UNIVERSIDAD NACIONAL DE MAR DEL PLATA

PROYECTO FINAL

Banco de pruebas para sistemas de decodificación implementado en SoC

Autor:
Luciano DELAUDE

Directores:
Mg. Ing. Mónica Cristina LIBERATORI
Ing. Leonardo Oscar COPPOLILLO

Proyecto final presentado para cumplir los requerimientos para la adquisición del titulo de Ingeniero Electrónico

en el

Laboratorio de Comunicaciones
Departamento de Electrónica

10 de octubre de 2022

UNIVERSIDAD NACIONAL DE MAR DEL PLATA

PROYECTO FINAL

Banco de pruebas para sistemas de decodificación implementado en SoC

Autor:
Luciano DELAUDE

Directores:
Mg. Ing. Mónica Cristina LIBERATORI
Ing. Leonardo Oscar COPPOLILLO

Proyecto final presentado para cumplir los requerimientos para la adquisición del titulo de Ingeniero Electrónico

en el

Laboratorio de Comunicaciones
Departamento de Electrónica

10 de octubre de 2022

Agradecimientos

Agradezco principalmente a mis padres quienes me permitieron la posibilidad de estudiar fuera de mi ciudad todos estos años, sin ellos no podría haber logrado el objetivo en un proceso tan extenso.

A mis hermanos por siempre ser un ejemplo a seguir en todo lo que respecta a la vida y sobre todo academicamente. Siempre me brindaron su ayuda desde el inicio hasta el final de la carrera.

A mis directores por acompañarme y brindarme su conocimiento tanto en sus materias como en este proyecto. Así como también ayudarme a redactar este informe.

A mis amigos y compañeros de facultad que más allá de acompañarme con el proceso de estudio, también me acompañaron a disfrutar de otras actividades fuera de la universidad.

A mi familia en general por siempre alentarme a seguir adelante en los momentos difíciles y acompañarme a disfrutar todo este trayecto.

Por último, a mi por encontrar la motivación y la constancia para terminar mis estudios luego de tanto tiempo transcurrido.

Luciano

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Desde el nacimiento de la teoría de la información, iniciada por el trabajo realizado por Claude Shannon en 1948 [1], el objetivo principal de la codificación de errores en canales ha sido el de encontrar esquemas prácticos que puedan alcanzar la capacidad del canal, los cuales a su vez deben brindar la capacidad de afrontar las alteraciones que sufre la información al atravesar el mismo. En la actualidad, existen una gran variedad de códigos que cumplen con estas condiciones a tasas de error extremadamente bajas, como por ejemplo los recientemente descubiertos Códigos Polares, para los cuales se utiliza una decodificación soft, como también sucede con los tradicionales Códigos Turbo o los LDPC. Aunque también se utilizan códigos hard como Reed-Solomon o Hamming.

La decodificación soft utiliza una medida cuantificada de la confiabilidad relacionada con la información que proviene del canal.

El proyecto consistirá en el diseño de un banco de pruebas genérico, que permitirá evaluar la tasa de error binaria (BER), debido a que es un parámetro clave para medir el desempeño de dichos códigos. La idea del mismo se originó debido a la relación que existe entre la empresa Inphi y el Laboratorio de Comunicaciones de la UNMDP, ya que forma parte de un acuerdo para la realización de proyectos en conjunto.

Al desarrollar códigos de corrección, estos deben ser simulados para verificar el desempeño de los mismos. Dichas simulaciones se conocen como test-bench y pueden realizarse en computadoras [2]. El problema es que suelen necesitar de un uso intensivo y sus ejecuciones son muy prolongadas en el tiempo, por lo que se prefiere realizarlas sobre dispositivos programables lógicos haciendo uso de las tecnologías de aceleración de simulación mediante hardware. Estos dispositivos permiten su programación mediante lenguaje HDL (Hardware Description Language) e implican mejor performance en la simulación, reduciendo los tiempos de prueba. En general, se dispone de una unidad bajo verificación, un bloque encargado de generar estímulos de prueba y un bloque de monitoreo que se encarga de examinar las salidas del sistema.

Actualmente, en la era de las multi-millones de puertas ASICs, propiedad intelectual reutilizable y diseños SoCs, el proceso de verificación consume cerca del 70 % del diseño total. Dado este gran esfuerzo demandado por la verificación, hoy en día se busca realizar este trabajo en paralelo al desarrollo de los proyectos, por lo que se trata de lograr mayores niveles de abstracción para no enfocarse en detalles a bajo nivel.

Uno de los puntos clave a tener en cuenta cuando se realiza un test-bench es saber qué se desea verificar, ya que se puede verificar fácilmente si el diseño implementa o no la función deseada, identificando una discrepancia entre lo que se obtuvo y lo que se esperaba obtener.

La implementación se desarrollará sobre una placa FPGA (Field Programmable Gate Arrays), específicamente el kit de desarrollo DE-10 Standard de Altera el cual presenta un diseño en torno a SoC, ya que combina un procesador Cortex-A9 en conjunto con el diseño en FPGA.

El banco a desarrollar presentará el diagrama de bloques que se aprecia en la Fig. 1.1. Allí se pueden ver 2 bloques principales (HPS/PROCESADOR y FPGA).

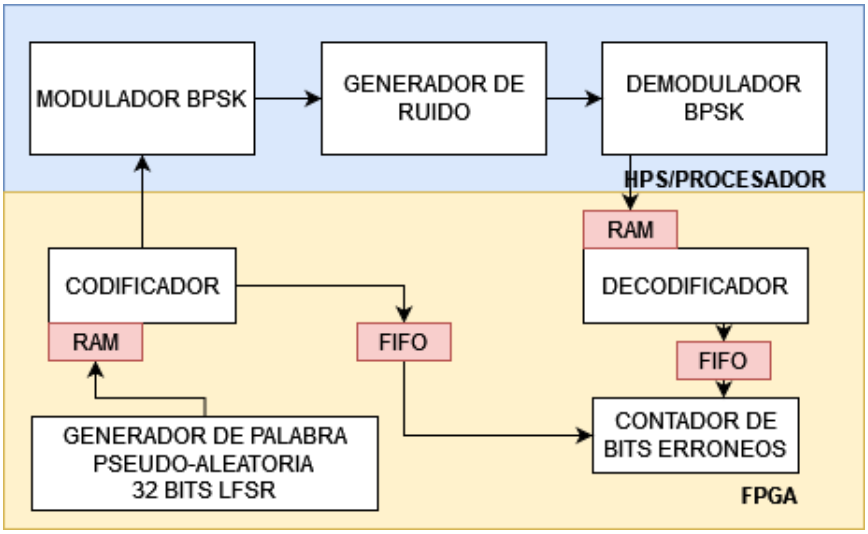


FIGURA 1.1: Banco de prueba para códigos implementado en FPGA.

Comenzando por el bloque FPGA, se puede apreciar el bloque inicial, el cual es un LFSR de 32 bits, siendo su código el mostrado en B.1.1. En la Fig. 1.2 se muestra un ejemplo de este LFSR. El mismo nos permite generar un símbolo aleatorio por vez, que luego es almacenado en la RAM asociada al codificador.

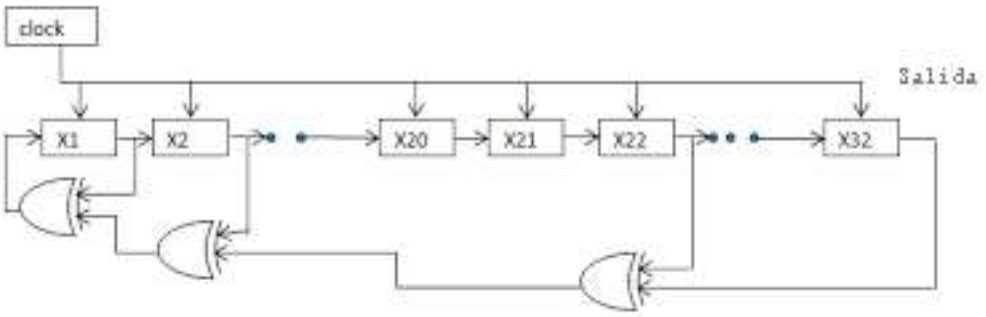


FIGURA 1.2: Registro de desplazamiento de 32 bits.

Una vez almacenados todos los símbolos requeridos para una palabra, se codifica la misma. Estos datos pasan al procesador, cuya funcionalidad se describirá más adelante.

Los datos obtenidos del procesador ingresan nuevamente a la FPGA mediante comunicación serie y el decodificador los almacena en RAM. Una vez decodificada la palabra, el codificador se comunicará con otro bloque a fin de verificar su funcionamiento, para ello, la palabra decodificada ingresará al contador de bits erróneos. En el contador se comparará bit a bit, la palabra decodificada con la enviada para determinar el número de bits recibidos erróneos, contra la cantidad total de bits

enviados, en cierto intervalo de tiempo. Esta elección de diseño será aplicable a la verificación de cualquier decodificador.

Volviendo a los demás bloques mostrados en la Fig. 1.1, la palabra codificada ingresa al procesador mediante comunicación serie y se almacena en la RAM del mismo. Mediante modulación unipolar NRZ (sin retorno a cero) se obtiene la palabra modulada, la cual es contaminada con muestras de la distribución de ruido elegida y demodulada mediante un comparador. El mismo consiste en comparar la señal recibida con un valor "x", en caso de que la misma se encuentre por encima de dicho valor se interpreta que el bit demodulado es un "1" y si es menor a "x" se lo interpreta como "0". Este valor "x" de umbral varía según la distribución de ruido elegida. La distribución deseada se obtiene mediante un sistema de segmentación jerárquica que será explicado en mayor detalle más adelante.

Capítulo 2

Aspectos teóricos

2.1. Canales de comunicación y codificación.

Un sistema de comunicaciones puede modelarse de forma general como en la (Fig. 2.1) [1]. Un emisor es una fuente de información que permite enviar mensajes a un destinatario. Esta información es codificada digitalmente de forma adecuada por la fuente, en mensajes de K bits. Un bloque transmisor es a continuación encargado de convertir esos mensajes en señales, de manera propicia para ser enviados por un canal de comunicaciones. En el canal estas señales sufren la influencia de perturbaciones del ambiente a través del cual se transmiten (ruido), que puede alterar la integridad del mensaje. Finalmente, un bloque receptor puede considerarse que realiza una operación inversa a la del transmisor, buscando recuperar correctamente la información generada inicialmente.

Para que la comunicación se lleve a cabo correctamente, los bloques transmisor y receptor realizan operaciones de control de errores. Los mensajes de K bits de longitud son codificados de distintas maneras, generalmente se agregan N-K bits de redundancia a los K bits de información, por lo que el mensaje final tendrá una longitud total de N bits. Además de esta redundancia, también es posible modular el mensaje para que el ruido afecte en menor medida al mensaje transmitido.

De esta manera, el bloque receptor podría tener 2 funciones. En primera instancia podría detectar la existencia de errores, y solicitar al transmisor el reenvío del mensaje, esperando recibir correctamente el mensaje con la nueva transmisión. Este método requeriría la existencia de un canal de retorno entre el transmisor y el receptor, con una demora asociada, y no se podría garantizar una mejora del canal durante la retransmisión. A este tipo de protocolo se lo denomina Requerimiento de Reconocimiento Automático (ARQ). Por otro lado, el receptor podría ser capaz de deducir el mensaje transmitido a partir de la señal recibida. En este caso se dice que el decodificador trabaja sobre el esquema de corrección de errores, y no se necesitaría un canal de retorno.

Los sistemas de control de errores, denominados de corrección hacia adelante (FEC) pertenecen a la segunda opción. Entre ellos el decodificador busca corregir

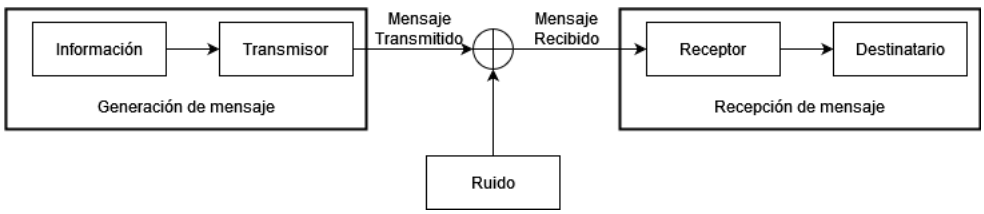


FIGURA 2.1: Sistema de comunicaciones generalizado.

hasta un cierto número de errores en el mensaje recibido. Si el número de bits con errores fuera inferior a cierto límite, el código permitiría recuperar el mensaje original. En cambio, si el mensaje tuviera más errores que los que el decodificador es capaz de corregir, el código colapsaría; el destinatario no tendría entonces forma de conocer íntegramente la información original. La capacidad correctora del código depende de su naturaleza, de las características del canal, y de la tasa de código empleada. Esta capacidad es además inversamente proporcional a la tasa de código. Con esta consideración, desde el punto de vista del control de errores se desearía una tasa lo más baja posible. Sin embargo, existe una situación de compromiso, ya que esto implicaría velocidades de transmisión efectivas más bajas y la comunicación no resultaría eficiente. Los mejores códigos permitirán alcanzar altas velocidades, manteniendo la probabilidad de error más baja.

2.2. Generación de canal.

Simular un sistema de comunicaciones implica pasar la información transmitida por un canal para luego ser decodificada del lado receptor. El canal puede modelarse a través de distintas distribuciones, para ajustarse al modelo de canal en que se quiera realizar la transmisión. Por ejemplo, la distribución uniforme es la más teórica y se utiliza en el canal binario simétrico (BSC), la gaussiana para canales cableados en donde se presenta el ruido blanco gaussiano (AWGN), las de Rayleigh y Rician para canales inalámbricos, etc. Estas distribuciones exigen métodos de generación de muestras de ruido muy precisos, asociadas a generadores de ruido aleatorio que producen dichas muestras a intervalos regulares.

Ha habido diversas propuestas sobre métodos para generación de distribuciones arbitrarias. En el caso de este trabajo se ha elegido un diseño de generación de números aleatorios basado en hardware, que usa la función distribución acumulativa inversa [3] (**ICDF**) para convertir una muestra x de una variable aleatoria uniforme en el rango [0,1), a una muestra de la función densidad de probabilidad deseada a través de $y = F^{-1}(x)$. El gran desafío es justamente el diseño de un algoritmo que evalúe esta función de la manera más precisa posible.

Generalmente se presenta un gran problema en el caso de las ICDFs no lineales, con zonas con derivadas de primer o mayor orden altas, por lo que se deben implementar métodos de segmentación apropiados para su aproximación.

2.2.1. Evaluación de funciones y segmentación

La evaluación de funciones matemáticas es usualmente utilizada en numerosas aplicaciones de comunicaciones, procesamiento digital de señales, gráficos por computadora y computo científico. Ejemplos de tales funciones incluyen funciones elementales como $ln(x)$ y $cos^{-1}(x)$, y funciones compuestas como $-(\frac{x}{2})log_2 x$ y $\sqrt{-ln(x)}$. Usualmente, entornos de software como C ó MATLAB proveen librerías para evaluar funciones en precisión de punto flotante. Sin embargo, las implementaciones en software sobre instrucciones de procesador son demasiado lentas para aplicaciones en tiempo real y/o intensivas numéricamente. El rendimiento de tales aplicaciones depende del diseño de una rápida y precisa unidad de evaluación de funciones en hardware, implementadas usualmente en matrices de compuertas programables en campo (FPGA) o un circuito integrado de aplicación especifica (ASIC).

La evaluación de funciones ha recibido un interés considerable en la comunidad de investigación. En particular, los métodos que involucran polinomios y splines

(polinomios por partes) se han utilizado extensivamente tanto en implementaciones de hardware como software. Las aproximaciones mediante splines se suelen preferir por encima de las aproximaciones sólo polinomiales debido al amplio rango de intercambios en el diseño que ofrecen incluyendo memoria, complejidad computacional y precisión [4].

Como se mencionó en 2.2, el método utilizado será la segmentación jerárquica, el mismo emplea jerarquías las cuales incluyen splines uniformes y splines no uniformes. Este tipo de segmentación es capaz de adaptarse a las no-linealidades de una función, resultando en una reducción significativa en el número de splines comparado con la segmentación uniforme. Cada spline contiene un conjunto de coeficientes de polinomios que se corresponden con una región particular de una función. En este caso, las prioridades de implementación apuntan a reducir al mínimo el retardo de direccionamiento de coeficientes generado por la segmentación. Además, se le permite al diseñador especificar un error requerido $\epsilon(req)$ y obtener automaticamente una segmentación que:

1. Cumpla esta tolerancia.
2. Requiere el mínimo número de segmentos M posible.
3. Permita llegar a una implementación en hardware eficiente.

2.2.2. Métodos de evaluación de funciones

Los métodos de evaluación de funciones pueden clasificarse en métodos iterativos y métodos no iterativos:

- Los métodos iterativos refinan sucesivamente la precisión de salida y son aplicables en implementaciones que requieren precisiones arbitrarias. Sin embargo, usualmente presentan latencias altas y bajos rendimientos, haciéndolos inadecuados para aplicaciones de alta performance.
- Los métodos no iterativos incluyen varios métodos, como las aproximaciones polinómicas, tablas de look-up directas, métodos de adición de tablas y aproximaciones racionales. Las **tablas de look-up directas** son utilizadas en métodos computacionales que requieren entradas de baja precisión. Los **métodos de adición de tablas** usan dos o más tablas de look-up paralelas seguidas por una suma de múltiples operandos. Las **aproximaciones polinómicas** incluyen la evaluación de un polinomio en un intervalo dado. La precisión de la aproximación puede controlarse mediante el grado del polinomio y la elección del intervalo. La **aproximación racional** es una generalización de las aproximaciones polinómicas en la cual la función se aproxima mediante un cociente de dos polinomios. Para un grado limitado de numerador y denominador, permite mayor precisión que las aproximaciones polinómicas, pero debido a la división, la complejidad del circuito crece considerablemente. Los métodos no iterativos se combinan usualmente con segmentación en la cual el rango de entrada se divide en multiples segmentos, cada uno asociado con un spline que contiene un conjunto de coeficientes particulares relacionados con su representación polinomial.

En este caso, se utiliza la aproximación polinomial combinada con segmentación lo que resulta en aproximación polinomial por partes [5]. Esto consiste en aproximar una función continua “f” con uno o más polinomios “p” de grado “d” en un intervalo [a,b). Los polinomios son de la forma:

$$p(x) = c_d x^d + c_{d-1} x^{d-1} + ... + c_1 x + c_0 \tag{2.1}$$

Donde "x" es la entrada. El objetivo es minimizar la distancia $||p - f||$ por lo que se utilizan aproximaciones de polinomios "minimax" (se minimiza el máximo error absoluto). La distancia para una aproximación minimax es:

$$||p - f||_\infty = \max_{a \leq x \leq b} |f(x) - p(x)| \tag{2.2}$$

Este polinomio minimax se obtiene de manera iterativa utilizando el algoritmo de Remez, el cual se utiliza generalmente para determinar coeficientes óptimos para filtros digitales. Por lo que, si se generaliza para multiples segmentos "u"

$$h_i(u_{i-1}, u_i) = \max_{p_i \epsilon P_i u_{i-1} \leq x \leq u_i} |f(x) - p_i(x)|. \; \forall \; i = 1, ..., m. \tag{2.3}$$

Siendo m ≥ 2 el número de segmentos contiguos en los cuales el intervalo [a,b) fue segmentado: $a = u_0 \leq u_1 \leq ... \leq u_m = b$ y P_i el conjunto de funciones p_i cuyos polinomios son de grado igual al grado "d" escogido. Por lo que el error minimax para cada segmento será $e_{max} = e_{max}(u) = \max_{1 \leq i \leq m} h_i(u_{i-1}, u_i)$. Es decir, la aproximación minimax consiste en minimizar e_{max} para todas las particiones "u" del intervalo [a,b).

Dentro de los métodos de segmentación utilizados, el más común es el método uniforme, en el cual todos los segmentos tienen la misma longitud. Además, la elección de la cantidad de segmentos está limitada a potencias de 2, lo que hace que el direccionamiento de los coeficientes sea más sencillo respecto a la segmentación no uniforme. Aunque a diferencia de estos, no permite que las longitudes de segmentos sean ajustadas a las características locales de la función no lineal. La ventaja de la segmentación no uniforme se presenta principalmente con funciones logarítmicas, que es el caso de la ICDF Rayleigh implementada en este proyecto, aunque podría utilizarse cualquier otra función que se desee segmentar. Para clarificar esto, se muestra en la Fig. 2.2 la ICDF de la distribución Rayleigh para una desviación estándar igual a 1, la cual se asocia con la Eq. 2.4. Como se puede ver, para los extremos de la función existen grandes alinealidades por lo que es necesario aplicar la ya mencionada segmentación no uniforme ya que esto se traducirá en una reducción de segmentos necesarios.

$$ICDF(x) = \sqrt{2}\sigma\sqrt{-\log(1 - x)} \tag{2.4}$$

2.2.3. Método de segmentación jerárquica, HSM

Un bloque importante es el segmentador, el cual emplea jerarquías que involucran splines que son funciones especiales definidas por polinomios por partes las cuales contienen un conjunto de coeficientes correspondientes a distintas regiones de la función segmentada. Cada región puede tener una longitud uniforme o la misma puede ser variable. Como se ha dicho, los segmentos de longitud variable pueden adaptarse a no linealidades de una función, lo que permite reducir la cantidad de segmentos requeridos para una misma implementación en comparación con los segmentos de longitud uniforme.

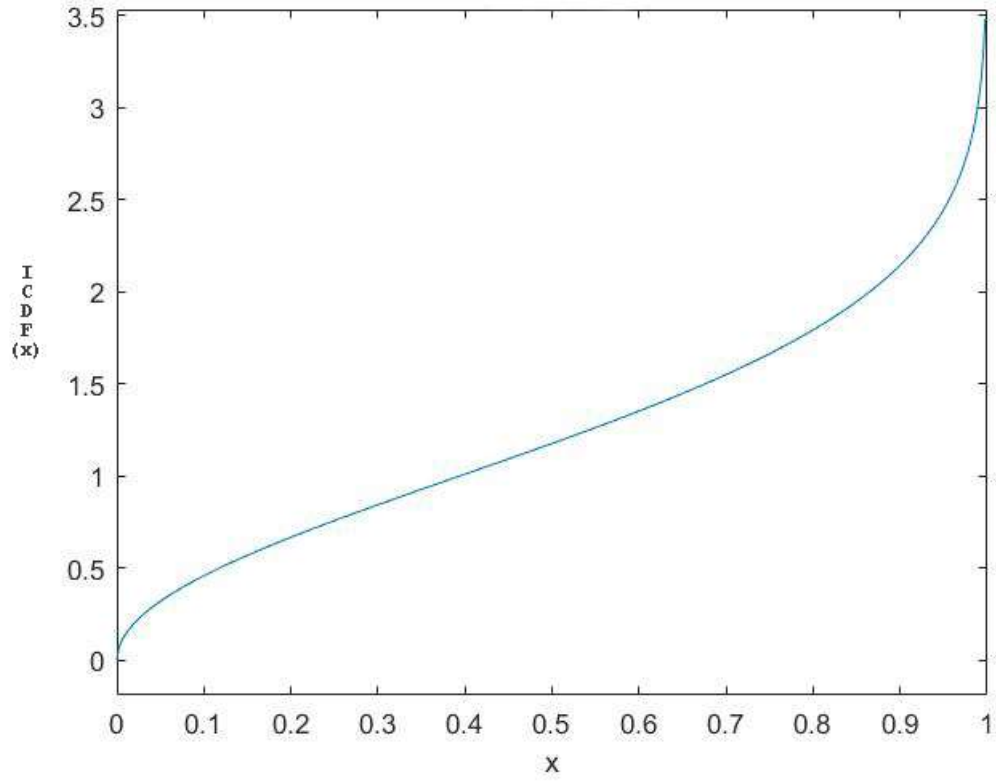


FIGURA 2.2: Función inversa acumulativa de la distribución Rayleigh.

El método HSM provee 4 esquemas de segmentación básicos: segmentación uniforme (US), segmentación en potencias de 2 para el lado izquierdo (P2SL), segmentación en potencias de 2 para el lado derecho (P2SR), segmentación en potencias de 2 para ambos extremos (P2SLR). En el caso de segmentación uniforme US los segmentos son todos del mismo tamaño Fig. 2.3. En P2SL, el tamaño del segmento crece en potencias de 2 desde el inicio del intervalo de entrada hacia el fin del intervalo Fig. 2.4, mientras que en P2SR el tamaño del segmento decrece en potencias de 2 desde el inicio del intervalo hacia el final Fig. 2.5. En P2SLR los tamaños de los segmentos crecen en potencias de 2 hasta el punto medio del intervalo y luego decrecen en potencias de 2 hasta el final Fig. 2.6. Por ej. para un rango de [a,b) y un total de 8 segmentos:

US:

$$[0; \frac{1}{8}; \frac{2}{8}; \frac{3}{8}; \frac{4}{8}; \frac{5}{8}; \frac{6}{8}; \frac{7}{8}; 1] * (b-a) + a$$

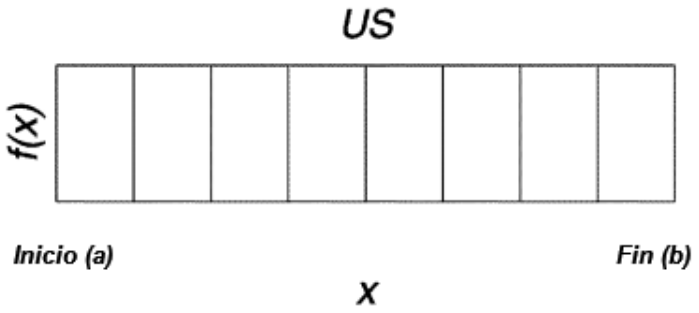


FIGURA 2.3: Ilustración de la segmentación uniforme US

P2SL:

$$[0; 2^{-7}; 2^{-6}; 2^{-5}; 2^{-4}; 2^{-3}; 2^{-2}; 2^{-1}; 1] * (b-a) + a$$

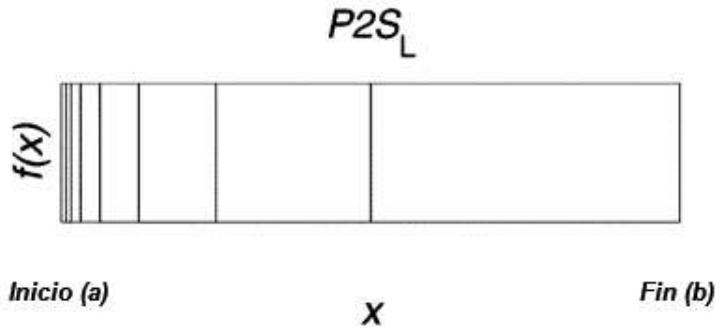


FIGURA 2.4: Ilustración de la segmentación en potencias de 2 concentrada en la izquierda P2SL

P2SR:

$$[0; 2^{-1}; 2^{-1}+2^{-2}; 2^{-1}+2^{-2}+2^{-3}; 2^{-1}+2^{-2}+2^{-3}+2^{-4}; 2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5};$$
$$2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}; 2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-7}; 1] * (b-a) + a$$

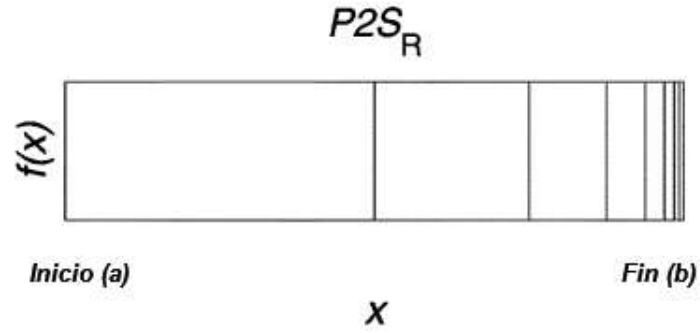


FIGURA 2.5: Ilustración de la segmentación en potencias de 2 concentrada en la derecha P2SR

P2SLR:

$$[0; 2^{-4}; 2^{-3}; 2^{-2}; 2^{-1}; 2^{-1}+2^{-2}; 2^{-1}+2^{-2}+2^{-3}; 2^{-1}+2^{-2}+2^{-3}+2^{-4}; 1] * (b-a)+a$$

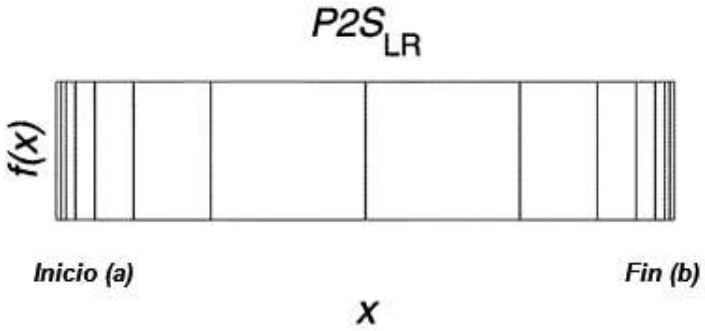


FIGURA 2.6: Ilustración de la segmentación en potencias de 2 para ambos extremos P2SLR

Este método se denomina jerárquico debido a que la segmentación es aplicada recursivamente. En el primer paso, denominado segmentación externa, el intervalo completo es subdividido utilizando uno de los cuatro esquemas mencionados previamente y en el segundo paso, llamado segmentación interna, cada subsegmento puede ser dividido nuevamente, utilizando el mismo, u otro de los cuatro esquemas. En esta implementación, el segundo paso se fija a una segmentación uniforme, ya que por lo investigado en [6] una segmentación de mayor cantidad de niveles no resultaría en una mejora significativa y de esta manera se reduce en gran medida la complejidad de la implementación.

La aplicación de este algoritmo exige ciertos parámetros de entrada: el intervalo a segmentar, el grado del polinomio válido para la aproximación por partes y error absoluto a la salida. Para cada segmento de la segmentación externa, se computan los coeficientes de Chebyshev para el polinomio de aproximación. Si el error de aproximación es mayor a uno previamente establecido, el número de segmentos de la segmentación interna se incrementa en sucesivas potencias de 2 hasta que el error obtenido sea menor que el requerido. Esto se debe a que a medida que se aumenta la cantidad de segmentos, para un mismo orden de polinomio, la interpolación se ajusta en mayor medida a la función que se quiere aproximar debido a que se consideran mayor cantidad de puntos de la función original. Por lo tanto, el error disminuye a medida que crece la cantidad de segmentos. De la misma manera, si se aumenta el orden del polinomio, el mismo coincidirá en mayor medida con la función a segmentar y se tendrá un menor error para la misma cantidad de segmentos. En este caso, se opta por modificar la cantidad de segmentos debido a que la implementación busca optimizar un desarrollo en hardware, y si se modificara el orden de los splines, ocurriría que se tendrían mayor cantidad de multiplicaciones las cuales son costosas a nivel hardware.

La estructura de implementación de la jerarquía se logra a través de memorias que almacenan información de cada nivel de la misma. Por este motivo, se precisa que la entrada x, de B_x bits se divida de acuerdo al número de niveles implicados. Cada partición tendrá un número de segmentos direccionables de acuerdo al tipo de segmentación y al ancho en bits asignado a cada nivel. A mayor cantidad de niveles utilizados, se acercará más al valor óptimo el número de segmentos. Sin embargo,

a mayor cantidad de niveles, aumenta la complejidad del sistema ya que se requieren mayor cantidad de memorias y direccionamientos, lo que aumenta el retardo en tiempo que se requiere para obtener un segmento especifico. Por los experimentos realizados en [6] se verifica que en la mayoría de los casos, una segmentación de dos niveles, es decir una segmentación externa y una interna, es más que suficiente ya que se obtiene un número de segmentos cercano al óptimo y mantiene la complejidad del particionado, junto con el retardo en valores aceptables. Para una segmentación HSM de dos niveles, el valor de entrada x, compuesto por B_x bits, se divide en tres partes: x_0, x_1 y x_2, cada una con su propia cantidad de bits B_{xi}. Los valores de x_0 y x_1 son utilizados para indexar la segmentación externa e interna respectivamente, mientras que x_2 es utilizado para la aritmética polinomial.

La determinación de la jerarquía de segmentación apropiada para una función dada juega un rol muy importante. Elegir la jerarquía equivocada puede resultar en una segmentación ineficiente, derivando en un número de segmentos inecesariamente grande. Una forma de encontrar la mejor jerarquía es aplicar todas las jerarquías de segmentación y tomar la que genere el menor número de segmentos "M". En lugar de esta aproximación, es posible otro método, el cual se inicia calculando los segmentos que presenten un error balanceado de la función dada, es decir todos los segmentos tendrán el mismo error respecto a la función a segmentar. Para la obtención de estos segmentos de error balanceado se utilizó el algoritmo mostrado en el apendice A.1. Luego, para cada uno de los 4 esquemas de segmentación posibles "Λ", se obtiene la varianza que se genera comparando cuáles segmentos del error balanceado, coinciden o se aproximan a los segmentos de cada una de las 4 segmentaciones. De esta manera, el esquema que resulte en la varianza más pequeña es el elegido, ya que una varianza pequeña indica que los segmentos del esquema elegido y los del error balanceado se aproximan de mejor manera a la función deseada. Recordando que la segmentación interna es uniforme y que se busca seleccionar la segmentación externa, se procede a mostrar el algoritmo utilizado en el pseudocódigo 2.1.

CÓDIGO 2.1: Pseudo-código 1 - Selección del mejor esquema de segmentación

```
1  //Parametros de entrada: Intervalo de entrada (a,b),
2                           Segmentos de error balanceado D⃗
3                           Cantidad de elementos en el vector de segmentos de error balanceado D⃗ DElements
4
5  //P2SL
6  Λ⃗(0;:) = [0 ; 2^(-DElements) ; 2^(-DElements+1) ; 2^(-DElements+2) ; ... ; 2^(-3) ; 2^(-2) ; 2^(-1) ; 1]*(b-a)+a
7
8  //P2SR
9  Λ⃗(1;:) = [0 ; 2^(-1) ; 2^(-1) + 2^(-2) ; 2^(-1) + 2^(-2) + 2^(-3) ; ... ;
10 2^(-1) + 2^(-2) +...+ 2^(-DElements + 1) ; 2^(-1) + 2^(-2) +...2^(-DElements + 1) + 2^(-DElements) ; 1]*(b-a)+a
11
12 //P2SLR
13 Λ⃗(2;:) = [0 ; 2^(-DElements/2) ; 2^(-DElements/2+1) ; ... ; 2^(-1) ; 2^(-1) + 2^(-2) ; ... ; 2^(-1) + 2^(-2) + ... +
14 2^(-DElements/2+1) ; 1]*(b-a)+a
15
16 //US
17 Λ⃗(3;:) = [0 ; 1/DElements ; 2/DElements ; ... ; (DElements - 2)/DElements ; (DElements - 1)/DElements ; 1]*(b-a)+a
18
19 for i = 0 to 3 do
20         for j = 0 to DElements do
21                 for k = 0 to length (D⃗ -1) do
22                         if Λ⃗(i,j)≤D⃗(k)<Λ⃗(i,j+1) then
23                                 Λ⃗_h(i,j)++
24                         end if
25                 end for
26         end for
27 end for
```

Otro problema importante es la determinación del número de bits B_{x0} a asignar para la segmentación externa λ_0. Si B_{x0} es demasiado pequeño, habrá una granularidad insuficiente en los segmentos externos al seguir las alinealidades de una función.

Por otro lado, si B_{x0} es muy grande, habrá demasiados segmentos externos y el número total de segmentos M será innecesariamente grande. En la Fig. 2.7 se da un ejemplo de como varía la cantidad de segmentos para la ICDF Rayleigh, a medida que aumenta B_{x0} con una precisión de 2^{-14} y una aproximación realizada mediante un polinomio de orden d = 2. En este caso se usa una segmentación PS2SLR como externa y US como interna, resultando en un B_{x1} máximo de 3 bits.

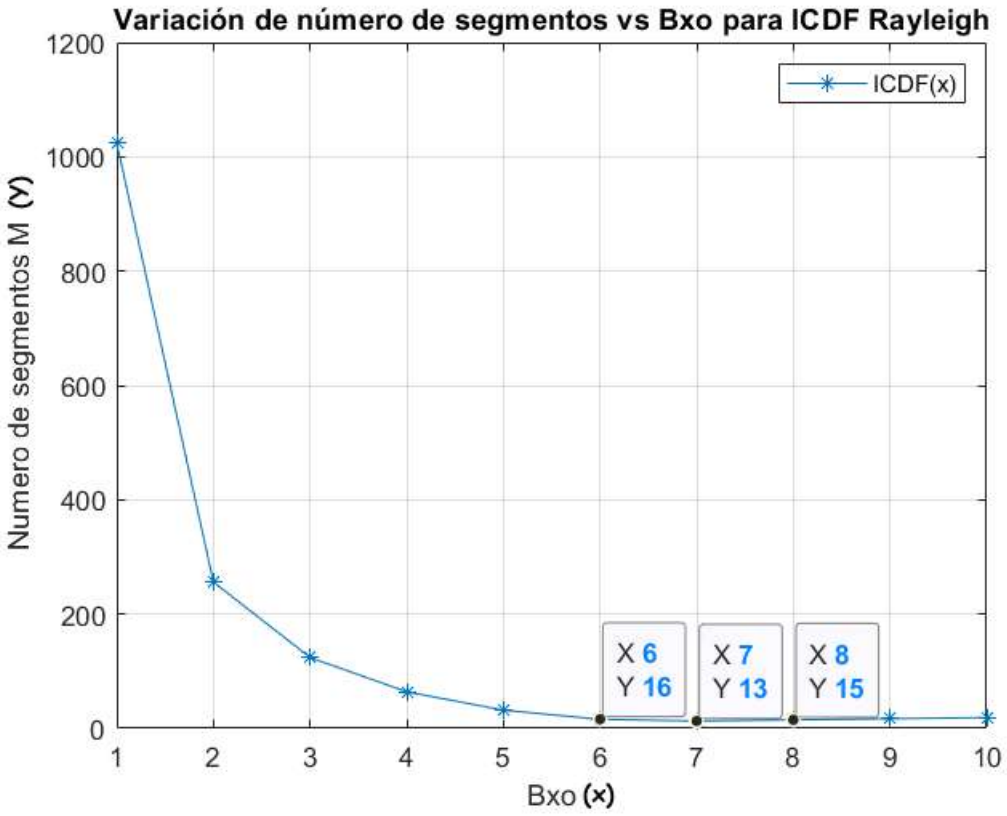


FIGURA 2.7: Número de segmentos M vs B_{x0} para ICDF Rayleigh con un error máximo de 2^{-14} y orden d = 2. Se alcanza el mínimo para B_{x0} = 7

De este modo la cantidad de bits de x_0 y x_1 quedaran determinados entonces por el orden del polinomio utilizado para interpolar y el error requerido.

Para obtener la Fig. 2.7 se utilizó el algoritmo dado por el pseudo-código 2.2, donde se especifican los parámetros de entrada necesarios.

CÓDIGO 2.2: Pseudocódigo 2 - Método de segmentación jerárquica

```
1  //Parametros de entrada: Funcion f a segmentar,
2                           Unidad en el ultimo lugar (ulp),
3                           Intervalo de entrada (a,b),
4                           Grado de polinomio d,
5                           Error requerido ϵ_req
6
7  //Se comienza por seleccionar la jerarquia de segmentacion H
8
9  D⃗ = GetBalancedErrorBoundaries(f,a,b,d,ulp,ϵ_req
10 H = SelectHierarchy(D⃗) //Algoritmo del pseudo-codigo anterior.
```

```
11
12 //Luego se procede a calcular el Bx0 optimo
13
14 λ⃗ = []
15 Bx0 = 0
16 M' = ∞
17 while 1 do
18         s0  = GetNumberOfλ0Segs(H,Bx0)
19         (λ⃗0) = Getλ Boundaries(H,Bx0,(a,b))
20         ROM0 = []
21         ROM1 = []
22         offset = 0
23         for i = 0 to s0 - 1 do
24                 A = λ⃗0(i)
25                 B = λ⃗0(i+1)
26                 (εmax,Coeffs) = Chebyshev(f,d,A,B,ulp)
27                 if εmax > εreq
28                 λ0SegSize = B - A
29                 Bx1 = 0
30                 while εmax > εreq
31                         Bx1 = Bx1 + 1
32                         λ1SegSize = λ0SegSize/2^Bx1
33                         λ1 = []
34                         λ1Coeffs⃗ = []
35                         s1 = 2^Bx1
36                         for j = 0 to s1 - 1 do
37                                 A = λ⃗0(i) + λ1SegSize * j
38                                 B = A + λ1SegSize
39                                 (εmax,Coeffs) = Chebyshev (f,d,A,B,ulp)
40                                 if εmax > εreq then
41                                 break
42                                 end if
43                                 λ1(j) = A
44                                 λ1Coeffs⃗(j) = Coeffs
45                         end for
46                 end while
47                 ROM0 = [ROM0; Bx1offset]
48                 offset = offset + 2^Bx1
49                 else
50                         λ⃗1 = A
51                         ROM1 = Coeffs
52                 end if
53                 λ⃗ = [λ⃗;λ⃗1]
54                 ROM1 = [ROM1;λ1Coeffs⃗]
55         end for
56         M = length(λ⃗)
57         if M > M' then
58                 M = M'
59                 λ⃗ = λ⃗'
60 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ROM0␣=␣ROM0'
61                 ROM1 = ROM1'
62                 break
63         else
```

```
64                  B_x0++
65                  M' = M
66                  λ⃗ ,␣=␣ λ⃗
67                  ROM0' = ROM0
68                  ROM1' = ROM1
69          end if
70  end while
```

Basándose en el algoritmo presentado anteriormente, primero se determina la segmentación apropiada (lineas 9 y 10). A continuación, se realiza la segmentación jerárquica mientras se busca el B_{x0} óptimo que minimice M. Inicialmente $B_{x0} = 0$, lo que se corresponde con segmentación uniforme (US). B_{x0} se incrementa hasta que se encuentra la segmentación que da el mínimo número de segmentos M (línea 14 a 70). El núcleo del algoritmo se encuentra en el lazo for de la línea 23 a la línea 55. Por cada segmento en la segmentación externa, se computan los coeficientes de Chebyshev para el polinomio de aproximación. Si el error de la aproximación ϵ_{max} es muy alto, el número de segmentos de la segmentación interna se incrementa en sucesivas potencias de 2 hasta que el ϵ_{max} de todos los segmentos es $\leq \epsilon_{req}$. Este proceso se realiza para todos los segmentos externos. El resultado final es el número de segmentos totales M, el vector $\overrightarrow{\lambda}$ que contiene los límites de los segmentos, ROM0 que se utiliza para indexar ROM1, y ROM1 que contiene los coeficientes de cada segmento.

De este modo, el algoritmo de segmentación requiere de los siguientes parámetros: la función a segmentar, el intervalo de entrada, el grado del polinomio d que realiza la aproximación por partes, el error requerido a la salida ϵ_{req} y la unidad en último lugar que se corresponde con el mínimo valor que se quiere representar, utilizando o no el máximo de bits disponibles, por ejemplo si se utilizan 32 bits como máximo, entonces ulp = 2^{-32}. Con este algoritmo se obtienen dos memorias ROM (ROM0 y ROM1) como salidas. ROM0 almacena B_{x1} y un offset correspondiente a cada segmento del primer nivel. El offset es simplemente el número de filas en ROM1 previo a la fila en ROM1 que se corresponde al segmento de primer nivel actual (B_{x1}). Para clarificar lo previamente mencionado, se presenta la tabla 2.1 un ejemplo donde se realiza una segmentación de dos niveles del tipo US. En este ejemplo, la segmentación externa se hace considerando $B_{x0} = 3$, es decir 8 segmentos de igual tamaño. Luego, Bx1 se toma en el rango de 0 a 3, de tal manera que cada segmento externo se particiona en 1, 2, 4 u 8 segmentos internos.

Indice	0	1	2	3	4	5	6	7
B_{x1}	0	1	2	2	3	1	0	1
offset	0	1	3	7	11	19	21	22

CUADRO 2.1: Variación de offset en función de B_{x1}

Como se puede ver, offset toma el máximo valor que nos permite calcular el B_{x1} del nivel previo en potencias de 2, es decir offset = $2^{B_{x1}}$. Por ejemlo, si $B_{x1} = 0$ –>Offset = $2^0 = 1$, y si $B_{x1} = 2$ –>Offset = $2^2 = 4$. Además, el offset se se acumula nivel a nivel, es decir, si offset en el nivel previo era 1 y en este nivel $B_{x1} = 1$ –>offset = 3 = 1 + 2.

2.2.4. Indexación de segmentos

Siendo B_x la cantidad de bits de x, al usar dos niveles de segmentación HSM, se divide x en tres: x_0, x_1, x_2. (x_0, x_1) indexan la segmentación externa e interna respectivamente, mientras que x_2 se usa para la aritmética polinomial. La dirección de los

segmentos de la primera partición u segmentación externa x_0, se obtiene detectando la cantidad de bits más significativos (MSB) en cero para los segmentos que presentan un cero en la posición más significativa, o el número de bits en alto para los que tienen un MSB igual a uno.

En la Fig. 2.8 y en la tabla 2.2 se muestra a través de un ejemplo el concepto de indexación con una segmentación externa P2SL. Para un ancho de bits totales Bx= 6, se toma una segmentación externa B_{x0} de 4 bits y una interna B_{x1} de 1 bit donde se tienen una cantidad de 5 segmentos externos ya que con P2SL, como se explicó previamente, se aumenta en uno la cantidad de segmentos cuando el bit no nulo se desplaza hacia la izquierda, y al dividir estos en 2, debido a que se utiliza segmentación uniforme y $B_{x1} = 1$, se obtienen 10 segmentos totales.

Así, B_{x0} da el número de bits utilizados para indexar los segmentos de la primera partición. $\hat{x}_0$ es el conjunto de bits que permanece constante dentro de un dado segmento y se ubican a la izquierda de la columna sombreada en la Fig. 2.8. B_{x1} es el ancho de bits adyacente que se usa para la siguiente partición y que se encuentra a la derecha de $\hat{x}_0$, por lo que el número de bits correspondientes al segundo nivel depende del valor de x_0, dado que x_0 determina el valor de $B_{\hat{x}0}$. Por ejemplo, para los segmentos 4 y 5, $B_{x0} = 4$, mientras que en este caso, como se encuentra desplazado el bit que define el segmento, $B_{\hat{x}0} = B_{x0} - Primerbitnonulo$ Y cómo este bit se encuentra en la tercera posición de izquierda a derecha en la Fig. 2.8, se tiene un valor de $B_{\hat{x}0} = 4 - 2 = 2$, por lo que se utilizaran los 3 bits más significativos o los tres primeros bits de izquierda a derecha para indexar el segmento externo. Lo que permite que B_{x1} sea el bit consecutivo al último tomado para indexar al segmento externo, es decir el cuarto bit más significativo ó el cuarto bit de izquierda a derecha. De esta manera, los bits restantes se utilizan para representar a X_2.

Como se mencionó previamente, el generador produce dos tablas: ROM0 y ROM1. ROM0 se precisa para el cálculo de las direcciones de ROM1 que almacena los coeficientes polinomiales para cada segmento. ROM0 almacena B_{x1} y el offset correspondiente a cada segmento externo, que es el número de filas de ROM1 previo a la fila en ROM1 correspondiente al segmento externo actual.

Se debe lograr un balance entre orden de polinomio y cantidad de segmentos, ya que si se tiene un mayor orden de polinomio, la interpolación posterior resultará en mayor número de multiplicaciones por valor generado. En cambio, a mayor número de segmentos, se requerirán ROMs más grandes para almacenar e indexar los mismos.

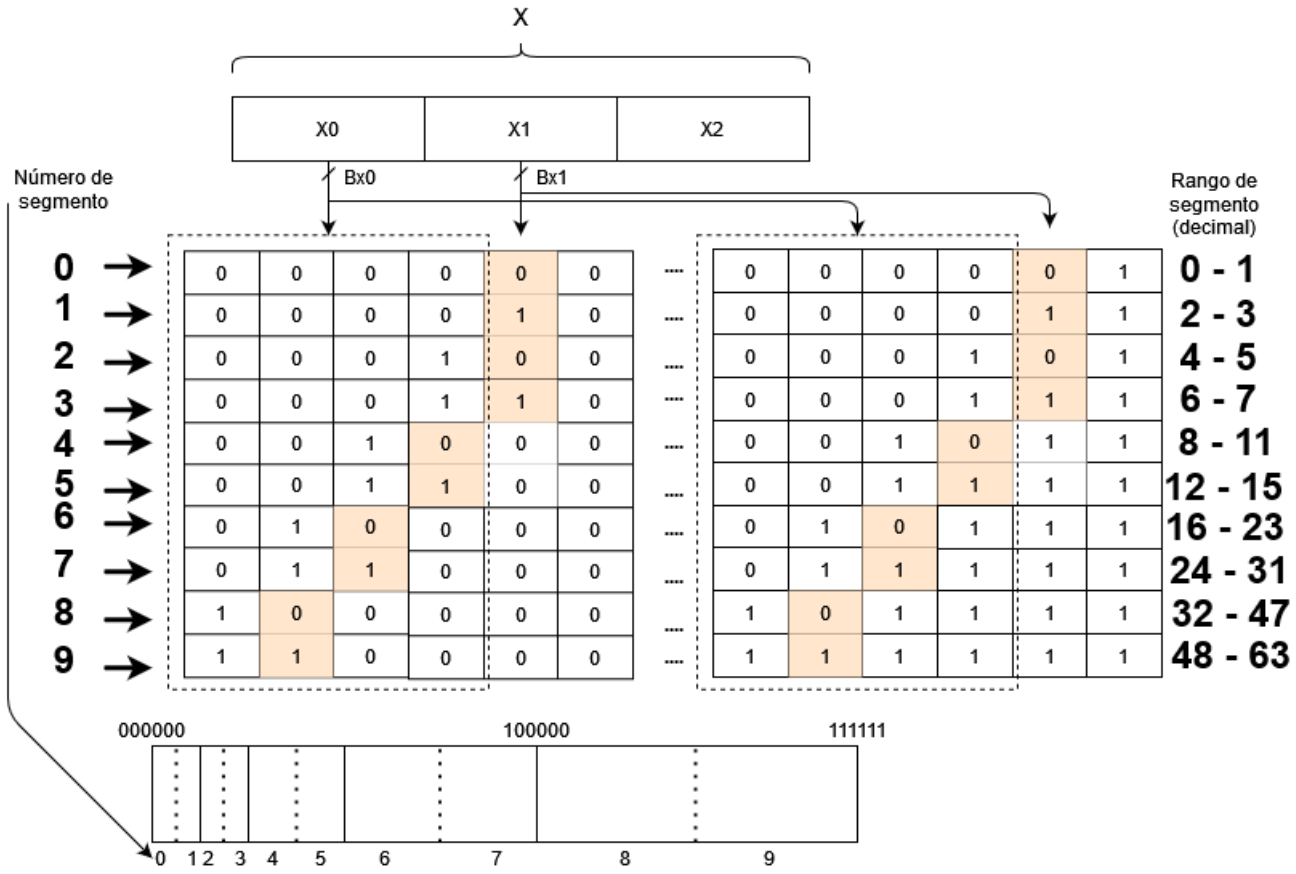


FIGURA 2.8: Rango de segmentos para un Bx = 6 con una segmentación externa B_{x0}= 4 y B_{x1} = 1.

Número de segmento	Rango (binario)	Rango (decimal)
0	000000-000001	0-1
1	000010-000011	2-3
2	000100-000101	4-5
3	000110-000111	6-7
4	001000-001011	8-11
5	001100-001111	12-15
6	010000-010111	16-23
7	011000-011111	24-31
8	100000-101111	32-47
9	110000-111111	48-63

CUADRO 2.2: Rango de segmentos para Bx = 6

2.2.5. Evaluación polinomial

Por último, resta realizar la evaluación polinomial. La misma se realiza utilizando el método de Horner el cual es un algoritmo que permite calcular el resultado de un polinomio, para un determinado valor de x. Este procedimiento permite evaluar funciones polinómicas de forma monomial, de manera que se reducen la cantidad de operaciones necesarias para llegar al resultado. Siendo el grado del polinomio "d" una resolución por sustitución requiere hasta $d^2 + d$ multiplicaciones y d sumas,

mientras que el algoritmo de Horner sólo requerirá d sumas y d multiplicaciones, ya que el resultado se obtiene de la siguiente forma:

$$y = ((C_d x + C_{d-1})x + ...)x + C_0 \tag{2.5}$$

Donde x es la entrada, d es el orden como se mencionó previamente y $C_{0,...,d}$ son los coeficientes del polinomio. Sea $x_{0,...,1}$ el conjunto de bits correspondientes a x_0 y x_1, y $B_{x0,...,1}$ el ancho de bits de este conjunto. x_2 debe ser escalado para ocupar el rango [0,1) ya que todas las ICDF presentan ese rango de entrada. Si x = [0,1), esto implica enmascarar los bits correspondientes a la indexación ($x_{0,...,1}$) y desplazar x $B_{x0,...,1}$ posiciones hacia la izquierda. Pero esto tiene un problema aparejado, ya que los coeficientes fueron calculados bajo la premisa que x sería utilizada para la evaluación polinomial y no x_2 como este caso. Considerando un segmento obtenido con un polinomio de orden 2, x_2 queda determinado por:

$$x_2 = (x - x_{0,...,1}) * 2^{B_{x0,...,1}} \tag{2.6}$$

Reordenando la ecuación, se tiene

$$x = \frac{x_2}{2^{B_{x0,...,1}}} + x_{0,...,1} \tag{2.7}$$

Y como sabemos, una ecuación polinomial de grado 2 tiene la forma:

$$y = C_2 x^2 + C_1 x + C_0 \tag{2.8}$$

Expresión que, combinada con (2.7) queda:

$$y = \frac{C_2}{2^{2B_{x0,...,1}}} x_2^2 + \frac{C_1 + 2C_2 x_{0,...,1}}{2^{B_{x0,...,1}}} x_2 + C_2 x_{0,...,1}^2 + C_1 x_{0,...,1} + C_0 \tag{2.9}$$

De los términos de orden 0, 1 y 2 se obtienen los nuevos coeficientes polinomiales, los cuales son:

$$C_2' = \frac{C_2}{2^{2B_{x0,...,1}}} \tag{2.10}$$

$$C_1' = \frac{C_1 + 2C_2 x_{0,...,1}}{2^{B_{x0,...,1}}} \tag{2.11}$$

$$C_0' = C_2 x_{0,...,1}^2 + C_1 x_{0,...,1} + C_0 \tag{2.12}$$

De esta manera, se obtiene el valor correcto de "y" evaluando la ecuación con x_2.

2.3. Códigos Reed-Solomon

2.3.1. Clasificación de los códigos Reed-Solomon

Los códigos Reed Solomon son códigos en bloque. En los códigos bloque, el mensaje transmitido se divide en bloques de datos y a cada bloque se le agrega una protección de información mediante símbolos llamados de chequeo o paridad, conformando el conjunto una palabra de código. A su vez se trata de códigos sistemáticos, lo que significa que el proceso de codificación no altera los símbolos del mensaje y los símbolos de protección se suman como una parte extra del bloque. En la Fig. 2.9 se presentan estos conceptos.

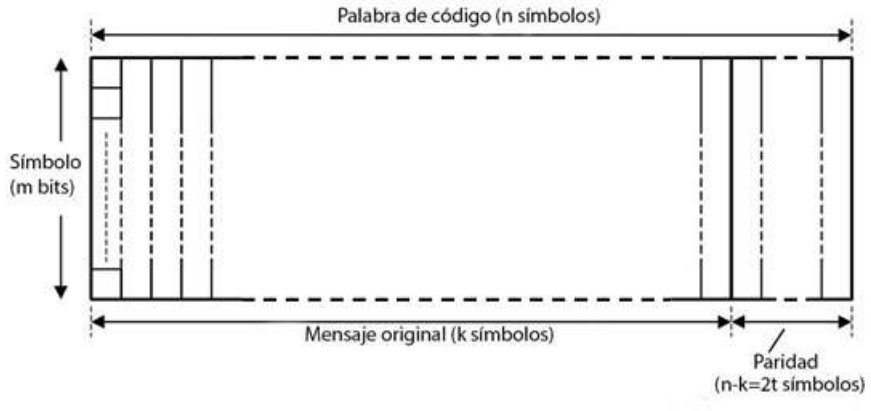


FIGURA 2.9: Definición de los mensajes para códigos Reed-Solomon.

Se trata de códigos **lineales**, ya que sumar dos palabras de código produce otra palabra de código y la palabra todos "0s" pertenece al código, y **cíclicos**, dado que un desplazamiento cíclico de cualquier palabra de código produce otra palabra del código. También, un código RS puede describirse como un código (n,k) donde n es la longitud de la palabra codificada y k es el número de símbolos de información en el mensaje. Además, se verifica $n \leq 2^m - 1$, donde m es el número de bits de un símbolo. Existen (n-k) símbolos de paridad y sólo t símbolos erróneos pueden ser corregidos en un bloque, y se cumple:

- $t = (n-k)\ /2$ para $(n-k)$ par
- $t = (n-k-1)\ /2$ para $(n-k)$impar

2.3.2. Campo de Galois

Elementos del campo de Galois

Debido a que los códigos RS son implementados en un campo matemático especial, conviene recordar algunas definiciones del mismo. Un Campo de Galois consiste en un conjunto de elementos (números), basados en un elemento primitivo, usualmente denotado como α, que toma los valores:

$$0, \alpha^0, \alpha^1, \alpha^2, \alpha^3, \ldots.., \alpha^{N-1} \tag{2.13}$$

que conforma un conjunto de 2^m elementos, donde $N = 2^m - 1$, y el campo se referencia como $GF\,(2^m)$. El valor de α es usualmente elegido igual a 2, aunque se pueden usar otros valores. Habiendo escogido α, las potencias superiores pueden obtenerse multiplicando por α en cada paso. Pero las reglas de operación no son las usuales, ya que cambia la multiplicación y división que usamos habitualmente en las operaciones decimales.

Además, cada elemento puede representarse como una expresión polinomial.

$$a_{m-1}x^{m-1} + ... + a_1 x + a_0 \tag{2.14}$$

Donde los coeficientes a_{m-1} hasta a_0 toman los valores 0 y 1. Por ejemplo, la representación en el campo de Galois con 16 elementos (GF(16), por lo que $m = 4$ es:

$$a_3x^3 \ + \ a_2x^2 \ + a_1x^1 + a_0x^0 \tag{2.15}$$

Adición y sustracción en el campo de Galois

Así, cuando se suman dos elementos de campo, en realidad se están sumando dos polinomios:

$$\begin{aligned}\left(a_{m-1}x^{m-1}+...+a_2x^2+a_1x^1+a_0x^0\right)+\left(b_{m-1}x^{m-1}+...+b_2x^2+b_1x^1+b_0x^0\right)\\ =c_{m-1}x^{m-1}+...+c_2x^2+c_1x^1+c_0x^0\end{aligned} \tag{2.16}$$

donde: $c_i = a_i + b_i$ para $0 \leq i \leq m-1$ Sin embargo, los coeficientes pueden tomar solamente los valores 0 y 1, entonces:

- $c_i = 0$ para $a_i = b_i$
- $c_i = 1$ para $a_i \neq b_i$

De este modo, los elementos en un campo de Galois se suman realizando la suma de módulo 2 de los coeficientes o en forma binaria, produciendo la OR-exclusiva bit a bit de los 2 números representados en formato binario. Por ejemplo, podemos sumar los elementos $x^3 + x$ y x^3+x^2+1 para producir $x^2 + x + 1$ como números binarios: $1010 + 1101 = 0111$ que en decimales es $10+13 = 7$. En el caso de la resta, se aplican las mismas reglas.

El polinomio generador de Campo

Una parte importante de la definición de un campo finito, y por lo tanto de un código Reed-Solomon, es el polinomio generador de campo ó polinomio primitivo $p(x)$. Se trata de un polinomio de grado m el cual es irreducible, esto es, un polinomio no factorizable. Forma parte del proceso de multiplicar dos elementos de campo juntos. Para un campo de Galois, de un tamaño particular, a veces se debe elegir entre dos polinomios de manera que no se generen resultados erróneos, ya que de otra manera los elementos generados a partir del mismo podrían no pertenecer al campo de Galois utilizado. Por ejemplo, para $GF(16)$ el polinomio de la Ec 2.17 es irreducible y será el utilizado como ejemplo en las próximas secciones:

$$p(x) = x^4 + x + 1 \tag{2.17}$$

Una alternativa posible al mismo sería:

$$p(x) = x^4 + x^3 + 1 \tag{2.18}$$

Construcción del campo de Galois

El interés principal en la adopción de un polinomio primitivo, es que todos los elementos no nulos del campo de Galois correspondiente pueden construirse considerando el hecho de que el elemento primitivo α es una raíz del polinomio generador de campo. Entonces:

$$p(\alpha) = 0 \rightarrow \alpha^4+\alpha+1 = 0 \tag{2.19}$$

ó $\alpha^4 = \alpha + 1$. A partir de esta condición se pueden generar los elementos del campo, teniendo en cuenta que la suma y la diferencia se definen como las operaciones ya presentadas para el campo de Galois. Así, a partir de la Ec. 2.19 es posible obtener el resto de los elementos del campo $GF(16)$con las potencias de α, usando $\alpha + 1$ para reemplazar el elemento α^4.

Siguiendo este procedimiento para los 15 elementos del $GF(16)$ se obtiene la Tabla 2.3.

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

CUADRO 2.3: Elementos de campo para un GF(16) con $p(x) = x^4 + x + 1$

Multiplicación y división en el campo de Galois

Así como la suma y la resta tienen reglas especiales, la multiplicación y la división también se puede pensar desde el punto de vista polinomial.

Dado que la multiplicación directa de dos polinomios de grado $m-1$ resulta en un polinomio de grado $2m-2$, que no es un elemento válido de $GF(2^m)$, se debe considerar la operación módulo el polinomio generador de campo $p(x)$para poder obtener un elemento del mismo como resultado. El producto módulo $p(x)$ se obtiene al dividir el polinomio resultante del producto por $p(x)$ y tomando el resto, lo que asegura que el resultado es siempre de grado $m-1$ o menor, y así un elemento de campo válido. Por ejemplo, si multiplicamos los valores 10 y 13 del $GF(16)$, representados por su expresión polinomial, se obtiene:

$$(x^3+x)\left(x^3+x^2+1\right) = x^6+x^5+x^3+x^4+x^3+x = x^6+x^5+x^4+x \qquad (2.20)$$

Para completar la multiplicación, el resultado del producto debe dividirse por x^4+x+1. La división de un polinomio por otro es similar a la división larga de polinomios. Así, el procedimiento consiste en multiplicar el divisor por un valor para hacerlo del mismo grado que el dividendo y luego se restan ambos términos, operación que en el caso de elementos de campo de Galois es lo mismo que la suma. Este proceso se repite utilizando el resto de cada etapa, hasta que los términos del dividendo se agotan o resultan en un resto de un orden menor al divisor. Por lo tanto, el cociente es la serie de valores utilizados para multiplicar el divisor en cada etapa, más el resto de la etapa final.

Esto puede verse más claramente organizando los términos de los polinomios en columnas de acuerdo a su exponente y luego el cálculo puede realizarse sumando 0 ’s y 1’ s.

	x^6	x^5	x^4	x^3	x^2	x^1	x^0
dividendo:	1	1	1	0	0	1	0
divisor * x^2:	1	0	0	1	1		
	-	-	-	-	-		
		1	1	1	1	1	
divisor * x:		1	0	0	1	1	
		-	-	-	-	-	
			1	1	0	0	0
divisor * 1:			1	0	0	1	1
			-	-	-	-	-
				1	0	1	1

Entonces, el cociente es $x^2 + x + 1$ y el resto, el cual es el producto de 10 y 13 que se calculó previamente, es $x^3 + x + 1$ (1011 en binario o 11 en decimal). Por lo que, podría escribirse:

$$10 * 13 = 11 \tag{2.21}$$

Otro método de multiplicación útil es el de los logaritmos que consiste en tomar la representación en forma de índices $10 = \alpha^9$ y $13 = \alpha^{13}$—>$10\ x13 = \alpha^9 x\ \alpha^{13} = \alpha^{(9+13) mod\ 15} = \alpha^{(22) mod\ 15} = \alpha^7$. Observando la Tabla 1, se obtiene que $\alpha^7 = 11$.

Una pequeña desventaja del método logarítmico es que el elemento de campo 0 no puede ser representado en la forma de índice. Se debe sensar entonces la presencia del valor cero y forzar el resultado correspondientemente.

La técnica logarítmica también puede ser utilizada para la división, por ejemplo:

$$11 \div 10 = \alpha^7 \div \alpha^9 = \alpha^{(7-9)\ mod\ (15)} = \alpha^{(-2)\ mod\ (15)} = \alpha^{13} = 13 \tag{2.22}$$

Sin embargo, la división de dos elementos de campo se realiza usualmente mediante la multiplicación del inverso del divisor por el número a dividir. La inversa de un elemento de campo se define como el valor que multiplicado por el elemento de campo produce un valor de 1 ($= a^0$). Es posible calcular la inversa de los elementos de campo usando la Tabla 1. Por ejemplo, $10 = \alpha^9$, entonces su inversa es $\alpha^{(-9)\ mod\ (15)} = \alpha^6 = 12$.

Escribiendo la operación en forma decimal se tiene $11 \div 10 = 11 * 12$, y luego el producto puede calcularse mediante cualquiera de los métodos mencionados previamente.

2.3.3. Construcción de un código Reed-Solomon

Los valores de los símbolos de mensaje y paridad de un código Reed-Solomon son elementos de un campo de Galois. Por este motivo, para un código basado en símbolos de m-bits, el campo de Galois tiene 2^m elementos.

El polinomio generador de código

Un código $RS\ (n,k)$ sobre un campo $GF(q)$ con capacidad de corrección t se construye formando el polinomio generador de código $g(x)$. Siendo α un elemento primitivo de $GF(q)$, sus potencias $\alpha,\ \alpha^2, ...,\ \alpha^{2t}$ son las raíces de $g(x)$. Dado que α^ies un elemento de $GF(q)$, su polinomio mínimo $\phi_i(x) = \left(x + \alpha^i\right)$. De este modo, se verifica:

$$g(x) = \left(x+\alpha^{b}\right)\left(x+\alpha^{2}\right).....\left(x+\alpha^{2t}\right) = g_0 + ... + g_{2t-1}X^{2t-1} + X^{2t} \tag{2.23}$$

Debe notarse que $g(x) = 0$ cuando $x = \alpha^i$. A su vez, las potencias α^ison raíces de $X^{q-1}+1$, es decir que $g(x)$divide a este polinomio, generando un código de longitud $n = q-1$, con exactamente $2t$ símbolos de paridad, siendo su distancia mínima $d_{mn} \geq 2t+1$, siendo $g(x)$un polinomio código con $2t+1$términos. Como su peso es $2t+1$, la distancia mínima es exactamente ese valor. Resumiendo, el código RS resultante cumplirá $n = q-1,\ (n-k) = 2t, d_{mn} = 2t+1$.

Ejemplos específicos son de gran ayuda para obtener un entendimiento completo del proceso que involucra la codificación y decodificación RS. Por ejemplo, para un código RS $(15,11)$, el bloque de longitud es de 15 símbolos, 11 de los cuales son símbolos de información y los 4 restantes son los símbolos de paridad. Debido a que $t = 2$, el código puede corregir errores en hasta 2 símbolos en 1 bloque. Sustituyendo por n en $n = 2^m - 1$, el valor de $m = 4$, por lo que cada símbolo consiste en una palabra de 4 bits y el código está basado en un campo de Galois con 16 elementos. En este caso se utilizará el polinomio generador de campo de la Ec. 2.15 y por lo tanto, los elementos de campo serán los mostrados en la Tabla 2.3.

El polinomio generador de código para corregir hasta 2 palabras de error, requiere 4 elementos consecutivos del campo como raíces, por lo que se puede elegir:

$$g(x) = \left(x+\alpha^{0}\right)\left(x+\alpha^{1}\right)\left(x+\alpha^{2}\right)\left(x+\alpha^{3}\right) = (x+1)(x+2)(x+4)(x+8) \tag{2.24}$$

Aplicando la propiedad distributiva en la forma polinomial:

$$\begin{aligned} g(x) &= (x+1)(x+2)(x+4)(x+8) = \left(x^2+3x+2\right)(x+4)(x+8) = \\ &= (x^3+7x^2+14x+8)(x+8) = x^4+15x^3+3x^2+x+12 \end{aligned} \tag{2.25}$$

Expresión que puede representarse como:

$$g(x) = x^4\alpha^0 + x^3\alpha^{12} + x^2\alpha^2 + x\alpha^0 + \alpha^6 \tag{2.26}$$

2.3.4. Codificación Reed-Solomon

El mensaje en forma de polinomio

Los k símbolos de información que forman el mensaje para ser codificado como un bloque pueden representarse por un polinomio $M(x)$de orden $k-1$, donde M_{k-1} es el primer símbolo del mensaje, por lo tanto:

$$M(x) = M_{k-1}x^{k-1} + ... + M_1x + M_0 \tag{2.27}$$

donde cada uno de los coeficientes M_{k-1}, ..., M_1,M_0 es un símbolo de mensaje de m-bits, esto es, un elemento de GF(2^m).

Para codificar el mensaje, el polinomio de mensaje primero es multiplicado por x^{n-k} y el resultado es dividido por el polinomio generador, $g(x)$. La división produce un cociente $q(x)$y un resto $r(x)$, donde $r(x)$ es hasta de grado $n-k-1$.

$$\frac{M(x) * x^{n-k}}{g(x)} = q(x) + \frac{r(x)}{q(x)} \tag{2.28}$$

Habiendo producido $r(x)$ por división, la palabra de código transmitida $T(x)$ puede formarse combinando $M(x)$ y $r(x)$ como sigue:

$$\begin{aligned} T(x) &= M(x) * x^{n-k} + r(x) \\ &= M_{k-1} * x^{n-1} + ... + M_0 * x^{n-k} + r_{n-k-1} * x^{n-k-1} + ... + r_0 \end{aligned} \tag{2.29}$$

Como se puede apreciar, la palabra de código es producida en la forma sistemática requerida.

Bases para la corrección de errores

Sumar el resto $r(x)$, asegura que el polinomio del mensaje codificado siempre sea exactamente divisible por el polinomio generador. Esto puede verse claramente al multiplicar la Ec. 2.28 por $g(x)$:

$$M(x) \ * \ x^{n-k} = g(x) * q(x) + r(x) \tag{2.30}$$

Reordenando se llega a:

$$M(x) \ * \ x^{n-k} + r(x) = g(x) * q(x) \tag{2.31}$$

donde el lado izquierdo de la igualdad es la palabra de código transmitida $T(x)$, y el lado derecho tiene a $g(x)$ como factor. Además, debido a que se eligió un polinomio generador que consiste de cierto número de factores, cada uno de estos es también un factor del polinomio de mensaje codificado y éste será divisible por los mismos con resto nulo. Si esto no ocurre, es decir si la división resulta en un resto distinto de 0, se puede concluir que el mensaje recibido presenta un error, ya que no forma parte de las palabras del código.

Ejemplo de codificación

A modo de ejemplo, se puede elegir un mensaje que consiste en 11 símbolos de 4 bits para el código (15,11). Por ejemplo, los valores 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 los cuales se desea codificar. Estos valores están representados por un mensaje polinomial:

$$x^{10} + 2x^9 + 3x^8 + 4x^7 + 5x^6 + 6x^5 + 7x^4 + 8x^3 + 9x^2 + 10x + 11 \tag{2.32}$$

Ahora, se desplaza este polinomio, multiplicando por x^4 para dar lugar a los 4 símbolos de paridad:

$$x^{14} + 2x^{13} + 3x^{12} + 4x^{11} + 5x^{10} + 6x^9 + 7x^8 + 8x^7 + 9x^6 + 10x^5 + 11x^4 \tag{2.33}$$

Luego este polinomio es dividido por el polinomio generador de código, para producir los cuatro símbolos de paridad, los cuales son el resto que se obtiene de la operación.

División polinomial

En el proceso de división, a cada paso el generador polinomial es multiplicado por un factor, para que el término más significativo sea el mismo que el resto del paso previo. Al restar, el término más significativo desaparece y un nuevo resto se forma. El proceso de división se puede representar entonces de la siguiente manera:

	x^{14}	x^{13}	x^{12}	x^{11}	x^{10}	x^9	x^8	x^7	x^6	x^5	x^4	x^3	x^2	x^1	x^0
	1	2	3	4	5	6	7	8	9	10	11	0	0	0	0
$*x^{10}$	1	15	3	1	12										
	-	-	-	-	-										
		13	0	5	9	6									
$*13x^9$		13	7	4	13	3									
		-	-	-	-	-									
			7	1	4	5	7								
$*7x^8$			7	11	9	7	2								
			-	-	-	-	-								
				10	13	2	5	8							
$*10x^7$				10	12	13	10	1							
				-	-	-	-	-							
					1	15	15	9	9						
$*1x^6$					1	15	3	1	12						
					-	-	-	-	-						
						0	12	8	5	10					
$*0x^5$						0	0	0	0	0					
						-	-	-	-	-					
							12	8	5	10	11				
$*12x^4$							12	8	7	12	15				
							-	-	-	-	-				
								0	2	6	4	0			
$*0x^3$								0	0	0	0	0			
								-	-	-	-	-			
									2	6	4	0	0		
$*2x^2$									2	13	6	2	11		
									-	-	-	-	-		
										11	2	2	11	0	
$*11x$										11	3	14	11	13	
										-	-	-	-	-	
											1	12	0	13	0
$*1$											1	15	3	1	12
											-	-	-	-	-
												3	3	12	12

Se aplica este proceso a 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, los 4 ceros de los símbolos de paridad y se obtiene el siguiente resto:

$$r\left(x\right) = 3x^3 + 3x^2 + 12x + 12 \tag{2.34}$$

Por su parte, el cociente q(x) no es requerido y se descarta. Así, el polinomio del mensaje codificado T(x) resulta:

$$\begin{gathered} x^{14} + 2x^{13} + 3x^{12} + 4x^{11} + 5x^{10} + 6x^9 + 7x^8 + 8x^7 \\ +9x^6 + 10x^5 + 11x^4 + 3x^3 + 3x^2 + 12x + 12 \end{gathered} \tag{2.35}$$

Versión pipeline

Para poder realizar la codificación en hardware de manera eficiente, usualmente se opera con datos utilizando arquitecturas pipelined, en las cuales se dispone de

múltiples procesamientos en paralelo, separando así un circuito en bloques y utilizando registros intermedios. De este modo, el cálculo de la división se lleva a cabo de una forma levemente alterada, ya que se efectúa usando los bits de mensaje uno por vez a medida que se presentan.

	x^{14}	x^{13}	x^{12}	x^{11}	x^{10}	x^{9}	x^{8}	x^{7}	x^{6}	x^{5}	x^{4}	x^{3}	x^{2}	x^{1}	x^{0}
	0	0	0	0											
	$\overline{1}$														
$*g(x)$	1→	15	3	1	12										
	-	-	-	-	-										
		15	3	1	12										
		$\overline{2}$													
$*g(x)$		13→	7	4	13	3									
		-	-	-	-	-									
			4	5	1	3									
			$\overline{3}$												
$*g(x)$			7→	11	9	7	2								
			-	-	-	-	-								
				14	8	4	2								
				$\overline{4}$											
$*g(x)$				10→	12	13	10	1							
				-	-	-	-	-							
					4	9	8	1							
					$\overline{5}$										
$*g(x)$					1→	15	3	1	12						
					-	-	-	-	-						
						6	11	0	12						
						$\overline{6}$									
$*g(x)$						0→	0	0	0	0					
						-	-	-	-	-					
							11	0	12	0					
							$\overline{7}$								
$*g(x)$							12→	8	7	12	15				
							-	-	-	-	-				
								8	11	12	15				
								$\overline{8}$							
$*g(x)$								0→	0	0	0	0			
								-	-	-	-	-			
									11	12	15	0			
									$\overline{9}$						
$*g(x)$									2→	13	6	2	11		
									-	-	-	-	-		
										1	9	2	11		
										$\overline{10}$					
$*g(x)$										11→	3	14	11	13	
										-	-	-	-	-	
											10	12	0	13	0
											$\overline{11}$				
$*g(x)$											1→	15	3	1	12
											-	-	-	-	-
												3	3	12	12

Con este arreglo, el primer mensaje, en este caso 1, se suma a los valores más significativos, inicialmente cero. El valor resultante, se multiplica por los coeficientes restantes del polinomio generador (15,3,1,12) para que los valores se sumen a los contenidos de las columnas restantes, que son inicialmente también cero. Luego el segundo mensaje (2) se suma a los contenidos de la columna más significativa siguiente (15) para producir 13. Este valor es multiplicado por los coeficientes del generador polinomial para dar los valores 7, 4, 13 y 3, y así consecutivamente.

2.3.5. Decodificación Reed-Solomon

Introducción de errores

Los errores pueden sumarse al polinomio de mensaje codificado $T(x)$, en la forma de un polinomio de error $E(x)$. Así el polinomio recibido $R(x) = T(x) + E(x)$. Donde $E(x) = E_{n-1}x^{n-1} + ... + E_1x + E_0$ y cada uno de los coeficientes $E_{n-1}.....E_0$ es un valor de m-bits, representado por un elemento del GF(2^m), con las posiciones de los errores en el código siendo determinadas por la potencia de x para ese término. Cabe destacar que, si más de $t = \frac{(n-k)}{2}$ de los E valores de error son no nulos, entonces la capacidad de corrección del código es excedida y los errores no son corregibles.

Los síndromes

En la sección 2.3.4 se demostró que la palabra de código transmitida es siempre divisible por el polinomio generador y que esta propiedad se extiende a los factores individuales del polinomio generador. Entonces, el primer paso en el proceso de decodificación es dividir el polinomio recibido por cada uno de los factores $(x + a^i)$ del polinomio generador de la Ec. 2.23, generándose un cociente y un resto:

$$\frac{R(x)}{x+\alpha^i} = Q(x) + \frac{S_i}{x+\alpha^i} \; Para \; b \; \leq i \leq b + 2t - 1 \tag{2.36}$$

El resto S_i resultante de estas divisiones es conocido como síndrome y puede escribirse como $S_0....S_{2t-1}$

Reordenando la ecuación anterior se obtiene:

$$S_i = Q_i(x) * \left(x + \alpha^i\right) + R(x) \tag{2.37}$$

Así, si $x = a^i$, Si se reduce a:

$$S_i = R\left(\alpha^i\right) = R_{n-1}(\alpha^i)^{n-1} + R_{n-2}(\alpha^i)^{n-2} + ... + R_1\alpha^i + R_0 \tag{2.38}$$

donde los coeficientes $R_{n-1}....R_0$ son los símbolos de la palabra de código recibida. Esto significa que cada uno de los valores de los síndromes puede obtenerse sustituyendo $x = a^i$ en el polinomio recibido, en lugar de aplicar la división de R(x) por $(x + a^i)$ para formar el resto.

La Ec. 2.38 puede re-escribirse como:

$$S_i = \left(... \left(R_{n-1}\alpha^i + R_{n-2}\right)\alpha^i + ... + R_1\right)\alpha^i + R_0 \tag{2.39}$$

En la forma de la Ec. 2.39, conocida como el ya mencionado método de Horner, el proceso se inicia multiplicando el primer coeficiente R_{n-1} por α^i. Entonces cada

coeficiente subsecuente es sumado al producto previo y la suma resultante se multiplica por α^i hasta que finalmente se suma R_0 . La gran ventaja de este método es que en el cálculo siempre se multiplica por el mismo valor α^i en cada etapa. Por este motivo se optó por realizar el cálculo de síndromes mediante este método.

Propiedades de los síndromes

Cabe destacar que los valores del síndrome son dependientes únicamente del patrón de error y no son afectados por los valores de los datos. Además, cuando no existe error, todos los síndromes valen 0. Matemáticamente, si sustituimos $x = \alpha^i$ en la ecuación $R(x) = T(x) + E(x)$, obtenemos:

$$R\left(\alpha^i\right) = E\left(\alpha^i\right) = S_i \tag{2.40}$$

ya que $T\left(\alpha^i\right) = 0$ debido a que $x + \alpha^i$ es un factor del polinomio generador.

Las ecuaciones de síndromes

Mientras que de la Ec. 2.38 es posible realizar el cálculo de los síndromes, mediante la relación existente entre estos y la palabra de código recibida, tal como se presenta en la Ec. 2.38, la relación entre los síndromes y el error polinomial puede ser usada para producir un conjunto de ecuaciones simultáneas, de las cuales pueden obtenerse los errores ocurridos en la transmisión. Para esto, el polinomio de error $E(x)$ es re-escrito para incluir sólo los términos que corresponden a los símbolos errados. Entonces, asumiendo v errores ocurridos, donde $v \leq t$:

$$E(x) = Y_1 x^{e_1} + Y_2 x^{e_2} + ... + Y_v x^{e_v} \tag{2.41}$$

donde $e_1 ....e_v$ identifican las ubicaciones de los errores en la palabra de código para las correspondientes potencias de x, $Y_1, ...Y_v$ representa la magnitud del error en esas ubicaciones.

Sustituyendo en la Ec. 2.40:

$$S_i = E\left(\alpha^i\right) = Y_1\alpha^{ie_1} + Y_2\alpha^{ie_2} + ... + Y_v\alpha^{ie_v} = Y_1 X_1^i + Y_2 X_2^i + ... + Y_v {X_v}^i \tag{2.42}$$

donde los $X_1 = \alpha^{e_1}, ..., X_v = \alpha^{e_v}$ se conocen como localizadores de error.

Entonces, las $2t$ ecuaciones de síndrome pueden ser escritas matricialmente de la siguiente manera:

$$\begin{bmatrix} S_0 \\ S_1 \\ \vdots \\ \vdots \\ S_{2t-1} \end{bmatrix} = \begin{bmatrix} {X_1}^0 & {X_2}^0 & \dots & {X_v}^0 \\ {X_1}^1 & {X_2}^1 & \dots & {X_v}^1 \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ {X_1}^{2t-1} & {X_2}^{2t-1} & \dots & {X_v}^{2t-1} \end{bmatrix} \times \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_v \end{bmatrix} \tag{2.43}$$

Es importante notar que los síndromes están escritos como S_0 ..... S_{2t-1} para corresponderse con las raíces α_0.... α_{2t-1}y las potencias de X son dependientes de haber elegido estas raíces en la Ec.2.23.

Ejemplo de decodificación - Parte 1 - Calculo de síndromes

Los pasos en el proceso de decodificación se ilustran por medio del ejemplo desarrollado del código Reed-Solomon (15,11) que fue utilizado en el proceso de codificación en la sub-sección 2.3.4.

Al introducir dos errores en el sexto (término x^9) y treceavo (término x^2) símbolo del mensaje codificado, se produce un polinomio de error con dos términos no nulos.

$$E(x) = E_9 x^9 + E_2 x^2 \tag{2.44}$$

y se puede elegir, por ejemplo, E_9 = 13 y E_2 = 2, de manera que se alteran 3 bits del sexto símbolo mientras que un sólo bit del treceavo símbolo es afectado. Aunque hay 4 bits erróneos, en términos de capacidad de corrección de error del código esto constituye sólo dos errores ya que está basado en el número de errores por símbolos erróneos. Por este motivo, estos errores deberían ser corregibles.

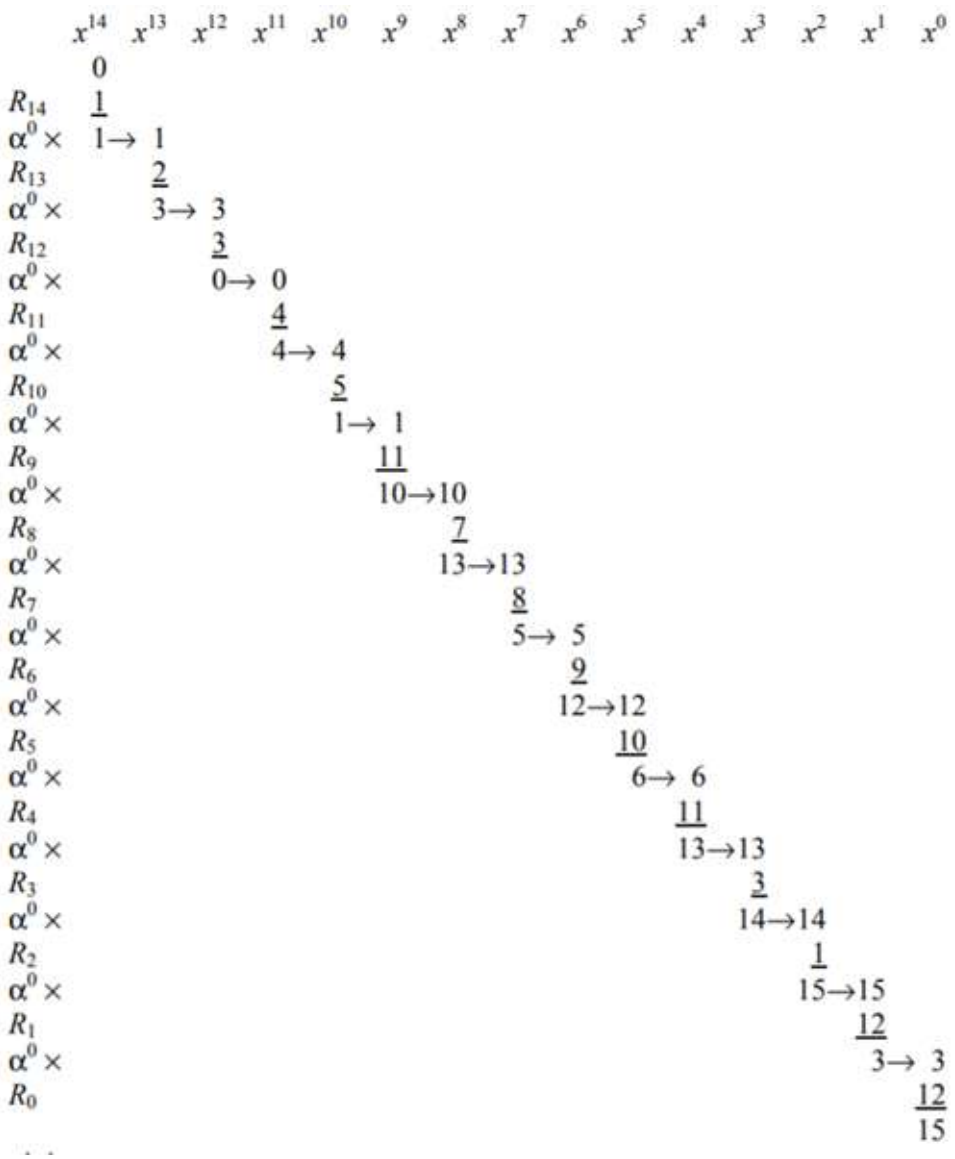
Sumar los errores al mensaje recibido genera el siguiente polinomio:

$$\begin{aligned} R(x) &= (x^{14} + 2x^{13} + 3x^{12} + 4x^{11} + 5x^{10} + 6x^9 + 7x^8 + 8x^7 + 9x^6 + 10x^5 + 11x^4 + 3x^3 + 3x^2 + 12x \\ &\quad +12 + (13x^9 + 2x^2) = (x^{14} + 2x^{13} + 3x^{12} + 4x^{11} + 5x^{10} + 11x^9 + 7x^8 + 8x^7 + 9x^6 \\ &\quad +10x^5 + 11x^4 + 3x^3 + x^2 + 12x + 12 \end{aligned} \tag{2.45}$$

o de una manera más sencilla: $1, 2, 3, 4, 5, (6 + 13), 7, 8, 9, 10, 11, 3, (3 + 2), 12, 12$

La sub-sección 2.3.5 mostraba que el síndrome que corresponde a cada raíz a^i del polinomio generador podría ser calculado dividiendo $\frac{R(X)}{x+a^i}$, o evaluando R(a^i). En el último caso, el método de Horner prueba una técnica eficiente.

Para el proceso de división directa, se utilizará un método de cálculo similar al de la sección 2.3.4. Sin embargo, el acercamiento de arquitectura pipelined de la sección 2.3.4 se ajusta mejor al hardware, por lo que el cálculo de S_0, que corresponde a la raíz a^0 , consiste en los siguientes pasos donde, en este caso, la multiplicación por α^0 (=1) es trivial:



Obteniendo $S_0 = 15$.

Alternativamente, se puede usar el método de substitución de la raíz en la Ec. 2.45. Así, para S_1 , sustituyendo a_1 por x y usando las equivalencias de la Tabla 2.3, se obtiene:

$$S_1 = (\alpha^1)^{14} + 2(\alpha^1)^{13} + 3(\alpha^1)^{12} + 4(\alpha^1)^{11} + 5(\alpha^1)^{10} + 11(\alpha^1)^9 + 7(\alpha^1)^8 + 8(\alpha^1)^7 + 9(\alpha^1)^6 + 10(\alpha^1)^5 + 11(\alpha^1)^4 + 3(\alpha^1)^3 + (\alpha^1)^2 + 12(\alpha^1) + 12 \tag{2.46}$$

O, usando el método de Horner:

$$\begin{aligned} S_2 = (((((((((((((1 * \alpha^2 + 2) * \alpha^2 + 3) * \alpha^2 + 4) * \alpha^2 + 5) * \alpha^2 + 11) * \\ \alpha^2 + 7) * \alpha^2 + 8) * \alpha^2 + 9) * \alpha^2 + 10) * \alpha^2 + 11) * \alpha^2 + 3) * \\ \alpha^2 + 1) * \alpha^2 + 12) * \alpha^2 + 12 \\ S_2 = 4 \end{aligned} \tag{2.47}$$

A su vez, el método de Horner puede ser escrito en una serie de pasos intermedios. Entonces, para S_3 donde $a_3 = 8$

$$
\begin{array}{rcrcrcr}
(& 0 & + & 1 &) & \times 8 & = & 8 \\
(& 8 & + & 2 &) & \times 8 & = & 15 \\
(& 15 & + & 3 &) & \times 8 & = & 10 \\
(& 10 & + & 4 &) & \times 8 & = & 9 \\
(& 9 & + & 5 &) & \times 8 & = & 10 \\
(& 10 & + & 11 &) & \times 8 & = & 8 \\
(& 8 & + & 7 &) & \times 8 & = & 1 \\
(& 1 & + & 8 &) & \times 8 & = & 4 \\
(& 4 & + & 9 &) & \times 8 & = & 2 \\
(& 2 & + & 10 &) & \times 8 & = & 12 \\
(& 12 & + & 11 &) & \times 8 & = & 13 \\
(& 13 & + & 3 &) & \times 8 & = & 9 \\
(& 9 & + & 1 &) & \times 8 & = & 12 \\
(& 12 & + & 12 &) & \times 8 & = & 0 \\
 & 0 & + & 12 & & & = & 12
\end{array}
$$

Por lo que $S_3 = 12$.

2.3.6. El polinomio localizador de error

Método directo

Se puede trabajar con los localizadores de error para armar un polinomio localizador. Este polinomio, denotado usualmente por $\sigma(x)$, se construye para tener los ubicadores de error X_1.....X_v como sus raíces, es decir, v factores de la forma $(x + X_j)$para $j = 1$ hasta v. Estos factores producen un polinomio de grado v con coeficientes σ_1 .... σ_v

$$\sigma(x) = \prod_{j=1}^{v} (x + X_j) = x^v + \sigma_1 x^{v-1} + ... + \sigma_{v-1} x + \sigma_v \tag{2.48}$$

Existe otra forma de cálculo, donde el polinomio suele denotarse como $\Lambda(x)$. En esta forma, se construye $\Lambda(x)$para tener v factores de la forma $(1 + X_j x)$, por lo tanto tiene las **inversas** X_1^{-1}, ... , X_v^{-1}de los v ubicadores de error, como sus raíces. Estos factores producen el polinomio de grado v con coeficientes Λ_1 ..... Λ_v:

$$\Lambda(x) = \prod_{j=1}^{v} (1 + X_j x) = 1 + \Lambda_1 x + ... + \Lambda_{v-1} x^{v-1} + \Lambda_v x^v \tag{2.49}$$

Sin embargo, se verifica que $\sigma(x) = x^v * \Lambda\left(\frac{1}{x}\right)$ por lo que los coeficientes σ_1 .... σ_v son los mismos que Λ_1 ..... Λ_v.

En el caso de $\Lambda(x)$, por cada error hay una raíz X_j^{-1}que hace $\Lambda(x) = 0$:

$$1 + \Lambda_1 X_j^{-1} + ... + \Lambda_{v-1} X_j^{-v+1} + \Lambda_v X_j^{-v} = 0 \tag{2.50}$$

Multiplicado la Ec. 2.50 por $Y_j X_j^{i+v}$:

$$Y_j X_j^{i+v} + \Lambda_1 Y_j X_j^{i+v-1} + ... + \Lambda_{v-1} Y_j X_j^{i+1} + \Lambda_v Y_j X_j^{i} = 0 \tag{2.51}$$

Ecuaciones similares pueden producirse para todos los errores, o sea diferentes valores de j. Sumando los términos se obtiene:

$$\sum_{j=1}^{v} Y_j X_j^{i+v} + \Lambda_1 \sum_{j=1}^{v} Y_j X_j^{i+v-1} + ... + \Lambda_v \sum_{j=1}^{v} Y_j X_j^{i} = 0 \tag{2.52}$$

y considerando que la suma de términos son los valores del síndrome:

$$S_{i+v} + \Lambda_1 S_{i+v-1} + ... + \Lambda_v S_i = 0 \tag{2.53}$$

Para distintos valores de i se obtienen ecuaciones similares, entonces se verifica:

$$S_{i+v} + \Lambda_1 S_{i+v-1} + ... + \Lambda_v S_i = 0 \; para \; i = 0, ..., 2t - v - 1 \tag{2.54}$$

por lo que se producen un conjunto de $2t - v$ ecuaciones simultáneas, algunas veces llamadas ecuaciones clave, con Λ_1 ..... Λ_v como incógnitas.

Para resolver este conjunto de ecuaciones para Λ_1 ..... Λ_v , se puede usar las primeras v ecuaciones, representadas por la siguiente ecuación matricial:

$$\begin{bmatrix} S_v \\ S_{v+1} \\ S_{v+2} \\ \vdots \\ S_{2v-1} \end{bmatrix} = \begin{bmatrix} S_{v-1} & S_{v-2} & S_{v-3} & \cdots & S_0 \\ S_v & S_{v-1} & S_{v-2} & \cdots & S_1 \\ S_{v+1} & S_v & S_{v-1} & \cdots & S_2 \\ \vdots & \vdots & \vdots & & \vdots \\ S_{2v-2} & S_{2v-3} & S_{2v-4} & \cdots & S_{v-1} \end{bmatrix} \times \begin{bmatrix} \Lambda_1 \\ \Lambda_2 \\ \Lambda_3 \\ \vdots \\ \Lambda_v \end{bmatrix} \tag{2.55}$$

Vale la pena destacar que hasta este momento v es desconocido. Debido a esto, es necesario calcular el determinante de la matriz para cada valor de v, empezando con $v = t$, hasta encontrar un determinante no nulo. Esto indica que las ecuaciones son independientes y pueden ser resueltas. Los coeficientes del polinomio ubicador de error Λ_1 ..... Λ_v pueden obtenerse invirtiendo la matriz, para resolver las ecuaciones.

El algoritmo Euclidiano

Otra técnica eficiente para obtener los coeficientes del polinomio localizador de error se basa en el método euclidiano para encontrar el máximo común divisor entre 2 números [7]. Así, se utiliza la relación entre los errores y los síndromes expresados en la forma de una ecuación basada en polinomios. Usualmente esta ecuación se denomina ecuación fundamental o clave y requiere que se introduzcan dos polinomios nuevos, los polinomios de síndrome y de magnitud de error.

El polinomio de los síndromes

Para usar en la ecuación fundamental, el polinomio de síndrome se define como:

$$S(x) = S_{b+2t-1} X^{2t-1} + ... + S_{b+1} x + S_b \tag{2.56}$$

donde los coeficientes son los $2t$ síndromes calculados de la palabra del código recibida usando la Ec.2.39.

El polinomio magnitud de magnitudes de error

El polinomio de magnitud de error puede escribirse como:

$$\Omega(x) = \Omega_{v-1}X^{v-1} + ... + \Omega_1 x + \Omega_0 \tag{2.57}$$

y sirve para determinar el valor de error que se sumó a los datos de entrada en la posición que indica el polinomio ubicador de error.

La ecuación fundamental

Con estos dos polinomios, la ecuación fundamental puede escribirse como:

$$\Omega(x) = [S(x)\,\Lambda(x)]\ mod\ x^{2t} \tag{2.58}$$

donde $S(x)$ es el polinomio de síndrome y $\Lambda(x)$ es el polinomio localizador de error. Cualquier término de grado x^{2t} o superior en el producto es ignorado debido a se tienen $2t$ símbolos de paridad en el mensaje transmitido, ya que es posible corregir t errores. Así:

$$\begin{aligned} \Omega_0 &= S_b \\ \Omega_1 &= S_{b+1} + S_b\Lambda_1 \\ &... \\ \Omega_{v-1} &= S_{b+v-1} + S_{b+v-2}\Lambda_1 + ... + S_b\Lambda_{v-1} \end{aligned} \tag{2.59}$$

Aplicando el método Euclidiano a la ecuación fundamental

Se sabe que el método Euclidiano puede encontrar el máximo común divisor d de dos elementos a. y b, tal que:

$$u * a + v * b = d \tag{2.60}$$

donde u y $v$son coeficientes producidos por el algoritmo.

En el caso de los polinomios mencionados, el producto de $S(x)$ de grado $2t-1$, con $\Lambda(x)$ de grado v, tendrá grado $2t+v-1$. Entonces el producto puede expresarse como:

$$S(x) * \Lambda(x) = F(x) * x^{2t} + \Omega(x) \tag{2.61}$$

en el cual los términos de x^{2t} y superiores, están representados por el término $F(x)$ y la parte restante está representada por $\Omega(x)$. Reordenando, para que los términos conocidos $S(x)$ y x^{2t} coincidan con los términos a y b de la Ec.2.60:

$$(\Lambda(x) * S(x) + F(x) * x^{2t} = \Omega(x) \tag{2.62}$$

El algoritmo consiste en dividir x^{2t}por $S(x)$ para producir un primer resto. En el siguiente paso $S(x)$ se vuelve el dividendo y el resto se vuelve el divisor para producir un nuevo resto. Este proceso se repite hasta que el grado del resto es menor que t. En este punto, tanto el resto $\Omega(x)$ y el factor de multiplicación $\Lambda(x)$ quedan disponibles como términos en el cálculo.

Ejemplo de decodificación - Parte 2 - Calculo del polinomio localizador de error

En esta etapa se desea encontrar los coeficientes del polinomio localizador de error. Para esto es necesario tener disponible el polinomio de síndrome, el cual fue calculado en el ejemplo de la sub-sección 2.3.5:

$$S_0 = 15,\ S_1 = 3,\ S_2 = 4\ y\ S_3 = 12 \tag{2.63}$$

por lo que el polinomio queda:

$$S(x) = S_3x^3 + S_2x^2 + S_1x + S_0 = 12x^3 + 4x^2 + 3x + 15 \tag{2.64}$$

El primer paso del algoritmo consiste en obtener el polinomio magnitud de error dividiendo x^{2t} (en este caso x^4) por S(x). Esta operación involucra una multiplicación de $S(X) * 10x$ $\left(10 = \frac{1}{12}\right)$ y una resta . Seguido por $S(x) * 6$ $\left(6 = \frac{14}{12}\right)$ y nuevamente se resta el resultado de la multiplicación, lo que resulta en un resto $6x^2 + 6x + 4$.

El otro procedimiento que se realiza en paralelo es el que permite obtener los coeficientes del polinomio localizador de error $\Lambda(x)$. Para ello se inicia con un valor inicial 1, debido a que se está dividiendo x^4 que luego es multiplicado por los mismos valores utilizados en el proceso de división (10 y 6) y estos se suman a un valor final el cual es $0 + 1 * (10x + 6) = 10x + 6$.

	Polinomio magnitud de error (x):						Polinomio ubicador de error (x):		
	x4	x3	x2	x1	x0	|	x2	x1	x0
dividendo:	1	0	0	0	0	|		0	0
divisor *10x:	1	14	13	12	0	|		10	0
	-	-	-	-	-	|	-	-	-
		14	13	12	0	|		10	0
divisor *6:		14	11	10	4	|		0	6
	-	-	-	-	-	|	-	-	-
resto:			6	6	4	|		10	6

Habiendo completado la primer división, y como el grado del resto no es menor a t (=2), se vuelve a dividir utilizando el divisor como el dividendo y el resto como el divisor, es decir, dividir $S(x)$ por el resto $6x^2 + 6x + 4$. En el primer paso, el resto es multiplicado por $2x$ $\left(2 = \frac{12}{6}\right)$ y sustraído, luego es multiplicado por 13 $\left(13 = \frac{8}{6}\right)$ y sustraído para producir el resto $3x + 14$. En el procedimiento de la derecha, el valor inicial anterior (1) se vuelve el sumando inicial y el resultado previo $(10x + 6)$ se multiplica por los valores utilizados en el proceso de división, esto produce $1 + (10x + 6) * (2x + 13) = 7x^2 + 7x + 9$

	Polinomio magnitud de error (x):						Polinomio ubicador de error (x):		
	x4	x3	x2	x1	x0	|	x2	x1	x0
dividendo:		12	4	3	15	|		0	1
divisor *2x:		12	12	8	0	|	7	12	0
	-	-	-	-	-	|	-	-	-
			8	11	15	|	7	12	1
divisor *13:			8	8	1	|		11	8
	-	-	-	-	-	|	-	-	-
resto:				3	14	|	7	7	9

En general, el proceso debería continuar repitiendo estos pasos, pero como el grado del resto es menor a t (=1), significa que el proceso finalizó. Los resultados buscados, extrayendo un valor constante de $\gamma = 9$, resultan:

$$\Lambda(x) = 14x^2 + 14x + 1 \; y \; \Omega(x) = 6x + 15 \tag{2.65}$$

Resolviendo el polinomio localizador de error - La búsqueda de Chien

Habiendo calculado los valores de los coeficientes, Λ_1 ..... Λ_v del polinomio localizador de error, es posible encontrar sus raíces. Si el polinomio es escrito en la forma:

$$\Lambda(x) = X_1 \left(x + X_1^{-1}\right) X_2 \left(x + X_2^{-1}\right) ... \tag{2.66}$$

claramente la función será cero si $x = X_1^{-1}$,X_1^{-2}, ... :

$$x = \alpha^{-e_1}, \alpha^{-e_2}, ... \tag{2.67}$$

Las raíces, y entonces los valores de X_1 ..... X_v , se obtienen por un método de prueba y error, conocido como la búsqueda de Chien [8], en la cual todos los posibles valores de las raíces, los elementos de campoα^i, $0 \leq i \leq n-1$, se sustituyen en la Ec. 2.49 y luego se evalúan los resultados. Si la expresión se reduce a cero, entonces ese valor de x es una raíz e identifica la posición del error. Debido a que el primer símbolo de la palabra de código corresponde al término x^{n-1} , la búsqueda se inicia con el valor $\alpha^{-(n-1)}$ (=α^1), entonces $\alpha^{-(n-2)}$ (=α^2), y continúa hasta α^0, el cual corresponde al último símbolo de la palabra de código.

Ejemplo de decodificación - Parte 3 - Busqueda de Chien

Como se explicó anteriormente en 2.3.6, el algoritmo de Chien permite determinar si existe un error en una determinada posición del polinomio ubicador de error (x). El método consiste en evaluar el polinomio para todos las palabras de código e identificar las que dan por resultado 0, ya que estas presentan un error.

Para identificar de una forma más sencilla las palabras que presentan error, es posible crear una tabla, tal como la tabla 2.4 en la que se presenta a modo de ejemplo todas las combinaciones posibles para un campo GF(16) .

Palabra de entrada (x)	Término x^2	Término x	Suma
α^{-14}	α^{13}	α^{12}	3
α^{-13}	α^{0}	α^{13}	13
α^{-12}	α^{2}	α^{14}	12
α^{-11}	α^{4}	α^{0}	3
α^{-10}	α^{6}	α^{1}	15
α^{-9}	α^{8}	α^{2}	0
α^{-8}	α^{10}	α^{3}	14
α^{-7}	α^{12}	α^{4}	13
α^{-6}	α^{14}	α^{5}	14
α^{-5}	α^{1}	α^{6}	15
α^{-4}	α^{3}	α^{7}	2
α^{-3}	α^{5}	α^{8}	2
α^{-2}	α^{7}	α^{9}	0
α^{-1}	α^{9}	α^{10}	12
α^{-0}	α^{11}	α^{11}	1

CUADRO 2.4: Tabla de valores para la búsqueda de Chien.

A partir de la primera fila, cada fila siguiente puede obtenerse a partir de la anterior multiplicando Λ_2 por α^2 y Λ_1 por α. Luego, sumando los términos resultantes se obtiene la columna de la derecha que resulta en un número no nulo, en el caso de un valor sin errores, o nulo si la palabra presenta un error. En esta tabla se puede ver que las dos palabras a las cuales se les introdujo (x^9 *y* x^2) error en el ejemplo, resultan en una suma igual a 0.

Además, es posible chequear estos resultados multiplicando los factores para los que se obtuvo el valor nulo:

$$\left(\alpha^9 x + 1\right)\left(\alpha^2 x + 1\right) = \alpha^{11}x^2 + \left(\alpha^9 + \alpha^2\right)x + 1 = 14x^2 + 14x + 1 \;=\; \Lambda\left(x\right) \quad (2.68)$$

tal como se presentó en la sección 2.3.6.

Así, la tabla 2.4 muestra el funcionamiento básico del algoritmo de Chien para encontrar las raíces del polinomio localizador de error.

2.3.7. Calculando los valores de error

Calculo directo

Cuando las ubicaciones de los errores X_1 ..... X_v son sustituidas en la Ec. 2.43 , las primeras v ecuaciones pueden ser resueltas realizando la inversa de la matriz para producir Y_1 ..... Y_v valores de error.

El algoritmo de Forney

Es interesante mencionar que existe un medio alternativo para calcular el valor de error Y_j habiendo establecido el polinomio localizador de error $\Lambda\left(x\right)$ y el polinomio de valor de error $\Omega\left(x\right)$. El algoritmo hace uso de la derivada del polinomio localizador de error.

La derivada del polinomio localizador de error

Dado un polinomio f(x) :

$$f(X) = 1 + f_1 x + f_2 x^2 + ... + f_v x^v \tag{2.69}$$

su derivada está dada por:

$$f'(x) = f_1 + 2f_2 x + ... + v f_v x^{v-1} \tag{2.70}$$

Sin embargo, para el polinomio ubicador de error $\Lambda(x)$, para $x = X_j^{-1}$, la derivada se reduce a:

$$\Lambda'\left(X_j^{-1}\right) = \Lambda_1 + \Lambda_3 X_j^{-2} + \Lambda_5 X_j^4 + ... \tag{2.71}$$

Esto equivale a establecer los términos pares del polinomio localizador de error a cero y dividir por $x = X_j^{-1}$.

Para el cálculo de la derivada de un polinomio dado sobre el campo de Galois, se requiere realizar un acercamiento especial debido a que no es posible hablar acerca de valores infinitesimales, teniendo en cuenta que no hay tales elementos que puedan utilizarse como infinitesimales en el campo de Galois [9]. Pero sí es posible utilizar un valor infinitesimal "formal" ε, que será utilizado para realizar las conversiones algebraicas.

Partiendo de la definición de la derivada formal de la función $f(x)$:

$$\frac{d}{dx} f(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon} \tag{2.72}$$

Para obtener derivadas de potencias de órdenes altos, se analiza la expresión $(x+\varepsilon)^k, k \geq 1$. Para elevar la expresión $x+\varepsilon$ a la k-ésima potencia, existe una fórmula en el álgebra tradicional: $(x+\varepsilon)^k = \sum_{i=0}^{k} C_k^i x^i \varepsilon^{k-i}$. El coeficiente binomial $C_k^i = \frac{k!}{i!(k-i)!}$representa la cantidad de sumandos comunes $x^i \varepsilon^{k-i}$(para cada $i = 0...k$), que se generan y se suman cuando elevamos la expresión a la k-ésima potencia.

A continuación, usando la aritmética característica de los campos de Galois

$$\mathrm{GF}(p^m): \overbrace{\underbrace{a+...+a}_{n \text{ sumandos}}}^{\mathrm{GF}(p^m)} = \overbrace{\lambda \cdot a}^{\mathrm{GF}(p^m)} \tag{2.73}$$

donde $\lambda = \overbrace{n \bmod p}^{\Re}$. En este caso $a = x^i \varepsilon^{k-i}$ y $n = C_k^i$, entonces:

$$\overbrace{\underbrace{x^i\varepsilon^{k-i} + ... + x^i\varepsilon^{k-i}}_{n \text{ sumandos}}}^{\mathrm{GF}(p^m)} = x^i\varepsilon^{k-i} \cdot \overbrace{(C_k^i \bmod p)}^{\Re} \tag{2.74}$$

Teniendo en cuenta lo mencionado previamente, es posible obtener la siguiente fórmula para la expresión de $(x+\varepsilon)^k, k \geq 1 : (x+\varepsilon)^k = \sum_{i=0}^{k} x^i \varepsilon^{k-i} \left(C_k^i \ mod \ p\right)$. Respectivamente, podemos obtener la derivada de la función:

$$f(x) = \beta x^k, k \geq 1 \Rightarrow \frac{d}{dx} f(x) = \beta \lim_{\varepsilon \to 0} \frac{(x+\varepsilon)^k - x^k}{\varepsilon} = \beta \lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \left(\left(\sum_{i=0}^{k} x^i \varepsilon^{k-i} \left(C_k^i \ mod \ p \right) \right) - x^k \right) \tag{2.75}$$

Si extraemos el sumando x^k de la sumatoria y consideramos que $C_k^k = 1$ y $\varepsilon^0 =$ 1obtenemos:

$$\frac{d}{dx}f(x) = \beta \lim_{\varepsilon \to 0} \frac{1}{\varepsilon}\left(\left(x^k + \sum_{i=0}^{k-1} x^i \varepsilon^{k-i}\left(C_k^i \bmod p\right)\right) - x^k\right) = \beta \lim_{\varepsilon \to 0} \frac{1}{\varepsilon}\left(\sum_{i=0}^{k-1} x^i \varepsilon^{k-i}\left(C_k^i \bmod p\right)\right) \tag{2.76}$$

Debe notarse que en el nominador, entre todos los términos de la sumatoria, sólo el sumando $x^{k-1}\varepsilon$ contiene ε en la primer potencia, el cual puede ser cancelado por el ε en el denominador. El resto de los sumandos contienen ε en potencias >1, por lo que no podrán ser cancelados por el denominador, pero serán cero por el límite de $\varepsilon -> 0$. Considerando esto:

$$\frac{d}{dx}\left(\beta x^k\right) = \beta \lim_{\varepsilon \to 0} \frac{1}{\varepsilon}\left(x^{k-1}\varepsilon\left(C_k^{k-1} \bmod p\right) + \left(\sum_{i=0}^{k-2} x^i \varepsilon^{k-i}\left(C_k^i \bmod p\right)\right)\right) = \beta\,(k \bmod p)\, x^{k-1}, k \leq 1 \tag{2.77}$$

Finalmente, tomando en cuenta lo previamente dicho, obtenemos la fórmula para el cálculo de la derivada del polinomio $\Psi(x) = \Psi_{k-1}x^{k-1} + ... + \Psi_1 x + \Psi_0$del campo de Galois GF($p^m$):

$$\frac{d}{dx}\Psi(x) = \frac{d}{dx}\left(\sum_{i=0}^{k-1} \Psi_i x^i\right) = \frac{d}{dx}\Psi_0 + \sum_{i=1}^{k-1} \Psi_i \frac{d}{dx}\left(x^i\right) = \sum_{i=1}^{k-1}\left(\Psi_i\,(i \bmod p) * x^{i-1}\right) \tag{2.78}$$

Y para el caso particular de un polinomio en el campo de Galois GF(2^m) , el cual es el caso de esta implementación, la ecuación queda de la siguiente manera:

$$\frac{d}{dx}\Psi(x) = \sum_{i=1}^{k-1}\left(\underbrace{\Psi_i \cdot \overbrace{(i \bmod 2)}^{\Re}}_{GF(2^m)} \cdot x^{i-1}\right) = \Psi_1 + \Psi_3 x^2 + \ldots + \begin{cases} \Psi_{k-1}x^{k-2}, & \text{Si (k-1) mod 2 = 1;} \\ \Psi_{k-2}x^{k-3}, & \text{Si (k-1) mod 2 = 0;} \end{cases} \tag{2.79}$$

Ecuación de Forney para las magnitudes de error

Los métodos para calcular los valores de error Y_1 .... Y_v basados en el algoritmo de Forney son más eficientes que el método directo para resolver ecuaciones de síndrome como se describió en la sección 2.3.7. De acuerdo al algoritmo de Forney, el valor de error está dado por:

$$Y_j = X_j^{1-b}\frac{\Omega\left(X_j^{-1}\right)}{\Lambda'\left(X_j^{-1}\right)} \tag{2.80}$$

donde $\Lambda'\left(X_j^{-1}\right)$es la derivada de $\Lambda(x)$ para $x = X_j^{-1}$.

Debería notarse que la Ec. 2.80 sólo da resultados válidos para posiciones de símbolo que contienen un error. Si el cálculo es hecho en otras posiciones, el resultado es generalmente no nulo e inválido. La búsqueda de Chien es por lo tanto todavía necesaria para identificar las posiciones del error.

Ejemplo de decodificación - Parte 4 - Algoritmo de Forney

El algoritmo de Forney permite calcular los valores de error en ubicaciones de error conocidas. Se basa en la interpolación de Lagrange donde, dado que habiendo calculado el polinomio evaluador de error $\Omega(x)$ es posible calcular el valor del error mediante:

$$e_j = -\frac{X_j^{1-c}\,\Omega(X_j^{-1})}{\Lambda'(X_j^{-1})} \tag{2.81}$$

En esta ecuación, el único valor desconocido es la derivada de $\Lambda(x)$ la cual se obtiene haciendo cero las potencias de x y dividiendo por x la ecuación de $\Lambda(x)$:

$$\Lambda(x) = 14x^2 + 14x + 1 \; --> \Lambda'(x_j^{-1}) = \frac{14x_j^{-1}}{x_j^{-1}} = 14 \tag{2.82}$$

Por lo que, de la Ec. 2.80 se deriva:

$$Y_j = x_j \frac{6x_j^{-1} + 15}{14} \tag{2.83}$$

Y para las ubicaciones de error que se obtuvieron previamente en la sub-sección 2.3.6, es decir x^9 y x^2, se pueden calcular los valores de error como sigue:

$$Y_6 = \alpha^9 \frac{6\alpha^{-9} + 15}{14} = 13 \tag{2.84}$$

$$Y_{13} = \alpha^2 \frac{6\alpha^{-2} + 15}{14} = 2 \tag{2.85}$$

que coincide con los errores introducidos.

Corrección de errores

Una vez localizados los símbolos que contienen los errores, identificados por X_j , y calculado los valores de Y_j de esos errores, los errores pueden ser corregidos añadiendo el polinomio de error $E(x)$ al polinomio $R(x)$ recibido. Debe tenerse en cuenta que por convención, el término de grado máximo del polinomio recibido corresponde al primer símbolo de la palabra de código recibida.

2.4. Memoria FIFO

Una memoria FIFO (First Input, First Output) se denomina a las memorias que permiten almacenar datos y extraerlos en el ordén que fueron almacenados, como se muestra en la Fig. 2.10. Generalmente, se realiza una analogía con las personas que esperan en una cola y van siendo atendidas en el orden de llegada, en otras palabras "primero en llegar, primero en ser atendido".

Las FIFO se usan comúnmente para almacenamiento y control de flujo, como es el caso de la implementación realizada en este proyecto, ya que se utilizan para controlar el flujo de datos de salida del codificador y el decodificador como se muestra en 1.1. Una FIFO consiste en un conjunto de punteros de lectura/escritura, almacenamiento y lógica de control. El almacenamiento puede ser SRAM (Static RAM), flip-flops, latches o cualquier otra forma de almacenamiento. Para FIFO de un tamaño importante se usa usualmente una SRAM de doble puerto, donde uno de los puertos se usa para la escritura y el otro para la lectura.

Las FIFOs pueden ser sincrónicos o asincrónicos. Un FIFO sincrónico maneja el mismo reloj tanto para las lecturas como para las escrituras. Un FIFO asicrónico es aquel que utiliza diferentes relojes uno para lectura y otro para la escritura.

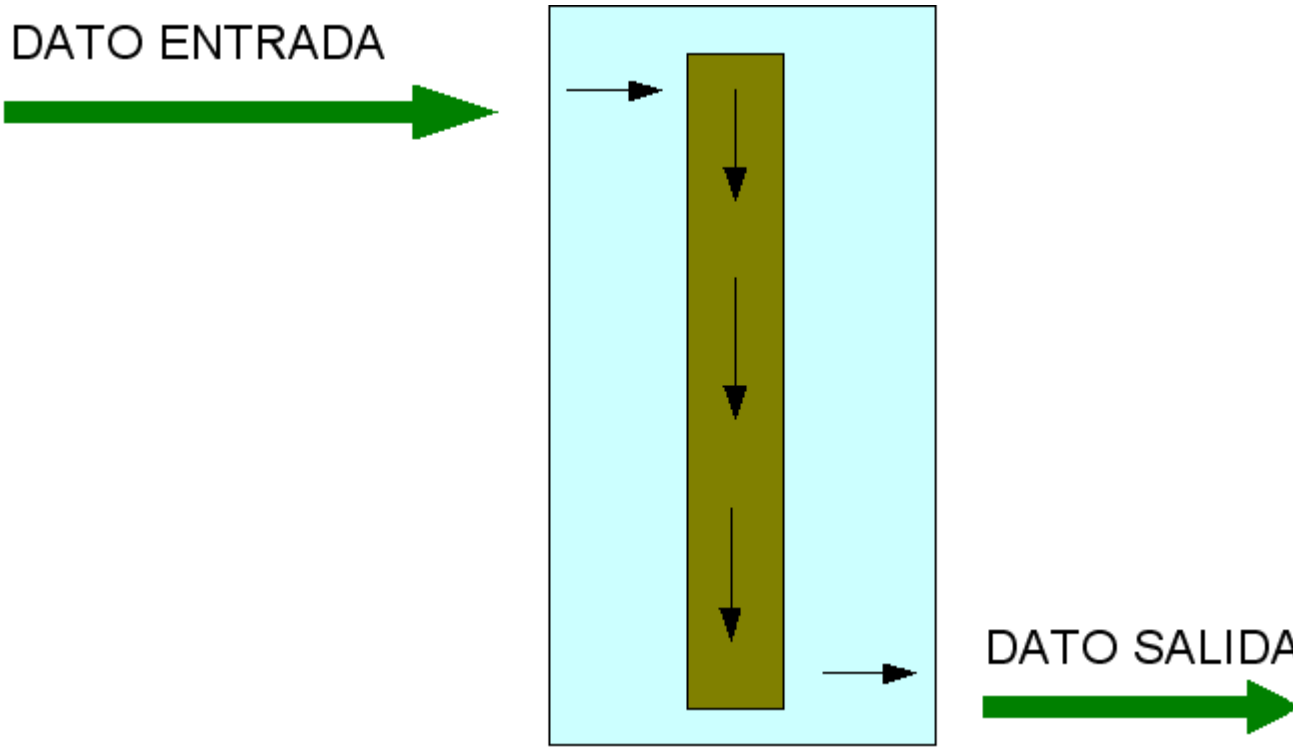


FIGURA 2.10: Funcionamiento de memoria FIFO.

Una memoria FIFO se puede usar para propósitos de sincronización, comportándose como una cola circular y, por lo tanto, contiene dos punteros:

- Puntero de lectura / registro de dirección de lectura.
- Puntero de escritura / registro de dirección de escritura.

Las direcciones de lectura y escritura se inician en la primera ubicación de la memoria. Además la cola FIFO comienza vacía.
En FIFO, se pueden enumerar:

- full (lleno): Cuando el registro de dirección de escritura alcanza al registro de dirección de lectura, la cola FIFO dispara la señal o bandera lleno.
- empty (vacío): Cuando el registro de dirección de lectura alcanza al registro de dirección de escritura, la cola FIFO dispara la señal o bandera vacío.
- almost full (casi lleno): Cuando el registro de dirección de escritura alcanza un registro definido por el usuario, el cual indica que la memoria se encuentra próxima a llenarse, se dispara la señal de casi lleno.
- almost empty (casi vacío): Cuando el registro de dirección de lectura alcanza un registro definido por el usuario, el cual indica que la memoria se encuentra próxima a vaciarse, se dispara la señal de casi vacío.

Cada FIFO, al igual que cualquier memoria, permite modificar su longitud y su ancho. Es decir, cada posición de memoria puede almacenar una cantidad de datos mayor o igual a 1 bit y a su vez, es posible definir una cantidad de posiciones de memoria variable. Aunque una vez creado el módulo en la FPGA, no es posible aumentar de manera dinámica estos valores.

Capítulo 3

Implementación práctica

3.1. Técnología de FPGA

3.1.1. Arquitectura

La arquitectura general de una FPGA consiste en arreglos de varios bloques programables (bloques lógicos) los cuales están interconectados entre sí con celdas de entrada/salida mediante canales de conexión verticales y horizontales, tal como muestra la Fig. 3.1. En general, se puede decir que posee una estructura bastante regular, aunque el bloque lógico y la arquitectura de ruteo varían de un fabricante a otro.

Las celdas configurables permiten implementar diversas funciones lógicas mediante la programación y luego síntesis, de circuitos lógicos combinacionales y sincrónicos. Algunas funciones simples pueden ser implementadas en una sola celda, mientras funciones más avanzadas pueden lograrse con la configuración e interconexión apropiada de celdas dentro de un conjunto. La estructura funcional interna de una celda depende del fabricante y de la familia de FPGA a la que pertenece. En general todas están compuestas por al menos una tabla de búsqueda o LUT, que permite implementar cualquier ecuación lógica hasta cierto número de entradas y una salida. Están integradas también por componentes como los flip-flops, cuya función principal suele ser registrar las salidas, multiplexores y circuitos aritméticos rápidos. Para ilustrar lo anterior, en la Fig. 3.2 se muestra la estructura de un módulo de lógica adaptable (ALM), tipo de celda lógica encontrada en las FPGA del fabricante Intel.

Las celdas no son los únicos componentes de las FPGA. Estas generalmente integran dispositivos dedicados, como bloques de memoria, bloques DSP, PLL y transceptores de alta velocidad, entre otros. Las FPGA pueden incluso incorporar microprocesadores en su encapsulado. Su arquitectura permite el ruteo eficiente de un gran número de conexiones entre estos componentes. Por otro lado, redes especiales distribuyen señales de reloj por todo el chip, con el objetivo de que las mismas lleguen a los componentes en el menor tiempo posible. Este conjunto de tecnologías permite al diseñador lograr diversas aplicaciones, en general de mayor velocidad que las realizadas en dispositivos multitarea como las computadoras.

3.1.2. Lenguajes de descripción de hardware

Para obtener aplicaciones en FPGA se utiliza un lenguaje de descripción de hardware, o HDL (Hardware Description Language), que permite un cierto nivel de abstracción sobre la implementación física final. Ejemplos de estos lenguajes son Verilog y VHDL. Si bien la sintaxis de estos lenguajes puede asemejarse a la de lenguajes de

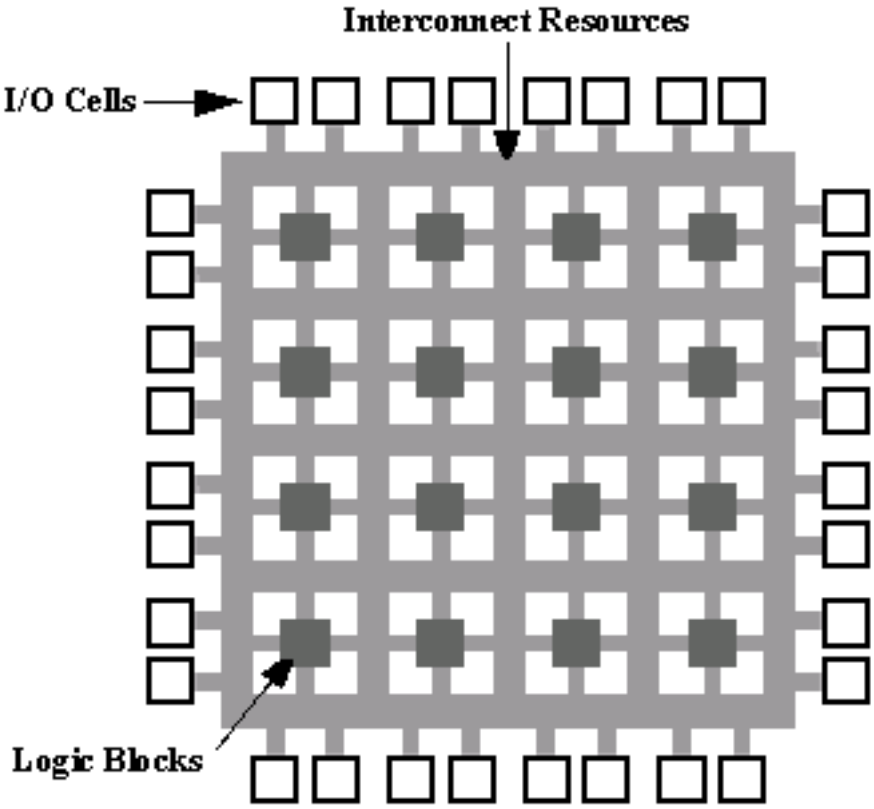


FIGURA 3.1: Arquitectura básica de un FPGA

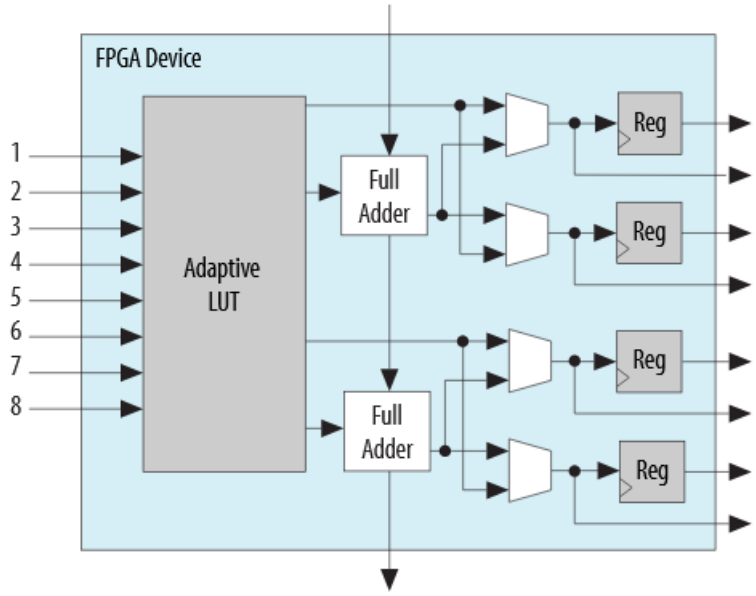


FIGURA 3.2: Componentes dentro de los ALM en FPGA de la familia Cyclone V

computación tradicionales, el paradigma de programación es distinto. En los archivos de HDL se describe el comportamiento deseado de elementos lógicos que trabajan en forma paralela y sus conexiones. Para escribir el código, el diseñador puede seguir uno de los siguientes estilos de programación:

- **Algorítmico o comportamental (behavioral)**: Se expresa la funcionalidad deseada, sin hacer referencia al hardware de bajo nivel.
- **Flujo de datos**: Se describen las funciones usando ecuaciones que se ejecutan de forma concurrente.
- **Estructural**: Se especifican componentes y sus interconexiones, siguiendo distintos niveles de jerarquía.
- **Mixto**: Se combinan algunos de los estilos anteriores.

3.1.3. Flujo de trabajo

Los entornos integrados de desarrollo (IDE) permiten llevar el lenguaje de descripci´on a un archivo binario que configura la FPGA, cumpliendo una serie de pasos:

1. **Compilación**: Se verifica que no haya errores de sintaxis en el código escrito por el usuario y permite el uso de bibliotecas desarrolladas por el fabricante u otros usuarios.
2. **Análisis y síntesis**: Se genera una netlist o interpretación RTL (Register Transfer Level) con distintos componentes lógicos interconectados. El entorno de desarrollo instancia módulos parametrizables a partir de distintas bibliotecas y efectúa optimizaciones lógicas.
3. **Elaboración (technology mapping**: Se traduce la netlist de forma especifica al tipo y cantidad de recursos utilizados en la FPGA seleccionada.
4. **Fitting o place and route**: Se definen las interconexiones de los componentes del dispositivo seleccionado.
5. **Ensamblado**: Se genera un archivo binario con la configuración deseada, listo para ser cargado en la memoria de la FPGA mediante el uso de un programador.

Los IDE tienen ademas otras funciones que son útiles para el desarrollo del sistema digital. Herramientas de simulación permiten verificar el comportamiento del diseño a nivel lógico o temporal. Herramientas de análisis temporal pueden otorgar estimaciones de las máximas frecuencias que se podrían alcanzar en el sistema. Por otro lado, se presentan detalles de la utilización o consumo de los recursos disponibles en el dispositivo, pudiendo indicar eventualmente la imposibilidad de lograr una elaboración para dicho dispositivo.

Las FPGA se destacan por su versatilidad y su velocidad. Permiten el desarrollo de aplicaciones complejas, con grandes requerimientos de procesamiento. Pueden también formar parte del diseño de aplicaciones para circuitos integrados dedicados, ya que comparten lenguajes y procesos de desarrollo semejantes. Los tiempos de desarrollo para FPGA se suelen medir en días/meses en lugar de meses/años, además de que los costos de desarrollo son inferiores en comparación con circuitos

dedicados como son los ASIC. En definitiva, es una tecnología apropiada para el desarrollo de sistemas como los de comunicaciones. Por ello su aparición es habitual en la bibliografía que trata temas relacionados con las mismos, como es el caso de este trabajo.

3.2. Criterios de diseño

Para la implementación se establecieron ciertos criterios u especificaciones, los cuales fueron los siguientes:

- **Calculo de tasa de error de bit**: La implementación debe ser capaz de brindar la tasa de error que presenta el código simulado, es decir, determinar la cantidad de bits erroneos a partir de la comparación del mensaje transmitido y el mensaje recibido.
- **Múltiples distribuciones de comunicación**: Se debe lograr simular distintos tipos de distribuciones de comunicaciones como la Gaussiana o Rayleigh.
- **Alta velocidad de procesamiento**: Se debe obtener la tasa de error en un tiempo considerable a la capacidad de la FPGA. La misma debe ser más rápida que realizar el procedimiento en una computadora.
- **Performance comparable a diseños teóricos**: El diseño debe poder calcular la tasa de error de bit para 1 millón de bits y la misma debe ser equivalente a lo calculado teóricamente.
- **Diseño portable**: El código fuente debe ser portable a distintas familias de FPGA.
- **Placa DE-10 Standard**: Debe implementarse el diseño en una placa Terasic DE-10 Standard facilitada por el Laboratorio de Comunicaciones (LAC) de la Facultad de Ingeniería de la Universidad Nacional de Mar del Plata.

3.3. Implementación en Matlab - Segmentador

3.3.1. Selección de jerarquía de segmentación

La determinación de la jerarquía de segmentación apropiada para una función dada juega un rol importante ya que si se elige la jerarquía equivocada puede resultar en una segmentación ineficiente, lo que llevaría a una cantidad de segmentos innecesariamente extensa. Por esta razón, como se mencionó en la sección 2.2.3, en la práctica se calcula el error de segmentación balanceado y se obtiene un histograma para saber cual de los 4 métodos de segmentación se adapta mejor a la función. Este proceso queda representado en el siguiente código realizado en Matlab, el cual realiza el procedimiento mencionado en el pseudo-código 2.1.

```
1 function [H,IH] =  Hierarchy_Scheme(a,b,errors)
2
3 %P2SL
4 A(1,:) = [0;0.0078125;0.015625;0.03125;0.0625;0.125;0.25;0.5;1]*(b-a)+a;
5
6 %P2SR
7 A(2,:) = [0;0.5;0.75;0.875;0.9375;0.96875;0.984375;0.9921875;1]*(b-a)+a;
8
```

```
9  %P2SLR
10
11 A(3,:) = [0;0.0625;0.125;0.25;0.5;0.75;0.875;0.9375;1]*(b-a)+a;
12
13 %US
14
15 A(4,:) = [0;0.125;0.25;0.375;0.5;0.625;0.75;0.875;1]*(b-a)+a;
16 %Genero un vector de ceros de 4 filas y el numero de intervalos (x1,x2) que
17 %cumplen con el error balanceado establecido
18 Ah = [zeros(4,8)];
19 %Checkeo que el SEGMENTO el cual tiene un ERROR BALANCEADO este dentro del
20 %intervalo de los vectores de prueba. Si no se cumple eso, no aumenta la
21 %varianza
22 for i = 1:4
23     for j = 1:8
24         for k = 1:(numel(errors))
25             if (A(i,j) < errors(k) && errors(k) < A(i,j+1))
26                 Ah(i,j)= Ah(i,j) + 1;
27             end
28         end
29     end
30 end
31 Varianza(1) = var(Ah(1,:));
32 Varianza(2) = var(Ah(2,:));
33 Varianza(3) = var(Ah(3,:));
34 Varianza(4) = var(Ah(4,:));
35 [~,IH] = min(Varianza);
36 H= A(IH,:);
37 end
```

Los parametros de entrada se corresponden con los del pseudo-código, siendo **(a,b)** el intervalo de validez de la función de entrada y **errors** el vector que contiene los límites de los segmentos con error balanceado. Luego, en las líneas 24 a 28 se tiene la comparación de los límites de los segmentos del error balanceado y los de cada jerarquía de segmentación. Para poder obtener la varianza, se crea una matriz con 4 filas (una fila por cada jerarquía de segmentación) y n columnas, siendo n la cantidad de valores contenidos en el vector **errors**. Luego se compara cada componente k del vector en cuestión, con los intervalos que se generan entre los elementos de cada jerarquía. Si k se encuentra dentro del intervalo, se aumenta en 1 el contador de la (fila , columna) referida a ese componente. Una vez finalizada la comparación con los 4 métodos de segmentación, se calcula la varianza para cada fila y la que presenta la menor varianza es la elegida (líneas 31 a 36). Los parámetros de salida de este programa son **H** el cual se corresponde con el vector equivalente a la segmentación elegida e **IH** que es el número asignado a la jerarquía seleccionada, siendo:

1. P2SL
2. P2SR
3. P2SLR
4. US

3.3.2. Cantidad de bits de segmentación y ROMs

Una vez definida la mejor segmentación, se debe determinar la cantidad de bits óptimos para la segmentación externa B_{x0}. Nuevamente, se procede a implementar

el diseño mostrado en la sección 2.2.3; especificamente el pseudo-código 2.2. Así, se desarrolló el siguiente código, el cual presenta cuatro parámetros de entrada: el intervalo de entrada **(a,b]**, la unidad en el último lugar **ulp**, el grado **d** del polinomio a utilizar para desarrollar los splines y el error máximo **Ereq** deseado.

```
1  function [M,T,To,ROM0,ROM1] = Hierarchical_Segmentation_Method(fun,a,b,ulp,degree,reg_error)
2  %Parameters function fun, input interval (a,b], unity in the last place
3  %ulp, polynomial degree d, Required error Ereq
4
5  %Select segmentation Hierarchy H
6
7  [~,~,~,Balanced_error_boundaries,~] = Boundaries(fun,a,b,degree,ulp,2^-10);
8  [~,IH] = Hierarchy_Scheme(a,b,Balanced_error_boundaries,length(Balanced_error_boundaries));
9  sprintf('Segmentacion utilizada IH=%d',IH)
10
11
12 %Find segmentation with the optimal Bxo
13 Bxo = 1;
14 done = 0;
15 Mprim = 999999999999;
16 while (~done)
17     To = GetToBoundaries(IH,Bxo,a,b);
18     so = numel(To)-1;
19     ROM0 = [];
20     ROM1 = [];
21     T = [];
22     offset = 0;
23     for i = 1:so
24         A = To(i);
25         B = To(i+1);
26         [Coeffs,Emax] = NewRemez(fun,reg_error,degree,A,B);
27         if Emax > reg_error
28             ToSegSize = B-A;
29             Bx1 = 0;
30             while Emax > reg_error
31                 Bx1 = Bx1 + 1;
32                 T1SegSize = ToSegSize/2^(Bx1);
33                 T1 = [];
34                 T1Coeffs = [];
35                 S1 = 2^Bx1;
36                 for j = 1:S1
37                     A = To(i) + T1SegSize * j;
38                     B = A+T1SegSize;
39                     [Coeffs,Emax] = NewRemez(fun,reg_error,degree,A,B);
40                     if Emax > reg_error
41                         break
42                     end
43                     T1(j) = A;
44                     T1Coeffs(j,:) = Coeffs;
45                 end
46             end
47             ROM0=[ROM0;Bx1,offset];
48             offset = offset + 2^(Bx1);
49         else
50             T1 = A;
51             T1Coeffs = Coeffs';
52         end
53         if (~isempty(T)) & (numel(T1(1,:)) < numel(T(1,:)))
54             T1 = [T1, zeros(1,(numel(T(1,:))-numel(T1)))];
55         elseif (~isempty(T)) & (numel(T1(1,:)) > numel(T(1,:)))
56             T = [T,zeros(numel(T(:,1)),(numel(T1)-numel(T(1,:))))];
57         end
58         T = [T;T1];
59         ROM1 = [ROM1;T1Coeffs];
60     end
61     M = length(T);
62     if M > Mprim
63         M = Mprim;
64         T = Tprim;
65         Bx1 = Bx1Prim;
66         To = Toprim;
67         ROM0 = ROM0prim;
68         ROM1 = ROM1prim;
69         sprintf('Valores finales M=%d and Bx0=%d',M,Bxo-1)
70         break
71     else
72         sprintf('M=%d and Bx0=%d',M,Bxo)
73         Bxo = Bxo +1;
74         Mprim = M;
75         Tprim = T;
76         Toprim = To;
77         Bx1Prim = Bx1;
78         ROM0prim = ROM0;
79         ROM1prim = ROM1;
80     end
81 end
82 end
```

En este caso, como en 2.2, se encuentra la jerarquía de segmentación apropiada (líneas 7 y 8) donde la función **Boundaries** permite obtener los límites de cada segmento, donde cada uno es calculado con una segmentación de error balanceado. A continuación, **Hierarchy Scheme** toma estos límites y determina cuál es la jerarquía

que mejor se adapta a la función en cuestión. Luego se aplica la segmentación jerárquica, a la vez que se busca el B_{x0} óptimo que minimice la cantidad de segmentos M. Para cada segmento en la segmentación exterior, se computan los coeficientes de Chebyshev mediante la función **NewRemez** que implementa el algoritmo de Remez [10]. El mismo consiste en una construcción iterativa de un conjunto de puntos $x_0, x_1, ..., x_{n+1}$ para aproximar una función f en el intervalo (a,b).

1. Se comienza con un conjunto inicial de puntos $x_0, x_1, ..., x_{n+1}$ en el intervalo [a,b].

2. Se considera el sistema de ecuaciones lineales

$$\begin{cases} p_0 + p_1 x_0 + p_2 x_0^2 + \cdots + p_n x_0^n - f(x_0) & = +\epsilon \\ p_0 + p_1 x_1 + p_2 x_1^2 + \cdots + p_n x_1^n - f(x_1) & = -\epsilon \\ p_0 + p_1 x_2 + p_2 x_2^2 + \cdots + p_n x_2^n - f(x_2) & = +\epsilon \\ \cdots & \cdots \\ p_0 + p_1 x_{n+1} + p_2 x_{n+1}^2 + \cdots + p_n x_{n+1}^n - f(x_{n+1}) & = (-1)^{n+1}\epsilon. \end{cases}$$

   El cual es un sistema de n + 2 ecuaciones lineales con n + 2 incógnitas: $p_0, p_1, ..., p_n$ y ϵ. Por lo tanto tendrá una solución $(p_0, p_1, ..., p_n, \epsilon)$ siendo ϵ el error que se tiene al realizar la diferencia entre la función aproximada y el polinomio generado para aproximarla. Resolviendo este sistema se obtiene el polinomio $P(x) = p_0 + p_1 x + ... + p_n x^n$.

3. Por último, calculamos el conjunto de puntos y_i en [a,b] donde P(x) - f(x) (diferencia entre el polinomio generado para aproximar la función P y la función aproximada f) tiene sus extremos, y se repite el proceso (paso 2), reemplazando los x_i's por los y_i's

Así, se obtuvo la siguiente función en Matlab:

```
function [P,Emax] = NewRemez(fun,error,n,a,b)
    H = sym('p',[n+1,1]);
    H = H(:);
    H = H(:);
    pts = a/2 + b/2 + ((b-a)/2) *cos(pi*(0:n+1)/(n+1));

    ratio = 2;
    Count = 1; threshold = 1.0 + error; %Diferencia tolerada entre error maximo y minimo
    while ratio > threshold
        %P-F = (-1)^i*epsilon con i = 0...n+1...
        %Se arman los vectores como con X^n y una vez evalaudos se multiplican

        %Creo vector de X^n
        for i = 0:n
            x = sym('x');
            x_vector(i+1) = x^i;
        end
        %Obtener X0,X1,X2.....Xn haciendo eval(x_vector)
        %reemplazando el valor de x previamente.
        %Se crea matriz de x y luego se evalua cada fila en X0, X1, X2....Xn.
        L = 0;
        for j = 0:n+1
            if L == 0 %Se concatenan los vectores de x_vector
               L = [x_vector];
            else
               L = [L;x_vector];
            end
        end
        %Se obtienen los valores del vector x(i) evaluados en los puntos de la ecuacion de Chebyshev.
        %Evaluamos F(X0,X1.....,Xn)
        for i = 1:n+2
            x = pts(i);
            L(i,:) = eval(L(i,:));
        end
        L = double(L);

        %Evaluo la funcion de entrada en los puntos de Chebyshev
```

```
38          for i = 1:n+2
39              fun_eval(i) = fun(pts(i));
40          end
41          %Se pasa a double para poder seguir operando y se transpone
42          fun_eval = double(fun_eval);
43          fun_eval = fun_eval(:);
44          %Se crea el vector de error "e" (alterna entre +e y -e)
45          e = ones(1,n+2).*(-1).^(1:n+2);
46          e = e(:);
47          %Se concatena el vector de error "e" a la matriz de polinomios
48          %que se utilizara para aproximar la funcion
49          L = [L,e];
50          %Matlab lo resuleve como A*X (X = x+e) = B lo por lo que
51          %hay que expresarlo como como X(c)*P+e = F(c)
52          %Se resuelve la matriz resultante usando "linsolve" como "A*X = B"
53          %siendo A = L y B = fun_eval
54          P = linsolve(L,fun_eval);
55          %Se transforma a simbolico el vector_x * P
56          %(los puntos que se obtuvieron de linsolve)
57          q_sym = x_vector*(P(1:n+1));
58          q = matlabFunction(q_sym);
59          %Realizamos la resta de q - f
60          fun_sym = sym(fun);
61          %Se obtiene la derivada (usando diff) de la funcion (P(x) - F(x))
62          %para mediante las raices, generar los nuevos puntos con los que se
63          %volvera a iterar o se utilizaran para la solucion final.
64          fun_der = matlabFunction(diff(q_sym - fun_sym));
65          [pts] = AllRootsOf(fun_der,a,b);
66          num_sols = numel(pts);
67          if num_sols > n+2
68              disp('Too many extreme values, try larger degree');
69          elseif num_sols == n
70              pts = [a,pts,b];
71          elseif num_sols == n+1
72              if abs((q(a)-fun(a))) > abs((q(b)-fun(b)))
73                  pts = [a,pts];
74              else pts = [pts,b];
75              end
76          elseif num_sols < n
77              disp('Not enough oscillations');
78          end
79          fdiff = abs(q(pts) - fun(pts));
80          %Se obtiene el error maximo de la matriz P, sabiendo que se
81          %encuentra en el final de la misma:
82          Emax = abs(P(end));
83          %Esto retorna el valor maximo y el minimo de la matriz de puntos
84          %(q(pts) - f(pts)) para asi saber si se cumple el ratio solicitado
85          %entre el algoritmo.
86          [maxfg,Imax] = max(fdiff);
87          [minfg,Imin] = min(fdiff);
88          ratio = maxfg/minfg;
89      end
90      P = P(1:n+1);
91  end
```

Aquí se realizan los pasos mencionados previamente, iniciando con un conjunto de puntos, los cuales son los coeficientes de Chebyshev (Línea 5) que aseguran una convergencia cuadrática [11]. Luego se resuelve el sistema de ecuaciones (Líneas 10 a 65). Cabe destacar que el algoritmo **AllRootsOf** de la línea 65 calcula todas las raíces de una función g (en este caso la derivada de P(x) - f(x)) en un intervalo $[a, b]$, asumiendo que ningún intervalo de la forma $[a + kh, a + (k + 1)h]$, donde $h = \frac{(b-a)}{200}$, contiene más de una raíz. Por último se comprueba si el error obtenido es menor que el requerido, si esto no es así, se continua iterando.

Volviendo a la función **Hierarchy Scheme**, si el error de aproximación es mayor al requerido (líneas 27 a 52), el número de segmentos de la segmentación interna se incrementa en sucesivas potencias de 2, hasta cumplir el error buscado. Este proceso se repite para todos los segmentos externos y finalmente se tiene como resultado el número total de segmentos M, el vector T que contiene los extremos de cada segmento, la ROM0 que se utiliza para direccionar a la ROM1 y la ROM1 con los coeficientes para cada segmento.

3.3.3. Representación gráfica de la segmentación realizada

A modo de ejemplo, se muestra de manera gráfica, en la Fig. 3.3, una segmentación resultante mediante el programa.

Aquí se puede apreciar que la segmentación determinada por el programa se adapta a la función, ya que las mayores alinealidades se encuentran en los extremos

de la misma. De esta manera, se segmentan en mayor medida el inicio y el final de la función, generando mayor cantidad de splines de orden 2, los cuales tienen menor rango de validez en comparación con los obtenidos en el centro de la gráfica.

3.4. Implementación System On Chip (SoC)

3.4.1. Consideraciones generales

Una vez generadas las memorias ROM0 y ROM1, se procedió a trabajar con la placa de desarrollo Terasic modelo DE-10 Standard. La placa basada en una FPGA Intel modelo Cyclone V 5CSXFC6D6F31C6N, del tipo *System On Chip* (SoC), se muestra en la Fig. 3.4. El procesador dedicado tipo HPS (Hard Processor System) es un dual-core Cortex-A9 de arquitectura ARM de 32 bits. Además la placa dispone de diversos periféricos, interfaces y puertos.

Para comprobar el funcionamiento de sistemas correctores de errores suele utilizarse el método Monte Carlo. Este implica, generar, codificar, modular y decodificar un número grande de mensajes para su procesamiento, y posterior análisis estadístico de resultados. Para realizar este procedimiento, se implementaron un codificador y decodificador Reed-Solomon en la FPGA, y se delegó la generación del canal al procesador. Se decidió por esta alternativa debido a que en primer lugar, la FPGA se comunica con el HPS de forma interna mediante puentes configurables de muy alta velocidad [12]. Estos puentes permiten el movimiento rápido y directo de datos, sin overhead, y evitando el desarrollo de protocolos a nivel de aplicación. Con esto se pueden reducir los intervalos de tiempo ociosos para la FPGA.

En segundo lugar, el sistema de pruebas permite independizarse de la computadora. Si bien, se podría establecer inicialmente una ejecución desde la computadora, y luego dejar correr un programa sobre la FPGA, se dispone del procesador embebido por lo que se aprovecha el mismo para realizar estas tareas. Además, el mismo cuenta con una capacidad de procesamiento suficiente para tareas básicas de control e interacción con el usuario, ya que funciona a una frecuencia de hasta 925 MHz y cuenta con 1 GB de RAM. El procesador permite correr un sistema operativo (OS) liviano, dedicando casi la totalidad de sus recursos al programa de pruebas. Las tareas pesadas de procesamiento son efectuadas por la lógica en la FPGA.

Por último, una implementación que hiciera uso de la combinación de FPGA y HPS era una excelente oportunidad para ganar experiencia en el desarrollo de aplicaciones híbridas, que aprovechan las ventajas de estas dos clases de dispositivos.

3.4.2. Arquitectura del banco de pruebas - General

En la placa se implementó un sistema de pruebas cuya arquitectura se encuentra en la Fig. 3.5. La misma se diseñó pensando un uso destinado a los investigadores del Laboratorio de Comunicaciones (LAC) de la Universidad Nacional de Mar del Plata.

En este sistema el procesador es el encargado del control general de todo el conjunto. Para ello se implementan en total cuatro sistemas de control. El sistema maestro se encuentra en el flujo del programa que corre en el procesador. En la FPGA se implementan los otros tres sistemas como máquinas de estado finito (FSM), dos de ellos controlados por un sistema general, el cual es esclavo del HPS. Las FSM operan de forma independiente entre sí, emulando el funcionamiento de un sistema de comunicaciones real, en el que transmisor y receptor se encuentran separados por un canal de comunicaciones.

La FPGA y el HPS pueden comunicarse entre sí usando tres buses bidireccionales independientes. Un bus de alta performance, denominado HPS-to-FPGA Bridge se encarga de pasar datos con el HPS como maestro del bus. Su contrapartida es el FPGA-to-HPS Bridge, que tiene las mismas características pero tiene a la FPGA como maestro. Estos buses trabajan con un ancho nativo de datos de 64 bits, pero pueden ser configurados para funcionar con anchos de 32 bits o de 128 bits. El tercer bus es el lightweight HPS-to-FPGA Bridge, que opera a velocidades inferiores y posee un ancho de datos fijo de 32 bits. Todos los buses se conectan al procesador a través de un switch interno, utilizando el protocolo Advanced Microcontroller Bus Architecture (AMBA) Advanced Extensible Interface (AXI) [12]. Para comunicarse a través de estos buses puede utilizarse también el protocolo de comunicaciones Avalon, cuyos diagramas de temporización deben ser respetados por cualquier lógica que se desee implementar.

La comunicación entre el maestro y el esclavo se efectúa mediante un sistema de hand-shake, conectados a través del lightweight bus HPS-FPGA. Cuando el HPS habilita el inicio del proceso, mediante flags permite en primera instancia el inicio de la codificación, una vez finalizada, se escribe un registro en el procesador que habilita el siguiente paso en la FPGA. De esta manera se continua hasta que se llega a una cantidad de palabras a decodificar preestablecida y se finaliza el proceso.

Para diseñar el sistema completo se trabajó en paralelo con el procesador y con la FPGA, por esta razón se implementó esta coordinación del procesador y de la FPGA a través de este sistema de control combinado, lo que permitió un intercambio de datos rápido y libre de errores.

Para acceder desde el programa a los registros de estado y a las memorias en FPGA, se utiliza la técnica de acceso por mapeo a memoria (Memory-mapped Access). Desde la perspectiva del programa, los distintos periféricos, registros de estado y control, o direcciones de memoria en la FPGA, son simplemente corrimientos u offsets relativos a una dirección de memoria base. El sistema operativo permite la correspondencia de un archivo en la memoria virtual del programa con direcciones de memoria reales a través de este mapeo.

3.4.3. Arquitectura del banco de pruebas - Software

Puesta en marcha del Sistema Operativo

Del lado del procesador, se decidió elaborar un programa que implementara mediante funciones las tareas de generación del canal, corriendo sobre un sistema operativo OS. Si bien la operación del OS consume recursos del procesador, la programación es mucho más sencilla que las realizadas en desarrollos del tipo *bare-metal*; estos desarrollos hacen uso del procesador de forma exclusiva, pero requieren del tratamiento en bajo nivel de múltiples registros de configuración y de control de dispositivos, tareas que un sistema operativo ya tiene resueltas, lo que reduce en gran medida los tiempos de implementación, a pesar de no obtenerse la máxima performance [13].

El sistema operativo utilizado es una distribución de Linux Debian liviana y destinada a sistemas embebidos, denominada Ångström. Dicha distribución hace una utilización relativamente restringida de los recursos del procesador para su funcionamiento y carece de un entorno gráfico, liberando recursos para las aplicaciones del usuario. Otros puntos destacables son la interfaz de usuario accesible de forma remota por terminal, y la disponibilidad de aplicaciones en los repositorios Linux accesibles desde conexión a Internet. Esto abre la posibilidad de acceder de forma

remota al sistema, desde otros lugares. La versión de Ångström utilizada en este proyecto (ver. 2014.12) es provista por Terasic como parte del paquete de soporte para la placa DE-10 Standard.

Para correr el OS en el procesador es necesario almacenar una imagen del mismo en una tarjeta de memoria microSD. La preparación de la tarjeta requiere del seguimiento de una serie de pasos para crear y modificar archivos y scripts necesarios para ejecutar el sistema operativo y que el mismo reconozca a la FPGA como dispositivo periférico disponible [13] [14] [15].

El programa que corre sobre el sistema operativo se compuso en lenguaje C++ utilizando un entorno de desarrollo y un editor de texto integrado en el SO del SoC. El IDE utilizado fue Code::Blocks versión 20.03 donde se desarrolló y testeo la mayor parte del programa. Una vez que se obtuvo una versión funcional, se copió el archivo a la SD de la FPGA y se ultimaron detalles utilizando el editor de texto Vi a la vez que se testeaban las funcionalidades del procesador.

Programa en C++ para generación del canal - Lineamientos generales

El software desarrollado permitió realizar la indexación de las memorias obtenidas mediante Matlab (ROM0 y ROM1) y la evaluación polinomial de los coeficientes almacenados en la última de estas ROMs, con el objetivo de replicar el procedimiento explicado teoricamente en 2.2.4 (donde se divide x en x0, x1 y x2 para indexar los distintos splines) y en 2.2.5 (donde se explica la evaluación matemática mediante la regla de Horner de los distintos polinomios). Para ello, se siguió la arquitectura mostrada en la Fig. 3.7. La misma consiste de una primera etapa en la que se generan los números aleatorios uniformes, luego se procede a decodificar las direcciones de los coeficientes utilizando la ROM0, la unidad de selección de bit, la unidad P2S (Power of 2 Segments) y por último, una etapa de evaluación polinomial incorporada con la tabla de coeficientes ROM1. La unidad de selección de bit consiste tomar el conjunto de bits de entrada B_x y, mediante desplazamientos hacia la izquierda, obtener los bits disponibles para representar a x_0, x_1 y x_2 respectivamente, como se muestra en la 3.6 para una segmentación de 2 niveles. Se comienza por la unidad P2S, indicando con λ_0 el tipo de segmentación utilizada en el nivel 0 (segmentación externa), y luego detectando la cantidad de "0's" o "1's" a la izquierda presenta x, es decir se identifica el segmento externo. A continuación, sabiendo la cantidad de bits utilizados por $B_{\hat{x}0}$ se realiza el desplazamiento a la izquierda, descartando estos bits y guardando el resto. El próximo paso consiste en obtener la cantidad B_{x1} de bits, los cuales se obtienen de la ROM0. Se realiza el desplazamiento a la izquierda nuevamente y terminamos obteniendo x_2.

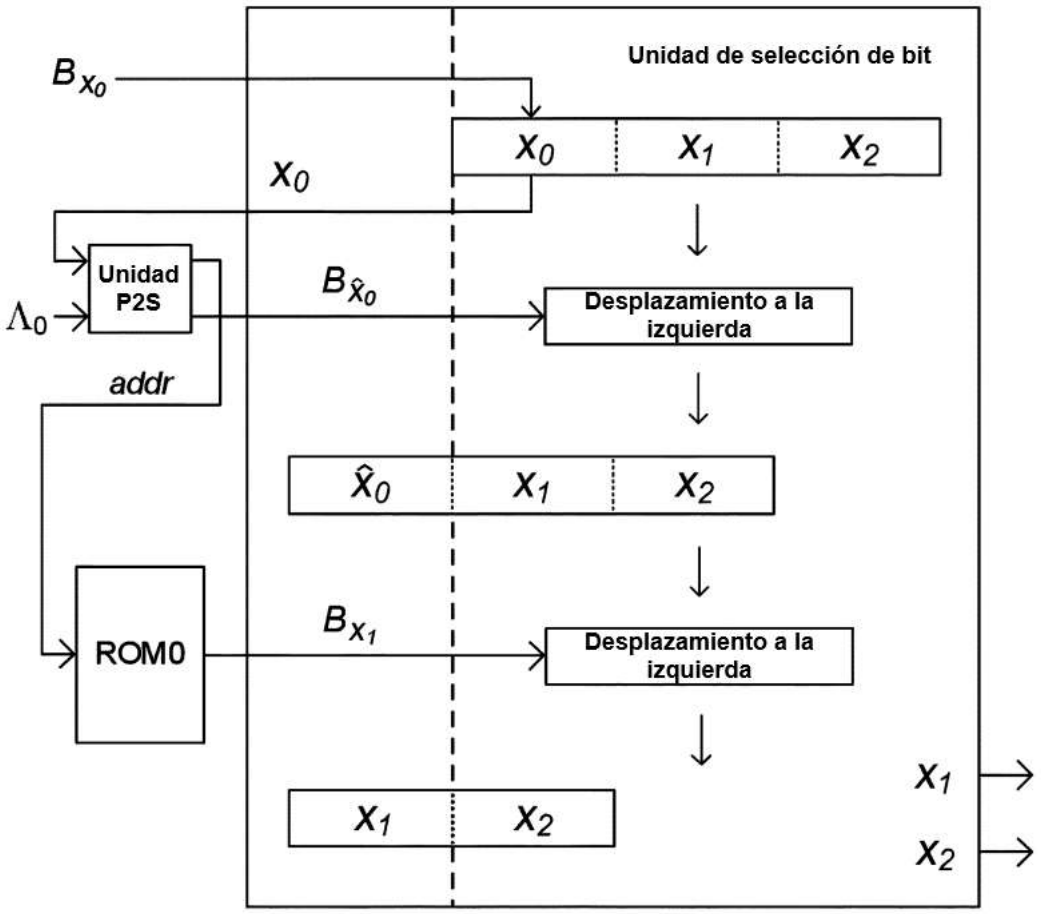


FIGURA 3.6: Representación de la unidad de selección de bit para una segmentación de 2 niveles.

La primera etapa utiliza el generador de números aleatorios uniformes (URNG) Tausworthe, presentado en la Fig. 3.8, elegido por sus grandes propiedades de aleatoriedad. Este circuito genera un número aleatorio uniforme llamado x. Esta implementación es la que se presenta teóricamente en [16] aunque los valores para su implementación son extraídos de la tabla 2 de [17]. Cabe aclarar que sus salidas son pseudoaleatorias, pero para un generador Tausworthe de 32 bits se logra una periodicidad de $2^{88} \approx 10^{25}$, más que suficiente para esta implementación.

La segunda etapa incluye la unidad P2S que calculará la dirección del segmento externo para un x_0 dado. El cálculo de las direcciones de los segmentos para una partición x_i dada, como se ha expresado anteriormente, se basa en detectar el número de ceros a la izquierda para segmentos que inician en cero, y el número de unos a la izquierda para segmentos que inician en uno.

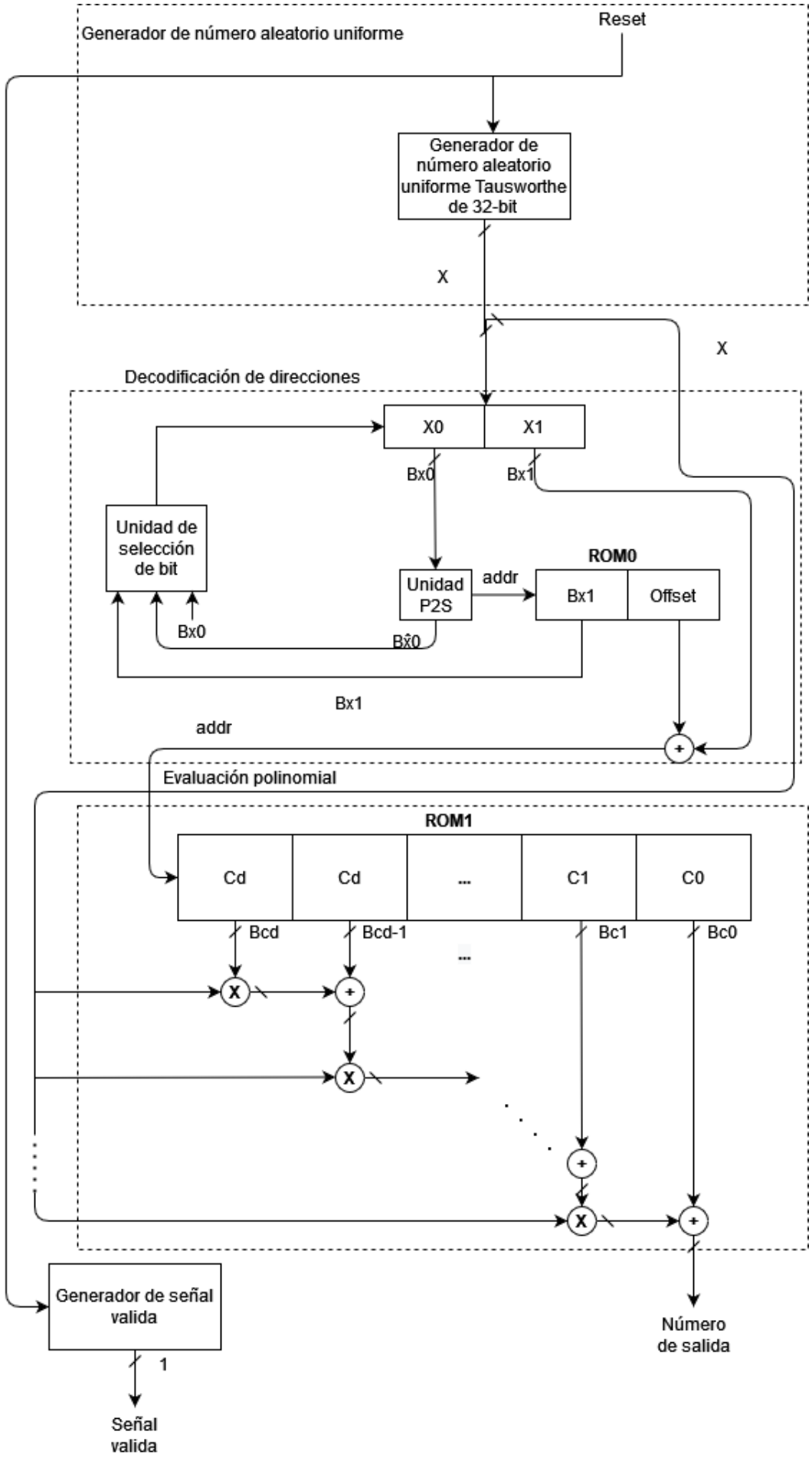


FIGURA 3.7: Arquitectura implementada para indexación e interpolación de ICDF.

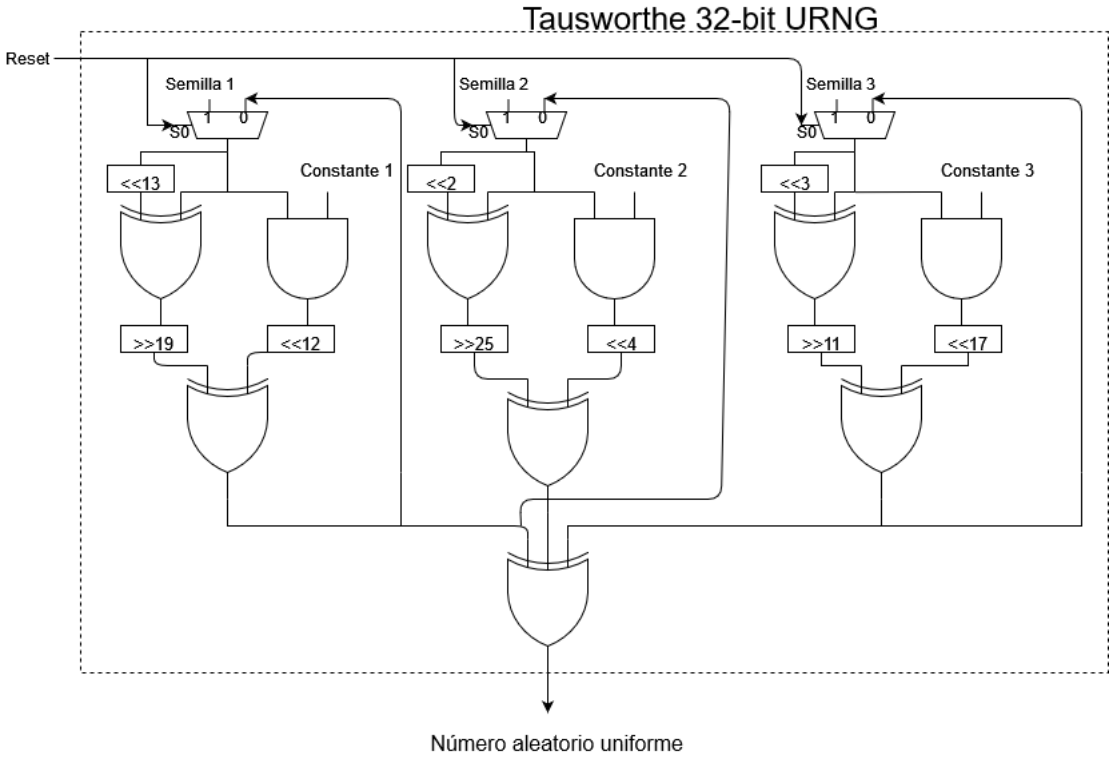


FIGURA 3.8: Arquitectura de un generador de número aleatorio uniforme Tausworthe de 32 bits de salida.

Específicamente, los números de segmentos (direcciones) pueden ser determinados a partir de las siguientes expresiones, las cuales fueron extraídas del trabajo realizado por Dong-U Lee et al. [6]:

$$P2S_{L\_addr} = \begin{cases} B_{xi} - LZD(x_i),\ si\ MSB(x_i) = 0 \\ B_{xi},\ si\ MSB(x_i) = 1 \end{cases} \tag{3.1}$$

$$P2S_{R\_addr} = \begin{cases} 0,\ si\ MSB(x_i) = 0 \\ LOD(x_i),\ si\ MSB(x_i) = 1 \end{cases} \tag{3.2}$$

$$P2S_{LR\_addr} = \begin{cases} B_{xi} - LZD(x_i),\ si\ MSB(x_i) = 0 \\ B_{xi} + LOD(x_i) - 1,\ si\ MSB(x_i) = 1 \end{cases} \tag{3.3}$$

Donde LZD(x_i), significa detector de ceros al inicio (Leading Zero Detector), LOD(x_i), detector de unos al inicio, (Leading Ones Detector), MSB(x_i) es el bit más significativo (Most Significant Bit) y B_{xi} determinan el número de ceros a la izquierda si x_i inicia con 0 o el número de unos a la izquierda si x_i inicia con 1.

Además, como en este caso se implementó el sistema de manera que la cantidad de bits utilizados para la segmentación externa x_0 sean variables,lo que provoca que el siguiente nivel de segmentación x_1 utilice los bits adyacentes a estos; se debe determinar el ancho o la cantidad de bits para cada segmento, con el objetivo de definir los bits disponibles para el número x_2 utilizado en la evaluación polinomial. El cómputo de estos bits x_0 se realiza de distinta manera según la segmentación utilizada. Cada caso se expresa en las siguientes ecuaciones:

$$US:\ B_{\hat{x}i} = B_{xi} \tag{3.4}$$

$$P2S_L\ B_{\hat{x}i} = \begin{cases} B_{xi},\ si\ P2S_L\_addr = 0 \\ B_{xi} - P2S_L\_addr + 1,\ Para\ el\ resto\ de\ los\ casos \end{cases} \tag{3.5}$$

$$P2S_R\ B_{\hat{x}i} = \begin{cases} B_{xi},\ si\ MSB(x_i) = 0 \\ P2S_R\_addr + 1,\ si\ MSB(x_i) = 1 \\ P2S_R\_addr,\ si\ P2S_R\_addr = s_i - 1 \end{cases} \tag{3.6}$$

$$P2S_{LR}\ B_{\hat{x}i} = \begin{cases} B_{xi},\ si\ P2S_{LR}\_addr = 0\ o\ s_i - 1 \\ B_{xi} - P2S_{LR}\_addr + 1,\ si\ MSB(x_i) = 0\ y\ P2S_{LR}\_addr \neq 0 \\ P2S_{LR}\_addr - B_{xi} + 2,\ Para\ el\ resto\ de\ los\ casos \end{cases} \tag{3.7}$$

Otro punto importante a mencionar es que es posible tener cualquier cantidad de niveles L en pasos de segmentación anidados Λ, siempre y cuando $\sum_{i=0}^{L} B_{xi} \leq B_x$. Cuantos más niveles se usen, más cerca estará el número total de segmentos M de ser óptimo. Pero aumentar la segmentación también aumenta la complejidad, ya que se deben anidar mayor cantidad de etapas en cascada, lo que aumenta el retardo para encontrar el segmento deseado. Por lo que generalmente se suele considerar una cantidad de niveles L=2, que consiste en una segmentación externa y una interna.

Por último, la tercera etapa consiste en la evaluación polinomial, que se realiza utilizando la regla de Horner mencionada en 2.2.5. En este caso, la implementación difiere de la realizada en [6] debido a que en este caso se utiliza una FPGA con SoC, el microcontrolador integrado permite operar con punto flotante de doble precisión. Lo que permite ahorrar una gran cantidad de hardware, ya que sino las multiplicaciones para lograr los distintos órdenes de magnitud llevarían a utilizar multiples multiplicadores, los cuales son implementados por sumadores en cadena y esto requeriría un mayor espacio disponible en el dispositivo, lo que en principio se busca minimizar, ya que el codificador y decodificador generalmente suelen ocupar todo el espacio disponible.

Programa en C++ para generación del canal - Análisis del software

Dicho esto, se procede a explicar el código desarrollado para obtener las funcionalidades de segmentación, interpolación y reconstrucción de la función segmentada.

```
1  #include <iostream>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/shm.h>
8  #include <sys/mman.h>
9  #include <boost/dynamic_bitset.hpp>
10 #include <math.h>          //Pow
11 #include "Include/taus176.hpp"
12
13 using namespace boost;
14
15 //Coeficientes
16 static const float ROM0 [10][13][2] = { {{3,0},{1,8},{1,10},{1,12},{1,14},{1,16},{2,18},{3,22},{3,30},{3,38},{3,46},{3,54},{2,62}},
17                                         {{3,0},{1,8},{1,10},{1,12},{1,14},{1,16},{2,18},{3,22},{3,30},{3,38},{3,46},{3,54},{2,62}},
18                                         {{2,0},{1,4},{1,6},{1,8},{1,10},{1,12},{2,14},{3,18},{3,26},{3,34},{3,42},{3,50},{2,58}},
19                                         {{2,0},{1,4},{1,6},{1,8},{1,10},{1,12},{2,14},{3,18},{3,26},{3,34},{3,42},{3,50},{2,58}},
20                                         {{2,0},{1,4},{1,6},{1,8},{1,10},{1,12},{2,14},{3,18},{3,26},{3,34},{3,42},{3,50},{2,58}},
21                                         {{1,0},{1,2},{1,4},{1,6},{1,8},{1,10},{2,12},{3,16},{3,24},{3,32},{3,40},{3,48},{2,56}},
22                                         {{1,0},{1,2},{1,4},{1,6},{1,8},{1,10},{2,12},{3,16},{3,24},{3,32},{2,40},{2,44},{2,48}},
23                                         {{2,0},{1,4},{1,6},{1,8},{1,10},{2,12},{3,16},{3,24},{2,32},{2,36},{3,40},{0,0},{0,0}},
24                                         {{1,0},{1,2},{1,4},{1,6},{1,8},{2,10},{2,14},{2,18},{2,22},{2,26},{3,30},{0,0},{0,0}},
25                                         {{1,0},{1,2},{1,4},{1,6},{1,8},{2,10},{2,14},{2,18},{2,22},{2,26},{3,30},{0,0},{0,0}} };
26
27 static const float C0 [10][66] = {{0.031284198864837978182329436549 48,0.041185307894009148821368171411 45,0.048979017230576897545812897760698,0.
28                                   {0.027884020326869964068183804783985,0.036709009792681351891374674778 49,0.043655646032330255867925927759643,0
```

```
                                    {0.035141465670946524235684904624577,0.046259570776656794288683727245370,0.055009115287572123154102854
                                    {0.031319864246628567605590376388136,0.041228885852366184416872840756696,0.049026925605952301201462262
                                    {0.027913858392020950149881031165933,0.036745283194116885872038125171457,0.043695293440144278296344282
                                    {0.035175036456742386770102370974 2,0.046296332876410772882991295773536,0.055044314469302253745031094920
                                    {0.031349784240959237269219528343456,0.041261650108247241441894459512696,0.049058296909706214372626220
                                    {0.027940524615048270562489562962583,0.036774484367921654592858971000169,0.043723253144018772720080789
                                    {0.035200750618249267154258319578 73,0.046315955925981699969895544199971,0.055051841823000019493505874 3
                                    {0.031372702011528225130021496624977,0.041279139169593082048237420167425,0.049065005670738640053407664

static const float C1 [10][66] = {{45.037578408383396322278713341802,34.388123242973001936206856044009,28.958044951642840203476225724444,
                                    {40.142589466313502555294689955 19,30.650589188916711691490490920842,25.810688569873843078994468669407,2
                                    {25.307119281003085831116550252773,19.327841511722695599928556475788,16.279805663106554192154362681322,
                                    {22.554993800654450808451656484976,17.225956879420259326707309810445,14.509392069880133746551109652501,
                                    {20.102159384440771106028478243388,15.352650229040692764215236820746,12.931509294023229017511766869575,
                                    {12.677422812004582652889439486898,9.686867139771607071452308446 1689,8.163103299463275064340450626 3323,
                                    {11.298764974317490938915398146491,8.633429425899677767 1200106851757,7.275373473727608342187522794 12,6.
                                    {10.070034883111218930562063178513 2,7.694552075144572178544422058621 4,6.484183433733146095789834362221 9
                                    {6.355018251565055287244376813 76,4.860505137819195908832625718787 3,4.099660940312315204892001929 6929,3.
                                    {5.663915978564121544991394330281 8,4.331929763885220552310784114 5247,3.653826659083692973695178807247 4

static const float C2 [10][66] = {{-5235.55176152164767700014635920 52,-2360.763495075959326641168445348 7,-1412.965512323704388109035789 9
                                    {-4666.516549510660297528374940156 9,-2104.179735210264425404602661728 9,-1259.394854156990959381801076
                                    {-1468.863892608480455237440764904,-661.812267208441426191711798310 28,-395.799536604719037313770968 46
                                    {-1309.126322277710642083548009395 6,-589.840804017622758692596107721 33,-352.756708311972886349394684 6
                                    {-1166.760062866771022527245804667 5,-525.696169930891187505039852112 53,-314.394747215448717270191991 7
                                    {-366.852502838349778357951436191 8,-165.024030414823471346608130261 3,-98.53116546205646386624721345 6
                                    {-326.957637311402663726767059415 58,-147.077821921888556744306697510 18,-87.815993653407744545802415 81
                                    {-291.401300983704913960536941885 95,-131.083246766550644224480492994 19,-78.266086726746465274118236 26
                                    {-91.412382869049466194155684206635,-40.979360582570592441697954200208,-24.378452538482576983369654 04
                                    {-81.471371989067719709964876528829,-36.522893563337355260500771692023,-21.727318695173078566540425 52
```

```
//LW Bus: PIO
#define FPGA_LW_BASE 0xFF200000
#define FPGA_LW_SPAN 0x00200000
//Light weight bus base
void *h2p_lw_virtual_base;
//Light weight read/write
volatile unsigned int * lw_noise_enable_ptr = NULL;
volatile unsigned int * lw_coded_buf_ptr_32 = NULL;
volatile unsigned int * lw_coded_buf_ptr_28 = NULL;
volatile unsigned int * lw_decoding_enable_ptr = NULL;
volatile unsigned int * lw_demodulated_buf_ptr_32 = NULL;
volatile unsigned int * lw_demodulated_buf_ptr_28 = NULL;
volatile unsigned int * lw_ack_finish_ptr = NULL;
volatile bool * lw_encoder_ack_ptr = NULL;
volatile bool * lw_noise_ack_ptr = NULL;
volatile bool * lw_decoding_ack_ptr = NULL;
volatile bool * lw_start_ptr = NULL;
volatile bool * lw_stop_ptr = NULL;
volatile int * lw_state_ptr = NULL;
volatile bool * lw_encoding_enable_ptr = NULL;
volatile bool * lw_idle_enable_ptr = NULL;
//Offset for write and read
#define ACK_FINISH 0x00000000
#define NOISE_ENABLE_WRITE 0x00000010
#define CODED_BUF_32 0x00000020
#define CODED_BUF_28 0x00000030
#define DECODING_ENABLE 0x00000040
#define DEMODULATED_BUF_32 0x00000050
#define DEMODULATED_BUF_28 0x00000060
#define ENCODER_ACK 0x00000070
#define NOISE_ACK 0x00000080
#define DECODING_ACK 0x00000090
#define START 0x000000a0
#define STOP 0x000000b0
#define STATE 0x000000c0
#define ENCODING_ENABLE 0x000000d0
#define IDLE_ENABLE 0x000000f0
// /dev/mem file id
int fd;


int Leading_detector (dynamic_bitset<> x,uint8_t n_bits){
  //Obtengo el numero de 0's a la izquierda y de 1's a la izquierda, dependiendo si el MSB = 0 o 1 respectivamente
  int LD = 0;
  int i;
  char flag;
  for (i=n_bits-1; i>=0; i--) {
    if ((!x.test(i))&&flag!='u') {
      LD++;
      flag = 'c';
    }
    else if ((x.test(i))&&flag!='c')
    {
      LD++;
      flag = 'u';
    }
    else break;
  }
  return LD;
}

float ICDF(uint32_t x,int IH, int SNR, uint8_t Bx0, uint8_t Bx1, uint8_t Tprim_size, uint8_t ROM0_size){
  dynamic_bitset<> x_bitset(32,x);
  uint8_t addr = 0;
```

```
127     uint8_t Bx_sombrero = 0;
128     uint8_t offset;
129     float y_2 = 0;
130      //P2S.... Aca voy a tener Ao = [0,1,2,3,....,n] y con esto y la seleccion de Bx'0 tengo que ir sacando la cantidad de bits
131     //para cada segmento. Depende la cantidad de bits que vaya a usar para Bx0, Bx1 y los que queden para Bx2
132     dynamic_bitset<> x0 (Bx0,x>>(32-Bx0)); //Hago Bx0 una serie de bits los cuales son los primeros (52-Bx0) bits de x
133     dynamic_bitset<> x1 (Bx1,x>>(32-Bx0-Bx1)); //Tomo los bits de x que siguen despues de Bx0
134
135
136     switch(IH)
137     {
138       //Segmentacion P2SL
139       case 1:
140       {
141         //Obtengo el valor para indexar la ROM0, es decir, tomo el coeficiente que se asocia con el Bx0--Bx1 que obtuve de x
142         if(!x0.test(x0.size()-1)){
143           addr = Bx0 - Leading_detector(x0,Bx0); //Tengo la cantidad de bits totales que tiene x0 y detecto los 1's o 0's a la izquierda
144         }
145         else
146         {
147           addr = Bx0;
148         }
149         //Calculo la cantidad de bits constantes de x0 para el segmento x
150         if (addr == 0)
151         {
152           Bx_sombrero = Bx0;
153         }
154         else
155         {
156           Bx_sombrero = Bx0 - addr +1;
157         }
158       }
159       break;
160
161       //Segmentacion P2SR
162       case 2:
163       {
164         //Obtengo el valor para indexar la ROM0
165         if(!x0.test(x0.size()-1)){
166             addr =  0;
167         }
168         else{
169             addr = Leading_detector(x0,Bx0);
170         }
171         //Calculo la cantidad de bits constantes de x0 para el segmento x
172         if(!x0.test(x0.size()-1)){
173           Bx_sombrero = Bx0;
174         }
175         else if(x0.test(x0.size()-1))
176           Bx_sombrero = addr +1;
177         else if(addr == Tprim_size)
178           Bx_sombrero = addr;
179       }
180       break;
181
182       //Segmentacion P2SLR
183       case 3:
184       {
185         //Obtengo el valor para indexar la ROM0
186         if(!x0.test(x0.size()-1)){
187           addr = Bx0 - Leading_detector(x0,Bx0);
188         }
189         else
190         {
191           addr = Bx0 + Leading_detector(x0,Bx0)-1;
192         }
193         //Calculo la cantidad de bits constantes de x0 para el segmento x
194         if (addr == 0 || addr == Tprim_size)
195         {
196           Bx_sombrero = Bx0;
197         }
198         else if (!x0.test(x0.size()-1) && addr != 0)
199         {
200           Bx_sombrero = Bx0 - addr +1;
201         }
202         else
203         {
204           if (addr < ROM0_size)
205           {
206           Bx_sombrero = addr - Bx0 + 2; // En este caso seria +2 si empezara contando de 0
207           }
208           else
209           {
210             Bx_sombrero = Bx0;
211             addr = ROM0_size - 1;
212           }
213         }
214       }
215       break;
216       //Segmentacion uniforme (US)
217       case 4: Bx_sombrero = Bx0;
218       break;
219           }
220   //Indexo la ROM0 con addr para obtener Bx1 y offset (Indexo como [fila][columna])
221     Bx1 = ROM0[SNR][addr][0];
222     offset = ROM0[SNR][addr][1];
223
224     if(Bx_sombrero > Bx0){
```

```
225       Bx_sombrero = Bx0;
226     }
227     x0 = (x0>>(Bx0-Bx_sombrero));
228     x1 = (x_bitset>>(32-Bx_sombrero-Bx1));
229     x0.resize(Bx_sombrero);
230     x1.resize(Bx1);
231 
232     dynamic_bitset<> x01(Bx_sombrero+Bx1,x>>(32-Bx_sombrero-Bx1));
233     dynamic_bitset<> x2(32-Bx_sombrero-Bx1,x);
234 
235     float x01_real = (x01.to_ulong())*pow(2,-Bx_sombrero-Bx1);
236     float x2_real = x2.to_ulong()*pow(2,-32+Bx_sombrero+Bx1);
237 
238 
239     float c0_prim = C0[SNR][x1.to_ulong()+offset-1]+C1[SNR][x1.to_ulong()+offset-1]*x01_real+
240                     C2[SNR][x1.to_ulong()+offset-1]*x01_real*x01_real;
241     float c1_prim = (C1[SNR][x1.to_ulong()+offset-1]+2*C2[SNR][x1.to_ulong()+offset-1]*x01_real)*pow(2,-(Bx_sombrero+Bx1));
242     float c2_prim = C2[SNR][x1.to_ulong()+offset-1]*pow(2,-2*(Bx_sombrero+Bx1));
243 
244     y_2 = (x2_real * c2_prim + c1_prim);
245     y_2 *= x2_real;
246     y_2 += c0_prim;
247 
248     return y_2;
249   };
250 
251   int main()
252   {
253       //-----------------VARIABLES DEL GENERADOR DE RUIDO-----------------------------------------------------------------------------
254       taus176 taus;
255       uint16_t seed;
256       uint8_t len = 60;
257       int IH = 0;
258       int SNR = 0;
259       int decoded_words = 0;
260       uint8_t Bx0 = 0;
261       uint8_t Bx1 = 0;
262       uint8_t Tprim_size = 0;
263       uint8_t ROM0_size = 0;
264       //-------------User Input-------------------------------------------------------------------------------------------------------
265       cout<<"Ingrese␣la␣semilla␣inicial␣del␣generador␣de␣ruido:␣";
266       cin >> seed;
267       //seed = 1;
268       cout<<"Ingrese␣la␣SNR␣del␣ruido␣de␣salida,␣entre␣9,␣8,␣7,␣6,␣5,␣4,␣3,␣2,␣1␣y␣0:␣";
269       cin >> SNR;
270 
271       uint32_t x[len];
272 
273 
274       //-----------------VARIABLES DEL MODULADOR--------------------------------------------------------------------------------------
275 
276       vector<float> TxT_LSB(32);
277       vector<float> TxT_MSB(28);
278 
279       std::vector<float> Rx_LSB(32);
280       std::vector<float> Rx_MSB(28);
281       dynamic_bitset<uint8_t> Rx_demod_LSB (32);
282       dynamic_bitset<uint8_t> Rx_demod_MSB (28);
283 
284       float threshold = 0.0;
285       //-----------------INDEXO LOS COEFICIENTES--------------------------------------------------------------------------------------
286       switch (SNR) {
287         case 0:
288               Tprim_size = 66;
289               ROM0_size = 13;
290               Bx0 = 7;
291               Bx1 = 3;
292               IH = 3;
293               threshold = 1.7;
294             break;
295       case 1:
296               Tprim_size = 66;
297               ROM0_size = 13;
298               Bx0 = 7;
299               Bx1 = 3;
300               IH = 3;
301               threshold = 1.7;
302             break;
303       case 2:
304             ROM0_size = 13;
305             Tprim_size = 62;
306             Bx0 = 7;
307             Bx1 = 3;
308             IH = 3;
309             threshold = 1.5;
310             break;
311       case 3:
312             Tprim_size = 62;
313             ROM0_size = 13;
314             Bx0 = 7;
315             Bx1 = 3;
316             IH = 3;
317             threshold = 1.3;
318             break;
319       case 4:
320             Tprim_size = 62;
321             ROM0_size = 13;
322             Bx0 = 7;
```

```
323             Bx1 = 3;
324             IH = 3;
325             threshold = 1.17;
326             break;
327       case 5:
328             Tprim_size = 60;
329             ROM0_size = 13;
330             Bx0 = 7;
331             Bx1 = 3;
332             IH = 3;
333             threshold = 1.05;
334             break;
335       case 6:
336             Tprim_size = 52;
337             ROM0_size = 13;
338             Bx0 = 7;
339             Bx1 = 3;
340             IH = 3;
341             threshold = 0.9421;
342             break;
343       case 7:
344             Tprim_size = 48;
345             ROM0_size = 11;
346             Bx0 = 6;
347             Bx1 = 3;
348             IH = 3;
349             threshold = 0.8397;
350             break;
351       case 8:
352             Tprim_size = 38;
353             ROM0_size = 11;
354             Bx0 = 6;
355             Bx1 = 3;
356             IH = 3;
357             threshold = 0.75;
358             break;
359       case 9:
360             Tprim_size = 38;
361             ROM0_size = 11;
362             Bx0 = 6;
363             Bx1 = 3;
364             IH = 3;
365             threshold = 0.6648;
366             break;
367
368    }
369
370       //----------------------Memory Mapping--------------------------------------
371       int i = 0;
372       //=== Get FPGA addresses ========
373       //open /dev/mem
374       if ((fd = open ("/dev/mem",(O_RDWR | O_SYNC))) == -1)
375       {
376           printf("ERROR: Can't Open memory. \n");
377           return (1);
378       }
379       //Get virtual address that maps to physical
380       //For light weight AXI bus
381       h2p_lw_virtual_base = mmap( NULL, FPGA_LW_SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, FPGA_LW_BASE);
382       if(h2p_lw_virtual_base == MAP_FAILED)
383       {
384           printf("ERROR: Can't map memory. \n");
385           close(fd);
386           return (1);
387       }
388       //Get the addresses that map to the two parallel ports on the light-weight bus
389       lw_noise_enable_ptr = (unsigned int *)(h2p_lw_virtual_base + NOISE_ENABLE_WRITE);
390       lw_coded_buf_ptr_32 = (unsigned int *)(h2p_lw_virtual_base + CODED_BUF_32);
391       lw_coded_buf_ptr_28 = (unsigned int *)(h2p_lw_virtual_base + CODED_BUF_28);
392       lw_decoding_enable_ptr = (unsigned int *)(h2p_lw_virtual_base + DECODING_ENABLE);
393       lw_demodulated_buf_ptr_32 = (unsigned int *)(h2p_lw_virtual_base + DEMODULATED_BUF_32);
394       lw_demodulated_buf_ptr_28 = (unsigned int *)(h2p_lw_virtual_base + DEMODULATED_BUF_28);
395       lw_ack_finish_ptr = (unsigned int *)(h2p_lw_virtual_base + ACK_FINISH);
396      lw_encoder_ack_ptr = (bool *)(h2p_lw_virtual_base + ENCODER_ACK);
397      lw_noise_ack_ptr = (bool *)(h2p_lw_virtual_base + NOISE_ACK);
398      lw_decoding_ack_ptr = (bool *)(h2p_lw_virtual_base + DECODING_ACK);
399      lw_start_ptr = (bool *)(h2p_lw_virtual_base + START);
400      lw_stop_ptr = (bool *)(h2p_lw_virtual_base + STOP);
401      lw_state_ptr = (int *)(h2p_lw_virtual_base + STATE);
402      lw_encoding_enable_ptr = (bool *)(h2p_lw_virtual_base + ENCODING_ENABLE);
403      lw_idle_enable_ptr = (bool *)(h2p_lw_virtual_base + IDLE_ENABLE);
404
405      //Espero que el mensaje este codificado...
406
407        uint32_t coded_buffer_32;
408        uint32_t coded_buffer_28;
409
410        //---------------Structural Code-----------------------------------------
411       taus.taus_set(&taus.state_1 ,seed);
412      // *(lw_demodulated_buf_ptr_32) = 0;
413      // *(lw_demodulated_buf_ptr_28) = 0;
414      do {
415        (*(lw_idle_enable_ptr) = 0);
416        (*(lw_encoding_enable_ptr) = 0);
417        (*(lw_noise_enable_ptr)) = 0;
418        (*(lw_decoding_enable_ptr)) = 0;
419        while (*(lw_state_ptr) != 0);
420        (*(lw_idle_enable_ptr) = 1);
```

```
421       (*(lw_start_ptr)) = 1;
422        (*(lw_stop_ptr)) = 0;
423        while (*(lw_state_ptr) != 1);
424        (*(lw_idle_enable_ptr) = 0);
425        (*(lw_encoding_enable_ptr) = 1);
426        while (*(lw_encoder_ack_ptr) == 0);
427         coded_buffer_32 = *(lw_coded_buf_ptr_32);
428         coded_buffer_28 = *(lw_coded_buf_ptr_28);
429         dynamic_bitset <> Tx_LSB (32,coded_buffer_32);
430         dynamic_bitset <> Tx_MSB (28,coded_buffer_28);
431         for (int i = 0 ; i < 32 ; ++i)
432            TxT_LSB[i] = 2.0*(float)Tx_LSB[i]-1.0; // Modulacion_LSB
433         for (int i = 0 ; i < 28 ; ++i)
434            TxT_MSB[i] = 2.0*(float)Tx_MSB[i]-1.0; // Modulacion_MSB
435         for (int i = 0; i < len; i++){
436          x[i] = taus.taus_get(&taus.state_1);
437          float y = ICDF(x[i],IH, SNR, Bx0, Bx1, Tprim_size, ROM0_size);
438
439          //-----------------------------DEMODULACION-----------------------------------------------------------------------
440           if (i<32){
441              Rx_LSB[i] = TxT_LSB[i] + y;
442              if (Rx_LSB[i]>= threshold)
443                  Rx_demod_LSB[i] = 1;
444              else
445                  Rx_demod_LSB[i] = 0;
446
447          }
448          else if (i> 31 & i<len) {
449              Rx_MSB[i-32] = TxT_MSB[i-32] + y;
450              if (Rx_MSB[i-32]>= threshold)
451                  Rx_demod_MSB[i-32] = 1;
452              else
453                  Rx_demod_MSB[i-32] = 0;
454
455          }
456          //-------------------------------Lo mando al PIO-------------------------------------------------------------------
457       }
458       *(lw_demodulated_buf_ptr_32) = Rx_demod_LSB.to_ulong();
459       *(lw_demodulated_buf_ptr_28) = Rx_demod_MSB.to_ulong();
460       while(*(lw_state_ptr) != 6);
461       (*(lw_encoding_enable_ptr) = 0);
462       *(lw_noise_enable_ptr) = 1;
463       while(*(lw_noise_ack_ptr) == 0);
464       while(*(lw_state_ptr) != 2);
465       *(lw_noise_enable_ptr) = 0;
466       *(lw_decoding_enable_ptr) = 1;
467       while(*(lw_decoding_ack_ptr) == 0);
468       decoded_words = decoded_words + 1;
469       *(lw_start_ptr) = 0;
470       *(lw_stop_ptr) = 1;
471       while(*(lw_state_ptr) != 0);
472       *(lw_decoding_enable_ptr) = 0;
473      } while (*(lw_ack_finish_ptr) == 0);
474     cout<<"Fin␣del␣codigo,␣se␣decodificaron:␣"<<decoded_words<<"␣palabras"<<endl;
475 };
```

En principio (líneas 1 a 12), se incluyen las librerías necesarias para utilizar funcionalidades especificas del lenguaje C++ como por ejemplo "stdio.h"que permite utilizar funcionalidades de entrada/salida estándar. En este caso se destacan la librería **boost/dynamic_bitset** la cual implementa funcionalidades que permiten operar de manera binaria con distintas variables. A su vez, la librería **sys/mman** permiten realizar el mapeo de memoria física a memoria virtual, necesario para realizar la interfaz entre el procesador y la FPGA. Por último, la librería **Include/taus176** que contiene las funciones relacionadas al generador tausworthe de 32 bits.

Luego (líneas 16 a 59), se cargan los coeficientes generados por el programa de MATLAB mencionado en la sección 3.3.1 en memoria RAM creando matrices de 3 dimensiones para ROM0 y 2 dimensiones para los coeficientes C0, C1 y C2, ya que se tienen distintos valores de coeficientes para distintas relaciones señal a ruido (SNR), los cuales permiten obtener una mejor representación del fenomeno que se busca simular y obtener graficas, como la de tasa de error de bit (Bit Error Rate) que permiten cuantificar la cantidad de bits erroneos que se tienen en una cantidad de bits transmitidos, la cual suele estar cercana al millón de bits o más. Esto en principio se implementó mediante una función que leía estos valores desde un archivo de texto (.txt) pero este proceso ralentizaba en gran medida la decodificación. Por lo que se optó por la alternativa en la que se disponen de los coeficientes en memoria desde que se inicia el programa y no se pierde procesamiento al cargarlos en la misma.

En las líneas 64 a 102 se realiza la definición de punteros y macros que permiten realizar el direccionamiento de todas las posiciones de memoria asociadas a los

puertos de entrada/salida de la FPGA.

Desde la línea 105 a la 123 se implementa la función que permite determinar los 0's o 1's más significativos según el caso, dependiendo si el bit más significativo es 0 o 1 respectivamente. El "Leading Detector"se utiliza cuando ingresa un nuevo valor de x, generado por el generador de números pseudoaleatorios Tausworthe. La detección del número "dominante a la izquierda"se lleva a cabo chequeando el valor del bit más significativo y luego desplazando un puntero desde esa posición hacia posiciones menos significativas o hacia la derecha. Siempre y cuando los bits consecutivos sean del mismo valor que el MSB, se continua aumentando la cuenta en la variable LD. Cuando un bit difiere, el programa sale del ciclo y retorna LD con la cantidad de 0's o 1's respectiva.

El programa principal se encuentra en las líneas 125 a la 248. Aquí se realizan las tareas asociadas a la selección de segmentación (líneas 137 a 218) según el valor de entrada obtenido del programa de Matlab. Dentro de este "switch/case", el cual es una instancia de selección del lenguaje C++, se implementan las ecuaciones 3.4 3.5 3.6 y 3.7 que permiten obtener la cantidad de bits requeridos para la segmentación externa. Posteriormente, en 220 y 221 se obtienen la cantidad de bits de B_{x1} y el offset para luego indexar los coeficientes. Esto se realiza indexando la matriz ROM0 con los valores referidos a la SNR elegida y la dirección previamente obtenida en el switch de las lineas 136 a 217. Es decir, [SNR] indica el primer conjunto de datos en $B_{x1} = ROM0[SNR][addr][0]$, por ejemplo, para el caso del primer conjunto de datos de ROM0 [(3, 0),(1, 8),(1, 10),(1, 12),(1, 14),(1, 16),(2, 18),(3, 22),(3, 30),(3, 38),(3, 46),(3, 54),(2, 62)], a continuación [addr] permite obtener 1 de estos pares, por ej si [addr] = 0 devolvería los valores (3,0) del conjunto mostrado previamente. Finalmente, el valor [0] permite obtener la cantidad de bits de $B_{x1} = 3$, es decir selecciona el primer valor del par que se obtuvo con [addr] = 0. Para el mismo ejemplo, el valor del offset sería el segundo del conjunto (3,0), es decir offset = 0.

En las líneas contiguas se realiza la adaptación de las variables para finalmente en las líneas 239 a 245 indexar los vectores de C0, C1 y C2 (239 a 241), y realizar la evaluación polinomial (243 a 245).

Por otro lado, en la función **main** (líneas 250 a 474) se implementan todas las declaraciones de variables globales, entradas de usuario, instrucciones de mapeo de memoria, entre otras. Esto siempre debe realizarse de esta manera y no en la función anterior, debido a que **main** es imprescindible en cualquier programa C++ ya que representa el punto de inicio de su ejecución. Aquí, en las líneas 285 a 367 se implementa una selección de parametros que depende de la relación señal a ruido elegida por el usuario.

A continuación, de 370 a 402 se realizan las tareas asociadas al mapeo de memoria. Donde primero se obtienen las direcciones de los puertos utilizados en la FPGA, luego se mapea la memoria virtual a la física usando la función **mmap** y por último se asocia a cada puntero la posición de memoria a la que van a apuntar.

Luego, se encuentra la programación estructural (líneas 410 a 474) donde primero, en la línea 410, se obtiene un estado con el que se inicia el generador tausworthe, con una semilla **seed** ingresada por el usuario. Enseguida, se inicia en la línea 413 el control de la máquina de estados general implementada en FPGA, en este caso se inicia seteando todos los **flags** en 0, de manera que el programa no inicie con señales espurias. Luego, una vez que se da **start** desde la FPGA, el programa inicia y se observa el estado y ajusta la bandera asociada a cada etapa según el caso. Por ejemplo, en la línea 418 se espera a que el estado sea distinto de 0, que se asocia con el estado S_START, ya que de esta manera se asegura que el proceso inició y se avanzó al siguiente estado, el cual sería S_ENCODING, asociado con el valor 1.

Una vez realizada la codificación, en las líneas 430 a 433 se realiza la modulación BPSK de los bits codificados recibidos de la FPGA. Aquí se separa en LSB y MSB debido a que se reciben 60 bits y el máximo posible para almacenar en una variable es de 32, por lo que se separan en dos variables de 32 y 28 bits respectivamente. Siendo 32 la parte menos significativa y 28 la más significativa.

El próximo paso consiste en sumar el ruido al mensaje modulado, por lo que en la línea 434 se crea un lazo que ejecuta la función ÏCDF"(línea 436), la cual devuelve un valor de muestra del ruido generado, por cada bit de mensaje recibidos. Es decir, para el caso particular de haber recibido 60 bits, este bucle genera 60 muestras de ruido y contamina cada uno de los bits individualmente.

Dentro de este ciclo for también se realiza la demodulación del mensaje contaminado con ruido (líneas 439 a 454). La misma se efectúa mediante una comparación con un umbral, de manera que si el valor del bit de mensaje + ruido es mayor ó igual que ese umbral, se considera que el bit recibido era un 1, en caso contrario, se considera que el bit recibido era un 0. Cabe destacar que el umbral varía según la relación señal a ruido elegida, ya que si se usara un mismo umbral para todos los casos, se introduciría error en la demodulación, debido a que no se tiene la misma probabilidad de que la distribución elegida tome los valores 0 o 1.

Una vez finalizada la demodulación, sólo resta devolver el mensaje a la FPGA para que sea decodificado y mantener el control de esta última etapa, lo cual se realiza de las líneas 457 a 472.

3.4.4. Arquitectura del banco de pruebas - Hardware

Memorias

Conjuntamente con el desarrollo de software mencionado previamente, se implementó en la FPGA el codificador y el decodificador en conjunto con sus respectivas máquinas de estado y una máquina de estados general.

Para el almacenamiento de los símbolos de entrada del codificador y el decodificador, el sistema utiliza los mismos tipos de RAM, generados a partir de bloques de memoria dedicados en la FPGA. Las mismas permiten ser configuradas para distintos anchos y cantidades de palabras. Para esta implementación se utilizan memorias de un puerto; sus tamaños son de de 4 bits de ancho por 16 palabras de profundidad. Las FSM del codificador y decodificador se encargan del control de las señales que manejan las memorias, respetando las formas de onda de la Fig. 3.9. A través de la comunicación mediante los registros de control, se asegura que la FPGA no escriba sobre una dirección de memoria ya utilizada, evitando así la posible pérdida de información. Como se mencionó en la sección 2.4 las salidas del codificador y decodificador se envían a dos sistemas de primera entrada, primera salida (FIFO). Si bien se podrían haber utilizado memorias RAM al igual que las entradas, la razón por la que no se hizo fue debido a que es necesario que ambas estén sincronizadas a la hora de calcular la tasa de error binaria y simplemente habilitando la lectura de ambas FIFOs en el mismo instante, se asegura este sincronismo. Por otra parte, también queda asegurado el vaciado una vez que la memoria FIFO es leída, por lo que no existirían símbolos residuales de una palabra procesada con anterioridad.

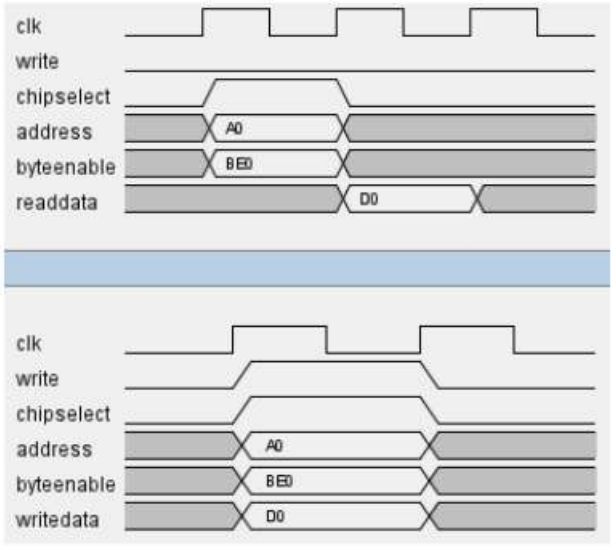


FIGURA 3.9: Formas de onda para lectura y escritura de memorias RAM

Máquinas de estado

Las máquinas de estado deben reconocer el instante en el que un mensaje termina de ser codificado o decodificado. Ni el codificador ni el decodificador cuentan con la capacidad de identificar el instante final, ya que son bloques combinacionales de procesamiento puro. En el caso de ambos, la latencia de un mensaje completo es generalmente mayor que el período de reloj con el que trabaja la FSM. Por lo tanto, se utiliza un contador configurable cuya función es mantener a la FSM en un estado de espera hasta que el mensaje correcto se estabilice a la salida. El control inicia el conteo una vez que el HPS habilita el comienzo de la codificación. Una vez codificado, se detiene el conteo hasta que el mensaje codificado atraviesa el canal simulado en el procesador. Cuando se tiene el mensaje contaminado con ruido, se envía el mismo de vuelta a la FPGA y se inicia el proceso de decodificación, rehabilitando el conteo.

Las máquinas de estado del codificador y del decodificador presentan grandes diferencias. Para el caso del codificador se tiene un estado inicial donde se espera el inicio de codificación mediante un registro "start". Una vez iniciado el proceso, se pasa al siguiente estado donde se realiza la codificación y una vez finalizada la misma, se llega al estado final donde se transmite un ack a la máquina de estados general. El diagrama de la máquina se muestra en la Fig. 3.10 y como se puede ver, es realmente sencillo ya que sólo se espera a que se generen los símbolos de paridad para sumarlos con el mensaje. A su vez, el código utilizado para realizar esta implementación se muestra en B.1.3

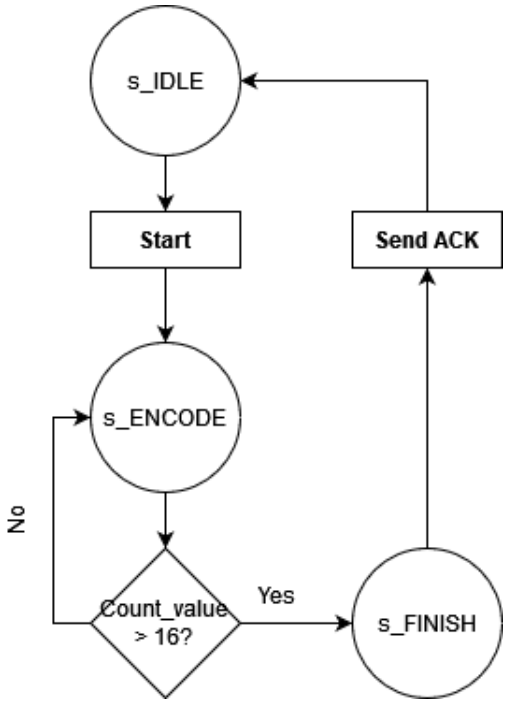


FIGURA 3.10: Máquina de estados - Codificador

Por su parte, el decodificador consta también de un estado inicial donde espera la habilitación del registro "start"pero luego realiza el proceso explicado en la sección 2.3, por esta razón se tiene un estado para el cálculo de los síndromes, otro para el algoritmo Euclidiano y el estado de finalización. Al igual que el codificador, también se envía un ack una vez decodificada la palabra. Esta máquina de estados se muestra en la Fig 3.11 y su código es B.1.4.

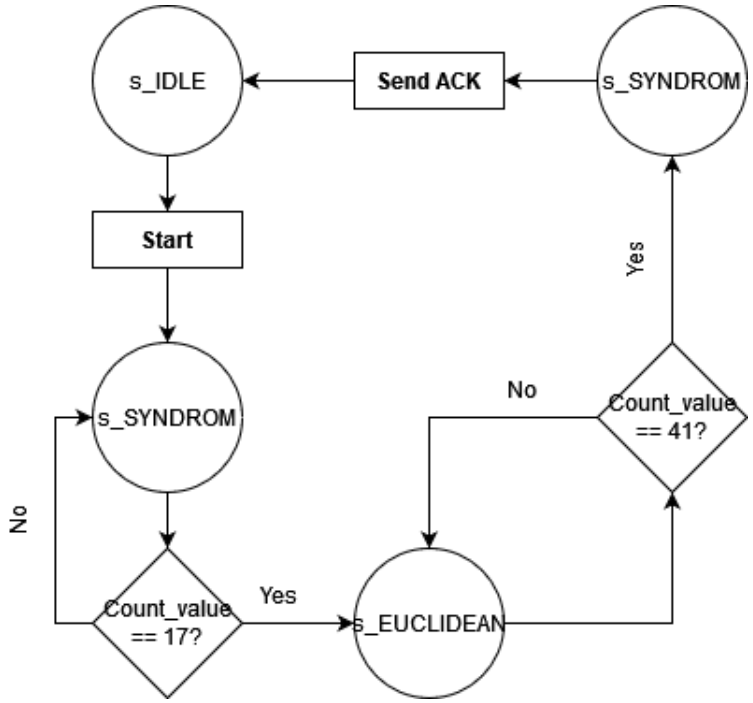


FIGURA 3.11: Máquina de estados - Decodificador

Por otra parte, también se implementó una máquina de estados general que permitió el control del sistema como un conjunto, la cual se muestra en la Fig.3.12 y su código es el B.1.5. A continuación se detallan cada uno de los estados:

s_IDLE: el sistema inicia y se mantiene en este estado hasta que recibe una señal de inicio o "start", lo que produce un avance a la siguiente etapa del proceso, el estado s_START/s_ENCODING.

s_START/s_ENCODING: en este caso se inicia la codificación de palabras utilizadas en el banco de pruebas. Estas últimas son generadas por un LFSR (Linear Feedback Shift Register) de 32 bits, de las cuales se toman los 4 bits menos significativos. Esto es para aumentar la aleatoriedad de mensajes de entrada al codificador, ya que con un LFSR de menor cantidad de bits, se repetirían las secuencias de palabras con mayor frecuencia. Cabe destacar que por cada "ciclo" (desde el estado s_IDLE a s_IDLE nuevamente) del sistema se realiza la codificación, contaminación con ruido y decodificación de una palabra por vez. Además, se llena un registro de 60 bits de longitud, con todos los símbolos que integran el mensaje codificado (información + paridad) el cual es leído antes del siguiente estado por el procesador, para realizar la modulación, demodulación y contaminación con ruido del mensaje. En conjunto con este registro, también se llena una memoria FIFO (First Input, First Output) la cual se utiliza en el estado s_DECODING, para comparar la palabra transmitida con la recibida. Finalmente, cuando el contador alcanza el valor 25, este estado habilita el siguiente.

s_NOISE: aquí, una vez que el procesador terminó de modular, contaminar y demodular el mensaje, se envía el mensaje contaminado a la memoría RAM del decodificador mediante un buffer de 60 bits. Para esto se descomponen los 60 bits recibidos del ARM en nibbles (porciones de 4 bits) ya que cada nibble se corresponde con 1 símbolo. Luego de que este proceso finaliza, se pasa al último estado.

s_DECODING: el último estado es el encargado de decodificar el mensaje recibido y a su vez, de realizar la comparación bit a bit de la palabra transmitida con la

decodificada. Para ello, en principio se espera la confirmación de que la decodificación finalizó, para luego llenar una memoria FIFO con la palabra decodificada, como el estado s_ENCODING, la cual se utilizará en el contador de bits erróneos para determinar si hubo errores en la misma o no. Una vez que la comparación finaliza, se aumenta el contador de palabras decodificadas y se repite el proceso hasta que se alcance el límite establecido de palabras a decodificar, en este caso 1 millón.

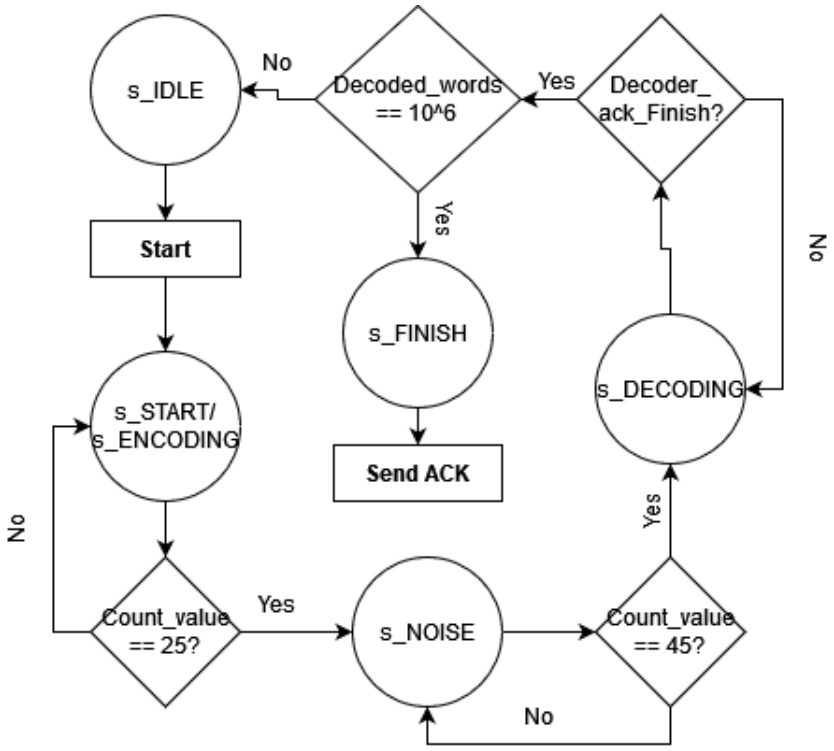


FIGURA 3.12: Maquina de estados finitos para el control general

Implementación de codificador Reed-Solomon

Como se mencionó en la sección 2.3.4 es necesario realizar el cociente entre el mensaje enviado, multiplicado por x^{n-k}, y el polinomio generador $g(x)$ para así obtener un resto $r(x)$ que representa los símbolos de paridad. Para ello, el cálculo de la división es desarrollado utilizando el circuito codificador convencional de la Fig. 3.13, cuyo código es el B.1.6. Cabe aclarar que todos los caminos de datos mostrados proveen valores de 4 bits para el caso de este ejemplo. Además, los sumadores realizan la suma bit a bit módulo-2 y cada sumador consiste en 4 compuertas XOR de 2 entradas.

Cada una de las entradas a los multiplicadores es un elemento de campo constante, que a su vez son los coeficientes del polinomio $g\,(x)$. Para un bloque particular, la información del polinomio de entrada ingresa en el codificador, símbolo por símbolo. Estos símbolos aparecen a la salida del codificador luego de cierta latencia, donde la lógica de control los realimenta a través de un sumador para producir la paridad deseada. Este proceso continúa hasta que los k símbolos del mensaje $M\,(x)$ hayan ingresado al codificador. Durante este lapso, la lógica de control a la salida habilita sólo el camino de datos de entrada, mientras mantiene el camino de paridad/realimentación, habilitado. Con una latencia de salida en el orden de un ciclo de reloj,

el último símbolo de datos aparece a la salida luego de $(k+1)$pulsos de reloj. Así, el resto de la división representado por los 4 símbolos de paridad, queda contenido en los registros tipo-D. La forma de onda de la señal de control cambia y entonces la compuerta AND evita la realimentación a los multiplicadores, de manera que los cuatro símbolos almacenados en el registro, se envían a la salida por el selector, en 4 ciclos de reloj consecutivos.

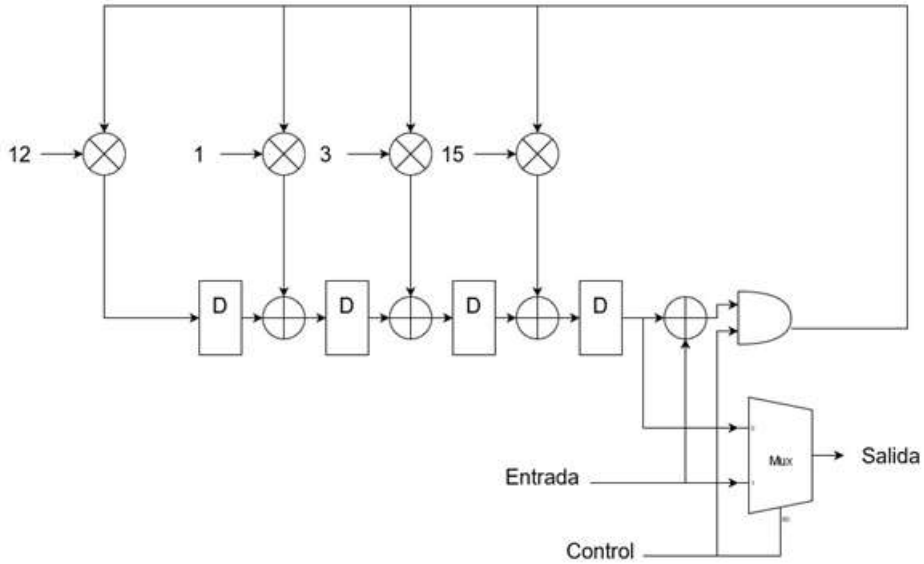


FIGURA 3.13: Codificador Reed-Solomon (15,11)

En cuanto a la implementación de los multiplicadores de la Fig. 3.13, la misma puede realizarse mediante distintos métodos, entre los cuales se pueden mencionar la implementación mediante lógica dedicada para multiplicación por constantes, la utilización de tablas de búsqueda o lookup como memorias de sólo lectura, ó multiplicadores completos de 16 bits. En este caso se optó por emplear multiplicadores completos, cómo el de la Fig. 3.16, ya que esta elección permite la reutilización del diseño para los distintos bloques.

Continuando con el mismo ejemplo, para la aproximación mediante lógica dedicada, se puede lograr la funcionalidad requerida utilizando una representación polinomial de la señal de entrada $a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a_0$. Luego, esta se multiplica por los polinomios formados por los valores 15, 3 ,1 y 12 de la Tabla 2.3. Esta operación implica una versión desplazada de la entrada para cada coeficiente no nulo del polinomio multiplicador. La versión desplazada produce valores en las columnas α^6, α^5 o α^4 , por lo que estos coeficientes son sustituidos por sus valores equivalentes de la Tabla 1 para lograr reducir y reutilizar el hardware. Los valores de los bits para cada una de las columnas α^3,α^2,α^1 y α^0 son sumados para contemplar las contribuciones requeridas para cada bit de salida. A modo de ejemplo, en la Fig. 3.14 se muestra la multiplicación por 15 (=α^3+α^2+α^1+1).

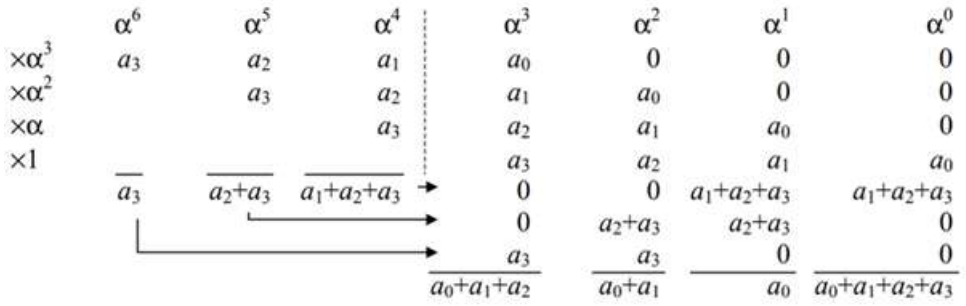


FIGURA 3.14: Multiplicación de constantes. Constante = 15

En este caso $\alpha^6 = \alpha^3+\alpha^2$, $\alpha^5 = \alpha^2+\alpha^1$ y $\alpha^4 = \alpha^1+\alpha^0$, por esta razón en la Fig. 3.14 se ve que los coeficientes que terminan en las columnas de la izquierda, luego son sumados por su equivalente en las demás columnas.

Como las sumas son módulo 2, se implementan simplemente con compuertas OR-exclusiva, ya que no se requiere de acarreo para obtener el resultado. La implementación en hardware de estos multiplicadores individuales se muestra en la Fig. 3.15.

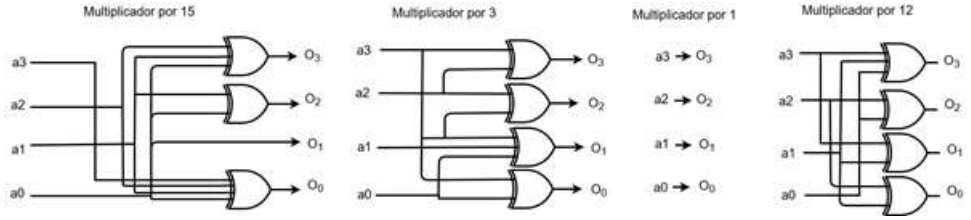


FIGURA 3.15: Multiplicadores por constantes de 4 bits

Alternativamente, cada multiplicador puede implementarse como una tabla de búsqueda o lookup con $2^m = 16$ entradas. Los valores de la tabla pueden obtenerse desplazando cíclicamente los elementos no nulos de la Tabla 2.3, de acuerdo al índice del factor a multiplicar. Esto es así debido a que el índice de multiplicación se suma al índice de la entrada, módulo 15, cambiando así los resultados de acuerdo con el valor que se multiplica. Sin embargo, los valores binarios de los coeficientes del polinomio de la entrada necesitan ordenarse en orden ascendente para coincidir con el direccionamiento binario de la tabla de look-up. Teniendo en cuenta estas consideraciones, se obtienen los resultados presentados en la Tabla 3.1.

		x15 = α^{12}	x3 = α^4	x1 = α^0	x12 = α^6
Forma de Indice	**Forma Decimal**	**Forma Decimal**	**Forma Decimal**	**Forma Decimal**	**Forma Decimal**
0	0	0	0	0	0
α^0	1	15	3	1	12
α^1	2	13	6	2	11
α^4	3	2	5	3	7
α^2	4	9	12	4	5
α^8	5	6	15	5	9
α^5	6	4	10	6	14
α^{10}	7	11	9	7	2
α^3	8	1	11	8	10
α^{14}	9	14	8	9	6
α^9	10	12	13	10	1
α^7	11	3	14	11	13
α^6	12	8	7	12	15
α^{13}	13	7	4	13	3
α^{11}	14	5	1	14	4
α^{12}	15	10	2	15	8

CUADRO 3.1: Tabla de lookup para los multiplicadores por constantes de la Fig. 3.15

Por su parte, los multiplicadores completos elegidos en este diseño, pueden implementarse mediante las técnicas mencionadas previamente, es decir mediante lógica dedicada ó bien utilizando tablas de look-up. A mayor cantidad de posiciones, medida por el valor de 2^{2m}, las tablas de lookup se vuelven más ineficientes ya que a medida que sea mayor la longitud de la palabra, la tabla aumentará exponencialmente.

En este caso se utiliza el arreglo mostrado en la Fig. 3.16, el cual presenta la implementación de un multiplicador de desplazamiento y suma, lo que resulta en el código mostrado en B.1.7. La primera etapa consta de un arreglo de compuertas AND que genera un conjunto de productos desplazados, los cuales producen 7 niveles de significancia. A continuación, las columnas XOR, se encargan de sumar los productos de cada nivel.

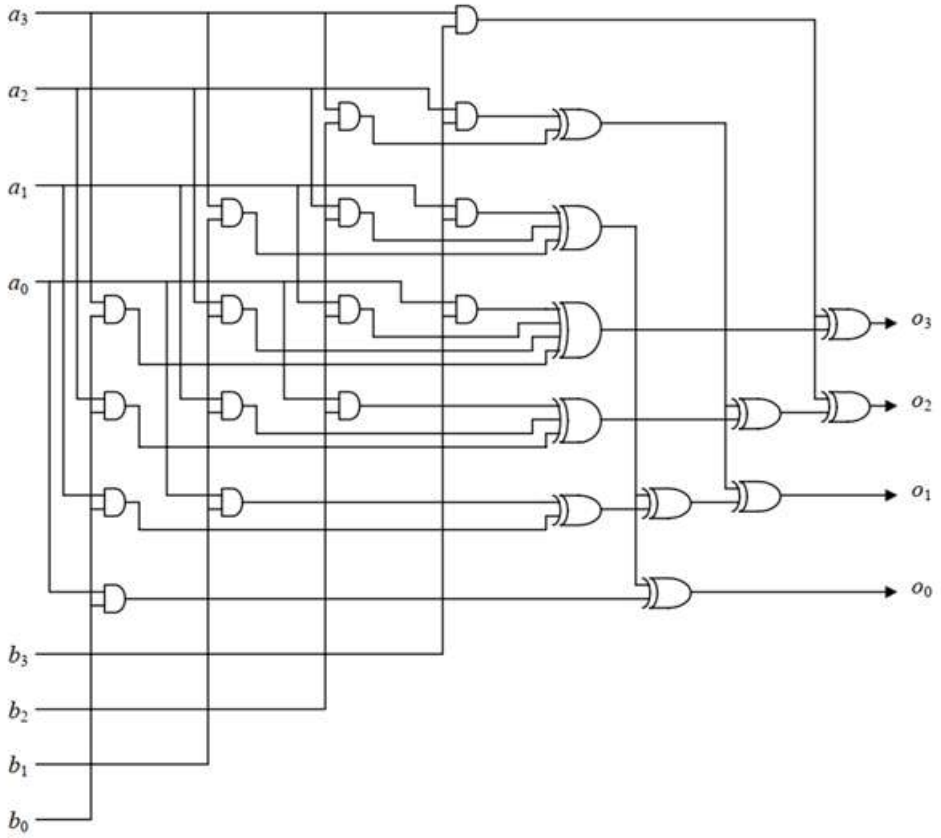


FIGURA 3.16: Multiplicador completo de 4 bits sin acarreo (GF(16)

Por ejemplo, suponiendo que multiplicamos A = 5 (0101 en binario) por B = 2 (0010 en binario) tendríamos $a_3 = 0,\ a_2 = 1,\ a_1 = 0,\ a_0 = 1$ y $b_3 = 0,\ b_2 = 0,\ b_1 = 1,\ b_0 = 0$ por lo que en los primeros 4 niveles, siendo & el símbolo de la operación AND, se tendría:

1. $AND_{1,1} = a_3\&b_0 = 0,\ AND_{1,2} = a_2\&b_0 = 0,\ AND_{1,3} = a_1\&b_0 = 0,\ AND_{1,4} = a_0\&b_0 = 0$
2. $AND_{2,1} = a_3\&b_1 = 0,\ AND_{2,2} = a_2\&b_1 = 1,\ AND_{2,3} = a_1\&b_1 = 0,\ AND_{2,4} = a_0\&b_1 = 1$
3. $AND_{3,1} = a_3\&b_2 = 0,\ AND_{3,2} = a_2\&b_2 = 0,\ AND_{3,3} = a_1\&b_2 = 0,\ AND_{3,4} = a_0\&b_2 = 0$
4. $AND_{4,1} = a_3\&b_3 = 0,\ AND_{4,2} = a_2\&b_3 = 0,\ AND_{4,3} = a_1\&b_3 = 0,\ AND_{4,4} = a_0\&b_3 = 0$

Luego, para el caso de la primer columna de 5 OR-exclusivas, ordenadas de arriba hacia abajo y siendo $\wedge$ el símbolo de la operación XOR, se tiene:

1. $XOR_1 = AND_{3,1} \ \wedge \ AND_{4,2} = 0$
2. $XOR_2 = AND_{4,3} \ \wedge \ AND_{3,2}; \wedge AND_{2,1} = 0$
3. $XOR_3 = AND_{4,4} \ \wedge \ AND_{3,3} \ \wedge \ AND_{2,2} \ \wedge \ AND_{1,1} = 1$
4. $XOR_4 = AND_{3,4} \ \wedge \ AND_{2,3} \ \wedge \ AND_{1,2} = 0$

5. $XOR_5 = AND_{2,4} \wedge AND_{1,3} = 1$

Finalmente, las salidas quedan formadas de la siguiente manera:

- $O_0 = XOR_2 \wedge AND_{1,4} = 0$
- $O_1 = (XOR_5 \wedge XOR_2) \wedge XOR_1 = 1$
- $O_2 = (XOR_4 \wedge XOR_1) \wedge AND_{4,1} = 0$
- $O_3 = XOR_3 \wedge AND_{4,1} = 1$

Finalmente, reordenando por significancia, se tiene el número $O_3O_2O_1O_0 = 1010$ en binario, que en decimal es 10 = 5*2 = A*B. Verificandosé el producto de los dos números ingresados.

Implementación de decodificador Reed-Solomon

Habiendo presentado el desarrollo teórico que sustenta la decodificación Reed Solomon, en esta sección se describirá un acercamiento específico a la decodificación por hardware basada en el algoritmo Euclidiano.

Un diagrama en bloques de las unidades principales de un decodificador Reed-Solomon reflejan las operaciones presentadas en la sección 2.3.5.

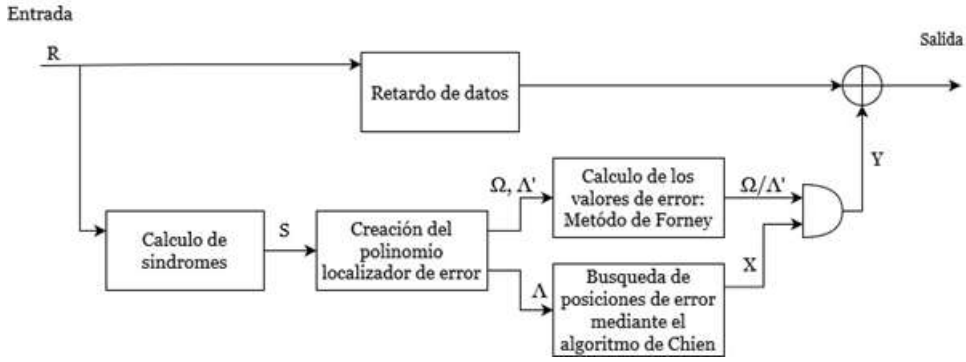


FIGURA 3.17: Diagrama de bloques decodificador Reed Solomon.)

Así, en la Fig. 3.17, la primer operación consiste en calcular los valores del síndrome de la palabra de código entrante. Estos valores son usados para encontrar los coeficientes del polinomio localizador de error Λ_1 ..... Λ_v y del polinomio de valor de error Ω_0 ..... Ω_{v-1} usando el algoritmo de Euclides. Las ubicaciones de error son identificadas por la búsqueda de Chien y los valores de error son calculados usando el método de Forney. Como estos cálculos involucran todos los símbolos de la palabra recibida, es necesario almacenar el mensaje hasta que los resultados de los cálculos estén disponibles. Entonces, para corregir los errores, cada valor de error es sumado en módulo 2 al símbolo en la ubicación apropiada en la palabra de código recibida.

Implementación en hardware para el calculo de síndromes

El arreglo de hardware usado para el cálculo de síndrome, mostrado en la Fig. 3.18, cuyo código es B.1.8, puede ser interpretado tanto como una división polinomial pipeline o como una implementación del método de Horner.

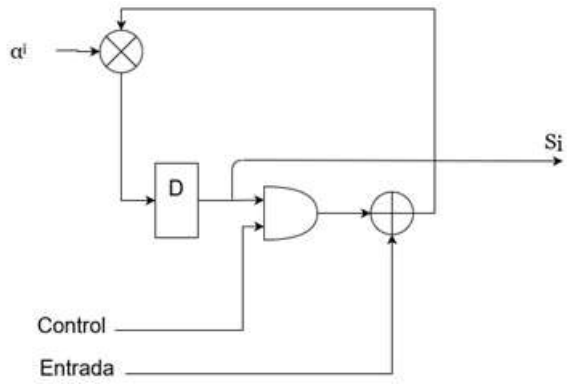


FIGURA 3.18: Hardware para calcular los síndromes

En el caso de la división polinomial, el proceso es básicamente el mismo que el descrito para codificación en la sección 2.3.4, excepto que en este caso se divide por un polinomio de grado 1. Por este motivo existe una realimentación de un término con un sólo multiplicador y sólo un registro. La única diferencia en este caso es que la compuerta AND se usa para prevenir que los contenidos del registro contribuyan al inicio de la palabra de código, lo que se logró en la Fig. 3.13 limpiando los registros al inicio de cada bloque de datos de entrada.

Alternativamente, este circuito puede verse como una implementación directa del método de Horner, en el cual el símbolo de entrada se suma a los contenidos del registro antes de ser multiplicado por a^i y el resultado realimentado al registro.

Claramente, los n símbolos de la palabra de código se deben acumular antes de que el valor del síndrome sea producido. Además, $2t$ circuitos de la forma de la Fig. 3.18, son requeridos, uno por cada valor de α^i, donde cada uno corresponde a una raíz del polinomio generador.

Implementación en hardware para el calculo del polinomio localizador de error

El algoritmo de Euclides puede implementarse utilizando el arreglo de la Fig. 3.19 en el cual todos los caminos de datos son de 4 bits. Para su implementación en FPGA se diseñó el código mostrado en B.1.9. Este arreglo presenta 2 secciones, tal como se presenta en la figura. Dichas secciones se corresponden con las operaciones del polinomio magnitud de error $\Omega(x)$, en el caso de la división, y el polinomio ubicador de error $\Lambda(x)$, en el caso de la multiplicación. El proceso se inicia cargando en los registros B el dividendo $\left(x^{2t} = x^4\right)$ y en los A los síndromes. Simultáneamente, los registros C y D se inician en 1 y 0 respectivamente.

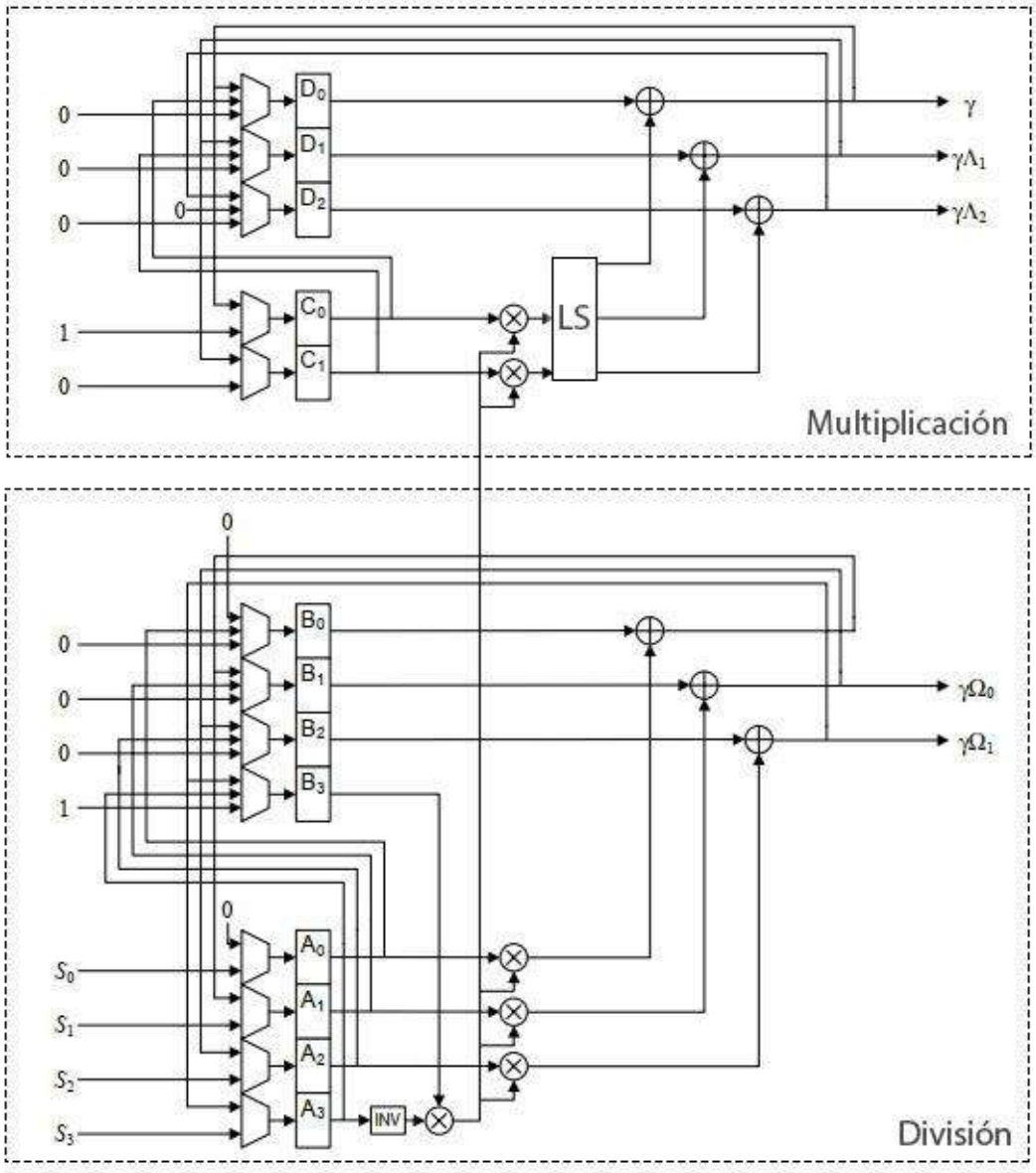


FIGURA 3.19: Procesador Euclideo

A cada paso, los datos de B_3 son multiplicados por el valor inverso de A_3 con el objetivo de realizar la división. El resultado de esta operación es utilizado en los demás multiplicadores. Luego, los resultados de estas últimas se suman con los contenidos de los registros B y D para formar los resultados intermedios. En el primer paso, los resultados de las multiplicaciones se realimentan en los registros B y D, y los contenidos de A y C son retenidos para mantener la señal estable. De otra manera se tendría un resultado erróneo debido a que estos valores cambiarían a la vez que se actualizan los registros B y D.

El paso dos se inicia con la transferencia de los datos de los registros A y C a los registros B y D respectivamente, y los resultados de las operaciones se cargan en los registros A y C.

El bloque LS, se encarga de desplazar los valores de los registros C para lograr distintos órdenes de significancia, de manera de simular una multiplicación por x.

Tomando como referencia el ejemplo de la sub-sección 2.3.6, los pasos y el estado de los registros quedarían de la manera que se muestra en la tabla 3.2.

Paso	A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0	C_1	C_0	D_2	D_1	D_0
1	12	4	3	15	1	0	0	0	0	1	0	0	0
2	12	4	3	15	14	13	12	0	0	1	0	10	0
3	6	6	4	0	12	4	3	15	10	6	0	0	1
4	6	6	4	0	8	11	15	0	10	6	7	12	1

CUADRO 3.2: Contenido de los registros para los cálculos del procesador euclídeo

Un detalle de esta implementación es que no es posible realizar una división por cero, ya que ello llevaría a error. Para ello se requiere añadir circuitería que permita sensar estas condiciones para alterar el resultado cuando se requiera.

Otro punto importante es que esta implementación está altamente paralelizada, de manera de aumentar la velocidad de procesamiento. Pero si se desea reducir el hardware en mayor medida, es posible reutilizar elementos ya que, como se ve en la Fig. 3.19, el multiplicador y el divisor son circuitos similares.

Implementación en hardware para la búsqueda de Chien

Para detallar la implementación circuital en este caso, se presenta en la Fig. 3.20 el procedimiento mencionado en la sub-sección 2.3.6. A su vez, el código donde se implementa este algoritmo es el B.1.10 El valor de cada término en el polinomio es calculado mediante la carga del valor del coeficiente Λ, el cual es multiplicado por cada potencia posible de α, $\alpha^0 = 1, \alpha^1 = 2$ *y* $\alpha^2 = 4$. Este producto es almacenado en los registros "D", realimentado para volver a multiplicar por α y luego de 15 pulsos de clock consecutivos, se logran evaluar todos los valores. Si alguno de los valores resultó en 0, el bloque de comparación "$= 0$" genera un pulso en alto determinando que se encontró un error en esa posición.

Otro detalle importante es la segunda salida del bloque, la cual es utilizada en el método de Forney para calcular las magnitudes del error, permite obtener la derivada de $\Lambda(x)$ evaluada en α^{-j} dividida por el mismo. Esto es posible ya que las raíces del polinomio generador de código inician con α^0, lo que permite utilizar la suma de los términos impares de $\Lambda(x)$ directamente.

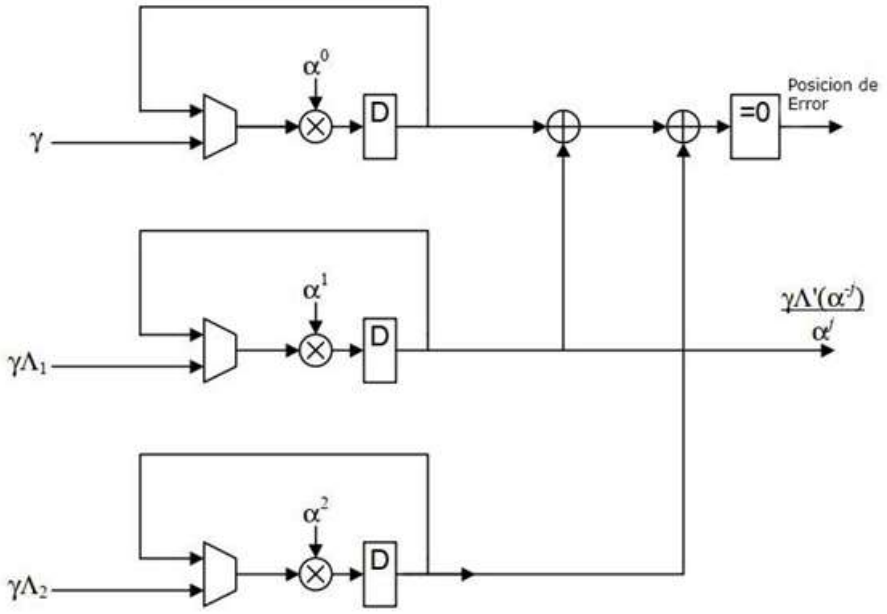


FIGURA 3.20: Implementación en hardware de la busqueda de chien

Implementación en hardware para el algoritmo de Forney

La implementación en hardware del algoritmo se presenta en la Fig. 3.21, de gran similitud con la realizada para la búsqueda de Chien. Su diseño en FPGA se muestra en B.1.11. En este caso, se ingresa con los coeficientes del polinomio evaluador de error, calculados en el ejemplo 2.3.6 $\Omega_1 = 6$ y $\Omega_0 = 15$, que luego se multiplican por $\alpha^1 = 2$ y $\alpha^0 = 1$ respectivamente.

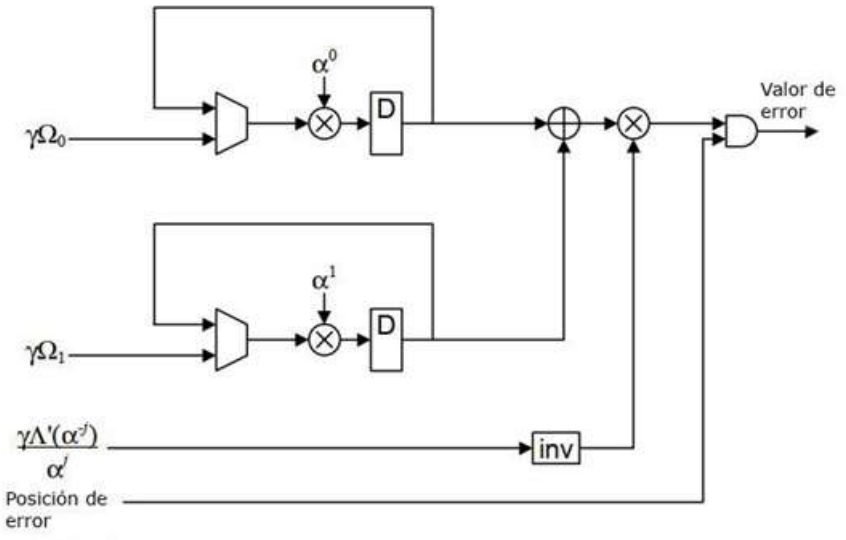


FIGURA 3.21: Implementación en hardware para calcular el valor de error - Algoritmo de Forney

Implementación en hardware para el retardo de datos

Por último, se comenta la arquitectura utilizada para obtener el retardo de datos de la Fig. 3.17. En este caso se utilizó la implementación realizada en [18] la cuál permite crear un buffer de longitud variable, con registros para anchos de palabras escalables, es decir es posible cambiar el largo y ancho de la memoria FIFO. A su vez, presenta flags para identificar si la memoria está casi llena, casi vacía, llena o vacía. Los flags de casi lleno y casi vacio pueden configurarse a gusto para que se disparen en la condición que el usuario desee. A su vez, presenta un contador de datos almacenados en la memoria y un flag de error. En este caso, se configura el módulo de manera que se puedan almacenar un máximo de 15 símbolos de 4 bits cada uno. Sabiendo esto, se muestra el código utilizado para este diseño en B.1.2.

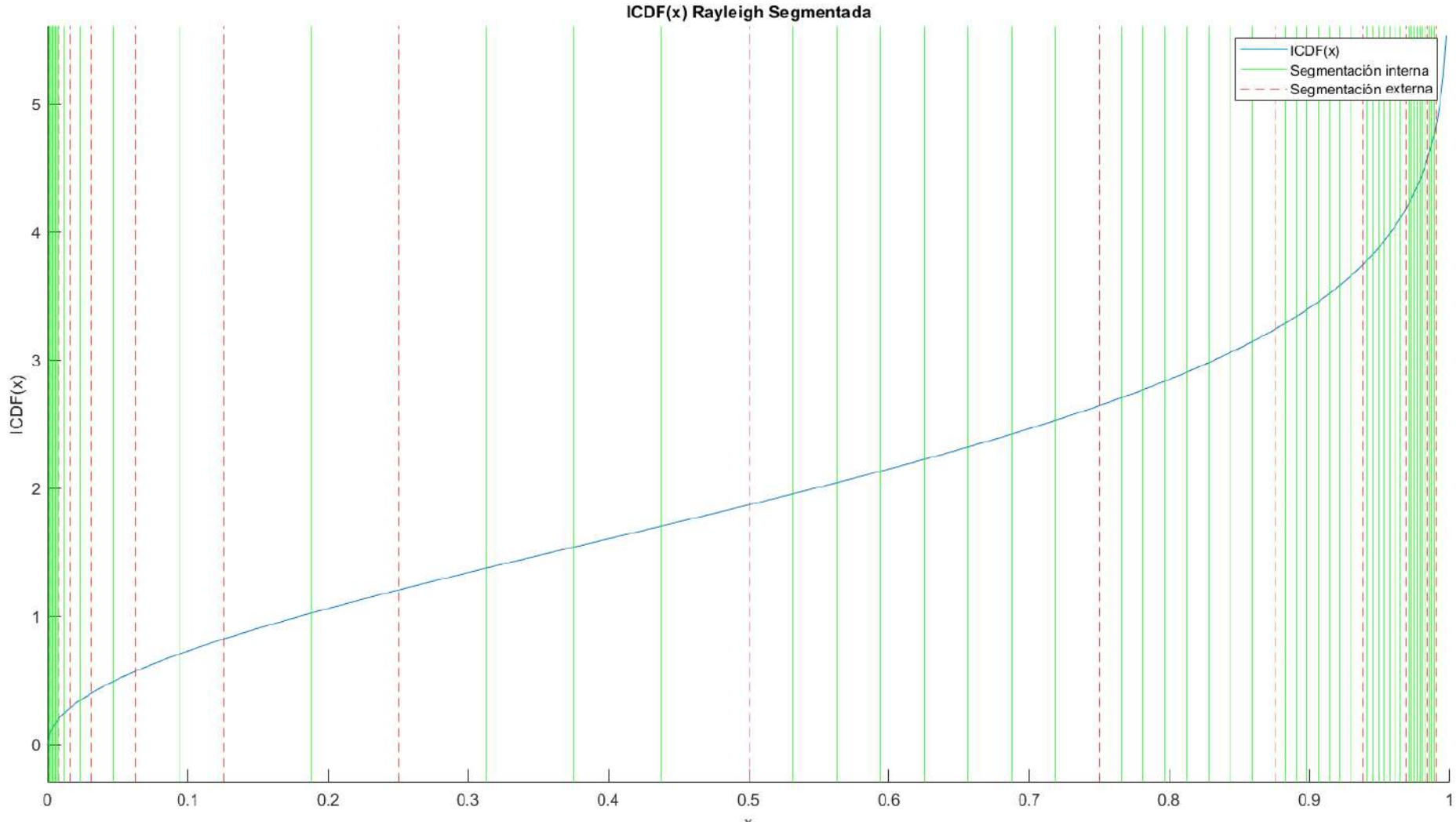


FIGURA 3.3: Segmentación de la ICDF Rayleigh para una SNR = 10dB, con splines de orden 2. En rojo segmentación externa y en negro segmentación interna

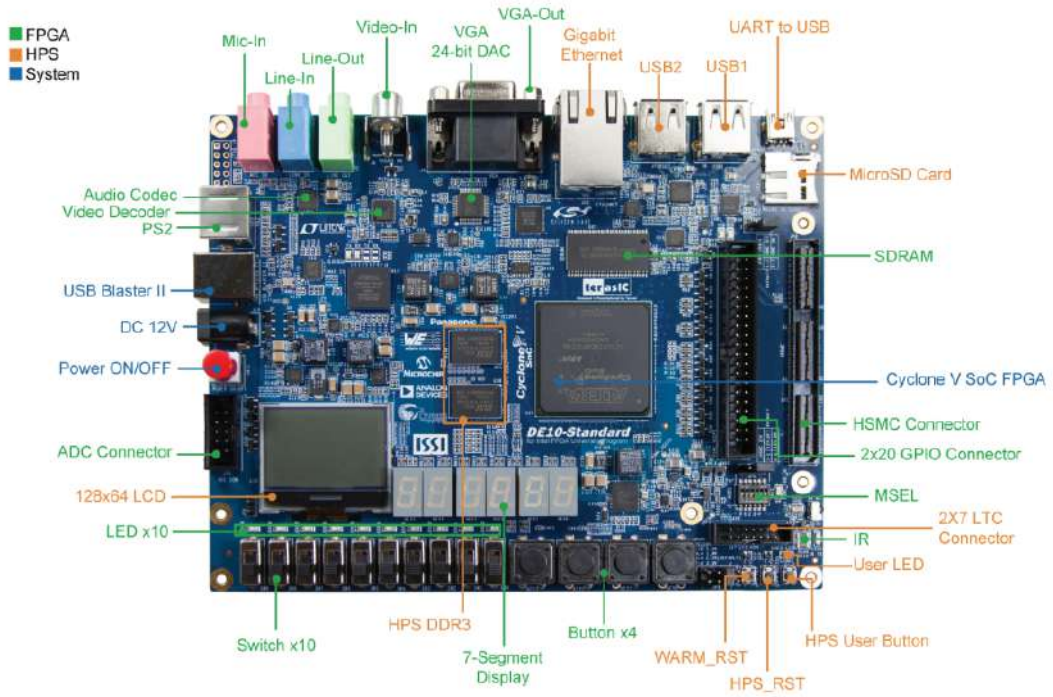


FIGURA 3.4: Placa de desarrollo Cyclone V + SoC

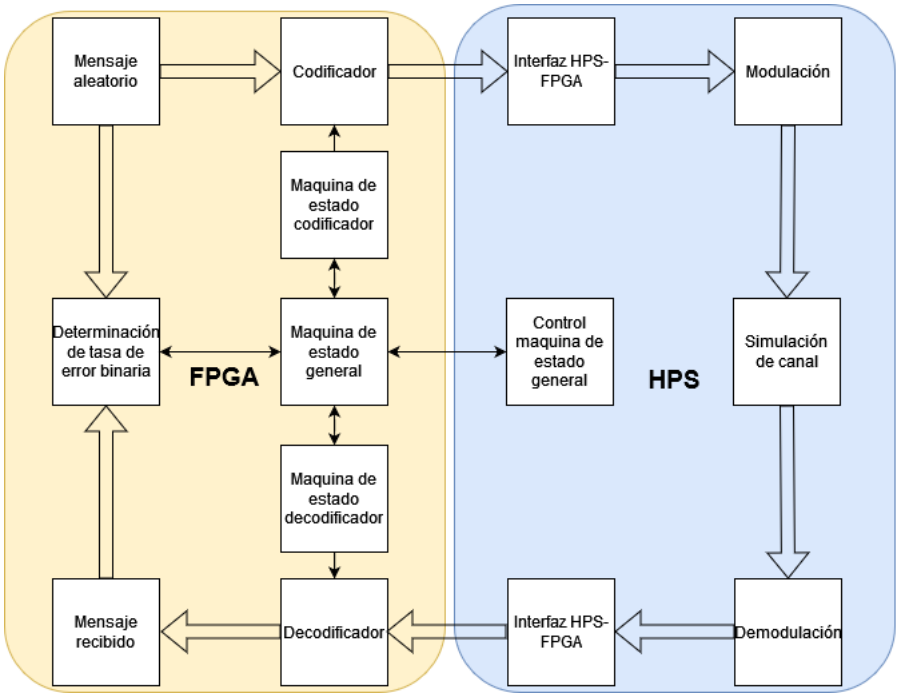


FIGURA 3.5: Arquitectura del sistema implementado

Capítulo 4

Simulaciones y Resultados

4.1. Simulación Reed-Solomon

Para verificar el funcionamiento deseado del sistema implementado en FPGA, se simuló el mismo utilizando la herramienta de diseño Vivado 2019.2 haciendo uso del lenguaje Verilog para crear los distintos módulos mencionados previamente.

Se comenzó por simular el cálculo de los mensajes codificados imitando el circuito de la Fig. 3.13. Mediante el uso de la herramienta Simulation es posible simular el comportamiento del circuito bajo ciertas condiciones impuestas por el archivo tb_TEST_BENCH_FSM, el cual es un banco de pruebas para analizar el correcto funcionamiento de las máquinas de estado, el codificador y el decodificador.

En primera instancia, se muestra una vista general de la simulación con el objetivo de mostrar el secuenciamiento y el sincronismo. En la Fig. **??** se puede apreciar que una vez que se levanta el reset y se da la señal de start, luego 4 ciclos de reloj, se inicia el contador principal, lo que permite que inicie el proceso. El hecho de que el sistema tarde 4 ciclos de reloj en iniciar se debe a que existe una latencia en la transferencia de datos aparejada con los registros, al cambiar de estado, se tardan 2 ciclos de reloj y luego la habilitación del conteo lleva 1 ciclo de reloj, más otro ciclo que se tarda en cargar el registro.

A su vez, se marca entre líneas amarillas la duración del proceso de codificación, el cuál se muestra en detalle en la Fig. **??**. Aquí, una vez habilitado "Encoder_start" se tardan otros 4 ciclos de reloj en iniciar la cuenta. Una vez iniciada, se habilita en simultáneo el control que permite la realimentación de los datos de entrada hacia los multiplicadores en el circuito de la Fig. 3.13.

En la Fig. **??** también se puede apreciar que ingresan 11 símbolos en la señal "Enabled_dataIn", las cuales son las mismas que se pueden ver en "Encoder_o_data" con la adición de los 4 símbolos de paridad al final. Si bien el último símbolo de paridad se mantiene en su valor, sólo se toma este valor 1 sola vez, ya que al siguiente se puede apreciar que se envía el ACK en "Encoder_ack", por lo que la máquina de estados del codificador se encuentra fuera del estado de codificación, debido a que el contador superó el valor 16 como se muestra en la Fig. 3.10. A su vez, en la sección de "Fifo_codificador" se muestra la carga de los datos de salida del codificador, en la memoria FIFO. En este caso se habilita el flag "Fifo_codificador_write_enable" lo que permite la escritura de la memoria. Por último, se muestran el estado de la memoria en "Fifo_codificador_memory" donde se pueden ver cargados los 15 símbolos a transmitir hacia el procesador y el flag que indica que la memoria ya no está vacía en "Fifo_codificador_empty_flag".

A continuación, en la Fig. **??** se muestra la carga de datos en la memoria RAM del decodificador. Este proceso se realiza antes de habilitar "Decoder_start" de manera de asegurar la correcta carga de datos y el proceso se lleve a cabo ordenadamente. Además, a la hora de decodificar, es necesario tener el puntero de la memoria

apuntando a la primer posición con datos, para evitar saltar o perder información. Además se destaca la introducción de 2 errores (en el primer y último símbolo) para verificar la capacidad de corrección máxima del decodificador. Todo este proceso se lleva a cabo en el estado "s_NOISE" de la Fig. fig:FSM-General

Por último, una vez iniciado el decodificador y procesados todos los símbolos como en 4.1.1 se procede a mostrar en la Fig. **??** el estado de las señales "Datos sin corregir", "Datos corregidos" y "Valor_de_correccion" donde se ve claramente que los dos errores se corrigen satisfactoriamente. Luego, los datos corregidos se cargan en el FIFO del decodificador, como muestra la señal "Fifo_decodificador_data_in". Una vez finalizada la carga, se envía el "Decoder_ACK" y a continuación se habilita la lectura de ambos FIFO's, los cuales son los datos de entrada en el contador de bits erróneos. Aquí "data_in" se corresponde con la salida del FIFO del codificador y "data_coded" con la del FIFO del decodificador. En este caso, como el error fue corregido "BER_counter" se mantiene en 0. Por último, se procede a aumentar la cantidad de palabras decodificadas en 1, como se puede ver en la señal "Decoded_words" y se resetean los registros del codificador, y decodificador para continuar con la siguiente palabra. Este proceso continua hasta que se decodifican la cantidad de palabras preestablecida.

4.1.1. Simulación en detalle de codificador y decodificador

En esta sección se presentan simulaciones del funcionamiento del codificador y decodificador para mostrar en detalle como funcionan ambos procesos. Esto se realiza, al igual que la sección anterior, para un caso particular utilizando un archivo de pruebas llamado "tb_Encoder_State_Machine" para el codificador y "tb_Decoder_State_Machine" en el caso del decodificador.

En la Fig. 4.5 se puede apreciar que la simulación se corresponde con el comportamiento buscado para el codificador, ya que se logran los datos de entrada y a continuación de los mismos, los símbolos de paridad requeridos para la corrección de los mensajes erróneos.

Cabe destacar que el símbolo 11 ocupa el doble de ciclos de reloj que los anteriores. Esto se debe a que al cambiar el selector, se debe esperar un ciclo de reloj adicional para que el cálculo de los símbolos de paridad estén disponibles, esto se debe al retardo de los registros D del decodificador. Si el cambio fuera instantáneo, los bits que conforman la paridad serían erróneos ya que su cálculo se realizaría sin considerar el último dato de entrada.

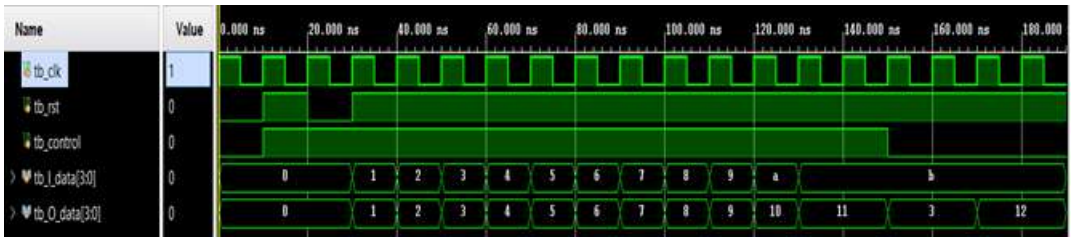


FIGURA 4.5: Formas de onda para codificador Reed Solomon.

Luego de la simulación del bloque codificador, se procedió a simular los distintos bloques que conforman el decodificador, presentados en la Fig. 3.17.

El circuito utilizado para el calculo de síndromes se replica 4 veces para lograr los 4 valores requeridos por las raíces del polinomio generador de código Ec.2.24.

En la Fig. 4.6 se muestra que, luego de que los cálculos de los síndromes finalizan, se transfieren los datos a los registros syndrom1_reg, syndrom2_reg, syndrom3_reg y syndrom4_reg para mantenerlos estables en el bloque siguiente.

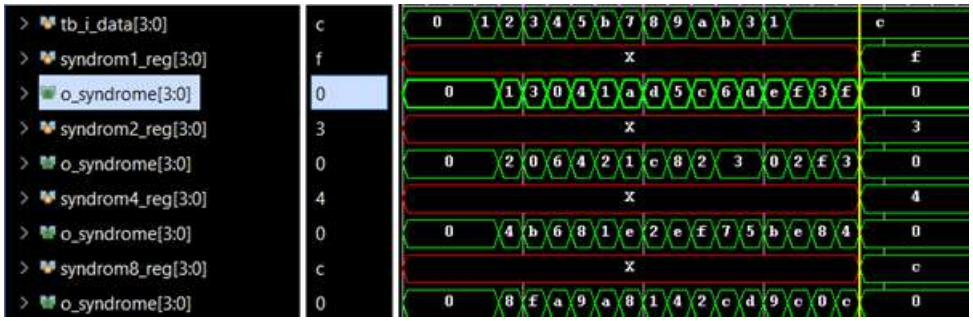


FIGURA 4.6: Simulación de cálculo de síndromes.

El bloque que procesa el algoritmo de Euclides es el más complejo y completo en lo que respecta a código y funciones.

Como se puede apreciar en la Fig. 4.7, los registros A, B, C y D toman los valores mostrados en la Tabla 3.2. Para lograr esto se debe hacer una correcta elección de los selectores de los multiplexores a la entrada de los registros de la Fig. 3.19. Para mejor explicación se presentan las Tablas 4.1 y 4.2 en las que se muestran los valores de los selectores y el estado de los registros.

	Control A/C	
Estado	0	1
Registro A	$A_0 = 0$ $A_1 = B_0 + \left(A_0 * \frac{B_3}{A_3}\right)$ $A_2 = B_1 + \left(A_1 * \frac{B_3}{A_3}\right)$ $A_3 = B_2 + \left(A_2 * \frac{B_3}{A_3}\right)$	$A_0 = S_0$ $A_1 = S_1$ $A_2 = S_2$ $A_3 = S_3$
Registro C	$C_0 = D_0 + LS_0$ $C_1 = D_1 + LS_1$	$C_0 = 1$ $C_1 = 0$

CUADRO 4.1: Estados de registros A y C.

	Control B/D			
Estado	00	01	10	11
Registro B	$B_0 = 0$ $B_1 = B_0 + \left(A_0 * \frac{B_3}{A_3}\right)$ $B_2 = B_1 + \left(A_1 * \frac{B_3}{A_3}\right)$ $B_3 = B_2 + \left(A_2 * \frac{B_3}{A_3}\right)$	$B_0 = A_0$ $B_1 = A_1$ $B_2 = A_2$ $B_3 = A_3$	$B_0 = 0$ $B_1 = 0$ $B_2 = 0$ $B_3 = 1$	$B_0 = B_0$ $B_1 = B_1$ $B_2 = B_2$ $B_3 = B_3$
Registro D	$D_0 = D_0 + LS_0$ $D_1 = D_1 + LS_1$ $D_2 = D_2 + LS_2$	$D_0 = C_0$ $D_1 = C_1$	$D_0 = 0$ $D_1 = 0$ $D_2 = 0$	$D_0 = D_0$ $D_1 = D_1$ $D_2 = D_2$

CUADRO 4.2: Estados de registros B y D.

Para el caso del registro C se utiliza un desplazador que permite sumar C_0 y C_1 a los 3 registros D en distintos instantes de tiempo. Es decir, permite lograr que (C_0,C_1) $\rightarrow$ (LS_0,LS_1) ó (C_0,C_1) $\rightarrow$ (LS_1,LS_2) quedando los cables no utilizados en "0".

Luego de que los registros se establecen en los valores finales, se llega a los resultados esperados en la salida del divisor y multiplicador, con $\Lambda(x) = 7x^2 + 7x + 9$ y $\Omega(x) = 3x + 14$ respectivamente.

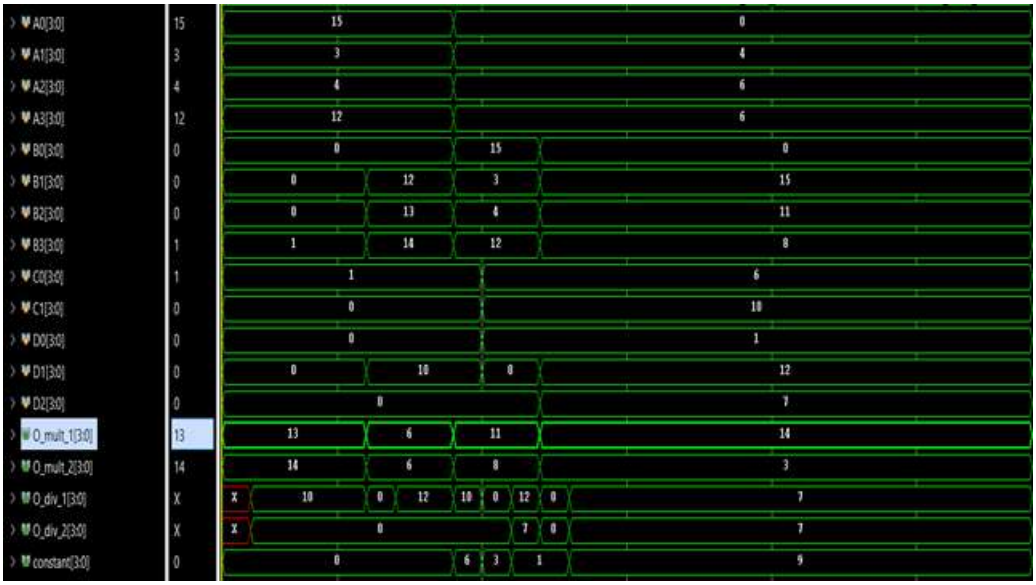


FIGURA 4.7: Registros y salidas del algoritmo Euclidiano.

Luego del cálculo de los coeficientes de Λ y Ω es posible identificar la posición del error, y calcular el valor del error generado para este caso. En la Fig. 4.8 se muestran las salidas del bloque de Chien en O_errorpos y de Forney en O_errorvalue. En ambos casos se sincroniza con la salida del sistema O_data, la cual ya presenta el mensaje corregido, para una mejor visualización de la suma que se realiza a la salida

del sistema entre O_errorvalue y sr_DelayedData. Este último se corresponde con la salida del bloque "Retardo de datos" de la Fig. 3.17 que se explicará a continuación.

Haciendo énfasis en los 2 errores introducidos, se ve claramente que se obtienen los símbolos enviados en el mensaje, ya que cuando un error es identificado por el circuito de la Fig. 3.20, la entrada "posición de error" habilita la salida y el valor de error se suma al símbolo del mensaje de entrada en la posición en la cual se presenta el mismo.

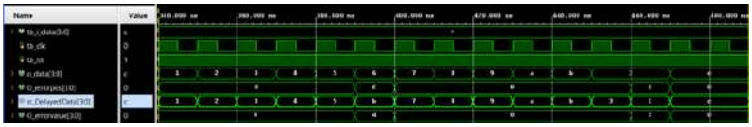


FIGURA 4.8: Salidas de los algoritmos de Chien y Forney.

En este caso, se busca disponer de una copia de los datos de entrada con un retardo controlado, de manera que permita corregir el mensaje al sumarle los valores de error que se obtienen del bloque de Forney. Para ello se implementa un registro desplazamiento que actúa como memoria. En primera instancia, un registro de 60 bits se carga con los datos de entrada y los almacena hasta que el resto de bloques terminen sus operaciones. Una vez que el bloque del algoritmo de Euclides dispone de una salida estable, se habilita la lectura y otro registro de 4 bits junto con un contador recorren el de 60 bits y permiten disponer del mensaje de entrada sincronizado con los valores de corrección. La salida sr_DelayedData de la Fig. 4.8 se corresponde con este registro de 4 bits.

La Tabla 4.3 muestra el funcionamiento de los bits de control que permiten leer y escribir esta pseudo memoria.

Acción/Bit	00	01	10	11
Reseteo de registros	X			X
Escritura		X		
Lectura			X	

CUADRO 4.3: Tabla de control para registro de retardo de datos.

4.2. Resultados experimentales

4.2.1. Determinación de tasa de error

Habiendo diseñado e implementado el sistema de pruebas, se realizaron evaluaciones de performance en cuanto a la capacidad de corrección de errores del conjunto codificador-decodificador. Para ello, se calculó la tasa de error binaria (BER) usando el método Monte Carlo. Se generaron, codificaron y decodificaron millones de mensajes para estimar estadísticamente probabilidades de encontrar errores de corrección. Las pruebas se realizaron con dos objetivos principales: verificar funcionalidades del sistema y efectuar comparaciones con resultados de referencia. En las pruebas la FPGA se encargó de codificar, decodificar los mensajes de prueba y determinar cuales bits resultaron erróneos en el mensaje recibido. El programa corriendo en el HPS se ocupó de simular e implementar los demás elementos del sistema de comunicaciones.

Para obtener los parámetros de referencia, se consideró el desarrollo de Viswanathan en [19]. Aquí define la performance de una modulación BPSK sobre un canal de Rayleigh. Para ello se utilizan las siguientes expresiones:

$$E_s = R_m R_c E_b \tag{4.1}$$

$$N_0 = \frac{(4-\Pi)\sigma^2}{2} \tag{4.2}$$

$$SNR = \frac{E_b}{N_0} \tag{4.3}$$

$$P_b = \frac{1}{2}(1 - \sqrt{\frac{\frac{E_b}{N_0}}{1+\frac{E_b}{N_0}}}) \tag{4.4}$$

Siendo E_s la energía de símbolo por bit modulado para sistemas M-arios de modulación, $R_m = \log_2(M)$, $R_c = k/N$ la tasa de código del sistema, N_0 la densidad espectral de potencia de ruido para una distribución Rayleigh, SNR la relación señal a ruido y P_b la BER teórica para un sistema de modulación BPSK sobre un canal de Rayleigh. En este caso, se asumieron los siguientes valores:

- $E_s = 1$
- $R_c = \frac{11}{15}$
- $R_m = 1$

Habiendo definido las ecuaciones y parámetros generales, se procedió a determinar los valores de dispersión espectral σ asociados a relaciones señal a ruido de 1 a 10 dB. Así, combinando las ecuaciones 4.1 4.2 4.3 se obtuvo:

$$\sigma = \frac{1}{\sqrt{\frac{11}{15}(\frac{4-\Pi}{2})10^{\frac{SNR[dB]}{10}}}} \tag{4.5}$$

Por lo que, para los valores mencionados previamente de relación señal a ruido, se obtuvieron los siguientes valores de σ

- $SNR = 1dB \rightarrow \sigma_1 \approx 1{,}5886$
- $SNR = 2dB \rightarrow \sigma_2 \approx 1{,}4158$
- $SNR = 3dB \rightarrow \sigma_3 \approx 1{,}2618$
- $SNR = 4dB \rightarrow \sigma_4 \approx 1{,}1246$
- $SNR = 5dB \rightarrow \sigma_5 \approx 1{,}0023$
- $SNR = 6dB \rightarrow \sigma_6 \approx 0{,}8933$
- $SNR = 7dB \rightarrow \sigma_7 \approx 0{,}7961$
- $SNR = 8dB \rightarrow \sigma_8 \approx 0{,}7096$
- $SNR = 9dB \rightarrow \sigma_9 \approx 0{,}6324$
- $SNR = 10dB \rightarrow \sigma_{10} \approx 0{,}5636$

Luego, estos valores de sigma se reemplazaron en la ecuación 2.4 y utilizando el segmentador implementado en MATLAB se determinaron los coeficientes para simular el canal para las distintas relaciones señal a ruido.

A continuación, se cargaron los coeficientes en la memoria RAM del programa implementado en el HPS, y para cada SNR se codificaron-decodificaron 1 millón de palabras.

Para chequear el correcto funcionamiento del sistema, se utilizó la herramienta "Signal Tap Logic Analyzer" de Quartus como se detalla en [20]. Esta herramienta permite implementar un analizador de señales digital, que permite ver las señales sintetizadas en la FPGA de manera continua. A modo de ejemplo, se muestra en la Fig.**??** las señales obtenidas mediante signal tap para una SNR = 6dB.

Finalmente, con el valor de los bits erróneos obtenidos mediante signal tap y la cantidad de palabras decodificadas, que para todos los casos resultó de 1000001 palabras, se procedió a calcular la tasa de error binaria P_b para compararla con la teórica obtenida mediante la Ec. 4.4. Estos resultados se muestran en la Fig. 4.10

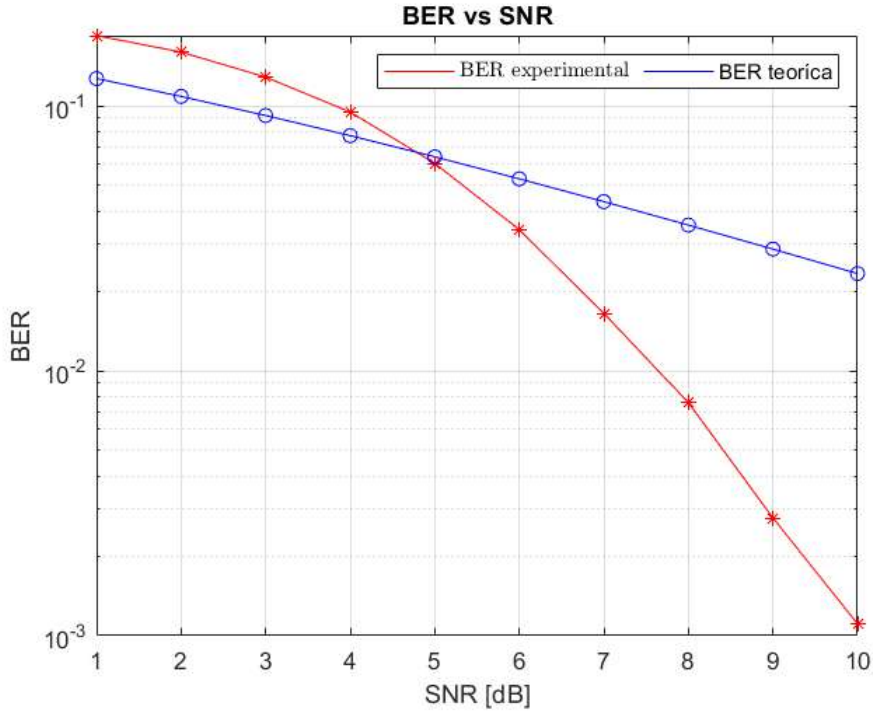


FIGURA 4.10: Comparación de tasas binarias de error para distintos valores de relación señal a ruido.

Como se puede ver, la curva experimental presenta una mejoría de performance luego de 5dB lo que era de esperarse, ya que para esta relación señal a ruido el mensaje no se encuentra tan contaminado con ruido, de manera que el decodificador es capaz de corregir t=2 símbolos erróneos. Otro punto a destacar es que el procesamiento del millón de palabras llevo aproximadamente unos 40 minutos, es decir para obtener el gráfico de la Fig. 4.10 se tuvieron que esperar 400 minutos. Si bien, para una primera implementación es más que aceptable, es posible mejorar este diseño transportando la parte del HPS a hardware, la cual era la idea en primera instancia, pero por cuestiones de falta de conocimiento no se logró llegar a cabo.

4.2.2. Utilización de la FPGA

Por otro lado el sistema de Quartus permite obtener una estimación de utilización del dispositivo. En este caso se muestra en la Tabla 4.4 la cantidad de componentes empleados en esta implementación.

	ALMs	Pines I/O	Total DLLs	Fmax (MHz)
Utilizados	3344	271	1	50
Total disponible	41910	499	4	50
Porcentaje de uso (%)	8	54	25	100

CUADRO 4.4: Porcentaje de utilización del sistema.

En la Tabla 4.4 puede verse que el porcentaje de utilización de elementos lógicos ó módulos de lógica adaptiva (ALM) es prácticamente nulo. Esto se debe a que el diseño del codificador-decodificador, con sus multiplicadores y máquinas de estado, son diseños pequeños, y personalizados. La implementación manual permite reducir en gran medida la cantidad de elementos utilizados por el sistema. Por otro lado, se utilizaron un 54 % de los pines de entrada/salida (Input/Output) los cuales están vinculados con el uso del HPS, ya que son necesarios para realizar la comunicación entre el procesador y la FPGA. Por último, se utilizó un DLL (Delayed Locked Loop) para generar el reloj del sistema. Para profundizar un poco más en la utilización de la placa se muestra en la Fig. 4.11 la distribución de los módulos utilizados en el proyecto.

Aquí se aprecian las dimensiones del codificador-decodificador comprendidas por los bloques rojos, específicamente el modulo **Test_Bench_FSM : Test_bench_u**. A simple vista se puede ver que comprenden la menor sección ocupada en la FPGA. A su vez, se muestra en los bloques de color magenta la sección necesaria para la implementación de los bloques requeridos por el procesador de la FPGA, el cual se encuentra comprendido en la sección donde se encuentra la leyenda **ARM A9 Subsystem**, siendo el nombre del módulo **HPS : u0**. Estos bloques se encuentran cercanos a las memorias de la placa ya que se esto acelera la comunicación con el procesador. Por último, en violeta se puede ver el módulo utilizado para recopilar datos y realizar las pruebas, el cual es una herramienta brindada por quartus para implementar un sistema de adquisición de datos digital, en el gráfico se encuentra como **sld_signaltap : auto_signaltap_0**. Se puede ver que este bloque abarca gran medida del dispositivo debido a que se necesitaban adquirir 1 millón de datos por muestra, por lo que para ver detalladamente los datos recopilados se necesitan grandes memorias y buffers.

De esta manera, se verifica que el funcionamiento del sistema fue satisfactorio y que es posible implementar codificadores-decodificadores de mayor tamaño.

4.2.3. Bloques en vista de compuertas

Otra herramienta interesante que presenta el software de diseño en FPGA, es que permite ver mediante que compuertas especificamente fue implementado cada bloque. En la Fig. 4.12 se muestra el módulo referido al banco de pruebas, es decir al codificador/decodificador junto con sus maquinas de estado y resto de componentes necesarios para el correcto funcionamiento del mismo.

Como se mencionó en el párrafo anterior, es posible apreciar los bloques que componen este modulo, los cuales en su mayoría son multiplexores, también se pueden ver el contador de palabras codificadas, sumadores, las memorias FIFO, el LFSR

de 32 bits, los contadores binarios utilizados por las máquinas de estado y el contador de BER. Estos últimos 4, junto con las maquinas de estado se aprecian como un bloque, pero en el detalle son circuitos más complejos como se explicó previamente en el capítulo 3.

A su vez, es posible ver los bloques que forman parte del módulo que conforman al procesador. Estos se muestran en la Fig. 4.13.

En este caso, a la entrada se puede ver el bloque mencionado anteriormente, esto es así debido a que la comunicación para el control del programa se hace directamente al procesador y se necesita que se encuentre en el "Top level.º el máximo nivel de encapsulamiento para así disponer de un fácil acceso al mismo y reducir los retardos de las señales. Luego, dentro de un gran recuadro se aprecia la implementación realizada en quartus mediante la herramienta "Qsys"que permite el diseño personalizado de un procesador y los PIO (puertos de entrada/salida) que serán utilizados por el mismo.

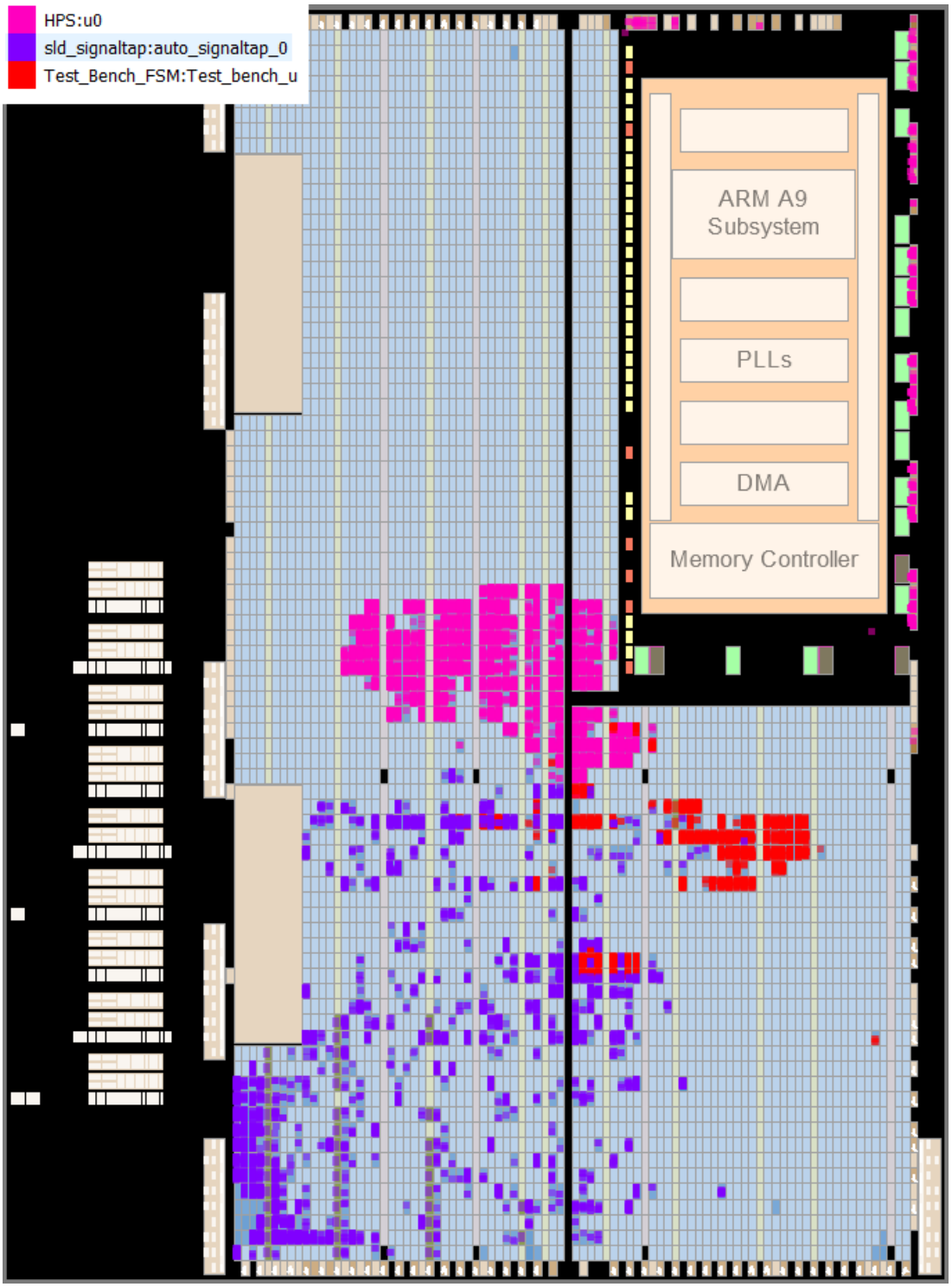


FIGURA 4.11: Utilización del chip donde se puede ver la sección ocupada por cada modulo.

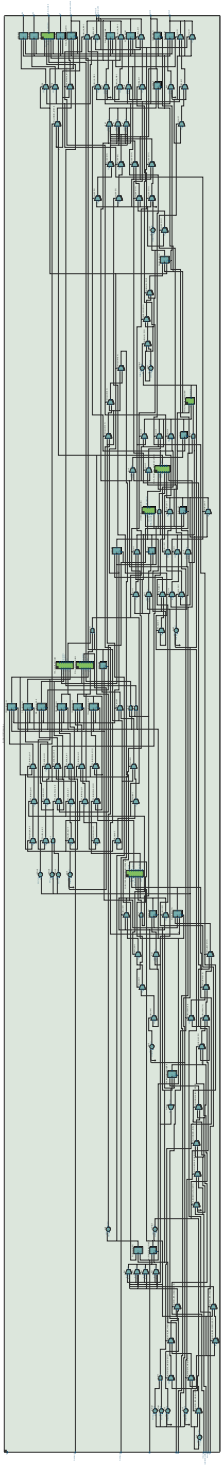


FIGURA 4.12: Vista RTL del hadware diseñado en FPGA para el banco de pruebas.

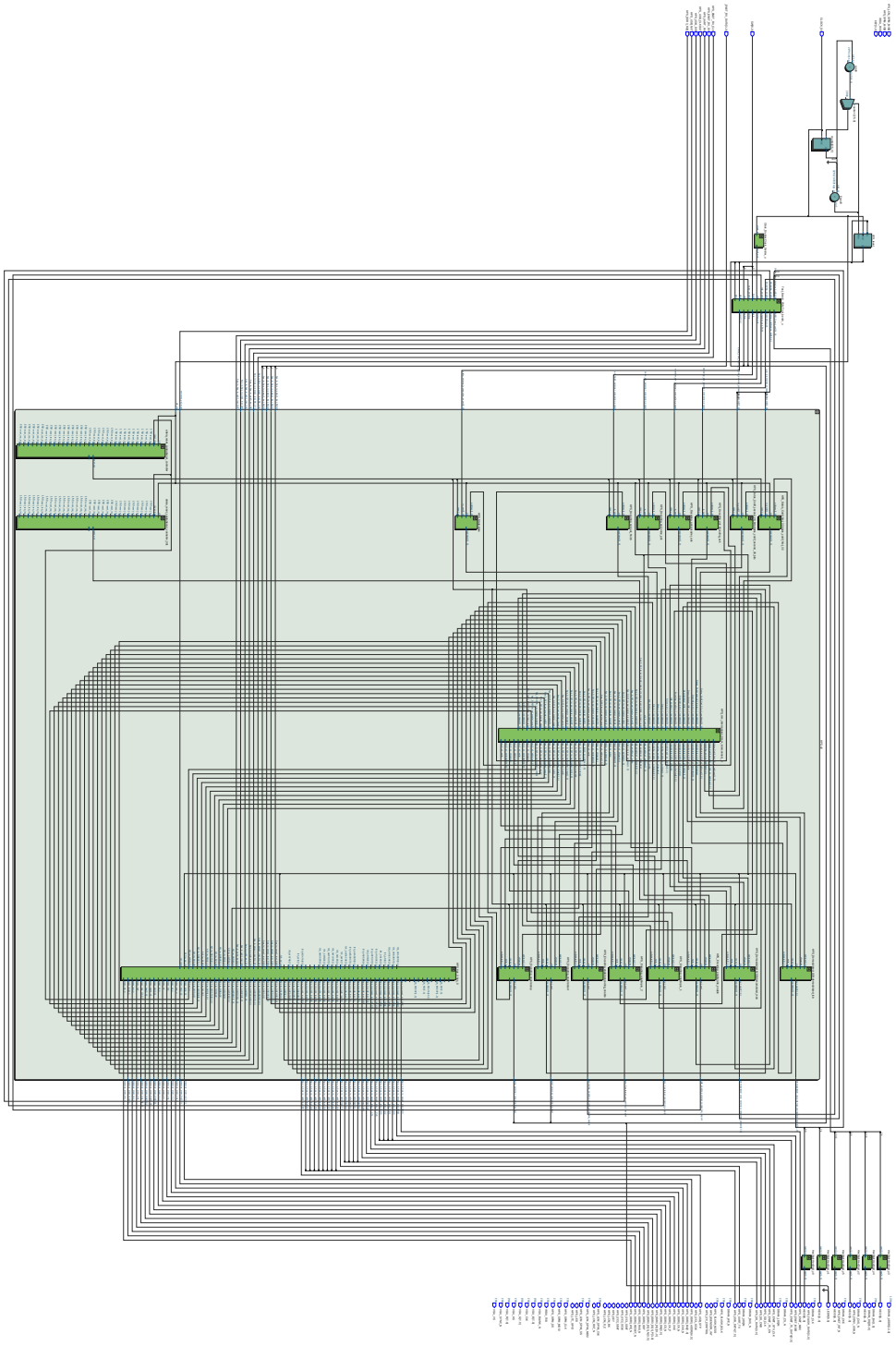


FIGURA 4.13: Vista RTL del hadware diseñado en FPGA para el procesador.

Capítulo 5

Conclusiones y consideraciones a futuro

En este trabajo se diseñó e implementó en una placa de desarrollo FPGA un banco de pruebas para sistemas de decodificación y corrección de errores. Se buscó en principio un diseño que permita su funcionamiento a una velocidad mayor que la que una computadora es capaz de procesar y desde el punto de vista de la FPGA, se logró ese objetivo. Por otro lado, la implementación en el HPS fue más lenta de lo esperado, si bien resultó aceptable, para el calibre de estos dispositivos, se esperaba lograr velocidades mayores. Esto generó un cuello de botella en la integración final del sistema, ya que se debía esperar que el procesador termine sus tareas para poder pasar a la etapa de decodificación. Por esta razón, se plantea modificar la implementación del HPS y adaptarla totalmente a hardware como se muestra en el trabajo de Guanxi Liu [21]. En este caso, la mayor diferencia entre el diseño de este proyecto y el de Liu, radica en que la implementación de Liu es particular para la generación de ruido gaussiano y el bloque que permite identificar los ceros, y unos a la izquierda no fue diseñado de forma genérica, sino que se aprovecho la simetría de la ICDF gaussiana para reducir este bloque a un enmascaramiento de bits. Pero se podría crear un detector de ceros y unos a la izquierda igual que la implementación de software para poder trabajar con otras funciones inversas acumulativas. En este caso se diseño en software como se mencionó previamente, lo que lo hace más lento que la implementación de Liu, pero más versatil por el hecho de detectar tanto ceros como unos a la izquierda, de esta manera se contemplan todas las funciones posibles. Por lo que el sistema presenta mayor versatilidad respecto a la cantidad de funciones que puede procesar, pero con un rendimiento reducido.

A su vez, la implementación del segmentador en MATLAB puede reutilizarse para nuevos diseños ya que es independiente del sistema creado. Como así también migrarse a otro lenguaje y/o entorno con el objetivo de ampliar su capacidad y velocidad. Otro punto importante respecto al segmentador es que sólo permite una segmentación interna uniforme, lo que sería interesante a futuro es agregar la posibilidad de permitir combinaciones de segmentaciones diferentes, de manera que se pueda adaptar de mejor manera la segmentación a la función trabajada.

Con respecto al par de codificador-decodificador utilizado, el diseño seguido en esta implementación fue el más pequeño posible en términos de capacidad de corrección y cantidad de bits codificados, en comparación con otros análisis como [22], [23] los cuales implementan en el caso de Kumar se estudiaron códigos RS(255, 245) a RS(400,240) y en el de Wai se utiliza un RS(255, 239) que comparado con el desarrollado en este caso (RS(15,11)) presentan mayor tamaño. Esto se debió a que en principio sólo se buscaba verificar el correcto funcionamiento del resto de los componentes, pero es posible ampliar este diseño para permitir códigos Reed-Solomon de mayor tamaño, para esto deberían reutilizarse/replicarse módulos básicos como

los multiplicadores y los FIFO, pero se deberían rediseñar/modificar los controles y maquinas de estado. También se pueden utilizar los bloques genéricos como el LFSR o el bloque que permite determinar la cantidad de bits erróneos para otros tipos de codificación que no sean Reed-Solomon.

La implementación en el HPS permite dar gran versatilidad a los diseños de FPGA ya que habilita a operar con elementos que en hardware lleva tiempo implementar, así como también resulta más complejo hacerlo. Si bien, como se mencionó previamente, es preferible implementar el interpolador en hardware, resultaría interesante crear una interfaz de control mediante el procesador adaptable, de manera que permita seleccionar distintos tipos de pares de codificadores-decodificadores implementados en la FPGA. Así como también, los distintos tipos de canales posibles a utilizar y la selección de varios modelos de modulación.

Por último, se podría reutilizar las entidades sintetizadas en este proyecto en otras arquitecturas. Como así también, ensayar con éstas diseños sincrónicos que las activaran repetidamente durante el procesamiento de cada mensaje, requiriendo en consecuencia menos recursos. Si se quisiera utilizar mayor cantidad de área y no minimizar como esta implementación, se podrían buscar mayores valores de capacidad de procesamiento implementando pipelines. Otra posibilidad podría ser aprovechar algunos módulos programados para experimentar con alguna implementación híbrida.

Apéndice A

Apendice A

A.1. Programa para calculo de segmentos balanceados

```
%Inputs: a,b,d,f,e_max,ulp
%Output: u()
function [P_end,x1,x2,u,errors] = Boundaries(fun,a,b,d,ulp,reg_error)
    %Con esto calculo los segmentos que tengan el mismo error o menor que
    %el reg_error (los segmentos externos). u tienen los intervalos (x1,x2)
    %de segmentos con errores balanceados
    x1=a; x2 = b; m = 1; done = 0; check_x2 = 0; prev_x2 = a;
    oscillating = 0; interval = [a b]; i = 1; errors = []; P_end =[];
    while (~done)
        %A=remez(fun, fun_der,interval,order)
        [P,error]= NewRemez(fun, d, x1, x2);
       % error = abs(A(end));
        if (error <= reg_error)
            if (x2 ==b)
                u(m) = x2;
                done = 1;
            else
                if (oscillating)
                    errors(m) = error;
                    u(m) = x2;
                    P_end(m,:) = P;
                    prev_x2 = x2;
                    x1 = x2;
                    x2 = b;
                    m = m + 1;
                    oscillating = 0;
                else
                    change_x2 = abs(x2-prev_x2)/2;
                    prev_x2 = x2;
                    if(change_x2 > ulp)
                        x2 = x2 + change_x2;
                    else
                        x2 = x2 + ulp;
                    end
                end
            end
        else
            change_x2 = abs(x2 - prev_x2)/2;
            prev_x2 = x2;
            if(change_x2 > ulp)
                x2 = x2 - change_x2;
               % interval = [x1 x2];
```

```
%                   a = x1;
%                   b= x2;
              else
                  x2 = x2 - ulp;
                 % interval = [x1 x2];
%                   a = x1;
%                   b= x2;
                  if (check_x2 == x2)
                      oscillating = 1;
                  else
                      check_x2 = x2;
                  end
              end
          end
      end
end
```

A.2. Generador Tausworthe de 32 bits

```
/* Maximally equidistributed combined Tausworthe generator */

/*
 * Copyright (C) 2014, Guangxi Liu <guangxi.liu@opencores.org>
 *
 * This source file may be used and distributed without restriction provided
 * that this copyright statement is not removed from the file and that any
 * derivative work contains the original copyright notice and the associated
 * disclaimer.
 *
 * This source file is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License,
 * or (at your option) any later version.
 *
 * This source is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this source; if not, download it from
 * http://www.opencores.org/lgpl.shtml
 */


#include "Include/taus176.hpp"
/* Set state using seed */
/* Update state */
unsigned long long taus176::taus_get(taus_state_t *state_1)
{
    b= (((state_1->z1 << 5) ^ state_1->z1) >> 39);
    state_1->z1 = (((state_1->z1 & 18446744073709551614ULL) << 24) ^ b);
    b = (((state_1->z2 << 19) ^ state_1->z2) >> 45);
    state_1->z2 = (((state_1->z2 & 18446744073709551552ULL) << 13) ^ b);
    b = (((state_1->z3 << 24) ^ state_1->z3) >> 48);
    state_1->z3 = (((state_1->z3 & 18446744073709551104ULL) << 7) ^ b);

```

```
39      return (state_1->z1 ^ state_1->z2 ^ state_1->z3);
40  };
41  void taus176::taus_set(taus_state_t *state_1, unsigned long s)
42  {
43      if (s == 0)     s = 1;     /* default seed is 1 */
44
45      state_1->z1 = LCG(s);
46      if (state_1->z1 < 2ULL)     state_1->z1 += 2ULL;
47      state_1->z2 = LCG(state_1->z1);
48      if (state_1->z2 < 64ULL)     state_1->z2 += 64ULL;
49      state_1->z3 = LCG(state_1->z2);
50      if (state_1->z3 < 512ULL)     state_1->z3 += 512ULL;
51
52      /* "warm it up" */
53      taus_get(state_1);
54      taus_get(state_1);
55      taus_get(state_1);
56      taus_get(state_1);
57      taus_get(state_1);
58      taus_get(state_1);
59      taus_get(state_1);
60      taus_get(state_1);
61      taus_get(state_1);
62      taus_get(state_1);
63  };
```

Apéndice B

Apendice B

B.1. Código Verilog

B.1.1. Codificador Reed Solomon - LFSR 32 Bits máximo

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 17.09.2021 11:57:18
7  // Design Name:
8  // Module Name: LFSR32BitsMax
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////////////////////
21
22
23 module LFSR32BitsMax
24   #(
25     parameter [4:0] NUM_BITS = 32
26   )
27   (
28     input i_Clk,
29     input i_Enable,
30
31    // Optional Seed Value
32    input i_Seed_DV,
33    input [NUM_BITS-1:0] i_Seed_Data,
34
35    output [NUM_BITS-1:0] o_LFSR_Data,
36    output o_LFSR_Done
37    );
38
39   reg [NUM_BITS:1] r_LFSR = 0;
40   reg              r_XNOR;
41
42
43   // Purpose: Load up LFSR with Seed if Data Valid (DV) pulse is detected.
44   // Othewise just run LFSR when enabled.
45   always @(posedge i_Clk)
46     begin
47       if (i_Enable == 1'b1)
48         begin
49           if (i_Seed_DV == 1'b1)
50             r_LFSR <= i_Seed_Data;
51           else
52             r_LFSR <= {r_LFSR[NUM_BITS-1:1], r_XNOR};
53         end
54     end
55
56   // Create Feedback Polynomials.  Based on Application Note:
57   // http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf
58   always @(*)
59     begin
60       case (NUM_BITS)
61         3: begin
62           r_XNOR = r_LFSR[3] ^~ r_LFSR[2];
63         end
64         4: begin
65           r_XNOR = r_LFSR[4] ^~ r_LFSR[3];
66         end
67         5: begin
68           r_XNOR = r_LFSR[5] ^~ r_LFSR[3];
69         end
70         6: begin
```

```
71            r_XNOR = r_LFSR[6] ^~ r_LFSR[5];
72          end
73          7: begin
74            r_XNOR = r_LFSR[7] ^~ r_LFSR[6];
75          end
76          8: begin
77            r_XNOR = r_LFSR[8] ^~ r_LFSR[6] ^~ r_LFSR[5] ^~ r_LFSR[4];
78          end
79          9: begin
80            r_XNOR = r_LFSR[9] ^~ r_LFSR[5];
81          end
82          10: begin
83            r_XNOR = r_LFSR[10] ^~ r_LFSR[7];
84          end
85          11: begin
86            r_XNOR = r_LFSR[11] ^~ r_LFSR[9];
87          end
88          12: begin
89            r_XNOR = r_LFSR[12] ^~ r_LFSR[6] ^~ r_LFSR[4] ^~ r_LFSR[1];
90          end
91          13: begin
92            r_XNOR = r_LFSR[13] ^~ r_LFSR[4] ^~ r_LFSR[3] ^~ r_LFSR[1];
93          end
94          14: begin
95            r_XNOR = r_LFSR[14] ^~ r_LFSR[5] ^~ r_LFSR[3] ^~ r_LFSR[1];
96          end
97          15: begin
98            r_XNOR = r_LFSR[15] ^~ r_LFSR[14];
99          end
100         16: begin
101           r_XNOR = r_LFSR[16] ^~ r_LFSR[15] ^~ r_LFSR[13] ^~ r_LFSR[4];
102           end
103         17: begin
104           r_XNOR = r_LFSR[17] ^~ r_LFSR[14];
105         end
106         18: begin
107           r_XNOR = r_LFSR[18] ^~ r_LFSR[11];
108         end
109         19: begin
110           r_XNOR = r_LFSR[19] ^~ r_LFSR[6] ^~ r_LFSR[2] ^~ r_LFSR[1];
111         end
112         20: begin
113           r_XNOR = r_LFSR[20] ^~ r_LFSR[17];
114         end
115         21: begin
116           r_XNOR = r_LFSR[21] ^~ r_LFSR[19];
117         end
118         22: begin
119           r_XNOR = r_LFSR[22] ^~ r_LFSR[21];
120         end
121         23: begin
122           r_XNOR = r_LFSR[23] ^~ r_LFSR[18];
123         end
124         24: begin
125           r_XNOR = r_LFSR[24] ^~ r_LFSR[23] ^~ r_LFSR[22] ^~ r_LFSR[17];
126         end
127         25: begin
128           r_XNOR = r_LFSR[25] ^~ r_LFSR[22];
129         end
130         26: begin
131           r_XNOR = r_LFSR[26] ^~ r_LFSR[6] ^~ r_LFSR[2] ^~ r_LFSR[1];
132         end
133         27: begin
134           r_XNOR = r_LFSR[27] ^~ r_LFSR[5] ^~ r_LFSR[2] ^~ r_LFSR[1];
135         end
136         28: begin
137           r_XNOR = r_LFSR[28] ^~ r_LFSR[25];
138         end
139         29: begin
140           r_XNOR = r_LFSR[29] ^~ r_LFSR[27];
141         end
142         30: begin
143           r_XNOR = r_LFSR[30] ^~ r_LFSR[6] ^~ r_LFSR[4] ^~ r_LFSR[1];
144         end
145         31: begin
146           r_XNOR = r_LFSR[31] ^~ r_LFSR[28];
147         end
148         32: begin
149           r_XNOR = r_LFSR[32] ^~ r_LFSR[22] ^~ r_LFSR[2] ^~ r_LFSR[1];
150         end
151
152       endcase // case (NUM_BITS)
153     end // always @ (*)
154
155
156   assign o_LFSR_Data = r_LFSR[NUM_BITS:1];
157
158   // Conditional Assignment (?)
159   assign o_LFSR_Done = (r_LFSR[NUM_BITS:1] == i_Seed_Data) ? 1'b1 : 1'b0;
160
161 endmodule
```

B.1.2. Memoria FIFO

```
1  ////////////////////////////// FIFO RAM MEMORY ////////////////////////////////////--
2  // ***********************************************************************
3  // FileName: FIFO_v.v
```

```
4   // FPGA: Lattice ECP2-70E
5   // IDE: Lattice Diamond ver 2.0.1
6   //
7   // HDL IS PROVIDED "AS IS." DIGI-KEY EXPRESSLY DISCLAIMS ANY
8   // WARRANTY OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING BUT NOT
9   // LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
10  // PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL DIGI-KEY
11  // BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT OR CONSEQUENTIAL
12  // DAMAGES, LOST PROFITS OR LOST DATA, HARM TO YOUR EQUIPMENT, COST OF
13  // PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR SERVICES, ANY CLAIMS
14  // BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF),
15  // ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER SIMILAR COSTS.
16  // DIGI-KEY ALSO DISCLAIMS ANY LIABILITY FOR PATENT OR COPYRIGHT
17  // INFRINGEMENT.
18  //
19  // Version History
20  // Version 1.0 28/7/2013 Tony Storey
21  // Initial Public Release
22
23  `timescale 1ns/ 100 ps
24
25  // buffer length must be less than or equal to address space as in  BUFF_L <or= 2^(ADDR_W)-1
26  module FIFO_v #(parameter ADDR_W = 4, DATA_W = 24, BUFF_L = 16, ALMST_F = 3, ALMST_E = 3)
27                          (
28                          output reg              [DATA_W- 1      :       0]              data_out,
29                          output reg              [ADDR_W         :       0]              data_count,
30                          output reg                                                              empty,
31                          output reg                                                              full,
32                          output reg                                                              almst_empty,
33                          output reg                                                              almst_full,
34                          output reg                                                              err,
35                          input  wire     [DATA_W -1      :       0]              data_in,
36                          input  wire                                                             wr_en,
37                          input  wire                                                             rd_en,
38                          input  wire                                                     n_reset,
39                          input  wire                                                             clk
40                          );
41
42
43  ////--------------- internal variables ---------------------------------------------------
44
45                          reg                             [DATA_W-1 : 0]  mem_array [0 : (2**ADDR_W)-1];
46                          reg                                     [ADDR_W-1 : 0]  rd_ptr, wr_ptr;
47                          reg                                     [ADDR_W-1 : 0]  rd_ptr_nxt, wr_ptr_nxt;
48                          reg                                                             full_ff, empty_ff;
49                          reg                                                             full_ff_nxt, empty_ff_nxt;
50                          reg                                                             almst_f_ff, almst_e_ff;
51                          reg                                                             almst_f_ff_nxt, almst_e_ff_nxt;
52                          reg                                     [ADDR_W : 0]    q_reg, q_nxt;
53                          reg                                                             q_add, q_sub;
54  //// -------------------------------------------------------------------------------------------
55
56
57  //// Always block to update the states
58  //// -------------------------------------------------------------------------------------------
59          always @ (posedge clk)
60                  begin   :       reg_update
61                          if(n_reset == 1'b 0)
62                                  begin
63                                          rd_ptr <= {(ADDR_W-1){1'b 0}};
64                                          wr_ptr <= {(ADDR_W-1){1'b 0}};
65                                          full_ff <= 1'b 0;
66                                          empty_ff <= 1'b 1;
67                                          almst_f_ff <= 1'b 0;
68                                          almst_e_ff <= 1'b 1;
69                                          q_reg <= {(ADDR_W){1'b 0}};
70                                  end
71                          else
72                                  begin
73                                          rd_ptr <= rd_ptr_nxt;
74                                          wr_ptr <= wr_ptr_nxt;
75                                          full_ff <= full_ff_nxt;
76                                          empty_ff <= empty_ff_nxt;
77                                          almst_f_ff <= almst_f_ff_nxt;
78                                          almst_e_ff <= almst_e_ff_nxt;
79                                          q_reg <= q_nxt;
80                                   end
81                  end     // end of always
82
83  //// Control for almost full and almost emptly flags
84  //// -------------------------------------------------------------------------------------------
85          always @ ( almst_e_ff, almst_f_ff, q_reg)
86                  begin   :       Wtr_Mrk_Cont
87                          almst_e_ff_nxt = almst_e_ff;
88                          almst_f_ff_nxt = almst_f_ff;
89                     //// check to see if wr_ptr is ALMST_E away from rd_ptr (aka almost empty)
90                          if(q_reg < ALMST_E)
91                                  almst_e_ff_nxt = 1'b 1;
92                          else
93                                  almst_e_ff_nxt = 1'b 0;
94
95                          if(q_reg > BUFF_L-ALMST_F)
96                                  almst_f_ff_nxt = 1'b 1;
97                          else
98                                  almst_f_ff_nxt = 1'b 0;
99
100                 end     // end of always
101
```

```
//// Control for read and write pointers and empty/full flip flops
        always @ (wr_en, rd_en, wr_ptr, rd_ptr, empty_ff, full_ff, q_reg)
                begin

                        wr_ptr_nxt = wr_ptr ;           //// no change to pointers
                        rd_ptr_nxt = rd_ptr;
                        full_ff_nxt = full_ff;
                        empty_ff_nxt = empty_ff;
                        q_add = 1'b 0;
                        q_sub = 1'b 0;
                ////---------- check if fifo is full during a write attempt, after a write increment counter
                ////-------------------------------------------------
                        if(wr_en == 1'b 1 & rd_en == 1'b 0)
                                begin
                                        if(full_ff == 1'b 0)
                                                begin
                                                        if(wr_ptr < BUFF_L-1)
                                                                begin
                                                                        q_add = 1'b 1;
                                                                        wr_ptr_nxt = wr_ptr + 1;
                                                                        empty_ff_nxt = 1'b 0;
                                                                end
                                                        else
                                                                begin
                                                                        wr_ptr_nxt = {(ADDR_W-1){1'b 0}};
                                                                        empty_ff_nxt = 1'b 0;
                                                                end
                                                        //// check if fifo is full
                                                        if( (wr_ptr+1 == rd_ptr) || ((wr_ptr == BUFF_L-1) && (rd_ptr == 1'b 0)))
                                                                full_ff_nxt = 1'b 1;
                                                end
                                end

                ////---------- check to see if fifo is empty during a read attempt, after a read decrement counter
                ////-------------------------------------------------------
                        if( (wr_en == 1'b 0) && (rd_en == 1'b 1))
                                begin
                                        if(empty_ff == 1'b 0)
                                                begin
                                                        if(rd_ptr < BUFF_L-1 )
                                                                begin
                                                                        if(q_reg > 0)
                                                                                q_sub = 1'b 1;
                                                                        else
                                                                                q_sub = 1'b 0;
                                                                        rd_ptr_nxt = rd_ptr + 1;
                                                                        full_ff_nxt = 1'b 0;
                                                                end
                                                        else
                                                                begin
                                                                        rd_ptr_nxt = {(ADDR_W-1){1'b 0}};
                                                                        full_ff_nxt = 1'b 0;
                                                                end

                                                        //// check if fifo is empty
                                                        if( (rd_ptr  + 1 == wr_ptr) || ((rd_ptr == BUFF_L -1) && (wr_ptr == 1'b 0 )))
                                                                empty_ff_nxt = 1'b 1;
                                                end
                                end

                //// ------------------------------------------------------------
                        if( (wr_en == 1'b 1) && (rd_en == 1'b 1))
                                begin
                                        if(wr_ptr < BUFF_L -1)
                                                wr_ptr_nxt = wr_ptr  + 1;
                                        else
                                                wr_ptr_nxt =  {(ADDR_W-1){1'b 0}};

                                        if(rd_ptr < BUFF_L -1)
                                                rd_ptr_nxt = rd_ptr + 1;
                                        else
                                                rd_ptr_nxt = {(ADDR_W-1){1'b 0}};
                                end

                end  // end of always


//// Control for memory array writing and reading
//// ------------------------------------------------------------------
        always @ (posedge clk)
                begin           :               mem_cont
                        if( n_reset == 1'b 0)
                                begin
                                        mem_array[rd_ptr] <=  {(DATA_W-1){1'b 0}};
                                        data_out <= {(DATA_W-1){1'b 0}};
                                        err <= 1'b 0;
                                end
                        else
                                begin
                                        ////  if write enable and not full then latch in data and increment wright pointer
                                        if( (wr_en == 1'b 1) && (full_ff == 1'b 0) )
                                                begin
                                                        mem_array[wr_ptr] <=  data_in;
                                                        err <= 1'b 0;
                                                end
                                        else if( (wr_en == 1'b 1) && (full_ff == 1'b 1))      ////  check if full and trying to write
                                                err <= 1'b 1;

```

```
200                                     //// if read enable and fifo not empty then latch data out and increment read pointer
201                                     if( (rd_en == 1'b 1) && (empty_ff == 1'b 0))
202                                             begin
203                                                     data_out <= mem_array[rd_ptr];
204                                                     err <= 1'b 0;
205                                             end
206                                     else if( (rd_en == 1'b 1) && (empty_ff == 1'b 1))
207                                             err <= 1'b 1;
208
209                             end     // end else
210             end     // end always
211
212
213 //// Combo Counter with Control Flags
214 //// -------------------------------------------------------------------------------------------
215         always @ ( q_sub, q_add, q_reg)
216                 begin   :       Counter
217                         case( {q_sub , q_add} )
218                                 2'b 01 :
219                                                 q_nxt = q_reg + 1;
220                                 2'b 10 :
221                                                 q_nxt = q_reg - 1;
222                                 default :
223                                                 q_nxt = q_reg;
224                         endcase
225                 end     // end of always
226
227 //// Connect internal regs to ouput ports
228 //// -------------------------------------------------------------------------------------------
229         always @ (full_ff, empty_ff, almst_e_ff, almst_f_ff, q_reg)
230                 begin
231                         full = full_ff;
232                         empty = empty_ff;
233                         almst_empty = almst_e_ff;
234                         almst_full = almst_f_ff;
235                         data_count = q_reg;
236                 end
237
238 endmodule
```

B.1.3. Codificador Reed Solomon - Maquina de estados

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 15.09.2021 18:35:25
7  // Design Name:
8  // Module Name: Encoder_State_Machine
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////////////////////
21
22
23 module Encoder_State_Machine
24     #(
25         parameter [6:0] MAX_COUNT = 6'd63,
26         parameter [2:0] num_bits = 4,
27         parameter [1:0] s_IDLE = 2'b00,
28         parameter [1:0] s_ENCODE = 2'b01,
29         parameter [1:0] s_WAIT = 2'b10,
30         parameter [1:0] s_FINISH = 2'b11
31     )
32     (
```

```
33          //Entradas
34          input clk, rst, start, stop, finish,
35          input [num_bits - 1:0]                  dataIn,
36          //Salidas
37          output reg     ack, offline, online,
38          output [num_bits-1:0]                  o_data
39      );
40
41      //Declaracin de variables internas-------------------------------------
42      reg  [1:0]                    state = 2'b00, next_state = 2'b00;
43      //Encoder--------------------------------------------------------------
44      wire [num_bits - 1:0]                       Encoder_o_data;
45      wire [num_bits - 1:0]                     Enabled_dataIn;
46      reg                                 fsm_control;
47      //RAM y SealesAdicionales----------------------------------------------
48      reg                                       RAM_writeEnable;
49      reg [12:0]                                    RAM_address;
50      reg                                       count_en = 1'b0;
51      reg                                       rst_counter = 1'b0;
52      wire[6:0]                                   counter_value;
53      reg [num_bits - 1 :0]                              input_data_enable;
54      reg [num_bits - 1 :0]                             output_data_enable;
55      always @(posedge clk or negedge rst) begin
56          if (~rst) begin
57              state <= s_IDLE;
58          end
59          else begin
60              //Asigno por defecto las variables y el estado
61              state <= next_state;
62              offline <= 1'b0;
63              online <= 1'b1;
64              ack <= 1'b0;
65              case (state)
66                  s_IDLE: begin
67                      offline <= 1'b1;
68                      fsm_control <= 1'b0;
69                      online <= 1'b0;
70                      input_data_enable <= {num_bits{1'b0}};
71                      if (start) begin
72                          next_state <= s_ENCODE;
73                          rst_counter <= 1'b1;
74                          RAM_address <= 12'd0;
75                          RAM_writeEnable <= 1'b0;
76                      end
77                      else if (stop) begin
78                          RAM_address <= 12'd0;
79                          RAM_writeEnable <= 1'b0;
80                          next_state <= s_WAIT;
81                      end
82                      else
83                          next_state <= s_IDLE;
84                  end
85                  s_ENCODE: begin
86                      if(counter_value < 11) begin
87                          count_en <= 1'b1;
88                          RAM_writeEnable <= 1'b1;
89                          fsm_control <= 1'b1;
```

```
90                          input_data_enable <= {num_bits{1'b1}};
91  //                        if (counter_value > 0) begin
92                              RAM_address <= RAM_address + 12'd1;
93  //                        end
94                      end
95                      else if (counter_value > 10 && counter_value < 13) begin
96                          fsm_control <= 1'b0;
97                          input_data_enable <= {num_bits{1'b0}};
98                          if (counter_value == 11)
99                            RAM_address <= RAM_address + 12'd1;
100                     end
101                     else if (counter_value > 12 && counter_value < 17) begin
102                         RAM_address <= RAM_address + 12'd1;
103                     end
104                     else begin
105                         count_en <= 1'b0;
106                         next_state <= s_FINISH;
107                     end
108                 end
109                 s_WAIT: begin
110                         if(counter_value > 17 && counter_value < 33) begin
111                             count_en <= 1'b1;
112                             RAM_address <= RAM_address + 12'd1;
113                         end
114                         else
115                             next_state <= s_FINISH;
116                 end
117                 s_FINISH: begin
118                         RAM_writeEnable <= 1'b0;
119 //                        output_data_enable <= 8'd0;
120                         next_state <= s_IDLE;
121                         rst_counter <= 1'b1;
122                         ack <= 1'b1;
123                 end
124               endcase
125             end
126         end
127
128     assign Enabled_dataIn = dataIn & input_data_enable;
129
130 Encoder
131     u_Encoder
132         (.clk       (clk),
133         .rst        (rst),
134         .control    (fsm_control),
135         .I_data     (Enabled_dataIn),
136         .O_data     (Encoder_o_data));
137
138 binary_counter_fsm
139     #(
140         .MAX_COUNT          (MAX_COUNT)
141     )
142     u_binary_counter
143     (
144         .clk                (clk),
145         .rst                (rst_counter),
146         .enable           (count_en),
```

```
147             .count_value      (counter_value)
148         );
149
150     RAM1PORT
151     u_RAM1PORT
152             (.address(RAM_address),
153              .clock (clk),
154              .wren(RAM_writeEnable),
155              .data(Encoder_o_data),
156              .q   (o_data));
157
158     //BRAM
159     //     BRAM_i
160     //         (.BRAM_PORTA_0_addr (RAM_address),
161     //          .BRAM_PORTA_0_clk         (clk),
162     //          .BRAM_PORTA_0_din   (Encoder_o_data),
163     //          .BRAM_PORTA_0_dout   (o_data),
164     //          .BRAM_PORTA_0_we(RAM_writeEnable));
165     endmodule
```

B.1.4. Decodificador Reed Solomon - Maquina de estados

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date: 19.08.2021 19:29:21
7   // Design Name:
8   // Module Name: State_machine
9   // Project Name:
10  // Target Devices:
11  // Tool Versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21
22  module Decoder_State_machine
23      #(
24          parameter [6:0] MAX_COUNT = 6'd63,
25          parameter [2:0] num_bits = 4,
26          parameter [1:0] s_IDLE = 2'b00,
27          parameter [1:0] s_SYNDROM = 2'b01,
28          parameter [1:0] s_EUCLIDEAN = 2'b10,
29          parameter [1:0] s_FINISH = 2'b11
30      )
31      (
32         //Entradas
33         input clk, rst, start, stop, finish,
34         input [3:0]  dataIn,
35         input [3:0]  mem_address,
36         //Salidas
37         output reg ack_lock/* synthesis noprune */, ack_finish, offline, online, /* synthesis noprune */
38         output [3:0] o_data,
39         output reg       output_data_flag /* synthesis noprune */,
40         output reg       set_rst /* synthesis noprune */
41      );
42      //Declaraci?n de variables internas
43      reg  [1:0]                  state = 2'b00, next_state = 2'b00 /* synthesis noprune */;
44
45      //Syndrom-------------------------------------------------------------------
46      reg                                 s_control = 1'b0/* synthesis noprune */;
47      reg [num_bits - 1 : 0]            s_constant1 = {(num_bits - 1){1'b0}} /* synthesis noprune */;
48      reg [num_bits - 1 : 0]            s_constant2 = {(num_bits - 1){1'b0}} /* synthesis noprune */;
49      reg [num_bits - 1 : 0]            s_constant3 = {(num_bits - 1){1'b0}} /* synthesis noprune */;
50      reg [num_bits - 1 : 0]            s_constant4 = {(num_bits - 1){1'b0}} /* synthesis noprune */;
51
52      //Euclidean-----------------------------------------------------------------
53      reg   [1:0]                        euc_shift = 2'b00;
54      reg                               euc_control = 1'b0/* synthesis noprune */;
55      reg   [1:0]                     euc_control_2 = 2'b00 /* synthesis noprune */;
56      reg                             euc_control_3 = 1'b0/* synthesis noprune */;
57      reg   [1:0]                     euc_control_4 = 2'b00 /* synthesis noprune */;
58      reg                                   euc_sel = 1'b0 /* synthesis noprune */;
59      reg                                 euc_sel_2 = 1'b0/* synthesis noprune */;
60
61      //ChienSearch---------------------------------------------------------------
```

```
62      reg                                     chien_sel = 1'b0;
63
64      //ForneyAlgorithm------------------------------------------------------------
65      reg                                   forney_sel = 1'b0;
66
67      //RAM y Se?alesAdicionales---------------------------------------------------
68      reg                                     RAM_writeEnable /* synthesis noprune */;
69      reg [3:0]                                   RAM_address/* synthesis noprune */;
70      reg                                    count_en = 1'b0;
71      wire    [num_bits -1 : 0]          euclidean_mult2_output;
72      wire  [6:0]                              counter_value /* synthesis noprune */;
73      reg                                  change_mult_output;
74      reg                                    one_div_flag_reg/* synthesis noprune */;
75      reg [3:0]                               input_data_enable /* synthesis noprune */;
76      reg [3:0]                              output_data_enable/* synthesis noprune */;
77
78
79     //Procedimiento con clock y reset sincronico
80      always @ (posedge clk or negedge rst) begin
81          if (~rst) begin
82              state <= s_IDLE;
83              next_state <= s_IDLE;
84              set_rst <= 1'b0;
85              one_div_flag_reg <= 1'b0;
86              change_mult_output <= 1'b0;
87              input_data_enable <= 4'd0;
88              output_data_enable <= 4'd0;
89              output_data_flag <= 1'b0;
90              RAM_address <= 4'd0;
91              RAM_writeEnable <= 1'b0;
92              count_en <= 1'b0;
93              forney_sel <= 1'b0;
94              chien_sel <= 1'b0;
95              euc_shift <= 1'b0;
96              euc_control <= 1'b0;
97              euc_control_2 <= 2'b00;
98              euc_control_3 <= 1'b0;
99              euc_control_4 <= 2'b00;
100             euc_sel <= 1'b0;
101             euc_sel_2 <= 1'b0;
102             s_control <= 1'b0;
103             s_constant1 <= {(num_bits - 1){1'b0}};
104             s_constant2 <= {(num_bits - 1){1'b0}};
105             s_constant3 <= {(num_bits - 1){1'b0}};
106             s_constant4 <= {(num_bits - 1){1'b0}};
107         end
108         else begin
109             //Asigno por defecto las variables y el estado
110             set_rst <= 1'b1;
111             state <= next_state;
112             ack_lock <= 1'b0;
113             ack_finish <= 1'b0;
114             offline <= 1'b0;
115             online <= 1'b1;
116             case (state)
117                 s_IDLE: begin
118                     offline <= 1'b1;
119                     online <= 1'b0;
120                     s_control <= 1'b0;
121                     ack_finish <= 1'b0;
122                     set_rst <= 1'b0;
123                     RAM_address <= mem_address;
124                     RAM_writeEnable <= 1'b1;
125                     if (start) begin
126                         next_state <= s_SYNDROM;
127                         RAM_address <= 4'd0;
128                         RAM_writeEnable <= 1'b0;
129                         set_rst <= 1'b1;
130                     end
131                     else
132                         next_state <= s_IDLE;
133                 end
134
135                 s_SYNDROM: begin
136                     if (counter_value <15) begin
137                         count_en <= 1'b1;
138                         input_data_enable <= 4'd15;
139                         s_constant1 <= 4'b0001;
140                         s_constant2 <= 4'b0010;
141                         s_constant3 <= 4'b0100;
142                         s_constant4 <= 4'b1000;
143 //                          sr_readwrite <= 2'b01;
144                         RAM_address <= RAM_address + 4'd1;
145                         if (counter_value == 7'd1) begin
146                             s_control <= 1'b1;
147                         end
148
149 //Puedo hacer otro contador que se active con este contador
150                     end
151
152                     else if (counter_value == 15) begin
153                         s_constant1 <= 4'b0001;
154                         s_constant2 <= 4'b0001;
155                         s_constant3 <= 4'b0001;
156                         s_constant4 <= 4'b0001;
157                         // RAM_address <= RAM_address;
158                     end
159
```

```
160                     else if (counter_value == 16) begin
161                         ack_lock <= 1'b1;
162                         s_constant1 <= 4'b0000;
163                         s_constant2 <= 4'b0000;
164                         s_constant3 <= 4'b0000;
165                         s_constant4 <= 4'b0000;
166                         count_en <= 1'b0;
167                     end
168                     else begin
169 //                          sr_readwrite <= 2'b00;
170                         next_state <= s_EUCLIDEAN;
171                         s_control <= 1'b0;
172                     end
173                 end
174 
175                 s_EUCLIDEAN: begin
176                     if (counter_value == 17 || counter_value == 18) begin
177                         ack_lock <= 1'b0;
178                         count_en <= 1'b1;
179                         euc_control <= 1'b1;
180                         euc_control_2 <= 2'b10;
181                         euc_control_3 <= 1'b1;
182                         euc_control_4 <= 2'b10;
183                         if(I_A3_zero_flag)
184                             euc_shift <= 2'b10;
185                         else
186                             euc_shift <= 2'b01;
187                     end
188 
189                     else if (counter_value == 19) begin
190                         euc_control_2 <= 2'b00;
191                         euc_control_4 <= 2'b00;
192                     end
193 
194                     else if (counter_value == 20) begin
195                         euc_control_2 <= 2'b11;
196                         euc_control_4 <= 2'b11;
197                         RAM_address <= 4'd0;
198                         if(I_A3_zero_flag)
199                             euc_shift <= 2'b01;
200                         else
201                             euc_shift <= 2'b00;
202                     end
203                     //Ac? tengo que agregar 1 estado de espera para decidir si
204                     //Sigo calculando o ya termin? la cuenta
205 
206                     else if (counter_value == 21) begin
207                         if(euclidean_mult2_output == 4'b0) begin
208                             one_div_flag_reg <= 1'b1;
209                             change_mult_output <= 1'b1;
210                             RAM_address <= RAM_address + 4'd1;
211                         end
212                     end
213 
214                    else if (counter_value == 22) begin
215                         if (one_div_flag_reg == 1'b1) begin
216                             chien_sel <= 1'b1;
217                             forney_sel <= 1'b1;
218                             RAM_address <= RAM_address + 4'd1;
219 //                            sr_readwrite <= 2'b10;
220 //                            euc_control_2 <= 2'b00;
221                         end
222                         else begin
223                             if(I_A3_zero_flag) begin
224                                 euc_control <= 1'b1;
225                                 euc_control_2 <= 2'b00;
226                                 euc_control_4 <= 2'b00;
227                             end
228                             else begin
229                                 euc_control <= 1'b0;
230                                 euc_control_2 <= 2'b01;
231                                 euc_control_4 <= 2'b11;
232                             end
233                             euc_shift <= 2'b00;
234                             euc_sel <= 1'b1;
235                         end
236                     end
237                     else if (counter_value == 23) begin
238                         if (one_div_flag_reg == 1'b1) begin
239                             chien_sel <= 1'b0;
240                             forney_sel <= 1'b0;
241                             RAM_address <= RAM_address + 4'd1;
242                             output_data_enable <= 4'd15;
243                             output_data_flag <= 1'b1;
244                         end
245                         else begin
246                             euc_control_2 <= 2'b11;
247                             euc_sel <= 1'b0;
248                             euc_sel_2 <= 1'b1;
249                             euc_control_3 <= 1'b0;
250                             if (I_A3_zero_flag) begin
251                                 euc_shift <= 2'b00;
252                                 euc_control_4 <= 2'b11;
253                             end
254                             else begin
255                                 euc_control_4 <= 2'b01;
256                                 euc_shift <= 2'b01;
257                             end
```

```
258                         end
259                     end
260
261                     else if (counter_value == 24) begin
262                         if (one_div_flag_reg == 1'b1) begin
263                             RAM_address <= RAM_address + 4'd1;
264                         end
265                         else begin
266                             euc_sel_2 <= 1'b0;
267                             if (I_A3_zero_flag) begin
268                                 euc_control_4 <= 2'b00;
269                                 euc_shift <= 2'b11;
270                             end
271                             else
272                                 euc_control_4 <= 2'b11;
273                         end
274                     end
275
276                     else if (counter_value == 25) begin
277                         if (one_div_flag_reg == 1'b1) begin
278                             RAM_address <= RAM_address + 4'd1;
279                         end
280                         else begin
281                             if(I_A3_zero_flag) begin
282                                 euc_control_2 <= 2'b11;
283                                 euc_control_4 <= 2'b11;
284                             end
285                             else begin
286                                 euc_control_2 <= 2'b00;
287                                 euc_control_4 <= 2'b00;
288                             end
289                             RAM_address <= 4'd0;
290                         end
291                     end
292
293                     else if (counter_value == 26) begin
294                        if (one_div_flag_reg == 1'b1) begin
295                             RAM_address <= RAM_address + 4'd1;
296                         end
297                         else begin
298                             euc_control_2 <= 2'b11;
299                             euc_control_4 <= 2'b11;
300                             euc_shift <= 1'b0;
301                             RAM_address <= RAM_address + 4'd1;
302                             output_data_enable <= 4'd15;
303                             if (I_A3_zero_flag)
304                                 euc_shift <= 2'b11;
305                             else
306                                 euc_shift <= 2'b00;
307                         end
308
309                     end
310
311                     else if (counter_value == 27) begin
312                        if (one_div_flag_reg == 1'b1) begin
313                             RAM_address <= RAM_address + 4'd1;
314                         end
315                         else begin
316                             chien_sel <= 1'b1;
317                             forney_sel <= 1'b1;
318                             RAM_address <= RAM_address + 4'd1;
319                             output_data_enable <= 4'd15;
320
321                         end
322                     end
323
324                     else if (counter_value == 28) begin
325                        if (one_div_flag_reg == 1'b1) begin
326                             RAM_address <= RAM_address + 4'd1;
327                         end
328                         else begin
329                             chien_sel <= 1'b0;
330                             forney_sel <= 1'b0;
331                             RAM_address <= RAM_address + 4'd1;
332                             output_data_enable <= 4'd15;
333                             output_data_flag <= 1'b1;
334                         end
335                     end
336                     else if (counter_value > 28 && counter_value < 41)
337                         RAM_address <= RAM_address + 4'd1;
338                         if (counter_value > 37 && one_div_flag_reg == 1'b1) begin
339                             output_data_flag <= 1'b0;
340                             output_data_enable <= 4'd0;
341                         end
342                         if (counter_value == 41) begin
343                             next_state <= s_FINISH;
344                         end
345                 end
346
347                     s_FINISH: begin
348                     //Ac? tendr?a que enviarle el ack al procesador y resetear o finalizar seg?n el caso.
349                         set_rst <= 1'b0;
350                         count_en <= 1'b0;
351                         output_data_enable <= 4'd0;
352                         output_data_flag <= 1'b0;
353                         ack_finish <= 1'b1;
354                         next_state <= s_IDLE;
355                     end
```

```
356                 endcase
357               end
358             end
359
360  Decoder_Top
361       #(
362         .num_bits                 (num_bits)
363        )
364        u_Decoder_Top
365        (
366         .i_data                    (dataIn),
367         .t_clk                       (clk),
368         .t_rst                       (set_rst),
369         .s_control                (s_control),
370         .ack                           (ack_lock),
371         .s_constant1              (s_constant1),
372         .s_constant2              (s_constant2),
373         .s_constant3              (s_constant3),
374         .s_constant4              (s_constant4),
375         .euc_enableshift            (euc_shift),
376         .euc_control              (euc_control),
377         .euc_control_2          (euc_control_2),
378         .euc_control_3          (euc_control_3),
379         .euc_control_4          (euc_control_4),
380         .euc_sel                      (euc_sel),
381         .euc_sel_2                  (euc_sel_2),
382         .chien_sel                  (chien_sel),
383         .forney_sel                (forney_sel),
384         .change_mult_output      (change_mult_output),
385 //        .sr_readwrite           (sr_readwrite),
386         .RAM_writeEnable          (RAM_writeEnable),
387         .RAM_address              (RAM_address),
388         .input_data_enable          (input_data_enable),
389         .output_data_enable         (output_data_enable),
390         .euclidean_mult2_output (euclidean_mult2_output),
391         .I_A3_zero_flag           (I_A3_zero_flag),
392         .o_data                       (o_data)
393       );
394
395  binary_counter_fsm
396      #(
397          .MAX_COUNT            (MAX_COUNT)
398      )
399      u_binary_counter
400      (
401          .clk                  (clk),
402          .rst                  (set_rst),
403          .enable             (count_en),
404          .count_value       (counter_value)
405      );
406
407  endmodule
```

B.1.5. Maquina de estados de control general

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date: 20.09.2021 18:10:07
7   // Design Name:
8   // Module Name: Test_Bench_FSM
9   // Project Name:
10  // Target Devices:
11  // Tool Versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21
22
23  module Test_Bench_FSM
24          #(
25          //FSM STATES
26          parameter [2:0] s_IDLE = 3'b000,
27          parameter [2:0] s_START = 3'b001,
28          parameter [2:0] s_DECODING = 3'b010,
29          parameter [2:0] s_NOISE = 3'b110,
30          parameter [2:0] s_FINISH = 3'b011,
31          parameter [2:0] s_WAIT  = 3'b100,
32
33
34          //WORD SIZE
35          parameter num_bits = 4,
36          //FIFO PARAMETERS
37          parameter FIFO_ADDR_W = 4,
38          parameter FIFO_DATA_W = 4,
39          parameter FIFO_BUFF_L = 15,
40          parameter FIFO_ALMST_F = 7,
41          parameter FIFO_ALMST_E = 1,
```

```
42
43          //BER PARAMETERS
44          parameter BER_Counter_Size = 32,
45
46          //BINARY COUNTER PARAMETER
47          parameter Counter_MAX_COUNT = 8,
48
49          //LFSR PARAMETERS
50          parameter LFSR_SIZE = 7
51
52          )
53          (
54              input clk, rst, start, stop, finish, noise_enable, Decoding_enable, Encoding_enable, Idle_enable,
55                          input [59:0]            Datos_demodulados_buf,
56              output reg     ack, offline, online, Noise_ack, Decoding_ack, Encoder_ack_FSM,
57                          output reg  [2:0]                   state = 3'b000,
58                          //Input words RAM Variables-------------------------------------------------------
59                          output reg        [59 : 0]                                    Datos_codificados_buf, /* synthesis noprune */
60                          output [BER_Counter_Size - 1 : 0]               BER_Count_out /* synthesis noprune */
61          );
62
63
64          //Structural code-----------------------------------------------------------
65          reg  [2:0]                  next_state = 3'b000;
66          reg                         set_rst = 1'b0;
67          //ENCODER Variables---------------------------------------------------------
68          reg                     Encoder_start = 1'b0, Encoder_stop = 1'b0, Encoder_finish = 1'b0;
69          wire                    Encoder_offline, Encoder_online;
70          wire [num_bits - 1 : 0] Encoder_o_data;
71
72          reg [num_bits - 2 : 0]  Encoded_words = {num_bits-2{1'b0}};
73          //DECODER Variables---------------------------------------------------------
74          reg                     Decoder_start = 1'b0, Decoder_stop = 1'b0, Decoder_finish = 1'b0 /* synthesis noprune */;
75          reg [num_bits - 1 : 0]  Decoder_dataIn = {(num_bits){1'b0}} /* synthesis noprune */;
76          wire                    Decoder_ack_finish, Decoder_offline, Decoder_online;
77          wire [num_bits - 1 : 0] Decoder_o_data;
78          reg [num_bits - 1 : 0]  Input_Words_RAM_addr_Deco /* synthesis noprune */;
79          reg [19 : 0]                    Decoded_words = {(20){1'b0}};
80          wire                    Decoder_o_data_enabled;
81          //BER Variables-------------------------------------------------------------
82          reg                     BER_enable;
83          //LFSR32BitsMax Variables---------------------------------------------------
84          reg                     LFSR_Enable = 1'b0 /* synthesis noprune */ ,LFSR_Seed_Valid = 1'b0;
85          reg [LFSR_SIZE - 1 : 0]  LFSR_Seed_Data = {(LFSR_SIZE-1){1'b0}};
86          wire[LFSR_SIZE - 1 : 0]  LFSR_Output_Data;
87          wire                    LFSR_Done;
88
89          //FIFO DECODED Variables----------------------------------------------------
90          reg [FIFO_DATA_W - 1 : 0]  FIFO_decoded_Data_in = {FIFO_DATA_W{1'b0}};
91          reg                        FIFO_decoded_wr_en = 1'b0, FIFO_decoded_rd_en = 1'b0;
92          wire [FIFO_DATA_W - 1 : 0] FIFO_decoded_data_out;
93          wire [FIFO_ADDR_W : 0]     FIFO_decoded_data_count;
94          wire                       FIFO_decoded_empty, FIFO_decoded_full,
95                                     FIFO_decoded_almst_empty, FIFO_decoded_almst_full,
96                                     FIFO_decoded_err;
97          //FIFO CODED Variables------------------------------------------------------
98          reg [FIFO_DATA_W - 1 : 0]  FIFO_coded_Data_in = {FIFO_DATA_W{1'b0}};
99          reg                        FIFO_coded_wr_en = 1'b0, FIFO_coded_rd_en = 1'b0;
100         wire [FIFO_DATA_W - 1 : 0] FIFO_coded_data_out;
101         wire [FIFO_ADDR_W : 0]     FIFO_coded_data_count;
102         wire                       FIFO_coded_empty, FIFO_coded_full,
103                                    FIFO_coded_almst_empty, FIFO_coded_almst_full,
104                                    FIFO_coded_err;
105                 reg    [59:0]                                   Datos_demodulados_buf_reg;
106         //Binary Counter Variables--------------------------------------------------
107         reg                                     count_en = 1'b0;
108         wire  [Counter_MAX_COUNT - 1 :0]          counter_value;
109
110
111         always @(posedge clk or negedge rst) begin
112             if(~rst) begin
113                 state <= s_IDLE;
114                 next_state <= s_IDLE;
115                 set_rst <= 1'b1;
116                 FIFO_coded_Data_in <= {FIFO_DATA_W{1'b0}};
117                 FIFO_coded_wr_en <= 1'b0;
118                 FIFO_coded_rd_en <= 1'b0;
119                 FIFO_decoded_Data_in <= {FIFO_DATA_W{1'b0}};
120                 FIFO_decoded_wr_en <= 1'b0;
121                 FIFO_decoded_rd_en <= 1'b0;
122                 count_en <= 1'b0;
123                 Encoder_start <= 1'b0;
124                 Decoder_start <= 1'b0;
125                 Datos_codificados_buf <= 60'd0;
126                 LFSR_Enable <= 1'b0;
127                 BER_enable <= 1'b0;
128                 Decoded_words <= {(20){1'b0}};
129                 Input_Words_RAM_addr_Deco <= {(num_bits){1'b0}};
130                 Decoder_dataIn <= {(num_bits){1'b0}};
131                                         Datos_demodulados_buf_reg <= 59'd0;
132             end
133             else begin
134                 //Asigno por defecto las variables y el estado
135                 state <= next_state;
136                 offline <= 1'b0;
137                 online <= 1'b1;
138                                         ack <= 1'b0;
139                         case (state)
```

```
140                     s_IDLE: begin
141                                     if(Idle_enable) begin
142                             offline <= 1'b1;
143                             Decoder_start <= 1'b0;
144                                             Encoder_start <= 1'b0;
145                             online <= 1'b0;
146                                             Noise_ack <= 1'b0;
147                                             Decoding_ack <= 1'b0;
148                                             Encoder_ack_FSM <= 1'b0;
149                             set_rst <= 1'b1;
150                             if(start) begin
151                                 next_state <= s_START;
152                                             LFSR_Enable <= 1'b1;
153                                 Input_Words_RAM_addr_Deco <= 4'b0;
154                             end
155
156                             else if(stop) begin
157                                 Encoder_start <= 1'b0;
158                                 Decoder_start <= 1'b0;
159                                             set_rst <= 1'b0;
160                             end
161                                     end
162                                     else begin
163                                             next_state <= s_IDLE;
164                                             Noise_ack <= 1'b0;
165                                             Encoder_ack_FSM <= 1'b0;
166                                             set_rst <= 1'b0;
167                                     end
168                     end
169                     s_START: begin
170                                     if(Encoding_enable) begin
171                                             count_en <= 1'b1;
172                                         if(counter_value == 1)
173                                                 Encoder_start <= 1'b1;
174                             if(counter_value > 7 && counter_value < 23) begin
175                                 Datos_codificados_buf <= {Datos_codificados_buf [55:0],Encoder_o_data};
176                                             FIFO_coded_Data_in <= Encoder_o_data;
177                                 FIFO_coded_wr_en <= 1'b1;
178                                             end
179                                             else if (counter_value > 23)
180                                                     FIFO_coded_wr_en <= 1'b0;
181                                             if(Encoder_ack == 1'b1) begin
182                                                     Encoded_words <= Encoded_words + 3'd1;
183                                                     count_en <= 1'b0;
184                                                     next_state <= s_NOISE;
185                                                     Encoder_start <= 1'b0;
186                                                     Encoder_ack_FSM <= 1'b1;
187                                             end
188                                     end
189                                     else begin
190                                             count_en <= 1'b0;
191                                             Encoder_start <= 1'b0;
192                                             FIFO_coded_wr_en <= 1'b0;
193                                             next_state <= s_START;
194                                     end
195                     end
196
197                     s_NOISE : begin
198                                     if(noise_enable) begin
199                             if (counter_value > 27 & counter_value < 44) begin
200                                             Encoder_ack_FSM <= 1'b0;
201                                             Datos_demodulados_buf_reg <= Datos_demodulados_buf;
202                                 if(counter_value == 28)
203                                             Datos_demodulados_buf_reg <= Datos_demodulados_buf;
204                                 else if (Input_Words_RAM_addr_Deco < 15) begin
205                                     Datos_demodulados_buf_reg <= Datos_demodulados_buf_reg << 4 ;
206                                     Decoder_dataIn <= Datos_demodulados_buf_reg [59 : 56];
207                                     Input_Words_RAM_addr_Deco <= Input_Words_RAM_addr_Deco + 4'd1;
208                                 end
209                             end
210                             else if (counter_value == 45)  begin
211                                                     next_state <= s_DECODING;
212                                                     Noise_ack <= 1'b1;
213                                                     count_en <= 1'b0;
214                                             end
215                                             else     begin
216                                                 count_en <= 1'b1;
217                                                 Input_Words_RAM_addr_Deco <= 4'd0;
218                                 Decoder_dataIn <= 4'd0;
219                                             end
220                                     end
221                     end
222
223                     s_DECODING: begin
224                                     if(Decoding_enable) begin
225                                             Noise_ack <= 1'b0;
226                                                 count_en <= 1'b1;
227                                                 if (counter_value == 46)
228                                                         Decoder_start <= 1'b1;
229                                                 if(Decoder_o_data_enabled) begin
230                                                         FIFO_decoded_Data_in <= Decoder_o_data;
231                                                         FIFO_decoded_wr_en <= 1'b1;
232                                                 end
233                                                 if (FIFO_decoded_data_count == 14) begin
234                                                         BER_enable <= 1'b1;
235                                                         FIFO_decoded_wr_en <= 1'b0;
236                                                         FIFO_decoded_rd_en <= 1'b1;
237                                                         FIFO_coded_rd_en <= 1'b1;
```

```
                                                    end
                                                    if(Decoder_ack_finish)
                                                            Decoder_start <= 1'b0;
                                                    if (BER_enable && FIFO_coded_empty && FIFO_decoded_empty) begin
                                                            BER_enable <= 1'b0;
                                                            set_rst <= 1'b0;
                                                            FIFO_decoded_rd_en <= 1'b0;
                                                            FIFO_coded_rd_en <= 1'b0;
                                                            if(counter_value > 70)
                                                                Decoded_words <= Decoded_words + 3'd1;
                                                            else
                                                                    Decoded_words <= Decoded_words;
                                                                Decoder_start <= 1'b0;
                                                                Decoding_ack <= 1'b1;
                                                            if(Decoded_words == 20'd1000000) begin
                                                                next_state <= s_FINISH;
                                                                count_en <= 1'b0;
                                                        end
                                                            else begin
                                                                    next_state <= s_IDLE;
                                                                    count_en <= 1'b0;
                                                        end
                                                    end
                                                end
                        end
                                            s_FINISH : begin
                                                    ack <=1'b1;
                                            end
                    endcase
                end
            end

Encoder_State_Machine
    #(
        .num_bits (num_bits)
    )
    u_Encoder_State_Machine
    (
        .clk        (clk),
        .rst        (rst && set_rst),
        .start      (Encoder_start),
        .stop       (stop),
        .finish     (Encoder_finish),
        .dataIn     (LFSR_Output_Data),
        .ack        (Encoder_ack),
        .offline    (Encoder_offline),
        .online     (Encoder_online),
        .o_data     (Encoder_o_data)
    );

Decoder_State_machine
    #(
        .num_bits (num_bits)
    )
    u_Decoder_State_machine
    (
        .clk        (clk),
        .rst        (rst && set_rst),
        .start      (Decoder_start),
        .stop       (Decoder_stop),
        .finish     (Decoder_finish),
        .mem_address(Input_Words_RAM_addr_Deco),
        .dataIn     (Decoder_dataIn),
        .ack_lock   (Decoder_ack_lock),
        .ack_finish (Decoder_ack_finish),
        .offline    (Decoder_offline),
        .online     (Decoder_online),
        .o_data     (Decoder_o_data),
        .output_data_flag (Decoder_o_data_enabled)
    );

BER
    #(
        .word_size      (num_bits),
        .counter_size   (BER_Counter_Size)
    )
    u_BER
    (
        .data_in        (FIFO_decoded_data_out),
        .data_coded     (FIFO_coded_data_out),
        .clk            (clk),
        .rst            (rst),
        .enable         (BER_enable),
        .BER_counter    (BER_Count_out)
    );

LFSR32BitsMax
    #(
      .NUM_BITS(7'd32)
     )
   u_LFSR32BitsMax
     (.i_Clk        (clk),
          .rst          (rst),
      .i_Enable     (LFSR_Enable),
      .i_Seed_DV    (LFSR_Seed_Valid),
      .i_Seed_Data  (32'd63), // Replication
      .o_LFSR_Data  (LFSR_Output_Data),
      .o_LFSR_Done  (LFSR_Done)
```

```
336         );
337
338 FIFO_v
339     #(
340     .ADDR_W       (FIFO_ADDR_W),
341     .DATA_W       (FIFO_DATA_W),
342     .BUFF_L       (FIFO_BUFF_L),
343     .ALMST_F      (FIFO_ALMST_F),
344     .ALMST_E      (FIFO_ALMST_E)
345     )
346     u_FIFO_data_decoded
347     (
348     .data_out     (FIFO_decoded_data_out),
349     .data_count   (FIFO_decoded_data_count),
350     .empty        (FIFO_decoded_empty),
351     .full         (FIFO_decoded_full),
352     .almst_empty(FIFO_decoded_almst_empty),
353     .almst_full   (FIFO_decoded_almst_full),
354     .err          (FIFO_decoded_err),
355     .data_in      (FIFO_decoded_Data_in),
356     .wr_en        (FIFO_decoded_wr_en),
357     .rd_en        (FIFO_decoded_rd_en),
358     .n_reset      (rst && set_rst),
359     .clk          (clk)
360     );
361
362 FIFO_v
363     #(
364     .ADDR_W       (FIFO_ADDR_W),
365     .DATA_W       (FIFO_DATA_W),
366     .BUFF_L       (FIFO_BUFF_L),
367     .ALMST_F      (FIFO_ALMST_F),
368     .ALMST_E      (FIFO_ALMST_E)
369     )
370     u_FIFO_data_coded
371     (
372     .data_out     (FIFO_coded_data_out),
373     .data_count   (FIFO_coded_data_count),
374     .empty        (FIFO_coded_empty),
375     .full         (FIFO_coded_full),
376     .almst_empty(FIFO_coded_almst_empty),
377     .almst_full   (FIFO_coded_almst_full),
378     .err          (FIFO_coded_err),
379     .data_in      (FIFO_coded_Data_in),
380     .wr_en        (FIFO_coded_wr_en),
381     .rd_en        (FIFO_coded_rd_en),
382     .n_reset      (rst && set_rst),
383     .clk          (clk)
384     );
385 binary_counter_fsm
386     u_binary_counter
387     (
388     .clk          (clk),
389     .rst          (rst && set_rst),
390     .enable       (count_en),
391     .count_value(counter_value)
392     );
393 //RAM PARA XILINX:
394 //  Input_Words_RAM
395 //    Input_Words_RAM_i
396 //        (.BRAM_PORTA_0_addr(Input_Words_RAM_addr),
397 //         .BRAM_PORTA_0_clk(clk),
398 //         .BRAM_PORTA_0_din(Encoder_o_data),
399 //         .BRAM_PORTA_0_dout(Input_Words_RAM_dout),
400 //         .BRAM_PORTA_0_we(Input_Words_RAM_we));
401 endmodule
```

B.1.6. Codificador Reed Solomon - Implementación

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:
4  // Engineer:
5  //
6  // Create Date: 31.03.2021 15:44:23
7  // Design Name:
8  // Module Name: Encoder
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
```

```
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////////////////////
21
22
23 module Encoder
24     #(
25         parameter num_bits = 4
26     )
27     (
28         input [num_bits - 1 : 0] I_data,
29         input                    control, clk, rst,
30         output reg [num_bits - 1 : 0] O_data
31     );
32
33         reg [num_bits - 1 : 0]  reg_d_0;
34         reg [num_bits - 1 : 0]  reg_d_1;
35         reg [num_bits - 1 : 0]  reg_d_2;
36         reg [num_bits - 1 : 0]  reg_d_3;
37         wire [num_bits - 1 : 0]  and_output;
38         wire [num_bits-1 : 0]    sum_output_0;
39         wire [num_bits-1 : 0]    sum_output_1;
40         wire [num_bits-1 : 0]    sum_output_2;
41         wire [num_bits-1 : 0]    sum_output_3;
42         wire [num_bits-1 : 0]    mult0_output;
43         wire [num_bits-1 : 0]    mult1_output;
44         wire [num_bits-1 : 0]    mult2_output;
45         wire [num_bits-1 : 0]    mult3_output;
46         always@(posedge clk) begin
47             if(~rst) begin
48                 reg_d_0 <= {(num_bits-1){1'b0}};
49                 reg_d_1 <= {(num_bits-1){1'b0}};
50                 reg_d_2 <= {(num_bits-1){1'b0}};
51                 reg_d_3 <= {(num_bits-1){1'b0}};
52                 O_data <= {(num_bits-1){1'b0}};
53             end
54             else begin
55                 reg_d_0 <= sum_output_1;
56                 reg_d_1 <= sum_output_2;
57                 reg_d_2 <= sum_output_3;
58                 reg_d_3 <= mult3_output;
59                 if (control)
60                     O_data <= I_data;
61                 else
62                     O_data <= reg_d_0;
63             end
64         end
65         assign and_output = (control & rst) ? sum_output_0 : 4'b0000;
66
67 Mod2_Adder
68 u_mod2_Adder_0(
69     .I_A    (reg_d_0),
70     .I_B    (I_data),
71     .O_sum (sum_output_0)
72     );
73
```

```
74  Mod2_Adder
75  u_mod2_Adder_1(
76      .I_A    (reg_d_1),
77      .I_B    (mult0_output),
78      .O_sum   (sum_output_1)
79      );
80
81  Mod2_Adder
82  u_mod2_Adder_2(
83      .I_A    (reg_d_2),
84      .I_B    (mult1_output),
85      .O_sum (sum_output_2)
86      );
87
88  Mod2_Adder
89  u_mod2_Adder_3(
90      .I_A    (reg_d_3),
91      .I_B    (mult2_output),
92      .O_sum (sum_output_3)
93      );
94
95  Multiplier
96  u_Multiplier_0(
97      .I_A     (and_output),
98      .I_B        (4'b1111),
99      .rst            (rst),
100     .O_mult (mult0_output)
101     );
102
103 Multiplier
104 u_Multiplier_1(
105     .I_A     (and_output),
106     .I_B        (4'b0011),
107     .rst            (rst),
108     .O_mult (mult1_output)
109     );
110
111 Multiplier
112 u_Multiplier_2(
113     .I_A     (and_output),
114     .I_B        (4'b0001),
115     .rst            (rst),
116     .O_mult (mult2_output)
117     );
118
119 Multiplier
120 u_Multiplier_3(
121     .I_A     (and_output),
122     .I_B        (4'b1100),
123     .rst            (rst),
124     .O_mult     (mult3_output)
125     );
126 endmodule
```

B.1.7. Multiplicador completo de 4 bits sin acarreo

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Company:
```

```
// Engineer:
//
// Create Date: 13.01.2021 20:16:09
// Design Name:
// Module Name: Multiplier
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Multiplier
    #(
        parameter num_bits = 4
    )
    (
        input [num_bits -1 : 0] I_A,
        input [num_bits -1 : 0] I_B,
//        input                   clk,
        input                   rst,
        output [num_bits-1 : 0] O_mult
    );
    wire [num_bits-1 : 0]    level_b0;
    wire [num_bits-1 : 0]    level_b1;
    wire [num_bits-1 : 0]    level_b2;
    wire [num_bits-1 : 0]    level_b3;
    wire [num_bits   : 0] level_1_xor;
    wire [num_bits-3 : 0] level_2_xor;
    wire [num_bits-3 : 0] level_3_xor;
    wire [num_bits-3 : 0] level_4_xor;


    //Level B0
    assign level_b0[num_bits-1] = I_A[num_bits-1] & I_B[num_bits-4]; //a3b0
    assign level_b0[num_bits-2] = I_A[num_bits-2] & I_B[num_bits-4];//a2b0
    assign level_b0[num_bits-3] = I_A[num_bits-3] & I_B[num_bits-4];//a1b0
    assign level_b0[num_bits-4] = I_A[num_bits-4] & I_B[num_bits-4];//a0b0
    //Level B1
    assign level_b1[num_bits-1] = I_A[num_bits-1] & I_B[num_bits-3];//a3b1
    assign level_b1[num_bits-2] = I_A[num_bits-2] & I_B[num_bits-3];//a2b1
    assign level_b1[num_bits-3] = I_A[num_bits-3] & I_B[num_bits-3];//a1b1
    assign level_b1[num_bits-4] = I_A[num_bits-4] & I_B[num_bits-3];//a0b1
    //Level B2
    assign level_b2[num_bits-1] = I_A[num_bits-1] & I_B[num_bits-2];//a3b2
    assign level_b2[num_bits-2] = I_A[num_bits-2] & I_B[num_bits-2];//a2b2
    assign level_b2[num_bits-3] = I_A[num_bits-3] & I_B[num_bits-2];//a1b2
    assign level_b2[num_bits-4] = I_A[num_bits-4] & I_B[num_bits-2];//a0b2
    //Level B3
    assign level_b3[num_bits-1] = I_A[num_bits-1] & I_B[num_bits-1];//a3b3
    assign level_b3[num_bits-2] = I_A[num_bits-2] & I_B[num_bits-1];//a2b3
    assign level_b3[num_bits-3] = I_A[num_bits-3] & I_B[num_bits-1];//a1b3
    assign level_b3[num_bits-4] = I_A[num_bits-4] & I_B[num_bits-1];//a0b3

    //Level 1 - XOR
    assign level_1_xor[num_bits] = level_b3[num_bits-2] ^ level_b2[num_bits-1];
    assign level_1_xor[num_bits-1] = (level_b3[num_bits-3] ^ level_b2[num_bits-2]) ^ level_b1[num_bits-1];
    assign level_1_xor[num_bits-2] = ((level_b3[num_bits-4] ^ level_b2[num_bits-3]) ^ level_b1[num_bits-2]) ^ level_b0[num_bits-1];
    assign level_1_xor[num_bits-3] = (level_b2[num_bits-4] ^ level_b1[num_bits-3]) ^ level_b0[num_bits-2];
    assign level_1_xor[num_bits-4] = level_b1[num_bits-4] ^ level_b0[num_bits-3];

    //Level 2 - XOR
    assign level_2_xor[num_bits-3] = level_1_xor[num_bits-1] ^ level_1_xor[num_bits-4];
    assign level_2_xor[num_bits-4] = level_1_xor[num_bits-1] ^ level_b0[num_bits-4];

    //Level 3 - XOR
    assign level_3_xor[num_bits-3] = level_1_xor[num_bits] ^ level_1_xor[num_bits-3];
    assign level_3_xor[num_bits-4] = level_1_xor[num_bits] ^ level_2_xor[num_bits-3];

    //Level 4 - XOR
    assign level_4_xor[num_bits-3] = level_b3[num_bits-1] ^ level_1_xor[num_bits-2];
    assign level_4_xor[num_bits-4] = level_b3[num_bits-1] ^ level_3_xor[num_bits-3];

    assign O_mult[num_bits-1] = (~rst) ? 1'b0 : level_4_xor[num_bits-3];
    assign O_mult[num_bits-2] = (~rst) ? 1'b0 : level_4_xor[num_bits-4];
    assign O_mult[num_bits-3] = (~rst) ? 1'b0 : level_3_xor[num_bits-4];
    assign O_mult[num_bits-4] = (~rst) ? 1'b0 : level_2_xor[num_bits-4];

endmodule
```

B.1.8. Decodificador Reed Solomon - Calculo de síndromes

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
```

```
5  //
6  // Create Date: 31.12.2020 20:31:22
7  // Design Name:
8  // Module Name: Syndrom
9  // Project Name:
10 // Target Devices:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //////////////////////////////////////////////////////////////////////////////////
21
22
23 module Syndrom
24     #(
25         parameter datapaths = 4
26     )
27     (
28     input  [datapaths -1 : 0]     i_symbol,
29     input                          control,
30     input                              clk,
31     input                              rst,
32     input  [datapaths -1 : 0] num_constant,
33     output [datapaths -1 : 0]   o_syndrome
34     );
35     //Internal signals
36     wire  [datapaths-1 : 0] sum_output;
37     wire  [datapaths-1 : 0] mult_output;
38     wire  [datapaths-1 : 0] and_output;
39     reg   [datapaths-1 : 0] reg_d;
40
41     always @(posedge clk) begin
42         if(~rst) begin
43             reg_d <= {(datapaths-1){1'b0}};
44         end
45         else begin
46              reg_d <= mult_output;
47         end
48     end
49     assign and_output[datapaths-1] = control & reg_d[datapaths-1];
50     assign and_output[datapaths-2] = control & reg_d[datapaths-2];
51     assign and_output[datapaths-3] = control & reg_d[datapaths-3];
52     assign and_output[datapaths-4] = control & reg_d[datapaths-4];
53
54     assign o_syndrome = and_output;
55
56
57 Mod2_Adder
58 u_mod2_Adder(
59     .I_A    (i_symbol),
60     .I_B  (and_output),
61     .O_sum (sum_output)
```

```
62      );
63
64  Multiplier
65  u_Multiplier(
66      .I_A      (sum_output),
67      .I_B    (num_constant),
68      .rst              (rst),
69      .O_mult  (mult_output)
70      );
71  endmodule
```

B.1.9. Decodificador Reed Solomon - Algoritmo Euclidiano

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date: 17.01.2021 19:35:33
7   // Design Name:
8   // Module Name: Euclidean
9   // Project Name:
10  // Target Devices:
11  // Tool Versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21
22
23  module Euclidean
24      #(
25          parameter num_bits = 4
26      )
27      (
28          input [num_bits-1 : 0] I_A0,I_A1,I_A2,I_A3,
29          input [num_bits-1 : 0] I_B0,I_B1,I_B2,I_B3,
30          input [num_bits-1 : 0] I_C0,I_C1,
31          input [num_bits-1 : 0] I_D0,I_D1,I_D2,
32          input                       clk,
33          input                       rst,
34          input [1:0]              enable_shift,
35          input                       control, //Osea cuando control valga 0, asigno sindromes. Cuando valga 1, los otros valores
36          input [1:0]                 control_2,
37          input                       control_3,
38          input [1:0]                 control_4,
39          input                       sel,
40          input                       sel_2,
41          output [num_bits-1 : 0] O_mult_0, O_mult_1, O_mult_2,
42          output [num_bits-1 : 0] O_div_1, O_div_2,
43          output reg                 I_A3_zero_flag = 1'b0,
44          output [num_bits-1 : 0] constant
45      );
46          reg [num_bits-1 : 0] A0, A1, A2, A3/* synthesis noprune */;
47          reg [num_bits-1 : 0] B0, B1, B2, B3 /* synthesis noprune */;
48          reg [num_bits-1 : 0] C0, C1 /* synthesis noprune */;
49          reg [num_bits-1 : 0] D0, D1, D2 /* synthesis noprune */;
50          wire[num_bits-1 : 0] A_Inverted;
51          wire[num_bits-1 : 0] A0_B0_sum,A1_B1_sum,A2_B2_sum;
52          wire[num_bits-1 : 0] s_C0,s_C1,s_C2;
53          wire[num_bits-1 : 0] multA0_output, multA1_output, multA2_output,multA_inverted_output;
54          wire[num_bits-1 : 0] Cx_D0_sum, Cx_D1_sum, Cx_D2_sum;
55          wire[num_bits-1 : 0] multC0_output, multC1_output;
56
57          always@(posedge clk) begin
58              if(control) begin
59                                      if(I_A3 != 4'b0000) begin
60                          A0 <= I_A0;
61                          A1 <= I_A1;
62                          A2 <= I_A2;
63                          A3 <= I_A3;
64                          I_A3_zero_flag <= 1'b0;
65                                      end
66                                      else begin
67                          A0 <= 4'b0000;
68                          A1 <= I_A0;
69                          A2 <= I_A1;
70                          A3 <= I_A2;
71                          I_A3_zero_flag <= 1'b1;
72                                      end
73              end
74              else if (control == 1'b0 & sel == 1'b1) begin
75                  A0 <= 4'b0000;
76                  A1 <= A0_B0_sum;
77                  A2 <= A1_B1_sum;
```

```
78                  A3 <= A2_B2_sum;
79              end
80              if (control_2 == 2'b00) begin
81                  B0 <=  4'b0000;
82                  B1 <=  A0_B0_sum;
83                  B2 <=  A1_B1_sum;
84                  B3 <=  A2_B2_sum;
85
86              end
87              else if (control_2 == 2'b01) begin
88                  B0 <=  A0;
89                  B1 <=  A1;
90                  B2 <=  A2;
91                  B3 <=  A3;
92              end
93              else if (control_2 == 2'b10) begin
94                  B0 <=  I_B0;
95                  B1 <=  I_B1;
96                  B2 <=  I_B2;
97                  B3 <=  I_B3;
98              end
99
100             else begin
101                B0 <= B0;
102                B1 <= B1;
103                B2 <= B2;
104                B3 <= B3;
105             end
106
107             if(control_3 == 1'b1) begin
108                 C0 <= I_C0;
109                 C1 <= I_C1;
110             end
111             else if (control_3 == 1'b0 & sel_2 == 1'b1) begin
112                 C0 <= Cx_D0_sum;
113                 C1 <= Cx_D1_sum;
114             end
115
116             if(control_4 == 2'b00) begin
117                 D0 <=  Cx_D0_sum;
118                 D1 <=  Cx_D1_sum;
119                 D2 <=  Cx_D2_sum;
120             end
121
122             else if(control_4 == 2'b01) begin
123                 D0 <=  C0;
124                 D1 <=  C1;
125                 D2 <=  4'b0000;
126             end
127
128             else if(control_4 == 2'b10) begin
129                 D0 <=  I_D0;
130                 D1 <=  I_D1;
131                 D2 <=  I_D2;
132             end
133
134             else if(control_4 == 2'b11) begin
135                 D0 <= D0;
136                 D1 <= D1;
137                 D2 <= D2;
138             end
139         end
140
141         assign O_mult_0 = A0_B0_sum;
142         assign O_mult_1 = A1_B1_sum;
143         assign O_mult_2 = A2_B2_sum;
144         assign constant = Cx_D0_sum;
145         assign O_div_1 = Cx_D1_sum;
146         assign O_div_2 = Cx_D2_sum;
147 Inverter
148 u_inverter(
149     .number             (A3),
150     .num_inverted       (A_Inverted)
151     );
152
153 Mod2_Adder
154 u_mod2_AdderB0(
155     .I_A             (B0),
156     .I_B  (multA0_output),
157     .O_sum    (A0_B0_sum)
158     );
159
160 Mod2_Adder
161 u_mod2_AdderB1(
162     .I_A               (B1),
163     .I_B   (multA1_output),
164     .O_sum     (A1_B1_sum)
165     );
166
167 Mod2_Adder
168 u_mod2_AdderB2(
169     .I_A               (B2),
170     .I_B   (multA2_output),
171     .O_sum     (A2_B2_sum)
172     );
173
174 Mod2_Adder
175 u_mod2_AdderD0(
```

```
176         .I_A              (D0),
177         .I_B            (s_C0),
178         .O_sum    (Cx_D0_sum)
179         );
180
181     Mod2_Adder
182     u_mod2_AdderD1(
183         .I_A               (D1),
184         .I_B             (s_C1),
185         .O_sum     (Cx_D1_sum)
186         );
187
188     Mod2_Adder
189     u_mod2_AdderD2(
190         .I_A               (D2),
191         .I_B             (s_C2),
192         .O_sum     (Cx_D2_sum)
193         );
194
195     Multiplier
196     u_Multiplier_A0(
197         .I_A               (A0),
198         .I_B   (multA_inverted_output),
199         .rst               (rst),
200         .O_mult (multA0_output)
201         );
202     //Terminar de conectar multiplicadores
203     Multiplier
204     u_Multiplier_A1(
205         .I_A               (A1),
206         .I_B    (multA_inverted_output),
207         .rst               (rst),
208         .O_mult (multA1_output)
209         );
210
211     Multiplier
212     u_Multiplier_A2(
213         .I_A               (A2),
214         .I_B   (multA_inverted_output),
215         .rst               (rst),
216         .O_mult (multA2_output)
217         );
218
219     Multiplier
220     u_Multiplier_Inverter(
221         .I_A     (A_Inverted),
222         .I_B               (B3),
223         .rst               (rst),
224         .O_mult (multA_inverted_output)
225         );
226
227     Multiplier
228     u_Multiplier_C0(
229         .I_A               (C0),
230         .I_B   (multA_inverted_output),
231         .rst               (rst),
232         .O_mult (multC0_output)
233         );
234     //Terminar de conectar multiplicadores
235     Multiplier
236     u_Multiplier_C1(
237         .I_A               (C1),
238         .I_B (multA_inverted_output),
239         .rst               (rst),
240         .O_mult (multC1_output)
241         );
242
243     Shifter
244     u_Shifter(
245             .clk         (clk),
246             .enable      (enable_shift),
247             .I_C0        (multC0_output),
248             .I_C1        (multC1_output),
249             .O_C0        (s_C0),
250             .O_C1        (s_C1),
251             .O_C2        (s_C2)
252             );
253     endmodule
```

B.1.10. Decodificador Reed Solomon - Búsqueda de Chien

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
5   //
6   // Create Date: 16.02.2021 22:50:20
7   // Design Name:
8   // Module Name: Chien_Search
9   // Project Name:
10  // Target Devices:
11  // Tool Versions:
12  // Description:
13  //
14  // Dependencies:
15  //
```

```
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////
21
22
23  module Chien_Search
24     #(
25          parameter num_bits = 4
26      )
27      (
28          input [num_bits-1 : 0] I_alfa0,I_alfa1,I_alfa2,
29          input                  clk,rst,sel,
30          output [num_bits-1 : 0] O_errorpos, O_terminointermedio
31      );
32
33      reg [num_bits-1 : 0]      reg_d_0;
34      reg [num_bits-1 : 0]      reg_d_1;
35      reg [num_bits-1 : 0]      reg_d_2;
36      wire [num_bits-1 : 0]     mux0_out;
37      wire [num_bits-1 : 0]     mux1_out;
38      wire [num_bits-1 : 0]     mux2_out;
39      wire [num_bits-1 : 0]     sum1_out;
40      wire [num_bits-1 : 0]     sum2_out;
41      wire [num_bits-1 : 0]     mult0_output;
42      wire [num_bits-1 : 0]     mult1_output;
43      wire [num_bits-1 : 0]     mult2_output;
44
45      //Voy a alternar con el mux despus de la primer iteracion creo (el flag habilita la
46      //realimentacin)
47      always @(posedge clk) begin
48          if (~rst) begin
49              reg_d_0 <= 4'b0000;
50              reg_d_1 <= 4'b0000;
51              reg_d_2 <= 4'b0000;
52          end
53          else begin
54              reg_d_0 <= mult0_output;
55              reg_d_1 <= mult1_output;
56              reg_d_2 <= mult2_output;
57          end
58      end
59
60      assign mux0_out = (sel) ? I_alfa0 : reg_d_0;
61      assign mux1_out = (sel) ? I_alfa1 : reg_d_1;
62      assign mux2_out = (sel) ? I_alfa2 : reg_d_2;
63
64      assign O_errorpos = (sum2_out) ? 4'b0000 : 4'b1111;
65      assign O_terminointermedio = reg_d_1;
66
67  Mod2_Adder
68  u_mod2_Adder_1(
69      .I_A    (reg_d_0),
70      .I_B  (reg_d_1),
71      .O_sum (sum1_out)
72      );
73
74  Mod2_Adder
75  u_mod2_Adder_2(
76      .I_A    (sum1_out),
77      .I_B  (reg_d_2),
78      .O_sum (sum2_out)
79      );
80
81  Multiplier
82  u_Multiplier_0(
83      .I_A       (mux0_out),
84      .I_B             (4'b0001),
85      .rst           (rst),
86      .O_mult (mult0_output)
87      );
88
89  Multiplier
90  u_Multiplier_1(
91      .I_A       (mux1_out),
92      .I_B             (4'b0010),
93      .rst           (rst),
94      .O_mult (mult1_output)
95      );
96
97  Multiplier
98  u_Multiplier_2(
99      .I_A       (mux2_out),
100     .I_B             (4'b0100),
101     .rst           (rst),
102     .O_mult (mult2_output)
103     );
104 endmodule
```

B.1.11. Decodificador Reed Solomon - Algoritmo de Forney

```
1   `timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   // Company:
4   // Engineer:
```

```
5   //
6   // Create Date: 17.02.2021 23:02:55
7   // Design Name:
8   // Module Name: Forney_Method
9   // Project Name:
10  // Target Devices:
11  // Tool Versions:
12  // Description:
13  //
14  // Dependencies:
15  //
16  // Revision:
17  // Revision 0.01 - File Created
18  // Additional Comments:
19  //
20  //////////////////////////////////////////////////////////////////////////////////


23  module Forney_Method
24      #(
25          parameter num_bits = 4
26      )
27      (   input                   clk,rst,sel,
28          input [num_bits-1 : 0] I_omega0,I_omega1,I_terminointermedio,I_errorpos,
29          output [num_bits-1 : 0] O_errorvalue
30      );
31
32      reg [num_bits-1 : 0]      reg_d_0;
33      reg [num_bits-1 : 0]      reg_d_1;
34      reg [num_bits-1 : 0]      reg_error;
35      wire [num_bits-1 : 0]     mux0_out;
36      wire [num_bits-1 : 0]     mux1_out;
37      wire [num_bits-1 : 0]     sum1_out;
38      wire [num_bits-1 : 0]     mult0_output;
39      wire [num_bits-1 : 0]     mult1_output;
40      wire [num_bits-1 : 0]     mult2_output;
41      wire [num_bits-1 : 0]     num_inverted;
42      wire [num_bits-1 : 0]     and_output;
43
44      //Voy a alternar con el mux despus de la primer iteracion creo (el flag habilita la
45      //realimentacin)
46
47      assign mux0_out = (sel) ? I_omega0 : reg_d_0;
48      assign mux1_out = (sel) ? I_omega1 : reg_d_1;
49
50      always @(posedge clk) begin
51          if (~rst) begin
52              reg_d_0 <= 4'b0000;
53              reg_d_1 <= 4'b0000;
54          end
55          else begin
56              reg_d_0 <= mult0_output;
57              reg_d_1 <= mult1_output;
58          end
59      end
60      //Le agrego este registro porque sino me queda desfasado por los otros registros
61      always @(posedge clk) begin
62          reg_error <= I_errorpos;
63      end
64
65      assign and_output[num_bits-1] = mult2_output[num_bits-1] & I_errorpos[num_bits-1];
66      assign and_output[num_bits-2] = mult2_output[num_bits-2] & I_errorpos[num_bits-2];
67      assign and_output[num_bits-3] = mult2_output[num_bits-3] & I_errorpos[num_bits-3];
68      assign and_output[num_bits-4] = mult2_output[num_bits-4] & I_errorpos[num_bits-4];
69
70      assign O_errorvalue = and_output;
71
72  Mod2_Adder
73  u_mod2_Adder_1(
74      .I_A    (reg_d_0),
75      .I_B    (reg_d_1),
76      .O_sum (sum1_out)
77      );
78
79  Multiplier
80  u_Multiplier_0(
81      .I_A            (mux0_out),
82      .I_B                (4'b0001),
83      .rst               (rst),
84      .O_mult         (mult0_output)
85      );
86
87  Multiplier
88  u_Multiplier_1(
89      .I_A       (mux1_out),
90      .I_B            (4'b0010),
91      .rst            (rst),
92      .O_mult (mult1_output)
93      );
94
95  Multiplier
96  u_Multiplier_2(
97      .I_A            (sum1_out),
98      .I_B            (num_inverted),
99      .rst            (rst),
100     .O_mult         (mult2_output)
101     );
102
```

```
103 Inverter
104 u_inverter(
105     .number          (I_terminointermedio),
106     .num_inverted    (num_inverted)
107     );
108 endmodule
```

Bibliografía

[1] Claude Shannon y Warren Weaver. «The Mathematical Theory of Communication». En: (), pág. 131.

[2] *The Ultimate Guide to FPGA Test Benches*. Dic. de 2020.

[3] Ray C C Cheung y John D Villasenor. «Hardware Generation of Arbitrary Random Number Distributions From Uniform Distributions Via the Inversion Method». En: 15.8 (2007), pág. 11.

[4] Dong-U Lee y col. «Optimizing Hardware Function Evaluation». En: *IEEE Trans. Comput.* 54.12 (dic. de 2005), págs. 1520-1531. ISSN: 0018-9340. DOI: 10.1109/TC.2005.201.

[5] J. R. Rice. *Rice J.R. - The approximation of functions. Linear theory. Volume 1-AW (1964)*. 1st ed. Massachusetts: Adisson-Wesley, 1964.

[6] Dong-U Lee y col. «Hierarchical Segmentation for Hardware Function Evaluation». En: *IEEE Trans. VLSI Syst.* 17.1 (ene. de 2009), págs. 103-116. ISSN: 1063-8210, 1557-9999. DOI: 10.1109/TVLSI.2008.2003165.

[7] Oliver Pretzel. *Error-Correcting Codes and Finite Fields*. Oxford Applied Mathematics and Computing Science Series. Oxford : New York: Clarendon Press ; Oxford University Press, 1992. ISBN: 978-0-19-859678-3.

[8] R. Chien. «Cyclic Decoding Procedures for Bose- Chaudhuri-Hocquenghem Codes». En: *IEEE Trans. Inform. Theory* 10.4 (oct. de 1964), págs. 357-363. ISSN: 0018-9448. DOI: 10.1109/TIT.1964.1053699.

[9] Pavel A Rahman. «Formal Derivative». En: *Journal of Physics* (2019), pág. 7.

[10] J. M. Muller. *Elementary Functions: Algorithms and Implementation*. 2nd ed. Boston: Birkhäuser, 2006. ISBN: 978-0-8176-4372-0.

[11] L. Veidinger. «On the Numerical Determination of the Best Approximations in the Chebyshev Sense». En: *Numer. Math.* 2.1 (dic. de 1960), págs. 99-105. ISSN: 0029-599X, 0945-3245. DOI: 10.1007/BF01386215.

[12] «Cyclone V Hard Processor System Technical Reference Manual». En: (), pág. 3501.

[13] Rene Beuchat y Sahand Kashani-Akhavan. «SoC-FPGA Design Guide [DE0-Nano-SoC Edition]». En: (), pág. 100.

[14] *Linux on HPS*. https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_peripherials/

[15] *Embedded Linux Beginners Guide*. https://rocketboards.org/foswiki/Documentation/EmbeddedLinu

[16] Pierre L'Ecuyer. «MAXIMALLY EQUIDISTRIBUTED COMBINED TAUSWORTHE GENERATORS». En: *Mathematics of Computation* 65.213 (ene. de 1996), págs. 203-213.

[17] Pierre L'Ecuyer. «Tables of Maximally Equidistributed Combined LFSR Generators». En: *Math. Comp.* 68.225 (ene. de 1999), págs. 261-270. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-99-01039-X.

[18] *FIFO Buffer Module with Watermarks (Verilog and VHDL) - eewiki / Logic*. https://forum.digikey.com/t/f buffer-module-with-watermarks-verilog-and-vhdl/13182. Mar. de 2021.

[19] Mathuranathan Viswanathan. «SIMULATION OF DIGITAL COMMUNICATION SYSTEMS USING MATLAB». En: (), pág. 258.

[20] «Intel® Quartus® Prime Standard Edition Handbook Volume 3 Verification». En: 3 (), pág. 76.

[21] Guangxi Liu. «Gaussian Noise Generator Core Specification». En: (), pág. 16.

[22] Sanjeev Kumar y Ragini Gupta. «Bit Error Rate Analysis of ReedSolomon Code for Efficient Communication System». En: *IJCA* 30.12 (sep. de 2011), págs. 11-15. ISSN: 09758887. DOI: 10.5120/3710-5174.

[23] Kenny Chung Chung Wai. «FPGA Implementation of Reed Solomon Codec for 40Gbps Forward Error Correction in Optical Networks». En: (), pág. 75.