Chapter 13
Finite-Field Arithmetic

Finite fields are used in different types of computers and digital communication systems. Two well-known examples are error-correction codes and cryptography. The traditional way of implementing the corresponding algorithms is software, running on general-purpose processors or on digital-signal processors. Nevertheless, in some cases the time constraints cannot be met with instruction-set processors, and specific hardware must be considered.

The operations over the finite ring Z_m are described in Sect. 13.1. Two multiplication algorithms are considered: "multiply and reduce" (Sect. 13.1.2.1) and "interleaved multiplication" (Sect. 13.1.2.2). The Montgomery multiplication, and its application to exponentiation algorithms, are the topics of Sect. 13.1.2.3. Section 13.2 is dedicated to the division over Z_p, where p is a prime. The proposed method is the "binary algorithm", an extension of an algorithm that computes the greatest common divider of two naturals. The operations over the polynomial ring $Z_2[x]/f(x)$ are described in Sect. 13.3. Two multiplication algorithms are considered: "multiply and reduce" (Sect. 13.3.2.1) and "interleaved multiplication" (Sect. 13.3.2.2). Squaring is the topic of Sect. 13.3.2.3. Finally, Sect. 13.4 is dedicated to the division over $GF(2^n)$.

As a matter of fact, only some of the most important algorithms have been considered. According to the Authors' experience they generate efficient FPGA implementations (Sect. 13.5). Furthermore, non-binary extension fields $GF(p^n)$ are not considered. A much more complete presentation of finite field arithmetic can be found in [1] and [2].

13.1 Operations Modulo *m*

Given a natural m, the set $Z_m = \{0, 1, \ldots, m-1\}$ is a ring whose operations are defined as modulo m.

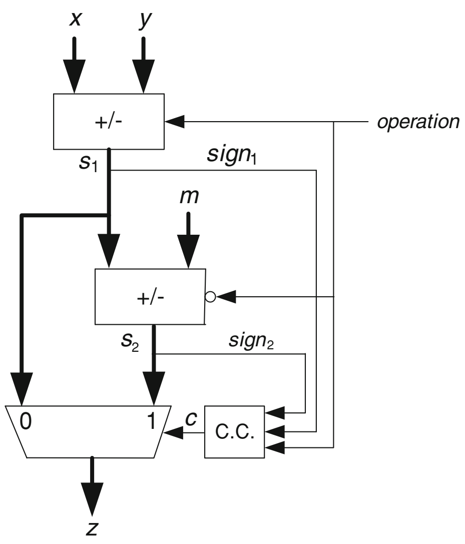

Fig. 13.1 Adder–subtractor modulo *m*

13.1.1 Addition and Subtraction Mod m

Given two natural numbers x and y belonging to Z_m, compute $z = (x + y)$ mod m. Taking into account that $0 \leq x + y < 2 \cdot m, z$ must be equal to either $x + y$ or $x + y - m$, the following algorithm computes z.

Algorithm 13.1: Mod *m* addition

```
z1 := x + y; z2 := z1 - m;
if z2 >= 0 then z := z2; else z := z1; end if;
```

As regards the computation of $z = (x - y)$ mod m, take into account that $-m < x - y < m$, so that z must be equal to either $x - y$ or $x - y + m$. The corresponding algorithm is the following.

Algorithm 13.2: Mod *m* subtraction

```
z1 := x - y; z2 := z1 + m;
if z1 < 0 then z := z2; else z := z1; end if;
```

The circuit of Fig. 13.1 is an adder-subtractor, which computes $z = (x + y)$ mod m if $operation = 0$ and $z = (x - y)$ mod m if $operation = 1$. It is described by the following VHDL model in which k is the number of bits of m.

```
 WITH operation SELECT s1 <=
   ('0'&x) + y WHEN '0', ('0'&x) - y WHEN OTHERS;
WITH operation SELECT s2 <=
   s1 + m WHEN '1', s1 - m WHEN OTHERS;
c <=
   (NOT(operation) AND s2(k)) OR (operation AND NOT(s1(k)));
WITH c SELECT z <=
   s1(k-1 DOWNTO 0) WHEN '1', s2(k-1 DOWNTO 0) WHEN OTHERS;
```

A complete VHDL model is *mod_m_AS.vhd* is available at the Authors' web page.

13.1.2 Multiplication Mod m

Given x and $y \in Z_m$, compute $z = x{\cdot}y$ mod m, where m is a k-bit natural.

13.1.2.1 Multiply and Reduce

A straightforward method consists of multiplying x by y, so that a $2k$-bit result *product* is obtained, and then reducing *product* mod m. For that, any combination of multiplier and mod m reducer can be used. For fixed values of m, specific combinational mod m reducers can be considered.

As an example, synthesize a mod m multiplier with $m = 2^{192} - 2^{64} - 1$. Any 192-by-192 multiplier can be used. A 384-bit to 192-bit mod m reducer can be synthesized as follows: given $x = x_{383} \cdot 2^{383} + x_{382} \cdot 2^{382} + \ldots + x_0 \cdot 2^0$, it can be divided up under the form

$$(x_{383} \cdot 2^{63} + x_{382} \cdot 2^{62} + \ldots + x_{320} \cdot 2^0)2^{320} + (x_{319} \cdot 2^{63} + x_{318} \cdot 2^{62} + \ldots + x_{256} \cdot 2^0)2^{256}$$
$$+ (x_{255} \cdot 2^{63} + x_{254} \cdot 2^{62} + \ldots + x_{192} \cdot 2^0)2^{192} + x_{191} \cdot 2^{191} + x_{190} \cdot 2^{190} + \ldots + x_0 \cdot 2^0.$$

Then, substitute 2^{320} by $2^{128} + 2^{64} + 1 \equiv 2^{320}$ mod m, 2^{256} by $2^{128} + 2^{64} \equiv 2^{256}$ mod m, and 2^{192} by $2^{64} + 1 \equiv 2^{192}$ mod m. So, $x \equiv x' + x'' + x''' + x''''$, where

$$\begin{aligned}
x' &= (x_{383} \cdot 2^{191} + x_{382} \cdot 2^{190} + \ldots + x_{320} \cdot 2^{128}) \\
&\quad + (x_{383} \cdot 2^{127} + x_{382} \cdot 2^{126} + \ldots + x_{320} \cdot 2^{64}) \\
&\quad + (x_{383} \cdot 2^{63} + x_{382} \cdot 2^{62} + \ldots + x_{320} \cdot 2^{0}), \\
x'' &= (x_{319} \cdot 2^{191} + x_{318} \cdot 2^{190} + \ldots + x_{256} \cdot 2^{128}) \\
&\quad + (x_{319} \cdot 2^{127} + x_{318} \cdot 2^{126} + \ldots + x_{256} \cdot 2^{64}), \\
x''' &= (x_{255} \cdot 2^{127} + x_{254} \cdot 2^{126} + \ldots + x_{192} \cdot 2^{64}) \\
&\quad + (x_{255} \cdot 2^{63} + x_{254} \cdot 2^{62} + \ldots + x_{192} \cdot 2^{0}), \\
x'''' &= x_{191} \cdot 2^{191} + x_{190} \cdot 2^{190} + \ldots + x_0 \cdot 2^0.
\end{aligned}$$

The sum $s = x' + x'' + x''' + x''''$ is smaller than $4 \times (2^{192} - 2^{64} - 1) = 4\,m$, so that x mod m is either s, $s - m$, $s - 2m$ or $s - 3m$.

The corresponding circuit is shown in Fig. 13.2 and is described by the following VHDL model.

```
s1 <= ('0' & x(383 DOWNTO 320) & x(383 DOWNTO 320) &
  x(383 DOWNTO 320)) + x(191 DOWNTO 0);
s2(192 DOWNTO 64)  <= ('0' & x(319 DOWNTO 256) &
  x(319 DOWNTO 256)) + x(255 DOWNTO 192);
s2(63 DOWNTO 0) <= x(255 DOWNTO 192);
s <= ('0'&s1) + s2;
z1 <= ('0'&s) + ("11"&minus_m);
z2 <= s + minus_2m;
z3 <= s(192 DOWNTO 0) + minus_3m;
PROCESS(z1, z2, z3, s)
BEGIN
  IF z1(194) = '1' THEN z <= s(191 DOWNTO 0);
  ELSIF z2(193) = '1' THEN z <= z1(191 DOWNTO 0);
  ELSIF z3(192) = '1' THEN z <= z2(191 DOWNTO 0);
  ELSE z <= z3(191 DOWNTO 0);
  END IF;
END PROCESS;
```

A complete VHDL model is *mod_p192_reducer2.vhd* is available at the Authors' web page.

In order to complete the multiplier design, any 192-bit by 192-bit multiplier can be used (Chap. 8). This is left as an exercise.

13.1.2.2 Interleaved Multiplier

Another option is to modify a classical left-to-right multiplication algorithm based on the following computation scheme

$$x \cdot y = (\ldots((0 \cdot 2 + x_{n-1} \cdot y) \cdot 2 + x_{n-2} \cdot y) \cdot 2 + \ldots + x_1 \cdot y) \cdot 2 + x_0 \cdot y.$$

Algorithm 13.3: Mod *m* multiplication, left-to-right algorithm

```
accumulator := 0;
for i in 0 .. k-1 loop
  accumulator := (accumulator·2 + $x_{k-i-1}$·y) mod m;
end loop;
product := accumulator;
```

The data path corresponding to Algorithm 13.3 is shown in Fig. 13.3. It is described by the following VHDL model.

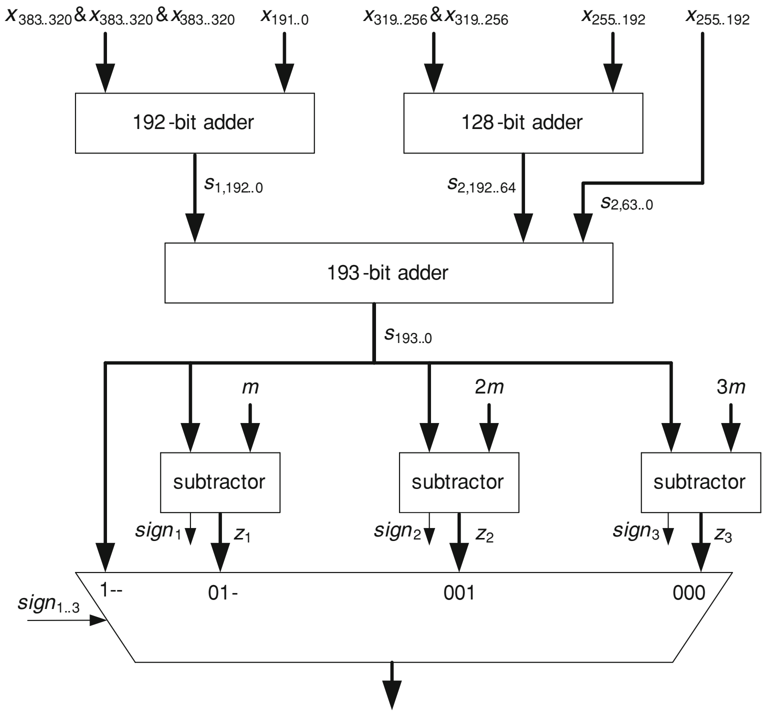

Fig. 13.2 Mod $2^{192} - 2^{64} - 1$ reducer

```
vector_xi <= (OTHERS => xi); xi_by_y <= y AND vector_xi;
s <= '0'&acc&'0' + xi_by_y;
z1 <= ('0'&s) + ("11"&minus_m);
z2 <= ('0'&s) + ('1'&minus_2m);
PROCESS(z1, z2, s)
BEGIN
  IF z1(k+2) = '1' THEN next_acc <= s(k-1 DOWNTO 0);
  ELSIF z2(k+2) = '1' THEN next_acc <= z1(k-1 DOWNTO 0);
  ELSE next_acc <= z2(k-1 DOWNTO 0);
  END IF;
END PROCESS;
register_acc: PROCESS(clk) ...
shift_register: PROCESS(clk) ...
xi <= int_x(k-1);
```

A complete VHDL model *mod-_m_multiplier.vhd-*, including a *k*-state counter and a control unit, is available at the Authors' web page.

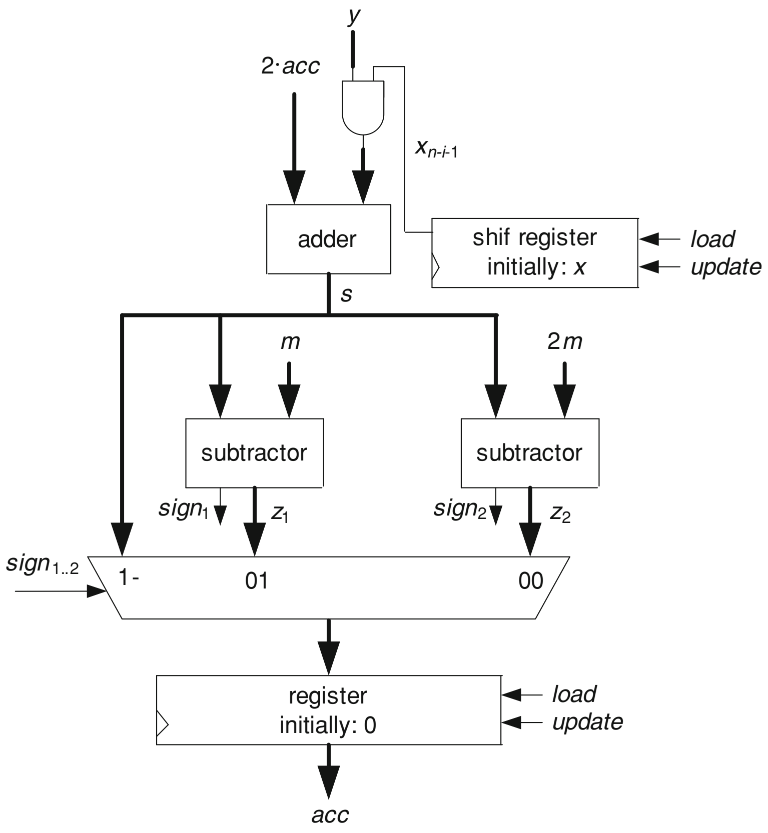

Fig. 13.3 Interleaved mod *m* multiplier

If ripple-carry adders are used, the minimum clock period is about $k \cdot T_{FA}$, so that the total computation time is approximately equal to $k^2 \cdot T_{FA}$. In order to reduce the computation time, the stored-carry encoding principle could be used [2]. For that, Algorithm 13.3 is modified: *accumulator* is represented under the form $acc_s + acc_c$; the conditional sum $(acc_s + acc_c) \cdot 2 + x_{n\text{-}k\text{-}i} \cdot y$ is computed in stored-carry form, and every sum is followed by a division by m, also in stored-carry form (Sect. 9.2.4), without on-the-fly conversion as only the remainder must be computed. The corresponding computation time is proportional to k instead of k^2. The design of the circuit is left as an exercise.

13.1.2.3 Montgomery Multiplication

Assume that m is an odd k-bit number. As m is odd, then $gcd(m, 2^k) = 1$, and there exists an element 2^{-k} of Z_m such that $2^k \cdot 2^{-k-} \equiv 1 \bmod m$. Define a one-to-one and onto application T from Z_m to Z_m:

$$T(x) = x.2^k \bmod m \text{ and } T^{-1}(y) = y.2^{-k} \bmod m.$$

The following properties hold true: $T((x + y) \bmod m) = (T(x) + T(y)) \bmod m$, $T((x - y) \bmod m) = (T(x) - T(y)) \bmod m$, $T(x \cdot y \bmod m) = T(x) \cdot T(y).2^{-k} \bmod m$. The latter suggests the definition of a new operation on Z_m, the so-called *Montgomery product MP* [3]:

$$MP(x, y) = x \cdot y \cdot 2^{-k} \bmod m.$$

Assume that the value $2^{2k} \bmod m$ has been previously computed. Then

$$T(x) = MP\left(x, 2^{2k} \bmod m\right) \text{ and } T^{-1}(y) = MP(y, 1).$$

The main point is that the Montgomery product MP is easier to compute than the mod m product. The following algorithm computes $MP(x, y)$.

Algorithm 13.4: Montgomery product

```
p := 0;
for i in 0 .. k-1 loop
  $q_i$ := ($p_0$ + $x_i \cdot y_0$) mod 2;
  p := (p + $x_i \cdot$y + $q_i \cdot$m)/2;
end loop;
if p ≥ m then z := p-m; else z := p; end if;
```

The data path corresponding to Algorithm 13.4 (without the final correction) is shown in Fig. 13.4. It is described by the following VHDL model.

```
q <= p(0) XOR (xi AND y(0));
vector_xi <= (OTHERS => xi); vector_q <= (OTHERS => q);
two_p <= ('0'&p) + (vector_xi AND y) +  (vector_q AND m);
next_p <= two_p(k+1 DOWNTO 1);
p_minus_m <= p + minus_m;
register_p: PROCESS(clk) ...
shift_register: PROCESS(clk) ...
xi <= int_x(0);
```

If ripple-carry adders are used, the total computation time is approximately equal to $k^2 \cdot T_{FA}$. In order to reduce the computation time, the stored-carry encoding principle could be used [2, 4].

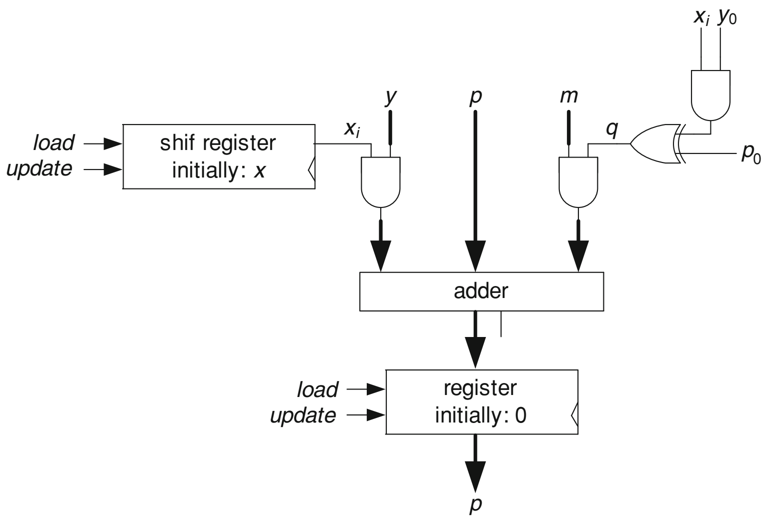

Fig. 13.4 Montgomery product

Algorithm 13.5: Montgomery product, carry-save addition

```
$p_c$ := 0; $p_s$ := 0;
for i in 0 .. k-1 loop
  $q_i$ := ($p_{c0}$ + $p_{s0}$ + $x_i \cdot y_0$) mod 2;
  ($p_c$, $p_s$) := ($p_c$ + $p_s$ + $x_i \cdot y$ + $q_i \cdot m$)/2;
end loop;
if p ≥ m then z := p-m; else z := p; end if;
```

The corresponding computation time is proportional to k instead of k^2. The design of the circuit is left as an exercise.

In order to compute $z = x{\cdot}y \bmod m$, the Montgomery product concept should be used in the following way: first (initial encoding) substitute x and y by $x' = T(x) = MP(x, 2^{2k} \bmod m)$ and $y' = T(y) = MP(y, 2^{2k} \bmod m)$; then compute $z' = MP(x', y')$; finally (result decoding) compute $z = T^{-1}(z') = MP(z, 1)$. This method is not efficient, unless many operations involving the same initial data are performed, in which case the initial encoding of those data is performed only once. Consider the following modular exponentiation algorithm; it computes $z = y^x \bmod m$ and is based on the following computation scheme:

$$z = y^{x_0 + x_1 \cdot 2 + x_2 \cdot 2^2 + \ldots + x_{k-1} \cdot 2^{k-1}} = y^{x_0} \cdot (y^2)^{x_1} \cdot (y^{2^2})^{x_1} \cdot \ldots \cdot (y^{2^{k-1}})^{x_{k-1}} \bmod m.$$

Algorithm 13.6: Mod *m* exponentiation, LSB-first

```
e := 1;
for i in 0 .. k-1 loop
  if $x_i$ = 1 then e := e·y mod m; end if;
  y := $y^2$ mod m ;
end loop;
z := e;
```

Mod *m* operations can be substituted by Montgomery products. For that, 1 is substituted by $T(1) = 2^k$ mod *m*, and *y* by $T(y) = MP(y,\ 2^{2k} \bmod m)$. Thus, assuming that 2^k mod *m* and 2^{2k} mod *m* have been previously computed, the following algorithm computes $z = x{\cdot}y$ mod *m*.

Algorithm 13.7: Modulo *m* exponentiation, Montgomery algorithm, LSB-first

```
te := $2^k$ mod m; ty := MP(y, $2^{2k}$ mod m);
for i in 0 .. k-1 loop
  if $x_i$ = 1 then te := MP(te, ty); end if;
  ty := MP(ty, ty);
end loop;
z := MP(te, 1);
```

The corresponding circuit is made up of two *Montgomery multipliers* working in parallel, with some kind of synchronization mechanism. A data-flow VHDL description *mod_m_exponentiation.vhd* is available at the Authors' web page. At each step both multipliers MP_1 and MP_2, with their corresponding $done_1$ and $done_2$ signals, are synchronized with a *wait* instruction:

```
WAIT UNTIL (done1 AND done2) = '1';
```

An MSB-first exponentiation algorithm could also be considered. For that, use the following computation scheme:

$$\begin{aligned} z &= y^{x_0 + x_1 \cdot 2 + x_2 \cdot 2^2 + \ldots + x_{k-1} \cdot 2^{k-1}} \\ &= ((\ldots(1^2 \cdot y^{x_{k-1}})^2 \cdot y^{x_{k-2}})^2 \cdot \ \ldots \ \cdot y^{x_1})^2 \cdot y^{x_0} \bmod m. \end{aligned}$$

Algorithm 13.8: Mod *m* exponentiation, MSB-first

```
e := 1;
for i in 1 .. k loop
  e := $e^2$ mod m;
  if $x_{k-i}$ = 1 then e := e·y mod m; end if;
end loop;
z := e;
```

As before, mod *m* multiplications can be substituted by Montgomery products. For that, 1 is substituted by $T(1) = 2^k \bmod m$, and y by $T(y) = MP(y, 2^{2k} \bmod m)$. There is a precedence relation between the two main operations $T(e) := T(e^2) = MP(T(e), T(e))$ and $T(e) := T(e \cdot y) = MP(T(e), T(y))$. Thus, a direct implementation includes only one *Montgomery multiplier*, but needs up to $2k$ cycles instead of k in the case of the LSB-first algorithm. The design of the corresponding circuit is left as an exercise.

13.2 Division Modulo *p*

If p is prime, than all non-zero elements of Z_p have a multiplicative inverse. Thus, given x and $y \neq 0$ in Z_p, there exists an element z of Z_p such that $z = x \cdot y^{-1} \bmod p$.

There are several types of algorithms that compute z. Some of them are generalizations of algorithms that compute the *greatest common divider*: Euclidean algorithm [5, 6], binary algorithm [7], plus-minus algorithm [8–10]. Another option is to substitute division by multiplications: according to the Fermat's little theorem $z = x \cdot y^{p-2} \bmod p$. As an example, the following binary algorithm computes $z = x \cdot y^{-1} \bmod p$. It uses four variables a, b, c and d, initially equal to p, y, 0 and x, respectively. At each step, a and b are updated in such a way that their *gcd* is unchanged and that b decreases. For that, observe that if b is even and a is odd, then $gcd(a, b) = gcd(a, b/2)$, and if both a and b are odd, then $gcd(a, b) = gcd(a, |b\text{-}a|) = gcd(b, |b\text{-}a|)$. As initially $a = p$ and $b = y$, where p is a prime, after a finite number of steps a is equal to 1. On the other hand, c and d are updated in such a way that $c \cdot y \equiv a \cdot x \bmod p$ and $d \cdot y \equiv b \cdot x \bmod p$. Initially, $c = 0$, $a = p \equiv 0 \bmod p$, $d = x$ and $b = y$, so that the mentioned relations are satisfied. It can be proven that if c and d are updated in the same way as a and b, both relations remain true. In particular, if $a = 1$, then $c \cdot y \equiv x \bmod p$, and $z = c$.

Algorithm 13.9: Mod p division, binary algorithm

```
a := p; b := y; c := 0; d := x;
while a ≠ 1 loop
  if b₀ = 0 then b := b/2; d := d·2⁻¹ mod p;
  elsif b ≥ a then b := b-a; d := (d-c) mod p;
  else (b, a) := (a-b, b); (d, c) := ((c-d) mod p, d);
  end if;
end loop;
z := c;
```

The corresponding circuit is made up of adders, registers and connection resources. A data-flow VHDL description *mod_p_division2.vhd* is available at the Authors' web page.

An upper bound of the number of steps before $a = 1$ is $4k$, k being the number of bits of p. So, if ripple-carry adders are used, the computation time is shorter than $4 \cdot k^2 \cdot T_{FA}$ (a rather pessimistic estimation).

13.3 Operations Over $\mathbf{Z_2[x]/f(x)}$

Given a polynomial $f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1x + f_0$, whose coefficients f_i belong to the binary field Z_2, the set of polynomials of degree smaller than m over Z_2 is a ring $Z_2[x]/f(x)$ whose operations are defined modulo $f(x)$.

13.3.1 Addition and Subtraction of Polynomials

Given two polynomials $a(x) = a_{m-1}x^{m-1} + \ldots + a_1x + a_0$ and $b(x) = b_{m-1}x^{m-1} + \ldots + b_1x + b_0$, then

$$a(x) + b(x) = a(x) - b(x) = c_{m-1}x^{m-1} + \ldots + c_1x + c_0,$$

where $c_i = (a_i + b_i) \bmod 2, \forall i \,\text{in}\{0, 1, \ldots, m-1\}$. In other words, the corresponding circuit is a set of m 2-input XOR gates working in parallel, which can be described by the following VHDL sentence:

```
c <= a XOR b;
```

13.3.2 Multiplication Modulo f(x)

Given two polynomials $a(x) = a_{m-1}x^{m-1} + \ldots + a_1x + a_0$ and $b(x) = b_{m-1}x^{m-1} + \ldots + b_1x + b_0$ of degree smaller than m, and a polynomial $f(\mathrm{x}) = x^m + f_{m-1}x^{m-1} + \ldots + f_1x + f_0$, compute $c(x) = a(x) \cdot b(x) \bmod f(x)$.

13.3.2.1 Multiply and Reduce

A straightforward method consists of multiplying $a(x)$ by $b(x)$, so that a polynomial $d(x)$ of degree smaller than $2m$-1 is obtained, and then reducing $d(x)$ mod $f(x)$.

The coefficients d_k of $d(x)$ are the following:

$$d_k = \sum\nolimits_{i=0}^{k} a_i \cdot b_{k-i},\ k = 0, 1, \ldots, m-1,$$
$$d_k = \sum\nolimits_{i=k}^{2m-2} a_{k-i+(m-1)} \cdot b_{i-(m-1)},\ k = m, m+1, \ldots, 2m-2.$$

The preceding equations can be implemented by a combinational circuit made up of 2-input AND gates and XOR gates with up to m inputs.

```
gen_ands: FOR k IN 0 TO m-1 GENERATE
  l1: FOR i IN 0 TO k GENERATE
  a_by_b(k)(i) <= a(i) AND b(k-i); END GENERATE;
END GENERATE;
gen_ands2: FOR k IN m TO 2*m-2 GENERATE
  l2: FOR i IN k TO 2*m-2 GENERATE
  a_by_b(k)(i) <= a(k-i+(m-1)) AND b(i-(m-1)); END GENERATE;
END GENERATE;
d(0) <= a_by_b(0)(0);
gen_xors: FOR k IN 1 TO 2*m-2 GENERATE
  l3: PROCESS(a_by_b(k),c(k))
      VARIABLE aux: STD_LOGIC;
  BEGIN
    IF (k < m) THEN aux := a_by_b(k)(0);
      FOR i IN 1 TO k LOOP aux := a_by_b(k)(i) XOR aux;
      END LOOP;
    ELSE aux := a_by_b(k)(k);
      FOR i IN k+1 TO 2*m-2 LOOP aux := a_by_b(k)(i) XOR aux;
      END LOOP;
    END IF;
    d(k) <= aux;
  END PROCESS;
END GENERATE;
```

It remains to reduce $d(x)$ modulo $f(x)$. Assume that all coefficients $r_{i,j}$, such that

$$x^{m+j} \bmod f(x) = r_{m-1,j}x^{m-1} + r_{m-2,j}x^{m-2} + \cdots + r_{1,j}x^1 + r_{0,j},$$

have been previously computed. Then the coefficients of $c(x) = a(x) \cdot b(x)$ mod $f(x)$ are the following:

$$c_j = d_j + \sum\nolimits_{i=0}^{m-2} r_{j,i} \cdot d_{m+i},\ j = 0, 1, \ldots, m-1. \qquad (13.1)$$

The preceding equations can be implemented by a combinational circuit made up of m XOR gates with up to m inputs. The number of gate inputs is determined by the maximum number of 1's within a column of the matrix $[r_{i,j}]$, and this depends on the chosen polynomial $f(x)$.

```
gen_xors: FOR j IN 0 TO m-1 GENERATE
  l1: PROCESS(d)
    VARIABLE aux: STD_LOGIC;
  BEGIN
    aux := d(j);
    FOR i IN 0 TO m-2 LOOP
    aux := aux XOR (d(m+i) and r(j)(i)); END LOOP;
    c(j) <= aux;
  END PROCESS;
END GENERATE;
```

A complete VHDL model *classic_multiplier.vhd*, including both the polynomial multiplier and the polynomial reducer, is available at the Authors' web page.

Every coefficient d_k is the sum of at most m products $a_i{\cdot}b_{k-i}$, and every coefficient c_j is the sum of, at most, m coefficients d_k. Thus, if tree structures are used, the computation time is proportional to log m. On the other hand, the cost is proportional to m^2. Hence, this type of multiplier is suitable for small values of m. For great values of m, the cost could be excessive and sequential multipliers should be considered.

13.3.2.2 Interleaved Multiplier

The following LSB-first algorithm computes $c(x) = a(x){\cdot}b(x)$ mod $f(x)$ according to the following computation scheme:

$$c(x) = b_0 \cdot a(x) + b_1 \cdot (a(x) \cdot x) + b_2 \cdot \left(a(x) \cdot x^2\right) + \ldots + b_{m-1} \cdot \left(a(x) \cdot x^{m-1}\right).$$

Algorithm 13.10: Interleaved multiplication, LSB-first

```
c(x) := 0;
for i in 0 .. m-1 loop
  c(x) := c(x) + $b_i$·a(x);
  a(x) := a(x)·x mod f(x);
end loop;
```

The first operation $c(x) + b_i{\cdot}a(x)$ is executed by the circuit of Fig. 13.5a. In order to compute $a(x){\cdot}x$ mod $f(x)$, use the fact that $x^m \bmod f(x) = f_{m-1}x^{m-1} + f_{m-2}x^{m-2} + \ldots + f_0$. Thus,

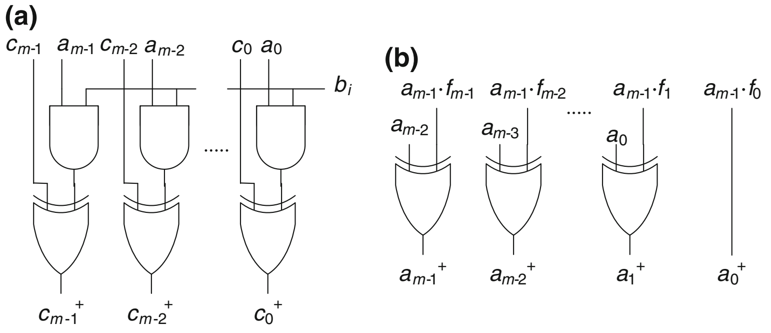
(a)
c_{m-1} a_{m-1} c_{m-2} a_{m-2} c_0 a_0
b_i
.....
c_{m-1}^+ c_{m-2}^+ c_0^+
(b)
$a_{m-1}\cdot f_{m-1}$ $a_{m-1}\cdot f_{m-2}$ $a_{m-1}\cdot f_1$ $a_{m-1}\cdot f_0$
a_{m-2} a_{m-3} a_0
.....
a_{m-1}^+ a_{m-2}^+ a_1^+ a_0^+

Fig. 13.5 Interleaved multiplier, computation of $c(x) + b_i{\cdot}a(x)$ and $a(x){\cdot}x$ mod $f(x)$

$$a(x) \cdot x \bmod f(x) = a_{m-1} \cdot \left(f_{m-1}x^{m-1} + f_{m-2}x^{m-2} + \ldots + f_0\right) + a_{m-2}x^{m-1} + a_{m-3}x^{m-2} + \ldots + a_0x.$$

The corresponding circuit is shown in Fig. 13.5b. For fixed $f(x)$, the products $a_{m\text{-}1}{\cdot}f_j$ must not be computed: if $f_j = 1$, $a_j^+ = \left(a_{j-1} + a_{m-1}\right)$ mod 2, and if $f_j = 0$, $a_j^+ = a_{j-1}$.

The circuit also includes two parallel registers a and c, a shift register b, an m-state counter, and a control unit. A complete VHDL model *interleaved_mult.vhd* is available at the Authors' web page.

13.3.2.3 Squaring

Given $a(x) = a_{m-1}x^{m-1} + \ldots + a_1x + a_0$, the computation of $c(x) = a^2(x)$ mod $f(x)$ can be performed with the algorithm of Sect. 13.3.2.1. The first step (multiply) is trivial:

$$\begin{aligned} d(x) = a^2(x) &= (a_{m-1}x^{m-1} + \ldots + a_1x + a_0)^2 \\ &= a_{m-1}x^{2(m-1)} + a_{m-2}x^{2(m-2)} + \ldots + a_1x^2 + a_0. \end{aligned}$$

Thus, $d_i = a_{i/2}$ if i is even, else $d_i = 0$. According to (13.1),

$$c_j = a_{j/2} + \sum_{\substack{0 \le i \le m-2 \\ m+i\ even}} r_{j,i} \cdot a_{(m+i)/2},\ j = 0, 2, 4, \ldots$$

$$c_j = \sum_{\substack{0 \le i \le m-2 \\ m+i\ even}} r_{j,i} \cdot a_{(m+i)/2},\ j = 1, 3, 5, \ldots$$

The cost of the circuit depends on the chosen polynomial $f(x)$, which, in turn, defines the matrix $[r_{i,j}]$. If $f(x)$ has few non-zero coefficients, as is the case of trinomials and pentanomials, then the matrix $[r_{i,j}]$ also has few non-zero coefficients, and the corresponding circuit is very fast and cost-effective. Examples of implementations are given in Sect. 13.5.

13.4 Division Over $GF(2^m)$

If $f(x)$ is irreducible, then all non-zero polynomials in $Z_2[x]/f(x)$ have a multiplicative inverse. Thus, given $g(x)$ and $h(x) \neq 0$ in $Z_2[x]/f(x)$, there exists a polynomial $z(x)$ in $Z_2[x]/f(x)$ such that $z(x) = g(x) \cdot h^{-1}(x) \bmod f(x)$.

There are several types of algorithms for computing $z(x)$. Some of them are generalizations of algorithms that compute the *greatest common divider*, like the Euclidean algorithm and the binary algorithm. Another option is to substitute the division by multiplications: according to the Fermat's theorem $z(x) = g(x) \cdot h^{q-2}(x) \bmod f(x)$ where $q = 2^m$. As an example, the following binary algorithm computes $z(x) = g(x) \cdot h^{-1}(x) \bmod f(x)$. It uses four variables $a(x)$, $b(x)$, $u(x)$ and $v(x)$, initially equal to $f(x)$, $h(x)$, 0 and $g(x)$, respectively. At each step, $a(x)$ and $b(x)$ are updated in such a way that their greatest common divider is unchanged and that the degree of $a(x) + b(x)$ decreases. For that, observe that if $b(x)$ is divisible by x and $a(x)$ is not, then $gcd(a(x), b(x)) = gcd(a(x), b(x)/x)$, and if neither $a(x)$ nor $b(x)$ are divisible by x, then $gcd(a(x), b(x)) = gcd(a(x), (a(x) + b(x))/x) = gcd(b(x), (a(x) + b(x))/x)$. As initially $a(x) = f(x)$ and $b(x) = h(x)$, where $f(x)$ is irreducible, after a finite number of steps $b(x) = 0$ and $a(x) = gcd(f(x), h(x)) = 1$. On the other hand, $u(x)$ and $v(x)$ are updated in such a way that $u(x) \cdot h(x) \equiv a(x) \cdot g(x) \bmod f(x)$ and $v(x) \cdot h(x) \equiv b(x) \cdot g(x) \bmod f(x)$. Initially, $u(x) = 0$, $a(x) = f(x) \equiv 0 \bmod f(x)$, $v(x) = g(x)$ and $b(x) = h(x)$, so that both equivalence relations are satisfied. It can be proven that if $u(x)$ and $v(x)$ are updated in the same way as $a(x)$ and $b(x)$, both relations remain true. In particular, if $a(x) = 1$, then $u(x) \cdot h(x) \equiv g(x) \bmod f(x)$, and $z(x) = u(x)$.

Algorithm 13.11: Mod $f(x)$ division, binary algorithm

```
a(x) := f(x); b(x) := h(x); u(x) := 0; v(x) := g(x);
while b(x) ≠ 0 loop
  if $b_0$ = 0 then b(x) := b(x)/x; v(x) := v(x)·$x^{-1}$ mod f(x);
  elsif degree(a(x)) ≥ degree(b(x)) then
    (a(x), b(x), u(x), v(x)) :=
    (b(x), (a(x)+b(x))/x, v(x), (u(x)+v(x))·$x^{-1}$ mod f(x));
  else
    b(x) := (a(x)+b(x))/x; v(x) := (u(x)+v(x))·$x^{-1}$ mod f(x));
  end if;
end loop;
z(x) := u(x);
```

Given a polynomial $w(x)$, then $w(x) \cdot x^{-1} \bmod f(x) = (w(x) + w_0 \cdot f(x))/x$. A data path for executing the preceding algorithm is shown in Fig. 13.6. The small circles connected to $f_1, f_2, \ldots$, represent programmable connections: if $f_i = 1$, the rightmost input of the corresponding XOR gate is connected to w_0, else it is connected to 0.

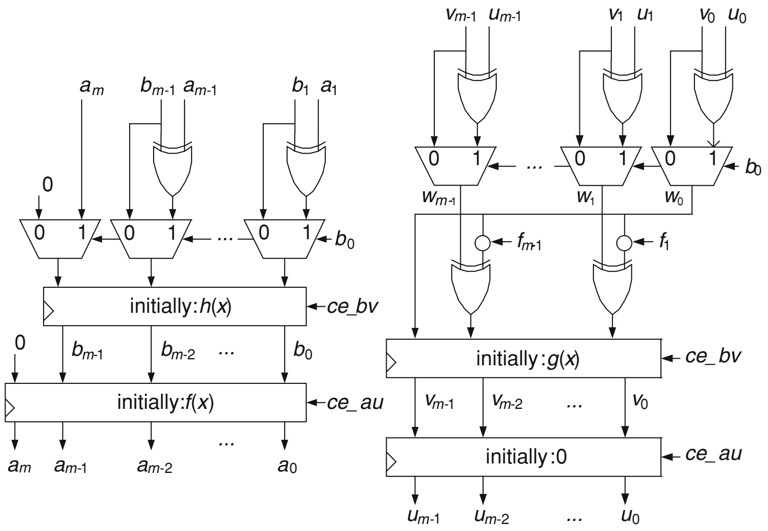

Fig. 13.6 Binary algorithm for polynomials

A drawback of the proposed algorithm is that the degrees of $a(x)$ and $b(x)$ must be computed at each step. A better option is to use upper bounds α and β of the degrees of $a(x)$ and $b(x)$.

Algorithm 13.12: Mod $f(x)$ division, binary algorithm, version 2

```
a(x) := f(x); b(x) := h(x); u(x) := 0; v(x) := g(x);
α := m; β := m-1;
while β ≥ 0 loop
  if b₀ = 0 then b(x) := b(x)/x; v(x) := v(x)·x^-1 mod f(x);
    β := β-1;
  elsif α ≥ β then (a(x), b(x), u(x), v(x)) :=
    (b(x), (a(x)+b(x))/x, v(x), (u(x)+v(x))·x^-1 mod f(x));
    (α, β) := (β, α-1);
  else
    b(x) := (a(x)+b(x))/x; v(x) := (u(x)+v(x))·x^-1 mod f(x));
    β := β-1;
  end if;
end loop;
z(x) := u(x);
```

The data path is the same as before (Fig. 13.6). An upper bound of the number of steps is $2m$. As the operations are performed without carry propagation, the computation time is proportional to m. A data-flow VHDL description *mod_f_division2.vhd* is available at the Authors' web page.

In the preceding binary algorithms, the number of steps is not fixed; it depends on the input data values. This is an inconvenience when optimization methods, such as digit-serial processing (Chap. 3), are considered. In the following algorithm [11] α and β are substituted by $count = |\alpha\text{-}\beta\text{-}1|$, and a binary variable *state* equal to 0 if $\alpha > \beta$, and equal to 1 if $\alpha \leq \beta$.

Algorithm 13.13: Mod *f*(*x*) division, binary algorithm, version 3

```
a(x) := f(x); b(x) := h(x); u(x) := 0; v(x) := g(x);
count := 0; state := 0;
for i in 1 .. 2m loop
  if state = 0 then count := count+1;
    if $b_0$ = 0 then b(x) := b(x)/x; v(x) := v(x)·$x^{-1}$ mod f(x);
    else (a(x), b(x), u(x), v(x)) :=
      (b(x), (a(x)+b(x))/x, v(x), (u(x)+v(x))·$x^{-1}$ mod f(x));
       state := 1;
    end if;
  else count := count -1;
    if $b_0$ = 0 then b(x) := b(x)/x; v(x) := v(x)·$x^{-1}$ mod f(x);
    else b(x) := (a(x)+b(x))/x;
      v(x) := (u(x)+v(x))·$x^{-1}$ mod f(x));
    end if;
    if count = 0 then state := 0; end if;
  end if;
```

The data path is still the same as before (Fig. 13.6), and the number of steps is 2*m*, independently of the input data values. As the operations are performed without carry propagation, the computation time is proportional to *m*. A data-flow VHDL description *mod_f_division3.vhd* is available at the Authors' web page. Furthermore, several digit-serial implementations, with different digit definitions, are given in Sect. 13.5.

13.5 FPGA Implementations

Several circuits have been implemented within a Virtex 5–2 device. The times are expressed in *ns* and the costs in numbers of Look Up Tables (LUTs) and flip-flops (FFs). All VHDL models are available at the Authors' web page.

Two combinational multipliers (multiply and reduce) have been implemented (Table 13.1).

For greater values of the degree *m* of *f*(*x*), sequential implementations should be considered. Several interleaved multipliers have been implemented (Table 13.2).

In the case of the squaring operation, combinational circuits can be used, even for great valued of *m* (Table 13.3).

Table13.1 Classic mod $f(x)$ multipliers

m	LUTs	Delay
8	37	3.2
64	2,125	5.3

Table 13.2 Interleaved mod $f(x)$ multipliers

m	FFs	LUTs	Period	Total time
8	32	34	1.48	13.3
64	201	207	1.80	117.0
163	500	504	1.60	262.4
233	711	714	1.88	439.9

Table 13.3 Mod $f(x)$ squaring (square and reduce)

m	LUTs	Delay
8	8	0.7
64	129	0.7
163	163	0.7
233	153	0.7

Table 13.4 Mod $f(x)$ dividers

m	FFs	LUTs	Period	Total time
8	43	34	2.41	41.0
64	273	157	2.63	339.3
163	673	370	2.96	967.9
233	953	510	2.98	1391.7

Several sequential mod $f(x)$ dividers (Sect. 13.4) have been implemented (Table 13.4).

13.6 Exercises

1. Generate the VHDL model of a reducer modulo 239.
2. Generate the VHDL model of a multiplier modulo $m = 2^{192} - 2^{64} - 1$.
3. Design an interleaved modulo m multiplier using the stored-carry principle.
4. Design a modulo p divider based on Fermat's little theorem.
5. Generate the VHDL model of a multiplier over $Z_2[x]/f(x)$ where $f(x) = x^8 + x^4 + x^3 + x + 1$.

6. Generate the VHDL model of a divider over $Z_2[x]/f(x)$ where $f(x) = x^8 + x^4 + x^3 + x + 1$.
7. Design a squarer over $Z_2[x]/f(x)$ where $f(x) = x^8 + x^4 + x^3 + x + 1$.

References

1. Rodríguez-Henríquez F, Saqib N, Díaz-Pérez A, Koç ÇK (2006) Cryptographic algorithms on reconfigurable hardware. Springer, Heidelberg
2. Deschamps JP, Imaña JL, Sutter G (2009) Hardware implementation of finite-field arithmetic. McGraw-Hill, New York
3. Montgomery PL (1985) Modular multiplication without trial division. Math Comput 44: 519–521
4. Sutter G, Deschamps JP, Imaña JL (2011) Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation. IEEE Trans Industr Electron 58(7):3101–3109
5. Hankerson D, Menezes A, Vanstone S (2004) Guide to elliptic curve cryptography. Springer, Heidelberg
6. Menezes A, van Oorschot PC, Vanstone S (1996) Handbook of applied cryptography. CRC Press, Boca Raton
7. Knuth DE (1981) The art of computer programming, Seminumerical algorithmsvol, vol 2. Addison-Wesley, Reading
8. Deschamps JP, Bioul G, Sutter G (2006) Synthesis of arithmetic circuits. Wiley, New York
9. Meurice de Dormale G, Bulens Ph, Quisquater JJ (2004) Efficient modular division implementation. Lect Notes Comp Sci 3203:231–240
10. Takagi N (1998) A VLSI algorithm for modular division based on the binary GCD algorithm. IEICE Trans Fundam Electron Commun Comp Sci 81-A(5):724–728
11. Kim C, Hong C (2002) High-speed division architecture for GF(2^m). Electron Lett 38: 835–836