CONSTRUCCIÓN DE UN DECODIFICADOR REED-SOLOMON EN VHDL

Efrén Clemente Cuervo.
CINVESTAV-IPN
Departamento de Ingeniería Eléctrica
Sección de Computación

INTRODUCCIÓN.

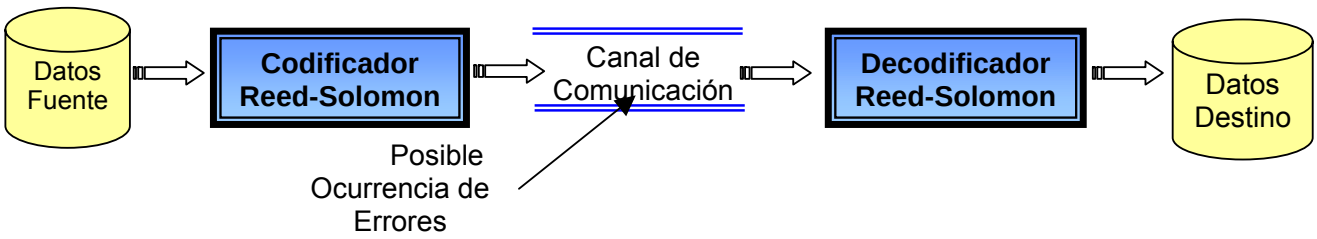
El código Reed-Solomon es un código corrector de errores basado en bloques con un amplio rango de aplicaciones en comunicaciones digitales y almacenamiento. El código fue inventado en 1960 por Irnving S. Reed y Gustave Solomon, miembros del MIT Lincoln Laboratory, en su articulo publicado "Polynomial Codes over Certain Finite Fields". Mas sin embargo la clave para hacer del código Reed-Solomon una aplicación tecnológica fue la implementación de un algoritmo eficiente de decodificación el cual fue realizado por Elwyn Berlekamp de la universidad de Berkeley

Actualmente los códigos Reed-Solomon se utilizan para corregir errores en varios sistemas incluyendo:

- ✓ Dispositivos de Almacenamiento (Cintas, Discos Compactos, DVD, códigos de barras)
- ✓ Comunicaciones inalámbricas o móviles (Telefonía celular, enlaces d microondas, etc.)
- ✓ Comunicaciones satelitales
- ✓ Televisión Digital/DVB
- ✓ Módem de alta velocidad como ADSL, x DSL, etc.

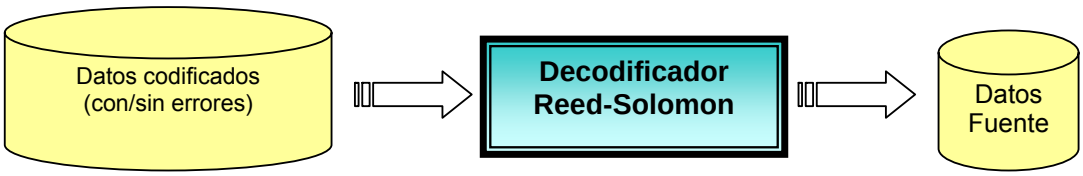
De ahí el interés tanto de conocer como de desarrollar un proyecto en donde aplique este tipo de algoritmo de “corrección de errores” en VHDL.

La idea típica de implementar un sistema de código Reed- Solomon seria la siguiente:



OBJETIVO.

1. Implementar un decodificador en VHDL el cual sea capaz de corregir los errores en un código Reed-Solomon y obtener a su salida un flujo de datos correctos de acuerdo a los Datos Fuente.



Un decodificador Reed-Solomon intenta identificar la posición y magnitud de hasta t errores y corregir los errores. La arquitectura general para decodificar códigos Reed-Solomon se muestra en el siguiente diagrama:

1. Cálculo del Síndrome:

Este es un cálculo similar al cálculo de paridad. Un código de palabra Reed-Solomon tiene 2t síndromes que dependen solamente de los errores (no de la palabra transmitida). Los síndromes pueden ser calculados al sustituir las 2t raíces del polinomio generador g(x) en r(x).

2. Encontrar el numero de errores y localizarlos:

Encontrar el lugar del símbolo erróneo implica resolver de forma simultánea ecuaciones con t incógnitas. Varios algoritmos rápidos existen para realizar lo anterior. Estos algoritmos toman ventaja de la estructura matricial especial de los códigos Reed-Solomon y reducen de gran forma el esfuerzo computacional requerido.

a) Encontrar un polinomio localizador de error:

Esto se puede lograr utilizando el algoritmo Berlekamp-Massey pero por sencillez en este proyecto se usará el algoritmo de Euclides; ya que este de acuerdo a la documentación revisada tiende a ser el más utilizado en la práctica debido a que es más fácil de implementar, sin embargo el algoritmo Berlekamp-Massey tiende a llevar a una implementación hardware y software más eficientes.

3. Encontrando los valores del símbolo erróneo:

Nuevamente, esto implica resolver ecuaciones con t incógnitas. Para poder encontrar los valores verdaderos, que deberán ser sustituidos en las posiciones correspondientes para así poder reproducir el mensaje correcto que se intento enviar. Esto se hace con el algoritmo de búsqueda de Chien.

DESARROLLO.

Código Reed-Solomon.

Sabemos que un código Reed-Solomon se especifica como RS(*n*, *k*) con símbolos de *s* bits. Lo anterior significa que el codificador toma *k* símbolos de los *s* bit y añade símbolos de paridad para hacer una palabra de código de *n* símbolos. Existen *n-k* símbolos de paridad de *s* bits cada uno. Un decodificador puede corregir hasta t símbolos que contienen errores en una palabra de código, donde *2t=n-k*. El siguiente diagrama muestra una típica palabra de código Reed-Solomon:

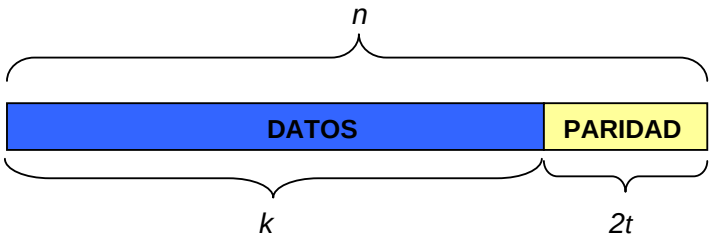


Figura 1: Diagrama de Una Palabra de Código Reed-Solomon

Errores de Símbolo.

Un error de símbolo ocurre cuando un bit en un símbolo es erróneo o cuando todos los bits en un símbolo se encuentran erróneos.

Ejemplo: RS(255,223) pude corregir 16 errores de símbolos. En el peor caso, errores de 16 bits pueden ocurrir, cada uno en un símbolo distinto (byte) de forma que el decodificador corrige errores de 16 bits. En el mejor caso, 16 errores de byte completos ocurren de tal forma que el decodificador corrige 16x8 errores de bit. Los códigos Reed-Solomon son particularmente útiles para corregir el llamado “burst error” (cuando una serie de bits en el código de palabra se reciben con error).

Decodificación

Los procedimientos algebraicos de decodificación de Reed-Solomon pueden corregir errores y datos perdidos. Un "borrado" ocurre cuando la posición de un símbolo errado es conocido. Un decodificador puede corregir hasta t errores o hasta 2t "borrados". Ahora bien a nuestro decodificador llegará una palabra r(x) la cual es la original (transmitida) c(x) más los errores e(x):

$$r(x)=c(x)+e(x)$$

Podemos apreciar muy fácilmente que la el codeword correcto estaría dado por:

$$c(x)=r(x)- e(x)$$

Y obviamente e(x) es totalmente desconocido, por lo que para poder encontrar el valor de e(x) tendremos que:

1. Calcular los Síndromes:

Los Síndromes se obtiene a partir de evaluar el codeword recibido r(x) donde $x=\alpha^j$, para *j=1…2t*. Así podemos observar que:

$$S_j = e(\alpha^j) = r(\alpha^j) - c(\alpha^j) = r(\alpha^j)$$

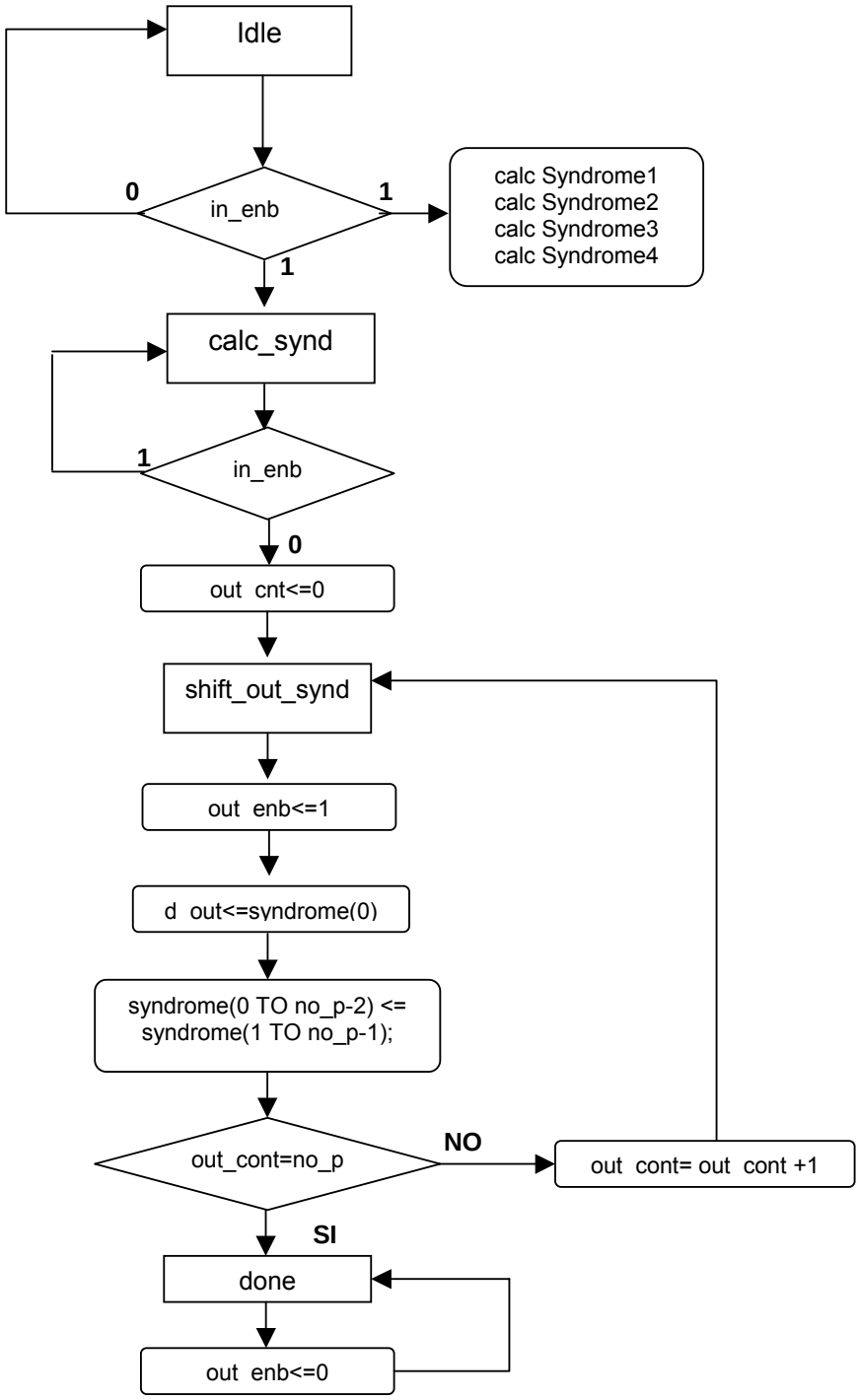
Así tenemos pues que el calculo de los síndromes es dado por:

$$S_i = \sum_{j=0}^{n} r_j \alpha^{i \cdot j} \qquad 0 \leq i < 2t$$

donde $r_j \in GF(2^m)$ son los símbolos recibidos

Implementación del cálculo de los Síndromes en VHDL:

Primeramente la implementación estaría dada por la siguiente maquina de estados:



De acuerdo a este diagrama podemos observar que el módulo del cálculo del síndrome actuará de la siguiente manera:

1. Se inicilizara en el estado ***idle.***

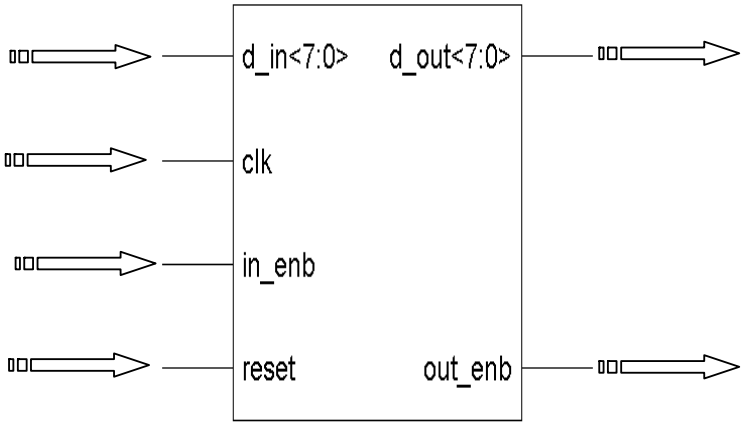
2. Al detectar que in_enb=1 entonces se entra al estado de recepcion y calculo del síndrome (*calc_synd*) para cada una de las raices, esto se realizará de la siguiente manera: se toma de la entrada in_d (la cual sera el termino m del codeword que se esta recibiendo) y ese término es evaluado multplicandolo por el valor de $\alpha^1, \alpha^2, \alpha^3, \alpha^4$ los cuales serían respectivamente 00000010, 00000100, 00001000, 00010000; y sumado a su vez al valor anterior, esto se realizara 32 veces ya que es el tamaño de nuestro codeword, en el siguiente código se puede observar lo antes mencionado:

```
Syndrome(0)= mul1(Syndrome(0)) XOR  ind_d
Syndrome(1)= mul2(Syndrome(1)) XOR  ind_d
Syndrome(2)= mul3(Syndrome(2)) XOR  ind_d
Syndrome(3)= mul4(Syndrome(3)) XOR  ind_d
```

3. Cuando la transmición del codeword finaliza entonces in_enb es puesto a 0 y pasamos al estado *shift_out_synd* el cual sera el encargado de enviar de manera serial los syndromes calculados para su posterior procesamiento.

4. Una vez enviados cada uno de los sindromes nuestro estado final es *done* y hasta ahí termina el proceso del calculo de los sindromes.

Dado como resultado el siguiente bloque, el cual correspondera al modulo del calculo de los Syndormes:



2. Encontrar el numero de errores y localizarlos:

a) Encontrar un polinomio localizador de error(Sigma) a partir del algoritmo de Euclides:

El primer paso para encontrar ***Sigma*** es aplicar el algoritmo de Euclides el cual para campos finitos lo podemos especificar como:

Teniendo un campo F y siendo a(x) y b(x) dos polinomios del campo y siendo b(x) diferente de 0 tenemos que entonces hay únicamente dos polinomios que satisfacen:

$$(1)\ a(x) = b(x)q(x) + r(x);$$
$$(2)\ \deg(r(x)) < \deg(b(x)).\ \square$$

De esto sabemos que:

$$\begin{aligned} a(x) &= 1 \cdot a(x) + 0 \cdot b(x); \\ b(x) &= 0 \cdot a(x) + 1 \cdot b(x). \end{aligned}$$

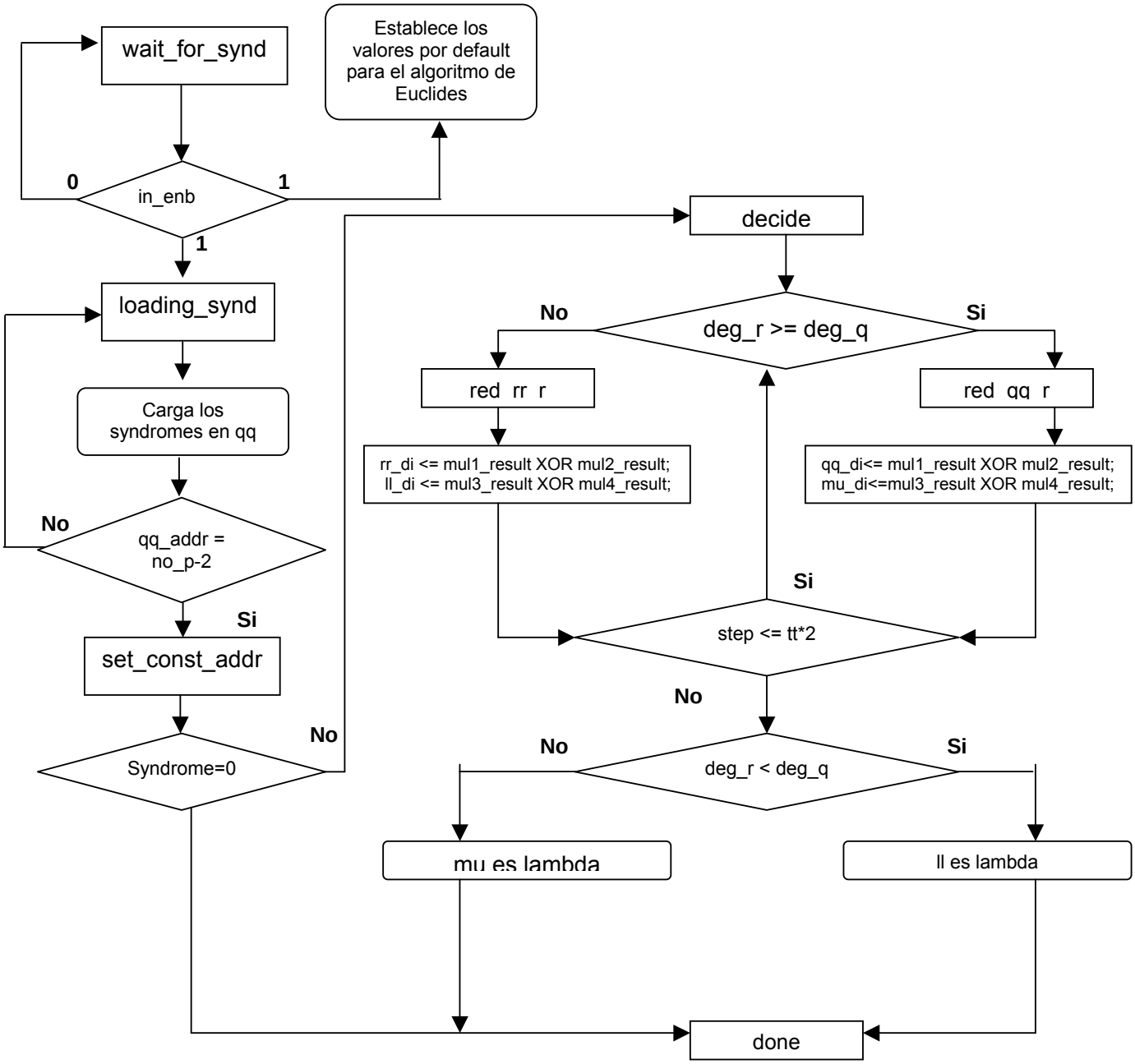
Ahora bien si asumimos que el grado(a(x))>= al grado(b(x)) con a(x)≠0 En el paso i se construye la ecualcion:

$$\mathbf{E_i} : r_i(x) = s_i(x)a(x) + t_i(x)b(x).$$

La ecuación E_i es construida a partir de E_{i-1} y E_{i-12} y la apropiada inicialización esta dada por (1) y (2)

$$\begin{aligned} r_{-1}(x) = a(x); \quad s_{-1}(x) = 1; \quad t_{-1}(x) = 0; \\ r_0(x) = b(x); \quad s_0(x) = 0; \quad t_0(x) = 1. \end{aligned}$$

Después dela inicialización los pasos son iterativos.


wait_for_synd
Establece los valores por default para el algoritmo de Euclides
0
in_enb
1
1
loading_synd
Carga los syndromes en qq
No
qq_addr = no_p-2
Si
set_const_addr
No
Syndrome=0
decide
No
deg_r >= deg_q
Si
red rr r
red qq r
rr_di <= mul1_result XOR mul2_result;
ll_di <= mul3_result XOR mul4_result;
qq_di<= mul1_result XOR mul2_result;
mu_di<=mul3_result XOR mul4_result;
Si
step <= tt*2
No
No
deg_r < deg_q
Si
mu es lambda
ll es lambda
done

Para este modulo hay que hacer notar que en primer lugar se crearon 4 memorias de tamaño (2t x m) estas memorias serán llamadas qq_ram, rr_ram, mu_ram y ll_ram. La memoria qq_ram contendrá en primera instancia los síndromes para a partir de ahí empezar el calculo. A continuación se presenta un diagrama muy general(omitiendo algunos pasos) de lo que es la maquina de estados para el algoritmo de euclides.

Como se puede apreciar en el diagrama el calculo de Euclides se comportará de la siguiente manera:

1. Cuando ***in_enb*** es puesto a 1 comienza a recibir los Síndromes calculados, y los escribe uno por uno en la memoria qq_ram, una vez terminada la carga de los síndromes se carga en el bus de multiplicación los valores iniciales. Verificamos que el síndrome no sea 0, en caso de que lo sea el codeword transmitido es correcto y se finaliza el proceso, en caso de que sea diferente de cero entonces procedemos al calculo de lambda y pasamos al estado ***decide***

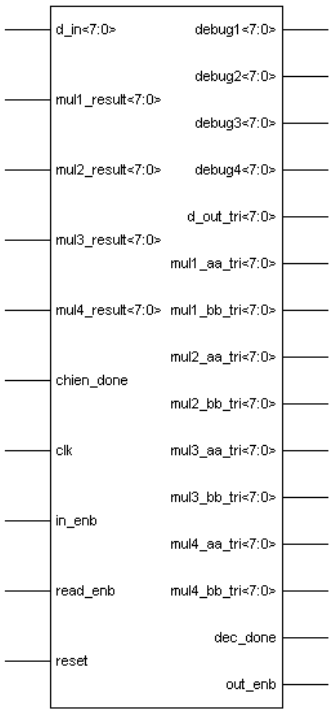
2. Si el grado del polinomio deg_r >= deg_q, trata de reducir rr

3. En caso contrario trata de reducir qq

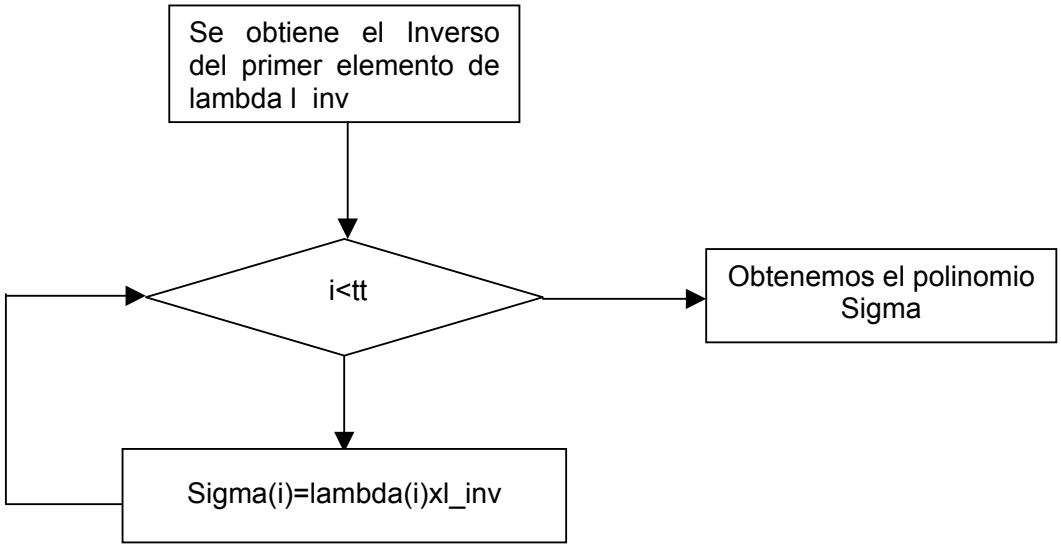
4. El proceso 2 o 3 se repetiran hasta que hayamos hecho 2t pasos

5. Por ultimo si deg_r < deg_q lambda es igual a ll_ram en caso contrario lambda debería ser igual a mu_ram.

En el siguiente bloque, se muestra el bloque para la implementacion del calculo de Euclides para obtener Labda:



El resultado de esta operación será un polinomio que llamaremos ***lambda*** a este polinomio habrá que aplicarle su inverso, para esto utilizaremos el módulo **gfnorm** el cual será el encargado de "normalizar", el proceso de normalizar se describe en el siguiente diagrama, el cual generaliza este proceso:



Una vez realizado este procedimiento obtendríamos el polinomio llamado Sigma el cual no es otra cosa sino la normalización de lambda.

$$\sigma(z) = z^r + \sigma_1 z^{r-1} + \cdots + \sigma_r = (z - \alpha^{e_1})(z - \alpha^{e_2}) \cdots (z - \alpha^{e_r})$$

Después de obtener Sigma obtenemos omega evaluando:

Sigma(x)Syndrome(x)=Omega(x)

Donde Omega será nuestro polinomio Evaluador del error.

Algoritmo de Chien Con Forney

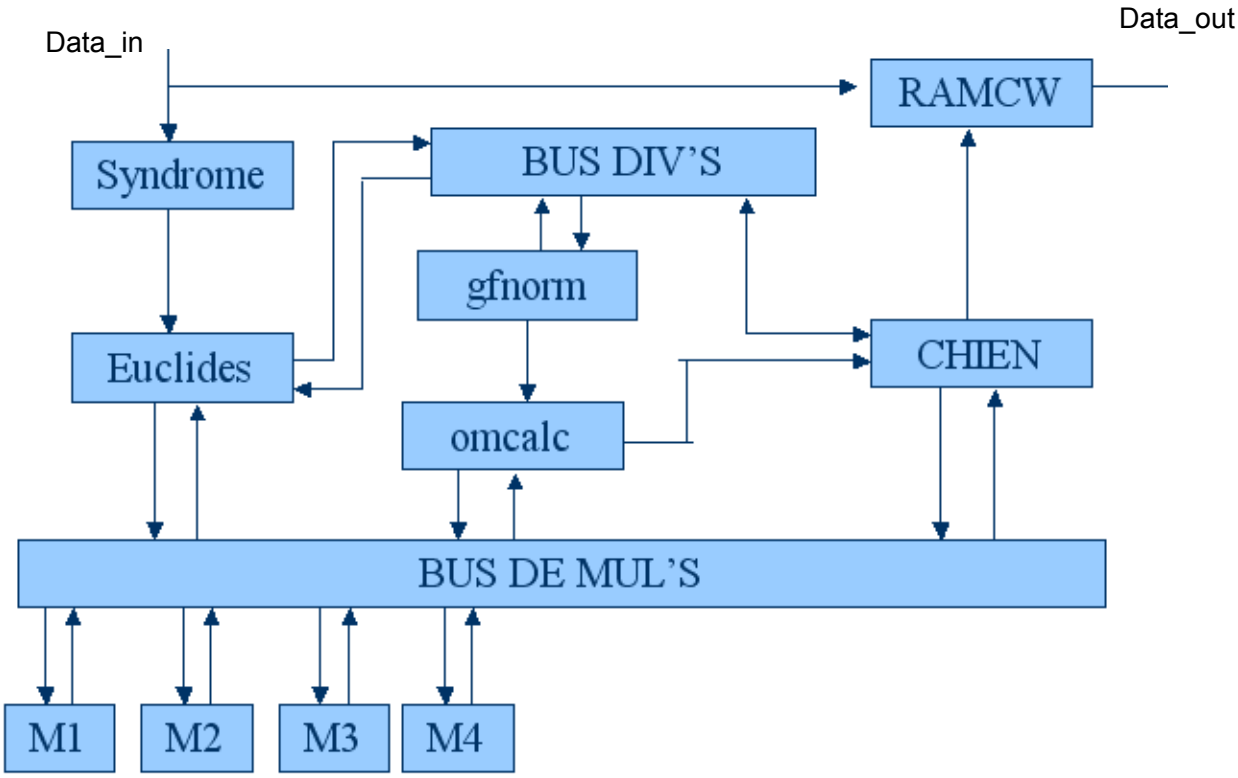
Un método que es muy comunmente usado es la sustitución de uno en uno, de todos los elementos dentro de σ(z). A esto se le llama la busqueda de chen.

Despues aplicamos forney con la siguiente formula:

$$e_i = \frac{\Omega(\alpha^{-i})}{\Lambda'(\alpha^{-i})} \cdot \alpha^{l}$$

Este sería el ultimo paso antes de sumar e(x) al codeword recibido, Como se puede observar el denominador es la derivada de lambda, Lo que significa que las potencias de 2 serán eliminadas.

Estructura General(VHDL)



APÉNDICE A. IMPLEMENTACION EN MAPLE

Comenzamos, usando la librería de álgebra lineal

```
> restart;
> with(linalg);
```

Primeo definimos las variables principales del decodificador, entre ellas están: P(El polinomio irreducible), G(El polinomio generatriz), t (el numero de errores que el código puede corregir) , fs (El tamaño del campo), el arreglo del alfabeto mapeado que estamos usando y por ultimo el procedimiento rscoeff; estos dos ultimos fueron tomados del algoritmo de Codificación Reed-Solomon implementado por Roberto Linares.

```
> p:=x^8+ x^4+x^3+ x^2+1:p:= unapply(p,x):eval(p);
```

$$x \to x^8 + x^4 + x^3 + x^2 + 1$$

```
> g:=(x-a)*(x-a^2)*(x-a^3)*(x-a^4):g:= unapply(g,x):eval(g);
```

$$x \to (x - a)\,(x - a^2)\,(x - a^3)\,(x - a^4)$$

```
> t:=2;fs:=256;
```

$$t := 2$$
$$fs := 256$$

para la interpretación de coeficientes

```
> rscoeff := proc(f, x, p, a)
>   local g, i, j, ng, cg, fs, field, ftable;
>   fs := 2^(degree(p));
>   field := linalg[vector](fs);
>   for i from 1 to fs-1 do
>     field[i] := Powmod(a, i, p, a) mod 2:
>   od:
>   field[fs] := 0;
>   ftable := table();
>   for i from 1 to fs-1 do
>     ftable[ field[i] ] := a^i:
>   od:
>   ftable[ field[fs] ] := 0;
>   g := expand(f) mod 2;
>   ng := 0;
>   for j from 0 to degree(g,x) do
>     cg := coeff(g, x, j):
>     cg := ftable[Rem(numer(cg),p,a) mod 2 ]/ftable[ Rem(denom(cg),p, a) mod 2 ];
>     if degree(cg,a) < 0 then
>       cg := cg * a^(fs-1);
>     fi:
>     if degree(cg,a) = (fs-1) then
>       cg := cg/a^(fs-1);
```

```
>      fi:
>      ng := ng + cg*x^j:
>    od:
>    g := sort(ng mod 2, x);
>    RETURN(g);
> end:

> alphabet := array(0..fs-1);
```

$$alphabet := array(0 .. 31, [])$$

```
> allist := [0, `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, `I`, `J`, `K`, `L`, `M`, `N`, `O`, `P`, `Q`, `R`, `S`, `T`, `U`, `V`,`W`,`X`,`Y`, `Z`,` `, `.`];
```

$$allist := [0, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, \ , .]$$

```
> atable := table():
> for i from 1 to nops(allist) do
>    alphabet[i-1] := allist[i]:
>    atable[ alphabet[i-1] ] := a^(i-1):
> od:
```

Supongamos que recibimos el siguiente mensaje(r1 es un código correcto y r2 es un código erróneo):

```
>r1 := a^19*x^23+ a^17*x^22+ a^18*x^21+ a^18*x^20+ a^29*x^19+ a^23*x^18+ a^24*x^17+ a^29*x^16+a^18*x^15+a^5*x^14+x^13+a^11*x^12+a^21*x^11+a^9*x^10+a^5*x^9+a^3*x^8+a^28*x^6+a^3*x^5+a^24*x^4+a^29*x^3+a^24*x^2+a^16*x+a^23;
> r2 :=  a^19*x^23+ a^17*x^22+ a^18*x^21+ a^18*x^20+ a^28*x^19+ a^23*x^18+ a^24*x^17+ a^29*x^16+a^18*x^15+a^5*x^14+x^13+a^11*x^12+a^21*x^11+a^2*x^10+a^13*x^9+a^2*x^8+ a^28*x^6+a^3*x^5+a^24*x^4+a^29*x^3+a^24*x^2+a^16*x+a^23;
```

$$r1 := a^{19}x^{23} + a^{17}x^{22} + a^{18}x^{21} + a^{18}x^{20} + a^{29}x^{19} + a^{23}x^{18} + a^{24}x^{17} + a^{29}x^{16} + a^{18}x^{15} + a^{5}x^{14} + x^{13} + a^{11}x^{12} + a^{21}x^{11} + a^{9}x^{10} + a^{5}x^{9} + a^{3}x^{8} + a^{28}x^{6} + a^{3}x^{5} + a^{24}x^{4} + a^{29}x^{3} + a^{24}x^{2} + a^{16}x + a^{23}$$

$$r2 := a^{19}x^{23} + a^{17}x^{22} + a^{18}x^{21} + a^{18}x^{20} + a^{28}x^{19} + a^{23}x^{18} + a^{24}x^{17} + a^{29}x^{16} + a^{18}x^{15} + a^{5}x^{14} + x^{13} + a^{11}x^{12} + a^{21}x^{11} + a^{2}x^{10} + a^{13}x^{9} + a^{2}x^{8} + a^{28}x^{6} + a^{3}x^{5} + a^{24}x^{4} + a^{29}x^{3} + a^{24}x^{2} + a^{16}x + a^{23}$$

```
> r:=r2;
```

$$r := a^{19}x^{23} + a^{17}x^{22} + a^{18}x^{21} + a^{18}x^{20} + a^{28}x^{19} + a^{23}x^{18} + a^{24}x^{17} + a^{29}x^{16} + a^{18}x^{15} + a^{5}x^{14} + a^{13} + x^{11} + a^{12}x^{12} + a^{21}x^{11} + a^{2}x^{10} + a^{13}x^{9} + a^{2}x^{8} + a^{28}x^{6} + a^{3}x^{5} + a^{24}x^{4} + a^{29}x^{3} + a^{24}x^{2} + a^{16}x + a^{23}$$

Con el siguiente comando convertimos el resultado previo en una función

```
> r := unapply(r,x);
```

$$r := x \to a^{19}\, x^{23} + a^{17}\, x^{22} + a^{18}\, x^{21} + a^{18}\, x^{20} + a^{28}\, x^{19} + a^{23}\, x^{18} + a^{24}\, x^{17} + a^{29}\, x^{16} + a^{18}\, x^{15} + a^{5}\, x^{14} + x^{13} + a^{11}\, x^{12} + a^{21}\, x^{11} + a^{2}\, x^{10} + a^{13}\, x^{9} + a^{2}\, x^{8} + a^{28}\, x^{6} + a^{3}\, x^{5} + a^{24}\, x^{4} + a^{29}\, x^{3} + a^{24}\, x^{2} + a^{16}\, x + a^{23}$$

El siguiente paso es determinar si r(x) es el mensaje original mandado.

Calculando el Síndrome -

Recordemos que el crear un codeword c (x) (el cual representa un mensaje trasmitido y codificado con Reed-Solomon) esta dado por el producto del mensaje m(x)y el generador g(x). Que es, c(x) = m(x)g(x) donde *m(x)* esta dado por el polinomio del mensaje y *g(x)* esta dado por el polinomio generatriz.

Cuando un codeword ha transmitido de un lugar a otro, al llegar al destino de entrada no es posible saber su el mensaje transmitido corresponde al original que fue enviado.

Los valores de evaluar 1..2 t son llamados síndromes. Estos proveen la base para verificar si un polinomio r(x) es un codeword. Si sustituimos en r(x) y todo nos da 0 entonces, entonces r(x) es un codeword valido y ahí terminamos; pero si sustituimos en r(x) y alguna evaluación es diferente de 0, entonces r(x) no es un codeword valido y es necesario encontrar los errores y corregirlos.

A continuación calculamos los síndromes para nuestro ejemplo donde t = 4 errores.

```
> Sa := [ seq( Rem(r(a^i), p(a), a) mod 2 , i = 1..2*t ) ];
```

$$Sa := [a^{3} + 1,\ a^{4} + a^{3} + 1,\ a^{3},\ a^{4} + a^{2} + 1,\ a^{4} + a^{3} + a + 1,\ a^{2} + a,\ a^{4} + a^{2} + a,\ a]$$

Si hubiéramos obtenido una lista de ceros con el comando anterior significaría que el codeword es verdadero, pero, como se puede apreciar, hemos obtenido otra cosa lo cual nos lleva a continuar buscando los errores y corregirlos.

Localizando y corrigiendo los errores.

Nuestros objetivos en este proceso es el de crear un polinomio al cual llamaremos R(z) (sin confundirlo con r(x) el cual será conocido como polinomio evaluador del error y un polinomio V(z) llamado error polinomio localizador de error.

Primero formamos un polinomio donde los coeficientes son los síndromes encontrados anteriormente

```
> S := sum('Sa[j+1] *z^j', 'j' = 0..2*t-1);
```

$$S := a^{3} + 1 + (a^{4} + a^{3} + 1)\, z + a^{3}\, z^{2} + (a^{4} + a^{2} + 1)\, z^{3} + (a^{4} + a^{3} + a + 1)\, z^{4} + (a^{2} + a)\, z^{5} + (a^{4} + a^{2} + a)\, z^{6} + a\, z^{7}$$

```
> S := unapply(S,z);
```

$$S := z \to a^{3} + 1 + (a^{4} + a^{3} + 1)\, z + a^{3}\, z^{2} + (a^{4} + a^{2} + 1)\, z^{3} + (a^{4} + a^{3} + a + 1)\, z^{4} + (a^{2} + a)\, z^{5} + (a^{4} + a^{2} + a)\, z^{6} + a\, z^{7}$$

El siguiente paso requiere ejecutar el algoritmo de Euclides en nuestro caso se copió un código ya realizado el cual nos permite aplicarlo a nuestros polinomios

```
>
> rseuclid := proc(t, f, g, z, p, a)
>   local q, r, rm1, rp1, um1, u, up1, vm1, v, vp1, i;
>   rm1 := sort(Expand(f) mod 2);
>   r := sort(Expand(g) mod 2);
>   um1 := 1;
>   u := 0;
>   vm1 := 0;
>   v := 1;
>   while degree(r,z) >= t do
>     print(`degree(r,z)=`,degree(r,z));
      rp1 := Rem(rm1, r, z, 'q') mod 2;
>     rp1 := rscoeff(rp1, z, p, a);
>     q := rscoeff(q, z, p, a);
>     vp1 := expand(vm1 - v*q) mod 2;
>     vm1 := v;
>     v := sort(vp1, z);
>     v := rscoeff(v, z, p, a);
>     up1 := expand(um1 - u*q) mod 2;
>     um1 := u;
>     u := sort(up1);
>     u := rscoeff(u, z, p, a);
>     rm1 := r;
>     r := sort(rp1, z);
>     print(`Q = `, q, `  R = `, r, `  V = `, v, `  U =          `, u);
>   od;
>   print();
>   RETURN(q, r, v, u):
> end:

> f := z^(2*t);
```

$$f := z^{8}$$

```
> res := rseuclid(t, f, S(z), z, p(a), a);
```

$$degree(r,z) = 7$$

$$Q = a^{30} z + a^{25}, \quad R = a^{30} z^{6} + a^{2} z^{5} + a^{26} z^{4} + a^{5} z^{3} + a^{9} z^{2} + a^{23} z + a^{30}, \quad V = a^{30} z + a^{25},$$

$$U = \quad , \qquad 1$$

$$degree(r,z) = 6$$

$$Q = a^{2} z + a^{16}, \quad R = a^{8} z^{5} + a^{18} z^{4} + a^{7} z^{3} + a^{3} z^{2} + z + a^{11}, \quad V = a z^{2} + a^{4} z + a^{6},$$

$$U = \quad , \qquad a^{2} z + a^{16}$$

$$degree(r,z) = 5$$

$$Q = a^{22} z + a^{10}, \quad R = a^{2} z^{4} + a^{11} z^{2} + a^{18} z + a^{14}, \quad V = a^{23} z^{3} + a^{2} z^{2} + a^{6} z + a^{9}, \quad U = \quad ,$$

$$a^{24} z^{2} + a^{10} z + a^{29}$$

$$degree(r,z) = 4$$

$$Q = a^{6} z + a^{16}, \quad R = a^{13} z^{3} + a z^{2} + a^{27} z + a^{19}, \quad V = a^{29} z^{4} + a^{28} z^{2} + a^{25} z + a^{14},$$

$$U = \quad , \qquad a^{30} z^{3} + a^{25} z^{2} + a^{20} z + a^{11}$$

$$res := a^{6} z + a^{16},\ a^{13} z^{3} + a z^{2} + a^{27} z + a^{19},\ a^{29} z^{4} + a^{28} z^{2} + a^{25} z + a^{14},\ a^{30} z^{3} + a^{25} z^{2} + a^{20} z + a^{11}$$

> degree(sort(Expand(g) mod 2),z);

$$0$$

Para hacer la corrección de los errores solo necesitamos R y V los cuales se encuentran en la posición 2 y 3 respectivamente en el arreglo *res*.

> R := res[2];

$$R := a^{13} z^{3} + a z^{2} + a^{27} z + a^{19}$$

> R := unapply(R,z);

$$R := z \rightarrow a^{13} z^{3} + a z^{2} + a^{27} z + a^{19}$$

> V := res[3];

$$V := a^{29} z^{4} + a^{28} z^{2} + a^{25} z + a^{14}$$

> V := unapply(V,z);

$$V := z \rightarrow a^{29} z^{4} + a^{28} z^{2} + a^{25} z + a^{14}$$

Pues bien, los símbolos erróneos nos los dan los exponentes de las raíces de la siguiente manera:

```
> for i from 1 to fs-1 do
>    if (Rem(V(a^i), p(a), a) mod 2) = 0 then
>      print([a^i],` es raiz de `,V(z),` la poscion del error esta en:`,degree(a^(fs-1)/a^i,a)),`]`;
>    fi:
> od:
```

$$[a^{12}],\ es\ raiz\ de\ ,\ a^{29} z^{4} + a^{28} z^{2} + a^{25} z + a^{14},\ la\ poscion\ del\ error\ esta\ en:,\ 19$$
$$[a^{21}],\ es\ raiz\ de\ ,\ a^{29} z^{4} + a^{28} z^{2} + a^{25} z + a^{14},\ la\ poscion\ del\ error\ esta\ en:,\ 10$$
$$[a^{22}],\ es\ raiz\ de\ ,\ a^{29} z^{4} + a^{28} z^{2} + a^{25} z + a^{14},\ la\ poscion\ del\ error\ esta\ en:,\ 9$$
$$[a^{23}],\ es\ raiz\ de\ ,\ a^{29} z^{4} + a^{28} z^{2} + a^{25} z + a^{14},\ la\ poscion\ del\ error\ esta\ en:,\ 8$$

Ahora calculamos las posiciones obteniendo la derivada de V(z) con respecto de z

> V(z);

$$a^{29} z^{4} + a^{28} z^{2} + a^{25} z + a^{14}$$

> Vp := diff(V(z), z) mod 2;

$$Vp := a^{25}$$

```
> Vp := unapply(Vp, z);
```

$$Vp := z \rightarrow a^{25}$$

Ahora calculamos los coeficientes para todos los errores

```
> e8 := R(a^23)/Vp(a^23):
> e9 := R(a^22)/Vp(a^22):
> e10 := R(a^21)/Vp(a^21):
> e19 := R(a^12)/Vp(a^12):
```

Por ultimo formamos el polinomio de corrección y usamos rscoeff convirtiendolo a una forma mas fácil con la que podamos trabajar

```
> e := e8*x^8 + e9*x^9 + e10*x^10 + e19*x^19:
> e := unapply(rscoeff(e, x, p(a), a), x);
```

$$e := x \rightarrow a^{11} x^{19} + a^{18} x^{10} + a^{24} x^{9} + a^{16} x^{8}$$

Obteniendo este valor sumamos el polinomio e(x) al polinomio recibido r(x), c(x) = r(x) + e(x) .

```
> c := unapply( rscoeff(r(x) + e(x), x, p(a), a), x);
```

$$c := x \rightarrow a^{19} x^{23} + a^{17} x^{22} + a^{18} x^{21} + a^{18} x^{20} + a^{29} x^{19} + a^{23} x^{18} + a^{24} x^{17} + a^{29} x^{16} + a^{18} x^{15} + a^{5} x^{14} + x^{13} + a^{11} x^{12} + a^{21} x^{11} + a^{9} x^{10} + a^{5} x^{9} + a^{3} x^{8} + a^{28} x^{6} + a^{3} x^{5} + a^{24} x^{4} + a^{29} x^{3} + a^{24} x^{2} + a^{16} x + a^{23}$$

Por ultimo podríamos volver a calcular los síndromes para saber si el mensaje es, ahora si, correcto

```
> seq(Rem(c(a^i), p(a), a) mod 2, i = 1..2*t);
```

$$0, 0, 0, 0, 0, 0, 0, 0$$

Obtenemos m(x) usando de nuevo el comando Quo el cual divide el codeword c(x) por el polinomio generatriz g(x).

```
> m := unapply(rscoeff(Quo(c(x),rscoeff(Expand(g(x)) mod 2, x, p(a),a), x) mod 2, x, p(a), a), x);
```

$$m := x \rightarrow a^{19} x^{15} + a^{5} x^{14} + a^{4} x^{13} + a^{15} x^{12} + a^{3} x^{11} + a^{14} x^{10} + a^{15} x^{9} + a^{13} x^{8} + a^{15} x^{7} + a^{12} x^{6} + a^{15} x^{5} + a^{19} x^{4} + a^{4} x^{3} + a^{5} x^{2} + a^{5} x + a^{18}$$

Finalmente mapeamos los valores a la tabla del alfabeto para obtener el mensaje:

```
> mlist := []:
> for i from 0 to degree(m(x), x) do
>    if coeff(m(x), x, i) <> 0 then mlist := [ op(mlist), alphabet[ degree(coeff(m(x), x, i)) ] ]:
>       else mlist := [ op(mlist), alphabet[ coeff(m(x), x, i) ] ]:
>    fi:
> od:
>
> mlist;
```

$$[R, E, E, D, S, O, L, O, M, O, N, C, O, D, E, S]$$

APÉNDICE B. IMPLEMENTACION EN C

Para la comprobación e implementación en VHDL se desarrolló un programa en ANSI C el cual es un codificador y decodificador parametrizable de códigos Reed-Solomon. Siendo este la base de la implementación en VHDL. Para su desarrollo se baso en el código fuente implementado por Phil Karn y Robert Morelos-Zaragoza.

El programa **rs_c** esta compuesto de 3 sub-módulos funcionales los cuales son:

- ***rs_galois.c***

  Este módulo es el encargado de generar el campo 2^m ,el polinomio generador, así como de proveer la funcionalidad de dicho campo para los demás módulos. Las Funciones principales del modulo se describen a continuación:

  void get_Pp(int mm);
  Obtiene el polinomio generador a partir del valor de m. Los polinomios que se crean son únicos para cada campo, y estos fueron tomados de la literatura encontrada en internet.

  void generate_gf(int mm);
  Genera los elementos del campo en base al polinomio generador creando una tabla que contendrá en base a los índices el valor "binario" que le corresponde a dicho elemento.

  int modnn(int x);
  Obtiene el modulo de x base nn.

  int gf_inv(int m1)
  Obtiene el inverso del elemento m1.

  int gf_mul_tab(int m1, int m2);
  Realiza una multiplicación usando el método de Karatsuba.

  void int_to_vector(int gf_el, int *vect);
  Transforma el valor de entero al vector correspondiente dentro del campo, todo esto dependiendo del campo en el que se encuentre.

  int vector_to_int(int *vect);
  Transforma el valor de un vector a entero.

  void print_vector(int *vect,int dir);
  Imprime el vector tal en la pantalla.

  void print_ind_el(int ind_el);
  Imprime el índice que le corresponde al elemento solicitado el cual es un entero.

  void print_poly_el(int poly_el, int mode);
  Imprime el polinomio de forma binaria.

- ***rs_utils.c***
  Este modulo es el que contiene algunas utilidades que era conveniente separar del modulo principal. Las cuales son:

  void init_rs(int nn, int kk, int **Gg_ind, int **Gg_poly);
  Es donde se inicializan las variables requeridas, así mismo esta función es la encargada de crear ,con las funciones del modulo rs_galois.c , el campo y el polinomio generador.

  int rs_encode_tab(int *data, int *bb, int nn, int kk, int *Gg_poly);
  Realiza la codificación ReedSolomon

  int rs_syndrome(int *data, int *s, int nn, int kk);
  Obtiene los síndromes del codeword recibido.

- ***rs_euclid.c***
  Este modulo es el que realiza los cálculos mas fuertes de la descodificación, básicamente con este módulo es con el que se localizan los errores y se encuentran los valores correctos. Las funciones principales de este modulo son:

  void print_p_vect(int *poly, int cnt);
  Imprime un vector dando como salida su valor entero de acuerdo al campo y su presentación de "forma binaria", es decir de forma polinomial.

  int euclid(int nn, int kk, int *synd, int *lambda, int verbose);
  Sin mas que decir realiza el algoritmo de euclides obteniendo lambda el cual será nuestro polinomio localizador de errores.

  int rs_errval(int nn, int kk, int *synd, int *lambda, int *errval, int verbose);
  Con esta función se calcula el vector de error por el algoritmo de búsqueda de Chien y el algoritmo de Forney.

A continuación se presenta el listado del código escrito en ANSI C.

LISTADO DEL CODIGO FUENTE DE RS_C.C

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#include "rs_utils.h"
#include "rs_galois.h"
#include "rs_euclid.h"

#define PRINT_MODE 1

int main( int argc, char *argv[] )

{
 int i,offset,err_pattern;

 //int optind;

 int e,dec_err_cnt,err_cnt = 0;
 int mm,nn,kk,no_p;
 int *cw,*recvd,*corrected,*synd;
 int *errval,*lambda;
 int *Gg_ind, *Gg_poly;
 int verbose;
 int euclid_flag = 1;

 e = 1;                 //Numero de errores
 offset = 0;            //direccion del error
 mm = 8;                //Tamño de los simbolos
 nn = 32;               //tamaño de codeword
 kk = 28;               //Simbolos de datos
  verbose=10;           //Nivel de debugeo



 printf("RS(%d,%d) Codificador/Decodificaor:\n",nn,kk);
 no_p = nn-kk;
 cw = (int*) malloc( nn * sizeof(int));
 recvd = (int*) malloc( nn * sizeof(int));
 synd = (int*) malloc( (no_p+1) * sizeof(int));
 errval = (int*) malloc( nn * sizeof(int));
 corrected = (int*) malloc( nn * sizeof(int));
 lambda = (int*) malloc( (no_p+1) * sizeof(int));

 for(i=0;i<nn;i++) {cw[i] = 0;errval[i] = 0;}
 for(i=0;i<kk;i++) cw[i] = i;

 for(i=0;i<no_p+1;i++) synd[i] = 0;

 generate_gf(mm);
 init_rs(nn,kk,&Gg_ind,&Gg_poly);

 if(verbose > 1) {
        printf("Polinomio generador:\n");
        for (i = 0; i <= nn-kk; i++) {
               printf("Indice  %4d, \t\t Forma Polinomial: ",Gg_ind[i]);
               print_poly_el(Gg_poly[i],PRINT_MODE);
        }
        printf("\n");
 }

 /* Datos para la prueba*/
 cw[0] = 0x0;
 rs_encode_tab(cw,&cw[kk],nn,kk,Gg_poly);
 for(i=0;i<nn;i++) recvd[i] = cw[i];

 /* Agrga errores*/
 err_pattern = 100;
 for(i=offset;i<e+offset;i++) {
        recvd[i] ^= err_pattern;
 }
```

```
if((verbose > 0) && (e > 0)) {
        printf("Agregando un error en la posicion: ");
        for(i=0;i<e;i++) printf("%d ",i+offset);
        printf("\n");
 }
 if(verbose > 4) {
        printf("Despues de codificar con errores nos queda:\n");
        for (i = 0; i < nn; i++) {
               print_poly_el(recvd[i],PRINT_MODE);
        }
        printf("\n");
 }

 if (rs_syndrome(recvd,synd,nn,kk) != -1) {
        printf("syndrome igual a zero, no errores :) !\n");
        exit(0);
 }
 if(verbose > 1) {
        printf("Syndrome:\n");
        for (i = 0; i <= nn-kk; i++) {
               printf("indice %4d, \t\t   Forma Polinomial ",synd[i]);
               print_poly_el(Alpha_to[synd[i]],PRINT_MODE);
        }
        printf("\n");
 }

 /* calcula lambda: */
 if(euclid(nn,kk,synd,lambda,verbose) < 1) {
        printf("Grado de lambda es 0, no errores!\n");
        exit(0);
 }
 err_cnt = rs_errval(nn,kk,synd,lambda,errval,verbose);

 if(!err_cnt) {
        printf(" :(  imposible corregir error!\n");
        exit(-1);
 }

 /* correccion: */
 for (i = 0; i < nn; i++) corrected[i] = recvd[i] ^ errval[i];

 dec_err_cnt = 0;
 for(i=0;i<nn;i++) if(corrected[i] != cw[i]) dec_err_cnt++;
 if(verbose > 7) {
        printf("Despues de decodifiacar:\n");
        for (i = 0; i < nn; i++) {
               print_poly_el(corrected[i],PRINT_MODE);
        }
        printf("\n");
 }

 printf("Simbolos despues de decodificar: %d\n",dec_err_cnt);

 return 0;

}/*main*/
```

LISTADO DEL CODIGO FUENTE DE RS_EUCLIDES.C

```
/**************************************************************************
*
*  funciones para el decodificador Reed-Solomon con el algoritmo de euclides
*
*
**************************************************************************/

#include <stdio.h>
#include <math.h>

#include "rs_euclid.h"
#include "rs_galois.h"
#include "rs_utils.h"

#define PRINT_MODE 2

/**************************************************************************/
/*
   Retorna el grado del polinomio localizador lambda
   el maximo grado no debe de ser no_p/2
*/

int euclid(int nn, int kk, int *synd, int *lambda, int verbose)
{
 int i,k,i_start,step,shift;
 int no_p,tt;
 int deg_r,deg_q,deg_l,m_cnt;

 int rr[RS_NN_MAX],qq[RS_NN_MAX],ll[RS_NN_MAX],mu[RS_NN_MAX];
 int rr_n[RS_NN_MAX],qq_n[RS_NN_MAX],ll_n[RS_NN_MAX],mu_n[RS_NN_MAX];

 /* numero de simbolos en la paridad*/
 no_p = nn-kk;


 for (i = 0; i <= no_p; i++) {
        rr_n[i] = 0;
        qq_n[i] = 0;
        ll_n[i] = 0;
        mu_n[i] = 0;
 }

 /* iniciamos euclides*/
 for (i = 0; i < no_p; i++) {
        qq[i] = Alpha_to[synd[i+1]];
 }
 qq[nn-kk] = 0;
 for (i = 0; i <= no_p; i++) {
        rr[i] = 0;
        ll[i] = 0;
        mu[i] = 0;
 }
 rr[0] = 1;
 mu[nn-kk-1] = 1;
 no_p = nn-kk;
 tt = no_p/2;
 deg_r = no_p;
 deg_q = no_p-1;

 if(verbose > 3) {
        printf("Inicializacion:\n");
        printf("rr     ");
        print_p_vect(rr,no_p);
        printf("qq     ");
        print_p_vect(qq,no_p);
        printf("ll     ");
        print_p_vect(ll,no_p);
        printf("mu     ");
        print_p_vect(mu,no_p);
```

```
        printf("deg_r %d deg_q %d\n\n",deg_r,deg_q);
        printf("\n");
}

step = 1;
while(step <= tt*2) {

        if(step < no_p) shift = 1;
        else shift = 0;

        if((deg_r < tt) || (deg_q < tt)) {
                if(shift) {
                        for (i = 0; i < no_p; i++) {
                                rr_n[i] = rr[i+1];
                                ll_n[i] = ll[i+1];
                        }
                }
                step++;
                continue;
        }
        if(deg_r < deg_q) { /* trata de reducir qq */
                if(verbose) printf("reduciendo qq en el paso %d !\n",step);
                if(rr[0] == 0) {
                        if(verbose) printf("algunos coefficientes de rr son 0 !\n");
                        for (i = 0; i < no_p; i++) {
                                rr_n[i] = rr[i+shift];
                                ll_n[i] = ll[i+shift];
                        }
                        if(shift) {
                                rr_n[no_p] = 0;
                                ll_n[no_p] = 0;
                        }
                        deg_r--;
                        for (i = 0; i <= no_p; i++) {
                                qq_n[i] = qq[i];
                                mu_n[i] = mu[i];
                        }

                } else {
                        for (i = 0; i < no_p; i++) {
                                qq_n[i]=gf_mul_tab(qq[i+shift],rr[0])^gf_mul_tab(rr[i+shift],qq[0]);
                                mu_n[i]=gf_mul_tab(mu[i+shift],rr[0])^gf_mul_tab(ll[i+shift],qq[0]);
                        }
                        if(shift) {
                                qq_n[no_p] = 0;
                                mu_n[no_p] = 0;
                        }
                        deg_q--;
                        for (i = 0; i <= no_p; i++) {
                                rr_n[i] = rr[i];
                                ll_n[i] = ll[i];
                        }
                }

        } else { /* deg_r >= deg_q, trata de reducir rr */

                if(qq[0] == 0) {
                        if(verbose) printf("algunos coefficientes de qq son 0 !\n");
                        for (i = 0; i < no_p; i++) {
                                qq_n[i] = qq[i+shift];
                                mu_n[i] = mu[i+shift];
                        }
                        if(shift) {
                                qq_n[no_p] = 0;
                                mu_n[no_p] = 0;
                        }
                        deg_q--;
                        for (i = 0; i <= no_p; i++) {
                                rr_n[i] = rr[i];
                                ll_n[i] = ll[i];
                        }
                } else {
```

```
                        for (i = 0; i < no_p; i++) {
                                rr_n[i]=gf_mul_tab(rr[i+shift],qq[0])^gf_mul_tab(qq[i+shift],rr[0]);
                                ll_n[i]=gf_mul_tab(ll[i+shift],qq[0])^gf_mul_tab(mu[i+shift],rr[0]);
                        }
                        if(shift) {
                                rr_n[no_p] = 0;
                                ll_n[no_p] = 0;
                        }
                        deg_r--;
                        for (i = 0; i <= no_p; i++) {
                                qq_n[i] = qq[i];
                                mu_n[i] = mu[i];
                        }
                }
        }

        /* Asigna los nuevos valores */
        for (i = 0; i <= no_p; i++) {
                rr[i] = rr_n[i];
                qq[i] = qq_n[i];
        }
        for (i = 0; i <= no_p; i++) {
                ll[i] = ll_n[i];
                mu[i] = mu_n[i];
        }

        if(verbose > 3) {
                printf("Paso %d calculo parcial:\n",step);
                printf("rr     ");
                print_p_vect(rr,no_p);
                printf("qq     ");
                print_p_vect(qq,no_p);
                printf("ll     ");
                print_p_vect(ll,no_p);
                printf("mu     ");
                print_p_vect(mu,no_p);
                printf("Grado de r %d Grado de q %d\n\n",deg_r,deg_q);
        }

        step++;
 }

 deg_l = tt;
 if(deg_r < deg_q) { /* ll is lambda */
        for (i = 0; i <= deg_l; i++) lambda[i] = ll[i];
 } else {           /* mu is lambda */
        for (i = 0; i <= deg_l; i++) lambda[i] = mu[i];
 }
 i_start = 0;
 while((i_start <= no_p) && (lambda[i_start] == 0)) i_start++;
 if(i_start == (no_p+1)) {
        return(0);
 }
 if(verbose && i_start) printf("Eliminadno  %d ceros de lambda\n",i_start);
 /* shift lambda to eliminate leading zeros */
 deg_l -= i_start;
 for (i = 0; i <= deg_l; i++) lambda[i] = lambda[i+i_start];
 for (; i <= tt; i++) lambda[i] = 0;
 return(deg_l);

}/*euklid*/

/*****************************************************************************/
/*
   Calcula el vector de error por el algoritmo de busqueda de Chien
   y el algoritmo de Forney
*/

int rs_errval(int nn, int kk, int *synd, int *lambda, int *errval, int verbose)
{
 int i,k,no_p,tt,root_cnt;
 int nn_short,factor;
 int sigma_sum,omega_sum,ll_der_sum,val,inv;
 int *sigma,*omega,*ll_der;
```

```
no_p = nn-kk;
tt = no_p/2;

/* Pido memoria*/
sigma = (int*) malloc( (tt+1) * sizeof(int));
omega = (int*) malloc( (tt+1) * sizeof(int));
ll_der = (int*) malloc( (tt+1) * sizeof(int));

/* Calculo sigma*/
inv = gf_inv(lambda[0]);
for (i = 0; i <= tt; i++) sigma[i] = gf_mul_tab(lambda[i],inv);

if(verbose) {
        printf("sigma (normalized lambda):\n");
        for (i = 0; i <= tt; i++) print_poly_el(sigma[i],PRINT_MODE);
        printf("\n");
}

/* calculo el polinomio de error(omega) a partir de los syndromes y de sigma*/
for (i = 0; i < tt+1; i++) omega[i] = 0;
for (i = 0; i < tt; i++) {
        sigma_sum = 0;
        for (k = 0; k <= i; k++)
                sigma_sum ^= gf_mul_tab(sigma[k],Alpha_to[synd[i+1-k]]);
        omega[i] = sigma_sum;
}

if(verbose) {
        printf("omega:\n");
        for (i = 0; i < tt+1; i++) print_poly_el(omega[i],PRINT_MODE);
        printf("\n");
}

/* calculo la derivada formal de lambda teniendo en cuenta que:
  (x**2)' = 2 * x, 2 mod 2 = 0
  => las potencias de 2 seran elimnadas
*/

for (i = 0; i <= tt; i++) ll_der[i] = 0;
for (i = 1; i <= tt; i+=2) {
        ll_der[i-1] = sigma[i];
}

if(verbose) {
        printf("Derivada de lamba:\n");
        for (i = 0; i < tt; i++) print_poly_el(ll_der[i],PRINT_MODE);
        printf("\n");
}


nn_short = gf_nn_max - nn;
if(nn_short) {
        for (k = 0; k <= tt-1; k++) {
                factor = Alpha_to[((k+1)*nn_short)%gf_nn_max];
                sigma[k] = gf_mul_tab(sigma[k],factor);
                ll_der[k] = gf_mul_tab(ll_der[k],factor);
                omega[k] = gf_mul_tab(omega[k],factor);
        }
        /* se incrementa el grado de sigma*/
        factor = Alpha_to[((tt+1)*nn_short)%gf_nn_max];
        sigma[tt] = gf_mul_tab(sigma[tt],factor);
}


/* Busqueda de chien, integrado el algoritmo de Forney para obtener el valor del error*/
root_cnt = 0;
for (i = 0; i < nn; i++) {
        sigma_sum = 0;
        omega_sum = 0;
        ll_der_sum = 0;
        val = 0;
        /* Se evalua sigma*/
        for (k = 0; k <= tt; k++) {
```

```
                sigma[k] = gf_mul_tab(sigma[k],Alpha_to[k+1]);
                sigma_sum ^= sigma[k];
        }
        /* Se evalua omega y lambda en el paso i*/
        for (k = 0; k <= tt-1; k++) {
                omega[k] = gf_mul_tab(omega[k],Alpha_to[k+1]);
                omega_sum ^= omega[k];
                ll_der[k] = gf_mul_tab(ll_der[k],Alpha_to[k+1]);
                ll_der_sum ^= ll_der[k];
        }


        if(!sigma_sum) {
                /* calculo el valor del error*/
                if(ll_der_sum != 0) {
                        inv = gf_inv(ll_der_sum);
                        val = gf_mul_tab(omega_sum,inv);
                } else {
                        val = omega_sum;
                }

                if(verbose) {
                        printf("Raiz encontrada en %d, valor del error: ",i);
                        print_poly_el(val,PRINT_MODE);
                }
                errval[i] = val;
                root_cnt++;
        }
 }
 return(root_cnt);

}/*rs_errval*/



/*************************************************************************/
/*
   Imprime el polinomio como vector para debugeo :)
*/
void print_p_vect(int *poly, int cnt)
{
 int i;

 for (i = cnt-1; i >= 0 ; i--) printf("%2d,",Index_of[poly[i]]);
 printf("     Forma Polinomial: ");
 for (i = cnt-1; i >= 0 ; i--) printf("%02d,",poly[i]);
 printf("\n");
 return;
}/*print_p_vect */
```

LISTADO DEL CODIGO FUENTE DE RS_UTILS.C

```
/*****************************************************************************
* project:
*
* description:
*  Reed-Solomon codificador y decodificador, basado el codigo de Phil Karn/Robert
*  Morelos-Zaragoza "new_rs_erasures.c".
*
*****************************************************************************/
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include "rs_galois.h"
#include "rs_utils.h"

/*****************************************************************************/

int rs_encode_tab(int *data, int *bb, int nn, int kk, int *Gg_poly)
{
 int i,j,no_p;
 int feedback;

 no_p = nn-kk;

 CLEAR(bb,no_p);

 for (i = 0; i < kk; i++) {
        feedback = data[i] ^ bb[0];
        for (j = 0; j < no_p-1; j++)
               bb[j] = bb[j+1] ^ gf_mul_tab(Gg_poly[no_p-1-j],feedback);
        bb[no_p-1] = gf_mul_tab(Gg_poly[0],feedback);
 }

 return 0;

}/*rs_encode_tab*/



/*****************************************************************************/
/*

   multplicacion y division por alfa^^i
   el syndrome es almacenada con el valor de su indice
*/
int rs_syndrome(int *data, int *s, int nn, int kk)
{
 int i,j,no_p;
 int sum,product;

 no_p = nn-kk;

 /* reset A0: */
 for (j = 1; j <= no_p; j++) s[j] = A0;

 for (i = 0; i < nn; i++) {
        for (j = 1; j <= no_p; j++) {
               if (s[j] != A0) product = modnn(j+s[j]);
               else product = A0;
               sum = Alpha_to[product] ^ data[i];
               s[j] = Index_of[sum];
        }
 }

 /* Checa los errores: */
 for (j = 1; j <= no_p; j++) {
        if(s[j] != A0) return(-1);
 }
 return(0);

}/*rs_syndrome*/
```

```
/*****************************************************************************/
void init_rs(int nn, int kk, int **Gg_ind, int **Gg_poly)
{
 int i, j;

 if(nn > gf_nn_max) {
        printf("valor nn > pow(2,gf_mm)-1 !\n");
        exit(-1);
 }

 if(kk > nn) {
        printf("valor de kk> nn !\n");
        exit(-1);
 }

 *Gg_poly = (int*) malloc( (nn - kk + 1) * sizeof(int));
 *Gg_ind = (int*) malloc( (nn - kk + 1) * sizeof(int));

 (*Gg_poly)[0] = Alpha_to[B0];
 (*Gg_poly)[1] = 1;
 for (i = 2; i <= nn - kk; i++) {
        (*Gg_poly)[i] = 1;
        for (j = i - 1; j > 0; j--)
               (*Gg_poly)[j] = (*Gg_poly)[j - 1] ^ gf_mul_tab((*Gg_poly)[j], Alpha_to[B0+i-1]);

        (*Gg_poly)[0] = gf_mul_tab((*Gg_poly)[0], Alpha_to[B0+i-1]);
 }
 /* Genero el indice  */
 for (i = 0; i <= nn - kk; i++)
        (*Gg_ind)[i] = Index_of[(*Gg_poly)[i]];

 return;

}/* init_rs */
```

LISTADO DEL CODIGO FUENTE DE RS_GALOIS.C

```
/*****************************************************************************
*
* description:
*  Aritmetica de Galois
*
* modified:
*
*****************************************************************************/
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include "rs_galois.h"

/*****************************************************************************/
/* m1 y el resultado son polinomios*/
int gf_inv(int m1)
{

 if(m1 == 1) return(1);
 if(m1 == 0) {
        printf("Error: division por cero !\n");
        exit(-1);
 }
 return(Alpha_to[gf_nn_max-Index_of[m1]]);

}/* gf_inv */

/*****************************************************************************/
/* m1, m2 and result in poly-form ! */
int gf_mul_tab(int m1, int m2)
{
 int m1_i,m2_i,prod_i,result;

 if((m1 == 0) || (m2 == 0)) return(0);

 m1_i = Index_of[m1];
 m2_i = Index_of[m2];
 prod_i = modnn(m1_i + m2_i);
 result = Alpha_to[prod_i];
 return(result);

}/* gf_mul_tab */

/*****************************************************************************/
/* De entero a vector*/
void int_to_vector( int int_el, int *vect)
{
 int i;

 for(i=0;i<gf_mm;i++) vect[i] = (int_el>>i)&1;
 return;

}/* int_to_vector */


/*****************************************************************************/
/* de vector a entero ! */
int vector_to_int(int *vect)
{
 int i;
 int result;

 result = 0;
 for(i=0;i<gf_mm;i++) if(vect[i]) result |= (1<<i);
 return(result);

}/* vector_to_int */
```

```
/*****************************************************************************/
void print_vector(int *vect,int dir)
{
 int i;
 if(dir) for(i=gf_mm-1;i>=0;i--) printf("%d",vect[i]);
 else for(i=0;i<gf_mm;i++) printf("%d",vect[i]);

 printf("\n");

 return;

}/* print_vector */

/*****************************************************************************/
/* Calcula  x % NN, donde n NN es 2**gf_mm - 1,
 *
 */
int modnn(int x)
{
 while (x >= gf_nn_max) {
        x -= gf_nn_max;
        x = (x >> gf_mm) + (x & gf_nn_max);
 }
 return x;
}/* modnn */

/*****************************************************************************/
void generate_gf(int mm)
{
 int i, mask;

 get_Pp(mm);
 gf_mm = mm;
 gf_nn_max = pow(2,mm)-1;

 A0 = gf_nn_max;
 Alpha_to = (int*) malloc( (gf_nn_max + 1) * sizeof(int));
 Index_of = (int*) malloc( (gf_nn_max + 1) * sizeof(int));

 for (i = 0; i <= gf_nn_max; i++ ) Alpha_to[i] = 0;
 for (i = 0; i <= gf_nn_max; i++ ) Index_of[i] = 0;

 mask = 1;
 Alpha_to[mm] = 0;
 for (i = 0; i < mm; i++) {
        Alpha_to[i] = mask;
        Index_of[Alpha_to[i]] = i;
        /* Si Pp[i] == 1 entonces esterm @^i en la representacionde polinomio*/
        if (Pp[i] != 0)
               Alpha_to[mm] ^= mask; /* operacion de XOR */
        mask <<= 1;
 }
 Index_of[Alpha_to[mm]] = mm;

 mask >>= 1;
 for (i = mm + 1; i < gf_nn_max; i++) {
        if (Alpha_to[i - 1] >= mask)
               Alpha_to[i] = Alpha_to[mm] ^ ((Alpha_to[i - 1] ^ mask) << 1);
        else
               Alpha_to[i] = Alpha_to[i - 1] << 1;
        Index_of[Alpha_to[i]] = i;
 }
 Index_of[0] = A0;
 Alpha_to[gf_nn_max] = 0;
 return;

}/*generate_gf*/

/*****************************************************************************
*
* Polinomios generatrices ya esteblecidos tomados de la literarura en internet
*
*
/*****************************************************************************/
```

```
void get_Pp(int mm)
{
 int i;

 Pp = (int*) malloc( (mm + 1) * sizeof(int));

 for (i = 0; i <= mm; i++) Pp[i] = 0;

 Pp[0] = 1;
 Pp[mm] = 1;

 switch(mm) {
        case 2:
               Pp[1] = 1;
               break;
        case 3:
               Pp[1] = 1;
               break;
        case 4:
               Pp[1] = 1;
               break;
        case 5:
               Pp[2] = 1;
               break;
        case 6:
               Pp[1] = 1;
               break;
        case 7:
               Pp[3] = 1;
               break;
        case 8:
               Pp[2] = 1;
               Pp[3] = 1;
               Pp[4] = 1;
               break;
 }/*switch*/


}/*get_Pp*/

/*****************************************************************************/
/* Dado un indice Imprime un el elemento del campo en su forma polinomial */
void print_ind_el(int ind_el)
{
 int k;

 for (k = gf_mm-1; k >= 0; k--) printf("%d",(Alpha_to[ind_el]>>k)&1);
 printf("\n");
 return;

}/*print_ind_el*/

/*****************************************************************************/
void print_poly_el(int poly_el, int mode)
{
 int k;

 switch(mode) {
        case 0:
               for (k = gf_mm-1; k >= 0; k--) printf("%d",(poly_el>>k)&1);
               printf("\n");
               break;
        case 1:
               printf("%02x ",poly_el);
               for (k = gf_mm-1; k >= 0; k--) printf("%d",(poly_el>>k)&1);
               printf("\n");
               break;
        case 2:
               printf("%3d, %02x ",Index_of[poly_el],poly_el);
               for (k = gf_mm-1; k >= 0; k--) printf("%d",(poly_el>>k)&1);
               printf("\n");
               break;

 }
```

```
  return;

}/*print_poly_el*/

/*****************************************************************************/
/* m1, m2 and result in poly-form ! */
int gf_mul(int a_const, int m2, int verbose)
{
  int ai,bi,si,sumi,i,k;
  int *a,*b,*c,*cnt;
  int *subs_v;
  int result;
     typedef struct prod_term
  {int a,b;} prod_term_t;
  prod_term_t logic_t[MM_MAX][MM_MAX*MM_MAX];
  int subs;


  a = (int*) malloc(gf_mm * sizeof(int));
  b = (int*) malloc(gf_mm * sizeof(int));
  c = (int*) malloc(gf_mm * sizeof(int));
  subs_v = (int*) malloc(gf_mm * sizeof(int));

  cnt = (int*) malloc(gf_mm * sizeof(int));
  for(i=0;i<gf_mm;i++) cnt[i] = 0;

  int_to_vector(a_const,a);
  int_to_vector(m2,b);

  /* generamos la tabla */
  for(bi=0;bi<gf_mm;bi++) {
         for(ai=0;ai<gf_mm;ai++) {
                sumi = ai+bi;
                if(sumi > gf_mm-1) {
                       /* Reducel el grado del polinomio por substitucion*/
                       subs = Alpha_to[sumi];
                       int_to_vector(subs,subs_v);
                       /* agrega el producto de los terminos del polinomio sustituido*/
                       for(si=0;si<gf_mm;si++) {
                              if(subs_v[si]) {
                                     logic_t[si][cnt[si]].a = ai;
                                     logic_t[si][cnt[si]].b = bi;
                                     cnt[si] += 1;
                              }
                       }
                } else {
                       logic_t[sumi][cnt[sumi]].a = ai;
                       logic_t[sumi][cnt[sumi]].b = bi;
                       cnt[sumi] += 1;
                }
         }
  }

  for(i=0;i<gf_mm;i++) c[i] = 0;

  /* Resultados*/
  for(i=0;i<gf_mm;i++) {
         for(k=0;k<cnt[i];k++) c[i] ^= a[logic_t[i][k].a] * b[logic_t[i][k].b];

  }

  result = vector_to_int(c);;

  free(a);
  free(b);
  free(c);
  free(subs_v);

  return(result);

}/* gf_mul */
```

REFERENCIAS.

[1] *Martyn Riley and Lain Richardson, "Red-Solomon Codes",*
http://www.4i2i.com/reed_solomon_codes.htm

[2] *J. Toledo*, "*Introducción a los códigos correctores de errores"*
http://personales.mundivia.es/jtoledo/angel/error/error51.htm

[3] "Wikipedia the free encyclopedia", http://en.wikipedia.org/wiki/Reed-Solomon_error_correction

[4] "Reed Solomon Module for Calculus I "North Carolina A & T State University and High Point University, http://www.radford.edu/~npsigmon/grant/rscode/rshtml/reeds_0.htm