are two m -bit registers, both are hardwired according to the primitive polynomial used to generate field elements (see Figure 13). The register A is loaded with 1 while the register B is loaded with β . Keeping clocking both registers until the content of register B equal 1. Then the content of register A is β^{-1} . The operation may take at most 2^m-1 clock cycles, since the operation is cyclic.

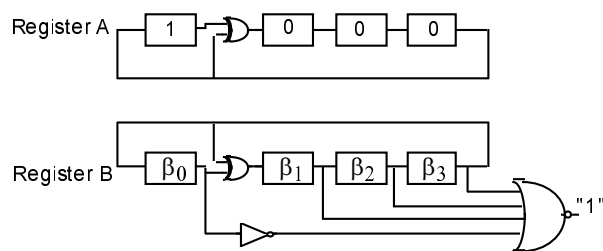


Figure 13 A bit-serial shift register inversion circuit

This technique is also easy to be adapted to compute multiplication-inversion and is also superior for the application that requires low space, low power and allows N clock cycles for operation (such as a time domain decoding system [9]). The complexity of the circuit is summarised as :

$$T_{ser_sft_inv} = 2^m - 1 \quad (33a)$$

$$S_{ser_sft_inv} = 2\{mS_{DFF} + (m-1)S_{XOR}\} \quad (33b)$$

$$\tau_{ser_sft_inv} = (2^m - 1)(\tau_{DFF}) \quad (33c)$$

3. Summary

Several techniques for implementing operations in the finite field are detailed in this paper. Among those, multiplications seem to be more complicated than others. They are, hence, realised in many ways. Two main themes that considerably effect the hardware implementation are recognised as field element representation, and bit-serial and bit-parallel approaches. Standard basis representation are used conventionally due to its ease in being well-understood. However, normal basis and dual basis representation are also have proved their simplicity in some hardware realisations which of

course their utilisation are very much depend on applications and available technology.

Inversion, can also be complicated, especially when it is not able to implemented using ROM based look-up table. A shift register technique proposed in the last section can be very efficient in term of its small size, but it is suitable for the system that allows N clocks for such an operation.

Compared to the bit-parallel technique, a bit-serial implementation always requires less hardware, but on the other sides it takes more clock cycles to complete its operation which also implies its higher latency. More hardware can be saved in many cases that operands are of fixed-coefficients. This can be obviously seen in the cases of fixed-coefficient multiplication, squaring and square rooting.

Reference

- [1] C. C. Wang, T. K. Truong, H. M. Shao, L. J. Deutch, L. J. Omura and I. S. Reed, "VLSI Architecture for Computing Multiplication and Inversion in $GF[2^m]$," IEEE Trans., vol. C-34, pp. 709-717, September 1985.
- [2] Y. R. Shayan and T. Le-Ngoc, "The least complex parallel Massey-Omura multiplier and its LCA and VLSI design," IEE Proc. vol. 136, Pt. G, No. 6, pp. 345-349, December 1989.
- [3] E. R. Berlekamp, "Bit-serial Reed-Solomon Encoder," IEEE Trans. Inform. Theory, vol. IT-28, pp. 869-874, November 1982.
- [4] I. S. Hsu, I. S. Reed, T. K. Truong, K. Wang, C. S. Yeh and L. J. Deutch, "The VLSI Implementation of a Reed-Solomon Encoder Using Berlekamp's Bit-serial multiplier Algorithm," IEEE Trans., vol. C-33, No. 10, pp. 906-911, October 1984.
- [5] W. W. Peterson and E. J. Weldon Jr., Error Correcting Codes, The M.I.T. Press, Cambridge, Mass, 1972.
- [6] B. Arambepola and S. Choomchuay, "Algorithm and Architecture for Reed-Solomon Codes," GEC Journal of Research, Vol. 9, No. 3, 1992, pp. 172-184.
- [7] R. E. Blahut, Theory and practice of Error Control Codes, Addison-Wesley, Reading, MA, 1983.

- [8] E. R. Berlekamp, Algebraic Coding Theory, Mc Graw-Hill, 1968.
- [9] S. Choomchuay and B. Arambepola, "Algorithms and Architectures for Time Domain Reed-Solomon Decoding," IEE Proc. - I, vol. 140, No. 3, pp. 189-196, June 1993.
- [10] LSI Logic Corp., "1.0 Micron Array-based Products Data Book," LSI Logic Corporation, CA, September 1991.

Appendix

1) Complexity measurement

The hardware complexity discussed through out this paper can be illustrated based on macrocells, 1.0 micron CMOS process [10]. The area complexity is given in term of gate count. The complexity of a single gate is equivalent to 2-4 MOS transistors. The delay time referred to here can be either low to high (t_{PLH}) or high to low (t_{PHL}) propagation delay time depends on which one is greater. Some of the gates and registers are given in the table below. More details or different devices parameter can be found in the manual [10].

Devices	Area	Delay propagation [†]	
		Fan out=1	Fan out=2
Inverter	1	0.36	0.44
2-input AND	2	0.58	0.62
2-input XOR	3	0.85	0.94
3-input XOR	6	0.99	1.03
2->1 MUX	4	0.83	0.87
D flip-flop	7	1.05	1.09
D latch	5	0.77	0.86

[†] Vdd = 5V±10%, Ambient temp. = 25 °C.

2). Finite Field table look-up ROM

In the standard basis element representation, each element belongs to the cyclic multiplicative group generated by the primitive root. Hence, there are 2^m-1 elements which can be referred to by its power, i.e., $\beta = \alpha^k$, the k^{th} location contains α^k . Represent the filed element by its power this way, some operations can be made simpler as follows.

let $A = \alpha^i$ be stored at the location i and let $B = \alpha^j$ be stored at the location j ;

a.) Multiplication :

$$D = A \cdot B = \alpha^i \cdot \alpha^j = \alpha^{(i+j)}.$$

The location $(i+j) \bmod (2^m-1)$ then contains the result.

b.) Inversion :

$$D = B^{-1} = \alpha^{-j} = \alpha^{(2^m-j-1)}.$$

The location $(2^m-j-1) \bmod (2^m-1)$ then contains the inversion result.

c.) Division :

$$D = AB^{-1} = \alpha^i \cdot \alpha^{-j} = \alpha^{(i-j)}.$$

The location $(i-j) \bmod (2^m-1)$ then contains the division result.

d.) Addition :

The addition is rather more intricate than the ordinary way of element representation. This can be seen from an equation given below.

$$D = A + B = \alpha^i + \alpha^{-j} = \alpha^i (1 + \alpha^{j-i}).$$

Similar procedures can be applied to powers and roots. The representation of a field element by its power is known as logarithmic representation. The operation is performed using its power (or index). Conversion back and forth of such a representation can be accomplished using a look up table which can be implemented using ROM The ROM size is m -bit \times 2^m locations. Basically, a ROM consists of a decoder and a data matrix. The decoder which consists of $2m$ AND gate of m inputs each, transforms m -bit inputs (addresses) into 2^m address locations. Data bits are buffered before they are actually read out.

With the use of memory (or table), multiplications and inversions could be reduced to convention and subtraction of the exponents $\bmod 2^m-1$. One would have two memories, one contains logs, and the other contains the antilogs. Another variation of using the table technique is that, rather than compute on the value, one can directly compute on the address. For instance, one want to compute $\alpha^i \cdot \alpha^j$, he can go to the address i of the log table and count up by j (2^m-1 modulo counter), that is, he will get $\alpha^{(i+j)}$.