Bulletin of the *Transilvania* University of Braşov • Vol. 8 (57) No. 1 - 2015
Series I: Engineering Sciences

A METHOD TO HANDLE BCH(n,k,t) ALGORITHM OVER LARGE GF(n) IN PRACTICAL HARDWARE IMPLEMENTATIONS

A. STANCIU[1] T. CIOCOIU[1] F. MOLDOVEANU[1]

Abstract: *This paper presents an approach to handle with elements from GF(2^n), in hardware implementation with minimum costs of area. The method is described by exemplifying with minimum costs of area. The method is described by exemplifying the practical implementation of the BCH(n,k,t) scheme over GF(2^n), where n is large (n > 6), on reconfigurable FPGA hardware with minimum costs of area. There are many papers in the open literature which presents hardware implementations of algorithms over GF(2^n) but none of them addresses the problem of hardware resources employed. There are many situations in which an area optimized implementation is more suitable than a speed optimized implementation.*

Key words: *Galois Field, BCH, polynomial, hardware implementation.*

1. Introduction

There are many applications in which a BCH (Bose Chaudhuri Hochquenghem) hardware implementation is more suitable than a software implementation. One example it is the case in which we have a secret key generated with the help of silicon physical unclonable functions based on process variations which appear during the physical execution of an integrated circuit [5]. The method of generating the secret key is out of scope in this paper. The secret key may be used to uniquely identify the integrated circuits using two phases: 1) the enrollment phase (Figures 1) the authentication phase (Figure 2).

The phase 1 illustrated in Figure 1, is used only once for an integrated circuit. It generates the 128-bits length identifier based on ring oscillators. From this 128-bits length sequence a helper data is generated, which will be used each time for the circuit authentication. This stage corresponds to the encoding stage BCH.

The authentication stage, resumed in Figure 2, involves the reconstruction of the identification sequence. The 128-bits length sequence is regenerated using the same ring oscillators as in the enrollment phase. The new 128-bits length sequence is corrected, in case it contains a number of accepted errors (maximum 10), using the helper data and the BCH decoding algorithm.

The field GF(2^n) is defined by a set of 2^n unique elements that is closed under both addition and multiplication, in which every non-zero element has a multiplicative inverse and every element has an additive

[1] Dept. of Automation and Information Technology, *Transilvania* University of Braşov.

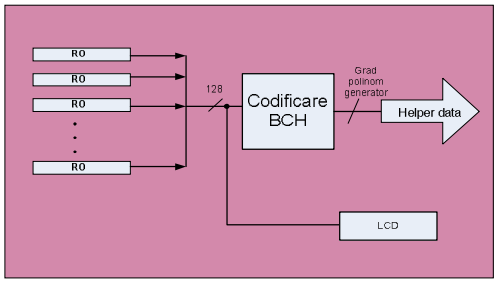

Fig. 1. *The enrollement phase*

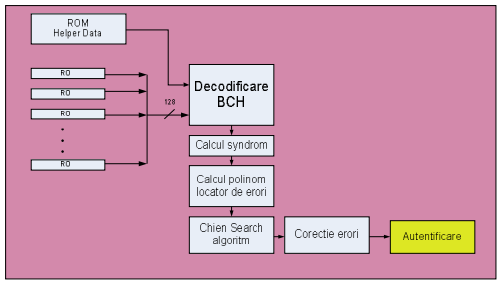

Fig. 2. *The authentication phase*

inverse. As with any field, addition and multiplication are associative, distributive and commutative [4]. The field GF(2^n) is defined over an irreducible polynomial of degree n with coefficients in GF(2^n). The primitive polynomial has a root α, named primitive root where $\alpha^{2n-1} - 1 = 1$ and α^i, where $i < 2 - 1$ generates a different element from GF(2^n). The Galois field GF(2^n) may be represented by the set of all polynomials of degree at most n-1, with binary coefficients, as can be seen in Table 1. The first step is to consider which representation of the elements would be used in the implementation: the representation as the power of α or the representation as the binary vectors.

Table 1

Examples of Galois Field Elements, n = 8

Elements	Polinomyals	Binary vectors
α^{127}	$\alpha^6 + \alpha^5 + \alpha^2 + \alpha$	01100110
α^{128}	$\alpha^7 + \alpha^6 + \alpha^3 + \alpha^2$	11001100
α^{130}	$\alpha^4 + \alpha^2 + \alpha + 1$	00010111
α^{131}	$\alpha^5 + \alpha^3 + \alpha^2 + 1$	00101110

In order to analyze this challenge we exemplify the addition, substraction, multiplication and division in GF(2^4), considering two elements α^{10} = 0111 and α^{11} = 1110.

Addition is the same as substraction and is easily implemented using XOR and operands in the form of binary vectors $\alpha^{10} + \alpha^{11} = 0111 \; xor \; 1110 = 1001$; if we search through GF(2^4) elements we find $\alpha^{14} = 1001$.

It is obvius that for addition/substraction is more convenient to represent the elements in the binary vector form.

Multiplication

May be done using power of α format:

$$\alpha^{10} \cdot \alpha^{11} = \alpha^{21} = \alpha^{15} \cdot \alpha^6 = 1 \cdot \alpha^6 = \alpha^6 = 1100. \quad (1)$$

May be done considering binary vector form:

$$(0111 \cdot 1110) \bmod 10011 = 101010 \bmod 10011 = 1100 = \alpha.$$

The multiplication is less expensive when we consider the power of α format because we can use a dedicated multiplier for natural values and a substraction of $2^4 - 1$.

Division in GF(2^4) is a multiplication between the divident and inverse of the divisor. The inverse may be computed using the extended euclidian algorithm:

$$\frac{\alpha^{11}}{\alpha^{10}} = \alpha^{11} \cdot (\alpha^{10})^{-1} = \alpha^{11} \cdot \alpha^5 = \alpha^{16} = \alpha = 0010.$$

We believe that an approach which combines the usage of the two representation form is the most suitable for an implementation optimized in terms of costs of area. In the case of BCH(128,10) or BCH(256, 25) we can store the Galois

elements in memories where the address represents the power of α and the value at the address represents the binary vector. In cases where n is larger, such as elliptic curve crypotgraphy with n = 163, the values are not stored in memories, they are generated immediately when they are involved in computations.

2. BCH Algorithm

2.1. Coding

- Choose a primitive polynomial of degree n, and construct GF(2^n);
- Find the minimal polynomial $m_i(x)$ of α^i for $i = 1,2,..2*t$;
- Obtain the generator polynomial $g(x)$ which is the least common multiple of minimal polynomials;
- Determine the degree of the generator polynomial;
- Translate the q length message that we want to encode in a polynomial form of degree q. Add in the right part of the polynomial form a number of zeros equals with the degree of the generator polynomial;
- The previously obtained polynomial is divided by generator polynomial;
- The remainder of this division;
- represents the helper data which will be used for error correction and detection [1].

2.2. Decoding

There are many algorithms for decoding BCH codes. The most ones follow this general outline:

- Calculate the syndromes m_j for the received vector;
- Determine the number of errors t and the error locator polynomial from the syndromes;
- Calculate the roots of the error location polynomial to find the error locations;
- Calculate the error values at those error locations;
- Correct the errors [1].

3. Generating the Elements of Galois Field

The first step is to generate the elements of Galois Group using the chosen primitive polynomial. Binary representation of the elements are stored in the memory. It is easy to generate these elements in hardware. The multiplication of α element represents a shifting operation of the previous value. If the MSB bit is shifted to the n+1 position, the primitive polynomial will be subtracted from the result.

Generally, a large n in the case of BCH code is 7 or 8 which means that there are 2^7 or 2^8 elements in Galois Field that can be stored in block memories from reconfigurable hardware.

In cases of larger n such as n = 163, used in elliptic curve cryptosystems we can generate this elements immediately, when we needed in computations. This it will take some clock cycles, but we still have a highest frequency of design due to the simplicity of operations.

4. Computing Galois Field Minimal Polynomials

In the entire algorithm we consider the use of a small restricted area for the FPGA hardware resources. In order to obtain this, we used BRAM memories to store the polynomial coefficients and the sequentiality of some parts of the implemented algorithm.

The algorithm presented below uses only one polynomial multiplier circuit and 2 BRAM memories for storing intermediate results. The size of each memory is $n * 2^n$. The i address memory will store the coefficient of x^i. The x^i coefficient is a sum of powers of α root, e.q $\alpha^{177} + \alpha^{87} + \alpha^3 + 1$. These coefficients are stored in memories as a 2^n bits length binary sequence, where the position in that value of 1 appears represents the power of α.

After the coefficients are stored, the sum

of these terms will be realized using the rules of the Galois Field. The algorithm for computing the α^i minimal polynomial, suitable for a hardware implementation with minimum cost of area is presented in Figure 3.

Exemplification is made using GF(2^4) generated by the primitive polynomial $x^4 + x + 1$. The minimal polynomial for α^3 is computed using the formula:

$$m_3(x) = (x+\alpha^3)\cdot(x+\alpha^{3^2}) \cdot(x+\alpha^{3^3})\cdot(x+\alpha^{3^4}). \quad (2)$$

The maximum power of α is 16, so we consider memories of size 8x16.

Phase 1 (Figures 4 and 5): the initialization of memories that will store the intermediate results. The *mem_minimal_polynomai_nx2^n_inst*1 contains the coefficients of the polynom x + α^3.

Algorithm for minimal polynomials

for(i=0; i<2*t-1; i++) //t is the maximum number of errors which must be corrected
//the memories are initialized
- mem_power_of_alfa_8x8 – store the power of α^i : i, i*2, i* 2^2, ..i* 2^n, where n is the Galois field order
- mem_power_of_alfa_valid_2^nx1 – store 1 if the power of alfa was already taken into consideration. The memories has the role to check if one power was already obtained.
- mem_minimal_polynomial_$nx2^n$_inst1, mem_minimal_polynomial_$nx2^n$_inst2 used for storing the intermediate coefficients of minimal polynomial
  //computing power of alfa
- for (k=1; k<n; k++)
  - new_power:= k<<n;
  - while (new_power > 2^n-1) new_power:= new_power – (2^n-1);

  //computing the minimal polynomial
- for (power=0; power<n; power++)
  - read one α power form memory
  - initialize memories for intermediate coefficients mem_minimal_polynomial_$nx2^n$_inst1, mem_minimal_polynomial_$nx2^n$_inst2,
  - for(addr=0; addr<m-1; addr++) // m represents the degree of the calculated polynomial
    - reg:=0;
    - read coefficient from address addr, coefficient[addr]
      - for (j=0; j< 2^n-1; j++)
        - if (coefficient[addr][j] == 1)
          - reg[addr+power]:=1;
        - if (addr>0)
          - read coefficient[addr-1];
          - reg:= reg XOR coefficient[addr-1];
    - write reg in memory at address addr

  //compute the coefficient using the arithmetic of the Galois field
- for (addr=0; addr<m-1; addr++)
  - read coefficient[addr];
  - new_coeficient:= 0;
  - for (i=0; i< 2^n-1; i++)
    - if (coefficient[addr][1]==1)
      - GF_data – translate I in polynomial form;
      - New_coefficient := new_coefficient XOR GF_data;
    - Write new_coefficient;

Fig. 3. *Algorithm for minimal polynomials*

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
.																
.																
.																
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 4. *Phase 1: mem_minimal_polynomial_nx2^n_inst*

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
.																
.																
.																
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 5. *Phase 1: mem_minimal_polynomial_nx2^n_inst2*

Phase 2 (Figures 6 and 7): the content of the first memory is multiplied with the polynomial $x+\alpha^6$. The result is $x^2+(\alpha^3+\alpha^6)*x+\alpha^9$. It may be observed that the polynom $x+\alpha^6$ has two coefficients: 1 and α^6. In order to obtain the coefficients of the polynomial result we proceed as following:

- Read the coefficient at the address *i*. The new value of this coefficient may be modified in two situations: 1) by the free term 2) by getting a new term as a result of the multiplication between the current coefficient and the coefficient of another term with a lower degree For these changes we do the following:

  ○ we go through the coefficient values and modify the position which are set in 1. The new values of 1 will be on the position equivalent with the old position + power of α.

- If the *i* address is higher than 0, the sum obtained previously it will be added with the 1 value of the coefficient term with one degree lower;

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
.																
.																
.																
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 6. *Phase 2: mem_minimal_ polynomial_nx2^n_inst1*

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
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	1	0	0	1	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
.																
.																
.																
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 7. *Phase 2 mem_minimal_ polynomial_nx2^n_inst2*

- For example computing the coefficient of the term with degree 1:

  ○ We read the value from the address 1 - 0000000000000001.

  ○ The coefficient *i* = 1, it will be multiply with α^9. The new value is: 0000001000000000.

  ○ We read the coefficient from the addres 0 - 0000000000001000. This will be added at the previously computed value, obtaining 0000001000001000, which is equivalent with $\alpha^3+\alpha^9$.

- The other coefficients are computed similar, the results are stored in the second instance of the memory.

Phase 3: This phase is similar with the previous phase except that now we will read the coefficients from the second instance of memory and we will store the results in the first instance of memory. During these phases we alternate the two memories for reading and writing.

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
2	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1
3	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
.																
.																
.																
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 8. *Phase4: mem_minimal_polynomial_nx2^n_inst2*

The coefficients of the minimal polynomial are stored as a sum of power of α:

$$x^4+x^3(\alpha^{12}+\alpha^9+\alpha^6+\alpha^3)+x^2(\alpha^6+\alpha^3+1)+x\alpha^3+1.$$

The next step is to apply arithmetic Galois rules in order to obtain the minimal polynomial for the α^3 term: $x^4+x^3+x^2+x+1$.

5. Generator Polynomial

The generator polynomial is computed with the formula:

$$g(x) = c.m.m.m.c\{m_1(x), m_3(x),..m_{2t-1}(x)\}. \quad (3)$$

The maximum degree is $t*n$, where t is the number of independent error which may appear and must be corrected and n give the order of GF(2^n). The minimal polynomials are co prime.

In our hardware implementation was used only one instance of multiply module. One of the operands is the previous result and the other is a minimal polynomial. The multiply operation is repeated until all the minimal polynomial were multiplied.

6. Computing the Syndrome Polynomial

If the number of maximum error which may be corrected is t we calculate $2*t$ syndrome polynomials. The syndrome polynomials are computed as a remainder from division between the message and the minimal polynomials. In Galois Fields there are different elements with the same minimal polynomials so the number of syndrome polynomials is less than $2*t$, where t is the maximum number of error that may be detected and corrected.

For example we consider:

- The message without errors: 10100110111;
- The message with errors: 10001110111;
- Consider that this code may correct maximum 2 errors, so we have 4 minimal polynomial:

$$\begin{aligned} m_1(x) &= x^4 + x + 1, \\ m_2(x) &= x^4 + x + 1, \\ m_3(x) &= x^4 + x^3 + x^2 + x + 1, \\ m_4(x) &= x^4 + x + 1. \end{aligned} \quad (4)$$

- We have three identical minimal polynomials so we will do only two division;
- We obtain the following syndrome polynomial:

$$\begin{aligned} S_1(x) &= x^3 + 1, \\ S_2(x) &= x^3 + 1, \\ S_3(x) &= x^3 + x, \\ S_4(x) &= x^3 + 1. \end{aligned} \quad (5)$$

7. Reducing Syndrome Polynomial as a Power of α Element

We need to calculate the syndrome polynomial in a point equivalent with a power of α:

$$\begin{aligned} S_1(\alpha) &= \alpha^3 + 1 = \alpha^{14}, \\ S_2(\alpha^2) &= \alpha^6 + 1 = \alpha^4\alpha^2 + 1 \\ &= \alpha^3 + \alpha^2 + 1 = \alpha^{13}, \\ S_3(\alpha^3) &= \alpha^9 + \alpha^3 = \alpha^4\alpha^4\alpha + \alpha^3 \\ &= (\alpha + 1)(\alpha + 1)\alpha + \alpha^3 \\ &= \alpha^3 + \alpha + \alpha^3 = \alpha. \end{aligned}$$

Multiplication and division were implemented in hardware using classical algorithms presented in [2] and [3].

8. Implementation Results and Conclusions

We optimized the process of implementation of the BCH(n,k,t) algorithm by: the choice of the algorithm with minimum costs in term of hardware resource usage, the use of BRAM memories for storing the polynomial coefficient powers and the sequentiality of some parts of the implemented algorithm.

A summary of the usage area, which was obtained after the synthesis, is presented in Figures 10 and 11.

Algorithm

input syndrom_polynom[n], power_of_alfa
for (i=0; i<n; i++)
  if (syndrome_polynom[i] == 1)
    new_power:= i* power_of_alfa;
    new_power_polynom:= the polynom representation for α^{new_power};
new_syndrom_polynom:= new_syndrom_polynom ^\ new_power_polynom;
search the power of α, i where α^i has the polynomial expression equivalent with new_syndrom_polynom. Store the power of α in a memory.

Fig. 9. *Reducing syndrome polynomial*

Selected Device: Virtex 4, 4vsx35ff668-10	
Number of Slices:	4404 out of 15360 28%
Number of Slice Flip Flops:	4210 out of 30720 13%
Number of 4 input LUTs:	8266 out of 30720 26%
Number of IOs:	224
Number of bonded IOBs:	19 out of 448 4%
Number of FIFO16/RAMB16s:	18 out of 192 9%
Number used as RAMB16s:	18
Number of GCLKs:	3 out of 32 9%
Number of DCM_ADVs:	1 out of 8 12%

Fig. 10. *Authentication method*

Selected Device: Virtex 4, 4vsx35ff668-10	
Number of Slices:	4587 out of 15360 29%
Number of Slice Flip Flops:	4320 out of 30720 14%
Number of 4 input LUTs:	8551 out of 30720 27%
Number of IOs:	224
Number of bonded IOBs:	7 out of 448 1%
Number of FIFO16/RAMB16s:	12 out of 192 6%
Number used as RAMB16s:	12
Number of GCLKs:	3 out of 32 9%
Number of DCM_ADVs:	1 out of 8 12%

Fig. 11. *Enrollment method*

The test results of the BCH(*n,k,t*) were realized on a device from family VIRTEX 4 FPGA-XC4VSX35 and on a device from family Spartan 3E-XC3S500E device. The authentication method uses 78% of the resources on a low FPGA chip as XC3S500E, which makes it impossible to be used on such low-FPGAs. Instead good results are obtained for Virtex 4 family FPGA.

Acknowledgment

We hereby acknowledge the structural founds project PRO-DD (POS-CCE, O.2.2.1, ID 123, SMIS 2637, ctr. No 11/2009) for providing the infrastructure used in this work and the project ID137070 financed from the European Social Fund and by the Romanian Government.

References

1. Chien, R.T.: *Cyclic Decoding Procedure for the Bose-Chaudhuri-Hocquenghem Codes*. In: IEEE Trans. Inf. Thery **IT-10** (1964) No. 4, p. 357-363.
2. Forney, D.: *MIT Principles of Digital Communication II*. Chapter 7, Video Lectures and Notes.
3. Freudenberger, J., Spinner, J.: *Mixed Serial/Parallel Hardware Implementation of the Berlekamp Massey Algorithm for BCH Decoding in Flash Controller Applications*. In: Signal, Systems, and Electronics (ISSSE), International Symposium on, 3-5 Oct, 2012, IEEE, doi: 10.1109/ ISSSE.2012.6374329.
4. Guajardo, J., et al.: *Efficient Hardware Implementation of Finite Fields with Applications to Cryptography*. In: Acta Applicandae Mathematica **96** (2006) Issue 1-3, p. 75-118.
5. Maes, R., Herrewege, A., Verbauwhede, I.: *PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator, Cryptographic Hardware and Embedded Systems*. CHES 2012, Lecture Notes in Computer Science **7428** (2012), p. 302-319.