

# **Implementación de un codificador y decodificador Reed-Solomon a través de una FPGA**

*DIEGO ANTONIO MONTAÑEZ SÁENZ*

*JUAN SEBASTIÁN VIZCAÍNO CASTRO*



**Universidad Distrital Francisco José de Caldas  
Facultad de Ingeniería Proyecto Curricular de Ingeniería Electrónica  
Bogotá D.C.  
2013**

# **Implementación de un codificador y decodificador Reed-Solomon a través de una FPGA**

*DIEGO ANTONIO MONTAÑEZ SÁENZ*

*JUAN SEBASTIÁN VIZCAÍNO CASTRO*

Trabajo de grado presentado como requisito parcial para optar al título de:

*Ingeniero Electrónico.*

Director:

MSc. Iván Ladino Vega

Codirector:

MSc. Juan Carlos Gómez

**Universidad Distrital Francisco José de Caldas  
Facultad De Ingeniería, Proyecto Curricular De Ingeniería Electrónica  
Bogotá D.C.  
2013**



Nota de aceptación:

---

---

---

---

---

---

---

Firma del Director del Trabajo de Grado

---

Firma del Co-director del Trabajo de Grado

---

Firma del jurado

---

Firma del jurado

Bogotá, Abril 22 de 2013

## Dedicatoria

A mis padres quienes siempre me han apoyado en todo. A mi hermana por su apoyo incondicional y motivación permanente. A Daniel y Camilo quienes me acompañaron y apoyaron durante toda la carrera y a todos mis compañeros de quienes aprendí y viví muchas experiencias.

**Juan Sebastián Vizcaíno Castro**

Ante todo a Dios por regalarme la vida y permitirme estudiar esta maravillosa carrera, a mi familia por el sacrificio y apoyo condicional que me prestaron durante todo este proceso, a mi compañero de trabajo de grado por el compromiso adquirido y a mis demás compañeros y amigos que dejaron una huella imborrable durante estos cinco años de formación académica.

**Diego Antonio Montañez Sáenz**

# Agradecimientos

Agradecimientos a nuestro director de tesis, MSc. Iván Ladino Vega, por su dedicación, apoyo, sugerencias y correcciones, que hicieron posible el desarrollo de este trabajo.

Al MSc. Juan Carlos Gómez por su, por su paciencia, apoyo y participación durante el proceso de la investigación.

A nuestras familias y amigos por su paciencia e incondicional apoyo. Y ante todo, a Dios por permitir que lográramos culminar esta etapa de nuestras vidas de manera exitosa.

# Resumen

Los códigos de detección y corrección de errores han sido ampliamente utilizados en las últimas décadas, por que gracias a ellos se pueden garantizar transmisiones confiables y así evitar la perdida de información. Gracias a estos códigos, es posible lograr transmitir a distancias tan lejanas como las encontradas en el espacio exterior y sin perdida de la información, como lo hacen las sondas espaciales. Debido a esto, el estudio de los códigos de corrección y detección de errores se ha venido estudiando desde la década de los 60's hasta hoy.

Dentro de todos los códigos que se han propuesto e implementado, hay uno en particular que esta presente en la mayoría de las aplicaciones. El código Reed-Solomon ha sido aplicado en áreas como la telefonía móvil, dispositivos de almacenamiento de información como los CD, sondas espaciales (la sonda Gallileo a Júpiter en 1989, la sonda Magallanes a Venus ese mismo año o la sonda Ulises al Sol en 1990, por citar algunos ejemplos) y por supuesto en las transmisiones por satélite Digital Video Broadcasting (DVB), la televisión digital ISDB-T, así como en los sistemas de xDLS de comunicación por cable.

Este libro trata sobre la implementación de los códigos Reed-Solomon, en el primer capítulo se exponen los objetivos, continuando con la introducción a los diferentes tipos de códigos de corrección y detección de errores. Luego se explica la importancia de los campos de Galois en la construcción de estos códigos y como se implementan las operaciones de suma y multiplicación para poder trabajar con los códigos Reed-Solomon. Posteriormente, se describe la idea detrás de la codificación Reed-Solomon, se describe el algoritmo de codificación y la implementación del mismo, generalizándolo para la creación de cualquier código Reed-Solomon que se deseé, luego se da inicio a la explicación del decodificador y los diferentes módulos que lo componen y del mismo modo, en el decodificador se generaliza toda la implementación y algoritmo para poder construir el decodificador para cualquier código Reed-Solomon. En el capítulo seis se describe la comunicación usada entre la FPGA y el ordenador para poner a prueba el código Reed-Solomon y se da una explicación del panel de control que se encarga de la configuración del codificador, modulo de ruido y decodificador dentro de la FPGA. En el capítulo siguiente se explica como se genero el ruido dentro de la FPGA para fines de pruebas y validación del código. Por ultimo se exponen los resultados y conclusiones de la implementación y se proponen algunos trabajos futuros en el área.

# Abstract

Detection and correction codes have been widely used in the last decades, because of them it is possible to guarantee reliable transmissions and in this way, avoiding the loss of information. Thanks to these codes, it is possible to send information to so long distances as those we can find in the space and with no loss of information. Because of this, the study of the correction and detection codes has been studied from 60's to now.

Among all codes which have been implemented and purposed, there is one which is presented in the most applications. Reed-Solomon codes have been implemented in areas such as mobile telephony, storage devices (CD), space crafts (the Galileo spacecraft to Jupiter in 1989, the Magellan *spacecraft* to Venus same year or the Ulysses spacecraft to the sun in 1990) and of course on satellite transmissions Digital Video Broadcasting (DVB), digital television ISDB-T and on xDLS systems of wired communications.

This book is about the implementation of Reed-Solomon codes, in the first chapter the objectives of this thesis are exposed, continuing with the introduction of the different kind of error correction and detection codes. After that, we explain the importance of the Galois Fields to build up these codes and how the addition and multiplication operation are implemented in order to work with Reed-Solomon codes. Later, it is described the idea behind Reed-Solomon coding, the coding algorithm is shown along with its implementation. This implementation is generalized to any Reed-Solomon code. After that we explain Reed-Solomon decoding and each module that compose it and in the same way as coding we generalize the algorithm and the implementation to be able to build up any Reed-Solomon decoding. In the chapter 6 it is described the type of communication that was used between FPGA and the computer which is used to test the Reed-Solomon code, besides, we explain about the control panel that handles the configuration of coding, noise module and decoding inside FPGA in order to validate the code. Finally, we expose the results and conclusions of the implementation and it is proposed some future works in the area.

# Contenido

<b>INTRODUCCIÓN.....</b>	14
1.1.    Justificación.....	17
1.2.    Objetivos.....	18
1.2.1. <i>Objetivos específicos.....</i>	18
<b>2. CAMPOS DE GALOIS Y SU IMPORTANCIA EN LOS CÓDIGOS DE CORRECCIÓN Y DETECCIÓN DE ERRORES.....</b>	19
2.1.    Definición de un campo de Galois.....	20
2.1.1. Algoritmo e implementación .....	24
2.2.    Operaciones en campos de Galois.....	27
2.2.1. Suma.....	27
2.2.1.1. Implementación.....	27
2.2.2. Multiplicación.....	28
2.2.2.1. Algoritmo e implementación.....	28
2.2.3. Potenciación.....	29
2.2.3.1. Algoritmo e implementación.....	29
2.2.4. Inverso.....	30
<b>3. CODIFICADOR REED-SOLOMON.....</b>	31
3.1.    Construcción del polinomio generador.....	31
3.2.    Calculo de $p(x)$ .....	33
3.3.    Sistema de corrimiento de registros de (n-k) etapas.....	33
<b>4. DECODIFICADOR REED-SOLOMON.....</b>	36
4.1.    Síndrome.....	38
4.1.1. Algoritmo e implementación.....	39
4.2.    Localización del error.....	43
4.2.1. Algoritmo de Berlekamp-Massey.....	45
4.2.2. Implementación .....	48
4.3.    Calculo de las raíces del polinomio localizador del error.....	53
4.3.1. Algoritmo propuesto.....	53
4.3.2. Algoritmo de Chien.....	56
4.4.    Calculo del valor de los errores.....	57

4.4.1.	Algoritmo de Forney.....	58
4.4.2.	Calculo de la derivada de $\sigma(x)$ .....	61
4.5.	Corrección del error.....	61
4.5.1.	Algoritmo e implementación.....	62
<b>5.</b>	<b>GENERACION DE RUIDO EN LA FPGA.....</b>	<b>64</b>
5.1.	Generador de números pseudo-aleatorios.....	64
5.1.1.	Serie pseudo-aleatoria.....	64
5.1.2.	Generando una secuencia pseudo-aleatoria.....	64
5.2.	Error de bit y error de ráfaga.....	64
5.3.	Modulo generador de error para prueba del código Reed-Solomon.....	67
<b>6.</b>	<b>SOFTWARE.....</b>	<b>73</b>
6.1.	Comunicación entre el software y la FPGA.....	74
6.1.1.	Modulo de recepción.....	74
6.1.2.	Modulo de transmisión.....	76
6.2.	Descripción del panel de control.....	78
6.3.	Generación de entidades VHDL a través del software.....	81
6.4.	Calculadora de campos finitos.....	83
<b>7.</b>	<b>RESULTADOS Y DISCUSION.....</b>	<b>87</b>
7.1	Funcionamiento del codificador.....	89
7.2	Funcionamiento del decodificador.....	92
7.3	Funcionamiento del síndrome.....	92
7.4	Funcionamiento del Berlekamp-Massey.....	95
7.5	Funcionamiento del Chien.....	99
7.6	Funcionamiento del Forney.....	102
7.7	Funcionamiento del decodificador.....	106
7.8	Consumo de hardware. ....	109
7.9	Calculadora.....	112
<b>8.</b>	<b>CONCLUSIONES Y TRABAJO FUTURO.....</b>	<b>115</b>
<b>9.</b>	<b>ANEXOS.....</b>	<b>117</b>
<b>10.</b>	<b>GLOSARIO.....</b>	<b>121</b>
<b>11.</b>	<b>BIBLIOGRAFÍA.....</b>	<b>122</b>

# Lista de imágenes

Figura. 1.1 Diagrama de bloques de todo el sistema implementado.....	20
Figura. 2.1 Comportamiento cíclico del campo de Galois.....	20
Figura. 2.2 Circuito propuesto para la generación de un $GF(2^m)$ .....	20
Figura. 2.3 Circuito propuesto para la generación de un $GF(2^4)$ .....	20
Figura. 2.4 Diagrama de flujo para la generación del campo de Galois.....	20
Figura. 2.5 Circuito para realizar la suma en un Campo de Galois.....	20
Figura. 2.6 Circuito propuesto para realizar la multiplicación en un campo de Galois.....	20
Figura. 2.7 Circuitos en un $GF(2^4)$ para los conversores.....	20
Figura. 2.8 Circuito para realizar la operación de potencia dentro de un campo de Galois.....	20
Figura. 2.9 Circuito para realizar la inversa dentro de un campo de Galois.....	20
Figura. 3.1 Cada cuadro representa un símbolo dentro de una cadena de bits.....	20
Figura. 3.2 Cadena de símbolos de paridad.....	20
Figura. 3.3 Al mensaje se le añaden los bits de paridad, formando una palabra código.....	20
Figura. 3.4 LFSR utilizado para realizar la división.....	20
Figura. 4.1 Diagrama de bloques del decodificador.....	20
Figura. 4.2 Circuito propuesto para el cálculo de un coeficiente de $S(X)$ .....	20
Figura. 4.3 Note que este circuito calcula el polinomio $S(X)$ en $n$ ciclos de reloj.....	20
Figura. 4.4 Diagrama de flujo para el cálculo de un coeficiente de $S(x)$ .....	20
Figura. 4.5 Diagrama de flujo del algoritmo Berlekamp-Massey propuesto en [2].....	20
Figura. 4.6 Circuito propuesto para calcular $d$ .....	20
Figura. 4.7 Primera bifurcación del algoritmo.....	20
Figura. 4.8 Se realiza el proceso A cuando la condición ( $d=0$ ) es positiva.....	20
Figura. 4.9 Proceso B, note que el proceso B1 se realiza independiente del resultado de la bifurcación.....	20
Figura. 4.10 Circuito encargado de ejecutar el proceso D. ....	20
Figura. 4.11 Circuito propuesto para el cálculo de $dd_m^{-1}p(x)$ . ....	20
Figura. 4.12 Circuito propuesto para el cálculo de $\mathbf{c}(x) + \mathbf{h}(x)$ .....	20
Figura. 4.13 Circuito propuesto para el cálculo de las raíces de $\sigma(x)$ . ....	20
Figura. 4.14 Representación circuito de la búsqueda de Chien. ....	20
Figura. 4.15 Circuito propuesto para el cálculo de la ecuación clave del algoritmo de Forney...20	20
Figura. 4.16 Circuito propuesto para el cálculo de la derivada de $\sigma(x)$ .....	20
Figura. 4.17 Circuito propuesto para efectuar la corrección del error.....	20
Figura. 5.1 LFRS para generar una secuencia pseudo-aleatoria.....	20
Figura. 5.2 Error de bit en una transmisión serial.....	20
Figura. 5.3 Error de ráfaga en una transmisión serial.....	20
Figura. 5.4 Distribución de error para un código 7-3 con sigma de 0.5.....	20

Figura. 5.5 Parte de la implementación del generador de ruido.....	20
Figura. 5.6 Diagrama de bloques del canal implementado dentro de la FPGA.....	20
Figura. 5.7 Función $X^2$ (representación para un código con un n mayor a 16).....	20
Figura 5.8 Función $X^2$ , representación para un código con un n=7.....	20
Figura. 5.9 Diagrama de flujo del proceso para llevar a cabo la configuración y puesta en funcionamiento del codificador y decodificador RS. Asociado al software.....	20
Figura. 6.1 Diagrama de conexionado del sistema implementado.....	20
Figura. 6.2 Trama transmitida.....	20
Figura. 6.3 Diagrama de flujo del receptor (UART).....	20
Figura. 6.4 Diagrama de flujo transmisión (UART).....	20
Figura. 6.5 Multiplexor a la salida del modulo de trasmisión.....	20
Figura. 6.6 Entidad - modulo de comunicación UART.....	20
Figura. 6.7 UCF utilizado en la implementación.....	20
Figura. 6.8 Configuración del error en el software de control.....	20
Figura. 6.9 Diagrama de clases del software.....	20
Figura. 7.1 Proceso de generación de archivos .vhdl.....	20
Figura. 7.2 Generando las entidades VHDL a partir del panel de control.....	20
Figura. 7.3 Entidades VHDL generadas por el software y listas para simular.....	20
Figura. 7.4 Circuito del codificador RS(15,9).....	20
Figura. 7.5 Simulación del funcionamiento del codificador.....	20
Figura. 7.6 Circuito de implementación del síndrome.....	20
Figura. 7.7 Simulación del funcionamiento del síndrome.....	20
Figura. 7.8 Circuito del algoritmo Berlekamp-Massey—Parte A.....	20
Figura. 7.9 Circuito del algoritmo Berlekamp-Massey—Parte B.....	20
Figura. 7.10 Simulación del funcionamiento del Berlekamp-Massey.....	20
Figura. 7.11 Circuito del algoritmo de Chien.....	20
Figura. 7.12 Simulación del funcionamiento del algoritmo de Chien.....	20
Figura. 7.13 Circuito del algoritmo Forney.....	20
Figura. 7.14 Simulación del funcionamiento del algoritmo de Forney.....	20
Figura. 7.15 Circuito del algoritmo de corrección.....	20
Figura. 7.16 Simulación del funcionamiento del algoritmo de corrección.....	20
Figura. 7.17 Circuito decodificador RS(15,9).....	20
Figura. 7.18 Simulación del decodificador RS(15,9).....	20
Figura. 7.19 Resultados del panel de control.....	20
Figura. 7.20 Incremento del número de slices con el incremento en el número de bits por símbolo.....	20
Figura. 7.21 Incremento del número de flip-flops con el incremento en el número de bits por simbolo.....	20

Figura. 7.22 Incremento del número de LUTs con el incremento en el número de bits por símbolo.....	20
Figura. 7.23 Incremento del número de slices con el incremento en el número de símbolos de paridad.....	20
Figura. 7.24 Incremento del número de flip-flops con el incremento en el número de símbolos de paridad.....	20
Figura. 7.25 Incremento del número de LUTS con el incremento en el número de símbolos de paridad.....	20
Figura. 7.26 Validación del funcionamiento de la calculadora modo alfas.....	20
Figura. 7.27 Validación del funcionamiento de la calculadora modo binario.....	20
Figura. 7.28 Validación del funcionamiento de la calculadora modo conversor.....	20
Figura. A1 Spartan 3 <sup>a</sup> AN apariencia física.....	20
Figura. B1 Conversor USB-SERIAL utilizado en la comunicación. Apariencia física.....	20

## Lista de tablas

Tabla 2.1 Definicion de los primeros elementos del campo.....	20
Tabla 2.2 Lista de polinomios primitivos con su representación binaria a utilizar.....	20
Tabla 2.3 Prueba de escritorio del circuito propuesto para generar un $GF(2^4)$ .....	20
Tabla 2.4 Tabla de suma para un $GF(2^4)$ .....	20
Tabla 2.5 Tabla de multiplicación para un $GF(2^4)$ .....	20
Tabla 4.1 Salida de la entidad contador para calcular los n-k coeficientes del síndrome.....	20
Tabla 4.3. Salidas del contador del síndrome.....	20
Tabla 4.4. Variación de $c(x) = c(x) + x^l h(x)$ con respecto a $l$ .....	20
Tabla 4.5. Salidas de los contadores que son multiplicadas por los coeficientes de $\sigma(x)$ representados como $c_i$ para $i = 1, 2, \dots, v$ .....	20
Tabla 4.6. Calculo de la inversa a través de una tabla.....	20
Tabla 5.1 Bits de realimentación según longitud de LFSR.....	20
Tabla 6.1 Comandos de inicio para la FPGA.....	20
Tabla 7.1 Prueba de escritorio codificador RS(7,3).....	20
Tabla 7.2 Prueba de escritorio síndrome RS(15,9).....	20
Tabla 7.3 Prueba de escritorio circuito del algoritmo Berlekamp-Massey.....	20
Tabla 7.4 Prueba de escritorio circuito del algoritmo Chien.....	20
Tabla 7.5 Prueba de escritorio circuito de algoritmo de Forney.....	20
Tabla 7.6 Prueba de escritorio algoritmo de corrección.....	20
Tabla 7.7 Resumen del consumo de hardware dependiendo de la variación de bits por símbolo.....	20
Tabla 7.8 Resumen del consumo de hardware dependiendo de los símbolos de paridad.....	20

# 1

## Introducción

Hoy en día la ingeniería en telecomunicaciones se ha dedicado a desarrollar sistemas de comunicaciones donde la rapidez y confiabilidad resultan fundamentales para responder a la demanda y necesidades de la sociedad actual. Uno de los problemas a los que se ha enfrentado es a los errores que se presentan en dichos sistemas debido a las características no ideales del canal de transmisión, por esto, uno de los retos de la industria de las telecomunicaciones es desarrollar sistemas libres de errores. Para esto se han desarrollado códigos que permiten la detección de los mismos para su posterior corrección, ya que en muchas ocasiones es preferible corregir posibles errores en el mensaje a tener que retransmitir este. La implementación de estos códigos se ha convertido en una de las disciplinas más relevantes cuando de diseñar soluciones de telecomunicaciones se trata.

Aunque el desarrollo de estos códigos se asoció inicialmente con la solución a los problemas que confrontaba (y confronta) la transmisión de información en sistemas de telecomunicaciones, aparecieron posteriormente nuevos campos de aplicación, por ejemplo, almacenamiento y protección de datos en memorias de computadoras, discos y cintas magnéticas, protección del funcionamiento de circuitos digitales contra el ruido, entre otros. En aplicaciones militares, donde también son ampliamente usados, tienen como propósito proteger la información contra interferencias provenientes del enemigo. También se utilizan para resolver problemas de potencia dada las limitaciones que muchos de los sistemas de comunicaciones tienen en cuanto a la potencia de la señal que pueden proveer, por ejemplo, puede resultar muy costoso transmitir alta potencia en sistemas de comunicaciones satelitales.

El desarrollo de estos códigos en las aplicaciones anteriormente mencionadas y en las demás aplicaciones en las que se ve involucrada la transmisión de datos se ha dado gracias al desarrollo de herramientas matemáticas como los campos de Galois , ya que el funcionamiento de estos códigos consiste en añadir bits de redundancia al mensaje, que sirven para determinar la ubicación de los errores en el mismo si es que los hay, para entonces pasar a su posterior corrección si es posible en el receptor y así recuperar el mensaje original. Las operaciones

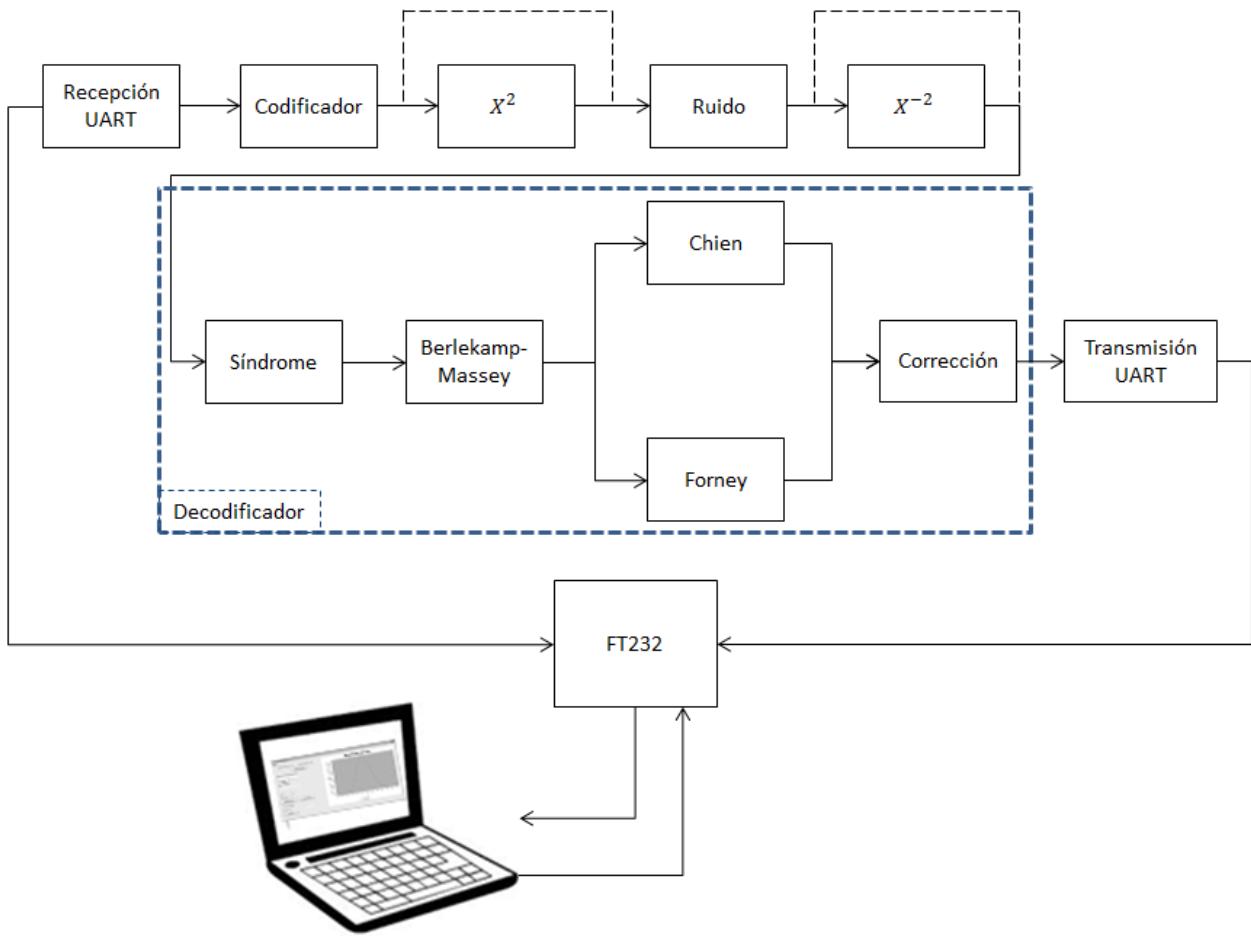
anteriores, detección y corrección, de los posibles errores, están fundamentadas en los procedimientos a que se someten los elementos de dichos campos.

Dentro de estos códigos encontramos los códigos BCH los cuales son códigos de corrección de múltiple error (multiple-error correcting codes) y fueron desarrollados por Bose and Ray-Chaudhuri (1960), Hocquenghem (1959) y Reed y Solomon (1960). Estos últimos desarrollaron una clase especial de códigos no binarios llamados códigos Reed-Solomon<sup>[1]</sup>. Los códigos Reed-Solomon son los más utilizados ya que se encuentran en sistemas de comunicaciones móviles como los celulares, en comunicaciones satelitales, en televisión digital (DVB), en módems de alta velocidad como ADSL y dispositivos de almacenamiento como DVD.

Estos códigos pueden ser implementados por hardware o por software dependiendo de la velocidad y el rendimiento que se requiera. La detección y corrección de errores cuando la tasa de transmisión es alta, demanda que los códigos exhiban un alto rendimiento, por lo tanto, se requiere de una implementación mediante hardware, mientras que sistemas con una baja tasa de transmisión de datos pueden soportar una implementación mediante software. Los códigos Reed-Solomon son implementados en hardware combinando así las dos bondades descritas anteriormente.

La implementación en hardware de los códigos Reed-Solomon usualmente se realiza por medio de FPGA debido a la velocidad de procesamiento que esta ofrece. Para lograr tal cosa resulta conveniente diseñar los circuitos digitales por medio de lenguajes de descripción de hardware como VHDL, Verilog, entre otros, ya que, por medio de estos se realiza la programación de dispositivos lógicos programables (PLDs, CPLD y FPGA), los cuales permiten una depuración y prueba de diseños digitales de alta complejidad.

En este proyecto se implementó a través de una FPGA un codificador y decodificador Reed-Solomon RS para detectar y corregir errores en canales lineales y no lineales bajo diferentes condiciones de ruido. El serial que posee la tarjeta de desarrollo fue utilizado para enviar los datos al codificador y recibirlas desde el decodificador, visualizando estos datos una vez recibidos en un panel simulado por medio de una herramienta de computación, la figura 1.1 muestra el diagrama de todo el sistema que se pretende realizar. Esta herramienta será de gran utilidad en los laboratorios de la Universidad Distrital Francisco José de Caldas ya que, les permitirá a los estudiantes hacer estudio del código RS en los cursos de comunicaciones o sobre las aplicaciones anteriormente mencionadas. Además, a partir de esta versión, será posible implementar futuros desarrollos sobre este tipo de técnica de corrección y detección de errores tanto académicas como de mercado.



*Figura. 1.1 Diagrama de bloques de todo el sistema implementado.*

## 1.1 Justificación

---

Es importante que en los laboratorios de la Universidad Distrital Francisco José de Caldas se cuente con los elementos necesarios para que los estudiantes puedan afianzar y desarrollar sus capacidades, en específico herramientas que les sirvan para poner en práctica los conocimientos adquiridos.

Teniendo en cuenta lo anteriormente expuesto, es importante desarrollar un codificador y un decodificador Reed-Solomon que sirva como herramienta de apoyo en los cursos de comunicaciones, telecomunicaciones y grupos de investigación referentes al área.

La implementación de dicho codificador y decodificador es importante hacerlo sobre una FPGA[9] teniendo en cuenta la capacidad y velocidad de procesamiento que esta nos ofrece a un costo bajo, además de ser un elemento con el que la universidad cuenta en este momento. Por lo cual se puede afirmar que el proyecto es económicamente viable ya que es algo concreto que se pretende realizar con recursos que la Universidad posee, además de ser ejemplo para otras instituciones académicas en el fomento de elaboración de herramientas propias para desarrollos sobre estas.

Además de lo anterior se debe tener en cuenta que existe un mercado laboral en el cual las exigencias son cada vez mayores, el estudio y profundización en códigos Reed-Solomon proporcionará herramientas muy útiles para desenvolverse en dicho ámbito, ya que el conocimiento de estos códigos es una necesidad laboral debido a la cantidad de aplicaciones en los que se encuentran hoy en día como lo son las comunicaciones satelitales. A la vez este trabajo de grado servirá para incentivar a la comunidad académica a realizar investigaciones alrededor de los códigos BCH, temática que poco se ha estudiado y profundizado en la Universidad Distrital Francisco José de Caldas, además que servirá de herramienta para futuros desarrollos que se quieran elaborar en la universidad.

## 1.2 Objetivos

---

Implementar un codificador y decodificador Reed-Solomon RS a partir de los campos de Galois en canales lineales y no lineales bajo diferentes condiciones de ruido con el objeto de generar, detectar y corregir errores en sistemas de comunicaciones digitales a través de una FPGA.

### 1.2.1. Objetivos específicos.

- Elaborar una herramienta que calcule el campo de Galois y realice las operaciones de suma y multiplicación del campo; necesaria para facilitar el diseño del codificador y decodificador R-S.
- Diseñar el codificador y decodificador Reed-Solomon que va a ser implementado.
- Implementar sobre una FPGA el codificador y decodificador Reed-Solomon.
- Simular un canal lineal y un canal no lineal, con diferentes tipos de ruidos mediante la FPGA con el fin de probar el codificador y decodificador Reed-Solomon que fue implementado.
- Diseñar mediante el uso de una herramienta de computación un panel de control en forma de banco de prueba para probar el funcionamiento del codificador y decodificador Reed-Solomon ya implementado.

# 2

## Campos de Galois y su importancia en los códigos de corrección y detección de errores.

La historia de los códigos de transmisión de datos (data-transmission codes) empezó en 1948 con la publicación de un famoso artículo hecho por Claude Shannon. Shannon mostró que cualquier canal de comunicación tiene un número  $C$  asociado a la capacidad del canal (medido en bit por segundo) de lo cual, siempre que la tasa de transmisión requerida  $R$  sea menor que  $C$  se puede diseñar un sistema de comunicación para el canal que posea una probabilidad de error tan pequeña como se desee. Una importante conclusión de la teoría de la información de Shannon es que es una pérdida de tiempo tratar de mejorar la probabilidad de error en el proceso de modulación y demodulación, ya que es más barato y más efectivo usar un poderoso código de transmisión de datos. Sin embargo Shannon no dijo cómo encontrar estos códigos solo nos demostró la existencia de ellos y su rol [4].

Los códigos de transmisión de datos son un tema simple y difícil al mismo tiempo. Es simple en el sentido de que cualquier persona calificada puede entenderlo fácilmente, pero es difícil a la hora de desarrollar una solución al problema. Suponga que toda la información puede ser representada como datos binarios, es decir una secuencia de unos y ceros. Estos datos binarios son transportados a través de un canal binario que ocasionalmente produce errores en la transmisión. El propósito de un código es agregar símbolos extras a los datos, de modo que los errores puedan ser encontrados y corregidos en el receptor. Es decir, una secuencia de datos es representada por una secuencia de datos más larga, que contenga la suficiente redundancia para proteger los datos.

A partir de la matemática desarrollada por Galois, se logra tratar matemáticamente este problema y a partir de esto encontrar diferentes soluciones al problema de codificación que se necesita en los canales de comunicación.

## 2.1. Definición de un campo de Galois.

---

El funcionamiento de los códigos no-binarios<sup>1</sup> como el Reed Solomon (RS) está basado en la matemática de los campos finitos denominados campos de *Galois*<sup>2</sup>. Un campo de *Galois* está formado a partir de polinomios denominados *polinomios primitivos*, de dichos polinomios se crean los campos de Galois de  $p$  elementos denotados  $GF(p)$ , cabe mencionar que  $p$  debe ser un número primo para que el campo cumpla las propiedades de suma y multiplicación por cerradura.<sup>3</sup>

- *Cerradura por suma:* Para cada elemento  $a, b$  y  $c$  que pertenecen al campo,  $c = a + b$  pertenece al campo. En la ecuación 2.1 se muestra el elemento entidad para la suma que corresponde al elemento nulo

$$a + (-a) = (-a) + a = 0 \quad (2.1)$$

- *Cerradura por multiplicación:* Para cada elemento  $a, b$  y  $c$  que pertenecen al campo,  $c = a * b$  pertenece al campo. El elemento entidad en la multiplicación se define en la ecuación 2.2

$$a * e = e * a = a \quad (2.2)$$

A partir de un  $GF(p)$  se puede crear un campo de *Galois* extendido denotado como  $GF(p^m)$ , donde  $m$  puede ser cualquier número real positivo, el campo inicial  $GF(p)$  entra a ser un subconjunto de  $GF(p^m)$ . En la práctica se utiliza comúnmente los campos de *Galois* extendidos que surgen a partir de  $GF(2)$  denominado campo de *Galois* binario, debido a que solamente posee dos elementos el cero y el uno, precisamente por esta razón es que son los más utilizados en la implementación ya que la manipulación binaria que se puede hacer con estos campos es mucho más sencilla.

Al realizar el campo extendido  $GF(2^m)$  se crea un campo con  $2^m$  elementos, incluyendo el elemento nulo, los elementos de estos campos son los utilizados en la construcción de los

---

<sup>1</sup> No-Binario hace referencia a que los símbolos del Campo no son solo un uno o un cero.

<sup>2</sup> Los campos de *Galois* son conjuntos de números que cumplen ciertas propiedades como lo son: cerradura por adición y por multiplicación, es decir que cualquier operación que se realice entre dos elementos del campo debe dar como resultado un elemento del mismo campo, recordando que el número de elementos llamados símbolos es finito. También es importante mencionar que el inverso de todos los elementos del campo existe, así como también existe el elemento nulo, al realizar la suma de dos elementos del campo que son iguales se tiene como resultado el elemento nulo.

<sup>3</sup> Para profundizar sobre la construcción de campos de *Galois* a partir de Polinomios primitivos y toda la matemática que rodea este tema se recomienda ir a [8].

códigos Reed Solomon. Al ser  $GF(2)$  un subconjunto del campo extendido  $GF(2^m)$  los elementos del este campo también son elementos del campo extendido, por lo que los números 0 y 1 son los primeros elementos conocidos, sin embargo se sabe que deben existir  $2^m - 2$  elementos adicionales para completar el conjunto de componentes del campo extendido; estos elementos serán denotados de ahora en adelante por medio del símbolo  $\alpha$ , donde cada elemento del campo diferente de cero puede ser denotado por una potencia de  $\alpha$ .

A partir de lo anterior conociendo que todos los elementos del campo diferentes de cero se pueden representar por una potencia de  $\alpha$  se podría empezar definiendo que el elemento uno corresponde a  $\alpha^0$ , siguiendo con la lógica se empieza a construir los elementos del campo multiplicando el último elemento encontrado por  $\alpha$ , obteniendo como resultado un conjunto de  $F$  elementos representados en la ecuación 2.3.

$$F = \{0, \alpha^0, \alpha^1, \alpha^2, \dots, \alpha^j, \dots\} \quad (2.3)$$

Recordando las propiedades de un campo de *Galois* tenemos que el número de elementos del campo debe ser finito, por lo que uno los elementos generados en el conjunto  $F$  no son del todo diferentes y debe existir un punto donde se empiecen a repetir de forma cíclica como se muestra en la figura 1.

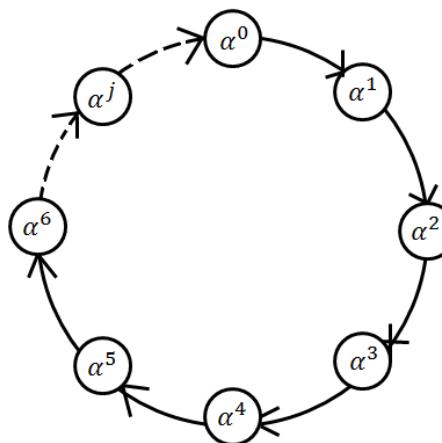


Figura. 2.1 Comportamiento cíclico del campo de *Galois*.

Partiendo de lo anteriormente expuesto y teniendo en cuenta que se mencionó que los elementos faltantes del campo por conocer son  $2^m - 2$ , se puede definir que el último elemento del campo se puede nombrar de la forma  $\alpha^{(2^m-2)}$ , de acuerdo al diagrama de la figura 1 el elemento que sigue en el campo es  $\alpha^{(2^m-1)}$  que debe ser igual a  $\alpha^0$  como se caracteriza por medio del polinomio irreducible mostrado en la ecuación 2.4.

$$\alpha^{(2^m-1)} + 1 = 0 \quad (2.4)$$

De la ecuación 2.4 se puede obtener la ecuación 2.5.

$$\alpha^{(2^m-1)} = 1 = \alpha^0 \quad (2.5)$$

Usando la definición de este polinomio, cualquier elemento del campo que tiene una potencia igual o mayor a  $2^m - 1$  puede ser reducido a un elemento con una potencia menor que  $2^m - 1$ , como se muestra en la ecuación 2.6.

$$\alpha^{(2^m+n)} = \alpha^{(2^m-1)}\alpha^{(n+1)} = \alpha^0\alpha^{(n+1)} = \alpha^{(n+1)} \quad (2.6)$$

A partir de 2.6 se puede redefinir el conjunto F como se muestra en la ecuación 2.7.

$$F=\{0, \alpha^0, \alpha^1, \alpha^2, \dots, \alpha^{2^m-2}, \alpha^0, \alpha^1, \dots\} \quad (2.7)$$

Cada uno de los  $2^m$  elementos del campo  $GF(2^m)$  puede ser representado por un polinomio distinto de grado  $m-1$  o menor, donde los coeficientes de dicho polinomio pertenecen al campo  $GF(2)$ . Se denota cada elemento diferente de cero de  $GF(2^m)$  como un polinomio,  $\alpha_i(x)$ , donde al menos uno de los coeficientes  $m$  de  $\alpha_i(x)$  es diferente de cero. Para  $i = 0, 1, 2, \dots, 2^m - 2$  como se muestra en la ecuación 2.8.

$$\alpha^i = \alpha_{i,0} + \alpha_{i,1}X + \alpha_{i,2}X^2 + \dots + \alpha_{i,m-1}X^{m-1} \quad (2.8)$$

### Creación de campos de Galois extendidos $GF(2^m)$

Se necesitan una clase especial de polinomios para definir un campo de Galois llamados, polinomios primitivos. Estos polinomios tienen la característica de que al dividirlos por  $X^n + 1$  donde  $n = 2m - 1$  se obtiene como resultado un residuo igual a cero y al dividirlos por  $n < 2m - 1$  el resultado generara un residuo diferente de cero.

Por ejemplo si se quisiera generar un código del RS(255,247) se debe utilizar un campo de  $GF(2^m)$  con  $m=8$  ya que se requiere que el número de símbolos sea igual al número de elementos del campo<sup>4</sup>, que para este caso es de 255 más el elemento nulo. Según la literatura el polinomio primitivo para generar  $GF(2^8)$  se define en la ecuación 2.9.

$$p(x) = 1 + X^2 + X^3 + X^4 + X^8 \quad (2.9)$$

Se define  $\alpha$  como una raíz del polinomio  $p(x)$  como se muestra en la ecuación 2.10.

$$p(\alpha) = 0 \quad (2.10)$$

---

<sup>4</sup> Puede ser igual o mayor, ya que existen códigos como el RS(249,237) que también funcionan sobre un  $RS(2^8)$ .

$$1 + \alpha^2 + \alpha^3 + \alpha^4 + \alpha^8 = 0 \quad (2.11)$$

A partir de la ecuación 2.11 se define.

$$1 + \alpha^2 + \alpha^3 + \alpha^4 = -\alpha^8 \quad (2.12)$$

Dado que en un campo binario  $-1=1$

$$1 + \alpha^2 + \alpha^3 + \alpha^4 = \alpha^8 \quad (2.13)$$

Se definen los primeros elementos como se muestran en la tabla 1, a partir de la teoría presentada anteriormente.

$\alpha^0$	1
$\alpha^1$	X
$\alpha^2$	$X^2$
$\alpha^3$	$X^3$
$\alpha^4$	$X^4$
$\alpha^5$	$X^5$
$\alpha^6$	$X^6$
$\alpha^7$	$X^7$

Tabla 2.1 Definición de los primeros elementos del campo.

A partir de los primeros elementos se pueden construir el resto de elementos del campo.

Por ejemplo partiendo de la ecuación 2.13 se obtiene  $\alpha^8$  como se muestra en la ecuación 2.14.

$$\alpha^8 = X^4 + X^3 + X^2 + 1 \quad (2.14)$$

Si multiplicamos por  $\alpha$  la ecuación 2.14 obtenemos  $\alpha^9$ , como se muestra en la ecuación 2.15.

$$\alpha^8\alpha = \alpha(\alpha^4 + \alpha^3 + \alpha^2 + 1) = \alpha^5 + \alpha^4 + \alpha^3 + \alpha \quad (2.15)$$

$$\alpha^9 = \alpha^5 + \alpha^4 + \alpha^3 + \alpha \quad (2.16)$$

$$\alpha^9 = X^5 + X^4 + X^3 + X \quad (2.17)$$

De la forma como se generó  $\alpha^9$ , se procede continuamente hasta generar todos los elementos del campo.

Para un diseño computacionalmente viable se escriben los coeficientes del polinomio formando un número binario de 8 bits, donde el grado de X representa la posición en el arreglo de 8 bits. Es decir, que el coeficiente que acompaña a  $X^7$  es representado por el bit más significativo mientras que coeficiente que acompaña a  $X^0$  se representa con el bit menos significativo del byte.

### 2.1.1. Algoritmo e implementación para generar un campo de Galois(n,k)

El campo de *Galois* se creó por medio del software reconfigurable desarrollado, del circuito propuesto en [XX] se propone una generalización del sistema para poder crear cualquier campo a partir de los polinomios primitivos  $p(x)$  conocidos para  $m$  entre 3 y 8, los cuales se muestran en la tabla 2.2.

<b>X</b>	<b>Polinomio primitivo</b>	<b>Representación binaria <math>p(x)</math></b>
3	$X^3 + X + 1$	11
4	$X^4 + X + 1$	11
5	$X^5 + X^2 + 1$	101
6	$X^6 + X + 1$	11
7	$X^7 + X^3 + 1$	1001
8	$X^8 + X^4 + X^3 + X^2 + 1$	11101

**Tabla 2.2** Lista de polinomios primitivos con su representación binaria a utilizar.

En la tabla también se muestra la representación binaria de  $p(x)$ , que se utilizó en el diseño del circuito. En la figura 2.2 se muestra el circuito propuesto para generar cualquier campo de Galois.

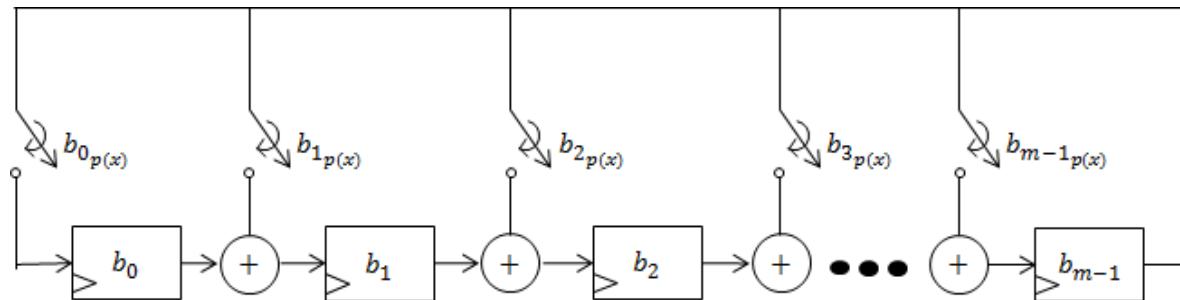


Figura. 2.2 Circuito propuesto para la generación de campos de Galois  $GF(2^m)$ .

Los valores  $b_{y_{p(x)}}$  con  $y$  entre 0 y  $m-1$  corresponden a los bits de la representación binaria de  $p(x)^5$ , si el circuito está abierto se tiene que la entrada del sumador es cero por lo que se podría omitir, por ejemplo si se desea crear un  $GF(2^4)$  se debe utilizar el  $p(x) = X^4 + X + 1$  cuya representación binaria es “11”, el circuito a utilizar se muestra en la figura 2.3.

<sup>5</sup> Note que en la representación binaria no se tiene en cuenta el bit más significativo ya que este es siempre uno y no tiene relevancia en el diseño del circuito.

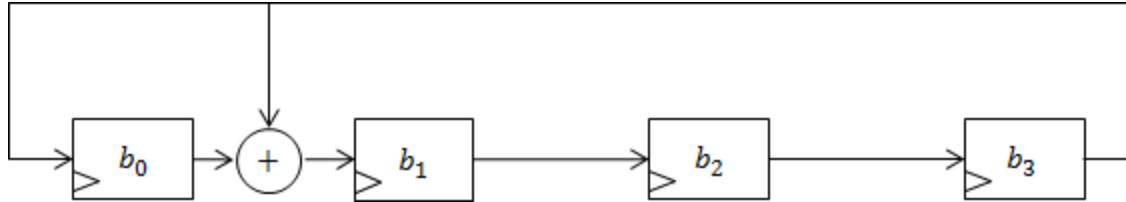


Figura. 2.3 Circuito propuesto para la generación de un  $GF(2^4)$

A partir del circuito presentado en la Figura 2.3 y teniendo en cuenta que la validación de los resultados en este libro se basa en un  $GF(2^4)$  se desarrolló la prueba de escritorio para verificar la validez del circuito y determinar los elementos del campo, partiendo de que los registros se cargan con el primer elemento  $\alpha^0$  correspondiente a "0001". La prueba de escritorio se muestra en la tabla 2.3.

<i>Clk</i>	<i>b3</i>	<i>b2</i>	<i>b1</i>	<i>b0</i>	<i>s_alpha</i>	<i>s_binario</i>	<i>s_decimal</i>	<i>rep_polinomial</i>
0	0	0	0	1	$\alpha^0$	0001	1	1
1	0	0	1	0	$\alpha^1$	0010	2	$X$
2	0	1	0	0	$\alpha^2$	0100	4	$X^2$
3	1	0	0	0	$\alpha^3$	1000	8	$X^3$
4	0	0	1	1	$\alpha^4$	0011	3	$X + 1$
5	0	1	1	0	$\alpha^5$	0110	6	$X^2 + X$
6	1	1	0	0	$\alpha^6$	1100	12	$X^3 + X^2$
7	1	0	1	1	$\alpha^7$	1011	11	$X^3 + X + 1$
8	0	1	0	1	$\alpha^8$	0101	5	$X^2 + 1$
9	1	0	1	0	$\alpha^9$	1010	10	$X^3 + X$
10	0	1	1	1	$\alpha^{10}$	0111	7	$X^2 + X + 1$
11	1	1	1	0	$\alpha^{11}$	1110	14	$X^3 + X^2 + X$
12	1	1	1	1	$\alpha^{12}$	1111	15	$X^3 + X^2 + X + 1$
13	1	1	0	1	$\alpha^{13}$	1101	13	$X^3 + X^2 + 1$
14	1	0	0	1	$\alpha^{14}$	1001	9	$X^3 + 1$
15	0	0	0	1	$\alpha^{15}$	0001	1	1

Tabla 2.3 Prueba de escritorio del circuito propuesto para generar un  $GF(2^4)$

De la prueba de escritorio se puede verificar el funcionamiento del circuito y la propiedad finita del campo de Galois ya que si observamos con detalle podemos ver que  $\alpha^{15}$  es igual a  $\alpha^0$ . A partir de esto se definen las operaciones de suma y multiplicación para dicho campo presentadas en las tablas 2.2 y 2.3 respectivamente.

Para ofrecer una mayor claridad, en la construcción del campo se muestra el diagrama de flujo de la figura 2.4.

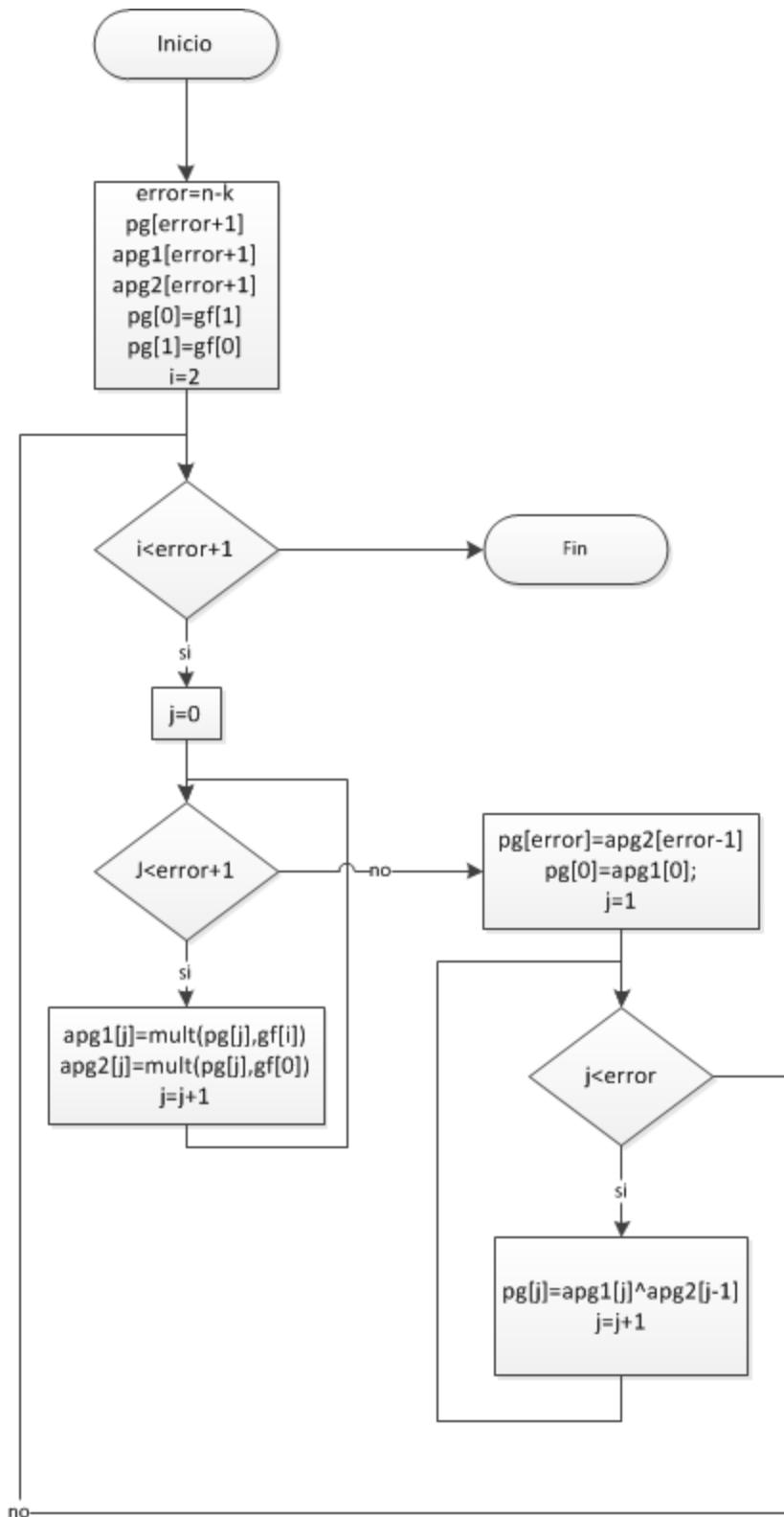


Figura. 2.4 Diagrama de flujo para la generación del campo de Galois.

## 2.2. Operaciones en campos de Galois

---

### 2.2.1 Suma

La adición de dos elementos del campo finito de *Galois* está definida como la suma en modulo-2 de cada uno de los coeficientes del polinomio de sus respectivas potencias, de la misma forma la multiplicación se efectúa como en un polinomio real pero las operaciones entre los coeficientes de los polinomios se realizan en modulo-2.

+	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$
$\alpha^0$	0	$\alpha^4$	$\alpha^8$	$\alpha^{14}$	$\alpha^1$	$\alpha^{10}$	$\alpha^{13}$	$\alpha^9$	$\alpha^2$	$\alpha^7$	$\alpha^5$	$\alpha^{12}$	$\alpha^{11}$	$\alpha^6$	$\alpha^3$
$\alpha^1$	$\alpha^4$	0	$\alpha^5$	$\alpha^9$	$\alpha^0$	$\alpha^2$	$\alpha^{11}$	$\alpha^{14}$	$\alpha^{10}$	$\alpha^3$	$\alpha^8$	$\alpha^6$	$\alpha^{13}$	$\alpha^{12}$	$\alpha^7$
$\alpha^2$	$\alpha^8$	$\alpha^5$	0	$\alpha^6$	$\alpha^{10}$	$\alpha^1$	$\alpha^3$	$\alpha^{12}$	$\alpha^0$	$\alpha^{11}$	$\alpha^4$	$\alpha^9$	$\alpha^7$	$\alpha^{14}$	$\alpha^{13}$
$\alpha^3$	$\alpha^{14}$	$\alpha^9$	$\alpha^6$	0	$\alpha^7$	$\alpha^{11}$	$\alpha^2$	$\alpha^4$	$\alpha^{13}$	$\alpha^1$	$\alpha^{12}$	$\alpha^5$	$\alpha^{10}$	$\alpha^8$	$\alpha^0$
$\alpha^4$	$\alpha^1$	$\alpha^0$	$\alpha^{10}$	$\alpha^7$	0	$\alpha^8$	$\alpha^{12}$	$\alpha^3$	$\alpha^5$	$\alpha^{14}$	$\alpha^2$	$\alpha^{13}$	$\alpha^6$	$\alpha^{11}$	$\alpha^9$
$\alpha^5$	$\alpha^{10}$	$\alpha^2$	$\alpha^1$	$\alpha^{11}$	$\alpha^8$	0	$\alpha^9$	$\alpha^{13}$	$\alpha^4$	$\alpha^6$	$\alpha^0$	$\alpha^3$	$\alpha^{14}$	$\alpha^7$	$\alpha^{12}$
$\alpha^6$	$\alpha^{13}$	$\alpha^{11}$	$\alpha^3$	$\alpha^2$	$\alpha^{12}$	$\alpha^9$	0	$\alpha^{10}$	$\alpha^{14}$	$\alpha^5$	$\alpha^7$	$\alpha^1$	$\alpha^4$	$\alpha^0$	$\alpha^8$
$\alpha^7$	$\alpha^9$	$\alpha^{14}$	$\alpha^{12}$	$\alpha^4$	$\alpha^3$	$\alpha^{13}$	$\alpha^{10}$	0	$\alpha^{11}$	$\alpha^0$	$\alpha^6$	$\alpha^8$	$\alpha^2$	$\alpha^5$	$\alpha^1$
$\alpha^8$	$\alpha^2$	$\alpha^{10}$	$\alpha^0$	$\alpha^{13}$	$\alpha^5$	$\alpha^4$	$\alpha^{14}$	$\alpha^{11}$	0	$\alpha^{12}$	$\alpha^1$	$\alpha^7$	$\alpha^9$	$\alpha^3$	$\alpha^6$
$\alpha^9$	$\alpha^7$	$\alpha^3$	$\alpha^{11}$	$\alpha^1$	$\alpha^{14}$	$\alpha^6$	$\alpha^5$	$\alpha^0$	$\alpha^{12}$	0	$\alpha^{13}$	$\alpha^2$	$\alpha^8$	$\alpha^{10}$	$\alpha^4$
$\alpha^{10}$	$\alpha^5$	$\alpha^8$	$\alpha^4$	$\alpha^{12}$	$\alpha^2$	$\alpha^0$	$\alpha^7$	$\alpha^6$	$\alpha^1$	$\alpha^{13}$	0	$\alpha^{14}$	$\alpha^3$	$\alpha^9$	$\alpha^{11}$
$\alpha^{11}$	$\alpha^{12}$	$\alpha^6$	$\alpha^9$	$\alpha^5$	$\alpha^{13}$	$\alpha^3$	$\alpha^1$	$\alpha^8$	$\alpha^7$	$\alpha^2$	$\alpha^{14}$	0	$\alpha^0$	$\alpha^4$	$\alpha^{10}$
$\alpha^{12}$	$\alpha^{11}$	$\alpha^{13}$	$\alpha^7$	$\alpha^{10}$	$\alpha^6$	$\alpha^{14}$	$\alpha^4$	$\alpha^2$	$\alpha^9$	$\alpha^8$	$\alpha^3$	$\alpha^0$	0	$\alpha^1$	$\alpha^5$
$\alpha^{13}$	$\alpha^6$	$\alpha^{12}$	$\alpha^{14}$	$\alpha^8$	$\alpha^{11}$	$\alpha^7$	$\alpha^0$	$\alpha^5$	$\alpha^3$	$\alpha^{10}$	$\alpha^9$	$\alpha^4$	$\alpha^1$	0	$\alpha^5$
$\alpha^{14}$	$\alpha^3$	$\alpha^7$	$\alpha^{13}$	$\alpha^0$	$\alpha^9$	$\alpha^{12}$	$\alpha^8$	$\alpha^1$	$\alpha^6$	$\alpha^4$	$\alpha^{11}$	$\alpha^{10}$	$\alpha^5$	$\alpha^2$	0

Tabla 2.4 Tabla de suma para un  $GF(2^4)$

#### Implementación del circuito de la Suma en un $GF(n,k)$

El circuito para realizar la suma es sumamente sencillo, la propuesta se muestra en la figura 5.



Figura. 2.5 Circuito para realizar la suma en un Campo de Galois.

Del circuito se puede mencionar que es una compuerta lógica XOR bit a bit, donde  $a$  y  $b$  son el valor en binario del símbolo y  $suma$  la salida de la operación.

## 2.2.2 Multiplicación

$\times$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$
$\alpha^0$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$
$\alpha^1$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$
$\alpha^2$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$
$\alpha^3$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$
$\alpha^4$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$
$\alpha^5$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$
$\alpha^6$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$
$\alpha^7$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$
$\alpha^8$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$
$\alpha^9$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$
$\alpha^{10}$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$
$\alpha^{11}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$
$\alpha^{12}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$
$\alpha^{13}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$
$\alpha^{14}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$

Tabla 2.5 Tabla de multiplicación para un  $GF(2^4)$

### 3.2.2.1 Algoritmo e implementación del circuito de la Multiplicación en un $GF(n,k)$

Recordando la teoría de las operaciones en un campo de Galois y observando los resultados obtenidos en las tablas 2.4 y 2.5 que la demuestran, podemos analizar que la multiplicación es más sencilla si la realizamos con la representación en *alfas* de los símbolos y la suma en la representación binaria. Cabe mencionar que la entrada de todos los bloques está en la representación binaria por lo que para realizar la implementación de la multiplicación en un campo de Galois se propone el circuito mostrado en la figura 2.6, siendo  $a$  y  $b$  los dos símbolos en su forma binaria a multiplicar y *mult* el resultado del cálculo.

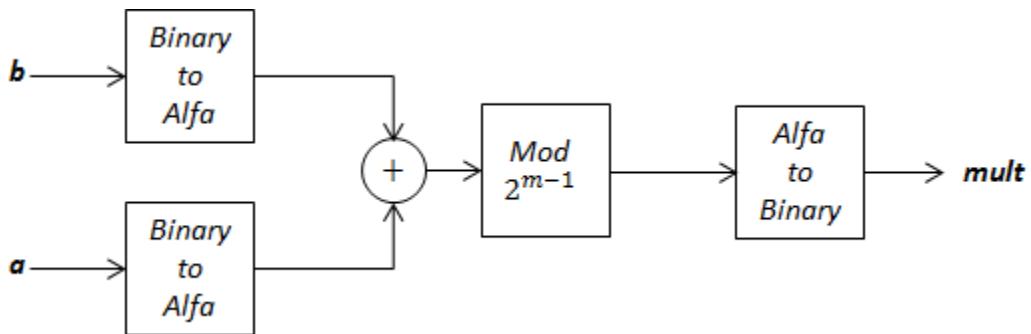


Figura. 2.6 Circuito propuesto para realizar la multiplicación en un campo de Galois

Cabe mencionar que la suma presente en el circuito es una suma normal en el campo de los números reales. El bloque que se añade para realizar la operación en  $\text{mod}(2^{m-1})$ , es para evitar que el resultado se desborde del tamaño del campo; es decir con este control se le da la propiedad de cerradura por multiplicación.

Los bloques *binary to alfa* y *alfa to binary*, son bloques creados para realizar las conversiones del valor del símbolo en alfa<sup>6</sup> al valor del símbolo en binario y viceversa. En la figura 2.7 se presenta un ejemplo de estos bloques para un  $GF(2^4)$ .<sup>7</sup>

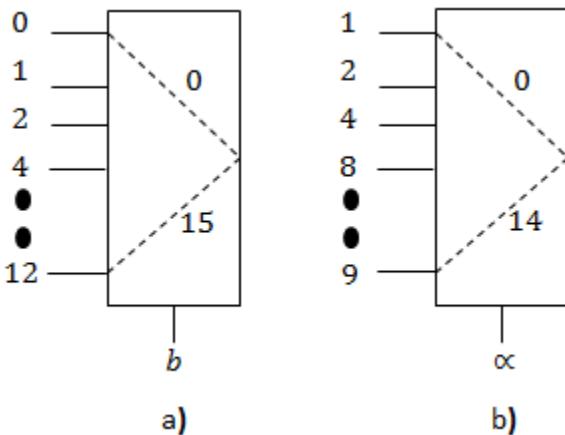


Figura. 2.7 Circuitos en un  $GF(2^4)$  para los conversores a) De binario a alfa b) De alfa a binario

### 2.2.3 Potenciación

**Implementación y algoritmo para realizar la Potencia.**

En la figura 2.8 se presenta el circuito propuesto para realizar el cálculo de la potencia dentro de un campo de Galois, definida esta como se muestra en la ecuación 1.

$$\text{alfa} = \text{raiz}^{\text{potencia}} \quad (1)$$

---

<sup>6</sup> Cuando se habla del valor del símbolo en alfa se hace referencia al valor de la potencia, por ejemplo si el símbolo es  $\alpha^0$ , el valor en alfa es 0 y el valor en binario es 1.

<sup>7</sup> Para realizar la descripción de hardware se realizaron Bancos de Registros estáticos que se crean por medio del Software desarrollado.

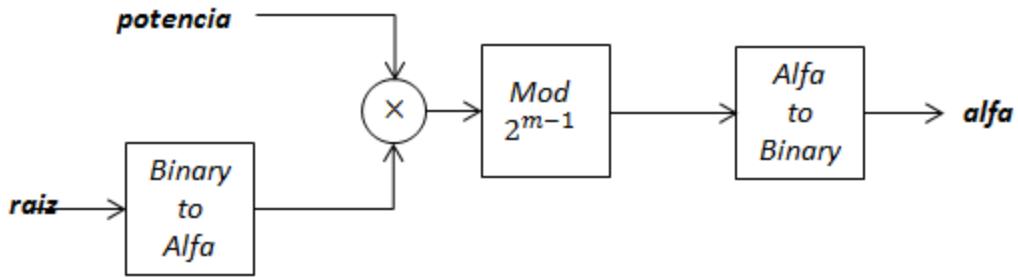


Figura. 2.8 Circuito para realizar la operación de potencia dentro de un campo de *Galois*.

#### 2.2.4 Inversa

En la figura 2.9 finalmente se muestra el circuito propuesto para realizar el cálculo de inversa dentro de un campo de *Galois* siendo este como se describe en la ecuación 2.

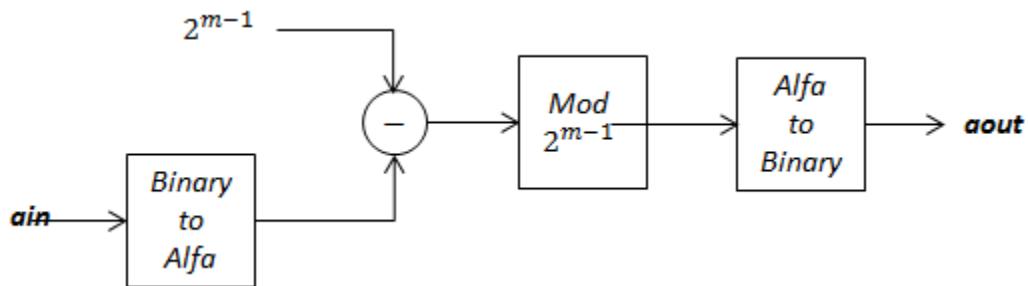


Figura. 2.9 Circuito para realizar la inversa dentro de un campo de *Galois*.

$$aout = \frac{1}{ain} (2)$$

Esta tarea en la descripción de hardware se puede realizar directamente por medio de un banco de registros, donde el selector del banco sea *ain* y la salida del banco sea *aout*, el valor de los registros se obtienen por medio del software desarrollado, siendo este estático y calculado por medio del circuito mostrado en la figura 2.9.

# 3

## Codificador Reed-Solomon

### 3.1. Construcción del polinomio generador.

---

Un código Reed-Solomon se expresa de la forma  $R(n, k)$ , donde  $n$  es el número de símbolos del campo y  $k$  el número de símbolos que componen el mensaje. El número de símbolos de paridad está dado por  $n - k = 2t$ , donde  $t$  representa el número máximo de símbolos erróneos que puede corregir el código. Para adicionar estos  $2t$  símbolos de paridad se utiliza un polinomio que los genere, llamado polinomio generador. El polinomio generador para un código R-S puede ser expresado de la siguiente forma.

$$g(x) = \sum_{i=0}^{2t} g_i X^i$$

$$g(x) = g_0 + g_1 X + g_2 X^2 + \cdots + g_{2t-1} X^{2t-1} + X^{2t} \quad (3.1)$$

Note que el grado del polinomio generador  $g(x)$  es igual al número de símbolos de paridad<sup>8</sup>. Por consiguiente deben existir  $2t$  raíces para  $g(x)$ , siendo estas potencias sucesivas de  $\alpha$ , es decir que las raíces de  $g(x)$  serán  $\alpha + \alpha^2, \dots, \alpha^{2t}$ . No es necesario empezar con la raíz  $\alpha$ , se puede empezar con cualquier otra raíz sin ningún problema. De modo que también podemos representar a  $g(x)$  en términos de sus raíces como se describe a continuación.

$$g(x) = \prod_{i=0}^{2t} (X - \alpha^i)$$

---

<sup>8</sup> Los códigos R-S son un subconjunto de los códigos BCH (Bose, Chaundhuri, y Hocquenghem), de modo que no es de sorprenderse que exista relación entre el numero de símbolos de paridad y el grado del polinomio generador.

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

A partir de la representación de  $g(x)$  en términos de sus raíces, es sencillo construir el polinomio en su forma expandida (ecuación 3.1.) para cualquier código R-S. A partir de ahora vamos a describir el algoritmo utilizado para generar los símbolos de paridad que se añadirán al mensaje.

Los códigos R-S son códigos cíclicos que se codifican en forma sistemática, análoga al proceso de codificación binaria [1]. Si se mira de una manera práctica, se tiene una cadena de  $k$  símbolos que representan el mensaje como en la figura 3.1. Y se desea añadir al mensaje una cadena de  $n - k$  bits de paridad (figura 3.2) para formar una palabra código ó *codeword* (figura 3.3)

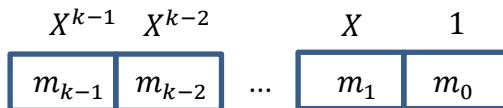


Figura. 3.1 Cada cuadro representa un símbolo dentro de una cadena de bits.

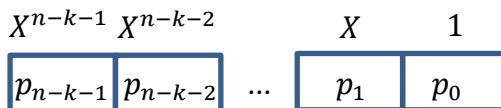


Figura. 3.2 Cadena de símbolos de paridad.

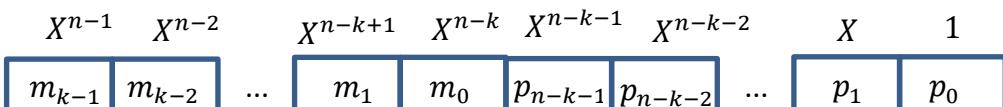


Figura. 3.3 Al mensaje se le añaden los bits de paridad, formando una palabra código (codeword).

Note que cada una de las posiciones dentro de las cadenas de símbolos representa uno de los literales del polinomio, vamos a representar lo que acabamos de hacer de una manera matemática para poder aclarar mejor este concepto.

Tenemos un polinomio que representa el mensaje y otro que representa los bits de paridad de la siguiente forma:

$$m(x) = \sum_{i=0}^{k-1} m_i X^i \quad p(x) = \sum_{i=0}^{n-k-1} p_i X^i$$

Ahora se desea formar una palabra código añadiendo símbolos de paridad al mensaje, note que si sumamos directamente estos polinomios se modificaría el mensaje ya que se sumarian los coeficientes de  $m(x)$  con los de  $p(x)$  y se perdería el mensaje, por ende se multiplica a  $m(x)$

por  $X^{n-k}$ , de este modo se logra unir el mensaje con los símbolos de paridad sin afectar al mismo. De esta forma un *codeword* queda definido como:

$$m(x) = m_0 + m_1X + \cdots + m_{k-2}X^{k-2} + m_{k-1}X^{k-1}$$

$$m(x) * X^{n-k} = m_0X^{n-k} + m_1X^{n-k+1} + \cdots + m_{k-2}X^{n-2} + X^{n-1}$$

$$p(x) = p_0 + p_1X + \cdots + p_{n-k-2}X^{n-k-2} + p_{n-k-1}X^{n-k-1}$$

$$U(x) = m(x) * X^{n-k} + p(x) \quad (3.2)$$

$$U(x) = p_0 + p_1X + \cdots + p_{n-k-2}X^{n-k-2} + p_{n-k-1}X^{n-k-1} + m_0X^{n-k} + m_1X^{n-k+1} + \cdots + m_{k-2}X^{n-2} + m_{k-1}X^{n-1} \quad (3.3)$$

La ecuación (3.2) es lo que deseamos a la salida de nuestro codificador Reed-Solomon, nuestro mensaje más la información de paridad que va a proteger nuestra información. La ecuación (3.3) muestra la expansión de la ecuación (3.2).

### 3.2. Calculo de $p(x)$

---

Para calcular  $p(x)$  dividimos a  $m(x)X^{n-k}$  por el polinomio generador  $g(x)$  lo cual puede ser escrito de la siguiente forma:

$$m(x)X^{n-k} = q(x)g(x) + p(x) \quad (3.4)$$

Donde  $q(x)$  y  $p(x)$  son el coeficiente y residuo respectivamente. Note que el residuo de esta división son nuestros símbolos de paridad. La ecuación 3.4 también puede ser expresada como:

$$p(x) = m(x)X^{n-k} \text{ modulo } g(x) \quad (3.5)$$

Es decir que lo que se pretende hacer con el codificador es obtener el residuo al dividir  $m(x)X^{n-k}$  por el polinomio generador  $g(x)$  y luego sumarlo a  $m(x)X^{n-k}$  para obtener a su salida la ecuación (3.2) como se había mencionado anteriormente. La división se puede realizar mediante un sistema de corrimiento de registros que se explicará a continuación.

### 3.3. Sistema de corrimiento de registros de $(n-k)$ etapas

---

Se usa un circuito para realizar la codificación R-S( $n,k$ ) de una secuencia de símbolos partiendo del polinomio generador descrito en la ecuación (3.1). El circuito que se requiere para lograr esta división es un LFSR (Linear Feedback Shift Register) como el que se muestra en la figura 3.4.

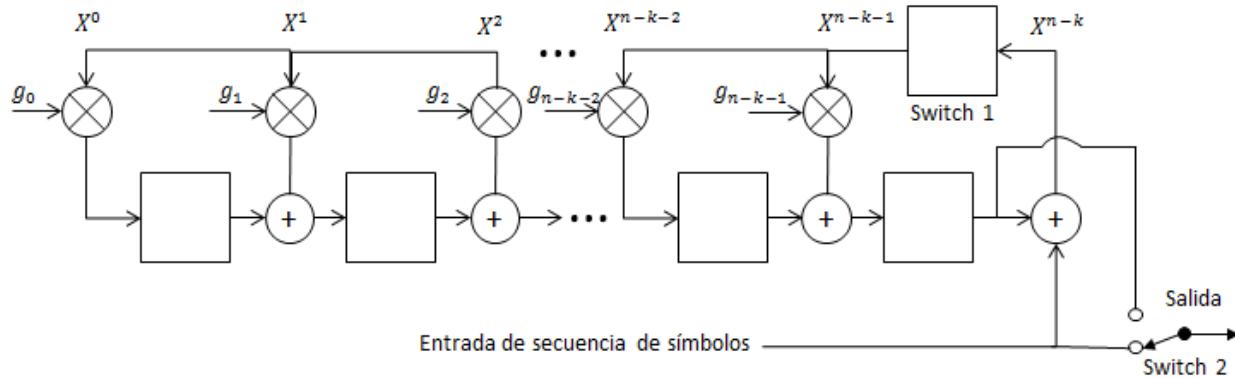


Figura. 3.4 LFSR utilizado para realizar la división

Cada uno de los registros (representados como un cuadrado), guarda los  $m$  bits que componen cada uno de los símbolos. El circuito mostrado en la figura 3.4 se encarga de añadir los bits de paridad, codificando de este modo el mensaje[19]. Su funcionamiento se describe a continuación.

1. El switch 1 permanece cerrado durante los primeros  $k$  ciclos de reloj, permitiendo alimentar los  $(n-k)$  registros con los símbolos que componen el mensaje.
2. El switch 2 permanece en la posición mostrada en la figura 4 durante los primeros  $k$  ciclos de reloj con el fin de permitir la transferencia directa de los símbolos del mensaje a la salida.
3. Después de que se ha transferido el  $k$ -ésimo símbolo a la salida, el switch 1 y 2 comutan.
4. El residuo, el cual está contenido en los  $(n-k)$  registros del circuito empieza a moverse a la salida.
5. El número total de ciclos que toma en realizarse la codificación es  $n$  y el contenido a la salida representa el polinomio del *codeword*  $m(x) * X^{n-k} + p(X)$ , donde  $p(X)$  representa los símbolos de paridad y  $m(x)$  los símbolos del mensaje a transmitir.

Hay que tener en cuenta que todas las operaciones de suma y multiplicación son diferentes para cada código, como se mostró en capítulos anteriores y que estas deben ser definidas para cada uno de los campos. Debido a esto realizar una prueba de escritorio de los algoritmos de división mediante LFSR resulta tedioso, pero se recomienda al estudiante hacerlo con un código pequeño, para que comprenda mejor el funcionamiento del mismo.

Las raíces del polinomio generador deben ser las mismas que las del *codeword*, ya que un *codeword* valido se puede expresar de la forma:

$$U(x) = m(x)g(x) \quad (3.6)$$

De modo que si se evalúan las raíces de  $g(x)$  en cualquier  $U(x)$  y  $U(x)$  es un *codeword* valido, estas deben hacer cero al polinomio  $U(x)$ . Es decir:

$$U(\alpha) = U(\alpha^2) = U(\alpha^3) = \dots = U(\alpha^{2t})=0$$

# 4

## Decodificador Reed-Solomon

El mensaje a transmitir ha sufrido una codificación RS( $n,k$ ) convirtiéndose en una de las palabras código de un campo finito. Esta palabra código (*codeword*) es transmitida a través de un canal, ya sea por el espacio, por cables de telefonía e internet, caminos de cobre dentro de una placa de circuito impreso o un dispositivo de almacenamiento como DVD, CD-ROM, un disco duro, etc.. El *codeword* transmitido se ve afectado por diferentes factores dentro del canal, como ruido.

Las señales que viajan a través de un canal, son corrompidas. Por ejemplo una señal puede verse afectada por ruido aditivo, experimentar algún retraso, un problema de jitter, sufrir de atenuación debido a la distancia de propagación y/o adquirir un nivel dc; además pueden producirse patrones de interferencia constructiva y destructiva al aparecer objetos en el camino de la señal o generarse interferencias debido a otros canales usados en el mismo medio. Todos estos fenómenos mencionados anteriormente pueden ocurrir todos al mismo tiempo. [2]

Debido a estos factores el *codeword* puede llegar a presentar errores en los símbolos que lo componen.

Para un *codeword* de  $n$  símbolos el patrón del error,  $e(x)$ , puede ser descrito con un polinomio de la siguiente forma.

$$e(x) = \sum_{i=0}^{n-1} e_i X^i$$

$$e(x) = e_0 X^0 + e_1 X^1 + e_2 X^2 + \cdots + e_{n-2} X^{n-2} + e_{n-1} X^{n-1} \quad (4.1)$$

Cabe aclarar que este polinomio solo puede tener  $t$  factores diferentes de cero, ya que como vimos anteriormente  $t$  es el límite de corrección del código RS. De lo contrario el error sobrepasará la capacidad de corrección del código. Denotamos a  $r(x)$  como el polinomio del *codeword* que hemos recibido y que ha sido corrompido por el canal, el cual puede ser representado como la suma del *codeword* original más el patrón de error de la siguiente forma:

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

Luego  $r(x)$  es:

$$r(x) = \sum_{i=0}^{n-1} u_i X^i + \sum_{i=0}^{n-1} e_i X^i$$

$$r(x) = \sum_{i=0}^{n-1} X^i (u_i + e_i)$$

$$r(x) = X^0(u_0 + e_0) + X^1(u_1 + e_1) + \cdots + X^{n-2}(u_{n-2} + e_{n-2}) + X^{n-1}(u_{n-1} + e_{n-1})$$

En  $r(x)$  existen  $2t$  incógnitas que deseamos averiguar,  $t$  localizaciones de los errores y  $t$  valores de los mismos. Note una diferencia importante entre la codificación no binaria y la binaria, en la codificación binaria basta con encontrar la localización del error, ya que al hacerlo, solo bastará con negar el bit erróneo para corregir el error, mientras que, en la codificación no binaria es necesario conocer el valor del error y su respectiva localización. Por ende al existir  $2t$  incógnitas, son necesarias  $2t$  ecuaciones para encontrarlas.

Se pretende entonces diseñar un decodificador Reed-Solomon para poder corregir estos errores. El decodificador Reed-Solomon está compuesto de 5 procesos como se muestran en la figura 4.1.

- 1) Síndrome.
- 2) Berlekamp-Massey.
- 3) Chien.
- 4) Forney.
- 5) Corrección.

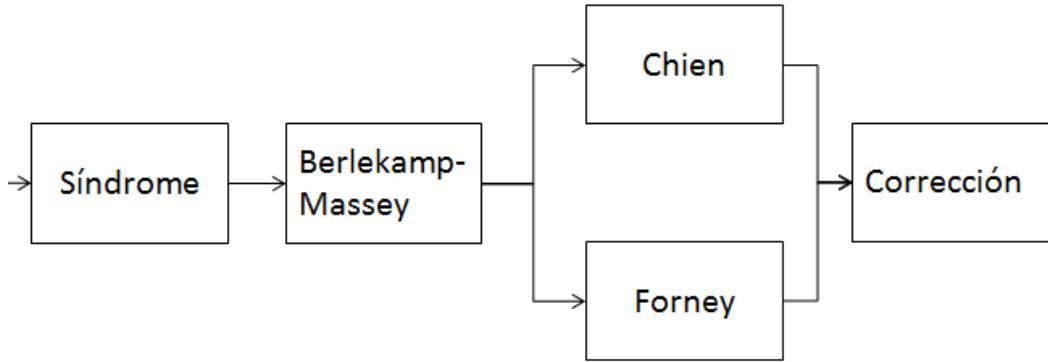


Figura. 4.1 Diagrama de bloques del decodificador.

Cada uno de estos procesos dura  $n$  pulsos de reloj. Es decir que un símbolo que ingrese al decodificador se tarda  $4 * n$  pulsos de reloj en ser procesado completamente. Note también que no es necesario generar una cola entre *codewords* en el proceso ya que la longitud de cada uno es de  $n$  y por ende mientras un codeword está siendo procesado en el Berlekamp-Massey el siguiente está siendo procesado en el síndrome por dar un ejemplo.

Lo primero que deseamos saber cuando llega un *codeword* es si tiene o no un error, para ello se evalúa el síndrome de este.

## 4.1. Síndrome

---

El síndrome es el resultado de una verificación en la paridad del *codeword*  $r$  que llega al receptor para determinar si este pertenece al conjunto de *codewords* del campo[17]. Si  $r$  es un *codeword* que pertenece al campo, el síndrome  $S$  será igual a cero, de lo contrario significa que  $r$  es un codeword que no pertenece al conjunto de codewords validos y por ende que este posee uno o más errores<sup>9</sup>. El síndrome  $S$  esta compuesto por  $n - k$  simblos,  $\{S_i\}$  ( $i = 0, 1, 2, \dots, n - k$ ). El cálculo del síndrome es bastante sencillo partiendo de la ecuación (6).

Se sabe que cada *codeword* es múltiplo del polinomio generador  $g(x)$  y por ende las raíces de  $g(x)$  son también las de  $U(x)$ . Luego  $r(x)$  puede escribirse como  $r(x) = U(x) + e(x)$ , note que si  $r(x)$  es evaluado en cada una de sus raíces y estas son igual a cero,  $r(x) = U(x)$  y por

---

<sup>9</sup> Similar al caso binario.

ende se tratará de un *codeword* valido. Cualquier error resultará en un cálculo del síndrome  $S$  diferente de cero. El cálculo del síndrome puede expresarse como:

$$S_i = r(x)|_{x=\alpha^i} = r(\alpha^i) \quad i = 1, 2, \dots, n - k \quad (4.2)$$

Note también que  $U(x)$  siempre va a ser cero cuando se evalúen las raíces de  $g(x)$ , por ende el evaluar la ecuación (4.2) es equivalente a evaluar cada una de las raíces en  $e(x)$  como se muestra en la ecuación (4.3).

$$S_i = e(x)|_{x=\alpha^i} = e(\alpha^i) \quad i = 1, 2, \dots, n - k \quad (4.3)$$

#### 4.1.1 Algoritmo e implementación

Como vimos en la explicación anterior, el cálculo del síndrome se resume en evaluar las raíces del polinomio generador  $g(x)$  en  $r(x)$ . Para ello se pueden implementar distintas alternativas. Por ejemplo, se puede guardar todo  $r(x)$  (lo que tardará  $n$  ciclos de reloj) y una vez que este guardado remplazar todas las raíces de  $g(x)$  en  $r(x)$  y así obtener  $S(x)$ , esto implica tener  $n$  registros para guardar  $r(x)$ ,  $n$  multiplicadores y  $n - 1$  sumadores para obtener un solo coeficiente de  $S$ , es decir que para obtener todo  $S(x)$  se gastarían  $i = 1, 2, \dots, n - k$  veces los recursos anteriormente descritos. A pesar de que es un procedimiento valido que obtiene el resultado deseado, existen alternativas que logran el mismo calculo en el mismo tiempo y tienen un mejor uso del hardware, el cual es importante para reducir costos y realizar la implementación sobre dispositivos de lógica reprogramable como una FPGA.

El algoritmo propuesto se desarrolla a partir del circuito mostrado en la figura (4.2), el cual calcula el valor de un coeficiente de  $S(x)$  en  $n$  ciclos de reloj. El circuito evalúa cada una de las raíces de  $g(x)$  en  $r(x)$ , a medida que  $r(x)$  va entrando al decodificador y las va sumando y almacenando en el registro de salida, de esta forma se obtiene el valor de un coeficiente de  $S(x)$  en  $n$  ciclos de reloj.

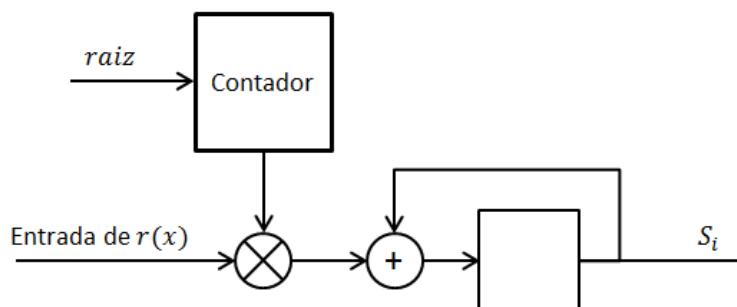


Figura. 4.2 Circuito propuesto para el calculo de un coeficiente de  $S(x)$ .

La entidad contador se encarga de arrojar la raíz evaluada en  $r(x)$  para que sea multiplicada por su coeficiente que acompaña al literal  $X$ . Para el primer símbolo que llega al decodificador, el cual se representa como  $r_{n-1}X^{n-1}$  (entra primero el símbolo más significativo), se evalúa la raíz en el literal  $X^{n-1}$  y se multiplica por su coeficiente que la acompaña  $r_{n-1}$ . Es decir que para el primer término de  $S$  y para el primer símbolo que llega al decodificador, contador debe entregar a su salida  $(\alpha)^{n-1}$ , para el segundo símbolo  $(\alpha)^{n-2}$  y así sucesivamente. Esto se ilustra mejor en la tabla 4.1.

Reloj	Entrada	Contador $S_1$	Contador $S_2$	...	Contador $S_{n-k-1}$	Contador $S_{n-k}$
1	$r_{n-1}X^{n-1}$	$(\alpha)^{n-1}$	$(\alpha^2)^{n-1}$	...	$(\alpha^{n-k-1})^{n-1}$	$(\alpha^{n-k})^{n-1}$
2	$r_{n-2}X^{n-2}$	$(\alpha)^{n-2}$	$(\alpha^2)^{n-2}$	...	$(\alpha^{n-k-1})^{n-2}$	$(\alpha^{n-k})^{n-2}$
:	:	:	:	:	:	:
$n-1$	$r_1X$	$(\alpha)^1$	$(\alpha^2)^1$	...	$(\alpha^{n-k-1})^1$	$(\alpha^{n-k})^1$
$n$	$r_0$	$(\alpha)^0$	$(\alpha)^0$	...	$(\alpha^{n-k-1})^0$	$(\alpha^{n-k})^0$

Tabla 4.1 Salida de la entidad contador para calcular los  $n-k$  coeficientes del síndrome.

Cuando se computa el primer término que llega al decodificador  $r_{n-1}(\alpha)^{n-1}$ , este es guardado en el registro, luego en el pulso dos se evalúa  $r_{n-1}(\alpha)^{n-2}$  y se suma con el valor guardado en el pulso anterior y se actualiza el valor del registro con  $r_{n-1}(\alpha)^{n-2} + r_{n-1}(\alpha)^{n-1}$ ; este proceso se realiza  $n$  veces simultáneamente para los  $i = 1, 2, \dots, n - k$  coeficientes del síndrome, como puede ilustrarse en la figura 4.3.

Note que de manera práctica en un código RS(7,3), para el cálculo del segundo coeficiente de  $S(x)$  el primer valor a la salida del Contador  $S_2$  será  $\alpha^{12}$  como lo describe la tabla 4.1., el cual es equivalente a  $\alpha^5$ . Se desarrolló un algoritmo sencillo que realiza el ajuste respectivo. Escribamos los exponentes de la salida del Contador  $S_2$  en forma binaria (Tabla 4.2.).

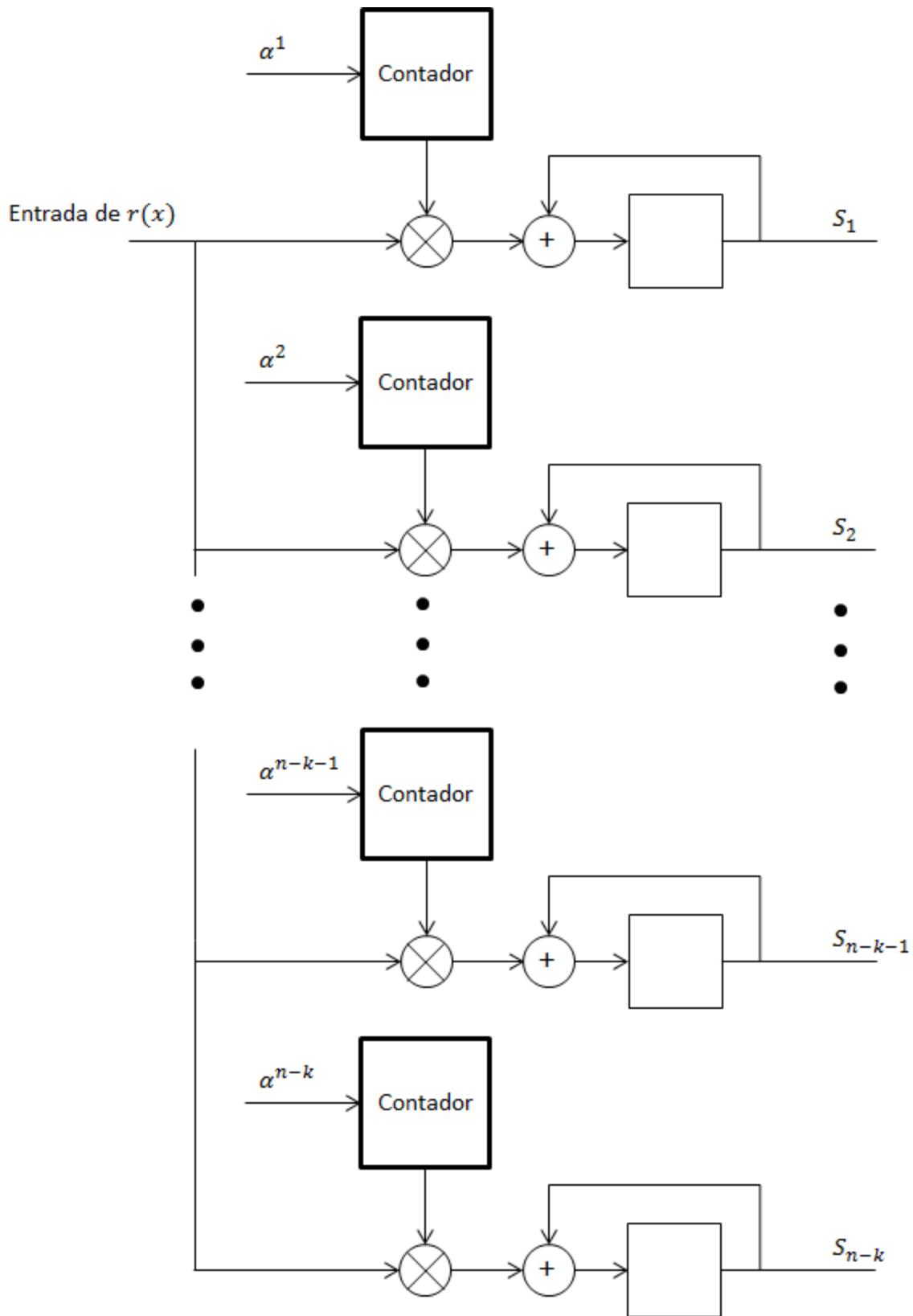


Figura. 4.3 Note que este circuito calcula el polinomio  $S(X)$  en  $n$  ciclos de reloj.

El exponente más alto que puede llegar a aparecer, se producirá al evaluar la última raíz de  $g(x)$  en el primer ciclo de reloj, es decir  $(\alpha^{n-k})^{n-1}$ , que para un código RS(7,3) representa un  $\alpha^{24}$ , por ende se usan 5 bits para representar cualquier exponente dentro de este rango. De forma general la cantidad de bits necesaria para representar el exponente más alto dentro de contador está dada por:

$$p = \lceil \log_2((n - k)(n - 1)) \rceil$$

Luego de representar el exponente con  $p$  bits, se separa este en B1 y B2 como muestra la tabla 4.2., y se suma B1 + B2 para obtener el exponente deseado a la salida de la entidad contador para un código RS(7,3). Se debe tener en cuenta el caso para el cual B2 es igual a todos sus bits en '1' donde se debe colocar una condición que convierta este valor a  $\alpha^0$ .

A1	A2	B	B1	B2	B2+B1
$\alpha^{12}$	$\alpha^5$	01100	01	100	101
$\alpha^{10}$	$\alpha^3$	01010	01	010	011
$\alpha^8$	$\alpha^1$	01000	01	000	001
$\alpha^6$	$\alpha^6$	00110	00	110	110
$\alpha^4$	$\alpha^4$	00100	00	100	100
$\alpha^2$	$\alpha^2$	00010	00	010	010
$\alpha^0$	$\alpha^0$	00001	00	001	001

*Tabla 4.3. A1 representa la salida del contador partiendo de la tabla 4.1. A2 es la salida real del contador, B la representación del exponente de A1 en binario con  $p$  bits, B1 representa los  $p-m$  bits más significativos de B y B2 los  $m$  bits menos significativos de B. Para un código RS(7,3).*

Este procedimiento se puede aplicar a cualquier código RS( $n,k$ ) si se desea implementar el algoritmo aquí propuesto. En la figura 4.4 se muestra un diagrama de flujo para el cálculo de un coeficiente de  $S(x)$ , que puede ayudar a aclarar todo lo explicado anteriormente.

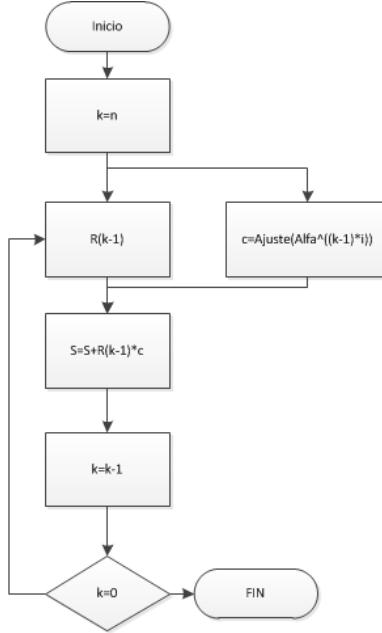


Figura. 4.4 Diagrama de flujo para el cálculo de un coeficiente de  $S(x)$ .

Dónde:

- $c$  representa la salida del contador.
- $R$  el polinomio  $r(x)$ .
- $S$  el coeficiente  $i$  de  $S(x)$ .

## 4.2. Localización del error

---

Suponga que hay  $v$  errores en el *codeword* en las posiciones  $X^{j_1}, X^{j_2}, \dots, X^{j_v}$ . De esta forma podemos escribir el polinomio del error  $e(x)$  mostrado en la ecuación (4.1) como:

$$e(x) = \sum_{i=1}^v e_{j_i} X^{j_i}$$

$$e(x) = e_{j_1} X^{j_1} + e_{j_2} X^{j_2} + \dots + e_{j_{(v-1)}} X^{j_{(v-1)}} + e_{j_v} X^{j_v} \quad (4.4)$$

Los índices  $1, 2, \dots, v - 1, v$  se refieren al primer, segundo,...  $v$ -ésimo error y los índices  $j$ , se refieren a la localización del error. Para lograr corregir el *codeword* cada localización  $X^{j_i}$  y cada valor del error  $e_{j_i}$ , donde  $i = 1, 2, \dots, v$  deben ser determinados. Definimos un número para la localización del error denotado como  $\beta_l = \alpha^{j_l}$  y luego se obtienen los  $n - k = 2t$  símbolos del síndrome por la sustitución de  $\alpha^i$  dentro del polinomio que ha sido recibido para  $i = 1, 2, \dots, 2t$ .

$$\begin{aligned}
S_1 &= r(\alpha) = e_{j_1}\beta_1 + e_{j_2}\beta_2 + \cdots + e_{j_v}\beta_v \\
S_2 &= r(\alpha^2) = e_{j_1}\beta_1^2 + e_{j_2}\beta_2^2 + \cdots + e_{j_v}\beta_v^2 \\
&\vdots \\
&\vdots \\
S_i &= r(\alpha^{2t}) = e_{j_1}\beta_1^{2t} + e_{j_2}\beta_2^{2t} + \cdots + e_{j_v}\beta_v^{2t} \quad (4.5)
\end{aligned}$$

Hay  $2t$  valores desconocidos ( $t$  valores de error y  $t$  localizaciones del error), y  $2t$  ecuaciones simultaneas. Sin embargo estas  $2t$  ecuaciones simultaneas no pueden ser resultas de manera convencional ya que no son lineales (debido a que algunos de los valores desconocidos tienen exponentes mayores a uno). Cualquier técnica que resuelva este sistema de ecuaciones es conocida como un *algoritmo de decodificación Reed-Solomon*.

Una vez un vector de síndromes no nulo (uno o más símbolos son diferentes de cero) ha sido calculado, lo siguiente es descubrir donde está el error, para ello se define un polinomio localizador del error  $\sigma(x)$  como:

$$\sigma(x) = (1 + \beta_1 X)(1 + \beta_2 X) \cdots (1 + \beta_v X) \quad (4.6)$$

Las raíces de  $\sigma(x)$  son  $\frac{1}{\beta_1}, \frac{1}{\beta_2}, \dots, \frac{1}{\beta_v}$ . El reciproco de las raíces de  $\sigma(x)$  son los simbolos que representan la localización del error en  $e(x)$ . Luego, usando técnicas de modelamiento auto-regresivo, formamos una matriz a partir de los síndromes, donde los primeros  $t$  síndromes son usados para predecir el siguiente síndrome. Esto lo podemos observar a continuación:

$$\begin{bmatrix} S_1 & S_2 & \cdots & S_{t-1} & S_t \\ S_2 & S_3 & \ddots & S_1 & S_1 \\ \vdots & & \ddots & \vdots & \vdots \\ S_{t-1} & S_t & \cdots & S_{2t-3} & S_{2t-2} \\ S_t & S_{t+1} & \cdots & S_{2t-2} & 2t-1 \end{bmatrix} \begin{bmatrix} \sigma_t \\ \sigma_{t-1} \\ \vdots \\ \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_{t+1} \\ -S_{t+2} \\ \vdots \\ -S_{2t-1} \\ -S_{2t} \end{bmatrix} \quad (4.7)$$

Se aplica el modelo auto-regresivo de la ecuación (4.7), note que el tamaño de esta matriz es de  $t \times t$ . El procedimiento matemático para hallar la localización del error a partir de la ecuación (4.7) es el mismo que se usa para resolver cualquier sistema de ecuaciones. Es decir que la solución a la ecuación (4.7) la podemos representar como:

$$\begin{bmatrix} \sigma_t \\ \sigma_{t-1} \\ \vdots \\ \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_{t+1} \\ -S_{t+2} \\ \vdots \\ -S_{2t-1} \\ -S_{2t} \end{bmatrix} \left[ \begin{array}{ccccc} S_1 & S_2 & \dots & S_{t-1} & S_t \\ S_2 & S_3 & & S_1 & S_1 \\ \vdots & & \ddots & \vdots & \\ S_{t-1} & S_t & \dots & S_{2t-3} & S_{2t-2} \\ S_t & S_{t+1} & \dots & S_{2t-2} & 2t-1 \end{array} \right]^{-1}$$

Encontrados los coeficientes de  $\sigma(x)$  podemos fácilmente obtener el polinomio localizador del error.

#### 4.2.1 Algoritmo de Berlekamp-Massey

Para poder resolver la ecuación (4.7) y obtener el polinomio localizador del error,  $\sigma(x)$ , existen diferentes algoritmos dentro de los cuales se destacan el “*Algoritmo de Euclides*” y el “*El algoritmo de Berlekamp-Massey*” y se han hecho algunas mejoras a estos como el RIBM[10]. El algoritmo de Berlekamp-Massey se describe a continuación y fue tomado de [3].

##### **Algoritmo de Berlekamp-Massey**

Entrada:  $S_1, S_2, \dots, S_N$

Inicialización:

$L = 0$  (Longitud actual del LFSR)

$c(x) = 1$  (conexión actual del polinomio)

$p(x) = 1$  (conexión del polinomio antes del último cambio)

$l=1$  ( $l$  es  $k - m$ , la cantidad de corrimiento)

$d_m = 1$  (Antes de la discrepancia).

*for*  $k = 1$  to  $N$

$d = S_k + \sum_{i=1}^L c_i S_{k-i}$  (calcula la discrepancia).

*if* ( $d = 0$ ) (el polinomio no cambia).

$l = l + 1$

*else*

*if* ( $2L \geq k$ ) (Ninguna longitud cambia en la actualización)

$c(x) = c(x) - dd_m^{-1}x^l p(x)$

$l = l + 1$

*else* (actualiza  $c$  con el cambio de longitud)

$t(x) = c(x)$  (guardamos temporalmente  $c$  en  $t$ .)

$c(x) = c(x) - dd_m^{-1}x^l p(x)$

$L = k - L$

$p(x) = t(x)$

$d_m = d$

$l = 1$

*end*  
*end*  
*end*

Este algoritmo puede representarse de una manera más clara como muestra el diagrama de flujo de la figura 4.5.

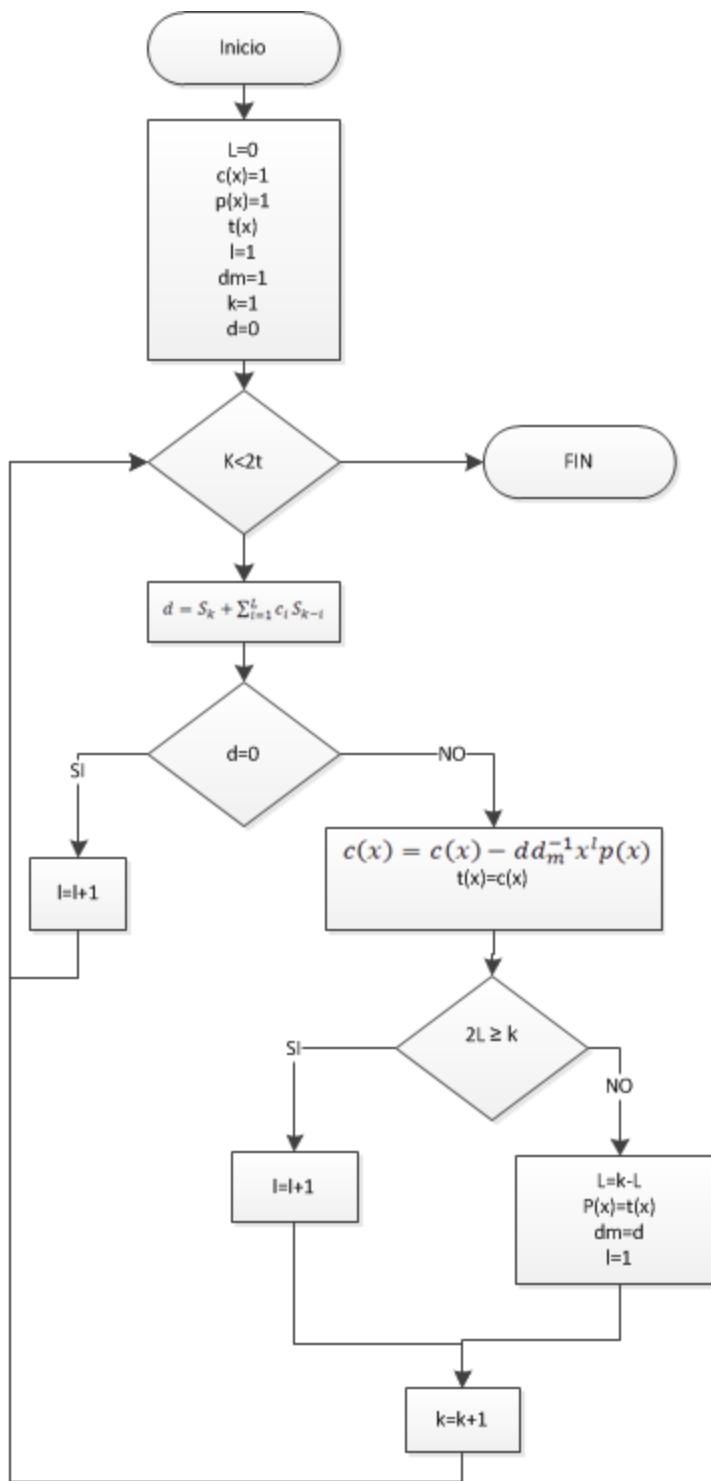


Figura. 4.5 Diagrama de flujo del algoritmo Berlekamp-Massey propuesto en [2]. Note que se hizo una pequeña modificación para calcular  $c(x) = c(x) - dd_m^{-1}x^l p(x)$  solo una vez.

Como la implementación que estamos haciendo es hecha sobre hardware, se traduce este diagrama de flujo a un circuito que pueda ser implementado en una FPGA.

## 4.2.2 Implementación

El cálculo de  $d$  para cualquier código RS se puede implementar mediante el circuito propuesto en la figura 6.

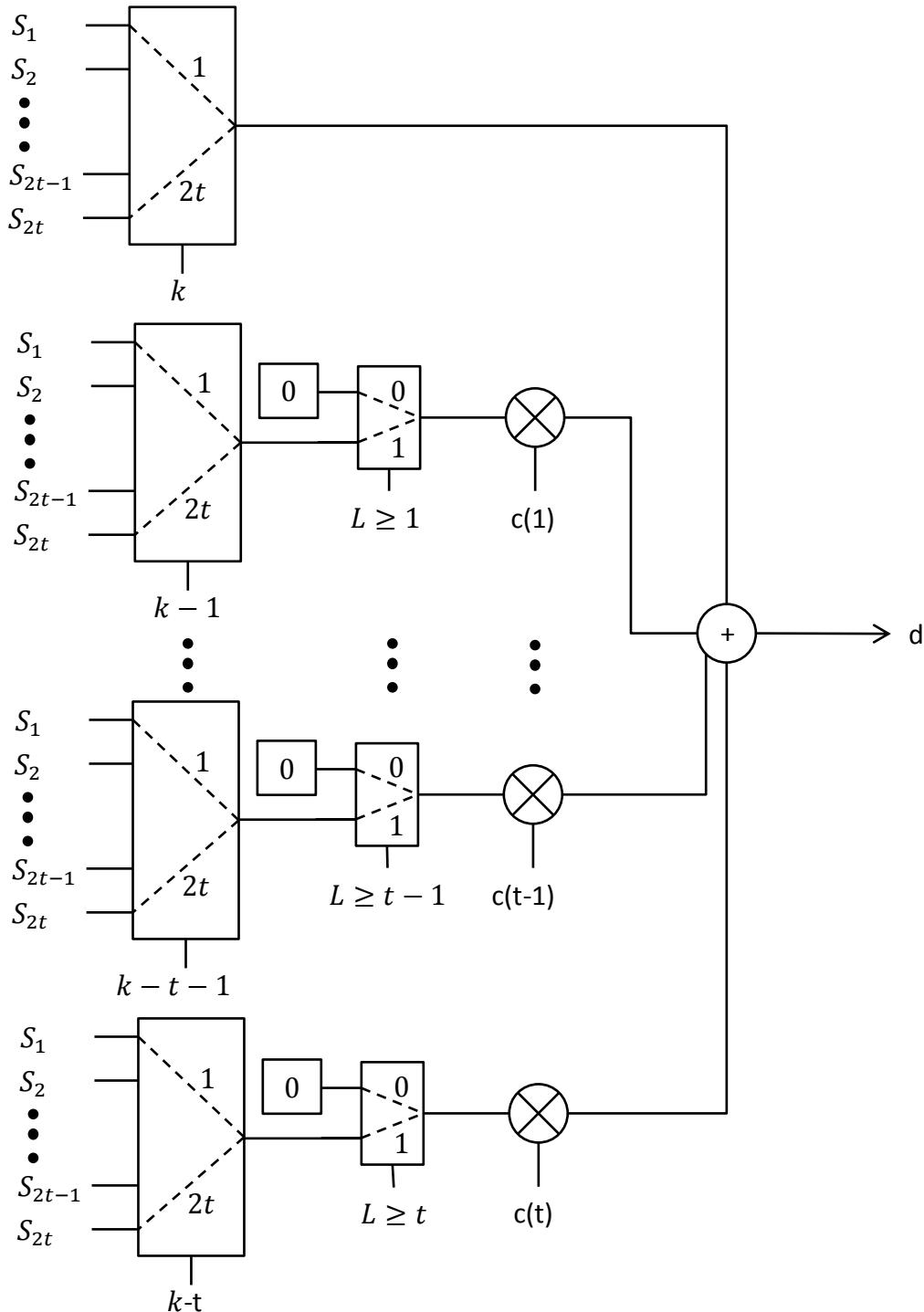
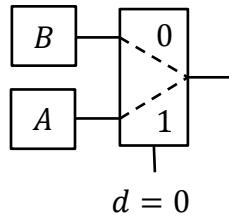


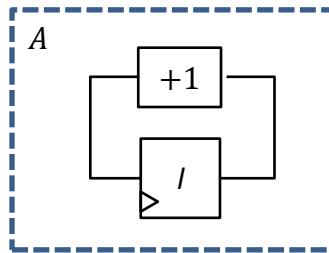
Figura. 4.6 Circuito propuesto para calcular  $d$ .

Luego de que  $d$  se calcula, se llega a una bifurcación donde se pregunta por la condición ( $d=0$ ), y dependiendo de esta, se cambia o no el polinomio  $c(x)$ . Esta bifurcación nos lleva a realizar dos procesos, uno que llamaremos A para cuando es afirmativa y otro que llamaremos B cuando esta condición no se cumpla. Circuitalmente se representa como:



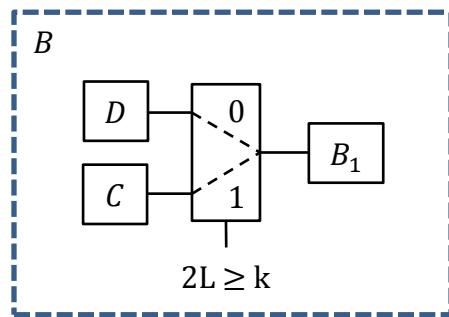
*Figura. 4.7 Primera bifurcación del algoritmo.*

El proceso A simplemente consiste en incrementar el valor de  $l$  en uno y puede ser descrito como:



*Figura.4.8 Se realiza el proceso A cuando la condición ( $d=0$ ) es positiva.*

El proceso B es más complejo ya que posee una bifurcación dentro de este, la cual depende de la condición ( $2L \geq k$ ). Esta bifurcación gestiona dos procesos, uno que llamaremos C cuando la condición se cumple y otro que llamaremos D cuando no lo hace. Es decir que el proceso B podemos representarlo como se muestra en la figura 4.9.



*Figura. 4.9 Proceso B, note que el proceso B1 se realiza independiente del resultado de la bifurcación.*

Como se puede evidenciar en el diagrama de flujo de la figura 4.5. El proceso que no depende de la bifurcación es la actualización de  $c(x)$  y al cual se a nombrado como proceso  $B_1$ . Los procesos  $D$ ,  $C$  y  $B_1$  se describen a continuación.

El proceso C es idéntico al proceso A y el circuito resultante es idéntico al mostrado en la figura 4.8. El proceso de D es mostrado en la figura 4.10.

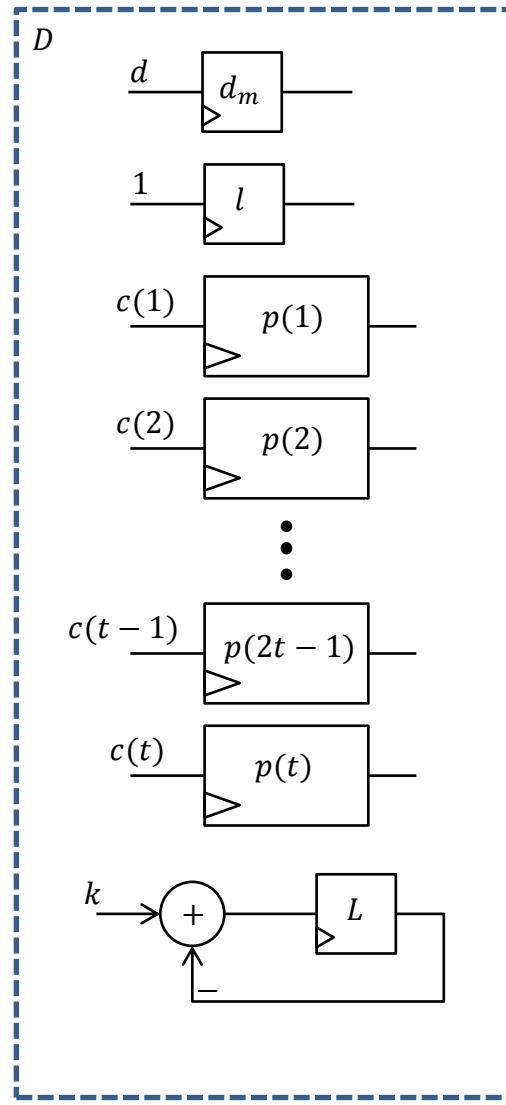


Figura. 4.10 Circuito encargado de ejecutar el proceso D.

La actualización de  $c(x)$  se describe mediante  $c(x) = c(x) - dd_m^{-1}x^l p(x)$ , este calculo se divide en dos partes, primeros se calcula el termino  $h(x) = dd_m^{-1}p(x)$  como se muestra en la figura 4.11. Note que la entidad INV obtiene el inverso de  $d_m$  que luego es multiplicado por  $d$  y el polinomio  $p(x)$ , el resultado de este calculo es un polinomio del mismo grado de  $p(x)$  y al que llamaremos  $h(x)$ .

La segunda parte del proceso consiste en sumar  $c(x) + x^l h(x)$ , note que si el termino  $x^l$  fuera igual a uno,  $c(x)$  y  $h(x)$  se sumarian de la manera habitual (como una suma entre vectores), sin embargo, el termino  $x^l$  realiza un corrimiento a la izquierda que depende de  $l$ .

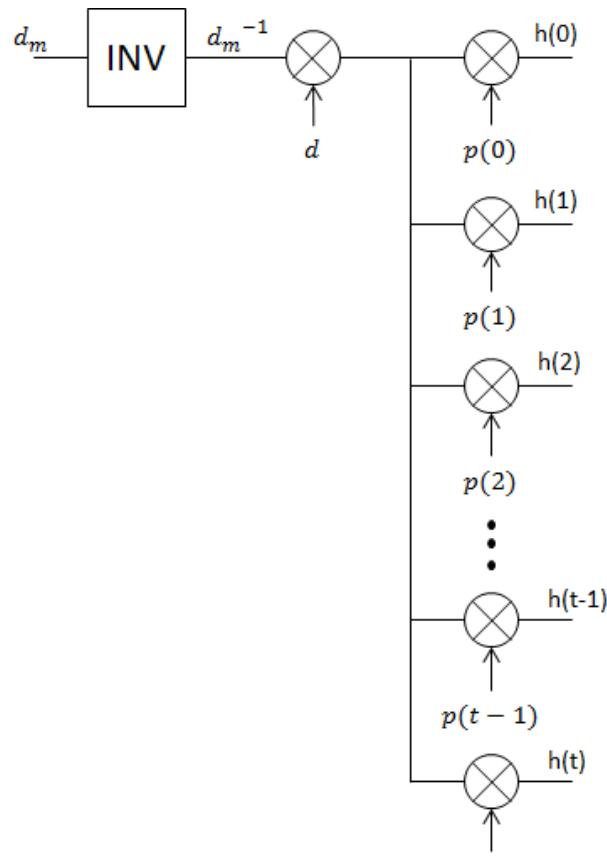


Figura. 4.11 Circuito propuesto para el cálculo de  $dd_m^{-1}p(x)$ .

La función de los multiplexores del circuito de la figura 4.12 es el de realizar este corrimiento de los términos de  $h(x)$  dependiendo del valor de  $l$ . Para evidenciar esto mejor, veamos como el término  $c(x) = c(x) + x^l h(x)$  va variando en cómputo en la tabla 4.4.

$l$	$c_t$	$c_{t-1}$	...	$c_2$	$c_1$
1	$c_t + h_{t-1}$	$c_{t-1} + h_{t-2}$	...	$c_2 + h_1$	$c_1 + h_0$
2	$c_t + h_{t-2}$	$c_{t-1} + h_{t-3}$	...	$c_2 + h_0$	$c_1$
:	:	:	⋮	⋮	⋮
$t - 1$	$c_t + h_1$	$c_{t-1} + h_0$	...	$c_2$	$c_1$
$t$	$c_t + h_0$	$c_{t-1}$	...	$c_2$	$c_1$

Tabla 4.4 variación de  $c(x) = c(x) + x^l h(x)$  con respecto a  $l$ .

Note como dependiendo del valor de  $l$  los coeficientes de  $h(x)$  que se suman con los de  $c(x)$  van variando. Finalmente en este circuito se termina de calcular  $c(x)$  y el proceso  $B_1$  concluye. Todos los procesos descritos anteriormente se repiten  $k$  veces ( $k = 1, 2, \dots, 2t$ ) hasta que finalmente se obtiene el LFSR capaz de generar todos los síndromes y con esto el polinomio localizador del error,  $\sigma(x)$ .

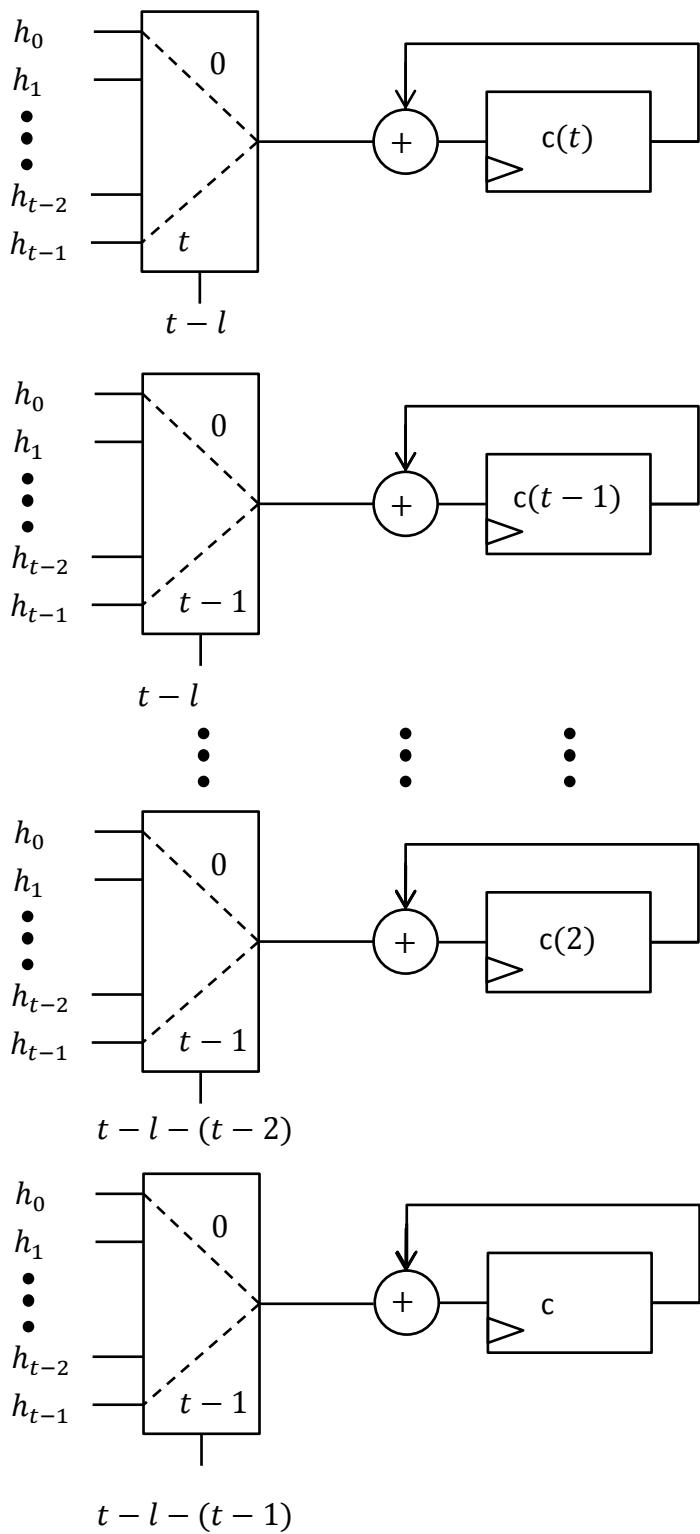


Figura. 4.12 Circuito propuesto para el cálculo de  $c(x) + x^l h(x)$ ,

## 4.3. Calculo de las raíces del polinomio localizador del error

---

Al obtener el polinomio localizador del error, basta con encontrar las raíces de este que nos indican la posición del error dentro del *codeword* que fue transmitido. Si recordamos la ecuación (4.6) (que mostramos por comodidad a continuación), recordamos que el inverso de las raíces de  $\sigma(x)$  ( $\beta_1, \beta_2, \dots, \beta_{v-1}, \beta_v$ ) representan los valores donde el error está localizado.

$$\sigma(x) = (1 + \beta_1 X)(1 + \beta_2 X) \dots (1 + \beta_v X)$$

Cualquier elemento del campo puede ser una raíz de  $\sigma(x)$ , por ende podemos determinar las raíces de  $\sigma(x)$  evaluando cada uno de los elementos del campo en  $\sigma(x)$  como se muestra a continuación.

$$\sigma(\alpha) = c_v(\alpha)^v + c_{v-1}(\alpha)^{v-1} + \dots + c_1(\alpha) + c_0$$

$$\sigma(\alpha^2) = c_v(\alpha^2)^v + c_{v-1}(\alpha^2)^{v-1} + \dots + c_1(\alpha^2) + c_0$$

⋮

$$\sigma(\alpha^{n-2}) = c_v(\alpha^{n-2})^v + c_{v-1}(\alpha^{n-2})^{v-1} + \dots + c_1(\alpha^{n-2}) + c_0$$

$$\sigma(\alpha^{n-1}) = c_v(\alpha^{n-1})^v + c_{v-1}(\alpha^{n-1})^{v-1} + \dots + c_1(\alpha^{n-1}) + c_0$$

A los elementos del campo que hacen que  $\sigma(x)$  se haga cero, se les debe calcular su inverso para finalmente poder identificar la posición del error dentro del *codeword* que fue recibido. Note que esta parte del proceso de decodificación es bastante sencilla y para este se pueden utilizar varios métodos de cómputo como se verá a continuación.

### 4.3.1. Algoritmo propuesto

Note que este problema se resume en evaluar todos los elementos del campo en  $\sigma(x)$  e identificar cuando este se hace cero. Este problema se puede resolver en un solo ciclo de reloj, ya que no se requiere de utilizar algún resultado que este almacenado en memoria, sin embargo, al hacer esto el costo en hardware sería muy alto. Note que si se desea evaluar un solo elemento del campo en un solo ciclo de reloj, este consumirá  $t$  multiplicaciones y  $t + 1$  sumas, es decir, que para evaluar todos los elementos del campo se necesitarían  $t * n$  multiplicaciones y  $(t + 1) * n$  sumas, lo cual es un gasto de hardware bastante alto. Cabe

aclarar, que si lo que se desea es un procesamiento rápido en el decodificador y es posible implementar toda esta cantidad de operaciones, este algoritmo puede considerarse como una alternativa.

Sin embargo al usar códigos con un amplio número de símbolos en dispositivos de lógica reprogramable como las FPGA, es probable que no se pueda implementar toda la cantidad de hardware descrita anteriormente.

Se propone un algoritmo que permita realizar la búsqueda con la menor cantidad de hardware posible, sacrificando velocidad de procesamiento en el decodificador. Para ello se utiliza el hardware necesario para evaluar un solo elemento del campo en  $\sigma(x)$  y se itera para los  $n$  elementos del campo, realizando todo el cálculo en  $n$  ciclos de reloj. Podemos utilizar un procedimiento lógico similar al circuito de la figura 4.3 que se usó para el cálculo del síndrome con algunas variaciones para guardar las raíces de  $\sigma(x)$  en el instante en el que un elemento del campo haga  $\sigma(x)$  cero. En la figura 4.13 se muestra el circuito propuesto, note que al igual que en el síndrome se tiene un bloque contador necesario para poder realizar las multiplicaciones de los coeficientes del polinomio localizador con el elemento del campo el cual debe ser elevado dependiendo del literal  $X^i$  para  $i = 1, 2, \dots, v$  que este evaluando. Para ello las salidas de contador dependen de una entrada que indica la potencia a la que deben ser elevados todos los elementos del campo que van ser multiplicados por ese coeficiente. Esto se puede evidenciar de una manera más clara en la tabla 4.

Contador $c_1$	Contador $c_2$	Contador $c_3$	...	Contador $c_{v-1}$	Contador $c_v$
$\alpha^0$	$\alpha^0$	$\alpha^0$	...	$\alpha^0$	$\alpha^0$
$\alpha^1$	$\alpha^2$	$\alpha^3$	...	$\alpha^{v-1}$	$\alpha^v$
$\alpha^2$	$\alpha^4$	$\alpha^6$	...	$\alpha^{2(v-1)}$	$\alpha^{2v}$
:	:	:	:	:	:
$\alpha^{n-2}$	$\alpha^{2(n-2)}$	$\alpha^{3(n-2)}$	...	$\alpha^{(v-1)(n-2)}$	$\alpha^{v(n-2)}$
$\alpha^{n-1}$	$\alpha^{2(n-1)}$	$\alpha^{3(n-1)}$	...	$\alpha^{(v-1)(n-1)}$	$\alpha^{v(n-1)}$

Tabla 2.5 Salidas de los contadores que son multiplicadas por los coeficientes de  $\sigma(x)$  representados como  $c_i$  para  $i = 1, 2, \dots, v$ .

Luego de que la multiplicación de cada uno de los coeficientes se calcula, se suman todas estas para calcular el valor de  $\sigma(\alpha^i)$  y verificar si esta es o no una raíz del  $\sigma(x)$ . Cuando se encuentra una raíz de  $\sigma(x)$  se guarda en el primer registro  $1/\beta_2$  y se incrementa  $w$  en uno, con ello se pretende guardar la siguiente raíz en el registro  $1/\beta_2$  y con ello almacenar cada una de las raíces que se encuentren, de este modo pueden ser utilizadas en la siguiente etapa del decodificador RS.

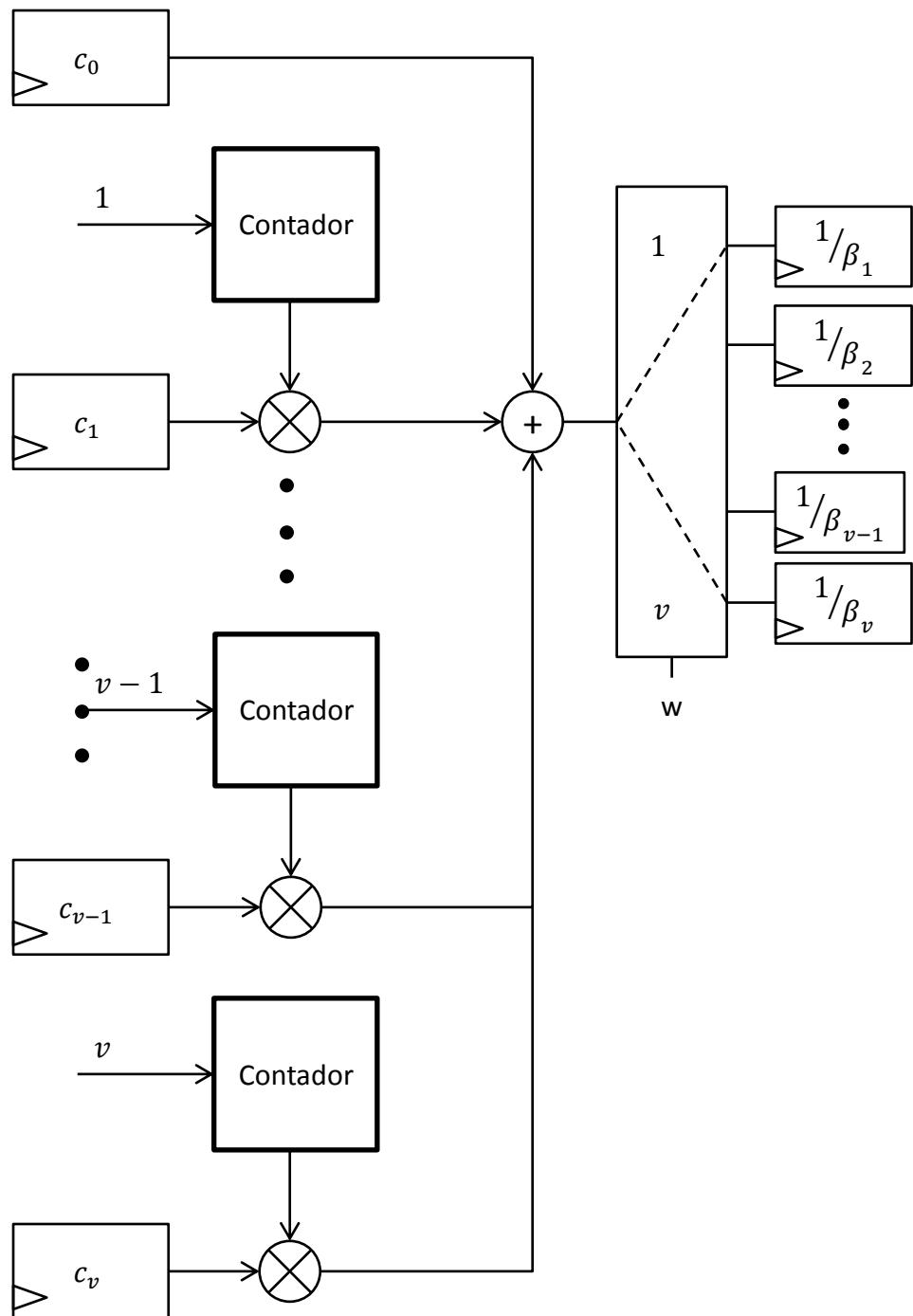


Figura. 4.13 Circuito propuesto para el cálculo de las raíces de  $\sigma(x)$ .

### 4.3.2. Algoritmo de Chien

Uno de los algoritmos que más se encuentra en la literatura para evaluar polinomios, es el algoritmo de Chien o búsqueda de Chien. Este algoritmo se destaca por ser uno de los algoritmos más eficientes en el uso del hardware a la hora de implementar un algoritmo para la búsqueda de las raíces de un polinomio en un campo finito. Retomemos el polinomio localizador expresado como:

$$\sigma(X) = \sigma_0 + \sigma_1X + \cdots + \sigma_{v-1}X^{v-1} + \sigma_vX^v = 1 + \sigma_1X + \cdots + \sigma_{v-1}X^{v-1} + \sigma_vX^v$$

Se evalúa el polinomio con la sucesión  $X = 1, X = \alpha, X = \alpha^2, \dots, X = \alpha^{n-1}$  lo cual produce:

$$\sigma(1) = 1 + \sigma_1(1) + \sigma_2(1)^2 + \cdots + \sigma_{v-1}(1)^{v-1} + \sigma_v(1)^v$$

$$\sigma(\alpha) = 1 + \sigma_1(\alpha) + \sigma_2(\alpha)^2 + \cdots + \sigma_{v-1}(\alpha)^{v-1} + \sigma_v(\alpha)^v$$

$$\sigma(\alpha^2) = 1 + \sigma_1(\alpha^2) + \sigma_2(\alpha^2)^2 + \cdots + \sigma_{v-1}(\alpha^2)^{v-1} + \sigma_v(\alpha^2)^v$$

⋮

$$\sigma(\alpha^{n-1}) = 1 + \sigma_1(\alpha^{n-1}) + \sigma_2(\alpha^{n-1})^2 + \cdots + \sigma_{v-1}(\alpha^{n-1})^{v-1} + \sigma_v(\alpha^{n-1})^v$$

El cálculo de esta secuencia puede ser representada de forma eficiente como se muestra en la figura 4.14.

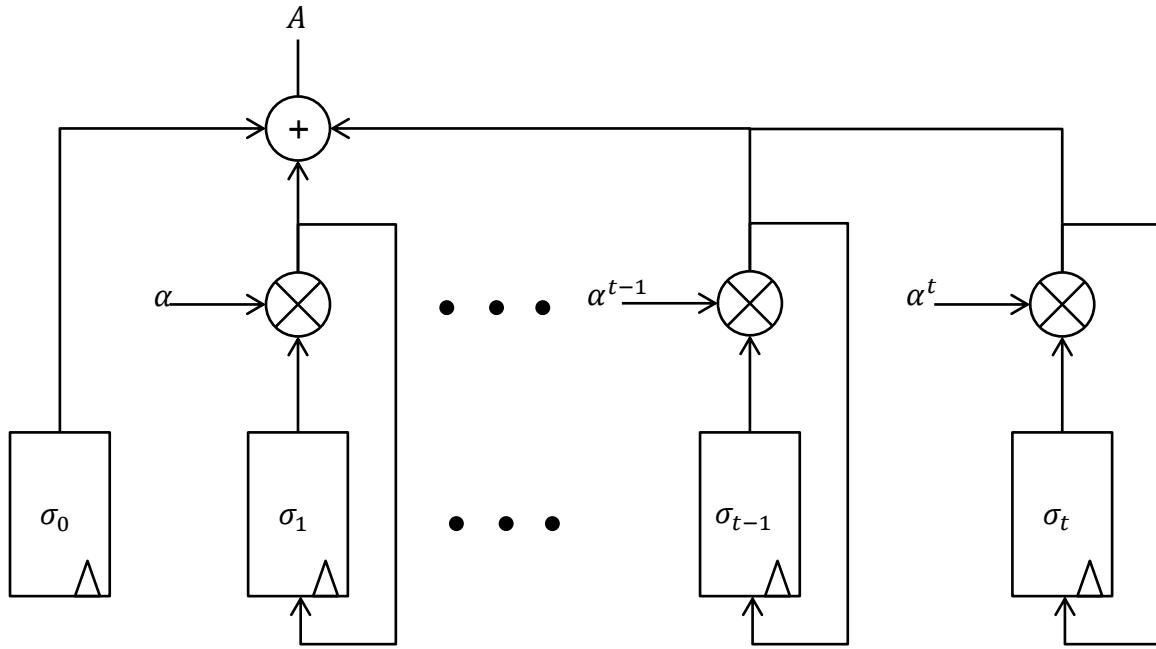


Figura. 4.14 Representación circuito de la búsqueda de Chien

Los registros  $\sigma_0, \sigma_1, \dots, \sigma_{v-1}, \sigma_v$  se inicializan con los coeficientes del polinomio localizador y luego se multiplican por la primera raíz evaluada en cada uno de los coeficientes ( $\sigma_0 * \alpha^0, \sigma_1 * \alpha^1, \dots, \sigma_{v-1} * \alpha^{v-1}, \sigma_v * \alpha^v$ ) y se actualiza cada uno de los registros para repetir  $n$  veces este procedimiento y evaluar cada uno de los elementos del campo. Note que a diferencia del algoritmo anterior el valor por el que se multiplica cada uno de los coeficientes es un valor constante[12], reduciendo aún más el costo en hardware con comparación con el anterior algoritmo.

#### 4.4. Cálculo del valor de los errores

---

Un error ha sido definido como  $e_{j_l}$  donde el índice  $j$  hace referencia a la localización del error y el índice  $l$  identifica el  $l$ -ésimo error. Ya que a cada error le corresponde una posición en particular, la notación puede ser simplificada denominando al error simplemente como  $e_l$ . Para determinar los valores de los errores  $e_1, e_2, \dots, e_{v-1}, e_v$  asociados con las localizaciones  $\beta_1, \beta_2, \dots, \beta_{v-1}, \beta_v$  cualquiera de las  $2t$  ecuaciones del síndrome (ecuación 11) pueden ser usadas. Debido a que tenemos  $t$  incógnitas y  $2t$  ecuaciones solo se requiere utilizar  $t$  ecuaciones de las  $2t$  del síndrome (por notación utilizaremos las  $t$  primeras ecuaciones disponibles), como mostramos en la ecuación 4.8.

$$S_1 = r(\alpha) = e_{j_1}\beta_1 + e_{j_2}\beta_2 + \dots + e_{j_v}\beta_v$$

$$S_2 = r(\alpha^2) = e_{j_1} \beta_1^2 + e_{j_2} \beta_2^2 + \cdots + e_{j_v} \beta_v^2$$

•  
•  
•

$$S_t = r(\alpha^t) = e_{j_1} \beta_1^t + e_{j_2} \beta_2^t + \cdots + e_{j_v} \beta_v^t \quad (4.8)$$

Podemos escribir las ecuaciones (4.8) en forma matricial:

$$\begin{bmatrix} \beta_1 & \beta_2 & \dots & \beta_{v-1} & \beta_t \\ \beta_1^2 & \beta_2^2 & \dots & \beta_{v-1}^2 & \beta_v^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \beta_1^{t-1} & \beta_2^{t-1} & \dots & \beta_{v-1}^{t-1} & \beta_v^{t-1} \\ \beta_1^t & \beta_2^t & \dots & \beta_{v-1}^t & \beta_v^t \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{t-1} \\ e_t \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_{t-1} \\ S_t \end{bmatrix} \quad (4.9)$$

Para resolver el sistema de ecuaciones de la ecuación (4.9) podemos hacerlo de la manera usual, encontrando la inversa de la primera matriz y multiplicándola por la matriz de síndromes de la derecha para encontrar el valor de cada uno de los errores como se muestra en la ecuación (4.10).

$$\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{t-1} \\ e_t \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_{t-1} \\ S_t \end{bmatrix} \begin{bmatrix} \beta_1 & \beta_2 & \dots & \beta_{v-1} & \beta_t \\ \beta_1^2 & \beta_2^2 & \dots & \beta_{v-1}^2 & \beta_v^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \beta_1^{t-1} & \beta_2^{t-1} & \dots & \beta_{v-1}^{t-1} & \beta_v^{t-1} \\ \beta_1^t & \beta_2^t & \dots & \beta_{v-1}^t & \beta_v^t \end{bmatrix}^{-1} \quad (4.10)$$

Un algoritmo que resuelve este sistema de ecuaciones es el algoritmo de Forney que se explicará a continuación.

#### 4.4.1. Algoritmo de Forney

Note que la matriz de localizaciones de los errores en la ecuación 15 es una matriz de Vandermonde[7]. Existen algoritmos rápidos para resolver sistemas que incluyan matrices de Vandermonde. Uno de estos algoritmos que se aplica en la mayoría de códigos BCH o RS es el algoritmo de Forney.

Antes de presentar la fórmula es necesario recordar ciertas definiciones. El polinomio del síndrome es definido por:

$$s(X) = S_1 + S_2X + S_3X^2 + \cdots + S_{2t}X^{2t-1} = \sum_{j=0}^{2t-1} S_{j+1} + X^j$$

Similar al caso de la localización del error, un polinomio del valor del error  $\Omega(X)$  debe ser definido, se define entonces  $\Omega(X)$  como:

$$\Omega(X) = S(X)\sigma(X)(\text{mod } X^{2t}) \quad (4.11)$$

A esta ecuación se le conoce como *ecuación clave* [11]. Note que el efecto de computar con  $\text{mod } X^{2t}$  esta ecuación, se hace con el fin de descartar todos los términos con grado  $2t$  o mayor.

Necesitamos definir la derivada en un campo finito. Sea  $f(X) = f_0 + f_1X + f_2X^2 + \cdots + f_tX^t$  con coeficientes en algún campo  $F$ . La derivada formal  $f'(X)$  de  $f(X)$  es calculada usando las reglas convencionales de la diferenciación de polinomios:

$$f'(X) = f_1 + 2f_2X + 3f_3X^2 + \cdots + tf_tX^{t-1} \quad (4.12)$$

Donde, como es usual,  $mf_i$  para  $m \in \mathbb{Z}$  y  $f_i \in F$  se describe como una suma repetitiva:

$$mf_i = \underbrace{f_i + f_i + f_i + \cdots + f_i + f_i}_{m \text{ sumas}}$$

No hay ningún tipo de implicación para la diferenciación formal, esto simplemente corresponde a la manipulación formal de los símbolos. Basados en la definición anterior se puede demostrar que muchas de las reglas convencionales de la diferenciación aplican para los campos finitos. Teniendo en cuenta lo mostrado anteriormente note que si  $f(X) \in F[X]$ , donde  $F$  es un campo de característica 2, entonces  $f'(X)$  no tiene potencias impares, ya que el coeficiente que las acompaña seria par y el producto de cualquier término por un numero par es nulo para un campo de característica 2. Podemos aclarar un poco este concepto con los siguientes ejemplos:

$$\begin{aligned} 2\alpha^i &= \alpha^i + \alpha^i = 0 \\ 4\alpha^i &= \alpha^i + \alpha^i + \alpha^i + \alpha^i = 0 \end{aligned} \quad (4.13)$$

Finalmente, el valor de los errores en un decodificador RS puede ser computado a través de la ecuación 4.14 que describe el algoritmo de Forney.

$$e_{j_l} = -\frac{\Omega(X_l^{-1})}{\sigma'(X_l^{-1})} \quad (4.14)$$

Donde  $\sigma'(X)$  es la derivada formal de  $\sigma(X)$ .

Se debe calcular esta ecuación para cada uno de los errores. Empezamos calculando la *ecuación clave* (ecuación 4.11), esta ecuación se calcula a partir del circuito mostrado en la figura 4.15.

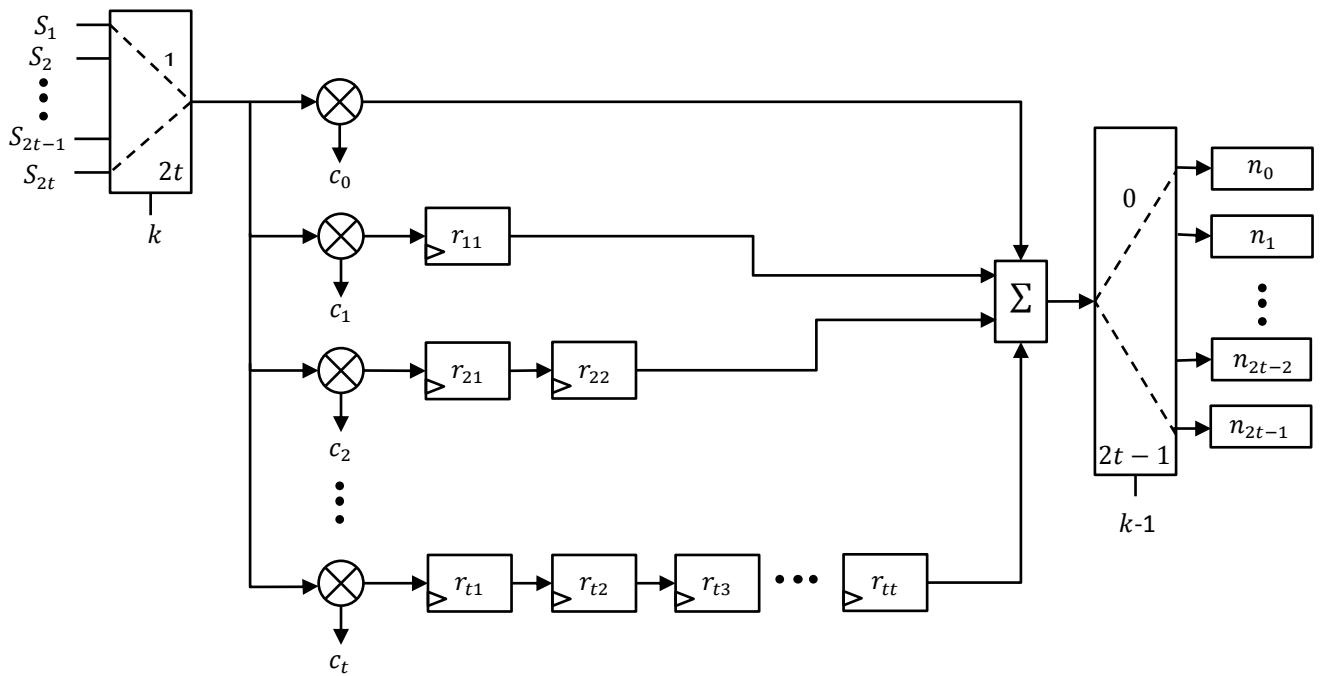


Figura. 4.15 Circuito propuesto para el cálculo de la ecuación clave del algoritmo de Forney.

Para ilustrar un poco mejor al lector a cerca del funcionamiento de este algoritmo, empecemos por analizar lo que debe hacer este. Si realizamos la expansión de  $S(X)\sigma(X)$  tenemos:

$$S(X)\sigma(X) = S_1\sigma_0 + S_1\sigma_1X + S_1\sigma_2X^2 + \dots + S_1\sigma_vX^v +$$

$$S_2\sigma_0X + S_2\sigma_1X^2 + S_2\sigma_2X^3 + \dots + S_2\sigma_vX^{v+1} +$$

$$S_3\sigma_0X^2 + S_3\sigma_1X^3 + S_3\sigma_2X^4 + \dots + S_3\sigma_vX^{v+2} +$$

⋮

$$S_{2t}\sigma_0X^{2t-1} + S_{2t}\sigma_1X^{1+(2t-1)} + S_{2t}\sigma_2X^{2+2t-1} + \dots + S_{2t}\sigma_vX^{v+(2t-1)}$$

Como sabemos solo se deben escoger los términos con potencias menores a  $2t$ , la inserción de los registros "r" permiten controlar esto. Note como se van actualizando cada uno de los coeficientes del polinomio  $n(X)$ :

$$k = 1; \quad n_0 = S_1\sigma_0$$

$$k = 2; \quad n_1 = S_2\sigma_0 + S_1\sigma_1$$

$$k = 3; \quad n_2 = S_3\sigma_0 + S_2\sigma_1 + S_1\sigma_2.$$

⋮

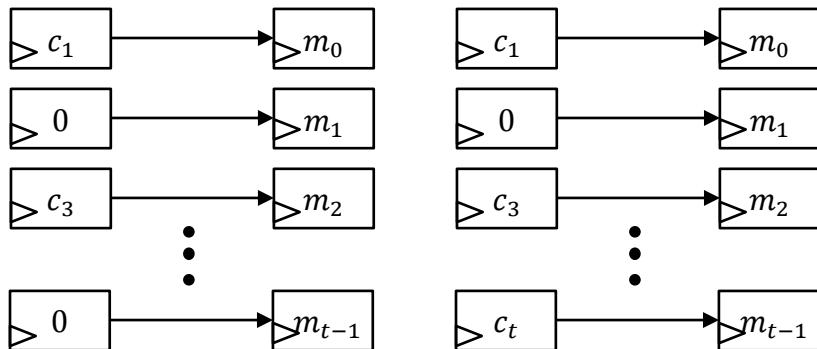
$$k = 2t; \quad n_{2t-1} = S_{2t}\sigma_0 + S_{2t-1}\sigma_1 + S_{2t-2}\sigma_2 + \cdots + S_t\sigma_v$$

Si compara  $n(X)$  con  $S(X)\sigma(X)$  notará que los términos con potencias mayores o iguales a  $2t$  han sido descartados, y por consiguiente  $n(x) = \Omega(X) = S(X)\sigma(X)(\bmod X^{2t})$ .

#### 4.4.2. Calculo de la derivada de $\sigma(x)$

El cálculo de la derivada en un campo finito es muy sencillo, si revisa el  $t$ -ésimo término de la ecuación 4.12 notará, que la derivada en un campo finito consiste en realizar un corrimiento a la derecha y hacer cero todos los registros impares (potencias impares) del polinomio que representa la derivada. Esto se puede evidenciar en el circuito de la figura 4.16. Si presta atención, notará que ninguno de los coeficientes de  $m(X)$  debe ser calculado de manera especial, solo equivale a alguno de los coeficientes de  $c(x)$  que han sido calculados previamente.

Note que el término que se almacena en el registro  $m_{t-1}$  depende de si  $t$  es par o impar, ya que por lo que evidenciamos en los ejemplos de la ecuación 4.13 si  $t$  es par  $m_{t-1}$  será cero, de lo contrario  $m_{t-1}$  guardara el valor de  $c_t$ .



*Figura. 4.16 Circuito propuesto para el cálculo de la derivada de  $\sigma(X)$ , a la izquierda el caso para  $t$  par, a la derecha el caso para  $t$  impar.*

Teniendo los polinomios  $m(X) = \sigma'(X)$  y  $\Omega(X)$  es posible realizar el cálculo del valor de cada error con la ecuación 20.

#### 4.5. Corrección del error

---

De la ecuación 4.4 y la ecuación 4.1, se estima el polinomio del error  $\hat{e}(x)$ :

$$\hat{e}(X) = e_{j_1}X^{j_1} + e_{j_2}X^{j_2} + \cdots + e_{j_{(v-1)}}X^{j_{(v-1)}} + e_{j_v}X^{j_v}$$

Si recordamos el *codeword* recibido  $r(X)$  puede ser expresado como:

$$r(X) = U(X) + e(X)$$

Note que para recuperar el *codeword* original  $U(X)$  basta con encontrar el valor del error y sumárselo a  $r(X)$  (recuerde que la resta es equivalente a una suma en operaciones de modulo 2):

$$r(X) = r(X) + \hat{e}(X) = U(X) + e(X) + \hat{e}(X)$$

#### 4.5.1. Algoritmo e implementación

Luego de haber calculado  $\Omega(X)$  y  $\sigma'(X)$  basta con implementar la ecuación 4.14 para calcular el valor de cada error y poder obtener  $e(X)$  para sumarlo con  $r(X)$  y finalmente recuperar  $U(X)$ . Para ello se implementa el circuito de la figura 4.17.

El primer selector arroja las posiciones del error, las cuales se les calcula su inversa. El cálculo de la inversa se realiza de manera sencilla por medio de una tabla, como la tabla 4.6. Note que básicamente el inverso se calcula con la diferencia entre la cantidad de elementos del campo  $n$  y el exponente del  $\alpha$  al que se le desea calcular la inversa.

IN	$\alpha^0$	$\alpha^1$	$\alpha^2$	...	$\alpha^{n-2}$	$\alpha^{n-1}$
OUT	$\alpha^0$	$\alpha^{n-1}$	$\alpha^{n-2}$	...	$\alpha^2$	$\alpha^1$

Tabla 4.6 Cálculo de la inversa a través de una tabla.

Posteriormente, estos elementos inversos son evaluados en  $\Omega(X)$  (registros  $n$ ) y  $\sigma'(X)$  (registros  $m$ ), de esta manera se obtienen los términos  $\Omega(X_l^{-1})$  y  $\sigma'(X_l^{-1})$  necesarios para evaluar la ecuación 4.14 que nos permite el cálculo del valor del error. En el denominador de la ecuación 4.14 se encuentra el término  $\sigma'(X_l^{-1})$ , por ende se le debe hallar la inversa de este para efectuar un producto entre  $\Omega(X_l^{-1})[\sigma'(X_l^{-1})]^{-1}$  que es más sencillo que evaluar la división. Finalmente después de obtener el valor del error este es sumado a la entrada  $r(X)$  para realizar la corrección del símbolo si es necesaria.

Note que las localizaciones del error están sincronizadas con la posición del símbolo en la entrada, por ende solo las localizaciones del error no nulas tendrán un efecto en la corrección, de lo contrario la entrada se sumará con cero y no producirá ninguna corrección ya que esta no es requerida.

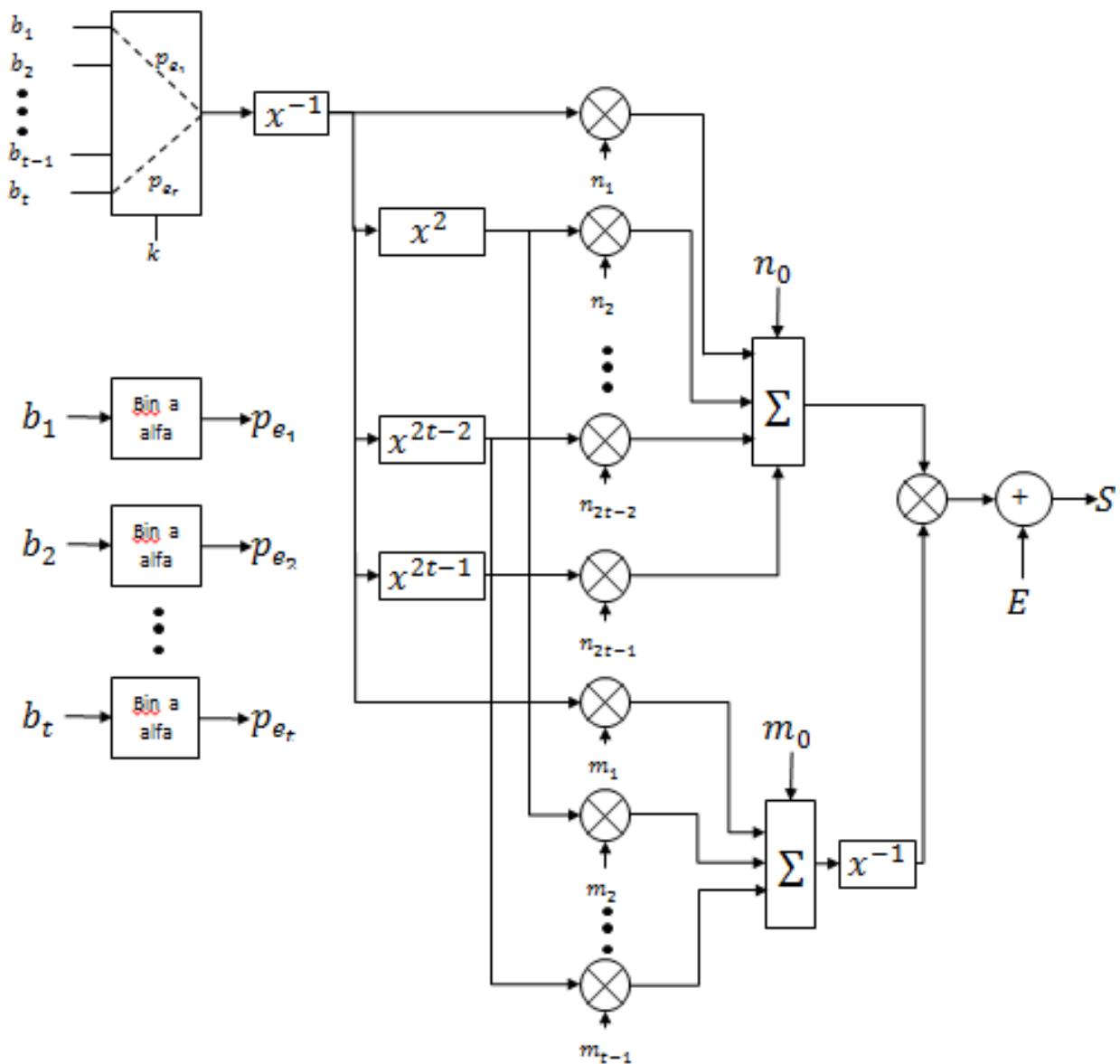


Figura. 4.17 Circuito propuesto para efectuar la corrección del error.

# 5

## Generación de ruido en la FPGA

### 5.1. Generador de números pseudo-aleatorios

---

Se requiere diseñar un circuito que genere una secuencia pseudo-aleatoria de números y con esta simular un generador de ruido Gaussiano capaz de corromper la transmisión de los bits en el canal de comunicación, dentro de la FPGA.

#### 5.1.1. Serie pseudo-aleatoria

La aleatoriedad se puede definir como incertidumbre y la imposibilidad de predecir un evento. Los números aleatorios se dividen en números aleatorios reales y números pseudo-aleatorios. Los números aleatorios reales se pueden considerar como aquellos que no son posibles predecir completamente y los cuales solo se producen por procesos físicos de naturaleza estocástica [4]. Los números pseudo-aleatorios comúnmente son generados iterando una fórmula matemática, tan pronto como la fórmula y el valor inicial son determinados, la secuencia de números pseudo-aleatorios puede ser determinada con exactitud y volverse completamente predecible. Debido a que es muy difícil implementar una secuencia de números realmente aleatorios se utilizan secuencias de números pseudo-aleatorios en la mayoría de las aplicaciones, ya que si el valor inicial y la fórmula matemática empleada para generar la secuencia de números es desconocida, esta no puede ser predicha, debido a esto las secuencias pseudo-aleatorios son usadas en la mayoría de las aplicaciones[16].

### 5.1.2. Generación de una secuencia pseudo-aleatoria

Se utiliza un LFSR (registro de desplazamiento con realimentación lineal) para generar la secuencia pseudo-aleatoria, esta se genera al introducir un circuito de realimentación en el desplazamiento del registro a la izquierda. El resultado de esta implementación es la generación de una secuencia pseudo-aleatoria de  $n$  bits con  $2^n - 1$  posibles valores donde  $n$  es el número de bits del registro a desplazar y realimentar. Esta implementación tienen la particularidad de que los resultados obtenidos van a estar en un campo finito. Las operaciones que se realizan al introducir la realimentación en el primer bit del registro se basan en operaciones de módulo 2 por esta razón el primer bit del registro es el resultado de operar dos o más bits del registro completo como se muestra en la figura 26.

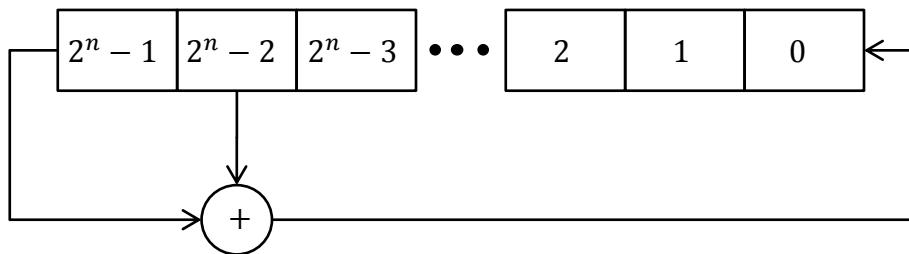


Figura. 5.1 LFRS para generar una secuencia pseudo-aleatoria.

Este registro cambia por cada pulso de reloj y la secuencia que se repite cada  $2^n - 1$  ciclos de reloj. A partir de la teoría de campos de Galois se puede demostrar que para cualquier valor de  $n$  existe al menos una ecuación capaz de generar un secuencia aleatoria que se repite cada  $2^n - 1$  ciclos de reloj y en donde el único estado que no es visitado es el todos los bits en cero. De hecho si llegara a pasar que el registro tiene todos sus bits en cero la secuencia se estancaría ya que no es posible sacar el registro de ese estado, debido a que la suma en modulo dos se representa con la operación XOR.

Por esta razón siempre se debe dar un valor inicial diferente de cero al registro, que comúnmente se conoce como semilla. Esta semilla puede ser cualquier número dentro del conjunto de posibilidades que puede tomar el registro exceptuando el cero. El uso de un LFRS es una implementación óptima para solucionar este problema ya que el uso del hardware es el mejor al solo utilizar un registro de bits y unas cuantas operaciones lógicas, a diferencia de la implementación de un contador.

El circuito de realimentación depende del número de bits que tenga el registro. Los bits a tomar para ser operados y realimentar la entrada ya están determinados sobre una base ad hoc[5].

Tamaño del LFSR	Realimentación
2	b0 XOR b1
3	b1 XOR b2
4	b2 XOR b3
5	b2 XOR b4
6	b4 XOR b5
7	b3 XOR b6
8	b3 XOR b4 XOR b5 XOR b6
12	b5 XOR b7 XOR b10 XOR b11
16	b10 XOR b11 XOR b12 XOR b15
32	b9 XOR b19 XOR b30 XOR b31
64	b59 XOR b60 XOR b62 XOR b63
128	b98 XOR b110 XOR b125 XOR b127

Tabla 5.1 Bits de realimentación según longitud de LFSR.

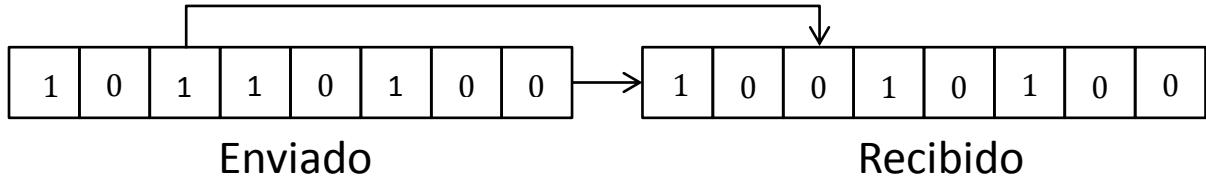
Las expresiones de la tabla 5.1 proveen lo que se conoce como máxima longitud del LFSR, y además se garantiza que se genere la secuencia de  $2^n - 1$  valores con duración de  $2^n - 1$  ciclos de reloj. De lo contrario podría realizarse una realimentación que genere menos de los  $2^n - 1$  símbolos posibles. Como se mencionó antes, cualquiera de los generadores pseudo-aleatorios, no son exactamente aleatorios por que la secuencia se repite cada cierto tiempo, sin embargo se utilizan por que los períodos en que se repiten son muy largos que pueden considerarse aleatorias las secuencias. Por ejemplo para un reloj de 50 MHz y un LFSR de 60 bits de longitud, la secuencia tardaría en repetirse 741 años lo cual es suficientemente largo para considerarse aleatorio. En el caso de un registro de 64 bits la secuencia se repetiría cada 11861 años.

## 5.2. Error de bit y error de ráfaga

---

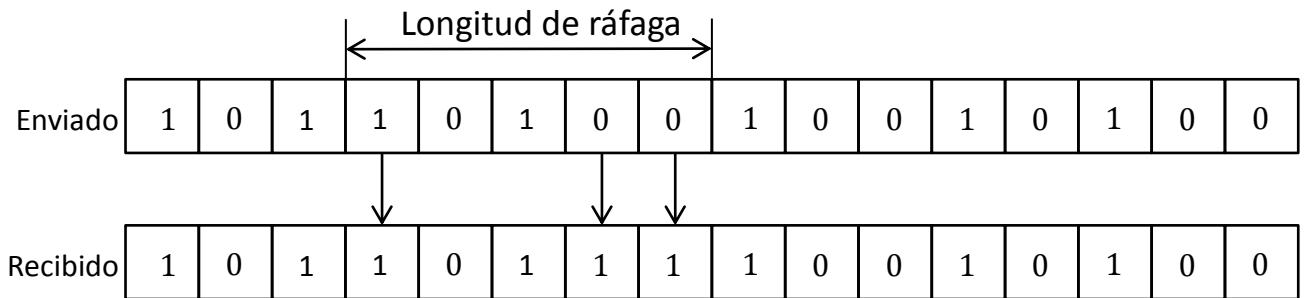
Existen dos tipos de errores que se presentan en las transmisiones seriales. El error de bit y el error en ráfaga.

El error de bit consiste en la modificación de un único bit dentro de la secuencia de bits en el canal de transmisión y es el tipo de error menos probable ya que el intervalo de bit es de  $1/f$ , el ruido debería tener una duración muy corta para afectar solo a un bit. En una transmisión en paralelo existe una mayor probabilidad de que exista un error de bit, ya que un cable podría dañarse y uno de los bits se corrompería debido a esto. En la figura 5.2 se puede apreciar un ejemplo.



*Figura. 5.2 Error de bit en una transmisión serial.*

El error en ráfaga por el contrario significa que dos o más bits han cambiado en una unidad de datos[13]. Los errores en ráfaga no significan que los errores se produzcan consecutivamente. La longitud de la rafa se mide desde el primer bit que se vio afectado hasta el último, sin importar si en la mitad hay bits que no se vieron afectados por la rafaga. En la figura 5.3 podemos apreciar un ejemplo de esto.



*Figura. 5.3 Error de ráfaga en una transmisión serial.*

Partiendo de estas definiciones se desarrolló un módulo dentro de la FPGA capaz de corromper los datos entre el codificador y el decodificador para probar la capacidad de corrección del código Reed-Solomon, para esto se generó una distribución gaussiana de ruido para la cadena de símbolos a transmitir.

### 5.3. Modulo generador de error para prueba del código Reed-Solomon

---

A cada uno de los bits dentro del mensaje enviado se le asigna una probabilidad de error como se mencionó antes. Se sabe que la cantidad de bits en el mensaje es equivalente a:

$$[\log_2 n] * n$$

A partir de esto se calcula una densidad de probabilidad Gaussiana, representada mediante la siguiente ecuación:

$$d(bit) = 65335 * e^{-\left(\left(\sigma^{-1}\left(bit - \frac{m*n-1}{2}\right)\right)^2\right)} \quad (5.1)$$

Donde:

*bit* es el bit dentro del codeword.

*m* es la cantidad de bits por símbolos.

*n* es la cantidad de símbolos por *codeword*.

$\sigma$  es una constante que define la apertura de la función.

En la figura 5.4 podemos observar una función generadora de probabilidad para un código RS (7,3). Note como el rango se extiende desde el primer bit hasta el último en el mensaje a transmitir.

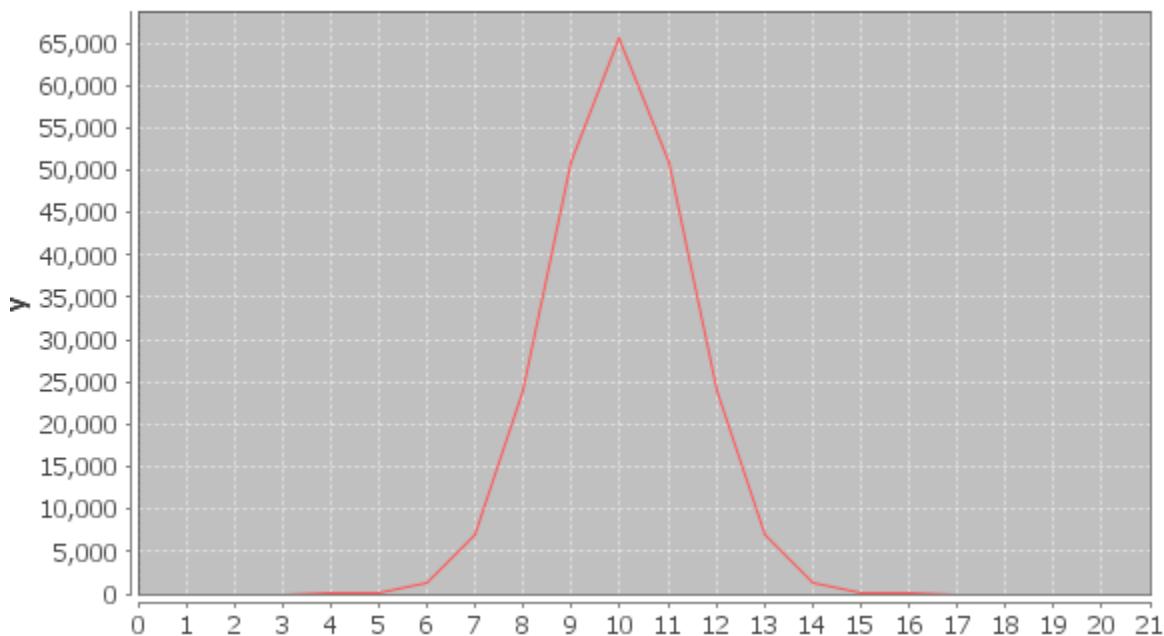


Figura.5.4 Densidad de probabilidad de error para un código 7-3 con sigma de 0.5.

Escalamos esta distribución en 65535 para realizar una comparación dentro de la FPGA con un registro de 16 bits que se va modificando aleatoriamente. A cada uno de los 21 bits se le asigna un valor de probabilidad como se muestra en la figura 5.4 y de esta forma se emula el ruido dentro de la FPGA. El circuito de la figura 5.5 muestra como se corrompe un solo bit de los 21 mostrados por la figura 5.4. Note que si el valor de probabilidad asignado al bit (en la figura 5.5 Prob. b0) es mayor a la mitad de 65535 la probabilidad de que se genere un bit corrupto en el canal es mayor al 50%. Para resumir la probabilidad de cualquiera de los bits se definiría como:

$$\text{probabilidad de corromper el bit} = \frac{\text{Probabilidad asignada al bit}}{65535} \quad (5.2)$$

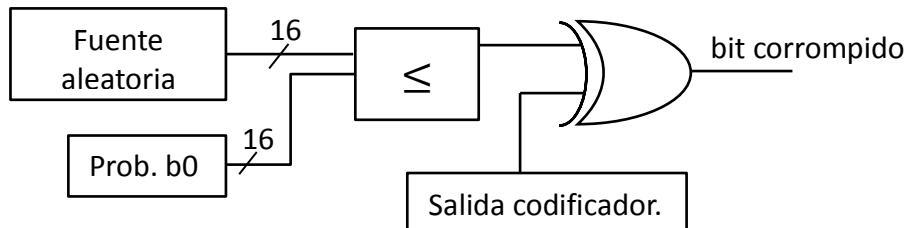


Figura. 5.5 parte de la implementación del generador de ruido.

Cada una de estas probabilidades es calculada en el software que se explica en el capítulo 6 y es enviada a la FPGA donde se construye un arreglo de probabilidades que representa la distribución de ruido Gaussiano mostrada en el software y que el usuario puede modificar a su gusto. Para esto se configuro la FPGA para que funcionara en dos modos, un modo de configuración y otro de transmisión. En el modo de configuración se fija la densidad de probabilidad del error y el tipo de canal; si es lineal o no lineal. Luego de fijar estos parámetros se pasa al modo de transmisión donde se puede poner a prueba el código Reed-Solomon dependiendo del ruido y estilo del canal configurados.

## 5.4. Añadiendo una no-linealidad al canal.

---

El canal no-lineal se modelo con una función cuadrática por la cual pasan los símbolos antes de pasar por el módulo de ruido. Al ser afectados estos por el ruido pasan por la función inversa y de allí a la entrada del codificador como muestra la figura. 5.6.



Figura. 5.6 Diagrama de boques del canal implementado dentro de la FPGA.

Note que si en la configuración se fija un canal no lineal los datos toman las trayectorias de las líneas puntuadas evitando que se aplique una no-linealidad a estos.

El bloque que le agrega la no linealidad se puede representar con la función cuadrática  $X^2$  como se muestra en la figura 5.7. Note que a primera vista parece que hubiera un error ya que se parece a una función lineal, no obstante, si se presta atención, al elevar un elemento al cuadrado lo que esta haciendo es multiplicar su exponente por dos y como lo que se esta

graficando son esos exponentes se produce como resultado una recta con pendiente 2 para el caso de un código con un  $n$  mayor a 16. Sin embargo si graficamos la función  $X^2$  para un código con un  $n$  de 7 podremos observar como al ser un campo finito los alfas mayores a  $\alpha^6$  son equivalentes a alfas entre  $\alpha^0$  y  $\alpha^6$  debido a las propiedades del campo de Galois. Esta representación de la función  $X^2$  en un campo de Galois de 7 símbolos es más interesante y es mostrada en la figura. 5.8. Note además que es una función periódica.

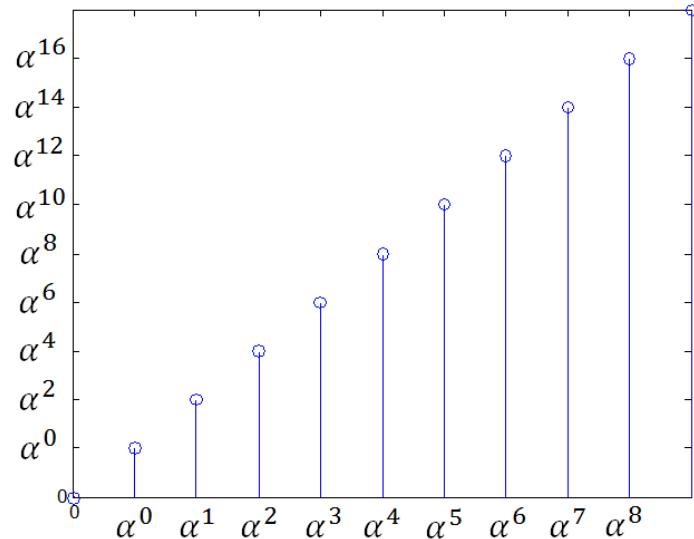


Figura.5.7 Función  $x^2$ (representación para un código con un  $n$  mayor a 16)

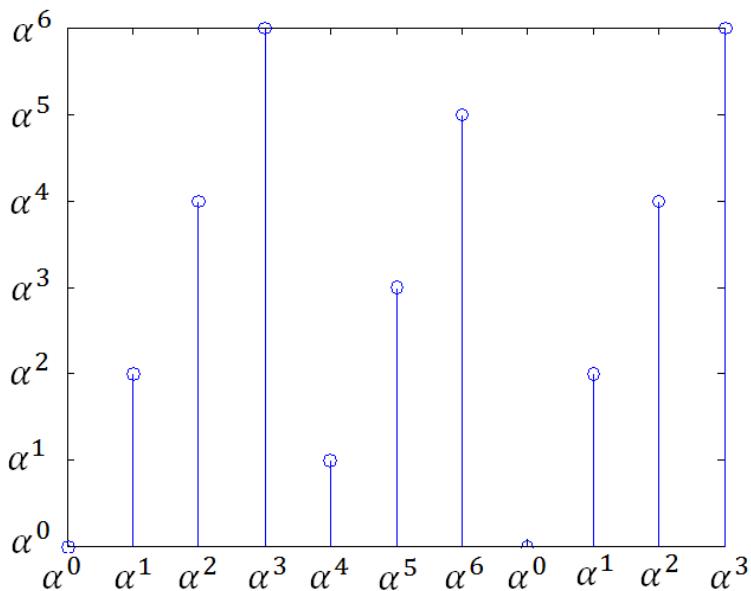


Figura.5.8 Función  $X^2$ , representación para un código con un  $n = 7$ .

## **SINTESIS DE LA IMPLEMENTACION SOBRE LA FPGA.**

Como se ha mencionado a través de los últimos capítulos dentro de la FPGA se producen más procesos que solo la codificación y decodificación del código Reed-Solomon. Estos pueden resumirse en:

- 1) Fijar la distribución de ruido y tipo de canal (Modo de configuración).
- 2) Modo transmisión.
- 3) Recibir datos y enviarlos al codificador.
- 4) Enviar datos de la salida del codificador al generador de ruido.
- 5) De la salida del generador de ruido enviarlos al decodificador y al software para comprobar la probabilidad de error.
- 6) De la salida del codificador al software para calcular el porcentaje de símbolos corregidos.

Para poder llevar acabo cada uno de estos procesos se implementó un FSM que controla todo el proceso durante una prueba al código. Describir todo el proceso mediante un diagrama de flujo o de estados resultaría muy complejo por lo que se muestra en la figura 5.9 un diagrama de flujo simplificando el proceso.

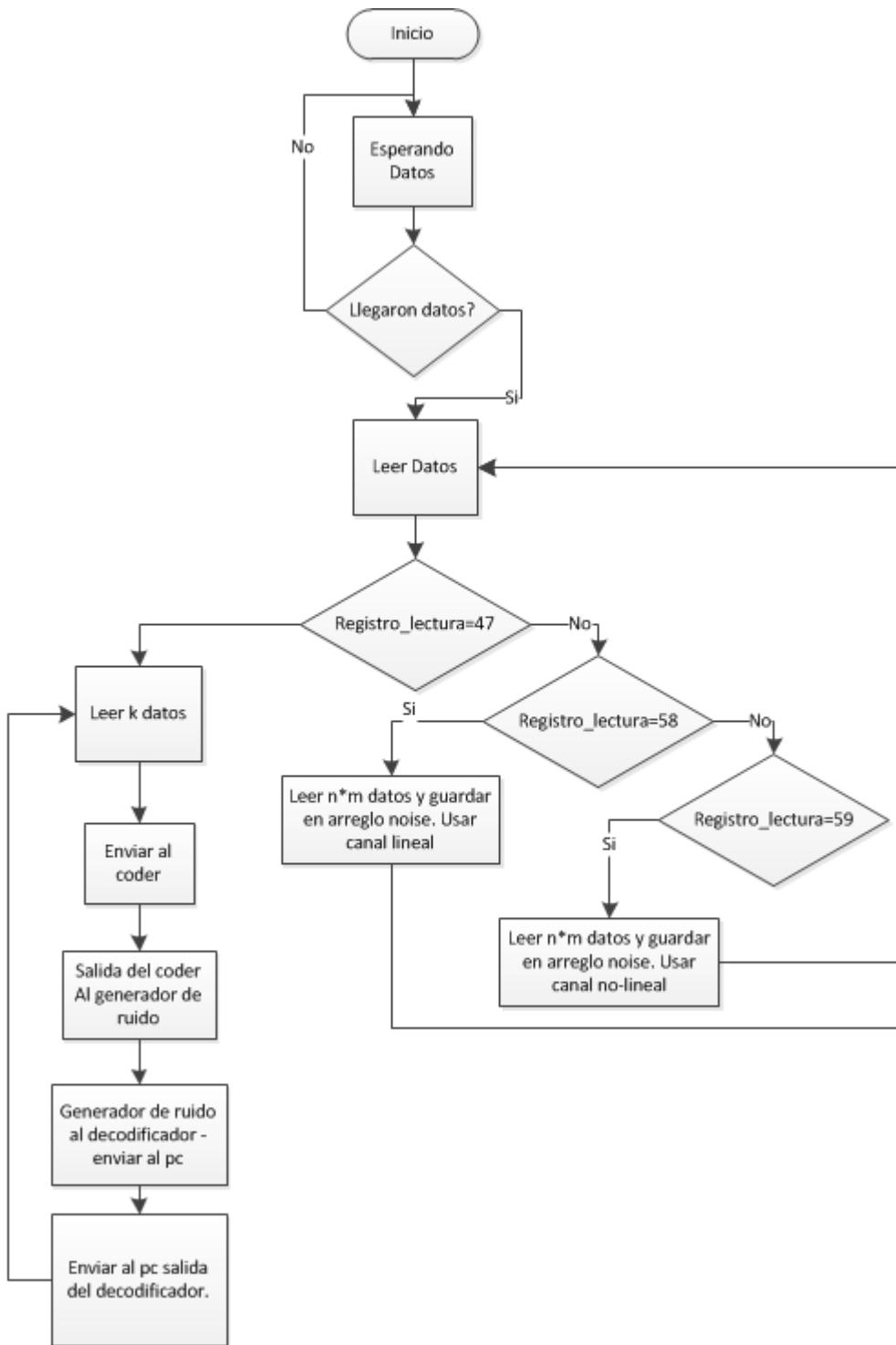


Figura. 5.9 Diagrama de flujo del proceso para llevar a cabo la configuración y puesta en funcionamiento del codificador y decodificador RS. Asociado al software.

# 6

## Software

Para controlar la configuración aplicada a la FPGA y realizar pruebas al codificador y decodificador Reed-Solomon se desarrolló un software que envía datos a la FPGA a través de un conversor USB-Serial que esta conectado a la FPGA. Esta tiene implementado un módulo de comunicación que entrega estos datos a un FSM que controla todo el flujo de datos dentro de la FPGA. La FPGA entiende dos comandos básicos al iniciar:

Comando	Acción
47	Entrar en modo de transmisión.
58	Entrar en modo comando.

Tabla 6.1 Comandos de inicio para la FPGA.

**Modo de transmisión:** En el modo de transmisión la FPGA recibe *codewords* desde el software para que sean procesador por el codificador, pasen por el módulo de ruido entren al decodificador y retornen al PC.

**Modo de configuración:** En el modo de configuración la FPGA recibirá  $2(n * [\log_2 n])$  datos provenientes del PC para fijar las  $(n * [\log_2 n])$  probabilidades del vector de ruido. Recibe dos datos por probabilidad ya que estas son valores de 16 bits y la transmisión solo permite enviar 8 bits. El diagrama de la figura 6.1 explica como esta conectado físicamente el sistema implementado.

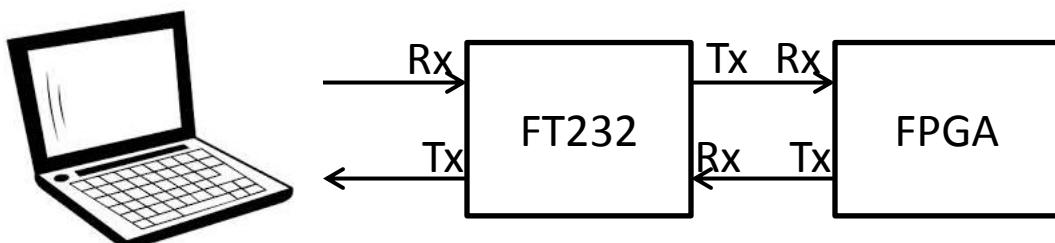


Figura. 6.1 Diagrama de conexionado del sistema implementado.

## 6.1. Comunicación entre el panel de control y la FPGA

Para la comunicación entre el computador donde se estará el panel de control para realizar las pruebas del codificador y decodificador Reed-Solomon, se utiliza una comunicación UART (Universal Asynchronous Receiver/Transmitter). Para esto se diseñan dos módulos en VHDL (uno de recepción y otro de transmisión). Esta comunicación es esencial para lograr la configuración de los parámetros de ruido y enviar y recibir los *codewords* con los que se hacen las pruebas del codificador y decodificador Reed-Solomon.

### 6.1.1. Módulo de recepción

Para entender como recibir una palabra transmitida a través de este protocolo lo primero que se debe entender es lo que llega al receptor. La trama de bits mostrada en la figura. 6.2 muestra un ejemplo de esto.

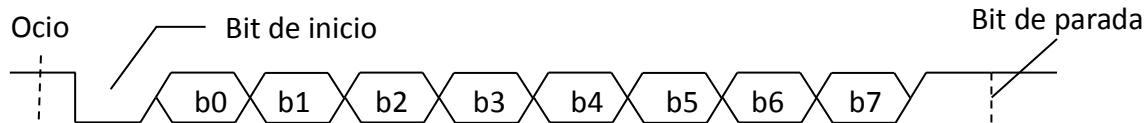


Figura. 6.2 Trama transmitida.

Note que el canal entre el PC y la FPGA siempre está en '1', indicando un estado de ocio en el que solo se espera por el bit de inicio, un '0' que indica el comienzo de la transmisión[14]. Luego se almacenan cada uno de los 8 bits en un registro hasta que llega el bit de parada y deja al canal de nuevo en estado de ocio. Debemos leer cada uno de estos bits exactamente en la mitad de su tiempo para evitar errores de lectura, para ello se utiliza un sobre-muestreo; es decir un reloj más rápido que la velocidad de transmisión para poder dividir el tiempo de un bit en varios pulsos de reloj. Se utiliza un reloj 16 veces más rápido que la velocidad de transmisión y se aprovecha el bit de parada para sincronizar un contador que nos sirva para determinar cuando estamos exactamente en la mitad de un bit para poder leerlo con exactitud, luego de obtener el bit se utiliza un segundo contador que sirve de selector para guardar los bits en un registro para su posterior lectura. Este procedimiento se entiende con más claridad al ver el siguiente diagrama de flujo de la figura 6.3. Note que se utilizan 4 estados:

- Estado de ocio (estado="00").
- Estado de sincronización (estado="01").
- Estado de lectura (estado="10").
- Estado de bit de parada (estado="11").

Estos son descritos en una entidad VHDL sintetizando una máquina de estados finitos (FSM), para lograr el objetivo. Hay que tener en cuenta que como la transmisión es asincrónica, ambos dispositivos (PC y FPGA) deben ponerse de acuerdo en el baudrate que van a manejar.

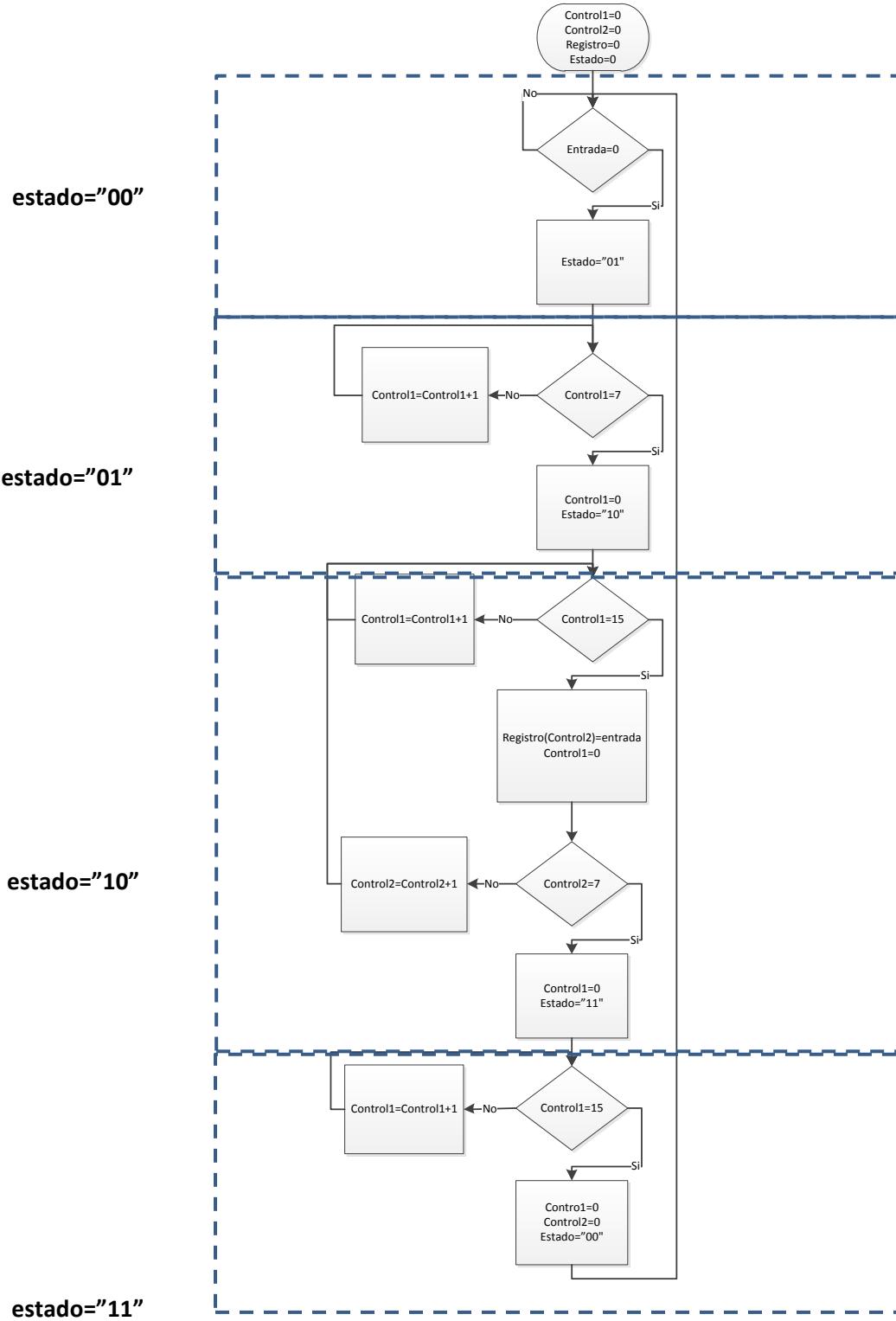


Figura. 6.3 Diagrama de flujo del receptor (UART)

### 6.1.2. Módulo de transmisión

El módulo de transmisión es muy parecido al módulo de recepción, posee el mismo número de estados y la estructura de su diagrama de flujo es casi idéntica.

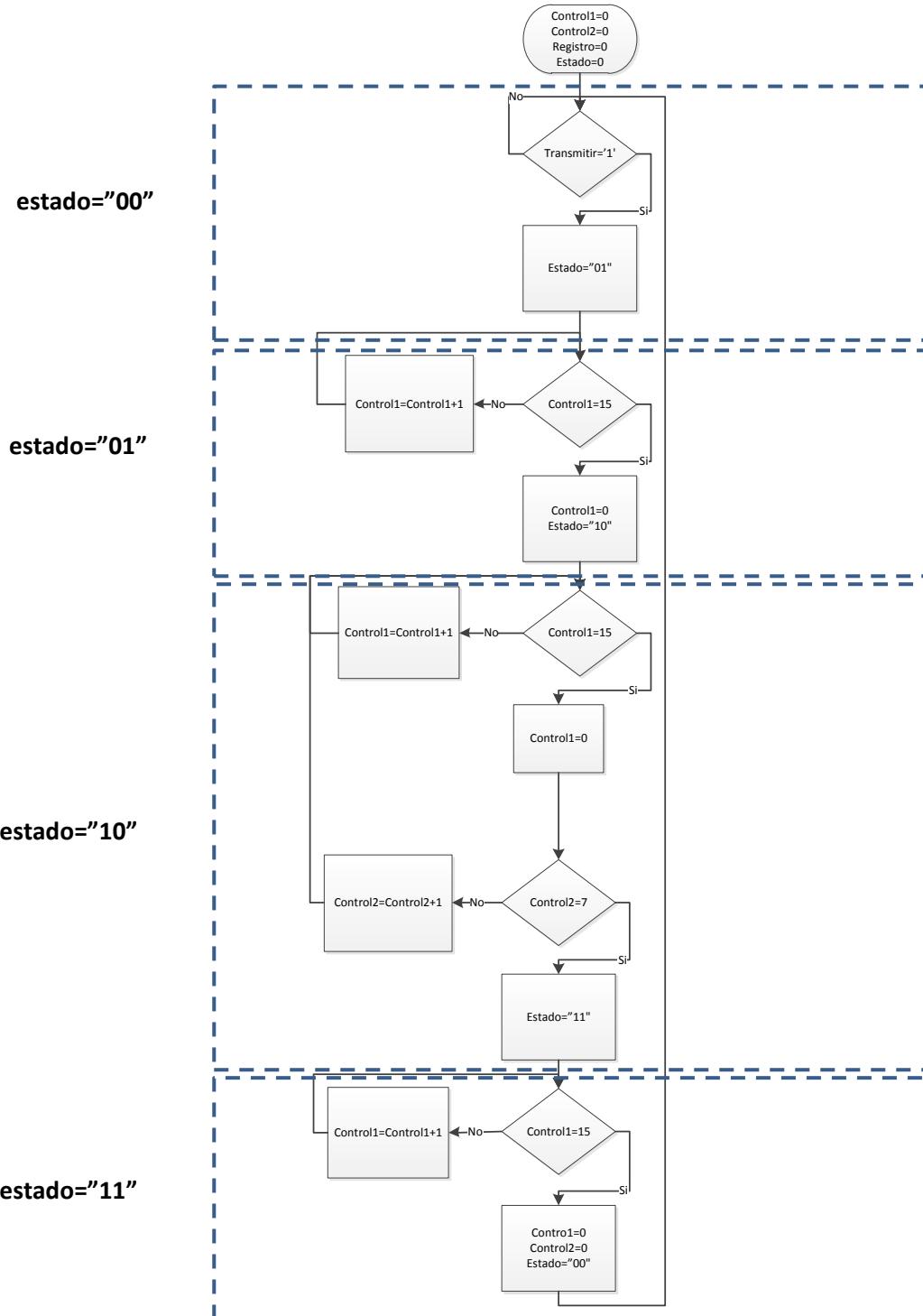
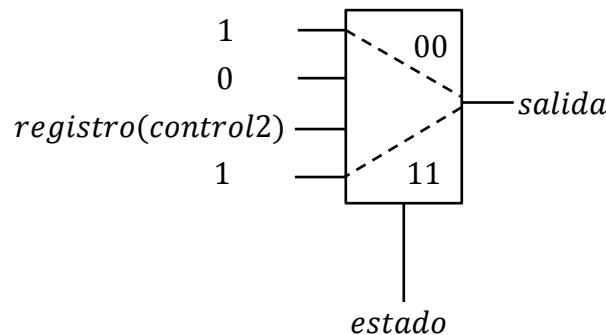


Figura. 6.4 Diagrama de flujo transmisión (UART).

Además de la máquina de estados que utiliza el diagrama de la figura 6.4. Se necesita un circuito que dependiendo del estado en el que se encuentre la trasmisión genere la salida del circuito de la siguiente forma:

- '1' para el estado de ocio (estado="00").
- '0' para el bit de inicio (estado="01").
- Salida=registro (control2) (estado="10").
- '1' para el bit de parada (estado="11").

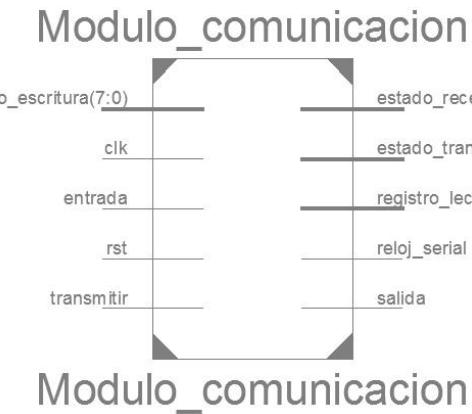
El circuito que describe este comportamiento es un multiplexor como el mostrado en la figura 6.5.



*Figura. 6.5 Multiplexor a la salida del módulo de trasmisión.*

Donde registro corresponde al registro donde se almacena el Byte a transmitir y el cual tiene de selector a control2 que selecciona el bit a trasmitir del registro.

Note también que para empezar la trasmisión, la entrada “trasmitir” debe estar en ‘1’, de esta forma se controla cuando enviar lo que está en el registro. Ambos módulos el de recepción y el de transmisión se integran bajo una misma entidad para su uso en el resto de la implementación como se muestra en la figura 6.6.



*Figura. 6.6 Entidad - módulo de comunicación UART.*

Es importante mencionar la configuración del archivo UCF que se utilizó en la implementación sobre la FPGA. Básicamente a la FPGA se le configuran una entrada y una salida que corresponden a la entrada y salida del módulo de comunicación y los LEDs que se enciende cuando la FPGA está lista para operar. La configuración del archivo UCF se ve en la figura 6.7. Este archivo también es generado por el software cuando se genera cualquier código RS.

```
#Entradas externas
NET "rst" LOC = "U10";
NET "entrada" LOC = "E16" | IOSTANDARD = LVCMOS33;

#Salidas externas
NET "leds(7)" LOC = "W21";
NET "leds(6)" LOC = "Y22";
NET "leds(5)" LOC = "V20";
NET "leds(4)" LOC = "V19";
NET "leds(3)" LOC = "U19";
NET "leds(2)" LOC = "U20";
NET "leds(1)" LOC = "T19";
NET "leds(0)" LOC = "R20";
NET "salida" LOC = "F15" | IOSTANDARD = LVCMOS33;

#Entradas internas
NET "clk" LOC = "E12" | IOSTANDARD = LVCMOS33;
```

Figura. 6.7 UCF utilizado en la implementación.

## 6.2. Descripción del panel de control

---

Después de haber realizado la comunicación entre el ordenador y la FPGA, se desarrolló un programa para configurar el módulo de error dentro de la FPGA y probar los códigos Reed-Solomon que se generen con este. Para esto se utilizó una API para graficar el error, donde el usuario puede introducir los parámetros sigma y un corrimiento, y esta genera una representación de una distribución de error Gaussiana como se ve en la figura 6.8.

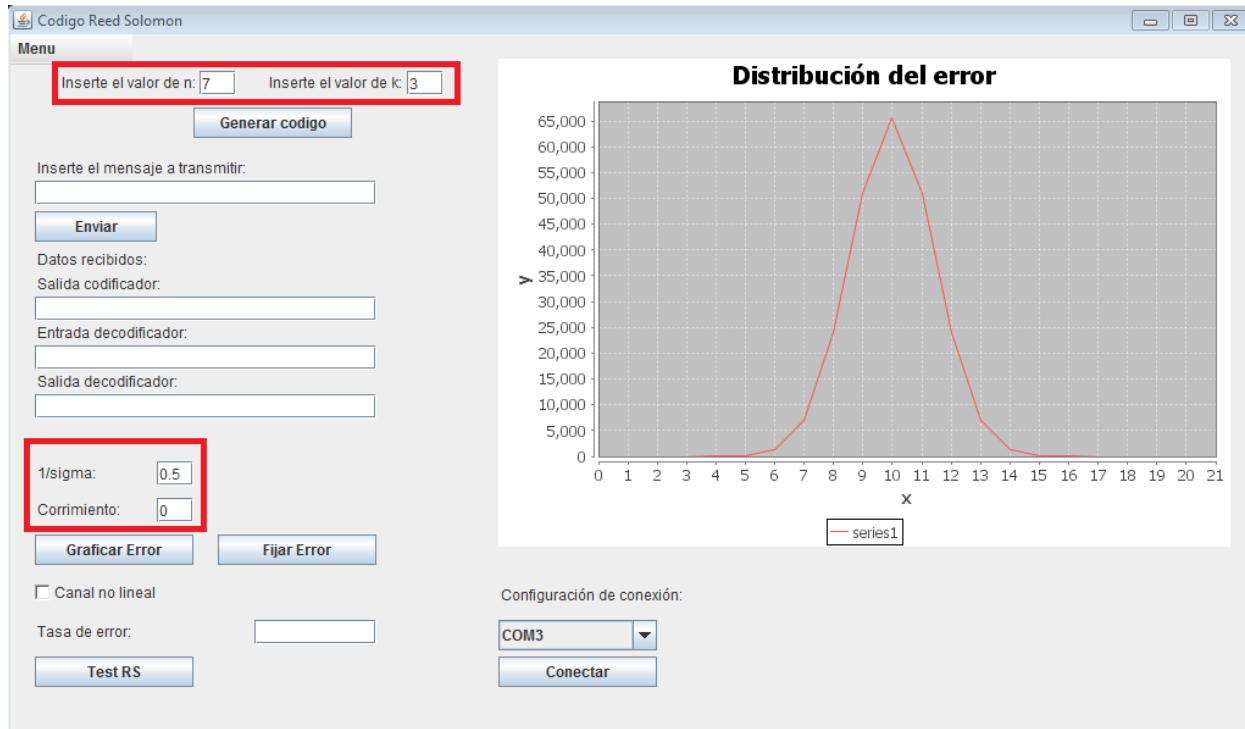


Figura. 6.8 Configuración del error en el software de control.

Note que como se mencionó en el capítulo anterior la distribución depende del código que se vaya a utilizar. Ya que es aquí donde elaborara el vector de probabilidades para asignarlas dentro de la FPGA a cada uno de los bits que salen del codificador. Luego de haber construido la probabilidad de error deseada, se da clic en el botón “Fijar error”, el software envía la información a la FPGA poniendo automáticamente a esta en modo de transmisión. Adicionalmente se le puede añadir la opción de canal no lineal permitiendo que dentro de la FPGA se le introduzca una no-linealidad al canal entre el codificador y el decodificador.<sup>10</sup>

A partir del momento en el cual la FPGA entra en modo de transmisión, el usuario puede insertar un mensaje con el cual desee hacer el experimento y en respuesta la FPGA retorna los símbolos a la entrada del decodificador (estos ya pasaron por el módulo de ruido) y los símbolos a la salida del decodificador, de esta manera se pueden visualizar los símbolos que se han dañado debido al módulo de error y como estos son corregidos por el decodificador.

Además se tiene un modo de test, en el cual se envían automáticamente 1000 símbolos (todos estos aleatorios) a la FPGA con los cuales se calcula el error de cada uno de los bits que entran al decodificador haciendo una comparación entre cada uno de los datos que están llegando y lo que debería salir del codificador, de esta manera se reconstruye la distribución de los errores para cada uno de los bits y se representa gráficamente en el software. La otra mitad de la

<sup>10</sup> La descripción de la no-linealidad que se agrega al canal fue descrita en el capítulo anterior.

información que llega desde la FPGA que son los símbolos provenientes de la salida del decodificador, nos permiten analizar el porcentaje de palabras enviadas corregidas, que se visualiza en el cuadro de texto con el label “Tasa de error”.

Si se desea establecer una distribución de error no Gaussiana se puede realizar enviando en orden los siguientes comandos:

- 1) Enviar un 58. (Indica a la FPGA que vamos a fijar el vector de ruido).
  - 2) Enviar parejas de valores que representan números de 16 bits para cada fijar cada una de las probabilidades de error de los bits; donde el primer Byte es el más significativo.

## Ejemplo:

Se desea asignarle al segundo bit de un código 7-3 una probabilidad del 80% para que este se corrompa. Primero se calcula el valor que se desea enviar de la siguiente forma:

$$\begin{aligned} \text{probabilidadenbit} &= 65535 * \%error \\ \text{probabilidadenbit} &= 65535 * 0.9 = 58982 \end{aligned}$$

Luego se separa este número en un par de números de 8 bits:

$$MSB = \text{probabilidadenbit}/256$$

$$LSB = \text{probabilidadenbit}\%256$$

$$MSB = \frac{58982}{256} = 230$$

$$LSB = 58982 \% 256 = 102$$

Finalmente el arreglo a ser enviado es el siguiente:

Luego para pasar al modo de transmisión se envía el número 47. A partir de ahora se puede ejecutar una prueba con el botón “Test RS” o empezar individualmente a enviar palabras código para ver la codificación y decodificación de esta.

Este software se realizó con estas características con el propósito de poder evaluar el comportamiento del error y la tasa de error cuando el codificador y el decodificador estuvieran separados físicamente. Ya que de esta forma es posible poner a prueba este codificador Reed-Solomon en canales reales de comunicación.

### 6.3. Generación de entidades VHDL a través del software

---

La generación de VHDL a través del software nos permitió una flexibilidad que no podíamos lograr con la instrucción GENERIC de VHDL. A partir del software escribimos los archivos .vhd que describen el funcionamiento de cualquier código RS. Durante el desarrollo de la implementación y el estudio de los códigos Reed-Solomon fue posible identificar los parámetros que modificaban el hardware dependiendo de un código y otro en las diferentes etapas de codificación de decodificación del código. A partir de esto se escribió el código necesario para generar los elementos básicos de cada etapa y mediante ciclos repetitivos adicionar cantidades deseadas de estos elementos.

Para la implementación de la generación de estos archivos mediante el software se hizo una división por clases, en donde cada clase representaba la creación de una entidad VHDL. Es decir que se crearon tantas clases como entidades VHDL existen para la generación del codificador y decodificador Reed-Solomon, estas se pueden observar en la figura 6.9. Estas clases se crearon estáticas con el fin de poderlas usar sin instanciar objetos y poderlas utilizar más fácilmente. La estructura básica de estas clases consiste en una función estática llamada vhd que se encarga de escribir todo el código VHDL dentro de una variable tipo String y dependiendo de los parámetros del código ciertas partes de las cadenas de texto que se adicionan a la variable tipo String se generan a partir de ciclos que crean los elementos necesarios para la descripción correcta de cada entidad dependiendo del valor de  $n$  y  $k$  del código Reed-Solomon que se quiera describir. Luego de que se ha llenado la variable tipo String con todo lo que se desea escribir en el archivo .vhd se crea un archivo .vhd a partir de un objeto FileWriter que es instanciado a partir de una de las clases Java pertenecientes al paquete Java.io.

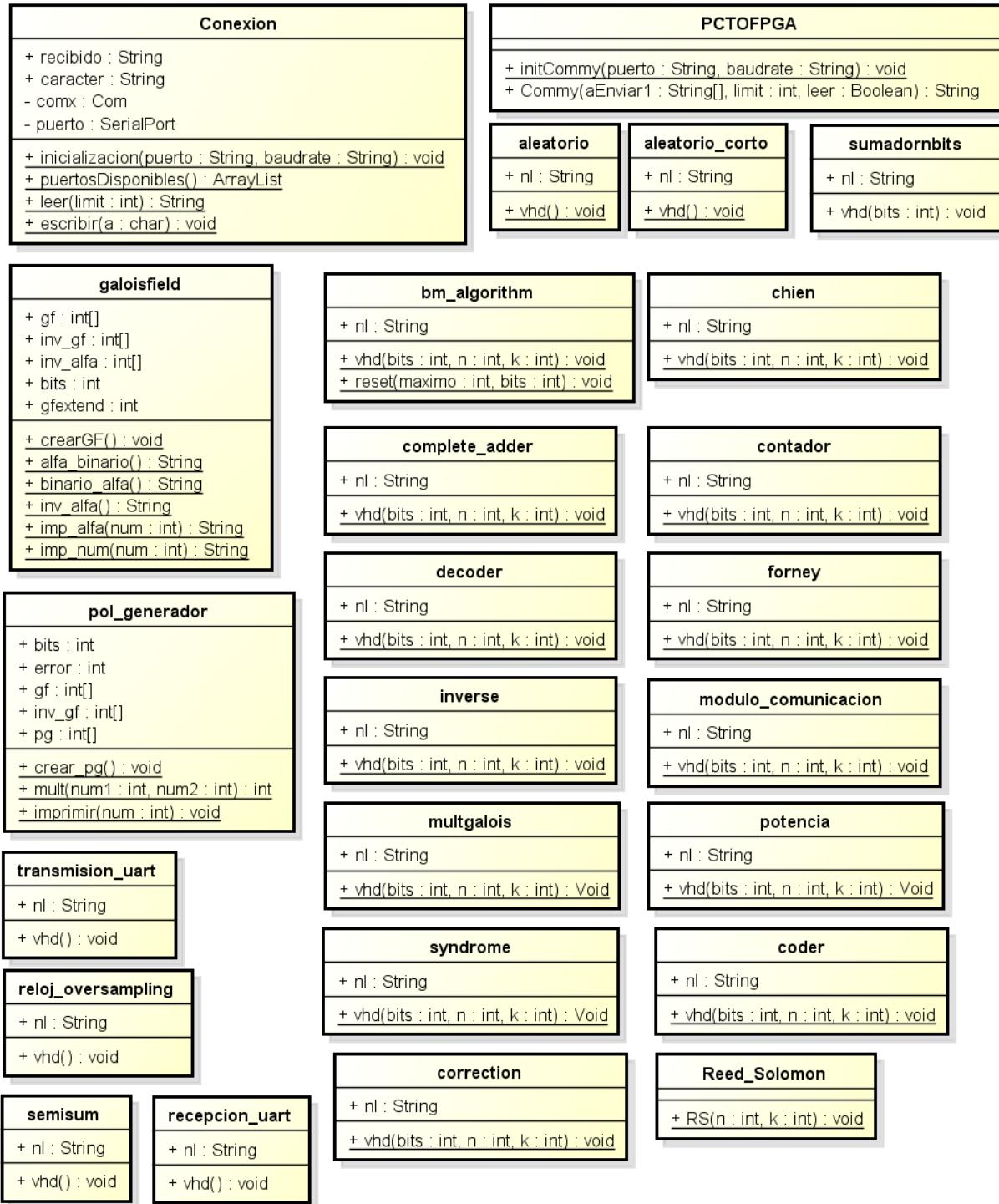


Figura. 6.9 Diagrama de clases del software

## 6.4. Calculadora de campos finitos

Se desarrolla la calculadora para realizar las operaciones esenciales dentro de un campo de Galois las cuales son la suma y la multiplicación.

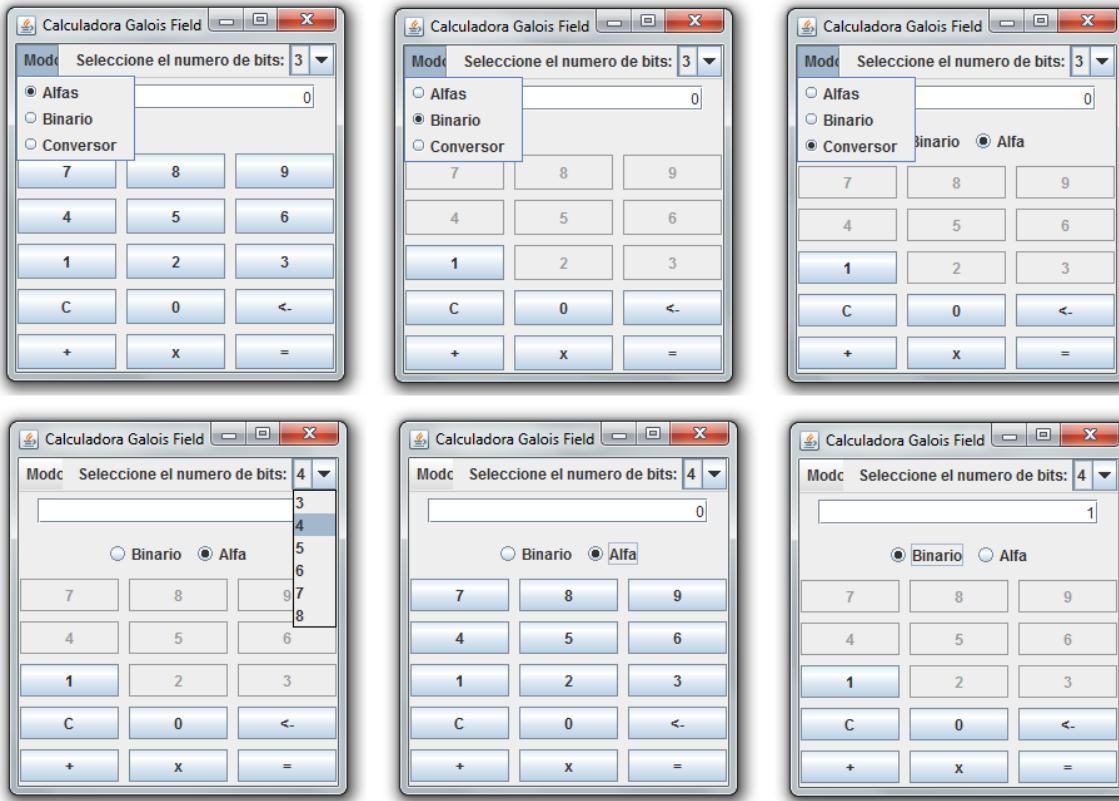


Figura. 6.10 Interfaz gráfica calculadora desarrollada

En la figura 6.10 se muestra la interfaz desarrollada<sup>11</sup>, se puede escoger el número de  $m$ , es decir el número de bits que tendrán los elementos del campo, por lo que la calculadora opera desde un campo  $GF(2^3)$  hasta un campo  $GF(2^8)$ . Además de esto se tienen tres modos de operación:

- *Alfas*: En este modo se ingresa los símbolos a operar en forma de alfa.
- *Binario*: En este modo se ingresa los símbolos a operar a en su representación binaria.
- *Conversor*: En este modo se pueden realizar la conversión de símbolos de su representación binaria a la representación en alfa y viceversa.

<sup>11</sup> De la interface cabe destacar que no esta habilitado el modo de entrada por teclado, es decir la única forma de ingresar los datos son por medio de los botones creados, a la vez dependiendo del modo en que se encuentra la calculadora se habilitan o deshabilitan ciertos botones con el fin de que el usuario no ingrese datos que el algoritmo no sea capas de procesar. También se crearon botones para realizar borrado ( $<-$ ) y para limpiar los valores de las entradas (C).

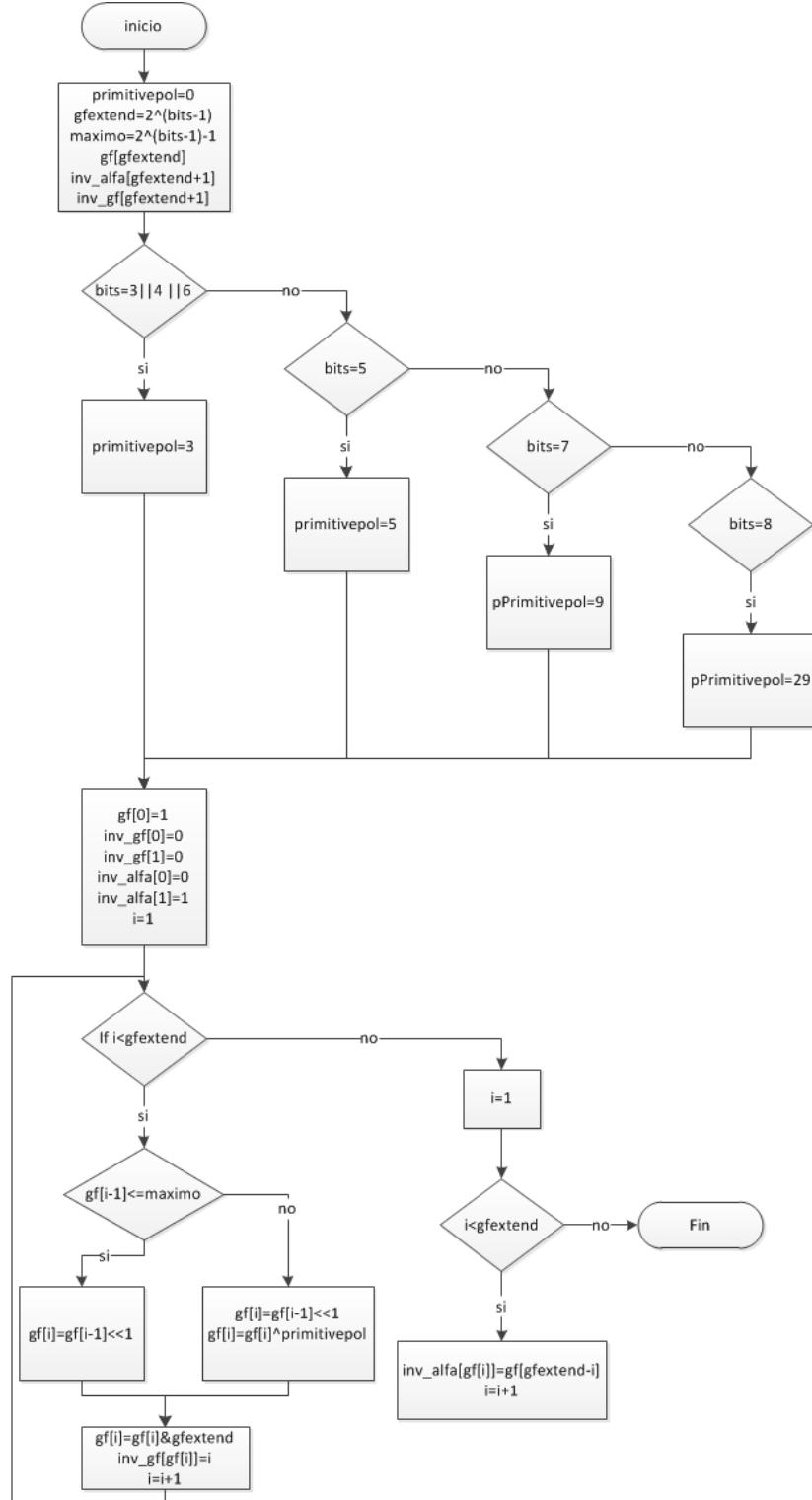


Figura. 6.11 Diagrama de flujo generador campo de Galois

El valor de  $m$  por defecto es 3, por lo que el software automáticamente genera un campo  $GF(2^3)$ , el cual se diseña a partir del diagrama de flujo mostrado en la figura 6.11.

La entrada del algoritmo es la captura del dato que tiene el *ComboBox*<sup>12</sup>, cuando se detecta un cambio en el valor del componente se corre nuevamente el algoritmo por lo que se puede cambiar el tamaño del campo de *Galois* sin ningún problema. Cabe anotar que el algoritmo crea 3 matrices *gf* donde se guardan los elementos del campo en binario y cuyo índice es el valor en alfa, *inv\_gf* donde se guarda los elementos del campo en alfa y cuyo índice es la representación binaria e *inv\_alfa* donde se guardan los valores del campo en binario y cuyo índice es el alfa del valor del inverso del símbolo. Para poder realizar que el campo de *Galois* sea reconfigurable es esencial la señal *primitivepol* creada para darle la flexibilidad necesaria al algoritmo quedando acorde a lo deseado.

Las operaciones tanto de suma como de multiplicación se hacen en principio con los símbolos en la representación de alfas, por lo que si la calculadora esta en modo *binario* los símbolos deben pasar por una función *convertir* la cual pasa el valor del símbolo en binario a su equivalente en *alfas* y se trata el problema como si la calculadora estuviera en modo *alfas*; de la misma manera al final se debe pasar el símbolo a su representación binaria por medio de la función *desconvertir*. La descripción de este proceso se representa por medio del diagrama de flujo mostrado en la figura 6.12, donde numero 1 y numero 2 son los valores que se leen por teclado siendo estos los que el usuario desea operar.

Del diagrama se puede comentar que la señal *opción* hace referencia al modo en el que se encuentra la calculadora, si estamos en modo *alfas* la señal es uno, si se esta en modo *binario* la señal es dos y finalmente si esta en modo *conversor* el valor de la señal es tres. Por otro lado el valor de la señal *oper* hace referencia a la operación que se desea hacer, si es una suma el valor de la señal toma el valor de uno y si es una multiplicación la señal se convierte en dos.

Para darle funcionalidad al modo *conversor* se hace uso de las funciones *convertir* y *desconvertir* anteriormente mencionadas, cuando se detecta que el evento del *RadioButton* cambio se define en que estado esta tomando el valor ingresado por el usuario y pasándolo por la función *conversor* si el estado del *RadioButton* paso de binario a alfa o por la función *desconversor* en caso contrario.

---

<sup>12</sup> Independiente del modo en que se encuentre la calculadora el dato que se captura es un decimal, por lo que si la calculadora se encuentra en modo *alfas* se lee la cadena de bits como si fuera un numero decimal.

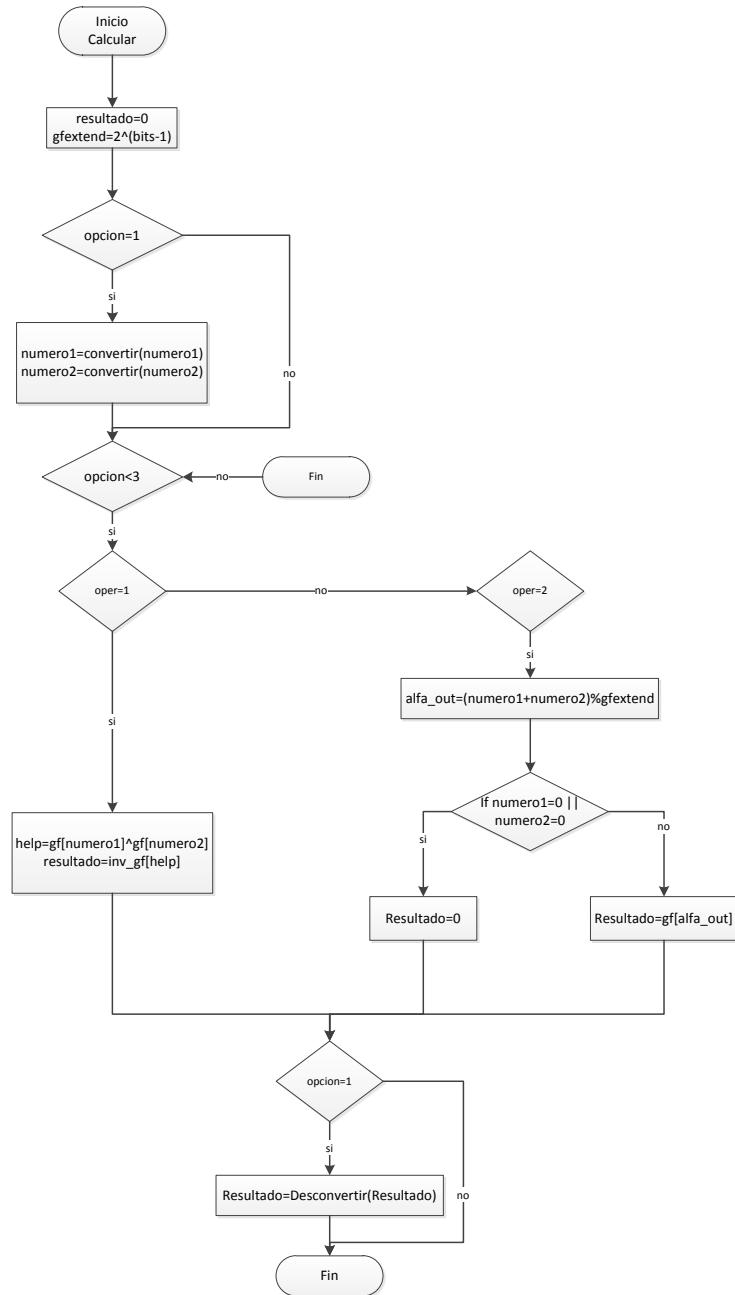


Figura. 6.12 Diagrama de flujo para realizar Multiplicación y suma en un campo de *Galois*.

Es importante mencionar que algoritmo parte del hecho que las variables de entrada *numero 1* y *numero2* pertenecen al campo de *Galois*<sup>13</sup>, además de que han sido capturadas por medio del *TextBox*.

<sup>13</sup> Para garantizar esto se crea una función denominada validación, la cual recibe un dato y verifica que ese dato sea un símbolo del campo de *Galois* sobre el que se esté realizando las operaciones en ese momento.

# 7

## Resultados y discusión

A partir del Software realizado se genera un código RS(15,9), para esto se crea una carpeta llamada “Código RS(15,9)”. Donde se copia el ejecutable del software como se ve en la figura 7.1.

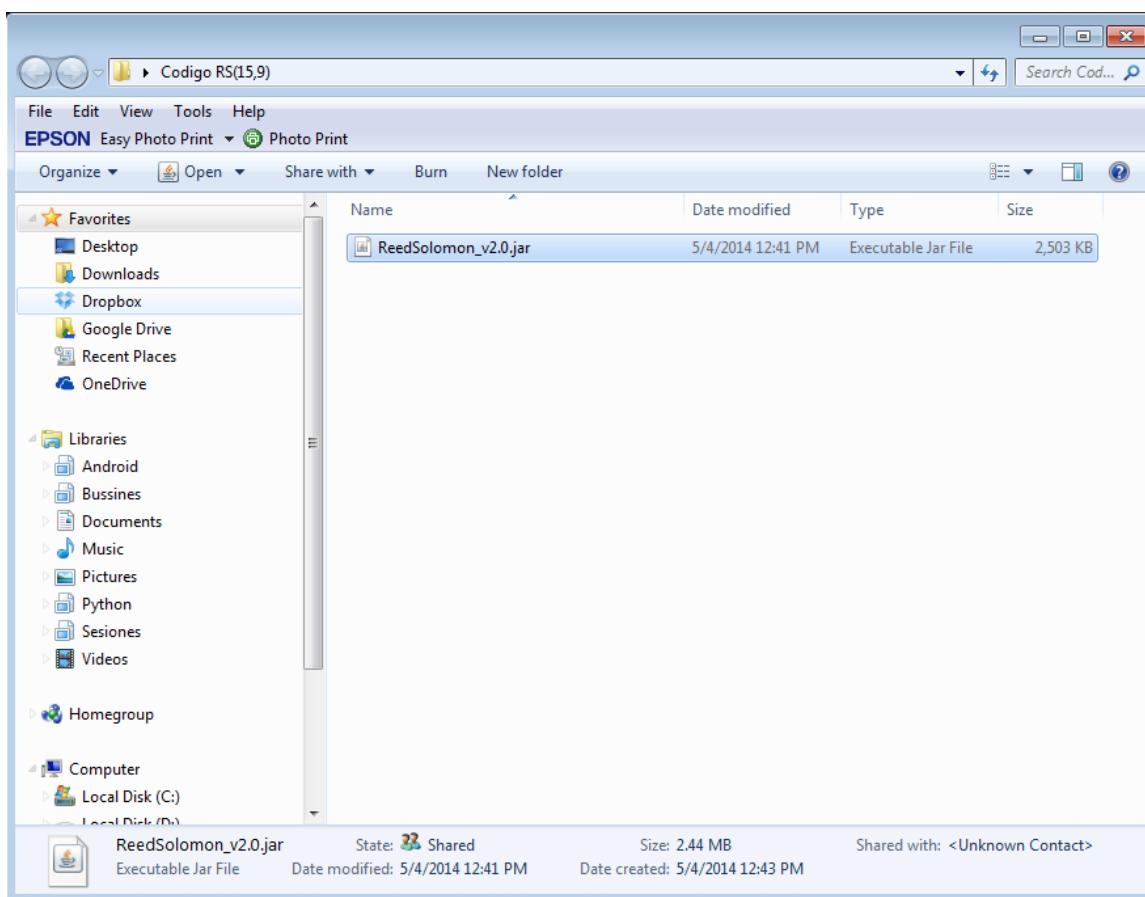
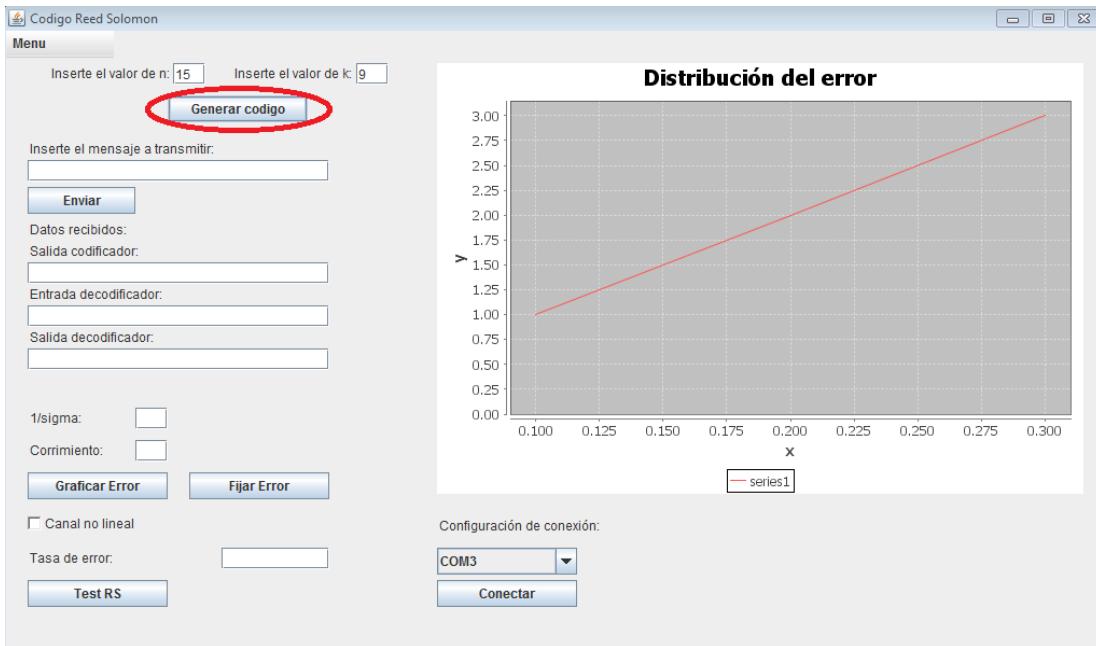
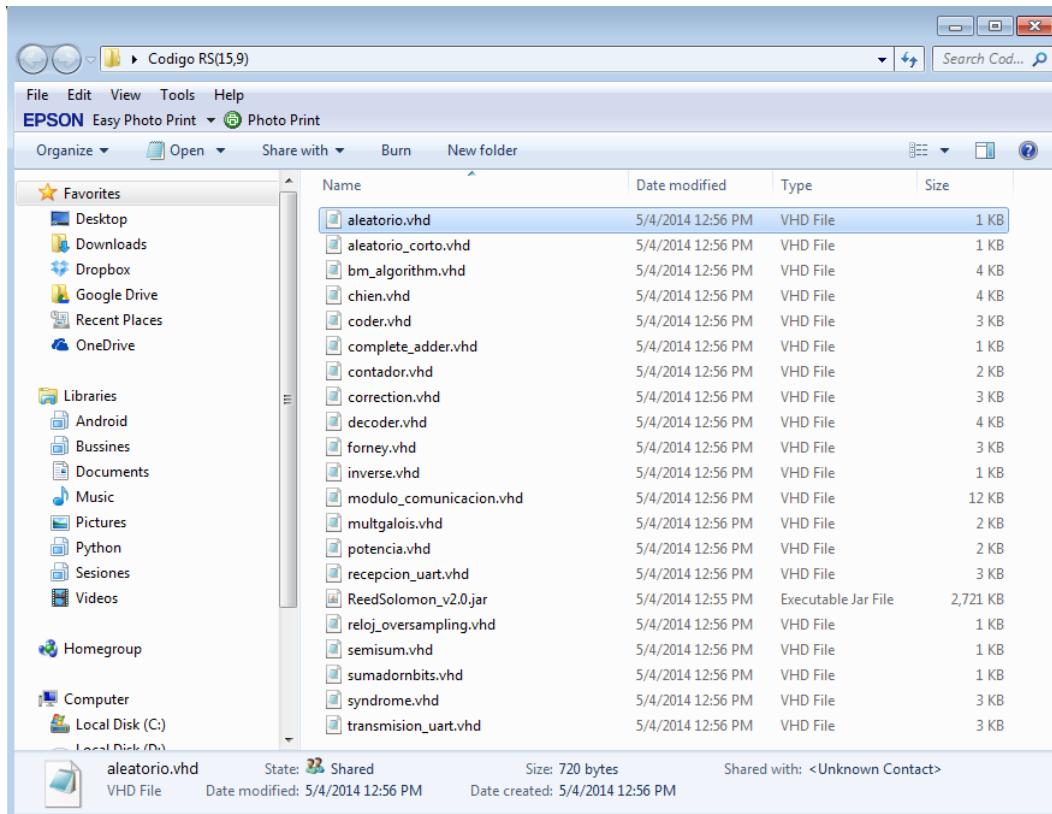


Figura. 7.1 Proceso de generación de archivos .vhf

Luego se ejecuta el programa y se crean los archivos .vhd que contienen las entidades VHDL para la prueba, como se ve en la figura. 7.2 y la figura 7.3.



**Figura. 7.2 Generando las entidades VHDL a partir del panel de control.**



**Figura. 7.3 Entidades VHDL generadas por el software y listas para simular.**

Se procede a validar via simulación cada uno de los bloques que compone el código, empezando por verificar el funcionamiento del codificador y luego el funcionamiento del decodificador.

## 7.1. Funcionamiento del codificador.

A partir del circuito generalizado de la figura 3.4 para ilustrar el funcionamiento del codificador, se define el circuito para un codificador RS(15,9) el cual se muestra en la figura 7.4 y se asocia a la entidad VHDL generada por el software, asociando las señales internas de la entidad al circuito propuesto.

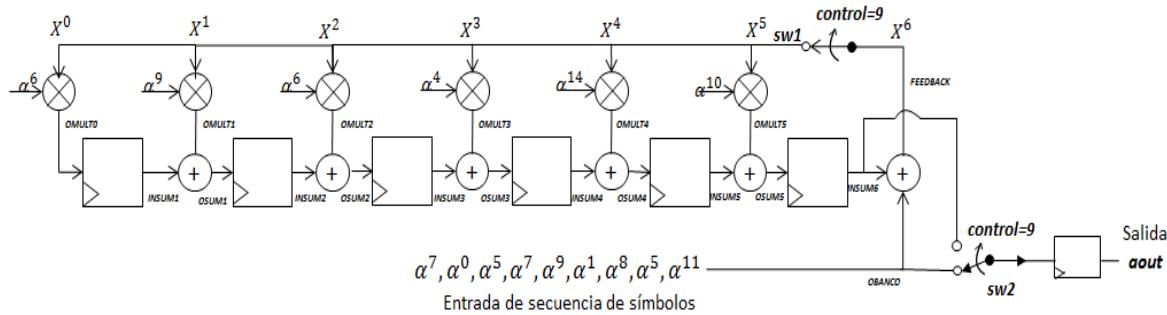


Figura. 4.4 Circuito del Codificador RS(15,9)

Se define un mensaje  $m(x)$ , el cual será transmitido por el codificador.

$$m(x) = \alpha^{11} X^8 + \alpha^5 X^7 + \alpha^8 X^6 + \alpha^1 X^5 + \alpha^9 X^4 + \alpha^7 X^3 + \alpha^5 X^2 + \alpha^0 X + \alpha^7 \quad (7.1)$$

$$m(x) = (1110)X^8 + (0110)X^7 + (0101)X^6 + (0010)X^5 + (1010)X^4 + (1011)X^3 + (0110)X^2 + (0001)X + (1011) \quad (7.2)$$

Del diagrama de bloques diseñado que representa el código en VHDL se realiza una prueba de escritorio incluyendo las señales más relevantes del circuito la cual es mostrada en la tabla 7.1. Cabe mencionar que por cada pulso de reloj va ingresando un coeficiente del mensaje para que sea procesado, es necesario que después de que entran todos los coeficientes al circuito la señal *obanco* se haga cero para no tener errores en el codificador; esto se controla por media de la señal *control* y se puede observar con más detalle en el código VHDL que el software genera para la entidad codificador. Las operaciones de multiplicación y suma tiene que realizarse claramente bajo el  $GF(2^4)$ , además de esto es importante resaltar que para hacer funcional el circuito es decir funcione cíclicamente y pueda recibir más mensajes, cuando *control* llega a catorce se deben reiniciar todos los registros que hacen parte del proceso incluyendo el mismo control. De esta manera el circuito queda preparado para recibir un nuevo mensaje y procesarlo, sin embargo no debemos olvidar que entre cada mensaje a procesar por

el codificador debe haber un tiempo de espera de seis símbolos, intervalo que corresponde a los símbolos de redundancia que añade el codificador al mensaje.

<b>clk</b>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<b>control</b>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0
<b>obanco</b>	$\alpha^{11}$	$\alpha^5$	$\alpha^8$	$\alpha^1$	$\alpha^9$	$\alpha^7$	$\alpha^5$	$\alpha^0$	$\alpha^7$	0	0	0	0	0	0	0
<b>sw2</b>	$\alpha^{11}$	$\alpha^5$	$\alpha^8$	$\alpha^1$	$\alpha^9$	$\alpha^7$	$\alpha^5$	$\alpha^0$	$\alpha^7$	$\alpha^{11}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{12}$	$\alpha^0$	0
<b>sw1</b>	$\alpha^{11}$	$\alpha^9$	$\alpha^0$	$\alpha^0$	$\alpha^{13}$	$\alpha^4$	$\alpha^4$	$\alpha^4$	$\alpha^9$	0	0	0	0	0	0	0
<b>omult0</b>	$\alpha^2$	$\alpha^0$	$\alpha^6$	$\alpha^6$	$\alpha^4$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^0$	0	0	0	0	0	0	0
<b>omult1</b>	$\alpha^5$	$\alpha^3$	$\alpha^9$	$\alpha^9$	$\alpha^7$	$\alpha^{13}$	$\alpha^{13}$	$\alpha^{13}$	$\alpha^5$	0	0	0	0	0	0	0
<b>omult2</b>	$\alpha^2$	$\alpha^0$	$\alpha^6$	$\alpha^6$	$\alpha^4$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^0$	0	0	0	0	0	0	0
<b>omult3</b>	$\alpha^0$	$\alpha^{13}$	$\alpha^4$	$\alpha^4$	$\alpha^2$	$\alpha^8$	$\alpha^8$	$\alpha^8$	$\alpha^{13}$	0	0	0	0	0	0	0
<b>omult4</b>	$\alpha^{10}$	$\alpha^8$	$\alpha^{14}$	$\alpha^{14}$	$\alpha^{12}$	$\alpha^3$	$\alpha^3$	$\alpha^3$	$\alpha^8$	0	0	0	0	0	0	0
<b>omult5</b>	$\alpha^6$	$\alpha^4$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^8$	$\alpha^{14}$	$\alpha^{14}$	$\alpha^{14}$	$\alpha^4$	0	0	0	0	0	0	0
<b>insum1</b>	0	$\alpha^2$	$\alpha^0$	$\alpha^6$	$\alpha^6$	$\alpha^4$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^0$	0	0	0	0	0	0
<b>Insum2</b>	0	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^5$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^9$	$\alpha^9$	$\alpha^{12}$	$\alpha^0$	0	0	0	0	0
<b>insum3</b>	0	$\alpha^2$	$\alpha^{10}$	0	$\alpha^{10}$	$\alpha^8$	0	$\alpha^{14}$	$\alpha^{13}$	$\alpha^7$	$\alpha^{12}$	$\alpha^0$	0	0	0	0
<b>insum4</b>	0	$\alpha^0$	$\alpha^{14}$	$\alpha^2$	$\alpha^4$	$\alpha^4$	0	$\alpha^8$	$\alpha^6$	0	$\alpha^7$	$\alpha^{12}$	$\alpha^0$	0	0	0
<b>insum5</b>	0	$\alpha^{10}$	$\alpha^2$	$\alpha^0$	$\alpha^{13}$	$\alpha^6$	$\alpha^7$	$\alpha^5$	$\alpha^{13}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{12}$	$\alpha^0$	0	0
<b>Insum6</b>	0	$\alpha^6$	$\alpha^2$	$\alpha^4$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^8$	$\alpha^1$	$\alpha^0$	$\alpha^{11}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{12}$	$\alpha^0$	0
<b>osum1</b>	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^5$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^9$	$\alpha^9$	$\alpha^{12}$	$\alpha^0$	0	0	0	0	0	0
<b>osum2</b>	$\alpha^2$	$\alpha^{10}$	0	$\alpha^{10}$	$\alpha^8$	0	$\alpha^{14}$	$\alpha^{13}$	$\alpha^7$	$\alpha^{12}$	$\alpha^0$	0	0	0	0	0
<b>osum3</b>	$\alpha^0$	$\alpha^{14}$	$\alpha^2$	$\alpha^4$	$\alpha^4$	0	$\alpha^8$	$\alpha^6$	0	$\alpha^7$	$\alpha^{12}$	$\alpha^0$	0	0	0	0
<b>osum4</b>	$\alpha^{10}$	$\alpha^2$	$\alpha^0$	$\alpha^{13}$	$\alpha^6$	$\alpha^7$	$\alpha^5$	$\alpha^{13}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{12}$	$\alpha^0$	0	0	0
<b>osum5</b>	$\alpha^6$	$\alpha^2$	$\alpha^4$	$\alpha^{10}$	$\alpha^3$	$\alpha^8$	$\alpha^1$	$\alpha^0$	$\alpha^{11}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{12}$	$\alpha^0$	0	0
<b>feedback</b>	$\alpha^{11}$	$\alpha^9$	$\alpha^0$	$\alpha^0$	$\alpha^{13}$	$\alpha^4$	$\alpha^4$	$\alpha^4$	$\alpha^9$	$\alpha^{11}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{12}$	$\alpha^0$	0
<b>aout</b>	0	$\alpha^{11}$	$\alpha^5$	$\alpha^8$	$\alpha^1$	$\alpha^9$	$\alpha^7$	$\alpha^5$	$\alpha^0$	$\alpha^7$	$\alpha^{11}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{12}$	$\alpha^0$

Tabla 7.1. Prueba de escritorio Codificador RS(7,3)

La salida del codificador se muestra por medio de la señal *aout* siendo la entrada *ain*, como se puede observar en la prueba de escritorio los primeros nueve símbolos de la salida son la misma entrada y los siguientes seis símbolos corresponden a la redundancia calculada por el circuito.

En la figura 7.5 se muestra la simulación<sup>14</sup> del funcionamiento del codificador, se muestran cada una de las señales de la entidad para entender con más claridad el funcionamiento del circuito. Al comparar con los resultados obtenidos en la prueba de escritorio se puede observar que cada señal tiene el valor que se esperaba en cada uno de los pulsos de reloj<sup>15</sup>. También se evidencia el retraso de un pulso de reloj que tiene la entrada del codificador con respecto a la

<sup>14</sup> Es importante mencionar que se simulo la entidad principal *módulo de comunicación* por lo que al realizar el archivo de simulación hay que tener en cuenta que la entrada se debe realizar bit por bit y se tiene que tener presente los tiempos de transmisión y recepción 781250 bps (95.367 kB/s) de cada dato para no tener ningún inconveniente. Sin embargo si se desea simular solo la entidad se puede realizar sin ningún inconveniente.

<sup>15</sup> El reloj que se utilizo es de 50 MHz por lo que la duración de cada ciclo de reloj es de 20 ns.

salida. Según la simulación y la prueba de escritorio el *codeword* que sale del codificador se representa por el polinomio descrito en la ecuación 7.1.

$$U(x) = \alpha^{11} X^{14} + \alpha^5 X^{13} + \alpha^8 X^{12} + \alpha^1 X^{11} + \alpha^9 X^{10} + \alpha^7 X^9 + \alpha^5 X^8 + \alpha^0 X^7 + \alpha^7 X^6 + \alpha^{11} X^5 + \alpha^{14} X^4 + 0X^3 + \alpha^7 X^2 + \alpha^{12} X + \alpha^0 \quad (7.3)$$

$$U(x) = (1110)X^{14} + (0110)X^{13} + (0101)X^{12} + (0010)X^{11} + (1010)X^{10} + (1011)X^9 + (0110)X^8 + (0001)X^7 + (1011)X^6 + (1110)X^5 + (1001)X^4 + (0000)X^3 + (1011)X^2 + (1111)X + (0001) \quad (7.4)$$

Finalmente podemos observar que todos los registros se resetean al terminar el circuito la codificación del primer mensaje a la espera de uno nuevo, de hecho se puede ver que al terminar la codificación de la primera palabra, se reinicia el circuito y comienza a decodificar el segundo mensaje siendo todos los coeficientes de este  $\alpha^7$ .

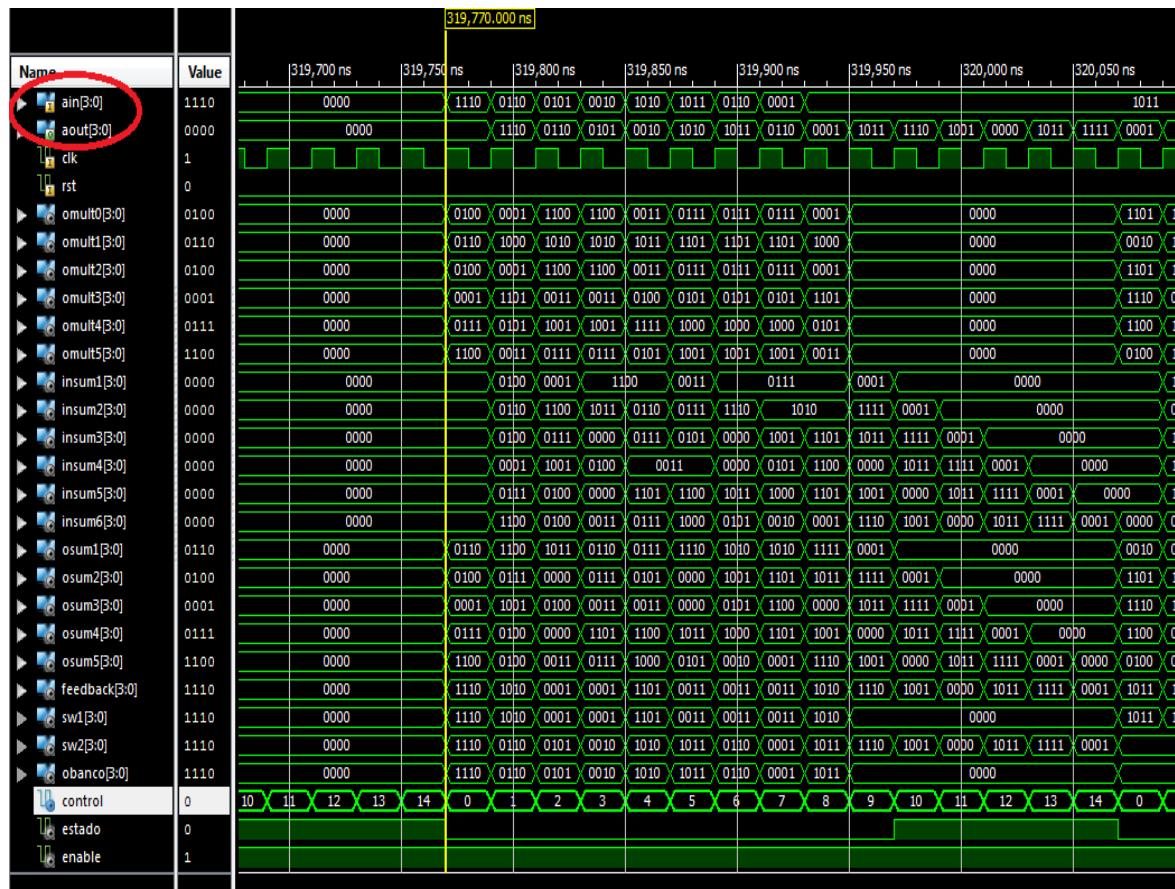


Figura. 7.5 Simulación del funcionamiento del codificador

## 7.2 Funcionamiento del decodificador.

---

A partir del arreglo de probabilidades que representa la distribución del ruido podemos manipularlo para corromper algunos símbolos del *codeword* que viene del codificador, para este caso en específico se manipulará el arreglo de tal forma que simularemos un canal que agregue el ruido mostrado en la ecuación 7.5.

$$e(x) = (0001)X^{14} + (0000)X^{13} + (0000)X^{12} + (0000)X^{11}(0000)X^{10} + (0000)X^9 + (0000)X^8 + (0000)X^7 + (0001)X^6 + (0000)X^5(0000)X^4 + (0000)X^3 + (0000)X^2 + (0001)X + (0000) \quad (7.5)$$

Partiendo de lo anterior el *codeword* que llega a la entrada del decodificar se puede describir a partir de la ecuación 7.6.

$$r(x) = U(x) + e(x) \quad (7.6)$$

$$\begin{aligned} r(x) = & (1110)X^{14} + (0110)X^{13} + (0101)X^{12} + (0010)X^{11}(1010)X^{10} + (1011)X^9 + (0110)X^8 + \\ & (0001)X^7 + (1011)X^6 + (1110)X^5(1001)X^4 + (0000)X^3 + (1011)X^2 + (1111)X + (0001) + \\ & (0001)X^{14} + (0000)X^{13} + (0000)X^{12} + (0000)X^{11}(0000)X^{10} + (0000)X^9 + (0000)X^8 + \\ & (0000)X^7 + (0001)X^6 + (0000)X^5(0000)X^4 + (0000)X^3 + (0000)X^2 + (0001)X + (0000) \end{aligned} \quad (7.7)$$

$$r(x) = (1111)X^{14} + (0110)X^{13} + (0101)X^{12} + (0010)X^{11}(1010)X^{10} + (1011)X^9 + (0110)X^8 + (0001)X^7 + (1010)X^6 + (1110)X^5(1001)X^4 + (0000)X^3 + (1011)X^2 + (1110)X + (0001) \quad (7.8)$$

$$r(x) = \alpha^{12} X^{14} + \alpha^5 X^{13} + \alpha^8 X^{12} + \alpha^1 X^{11} + \alpha^9 X^{10} + \alpha^7 X^9 + \alpha^5 X^8 + \alpha^0 X^7 + \alpha^{10} X^6 + \alpha^{11} X^5 + \alpha^{14} X^4 + 0X^3 + \alpha^7 X^2 + \alpha^{11} X + \alpha^0 \quad (7.9)$$

A partir del polinomio obtenido en la ecuación 7.9 se trabaja el decodificador para comprobar la funcionalidad del mismo observando que recupere el mensaje original que ingreso al codificador. Se analiza cada uno de los bloques del codificador por separado y así poder validar la importancia de cada uno de estos y el papel que cumplen dentro del codificador.

## 7.3. Funcionamiento del síndrome.

---

En la figura 7.6 se muestra el circuito implementado para generar la funcionalidad del Síndrome para un código RS(15,9), el cual es descrito por medio de la entidad VHDL que genera el software. La entrada del circuito es el polinomio  $r(x)$  descrito en las ecuaciones 7.8 y 7.9, y las salidas son los valores de los síndromes  $S_0, S_1, S_2, S_3, S_4$  y  $S_5$ ; para mantener un orden en el decodificador las salidas de todos los bloques que lo componen se cargan cuando este termina su trabajo, es decir por ejemplo para este caso los síndromes se cargan en la salida cuando pasan los 15 ciclos de reloj que se demora este bloque en hacer su tarea.

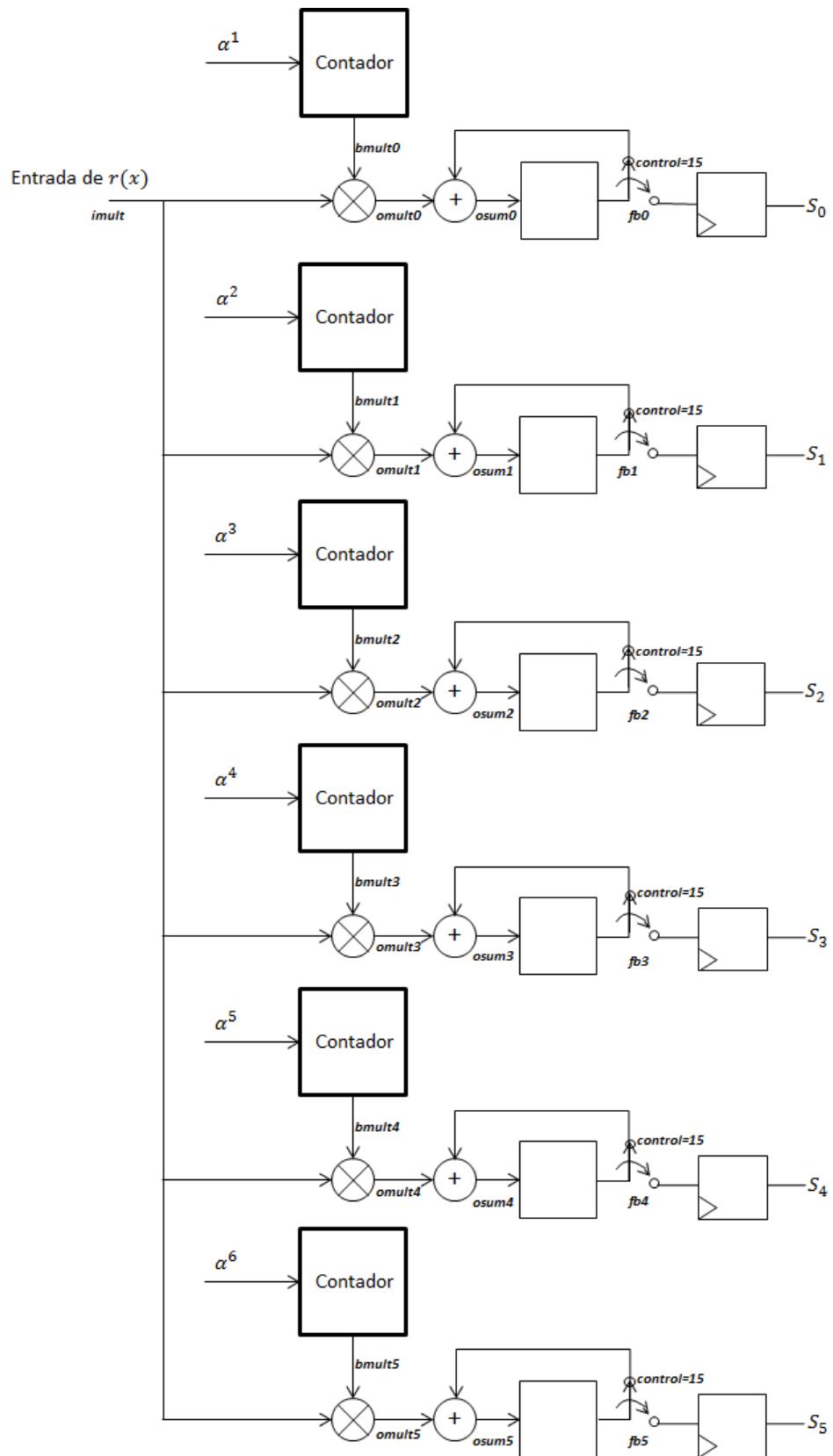


Figura. 7.6 Circuito de implementación del Síndrome

Del circuito de la figura 7.6, cabe destacar los conmutadores que permiten el paso de la información hacia los registros que cargan el valor de la salida, a partir del comportamiento de este circuito se desarrolló la prueba de escritorio que se presenta en la tabla 7.2.

<b>clk</b>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0
<b>control</b>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
<b>imult</b>	$\alpha^{12}$	$\alpha^5$	$\alpha^8$	$\alpha^1$	$\alpha^9$	$\alpha^7$	$\alpha^5$	$\alpha^0$	$\alpha^9$	$\alpha^{11}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{11}$	$\alpha^0$	0
<b>fb0</b>	0	$\alpha^{11}$	$\alpha^5$	0	$\alpha^{12}$	$\alpha^6$	$\alpha^{11}$	$\alpha^4$	$\alpha^3$	$\alpha^{14}$	$\alpha^7$	$\alpha^4$	$\alpha^4$	$\alpha^{14}$	$\alpha^5$	0
<b>fb1</b>	0	$\alpha^{10}$	$\alpha^8$	$\alpha^0$	$\alpha^2$	$\alpha^{13}$	$\alpha^9$	$\alpha^5$	$\alpha^{12}$	$\alpha^4$	$\alpha^{12}$	$\alpha^2$	$\alpha^2$	$\alpha^9$	$\alpha^{10}$	0
<b>fb2</b>	0	$\alpha^9$	$\alpha^4$	$\alpha^9$	$\alpha^{14}$	$\alpha^4$	0	$\alpha^{14}$	$\alpha^8$	$\alpha^9$	$\alpha^2$	$\alpha^9$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	0
<b>fb3</b>	0	$\alpha^8$	$\alpha^9$	$\alpha^2$	$\alpha^8$	$\alpha^5$	$\alpha^7$	0	$\alpha^{13}$	$\alpha^8$	$\alpha^{10}$	$\alpha^5$	$\alpha^5$	$\alpha^{10}$	$\alpha^5$	0
<b>fb4</b>	0	$\alpha^7$	$\alpha^6$	$\alpha^{14}$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^8$	$\alpha^2$	$\alpha^1$	$\alpha^3$	$\alpha^2$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^4$	$\alpha^0$	0
<b>fb5</b>	0	$\alpha^6$	$\alpha^{14}$	$\alpha^{12}$	$\alpha^2$	$\alpha^{11}$	$\alpha^6$	$\alpha^{14}$	$\alpha^5$	$\alpha^{10}$	$\alpha^{14}$	$\alpha^6$	$\alpha^6$	$\alpha^{12}$	$\alpha^7$	0
<b>bmult0</b>	0	$\alpha^{13}$	$\alpha^{12}$	$\alpha^{11}$	$\alpha^{10}$	$\alpha^9$	$\alpha^8$	$\alpha^7$	$\alpha^6$	$\alpha^5$	$\alpha^4$	$\alpha^3$	$\alpha^2$	$\alpha^1$	$\alpha^0$	0
<b>bmult1</b>	0	$\alpha^{11}$	$\alpha^9$	$\alpha^7$	$\alpha^5$	$\alpha^3$	$\alpha^1$	$\alpha^{14}$	$\alpha^{12}$	$\alpha^{10}$	$\alpha^8$	$\alpha^6$	$\alpha^4$	$\alpha^2$	$\alpha^0$	0
<b>bmult2</b>	0	$\alpha^9$	$\alpha^6$	$\alpha^3$	$\alpha^0$	$\alpha^{12}$	$\alpha^9$	$\alpha^6$	$\alpha^3$	$\alpha^0$	$\alpha^{12}$	$\alpha^9$	$\alpha^6$	$\alpha^3$	$\alpha^0$	0
<b>bmult3</b>	0	$\alpha^7$	$\alpha^3$	$\alpha^{14}$	$\alpha^{10}$	$\alpha^6$	$\alpha^2$	$\alpha^{13}$	$\alpha^9$	$\alpha^5$	$\alpha^1$	$\alpha^{12}$	$\alpha^8$	$\alpha^4$	$\alpha^0$	0
<b>bmult4</b>	0	$\alpha^5$	$\alpha^0$	$\alpha^{10}$	$\alpha^5$	$\alpha^0$	0									
<b>bmult5</b>	0	$\alpha^3$	$\alpha^{12}$	$\alpha^6$	$\alpha^0$	$\alpha^9$	$\alpha^3$	$\alpha^{12}$	$\alpha^6$	$\alpha^0$	$\alpha^9$	$\alpha^3$	$\alpha^{12}$	$\alpha^6$	$\alpha^0$	0
<b>omult0</b>	$\alpha^{11}$	$\alpha^3$	$\alpha^5$	$\alpha^{12}$	$\alpha^4$	$\alpha^1$	$\alpha^{13}$	$\alpha^7$	$\alpha^0$	$\alpha^1$	$\alpha^3$	0	$\alpha^9$	$\alpha^{12}$	$\alpha^0$	0
<b>omult1</b>	$\alpha^{10}$	$\alpha^1$	$\alpha^2$	$\alpha^8$	$\alpha^{14}$	$\alpha^{10}$	$\alpha^6$	$\alpha^{14}$	$\alpha^6$	$\alpha^6$	$\alpha^7$	0	$\alpha^{11}$	$\alpha^{13}$	$\alpha^0$	0
<b>omult2</b>	$\alpha^9$	$\alpha^{14}$	$\alpha^{14}$	$\alpha^4$	$\alpha^9$	$\alpha^4$	$\alpha^{14}$	$\alpha^6$	$\alpha^{12}$	$\alpha^{11}$	$\alpha^{11}$	0	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	0
<b>omult3</b>	$\alpha^8$	$\alpha^{12}$	$\alpha^{11}$	$\alpha^0$	$\alpha^4$	$\alpha^{13}$	$\alpha^7$	$\alpha^{13}$	$\alpha^3$	$\alpha^1$	$\alpha^0$	0	$\alpha^0$	$\alpha^0$	$\alpha^0$	0
<b>omult4</b>	$\alpha^7$	$\alpha^{10}$	$\alpha^8$	$\alpha^{11}$	$\alpha^{14}$	$\alpha^7$	$\alpha^0$	$\alpha^{11}$	$\alpha^9$	$\alpha^6$	$\alpha^4$	0	$\alpha^2$	$\alpha^1$	$\alpha^0$	0
<b>osum0</b>	$\alpha^{11}$	$\alpha^5$	0	$\alpha^{12}$	$\alpha^6$	$\alpha^{11}$	$\alpha^4$	$\alpha^3$	$\alpha^{14}$	$\alpha^7$	$\alpha^4$	$\alpha^4$	$\alpha^{14}$	$\alpha^5$	$\alpha^{11}$	0
<b>osum1</b>	$\alpha^{10}$	$\alpha^8$	$\alpha^0$	$\alpha^2$	$\alpha^{13}$	$\alpha^9$	$\alpha^5$	$\alpha^{12}$	$\alpha^4$	$\alpha^{12}$	$\alpha^2$	$\alpha^2$	$\alpha^9$	$\alpha^{10}$	$\alpha^5$	0
<b>osum2</b>	$\alpha^9$	$\alpha^4$	$\alpha^9$	$\alpha^{14}$	$\alpha^4$	0	$\alpha^{14}$	$\alpha^8$	$\alpha^9$	$\alpha^2$	$\alpha^9$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	0
<b>osum3</b>	$\alpha^8$	$\alpha^9$	$\alpha^2$	$\alpha^8$	$\alpha^5$	$\alpha^6$	0	$\alpha^{13}$	$\alpha^8$	$\alpha^{10}$	$\alpha^5$	$\alpha^5$	$\alpha^{10}$	$\alpha^5$	$\alpha^{10}$	0
<b>osum4</b>	$\alpha^7$	$\alpha^6$	$\alpha^{14}$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^8$	$\alpha^2$	$\alpha^1$	$\alpha^3$	$\alpha^3$	$\alpha^2$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^4$	$\alpha^0$	0
<b>osum5</b>	$\alpha^6$	$\alpha^{14}$	$\alpha^{12}$	$\alpha^2$	$\alpha^{11}$	$\alpha^6$	$\alpha^{10}$	$\alpha^5$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^{14}$	$\alpha^6$	$\alpha^6$	$\alpha^{12}$	$\alpha^7$	0
<b>s0</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^{10}$	
<b>s1</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^5$	
<b>s2</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^{12}$	
<b>s3</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^{10}$	
<b>s4</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
<b>s5</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^9$	

Tabla 7.2 Prueba de escritorio Síndrome RS(15,9)

Se observa que el circuito funciona según lo requerido generando los valores de los síndromes al final de los 15 ciclos de reloj, quedando estos valores disponibles para ser usados por el Bloque que realiza el algoritmo de Berlekamp-Massey. Se simula la entidad que genera el software para observar que el código implementado este acorde con el circuito diseñado, cabe mencionar que en el circuito se colocaron todas las señales internas de la entidad para comprender de forma correcta la solución.

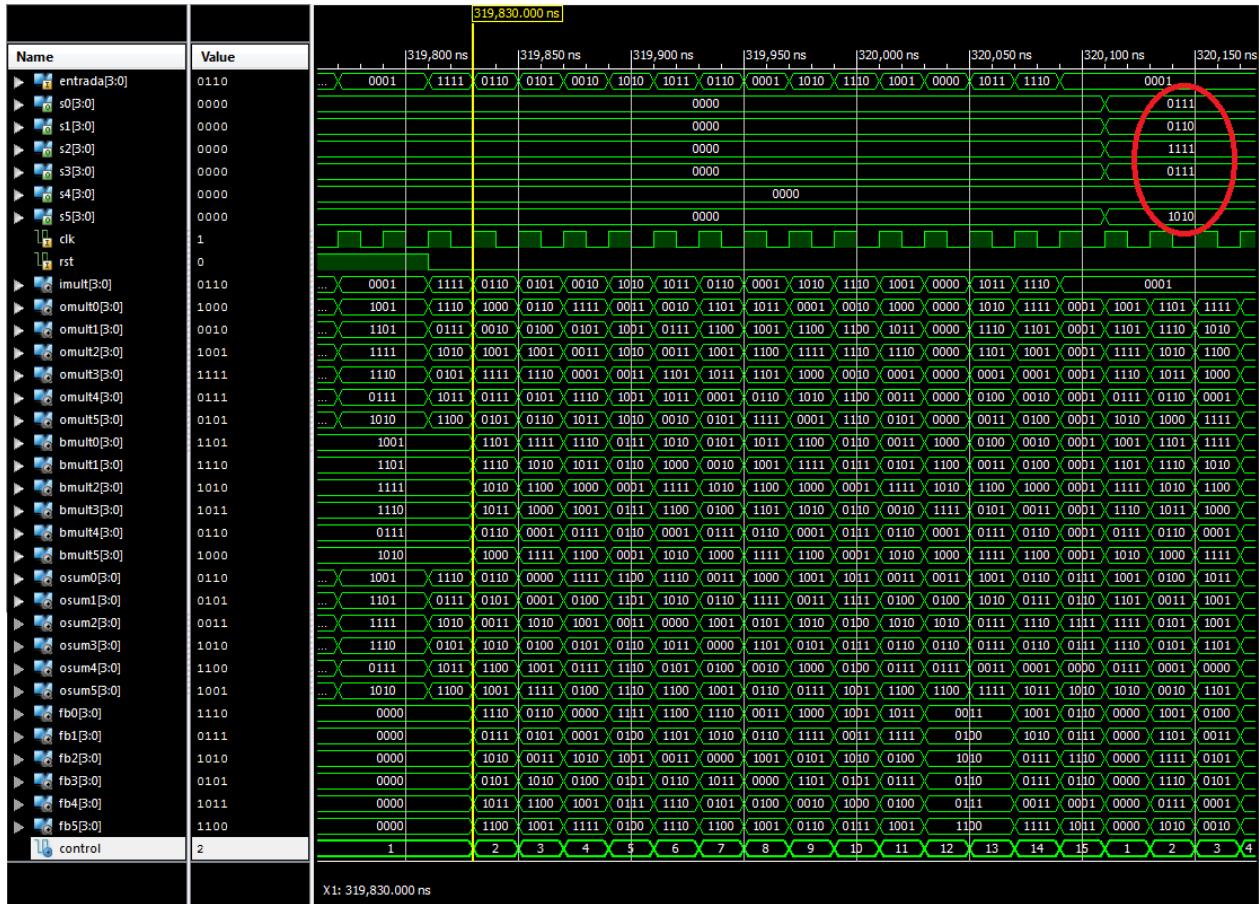


Figura. 7.7 Simulación del funcionamiento del Síndrome

En la figura 7.8 se presenta la simulación del funcionamiento de la entidad síndrome, la entrada del circuito es la señal *entrada* que representa el polinomio  $r(x)$ , la señal control se inicializa en 1, y cuando esta llega a 15 se resetean todos los parámetros del circuito cargando la salida de los síndromes. Se puede observar que los 6 síndromes corresponden a los que se calcularon por medio de la prueba de escritorio, estos valores duran cargados en el registro los siguientes 15 ciclos de reloj hasta que el circuito vuelve a generar nuevos valores, este tiempo es suficiente para que el circuito del *Berlekamp-Massey* pueda realizar su trabajo, recordando que las entradas de este circuito son los 6 síndromes calculados.

## 7.4. Funcionamiento del Berlekamp-Massey.

En las figuras 7.8 y 7.9 se presenta el circuito diseñado a partir del circuito generalizado presentado en las figuras de la 4.6 a la 4.12 para caracterizar el funcionamiento del algoritmo de *Berlekamp-Massey* de un código RS(15,9). Se observan todas las señales internas que componen en el circuito que son analizadas por medio de la prueba de escritorio presentada en la tabla 7.3, cabe mencionar que las entradas de este circuito corresponden a los valores

calculados en la entidad síndrome y la salida son los coeficientes del polinomio localizador del error  $c_1$ ,  $c_2$  y  $c_3$ <sup>16</sup>.

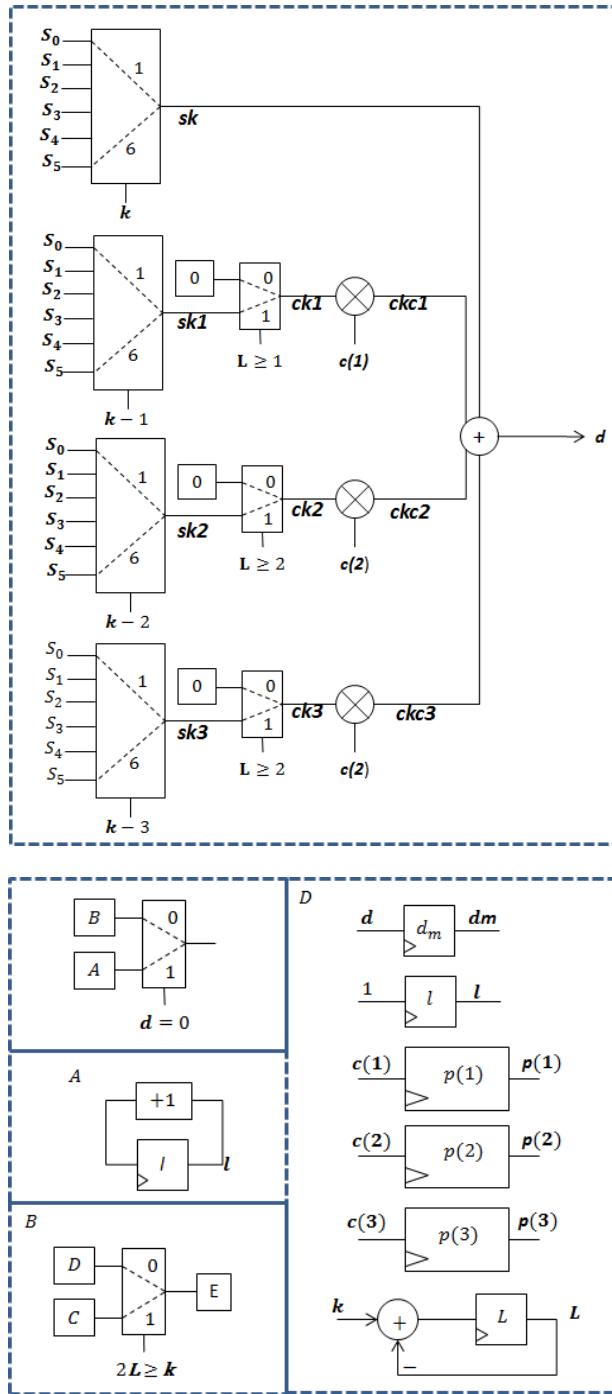


Figura. 7.8 Circuito del Algoritmo Berlekamp Massey – Parte A

<sup>16</sup> El valor de  $c_0$  no se calcula ya que este es siempre  $\alpha^0$  sin importar el código Reed Solomon que se esté realizando, sin embargo si se tiene en cuenta como constante y es utilizado en las siguientes entidades.

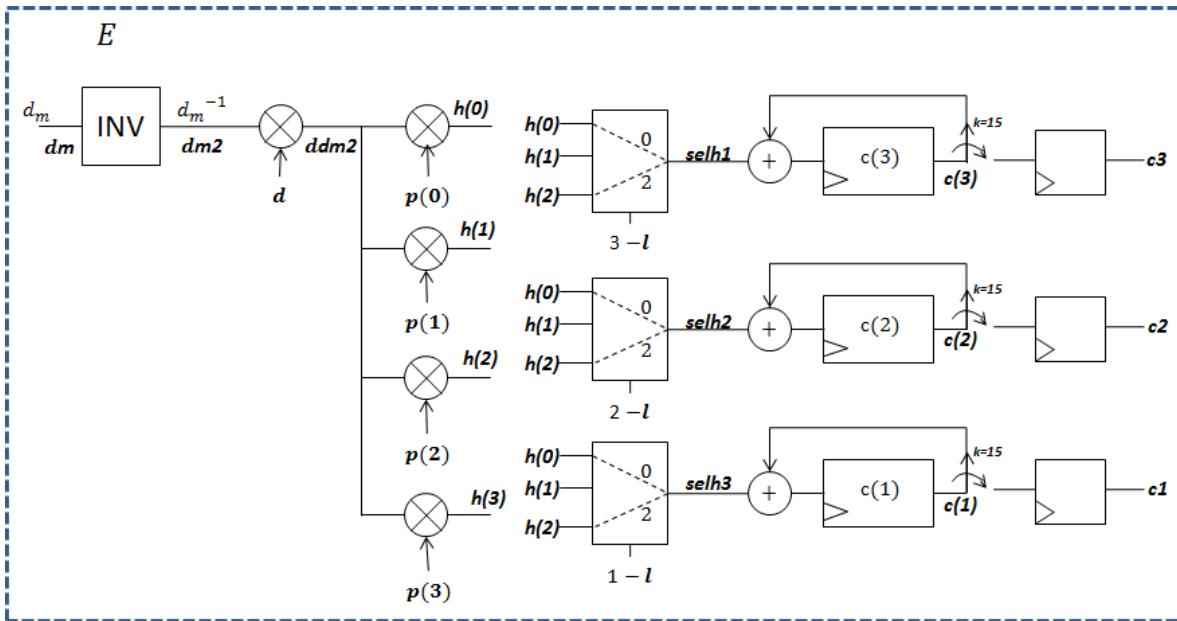


Figura. 7.9 Circuito del Algoritmo Berlekamp Massey – Parte B

<b>clk</b>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<b>k</b>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1
<b>I</b>	0	1	1	2	2	3	3	3	3	3	3	3	3	3	3	0
<b>c(3)</b>	0	0	0	0	0	$\alpha^6$	0									
<b>c(2)</b>	0	0	0	$\alpha^1$	$\alpha^1$	$\alpha^6$	0									
<b>c(1)</b>	0	$\alpha^{10}$	0													
<b>c(0)</b>	$\alpha^0$	0														
<b>p3</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>p2</b>	0	0	0	0	0	$\alpha^1$	0									
<b>p1</b>	0	0	0	$\alpha^{10}$	0											
<b>p0</b>	$\alpha^0$	0														
<b>h3</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>h2</b>	0	0	0	0	0	0	$\alpha^9$	$\alpha^9$	$\alpha^9$	0	0	0	0	0	0	0
<b>h1</b>	0	0	0	0	$\alpha^6$	0	$\alpha^3$	$\alpha^3$	$\alpha^3$	0	0	0	0	0	0	0
<b>h0</b>	$\alpha^{10}$	0	$\alpha^1$	0	$\alpha^{11}$	0	$\alpha^8$	$\alpha^8$	$\alpha^8$	0	0	0	0	0o	0o	0
<b>lp</b>	1	1	2	1	2	1	2	2	2	2	2	2	2	2	2	0
<b>dm</b>	$\alpha^0$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{11}$	$\alpha^7$	0									
<b>d</b>	$\alpha^{10}$	0	$\alpha^{11}$	0	$\alpha^7$	0	$\alpha^0$	0	0	0	0	0	0	0	0	0
<b>dm2</b>	$\alpha^0$	$\alpha^5$	$\alpha^5$	$\alpha^4$	$\alpha^4$	$\alpha^8$	0									
<b>ddm2</b>	$\alpha^{10}$	0	$\alpha^1$	0	$\alpha^{11}$	0	$\alpha^8$	0	0	0	0	0	0	0	0	0
<b>c3</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^6$
<b>c2</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^6$
<b>c1</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^{10}$

Tabla 7.3 Prueba de escritorio circuito del algoritmo Berlekamp Massey

De la prueba de escritorio y del circuito propuesto para el Berlekamp Massey de un RS(15,9) cabe mencionar que el valor con el cual se inicia la variable de control  $k$  es igual a 1, y como se mencionó anteriormente los valores de la salida se cargan cuando la variable de control es igual a 15, es decir cuando el circuito termino su trabajo por lo menos con esa palabra.

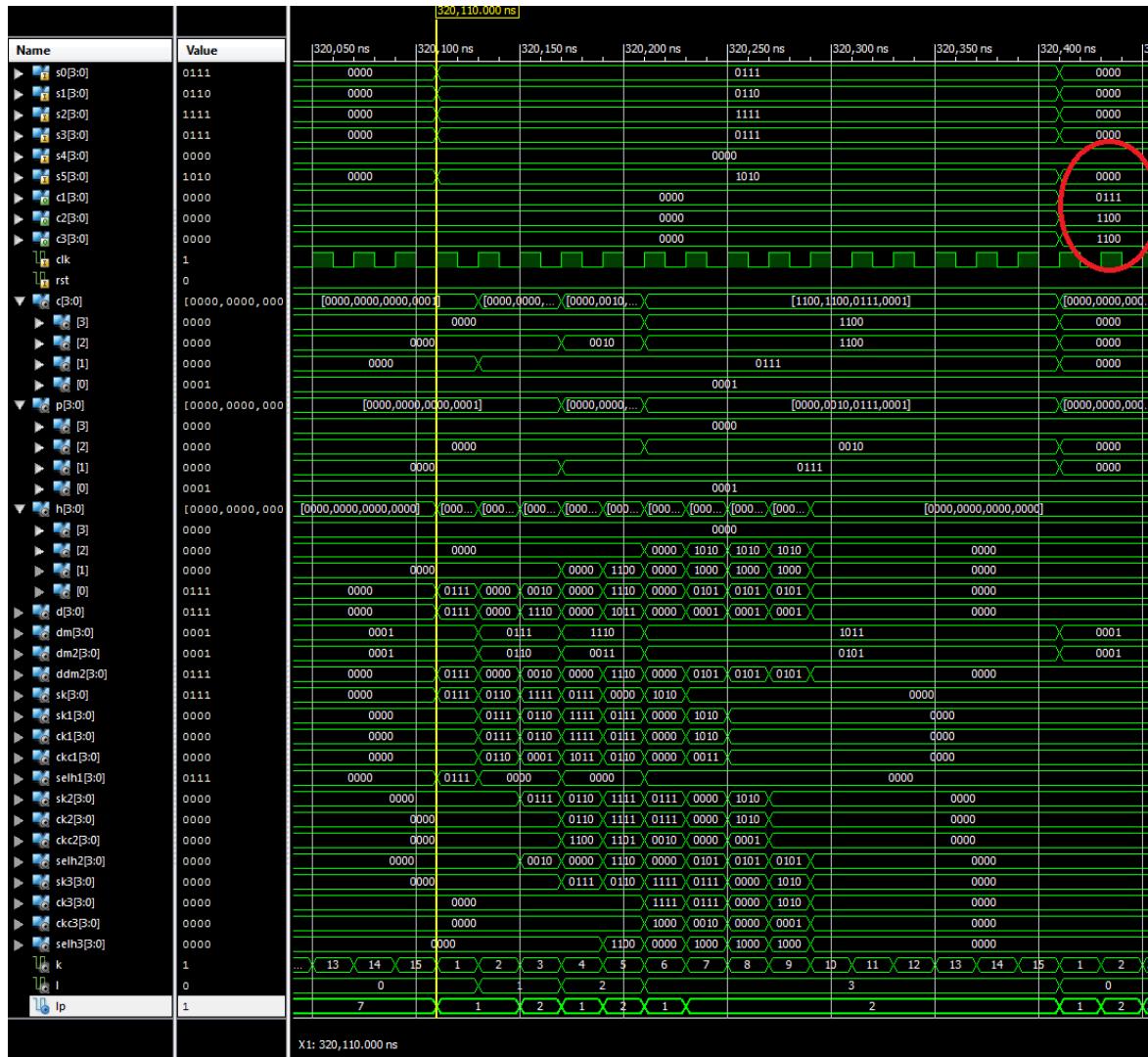


Figura. 7.10 Simulación del funcionamiento del *Berlekamp Massey*

En la figura 7.10 se puede observar la simulación del funcionamiento del algoritmo Berlekamp Massey de la entidad generada por el software desarrollado y que corresponde a la descripción de *hardware* del circuito propuesto. Los valores de  $p$ ,  $h$  y  $c$  fueron tomados como bancos de señales por practicidad a la hora del desarrollo del código. Se puede comparar los resultados obtenidos en la simulación con los de la prueba de escritorio y determinar que la entidad creada en VHDL cumple con su objetivo, generando los coeficientes del polinomio localizador del error  $c(x)$  en el número de ciclos de reloj requerido cargando las salidas en el momento necesario para que sean utilizadas por los circuitos del Chien y Forney.

De la simulación se puede determinar por tanto que el polinomio localizador del error para este *codeword* es el que se presenta en la ecuación 7.10.

$$c(x) = (1100)X^3 + (1100)X^2 + (0111)X + (0001) \quad (10)$$

$$c(x) = \alpha^6 X^3 + \alpha^6 X^2 + \alpha^{10} X + \alpha^0 \quad (11)$$

## 7.5 Funcionamiento del Chien.

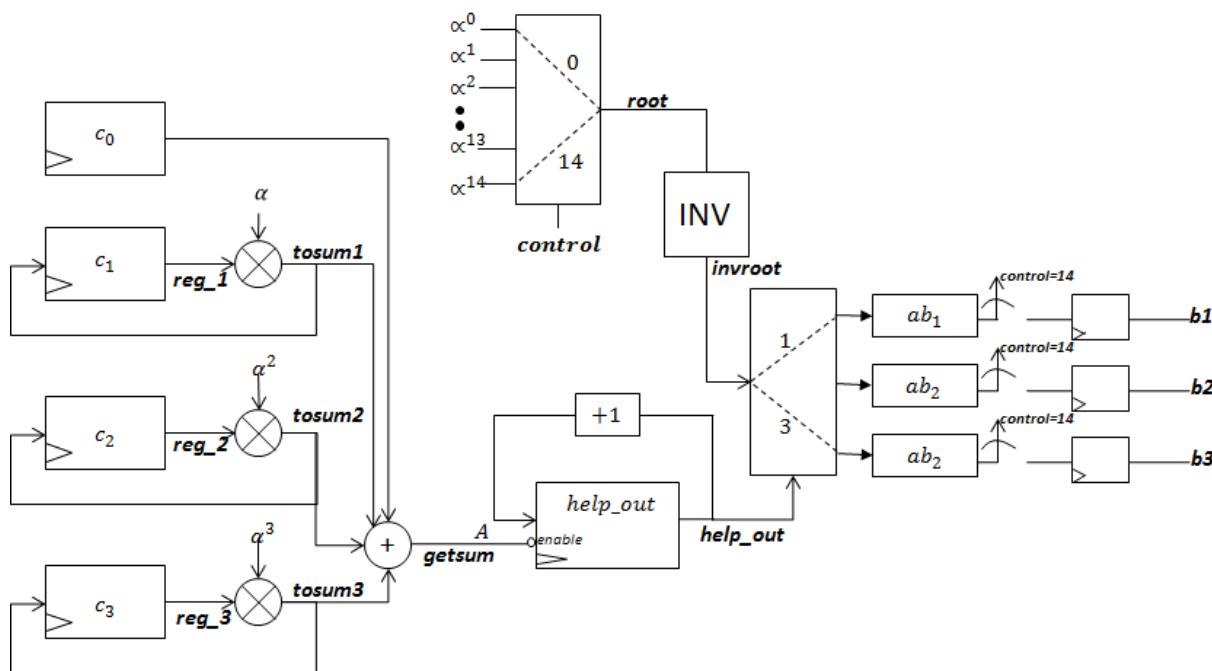


Figura. 7.11 Circuito del Algoritmo de Chien

En la figura 7.11 se presenta el circuito diseñado del algoritmo de Chien para un código RS(15,9) a partir de la generalización presentada en la figura 4.14, se añade la parte del circuito de la derecha para poder controlar que los valores de la salida del circuito se carguen solamente cuando este termina su trabajo es decir al cabo de 15 ciclos de reloj. Cabe recordar que el Chien es el encargado de encontrar las raíces del polinomio localizador del error, por lo que las entradas corresponden a los coeficientes de dicho polinomio excepto  $c_0$  que se toma como una constante interna dentro del circuito y las salidas son el valor de inversa de las raíces anteriormente mencionadas que corresponden al valor de la posición del error dentro del *codeword*, como el polinomio localizador es de grado 3 deben existir 3 raíces y por tanto esta entidad tiene 3 salidas.

<i>clk</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>control</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0
<i>reg_1</i>	$\alpha^{10}$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	0
<i>reg_2</i>	$\alpha^6$	$\alpha^6$	$\alpha^8$	$\alpha^{10}$	$\alpha^{12}$	$\alpha^{14}$	$\alpha^1$	$\alpha^3$	$\alpha^5$	$\alpha^7$	$\alpha^9$	$\alpha^{11}$	$\alpha^{13}$	$\alpha^0$	$\alpha^2$	0
<i>reg_3</i>	$\alpha^6$	$\alpha^6$	$\alpha^9$	$\alpha^{12}$	$\alpha^0$	$\alpha^3$	$\alpha^6$	$\alpha^9$	$\alpha^{12}$	$\alpha^0$	$\alpha^3$	$\alpha^6$	$\alpha^9$	$\alpha^{12}$	$\alpha^0$	0
<i>tosum1</i>	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	0
<i>tosum2</i>	$\alpha^6$	$\alpha^8$	$\alpha^{10}$	$\alpha^{12}$	$\alpha^{14}$	$\alpha^1$	$\alpha^3$	$\alpha^5$	$\alpha^7$	$\alpha^9$	$\alpha^{11}$	$\alpha^{13}$	$\alpha^0$	$\alpha^2$	$\alpha^4$	0
<i>tosum3</i>	$\alpha^6$	$\alpha^9$	$\alpha^{12}$	$\alpha^0$	$\alpha^3$	$\alpha^6$	$\alpha^9$	$\alpha^{12}$	$\alpha^0$	$\alpha^3$	$\alpha^6$	$\alpha^9$	$\alpha^{12}$	$\alpha^0$	$\alpha^3$	0
<i>getsum</i>	$\alpha^5$	0	$\alpha^5$	$\alpha^1$	$\alpha^{14}$	$\alpha^{11}$	$\alpha^0$	$\alpha^6$	$\alpha^4$	0	$\alpha^8$	$\alpha^9$	$\alpha^2$	$\alpha^0$	0	0
<i>root</i>	$\alpha^0$	$\alpha^1$	$\alpha^2$	$\alpha^3$	$\alpha^4$	$\alpha^5$	$\alpha^6$	$\alpha^7$	$\alpha^8$	$\alpha^9$	$\alpha^{10}$	$\alpha^{11}$	$\alpha^{12}$	$\alpha^{13}$	$\alpha^{14}$	0
<i>invroot</i>	$\alpha^{14}$	$\alpha^{13}$	$\alpha^{12}$	$\alpha^{11}$	$\alpha^{10}$	$\alpha^9$	$\alpha^8$	$\alpha^7$	$\alpha^6$	$\alpha^5$	$\alpha^4$	$\alpha^3$	$\alpha^2$	$\alpha^1$	$\alpha^0$	0
<i>ab1</i>	0	$\alpha^{14}$	0													
<i>ab2</i>	0	0	0	0	0	0	0	0	0	$\alpha^6$	$\alpha^6$	$\alpha^6$	$\alpha^6$	$\alpha^6$	$\alpha^6$	0
<i>ab3</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^1$	0	
<i>b1</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^{14}$	
<i>b2</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^6$	
<i>b3</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^1$	

Tabla 7.4 Prueba de escritorio circuito del algoritmo de Chien

En la figura 7.12 se muestran los resultados de la simulación de la entidad *Chien* generada por el software desarrollado y que representa el circuito propuesto en la figura 7.11, se puede observar que la variable *control* empieza en 0 y cuando llega a 14 los registros vuelven a sus valores iniciales con el fin de quedar preparado para procesar otra palabra.

Se observa en la simulación los valores de los coeficientes del polinomio localizador del error que llegan del Chien, así como el cálculo de los valores de las raíces de dicho polinomio después del

procesamiento que dura 15 ciclos de reloj correspondientes a la longitud de la palabra. Al comparar los valores obtenidos en la prueba de escritorio con los resultados de la simulación se puede decir que la descripción en VHDL desarrollada está acorde con el circuito propuesto logrando así las salidas esperadas correspondientes a las inversas de las raíces que representan la posición de los errores.



Figura. 7.12 Simulación del funcionamiento del Algoritmo de *Chien*

Si se recuerda la ecuación 7.9 se puede ver que los errores se presentan en las posiciones 14, 6 y 1 del *codeword*, que deben corresponder al valor de  $\alpha$  de cada una de las salidas del código *Chien* como efectivamente se puede observar al pasar las salidas binarias del circuito a su correspondiente valor en  $\alpha$  revisando la tabla 2.2.

## 7.6. Funcionamiento del Forney.

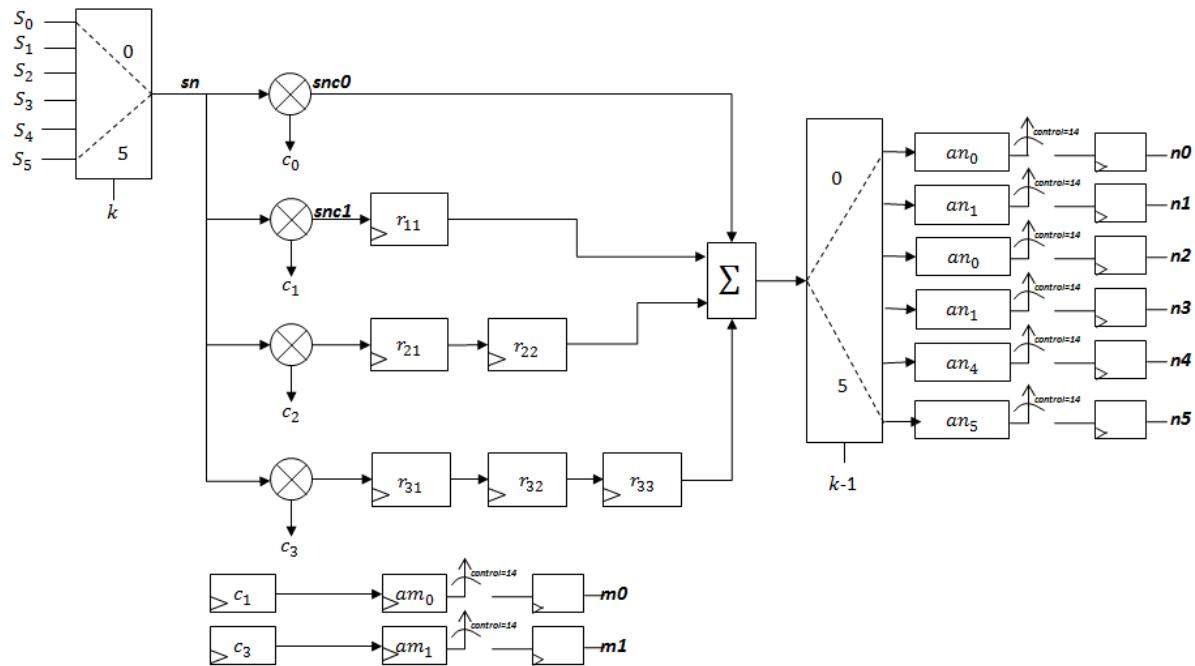


Figura. 7.13 Circuito del Algoritmo Forney

En la figura 7.13 se muestra el circuito propuesto para el desarrollo del algoritmo de *Forney*, producto del circuito generalizado mostrado en las figuras 4.15 y 4.16, cabe recordar que el *Forney* es el encargado de encontrar el polinomio del valor del error. En este diseño las salidas del circuito corresponden a los coeficientes del numerador  $n$  y los coeficientes del denominador  $m$  de dicho polinomio.

En la tabla 7.5 se puede observar la prueba de escritorio realizada para el circuito de la figura 7.3, la variable de control  $k$  se inicializa en 0, se verifica que el circuito tenga una duración de 15 ciclos de reloj para que se sincronice con el resto de los elementos del decodificador.

<b>Clik</b>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<b>K</b>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0
<b>sn</b>	$\alpha^{10}$	$\alpha^5$	$\alpha^{12}$	$\alpha^{10}$	0	$\alpha^9$	0	0	0	0	0	0	0	0	0	0
<b>snc0</b>	$\alpha^{10}$	$\alpha^5$	$\alpha^{12}$	$\alpha^{10}$	0	$\alpha^9$	0	0	0	0	0	0	0	0	0	0
<b>snc1</b>	$\alpha^5$	$\alpha^0$	$\alpha^7$	$\alpha^5$	0	$\alpha^4$	0	0	0	0	0	0	0	0	0	0
<b>snc2</b>	$\alpha^1$	$\alpha^{11}$	$\alpha^3$	$\alpha^1$	0	$\alpha^0$	0	0	0	0	0	0	0	0	0	0
<b>snc3</b>	$\alpha^1$	$\alpha^{11}$	$\alpha^3$	$\alpha^1$	0	$\alpha^0$	0	0	0	0	0	0	0	0	0	0
<b>r11</b>	0	$\alpha^5$	$\alpha^0$	$\alpha^7$	$\alpha^5$	0	$\alpha^4$	0	0	0	0	0	0	0	0	0
<b>r21</b>	0	$\alpha^1$	$\alpha^{11}$	$\alpha^3$	$\alpha^1$	0	$\alpha^0$	0	0	0	0	0	0	0	0	0
<b>r22</b>	0	0	$\alpha^1$	$\alpha^{11}$	$\alpha^3$	$\alpha^1$	0	$\alpha^0$	0	0	0	0	0	0	0	0
<b>r31</b>	0	$\alpha^1$	$\alpha^{11}$	$\alpha^3$	$\alpha^1$	0	$\alpha^0$	0	0	0	0	0	0	0	0	0
<b>r32</b>	0	0	$\alpha^1$	$\alpha^{11}$	$\alpha^3$	$\alpha^1$	0	$\alpha^0$	0	0	0	0	0	0	0	0

<b>r33</b>	0	0	0	$\alpha^1$	$\alpha^{11}$	$\alpha^3$	$\alpha^1$	0	$\alpha^0$	0	0	0	0	0	0
<b>n</b>	$\alpha^{10}$	0	$\alpha^6$	0	0	0	$\alpha^0$	$\alpha^0$	$\alpha^0$	0	0	0	0	0	0
<b>an0</b>	0	$\alpha^{10}$	0												
<b>an1</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>an2</b>	0	0	0	$\alpha^6$	0										
<b>an3</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>an4</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>an5</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>am0</b>	$\alpha^{10}$	0													
<b>am1</b>	$\alpha^6$	0													
<b>n0</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^{10}$
<b>n1</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>n2</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^6$
<b>n3</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>n4</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>n5</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<b>m0</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^{10}$
<b>m1</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	$\alpha^6$

Tabla 7.5 Prueba de escritorio circuito del Algoritmo de Forney.

En la figura 7.14 se muestra el resultado de la simulación del Algoritmo de Forney de la entidad que se crea por medio del software desarrollado, para que este circuito desarrolle su función correctamente es indispensable que los parámetros de entrada estén disponibles en el momento justo como se evidencia que está sucediendo en la simulación; de lo anterior se puede decir que en el momento que el Forney empiece a procesar determinada palabra, esta palabra tuvo que haber pasado ya por los circuitos del *Sindrome* y del *Berlekamp Massey* obteniendo los valores de los coeficientes del polinomio localizador del error así como de los síndromes.

Al comparar los resultados de la simulación con los de la prueba de escritorio, se observa que las salidas son las deseadas y en el momento deseado, a partir de estas podemos definir el polinomio del valor del error  $\Omega(x)$ .

$$\Omega(x) = \frac{\text{num}(x)}{\text{den}(x)} \quad (7.12)$$

$$\Omega(x) = \frac{n_5 X^5 + n_4 X^4 + n_3 X^3 + n_2 X^2 + n_1 X + n_0}{(1100)X + (0111)} \quad (7.13)$$

$$\Omega(x) = \frac{(0000)X^5 + (0000)X^4 + (0000)X^3 + (1100)X^2 + (0000)X + (0111)}{(1100)X + (0111)} \quad (7.14)$$

$$\Omega(x) = \frac{\alpha^6 X^2 + \alpha^{10}}{\alpha^6 X + \alpha^{10}} \quad (7.15)$$

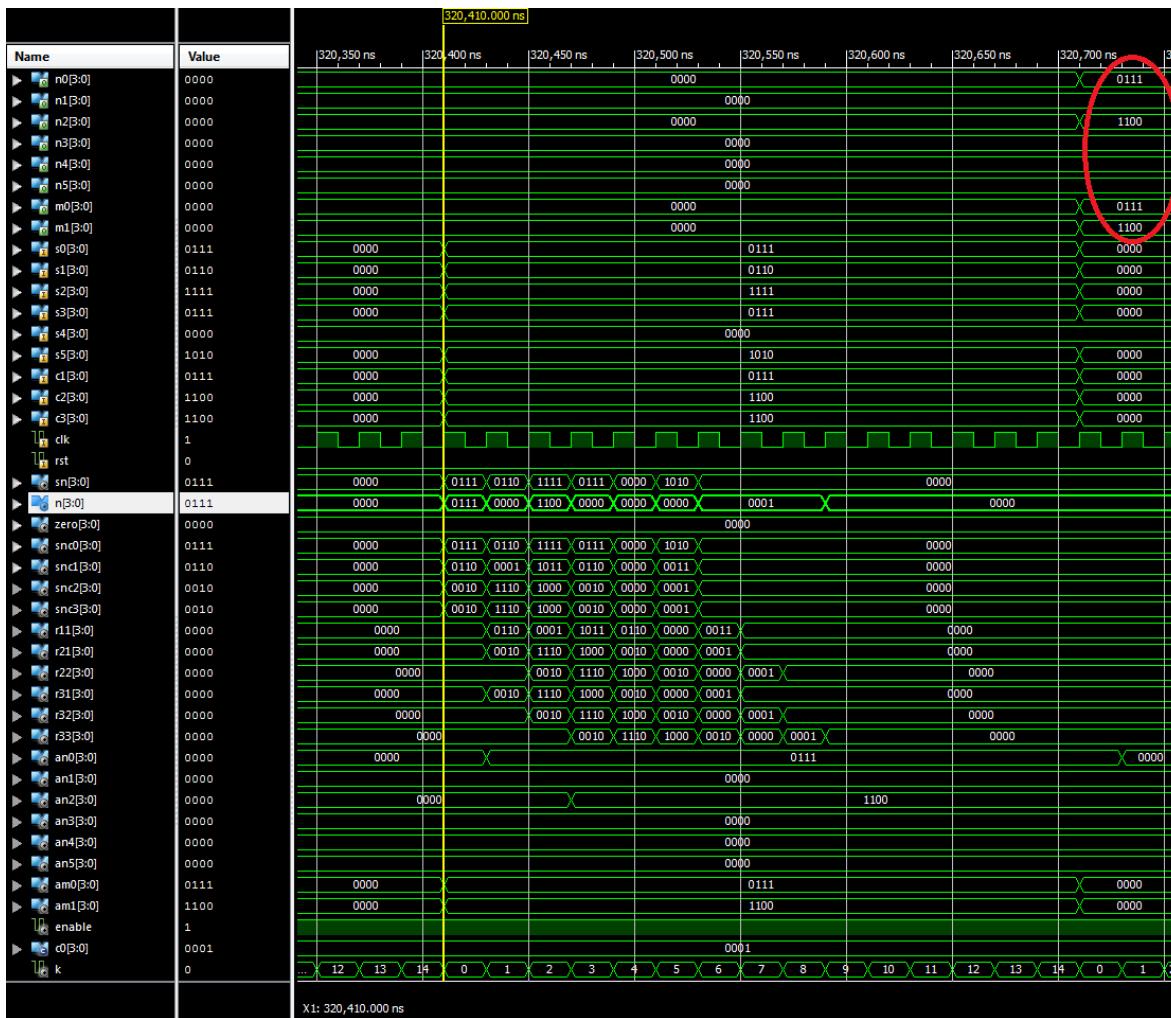


Figura. 7.14 Simulación del funcionamiento del Algoritmo de Forney

### 1.1 Funcionamiento del algoritmo de corrección

En la figura 7.15 se presenta el circuito propuesto para el algoritmo de corrección para un código RS(15,9), a partir de la generalización planteada en la figura 4.16. Se calcula el valor del error y cuando se encuentra que en determinada posición hay un error se cambia el símbolo de la salida.

En la tabla 7.6 se realiza la prueba de escritorio del algoritmo propuesto, a diferencia de los circuitos anteriores la señal de control se inicializa en 14 y va decrementando, cuando llega a 0 se restauran los valores de los registros por defecto para procesar una nueva palabra. A la vez desde el primer ciclo de reloj el circuito empieza a corregir y generar los símbolos corregidos, tarea que se demora 15 ciclos de reloj mientras procesa todo el codeword. En este circuito se evalúan las 3 raíces encontradas en el Chien en el polinomio del error obtenido en el Forney.

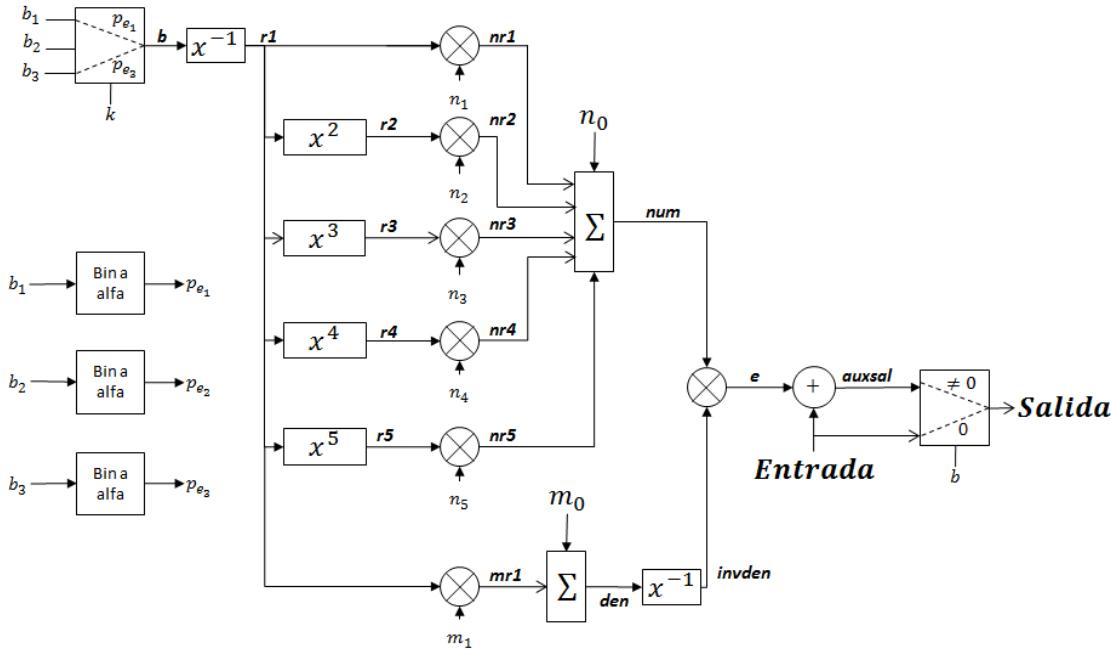


Figura. 7.15 Circuito del algoritmo de corrección.

<i>Clk</i>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
<i>K</i>	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>pe1</i>	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
<i>pe2</i>	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
<i>pe3</i>	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
<i>B</i>	$\alpha^{14}$	0	0	0	0	0	0	0	$\alpha^6$	0	0	0	0	$\alpha^1$	0
<i>r1</i>	$\alpha^1$	0	0	0	0	0	0	0	$\alpha^9$	0	0	0	0	$\alpha^{14}$	0
<i>r2</i>	$\alpha^2$	0	0	0	0	0	0	0	$\alpha^3$	0	0	0	0	$\alpha^{13}$	0
<i>r3</i>	$\alpha^3$	0	0	0	0	0	0	0	$\alpha^{12}$	0	0	0	0	$\alpha^{12}$	0
<i>r4</i>	$\alpha^4$	0	0	0	0	0	0	0	$\alpha^6$	0	0	0	0	$\alpha^{11}$	0
<i>r5</i>	$\alpha^5$	0	0	0	0	0	0	0	$\alpha^0$	0	0	0	0	$\alpha^{10}$	0
<i>nr1</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<i>nr2</i>	$\alpha^8$	0	0	0	0	0	0	0	$\alpha^9$	0	0	0	0	$\alpha^4$	0
<i>nr3</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<i>nr4</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<i>nr5</i>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
<i>Num</i>	$\alpha^1$	$\alpha^{10}$	$\alpha^{13}$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^2$	$\alpha^{10}$						
<i>m0</i>	$\alpha^{10}$														
<i>m1</i>	$\alpha^6$														
<i>Den</i>	$\alpha^1$	$\alpha^{10}$	$\alpha^{13}$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^{10}$	$\alpha^2$	$\alpha^{10}$						
<i>invden</i>	$\alpha^{14}$	$\alpha^5$	$\alpha^2$	$\alpha^5$	$\alpha^5$	$\alpha^5$	$\alpha^5$	$\alpha^{13}$	$\alpha^5$						
<i>E</i>	$\alpha^0$														
<i>entrada</i>	$\alpha^{12}$	$\alpha^5$	$\alpha^8$	$\alpha^1$	$\alpha^9$	$\alpha^7$	$\alpha^5$	$\alpha^0$	$\alpha^9$	$\alpha^{11}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{11}$	$\alpha^0$
<i>auxsal</i>	$\alpha^{11}$	$\alpha^5$	$\alpha^8$	$\alpha^1$	$\alpha^9$	$\alpha^9$	$\alpha^5$	$\alpha^0$	$\alpha^9$	$\alpha^{11}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{12}$	$\alpha^0$
<i>salida</i>	$\alpha^{12}$	$\alpha^5$	$\alpha^8$	$\alpha^1$	$\alpha^9$	$\alpha^7$	$\alpha^5$	$\alpha^0$	$\alpha^9$	$\alpha^{11}$	$\alpha^{14}$	0	$\alpha^7$	$\alpha^{11}$	$\alpha^0$

Tabla 7.6 Prueba de escritorio algoritmo de corrección

En la figura 7.16 se presenta los resultados de la simulación de la entidad generada por el software desarrollado que representa el circuito propuesto en la figura 7.15. Se comparan los resultados obtenidos con la prueba de escritorio y se evidencia que el objetivo se cumplió y el código corrigió los tres símbolos que se habían corrompido. A partir de esto se reconstruye el *codeword* enviado por el codificador y se recupera el mensaje emitido.

$$\hat{U}(x) = (1110)X^{14} + (0110)X^{13} + (0101)X^{12} + (0010)X^{11} + (1010)X^{10} + (1011)X^9 + (0110)X^8 + (0001)X^7 + (1011)X^6 + (1110)X^5 + (1001)X^4 + (0000)X^3 + (1011)X^2 + (1111)X + (0001) \quad (7.16)$$

$$\hat{m}(x) = \alpha^{11} X^8 + \alpha^5 X^7 + \alpha^8 X^6 + \alpha^1 X^5 + \alpha^9 X^4 + \alpha^7 X^3 + \alpha^5 X^2 + \alpha^0 X + \alpha^7 \quad (7.17)$$

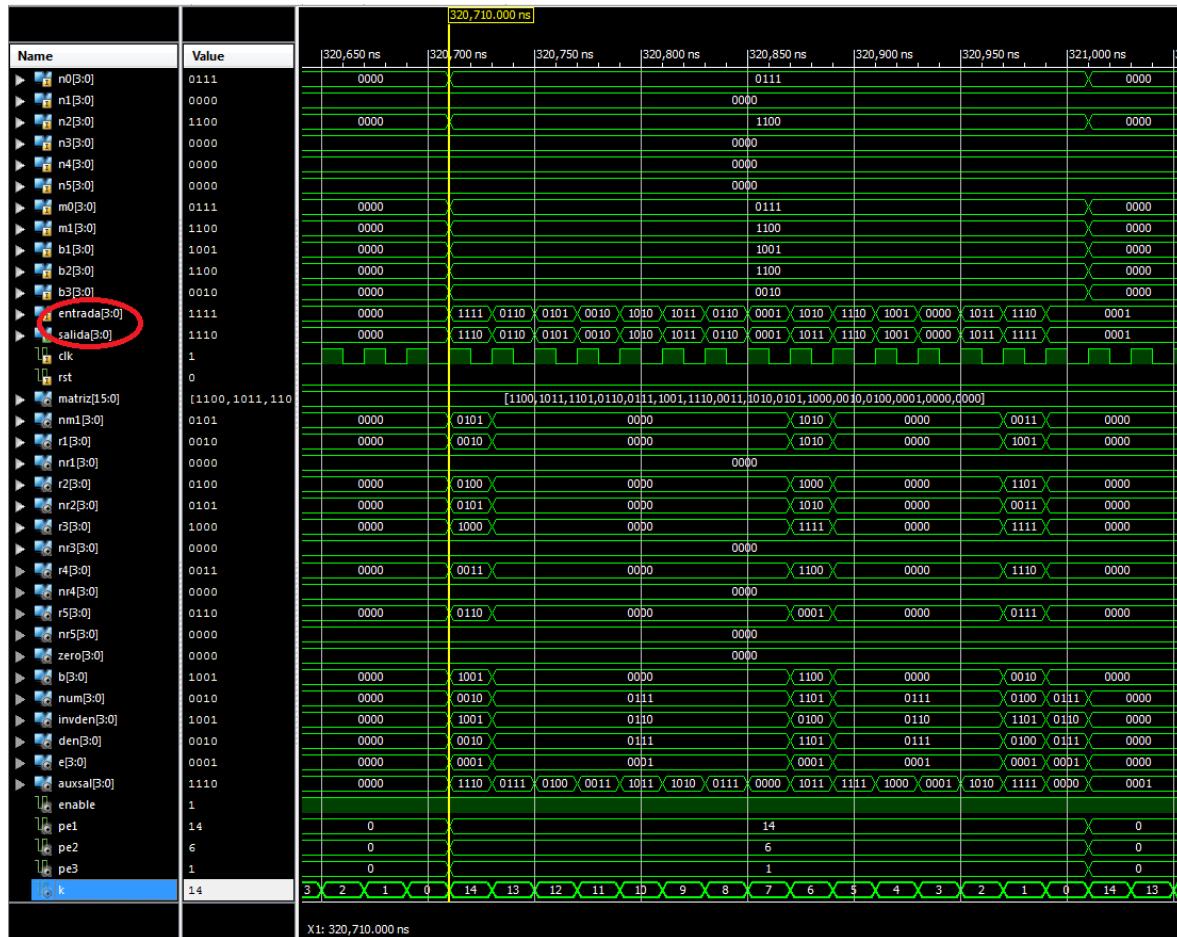


Figura. 7.16 Simulación del funcionamiento del algoritmo de corrección

## 7.7 Funcionamiento del decodificador.

Ahora observaremos el comportamiento al decodificar en general. En la figura 7.17 se encuentra el circuito propuesto para un código RS(15,9), se debe tener en cuenta que los síndromes necesitan estar disponibles para la entidad *Forney*, luego estos deben tener un

persistencia de 15 pulsos para asegurarnos que efectivamente puedan ser leídos por ambas entidades.

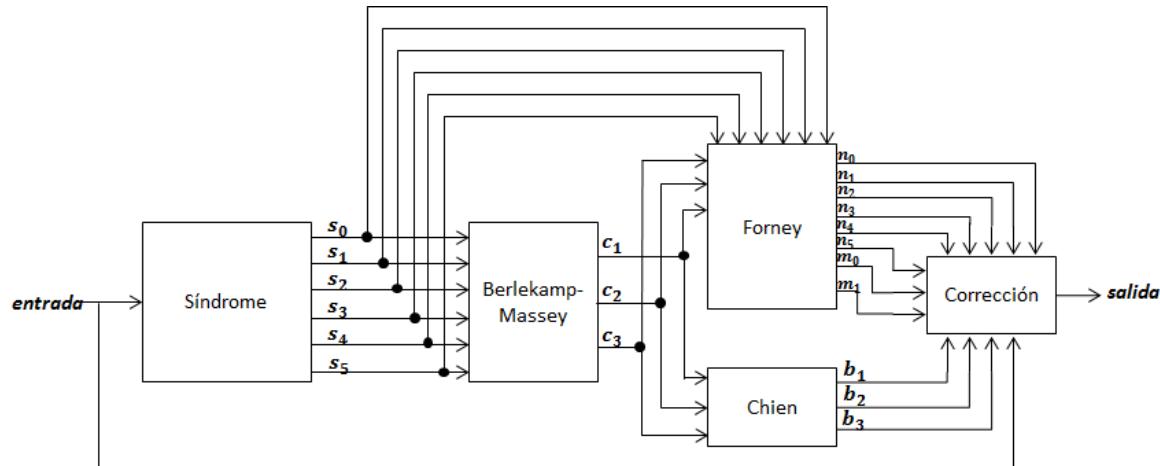


Figura. 7.17 Circuito Decodificador RS(15,9)

En la figura 7.18 se muestra la simulación del decodificador observando su correcto funcionamiento.

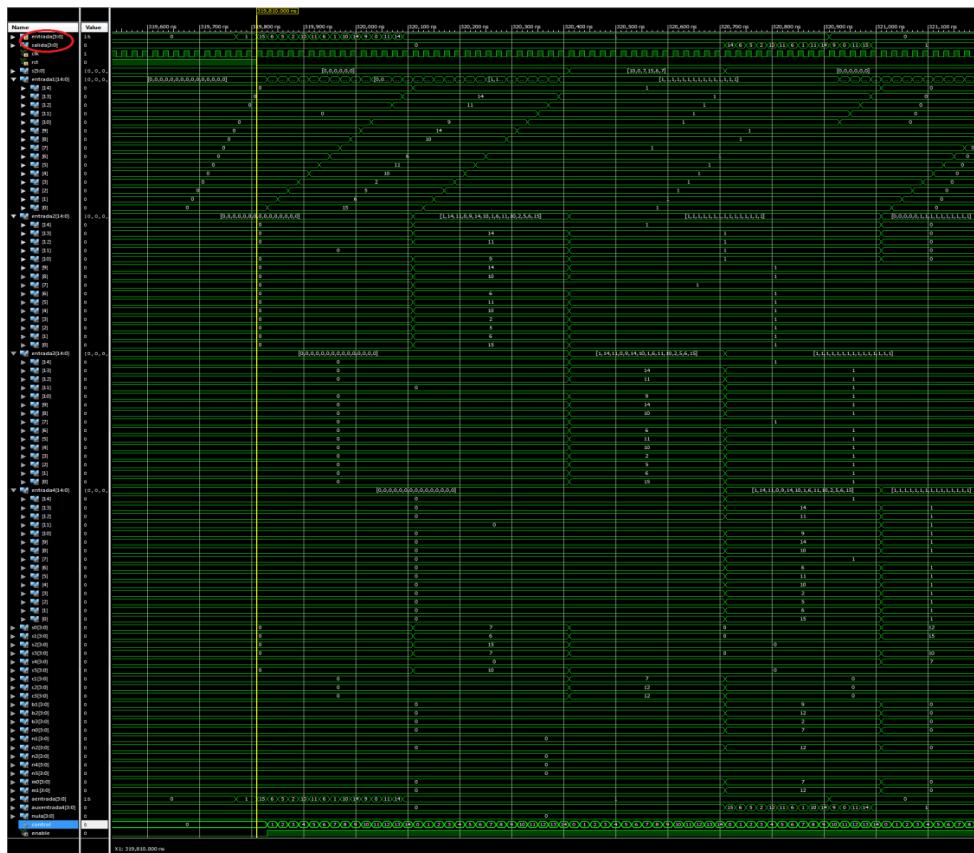


Figura. 7.18 Simulación del Decodificador RS(15,9)

## 7.8 Prueba de la implementación a través del software.

Para comprobar que el codificador y decodificador Reed-Solomon está funcionando correctamente y corrige no solo el *codeword* que se envió en la simulación sino cualquier *codeword* del código RS(15,9) se fija la misma distribución de ruido utilizada en la simulación en donde se corrompían 3 símbolos y probamos el codificador y decodificador enviando 1000 símbolos escogidos aleatoriamente, a partir de esto se calcula la tasa de error y se verifica la distribución de error en el canal.

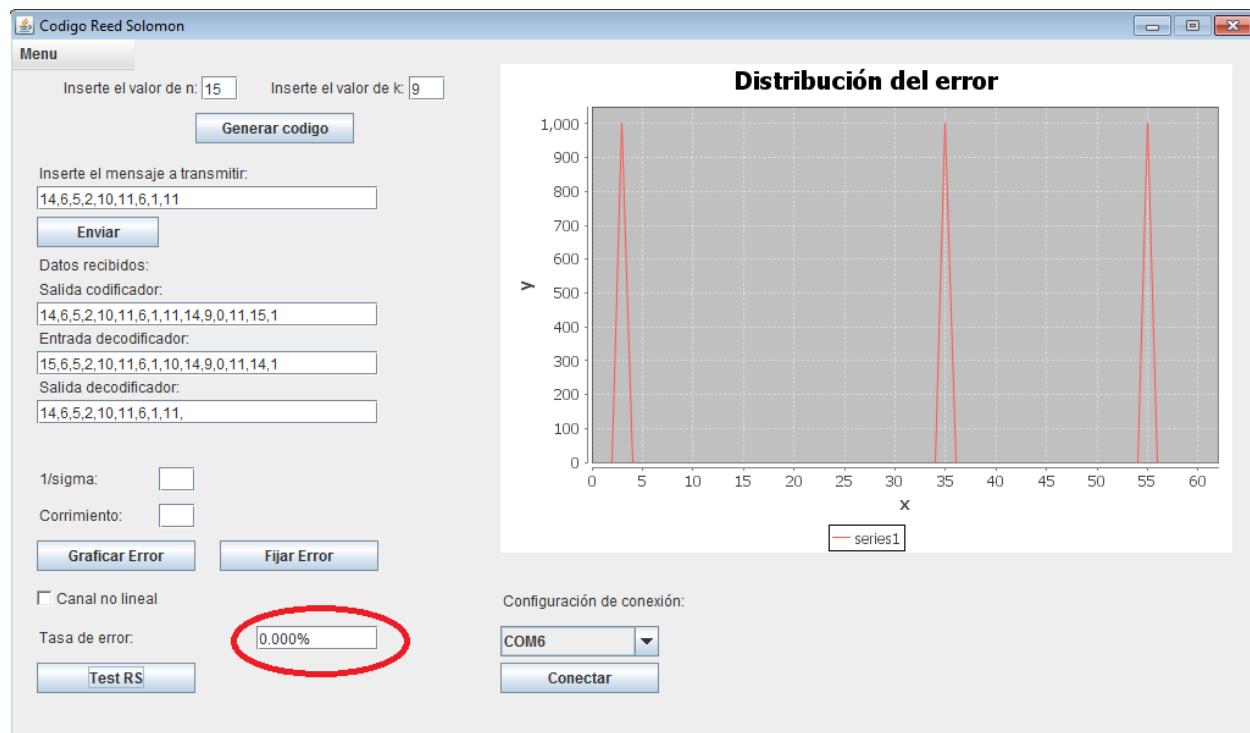


Figura. 7.19 Resultados del panel de control.

Como se puede ver en la figura 7.19 la taza de error es del 0% es decir que se corrigieron los 1000 *codewords* aleatorios que fueron enviados a la FPGA. Además en la gráfica que entrega el software que muestra la distribución de ruido en el canal se evidencia como se dañaron los 3 símbolos en cada uno de los 1000 experimentos que se hicieron.

En la siguiente prueba se dañaron todos los bits de los 3 símbolos que se dañaron anteriormente, esto con el fin de comprobar que el código Reed-Solomon al ser un código no-binario, no importa si se daña un solo bit del símbolo o todos, ya que este código corrige es por símbolos y no por bits.

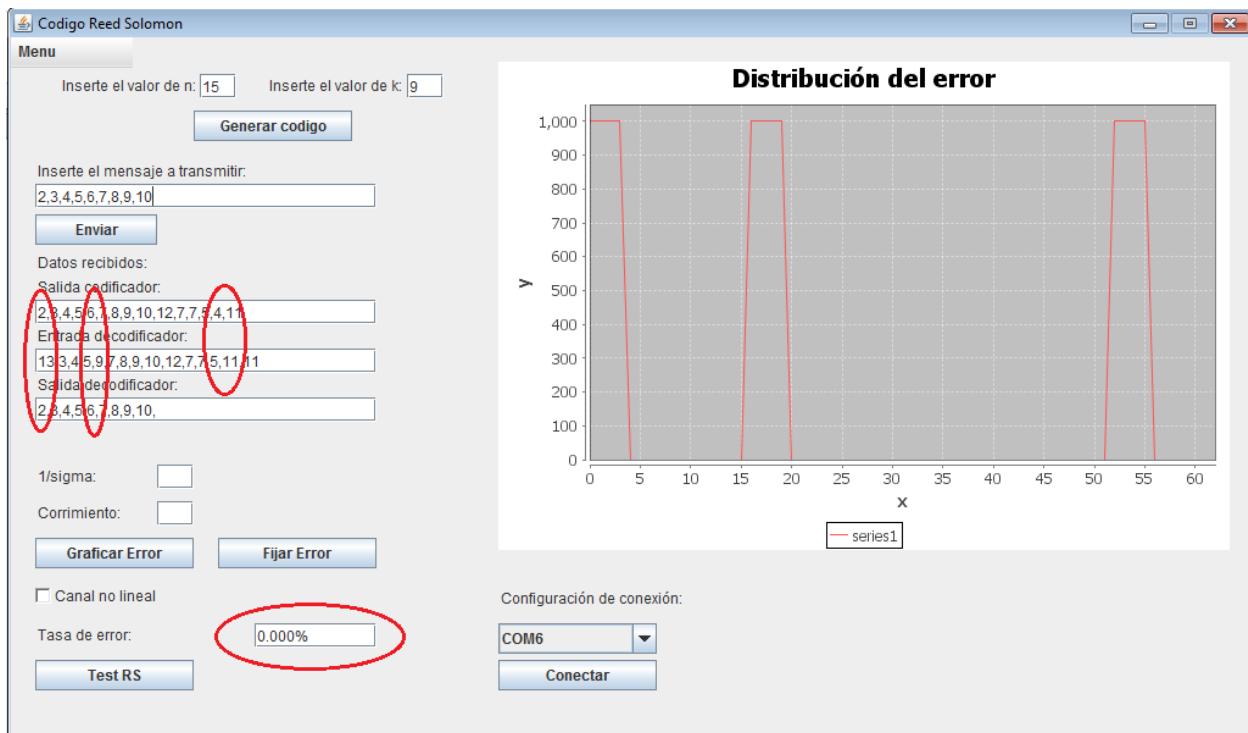
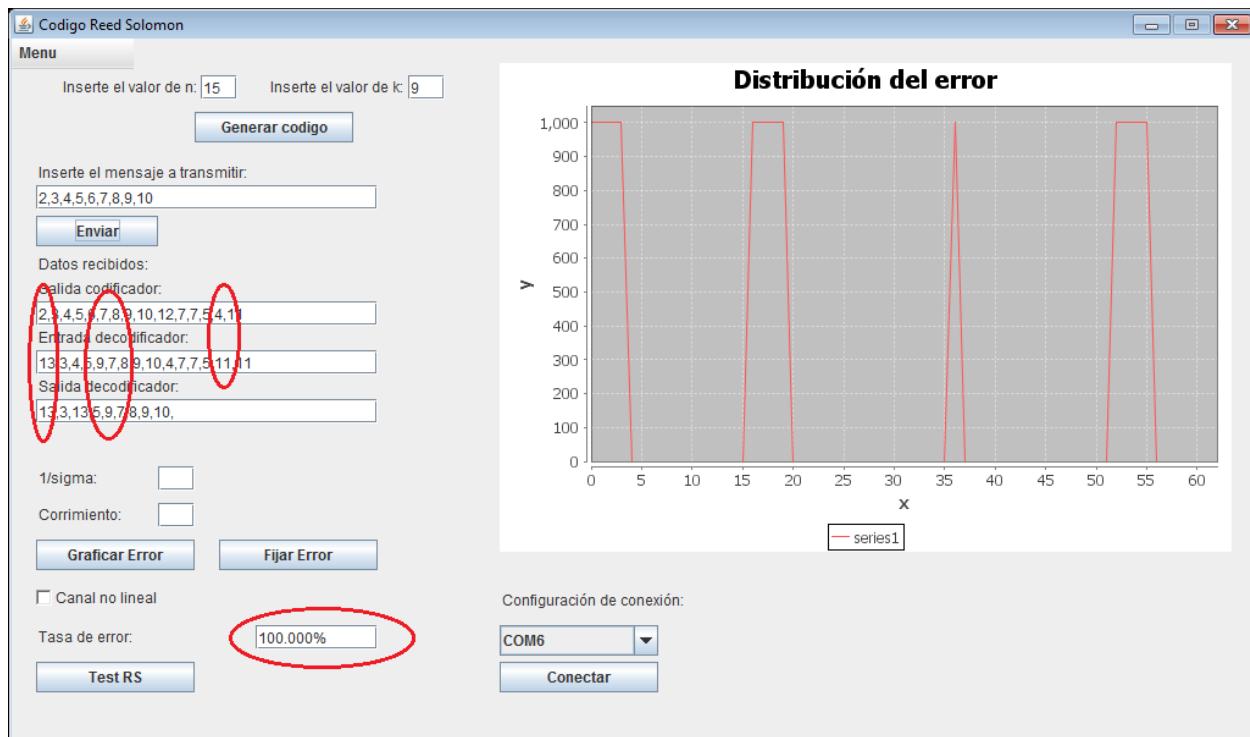


Figura. 7.20 Resultados del panel de control al dañar todos los bits de los símbolos 1, 5 y 14.

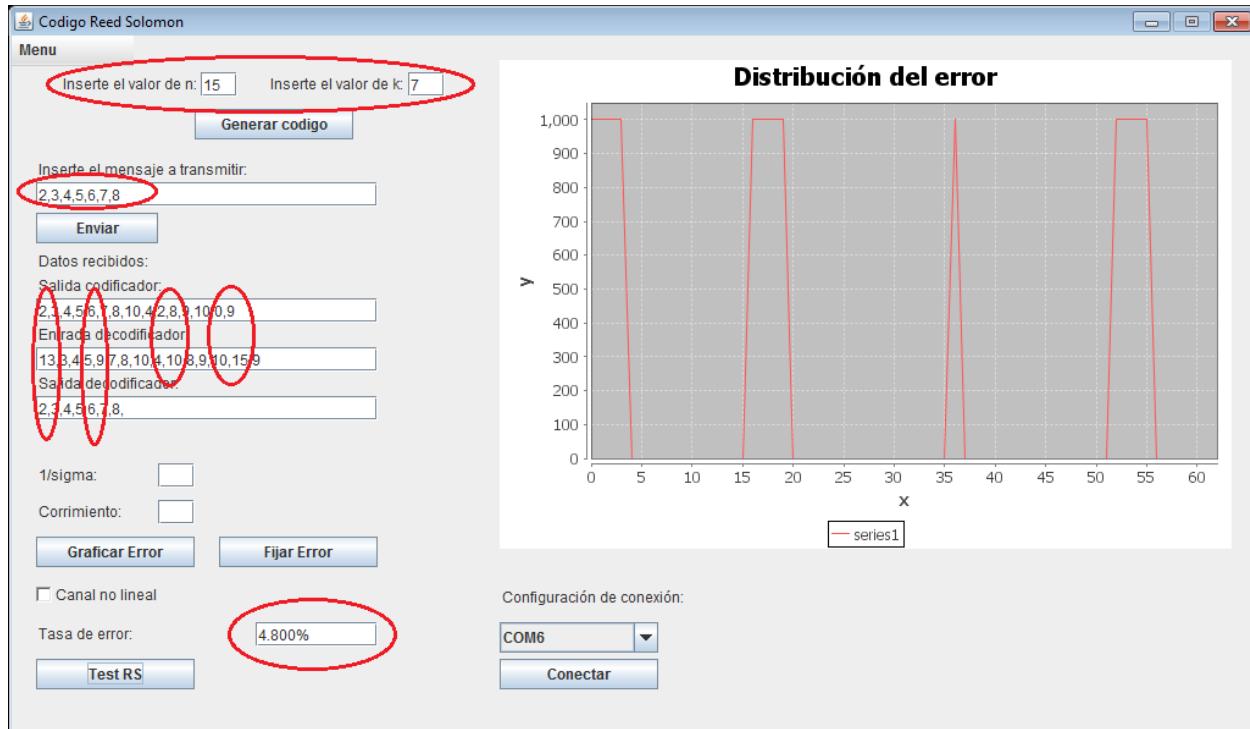
Note que el decodificador volvió a recuperar todos los mensajes corrompidos con una tasa de éxito del 100%.

Ahora dañamos un cuarto bit para analizar que sucede al superar la capacidad de corrección del código, que para este caso es de 3 símbolos. Note que el decodificador no puede corregir los 4 datos generando una tasa de error del 100%. Esto se puede evidenciar en la figura 7.21.

¿Qué podemos hacer si nuestro canal esta dañando 4 símbolos y nuestro código solo corrige 3?, la solución a este problema consiste en variar el parámetro  $k$  del código Reed-Solomon para aumentar la capacidad de corrección. Para ello generamos un código 15-7 que es capaz de corregir 4 símbolos y lo implementamos sobre la FPGA. Al repetir la prueba podemos ver en la figura 7.22 como cambian los resultados drásticamente. Note como se paso de una tasa de error del 100% a una tasa de error del 4.8 %, se entiende entonces lo que quería decir Shannon, que uno puede lograr establecer una tasa de error deseada a partir del uso de códigos de detección y corrección de errores. Solo variando el parámetro  $k$  se logra establecer la capacidad de corrección que se desee sobre el canal.



*Figura. 7.21 Resultados del panel de control al dañar un bit extra.*



*Figura. 7.22 Resultados del panel de control al cambiar a un código 15-7.*

Ahora modificamos de nuevo el parámetro k del código Reed-Solomon generando un código RS(15,5), de esta forma la capacidad de corrección aumentara a 5 símbolos. Los resultados se ven en la figura 7.23

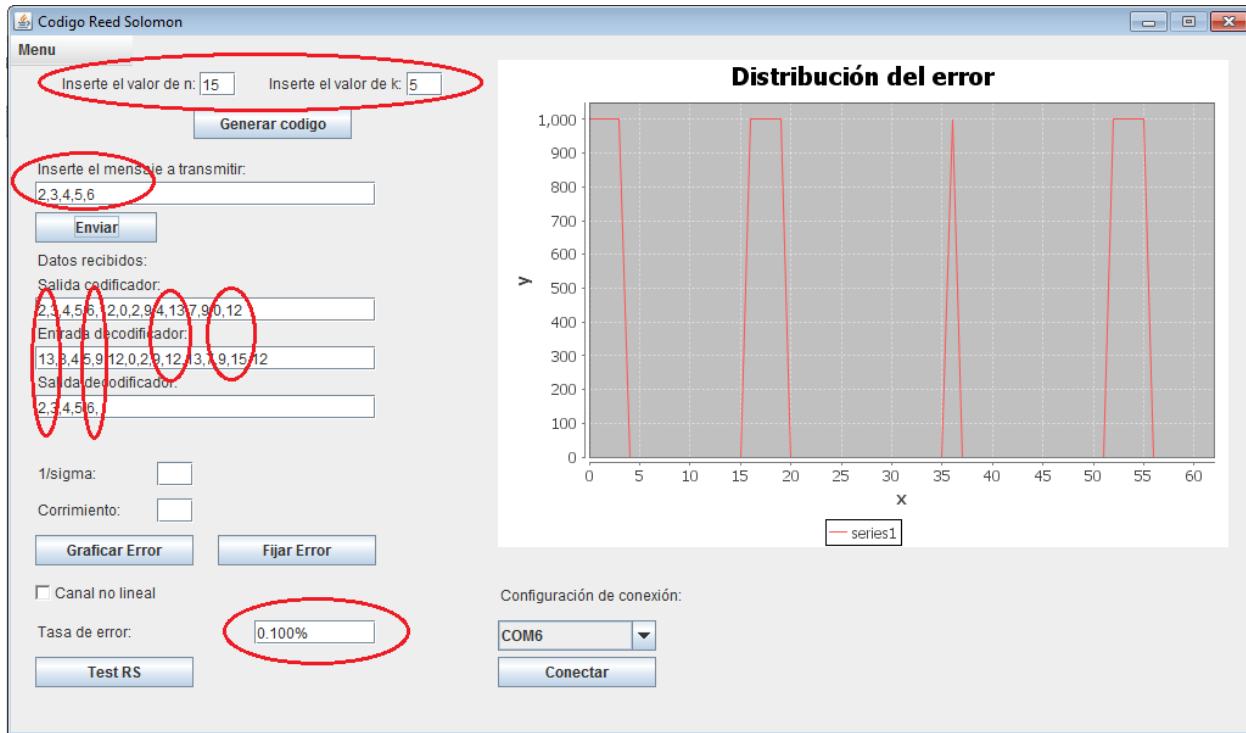


Figura. 7.23 Resultados del panel de control al cambiar a un código 15-7.

Esta vez obtenemos una tasa de error del 0.1%. Sin embargo al bajar el parámetro k necesitaremos enviar más palabras código para transmitir la misma cantidad de información. Si es posible aumentar la velocidad de trasmisión esta sería capaz de compensar la perdida de información útil que se pierde al agregar más capacidad de corrección al código.

Con los dos últimos ejemplos nos damos cuenta que es posible transmitir a una mayor distancia con la misma cantidad de potencia, aumentando la capacidad de corrección del código. La potencia que se ahorra al aumentar la capacidad de corrección se conoce con el término de potencia del código.

## 7.8. Consumo de hardware

Se intentó parametrizar el consumo de hardware a partir de la variación del número de bits y de la cantidad de bits de paridad. Para ellos se hicieron 5 pruebas solo variando la cantidad de bits dentro de los símbolos sin variar la cantidad de símbolos de paridad del código.

Código	Nº de Bits	Nº de Slices	Nº de Flip-Flops	Nº de LUTs de 4 entradas.
RS(7,3)	3	1056	766	1658
RS(15,11)	4	1913	1710	2543
RS(31,27)	5	6560	3884	9878
RS(63,59)	6	11964	8891	15883
RS(127,123)	7	41247	20192	63861

Tabla 3.7 Resumen del consumo de hardware dependiendo de la variación de bits por símbolo.

Las ecuaciones resultantes para cada uno de los parámetros de consumo en donde la variable que se modificó fue el número de bits por símbolo fueron las siguientes:

- $Nº\ de\ Slices = 4114.9x^2 - 32106x + 61975$
- $Nº\ de\ Flip - Flops = 1873.5x^2 - 14132x + 26894$
- $Nº\ de\ LUTs\ de\ 4\ entradas = 7237.1x^2 - 59711x + 123165$

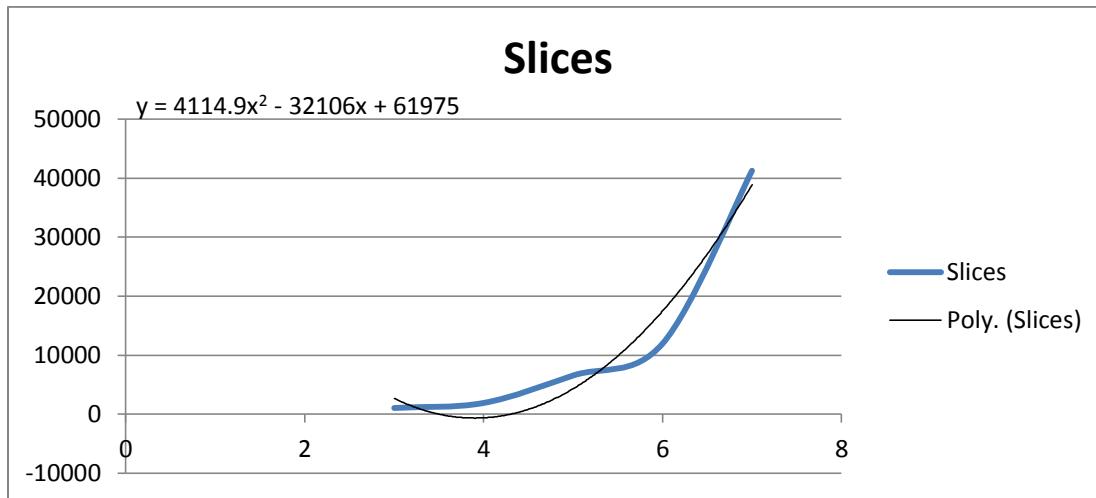


Figura. 7.20 Incremento del número de slices con el incremento en el número de bits por símbolo.

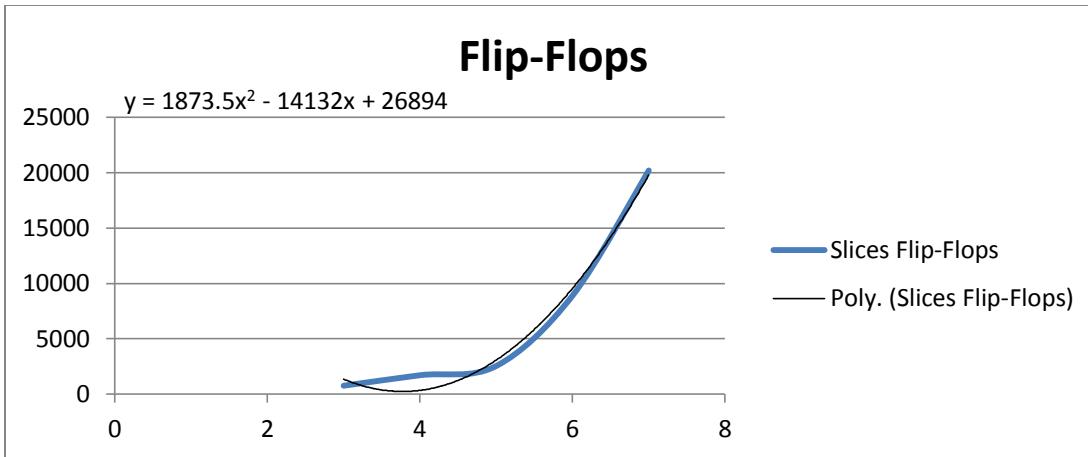


Figura. 7.21 Incremento del número de flip-flops con el incremento en el número de bits por símbolo.

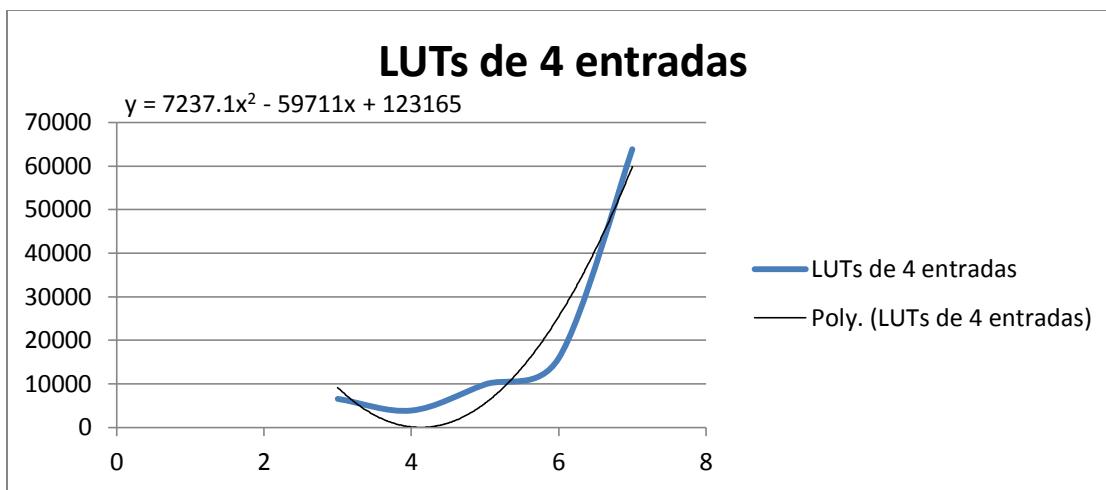


Figura. 7.22 Incremento del número de LUTs con el incremento en el número de bits por símbolo.

Los resultados para la variación de la cantidad de símbolos de paridad y cantidad de bits fija fueron las siguientes. Para un  $n = 15$

Código	Nº de símbolos de paridad.	Nº de Slices	Nº de Flip-Flops	Nº de LUTs de 4 entradas.
RS(15,11)	4	1913	1710	2543
RS(15,9)	6	2189	1782	3121
RS(15,7)	8	2394	1848	3506
RS(15,5)	10	2645	1920	4042
RS(15,3)	12	2839	1993	4394

Tabla 7.4 Resumen del consumo de hardware dependiendo de los símbolos de paridad.

Las ecuaciones resultantes para cada uno de los parámetros de consumo en donde la variable que se modificó fue el número símbolos de paridad para un  $n = 15$  fueron las siguientes:

- $slices = 115.4x + 1472.8$
- $Flip - Flops = 35.2x + 1569$
- $LUTS = 231.15x + 1672$

Hay que tener en cuenta que estas ecuaciones solo sirven para el caso de un  $n = 15$  las pendientes y punto de corte de esta recta dependerán del valor de  $n$ . Lo que importa destacar con este análisis es el comportamiento lineal que tiene el incremento de los símbolos de paridad en el consumo del hardware. Mientras que la modificación del número de bits por símbolo tienen un comportamiento con tendencia exponencial.

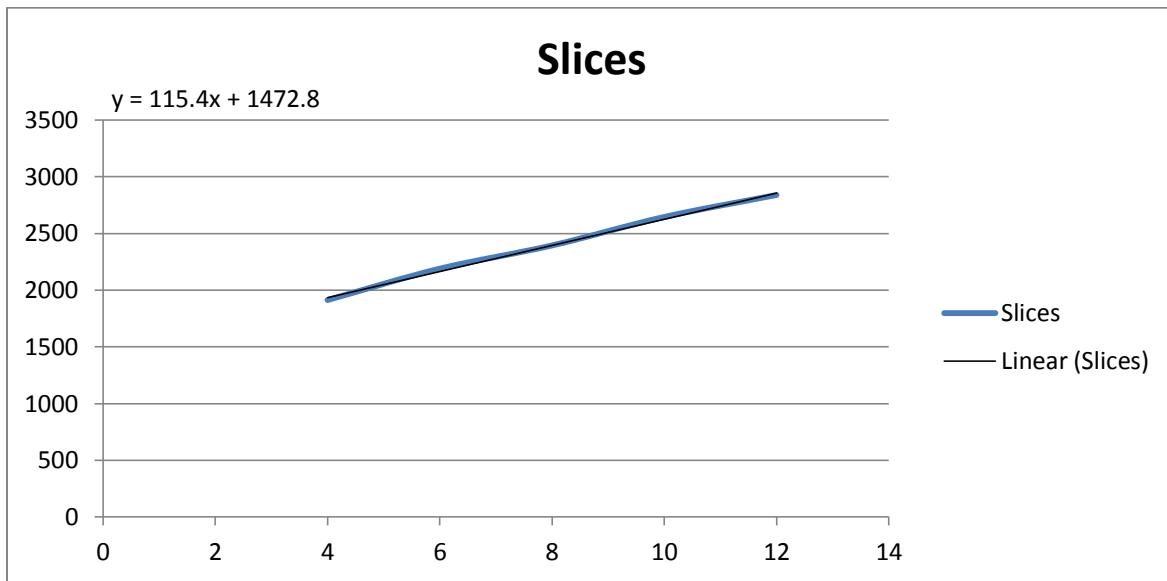


Figura. 7.23 Incremento del número de slices con el incremento en el número de símbolos de paridad.

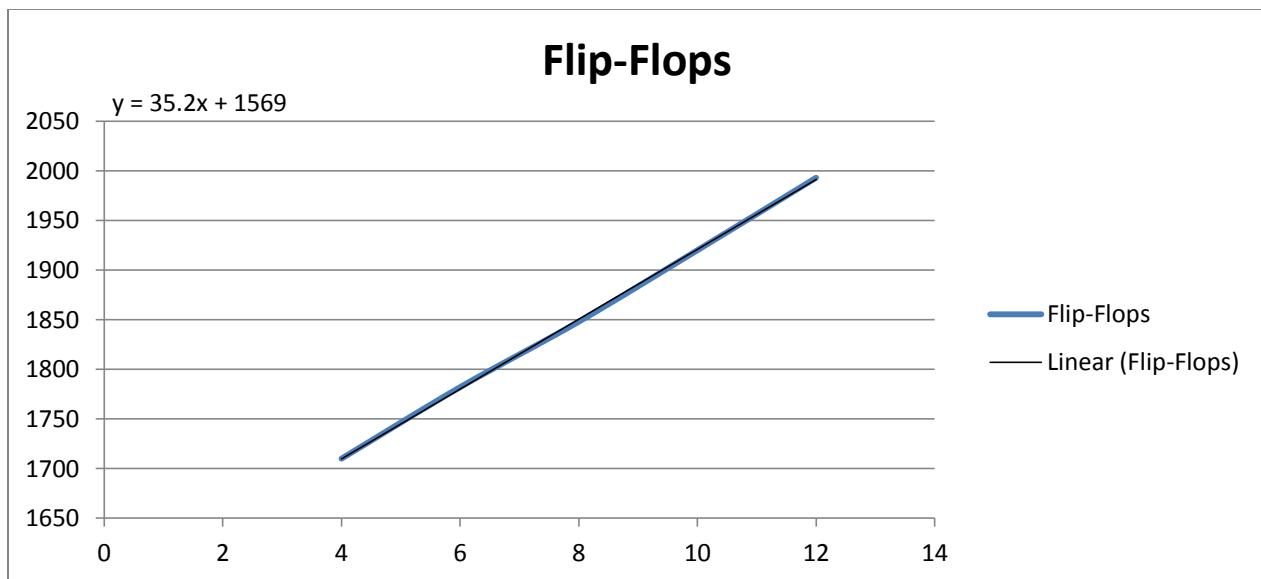


Figura. 7.24 Incremento del número de flip-flops con el incremento en el número de símbolos de paridad.

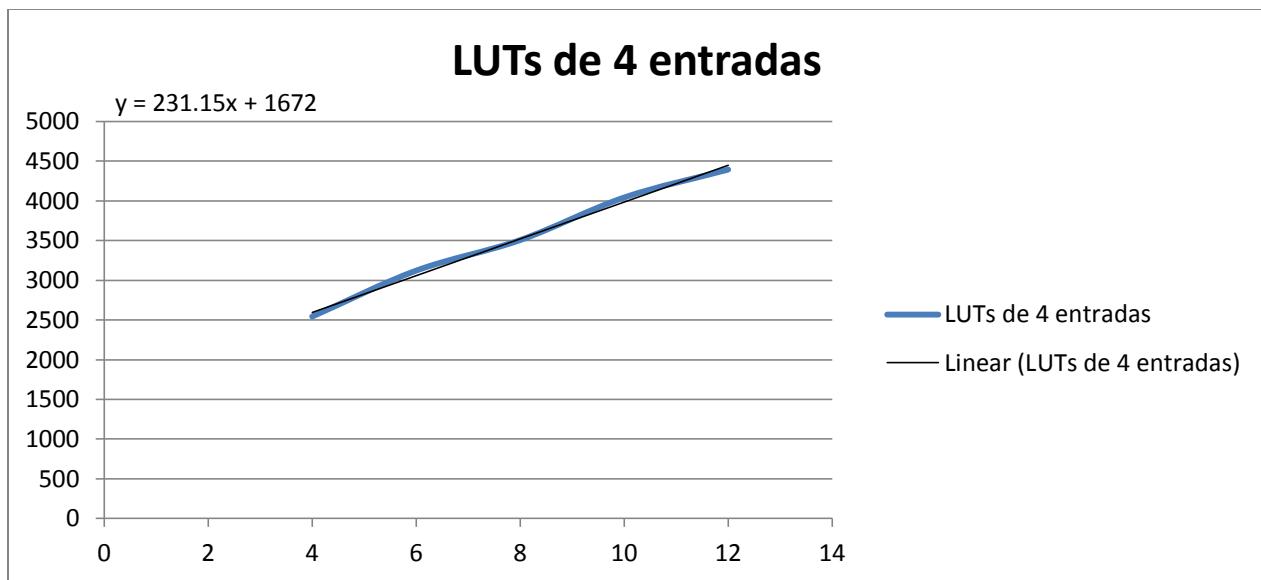


Figura. 7.25 Incremento del número de LUTs con el incremento en el número de símbolos de paridad.

## 7.9. Calculadora

Se realiza la validación del funcionamiento de la calculadora, primero se selecciona el modo de funcionamiento *alfas*, se valida la operación descrita en la ecuación 7.18 basados en la tabla 2.3, los resultados se muestran en la figura 7.26.

$$\alpha^5 + \alpha^{12} = \alpha^{14} \text{ (7.18)}$$

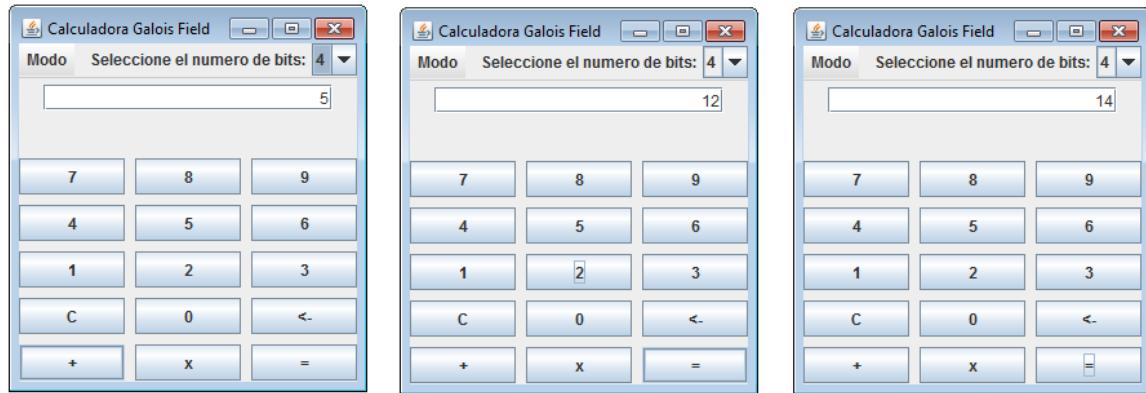


Figura. 7.26 Validación del funcionamiento de la calculadora modo *alfas*.

Se realiza la validación del funcionamiento del modo *binario*, por medio de la operación mostrada en la ecuación 7.19 basada en la tabla 2.4, es importante anotar que cuando se cambia de modo todos los parámetros de la calculadora retornan a su valor inicial, además podemos observar que como en este modo solo necesitamos ingresar 0 o 1, los botones correspondientes a otros números se bloquean para que el usuario no digite un valor erróneo. Los resultados de la validación se pueden observar en la figura 7.27.

$$\alpha^{10} \times \alpha^4 = \alpha^{14} \quad (7.19)$$

$$(0111) \times (0011) = (1001) \quad (7.20)$$

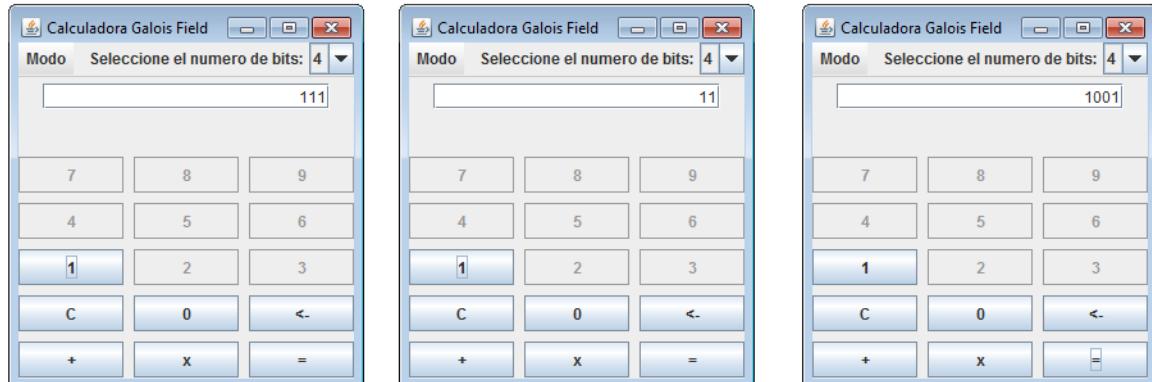


Figura. 7.27 Validación del funcionamiento de la calculadora modo *binario*

En la figura 7.28, se muestran los resultados de la validación del modo *conversor*, se utilizo la ecuación descrita en la ecuación 7.21 para corroborar el correcto funcionamiento del modulo.

$$\alpha^7 = 1011 \quad (7.21)$$

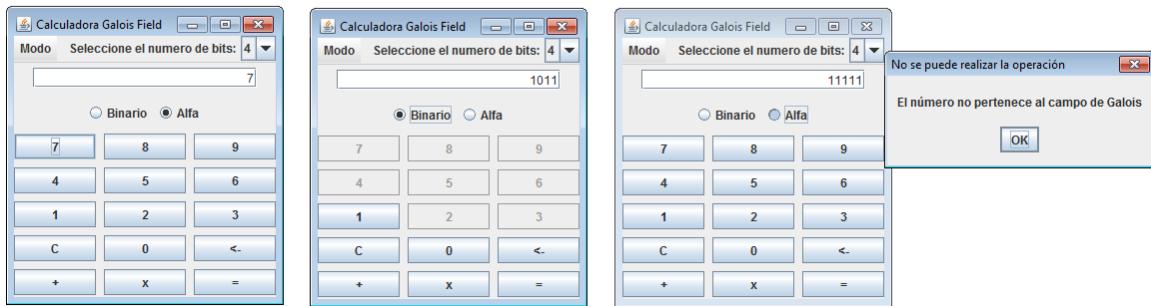


Figura. 7.28 Validación del funcionamiento de la calculadora modo **conversor**.

También se puede observar que si se ingresa un valor que no corresponde a ningún símbolo dentro del campo de *Galois* se muestra un mensaje en la pantalla informando el evento encontrado.

# 8

## Conclusiones y trabajo futuro

- La elaboración de una herramienta de cálculo sobre campos de Galois es fundamental a la hora de abordar temas que incluyan este tipo de matemática, ya que la dificultad para entender cualquier tema sin tener a la mano una herramienta de este tipo, dificulta el proceso aprendizaje considerablemente.
- El entendimiento de la matemática de Galois es esencial para poderse introducir en temas de codificación, encriptación y compresión de información.
- La implementación de un código Reed-Solomon es compleja para códigos muy extensos como un RS(255,247), por esto es conveniente construir con código más simple y generalizarlo para generar cualquier código deseado independiente de su complejidad.
- La velocidad de la implementación puede aumentarse si en la entidad que calculan el valor de las raíces del polinomio localizador del error y la entidad que calcula el polinomio del valor del error, se hacen sin elementos secuenciales. Teniendo en cuenta que esto produciría un incremento significativo en el uso del hardware dentro de la FPGA.
- Se evidencia que el código Reed-Solomon es eficiente frente a errores en ráfaga. Sin embargo si el canal presenta una distribución de error capaz de corromper simultáneamente más símbolos que los que son posibles corregir, la eficiencia del código disminuye considerablemente.
- El uso de software para generar las entidades VHDL permite abrir un campo de posibilidades al poder generar una descripción de hardware reconfigurable, la cual antes estaba restringida al uso de GENERICS dentro de las entidades VHDL.

- El uso de software para probar el código Reed-Solomon permite tener un control total de las variables a las que este se enfrenta y evaluar mejor su capacidad de corrección.

## 8.1. Trabajos futuros

---

- Se propone la implementación de codificadores y decodificadores de códigos Reed-Solomon, Turbo códigos y códigos de baja densidad, mediante el uso de algoritmos de aprendizaje como los basados en redes neuronales y algoritmos genéticos. Ya que como las entradas y salidas del codificador y decodificador son conocidas es posible entrenar sistemas de aprendizaje para lograr el mismo propósito.
- Separar el codificador y decodificador físicamente con el fin de realizar las pruebas del canal y evaluar la capacidad de corrección y potencia del código en un ambiente real.
- Evaluar otros algoritmos para la localización del error como el “Euclidean Algorithm”.
- Mover las tablas que se usan para la conversión de binario a alfas a la memoria RAM de la FPGA con el fin de consumir menos hardware.

# 9

## Anexos

# ANEXO A

FPGA utilizada en la implementación.



*Figura A1 Spartan 3A3AN apariencia física.[18]*

## CARACTERISTICAS PRINCIPALES

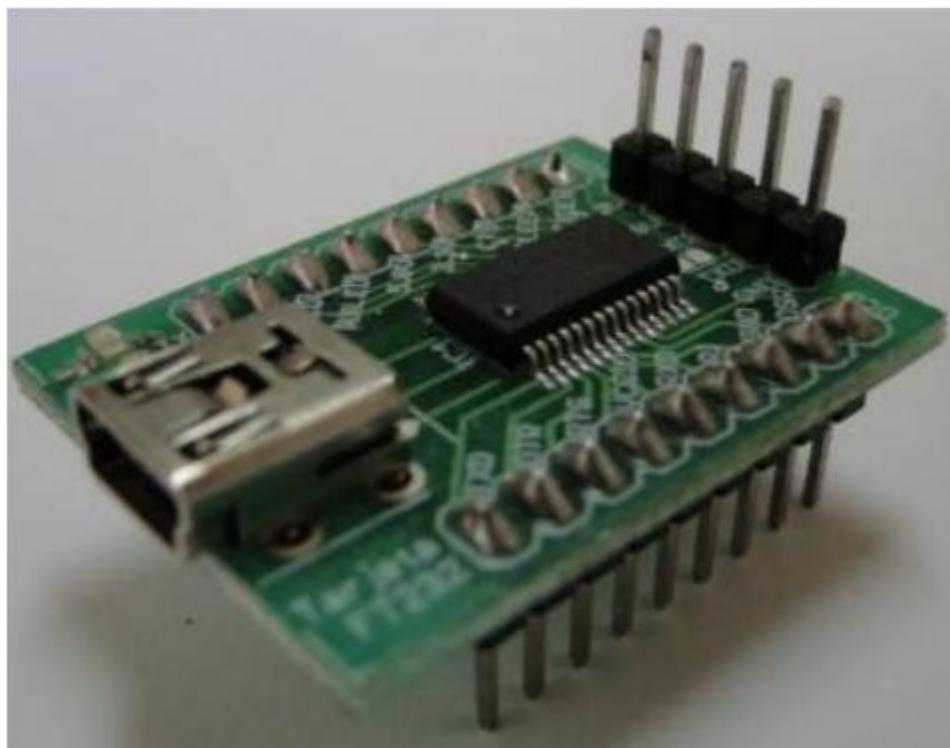
<b>Reloj</b>	<ul style="list-style-type: none"> <li>• Cristal de 50 MHz en la tarjeta de desarrollo.</li> <li>• Espacio para reloj de usuario.</li> </ul>
<b>Memoria</b>	<ul style="list-style-type: none"> <li>• 4 Mbit Platform Flash PROM</li> <li>• 32M x 16 DDR2 SDRAM</li> <li>• 32 Mbit parallel Flash</li> <li>• 2-16 Mbit SPI Flash devices</li> </ul>
<b>Dispositivos de interfaces análogas.</b>	<ul style="list-style-type: none"> <li>• 4-channel D/A converter</li> <li>• 2-channel A/D converter</li> <li>• Signal amplifier</li> </ul>
<b>Interfaces y conectores.</b>	<ul style="list-style-type: none"> <li>• Ethernet 10/100 PHY</li> <li>• JTAG USB download port</li> <li>• Two 9-pin RS-232 serial port</li> <li>• PS/2-style mouse/keyboard port</li> <li>• 15-pin VGA connector capable of 4,096 colors</li> <li>• One FX2 100-pin and two 6-pin expansion connectors</li> <li>• 20 user I/O available on standard header pins</li> <li>• Stereo mini-jack for PWM audio</li> <li>• Rotary/push button function switch</li> <li>• Eight individual LED outputs</li> <li>• Four slider switches, four push-button switches.</li> </ul>
<b>Display</b>	<ul style="list-style-type: none"> <li>• 16 character, 2-Line LCD</li> </ul>

# ANEXO B

## CONVERSOR USB-SERIAL USADO PARA LA COMUNICACIÓN ENTRE LA FPGA y EL PC.

### DESCRIPCION

Tarjeta para el FT232RL, circuito integrado conversor de interfaz serial USB-UART. Se alimenta directamente del puerto USB, y su pin VCCIO puede ser puesto a 3.3 V por medio del solder-jumper (SJ-1, ubicado en la cara superior entre el FT232 y la regleta de 5 pines).[6]



*Figura. B1 Conversor USB-SERIAL utilizado en la comunicación. Apariencia física.*

## CARATERISITICAS PRINCIPALES

- Chip USB a interfaz de transferencia de datos serial asincronos.
- Protocolo USB manejado enteramente desde el chip. Ningún firmware de programación requerida.
- 1024 EEPROM integrada y configuración CBUS I/O.
- Resistencias de terminación USB integradas.
- Reloj integrado ningún cristal externo requerido más habilitar la salida del reloj para sincronizar con FPGA o MCU
- Tasas de transferencia de datos desde 300 baudios hasta 3 Mbaudios (RS422, RS485, RS232 ) en niveles TTL.
- Buffer de recepción de 128 bytes y buffer de transmisión de 256 bytes utilizando tecnología de buffer smoothing.
- FTDI's royalty-free Virtual Com Port (VCP) y Direct (D2XX) drivers eliminando el requerimiento de usar driver USB en la mayoría de los casos. Característica USB FT232R-ID™.
- Pines CBUS I/O configurables.
- Transmite y recibe señales de drive de LED.
- Interfaz UART soporta 7 o 8 bits de datos, 1 o 2 bits de parada and par / impar / marca / espacio / no paridad.
- FIFOs de recepción y buffers de transmisión para un alto rendimiento.
- Opciones de bit sincrónico y asíncrono con RD# and WR# strobes.
- Dispositivo con serial USB único. Soporta bus auto-alimentado ,y configuraciones USB de alto rendimiento
- Convertido a nivel de 3.3 V integrado para I/O USB.
- Convertidor de nivel a UART integrado y CBUS para interfaces entre +1.8V y +5V.
- True 5V/3.3V/2.8V/1.8V CMOS drive salida y entrada TTL.
- Pines de I/O con drive fuerte de salida
- Circuito integrado de power-on-reset.
- Filtro de alimentación AVCC integrado - no es requerido ningún filtro externo.
- Opción de conversión de señal UART. +3.3V (usando oscilador externo) a +5.25V (oscilador interno). Una sola fuente de alimentación requerida.
- Operación baja y suspensión de corriente USB.
- Bajo consumo de ancho de banda USB.
- Compatible con controladores de host UHCI/OHCI/EHCI .
- Compatible con USB 2.0 Full Speed.
- -40°C a 85°C operación de rango extendido.
- Disponible en encapsulado de 28 pines compacto Pb-free SSOP y paquetes QFN-32 (ambos RoHS esclavos).

# 10

## Glosario

- **Codeword:** Conjunto de símbolos de un campo de Galois que forman el mensaje a transmitir.
- **Símbolo:** Conjunto de bits que representan un elemento de un campo de Galois y que tienen como longitud  $\lceil \log_2 n \rceil$ . Donde  $n$  es la cantidad de codewords que contiene el código.
- **Alfas:** Elemento de un campo extendido de Galois, utilizado para representar matemáticamente los símbolos dentro de un campo de Galois.
- **Código:** Conjunto de palabras código (codewords) que pueden ser transmitidas.
- **Campo:** Conjunto de elementos que poseen las propiedades de suma, resta, multiplicación, división y que el resultados de estas son otro elemento dentro del conjunto que lo componen.
- **Campo de Galois:** Conjunto de elementos finitos que cumplen los axiomas de suma por cerradura y multiplicación por cerradura.
- **Campo de Galois extendido:**
- **Potencia del código:** Es la potencia que se ahorra al implementar un código en una transmisión. Es decir que si se necesita una potencia P1 para trasmisir correctamente y con la implementación del código se necesita una potencia P2. La diferencia entre P1 y P2 es la potencia del código.
- **LFSR:** (Linear Feedback Shift Register) Arreglo de registros a los que se realimenta su entrada con la señal de alguno de los registros que lo conforma.
- **GENERICs:** Instrucción dentro de VHDL que define y declara constantes y propiedades del modulo al que pertenece.
- **UART:** (universal asynchronous receiver/transmitter) Protocolo de comunicación utilizado para las transmisión serial de datos. Ideal para hacer conversiones de datos serial en paralelo y viceversa.

# 11

## Bibliografía

- [1] Sklar, B., *Digital Communications: Fundamentals and Applications, Second Edition* (Upper Saddle River, NJ: Prentice-Hall, 2001).
- [2] Moon, T. K. (2005). *Error Correction Coding Mathematical Methods and Algorithms*. Hoboken, New Jersey: John Wiley & Sons, Inc. Pagina 8.
- [3] Moon, T. K. (2005). *Error Correction Coding Mathematical Methods and Algorithms*. Hoboken, New Jersey: John Wiley & Sons, Inc. Pagina 258.
- [4] Shen Hai-Wei; Li Jin-ping, "A High-Speed and Long-Period Combined Pseudo-random Number Generator," *Computational Intelligence and Design, 2009. ISCID '09. Second International Symposium on*, vol.1, no., pp.112,114, 12-14 Dec. 2009
- [5] Cristian Sisterna, (2003), "Generador de Secuencia Binaria Pseudo Aleatoria", Nota Técnica 12,  
Disponible en: [http://c7t-hdl.com/Docs/C7T\\_NT12\\_PRBS\\_LFSR.pdf](http://c7t-hdl.com/Docs/C7T_NT12_PRBS_LFSR.pdf)
- [6] [Online][Citado 2013-12-07]  
Disponible en: <http://www.sigmaelectronica.net/manuals/TARJETA%20FT232.pdf>
- [7] Enrique Ponsoda, Rafael Company.(2000).Algebra, Valencia:Ed.Univ.Politéc. Pagina 121.
- [8] Richard E. Blahut, *Algebraic Code for Data Transmission*, Cambridge University Press, New York, USA, 2003, pp.5.
- [9] S Kaulgud, M Mukherjee, "FPGA Implementation of Reed-Solomon Codes" International Conference and Workshop on Emerging Trends in Technology– TCET, Mumbai, India. 2011.

- [10] Liu Tong; Zhang Chuan, "Optimization design of reed-solomon decoder based on FPGA," *Electronics, Communications and Control (ICECC), 2011 International Conference on* , vol., no., pp.368,371, 9-11 Sept. 2011.
- [11] Moon, T. K. (2005). *Error Correction Coding Mathematical Methods and Algorithms*. Hoboken, New Jersey: John Wiley & Sons, Inc. Pagina 262-264.
- [12] Moon, T. K. (2005). *Error Correction Coding Mathematical Methods and Algorithms*. Hoboken, New Jersey: John Wiley & Sons, Inc. Pagina 248-249.
- [13] Monica Borda.(2011).Fundamentals in Information an Theory Coding.Springer. Pagina 49.
- [14] Pong P. Chu.(2008).FPGA Prototyping by VHDL examples.Cleveland.John Willey & Sons, Inc. New Jersey.Pagina 163.
- [15] Michael J. Miller. Branka Vucetic.Les Berry.(1993).Satelite communications:Mobile and Fixed Services.Springer.pagina 134.
- [16] Liliana Blanco Castañeda.(2004).Probabilidad.Universidad Nacional de Colombia.pagina 295.
- [17] Vera Pless. (2011).Introduction to the theory of error correcting codes. John Willey & Sons. Página 17.
- [18] [Online][Citado 2013-12-07]  
Disponible en: <http://www.gta.ufrj.br/ensino/EEL480/spartan3/ug334.pdf>
- [19] Sklar, B. (2001), *Digital Communications: Fundamentals and Applications, Second Edition* (Upper Saddle River, NJ: Prentice-Hall). página 11.
- [20] John Proakis Masoud Salehi.(2008).Digital communications.McGraw-Hill Higher Education. Pagina 1.