Hardware Implementation of the Binary Method for Exponentiation in GF(2^m)

Mario Alberto García Martínez
Instituto Tecnológico de Orizaba
Av. ITO, Col. Zapata, Orizaba Ver. 94300
marioag@prodigy.net.mx

Guillermo Morales Luna
Sección de Computación, CINVESTAV, IPN
Av. IPN 2508
07300, México D.F.
gmorales@cs.cinvestav.mx

Francisco Rodríguez Henríquez
Sección de Computación, CINVESTAV, IPN
Av. IPN 2508
07300, México D.F.
frodriguez@cs.cinvestav.mx

Abstract

Exponentiation in finite or Galois fields, GF(2^m), is a basic operation for several algorithms in areas such as cryptography, error-correction codes and digital signal processing. Nevertheless the involved calculations are very time consuming, especially when they are performed by software. Due to performance and security reasons, it is often more convenient to implement cryptographic algorithms by hardware. In order to overcome the well-known drawback of little or inexistent flexibility associated to traditional Application Specific Integrated Circuits (ASIC) solutions, we propose an architecture using Field Programmable Gate Arrays (FPGA). A cheap but still flexible modular exponentiation can be implemented using these devices. We provide the VHDL description of an architecture for exponentiation in GF(2^m) based in the square-and-multiply method, called binary method, using two multipliers in parallel previously developed by ourselves. Our structure, compared with other designs reported earlier, introduces an important saving in hardware resources.

I. Introduction

Exponentiation operation in finite or Galois fields is fundamental in several cryptographic algorithms of generalized use at the moment, such as the Diffie-Helman protocol for key exchange [1], El-Gamal algorithm for digital signatures [2] or the RSA cryptosystem [7]. The calculations required for such algorithms imply a high consumption of processing time, especially when they are implemented by software. A conventional method for software exponentiation in finite fields makes use of the so called look-up tables. But this method cannot be efficiently implemented in VLSI circuits. For security and performance reasons, it is often more advantageous to develop cryptographic algorithms in hardware.

In recent years, several hardware algorithms and architectures have been proposed for computing exponentiation in GF(2^m) [8]-[12]. Some of them have been implemented in VLSI circuits and others in FPGA´s, taking advantage of the inherent characteristic of programmability of such devices. Nevertheless, most of those implementations operate with small values of word length since, for values greater to 8 bits the hardware requirements grow considerably

In order to avoid the little flexibility inherent to traditional ASIC designs, we propose an architecture especially tailored for FPGA implementations. Using such devices an economic and flexible structure for exponentiation can be obtained.

The square and multiply method for exponentiation, known as the *binary method* [3], is generally an accepted technique to compute exponentiation in finite fields.

Let us recall [6] that the multiplicative group of the Galois field $GF(2^m)$ is cyclic with 2^m-1 elements. Indeed a generator is an irreducible polynomial of degree m, with coefficients in the prime field $\mathbf{Z}_2$. Thus for any non-zero element M in $GF(2^m)$ and any integer exponent e, we have

$$M^e = M^{e \bmod (2^m - 1)}$$

Moreover, the product in $GF(2^m)$ is realized as the polynomial multiplication reduced modulo a chosen irreducible polynomial of degree m. Hence, the modular exponentiation operation gives, for any given non-zero element M in $GF(2^m)$ and any integer exponent e, with $e<2^m$, the element

$$R = M^{e} = \underbrace{M \cdots M}_{e \text{ times}} \bmod G \qquad (1)$$

where g is the irreducible polynomial representing $GF(2^m)$.

If we write the exponent e in base 2,

$$e = e_{m-1}e_{m-2} \ldots e_1 e_0$$

then we can express R as

$$R = \prod_{e^i=1} M^{2^i} \qquad (2)$$

There exist two main binary algorithms to evaluate the right hand side term in eq. (2): *MSB-first* and *LSB-first (Most and Least Significant Bit*, respectively). Each one of them depends on which is the first bit of the exponent to be scanned by the procedure. In this work we have used the LSB-approach because it is possible to obtain a parallelized version of it in order to compute the exponentiation operation defined in eq. (1). The exponentiation operation is computed using two multipliers designed previously in [4] as main building blocks.

In the remaining sections of this paper, we outline the LSB-first algorithm and we give two examples of it, then we introduce our proposed exponentiator algorithm and its corresponding hardware implementation. Thereafter we provide a complexity analysis of our design and finally we formulate some conclusions and comparisons with other designs previously reported in the literature.

II. Exponentiation Algorithm

Let $GF(2^m)$ be the Galois field over $GF(2) = \{0,1\}$ of order m. Let $g(x)$ be an m-degree irreducible polynomial generating the field $GF(2^m)$. Let α be a root of $g(x)$. Then the powers 1, α, α^2,. . .,α^{m-1} form a canonical basis in $GF(2^m)$.

Let M be an arbitrary element in $GF(2^m)$ expressed in canonical basis as:
and let

$$M = \sum_{i=0}^{m-1} m_i \alpha^i$$

$$e = \sum_{i=0}^{m-1} e_i 2^i = (e_{m-1}, e_{m-2}, \ldots, e_1, e_0); e_i = \{0,1\}$$

be an m-bit integer, $1 \le e \le 2^m$-1. It is enough to consider exponents of this size, since the multiplicative structure of $GF(2^m)$ is cyclic with 2^m-1 elements.

The power $R = M^e$, modulo the irreducible polynomial $G=g(x)$, is also in $GF(2^m)$ and, by using the binary exponentiation method [3], can be computed as shown in the next algorithm:

Algorithm: *(Exponentiation LSB-first)*

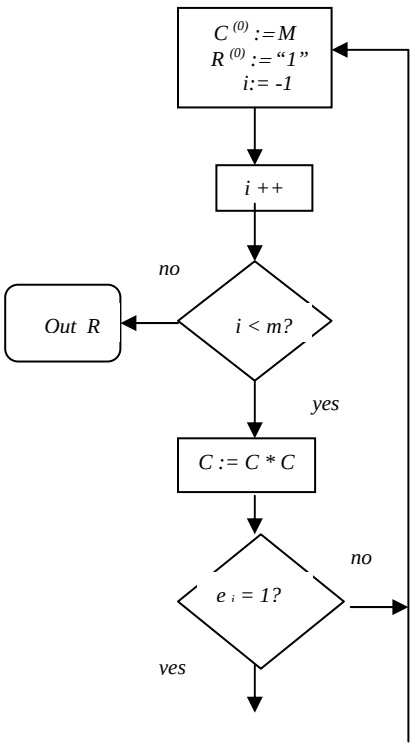
Input: *M, e, G*
Output: $R = M^e \pmod G$
==========
1.- $C := M; R := 1$;
2.- **for** $i := 0$ **to** $n-1$ **do**
2.*a*).- **if** $e_i := 1$ **then** $R := R*C \pmod G$
2.*b*).- $C := C*C \pmod G$
end for ;
3.- **return** *R*;
==========

Example
e = 1 1 1 1 1 0 1 0 = 250

e	***Step 2.a (R)***	***Step 2.b (C)***
0	1	$(M)^2 = M^2$
1	$1*M^2 = M^2$	$(M^2)^2 = M^4$
0	M^2	$(M^4)^2 = M^8$
1	$M^2 * M^8 = M^{10}$	$(M^8)^2 = M^{16}$
1	$M^{10} * M^{16} = M^{26}$	$(M^{16})^2 = M^{32}$
1	$M^{26} * M^{32} = M^{58}$	$(M^{32})^2 = M^{64}$
1	$M^{58} * M^{64} = M^{122}$	$(M^{64})^2 = M^{128}$
1	$M^{122} * M^{128} = M^{250}$	$(M^{128})^2 = M^{256}$

III. Modular Exponentiator Architecture

The flow chart of the binary algorithm is shown in figure1.



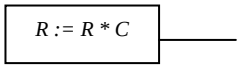


Figure 1. Flow chart of binary method

In figure 2, we propose a parallel architecture for exponentiation in $GF(2^m)$ based on that method.

As we can observe in the algorithm shown in figure 1, registers R and C are loaded initially with "1" and M respectively. Then, with each clock cycle, one multiplier will operate to calculate $C*C$; and, depending of e_i value, other multiplier will work to make the $R*C$ product whenever $e_i = 1$. The final result will be obtained in register R when the e_{m-1} bit be examined.

As we mentioned before, the algorithm described in figure 1 can be efficiently implemented in hardware by using the architecture shown in figure 2.

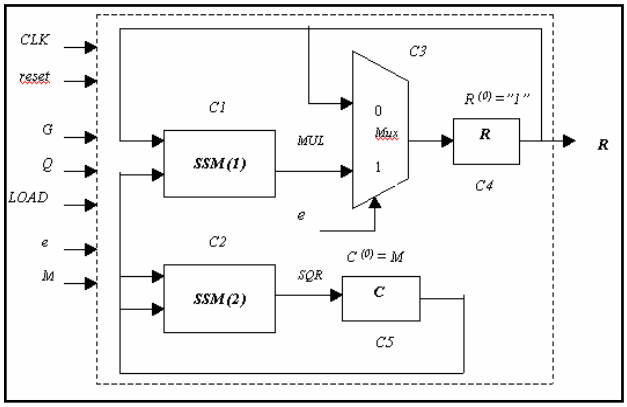


Figure 2. Exponentiator Architecture

That structure requires $2m$ multiplications and sm clock cycles for computing a modular exponentiation. The exponent is a m-bit word and s is the number of clock cycles required to calculate the multiplication.

We use a systolic and serial multiplier (*SSM*) [4] with time delay of $3m$-1 clock cycles as the main building block of the exponentiator design. The architecture uses two *SSM* that work in parallel form, two registers R and C of m-bits and a multiplexer that selects the corresponding operation of *SMM(1)*.

The signals and block labels shown in figure 2 stand for:

SSM=*Serial and Systolic Multiplier*
Mux=*Multiplexer*
R, C=*Registers with parallel load*
G = Modulus, **Q** = Control signal

This circuit is specified directly in VHDL. Its code is included in Appendix A.

IV. Design Comparisons

Table 1 shows the hardware requirements of our architecture and its comparison with [5] and [8].

Table 1. Comparison table

	[5]	[8]	Here
Multiplications	2(m-1)	2(m-1)	$2m$
Multipliers	m-1	2(m-1)	2
Squarers	m-1	------	--
Registers	----	------	2
Multiplexers	M	------	1
Time delay	m^2-m/2+1	$2m^2$ +2	$3m^2$ -m

As we can observe, our time delays tend to be 50% greater than those in [8] but we require a constant number of multipliers while in [8] this number grows linearly with m.

We have used the tools of program ISE 4.1i from Xilinx to describe the circuits with VHDL (*VHSIC-Hardware Description Language*, VHSIC in turn is *Very High Scale Integration Circuit*s) and also they will be used in the synthesis process and implementation in the FPGA. We have a prototype card whose device is a Virtex FPGA XSV300 from Xilinx which is integrated with 3072 CLB´s (*Configurable Logic Blocks*).

V. Conclusions

We have presented the architecture and VHDL description of a structure for finite field exponentiation that is based on the *LSB-first* version of the binary method. The architecture proposed here is a structure that operates using two multipliers in parallel and that is economic in terms of hardware requirements. If we compare our structure with other designs, as those reported in [5] and [8], we can see that some hardware complexity may be saved by using our design. Later we will have to physically implement it in a FPGA, which will be used as a basic element for specific algorithms in cryptography and error correction codes.

Acknowledgements: We thank the suggestions of anonymous referees who helped us to improve the final presentation and contents of this paper.

VI. References

[1] Diffie,W. and Hellman, M.E.: New directions in cryptography, *IEEE Trans. Inf. Theory*,1976, IT-22,(6), pp.644-654.

[2] Data Encryption Standard (DES*): Federal Information Processing Standards*. Publication 46-2, December 1993.

[3] Knuth, D. E.: *The art of computer programming, vol. II: Seminumerical algorithms*, (Addison-Wesley, MA, 1969).

[4] García-Martínez, M. A. and Morales-Luna G.: FPGA implementation to divide and multiply in *GF(2^m)*. In Mullen G. L., Stichtenoth H., Tapia-Recillas, H. (ed.s), *Finite Fields with Applications to Coding Theory, Cryptography and Related Areas*. Springer-Verlag, 2002.

[5] Jain, S. K., Song, L. and Parhi, K. K.: Efficient semisystolic architectures for finite field arithmetic. *IEEE Trans. on VLSI*, 1998, vol.6 (1), pp. 101-113.

[6] Lidl, R. and Niederreiter, H. : *Introduction to Finite Fields and their Applications*, Cambridge University Press, 1986.

[7] PKCS # 1, v2.1: RSA Cryptography Standard. RSA Laboratories, June 2002.

[8] Wang, C.L.: Bit-Level Systolic Array for Fast Exponentiation in GF(2^m), *IEEE Trans. on Comp.*,1994, vol.43(7), pp. 838-841.

[9] Kovac M. and Ranganathan N. : ACE: A VLSI Chip for Galois Field GF(2^m) Based Exponentiation. *IEEE Trans. on Circuits and Systems-II*, 1996, vol. 43, no. 4, pp. 289-297.

[10] Blum T. and Paar C. : Montgomery Modular Exponentiation on Reconfigurable Hardware. *14th IEEE Symposium on Computer Arithmetic*. 1999, Adelaide, Australia.

[11] Paar C. and Soria-Rodriguez P. : Fast Arithmetic for Public-Key Algorithms in Galois Fields with Composite Exponents. *IEEE Trans. on Computers*, 1999, vol. 48, no. 10, pp. 1025-1034.

[12] Lee K-J. And Yoo K-Y. : Linear systolic multiplier/squarer for fast exponentiation. *Information Processing Letters*. 2000,vol.76, pp. 105-111.

A. Exponentiator VHDL Description

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity exponentiator is                    --Main entity
   Port ( M,e,G,Q : in std_logic;
          R : out std_logic;               --R=M^e (mod G)
          CLK,reset : in std_logic);
end exponentiator;

architecture Behavioral of exponentiator is

   component SSM port(CLK,reset,G,Q,A,B:in std_logic;
                      C:out std_logic);
   end component;
   component Mux21 port(X,Y,e: in std_logic;
                        Z: out std_logic);
   end component;
   component REG port(CLK,reset,LOAD,
                      In_reg:in std_logic;
                     Out_reg: out std_logic);
   end component;

signal S_reg_C,S_reg_R,S_SQR,S_MUL,S_In_R:
       std_logic;
begin
C1:SMM port map
     (CLK=>CLK,reset=>reset,G=>G,Q=>Q,
     A=>S_reg_R,B=>S_reg_R,C=>S_MUL);
 C2: SMM port map
     (CLK=>CLK,reset=>reset,G=>G,Q=>Q,
     A=>S_reg_C,B=>S_reg_C,C=>S_SQR);
 C3: Mux21 port map(X=>S_reg_R,Y=>S_MUL,Z=>In_R);
 C4: REG port map
     (CLK=>CLK,reset=>reset,LOAD=>LOAD,
     In_reg=>S_In_R,Out_reg=>S_reg_R);
C5: REG port map
     (CLK=>CLK,reset=>reset,LOAD=>LOAD,
     In_reg=>S_In_R,Out_reg=>S_reg_R);

end Behavioral;
=========================================

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity REG is                              --Register entity
   Port ( CLK,set,reset,In_LOAD : in std_logic;
          Din_p: in std_logic_vector(m-1 to 0);
          Din_s: in std_logic;
          Dout: out std_logic);
end REG;

architecture behavior of REG is
signal Q_temp: std_logic_vector(m-1 down to 0);
begin
    Dout<="0";
    comb:process(In_LOAD,Din_s)
    begin
          if(In_LOAD="1") then Q_temp<=Din_p;end if;
    end process comb;

   state: process(CLK,set,reset)
   begin
      if(reset="1") then Q_temp<=(others=>"0");end if;
      if(set="1") then Q_temp<= (others=>"1");
      elsif(CLK'event and CLK="1") then
           Q_temp:=Din_p & Q_temp(m-1 down to 1);
      end if;
      Dout<= Q_temp(0);
    end process state;
end behavior;
=========================================

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Mux21 is                  --Mux entity
   Port ( X,Y,e : in std_logic;  Z: out std_logic);
end Mux21;

architecture behavior of mux21 is
begin
          Z<=Y when (e="1") else X;
```

```
end behavior;
=========================================
```
Note: The VHDL description of SSM can be consulted in [4]