6

Binary Finite Field Arithmetic

In this Chapter we review some of the most relevant arithmetic algorithm on binary extension fields $GF(2^m)$. The arithmetic over $GF(2^m)$ has many important applications in the domains of theory of code theory and in cryptography [221, 227, 380]. Finite field's arithmetic operations include: addition, subtraction, multiplication, squaring, square root, multiplicative inverse, division and exponentiation.

Addition and subtraction are equivalent operations in $GF(2^m)$. Addition in binary finite fields is defined as polynomial addition and can be implemented simply as the XOR addition of the two m-bit operands.

That is why we begin this Section with a review of the main algorithms reported in the open literature for perhaps the most important field arithmetic operation: field multiplication.

6.1 Field Multiplication

Let $A(x), B(x)$ and $C'(x) \in GF(2^m)$ and $P(x)$ be the irreducible polynomial generating $GF(2^m)$. Multiplication in $GF(2^m)$ is defined as polynomial multiplication modulo the irreducible polynomial $P(x)$, namely,

$$C'(x) = A(x)B(x) \bmod P(x).$$

One important factor for designing multipliers in binary extension fields is the way that field elements are represented, i.e, the sort of basis that is being used[1]. Indeed, field element representation has a crucial role in the design of architectures for arithmetic operations.

Besides the polynomial or canonical basis, several other bases have been proposed for the representation of elements in binary extension fields [221, 51, 390]. Among them, probably the most studied one is the Gaussian normal basis [281, 285, 164, 89, 405].

[1] More details about field element representation can be found in §4.2.

Even though efficient bit-parallel multipliers for both canonical and normal basis representation have been regularly reported in the specialized literature, in this Section we will mainly focus on polynomial basis multiplier schemes, mostly because they are consistently more efficient than their counterparts in other bases[2].

Traditionally, the space complexity of bit parallel multipliers is expressed in terms of the number of 2-input AND and XOR gates. For reconfigurable hardware devices though, the total number of CLBs and/or LUTs utilized by the design is preferred. Depending on their space complexity, bit parallel multipliers are classified into two categories: quadratic and subquadratic space complexity multipliers.

Several quadratic and subquadratic space complexity multipliers have been reported in literature. Examples of quadratic multipliers can be found in [220, 182, 389, 390, 350, 129, 352, 315, 129, 282, 391, 112, 201, 292, 283, 284, 247, 90, 146]. On the other hand, some examples of sub-quadratic multipliers can be found in [267, 268, 269, 270, 291, 86, 298, 117, 293, 349, 16, 106, 91, 377, 239]. This latter category offers low space complexity especially for large values of n and therefore they are in principle attractive for cryptographic applications.

Among the several approaches for computing the product $C'(x)$, we will study the following strategies,

- Two-Step multipliers
- Interleaving Multiplication
- Matrix-Vector Multipliers
- Montgomery Multiplier

In the case of two-step multipliers, first the polynomial product $C(x)$ of degree at most $2m-2$ is obtained as,

$$C(x) = A(x)B(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} b_i x^i) \tag{6.1}$$

Then, in a second step, the reduction operation needs to be performed in order to obtain the $m-1$ degree polynomial $C'(x)$, which is defined as

$$C'(x) = C(x) \bmod P(x) \tag{6.2}$$

It is noticed that once the irreducible polynomial $P(x)$ has been selected, the reduction step can be accomplished by using XOR gates only.

In the rest of this section different implementation aspects and several efficient methods for computing $GF(2^m)$ finite field multiplication are extensively studied. In § 6.1.1 the analysis of the school or classical method is presented. Subsection § 6.1.2 analyzes a variation of the classical Karatsuba-Ofman algorithm as one of the most efficient techniques to find the polynomial product of

[2] Examples of efficient normal basis multiplier designs recently published in the open literature can be found in [164, 89, 285, 281, 405, 352, 283].

product of Equation 6.1. In subsection § 6.1.3 we describe an efficient method to compute polynomial squaring in hardware, at a complexity cost of just $O(1)$. Subsections § 6.1.4 and § 6.1.5 explain an efficient hardware methodology that carries on the reduction step of Equation 6.2 considering three separated cases, namely, reduction with irreducible trinomials, pentanomials and arbitrary polynomials. Then in §6.1.6 a method that interleaves the steps of multiplication and reduction is presented. Subsection §6.1.7 outlines field multiplication methods that solve Equation 6.1 by reformulating it in terms of matrix-vector operations. Then, in §6.1.8, the binary field version of the Montgomery multiplier is discussed. Finally, §6.1.9 compares the most relevant binary field multiplier designs published up-to date. Designs are compared from the perspective of three different metrics, namely, speed, compactness and efficiency.

6.1.1 Classical Multipliers and their Analysis

Let $A(x), B(x)$ be elements of $GF(2^m)$, and let $P(x)$ be the degree m irreducible polynomial generating $GF(2^m)$. Then, the field product $C'(x) \in GF(2^m)$ can be obtained by first computing the polynomial product $C(x)$ as

$$C(x) = A(x)B(x) = \left(\sum_{i=0}^{m-1} a_i x^i\right)\left(\sum_{i=0}^{m-1} b_i x^i\right) . \tag{6.3}$$

Followed by a reduction operation, performed in order to obtain the $(m-1)$-degree polynomial $C'(x)$, which is defined as

$$C'(x) = C(x) \bmod P(x) . \tag{6.4}$$

Once the irreducible polynomial $P(x)$ is selected and fixed, the reduction step can be accomplished using only XOR gates. The classical algorithm formulates these two steps into a single matrix-vector product, and then reduces the product matrix using the irreducible polynomial that generates the field. The degree $2m-2$ polynomial $C(x)$ in (6.3) can be written as,

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{m-2} \\ c_{m-1} \\ c_m \\ c_{m+1} \\ \vdots \\ c_{2m-3} \\ c_{2m-2} \end{bmatrix} = \begin{bmatrix} a_0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ a_1 & a_0 & 0 & 0 & \cdots & 0 & 0 \\ a_2 & a_1 & a_0 & 0 & \cdots & 0 & 0 \\ \vdots & & \vdots & & \ddots & \vdots & \vdots \\ a_{m-2} & a_{m-3} & a_{m-4} & a_{m-5} & \cdots & a_0 & 0 \\ a_{m-1} & a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_1 & a_0 \\ 0 & a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & a_{m-2} & \cdots & a_3 & a_2 \\ \vdots & & \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & 0 & \cdots & 0 & a_{m-1} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-2} \\ b_{m-1} \end{bmatrix} \tag{6.5}$$

The computation of the field product $C'(x)$ in (6.4) can be accomplished by first computing the above matrix-vector product to obtain the vector C which has $2m-1$ elements. By taking into account the zero entries of the matrix, we obtain the gate complexity of the computation of $C(x)$ in Table 6.1.

Table 6.1. The Computation of $C(x)$ Using Equation (6.5)

Coordinates	AND Gates	XOR Gates	T_A	T_X
c_i for $0 \le i \le m-1$	$i+1$	i	1	$\log_2\lceil i+1 \rceil$
c_{m+i} for $0 \le i \le m-2$	$m-(i+1)$	$m-(i+1)-1$	1	$\log_2\lceil m-1-i \rceil$

Therefore, the total number of gates are found as

$$\text{AND Gates: } 1+2+\cdots+m+(m-1)+(m-2)+\cdots+2+1 = m^2 ,$$
$$\text{XOR Gates: } 1+2+\cdots+(m-1)+(m-2)+\cdots+2+1 = (m-1)^2 .$$

The AND gates operate all in parallel, and require a single AND gate delay T_A. On the other hand, the XOR gates are organized as a binary tree of depth $\log_2\lceil j \rceil$ in order to add j operands. The total time complexity is then found by taking the largest number of terms, which is equal to m for the computation of c_{m-1}. Therefore, the total complexity of computing the matrix-vector product (6.5) so that the elements c_i for $i = 0, 1, \ldots, 2m-2$ are all found is given as,

$$\begin{aligned} &\text{AND Gates} = m^2 \\ &\text{XOR Gates} = (m-1)^2 \\ &\text{Total Delay} = T_A + \lceil \log_2 m \rceil T_X . \end{aligned} \tag{6.6}$$

Notice that this computation must be followed by reduction modulo the irreducible polynomial $P(x)$. The reduction operation is discussed in Section 6.1.4.

6.1.2 Binary Karatsuba-Ofman Multipliers

Several architectures have been reported for multiplication in $GF(2^m)$. For example, efficient bit-parallel multipliers for both canonical and normal basis representation have been proposed in [136, 351, 241, 389, 20]. All these algorithms exhibit a space complexity $O(m^2)$. However, there are some asymptotically faster methods for finite field multiplications, such as the Karatsuba-Ofman algorithm [168, 268]. Discovered in 1962, it was the first algorithm to accomplish polynomial multiplication in under $O(m^2)$ operations [14]. Karatsuba-Ofman multipliers may result in fewer bit operations at the expense of some design restrictions, particularly in the selection of the degree of the generating irreducible polynomial m.

In [268], it was presented a Karatsuba-Ofman multiplier based on composite fields of the type $GF((2^n)^s)$ with $m = sn$, $s = 2^t$, t an integer. However, for certain applications, quite particularly, elliptic curve cryptosystems, it is important to consider finite fields $GF(2^m)$ where m is not necessarily a power of two. In fact, for this specific application some sources [145] suggest that, for security purposes, it is strongly recommended to choose degrees m primes for finite fields in the range $[160, 512]$.

In the rest of this subsection we will briefly describe a variation of the classic Karatsuba-Ofman Multiplier called *binary Karatsuba-Ofman multipliers* that was first presented in [293]. Binary Karatsuba-Ofman multipliers can be utilized arbitrarily, regardless the form of the required degree m.

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ of degree $m = rn$, with $r = 2^k$, k an integer. Let A, B be two elements in $GF(2^m)$. Both elements can be represented in the polynomial basis as,

$$\begin{aligned} A = \sum_{i=0}^{m-1} a_i x^i &= \sum_{i=\frac{m}{2}}^{m-1} a_i x^i + \sum_{i=0}^{\frac{m}{2}-1} a_i x^i \\ &= x^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} a_{i+\frac{m}{2}} x^i + \sum_{i=0}^{\frac{m}{2}-1} a_i x^i = x^{\frac{m}{2}} A^H + A^L \end{aligned}$$

and

$$\begin{aligned} B = \sum_{i=0}^{m-1} b_i x^i &= \sum_{i=\frac{m}{2}}^{m-1} b_i x^i + \sum_{i=0}^{\frac{m}{2}-1} b_i x^i \\ &= x^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} b_{i+\frac{m}{2}} x^i + \sum_{i=0}^{\frac{m}{2}-1} b_i x^i = x^{\frac{m}{2}} B^H + B^L. \end{aligned}$$

Then, using last two equations, the polynomial product is given as

$$C = x^m A^H B^H + (A^H B^L + A^L B^H) x^{\frac{m}{2}} + A^L B^L. \tag{6.7}$$

Karatsuba-Ofman algorithm is based on the idea that the product of last equation can be equivalently written as,

$$\begin{aligned} C = {} & x^m A^H B^H + A^L B^L + \\ & (A^H B^H + A^L B^L + (A^H + A^L)(B^L + B^H)) x^{\frac{m}{2}} \\ = {} & x^m C^H + C^L. \end{aligned} \tag{6.8}$$

Let us define

$$\begin{aligned} M_A &:= A^H + A^L; \\ M_B &:= B^L + B^H; \\ M &:= M_A M_B. \end{aligned} \tag{6.9}$$

Using Equation 6.8, and taking into account that the polynomial product C has at most $2m - 1$ coordinates, we can classify its coordinates as,

$$\begin{aligned} C^H &= [c_{2m-2}, c_{2m-3}, \ldots, c_{m+1}, c_m]; \\ C^L &= [c_{m-1}, c_{m-2}, \ldots, c_1, c_0]. \end{aligned} \tag{6.10}$$

Although (6.8) seems to be more complicated than (6.7), it is easy to see that Equation (6.8) can be used to compute the product at a cost of four polynomial additions and three polynomial multiplications. In contrast, when using equation (6.7), one needs to compute four polynomial multiplications and three polynomial additions. Due to the fact that polynomial multiplications are in general much more expensive operations than polynomial additions, it is valid to conclude that (6.8) is computationally simpler than the classic algorithm.

Algorithm 6.1 $mul2^k(C, A, B)$: $m = 2^k n$-bit Karatsuba-Ofman Multiplier

Require: Two elements $A, B \in GF(2^m)$ with $m = rn = 2^k n$, where A, B can be expressed as $A = x^{\frac{m}{2}} A^H + A^L, B = x^{\frac{m}{2}} B^H + B^L$.
Ensure: A polynomial $C = AB$ with up to $2m-1$ coordinates, where $C = x^m C^H + C^L$.
1: **if** $r == 1$ **then**
2: $C = mul\_n(A, B)$;
3: **Return**(C)
4: **end if**
5: **for** i from 0 to $\frac{r}{2} - 1$ **do**
6: $M_{Ai} = A_i^L + A_i^H$;
7: $M_{Bi} = B_i^L + B_i^H$;
8: **end for**
9: $mul2^k(C^L, A^L, B^L)$;
10: $mul2^k(M, M_A, M_B)$;
11: $mul2^k(C^H, A^H, B^H)$;
12: **for** i from 0 to $r - 1$ **do**
13: $M_i = M_i + C_i^L + C_i^H$;
14: **end for**
15: **for** i from 0 to $r - 1$ **do**
16: $C_{\frac{r}{2}+i} = C_{\frac{r}{2}+i} + M_i$;
17: **end for**
18: **Return**(C).

Karatsuba-Ofman's algorithm can be applied recursively to the three polynomial multiplications in (6.8). Hence, we can postpone the computations of the polynomial products $A^H B^H$, $A^L B^L$ and M, and instead we can split again each one of these three factors into three polynomial products. By applying this strategy recursively, in each iteration each degree polynomial multiplication is transformed into three polynomial multiplications with their degrees reduced to about half of its previous value.

Eventually, after no more than $\lceil \log_2(m) \rceil$ iterations, all the polynomial operands collapse into single coefficients. In the last iteration, the resulting bit

multiplications can be directly computed. Although it is possible to implement the Karatsuba-Ofman algorithm until the $\lceil \log_2 m \rceil$ iteration, it is usually more practical to truncate the algorithm earlier. If the Karatsuba-Ofman algorithm is truncated at a certain point, the remaining multiplications can be computed by using alternative techniques[3].

Let us consider the algorithm presented in Algorithm 6.1. If $r = 1$, then the product is trivially found in lines 1-3 as the result of the single n-bit polynomial multiplication $C = mul\_n(A, B)$. Otherwise, in the first loop of the algorithm (lines 4-6) the polynomials M_A and M_B of equation (6.9) are computed by a direct polynomial addition of $A^H + A^L$ and $B^H + B^L$, respectively. In lines 7-9, C^L, C^H and M, are obtained via $\frac{r}{2}$-bit polynomial multiplication. After completion of these polynomial multiplications, the final value of the lower half of C^L as well as the upper half of C^H are found. To find the final values of the upper half of the polynomial C^L and the lower half of C^H, we need to combine the results obtained from the multiplier blocks with the polynomials C^H, C^L and M, as described in equations (6.8) and (6.9). This final computation is implemented in lines 10 through 13 of figure 6.1.

Complexity Analysis

The space complexity of the Algorithm 6.1 can be estimated as follows. The computation of the loop in lines 4-6 requires $2(\frac{r}{2}) = r$ additions. The execution of lines 7-9, implies the cost of 3 $\frac{r}{2}$-bit polynomial multipliers. Finally, lines 10-13 can be computed with a total of $3r$ additions. Notice that if $n > 1$ the additions in Algorithm 6.1 need to be multi-bit operations. Noticing also that m-bit multiplications in $GF(2)$ can generate at most $(2m - 1)$-bit products, we can have an extra saving of four bit-additions in lines 11 and 13. Hence, the addition complexity per iteration of the $m = 2^k n$-bits Karatsuba-Ofman multiplier presented in Algorithm 6.1 is given as $r + 3r = 4r$ n-bit additions plus three times the number of additions needed in a $\frac{r}{2}$ multiplier block, minus four bit additions. Notice that for n-bit arithmetic, each one of these additions can be implemented using n XOR gates.

Recall that m is a composite number that can be expressed as $m = rn$, with $r = 2^k$, k an integer. Then, one can successively invoke $\frac{r}{2^i}$-bit multiplier blocks, 3^i times each, for $i = 1, 2, \ldots, \log_2 r$. After $k = \log_2 r$ iterations, all the multiplier operations will involve polynomial multiplicands with degree n. These multiplications can be then computed using an alternative technique, like the classic algorithm. By applying iteratively the analysis given above, one can see that the total XOR gate complexity of the $m = 2^k n$-bit hybrid Karatsuba-Ofman multiplier truncated at the n-bit operand level is given as

[3] such as the classical algorithm studied in §6.1.1 or other techniques

$$\begin{aligned}
\text{XOR Gates} &= M_{xor2^n} 3^{\log_2 r} + \sum_{i=1}^{\log_2 r} 3^{i-1}\left(\frac{8rn}{2^i} - 4\right) \\
&= M_{xor2^n} 3^{\log_2 r} + 8rn \sum_{i=1}^{\log_2 r} \frac{3^{i-1}}{2^i} - 4 \sum_{i=1}^{\log_2 r} 3^{i-1} \\
&= M_{xor2^n} 3^{\log_2 r} + \frac{8}{3} rn \sum_{i=1}^{\log_2 r} \frac{3^i}{2} - \frac{4}{3} \sum_{i=1}^{\log_2 r} 3^i \\
&= M_{xor2^n} 3^{\log_2 r} + 8rn\left(\frac{3}{2}^{\log_2 r} - 1\right) - 2(3^{\log_2 r} - 1) \\
&= M_{xor2^n} 3^{\log_2 r} + 8rn(r^{\log_2 \frac{3}{2}} - 1) - 2(r^{\log_2 3} - 1) \\
&= M_{xor2^n} r^{\log_2 3} + 8n(r^{\log_2 3} - 8r) - 2(r^{\log_2 3} - 1) \\
&= r^{\log_2 3}(8n - 2 + M_{xor2^n}) - 8rn + 2 \\
&= \left(\frac{m}{n}\right)^{\log_2 3} \left(8\frac{m}{r} - 2 + M_{xor2^n}\right) - 8m + 2.
\end{aligned}$$

Where M_{xor2^n} represents the XOR gate complexity of the block selected to implement the n-bit multipliers.

Similarly, notice that no AND gate is needed in Algorithm 6.1, except when the block selected to implement the n-bit multiplier is called. Let M_{and2^n} be the AND gate complexity of the block selected to implement the n-bit multiplier. Then, since this block is called exactly $3^{\log_2 r}$ times, we conclude that the total number of AND gates needed to implement the algorithm in 6.1 is given as,

$$\text{AND gates} = r^{\log_2 3} M_{and2^n} = \left(\frac{m}{n}\right)^{\log_2 3} M_{and2^n}$$

We give the time complexity of Algorithm 6.1 as follows. The execution of the first loop in lines 4-6 can be computed in parallel in a hardware implementation. Therefore, the required time for this part of the algorithm is of just 1 n-bit addition delay, which is equal to an XOR gate delay T_X. Lines 7-9, can also be implemented in parallel. Thus, the associated cost is of one $\frac{r}{2}$-bit multiplier delay. Notice that we cannot implement this second part of the algorithm in parallel with the first one because of the inherent dependencies of the variables. Finally, lines 10-13 can be computed with a delay of just $3T_X$. Hence, the associated time delay of the $m = 2^k n$-bit Karatsuba-Ofman multiplier of figure 6.1 is given as

$$\text{Time Delay} = T_{delay2^n} + \sum_{i=1}^{\log_2 r} 3 = T_{delay2^n} + 4T_X \log_2 r.$$

In this case it has been assumed that the block selected to implement the $GF(2^n)$ arithmetic has a T_{delay2^n} gate delay associated with it.

In summary, the space and time complexities of the m-bit Karatsuba-Ofman multiplier are given as

$$\begin{array}{l} \text{XOR Gates} \leq \left(\frac{m}{n}\right)^{\log_2 3}\left(8\frac{m}{r} - 2 + M_{xor2^n}\right) - 8m + 2 \ ; \\ \text{AND Gates} \leq \left(\frac{m}{n}\right)^{\log_2 3} M_{and2^n} \ ; \\ \text{Time Delay} \leq T_{delay2^n} + 4T_X \log_2\left(\frac{m}{n}\right) \ . \end{array} \tag{6.11}$$

As it has been mentioned above, the hybrid approach proposed here requires the use of an efficient multiplier algorithm to perform the n-bit polynomial multiplications. Let us recall that in §6.1.1 above, it was found that the space and time complexities for the classic n-bit multiplier are given as

$$\begin{array}{l} \text{XOR Gates} = (n-1)^2 \ ; \\ \text{AND Gates} = n^2 \ ; \\ \text{Time Delay} \leq T_{AND} + T_X \lceil \log_2 n \rceil \ . \end{array} \tag{6.12}$$

Combining the complexities given in equation (6.12), together with the complexities of equation (6.11) we conclude that the space and time complexities of the hybrid m-bit Karatsuba-Ofman multiplier truncated at the n-bit multiplicand level are upper bounded by

$$\begin{array}{lcl} \text{XOR Gates} & \leq & \left(\frac{m}{n}\right)^{\log_2 3}(8n - 2 + M_{xor2^n}) - 8m + 2 \\ & = & \left(\frac{m}{n}\right)^{\log_2 3}(n^2 + 6n - 1) - 8m + 2 \ ; \\ \text{AND Gates} & \leq & 3^{\log_2 r} M_{and2^n} \ = \ \left(\frac{m}{n}\right)^{\log_2 3} n^2 ; \\ \text{Time Delay} & \leq & T_{AND} + T_X(\log_2 n + 4\log_2 r) \ . \end{array} \tag{6.13}$$

Let us consider now the cases where m is a power of two, $m = rn = 2^k$, $k > 2$. Then, $n = 4$ is the most optimal selection for the hybrid Karatsuba-Ofman algorithm. For this case using equation (6.13) we obtain

$$\begin{array}{l} \text{XOR Gates} \leq \left(\frac{m}{n}\right)^{\log_2 3}(n^2 + 6n - 1) - 8m + 2 \\ \qquad = \left(\frac{2^k}{4}\right)^{\log_2 3}(4^2 + 6 \cdot 4 - 1) - 8 \cdot 2^k + 2 \\ \qquad = 13 \cdot 3^{k-1} - 2^{k+3} + 2; \\ \text{AND Gates} \leq \left(\frac{m}{n}\right)^{\log_2 3} n^2 \ = \ \left(\frac{2^k}{4}\right)^{\log_2 3} 4^2 \ = \ 16 \cdot 3^{k-2}; \\ \text{Time Delay} \leq T_{AND} + T_X(\log_2 n + 4\log_2 r) \ = \\ \qquad = T_{AND} + T_X(\log_2 4 + 4\log_2 2^{k-2}) \ = \ T_{AND} + T_X(4k - 6) \ . \end{array} \tag{6.14}$$

Table 6.2 shows the space and time complexities for the hybrid Karatsuba-Ofman multiplier using the results found in equation (6.14). The values of m presented in Table 6.2 correspond to the first ten powers of two, i.e., $m = 2^k$ for $i = 0, 1, \ldots, 9$. Notice that the multipliers for $m = 1, 2, 4$ are assumed to be implemented using the classical method only. As we will see, the complexities of the hybrid Karatusba multiplier for degrees $m = 2^k$ happen to be crucial to find the hybrid Karatsuba-Ofman complexities for arbitrary degrees of m.

Table 6.2. Space and Time Complexities for Several $m = 2^k$-bit Hybrid Karatsuba-Ofman Multipliers

m	r	n	AND gates	XOR gates	Time delay	Area (in NAND units)
1	1	1	1	0	T_A	1.26
2	1	2	4	1	$T_X + T_A$	7.24
4	1	4	16	9	$2T_X + T_A$	39.96
8	2	4	48	55	$6T_X + T_A$	181.48
16	4	4	144	225	$10T_X + T_A$	676.44
32	8	4	432	799	$14T_X + T_A$	2302.12
64	16	4	1296	2649	$18T_X + T_A$	7460.76
128	32	4	3888	8455	$22T_X + T_A$	23499.88
256	64	4	11664	26385	$26T_X + T_A$	72743.64
512	128	4	34992	81199	$30T_X + T_A$	222727.72

Binary Karatsuba-Ofman Multipliers

In order to generalize the Karatsuba-Ofman algorithm of Algorithm 6.1 for arbitrary degrees m, particularly m primes, let us consider the multiplication of two polynomials $A, B \in GF(2^m)$, such that their degree is less or equal to $m-1$, where $m = 2^k + d$.

$$A = \underbrace{[\overbrace{0, \ldots, 0, 0}^{2^{k+1}-d}, a_{2^k+d-1}, \ldots, a_{2^k+1}, a_{2^k}}_{A^H}, \overbrace{a_{2^k-1}, a_{2^k-2}, \ldots, a_2, a_1, a_0}^{A^L}];$$

$$A^H = [0, \ldots, 0, 0, a_{2^k+d-1}, \ldots, a_{2^k+1}, a_{2^k}];$$
$$A^L = [a_{2^k-1}, a_{2^k-2}, \ldots, a_2, a_1, a_0];$$

Fig. 6.1. Binary Karatsuba-Ofman Strategy

As a very first approach, we could pretend that both operands have 2^{k+1} coordinates each, where their respective $2^{k+1} - d$ most significant bits are all equal to zero. Figure 6.1 shows how the sub-polynomials A^H and A^L will be redefined according with this approach. If we partition the operands A and B as shown in Figure 6.1, then, in order to compute their polynomial multiplication, we can use the regular Karatsuba-Ofman algorithm with $m = 2^{k+1}$. Although this approach is obviously valid, it clearly implies the waste of several arithmetic operations, as some of the most significant bits of the operands are zeroes. However, if we were able to identify the extra arithmetic operations and remove them from the computation, we would then be able to find a quasi-optimal solution for arbitrary degrees of m. To see how this can

be done, consider the Algorithm 6.2, which has been adapted from Algorithm 6.1 previously studied.

Algorithm 6.2 *mulgen_d*(C, A, B): m-bit Binary Karatsuba-Ofman Multiplier

Require: Two elements $A, B \in GF(2^m)$ with m an arbitrary number, and where A, B can be expressed as $A = x^{\frac{m}{2}} A^H + A^L$, $B = x^{\frac{m}{2}} B^H + B^L$.

Ensure: A polynomial $C = AB$ with up to $2m-1$ coordinates, where $C = x^m C^H + C^L$.

1: $k = \lfloor \log_2 m \rfloor$;
2: $d = m - 2^k$;
3: **if** $d == 0$ **then**
4: $\quad C = Kmul2^k(A, B)$;
5: $\quad$ **return**(C);
6: **end if**
7: **for** i from 0 to $d-1$ **do**
8: $\quad M_{Ai} = A_i^L + A_i^H$;
9: $\quad M_{Bi} = B_i^L + B_i^H$;
10: **end for**
11: $mul2^k(C^L, A^L, B^L)$;
12: $mul2^k(M, M_A, M_B)$;
13: $mulgen\_d(C^H, A^H, B^H)$;
14: **for** i from 0 to $2^k - 2$ **do**
15: $\quad M_i = M_i + C_i^L + C_i^H$;
16: **end for**
17: **for** i from 0 to $2^k - 2$ **do**
18: $\quad C_{k+i} = C_{k+i} + M_i$;
19: **end for**
20: **Return**(C).

In lines 1-2 the values of the constants k, d such that $m = 2^k + d$, are computed. If $d = 0$, i.e, if m is a power of two, then the binary Karatsuba-Ofman Algorithm 6.2 reverts to the specialized Algorithm 6.1 presented previously. If that is not the case, Algorithm 6.2 uses the constants k and d to prevent us to compute unnecessary arithmetic operations. In lines 6-8, the d least significant bits of M_A and M_B of equation (6.9) are computed using the d non-zero coordinates of A^H and B^H. The remaining $k-d$ most significant bits of M_A and M_B are directly obtained from A^L and B^L, respectively. Notice that the operands, A^L, B^L, M_A and M_B are all 2^k-bit polynomials. Because of that, our algorithm invokes the multiplier of Algorithm 6.1 in lines 9 and 10. On the other hand, both operands A^H and B^H are d-bit polynomials, where d, in general, is not a power of two. Consequently, in line 11, the algorithm calls itself in a recursive manner. This recursive call is invoked using the operand's degree reduced to d. In each iteration the degree of the operands gets reduced,

and eventually, after a total of h iterations (where h is the hamming weight of the binary representation of the original degree m), the algorithm ends.

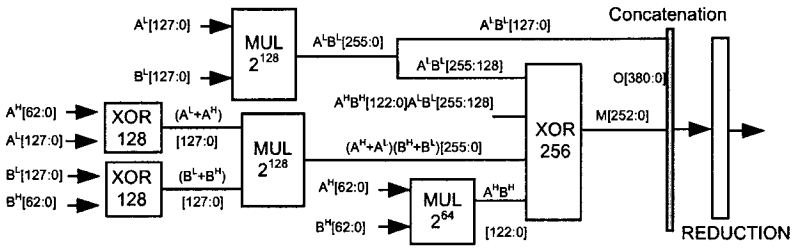

Fig. 6.2. Karatsuba-Ofman Multiplier $GF(2^{191})$

As a design example, consider the binary Karatsuba-Ofman multiplier shown in Figure 6.2. That circuit computes the polynomial multiplication of the elements A and $B \in GF(2^{191})$. Notice that for this case $m = 191 = 2^k + d = 2^7 + 63$. Since $(191)_2 = 10111111$, the Hamming weight h of the binary representation of m is $h = 7$. This implies that we would need a total of seven iterations in order to compute the multiplication using the generalized m-bit binary Karatsuba-Ofman multiplier.

However we can do much better by assuming that the d = 63 most significant leftover bits are 64 (implying $m = (192)_2 = 11000000$). Hence, algorithm 6.2 can finish the computation in only two iterations, as shown in Figure 6.2.

By using the complexity figures listed in Table 6.2, we can estimate the space and time complexities of the generalized 191-bit binary Karatsuba-Ofman multiplier as,

$$\begin{aligned} \text{Number of CLBs} &= 2MUL_X(128) + MUL_X(64) + C \\ &= 2 \cdot 3379 + 1171 + C \\ &= 7929 + C \\ \text{Delay} &= MUL_{delay}(2^{\lfloor \log_2 m \rfloor}) + O \\ &= MUL_{delay}(2^{\lfloor \log_2 191 \rfloor}) + O \\ &= MUL_{delay}(2^7) + O \end{aligned} \tag{6.15}$$

Where C and O represent the overhead in space and time, respectively, associated with the extra circuitry shown in Figure 6.2.

The generalized 191-bit binary Karatsuba-Ofman multiplier was implemented using Xilinx Foundation Series F4.1i software on Xilinx Virtex-E FPGA device XCV2600e-8bg560. The design is coded using VHDL, using library components and also by using Xilinx Coregenerator for design entry. The implementation occupied a total of 8721 slices and 576 I/O Blocks. We obtained a total path delay of 43 ηSec.

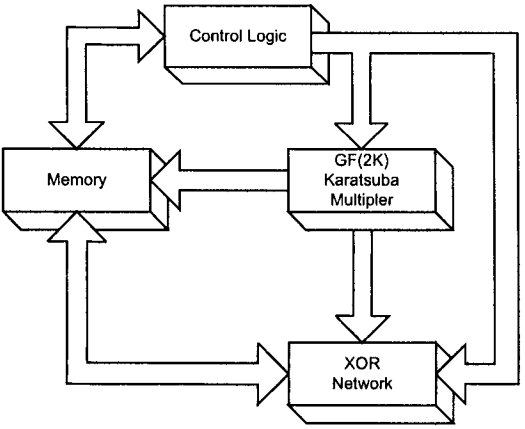

Fig. 6.3. Programmable Binary Karatsuba-Ofman Multiplier

Programmability

The schematic diagram shown in figure 6.2 illustrates two desirable characteristics of the binary Karatsuba-Ofman multipliers. First, it is possible to implement them using non-recursive architectures. In addition, since these algorithms are highly modular, it is possible to design non-parallel scalable implementations. By scalable implementations we mean configurations that allow the user to select the size m of the multiplicands that he/she wants to work with.

Consider the architecture shown in figure 6.3. We use a control logic block that allows us to execute the algorithm of figure 6.2 in a sequential manner. To do this, we also take advantage of the intrinsically modular nature of a 2^k-bit Karatsuba-Ofman multiplier, which can itself be programmed to compute multiplications that involve operands of a size that is any power of two lower than 2^k.

The partial multiplications obtained using this approach, are stored in a memory block as figure 6.3 shows. The control logic can then use these partial results to compute the remaining operations so that the total polynomial product can be obtained. Notice also, that the architecture shown in figure 6.3 can be programmed to implement multiplications with different operands' sizes.

6.1.3 Squaring

In this section we investigate some efficient methods to compute polynomial squaring, which is a special case of polynomial multiplication. Let us assume

that we have an element A given as $A = \sum_{i=0}^{m-1} a_i x^i$. Then the square of A is given as

$$C(x) = A(x)A(x) = A^2(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} a_i x^i) = \sum_{i=0}^{m-1} a_i x^{2i}. \quad (6.16)$$

The main implication of the above equation is that the first $k < m$ bits of A completely determine the first $2k$ bits of A^2. Notice also that half the bits of A^2 (the odd ones) are zeroes. Taking advantage of this feature, the hardware implementation shown in Figure 6.4 simply interleaves a zero value between each one of the original bits of A yielding the required squaring computation which must be followed by a reduction operation to be discussed in the next Subsection.

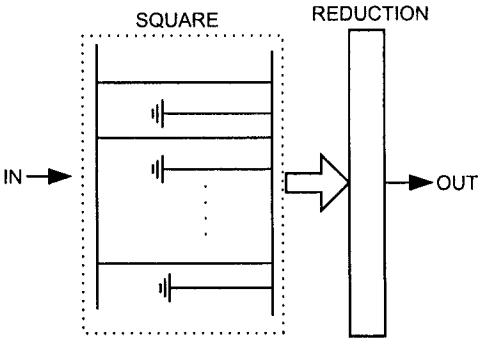

Fig. 6.4. Squaring Circuit

6.1.4 Reduction

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ and let $A(x), B(x) \in GF(2^m)$. Assuming that we already have computed the product polynomial $C(x)$ of Equation (6.1), by using any one of the methods described in the previous two subsections, we want to obtain the modular product C' of Equation (6.2). Recall that the polynomial product C and the modular product C', have $2m-1$ and m, coordinates, respectively, i.e.,

$$\begin{aligned} C &= [c_{2m-2}, c_{2m-3}, \ldots, c_{m+1}, c_m, \ldots, c_1, c_0]; \\ C' &= [c'_{m-1}, c'_{m-2}, \ldots, c'_1, c'_0]. \end{aligned} \quad (6.17)$$

Once the generating polynomial $P(x)$ has been selected, the reduction step that obtains C' from C can be computed by using XOR and shift operations only.

Reduction with Trinomials and Pentanomials

Let the field $GF(2^m)$ be constructed using the irreducible trinomial $P(x) = x^m + x^n + 1$ with a root α and $1 < n < \frac{m}{2}$. Let also $A(x), B(x)$ be elements in $GF(2^m)$. In order to obtain the modular product $C'(x)$ of (6.1), we use the property $P(\alpha) = 0$, and write

$$\begin{aligned} \alpha^m &= 1 + \alpha^n \ ; \\ \alpha^{m+1} &= \alpha + \alpha^{n+1} \ ; \\ &\vdots \\ \alpha^{2m-3} &= \alpha^{m-3} + \alpha^{m+n-3} \ ; \\ \alpha^{2m-2} &= \alpha^{m-2} + \alpha^{m+n-2} \ . \end{aligned} \tag{6.18}$$

The above $m-1$ set of identities suggests a method to obtain the m-coordinates of the modular product C' of Equation (6.2). We can partially reduce the $2m-1$ coordinates of C by reducing its most significant $m-1$ bits into its first $m+n-1$ bits, as indicated by (6.18). For instance, in order to obtain the first partially reduced coordinate c'_0 we just need to add the regular product coordinate c_m to the c_0 coordinate, yielding c'_0 as $c'_0 = c_0 + c_m$.

Similarly the whole set of $m+n-2$ partially reduced coordinates can be found as,

$$\begin{aligned} c'_0 &= c_0 + c_m \ ; \\ c'_1 &= c_1 + c_{m+1} \ ; \\ &\vdots \\ c'_{n-1} &= c_{n-1} + c_{m+n-1} \ ; \\ c'_n &= c_n + c_{m+n} + c_m \ ; \\ c'_{n+1} &= c_{n+1} + c_{m+n+1} + c_{m+1} \ ; \\ &\vdots \\ c'_{m-2} &= c_{m-2} + c_{2m-2} + c_{2m-n-2} \ ; \\ c'_{m-1} &= c_{m-1} + c_{2m-n-1} \ ; \\ c'_m &= c_m + c_{2m-n} \ ; \\ &\vdots \\ c'_{m+n-3} &= c_{m+n-3} + c_{2m-3} \ ; \\ c'_{m+n-2} &= c_{m+n-2} + c_{2m-2} \ . \end{aligned} \tag{6.19}$$

Notice that in the reduction process of (6.19), the constant coefficient of the irreducible generating trinomial $P(x)$ reflects its influence in the first $m-1$ partially reduced bits. The middle term of $P(x)$, on the other hand, affects the partially reduced bits of (6.19) in the range $[c'_n, c'_{m+n-2}]$.

We say that the coefficients in (6.19) have been partially reduced because in general, if $n > 1$, we still need to reduce the $n-1$ most significant reduced coordinates of (6.19). However, this same idea can be used repeatedly until the $m-1$ modular coordinates of (6.17) are obtained. Each time that this strategy is applied we reduce $m-n$ coordinates.

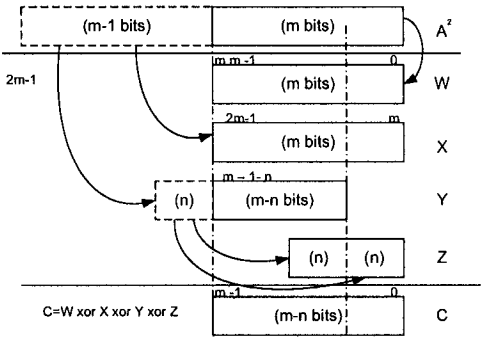

Fig. 6.5. Reduction Scheme

For hardware reconfigurable designs, we can implement above ideas as follows. According to Eq. (6.1) the polynomial product $C(x) = A(x)B(x)$, can be represented as a $2m$-coefficient vector as,

$$C(x) = \left[c'_{2m-1}\ c'_{m-2} \cdots c'_{m-1}\ c'_m\ ;\ c'_{m-1}\ c'_2 \cdots c'_1\ c'_0\right] \tag{6.20}$$

When working with an irreducible trinomial of the form $P(x) = x^m + x^n + 1$, it is convenient to consider the following four sub-vectors,

$$\begin{aligned} C' &= A \cdot B \bmod P(x) \\ &= C'_{[0,m-1]} + C'_{[m,2m-1]} + C'_{[m,2m-1-n]}x^n \\ &\quad + \left(C'_{[2m-n,2m-1]} + C'_{[2m-n,2m-1]}x^n\right) \end{aligned} \tag{6.21}$$

Thus, the reduction step can be computed by the addition of four terms,

$$\begin{aligned} W &= C'_{[0,m-1]} \\ X &= C'_{[m,2m-1]} \\ Y &= C'_{[m,2m-1-n]}x^n \\ Z &= C'_{[2m-n,2m-1]} + C'_{[2m-n,2m-1]}x^n \end{aligned}$$

This procedure is shown schematically in Fig. 6.5. Notice that for those designs implemented in hardware platforms, the modular reduction procedure just

outlined can be instrumented by using XOR logic gates only. Nevertheless, the exact computational complexity of this arithmetic operation depends on the explicit form of m and the middle coefficient n in the trinomial $P(x)$.

Although the strategy shown in Figure 6.5 has been designed for irreducible trinomials, it can be easily extended to irreducible pentanomials. For example, let us consider the finite field GF(2^{163})generated using the irreducible pentanomial $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$ [4]. The corresponding reduction procedure for this pentanomial is depicted in Fig. 6.6.

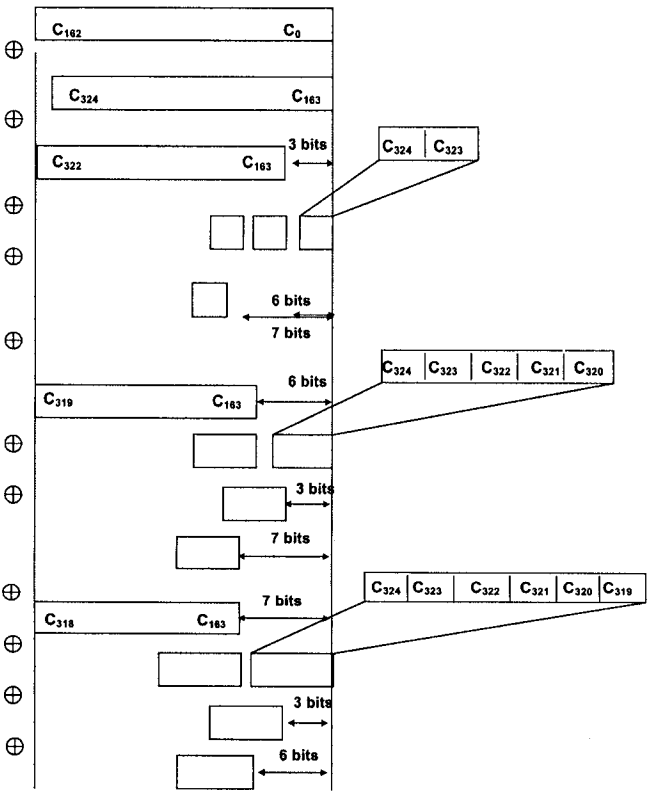

Fig. 6.6. Pentanomial Reduction

[4] This is a NIST recommended finite field for elliptic curve applications [253].

6.1.5 Modular Reduction with General Polynomials

The algorithms studied in the previous section are highly efficient for irreducible trinomials and/or pentanomials. However, when general irreducible polynomials are selected, i.e., irreducible polynomials with an arbitrary number of nonzero coefficients, the algorithms presented in last section are not efficient anymore. Because of that, we need to come out with alternative techniques to handle the reduction step. In this section we present a standard reduction method based in look-up tables specifically intended for general irreducible polynomials.

Recall that assuming that the polynomial product C with $2m-1$ coordinates is given, we would like to compute

$$C'(x) = C(x) \bmod P(x).$$

Our methods are based on the observation that since we are interested in the polynomial remainder of the above equation, we can safely add any multiple of $P(x)$ to $C(x)$ without altering the desired result. This simple observation suggests the following algorithm that can reduce k bits of the polynomial product C at once. Assume that the $m+1$ and $2m-1$ coordinates of P(x) and C(x), respectively, are distributed as follows:

$$\begin{array}{l} C = [c_{2m-2}, c_{2m-3}, \ldots, c_{2m-1-k}, c_{2m-2-k}, \ldots, c_1, c_0]; \\ P = [p_m, p_{m-1}, \ldots, p_1, p_0]. \end{array} \tag{6.22}$$

Then, there always exists a k-bit constant scalar S, such that

$$\begin{array}{rl} P = [& p_m,\ p_{m-1}, \ldots,\ p_{m-k+1}, p_{m-k}, \ldots, p_1, p_0]; \\ S \cdot P = [& c_{2m-2}, c_{2m-3}, \ldots, c_{2m-1-k}, p'_{m-k}, \ldots, p'_1, p'_0]; \end{array} \tag{6.23}$$

where $1 \leq k \leq m-1$. Notice that all the most significant k bits of the scalar multiplication $S \cdot P$ become identical to the corresponding ones of the number C. By left shifting the number $S \cdot P$ exactly $Shift = 2m-2-k-1$ positions, we can effectively reduce the number in C by k bits as shown in figure 6.7.

$$\begin{array}{rl} C & [c_{2m-2}, c_{2m-3}, \ldots, c_{2m-1-k}, c_{2m-2-k}, \ldots, c_{m-2}, c_{m-3}, \ldots, c_0] + \\ 2^{Shift}(S \cdot P) & [c_{2m-2}, c_{2m-3}, \ldots, c_{2m-1-k}, p'_{m-k}, \ldots, p'_0, 0, \ldots, 0] = \\ \hline & [0, 0, \ldots, 0, c'_{2m-k}, \ldots, c'_{m-2}, c_{m-3}, \ldots, c_0] \end{array}$$

Fig. 6.7. A Method to Reduce k Bits at Once

One can apply this strategy an appropriate number of times in order to reduce all the most $m-1$ significant coordinates of C.

In summary, the main design problems that we need to solve in order to implement the reduction method discussed here are:

- Given C and P as in (6.22), find the appropriate constant S that yields the most significant k bits of the operation SP, identical to the corresponding ones in C.
- Compute the scalar multiplication $S \cdot P$ of (6.23).
- Left shift the number $S \cdot P$ by $Shift$ positions, so that the result of the polynomial addition $C + 2^{Shift}(S \cdot P)$ ends up having k leading zeroes.

Both of the first two design problems, i.e., finding the constant S and computing the scalar product $S \cdot P$, can be solved efficiently by using a look-up table approach, provided that a moderated value of k be selected. In practice, we have found that a selection of $k = 8$ yields a reasonable memory/speed trade-off in the performance of the algorithm.

For all the 2^k different values that the k most significant bits of C can possibly take, we want to guarantee that the k most significant bits of the operation SP are identical to those of C. Hence, once that k has been fixed, we need to find a set of 2^k different scalars satisfying that requirement.

Algorithm 6.3 presents a method that, given the irreducible polynomial P and its degree m and the selected value of k, constructs a table containing all the 2^k scalars needed to obtain the required result.

Algorithm 6.3 Constructing a Look-Up Table that Contains All the 2^k Possible Scalars in Equation (6.23)

Require: The irreducible polynomial P; its degree m; and k, the number of bits to be reduced at once.
Ensure: A table *highdivtable* with 2^k scalars.
1: $highdivtable = 0$;
2: $N = 2^k - 1$;
3: $PMSBk = P_m P_{m-1} \ldots P_{m-k+1}$;
4: **for** i from 0 to N **do**
5: $A = Dec2Bin(i)$;
6: **for** j from 0 to k-1 **do**
7: **if** $A_j = 1$ **then**
8: $A = A + RightShift(PMSBk, j)$;
9: $highdivtable[i] = highdivtable[i] + 2^{k-1-j}$;
10: **end if**
11: **end for**
12: **end for**
13: **Return** (highdivtable)

The Algorithm 6.3 finds all the 2^k scalars needed by *reducing* each one of them using the k most significant bits of the irreducible polynomial P. For convenience, these bits are stored in the variable $PMSBk$ (see step 3 of Algorithm 6.3). Steps 4-9 find the appropriate scalar S for each one of all the 2^k possible values that the k MSB of C can take.

In line 5 the value of C to be processed is translated to its binary representation and stored in the temporary variable A. Then, in lines 6-9 each one of the k bits of A is scanned and reduced, if necessary, by using an appropriate shift version of $PMSBk$. Finally, in line 9 the $k-1-j$-th bit of the i-th entry in table *highdivtable* is set. At the end of the inner loop in lines 6-9, the i-th entry of *highdivtable* contains the scalar S that would obtain the result in (6.23), if the k most significant bits of C where equal to the number in A.

In order to compute the scalar multiplication $S \cdot P$ of (6.23), we use once again a look-up table approach as shown in Algorithm 6.4.

Algorithm 6.4 Generating a Look-Up Table that Contains All the 2^k Possible Scalars Multiplications $S \cdot P$

Require: The irreducible polynomial P; and k, the number of bits to be reduced at once.
Ensure: A table *Paddedtable*, with all the 2^k $S \cdot P$ possible products.
1: **for** i from 0 to k-1 **do**
2: Pshift[i] = LeftShift(P, i);
3: **end for**
4: $N = 2^k - 1$;
5: **for** i from 0 to N **do**
6: $S = Dec2Bin(i)$;
7: **for** j from 0 to k-1 **do**
8: **if** $S_j = 1$ **then**
9: $Paddedtable[i] = Paddedtable[i] + Pshift[k]$;
10: **end if**
11: **end for**
12: **end for**
13: **Return** (Paddedtable)

The algorithm in 6.4 is quite similar to Algorithm 6.3. In order to obtain all the 2^k scalar products of the irreducible polynomial P, the above algorithm finds first in lines 1-2 all the first 2^j multiples of P for $j = 0, 1, \ldots, k-1$. Then, in lines 4-9 all the 2^k scalars S are examined one by one and bit by bit, so that the scalar product $i \cdot P$ is stored in the i-th entry of the table *Paddedtable* for $i = 0, 1, \ldots, N = 2^k - 1$. Notice that each entry of *Paddedtable* has a size of $m + k$ bits, where m is the degree of the irreducible polynomial P.

Using the two look-up tables generated by Algorithms 6.3 and 6.4, we can easily obtain the modular reduction of the polynomial C by repeatedly implementing the operation $C + 2^{Shift}(S \cdot P)$.

Consider now Algorithm 6.5, where it has been assumed that the tables *Highdivtable* and *Paddedtable* have been previously computed and are available.

First, in line 1 given k and the degree m of the irreducible polynomial P, the number of iterations is computed and stored in the variable N_k. In line 2 it

Algorithm 6.5 Modular Reduction Using General Irreducible Polynomials

Require: The degree m of the irreducible polynomial; the operand C to be reduced; and k the number of bits that can be reduced at once.
Ensure: The field polynomial defined as $C = C \bmod P$, with a length of m bits.
1: $N_k = \lceil \frac{m-1}{k} \rceil$;
2: $shift = 2m - 2 - k - 1$;
3: **for** i from 0 to N_k **do**
4: $A = C_{n-k \cdot i} C_{(n-k \cdot i)-1} \ldots C_{(n-k \cdot i)-k+1}$;
5: $S = Highdivtable[A]$;
6: $Pshifted = LeftShift(Paddedtable[S], shift)$;
7: $C = C + Pshifted$;
8: $shift = shift - k$;
9: **end for**
10: **Return** C

is computed the amount of shift needed to apply properly the method outlined in figure 6.7. Then, in each iteration of the loop in lines 3-9, k bits of C are reduced. In line 4 the k bits of C to be reduced are obtained. This information is used in line 5 to compute the appropriate scalar S needed to obtain the result of equation (6.23). In line 6 the S-th entry of the table *Paddedtable* is left shifted $shift$ positions so that in line 7 the operation $C + 2^{Shift}(S \cdot P)$ can be finally computed allowing the effective reduction of k bits at once. Then, in line 8 the variable $shift$ is updated in order to continue the reduction process.

Algorithm 6.5 performs a total of $N_k = \lceil \frac{m-1}{k} \rceil$ iterations. At each iteration of the algorithm the look-up tables *Highdivtable* and *Paddedtable* are accessed once each. In line 7, and XOR addition is executed, implying that the complexity cost of the general reduction method discussed in this section is given as,

$$\begin{aligned} \text{Additions} &= 2N_k, \\ \text{Look-up table size (in bits)} &= 2^k(m + 2k) \ . \end{aligned} \tag{6.24}$$

6.1.6 Interleaving Multiplication

In this Subsection we discuss one of the simplest and most economical binary field multiplier schemes: the serial interleaving multiplication algorithm.

Multiplication by a Primitive Element

Let $P(x) = p_0 + p_1 x + p_1 x^2 + \ldots + p_{m-1} x^{m-1} + x^m$ be an m-degree irreducible polynomial over $GF(2)$. Let also α be a root of $p(x)$, i.e., $p(\alpha) = 0$. Then, the set $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ is a basis for $GF(2^m)$, commonly called the polynomial (canonical) basis of the field [221]. An element $A \in GF(2^m)$ is expressed in this basis as $A = \sum_{i=0}^{m-1} a_i \alpha^i$. Let $A(\alpha)$ be an arbitrary element of $GF(2^m)$.

Then, the product $C = \alpha \cdot A(\alpha)$ can be expressed as,

$$C = \alpha \left(a_0 + a_1\alpha + \ldots + a_{m-1}\alpha^{m-1}\right) = a_0\alpha + a_1\alpha^2 + \ldots + a_{m-1}\alpha^m. \quad (6.25)$$

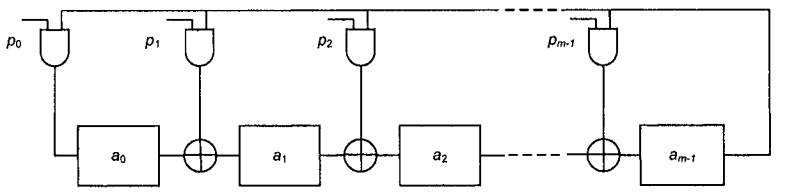

Fig. 6.8. $\alpha \cdot A(\alpha)$ Multiplication

Using the fact that α is a primitive root of the irreducible polynomial, we can write,

$$\alpha^m = p_0 + p_1\alpha + \ldots + p_{m-1}\alpha^{m-1}. \quad (6.26)$$

Substituting Eq. (6.26) into Eq. (6.25) we obtain,

$$C = c_0 + c_1\alpha + \ldots + c_{m-1}\alpha^{m-1},$$

where, $c_0 = a_{m-1}p_0$ and

$$d_i = a_{i-1} + a_{m-1}p_i,$$

for $i = 1, \ldots, m-1$. A realization of the above operation is shown in Fig. 6.8. The main building block is an m-tap LFSR register. That register is initially loaded with the m coordinates of the field element A, namely, $(a_0, a_1, a_2, \ldots, am-1)$. The signals p_i represent the coefficients of the irreducible polynomial. Notice that whenever a given polynomial coefficient is on, i.e., $p_i = 1$, then the corresponding branch of the circuit will be a *short circuit*. Otherwise, if $p_i = 0$ the branch acts as an *open circuit*. After m clock cycles, the new register content will be the value of the field element C.

Serial Multiplication

Using the multiplication procedure outlined above, the multiplication of two arbitrary field elements can be accomplished by using a procedure inspired in the well-know Horner's scheme.

Let us consider two arbitrary field elements A and B expressed in polynomial basis as,

$$A(\alpha) = \sum_{i=0}^{m-1} a_i\alpha^i, B(\alpha) = \sum_{i=0}^{m-1} b_i\alpha^i$$

Then, the product of $A \cdot B$ can be expressed as,

$$\begin{aligned} C(\alpha) &= A(\alpha)B(\alpha) \bmod P(\alpha) \\ &= A(\alpha)\left(\sum_{i=0}^{m-1} b_i\alpha^i\right) \bmod P(\alpha) \\ &= \left(\sum_{i=0}^{m-1} b_i A(\alpha)\alpha^i\right) \bmod P(\alpha) \end{aligned}$$

Therefore,

$$C(\alpha) = \left(b_0A(\alpha) + b_1A(\alpha)\alpha + b_2A(\alpha)\alpha^2 + \ldots + b_{m-1}A(\alpha)^{m-1}\right) \bmod P(\alpha).$$

Algorithm 6.6 shows the standard procedure for computing above equation using Horner's rule.

Algorithm 6.6 LSB-First Serial/Parallel Multiplier

Require: An irreducible polynomial $P(\alpha)$ of degree m, two elements $A, B \in GF(2^m)$.
Ensure: $C(\alpha) = A(\alpha)B(\alpha) \bmod P(\alpha)$.
1: $C = 0$;
2: **for** $i = 0$ to $m - 1$ **do**
3: $C = b_iA + C$;
4: $A = A\alpha^i \bmod P(\alpha)$;
5: **end for**
6: **Return(C)**.

The multiplier realization of Algorithm 6.6 is shown in Fig. 6.9. The architecture shown in Fig. 6.9 consists of two LFSR Register plus extra circuitry. As it was mentioned previously, the signals p_i in the first LFSR block represent the coefficients of the irreducible polynomial, and their values (either ones or zeroes) determine the LFSR structure. Furthermore, a gate array is included in order to compute the multiplication operation as is explained below. Initially the register C is set to zero, whereas the register in the upper part of Fig. 6.9 is loaded with the m coefficients of the field element A. Thereafter, when the clock signal is applied to the registers, the value of $A\alpha$ is generated. Then, B coefficients, namely, $b_0, b_1, b_2, \ldots, b_{m-1}$ are serially introduced in that order, thus generating the values $b_iA\alpha^i$, for $i = 0, 1, \ldots, m-1$, which are accumulated in register C until all the m product coefficients $c_0, c_1, c_2, \ldots, c_{m-1}$ are collected.

6.1.7 Matrix-Vector Multipliers

The GF(2^m) multiplication given by (6.1) can be described in terms of matrix-vector operations. There are mainly two different approaches based on matrix vector operations to compute a field product:

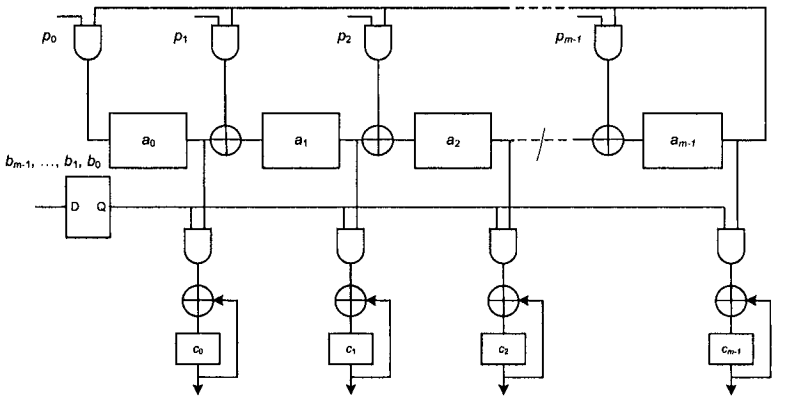

Fig. 6.9. LSB-First Serial/Parallel Multiplier

1. The polynomial multiplication part is performed by any method. Then, the resulting product is reduced by using a reduction matrix.
2. The polynomial multiplication and modular reduction parts are performed in a single step by using the so-called Mastrovito matrix.

Let $a(x)$ and $b(x)$ denote two degree m polynomials representing the elements in GF(2^m). Let $c(x) = a(x)b(x) \bmod P(x)$ denote their field product. The coefficient vectors of these polynomials are given by

$$\begin{aligned}\mathbf{a} &= [a_0, a_1, \cdots, a_{m-1}]^T \\ \mathbf{b} &= [b_0, b_1, \cdots, b_{m-1}]^T \\ \mathbf{c} &= [c_0, c_1, \cdots, c_{m-1}]^T.\end{aligned}$$

Also, let us define the polynomials

$$\begin{aligned}d(x) &= a(x)b(x) = d_0 + d_1 x + \cdots + d_{2m-2}x^{2m-2}\ , \\ d^{(L)}(x) &= d_0 + d_1 x + \cdots + d_{m-1}x^{m-1}\ , \\ d^{(H)}(x) &= d_m + d_{m+1}x + \cdots + d_{2m-2}x^{m-2}\ .\end{aligned} \tag{6.27}$$

The coefficient vectors representing these polynomials are

$$\begin{aligned}\mathbf{d} &= [d_0, d_1, \cdots, d_{2m-2}]^T\ , \\ \mathbf{d}^{(L)} &= [d_0, d_1, \cdots, d_{m-1}]^T\ , \\ \mathbf{d}^{(H)} &= [d_m, d_{m+1}, \cdots, d_{2m-2}]^T\ .\end{aligned}$$

The work in [284] reduces the polynomial multiplication $d(x)$ using an $(m \times m-1)$ reduction matrix $\mathbf{Q}$ to obtain the field product $c(x)$ as below:

$$\mathbf{c} = \mathbf{d}^{(L)} + \mathbf{Q} \cdot \mathbf{d}^{(H)} \ . \tag{6.28}$$

Mastrovito Multiplier

The so-called Mastrovito matrix is constructed from the coefficients of the first multiplicand and the irreducible polynomial defining the field. Then, the polynomial multiplication and modulo reduction steps are performed together using this matrix. The papers [351, 128, 401] follow the Mastrovito multiplication scheme outlined below,

$$\mathbf{c} = \mathbf{M} \cdot \mathbf{b} \ , \tag{6.29}$$

where $\mathbf{M}$ is the $(m \times m)$ Mastrovito matrix whose entries are the function of the coefficients of $a(x)$ and $P(x)$. The Mastrovito matrix $\mathbf{M}$ is related to the reduction matrix $\mathbf{Q}$ by

$$\mathbf{M} = \mathbf{L} + \mathbf{Q} \cdot \mathbf{U} \ , \tag{6.30}$$

where $\mathbf{L}$ and $\mathbf{U}$ are the following $(m \times m)$ and $(m-1 \times m)$ matrices:

$$\mathbf{L} = \begin{bmatrix} a_0 & 0 & 0 & \cdots & 0 & 0 \\ a_1 & a_0 & 0 & \cdots & 0 & 0 \\ a_2 & a_1 & a_0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_0 & 0 \\ a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_1 & a_0 \end{bmatrix} \ ,$$

$$\mathbf{U} = \begin{bmatrix} 0 & a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & \cdots & a_3 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & \cdots & 0 & a_{m-1} \end{bmatrix} \ . \tag{6.31}$$

This is because $d(x) = a(x)b(x)$ can be given in the vector notation by

$$\mathbf{d} = \begin{bmatrix} \mathbf{d}^{(L)} \\ \mathbf{d}^{(H)} \end{bmatrix} = \begin{bmatrix} \mathbf{L} \cdot \mathbf{b} \\ \mathbf{U} \cdot \mathbf{b} \end{bmatrix} \ .$$

Then, $\mathbf{c} = \mathbf{d}^{(L)} + \mathbf{Q} \cdot \mathbf{d}^{(H)} = \mathbf{L} \cdot \mathbf{b} + \mathbf{Q} \cdot \mathbf{U} \cdot \mathbf{b} = (\mathbf{L} + \mathbf{Q} \cdot \mathbf{U}) \cdot \mathbf{b} = \mathbf{M} \cdot \mathbf{b}$.

The Mastrovito and the reduction matrices are studied thoroughly in [284, 401] for various types of irreducible polynomials. In [351] a comprehensive study of the Mastrovito multiplier for irreducible trinomials was presented. Authors in [401] proposed a practical and systematic design approach for a general Mastrovito multiplier. In [388] it was shown that non-Mastrovito multipliers using direct modular reduction also provide competitive performance. Moreover, efficient non-Mastrovito multipliers for irreducible trinomials were also proposed.

6.1.8 Montgomery Multiplier

In this section we explain the Montgomery multiplication method in GF(2^m). Once again, let $P(x)$ be an irreducible polynomial over $GF(2)$ that defines the field GF(2^m). Rather than computing Eq.(6.1), the Montgomery multiplication calculates

$$C(x) = A(x)B(x)R^{-1}(x) \mod P(x) \tag{6.32}$$

where $R(x)$ is a fixed element and $gcd(R(x), P(x)) = 1$.

Because of Bezout's identity[5], one can find two polynomials $R^{-1}(x)$ and $P'(x)$ such that

$$R(x)R^{-1}(x) + P(x)P'(x) = 1 \tag{6.33}$$

where $R^{-1}(x)$ is the inverse of $R(x)$ modulo $P(x)$. These two polynomials can be calculated with the extended Euclidean algorithm. Koç and Acar [182, 388] selected $R(x) = x^m$ for high performance modular reduction in the Montgomery multiplication algorithm, which can be given as follows:

Algorithm 6.7 Montgomery Modular Multiplication Algorithm

Require: $A(x), B(x), R(x), P'(x)$
Ensure: $C(x) = A(x)B(x)R^{-1}(x) \mod P(x)$
1: $T(x) = A(x)B(x)$;
2: $U(x) = T(x)\ P'(x) \mod R(x)$;
3: $C(x) = [T(x) + U(x)P(x)]/R(x)$;
4: **Return** C

To prove the correctness of this algorithm we note that Step 2 implies that there exists a polynomial

$$U(x) = T(x)\ P'(x) + H(x)R(x)\ . \tag{6.34}$$

We write $C(x)$ in Step 3 by using (6.34) as follows:

$$\begin{aligned} C(x) &= \tfrac{1}{R(x)}[T(x) + T(x)\ P'(x)\ P(x) + H(x)R(x)\ P(x)] \\ &= \tfrac{1}{R(x)}[T(x)(1 + P'(x)\ P(x)) + H(x)R(x)\ P(x)]\ . \end{aligned}$$

From (6.33), we can write $1 + P(x)P'(x) = R(x)R^{-1}(x)$ and substitute it into our last expression

$$\begin{aligned} C(x) &= \tfrac{1}{R(x)}[T(x)R(x)R^{-1}(x) + H(x)R(x)\ P(x)] \\ &= T(x)R^{-1}(x) + H(x)\ P(x) \\ &= A(x)B(x)R^{-1} \mod P(x)\ . \end{aligned}$$

[5] For more details on Bezout's identity the reader is refer to §6.3.1.

The degree of $C(x)$ can be verified from Step 3 as follows:

$$\begin{aligned} deg[C(x)] &\leq max\{deg[T(x)], deg[U(x)] + deg[P(x)]\} - deg[R(x)] \\ &\leq max\{2m-2, deg[R(x)] - 1 + m\} - deg[R(x)] \\ &\leq max\{2m-2-deg[R(x)], m-1\} \ . \end{aligned}$$

Then, it can be concluded that $deg[C(x)] \leq m-1$, if $deg[R(x)] \geq m-1$. If we choose $R(x) = x^m$, the result $C(x)$ will be of degree $m-1$ at most.

It can be shown [182] that Algorithm 6.7 has an associated computational cost of $2m^2$ coefficient multiplications (ANDs) and $2m^2 - 3m - 1$ coefficient additions (XORs), whereas the total time complexity is $3T_A + (2\lceil \log_2 m \rceil + \lceil \log_2 (m-1) \rceil) T_X$.

6.1.9 A Comparison of Field Multiplier Designs

Table 6.3. Fastest Reconfigurable Hardware $GF(2^m)$ Multipliers

Work	Platform	Field	Cost	Cycles	$timings$	$\frac{bits}{Slices \times timings}$
KOM variant by [47], implemented by [326]	Virtex 2	$GF(2^{163})$	5307 CLBs	1	$12.56\eta S$	2.445M
KOM variant by [85], implemented by [326]	Virtex 2	$GF(2^{163})$	5409 CLBs	1	$13.37\eta S$	2.254M
KOM variant by [293], implemented by [326]	Virtex 2	$GF(2^{163})$	5840 CLBs	1	$14.73\eta S$	1.895M
KOM [106]	Virtex 2	240 bits	1480 CLBs	30	$378\eta S$	0.429M
Recursive Classical [106]	Virtex 2	240 bits	1582 CLBs	56	$523\eta S$	0.290M
KOM [117]	Virtex 2	240 bits	1660 CLBs	54	$655\eta S$	0.221M
Massey-Omura [118]	Virtex 2	240 bits	36857 LUTs	50	$800\eta S$	0.0336M (est.)

In this Subsection we compare some of the most representative designs of $GF(2^m)$ multipliers considering three metrics: speed, compactness and efficiency. Table 6.3 shows the fastest designs reported to date for $GF(2^m)$ field multiplication. It can be observed that Karatsuba-ofman Multipliers (KOM) are much faster than other schemes such as recursive classical multiplier or Massey-Omura scheme. This can be explained from the theoretical point of view from the fact that KOM algorithms enjoy of a sub-quadratic complexity.

In Table 6.4 we show a selection of some of the most compact reconfigurable hardware multiplier designs. It is noted that this category is dominated by the interleaved and Montgomery multiplier schemes.

Table 6.4. Most Compact Reconfigurable Hardware $GF(2^m)$ Multipliers

Work	Platform	Field	Cost	Cycles	$timings$	$\frac{bits}{Slices \times timings}$
Interleaved [104]	Virtex	$GF(2^{239})$	359 CLBs	239	3.1μS	0.215M
Montgomery [97]	Virtex	$GF(2^{233})$	425 CLBs (est)	466	2.81μS	0.195M
Class.+Montg. [18]	Virtex	$GF(2^{160})$	1049 CLBs	80	1.11μS	0.137M
Montgomery [18]	Virtex	$GF(2^{160})$	1427 CLBs	160	1.66μS	0.0675M
Interleaved [266]	Virtex	$GF(2^{210})$	420 CLBs (est)	210	12.3μS	0.042M

We measure efficiency by taking the ratio of number of bits processed over slices multiplied by the time delay achieved by the design, namely,

$$\frac{bits}{Slices \times timings}$$

For instance, consider the KOM variant design proposed by [47] and implemented by [326]. As is shown in Table 6.3, working over $GF(2^{163})$, that design achieved a time delay of just 12.56ηS at a cost of 5307 slices. Therefore its efficiency is calculated as,

$$\frac{bits}{Slices \times timings} = \frac{163}{5307 \times 12.56\eta} = 2.445M$$

When comparing the designs featured in Tables 6.3 and 6.4, it is noticed that the most efficient multiplier designs are the Karatsuba-Ofman multipliers variants as they were reported in [47, 85, 293]. This is a quite remarkable feature, which implies that the Karatsuba-Ofman multipliers represent both, the fastest and the most efficient of all multiplier designs studied in this Chapter.

6.2 Field Squaring and Field Square Root for Irreducible Trinomials

Let us consider binary extension fields constructed using irreducible trinomials of the form $P(x) = x^m + x^n + 1$, with $m \geq 2$. It is convenient to consider, without loss of generality, the additional restriction $1 \leq n \leq \lfloor \frac{m}{2} \rfloor$ [6].

[6] It is known that if $P(x) = x^m + x^n + 1$ is irreducible over $GF(2)$, so is $P(x) = x^m + x^{m-n} + 1$[228]. Hence, provided that at least one irreducible trinomial of degree m exists, it is always possible to find another irreducible trinomial such that its middle coefficient n satisfies the restriction $1 \leq n \leq \lfloor \frac{m}{2} \rfloor$.

The rest of this Section is organized as follows. First, in Subsection 6.2.1, we give the corresponding formulae needed for computing the field squaring operation when considering arbitrary irreducible trinomials. Those equations are then used in Subsection 6.2.2 to find the corresponding ones for the field square root operator.

6.2.1 Field Squaring Computation

Let $A = \sum_{i=0}^{m-1} a_i x^i$ be an arbitrary element of $GF(2^m)$. Then, according to Eq. (6.16) its square, A^2, can be represented by the $2m$-coefficient vector,

$$\begin{aligned} A^2(x) &= \left[0\ a_{m-1}\ 0\ a_{m-2} \cdots\ 0\ a_1\ 0\ a_0\right] \\ &= \left[a'_{2m-1}\ a'_{m-2} \cdots\ a'_{m-1}\ a'_m\ ;\ a'_{m-1}\ a'_2 \cdots\ a'_1\ a'_0\right] \end{aligned} \tag{6.35}$$

where $a'_i = 0$ for i odd. Hence, the upper half of A^2 (i.e., the m most significant bits) in Eq. (6.35) is mapped into the first m coordinates by performing addition and shift operations only.

In order to investigate the exact cost of the field squaring operation, we categorize all the irreducible trinomials over $GF(2)$ into four different types. For all four types considered and by means of Eqs. (6.35) and (6.21), the following explicit formulae for the field squaring operation were found,

Type I: Computing $C = A^2 \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m even, n odd and $n < \frac{m}{2}$,

$$c_i = \begin{cases} a_{\frac{i}{2}} + a_{\frac{m+i}{2}} & i \text{ even}, i < n \text{ or } i \geq 2n, \\ a_{\frac{i}{2}} + a_{\frac{m+i}{2}} + a_{m-n+\frac{i}{2}} & i \text{ even}, n < i < 2n, \\ a_{m+1-\frac{n+i}{2}} & i \text{ odd}, i < n, \\ a_{\frac{m-n+i}{2}} & i \text{ odd}, i \geq n, \end{cases} \tag{6.36}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.36) has an associated cost of $\frac{m+n-1}{2}$ XOR gates and $2T_x$ delays.

Type II: Computing $C = A^2 \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m even, n odd and $n = \frac{m}{2}$,

$$c_i = \begin{cases} a_{\frac{i}{2}} + a_{\frac{m+i}{2}} & i \text{ even}, i < n, \\ a_{\frac{i}{2}} & i \text{ even}, i > n, \\ a_{m+1-\frac{n+i}{2}} & i \text{ odd}, i < n, \\ a_{\frac{n+i}{2}} & i \text{ odd}, i \geq n, \end{cases} \tag{6.37}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.37) has an associated cost of $\frac{m+2}{4}$ XOR gates and one T_x delay.

Type III: Computing $C = A^2 \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m, n odd numbers and $n < \frac{m+1}{2}$,

$$c_i = \begin{cases} a_{\frac{i}{2}} & i \text{ even}, i < n, \\ a_{\frac{i}{2}} + a_{\frac{i}{2}+\frac{m-n}{2}} + a_{\frac{i}{2}+(m-n)} & i \text{ even}, n < i < 2n, \\ a_{\frac{i}{2}} + a_{\frac{i}{2}+\frac{m-n}{2}} & i \text{ even}, i \geq 2n, \\ a_{\frac{m+i}{2}} + a_{\frac{m+i}{2}+\frac{m-n}{2}} & i \text{ odd}, i < n, \\ a_{\frac{m+i}{2}} & i \text{ odd}, i \geq n, \end{cases} \tag{6.38}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.38) has an associated cost of $\frac{m-1}{2}$ XOR gates and $2T_x$ delays.

Type IV: Computing $C = A^2 \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m odd, n even and $n < \frac{m+1}{2}$,

$$c_i = \begin{cases} a_{\frac{i}{2}} + a_{\frac{i}{2}+m-\frac{n}{2}} & i \text{ even}, i < n, \\ a_{\frac{i}{2}} + a_{\frac{i}{2}+m-n} & i \text{ even}, n \leq i < 2n, \\ a_{\frac{i}{2}} & i \text{ even}, i \geq 2n, \\ a_{\frac{m+i}{2}} & i \text{ odd}, i < n, \\ a_{\frac{m+i}{2}} + a_{\frac{m+i}{2}-\frac{n}{2}} & i \text{ odd}, i > n, \end{cases} \tag{6.39}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.39) has an associated cost of $\frac{m+n-1}{2}$ XOR gates and one T_x delay.

The complexity costs found on Equations (6.36) through (6.39) are in consonance with the ones analytically derived in [386, 387].

6.2.2 Field Square Root Computation

In the following, we keep the assumption that the middle coefficient n of the generating trinomial $P(x) = x^m + x^n + 1$ satisfies the restriction $1 \leq n \leq \frac{m}{2}$.

Clearly, Eqs. (6.36)-(6.39) are a consequence of the fact that in binary extension fields, squaring is a linear operation. The linear nature of binary extension field squaring, allow us to describe this operator in terms of an $(m \times m)$-matrix as,

$$C = A^2 = MA \tag{6.40}$$

Furthermore, based on Eq. (6.40), it follows that computing the square root of an arbitrary field element A means finding a field element $D = \sqrt{A}$ such that $D^2 = MD = A$. Hence,

$$D = M^{-1}A \tag{6.41}$$

Eq. (6.41) is especially attractive for fields $GF(2^m)$ with order sufficiently large, i.e., $m >> 2$, where the matrixes M corresponding to Eqs. (6.36)-(6.39) are all highly spare (each row has at most three nonzero values).

Hence, for the trinomial types I, II, III and IV as described above, the element $D = \sqrt{A}$ given by Eq. (6.41) can be found by the computation of the inverse of the corresponding matrix M. Then using $\sqrt{A} = D = M^{-1}A$, we can determine the m coordinates of the field element as described bellow.

Type I: Computing D such that $D^2 = A \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m even, n odd, and $n < \frac{m}{2}$:

$$d_i = \begin{cases} a_{2i} + a_{2i+n} & i \leq \lfloor \frac{n}{2} \rfloor, \\ a_{2i} + a_{(2i+n) \bmod m} + a_{2i-n} & \lfloor \frac{n}{2} \rfloor < i < n, \\ a_{2i} + a_{(2i+n) \bmod m} & n \leq i < \frac{m}{2}, \\ a_{(2i+n) \bmod m} & \frac{m}{2} \leq i < m \end{cases} \tag{6.42}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.42) has an associated cost of $\frac{m+n-1}{2}$ XOR gates and $2T_x$ delays.

Type II: Computing D such that $D^2 = A \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m even, n odd and $n = \frac{m}{2}$:

$$d_i = \begin{cases} a_{2i} + a_{2i+\frac{m}{2}} & i < \frac{m+2}{4}, \\ a_{2i} & \frac{m+2}{4} \leq i < \frac{m}{2} \\ a_{(2i+\frac{m}{2}) \bmod m} & \frac{m}{2} \leq i < m \end{cases} \tag{6.43}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.43) has an associated cost of $\frac{m+2}{4}$ XOR gates and one T_x delay.

Type III: Computing D such that $D^2 = A \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m, n odd numbers and $n < \frac{m+1}{2}$,

$$d_i = \begin{cases} a_{2i} & i < \frac{n+1}{2}, \\ a_{2i} + a_{2i-n} & \frac{n+1}{2} \leq i < \frac{m+1}{2}, \\ a_{2i-n} + a_{2i-m} & \frac{m+1}{2} \leq i < \frac{m+n}{2}, \\ a_{2i-m} & \frac{m+n}{2} \leq i < m \end{cases} \tag{6.44}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.44) has an associated cost of $\frac{m-1}{2}$ XOR gates and one T_x delay.

Type IV: Computing D such that $D^2 = A \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m, odd, n even and $\lceil \frac{m-1}{4} \rceil \leq n < \lfloor \frac{m-1}{3} \rfloor$.

$$d_i = \begin{cases} a_{2i} + a_{2i+(m-n)} + a_{2i+(m-2n)} + a_{2i+(m-3n)} & i < \frac{4n-(m-1)}{2}, \\ a_{2i} + a_{2i+(m-n)} + a_{2i+(m-2n)} + a_{2i+(m-3n)} & \\ +a_{2i+(m-4n)} & \frac{4n-(m-1)}{2} \le i < \frac{n}{2}, \\ a_{2i} + a_{2i+(m-2n)} + a_{2i+(m-3n)} + a_{2i+(m-4n)} & \frac{n}{2} \le i < \frac{5n-(m-1)}{2}, \\ a_{2i} + a_{2i+(m-2n)} + a_{2i+(m-3n)} + a_{2i+(m-4n)} & \\ +a_{2i+(m-5n)} & \frac{5n-(m-1)}{2} \le i < n, \\ a_{2i} & n \le i \le \frac{m-1}{2}, \\ a_{2i-m} & \frac{m+1}{2} \le i < \frac{n+m+1}{2}, \\ a_{2i-m} + a_{2i-(m+n)} & \frac{n+m+1}{2} \le i < \frac{2n+m+1}{2} \\ a_{2i-m} + a_{2i-(m+n)} + a_{2i-(m+2n)} & \frac{2n+m+1}{2} \le i < \frac{3n+m+1}{2} \\ a_{2i-m} + a_{2i-(m+n)} + a_{2i-(m+2n)} + a_{2i-(m+3n)} & \frac{3n+m+1}{2} \le i < m \end{cases} \tag{6.45}$$

for $i = 0, 1, \cdots, m-1$. At first glance, Eq. (6.45) can be implemented with an XOR gate cost of,

$$3 \cdot \frac{4n-(m-1)}{2} + 4 \cdot \frac{m-3n-1}{2} + 3 \cdot \frac{4n-(m-1)}{2} + 4 \cdot \frac{m-3n-1}{2} + \frac{n}{2} + 2 \cdot \frac{n}{2} + 3 \cdot \frac{m-3n-1}{2} = 5 \cdot \frac{m-n-1}{2} - \frac{n}{2}.$$

However, taking advantage of the high redundancy of the terms involved in Eq. (6.45), it can be shown (after a tedious long derivation) that actually $\frac{m+n-1}{2}$ XOR gates are sufficient to implement it with a $2T_x$ gate delays.

Table 6.5. Summary of Complexity Results

Type	Trinomial $P(x) = x^m + x^n + 1$	Operation	XOR gates	Time delay
I	m even, n odd	Squaring	$(m+n-1)/2$	$2T_x$
II	m even, $n = m/2$	Squaring	$(m+2)/4$	T_x
III	m odd, n odd	Squaring	$(m-1)/2$	$2T_x$
IV	m odd, n even	Squaring	$(m+n-1)/2$	T_x
I	m even, n odd	Square root	$(m+n-1)/2$	$2T_x$
II	m even, $n = m/2$	Square root	$(m+2)/4$	T_x
III	m odd, n odd	Square root	$(m-1)/2$	T_x
IV	m odd, n even	Square root	$(m+n-1)/2$	$2T_x$

Table 6.5 summarizes the area and time complexities just derived for the cases considered. Furthermore, in Table 6.6 we list all preferred irreducible trinomials $P(x) = x^m + x^n + 1$ of degree $m \in [160, 571]$ with m a prime number. In all the instances considered the computational complexity of computing the square root operator is comparable or better than that of the field squaring.

6.2.3 Illustrative Examples

In order to illustrate the approach just outlined, we include in this Section several examples using first the artificially small finite field $GF(2^{15})$ and then more realistic fields, in terms of practical cryptographic applications.

Example 6.1. Field Square Root Computation over $GF(2^{15})$

Let us consider $GF(2^{15})$ generated with the irreducible Type III trinomial $P(x) = x^{15} + x^7 + 1$. As it was discussed before, one can find the square root of any arbitrary field element $A \in GF(2^{15})$ by applying Eq. (6.41). In order to follow this approach, based on Eq. (6.38), we first determine the matrix M of Eq. (6.40) as shown in Table 6.7. Then, the inverse matrix of M modulus two, M^{-1}, is obtained as shown in Table 6.8. Afterwards, the polynomial coefficients, in terms of the coefficients of A, corresponding to the field square $C = A^2$ and the field square root $D = \sqrt{A}$ elements can be found from Eqs. (6.40) and (6.41) as shown in Table 6.9.

As predicted by Eq. (6.38), field squaring can be computed at a cost of $(m-1)/2 = (15-1)/2 = 7$ XOR gates and one T_x delay. In the same way, the square root operation can be computed at a cost of $\frac{(m-1)}{2} = \frac{(15-1)}{2} = 7$ XOR gates with an incurred delay time of one T_x, which matches Eq. (6.44) prediction. It is noticed that in this binary extension field, computing a field square root requires the same computational effort than the one associated to field squaring.

Example 6.2. Field Square Root Computation over $GF(2^{162})$

Let us consider $GF(2^{162})$ generated using the irreducible Type II trinomial, $P(x) = x^{162} + x^{81} + 1$. Using the same approach as for the precedent example,

Table 6.6. Irreducible Trinomials $P(x) = x^m + x^n + 1$ of Degree $m \in [160, 571]$ Encoded as $m(n)$, with m a Prime Number

$m(n)$	Type	$m(n)$	Type	$m(n)$	Type
167(35)	III	281(93)	III	439(49)	III
191(9)	III	313(79)	III	449(167)	III
193(15)	III	337(55)	III	457(61)	III
199(67)	III	353(69)	III	463(93)	III
223(33)	III	359(117)	III	479(105)	III
233(74)	IV	367(21)	III	487(127)	III
239(81)	III	383(135)	III	503(3)	III
241(70)	IV	401(152)	IV	521(158)	IV
257(41)	III	409(87)	III	569(77)	III
263(93)	III	431(120)	IV		
271(70)	IV	433(33)	III		

we can obtain the square root polynomial coefficients of an arbitrary element A from the field $GF(2^{162})$ as,

$$d_i = \begin{cases} a_{2i} + a_{2i+81} & i < 41, \\ a_{2i} & 41 \leq i < 81 \\ a_{(2i+81) \bmod 162} & 81 \leq i \end{cases} \tag{6.46}$$

for $i = 0, 1, \cdots, 161$. As predicted by Eq. (6.43) the associated cost of the field square root computation for this field is given as, $\frac{(m+2)}{4} = \frac{(162+2)}{4} = 41$ XOR gates with an incurred delay time of one T_x.

Example 6.3. Field Square Root Computation over $GF(2^{233})$

Let $GF(2^{233})$ be a field generated with the Type III irreducible trinomial[7], $P(x) = x^{233} + x^{74} + 1$. The square root of any arbitrary field element A is given as,

Table 6.7. Squaring matrix M of Eq. (6.40)

$$M = \begin{bmatrix} 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&1&0&0&0&1&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&1&0&0&0&1&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&1&0&0&0&1&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&1&0&0&0&1&0&0&0 \end{bmatrix}$$

[7] This is a NIST recommended finite field for elliptic curve applications [253].

$$d_i = \begin{cases} a_{2i} + a_{2i+159} + a_{2i+85} + a_{2i+11} & i < 32, \\ a_{2i} + a_{2i+159} + a_{2i+85} + a_{2i+11} + a_{2i-63} & 32 \leq i < 37, \\ a_{2i} + a_{2i+85} + a_{2i+11} + a_{2i-63} & 37 \leq i < 69, \\ a_{2i} + a_{2i+85} + a_{2i+11} + a_{2i-63} + a_{2i-137} & 69 \leq i < 74, \\ a_{2i} & 74 \leq i \leq 116, \\ a_{2i-233} & 116 \leq i < 154, \\ a_{2i-233} + a_{2i-307} & 154 \leq i < 191 \\ a_{2i-233} + a_{2i-307} + a_{2i-381} & 191 \leq i < 228 \\ a_{2i-233} + a_{2i-307} + a_{2i-381} + a_{2i-455} & 228 \leq i < 233 \end{cases} \quad (6.47)$$

for $i = 0, 1, \cdots, 232$. Eq. (6.47) can be implemented with an XOR gate cost of $\frac{m+n-1}{2} = 153$ XOR gates with a $4T_x$ gate delay, which agrees with the value predicted by Eq. (6.45).

6.3 Multiplicative Inverse

Among customary finite field arithmetic operations, namely, addition, subtraction, multiplication and inversion of nonzero elements, the computation of the later is the most time-consuming one. Multiplicative inversion computation of a nonzero element $a \in GF(2^m)$ is defined as the process of finding the unique element $a^{-1} \in GF(2^m)$ such that $a \cdot a^{-1} = 1$.

Several algorithms for computing the multiplicative inverse in $GF(2^m)$ have been proposed in literature [153, 93, 356, 135, 399, 127, 296, 122]. In [135], multiplicative inverse is computed using an improved modification of

Table 6.8. Square Root Matrix M^{-1} of Eq. (6.41)

$$M^{-1} = \begin{bmatrix} 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1 \\ 0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \end{bmatrix}$$

the extended Euclidean algorithm called *almost inverse algorithm*. That iterative algorithm can compute the multiplicative inverse in approximately $2m$ clock cycles [135]. In [127] an architecture able to compute the Montgomery multiplicative inverse for both, $GF(p)$, for a prime p, and $GF(2^m)$ on a unified-field hardware platform was proposed.

Based on Fermat's Little Theorem (FLT) and using an ingenious rearrangement of the required field operations, the *Itoh-Tsujii Multiplicative Inverse Algorithm* (ITMIA) was presented in [153]. Originally, ITMIA was proposed to be applied over binary extension fields with *normal basis* field element representation. Since its publication however, several improvements and variations of it have been reported [93, 356, 399, 122, 296], showing that it can be used with other field element representations too.

Unfortunately enough, cryptographic designers have historically shown some resistance to use FLT-related techniques for computing multiplicative inverses when using polynomial basis representation. This phenomenon is probably due to three frequent misconceptions:

1. Computing multiplicative inverses by using FLT-related techniques is inefficient as those methods require many field multiplication and squaring operations;
2. ITMIA is a competitive design option only when using normal basis representation and;
3. The recursive nature of the ITMIA algorithm makes the parallelization of that algorithm rather difficult if not impossible, forcing the implementation of the ITMIA procedure in a sequential manner.

In the rest of this Section we describe efficient implementations of the binary Euclidean algorithm and the Itoh-Tsujii multiplicative inverse algorithm.

Table 6.9. Square and Square Root Coefficient Vectors

$$C = \begin{bmatrix} a_0 \\ a_8 + a_{12} \\ a_1 \\ a_9 + a_{13} \\ a_2 \\ a_{10} + a_{14} \\ a_3 \\ a_{11} \\ a_4 + a_8 + a_{12} \\ a_{12} \\ a_5 + a_9 + a_{13} \\ a_{13} \\ a_6 + a_{10} + a_{14} \\ a_{14} \\ a_7 + a_{11} \end{bmatrix}, \quad D = \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ a_6 \\ a_1 + a_8 \\ a_3 + a_{10} \\ a_5 + a_{12} \\ a_7 + a_{14} \\ a_1 + a_9 \\ a_3 + a_{11} \\ a_5 + a_{13} \\ a_7 \\ a_9 \\ a_{11} \\ a_{13} \end{bmatrix}$$

In §6.3.1 main implementation details of the binary Euclidean algorithm are explained. Then, S6.3.2 describes how the Itoh-Tsuii algorithm can be utilized for the efficient computation of multiplicative inverses.

6.3.1 Inversion Based on the Extended Euclidean Algorithm

Given two polynomials A and B, not both 0, we say that the greatest common divisor of A and B, is the highest polynomial $D = gcd(A, B)$ that divides both A and B. Based on the property $gcd = (A, B) = gcd(B \pm CA, A)$, the revered Extended Euclidean Algorithhm (EEA)[8] is able to find the unique polynomials G and H that satisfies Bezout's celebrated formula,

$$A \cdot G + B \cdot H = D,$$

where $D = gcd(A, B)$.

Several variations of the EEA have been proposed in the open literature [96, 127, 127, 10]. EEA variants include: the almost inverse algorithm, first proposed in [323], the Binary Euclidean Algorithm (BEA), the Montgomery inverse algorithm, etc. All those algorithms show a computational complexity proportional to the maximum of A and B polynomial degrees.

Algorithm 6.8 shows the binary algorithm as it was reported in [96]. That algorithm takes as inputs the irreducible polynomial P of degree m and the field element A of degree at most $m - 1$. It gives as output the field element A^{-1} such that

$$A \cdot A^{-1} \equiv 1 \bmod P.$$

In steps 4 and 10, the operands U and V are divided by x as many times as possible, respectively. Furthermore, the variables G and H are also divided by x in steps 5-8 and 11-14, respectively. Notice that in case that either G or H are not divisible by x, then an addition with the irreducible polynomial P must be performed first. Eventually, after approximately m iterations, either U or V are equal to 1, which is the condition for exiting the main loop. Either G or H will contain the required multiplicative inverse.

The number of iterations required by Algorithm 6.8 depends on several factors such as design's architecture, target platform and even the exact structure of the irreducible polynomial $P(x)$. Roughly speaking, the number of iterations N can be estimated as $N \approx m$, where m is the size of the finite field.

[8] Euclid's algorithm is proposed in his book *Elements* published 300 B.C. Nevertheless, some scholars are convinced that it was previously known by Aristotle and Eudoxus, some 100 years earlier than Euclid's times. According to Knuth, it can be considered the oldest nontrivial algorithm that has survived to modern era [178].

Algorithm 6.8 Binary Euclidean Algorithm

Require: An irreducible polynomial $P(X)$ of degree m, A polynomial $A \in GF(2^m)$.
Ensure: $A^{-1} \bmod P(x)$.

```
1: U = A; V = P; G = 1; H = 0;
2: while (u ≠ 1 AND v ≠ 1) do
3:    while x divides U do
4:       U = U/x;
5:       if x divides G then
6:          G = G/x;
7:       else
8:          G = (G+P)/x;
9:       end if
10:   end while
11:   while x divides V do
12:      V = V/x;
13:      if x divides G_2 then
14:         H = H/x;
15:      else
16:         H = (H+P)/x;
17:      end if
18:   end while
19:   if (deg(U)>deg(V)) then
20:      U = U + V; G = G + H;
21:   else
22:      V = V + U; H = H + G;
23:   end if
24: end while
25: if U=1 then
26:   Return(G);
27: else
28:   Return(H);
29: end if
```

6.3.2 The IToh-Tsujii Algorithm

In this Section we describe the Itoh-Tsujii Multiplicative Inversion Algorithm (ITMIA). We start deriving a recursive sequence useful for finding multiplicative inverses. Then, we briefly discuss the concept of *addition chains*, which together with the aforementioned recursive sequence yield an efficient version of the original ITMIA procedure.

Since the multiplicative group of the Galois field $GF(2^m)$ is cyclic of order $2^m - 1$, for any nonzero element $a \in GF(2^m)$ we have $a^{-1} = a^{2^m-2}$. Clearly,

$$2^m - 2 = 2(2^{m-1} - 1) = 2\sum_{j=0}^{m-2} 2^j = \sum_{j=1}^{m-1} 2^j.$$

The right-most component of above equalities allow us to express the multiplicative inverse of a in two ways:

$$\left[a^{2^{m-1}-1}\right]^2 = a^{-1} = \prod_{j=1}^{m-1} a^{2^j} \tag{6.48}$$

Let us consider the sequence $\left(\beta_k(a) = a^{2^k-1}\right)_{k\in\mathbb{N}}$. Then, for instance,

$$\beta_0(a) = 1 \quad , \quad \beta_1(a) = a,$$

and from the first equality at (6.48), $[\beta_{m-1}(a)]^2 = a^{-1}$.

It is easy to see that for any two integers $k, j \geq 0$,

$$\beta_{k+j}(a) = \beta_k(a)^{2^j}\beta_j(a). \tag{6.49}$$

Namely,

$$\begin{aligned}\beta_{k+j}(a) = a^{2^{k+j}-1} &= \frac{a^{2^{k+j}}}{a} = \frac{\left(a^{2^k}\right)^{2^j}}{a} \\ &= \left(\frac{a^{2^k}}{a}\right)^{2^j} \frac{a^{2^j}}{a} = \left(a^{2^k-1}\right)^{2^j} a^{2^j-1} \\ &= \beta_k(a)^{2^j}\beta_j(a)\end{aligned}$$

In particular, for $j = k$,

$$\beta_{2k}(a) = \beta_k(a)^{2^k}\beta_k(a) = \beta_k(a)^{2^k+1}. \tag{6.50}$$

Furthermore, we observe that this sequence is periodic of period m:

$$k_2 \equiv k_1 \bmod m \;\Rightarrow\; \beta_{k_2}(a) = \beta_{k_1}(a).$$

To see this, consider $k_2 = k_1 + nm$. Then, by eq. (6.49) and FLT,

$$\beta_{k_2}(a) = \beta_{k_2}(a)^{2^{nm}}\beta_{nm}(a) = \beta_{k_2}(a)^{2^{nm}} \cdot 1 = \beta_{k_2}(a).$$

Therefore, the sequence $(\beta_k(a))_k$ is completely determined by its values corresponding to the indexes $k = 0, \ldots, m-1$.

As a final remark, notice that for any two integers k, j, by eq. (6.49):

$$\beta_k(a) = \beta_{(k-(m-j))+(m-j)}(a) = \beta_{k+j-m}(a)^{2^{m-j}}\beta_{m-j}(a).$$

Since the sequence of β's is periodic, and the rising to the power 2^m coincides with the identity in $GF(2^m)$, we have

$$\beta_k(a) = \beta_{k+j}(a)^{2^{-j}}\beta_{m-j}(a). \tag{6.51}$$

Eq. (6.49) allows the calculation of a "current" $i(= k+j)$-th term as a recursive function of two previous terms, the k-th and the j-th in the sequence.

6.3.3 Addition Chains

Let us say that an *addition chain* for an integer $m-1$ consists of a finite sequence of integers $U = (u_0, u_1, \ldots, u_t)$, and a sequence of integer pairs $V = ((k_1, j_1), \ldots, (k_t, j_t))$ such that $u_0 = 1$, $u_t = m-1$, and whenever $1 \leq i \leq t$, $u_i = u_{k_i} + u_{j_i}$.

Example 6.4. Consider the case $e = m-1 = 193-1 = 192 = (11000000)_2$. Then, a binary addition chain with length $t = 8$ for that e is,

$$\begin{aligned} U &= (\ 1, \quad 2, \quad 4, \quad 8, \quad 16, \quad 32, \quad 64, \quad 128, \quad 192) \\ V &= (\quad (0,0), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (6,7)) \end{aligned}$$

i.e. the associated sequence is governed by the rule, $u_i = u_{i-1} + u_{i-1} = 2u_{i-1}$ for all but the final value which is obtained using $u_t = u_{t-1} + u_{t-2}$.

Another addition chain, also with length $t = 8$, is

$$\begin{aligned} U &= (\ 1, \quad 2, \quad 3, \quad 6, \quad 12, \quad 24, \quad 48, \quad 96, \quad 192) \\ V &= (\quad (0,0), (0,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7)) \end{aligned}$$

i.e. for all $i \neq 2$ the combinatorial rule is $u_i = u_{i-1} + u_{i-1} = 2u_{i-1}$, while $u_2 = u_0 + u_1$. □

The concept of addition chains leads us to a natural way to generalize the Itoh-Tsujii Algorithm, by using an addition chain for $m-1$ and relations (6.48) and (6.49) to compute $a^{-1} = [\beta_{m-1}(a)]^2$.

6.3.4 ITMIA Algorithm

Let a be any arbitrary nonzero element in the field $GF(2^m)$. Let us consider an addition chain U of length t for $m-1$ and its associated sequence V. Then the multiplicative inverse $a^{-1} \in GF(2^m)$ of a can be found by repeatedly applying eq's. (6.49) and/or (6.50). Hence, given $\beta_{u_0}(a) = a^{2^1-1} = a$, for each $u_i, 1 \leq i \leq t$, compute

$$\left[\beta_{u_{i_1}}(a)\right]^{2^{u_{i_2}}} \beta_{u_{i_2}}(a) = \beta_{u_{i_2}+u_{i_1}}(a) = \beta_{u_i}(a) = a^{2^{u_i}-1}$$

A final squaring step yields the required result since,

$$[\beta_{u_t}(a)]^2 = \left(a^{2^{m-1}-1}\right)^2 = (a^{2^m-2}) = a^{-1}.$$

Fig. 6.9 shows an algorithm that iteratively computes all the $\beta_{u_i}(a)$ coefficients in the exact order stipulated by the addition chain U as discussed above.

We assess the computational complexity of the algorithm shown in Fig. 6.9 as follows. The algorithm performs t iterations (where t is the length of the

addition chain U) and one field multiplication per iteration. Thus, we conclude that a total of t field multiplication computations are required. On the other hand, notice that at each iteration i, a total of $2^{u_{i_2}}$ field squarings are performed. Notice also that by definition, the addition chain guarantees that for each $u_i, 1 \leq i \leq t$, the relation $u_{i_2} = u_i - u_{i_1}$ holds. Hence, one can show by induction that the total number of field squaring operations performed right after the execution of the i-th iteration is $u_i - 1$. Therefore, at the end of the final iteration t, a total of $u_t - 1 = m - 2$ squaring operations have been performed. This, together with the final squaring operation, yield a total of $m - 1$ field squaring computations.

Summarizing, the algorithm of Fig. 6.9 can find the multiplicative inverse of any nonzero element of the field using exactly,

$$\begin{aligned} \#Multiplications &= t; \\ \#Squarings &= m - 1. \end{aligned} \tag{6.52}$$

Algorithm 6.9 Itoh-Tsujii Multiplicative Inversion Addition-Chain Algorithm

Require: An irreducible polynomial $P(X)$ of degree m, An element $a \in GF(2^m)$, an addition chain U of length t for $m-1$ and its associated sequence V.
Ensure: $a^{-1} \in GF(2^m)$.
1: $\beta_{u_0}(a) = a$;
2: **for** i from 1 to t **do**
3: $\beta_{u_i}(a) = \left[\beta_{u_{i_1}}(a)\right]^{2^{u_{i_2}}} \cdot \beta_{u_{i_2}}(a) \bmod P(X)$;
4: **end for**
5: **Return**$(\beta^2_{u_t}(a) \bmod P(X))$.

Example 6.5. Let us consider the binary field $GF(2^{193})$ using the irreducible trinomial $P(X) = X^{193} + X^{15} + 1$. Let $a \in GF(2^{193})$ be an arbitrary nonzero field element. Then, using the addition chain of Example 6.4, the algorithm of Fig. 6.9 would compute the sequence of $\beta_{u_i}(a)$ coefficients as shown in Table 6.3.4. Once again, notice that after having computed the coefficient $\beta_{u_8}(a)$, the only remaining step is to obtain a^{-1} which can be achieved as $a^{-1} = \beta^2_{u_8}(a)$. □

6.3.5 Square Root ITMIA

Let a be any arbitrary nonzero element in the field $GF(2^m)$. Let us consider an addition chain U of length t for $m-1$ and its associated sequence V. Then the multiplicative inverse of a, $a^{-1} \in GF(2^m)$, can be found as follows [295].

Given $\gamma_{u_0}(a) = a^{1-2^{-1}} = \sqrt{a}$, for each $u_i, 1 \leq i \leq t$, compute

Table 6.10. $\beta_i(a)$ Coefficient Generation for m-1=192

i	u_i	rule	$\left[\beta_{u_{i_1}}(a)\right]^{2^{u_{i_2}}} \cdot \beta_{u_{i_2}}(a)$	$\beta_{u_i}(a) = a^{2^{u_i}-1}$
0	1	–	–	$\beta_{u_0}(a) = a^{2^1-1}$
1	2	$2u_{i-1}$	$[\beta_{u_0}(a)]^{2^{u_0}} \cdot \beta_{u_0}(a)$	$\beta_{u_1}(a) = a^{2^2-1}$
2	3	$u_{i-1}+u_{i-2}$	$[\beta_{u_1}(a)]^{2^{u_0}} \cdot \beta_{u_0}(a)$	$\beta_{u_2}(a) = a^{2^3-1}$
3	6	$2u_{i-1}$	$[\beta_{u_2}(a)]^{2^{u_2}} \cdot \beta_{u_2}(a)$	$\beta_{u_3}(a) = a^{2^6-1}$
4	12	$2u_{i-1}$	$[\beta_{u_3}(a)]^{2^{u_3}} \cdot \beta_{u_3}(a)$	$\beta_{u_4}(a) = a^{2^{12}-1}$
5	24	$2u_{i-1}$	$[\beta_{u_4}(a)]^{2^{u_4}} \cdot \beta_{u_4}(a)$	$\beta_{u_5}(a) = a^{2^{24}-1}$
6	48	$2u_{i-1}$	$[\beta_{u_5}(a)]^{2^{u_5}} \cdot \beta_{u_5}(a)$	$\beta_{u_6}(a) = a^{2^{48}-1}$
7	96	$2u_{i-1}$	$[\beta_{u_6}(a)]^{2^{u_6}} \cdot \beta_{u_6}(a)$	$\beta_{u_7}(a) = a^{2^{96}-1}$
8	192	$2u_{i-1}$	$[\beta_{u_7}(a)]^{2^{u_7}} \cdot \beta_{u_7}(a)$	$\beta_{u_8}(a) = a^{2^{192}-1}$

$$\left[\gamma_{u_{i_1}(a)}\right]^{2^{-u_{i_2}}} \gamma_{u_{i_2}}(a) = \gamma_{u_{i_2}+u_{i_1}}(a) = \gamma_{u_i}(a) = a^{1-2^{-u_i}}$$

Where $\gamma_{\{u_t=m-1\}} = a^{1-2^{-(m-1)}} = a^{-1}$ gives the required result.

Fig. 6.10 shows an algorithm that iteratively computes all the $\gamma_{u_i}(a)$ coefficients in the exact order stipulated by the addition chain U as discussed above. We assess the computational complexity of the algorithm shown in Fig. 6.10 as follows. The algorithm performs one field multiplication in each of algorithm's t iterations, yielding a total of t field multiplication computations required. Furthermore, at each iteration i, a total of $2^{u_{i_2}}$ field square roots are performed. Since by definition, the addition chain guarantees that for each $u_i, 1 \leq i \leq t$, the relation $u_{i_2} = u_i - u_{i_1}$ holds, one can show that the total number of field square root operations performed right after the execution of the i-th iteration is $u_i - 1$. Therefore, a total of $u_t - 1 = m - 2$ square root operations must be performed. This, together with the initial square root operation, yield a total of $m - 1$ field square root computations.

Summarizing, the algorithm of Fig. 6.10 can find the inverse of any nonzero element of the field using exactly,

$$\begin{aligned} \#\textit{Multiplications} &= t; \\ \#\textit{Square root} &= m-1. \end{aligned} \tag{6.53}$$

Example 6.6. Following with our running example, let us consider the binary field $GF(2^{193})$ generated using the irreducible trinomial $P(X) = X^{193} + X^{15} + 1$. Let $a \in GF(2^{193})$ be an arbitrary nonzero field element. Then, the algorithm of Fig. 6.10 would compute the sequence of $\gamma_{u_i}(a)$ coefficients as shown in Table 6.3.5. The multiplicative inverse is given as $\gamma_{u_8} = a^{-1}$. □

Algorithm 6.10 Square Root Itoh-Tsujii Multiplicative Inversion Algorithm

Require: An irreducible polynomial $P(X)$ of degree m, An element $a \in GF(2^m)$, an addition chain U of length t for $m-1$ and its associated sequence V.
Ensure: $a^{-1} \in GF(2^m)$. **Procedure** SquareRoot_ITMIA($P(X), a, \{U, V\}$) {
1: $\gamma_{u_0}(a) = a^{1-2^{-1}} = \sqrt{a}$;
2: **for** i from 1 to t **do**
3: $\gamma_{u_i}(a) = \left[\gamma_{u_{i_1}}(a)\right]^{2^{-u_{i_2}}} \cdot \gamma_{u_{i_2}}(a) \bmod P(X)$;
4: **end for**
5: **Return**($\gamma_{u_t}(a) \bmod P(X)$)

Table 6.11. $\gamma_i(a)$ Coefficient Generation for m-1=192

i	u_i	rule	$\left[\gamma_{u_{i_1}}(a)\right]^{2^{-u_{i_2}}} \cdot \gamma_{u_{i_2}}(a)$	$\gamma_{u_i}(a) = a^{1-2^{-u_i}}$
0	1	–	–	$\gamma_{u_0}(a) = a^{1-2^{-1}}$
1	2	$2u_{i-1}$	$[\gamma_{u_0}(a)]^{2^{-u_0}} \cdot \gamma_{u_0}(a)$	$\gamma_{u_1}(a) = a^{1-2^{-2}}$
2	3	$u_{i-1} + u_{i-2}$	$[\gamma_{u_1}(a)]^{2^{-u_0}} \cdot \gamma_{u_0}(a)$	$\gamma_{u_2}(a) = a^{1-2^{-3}}$
3	6	$2u_{i-1}$	$[\gamma_{u_2}(a)]^{2^{-u_2}} \cdot \gamma_{u_2}(a)$	$\gamma_{u_3}(a) = a^{1-2^{-6}}$
4	12	$2u_{i-1}$	$[\gamma_{u_3}(a)]^{2^{-u_3}} \cdot \gamma_{u_3}(a)$	$\gamma_{u_4}(a) = a^{1-2^{-12}}$
5	24	$2u_{i-1}$	$[\gamma_{u_4}(a)]^{2^{-u_4}} \cdot \gamma_{u_4}(a)$	$\gamma_{u_5}(a) = a^{1-2^{-24}}$
6	48	$2u_{i-1}$	$[\gamma_{u_5}(a)]^{2^{-u_5}} \cdot \gamma_{u_5}(a)$	$\gamma_{u_6}(a) = a^{1-2^{-48}}$
7	96	$2u_{i-1}$	$[\gamma_{u_6}(a)]^{2^{-u_6}} \cdot \gamma_{u_6}(a)$	$\gamma_{u_7}(a) = a^{1-2^{-96}}$
8	192	$2u_{i-1}$	$[\gamma_{u_7}(a)]^{2^{-u_7}} \cdot \gamma_{u_7}(a)$	$\gamma_{u_8}(a) = a^{1-2^{-192}}$

6.3.6 Extended Euclidean Algorithm versus Itoh-Tsujii Algorithm

In order to assess the performance differences between multiplicative inverse computation via the Extended Euclidean Algorithm and the Itoh-Tsujii Algorithm, we performed the following experiment.

Using a Virtex 2 xc2v4000-6bf957 as a target device, we implemented Algorithms 6.8 and 6.9 for computing multiplicative inverses in the field $GF(2^m)$ generated using the irreducible trinomial $P(x) = x^{193} + x^{15} + 1$. Algorithm 6.8 was implemented according to the finite-state machine shown in Fig. 6.10, whereas the Itoh-Tsujii Algorithm was implemented using the architecture shown in Fig. 6.11. The implementation statistics obtained for each algorithm are summarized in Table 6.12.

According to Table 6.12, it can be observed that the BEA scheme represents a cheaper solution in terms of hardware resource requirements. Indeed, the BEA scheme utilizes just 12.02% of the area required by the ITMIA design. On the contrary, the ITMIA scheme outperforms the BEA scheme in timing performance, with a speedup of about 3.3 times. Therefore, consider-

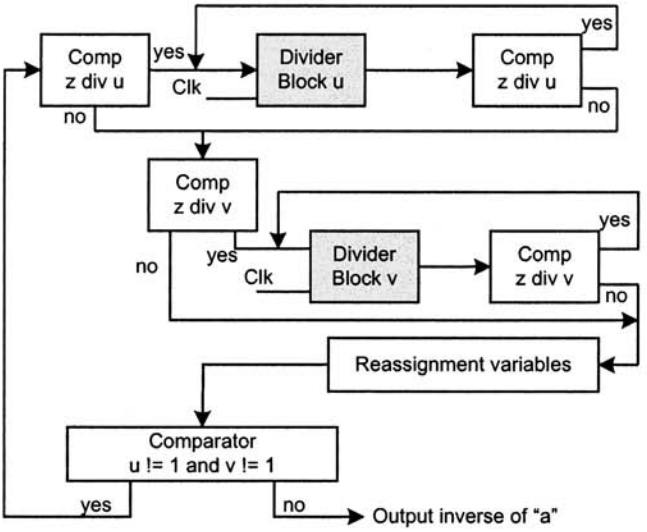

Fig. 6.10. Finite State Machine for the Binary Euclidean Algorithm

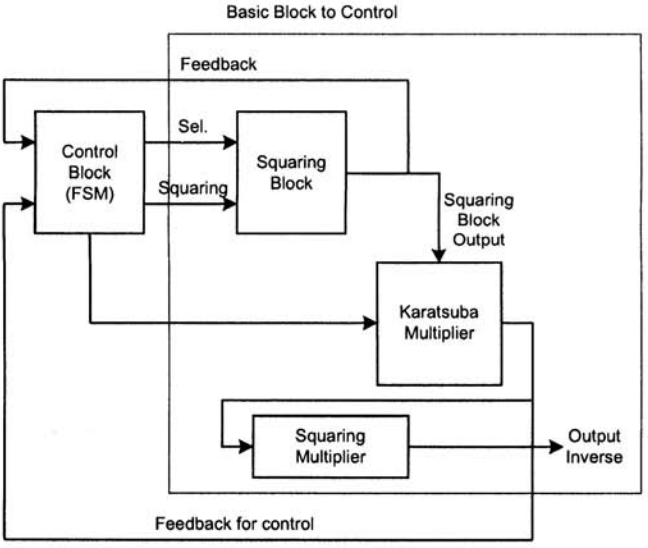

Fig. 6.11. Architecture of the Itoh-Tsujii Algorithm

Table 6.12. BEA Versus ITMIA: A Performance Comparison

Design	Cost	Cycles	Freq (MHz)	$timings$	$\frac{1}{Slices \times timings}$
BEA	1195	191	76.10	2509ηS	333.53
ITMIA	9945	40	55.25	724ηS	138.89
ITMIA without KOM Block	2345	40	55.25	724ηS	589.00

ing our customary efficiency figure of merit of $\frac{1}{Slices \times timings}$, we can see that the BEA solution is about 2.40 times more efficient than the ITMIA design.

Nevertheless, since for all practical cryptographic and code applications a binary extension field multiplier is a mandatory operator, we included the performance statistics of both, the ITMIA design considering the costs of the expensive Karatsuba-Ofman Multiplier (KOM) block and without considering it. In the case that the KOM block cost is taken out of the ITMIA statistics, Table 6.12 shows that the ITMIA solution becomes the most efficient option, providing An efficiency improvement of nearly 1.77 times with respect to the BEA design.

6.3.7 Multiplicative Inverse FPGA Designs

Table 6.13 shows the computational cost of several reported designs for the computation of multiplicative inversion over $GF(2^m)$ in hardware platforms. The *standard* Itoh-Tsujii algorithm using the architecture described here requires 28 clock cycles in the design reported in [295], thus computing the multiplicative inverse in about 1.32μS.

6.4 Other Arithmetic Operations

In this Section we briefly describe some important binary finite field arithmetic operations such as, the computation of the trace function, the half trace function and binary exponentiation. The first two operations are key building blocks for *halving* an elliptic curve point, which will be studied in §10.7.

6.4.1 Trace function

Given $C \in GF(2^m)$, the trace function can be defined as:

$$Tr(C) = C + C^2 + C^{2^2} + \ldots + C^{2^{m-1}} \tag{6.54}$$

Due to its linearity, the trace function can be implemented such that the execution time is $O(1)$ as [133],

Table 6.13. Design Comparison for Multiplicative Inversion in $GF(2^m)$

Work	Platform	Field	Cost	Cycles	Freq (MHz)	*timings*
BEA divisor [403]	0.18μm CMOS	$GF(2^{163})$	1.658 mm^2	198	400	0.495μS
BEA divisor [77]	0.18μm CMOS	$GF(2^{163})$	1.192 mm^2	326	460	0.709μS
ITMIA [248]	Xilinx Virtex 2	$GF(2^{193})$	9945	40	55.25	0.724μS
Parallel ITMIA [295]	Xilinx Virtex 2	$GF(2^{193})$	12021 CLBs	20	21.2	0.943μS
ITMIA [295]	Xilinx Virtex 2	$GF(2^{193})$	11081 CLBs	28	21.2	1.32μS
BEA [248]	Xilinx Virtex 2	$GF(2^{193})$	1195 CLBs	191	76.1	2.509μS
Montgomery Inversion [314]	0.18μm CMOS	160-bit	14.4K NANDs	1516	227.3	2.509μS
ITMIA [20]	Xilinx Virtex	$GF(2^{191})$	–	390	50	7.8μS (est.)
ITMIA [216]	Xilinx Virtex	$GF(2^{163})$		711	66	10.7μS
BEA [114]	0.25μm CMOS	$GF(2^{256})$	–	844	50	16.88μS (est.)

$$Tr(C) = Tr(\sum_{i=0}^{m-1} c_i x^i) = \sum_{i=0}^{m-1} c_i Tr(x^i) \tag{6.55}$$

As an example, consider the field defined by $GF(2^{163})$ with the reduction polynomial $p(x) = x^{163} + x^7 + x^6 + x^3 + 1$. Then, $Tr(x^i) = 1$ if and only if $i \in \{0, 157\}$. The implementation of the trace function in reconfigurable hardware only needs one XOR gate to add the bits *0 and 157* from the input polynomial.

6.4.2 Solving a Quadratic Equation over $GF(2^m)$

In order to solve a quadratic Equation (10.26), we may use the half-trace function. Let $C \in GF(2^m)$ be defined as $C(x) = \sum_{i=0}^{m-1} c_i x^i \in GF(2^m)$ with $Tr(C) = 0$ and m an odd integer, the half-trace function can be defined as:

$$H(C) = H(\sum_{i=0}^{m-1} c_i x^i) = \sum_{i=0}^{m-1} c_i H(x^i) \tag{6.56}$$

Therefore, by using the definition of the half trace equation 6.56. We can precompute the m half-traces of the field elements x^i for $i = 0, 1, \ldots, m-1$; and by arranging these Equations in a $m \times m$ matrix B, we may obtain the half-trace of an arbitrary element $C \in GF(2^m)$ by computing $H(C) = CB$.

6.4.3 Exponentiation over Binary Finite Fields

Exponentiation over binary finite fields is used for inverse computation via Fermat Little theorem [295] and key agreement schemes such as the Diffie-Hellman protocol, among other applications.

For binary extension fields $GF(2^m)$, generated using the m-degree irreducible polynomial $P(x)$, irreducible over $GF(2)$. Let e be an arbitrary m-bit positive integer e, with a binary expansion representation given as,

$$e = (1e_{m-2} \dots e_1 e_0)_2 = 2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i.$$

Then,

$$\begin{aligned} b = a^e &= a^{2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i} \qquad (6.57) \\ &= a^{2^{m-1}} \cdot a^{2^{m-2} e_{m-2}} \cdot \dots \cdot a^{2^1 e_1} \cdot a^{2^0 e_0} = a^{2^{m-1}} \cdot \prod_{i=0}^{m-2} a^{2^i e_i} \end{aligned}$$

Algorithm 6.11 MSB-first Binary Exponentiation

Require: The irreducible polynomial $P(x)$, $a \in GF(2^m)$, $e = (e_{m-1} \dots e_1 e_0)_2$
Ensure: $b = a^e \bmod P(x)$
1: $b = a$;
2: **for** $i = m - 2$ **downto** 0 **do**
3: $b = b^2$;
4: **if** $e_i == 1$ **then**
5: $b = b \cdot a \bmod P(x)$;
6: **end if**
7: **end for**
8: **Return** b

Binary strategies evaluate (6.57) by scanning the bits of the exponent e one by one, either from left to right (MSB-first binary algorithm) or from right to left (LSB-first binary algorithm) applying the so-called Horner's rule. Both strategies require a total of $m-1$ iterations. At each iteration a squaring operation is performed, and if the value of the scanned bit is one, a subsequent field multiplication is performed. Therefore, the binary strategy requires a total of $m-1$ squarings and $H(e)-1$ field multiplications, where $H(e)$ is the Hamming weight of the binary representation of e. The pseudo-code of the MSB-first binary algorithm is shown in Algorithm 6.11.

On the other hand, it is known from Fermat Little Theorem that for any nonzero $a \in GF(2^m)$, we have $a^{2^m-1} = 1$ which implies $a^{2^m} = a$ and by taking square root in both sides of the last relation we get $a^{2^{m-1}} = \sqrt{a} = a^{2^{-1}}$. In general, the i-th square-root of a, with $i \geq 1$ can be written as,

$$a^{2^{m-i}} = a^{2^{-i}}.$$

Hence, Eq. (6.57) can be reformulated in terms of the square root operator as,

$$a^e = a^{2^{m-1}} \cdot \prod_{i=0}^{m-2} a^{2^i e_i} = a^{2^{m-1}} \cdot a^{2^{m-2} e_{m-2}} \cdot \ldots \cdot a^{2^1 e_1} \cdot a^{2^0 e_0} \tag{6.58}$$

$$= a^{2^{-1}} \cdot a^{2^{-2} e_{m-2}} \cdot \ldots \cdot a^{2^{-(m-1)} e_1} \cdot a^{2^0 e_0} = \sqrt{a} \cdot \prod_{i=2}^{m-1} a^{2^{m-i} e_i} \cdot a^{e_0}$$

Algorithm 6.12 Square root LSB-first Binary Exponentiation

Require: The irreducible polynomial $P(x)$, $a \in GF(2^m)$, $e = (e_{m-1} \ldots e_1 e_0)_2$
Ensure: $b = a^e \bmod P(x)$
1: $b = a$;
2: $e_m = e_0$;
3: **for** $i = 1$ **to** m **do**
4: $b = \sqrt{b}$;
5: **if** $e_i == 1$ **then**
6: $b = b \cdot a \bmod P(x)$;
7: **end if**
8: **end for**
9: **Return** b

Therefore, the novel square root LSB-first binary strategy requires a total of $m - 1$ square root computations and $H(e) - 1$ field multiplications, where $H(e)$ is the Hamming weight of the binary representation of e. The pseudo-code of the square root LSB-first binary algorithm is shown in Algorithm 6.12. Algorithms 6.11 and 6.12 suggest a parallel version that can combine both ideas. This parallel version is especially attractive for hardware platforms implementations. Algorithm 6.13 shows this suggesting algorithm. Notice that both loop computations can be performed in parallel provided that the architecture has two independent field multiplier units. The computational time speedup can be estimated in about 50% when compared with Algorithms 6.11 and 6.12.

6.5 Conclusions

In this chapter, we addressed the problem of how to implement efficiently finite field arithmetic algorithms for reconfigurable hardware platforms. We included detailed analysis of complexities for binary field operations such as: multiplication, squaring, square root, multiplicative inverse computation, among others.

Algorithm 6.13 Squaring and Square Root Parallel Exponentiation

Require: The irreducible polynomial $P(x)$, $a \in GF(2^m)$, $e = (e_{m-1} \dots e_1 e_0)_2$
Ensure: $b = a^e \bmod P(x)$

```
1: b = c = 1 ;
2: e_m = 0 ;
3: N = ⌊m/2⌋ ;
4: for i = N downto 0 do          for j = N + 1 to m do
5:    b = b^2 ;                      c = √c;
6:    if e_i == 1 then               if e_j == 1 then
7:       b = b · a;                     c = c · a;
8:    end if
9: end for
10: b = b · c;
11: Return b
```

In §6.1, field multipliers algorithms were studied covering the whole spectrum of state-of-the-art strategies for computing that crucial arithmetic operation as efficiently as possible. That spectrum goes from the mighty fully bit-parallel Karatsuba-Ofman multiplier to the ultra compact interleaving multiplier which can be quite useful for constrained environments.

The most attractive feature of the Karatsuba-Ofman algorithm variation analyzed in §6.1.2, is that the degree m of the generating irreducible polynomial can be arbitrarily selected by the designer, allowing the usage of prime degrees. In addition, the new field multiplier leads to architectures which show a considerably improved space complexity when compared to traditional approaches. Moreover, the binary Karatsuba-Ofman multiplier leads to highly modular architectures that are well suited for both, VLSI and reconfigurable hardware implementations.

We studied in §6.1.4 a method able to perform the reduction step of field multipliers when an irreducible trinomial or pentanomial is used to generate the field. Moreover, we also presented a general method for accomplishing reduction when dealing with arbitrary irreducible polynomials.

In §6.2 a low-complexity bit-parallel algorithm for computing square roots over binary extension fields was studied. Although the method presented can be applied for any type of irreducible polynomials, we were particularly interested in studying the case of irreducible trinomials. Hence, in order to investigate the exact cost of the square root operator, we categorized irreducible trinomials over $GF(2)$ into four different types. For all four types considered, explicit area and time complexity formulae were found for both, field squaring and field square root operators. It was shown that for the important practical case of finite fields generated using irreducible trinomials, the square root operation can be performed with no more computational cost than the one associated to the field squaring operation.

In §6.3 we presented a performance comparison of two of the most popular algorithms for computing the field multiplicative inverse operation: the

Binary Euclidean Algorithm (BEA) and the Itoh-Tsujii Multiplicative Inverse Algorithm (ITMIA). It was shown that the Itoh-Tsujii strategy offers a competitive performance when implemented in hardware platforms. Furthermore, we combined the standard Itoh-Tsuii algorithm with the concept of addition chains. Then, we showed that for this version of the Itoh-Tsuii algorithm the multiplicative inverse of an arbitrary nonzero field element in GF(2^m) can be computed by performing exactly $m-1$ field squarings and t multiplications, where t is the step-length of the optimal addition-chain for m-1. One of the main conclusions of this Section is that according to Table 6.12 there is not a clear winner when comparing the BEA and the ITMIA methods.

Finally, in §6.4 some less popular field arithmetic operations were studied, such as, the computation of the trace function, the half trace function and binary field exponentiation. The first two operations are key building blocks for *halving* an elliptic curve point, which will be studied in §10.7.