6 Hardware implementation of Galois field operators

6.1 Gray map

The alphabet used in coding theory can be seen as a set of equally spaced points on a circumference. Generally, in communication channels, noise does not make all the errors equally probable, that is, it is much easier that a transmitted symbol is erroneously received as a near symbol rather than a far one. For example, it is much easier that on sending 4 we receive 3 or 5 rather than 1 or 10. With this hypothesis the Hamming distance (Chap. 1, Definition 1.5.7) is not the most suitable metric to measure errors.

Definition 6.1.1 Considered an alphabet in *GF(m)*, the *Lee weight* w_L of an integer i is defined by

$$w_L(i) = \min\{i, m-i\} \tag{6.1}$$

Definition 6.1.2 The *Lee distance* $d_L(x,y)$ is given by $w_L(x-y)$.

Suppose we are working in *GF(4)* with an alphabet composed of 0, 1, 2 and 3. Using the metric defined by the Lee distance, we note that for 0, 1 and 2 the weights are identical to the elements themselves, while this is not true for 3. In fact, 0 has weight 0, 1 has weight 1, 2 has weight 2, while 3 has weight 1.

In the case of multilevel memories, where the cell may be in one of the 2^m possible states, the first step in designing an ECC is encoding the alphabet of 2^m symbols through a binary alphabet.

The basic idea of Gray encoding is to map blocks of symbols of codewords in blocks of channel symbols, so that the most probable errors in the channel symbols cause a minimum number of error symbols in the codewords. In other words, we look for a natural map between $GF(2^m)$ and *GF(2)* so as to map the metrics constructed in the two fields, that is the Lee metrics into the Hamming metrics.

For example, we want to construct a Gray map between *GF(4)* and *GF(2)*. We introduce 3 functions from *GF(4)* towards *GF(2)* as described in Table 6.1.

Note that i written in the binary system is $(\alpha(i),\beta(i))$, besides $\varphi(i)=\alpha(i)+\beta(i)$, where the sum is intended to be a binary sum, that is an exclusive-OR.

Table 6.1. Functions needed to map *GF(4)* into *GF(2)*

$i \in GF(4)$	$\alpha(i)$	$\beta(i)$	$\varphi(i)$	$w_L(i)$
0	0	0	0	0
1	1	0	1	1
2	0	1	1	2
3	1	1	0	1

Definition 6.1.3 The *Gray map* is defined as $\theta(i)=(\beta(i),\varphi(i))$. The code $C'= \theta(C)$ is called *binary image* of C.

Note that generally C' is not linear since the Gray map is not linear. The crucial point is that the Gray map preserves the distance, that is, the Lee distance of two codewords in C is equal to the Hamming distance of their images.

Accordingly, the 4 states of a ML memory are respectively encoded as 11 10 00 01. Note that, using the decimal coding, the states would not have been encoded in the same way.

Definition 6.1.4 It is defined as *decimal coding* between $GF(2^m)$ and *GF(2)* the map that transcribes each decimal symbol of the alphabet $GF(2^m)$ in binary form.

In the case of decimal encoding of *GF(4)*, the states would have been encoded as 11 10 01 00.

With decimal encoding there is no continuity when shifting from one state to another (observe that the central transition involves an error of 2 bits). On the contrary, with the Gray encoding the transition from a state to another consecutive one involves an error of only one bit. Applying an error correction code with both encodings, we see that a code corrector of only one bit corrects all the transitions between adjacent states if a Gray encoding is used, while this is not true in the case of decimal encoding for which a code corrector of 2 bits is required.

In conclusion, the construction of the Gray map is equivalent to encode the states, so that the transition between adjacent states involves the change of only one bit.

6.2 Adders

As explained in Chap. 2, cyclic codes exist in Galois fields. To explain their implementation it is therefore fundamental to describe the hardware implementation of the Galois field operations.

It is easier to implement the arithmetic operations in the Galois field than the usual arithmetic operations, as they do not have any carries.

In order to describe them we need to keep in mind all the possible representations of the field elements, that is, the polynomial representation, the vector representation and the exponential representation (Chap. 2, Table 2.5).

In the case of the sum in $GF(q^m)$ the most useful representations are the vector and the polynomial ones. Adding two elements of the Galois field is equivalent to adding two polynomials, that is:

$$\begin{aligned}&\left(a_0 + a_1 q + \ldots + a_{m-1} q^{m-1}\right) + \left(b_0 + b_1 q + \ldots + b_{m-1} q^{m-1}\right) \\ &= c_0 + c_1 q + \ldots + c_{m-1} q^{m-1},\end{aligned} \tag{6.2}$$

where

$$c_i = a_i + b_i \qquad for \quad i = 0, \ldots, m-1 \tag{6.3}$$

If we consider only Galois fields of the type $GF(2^m)$ we have

$$\begin{cases} c_i = 0 & \text{if } a_i = b_i \\ c_i = 1 & \text{if } a_i \neq b_i \end{cases} \tag{6.4}$$

Thus, two Galois field elements are added either by modulo-two addition of their coefficients or, in binary form, producing the bit-by-bit exclusive-OR function of the two binary numbers.

The operation performed via software is equivalent to the binary sum of vector representations. The sum is implemented via hardware by the circuit shown in Fig. 6.1.

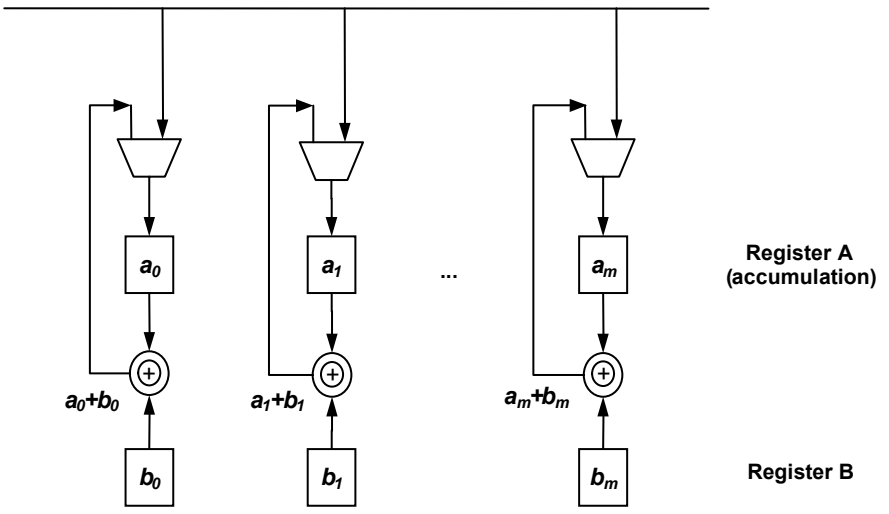

Fig. 6.1. Circuit that performs the sum of two Galois field elements

At the beginning the vector representation of the two elements to be added is loaded into the registers *A* and *B*. Their vector sum then appears at the input of register *A*. When the clock is pulsed, the sum is loaded into register *A* (register *A* is used as an accumulator).

In Appendix B (Sect. B.5.1) a routine implementing the sum is described.

6.3 Constant multipliers

The multiplication is a much more complicated operation than the addition. By multiplying two polynomials of degree *m–1* we get a polynomial of degree *2m–2* which is an invalid element of the Galois field.

For this reason, said *p(x)* the field generator polynomial, the multiplication is defined as the *product modulo p(x)*. The *product modulo p(x)* is obtained by dividing the product of the polynomials by *p(x)* and then keeping the remainder. Defined in this way, it is certain that the result is always of degree smaller than or at most equal to *m–1* and therefore a valid element of the field.

First of all, consider the multiplication by a fixed element of the Galois field. Suppose we are working in $GF(2^4)$, whose primitive polynomial is $p(x)=1+x+x^4$. We want to multiply β in $GF(2^4)$ by the fixed field element α^3.

To perform the multiplication in an efficient way, it is convenient to use the exponential representation. In fact, β will correspond to the exponential representation α^i and the multiplication is easily performed by adding the exponents, so the result of the product will be:

$$\delta = \beta \cdot \alpha^3 = \alpha^i \cdot \alpha^3 = \alpha^{3+i} \tag{6.5}$$

To perform this operation via software we need two tables. A table is necessary to have the exponential representation of β as α^i. Once the exponents are added, it is necessary to have the result as vector representation, so we need the other table, which lets us easily go from the exponential representation toward the vector representation. Substantially, the operation performed via software requires the storage of two tables of 2^m-1 elements.

In case we want to perform this operation via hardware, we need to implement the circuit performing the multiplication.

Since the hardware implementation of the adder requires the vector representation, we use this representation also for the multiplier.

Still considering the example described above, the element β is represented as

$$\beta = b_0 + b_1\alpha + b_2\alpha^2 + b_3\alpha \tag{6.6}$$

and the multiplication will be performed as

$$\alpha^3\beta = b_0\alpha^3 + b_1\alpha^4 + b_2\alpha^5 + b_3\alpha^6 \tag{6.7}$$

Remembering that the primitive element satisfies the minimum polynomial of the field we get the equality

$$\alpha^4 = \alpha + 1 \tag{6.8}$$

Substituting Eq. (6.8) in Eq. (6.7) we get:

$$\alpha^3 \beta = b_1 + (b_1 + b_2) \cdot \alpha + (b_2 + b_3) \cdot \alpha^2 + (b_0 + b_3) \cdot \alpha^3 \tag{6.9}$$

Taking into account this expression, it is possible to build the circuit represented in Fig. 6.2 which is able to multiply any element β in $GF(2^4)$ by α^3. To perform the multiplication, we first load the vector representation (b_0,b_1,b_2,b_3) of β into the register b_i. Then we pulse the clock. The new contents of the registers will be the vector representation of $\alpha_3 \beta$.

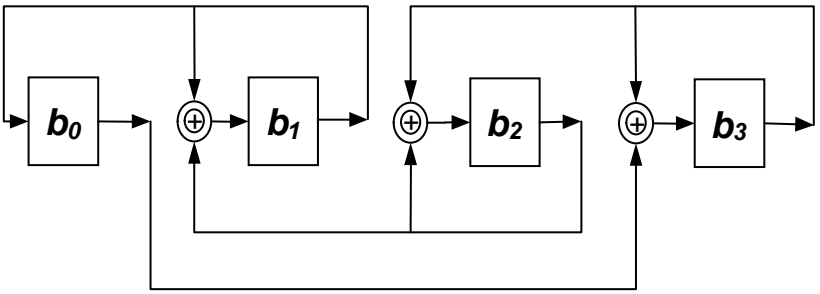

Fig. 6.2. Circuit that performs the product by α^3 of a field element in $GF(16)$

In general, every multiplication rule can be represented by a dedicated table. As an example, we will build the table in $GF(2^4)$ of the multiplication by the constant α^{12} represented in polynomial form (Fig. 6.3) as $\alpha^3 + \alpha^2 + \alpha + 1$.

The input bits contributing to a particular output bit are identified by the sum at the end of each column. Since the result can have degree 3 at most, the coefficients of degree 4, 5 and 6 must be written in the field, according to the field rule dictated by Eq. (6.8).

As the additions are modulo 2, these are implemented with exclusive-OR gates, as shown in Fig. 6.4.

Alternatively, each multiplier may be implemented as a look-up table with 2^m inputs which contains the result of the multiplication of each element by the constant. Obviously, this way becomes prohibitive as m increases.

	α^6	α^5	α^4	α^3	α^2	α^1	α^0
α^3	b_3	b_2	b_1	b_0	0	0	0
α^2		b_3	b_2	b_1	b_0	0	0
α^1			b_3	b_2	b_1	b_0	0
α^0				b_3	b_2	b_1	b_0
	b_3	b_2+b_3	$b_1+b_2+b_3$	0	0	$b_1+b_2+b_3$	$b_1+b_2+b_3$
				0	b_2+b_3	b_2+b_3	0
				b_3	b_3	0	0
				$b_0+b_1+b_2$	b_0+b_1	b_0	$b_0+b_1+b_2+b_3$

Fig. 6.3. Rule for the multiplication by α^{12} in *GF(16)*

A particular case concerns the multiplication in $GF(2^m)$ of an element β, represented by a polynomial of degree i, by x^j where $j<m-i$. The multiplication does not need to be performed modulo the field generator polynomial, since, as indicated by Eq. (6.10) the resultant element has degree equal to $m-1$ at most.

$$x^j \beta = x^j \left(a_0 + a_1 x + \ldots + a_i x^i\right) = a_0 x^j + a_1 x^{j+1} + \ldots + a_i x^{i+j} \tag{6.10}$$

Thus, the multiplication is reduced to a shift of the element β to the right of j positions.

Hereafter, we will see the construction of a generic multiplier, used to multiply any two field elements.

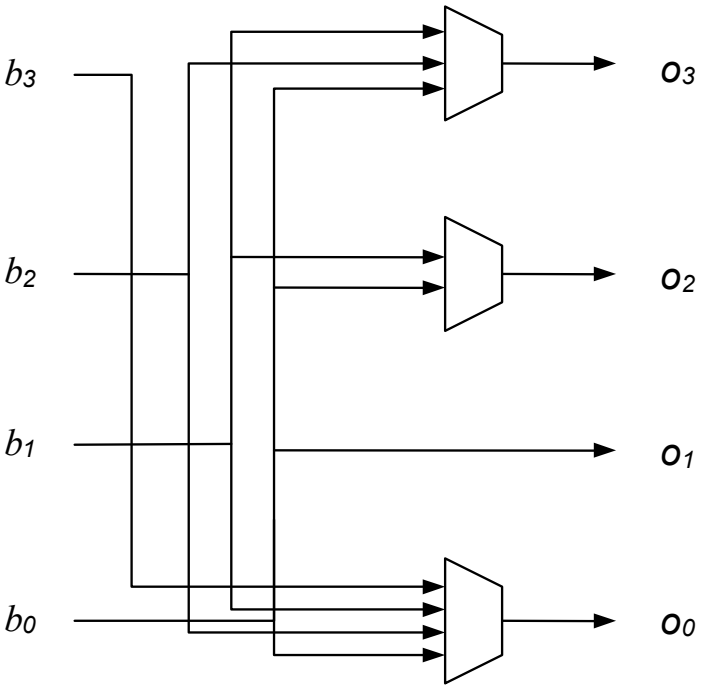

Fig. 6.4. Constant Multiplier by α^{12}

6.4 Full multipliers

When two generic elements of a Galois field $GF(q^m)$ are multiplied, the same considerations already made for the constant multiplier apply. In fact, also in this case, the multiplication is performed *modulo p(x)* as said above.

In order to implement this operation via software, it is useful to use the exponential representation. Similarly to the multiplication by constant, the method used consists in looking into a table for the representations of the two elements β and γ as powers of α and then adding their exponents. Finally, the result in exponential form is sought in the table again to have its vector representation.

$$\beta \cdot \gamma = \alpha^i \cdot \alpha^j = \alpha^{i+j} = \sigma \tag{6.11}$$

The hardware required to develop this operation is more complicated compared to the multiplication by constant. Always considering the case of $GF(2^4)$, two elements β and γ are expressed in polynomial form as

$$\beta = b_0 + b_1\alpha + b_2\alpha^2 + b_3\alpha^3 \tag{6.12}$$

$$\gamma = c_0 + c_1\alpha + c_2\alpha^2 + c_3\alpha^3 \tag{6.13}$$

The product $\beta\gamma$ can be expressed in the form described by Eq. (6.14), called Horner rule:

$$\beta \cdot \gamma = \left(\left(\left(c_3\beta\right)\cdot\alpha + c_2\beta\right)\cdot\alpha + c_1\beta\right)\cdot\alpha + c_0\beta \tag{6.14}$$

The circuit for the implementation of this equation, represented in Fig. 6.5, is composed of two blocks. The first one represents the multiplication by constant α, while the second one performs the other operations indicated in Eq. (6.14).

At the beginning the feedback shift registers of the circuit are empty while (b_0,b_1,b_2,b_3) and (c_0,c_1,c_2,c_3), i.e. the vector representations of β and γ, are loaded into the registers B and C respectively. Then the registers A and C are shifted four times. At the end of the first shift, register A contains $(c_3 b_0, c_3 b_1, c_3 b_2, c_3 b_3)$, i.e. the vector representation of $c_3\beta$. At the end of the second shift, register A contains the vector representation of $(c_3\beta)\alpha + c_2\beta$. At the end of the third shift, the content of register A forms the vector representation of $((c_3\beta)\alpha + c_2\beta)\alpha + c_1\beta$. At the end of the fourth shift, register A contains the product $\beta\gamma$ in vector form.

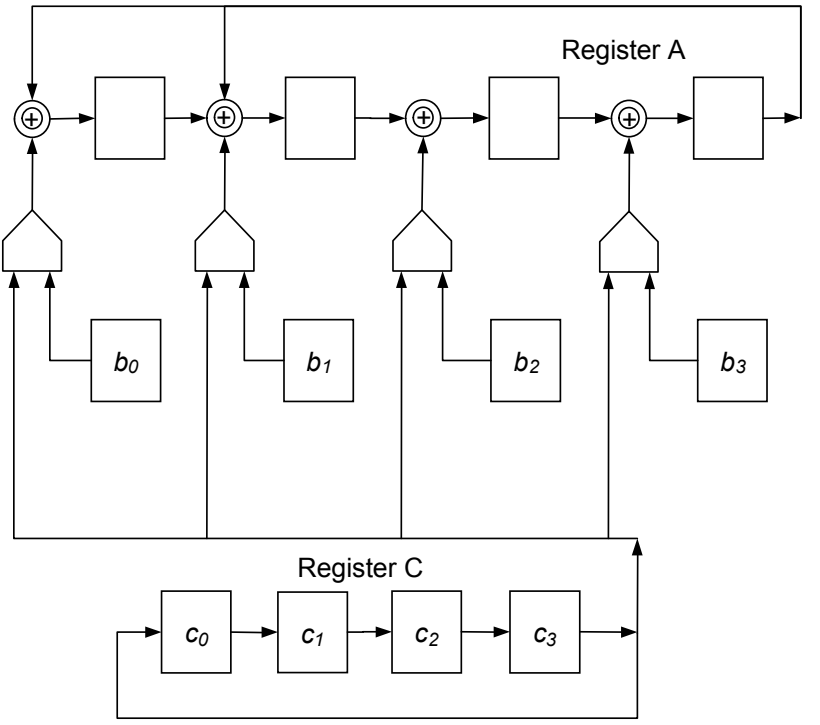

Fig. 6.5. Circuit for the multiplication of two generic elements in *GF(16)*

Also the multiplication of two elements of the Galois field $GF(2^m)$ can be represented by a dedicated table and is implemented as a combinatorial logic with 2^m inputs and m outputs.

Suppose, for example, we want to construct the full multiplier in $GF(2^4)$ with a combinatorial logic.

The multiplication of the two elements follows the scheme represented in Fig. 6.6. First the coefficients of the two elements are multiplied, thus getting 7 coefficients. The coefficients of degree higher than the third, indicated as *AB4*, *AB5* and *AB6* are summed to the coefficients of degree less than 4, according to the field rule dictated by Eq. (6.8). To create the combinatorial logic network, we build the matrix that performs the operation. The matrix contains m rows, which indicate the coefficients inside the vector: in other words, the first row indicates the coefficient of degree 0, the second one indicates the coefficient of degree 1 and so on. The columns are m^2: they indicate a_ib_j and are numbered so that the first m columns are a_0b_j, then the second m columns are a_1b_j and so on up to $a_{m-1}b_{m-1}$.

	α^6	α^5	α^4	α^3	α^2	α^1	$?^0$
α^3	b_3a_3	b_3a_2	b_3a_1	b_3a_0	0	0	0
α^2		b_2a_3	b_2a_2	b_2a_1	b_2a_0	0	0
α^1			b_1a_3	b_1a_2	b_1a_1	b_1a_0	0
α^0				b_0a_3	b_0a_2	b_0a_1	b_0a_0
	AB6	AB5	AB4	0	0	AB4	AB4
				0	AB5	AB5	0
				AB6	AB6	0	0

Fig. 6.6. Multiplication table of two generic elements of *GF(16)*

The matrix contains a 1 where the corresponding coefficient has to be added, otherwise there is a 0. In the example of $GF(2^4)$, the matrix is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{6.15}$$

The matrix described by Eq. (6.15) must then be multiplied by the vector c composed of m^2 coefficients as indicated by Eq. (6.16).

$$c = \left(a_0 b_0, a_0 b_1, \ldots, a_0 b_{m-1}, a_1 b_0, \ldots a_1 b_{m-1}, \ldots a_{m-1} b_{m-1}\right) \tag{6.16}$$

The advantage of this type of implementation is the speed. Anyway, as m increases it becomes prohibitively complex and costly.

In Appendix B (Sect. B.5.2) an implementation of a Galois field multiplication is described.

6.5 Divider

The division is the most complicated operation in a Galois field. Said β and γ two elements of the field, the operation β/γ is equivalent to the multiplication $\beta\, \gamma^{-1}$.

For a Galois field $GF(q^m)$ the following relation applies:

$$\alpha^{-i} = \alpha^{q^m - i - 1} \tag{6.17}$$

Thus, for each element it is simple to extract its inverse.

To implement the operation, we can use two different approaches. First of all, the operation may be implemented through a ROM which indicates the inverse of each field element. This "tabular" approach is quite easy if it is done via software, but it becomes prohibitive via hardware, especially in the case of very large field dimensions. In fact, the size of the required ROM is $2^m–1$ rows and m columns in case of fields $GF(2^m)$.

In the hardware case, it is possible to look for a dedicated combinatorial logic to perform the inversion, in the same way as the multiplier by constant was built. However, also in this case, as the field dimensions increase, the combinatorial logic becomes prohibitive.

When the field dimension is large, we prefer the following approach, by means of successive multiplications. The division is performed with $2m–1$ multiplications which alternate squares and multiplications by constant.

Suppose we want to perform the division of 2 elements a and x in $GF(2^4)$. The division follows Table 6.2 where the left column indicates the multiplication by the divisor x while the right column shows the successive powers to the square.

Table 6.2. Operations to be performed in order to have the inversion in GF(16)

a	a^2
a^2x	a^4x^2
a^4x^3	a^8x^6
a^8x^7	$a^{16}x^{14}$

At the end we get:

$$a^{16}x^{14} = a^{15}a\frac{x^{15}}{x} = \frac{a}{x} \tag{6.18}$$

exploiting the property (Sect. 2.3.1.1):

$$\alpha^{2^m-1} = 1 \tag{6.19}$$

Accordingly, whenever operations are implemented in an extended Galois field, i.e. $GF(2^m)$, which is the interesting case for applicative purposes, we need to keep in mind that the weight of a division is of $2m–1$ multiplications.

In Appendix B (Sect. B.5.3) a routine performing the division is described.

6.6 Linear shift register

Data rates in the hundreds or even thousands of megabits per second are common in many applications. Unfortunately, these date rates limit the technology of the device that may be used for the implementation of error control systems and, having fixed a given technology, they limit the complexity of the circuits. For this reason, it is important to implement the structures as repetitive as possible, by using simple exclusive-OR gates, switches or shift registers. Shift registers (SR) are some of the most simple digital circuits, since they are simply constituted by a collection of flip-flops connected in series. For this reason, given a technology, they can operate at a speed near to the maximum speed possible for a single gate.

In the binary case, shift registers are constituted by binary adders, that is exclusive-OR gates, a memory cell, which is a flip-flop, and multipliers by constant, which are represented either by the existence of a connection (multiplication by 1) or by the absence of a connection (multiplication by 0).

In the non binary case, the operators must be operators of the field $GF(2^m)$. Thus, a shift register is constituted by non binary adders of the type described in Sect. 6.2, by multipliers by constant of the type described in Sect. 6.3 and by memory cells containing more bits. The non binary memory cells are implemented as intuitively expected: a flip flop is devoted to the storage of each coefficient of the vector representation of the field element.

The product of two polynomials $m(x)$ and $g(x)$ may be calculated as a weighted sum of cyclic shifts of $g(x)$, as illustrated in Fig. 6.7.

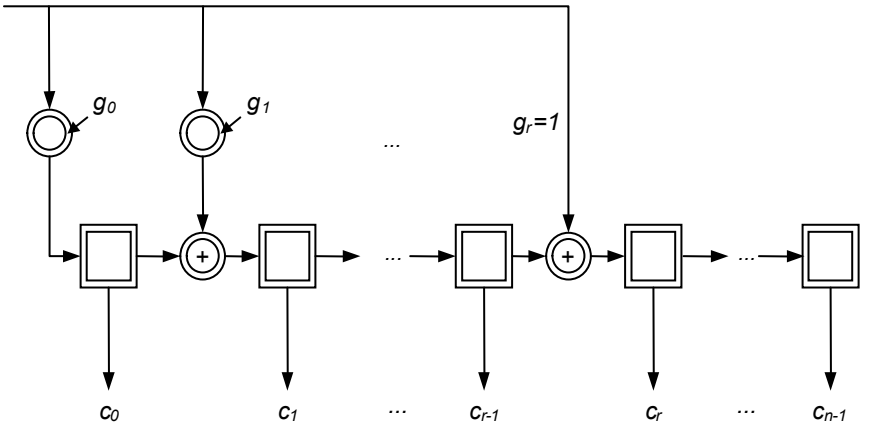

Fig. 6.7. Generic shift register for the multiplication of two polynomials

The coefficients of *m(x)* are loaded into the circuit in descending order of index. At each clock pulse a new coefficient is set as input and the contents of the memory cells are shifted of one cell to the right. When the final coefficient (m_0) has been loaded into the SR, the memory cells of the circuit contain the result of the polynomial multiplication.

We want, for instance, to calculate the product of $m(x)=1+x^2$ and $g(x)=1+x^2+x^3+x^4$. The circuit that implements the operation is represented in Fig. 6.8.

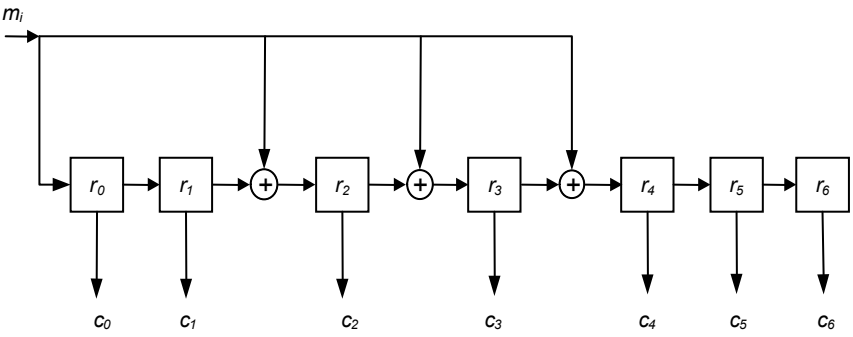

Fig. 6.8. Shift register for the binary multiplication of two polynomials

The contents of the SR cells during the multiplication are represented in Table 6.3.

As we will see later, shift registers are the basic elements encoding and decoding circuits of cyclic codes are constructed with.

Table 6.3. SR cells during the multiplication

SR cells	r_0	r_1	r_2	r_3	r_4	r_5	r_6
Initial state	0	0	0	0	0	0	0
Input $m_2=1$	1	0	1	1	1	0	0
Input $m_1=0$	0	1	0	1	1	1	0
Input $m_0=1$	1	0	0	1	0	1	1
Final state	1	0	0	1	0	1	1

Bibliography

T. C. Chen, S. W. Wei, H. J. Tsai, "Arithmetic Circuit for Finite Field GF(2^M)" in *US Patent 6,687,725 B1*, February 2004.

I. Dror, C. D. Gressel, M. Mostovoy, A. Molchanov, "Extending the Range of Computational Fields of Integers" in *US Patent US2002/0039418 A1*, April 2002.

S. V. Fedorenko, P. V. Trifonov, "Finding Roots of Polynomials Over Finite Fields" in *IEEE Transactions on communications*, Vol. 50, November 2002.

G. L. Feng, "A VLSI Architecture for Fast Inversion in *GF(2^m)*" in *IEEE Transactions on Computers*, Vol. 38, October 1989.

C. D. Gressel, A. Schevachman, E. Aizman, M. Slobodkin, S. Cooper, "Random Number Slip and Swap Generators" in *US Patent US2004/0205095 A1*, October 2004.

S. Lin, D. J. Costello, "Error Control Coding: Fundamentals and Applications", *F. F. Kuo Editor*, 1983.

M. Metha, V. Parmmar, E. Swartzlander Jr., "High-Speed Multiplier Design Using Multi-Input Counter and Compressor Circuits" in *IEEE Conference on Computer Arithmetic*, 1991.

C. Paar, "Optimized Arithmetic for Reed-Solomon Encoders" in *ISIT*, June 1997.

E. M. Popovici, P. Fitzpatrick, "Division Algorithm over GF(2^m)" in *Electronics Letters*, Vol. 34, September 1998.

M. Potkonjak, M. B. Srivastava, A. P. Chandrakasan, "Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination" in *IEEE Transactions on Computers-Aided Design of Integrated Circuits and Systems*, Vol. 15, February 1996.

R. Schober, W. H. Gerstacker, "The Zeros of Random Polynomials: Further Results and Applications" in *IEEE Transactions on Communications*, Vol. 50, June 2002.

S. B. Wicker, "Error Control for Digital Communication and Storage", *Prentice Hall*, 1995.