Projet de diplôme

Implémentation de codes de Reed-Solomon sur FPGA pour communications spatiales

Code correcteur d'erreurs

Réalisé par :

Samuele Dietler
Télécommunications
Traitement et transmission de l'information

Professeur responsable :

Prof. Stephan Robert
Télécommunications

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Hes·so
Haute Ecole Spécialisée de Suisse occidentale

iICT
INSTITUTE FOR INFORMATION AND COMMUNICATION TECHNOLOGIES

Résume du problème :

La tendance dans les missions spatiales est d'intégrer de plus en plus d'instruments de mesure à bord des satellites, ce qui implique une demande en largeur de bande plus importante. Malheureusement la voie descendante (satellite-terre) est limitée et un traitement de données s'avère nécessaire. De plus l'augmentation de données transférées attire potentiel hackers, surtout lorsque ces dernières dont d'ordre potentiellement stratégique. Il devient donc nécessaire de chiffrer les données échangées entre le satellite et la terre. Un standard pour la transmission des données télémétriques existe au niveau du codage. Un codeur-décodeur de Reed – Solomon est utilisé dans ce contexte et doit pouvoir être capable de fonctionner à très haut vitesse. Pour le chiffrage, l'algorithme Advanced Encryption Algorithm (AES) sera utilisé mais il ne doit pas être implémenté dans le cadre du travail de diplôme. Le but de ce travail est de se familiariser avec le design envisagé pour la construction du prototype dans un premier temps et de mettre sur pied une réalisation fonctionnel en temps réel sur FPGA(Field Programmable Gate Array).

Cahier des charges :

1. Etude détaille du codeur-décodeur RS et des différentes possibilités qu'il est possible d'implémenter en matériel pour être capable de travailler à haute vitesse
2. Réalisation de l'implémentation en FPGA
3. Réalisation d'un système de test permettant d'évaluer les performances de la solution choisie
4. Création de la page web pour le projet
5. Etablissement d'un rapport détaillé (support papier avec tous les software développés) et remise d'un CD lors de la défense de diplôme (rapport original(Word ou Latex), mesures, implémentations, présentation PowerPoint

Préambule

Les codes de Reed – Solomon sont des codes correcteurs d'erreurs utilisés dans tous les domaines requérant des données fiables. Typiquement, dans les communications spatiales, télévision numérique et stockage de données.

Les codes de Reed – Solomon permettent de corriger des erreurs et des effacements grâce à des symboles de contrôle ajoutés après l'information. Ce document traite l'aspect mathématique du codage et du décodage de ces codes. On présentera ainsi l'implémentation des différents blocs hardware pour le codage et le décodage avec des exemples pratiques.

Table des matières

Pages

1 Introduction

De nos jours, nous vivons dans un monde où les communications jouent un rôle primordial tant par la place qu'elles occupent dans le quotidien de chacun, que par les enjeux économiques et technologiques dont elles font l'objet. Nous avons sans cesse besoin d'augmenter les débits de transmission tout en gardant ou en améliorant la qualité de ceux-ci. Mais sans un souci de fiabilité, tous les efforts d'amélioration seraient vains car cela impliquerait forcément à ce que certaines données soient retransmises. C'est dans la course au débit et à la fiabilité que les codes correcteurs entrent en jeu…

Un code correcteur d'erreur permet de corriger une ou plusieurs erreurs dans un mot-code en ajoutant aux informations des symboles redondants, autrement dit, des symboles de contrôle. Différents codes possibles existent mais dans ce document on traitera seulement les codes de Reed – Solomon car pour le moment, ils représentent le meilleur compromis entre efficacité (symboles de parité ajoutés aux informations) et complexité (difficulté de codage).

Les codes de Reed – Solomon sont des codes non binaires qui travaillent dans les « champs de Galois ». Au début, on tentera d'expliquer brièvement les mathématiques qui se cachent derrière la théorie de Reed – Solomon. Ensuite, on traitera cette théorie par des mathématiques simples et la circuiterie hardware correspondante. Cette dernière sera le plus souvent suivie par des exemples consacrés à la compréhension des circuits logiques utilisés.

La théorie présentera deux méthodes de décodage des codes de Reed – Solomon. La première solution présentée sera la méthode de la division Euclidienne, quand à la deuxième méthode, elle mettra en évidence l'algorithme de Berlekamp – Massey.

Un chapitre sera consacré à la réalisation hardware des circuits traités en théorie. L'implémentation hardware sera effectuée en utilisant des circuits logiques, le tout traduit en langage VHDL. L'implémentation VHDL sera traitée à l'aide du logiciel Mentor Graphic's HDL Designer. L'implémentation hardware sera toujours suivie d'une simulation temporelle effectuée à l'aide du logiciel Mentor Graphic's « ModelSim ».

2 Applications mathématiques

2.1 Introduction

Le chapitre 2 sera consacré à une brève introduction aux mathématiques utilisées dans les codes de Reed – Solomon. On introduira la notion de groupe, d'anneau et de champ dans les mathématiques abstraites. Par la suite, on se concentrera sur les « champs de Galois » utilisés pour les codes de Reed – Solomon.

2.2 Groupe

Un groupe[1] G est un ensemble d'éléments défini par l'opération binaire $\bullet$. Chaque paire d'éléments, a et b, a un seul élément c défini par l'opération binaire $\bullet$ dans G, $c = a \bullet b$.
Un groupe doit respecter les contraintes suivantes:

- Le groupe doit être fermé, pour n'importe quel élément a et b dans G, l'opération binaire $c = a \bullet b$ doit toujours retourner un élément dans le même groupe G.
- La loi associative doit être applicable ; $a(bc) = (ab)c$ pour tout a,b,c compris dans G.
- Pour n'importe quel élément a dans G, il doit exister un élément tel que $a \bullet e = e \bullet a = a$. L'élément e est appelé l'élément identité de G.
- Pour tout élément a dans G, il doit exister son élément inverse a' tel que $a \bullet a' = e$.

Un groupe est dit « abélien » si la loi de commutation est valable, $ab = ba$ pour tous les éléments compris dans G.

2.3 Anneau

Un anneau[2] R est un ensemble d'éléments défini par deux opérations binaires appelées, *addition* et *multiplication*. Un anneau R est un groupe « abélien » selon l'operateur d'addition. La multiplication doit respecter les contraintes suivantes:

- La multiplication doit être associative, $a(bc) = (ab)c$ pour tout a,b,c compris dans R.
- La multiplication doit être distributive par rapport à l'addition, $a(b+c) = ab + ac$ et $(a+b)c = ac + bc$ pour tout a,b,c compris dans R

[1] Source : Cryptography and Network Security, Third Edition, William Stalling, Prentice Hall
Source : Error Control Coding, Second Edition, S. Lin et D.J. Costello, Prentice Hall
[2] Source : Cryptography and Network Security, Third Edition, William Stalling, Prentice Hall
Source : Error Control Coding, Second Edition, S. Lin et D.J. Costello, Prentice Hall

Un anneau est dit commutatif si sa multiplication est commutative, $ab = ba$ pour tout a,b compris dans R.

Un anneau est appelé domaine d'intégrité s'il possède un élément unité pour la multiplication et qu'il n'admet pas de division par zéro.

- Il y a un élément unité pour la multiplication tel que $a1 = 1a = a$ pour tout a compris dans R
- Pour tout a,b en R, $ab = 0$, il y a alors la possibilité que $a = 0$ ou $b = 0$

2.4 Corps ou champs

Un corps ou champ C est un domaine d'intégrité dans lequel tous les éléments non nuls sont inversibles. Pour un a non nul dans C, il existe un élément a^{-1} dans C tel que $aa^{-1} = 1$. Un corps possède toutes les propriétés définies aux sous-chapitres 2.2 et 2.3.

2.5 Champs de Galois

Les « champs de Galois » font partie d'une branche particulière des mathématiques qui modélise les fonctions du monde numérique. Ils sont très utilisés dans la cryptographie ainsi que pour la reconstruction des données comme on le verra dans le chapitre 3.
La dénomination « champ de Galois » provient du mathématicien français Galois qui en a découvert les propriétés fondamentales.

Il y a deux types de champs, les champs finis et les champs infinis. Les « champs de Galois » finis sont des ensembles d'éléments fermés sur eux-mêmes. L'addition et la multiplication de deux éléments du champ donnent toujours un élément du champ fini.

2.5.1 Eléments des champs de Galois

Un « champ de Galois »[3] consiste en un ensemble de nombres, ces nombres sont constitués à l'aide de l'élément base α comme suit :

$$0, 1, \alpha, \alpha^2, \alpha^3 ... \alpha^{N-1} \qquad \textbf{(2-1)}$$

En prenant $N = 2^m - 1$, on forme un ensemble de 2^m éléments. Le champ est alors noté $GF\left(2^m\right)$.

$GF\left(2^m\right)$ est formé à partir du champ de base $GF\left(2\right)$ et contiendra des multiples des éléments simples de $GF\left(2\right)$.

[3] Source : Reed – Solomon error correction, C. K. P Clarke, British Broadcast Corporation

En additionnant les puissances de α, chaque élément du champ peut être représenté par une expression polynomiale du type :

$$\alpha^{m-1}x^{m-1} + \alpha^{m-2}x^{m-2} + ... + \alpha x + \alpha^0 \qquad \textbf{(2-2)}$$

Avec :

$\alpha^{m-1}...\alpha^0$: éléments bases du $GF(2)$ (valeurs : $0,1$)

Sur les « champs de Galois », on peut effectuer toutes les opérations de base. L'addition dans un champ fini $GF(2)$ correspond à faire une addition modulo 2, donc l'addition de tous les éléments d'un « champ de Galois » dérivés du champ de base sera une addition modulo 2 (XOR). La soustraction effectuera la même opération qu'une addition, c'est-à-dire, la fonction logique « XOR ».
La multiplication et la division seront des opérations modulo « grandeur du champ », donc $\mathrm{mod}\left(2^m - 1\right)$.

2.5.1.1 Addition dans $GF(2)$

Dans ce sous-chapitre, on cherche à comprendre comment l'affirmation du chapitre 2.5.1 est possible. Considérons le tableau ci-dessous dans lequel'on fait l'addition binaire entre les deux éléments A et B du $GF(2)$:

A	B	Reste	Résultat
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Tableau 2-1: addition de deux éléments dans un $GF(2)$

En négligeant le reste dans le résultat final de l'addition, on constate que la somme entre deux éléments dans un $GF(2)$ donne une addition modulo 2, c'est-à-dire, une fonction logique « XOR ». Comme $GF(2)$ est le champ de base, cette relation sera valable pour tous les champs dérivés, c'est-à-dire, pour $GF\left(2^m\right)$.

2.5.1.2 Soustraction dans $GF(2)$

Ici, on cherche à comprendre comment une soustraction dans $GF(2)$ correspond à faire une addition dans le même champ. Considérons le tableau suivante dans lequel'on fait la soustraction binaire entre les deux éléments A et B:

A	B	Emprunte	Résultat
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

Tableau 2-2: soustraction de deux éléments dans un $GF(2)$

On constate que la soustraction dans $GF(2)$ effectue la même opération que l'addition dans le même champ, c'est-à-dire une opération logique « XOR ».

2.5.2 Polynôme primitif

Ce polynôme permet de construire le « champ de Galois » souhaité. Tous les éléments non nuls du champ peuvent être construits en utilisant l'élément α comme racine du polynôme primitif. Chaque m a peut être plusieurs polynômes primitifs $p(x)$, mais dans le tableau ci-dessous, on mentionne seulement les polynômes ayant le moins d'éléments. Les polynômes primitifs pour les principaux « champs de Galois » sont les suivants:

m	$P(X)$	m	$P(X)$
3	$1+X+X^3$	14	$1+X+X^6+X^{10}+X^{14}$
4	$1+X+X^4$	15	$1+X+X^{15}$
5	$1+X^2+X^5$	16	$1+X+X^3+X^{12}+X^{16}$
6	$1+X+X^6$	17	$1+X^3+X^{17}$
7	$1+X^3+X^7$	18	$1+X^7+X^{18}$
8	$1+X^2+X^3+X^4+X^8$	19	$1+X+X^2+X^5+X^{19}$
9	$1+X^4+X^9$	20	$1+X^3+X^{20}$
10	$1+X^3+X^{10}$	21	$1+X^2+X^{21}$
11	$1+X^2+X^{11}$	22	$1+X+X^{22}$
12	$1+X+X^4+X^6+X^{12}$	23	$1+X^5+X^{23}$
13	$1+X+X^3+X^4+X^{13}$	24	$1+X+X^2+X^7+X^{24}$

Tableau 2-3: polynômes primitifs dans $GF(2^m)$ [4]

[4] Source : Errror control coding, Modin

2.5.2.1 Exemple de construction d'un $GF\left(2^4\right)$

On veut construire tous les éléments du $GF\left(2^4\right)$ à partir du polynôme primitif :

$$p(x) = x^4 + x + 1$$

Comme mentionné au chapitre 2.5.2, l'élément α est racine du polynôme primitif, on peut en déduire :

$$\begin{aligned} p(x) &= x^4 + x + 1 \\ p(\alpha) &= \alpha^4 + \alpha + 1 \\ 0 &= \alpha^4 + \alpha + 1 \\ \alpha^4 &= \alpha + 1 \end{aligned}$$

Maintenant, il suffit de multiplier l'élément α par α à chaque étape et réduire par rapport à $\alpha^4 = \alpha + 1$ pour obtenir le champ complet. On aura besoin de 13 multiplications pour compléter le champ.

Les éléments d'un « champ de Galois » de $GF\left(2^4\right)$ sont :

Eléments	Formes polynomiales	Formes binaires	Formes décimales
0	0	0000	0
1	1	0001	1
α	α	0010	2
α^2	α^2	0100	4
α^3	α^3	1000	8
α^4	$\alpha + 1$	0011	3
α^5	$\alpha^2 + \alpha$	0110	6
α^6	$\alpha^3 + \alpha^2$	1100	12
α^7	$\alpha^3 + \alpha + 1$	1011	11
α^8	$\alpha^2 + 1$	0101	5
α^9	$\alpha^3 + \alpha$	1010	10
α^{10}	$\alpha^2 + \alpha + 1$	0111	7
α^{11}	$\alpha^3 + \alpha^2 + \alpha$	1110	14
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	1111	15
α^{13}	$\alpha^3 + \alpha^2 + 1$	1101	13
α^{14}	$\alpha^3 + 1$	1001	9

Tableau 2-4: éléments de $GF\left(2^4\right)$

2.5.2.2 Construction d'un champ de $GF\left(2^4\right)$ sous Matlab

Les éléments d'un « champ de Galois » peuvent être aussi calculés avec Matlab selon les instructions suivantes:

```
p=2;   % Nombre base du champ
m=4;   % Eléments

champ = gftuple([-1:p^m-2]',m,p); %Calcul du champ en binaire
```

Code 2-1: éléments dans $GF\left(2^4\right)$ sous Matlab

2.6 Dérivation formelle d'un polynôme dans un champ de Galois

Un polynôme d'ordre m est défini comme:

$$f(x) = f_m x^m + f_{m-1} x^{m-1} + \ldots + f_1 x + f_0 \qquad \textbf{(2-3)}$$

La dérivée formelle[5] du polynôme (2-3) dans un « champ de Galois » $GF\left(2^m\right)$ est définie avec les puissances paires de (2-3) :

$$\begin{aligned} f'(x) &= f_1 + 2 f_2 x + 3 f_3 x^2 + \ldots + m f_m x^{m-1} \\ f'(x) &= f_1 + 3 f_3 x^2 + 5 f_5 x^4 + \ldots \end{aligned} \qquad \textbf{(2-4)}$$

La dérivée est définie seulement par les puissances paires car les puissance impaires sont précédées de coefficients paires qui annulent les termes:

$$\begin{aligned} &2 = 1+1 = 0 \\ &4 = 2+2 = 1+1+1+1 = 0 \\ &\ldots \end{aligned} \qquad \textbf{(2-5)}$$

La dérivation doit respecter les règles suivantes :

- Si $f^2(x)$ divise $a(x)$ alors $f(x)$ divise $a'(x)$.
- La dérivation est une opération linéaire, elle doit respecter la règle de linéarité suivante : $\left[f(x)a(x) \right]' = f'(x)a(x) + f(x)a'(x)$.

[5] Source : PGZ Decoder, John Gill, Stanford University

3 Principe pratique de Reed – Solomon

3.1 Introduction

Les codes de Reed – Solomon sont des codes de détection et de correction des erreurs. Se sont des codes particuliers des codes BCH.

Les messages sont divisés en blocs dont on a ajouté des informations redondantes à chaque bloc permettant ainsi de diminuer la possibilité de retransmission. La longueur des blocs dépend de la capacité du codeur.

Le décodeur traite chaque bloc et corrige les éventuelles erreurs. A la fin de ce traitement, les données originelles seront restaurées.

Typiquement, la transmission des données dans un canal avec cette méthode est effectuée ainsi :

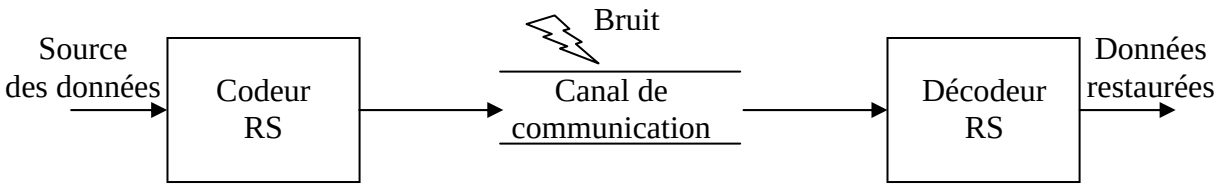

Figure 3-1: schéma général

3.2 Propriétés des codes Reed – Solomon

Ces codes ont une propriété importante, ils sont linéaires et font partie des codes BCH.
Le codeur prend *k* symboles de donnée (chaque symbole contenant *s* bits) et calcule les informations de contrôle pour construire *n* symboles, ce qui donne *n-k* symboles de contrôle. Le décodeur peut corriger au maximum *t* symboles, ou *2t=n-k*.

Le diagramme ci-dessous montre une trame constituée avec le codeur Reed – Solomon :

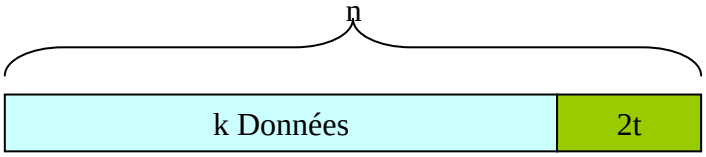

Figure 3-2: mot-code de Reed – Salomon

La longueur maximale d'un code de Reed – Solomon est définie comme :

$$n = k + 2t = 2^s - 1 \qquad \textbf{(3-1)}$$

Avec :

k : nombre de symboles d'information
$2t$: nombre de symboles de contrôle
s : nombre de bits par symbole

La distance minimale d'un code Reed – Solomon est :

$$d_{\min} = 2t + 1 \qquad \textbf{(3-2)}$$

Autre propriété des codes Reed – Salomon, ils sont cycliques, c'est-à-dire, que chaque mot-code décalé engendre un autre mot-code. Tous les codes cycliques peuvent être réduits en gardant la même capacité d'erreur, mais le nouveau code formé n'est alors pas cyclique.

De plus, les codes de Reed – Solomon sont des codes non-binaires. Les codes sont représentés sur des « champs de Galois » de $GF\left(2^m\right)$ et non pas sur des champs de $GF\left(2\right)$. Les symboles sont définis comme les coefficients du polynôme et le degré de x indique l'ordre. Ainsi, le symbole avec l'ordre le plus élevé est reçu/envoyé en premier et le dernier symbole reçu/envoyé est celui dont l'ordre est moindre.

3.2.1 Exemple de polynôme

Prenons le polynôme suivant :

$$\alpha^2 x^2 + \alpha^3 x + \alpha^6 \qquad \textbf{(3-3)}$$

Le premier coefficient entrant ou sortant du codeur/décodeur sera α, suivi de α^3 puis α^6.

3.2.2 Exemple du nombre de symboles corrigibles

Prenons un code de Reed – Solomon RS(15, 9), que l'on utilisera par la suite pour tous les autres exemples. L'objectif est de découvrir combien de bits sont utilisés pour chaque symbole et combien d'erreurs peut-on corriger.

RS(n,k)=RS(15,9) : *n* indiquant la longueur totale d'un bloc de Reed – Solomon, 15 symboles dans ce cas et *k* indique la longueur du bloc d'information, 9 symboles dans cet exemple.

La capacité de correction des erreurs du système est:

$$2t = n - k = 15 - 9 = 6$$

Donc :

$$t = \frac{n-k}{2} = 3$$

Ce code permettra de corriger 3 symboles. Le nombre de bits *s* par symbole est :

$$n = 2^s - 1$$

$$s = \frac{\ln(n+1)}{\ln(2)} = \frac{\ln(16)}{\ln(2)} = 4$$

Le nombre de bits utilisés pour coder les symboles est donc de 4. Ce qui nous amène à utiliser un « champ de Galois » de $GF\left(2^4\right)$.

3.3 Codage

3.3.1 Introduction

Le codage avec les codes de Reed – Solomon est effectué de la même façon que le codage à l'aide du CRC. La seule différence est que les codes de Reed – Solomon sont non-binaires (Reed – Solomon $GF\left(2^m\right)$), alors que le CRC est binaire, ($GF\left(2\right)$).

3.3.2 Théorie du Codage

L'équation clé définissant le codage systématique de Reed – Solomon(n,k) est :

$$c(x) = i(x)x^{n-k} + \left[i(x)x^{n-k}\right] \bmod \left(g(x)\right) \qquad \textbf{(3-4)}$$

Avec :

$c(x)$	: polynôme du mot-code, degré $n-1$
$i(x)$	: polynôme d'information, degré $k-1$
$\left[i(x)x^{n-k}\right] \bmod \left(g(x)\right)$	: polynôme de contrôle, degré $n-k-1$
$g(x)$	: polynôme générateur, degré $n-k$

Le codage systématique signifie que l'information est codée dans le degré élevé du mot-code et que les symboles de contrôle sont introduits après les mots d'information.

3.3.2.1 Polynôme générateur

Les symboles de contrôle sont générés à l'aide de polynômes particuliers, appelés polynômes générateurs. Tous les codes Reed – Solomon sont valables si et seulement si ils sont divisibles par leur polynôme générateur, $c(x)$ doit être divisible par $g(x)$.

Pour la génération d'un correcteur d'erreurs de t symboles, on devrait avoir un polynôme générateur de puissance α^{2t}. La puissance maximale du polynôme est déterminée grâce la distance minimale qui est $d_{\min} = 2t+1$. On devrait avoir $2t+1$ termes du polynôme générateur.

Le polynôme générateur est sous la forme :

$$g(x) = \left(x-\alpha^1\right)\left(x-\alpha^2\right)\ldots\left(x-\alpha^{2t}\right)$$
$$g(x) = g_{2t}x^{2t} + g_{2t-1}x^{2t-1} + \ldots + g_1x + g_0 \qquad \textbf{(3-5)}$$

3.3.2.1.1 Calcul des coefficients du polynôme générateur pour RS(15,9)

On veut calculer les coefficients du polynôme générateur pour le calcul des symboles de contrôle d'un code de RS(15,9) qui puisse corriger 3 erreurs, comme vu dans l'exemple 3.2.2.

La forme générale du polynôme générateur est :

$$g(x) = (x-\alpha^1)(x-\alpha^2)...(x-\alpha^{2t})$$

En développant cette équation, on trouve :

$$\begin{aligned} g(x) &= (x-\alpha^1)(x-\alpha^2)...(x-\alpha^{2t}) \\ &= (x-\alpha^1)(x-\alpha^2)(x-\alpha^3)(x-\alpha^4)(x-\alpha^5)(x-\alpha^6) \\ &= (x^2+\alpha^5 x+\alpha^3)(x^2+\alpha^7 x+\alpha^7)(x^2+\alpha^9 x+\alpha^{11}) \\ &= (x^4+\alpha^{13}x^3+\alpha^6 x^2+\alpha^3 x+\alpha^{10})(x^2+\alpha^9 x+\alpha^{11}) \\ &= x^6+x^5(\alpha^{13}+\alpha^9)+x^4(\alpha^6+\alpha^7+\alpha^{11})+x^3(\alpha^3+1+\alpha^9)+x^2(\alpha^{10}+\alpha^{12}+\alpha^2)+x(\alpha^4+\alpha^{14})+\alpha^6 \\ &= x^6+\alpha^{10}x^5+\alpha^{14}x^4+\alpha^4 x^3+\alpha^6 x^2+\alpha^9 x+\alpha^6 \end{aligned}$$

En prenant l'équivalence en décimal, on obtient :

$$g(x) = x^6+7x^5+9x^4+3x^3+12x^2+10x+12$$

3.3.3 Implémentation hardware du codeur

Pour comprendre comment la partie hardware fonctionne, on doit comprendre la définition mathématique du codage (3-4) et les différentes opérations effectuées.

Le codage est systématique, on doit effectuer une opération de décalage pour placer les informations dans le degré élevé du mot-code de sortie. Mathématiquement, le décalage est effectué selon la fonction :

$$i(x)\,x^{n-k} \qquad \textbf{(3-6)}$$

Avec :

$i(x)$: polynôme d'information

x^{n-k} : décalage du polynôme d'information de $n-k$ positions vers la gauche

Le deuxième terme de l'équation (3-4) est le reste de la division de $\frac{i(x)\,x^{n-k}}{g(x)}$. Cette division donnera les symboles de contrôle.

L'implémentation du codeur demandera deux opérations : un décalage et une division. Ces deux opérations peuvent être effectuées grâce à des registres à décalage et à des multiplexeurs.

3.3.3.1 Schéma

Schéma général du codage:

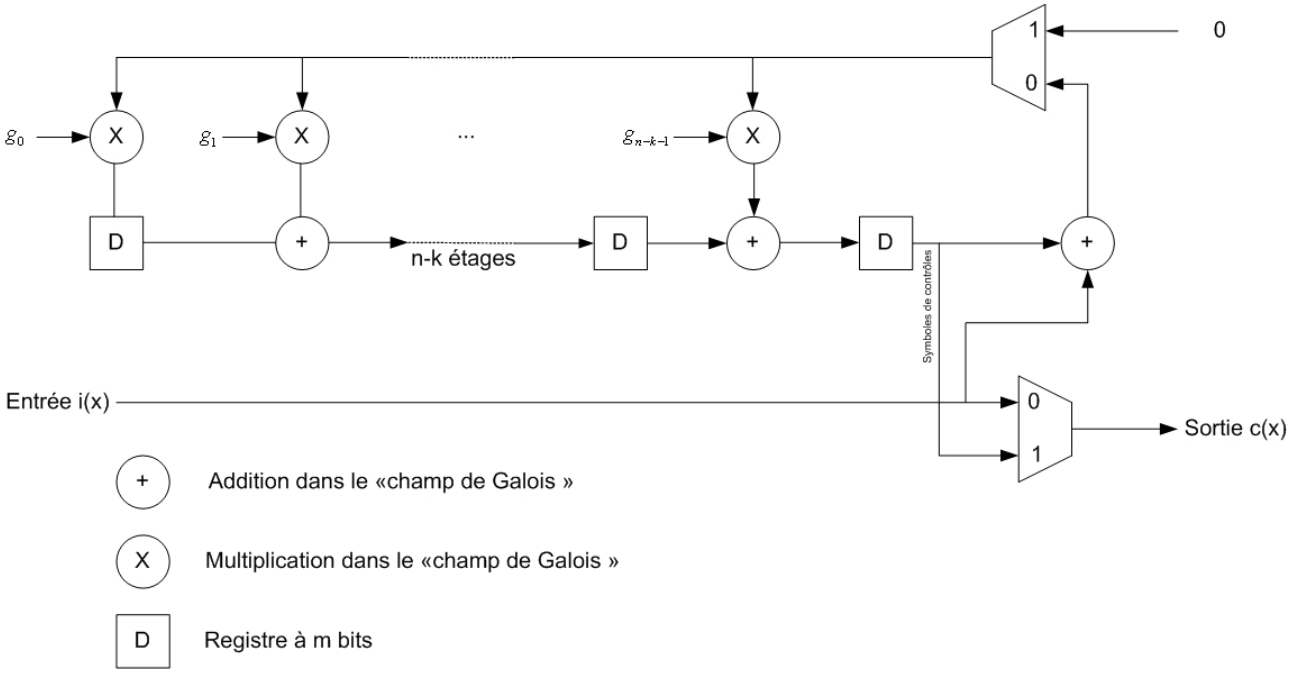

Figure 3-3: schéma de codage[6]

Les éléments utilisés pour le codage sont:

- Des additionneurs dans le « champ de Galois » $GF\left(2^m\right)$
- Des multiplicateurs dans le « champ de Galois » $GF\left(2^m\right)$
- Des multiplexeurs
- Des registres

3.3.3.1.1 Addition

Les additions sont définies dans le « champ de Galois » $GF\left(2^m\right)$, donc pour additionner deux éléments, on prendra la notation binaire de chaque élément et on les additionnera modulo 2. L'addition modulo 2 est une opération logique définie par l'opérateur logique « XOR » bit à bit. L'addition entre deux éléments d'un champ fini donnera toujours un élément dans le même champ.

[6] Source : Self-correcting codes conquer noise, Reed – Solomon codecs, S. S. Shab, S. Yaqub, F. Suleman,designfeature

3.3.3.1.1.1 Exemple

L'addition des deux symboles définis dans le « champ de Galois » de $GF\left(2^4\right)$.

On a :

$$\left.\begin{aligned} A(x) &= \gamma_3 x^3 + \gamma_2 x^2 + \gamma_1 x + \gamma_0 \\ B(x) &= \lambda_3 x^3 + \lambda_2 x^2 + \lambda_1 x + \lambda_0 \end{aligned}\right\} \gamma_3, \lambda_3, \gamma_2, \lambda_2, \gamma_1, \lambda_1, \gamma_0, \lambda_0 \text{ appartiennent à } GF(2)$$

L'addition dans un « champ de Galois » de $GF\left(2^4\right)$ se calcule ainsi:

$$\begin{aligned} Out &= A(x) + B(X) \\ &= x^3(\gamma_3 + \lambda_3) + x^2(\gamma_2 + \lambda_2) + x(\gamma_1 + \lambda_1) + \gamma_0 + \lambda_0 \\ Out(3) &= \gamma_3 + \lambda_3 \\ Out(2) &= \gamma_2 + \lambda_2 \\ Out(1) &= \gamma_1 + \lambda_1 \\ Out(0) &= \gamma_0 + \lambda_0 \end{aligned}$$

La figure 3-4 montre le schéma de l'addition :

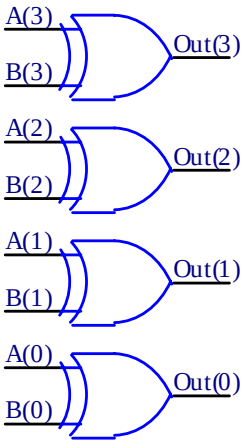

Figure 3-4: schéma de l'addition en $GF\left(2^4\right)$

3.3.3.1.2 Multiplication

Les multiplications utilisées dans les codes de Reed – Solomon sont des multiplications dans le « champ de Galois » $GF\left(2^m\right)$. La multiplication dans le « champ de Galois » est une opération modulaire, c'est-à-dire que la multiplication entre deux éléments d'un champ fini donnera toujours un élément dans le même champ. Il y a plusieurs techniques pour effectuer ce calcul, dans le cadre de ce projet on choisira de développer la multiplication entre deux polynômes[7] et ensuite de normaliser le résultat par rapport au polynôme primitif du champ choisi. La deuxième technique de multiplication est effectuée à l'aide d'une table de vérité.

3.3.3.1.2.1 Exemple

Multiplication de deux éléments dans le « champ de Galois » de $GF\left(2^4\right)$

$$\left.\begin{aligned}\beta_1 &= \gamma_3 z^3 + \gamma_2 z^2 + \gamma_1 z + \gamma_0 \\ \beta_2 &= \lambda_3 z^3 + \lambda_2 z^2 + \lambda_1 z + \lambda_0\end{aligned}\right\} \ \gamma_3,\lambda_3,\gamma_2,\lambda_2,\gamma_1,\lambda_1,\gamma_0,\lambda_0 \text{ appartiennent à } GF\left(2\right)$$

La multiplication dans ce cas doit être réduite selon le polynôme primitif :

$$p\left(z\right) = z^4 + z + 1$$

Réduction en z^4:

$$z^4 = z + 1$$

La multiplication donne :

$$\begin{aligned}\beta_1\beta_2 &= \left(\gamma_3 z^3 + \gamma_2 z^2 + \gamma_1 z + \gamma_0\right)\left(\lambda_3 z^3 + \lambda_2 z^2 + \lambda_1 z + \lambda_0\right) \\ &= \gamma_3\lambda_3 z^6 + z^5\left(\gamma_3\lambda_2 + \gamma_2\lambda_3\right) + z^4\left(\gamma_3\lambda_1 + \gamma_2\lambda_2 + \gamma_1\lambda_3\right) + z^3\left(\gamma_3\lambda_0 + \gamma_2\lambda_1 + \gamma_1\lambda_2 + \gamma_0\lambda_3\right) \\ &\quad + z^2\left(\gamma_2\lambda_0 + \gamma_1\lambda_1 + \gamma_0\lambda_2\right) + z\left(\gamma_1\lambda_0 + \gamma_0\lambda_1\right) + \gamma_0\lambda_0\end{aligned}$$

Réduction en z^4 :

$$\begin{aligned}z^6 &= z^4 z^2 = z^2\left(z+1\right) \\ z^5 &= z^4 z = z\left(z+1\right) \\ z^4 &= \left(z+1\right)\end{aligned}$$

[7] Source : Self-correcting codes conquer noise, Reed – Solomon codecs, S. S. Shab, S. Yaqub, F. Suleman,designfeature

En réduisant l'expression ci-dessus, on obtient :

$$\begin{aligned}\beta_1\beta_2 &= \left(\gamma_3 z^3+\gamma_2 z^2+\gamma_1 z+\gamma_0\right)\left(\lambda_3 z^3+\lambda_2 z^2+\lambda_1 z+\lambda_0\right)\\ &= \gamma_3\lambda_3 z^6+z^5\left(\gamma_3\lambda_2+\gamma_2\lambda_3\right)+z^4\left(\gamma_3\lambda_1+\gamma_2\lambda_2+\gamma_1\lambda_3\right)+z^3\left(\gamma_3\lambda_0+\gamma_2\lambda_1+\gamma_1\lambda_2+\gamma_0\lambda_3\right)\\ &\quad+z^2\left(\gamma_2\lambda_0+\gamma_1\lambda_1+\gamma_0\lambda_2\right)+z\left(\gamma_1\lambda_0+\gamma_0\lambda_1\right)+\gamma_0\lambda_0\\ &= z^3\left(\gamma_3\lambda_0+\gamma_2\lambda_1+\gamma_1\lambda_2+\gamma_0\lambda_3+\gamma_3\lambda_3\right)+z^2\left(\gamma_2\lambda_0+\gamma_1\lambda_1+\gamma_0\lambda_2+\gamma_3\lambda_3+\gamma_3\lambda_2+\gamma_{_2}\lambda_3\right)\\ &\quad+z\left(\gamma_1\lambda_0+\gamma_0\lambda+\gamma_3\lambda_1+\gamma_2\lambda_2+\gamma_1\lambda_3+\gamma_3\lambda_2+\gamma_3\lambda_3\right)+\gamma_0\lambda_0+\gamma_3\lambda_1+\gamma_2\lambda_2+\gamma_1\lambda_3\end{aligned}$$

La multiplication entre chaque terme est une opération logique « AND » et la somme est une addition modulo 2, donc une opération logique « XOR ».

Schéma de la multiplication en $GF\left(2^4\right)$:

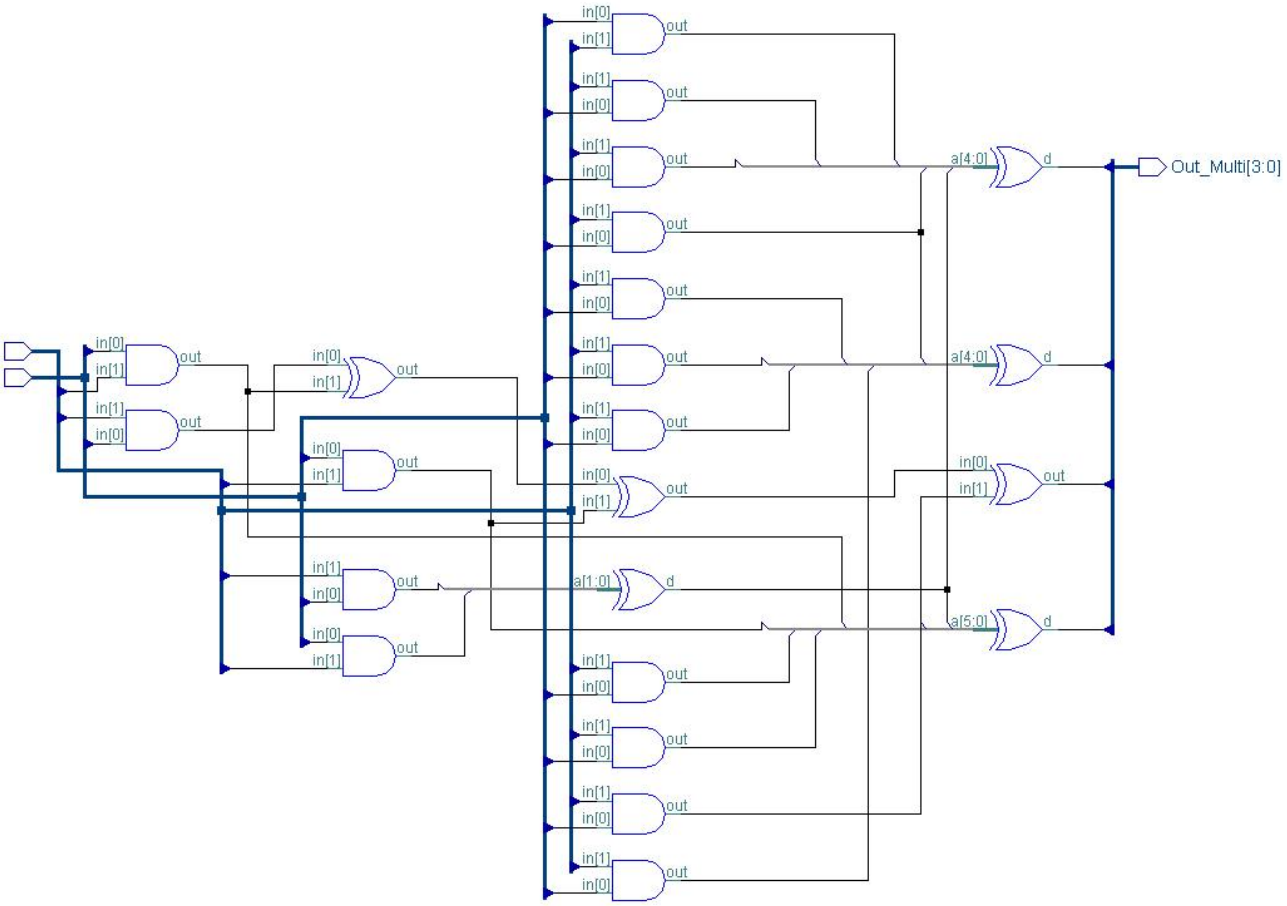

Figure 3-5: schéma de la multiplication en $GF\left(2^4\right)$

3.4 Décodage

3.4.1 Introduction

L'idée de base du décodeur de Reed – Solomon est de détecter une séquence erronée avec peu de termes, qui sommée aux données reçues, donne lieu à un mot-code valable.

Plusieurs étapes sont nécessaires pour le décodage de ces codes :

- Calcul du syndrome
- Calcul des polynômes de localisation des erreurs et de d'amplitude
- Calcul des racines et évaluation des deux polynômes
- Sommation du polynôme constitué et du polynôme reçu pour reconstituer l'information de départ sans erreur.

Le schéma de décodage est montré dans la figure 3-6 ci-dessous :

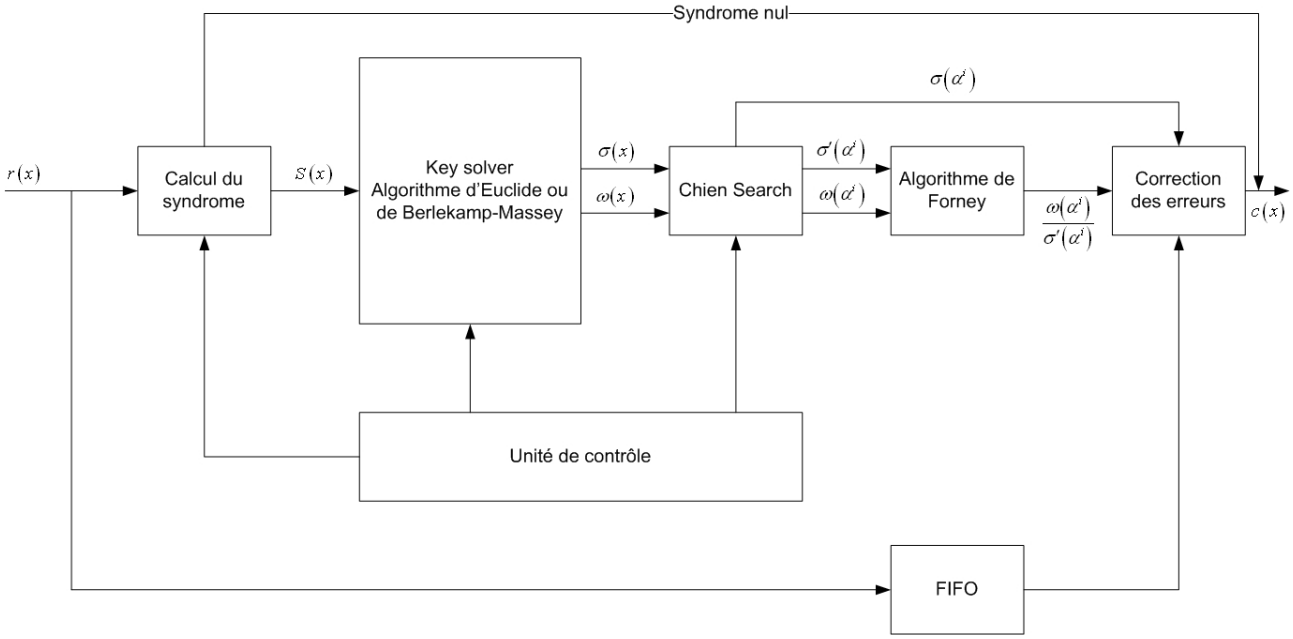

Figure 3-6: schéma du décodage

Avec :

$r(x)$: mot-code reçu

$S(x)$: syndrome calculé

$\omega(x)$: polynôme d'amplitude des erreurs

$\omega(\alpha^i)$: polynôme d'amplitude des erreurs évalué pour tous les éléments compris dans $GF(2^m)$

$\sigma(x)$: polynôme de localisation des erreurs

$\sigma(\alpha^i)$: polynôme de localisation des erreurs évalué pour tous les éléments compris dans $GF(2^m)$

$\sigma'(\alpha^i)$: dérivée du polynôme de localisation des erreurs évalué pour tous les éléments compris dans $GF(2^m)$

$\frac{\omega(\alpha^i)}{\sigma'(\alpha^i)}$: division entre le polynôme d'amplitude et la dérivée du polynôme de localisation des erreurs

$c(x)$: mot-code reconstitué

3.4.2 Généralité du décodage

Considérons un code de Reed – Solomon $c(x)$ correspondant au code transmis et soit $r(x)$ le code que l'on reçoit. Le polynôme d'erreurs introduit par le canal est défini comme :

$$\begin{aligned} e(x) &= r(x) - c(x) \\ &= r(x) + c(x) \\ &= e_0 + e_1 x + \ldots + e_{n-1} x^{n-1} \end{aligned} \quad \textbf{(3-7)}$$

Supposant que le polynôme des erreurs contienne v erreurs aux positions $x^{j_1}, x^{j_2}, \ldots, x^{j_v}$ avec $0 \le j_1 < j_2 < \ldots < j_v \le n-1$. On peut donc redéfinir le polynôme des erreurs comme :

$$e(x) = e_{j_1} x^{j_1} + e_{j_2} x^{j_2} + \ldots + e_{j_v} x^{j_v} \quad \textbf{(3-8)}$$

Avec :

$e_{j_1}, e_{j_2}, \ldots, e_{j_v}$: valeurs d'amplitude des erreurs

$x^{j_1}, x^{j_2}, \ldots, x^{j_v}$: emplacements des erreurs

A partir du polynôme $r(x)$ reçu, on peut calculer le polynôme du syndrome $S(x)$ qui nous indiquera la présence d'éventuelles erreurs. Si tous les coefficients du syndrome sont nuls, alors les étapes suivantes du décodage n'ont pas lieu d'être car le mot-code reçu ne contiendra pas d'erreurs. Par contre, si le syndrome est non nul, on devra calculer le polynôme de localisation des erreurs et le polynôme d'amplitude des erreurs. Il y a plusieurs méthodes de calcul de ces deux polynômes, dans le cadre de ce projet on ne traitera que deux méthodes, le décodage selon l'algorithme d'Euclide et le décodage selon l'algorithme de Berlekamp – Massey. Une fois les polynômes calculés en utilisant l'algorithme de Forney, on calculera les valeurs à soustraire pour obtenir le mot-code sans erreur.

3.4.3 Calcul du syndrome

Le calcul du syndrome est défini comme le reste de la division entre le polynôme reçu $r(x)$ et le polynôme générateur $g(x)$. Le reste indiquera la présence d'erreurs. Comme l'opération division est toujours une opération complexe par rapport à des sommes et des additions, on est amené à chercher une autre méthode pour le calcul du syndrome[8].

Le calcul du syndrome peut aussi être effectué par un processus itératif. Avant de pouvoir calculer le polynôme du syndrome, on doit attendre que l'on ait reçu tous les éléments du polynôme $r(x)$.

Comme :

$$r(x) = c(x) + e(x) \tag{3-9}$$

On obtient :

$$S_i = r(\alpha^i) = c(\alpha^i) + e(\alpha^i) = e(\alpha^i) \tag{3-10}$$

Par la relation (3-10) on peut définir les différentes équations qui lient le polynôme d'erreurs au syndrome :

$$S_1 = e_{j_1}\alpha^{j_1} + e_{j_2}\alpha^{j_2} + \ldots + e_{j_v}\alpha^{j_v}$$

$$S_2 = e_{j_1}\alpha^{2j_1} + e_{j_2}\alpha^{2j_2} + \ldots + e_{j_v}\alpha^{2j_v} \tag{3-11}$$

...

$$S_{2t} = e_{j_1}\alpha^{2tj_1} + e_{j_2}\alpha^{2tj_2} + \ldots + e_{j_v}\alpha^{2tj_v}$$

Le syndrome sous forme polynomiale sera :

$$S(x) = \ldots + S_{2t+1}x^{2t} + S_{2t}x^{2t-1} + \ldots + S_2 x + S_1 \tag{3-12}$$

Seuls les premiers $2t$ symboles du syndrome sont connus.

Si le code reçu $r(x)$ n'est pas affecté par des erreurs alors tous les coefficients du syndrome seront nuls ($r(x) = c(x)$).

[8] Source : Error Control Coding, Second Edition, S. Lin et D.J. Costello, Prentice Hall

3.4.3.1 Exemple

On a un code de RS(15,9), donc $15-9=6$ symboles de contrôle ($2t=6$). On reçoit le polynôme suivant $r(x)$ et on veut calculer le syndrome S :

$$r(x)=\alpha x^{14}+\alpha^2 x^{12}+\alpha^{13}x^4$$

$$\begin{aligned}S_1 &= r(x=\alpha)=\alpha(\alpha^{14})+\alpha^2(\alpha^{12})+\alpha^{13}(\alpha^4)\\ &=\alpha^{1+14}+\alpha^{2+12}+\alpha^{13+4}\\ &=1+\alpha^3+1+\alpha^2\\ &=\alpha^3+\alpha^2=\alpha^6\end{aligned}$$

$$S_2=r(x=\alpha^2)=\alpha\left((\alpha^2)^{14}\right)+\alpha^2\left((\alpha^2)^{12}\right)+\alpha^7\left((\alpha^2)^4\right)=\alpha^7$$

$$S_3=r(x=\alpha^3)=\alpha\left((\alpha^3)^{14}\right)+\alpha^2\left((\alpha^3)^{12}\right)+\alpha^7\left((\alpha^3)^4\right)=\alpha^{12}$$

$$S_4=r(x=\alpha^4)=\alpha\left((\alpha^4)^{14}\right)+\alpha^2\left((\alpha^4)^{12}\right)+\alpha^7\left((\alpha^4)^4\right)=0$$

$$S_5=r(x=\alpha^5)=\alpha\left((\alpha^5)^{14}\right)+\alpha^2\left((\alpha^5)^{12}\right)+\alpha^7\left((\alpha^5)^4\right)=\alpha$$

$$S_6=r(x=\alpha^6)=\alpha\left((\alpha^6)^{14}\right)+\alpha^2\left((\alpha^6)^{12}\right)+\alpha^7\left((\alpha^6)^4\right)=\alpha^8$$

3.4.3.2 Schéma de calcul du syndrome

Le schéma de la figure 3-7 calcule un symbole du syndrome de façon itérative :

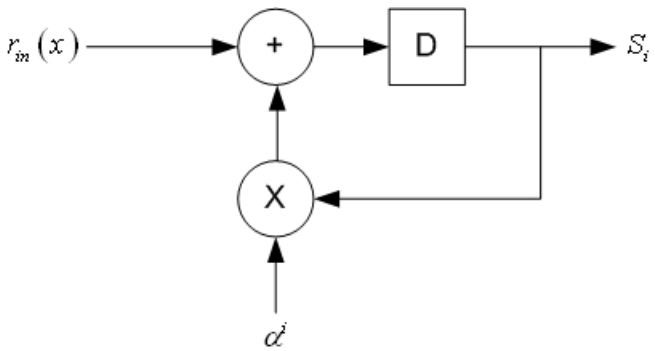

Figure 3-7: schéma pour le calcul du syndrome

On aura besoin de $2t$ schémas, comme celui de la figure 3-7, pour avoir le syndrome complet.

3.4.3.3 Exemple

Depuis la figure 3-7, on peut calculer la valeur du premier coefficient du polynôme S_1 selon le polynôme reçu : $r(x) = \alpha x^{14} + \alpha^2 x^{12} + \alpha^{13} x^4$

i	r_{in}	$D1$	S_{1out}	*Multi*
0	α	α	0	
1	0	α^2	α	α^2
2	α^2	α^6	α^2	α^6
3	0	α^7	α^6	α^7
4	0	α^8	α^7	α^8
5	0	α^9	α^8	α^9
6	0	α^{10}	α^9	α^{10}
7	0	α^{11}	α^{10}	α^{11}
8	0	α^{12}	α^{11}	α^{12}
9	0	α^{13}	α^{12}	α^{13}
10	α^{13}	α^2	α^{13}	α^2
11	0	α^3	α^2	α^3
12	0	α^4	α^3	α^4
13	0	α^5	α^4	α^5
14	0	α^6	α^5	α^6
15	-	-	α^6	α^7

Tableau 3-1: tableau du calcul du syndrome S1

Schéma :

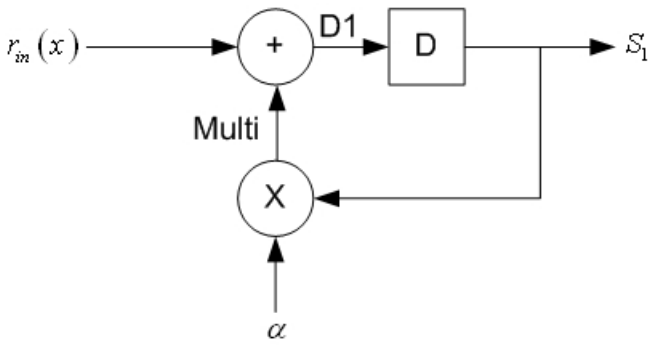

Figure 3-8: schéma avec signaux détaillés pour le calcul du syndrome

3.4.4 Euclide

3.4.4.1 Généralité du théorème d'Euclide

L'algorithme d'Euclide[9] est un algorithme récursif qui permet de trouver le plus grand diviseur commun de deux polynômes $r_0(x)$ et $r_1(x)$ dans le « champ de Galois »$GF(q)$.
Il existe deux polynômes $a(x)$et $b(x)$ en $GF(q)$ tels que :

$$MCD\left(r_0(x), r_1(x)\right) = a(x)r_0(x) + b(x)r_1(x) \qquad \textbf{(3-13)}$$

Avec :

$a(x)$ et $b(x)$ peuvent être calculés selon l'algorithme d'Euclide.

En donnant deux polynômes non nuls $r_0(x)$ et $r_1(x)$ en $GF(q)$, l'algorithme d'Euclide fonctionne de la façon suivante :

$$\begin{aligned} &\deg\left(r_1(x)\right) \le \deg\left(r_0(x)\right) \\ &a_0(x) = 1, b_0(x) = 0 \\ &a_1(x) = 0, b_1(x) = 1 \end{aligned} \qquad \textbf{(3-14)}$$

Avec :

$\deg\left(r_1(x)\right)$: degré du polynôme $r_1(x)$
$\deg\left(r_0(x)\right)$: degré du polynôme $r_0(x)$

Pour $i \geq 2$, on calcule le quotient $q_i(x)$ et le polynôme restant $r_i(x)$, la division est effectuée sur $r_{i-2}(x)$ et $r_{i-1}(x)$.

$$r_{i-2}(x) = q_i(x)r_{i-1}(x) + r_i(x) \qquad \textbf{(3-15)}$$

Avec :

$$\begin{aligned} &0 \le \deg\left(r_i(x)\right) < \deg\left(r_{i-1}(x)\right) \\ &a_i(x) = a_{i-2}(x) - q_i(x)a_{i-1}(x) \\ &b_i(x) = b_{i-2}(x) - q_i(x)b_{i-1}(x) \end{aligned} \qquad \textbf{(3-16)}$$

Les calculs se terminent lorsque $\deg\left(r_i\right) = 0$, le dernier polynôme non nul indique le plus grand diviseur commun.

[9] Source : Error-Control Block Codes, L.H. Charles LEE, Artech House Publishers

3.4.4.2 Correction d'erreurs avec Euclide[10]

Le polynôme de localisation des erreurs est défini comme :

$$\sigma(x) = \prod_{k=1}^{v}\left(1-\alpha^{i_k}x\right) = \sigma_v x^v + \sigma_{v-1}x^{v-1} + \ldots + \sigma_1 x + 1 \quad \textbf{(3-17)}$$

Le polynôme d'amplitude des erreurs se calculera de la suivant façon :

$$\omega(x) = S(x)\sigma(x) \quad \textbf{(3-18)}$$

Avec :

$\sigma(x)$: polynôme de localisation des erreurs, inconnu à ce stade

$\omega(x)$: polynôme d'amplitude, inconnu à ce stade

$S(x)$: polynôme syndrome, connu

Comme on connait seulement $2t$ symboles du polynôme du syndrome ($x^0 \ldots x^{2t-1}$), on devrait limiter le résultat à $2t$:

$$S(x)\sigma(x) = \omega(x) \bmod \left(x^{2t}\right) \quad \textbf{(3-19)}$$

Cette expression est l'équation clé pour les codes de Reed – Solomon. Si le nombre d'erreurs v dans le mot-code transmis $c(x)$ est plus petit ou égal à t, l'équation clé a une seule paire de solutions $\sigma(x)$ et $\omega(x)$. Les deux degrés des polynômes doivent respecter la contrainte qui suit :

$$\deg\left(\omega(x)\right) < \deg\left(\sigma(x)\right) \leq t \quad \textbf{(3-20)}$$

L'équation clé peut être résolue selon l'algorithme d'Euclide en appliquant $r_0(x) = x^{2t}$ et $r_1(x) = S(x)$. Le calcul du théorème d'Euclide nous donnera comme solution le polynôme de localisation des erreurs et le polynôme d'amplitude. L'algorithme d'Euclide, pour le calcul du polynôme de localisation des erreurs et le polynôme d'amplitude, est montré dans la figure 3-9.

[10] Source : Error Control Coding, Second Edition, S. Lin et D.J. Costello, Prentice Hall
Source : Error-Control Block Codes, L.H. Charles LEE, Artech House Publishers

Diagramme en flèches de l'algorithme d'Euclide:

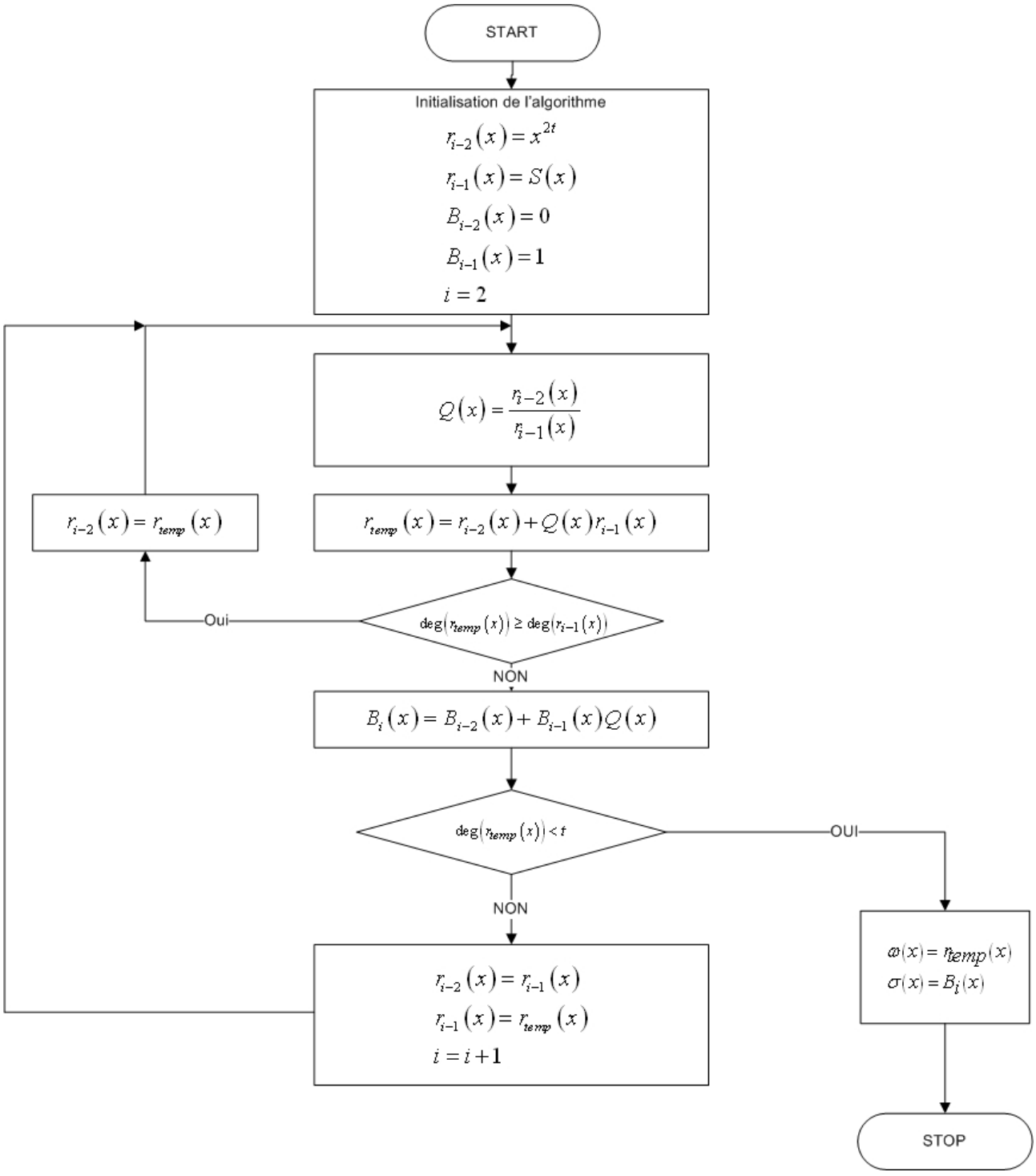

Figure 3-9: algorithme d'Euclide pour le calculant du polynôme de localisation et pour le polynôme d'amplitude

Le dernier reste de la division nous donnera le polynôme d'amplitude. Le polynôme de localisation des erreurs est donné selon la relation :

$$\sigma_i(x) = \sigma_{i-2}(x) + \sigma_{i-1}(x)Q_i(x) \qquad \textbf{(3-21)}$$

Avec :

$$\sigma_i(x) = B_i(x)$$

La théorie montre que l'on est obligé d'avoir deux blocs dans l'implémentation hardware. Un bloc qui effectue la division et qui donnera le polynôme d'amplitude des erreurs, et également un bloc de multiplication qui donnera le polynôme de localisation des erreurs.

3.4.4.3 Exemple de calcul

Considérons le code RS(15,9), comme on l'a calculé au chapitre 3.2.2, avec un mot-code $c(x) = 0$, un mot-code reçu $r(x) = \alpha x^{14} + \alpha^2 x^{12} + \alpha^{13} x^4$ et le syndrome calculé au chapitre 3.4.3.1. On cherche à calculer le polynôme de localisation des erreurs et le polynôme d'amplitude en utilisant l'algorithme d'Euclide.

On prend $r_0 = x^{2t} = x^6$ et $r_1(x) = \alpha^8 x^5 + \alpha x^4 + \alpha^{12} x^2 + \alpha^7 x + \alpha^6$

On commence à diviser $r_0(x)$ par $r_1(x)$:

$$\begin{array}{l|l}
x^6 + 0x^5 + 0x^4 + 0x^3 + 0x^2 + 0x + 0 & r_1(x) = \alpha^8 x^5 + \alpha x^4 + \alpha^{12} x^2 + \alpha^7 x + \alpha^6 \\
x^6 + \alpha^8 x^5 + 0x^4 + \alpha^4 x^3 + \alpha^{14} x^2 + \alpha^{13} x & \hline
\hline
\alpha^8 x^5 + 0x^4 + \alpha^4 x^3 + \alpha^{14} x^2 + \alpha^{13} x + 0 & \alpha^7 x + 1 \\
\alpha^8 x^5 + \alpha x^4 + 0x^3 + \alpha^{12} x^2 + \alpha^7 x + \alpha^6 & \\
\hline
\alpha x^4 + \alpha^4 x^3 + \alpha^5 x^2 + \alpha^5 x + \alpha^6 &
\end{array}$$

Comme $\deg(r_2) > t, \deg(r_2) > 3$ on continue l'algorithme. Le polynôme de localisation des erreurs intermédiaire sera :

$$\sigma_2 = \sigma_0(x) + \sigma_1(x)Q_2(x) = 0 + 1 * \alpha^7 x + 1 = \alpha^7 x + 1$$

La nouvelle opération à effectuer est $r_1(x)$ divisé par $r_2(x)$:

$$\begin{array}{l|l}
\alpha^8x^5+\alpha x^4+0x^3+\alpha^{12}x^2+\alpha^7x+\alpha^6 & \alpha x^4+\alpha^4x^3+\alpha^5x^2+\alpha^5x+\alpha^6 \\
\alpha^8x^5+\alpha^{11}x^4+\alpha^{12}x^3+\alpha^{12}x^2+\alpha^{13}x & \hline
\cline{1-1} & \alpha^7x+\alpha^5 \\
\alpha^6x^4+\alpha^{12}x^3+0x^2+\alpha^5x+\alpha^6 & \\
\alpha^6x^4+\alpha^9x^3+\alpha^{10}x^2+\alpha^{10}x+\alpha^{11} & \\
\cline{1-1} & \\
\alpha^8x^3+\alpha^{10}x^2+x+\alpha & \\
\end{array}$$

Comme $\deg(r_3)>t, \deg(r_3)>3$ on continue l'algorithme. Le polynôme de localisation des erreurs intermédiaire sera :

$$\sigma_3=\sigma_1(x)+\sigma_2(x)Q_3(x)=1+(\alpha^7x+1)(\alpha^7x+\alpha^5)=\alpha^{14}x^2+\alpha^2x+\alpha^{10}$$

La nouvelle opération à effectuer est $r_2(x)$ divisé par $r_3(x)$:

$$\begin{array}{l|l}
\alpha x^4+\alpha^4x^3+\alpha^5x^2+\alpha^5x+\alpha^6 & \alpha^8x^3+\alpha^{10}x^2+x+\alpha \\
\alpha x^4+\alpha^3x^3+\alpha^8x^2+\alpha^9x & \hline
\cline{1-1} & \alpha^8x+\alpha^{14} \\
\alpha^7x^3+\alpha^4x^2+\alpha^6x+\alpha^6 & \\
\alpha^7x^3+\alpha^9x^2+\alpha^{14}x+1 & \\
\cline{1-1} & \\
\alpha^{14}x^2+\alpha^8x+\alpha^{13} & \\
\end{array}$$

Comme $\deg(r_4(x))<t, \deg(r_4(x))<3$ on arrête l'algorithme. Le dernier reste de la division est le polynôme d'amplitude cherché :

$$\omega(x)=\alpha^{14}x^2+\alpha^8x+\alpha^{13}$$

Le polynôme de localisation des erreurs cherché est :

$$\sigma_4=\sigma_2(x)+\sigma_3(x)Q_4(x)=\alpha^7x+1+(\alpha^{14}x^2+\alpha^2x+\alpha^{10})(\alpha^8x+\alpha^{14})=\alpha^7x^3+\alpha^9x^2+x+\alpha^7$$

3.4.5 Berlekamp – Massey

3.4.5.1 Généralité de l'algorithme de Berlekamp – Massey [11]

De manière générale l'algorithme de Berlekamp – Massey permet de résoudre les identités de Newton de façon itérative. Dans le cadre des codes de Reed – Solomon, l'algorithme de Berlekamp – Massey permet de calculer le polynôme de localisation des erreurs.

L'identité de Newton à résoudre pour les codes de Reed – Solomon est :

$$\begin{aligned} S_{v+1} + \sigma_1 S_v + \sigma_2 S_{v-1} + \ldots + \sigma_v S_1 &= 0 \\ S_{v+2} + \sigma_1 S_{v+1} + \sigma_2 S_v + \ldots + \sigma_v S_2 &= 0 \\ \ldots \qquad\qquad\qquad\qquad & \ldots \\ S_{2t} + \sigma_1 S_{2t-1} + \sigma_2 S_{2t-2} + \ldots + \sigma_v S_{2t-v} &= 0 \end{aligned} \qquad \textbf{(3-22)}$$

En connaissant :

$$\begin{aligned} S_j &= \sum_{j=1}^{v} e_{j_i} \alpha^{j_i} \\ &= \sum_{j=1}^{v} \sigma_l S_{j-l} \end{aligned} \qquad \textbf{(3-23)}$$

On peut voir que cet algorithme est basé sur la synthèse du polynôme de localisation des erreurs en utilisant des registres à décalage pour la résolution des identités des Newton. En 1972 Massey développa un algorithme itératif basé sur le principe des registres à décalage de Berlekamp. Ce nouvel algorithme est appelé algorithme de Berlekamp – Massey et est défini selon le diagramme en flèches de la page suivante.

Registres à décalage :

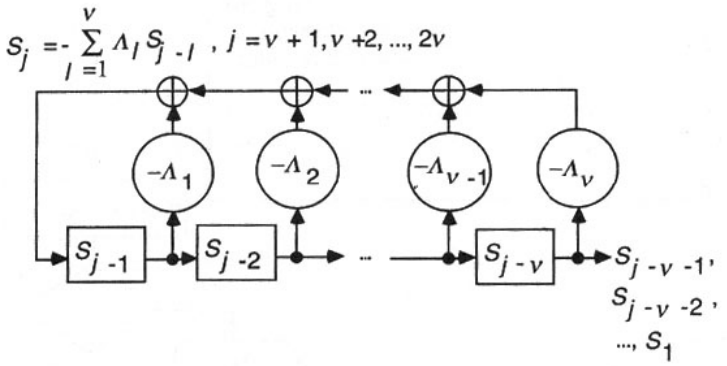

Figure 3-10: schéma de l'algorithme Berlekamp – Massey avec registres à décalage

[11] Source : Error Control Coding, Second Edition, S. Lin et D.J. Costello, Prentice Hall
Source : Error-Control Block Codes, L.H. Charles LEE, Artech House Publishers

Connaissant les syndromes, on peut appliquer le diagramme en flèches ci-dessous pour le calcul de $\sigma(x)$:

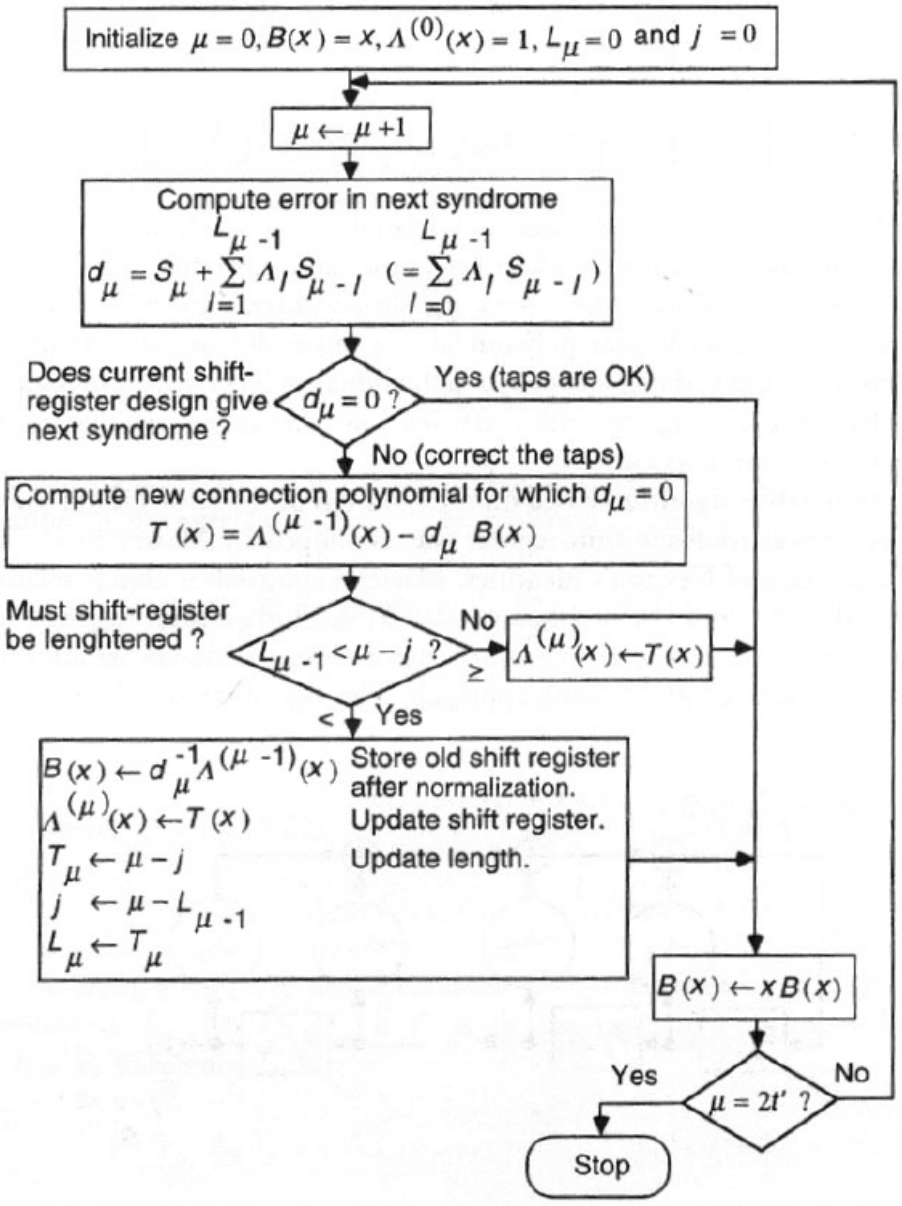

Figure 3-11: diagramme en flèches de l'algorithme de Berlekamp – Massey [12]

Avec :

$\Lambda^{(\mu)} = \sigma^{(\mu)}(x)$	: polynôme de localisation des erreurs après μ étapes
L_μ	: degré du polynôme de localisation des erreurs après μ étapes
d_μ	: incohérence après μ étapes
$T(x)$	: polynôme de connexion annulant l'incohérence

[12] Source : Error-Control Block Codes, L.H. Charles LEE, Artech House Publishers.

3.4.5.2 Correction d'erreurs avec Berlekamp-Massey[13]

Le polynôme localisateur est définit comme :

$$\begin{aligned}\sigma(x) &= \prod_{i=1}^{v}\left(1+\alpha^{j_i}x\right) \\ &= 1+\sigma_1 x+\sigma_2 x^2+\ldots+\sigma_v x^v\end{aligned} \qquad \textbf{(3-24)}$$

Avec :

$j_1, j_2, \ldots, j_v$: indiquant l'emplacement des erreurs

$\sigma_1, \sigma_2, \ldots, \sigma_v$: coefficients du polynôme de localisation des erreurs

A partir des relations (3-11) et (3-24), il est possible de comprendre comment le syndrome et le polynôme de localisation des erreurs sont liés par l'identité de Newton :

$$\begin{aligned} S_{v+1}+\sigma_1 S_v+\sigma_2 S_{v-1}+\ldots+\sigma_v S_1 &= 0 \\ S_{v+2}+\sigma_1 S_{v+1}+\sigma_2 S_v+\ldots+\sigma_v S_2 &= 0 \\ \ldots \qquad\qquad\qquad\qquad & \ldots \\ S_{2t}+\sigma_1 S_{2t-1}+\sigma_2 S_{2t-2}+\ldots+\sigma_v S_{2t-v} &= 0\end{aligned} \qquad \textbf{(3-25)}$$

Le but de l'algorithme de Berlekamp-Massey est de trouver le polynôme $\sigma(x)$ de degré le plus petit possible et satisfaisant l'identité de Newton. Le calcul de $\sigma(x)$ est effectué itérativement avec $2t$ étapes selon le diagramme en flèches de la figure 3-11.

Le polynôme calculé sera sous la forme :

$$\sigma(x) = \sigma_0+\sigma_1 x+\ldots+\sigma_t x^t \qquad \textbf{(3-26)}$$

La relation (3-12) nous indique :

$$\begin{aligned}S(x) &= \ldots+S_{2t+1}x^{2t}+S_{2t}x^{2t-1}+\ldots+S_2 x+S_1 \\ &= \sum_{j=1}^{\infty} S_j x^{j-1}\end{aligned} \qquad \textbf{(3-27)}$$

En sachant que :

$$S_j = \sum_{j=1}^{v} e_{j_i}\alpha^{j_i} \qquad \textbf{(3-28)}$$

[13] Source : Error Control Coding, Second Edition, S. Lin et D.J. Costello, Prentice Hall
Source : Error-Control Block Codes, L.H. Charles LEE, Artech House Publishers

On peut alors affirmer que le polynôme d'amplitude[14] sera de degré $t-1$:

$$\begin{aligned}\omega(x) &= \left[S(x)\sigma(x)\right] \bmod \left(x^t\right) \\ &= \left[\left(S_{2t}x^{2t-1}+\ldots+S_2x+S_1\right)\left(\sigma_t x^t+\ldots+\sigma_1 x+\sigma_0\right)\right] \bmod \left(x^t\right) \\ &= \left(S_t+\sigma_1 S_{t-1}+\ldots\right)x^{t-1}+\ldots+\left(S_2+\sigma_1 S_1\right)x+S_1\end{aligned} \quad \textbf{(3-29)}$$

3.4.5.2.1 Exemple

Reprenons le code RS(15,9) calculé chapitre 3.2.2 avec un mot-code $c(x)=0$, un mot-code reçu $r(x)=\alpha x^{14}+\alpha^2 x^{12}+\alpha^{13}x^4$ et le syndrome calculé au chapitre 3.4.3.1. Cette fois, on veut calculer le polynôme de localisation des erreurs et le polynôme d'amplitude en utilisant l'algorithme de Berlekamp – Massey.

Le polynôme de localisation des erreurs est calculé selon le diagramme en flèches, au chapitre 3.4.5.1. On développera seulement les trois premières itérations puisque le calcul est identique par la suite.

Initialisation de la machine :

$\mu=0$

$B(x)=x \qquad \sigma_0(x)=1 \qquad L_\mu=0 \qquad j=0$

Etape 1 :

$\mu=1$

$d_1=S_1+\sum_{l=1}^{L_{\mu-1}}\sigma_l S_{\mu-l}=S_1+\sum_{l=1}^{0}\sigma_l S_{\mu-l}=S_1=\alpha^6$

$T(x)=\sigma_{\mu-1}(x)-d_\mu B(x)=\sigma_0(x)+d_1B(x)=1+\alpha^6 x$

$L_{\mu-1}<\mu-j\,??\rightarrow 0<1-0\rightarrow 0<1 \;\Rightarrow OUI$

$B(x)=d_{\mu-1}^{-1}\sigma_{\mu-1}(x)=d_1^{-1}\sigma_0(x)=\alpha^{-6}=\alpha^9$

$\sigma_1(x)=T(x)=1+\alpha^6 x$

$T_1=\mu-j=1-0=1$

$j=\mu-L_0=1-0=1$

$L_1=T_1=1$

$B(x)=xB(x)=\alpha^9 x$

[14] Source : Error Control Coding, Second Edition, S. Lin et D.J. Costello, Prentice Hall

Etape 2 :

$$\mu = 1$$

$$d_2 = S_2 + \sum_{l=1}^{L_{\mu-1}} \sigma_l S_{\mu-l} = S_2 + \sum_{l=1}^{1} \sigma_l S_{\mu-l} = S_2 + \sigma_1 S_1 = \alpha^7 + \alpha^6 \alpha^6 = \alpha^2$$

$$T(x) = \sigma_{\mu-1}(x) - d_\mu B(x) = \sigma_1(x) + d_2 B(x) = 1 + \alpha^6 x + \alpha^2 (\alpha^9 x) = 1 + \alpha x$$

$$L_{\mu-1} < \mu - j \,??\, \rightarrow 1 < 2 - 1 \rightarrow 1 < 1 \quad \Rightarrow NON$$

$$\sigma_2(x) = T(x) = 1 + \alpha x$$

$$j = \mu - L_1 = 1 - 0 = 1$$

$$L_2 = L_1 = 1$$

$$B(x) = xB(x) = \alpha^9 x^2$$

Résumé des calculs:

μ	d_μ	$B(x)$	$\sigma_\mu(x)$	j	L_μ
0	-	x	1	0	0
1	α^6	$\alpha^9 x$	$\alpha^6 x + 1$	1	1
2	α^2	$\alpha^9 x^2$	$\alpha x + 1$	1	1
3	α^9	$\alpha^7 x^2 + \alpha^6 x$	$\alpha^3 x^2 + \alpha x + 1$	2	2
4	α^9	$\alpha^7 x^3 + \alpha^6 x^2$	$\alpha^9 x^2 + \alpha^4 x + 1$	2	2
5	α^{11}	$\alpha^{13} x^3 + \alpha^8 x^2 + \alpha^4 x$	$\alpha^3 x^3 + \alpha^{11} x^2 + \alpha^4 x + 1$	3	3
6	α	$\alpha^{13} x^4 + \alpha^8 x^3 + \alpha^4 x^2$	$x^3 + \alpha^2 x^2 + \alpha^8 x + 1$	3	3

Tableau 3-2: calcul du polynôme de localisation des erreurs

Une fois le polynôme de localisation des erreurs calculé:

$$\sigma(x) = x^3 + \alpha^2 x^2 + \alpha^8 x + 1$$

Reste à calculer le polynôme d'amplitude selon la relation (3-29) :

$$\begin{aligned}
\omega(x) &= \left[S(x)\sigma(x) \right] \bmod (x^t) \\
&= \left[\left(S_1 + S_2 x + \ldots + S_{2t} x^{2t-1} \right) \left(\sigma_0 + \sigma_1 x + \ldots + \sigma_t x^t \right) \right] \bmod (x^t) \\
&= \left(S_3 + \sigma_1 S_2 + \sigma_2 S_1 \right) x^2 + \left(S_2 + \sigma_1 S_1 \right) x + S_1 \\
&= \left(\alpha^{12} + 1 + \alpha^8 \right) x^2 + \left(\alpha^7 + \alpha^{14} \right) x + \alpha^6 \\
&= \alpha^7 x^2 + \alpha x + \alpha^6
\end{aligned}$$

3.4.6 Chien Search

Une fois le polynôme de localisation des erreurs calculé, on doit évaluer ses racines et sa dérivée. La dérivée formelle d'un polynôme dans $GF\left(2^m\right)$ est définie au chapitre 2.6.

L'évaluation des racines est effectuée avec l'algorithme appelé « Chien Search » qui est du type « brute force », c'est-à-dire, qu'il évalue toutes les possibilités. Par exemple pour un RS(15,9), on évalue le polynôme de localisation des erreurs et sa dérivée pour tous les éléments du « champ de Galois » $GF\left(2^4\right)$, sauf pour l'élément nul.

A la sortie de ce bloc, on obtiendra une séquence de symboles. Lorsque les symboles sont nuls, ceux-ci nous indiqueront qu'une racine a été détectée.

3.4.6.1 Schéma du Chien Search

Ce schéma permet de calculer les racines pour un polynôme de localisation des erreurs et pour sa dérivée (dérivation dans $GF\left(2^m\right)$ voir 2.6). Le schéma de la figure (3-12) calcule les racines d'un code de RS(15,9), donc en $GF\left(2^4\right)$.

Pendant n coups d'horloge, on évaluera le polynôme de localisation des erreurs et son polynôme dérivé. Chaque coup d'horloge indiquera une valeur différente de α^i, où i indique le numéro du coup d'horloge.

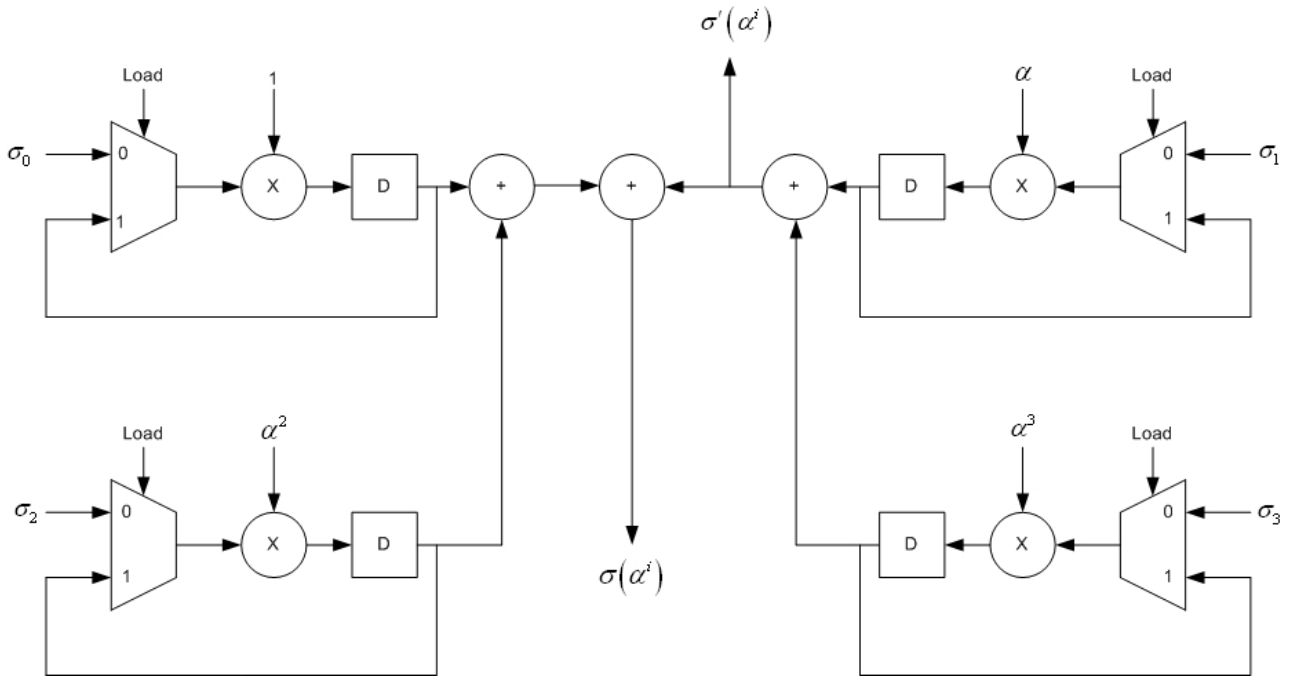

Figure 3-12: schéma du bloc Chien Search

3.4.6.2 Exemple

Le calcul des racines du polynôme de localisation des erreurs est établi au chapitre 3.4.4.3 selon « Chien Search ». Premièrement, on calculera les racines en substituant x pour les différents éléments de $GF\left(2^4\right)$, car on utilise un code RS(15,9). Ensuite, on testera le schéma pour comprendre son fonctionnement.

Soit le polynôme de localisation des erreurs suivant :

$$\sigma(x) = \alpha^7 x^3 + \alpha^9 x^2 + x + \alpha^7$$

On doit essayer tous les éléments possibles de $GF\left(2^4\right)$.

$$\sigma\left(\alpha^i\right) = \alpha^7 \left(\alpha^i\right)^3 + \alpha^9 \left(\alpha^i\right)^2 + \left(\alpha^i\right) + \alpha^7$$

Calcul des racines ; on montrera seulement les 4 premiers cas car après les calculs se répètent:

$$\begin{aligned}\sigma(\alpha) &= \alpha^7 (\alpha)^3 + \alpha^9 (\alpha)^2 + (\alpha) + \alpha^7 \\ &= \alpha^{10} + \alpha^{11} + \alpha + \alpha^7 \\ &= 0\end{aligned}$$

$$\begin{aligned}\sigma\left(\alpha^2\right) &= \alpha^7 \left(\alpha^2\right)^3 + \alpha^9 \left(\alpha^2\right)^2 + \left(\alpha^2\right) + \alpha^7 \\ &= \alpha^{13} + \alpha^{13} + \alpha^2 + \alpha^7 \\ &= \alpha^{12}\end{aligned}$$

$$\begin{aligned}\sigma\left(\alpha^3\right) &= \alpha^7 \left(\alpha^3\right)^3 + \alpha^9 \left(\alpha^3\right)^2 + \left(\alpha^3\right) + \alpha^7 \\ &= \alpha + 1 + \alpha^3 + \alpha^7 \\ &= 0\end{aligned}$$

$$\begin{aligned}\sigma\left(\alpha^4\right) &= \alpha^7 \left(\alpha^4\right)^3 + \alpha^9 \left(\alpha^4\right)^2 + \left(\alpha^4\right) + \alpha^7 \\ &= \alpha^4 + \alpha^2 + \alpha^4 + \alpha^7 \\ &= \alpha^{12}\end{aligned}$$

Résumé des calculs :

Elément	Résultat
α	0
α^2	α^{12}
α^3	0
α^4	α^{12}
α^5	α^8
α^6	α^6
α^7	α^3
α^8	α^7
α^9	α^{13}
α^{10}	α^{11}
α^{11}	0
α^{12}	1
α^{13}	α
α^{14}	α^9
1	α^7

Tableau 3-3: racines du polynôme de localisation des erreurs

Les racines peuvent être calculées selon le schéma de la figure 3-13. Dans le tableau 3-4, on montrera les résultats du calcul selon le schéma.

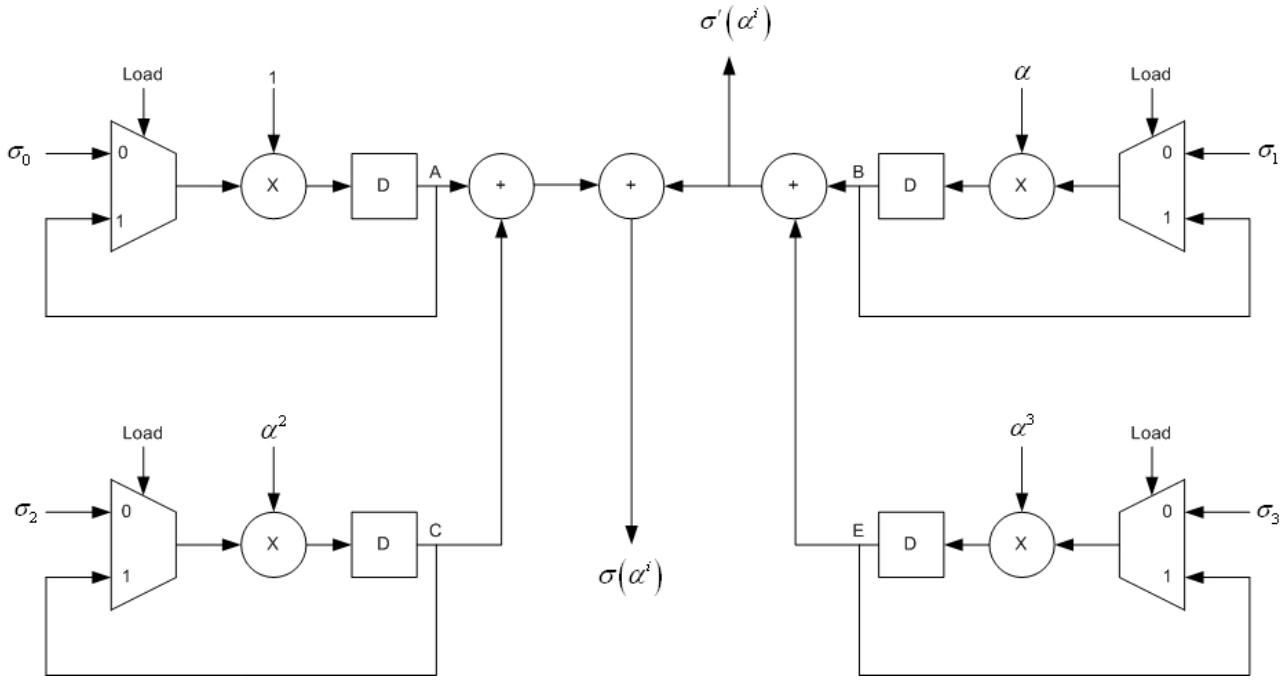

Figure 3-13: schéma avec signaux détaillés pour l'évaluation du polynôme de localisation et sa dérivée

Calcul des racines selon le schéma :

i	A	B	C	E	A+C	B+E	$\sigma'(\alpha^i)$	$\sigma(\alpha^i)$
0	0	0	0	0	0	0	0	0
α	α^7	α	α^{11}	α^{10}	α^8	α^8	α^8	0
α^2	α^7	α^2	α^{13}	α^{13}	α^5	α^{14}	α^{14}	α^{12}
α^3	α^7	α^3	1	α	α^9	α^9	α^9	0
α^4	α^7	α^4	α^2	α^4	α^{12}	0	0	α^{12}
α^5	α^7	α^5	α^4	α^7	α^3	α^{13}	α^{13}	α^8
α^6	α^7	α^6	α^6	α^{10}	α^{10}	α^7	α^7	α^6
α^7	α^7	α^7	α^8	α^{13}	α^{11}	α^5	α^5	α^3
α^8	α^7	α^8	α^{10}	α	α^6	α^{10}	α^{10}	α^7
α^9	α^7	α^9	α^{12}	α^4	α^2	α^{14}	α^{14}	α^{13}
α^{10}	α^7	α^{10}	α^{14}	α^7	α	α^6	α^6	α^{11}
α^{11}	α^7	α^{11}	α	α^{10}	α^{14}	α^{14}	α^{14}	0
α^{12}	α^7	α^{12}	α^3	α^{13}	α^4	α	α	1
α^{13}	α^7	α^{13}	α^5	α	α^{13}	α^{12}	α^{12}	α
α^{14}	α^7	α^{14}	α^7	α^4	0	α^9	α^9	α^9
1	α^7	1	α^9	α^7	1	α^9	α^9	α^7

Tableau 3-4: calcul des racines du polynôme de localisation des erreurs

3.4.7 Algorithme de Forney

Cet algorithme permet de construire le polynôme d'erreurs $e(x)$ à additionner avec le polynôme reçu $r(x)$ pour reconstituer le polynôme $c(x)$. Pour le calcul du polynôme $e(x)$, les polynômes $\sigma(\alpha^i), \sigma'(\alpha^i), \omega(\alpha^i)$ sont nécessaires. Le polynôme de localisation des erreurs et sa dérivée sont déjà évalués pour les différentes valeurs de α, il nous reste à évaluer $\omega(\alpha^i)$.

Une fois les différentes valeurs de $\omega(\alpha^i)$ calculées, on applique l'algorithme de Forney. Cet algorithme est défini comme :

$$e_i = \frac{\omega(\alpha^i)}{\sigma'(\alpha^i)} \qquad \textbf{(3-30)}$$

Avec :

$\omega(\alpha^i)$: polynôme d'amplitude évalué pour les valeurs de $GF(2^4)$

$\sigma'(\alpha^i)$: dérivée du polynôme de localisation des erreurs pour les valeurs de $GF(2^4)$

3.4.7.1 Schéma de l'algorithme de Forney et de la correction des erreurs

Le schéma pour le calcul de l'algorithme de Forney et pour la correction des erreurs est :

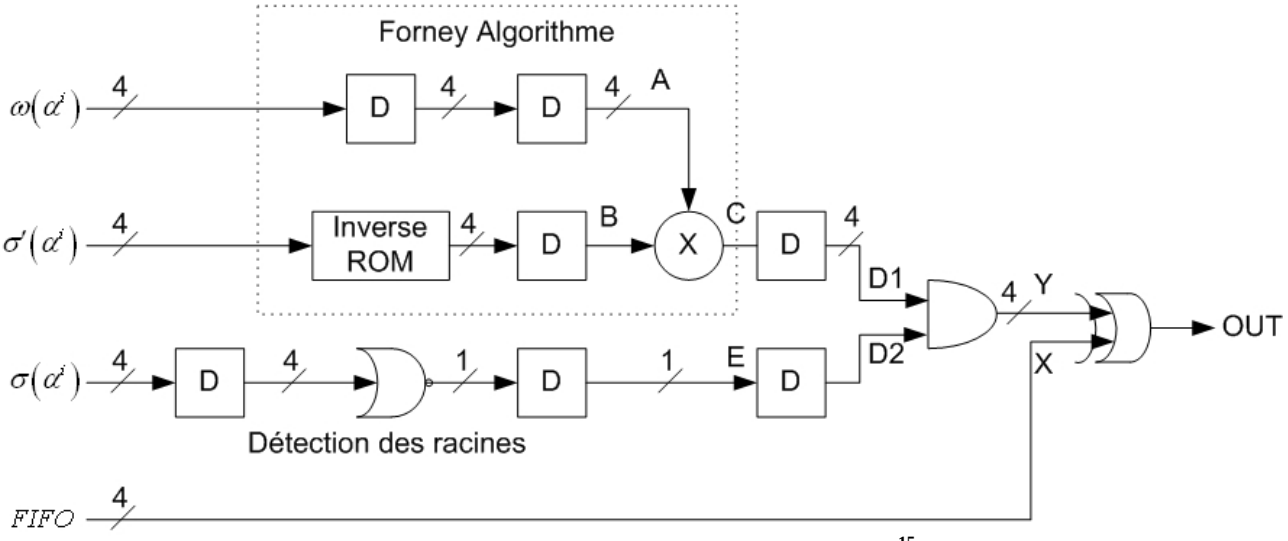

Figure 3-14: schéma de l'algorithme de Forney[15]

[15] Source : Reed – Solomon decoder for HDTV application, E. Aleman, E. Galema, M. Kettledon

3.4.7.1.1 Evaluation du polynôme d'amplitude

L'évaluation du polynôme d'amplitude $\omega(\alpha^i)$ peut être effectuée avec un schéma semblable à celui du « Chien Search ». Le schéma de la figure 3-15 évalue le polynôme d'amplitude pour un code RS(15,9).

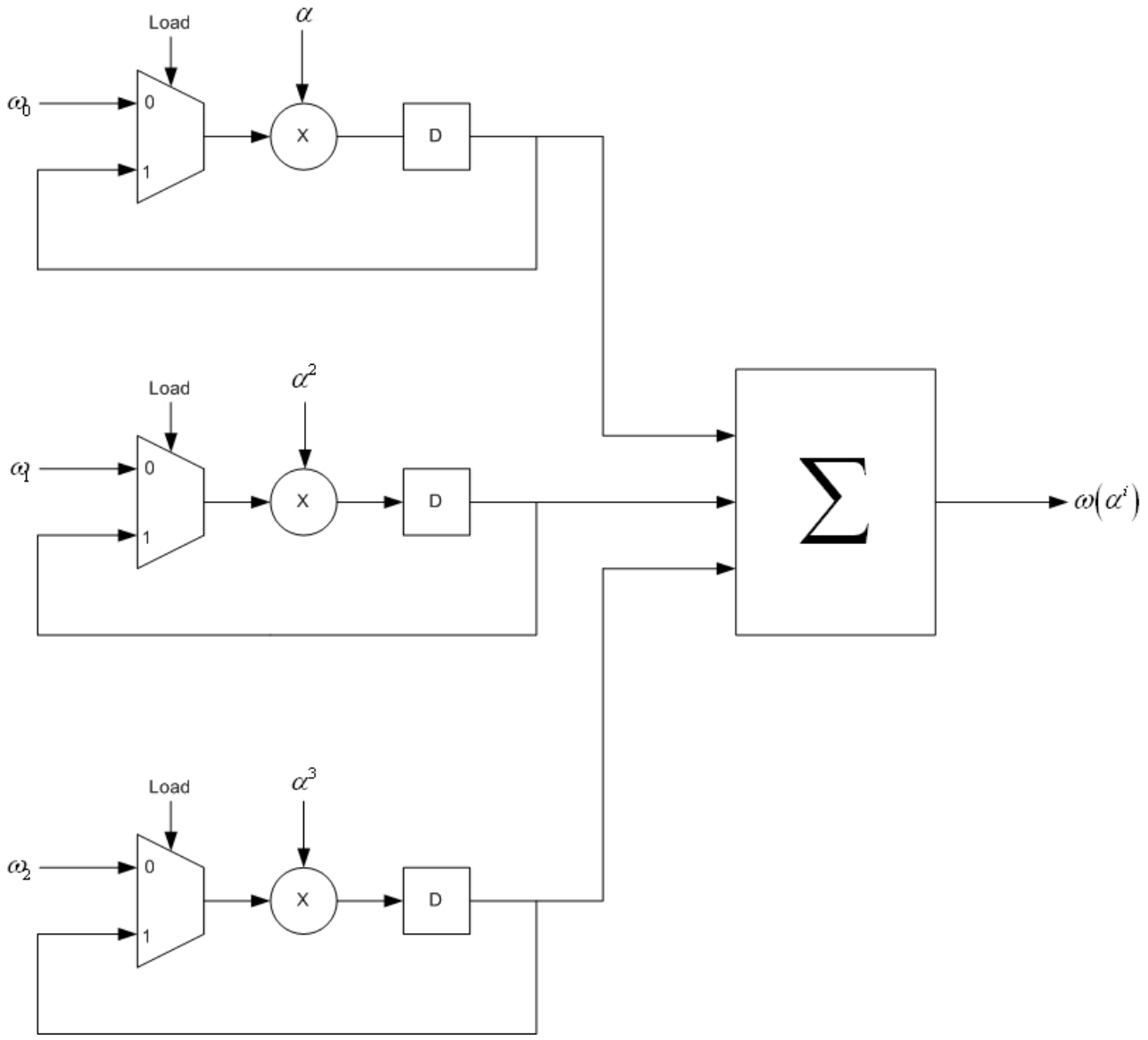

Figure 3-15: schéma pour le calcul des racines du polynôme d'amplitude

3.4.7.1.1.1 Exemple

On prend comme exemple le schéma de la figure 3-15 pour un code RS(15,9). Le but étant d'évaluer le polynôme d'amplitude :

$$\omega(x) = \alpha^{14}x^2 + \alpha^8 x + \alpha^{13}$$

On évalue le polynôme d'amplitude selon le schéma de la figure 3-15.

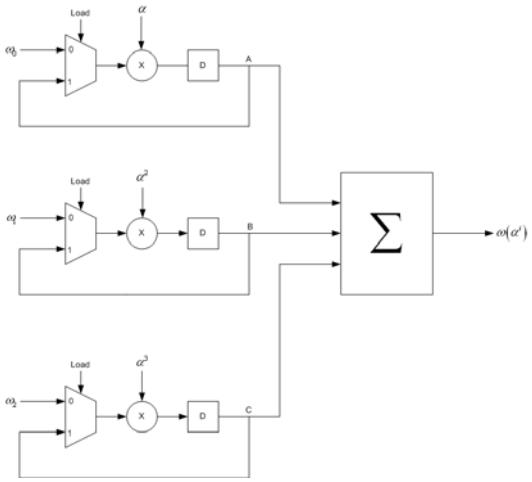

Figure 3-16: schéma avec signaux détaillés pour l'évaluation du polynôme d'amplitude

Résumé des calculs :

A	B	C	$\omega(\alpha^i)$
0	0	0	0
α^{14}	α^{10}	α^2	α^9
1	α^{12}	α^5	α^3
α	α^{14}	α^8	α^{11}
α^2	α	α^{11}	α^3
α^3	α^3	α^{14}	α^{14}
α^4	α^5	α^2	1
α^5	α^7	α^5	α^7
α^6	α^9	α^8	α^4
α^7	α^{11}	α^{11}	α^7
α^8	α^{13}	α^{14}	1
α^9	1	α^2	α^{12}
α^{10}	α^2	α^5	α^8
α^{11}	α^4	α^8	α^3
α^{12}	α^6	α^{11}	α^{13}
α^{13}	α^8	α^{14}	1

Tableau 3-5: évaluation du polynôme d'amplitude

3.4.7.1.2 Inversion avec ROM

Le calcul de l'algorithme de Forney requiert une division entre deux valeurs, $e_i = \frac{\omega(\alpha^i)}{\sigma'(\alpha^i)}$.

Une division peut être calculée en faisant une multiplication par l'élément inverse du dénominateur. On crée un ROM avec tous les éléments inverses de $GF(2^m)$ de manière à pouvoir effectuer la division avec une multiplication par un symbole inverse.

Le calcul de l'inverse dans un « champ de Galois » est défini comme suit :

$$\alpha^{-i} = \alpha^{element\_\max - i} \qquad \textbf{(3-31)}$$

3.4.7.1.2.1 Exemple des éléments inverses pour un $GF(2^4)$

En sachant que les éléments non nuls dans $GF(2^4)$ sont au nombre de 15, on peut calculer tous les éléments inverses.

Le calcul des inverses est effectué selon l'équation (3-31) :

$$\alpha^{-1} = \alpha^{15-1} = \alpha^{14}$$

$$\alpha^{-2} = \alpha^{15-2} = \alpha^{13}$$

...

Résumé des calculs :

Elément	Inverse
0	0
1	1
α^1	α^{14}
α^2	α^{13}
α^3	α^{12}
α^4	α^{11}
α^5	α^{10}
α^6	α^9
α^7	α^8
α^8	α^7
α^9	α^6
α^{10}	α^5
α^{11}	α^4
α^{12}	α^3
α^{13}	α^2
α^{14}	α

Tableau 3-6: tableau d'inversion pour $GF(2^4)$

3.4.7.1.3 Multiplication

La multiplication entre la valeur inverse de $\sigma'(\alpha^i)$ et $\omega(\alpha^i)$ nous donne la valeur de e_i.

$$e_i = \frac{\omega(\alpha^i)}{\sigma'(\alpha^i)} = \omega(\alpha^i)\sigma'(\alpha^{-i}) \qquad \textbf{(3-32)}$$

Effectuer une multiplication au lieu d'une division sert à économiser des coups d'horloge et à simplifier les circuits. La division faite à l'aide d'une multiplication inversée requiert un coup d'horloge, pour la lecture dans la mémoire, puis la multiplication est effectuée avec de la logique combinatoire, comme vu dans le paragraphe 3.3.3.1.2.

3.4.7.1.4 Détection du zéro

La détection du zéro est effectuée avec une porte logique « NOR ». Lorsque l'on aura un élément nul à l'entrée, donc une racine, on aura un « 1 » à la sortie. Cette détection sert uniquement à sélectionner les éléments pour la correction des erreurs.

3.4.7.1.5 La porte logique « AND »

La porte logique « AND » sert à sélectionner seulement les symboles qui devraient être corrigés. De cette façon on éliminera les erreurs du mot-code reçu.

3.4.7.1.6 La porte logique « XOR »

L'addition modulo 2 donnera toujours le symbole du polynôme reçu lorsque la valeur de $e_i = 0$. Quand la valeur de $e_i \neq 0$, on additionnera le symbole de e_i et de r_i pour éliminer l'erreur.

3.4.7.2 Exemple de calcul selon le schéma de Forney

On veut calculer le polynôme de correction selon le schéma de Forney pour le polynôme de localisation des erreurs et pour le polynôme d'amplitude du chapitre 3.4.4.3. Pour ce faire, on a fixé des points sur le schéma, de manière à mieux repérer les signaux en question.

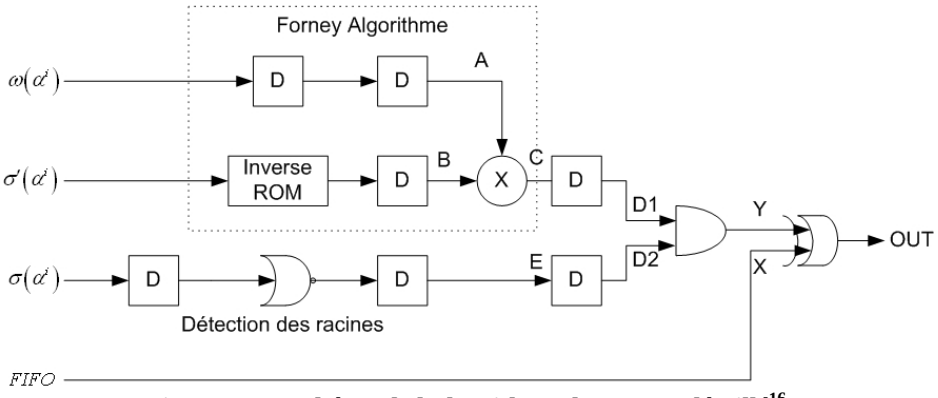

Figure 3-17: schéma de l'algorithme de Forney détaillé[16]

Les valeurs de $\omega(\alpha^i), \sigma(\alpha^i), \sigma'(\alpha^i)$ sont les valeurs calculées dans les exemples précédents, paragraphes 3.4.7.1.1.1 , 3.4.6.2.

Le tableau ci-dessous montre les différentes étapes :

i	A	B	C	E	D1	D2	X	Y	Out
0	α^9	α^7	α	1	0	0	-	0	-
1	α^3	α	α^4	0	α	1	α	α	0
2	α^{11}	α^6	α^2	1	α^4	0	0	0	0
3	α^3	0	0	0	α^2	1	α^2	α^2	0
4	α^{14}	α^2	α	0	0	0	0	0	0
5	1	α^8	α^8	0	α	0	0	0	0
6	α^7	α^{14}	α^6	0	α^8	0	0	0	0
7	α^4	α^5	α^9	0	α^6	0	0	0	0
8	α^7	α	α^8	0	α^9	0	0	0	0
9	1	α^9	α^9	0	α^8	0	0	0	0
10	α^{12}	α	α^{13}	1	α^9	0	0	0	0
11	α^8	α^{14}	α^7	0	α^{13}	1	α^{13}	α^{13}	0
12	α^3	α^3	α^6	0	α^7	0	0	0	0
13	α^{13}	α^6	α^4	0	α^6	0	0	0	0
14	1	α^6	α^6	0	α^4	0	0	0	0
15	-	-	-	-	α^6	0	0	0	0

Tableau 3-7: calcul de l'algorithme de Forney selon schéma

[16] Source : Reed – Solomon decoder for HDTV application, E. Aleman, E. Galema, M. Kettledon

3.5 Correction des erreurs et des effacements

Les codes de Reed – Solomon sont non seulement utilisés pour la correction des erreurs (chapitre 3.4), mais permettent aussi de corriger les effacements.

Un effacement[17] suit le même principe que lorsqu'on efface une lettre dans un mot à l'aide d'un effaceur. La lettre effacée dans le mot n'est pas connue, mais la position de celle-ci l'est. Les codes de Reed – Solomon permettent de corriger deux fois plus d'effacements que d'erreurs.

La séquence de décodage est presque la même que celle utilisée pour la correction des erreurs, la seule différence est qu'avant de calculer les syndromes, on doit substituer dans le polynôme reçu $r(x)$ les effacements avec des « 0 » avant de procéder au calcul du syndrome lui même. La première opération à effectuer pour le décodage des erreurs et des effacements, est l'évaluation du polynôme de localisation des effacements.

L'équation clé comme pour le décodage simple des erreurs peut être résolue selon différents algorithmes, ici on n'en traitera que deux :

1. Algorithme d'Euclide (chapitre 3.4.4)
2. Algorithme de Berlekamp – Massey (chapitre 3.4.5)

3.5.1 Capacité de correction

Comme vu dans le chapitre précédent, la capacité de correction des erreurs d'un code de Reed – Solomon est au maximum de :

$$v \le t \qquad \textbf{(3-33)}$$

Cette règle n'est plus valable lorsqu'on doit aussi corriger des effacements, car on devrait ajouter f effacements aux v erreurs. En supposant que le polynôme des erreurs ait $\eta = v + f$ non nuls valeurs aux positions $j_1, j_2 .... j_n$, le polynôme des erreurs sera défini comme:

$$e(x) = e_{j_\eta} x^{j_\eta} + e_{j_{\eta-1}} x^{j_{\eta-1}} + ... + e_{j_2} x^{j_2} + e_{j_1} x^{j_1} \qquad \textbf{(3-34)}$$

Avec :

$0 \le j_1 < j_2 ... < j_\eta < \eta$ et $2v + f \le t$

[17] Source : ECC Technologies, faqs

3.5.2 Résolution selon l'algorithme d'Euclide[18]

La définition du polynôme de localisation reste la même qu'au chapitre 3.4.4.2, on peut donc définir le polynôme de localisation des erreurs comme :

$$\begin{aligned}\sigma(x) &= \prod_{k=1}^{v}\left(1-\alpha^{i_k}x\right) \\ &= \sigma_v x^v + \sigma_{v-1}x^{v-1} + \ldots + \sigma_1 x + 1\end{aligned} \qquad \textbf{(3-35)}$$

Avec :

v : nombre d'erreurs dans le polynôme reçu $r(x)$

Comme la position des effacements est connue, le polynôme des effacements est défini comme :

$$\begin{aligned}\beta(x) &= \prod_{l=1}^{f}\left(1-\alpha^{i_l}x\right) \\ &= \beta_f x^f + \beta_{f-1}x^{f-1} + \ldots + \beta_1 x + 1\end{aligned} \qquad \textbf{(3-36)}$$

Avec :

f : nombre d'effacements dans le polynôme reçu $r(x)$

Le polynôme de localisation des effacements est défini de manière à ce que l'on puisse substituer les symboles effacés par des symboles aléatoires dans le polynôme reçu $r(x)$. Mais pour faciliter la tâche, on utilisera toujours le symbole nul « 0 ». Car la substitution des symboles effacés par des symboles aléatoires introduit des erreurs.

En mettant ensemble le polynôme de localisation des erreurs et le polynôme des effacements, on obtient un nouveau polynôme appelé, polynôme de localisation des erreurs et des effacements. Ce polynôme est défini comme :

$$\gamma(x) = \sigma(x)\beta(x) \qquad \textbf{(3-37)}$$

Avec :

$\gamma(x)$: polynôme de localisation des erreurs et des effacements

$\sigma(x)$: polynôme de localisation des erreurs, inconnu à ce stade

$\beta(x)$: polynôme de localisation des effacements, connu à ce stade

[18] Source : Error-Control Block Codes, L.H. Charles LEE, Artech House Publishers

On calcule le nouveau polynôme du syndrome avec les symboles effacés remplacés par des symboles nuls :

$$S(x) = S_{2t}x^{2t-1} + ... + S_2 x + S_1 \qquad \textbf{(3-38)}$$

La nouvelle équation clé à résoudre sera :

$$\sigma(x)\beta(x)S(x) = \omega(x) \bmod \left(x^{2t}\right) \qquad \textbf{(3-39)}$$

Avec :

$\omega(x)$: polynôme d'amplitude, inconnu à ce stade

$\sigma(x)$: polynôme de localisation des erreurs, inconnu à ce stade

$\beta(x)$: polynôme de localisation des effacements, connu à ce stade

$S(x)$: syndrome modifié, connu à ce stade

Le problème cette fois est de calculer $\sigma(x)$ et $\omega(x)$ de façon à ce que $\sigma(x)$ ait le plus petit degré v et que $\deg(\omega(x)) < v + f$.

Etant donné qu'on connaît $\beta(x)$ et $S(x)$, on peut les assembler pour former le nouveau polynôme $\Xi(x)$:

$$\Xi(x) = \left[\beta(x)S(x)\right] \bmod \left(x^{2t}\right)$$

Ici, on peut appliquer l'algorithme d'Euclide pour $r_0(x) = x^{2t}$ et $r_1(x) = \Xi(x)$. L'algorithme d'Euclide est appliqué jusqu'à :

$$\deg r_i(x) \leq \begin{cases} t + \dfrac{f}{2} & \text{pour } f \text{ pair} \\ t + \dfrac{f-1}{2} & \text{pour } f \text{ impair} \end{cases} \qquad \textbf{(3-40)}$$

Avec la même technique qu'au chapitre 3.4.7, on peut évaluer l'amplitude des erreurs et des effacements selon la relation :

$$e_{j_i} = \frac{\omega\left[\left(\alpha^i\right)\right]}{\gamma'\left[\left(\alpha^i\right)\right]} \qquad \textbf{(3-41)}$$

3.5.2.1 Résumé des opérations pour le décodage selon le théorème d'Euclide[19]

1. Evaluation du polynôme des effacements $\beta(x)$
2. Modifier le polynôme reçu $r(x)$ en substituant les valeurs effacées par des zéros et calculer le syndrome
3. Evaluation du syndrome modifié $\Xi(x) = \left[S(x)\beta(x) \right] \bmod \left(x^{2t} \right)$
4. Application du théorème d'Euclide avec $r_0(x) = x^{2t}$ et $r_1(x) = \Xi(x)$ pour le calcul du polynôme de localisation des erreurs $\sigma(x)$ et pour le polynôme d'amplitude $\omega(x)$
5. Calcul des racines selon « Chien Search » du polynôme des erreurs et des effacements $\gamma(x) = \sigma(x)\beta(x)$
6. Evaluation des amplitudes des erreurs et des effacements associés avec le polynôme de localisation des erreurs et des effacements. Ce calcul donnera le polynôme d'erreurs $e(x)$
7. Soustraction du polynôme $e(x)$ au polynôme reçu $r(x)$ pour la correction des erreurs et des effacements

[19] Source : Error-Control Block Codes, L.H. Charles LEE, Artech House Publishers

3.5.3 Résolution selon l'algorithme de Berlekamp – Massey[20]

La définition du polynôme de localisation reste la même qu'au chapitre 3.4.5.2, on peut donc définir le polynôme de localisation des erreurs par :

$$\begin{aligned}\sigma(x) &= \prod_{k=1}^{v}\left(1-\alpha^{i_k}x\right) \\ &= \sigma_v x^v + \sigma_{v-1}x^{v-1} + \ldots + \sigma_1 x + 1\end{aligned} \qquad \textbf{(3-42)}$$

Avec :

v : nombre d'erreurs dans le polynôme reçu $r(x)$

Comme la position des effacements est connue, le polynôme des effacements est défini comme :

$$\begin{aligned}\beta(x) &= \prod_{l=1}^{f}\left(1-\alpha^{i_l}x\right) \\ &= \beta_f x^f + \beta_{f-1}x^{f-1} + \ldots + \beta_1 x + 1\end{aligned} \qquad \textbf{(3-43)}$$

Avec :

f : nombre d'effacements dans le polynôme reçu $r(x)$

Le polynôme de localisation des effacements est défini de manière à ce qu'on puisse substituer les symboles effacés par des symboles aléatoires dans le polynôme reçu $r(x)$. Mais pour plus de commodités, on utilisera toujours le symbole nul « 0 », car la substitution des symboles effacés par des symboles aléatoires introduit des erreurs.

Le polynôme d'amplitude sera donc défini par :

$$\gamma(x) = \sigma(x)\beta(x) \qquad \textbf{(3-44)}$$

Avec :

$\gamma(x)$: polynôme de localisation des erreurs et des effacements

$\sigma(x)$: polynôme de localisation des erreurs, inconnue à ce stade

$\beta(x)$: polynôme de localisation des effacements, connue à ce stade

[20] Source : Error-Control Block Codes, L.H. Charles LEE, Artech House Publishers

On calcule le nouveau polynôme du syndrome avec les symboles effacés remplacés par des symboles nuls :

$$S(x) = S_{2t}x^{2t-1} + ... + S_2 x + S_1 \qquad \textbf{(3-45)}$$

En connaissant les $2t$ syndromes, le problème cette fois est de calculer le polynôme de localisation des erreurs qui satisfait la relation suivante :

$$\omega(x) = \left[S(x)\gamma(x)\right] \bmod \left(x^{2t}\right) \qquad \textbf{(3-46)}$$

Avec :

$\omega(x)$: polynôme d'amplitude

$\gamma(x)$: polynôme de localisation des erreurs, inconnu à ce stade

$S(x)$: syndrome modifié, connu à ce stade

Comme $\beta(x)$ est un facteur de $\gamma(x)$, on peut utiliser $\beta(x)$ pour initialiser l'algorithme de Berlekamp – Massey à la valeur de $\mu = f = \deg\left(\beta(x)\right)$ avec les symboles du syndrome pour le calcul du polynôme de localisation des erreurs et des effacements.

Avec la même technique qu'au chapitre 3.4.7, on peut évaluer l'amplitude des erreurs et des effacements selon la relation :

$$e_{j_i} = \frac{\omega\left[\left(\alpha^i\right)\right]}{\gamma'\left[\left(\alpha^i\right)\right]} \qquad \textbf{(3-47)}$$

3.5.3.1 Résumé des opérations pour le décodage selon Berlekamp – Massey[21]

1. Evaluation du polynôme des effacements $\beta(x)$
2. Modifier le polynôme reçu $r(x)$ en substituant les valeurs effacées par des zéros et calculer le syndrome
3. Remplacer le polynôme de localisation des erreurs $\sigma(x)$ par le polynôme de localisation des erreurs et des effacements dans le diagramme en flèches figure 3-11
4. Initialiser l'algorithme selon les paramètres suivants :
   $\mu = f$, $B(x) = x\beta(x)$, $\gamma^{(\mu)}(x) = \beta(x)$, $j = 0$ et $L_\mu = f$
5. Calcul du polynôme de localisation des erreurs et des effacements selon l'algorithme de Berlekamp – Massey en utilisant les coefficients du syndrome
6. Evaluation du polynôme d'amplitude $\omega(x)$
7. Calculer les racines du polynôme de localisation des erreurs et des effacements $\gamma(x)$
8. Déterminer les amplitudes des erreurs et des effacements avec le polynôme de localisation des erreurs et des effacements, le calcul donnera le polynôme d'erreur $e(x)$
9. Soustraction du polynôme $e(x)$ au polynôme reçu $r(x)$ pour la correction des erreurs et des effacements

[21] Source : Error-Control Block Codes, L.H. Charles LEE, Artech House Publishers

4 Implémentation hardware

4.1 Introduction

Dans les sous-chapitres qui suivent, on discutera et expliquera l'implémentation hardware choisie. On implémentera un code RS(15,9) comme vu dans tous les exemples du chapitre 3 . Ce code sera un simple correcteur d'erreurs, on ne traitera pas les effacements.

4.2 Flux de conception hardware

Les principales étapes pour la conception hardware d'un circuit logique sur FPGA ou CPLD sont :

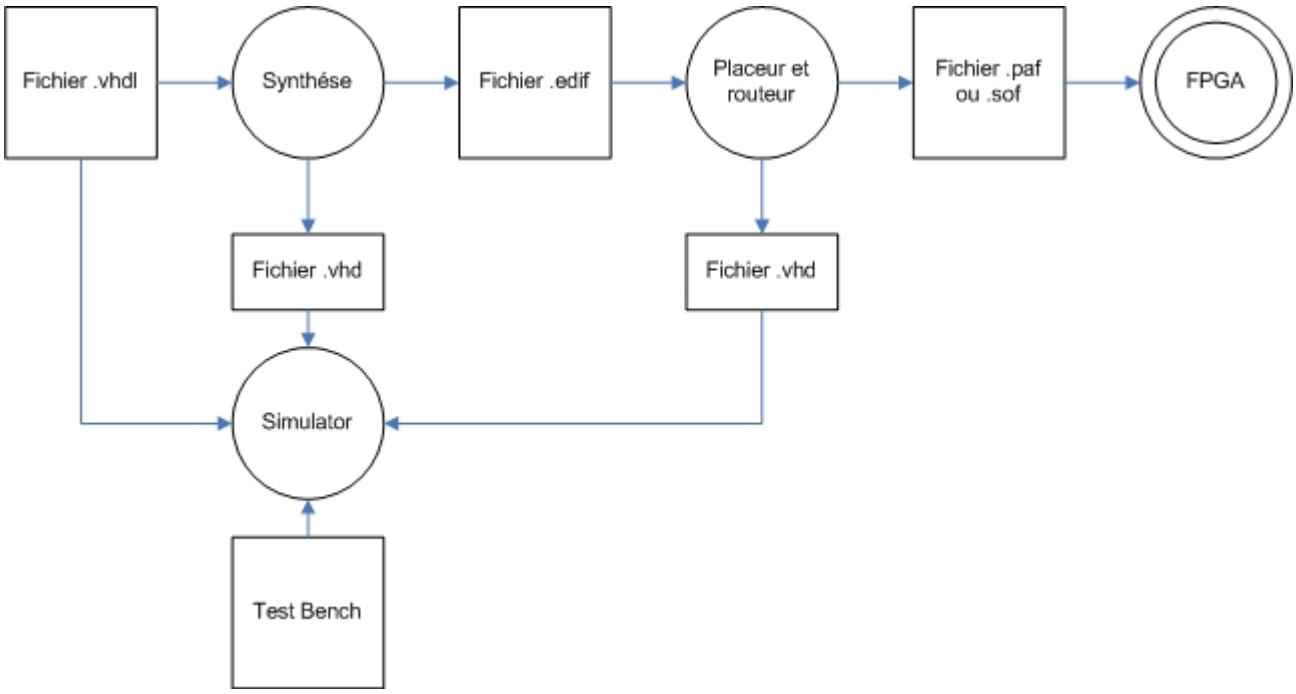

Figure 4-1: étape d'implémentation d'un circuit logique sur FPGA ou CPLD

1. Le fichier *.vhdl* contient la description du circuit traité
2. La simulation permet d'essayer le fichier décrit en *.vhdl*, afin de contrôler le résultat temporellement
3. La synthèse permet de faire la traduction du fichier *.vhdl* en fichier logic. Le fichier crée sera un fichier *.edif*
4. Le placement/routage permet de placer les interconnexions créées avec le fichier edif de synthèse pour une technologie donnée. Chaque fabricant fournit son propre logiciel. Le placement et routage fournira selon la technologie choisie un fichier *.paf* ou *sof*
5. Le programmeur permettra de programmer un FPGA ou un CPLD avec le fichier *.paf* ou *.sof*

4.3 Codeur

4.3.1 Eléments du codeur

Pour faciliter la tâche de codage, on divise le codeur en trois blocs, un bloc pour la sélection des entrées, un bloc qui calcule les symboles de contrôle, appelé *Parity_Block* et un bloc qui sert à choisir les bons signaux dans le bloc général, appelé *Logic_Control*.

L'élément top du codeur est :

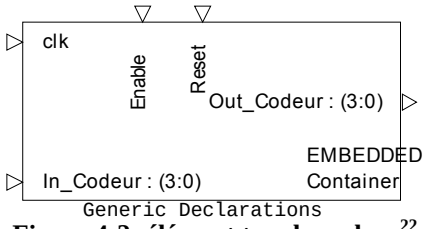

Figure 4-2: élément top du codeur[22]

Le codeur est composé à la base d'éléments simples comme expliqué au chapitre 3.3. Les éléments base utilisés pour le codage sont:

- Additionneurs
- Bascules D
- Multiplicateurs en $GF(2^4)$
- Multiplexeurs

4.3.1.1 Signaux de commande et d'entrée/sortie du codeur

Les signaux utilisés par le codeur se divisent en deux catégories différentes :

- les signaux de commandes
- les signaux d'entrée/sortie

Les signaux de commande représentent tous les signaux permettant au codeur de fonctionner correctement. Ces signaux sont :

- clk : signal d'horloge (*std_logic*)
- Enable : signal qui permet d'activer le codeur. Ce signal est activé quand les symboles à coder arrivent et reste actif jusqu'à ce que le dernier symbole codé soit sorti du codeur (*std_logic*)
- Reset : signal permettant la remise à zéro du codeur (*std_logic*)

[22] Voir CD, chemin : Reed-Solomon codes - FPGA\Exportations\Codeur\Codeur

Les signaux d'entrée/sortie constituent tous les signaux que le système doit traiter. Ces signaux sont :

- In_Codeur : signal d'entrée des symboles à coder. Cette entrée est sur 4 bits. (*std_logic_vector(3 downto 0)*)
- Out_Codeur : signal de sortie des symboles codés. Cette sortie est sur 4 bits (std_logic_vector(3 downto 0))

4.3.2 Bloc sélection des entrées

Le bloc de sélection des entrées sert à effectuer le décalage des informations situées dans les puissances élevées du polynôme et à placer les symboles de parité dans la place restante du polynôme. Ce phénomène est expliqué au chapitre 3.3. Pour ce faire, on utilise un *multiplexeur* que l'on commute sur l'entrée des symboles de parité dès que le dernier symbole d'information est entré dans le codeur.

Pendant les premiers « k » coups d'horloge, le *multiplexeur* sera sélectionné sur l'entrée des symboles d'information. Après avoir inséré le dernier symbole d'information dans le codeur (au « k-ième » coup d'horloge), le *multiplexeur* sélectionnera l'entrée des symboles de parité pour les « $n-k$ » coups d'horloge restants. Avant de pouvoir insérer le prochain bloc d'information, on devrait attendre que le dernier symbole de parité du bloc précédant soit sorti. On devrait donc attendre «$n+1$» coups d'horloge avant de pouvoir insérer le prochain bloc d'information.

On considère que dans le codeur entrent déjà des blocs d'information corrects de «k» symboles et on peut négliger le buffer d'entrée pour la sauvegarde des symboles. Les blocs de symboles arrivent avec la bonne cadence, «k» coup d'horloge pour les symboles d'information, puis une pause de «$n-k$» coups d'horloge est effectuée avant que le prochain symbole d'information entre dans le codeur.

Le multiplexeur est commandé par le bloc *Logic_Control*. Les symboles de parité sont calculés avec le *Parity_Block.*

4.3.3 Logic_Control

Le bloc *Logic_Control* est constitué des deux compteurs d'horloge qui s'alternent et qui commandent le signal *control*.

Le symbole utilisé pour ce bloc est :

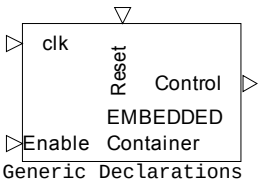

Figure 4-3: symbole du bloc Logic_Control

4.3.3.1 Signaux de commande et d'entrée/sortie du codeur

Les signaux de commande du bloc *Logic_Control* sont :

- clk : signal d'horloge (*std_logic*)
- Enable : signal qui permet d'activer le comptage. Ce signal est activé quand les symboles à coder arrivent et reste actif jusqu'à ce que le dernier symbole codé soit sorti du codeur (*std_logic*)
- Reset : signal permettant la remise à zéro du bloc *control_logic* (*std_logic*)

Le signal de sortie est :

- Control : signal de sortie est à « 0 » lors les 9 premiers coups d'horloge et mise à « 1 » pour les 6 coups d'horloge restants. Ce signal est commandé par deux compteurs d'horloge (*std_logic*)

4.3.3.2 Fonctionnement du bloc Logic_Control

Le premier compteur *count9* compte jusqu'à neuf en gardant le signal de *control* à « 0 », après qu'il ait fini de compter, il se réinitialise et met la sortie *logic* à « 1 ». Il passe la main au deuxième compteur *count6* qui, lui, compte jusqu'à six en gardant la valeur du signal *logic*. Une fois que *count6* termine de compter, il se réinitialise et met la sortie *logic* à « 0 », et il passe la main à *count9*. Cette opération est répétée tant que le signal *Enable* est actif.

Ce bloc est défini selon « VHDL » pur, aucun élément au schéma n'a été créé.

4.3.4 Parity_Block

Le *Parity_Block* est constitué de plusieurs registres, multiplicateurs et additionneurs qui calculent les symboles de parité à partir des informations et du polynôme générateur. Le symbole du bloc de parité est :

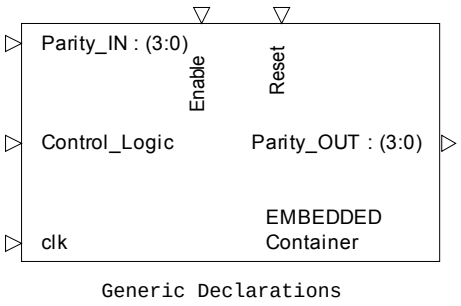

Figure 4-4: symbole du bloc de parité[23]

4.3.4.1 Signaux de commande et d'entrée/sortie du codeur

Les signaux de commande du bloc *Parity_Block* sont :

- clk : signal d'horloge (*std_logic*)
- Enable : signal qui permet d'activer le calcul des symboles de contrôle. Ce signal est activé quand les symboles à coder arrivent et reste active jusqu'à ce que le dernier symbole codé soit sorti (*std_logic*)
- Reset : signal permettant la remise à zéro du *Parity_Block* (*std_logic*)
- Control_Logic : signal qui permet la commande du *multiplexeur* interne (*std_logic*)

Les signaux entrée/sortie sont :

- Parity_In : signal d'entrée pour le codage des symboles qui doivent être codés. Cette entrée est sur 4 bits. (*std_logic_vector(3 downto 0)*)
- Parity_Out : signal de sortie des symboles codés. Cette sortie est sur 4 bits (std_logic_vector(3 downto 0))

4.3.4.2 Fonctionnement du bloc Parity_Block

Lors des premiers « k » coups d'horloge, les informations sont rentrées dans le *Parity_Block*. Après avoir entré le dernier symbole d'information (au « k-ième » coup d'horloge), on sortira les informations de contrôle sur la sortie du codeur lors des « n – k » coups d'horloge suivants.
A l'intérieur de ce bloc, on utilise un *multiplexeur* permettant de choisir l'entrée des symboles de parité pour les « k » premiers coups d'horloge, et qui met sur le bus un signal nul pour les « n – k » coups d'horloge restants. On utilise cette technique pour

[23] Voir CD, chemin : Reed-Solomon codes - FPGA\Exportations\Codeur\Parity_Bloc

sortir les symboles de contrôle sans avoir de résultats erronés et pour réinitialiser les registres.

4.3.4.3 Test bench du Parity_Block

Le test bench du *Parity_Block* est le même que celui du codeur complet, voir chapitre 4.3.5.

4.3.5 Test Bench du codeur complet

On effectue un test bench pour chaque bloc suivi d'un autre test général sur l'ensemble des blocs. Les coefficients de référence sont calculés à l'aide du logiciel « Matlab » selon la routine *rsenc(msg, n, k).*

Coefficients de référence pour le test bench général:

$$n = 15$$
$$k = 9$$
$$msg = \begin{bmatrix} 3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

La fonction de « Matlab » nous retourne le message codé suivant :

$$msg = \begin{bmatrix} 3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 15 & 15 & 14 & 15 & 14 & 5 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 2 & 1 & 3 & 12 & 15 & 11 \end{bmatrix}$$

4.3.6 Performance du codage

4.3.6.1 Ressources hardware

Les ressources hardware utilisées pour un code de RS(15,9) après placement et routage à l'aide du logiciel *Quartus II* sur une carte *ALTERA STRATIX EP1S20F484C5* sont :

Ressources	Usage
I/O pins	11 / 362 (3%)
Eléments logiques total	198 / 18,460 (1 %)
Eléments combinatoires	174
Eléments combinatoires avec registres	24
Registres utilisés	24

Tableau 4-1: ressources hardware utilisées

4.3.6.2 Débit binaire

Le débit binaire est calculé pour le code RS(15,9) avec FPGA *ALTERA STRATIX EP1S20F484C5* . Les temps de propagation sont calculés à l'aide de logiciel Quartus II.
L'horloge interne de la carte *EP1S20F484C5* est de $f = 332.7[MHz]$, ce qui donne une période d'environ $p \cong 3[ns]$. Le codage requiert 15 coups d'horloge, pendant cette période 9 symboles d'information suivis par 6 symboles de contrôle seront sortis du codeur. Comme on est dans le cas d'un $GF(2^4)$, on travaille sur 4 bits, donc on entrera au total pour le codage :

$$b = nbr\_symboles * nbr$$
$$b = 9 * 4 = 36[bits] \qquad \textbf{(4-1)}$$

Avec :

b	: nombre total de bits rentrés dans le codeur
$nbr\_symboles$	: nombre total de symboles rentrés dans le codeur
nbr	: nombre de bits utilisés

On a un temps total de latence de :

$$t = nbr\_hor * p$$
$$t = 15 * 3 * 10^{-9} = 45[ns] \qquad \textbf{(4-2)}$$

Avec :

t	: temps de codage
nbr_hor	: nombre de coups d'horloge nécessaires pour le codage
p	: période de l'horloge

Le débit binaire est défini comme :

$$d = \frac{b}{t}[bps] \qquad \textbf{(4-3)}$$

Avec :

d	: débit binaire en bits par seconde
b	: nombre de bits
t	: temps de codage

En se basant sur la relation (4-3) on peut déduire le débit du codage :

$$d = \frac{b}{t} = \frac{36}{45 * 10^{-9}} = 800000000 = 800[Mbps]$$

4.3.7 Codage : conclusion

Le codage fonctionne correctement selon la théorie vue au chapitre 3.3. Le codage choisi est un code de RS(15,9) permettant de comparer les résultats théoriques avec les résultats pratiques.

La simulation temporelle confirme que la démarche suivie pour constituer le codeur est correcte. Le codeur respecte les contraintes temporelles mentionnées dans les sous-chapitres précédants.

Les simulations des différents blocs du codeur sont effectuées avec une fréquence d'horloge de $f = 10[MHz]$. L'hardware n'a aucune influence sur la fréquence d'horloge choisie pour les simulations.

Le codage ne présente pas de vraies possibilités d'amélioration du débit binaire en changeant le type de codage ou les algorithmes, car il existe une seule façon de calculer les symboles de parité.

Pour améliorer sensiblement le débit binaire, on devrait changer le type de codage en prenant un codage plus performant, par exemple RS(255,223) ou des FPGA plus performantes. Plus la carte choisie est performante, plus les coûts seront élevés.

4.4 Décodeur

Le décodeur se compose de plusieurs blocs comme vu dans le chapitre 3.4.1. Chaque bloc sera expliqué en détail et les schémas utilisés dans les exemples théoriques seront repris pour les différents blocs. Suite à des problèmes d'implémentation le bloc de calcul des polynômes de localisation des erreurs et d'amplitude sera traité à l'aide de « Matlab ». Aucune implémentation « VHDL » ne serait faite pour ce bloc.

4.4.1 Syndrome_Bloc

Le bloc qui calcule les symboles du polynôme du syndrome est :

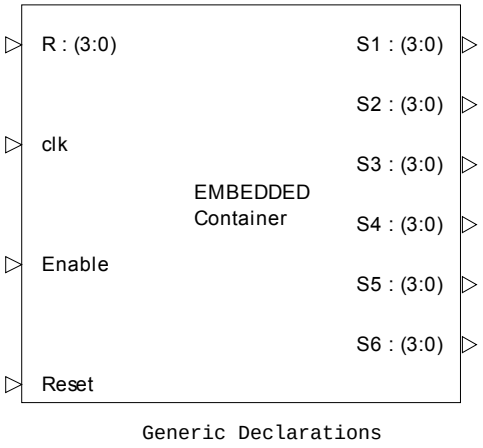

Figure 4-5: symbole du bloc calculant le syndrome[24]

Le bloc calculant le syndrome est composé de $2t$ blocs qui effectuent les mêmes opérations avec des valeurs différentes. Pour simplifier le schéma du *syndrome_bloc* on crée un autre symbole contenant un schéma simple qui sera utilisé autant de fois que l'on désire dans le bloc *syndrome_bloc*. Le bloc qui calcule un simple coefficient du syndrome est :

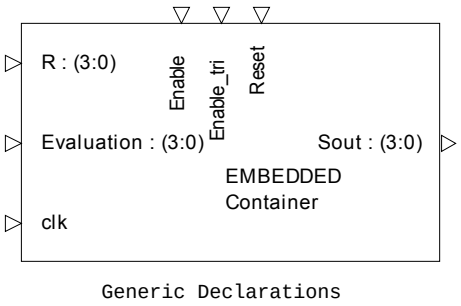

Figure 4-6: symbole du bloc calculant un coefficient du syndrome[25]

[24] Voir CD, chemin : Reed-Solomon codes - FPGA\Exportations\Decodeur\Syndrome_Bloc
[25] Voir CD, chemin : Reed-Solomon codes - FPGA\Exportations\Decodeur\Syndrome_simple

4.4.1.1 Signaux de commande et d'entrée/sortie syndrome_bloc

Les signaux de commande du *syndrome_bloc* et du *syndrome_simple* sont les mêmes :

- clk : signal d'horloge (*std_logic*)
- Enable : signal qui permet d'activer le calcul des symboles du syndrome. (*std_logic*)
- Reset : signal permettant la remise à zéro des blocs *syndrome_bloc* et *syndrome_simple* (*std_logic*)

Les signaux d'entrée/sortie du *syndrome_bloc* sont :

- R : entrée des symboles du mot-code reçu. Cette entrée est sur 4 bits (*std_logic_vector(3 downto 0)*)
- S1…S6 : sortie des symboles du syndrome calculé. Cette sortie est sur 4 bits (*std_logic_vector(3 downto 0)*)

Les signaux d'entrée/sortie pour *le syndrome_simple* sont :

- R : signal d'entrée des symboles du mot-code reçu. Cette entrée est sur 4 bits (*std_logic_vector(3 downto 0)*)
- Evaluation : signal qui permet de choisir la valeur pour laquelle le mot-code reçu sera évalué. Cette entrée est sur 4 bits (*std_logic_vector(3 downto 0)*)
- Enable_tri : signal commandant une porte tristate à l'intérieur du bloc pour avoir la sortie haute-impédance ou pour avoir le résultat du syndrome calculé (*std_logic*)
- Sout : signal de sortie du symbole calculé. Cette sortie est sur 4 bits (*std_logic_vector(3 downto 0)*)

4.4.1.2 Fonctionnement du syndrome_bloc

Le fonctionnement du bloc *syndrome_simple* est le même que celui de la figure 3-7. Les calculs effectués par ce bloc sont les mêmes que ceux de l'exemple 3.4.3.3 . La seule différence entre le schéma de l'exemple et implémentation est l'ajout d'une porte tristate permettant d'avoir le symbole calculé désiré au bon moment et non pas les valeurs intermédiaires. La porte tristate est commandée par le signal *En_tri* qui est généré depuis le bloc *counter* qu'on traitera dans le sous-chapitre suivant.
Le temps de calcul d'un coefficient du syndrome est de $n+1$ coups d'horloge.

Le *syndrome_bloc* est constitué de $2t$ blocs du *syndrome_simple* en parallèle qui calculent les différentes valeurs du syndrome.

4.4.1.3 Test Bench

Le test bench effectué sur le *syndrome_bloc* est le même que celui du chapitre 3.4.3.3. La référence pour l'exemple du chapitre 3.4.3.3 et pour le test bench est calculée avec « Matlab » selon la fonction :

$$S_i = polyval\left(r(x), \alpha^i\right)$$

Avec :

- S_i : symbole du syndrome
- Polyval : fonction « Matlab » qui évalue le polynôme
- $r(x)$: polynôme reçu
- α^i : valeur à tester pour le calcul du syndrome, $\alpha \le i \le \alpha^{2t}$

4.4.2 Bloc counter

Le bloc *counter* est un bloc constitué en « VHDL » pur, le symbole utilisé pour ce bloc est :

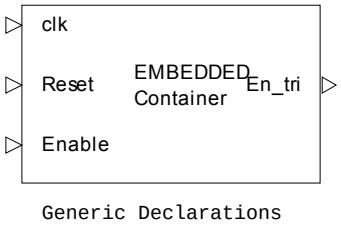

Figure 4-7: symbole du bloc counter

Ce bloc permet de commander les portes tristate à l'intérieur du *syndrome_bloc*.

4.4.2.1 Signaux de commande et d'entrée/sortie counter

Les signaux de commande du bloc *Logic_Control* sont :

- clk : signal d'horloge (*std_logic*)
- Enable : signal qui permet d'activer le comptage. Ce signal est activé dès que le premier symbole arrive dans le bloc *syndrome_bloc* et reste actif jusqu'à ce que le compteur ait fini de compter (*std_logic*)
- Reset : signal permettant la remise à zéro du *conteur* (*std_logic*)

Le signal de sortie est :

- En_tri : sortie de *counter*, il est mis à « 1 » à la fin du comptage. Cette sortie est sur 1 bit (*std_logic*)

4.4.2.2 Fonctionnement du bloc counter

Le bloc *counter* est un compteur d'horloge. Pendant les 15 premiers coups d'horloge la sortie *En_tri* sera gardée à « 0 » mais au but du 16ième coup d'horloge, la sortie sera mise à « 1 » de façon à rendre actives les sorties du *syndrome_bloc*.

4.4.3 Bloc d'Euclide

Le bloc d'Euclide permettant de calculer le polynôme de localisation des erreurs et le polynôme d'amplitude est beaucoup plus complexe que les autres blocs. Les ressources hardware et les temps de calcul de ce bloc seront supérieurs aux autres. Ce sous-chapitre présentera l'algorithme d'Euclide modifié qui permet le calcul du polynôme de localisation et le polynôme d'amplitude sans utiliser des divisions mais uniquement des multiplications et des décalages.

Une implémentation « Matlab[26] » sera fournie pour tester l'algorithme. Aucune implémentation « VHDL » synthétisable n'a été faite sur ce bloc car ça fonctionnalité a été découverte trop tardivement. Plusieurs essais sur différents autres blocs et algorithmes ont été effectués, mais aucun des résultats s'est avéré valable.

4.4.3.1 Algorithme modifié d'Euclide

L'algorithme d'Euclide modifié permet de calculer le polynôme de localisation des erreurs $\sigma(x)$ et d'amplitude $\omega(x)$ depuis l'équation clé $S(x)\sigma(x) = \omega(x) \bmod \left(x^{2t}\right)$ en connaissant le syndrome et le polynôme x^{2t}.

Les polynômes d'entrée de cet algorithme sont :

- Le syndrome $S(x)$, avec $2t$ symboles et de degré x^{2t-1}
- Le polynôme x^{2t}, avec des symboles nuls partout sauf pour le degré x^{2t}

Les polynômes de sortie de cet algorithme sont :

- Le polynôme de localisation des erreurs $\sigma(x)$
- Le polynôme d'amplitude $\omega(x)$

[26] Voir CD, chemin : Reed-Solomon codes - FPGA\Matlab\Euclide\Euclide Modifie

L'algorithme d'Euclide modifié est le suivant:

Initialisation

$R_0(x) = x^{2t}, Q_0(x) = S(x), L_0(x) = 0, U_0(x) = 1$

$\deg(R_0(x)) = 2t, \deg(Q_0(x)) = 2t-1$

$l_0 = \deg(R_0(x)) - \deg(Q_0(x))$

$i = 0, pas = 1$

Début

$pas = pas + 1$

$i = i + 1$

tant que $(pas \le 2t)$ ***boucle***

$pas = pas + 1$

$i = i + 1$

$a_{i-1} = premier\ coefficient\ du\ grade\ élevé\ non\ nul\ de\ R_{i-1}(x)$

$b_{i-1} = premier\ coefficient\ du\ grade\ élevé\ non\ nul\ de\ Q_{i-1}(x)$

Si $(\deg(R_i(x)) < t)$ ***alors***

$\sigma(x) = L_i(x)$

$\omega(x) = R_i(x)$

terminer l'algorithme

fin si

si $(l_{i-1} \ge 0)$ ***alors***

$R_i(x) = [b_{i-1}R_{i-1}(x)] - x^{|l_{i-1}|}[a_{i-1}Q_{i-1}(x)]$

$Q_i(x) = Q_{i-1}(x)$

$L_i(x) = [b_{i-1}L_{i-1}(x)] - x^{|l_{i-1}|}[a_{i-1}U_{i-1}(x)]$

$U_i(x) = U_{i-1}(x)$

sinon

$R_i(x) = [a_{i-1}Q_{i-1}(x)] - x^{|l_{i-1}|}[b_{i-1}R_{i-1}(x)]$

$Q_i(x) = R_{i-1}(x)$

$L_i(x) = [a_{i-1}U_{i-1}(x)] - x^{|l_{i-1}|}[b_{i-1}L_{i-1}(x)]$

$U_i(x) = L_{i-1}(x)$

fin si

$l_{i-1} = \deg(R_i(x)) - \deg(Q_i(x))$

fin tant que

$\sigma(x) = L_i(x)$

$\omega(x) = R_i(x)$

fin de l'algorithme

Code 4-1: algorithme modifié d'Euclide

4.4.4 Chien_bloc

Le bloc qui évalue le polynôme de localisation des erreurs et sa dérivée est :

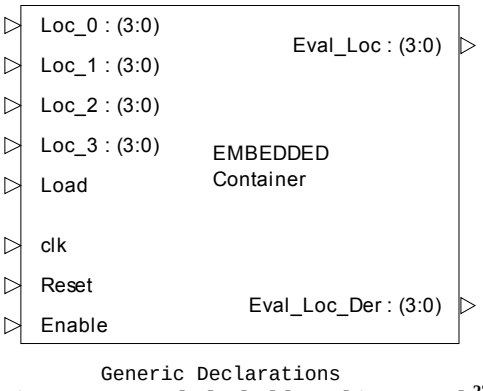

Figure 4-8: symbole du bloc Chien search[27]

Le schéma du *chien_bloc* est exactement le même que celui de la figure 3-12. Sur cette figure, on peut noter qu'on utilise la même séquence de calcul pour chaque coefficient des deux polynômes, mais avec des coefficients différents. On peut alors créer un bloc qui effectue ces opérations et les répètent autant de fois que l'on en a besoin dans le *chien_bloc* avec les différentes valeurs d'évaluation. Le symbole du bloc utilisé à l'intérieur du *chien_bloc* est :

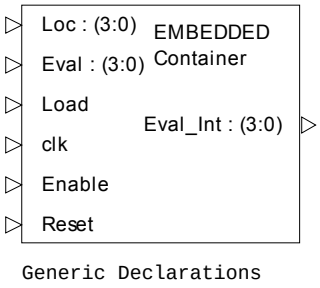

Figure 4-9: symbole du bloc *chien_int* utilisé à l'intérieur du *chien_bloc*[28]

[27] Voir CD, chemin : Reed-Solomon codes - FPGA\Exportations\Decodeur\Chien_Bloc
[28] Voir CD, chemin : Reed-Solomon codes - FPGA\Exportations\Decodeur\Chien_Bloc_Int

4.4.4.1 Signaux de commande et d'entrée/sortie du chien_bloc

Les signaux de commande du *chien_bloc* et *du chien_bloc_int* sont :

- clk : signal d'horloge (*std_logic*)
- Enable : signal qui permet d'activer l'évaluation. Ce signal est mis à « 1 » dès que les coefficients du polynôme de localisation arrivent dans le bloc *chien_bloc* et reste actif jusqu'à l'évaluation du dernier symbole (*std_logic*)
- Reset : signal permettant la remise à zéro du *chien_bloc* (*std_logic*)
- Load : signal commandant des multiplexeurs à l'intérieur de ce bloc (*std_logic*)

Du au fait que les entrée/sortie du *chien_bloc* et du *chien_bloc_int* sont légèrement différentes, on traitera les deux blocs séparément.

4.4.4.1.1 Signaux d'entrée/sortie du chien_bloc

Les signaux d'entrée/sortie du *chien_bloc* sont :

- Loc_0...Loc_3 : entrée des coefficients du polynôme de localisation des erreurs à évaluer. Cette entrée est sur 4 bits (*std_logic_vector(3 dowto 0)*)
- Eval_Loc : sortie des coefficients du polynôme de localisation des erreurs évalué pour différents α. Cette sortie est sur 4 bits (*std_logic_vector(3 dowto 0)*)
- Eval_Loc_Der : sortie des coefficients du polynôme de localisation des erreurs dérivé et évalué pour différents α. Cette sortie est sur 4 bits (*std_logic_vector(3 dowto 0)*)

4.4.4.1.2 Signaux d'entrée/sortie du chien_bloc_int

Les signaux d'entrée/sortie du *chien_bloc_int* sont

- Loc : entrée du coefficient du polynôme de localisation des erreurs. Cette entrée est sur 4 bits (*std_logic_vector(3 dowto 0)*)
- Eval : entrée de la valeur pour laquelle le coefficient du polynôme de localisation des erreurs doit être évalué. Cette entrée est sur 4 bits (*std_logic_vector(3 dowto 0)*)
- Eval_Int : sortie des coefficients évalués selon la valeur de *Eval*. Cette sortie est sur 4 bits (*std_logic_vector(3 dowto 0)*)

4.4.4.2 Fonctionnement du bloc chien_bloc

Le fonctionnement du bloc *chien_bloc* est le même que celui de la figure 3-12. Les calculs effectués par ce bloc sont les mêmes que ceux de l'exemple 3.4.6.2

Au début, le signal *Load* sera mis à « 0 » et cela permettra de charger les coefficients du polynôme de localisation des erreurs évalué pour la première valeur dans les registres. Après ce premier coup d'horloge, le signal *Load* sera mis à « 1 » et le restera pour les prochains n coups d'horloge de façon à pouvoir évaluer le polynôme pour tous les éléments du $GF\left(2^m\right)$. A chaque coup d'horloge, les différents symboles sortant des différents blocs de *chien_search_int* seront additionnés pour constituer les deux sorties.

Ce processus d'évaluation du polynôme de localisation des erreurs et de sa dérivée requiert un temps de calcul de $n+1$ coups d'horloge.

4.4.4.3 Test Bench

Le test bench effectué sur le *chien_bloc* est le même calcul que celui du chapitre 3.4.6.2. La référence pour l'exemple du chapitre 3.4.6.2 et pour le test bench est calculée par « Matlab » selon la fonction :

$$\sigma\left(\alpha^i\right) = polyval\left(\sigma\left(x\right), \alpha^i\right)$$
$$\sigma'\left(\alpha^i\right) = polyval\left(\sigma'\left(x\right), \alpha^i\right)$$

Avec :

- Polyval : fonction « Matlab » qui évalue un polynôme
- $\sigma\left(x\right)$: polynôme de localisation des erreurs
- $\sigma'\left(x\right)$: dérivée du polynôme de localisation des erreurs
- α^i : différentes valeurs de α selon le coup d'horloge, de α à 1

4.4.5 Eval_amplitude_bloc

Le bloc qui permet d'évaluer le polynôme d'amplitude est :

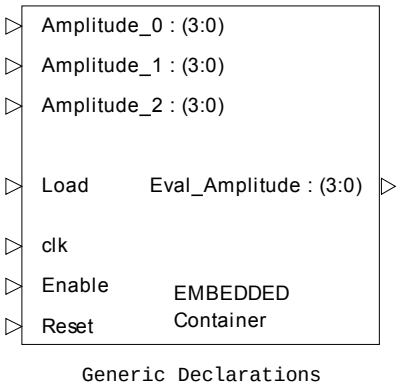

Figure 4-10: symbole du eval_amplitude_bloc[29]

Comme ce bloc effectue le même calcul que celui du *chien_bloc,* à l'intérieur de ce bloc on utilisera des blocs de *chien_bloc_int* avec des valeurs d'évaluation différentes.

4.4.5.1 Signaux de commande et d'entrée/sortie du eval_amplitude_bloc

Les signaux de commande du *eval_amplitude_bloc* sont :

- Clk : signal d'horloge (*std_logic*)
- Enable : signal qui permet d'activer l'évaluation. Ce signal est mis à « 1 » dès que les coefficients du polynôme d'amplitude arrivent dans le bloc *eval_amplitude_bloc* et le reste jusqu'à l'évaluation du dernier symbole (*std_logic*)
- Reset : signal permettant la remise à zéro du *eval_amplitude_bloc* (*std_logic*)
- Load : signal commandant des multiplexeurs à l'intérieur de ce bloc (*std_logic*)

Les signaux d'entrée/sortie de ce bloc sont :

- Amplitude_0…Amplitude_2 :coefficients du polynôme d'amplitude. Cette entrée est sur 4 bits (*std_logic_vector(3 downto 0)*)
- Eval_Amplitude : sortie du polynôme d'amplitude évalué pour différentes valeurs de α. Cette sortie est sur 4 bits (*std_logic_vector(3 downto 0)*)

[29] Voir CD, chemin : Reed-Solomon codes - FPGA\Exportations\Decodeur\Eval_amplitude_bloc

4.4.5.2 Fonctionnement du eval_amplitude_bloc

Le fonctionnement du *eval_amplitude_bloc* est le même que celui de la figure 3-15. Les calculs effectués par ce bloc sont les mêmes que ceux de l'exemple du chapitre 3.4.7.1.1.1.

Au début le signal, *Load* sera mis à « 0 » et cela permettra de charger les coefficients du polynôme d'amplitude des erreurs évalué pour la première valeur dans les registres. Après ce premier coup d'horloge, le signal *Load* sera mis à « 1 » et le restera pour les prochains n coups d'horloge de façon à pouvoir évaluer le polynôme pour tous les éléments du $GF\left(2^m\right)$. A chaque coup d'horloge, les différents symboles sortant des différents blocs de *chien_search_int* seront additionnés pour constituer la sortie.

Ce processus d'évaluation du polynôme d'amplitude des erreurs requiert un temps de calcul de $n+1$ coups d'horloge.

4.4.5.3 Test Bench

Le test bench effectué sur *eval_amplitude_bloc* est le même calcule que celui du chapitre 3.4.7.1.1.1. La référence pour le test bench sera donc le tableau 3-5.

4.4.6 Forney _bloc

Le bloc qui permet de déterminer le polynôme d'erreurs $e(x)$ et de corriger les erreurs est :

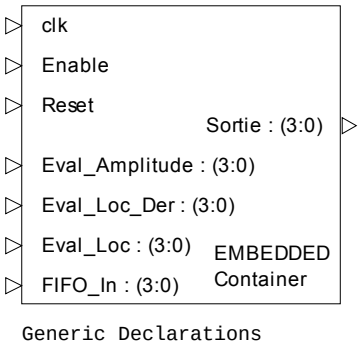

Figure 4-11 : symbole du forney_bloc[30]

A l'intérieur de ce bloc, on a différents éléments et un *ROM* qui permet de faire l'inversion afin d'effectuer la multiplication inverse. Ce bloc sera expliqué dans le sous-chapitre suivant.

[30] Voir CD, chemin : Reed-Solomon codes - FPGA\Exportations\Decodeur\Forney_bloc

4.4.6.1 Signaux de commande et d'entrée/sortie du forney_bloc

Les signaux de commande du *forney_bloc* sont :

- Clk : signal d'horloge (*std_logic*)
- Enable : signal qui permet d'activer la correction des erreurs. Ce signal est mis à « 1 » dès que le premier symbole arrive dans le *forney_bloc* et reste actif jusqu'à ce que le dernier symbole du polynôme de localisation évalué soit sorti (*std_logic*)
- Reset : signal permettant la remise à zéro du *forney_bloc* (*std_logic*)

Les signaux d'entrée/sortie du *forney_bloc* sont :

- Eval_Amplitude : entrée des symboles du polynôme d'amplitude évalué pour différents α. Cette entrée est sur 4 bits (*std_logic_vector(3 downto 0)*)
- Eval_Loc_Der : entrée des symboles de la dérivée du polynôme de localisation des erreurs évalué pour différents α. Cette entrée est sur 4 bits (*std_logic_vector(3 downto 0)*)
- Eval_Loc : entrée des symboles du polynôme de localisation des erreurs évalué pour différents α. Cette entrée est sur 4 bits (*std_logic_vector(3 downto 0)*)
- FIFO : entrée des symboles du polynôme reçu retardés par le temps de traitement des tous les autres blocs. Cette entrée est sur 4 bits (*std_logic_vector(3 downto 0)*)
- Sortie : sortie des symboles corrigés. Cette sortie est sur 4 bits (*std_logic_vector(3 downto 0)*)

4.4.6.2 Fonctionnement du forney_bloc

Le fonctionnement du *forney_bloc* est le même que celui à la figure 3-17. Les calculs effectués par ce bloc sont les mêmes que ceux de l'exemple du chapitre 3.4.7.2.

A chaque coup d'horloge, trois nouveaux symboles sont introduits dans le circuit, se sont ; $\sigma(\alpha^i)$, $\sigma'(\alpha^i)$ et $\omega(x)$. Sur le premier, on effectuera une opération logique « NOR » pour détecter quels symboles devraient être corrigés. Avec $\sigma^i(\alpha^i)$, on effectuera une inversion selon un certain *ROM* suivi d'une multiplication pour le symbole de $\omega(\alpha^i)$. Par le biais d'une porte logique « AND », les symboles de correction pourront corriger si nécessaire les symboles du polynôme reçu. La correction sera effectuée selon un porte logique « XOR ».

4.4.6.3 Test Bench

Le test bench effectué sur *forney_bloc* est le même calcule que celui du chapitre 3.4.7.2. La référence pour le test bench sera donc le tableau 3-7.

4.4.7 Inverse_ROM

Le symbole du bloc *Inverse_ROM* qui permet de faire l'inversion d'un symbole dans $GF\left(2^4\right)$ est :

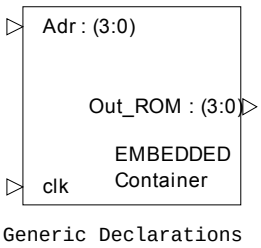

Figure 4-12: symbole de inverse_ROM

Ce symbole est utilisé dans l'algorithme de Forney.

4.4.7.1 Signaux de commande et d'entrée/sortie de inverse_ROM

Le signal de commande de ce bloc est :

- clk : signal d'horloge (*std_logic*)

Les signaux d'entrée/sortie du bloc *inverse_ROM* sont :

- Adr : signal qui permet d'adresser dans la *ROM* le bon symbole de sortie. Cette entrée est sur 4 bits (*std_logic_vector(3 downto 0)*)
- Out_ROM : sortie du symbole inverse adressé par la valeur du symbole d'entrée *Adr*. Cette sortie est sur 4 bits (*std_logic_vector(3 downto 0)*)

4.4.7.2 Fonctionnement du forney_bloc

Le fonctionnement est le même que celui de l'exemple du chapitre 3.4.7.1.2.1.

Dans ce bloc on a créé un tableau de constantes contenant toutes les valeurs inverses du $GF\left(2^4\right)$. Avec un *process* à chaque coup d'horloge, on lit le symbole présent sur l'entrée *Adr*. Ce symbole sera converti en entier et servira à adresser la bonne case du tableau contenant l'élément inverse du symbole *Adr*. Au prochain coup d'horloge, on sortira la valeur inverse et on lira le nouveau symbole à l'entrée *Adr* pour l'inversion.

4.4.7.3 Test Bench

Le test bench effectué sur *inverse_ROM* est le même calcul que celui du chapitre 3.4.7.1.2.1. La référence pour le test bench sera donc le tableau 3-6.

4.4.8 FIFO

Le *FIFO* sert à retarder le polynôme reçu de façon à l'additionner à son polynôme d'erreurs. La profondeur du *FIFO* n'est pas déterminable car le temps de traitement du bloc d'Euclide est inconnu.

4.4.9 Décodage : conclusion

Dans le bloc de décodage, les blocs testés présentent plus ou moins le même temps de calcul. Le bloc permettant de résoudre l'équation clé est le bloc effectuant le plus grand nombre de calculs. La majeure partie des opérations de ce bloc requiert beaucoup plus de ressources hardware que n'importe quel autre bloc traité ici.

Dans le cadre de ce projet la résolution selon l'algorithme d'Euclide modifié est choisie car elle présente un bon rapport entre complexité et efficacité. L'algorithme d'Euclide est plus facilement implémentable que l'algorithme de Berlekamp – Massey car la conception des différents blocs est plus simple. L'implémentation de test de l'algorithme d'Euclide modifié confirme que l'algorithme est valable et qu'il est possible d'effectuer le décodage selon cet algorithme.

Le temps de calcul, le débit et les ressources utilisés par le décodeur ne peuvent pas être calculés à ce stade car le bloc principal de la chaîne de décodage n'est pas implémentée.

La simulation temporelle des différents blocs confirme que le bon fonctionnement des différents blocs de décodage constitués dans la théorie du sous-chapitre 3.4. Les blocs du décodeur respectent les contraintes temporelles mentionnées dans les sous-chapitres précédents.

Les simulations des différents blocs du décodeur sont effectuées avec une fréquence d'horloge de $f = 10[MHz]$. La fréquence d'horloge choisie pour les simulations est n'influence en rien l'hardware utilisé.

5 Applications

Les codes de Reed – Solomon sont aujourd'hui dans toute application multimédia et dans toute application nécessitant une correction d'erreurs. Selon l'application on aura différents types de codes utilisés.

Les principaux domaines d'utilisations des codes de Reed – Solomon sont[31] :

- Dans la sauvegarde de données (sauvegarde magnétique, optique, etc…)
- Dans la communication mobile et les réseaux sans fils (wireless, etc…)
- Dans les communications satellitaires
- Dans la télévision numérique et la radio diffusion numérique (DVB)
- Dans les modems ADSL et VDSL

[31] Source : Reed – Solomon Encoder and Decoder Application

6 Conclusion

Dans les codes de Reed – Solomon, le décodage représente la tâche la plus complexe, tant au niveau théorique qu'au niveau hardware. Les ressources hardware utilisées par le décodeur sont beaucoup plus importantes que lors du codage. La complexité du codeur/décodeur dépend du type de codage choisi et des algorithmes choisis pour le décodage.

Le codage présente des débits binaires d'environ $800Mbps$ en choisissant un codage peu performant. En choisissant un code plus performant, comme par exemple un RS(255,223) proposé par les normes de télémétrie spatiale, le débit peut être sensiblement augmenté.

La partie principale du décodage est le bloc qui calcule l'équation clé et qui permet de calculer les polynômes de localisation des erreurs et le polynôme d'amplitude. Ce bloc requiert des efforts majeurs lors de d'implémentation et est gourmand en ressources hardware. Plusieurs solutions sont possibles pour résoudre l'équation clé, les deux méthodes plus répandues sont celles basées sur l'algorithme de Berlekamp – Massey et celles se basant sur l'algorithme d'Euclide. Ce dernier a été choisi pour l'implémentation dans le cadre de ce travail car il représente le meilleur rapport entre complexité et efficacité. L'algorithme de Berlekamp – Massey est traité seulement de façon théorique. Faute de temps, le bloc résolvant l'équation clé est présenté sous forme algorithmique et uniquement une simulation « Matlab » est effectuée.

Sans avoir implémenté au niveau hardware le bloc calculant l'équation clé, et dont le temps de calcul est supérieur aux autres blocs de part sa complexité, on peut affirmer que le décodage aurait un débit inférieur à celui du codage car il aura besoin de beaucoup plus de cycles d'horloge.

6.1 Améliorations

Les futures améliorations pour ce projet sont :

- Rendre fonctionnelle la partie de décodage en « VHDL ». Reste à coder le bloc d'Euclide
- Créer un bloc de contrôle commandant tous les blocs du décodage
- Changer le type de code pour le codage/décodage afin d'augmenter le débit binaire
- Rechercher des solutions pour l'amélioration du débit binaire, « pipeline » sur le décodeur possible.
- Recherche d'algorithmes plus performants pour le décodage

7 Acronymes et abréviations

Additionneur :	Elément numérique permettant de sommer deux symboles dans le « champ de Galois ».
Bascule D :	Elément mémoire qui permet de sauver des symboles à chaque coup d'horloge.
Code BCH :	Code de Bose – Chaudhuri – Hochquenghem, les codes de Reed – Solomon sont un cas particulier de code BCH
Code cyclique :	Chaque mot-code décalé engendre un autre mot-code.
CPLD	Complex Programmable Logic Device
Distance minimale	Paramètre permettant de déterminer le nombre d'erreurs qu'il est possible de détecter et corriger
FPGA :	Field Programmable Gate Array, chip logique programmable
Hardware :	Elément physique pouvant être implémenté.
Multiplicateur :	Elément numérique permettant de faire la multiplication de deux symboles dans un « champ de Galois ».
Multiplexeur :	Elément numérique permettant de choisir différents chemins selon un signal de commande précis.
ROM	Read Only Memory, mémoire accessible uniquement en lecture
RS(n,k) :	Codage selon Reed – Solomon avec « n » symboles sortants du codeur et « k » symboles d'information.
Software :	Equivalent anglais de *logiciel*
VHDL :	VHSIC Hardware Description Language.
VHSIC :	Very High Speed Integrated Circuit.

8 Annexes

Les annexes papiers :

Annexe A.1

Tableau résumant les différents travaux effectués

Annexe A.2

Tests bench des blocs de codage
Simulations des blocs de codage

Annexe A.3

Tests bench des blocs de décodage
Simulations des blocs de décodage

Annexe A.4

Simulation Matlab d'Euclide modifié

Le CD fournit en annexe contient les fichiers suivants :

Annexe C.1

Documentation utilisée pour ce travail ; .pdf, .ppt
Documents électroniques de mémoire

Annexe C.3

Codes VHDL du codeur
Simulations et tests bench du codeur

Annexe C.4

Codes VHDL du décodeur
Simulation et test bench du décodeur

Annexe C.5

Fichiers de simulation « Matlab » pour le calcul de polynôme de localisation des erreurs et d'amplitude basé sur l'algorithme d'Euclide

9 Tables et figures

9.1 Tables

Pages

9.2 Figures

Pages

10 Bibliographie

Chapitre 2

Livres

Charles Lee , L.H , Error-Control Block Codes, Artech House Publishers, 2000
William Stallings, Cryptography and Network Security, third edition, Prentice Hall, 2003
Shu Lin, Daniel J. Costello Jr , Error Control Coding, second edition, Prentice Hall, 2004
Cory S. Mondlin, Error control coding in DSL system, CRC Press LCC, 2004
Hervé Dedieu, Cours de théorie de l'information, EIVD, 2005

Sites Internet

Champ de Galois :
http://www.stanford.edu/class/ee387/handouts/galois.pdf
Reed-Solomon error correction :
http://www.bbc.co.uk/rd/pubs/whp/whp-pdf-files/WHP031.pdf
Forney algorithm:
http://www.stanford.edu/class/ee387/handouts/lect24.pdf

Chapitre 3

Livres

Charles LEE , L.H , Error-Control Block Codes, Artech House Publishers, 2000
Shu Lin, Daniel J. Costello Jr , Error Control Coding, second edition, Prentice Hall, 2004
Stephen B. WICKER, Vijay K. BHARGAVA, Reed – Solomon Codes and their application, IEEE Press, 1994

Sites Internet

Reed-Solomon codes:
http://www.4i2i.com/reed_solomon_codes.htm
Error and erasure correction :
http://www.stanford.edu/class/ee387/handouts/lect26.pdf
High speed electrical interface :
http://bwrc.eecs.berkeley.edu/Classes/EE290C_S04/lectures/Lect26_XAUI_PCIxs.5.pdf
On the high-speed VLSI implementation of errors-and-erasures correcting reed-solomon decoders:
http://delivery.acm.org/10.1145/510000/505326/p89zhang.pdf?key1=505326&key2=0048134311&coll=GUIDE&dl=GUIDE&CFID=59827818&CFTOKEN=63576591
Solving error location polynomial :
http://www.stanford.edu/class/ee387/handouts/chien.pdf
BCH decoding :
http://www.stanford.edu/class/ee387/handouts/lect23.pdf
Reed-Solomon solution with SPARTAN-II :
http://direct.xilinx.com/bvdocs/whitepapers/wp110.pdf
Digital Finite-Field Multiplier for Reed-Solomon Channel codes in GF(2^8) with programmable basis polynomial :
http://www.ece.iit.edu/~niliev/gf_mult_2003_conf.pdf
Self-correcting codes conquer noise Part 2: Reed-Solomon codecs :
http://www.edn.com/contents/images/315013.pdf
Reed-Solomon codes :
http://www.elektrobit.co.uk/pdf/reedsolomon.pdf
Reed-Solomon codes :
http://cwww.ee.nctu.edu.tw/course/channel_coding/chap6.pdf
Generalized Reed-Solomon codes :
http://www.math.msu.edu/~jhall/classes/codenotes/GRS.pdf
Power consuption of Reed-Solomon decoder algorithm :
http://epubl.luth.se/1402-1617/2002/289/LTU-EX-02289-SE.pdf
Coding theory :
http://www-rohan.sdsu.edu/~mosulliv/Courses/Coding02/RScodes.pdf
ECC Technologies :
http://members.aol.com/mnecctek/faqs.html

Chapitre 4

Sites Internet

An area-efficient VLSI architecture of a Reed-Solomon decoder/encoder for digital VCRs :
http://ieeexplore.ieee.org/iel3/30/13954/00642367.pdf?arnumber=642367
A new scalable VLSI architecture for Reed-Solomon decoder :
http://ieeexplore.ieee.org/iel4/5666/15173/00694898.pdf?arnumber=694898
A VLSI design of a pipeline Reed-Solomon decoder :
http://ieeexplore.ieee.org/iel6/8361/26343/01168233.pdf?arnumber=1168233
High-speed architectures for Reed-Solomon decoders :
http://portal.acm.org/citation.cfm?id=505522
A High Speed Reed-Solomon decoder Chip using Inversionless Decomposed Architecture for Euclidean Algorithm :
http://www.imec.be/esscirc/ESSCIRC2002/PDFs/C29.01.pdf
High-speed low-complexity Reed-Solomon decoder for optical communication :
http://soc.inha.ac.kr/mypapers/ITCAS2005_RSdec.pdf
Pipelined recursive modified Euclidean algorithm block for low-complexity,high-speed Reed–Solomon decoder :
http://soc.inha.ac.kr/mypapers/Eletter03_LEE.pdf
High-Speed VLSI Architecture for Parallel Reed–Solomon decoder :
http://soc.inha.ac.kr/mypapers/itvlsi03_lee.pdf
Modified Euclidean algorithm block for high-speed Reed-Solomon decoder :
http://soc.inha.ac.kr/mypapers/Eletter01.pdf

Chapitre 5

Sites Internet

Reed-Solomon encoder/decoder algorithm application
http://www.seasolve.com/products/reed-solomon/productinfo/1002.html