# VHDL and Cyclic Corrector Codes

France Mendez

CNET Grenoble (France Telecom)

## Abstract

Cyclic corrector codes, or "block codes", are often used in telecommunications systems. To facilitate the design of coding/decoding circuits using this type of code, we described the usual algorithms using VHDL. The mathematics used for these codes requires special packages to be created describing the functions on Galois Fields. The synthesis of components performing these functions provided the necessary information for choosing the model of architecture.

## Introduction

When studying cyclic corrector codes, we were confronted with multiple notations and numerous algorithms. The model written in VHDL allowed us to compare the different algorithms using a unique notation, and the simulation of VHDL descriptions enabled various processes to be explored. With these descriptions, we have been able to evaluate the performances of each decoding method. Using the results of the synthesis of algebraic basic operators, it has also been possible to help designers choose the optimum architecture.

All the algorithms used for decoding linear block codes require mathematics based on the Finite Fields. To understand the theory of the cyclic corrector codes, it is necessary to make a short presentation of the Finite, or Galois, Fields. First we will explain how VHDL makes it possible to synthesize operations over the Finite Fields. After we will describe how to model decoders using VHDL. Finally we will show how the modelling can help in the choice of the most appropriate architecture.

## 1. Presentation of Finite Fields

### 1.1. What are Finite Fields

The Galois Fields are vectorial fields with a finite number of elements. Each element A of the fields can be represented as a **vector** on the primitive element named "α" :

$$A = \sum_{j=0}^{M-1} a_j \alpha^j$$

where α is a root of a monic polynomial P :

$$P(x) = \sum_{j=0}^{M-1} p_j x^j \quad and \quad P(\alpha) = 0$$

As binary fields are normally used, $a_j$ and $p_j$ are 1 or 0. All the non-zero elements of the fields can also be represented as a power of α.

For example in the field GF($2^3$), formed with the primitive polynomial $x^3 + x + 1$, $\alpha^3 = \alpha + 1$ and the elements can thus have three representations :

| logarithmic | polynomial | "vector" |
|---|---|---|
| 1 | 1 | 0 0 1 |
| $\alpha^1$ | $\alpha$ | 0 1 0 |
| $\alpha^2$ | $\alpha^2$ | 1 0 0 |
| $\alpha^3$ | $\alpha + 1$ | 0 1 1 |
| $\alpha^4$ | $\alpha^2 + \alpha$ | 1 1 0 |
| $\alpha^5$ | $\alpha^2 + \alpha + 1$ | 1 1 1 |

|  |  |  |
|---|---|---|
| $\alpha^6$ | $\alpha^2 \quad +1$ | 1 0 1 |
| $\alpha^7$ | $1$ | 0 0 1 |

In a binary field the fundamental property of $\alpha$ is that $\alpha^{2^M-1}=1$. A binary Galois Field has only ($2^M$-2) non-zero elements. The zero element can not normally be represented as a power of $\alpha$. To have a complete logarithm-representation, we represent arbitrarily the vector ZERO with $\alpha^{2^M-1}$. This necessitates testing the zero value in each function.

We will now see how VHDL transforms this complicated mathematical theory into synthesizable components.

## 1.2. The VHDL modelling for Finite Fields

To describe the algebra in Galois Fields using VHDL, a package of new types is defined first, following by other packages for the algebraic functions.

As explained, there are three representations of each element, but the polynomial and the vector can be described using the same type called GFPM (subtype of bit vector). Another type named GF_ALPHA (subtype of integer) is used for the logarithm. Conversion functions allow translation between the representations.

```
library IEEE, ALGEBRE_IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
package GF2_TYPE is
-- Declaration of the order of the Field M
  constant M : NATURAL := 8;
-- declaration of INFINITE
  constant INFINITE : NATURAL := 2**M-1;
-- Field element as polynomial
   subtype GFPM is UNSIGNED(M-1 downto
0);
-- Field element as power of Alpha
   subtype GF_ALPHA is INTEGER
range 0 to INFINITE;
```

Two other types were defined :
- MOT to represent a vector of GF_ALPHA, to represent the polynomials with coefficients in GF($2^M$).
- GF_ALPHABIT : the representation of a GF_ALPHA in bit vector, i.e. the representation of an integer in bit vector.

For example, we often use the constant V_INFINITE, which is a vector with all the components at '1'. This constant is equivalent to the zero value in vectorial representation.

In another package the basic operations over Finite Fields are described : addition, product, division, inversion, exponentiation, and multiplication by an integer. In a binary field the subtraction does not exist. Addition is described for the GFPM'type and the GF_ALPHA'type.

```
library ALGEBRE_IEEE, IEEE;
use ALGEBRE_IEEE.GF2_TYPE.all;
use IEEE.std_logic_1164.all;
package GF2_SYNTH is
-- Definition of addition
-- of finite field elements
-- (polynomial representation)
        function PLUS_GFPM(OPA, OPB: in
GFPM)
                        return GFPM;
-- Definition of addition
-- of finite field elements
-- (power of ALPHA as UNSIGNED)
   function PLUS_GFLOG(OPA, OPB:
                  in GF_ALPHABIT)
                        return GF_ALPHABIT;
-- Definition of addition
-- of finite field elements
-- (power of ALPHA as INTEGER)
   function PLUS_GFLOG(OPK1, OPK2:
                    in GF_ALPHA)
                        return GF_ALPHA ;
```

As the decoding generally uses more products than additions, the other functions are only described for GF_ALPHA.

In polynomial representation, the **addition** is a simple EXCLUSIVE-OR.

```
-- Definition of addition in GF(2**M)
-- polynomial representation
        function PLUS_GFPM(OPA, OPB: in
GFPM)
                         return GFPM is
   begin
    return (OPA xor OPB);
   end PLUS_GFPM;
```

With the GF_ALPHA there are two possibilities : either convert A and B into GFPM, use an EXCLUSIVE-OR, and convert the result into GF_ALPHA; or use the ZECH'logarithm to perform the addition.

$$\alpha^{Z(x)} = \alpha^x + 1$$

Then :

$$\alpha^x + \alpha^y = \alpha^{(x+Z((y-x)\bmod(2^M-1)))\bmod(2^M-1)} =$$
$$\alpha^{(y+Z((x-y)\bmod(2^M-1)))\bmod(2^M-1)}$$

The first method uses three tables, the second only one, but the modulo'addition and modulo'subtraction are more complicated. The logic synthesis showed that the second method is more interesting when M is higher than 5 (the usual case is 8).

As an example, the function PLUS_GFLOG, which performs the addition of two elements of GF($2^M$) in logarithmic representation.

```
function PLUS_GFLOG(OPA, OPB:
             in GF_ALPHABIT)
                         return GF_ALPHABIT is
  variable ADDRESS, VALUE, A, B :
                      GF_ALPHABIT;
  begin
  --case of equality of the operands
   if OPA = OPB then
     return V_INFINITE;
  --case of null operand
   elsif OPA = V_INFINITE then
      return OPB;
   elsif OPB = V_INFINITE then
      return OPA;
  -- normal case
   elsif M < 6 then
     A := TO_GFPM(OPA);
     B := TO_GFPM(OPB);
     return TO_ALPHABIT(A xor B);
   else
    ADDRESS := SUB_MOD(OPB, OPA);
    VALUE := TO_ZECH(ADDRESS);
    return PLUS_MOD(VALUE, OPA);
   end if;
  end PLUS_GFLOG;
```

The **product** of GF_ALPHA also uses the addition modulo ($2^M$-1) :

$$C = \alpha^a \times \alpha^b = \alpha^{(a+b)\bmod(2^M-1)}$$

The **exponentiation** is like the product :

$$C = A^n = (\alpha^a)^n = \alpha^{(a\times n)\bmod(2^M-1)}$$

For the decoding processes, only **square** and **cube are** needed.

A special case is the **inversion**, i.e. when n = -1. However, we must be sure that A is non-zero if n is negative.

$$A^{-1} = \alpha^{2^M-1-a}$$

For the **division** :

$$C = A / B = \alpha^{(a-b)\bmod(2^M-1)} \quad \text{if } B \neq 0$$

For the **multiplication** of A **by an integer** n we can note that in a binary field $A + A = 0$. So if n is even the result is zero :

$$n \times A = \left| \begin{array}{ll} 0 & \text{if } n \text{ even} \\ A & \text{if } n \text{ odd} \end{array} \right.$$

All these functions can be modelled to be synthesized using a logic synthesizer.

```
-- function addition modulo (2**M-1)
-- for UNSIGNED
 function PLUS_MOD(OPA, OPB : in GFPM)
                    return GFPM;
-- function subtraction modulo (2**M-1)
-- for UNSIGNED
  function SUB_MOD(OPA, OPB : in GFPM)
                    return GFPM;
function PLUS_MOD(OPA, OPB : in GFPM)
                    return GFPM is
  variable R : GFPM;
    variable A, B : INTEGER range 0 to
INFINITE;
begin
    A := to_natural(OPA);
    B := to_natural(OPB);
    if (A + B) = 2*INFINITE then
      R := A0;
    elsif (A + B) >= INFINITE then
        R := conv_unsigned(A + B - INFINITE,
M);
    else
      R := conv_unsigned(A + B , M);
    end if;
    return R;
end PLUS_MOD;
function SUB_MOD(OPA, OPB: in GFPM)
                    return GFPM is
variable S : GFPM;
begin
    S := PLUS_MOD(OPA, not OPB);
    return S;
end SUB_MOD;
```

They sometimes use intermediate functions, like PLUS_MOD or SUB_MOD, to calculate the addition or subtraction of two integer modulo ($2^M$-1), which are described in the package.

A package has also been written containing the conversion tables usually used for various values of M. With this package we can synthesize the conversion functions between both representations.

Using all these packages, the decoding processes can now be described.

## 2. Decoding Block codes

A cyclic code is a set C verifying :

$$(c_0, \quad c_1, \quad \ldots, \quad c_{n-1}) \in \mathrm{C} \Rightarrow$$
$$(c_{n-1}, \quad c_0, \quad \ldots, \quad c_{n-2}) \in \mathrm{C}$$
$$C = (c_0, \quad c_1, \quad \ldots, \quad c_{n-1}) \approx C(x) =$$
$$c_0 + c_1x + \ldots + c_{n-1}x^{n-1}$$
$$C(x) \in \mathrm{C} \Rightarrow x \times C(x) \in \mathrm{C}$$

The $c_i$ are elements of the Finite Field GF($2^M$).

For each code there is a generator polynomial G(x). Each element C(x) of code C is a multiple of G(x) :

$$C(x) = x^r \times I(x) + R(x) = Q(x) \times G(x)$$

where I(x) is the polynomial containing the information before coding.

A Reed-Solomon code is normally named with the parameters (N, K, d):

N : the number of symbols in the transmitted coded word

d : the distance of the code

K : the number of information symbols

Usually N is $2^M$-1. A code of distance d can correct T errors if d≥ 2T+1. The code is generated using polynomial G(x) :

$$G(x) = \prod_{i=0}^{2T-1}\left(x - \alpha^i\right)$$

Several methods exist for decoding cyclic corrector codes. The most well known is the Berlekamp algorithm [2]. A combinational method can also be used[5], or a method developed by Youzhi in [9], which is like Berlekamp's, but without inversion. Another method uses a function similar to the Fourier Transform [4]. Not all the decoding methods are described in this paper. We only show how VHDL can help to describe different decoding methods, using component instantiation.

### 2.3. Modelling decoders

All the decoding processes are made of several blocks.

In the "frequency-domain" there is the direct method (Berlekamp or Youzhi) with :

1 - calculation of syndromes
2 - calculation of sigma
3 - calculation of error positions with the Chien search
4 - error evaluation
5 - correction

This case can be represented using the schema shown in figure 1.

In the case of the transform method (Blahut) the blocks are :

1 - "transformed errors"
2 - calculation of sigma
3 - calculation of last "transformed errors"
4 - correction

We can also use the "Time-domain" (Shayan) :

1 - calculation of sigma in time-domain
2 - correction

We have described each block as an entity. The architecture can be "functional" or "structural". All the entities are generic with N and T. M is a constant defined in the package GF2_TYPE.

For example :

```
library ALGEBRE_IEEE, IEEE;
use IEEE.std_logic_1164.all;
use ALGEBRE_IEEE.GF2_TYPE.all;
entity YOUZHI_RS is
   generic(T,                          -- number of
errors
        N : INTEGER);        --word length
   port(S : in MOT(2*T-1 downto 0);       --
syndrome
polynomial
       RS,                    -- reset
       H_BIT,                 -- bit clock
       H_SYMB,                -- symbol clock
           H_MOT : in STD_LOGIC;       -- word
clock
       SIGMA_PRET : out STD_LOGIC;
     -- control signal
           SIGMA, OMEGA : out MOT(T downto
0));
    -- polynomials sigma and omega
  end;
```

For the Syndrome component instantiation we use the "generate" VHDL instruction.

```
G_SYN : for J in 0 to 2*T-1 generate
```

```
   SYN : SYNDROME_RS
      generic map(J, N)
      port map(RI, RS, H_MOT, S(J));
 end generate G_SYN;
```

Using the functional architectures, we have simulated the different decoding processes. These architectures have also been used to evaluate the performances of a panel of six decoders [3]. See figure 2.

The results of the following table are based on the algebraic operators used in the algorithms.

| Decoder | Complexity | Latency |
|---|---|---|
| 1-serial | $7S_A$<br>$8S_M$<br>$1S_{INV}$<br>$1S_D$ | 2NT+(19/2)T+12 |
| 1-Parallel | $5(T+1)S_A$<br>$4(T+1)S_M$<br>$1S_{INV}$<br>$1S_D$ | 2NT+(13/2)T+7 |
| 2-Serial | $5S_A$<br>$8S_M$<br>$3S_D$ | 2NT+(19/2)T+12 |
| 2-Parallel | $(3T+5)S_A$<br>$(5T+9)S_M$<br>$(2T+3)S_D$ | 2NT+(13/2)T+7 |
| 3-Serial | $6S_A$<br>$11S_M$<br>$1S_D$ | 2NT+6T^2+(27/2)T+8 |
| 3-Parallel | $(2T+6)S_A$<br>$(5T+11)S_M$<br>$1S_D$ | 2NT+(13/2)T+7 |
| 4-Serial | $8S_A$<br>$7S_M$<br>$1S_{INV}$ | 3NT+3N+4T-2T^2+7 |
| 4-Parallel | $(N+6T+2)S_A$<br>$(5T+5)S_M$<br>$1S_{INV}$ | 2NT+2N+6T+10 |
| 5-Serial | $7S_A$<br>$7S_M$<br>$2S_D$ | 3NT+3N+4T-2T^2+7 |
| 5-Parallel | $(N+4T+2)S_A$<br>$(6T+7)S_M$<br>$2(T+1)S_D$ | 2NT+2N+6T+10 |
| 6-Serial | $4S_A$<br>$8S_M$<br>$1S_{INV}$ | 5N^2+10N+5 |
| 6-Parallel | $3S_A$<br>$7S_M$<br>$1S_{INV}$ | 2(N+1) |

- Table 1 -

| | |
|---|---|
| $S_A$ | : addition |
| $S_M$ | : product |
| $S_D$ | : division |
| $S_{INV}$ | : inversion |

## 2.4. Choice of architecture for synthesis

Using our VHDL-Synthesis tool we have evaluated components performing the functions on Galois Fields. In the usual case, the number of standard cells of each operator is :

| operator | M = 8 |
|---|---|
| log addition | 695 |
| log division | 110 |
| log product | 95 |
| log cube | 43 |
| log inversion | 13 |
| vectorial addition | 8 |

Based on Table 1, we obtain the following results in terms of number of standard cells for the different decoders :

| Decoder | T = 2 | T = 5 | T = 8 |
|---|---|---|---|
| 1-Serial | 5 748 | 5 748 | 5 748 |
| 1-Parallel | 11 688 | 23 253 | 34 818 |
| 2-Serial | 4 565 | 4 565 | 4 565 |
| 2-Parallel | 10 220 | 18 560 | 26 900 |
| 3-Serial | 5 325 | 5 325 | 5 325 |
| 3-Parallel | 9 055 | 14 650 | 20 245 |
| 4-Serial | 6 238 | 6 238 | 6 238 |
| 4-Parallel | 189 088 | 203 023 | 216 958 |
| 5-Serial | 5 750 | 5 750 | 5 750 |
| 5-Parallel | 187 335 | 198 045 | 208 755 |
| 6-Serial | 3 553 | 3 553 | 3 553 |

| | | | |
|---|---|---|---|
| 6-Parallel | 2 763 | 2 763 | 2 763 |

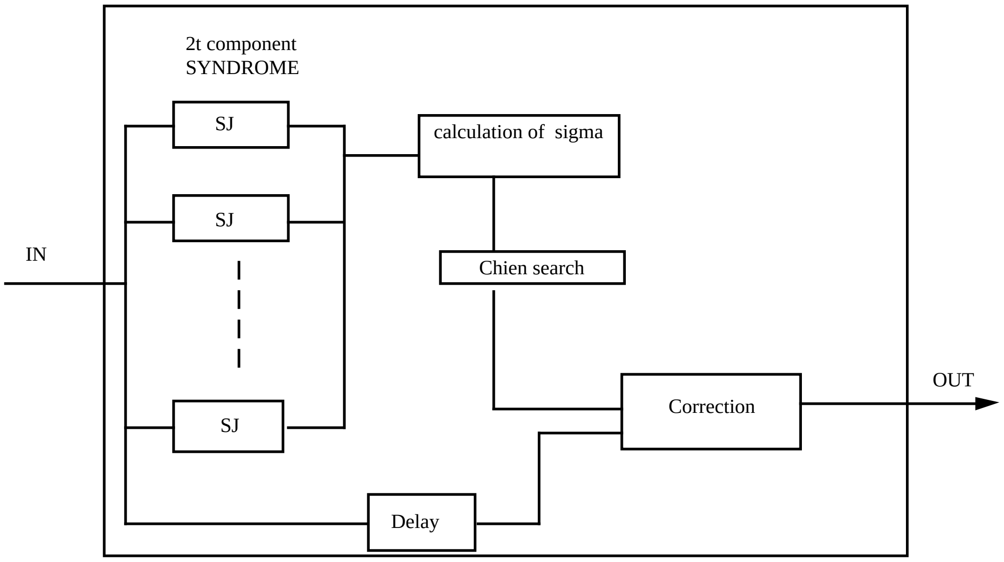

Figure 1

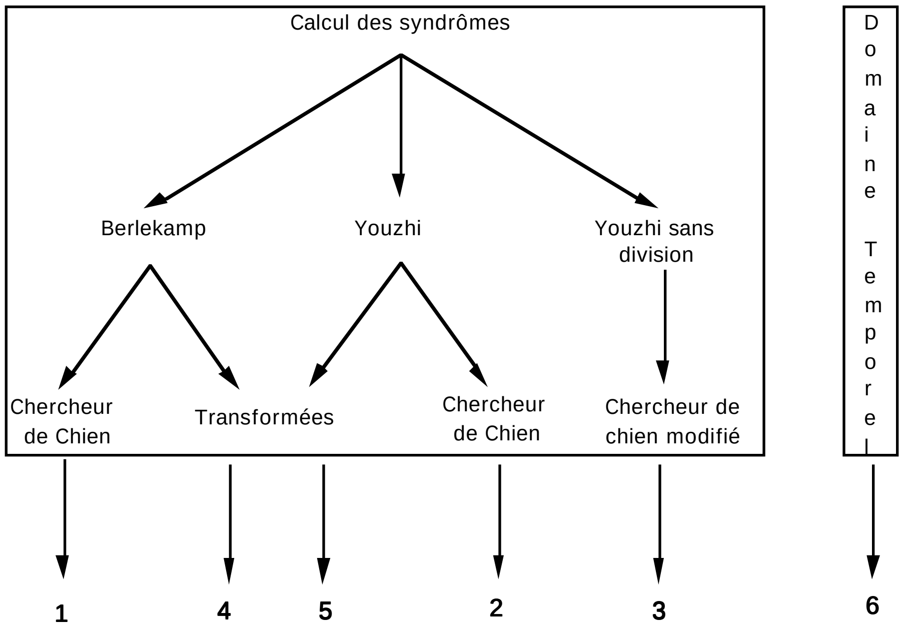

Figure 2

Table 1 also gives us approximate arrays in 0.7 $\mu$ technology :

| Decoder | T = 2 | T = 5 | T = 8 |
|---|---|---|---|
| 1-Serial | 10 mm2 | 10 mm2 | 10 mm2 |
| 1-Parallel | 20 mm2 | 41 mm2 | 61 mm2 |
| 2-Serial | 8 mm2 | 8 mm2 | 8 mm2 |
| 2-Parallel | 17 mm2 | 31 mm2 | 45 mm2 |
| 3-Serial | 9 mm2 | 9 mm2 | 9 mm2 |
| 3-Parallel | 15 mm2 | 25 mm2 | 34 mm2 |
| 4-Serial | 11 mm2 | 11 mm2 | 11 mm2 |
| 4-Parallel | 338 mm2 | 362 mm2 | 387 mm2 |
| 5-Serial | 10 mm2 | 10 mm2 | 10 mm2 |
| 5-Parallel | 334 mm2 | 352 mm2 | 370 mm2 |
| 6-Serial | 6 mm2 | 6 mm2 | 6 mm2 |

| | | | |
|---|---|---|---|
| 6-Parallel | 5 mm2 | 5 mm2 | 5 mm2 |

To choose an appropriate architecture, the designer must compare the area, but also the time performances. The logic Synthesizer provides the time period for the operators

| operator | M = 8 |
|---|---|
| log addition | 77,6 ns |
| log division | 1,1 ns |
| log product | 27,2 ns |
| log cube | 19,1 ns |
| log inversion | 17,8 ns |
| vectorial addition | 7,4 ns |

In the case T = 2, the results for the latency in clock'cycles are the following :

| Decoders | M=8 |
|---|---|
| 1-Serial/2-Serial | 1 055 |
| 1-Parallel/2-Parallel | 1 049 |
| 3-Parallel | 1 044 |
| 3-Serial | 1 083 |
| 4-Serial/5-Serial | 2 311 |
| 4-Parallel/5-Parallel | 1 558 |
| 6-Parallel | 514 |
| 6-Serial | 330 245 |

The complete results given in [3] and [7] will help the designer to choose the smallest and fastest Reed-Solomon decoder.

## Conclusion

The use of VHDL has allowed us to work in different steps. First the description of the algebra on Finite Fields; secondly the comparison between different decoding methods and the evaluation of the different performances; and thirdly the synthesis.

Finally, the very mathematical theory of cyclic corrector codes can become VLSI circuits with the help of VHDL and synthesis.

## Bibliography

[1] R. AIRIAU, J.M. BERGE, V. OLIVE, J. ROUILLARD : VHDL Du langage à la modélisation.. Presses Polytechniques et Universitaires Romandes 1990.
[2] E.R. BERLEKAMP : Algebraic Coding Theory . Mac Graw Hill 1968
[3] S. BERNARD : Architectures et performances de décodage des Codes Reed-Solomon.. Note technique CNET NT/CNS/CIT/115. 7/1993.
[4] R. BLAHUT : Digital transmission of information. Addison Wesley Publishing Co 1990
[5] M.CAND : Approche combinatoire du décodage BCH. Note technique CNET NT/CNS/CCI/103. 3/1992
[6] F. MENDEZ : Modélisation VHDL des codeurs/décodeurs BCH et Reed-Solomon.. Note technique CNET NT/CNS/CIT/113. 4/93.
[7] F. MENDEZ : Décodeurs Reed-Solomon : Aide au choix d'une architecture synthétisable en VHDL. Note technique CNET 12/93.
[8] Y. SHAYAN, T. LE-NGOC, V.K. BHARGAVA : A versatile Time-domain Reed-Solomon decoder. IEEE Journal on Selected Areas in Com. **8** N°8. (pp. 1535-1542) 10/1990.
[9] X. YOUZHI : Implementation of Berlekamp-Massey algorithm without inversion. IEE PROCEEDINGS-I **138** N°3 (pp. 138-140) 6/1991