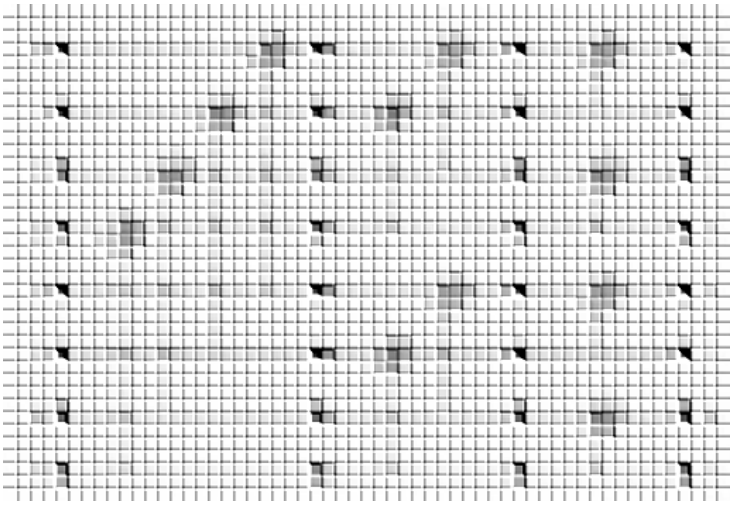
Universidad Nacional de Mar del Plata

Facultad de Ingeniería

Tesis de grado

Ingeniería Electrónica

Implementación en FPGA de códigos polares para corrección de errores en comunicaciones digitales



Autor:
Federico Guillermo Krasser

Directores:
Mg. Ing. Mónica Cristina Liberatori
Ing. Leonardo Oscar Coppolillo

Diciembre de 2018

Índice general

Índice de figuras

Índice de cuadros

Glosario

ALM *Adaptive Logic Module*. Denominación de los módulos de celdas lógicas utilizadas en FPGA del fabricante Intel/Altera. 21

AMBA *Advanced Microcontroller Bus Architecture*. 50

ARQ *Automatic Repeat Query*. Requerimiento de repetición automático. 15

ASIC *Application Specific Integrated Circuit*. Circuitos integrados que implementan una funcionalidad específica. 21–24, 56

AWGN *Additive White Gaussian Noise*. Ruido blanco aditivo gaussiano. 17, 24, 53, 59, 60

AXI *Advanced Extensible Interface*. 50

BEC *Binary Erasure Channel*. Canal binario de borrado. 17

BER *Bit Error Rate*. Tasa de error por bit. 59

BPSK *Binary phase-shift keying*. Modulación binaria por desplazamiento de fase. 59

BSC *Binary Symmetric Channel*. Canal binario simétrico. 17, 53, 62

capacidad (de un canal de comunicaciones) Límite superior de la velocidad de transmisión por un canal, asegurando una probabilidad de error arbitrariamente baja con el empleo de una codificación adecuada. 16

codificación sistemática Tipo de codificación para control de errores en la que el mensaje original aparece en orden en el mensaje codificado. 24

CRC *Cyclic Redundancy Check*. Código detector de errores del tipo cíclico. 23, 24

DSP *Digital Signal Processing*. Procesamiento digital de señales. 21

FEC *Forward Error Correction*. Corrección de errores hacia adelante. 15, 16, 23

FER *Frame Error Rate*. Tasa de error por bloque. 59

FFT *Fast Fourier Transform*. Transformada rápida de Fourier. 19

flip-flop Dispositivo básico de la electrónica digital, que permite almacenar un bit de información. 21, 40

FPGA *Field-programmable Gate Array*. Arreglo de compuertas programables. 13, 21–26, 33, 37, 40, 42, 47, 48, 50, 56, 58, 59, 65

FSM *Finite-state Machine*. Máquina de estado finito. 50, 51, 57

GHRD *Golden Reference Hardware Design*. Diseño de referencia provisto por el fabricante de una placa de desarrollo, para facilitar la tarea del programador o diseñador. 54

GPIO *General-Purpose Input/Output*. Entrada/Salida de Propósito General. Pines de funciones configurables en un circuito integrado. 47

HDL *Hardware Description Language*. Lenguaje de descripción de hardware. 22

HPS *Hard Processor System*. Procesador implementado con hardware dedicado y contenido en el mismo encapsulado de una FPGA. Se contrapone a los procesadores *soft* o *Soft Processors* que son implementados por la lógica reconfigurable de las FPGA. 47, 48, 50, 57–59

IDE *Integrated Development Environment*. Entorno gráfico que provee a un diseñador de un conjunto de herramientas necesarias para desarrollar un sistema. 22, 25, 53

LAC Laboratorio de Comunicaciones, Departamento de Electrónica y Computación, Facultad de Ingeniería, UNMDP. 25

LDPC *Low-density parity-check*. Comprobación de paridad de baja densidad. Clase de códigos correctores de errores. 23, 24, 32

LLR *Log-likelihood Ratio*. Relación de similitud logarítmica. 23–25, 27, 28, 32, 33, 39, 42, 43, 48, 53, 59

LR *Likelihood Ratio*. Relación de similitud. 19, 20, 23, 32

LSB *Least Significant Bit*. Bit menos significativo en una representación binaria. 36

LUT *Lookup Table*. Tabla de búsqueda. Memoria en la que se almacenan generalmente los resultados de funciones. Permite calcular funciones estableciendo a partir de sus entradas las direcciones de lectura. 21

MPE *Merged Processing Element*. Elemento o módulo de procesamiento combinado, que permite implementar más de una función. 23, 27, 28, 33, 37, 38, 43, 62

MSB *Most Significant Bit*. Bit más significativo en una representación binaria. 36

MUX Multiplexor. 36

netlist Lista de conexiones entre distintos elementos básicos de un circuito electrónico, usada en herramientas de desarrollo y simulación de circuitos. 22

OS *Operating System*. Sistema Operativo. 49, 52

PE *Processing Element*. Elemento o módulo de procesamiento de datos. 23

pipeline Tipo de segmentación de un sistema digital para aumentar la velocidad de operación, procesando simultáneamente varias partes de uno o varios grupos de datos. 23, 24, 26

PLL *Phase-locked Loop*. Lazo de control de fase. Permite generar señales de reloj de distintas frecuencias. 21, 54

RAM *Random-access Memory*. Memoria de acceso aleatorio. 49, 51, 54

ROM *Read-only Memory*. Memoria de sólo lectura. 54

RTL *Register-transfer Level*. Nivel de abstracción en el que un sistema digital se describe mediante componentes lógicos básicos y sus interconexiones. 5, 22, 26, 29, 41

SCD *Successive Cancellation Decoding*. Decodificación por cancelaciones sucesivas. 18, 23, 24, 26, 37, 43

skew Fenómeno indeseado en un circuito sincrónico, por el que una misma señal de reloj llega a diferentes componentes en distintos instantes. 21

SM Signo y magnitud. 32, 60

SNR *Signal to Noise Ratio*. Relación señal a ruido. 60, 64

SOC *System On Chip*. Circuito integrado que contiene un gran número de componentes de un sistema como microprocesadores, memoria y periféricos. 47

tasa de código Relación entre la cantidad de bits originales de información y la cantidad de bits del mensaje codificado. Representado por $Rc = {}^{K}/_{N}$. 16, 25

throughput Velocidad de procesamiento de datos. Generalmente expresada como TP, en unidades de bits por segundo (*bps*). 23, 24, 41

UART *Universal Asynchronous Receiver-Transmitter*. Transmisor-Receptor Asíncrono Universal. Dispositivo de comunicación dúplex en serie. 48

USB *Universal Serial Bus*. Bus de comunicación serie universal. 47

VHDL *Very High Speed Integrated Circuit Hardware Description Language*. Lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad. 22, 24, 28, 29, 40, 44

Capítulo 1

Introducción

En su trabajo de 1948, Claude Shannon introdujo algunos conceptos fundamentales de lo que se conoce como teoría de la información. Su trabajo propició el desarrollo vertiginoso de las comunicaciones digitales durante el resto del siglo XX y sentó las bases para el futuro.

Como uno de los pilares de esta teoría, los códigos de control de errores han permitido comunicaciones cada vez más confiables y rápidas. Su aplicación se diversificó y se materializó en tecnologías como las comunicaciones satelitales, la telefonía celular, Internet y los dispositivos de almacenamiento de datos, entre otros.

Apareciendo en el año 2009, los códigos polares han cobrado notoriedad por ser los primeros códigos que alcanzan el límite de capacidad de un canal. Otras características interesantes son el poseer un método de construcción explícito y una estructura regular. Por estos motivos en los últimos años los investigadores han estado buscando implementaciones rápidas y eficientes, llevando la teoría a la práctica efectiva. El último logro de los códigos polares es haber sido seleccionados para formar parte del estándar de telefonía celular 5G, cuyo despliegue debería efectuarse durante los años entrantes.

Las aplicaciones prácticas de los sistemas de comunicaciones y de control de errores están normalmente basadas en *hardware*. Durante los proyectos de desarrollo es generalmente utilizada la tecnología de arreglos de compuertas programables (FPGA). Ésta aparece hace aproximadamente 40 años y puede simplificarse su concepto como hardware que puede ser reconfigurado múltiples veces, habiendo salido ya de la fábrica. Su programación se hace en lenguajes de descripción de hardware. Estos lenguajes, de alto nivel, facilitan la tarea al diseñador, y acortan plazos y costos de desarrollo. El diseñador puede además utilizar la tecnología como paso intermedio en el camino hacia circuitos integrados dedicados (ASIC), o utilizarla como banco de simulación y pruebas de sistemas más complejos. La tendencia moderna se está manifestando en el uso de FPGA en combinación con procesadores tradicionales, actuando como aceleradores en tareas altamente demandantes y paralelizables, donde la velocidad de cálculo impera.

En este trabajo se alinean muchas de las tecnologías hasta aquí nombradas. Es el resultado de un proceso para llevar un tema abstracto a un uso práctico. En este proyecto se trabajó sobre la implementación mediante FPGA de códigos polares para corrección de errores. Como parte del mismo se desarrolla también un sistema automatizado de pruebas, cuya utilidad se extiende más allá de una aplicación particular de códigos correctores de errores.

A modo de guía, el presente informe se divide en seis capítulos, relacionados en gran medida con la cronología del proyecto. Los primeros capítulos de este informe presentan los principales conceptos referidos a la corrección de errores y los códigos polares. Posteriormente se tratan los aspectos relacionados al diseño y concepción del sistema en FPGA. El cuarto capítulo está dedicado al sistema desarrollado para automatizar la verificación de funcionamiento mediante la

simulación de canales de comunicaciones en sistemas embebidos. En el quinto capítulo se exponen los resultados obtenidos en la implementación práctica de códigos y canales ya previamente estudiados mediante programas de simulación. El último capítulo es dedicado a las conclusiones y a las consideraciones y mejoras que podrían aconsejarse para posteriores ampliaciones de este proyecto, o en proyectos similares. Finalmente se encuentran los anexos; en ellos se incluyen partes importantes del diseño como los códigos usados durante la implementación y líneas de código particularmente relevantes.

Capítulo 2

Aspectos teóricos

2.1. Canales de comunicaciones y codificación

Un sistema de comunicaciones puede modelizarse de forma general como en la (Fig. 2.1) [1]. Un emisor es una fuente de información que se desea que sea recibida por 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 una fuente de 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.

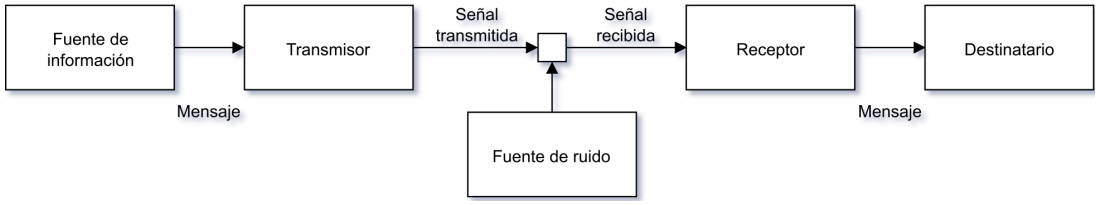


Figura 2.1: Sistema de comunicaciones generalizado

Para lograr la transmisión eficaz de información a través del canal, los bloques transmisor y receptor deben generalmente realizar operaciones de control de errores. Los mensajes de longitud K en bits deben tratarse de forma especial. Esto implica inicialmente el agregado de redundancia, aumentando la longitud de los mensajes a N bits. En segundo lugar, sobre los bits se realizan operaciones siguiendo un esquema codificador. El resultado es un mensaje codificado de N bits, que puede ingresar a un modulador para ser finalmente enviado por el canal de comunicaciones. En el lado receptor, el bloque decodificador busca obtener los K bits de mensaje original, a partir de los N bits sujetos al ruido.

Con este fin dicho bloque tendría dos opciones. En primer lugar podría simplemente detectar la existencia de errores, y solicitar al transmisor el reenvío del mensaje, con la esperanza de recibir esta vez un mensaje correcto. 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. Este tipo de esquema o protocolo es conocido como Requerimiento de Reconocimiento Automático (ARQ). En segundo lugar, el decodificador podría intentar deducir el mensaje original directamente a partir del mensaje recibido. En este caso el decodificador estaría trabajando en el modo denominado de corrección de errores, sin la necesidad de un canal de retorno.

Los sistemas de control de errores denominados de corrección de errores hacia adelante (FEC) pertecenen a la segunda opción. En ellos el decodificador busca corregir 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. Si, en cambio, un mensaje tuviera más errores que los que el código es capaz de corregir, el código colapsaría; el destinatario no tendría entonces forma de conocer en forma íntegra 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 las probabilidades de encontrar errores más bajas.

2.2. Códigos polares

Estos códigos FEC modernos pertenecen a la clase de los llamados códigos de bloque. En dicha clase todos los bits del mensaje codificado deben ser recibidos antes de poder iniciar su decodificación. Arikan demostró que los códigos polares alcanzan la capacidad de un canal binario simétrico (BSC), para longitudes de mensaje tendientes a infinito [2]. La capacidad de un canal es el límite superior de la velocidad de transmisión por un canal, asegurando una probabilidad de error arbitrariamente baja con una codificación apropiada [1]. La capacidad depende del tipo y ancho de banda del canal, así como también de las potencias de la señal y del ruido. En iguales condiciones, los canales más ruidosos son menos capaces. Un aspecto a destacar de los códigos polares es que son los primeros cuya habilidad de alcanzar la capacidad del canal es matemáticamente demostrable.

2.2.1. Polarización

El nombre de estos códigos proviene del efecto de polarización de un canal de comunicaciones. La polarización es una operación recursiva, en la que se aplican etapas de combinación y separación de canales. Si se utiliza sobre $N = 2^n$ copias independientes de un canal W, el resultado es N canales sintéticos polarizados. Cuando N es muy grande, los canales sintéticos tienden a presentar capacidades máximas o nulas.

A continuación se ilustran los procesos de polarización (Fig. 2.2). En los canales se tiene $W_N : X^N \to Y^N$, simbolizando la utilización de vectores de N bits, y se trabaja en el campo de Galois $\mathbb{F}^2$. En este campo las operaciones suma son operaciones or-exclusiva (XOR), representadas por los símbolos $\oplus$. En la figura los bloques R_N aplican la permutación denominada *reverse-shuffle*, que separa los bits de índices pares de los impares y los reagrupa en orden.

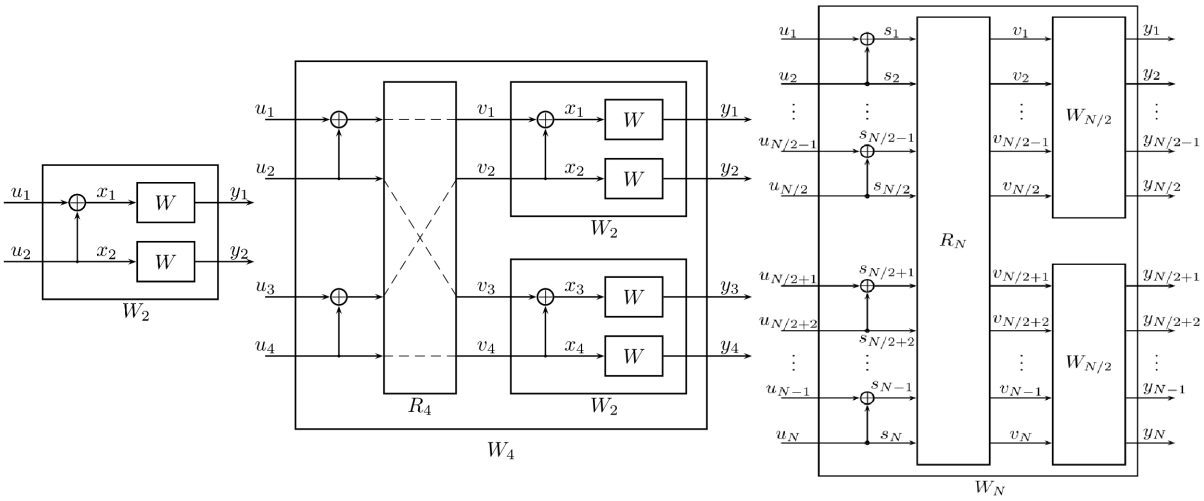


Figura 2.2: Canales W_N, construidos recursivamente a partir de canales $W_{N/2}$

2.2.2. Construcción

La polarización permite obtener canales casi perfectamente capaces. Es deseable entonces conocer cuáles son estos canales, y así enviar la información por los mismos. Esto puede lograrse con un procedimiento conocido como construcción del código polar, y es crítico para obtener la mejor performance para longitudes de código finitas. Para operar correctamente sobre un mensaje el codificador y el decodificador polar deben utilizar el mismo código.

En base a lo anterior puede decirse que un código polar es especificado con tres parámetros (N, K, F):

- La cantidad de bits del mensaje codificado, N.
- La cantidad de bits del mensaje sin codificar, K.
- El conjunto $\mathbb{F}$ de los $N - K$ índices de los canales menos capaces, denominados *frozen bits*.

A modo de ejemplo, un código polar puede ser el código $(8, 4, \{0, 1, 2, 5\})$. Éste puede también ser interpretado como $(8, 4, 11101000)$, empleando notación binaria y asociando la posición de cada 0 al índice de un frozen bit. Para longitudes de código mayores será más fácil visualizar los frozen bits si se emplea notación hexadecimal.

Los códigos polares no son universales. Para maximizar la performance cada código debe construirse en función de los parámetros del canal a utilizar (estado de diseño). En igualdad de condiciones, un código diseñado para un canal BSC con una probabilidad de transición $p = 0,5$ será diferente al construido para un canal de ruido blanco aditivo gaussiano (AWGN) con $SNR = 1$ dB.

Existen diversos métodos para construir códigos polares. Arikan propuso inicialmente, para canales binarios simétricos, un algoritmo recursivo sencillo basado en la evolución de los límites de Bhattacharyya a partir de un valor inicial [2]. Dicho valor inicial es función del tipo de canal y del estado de diseño, y puede utilizarse para otros canales como el binario de borrado (BEC) o el AWGN [3]. Para canales gaussianos, el algoritmo basado en los parámetros de Bhattacharyya encuentra códigos cuya performance es tan buena como la de otros métodos más complejos [4]. Por lo tanto será el empleado a lo largo de este trabajo para obtener todos los códigos que se utilicen.

Finalmente, habiendo construido un código polar, los valores de los K bits del mensaje original se posicionan de acuerdo a los elementos del conjunto $\mathbb{F}^c$, índices de los bits que no son frozen. Para los bits pertenecientes a $\mathbb{F}$, los valores asociados podrán disponerse de cualquier forma (estos bits se *congelan*), siempre que se respete la concordancia entre el codificador y el decodificador. En este trabajo se utilizan ceros como frozen bits, conforme a lo habitual en la bibliografía sobre códigos polares. Con los bits anteriores ordenados en sus posiciones correspondientes se obtiene el mensaje que incluye los frozen bits. Este mensaje es el que se dispone a la entrada del codificador polar, que se presenta en la siguiente sección de este trabajo.

2.2.3. Codificación

En los códigos de bloque la codificación puede considerarse como una operación de multiplicación vectorial tal que $\mathbf{x} = \mathbf{u}.\mathbf{G}$, donde:

- $\mathbf{u}$ es un vector fila de K elementos, correspondiente al mensaje sin codificar.
- $\mathbf{G}$ es una matriz de $K \times N$ elementos, llamada matriz generadora.
- $\mathbf{x}$ es un vector fila de N elementos, correspondiente al mensaje codificado.

Para los códigos polares se cumple $N = 2^n$ ($n \in \mathbb{N}$), $\mathbf{u}$ es un vector de N elementos, con K elementos correspondientes al mensaje original, y $N - K$ correspondientes a los frozen bits. La matriz generadora es una matriz cuadrada de dimensión $N \times N$ tal que $\mathbf{G} = \mathbf{F}^{\otimes \mathbf{n}}.\mathbf{B}$. Esta matriz se construye en parte aplicando la n-ésima potencia de Kronecker a la matriz $\mathbf{F}^{\otimes \mathbf{1}} = \left(\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right)$. Esta matriz $\mathbf{F}$ es el denominado núcleo o *kernel* polar, y se representa gráficamente en la Fig. 2.3.

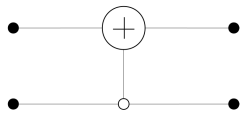


Figura 2.3: Representación gráfica del kernel polar

La operación de Kronecker es recursiva, ya que se cumple:

$$\mathbf{F}^{\otimes \mathbf{n}} = \begin{pmatrix} \mathbf{F}^{\otimes (\mathbf{n-1})} & 0 \\ \mathbf{F}^{\otimes (\mathbf{n-1})} & \mathbf{F}^{\otimes (\mathbf{n-1})} \end{pmatrix} \tag{2.1}$$

Por ejemplo, para $N = 8$:

$$\mathbf{F}^{\otimes \mathbf{3}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{2.2}$$

Por otro lado, $\mathbf{B}$ es un operador que realiza la permutación denominada *bit-reverse*. Esta permutación reordena los bits en función de la notación binaria de su índice. Si se expresa cada índice como dígitos binarios, tal que $b = b_0 b_1 \dots b_{n-1}$, usando esta operación se obtiene $b_{br} = b_{n-1} b_{n-2} \dots b_0$. Por ejemplo, para un vector $\mathbf{v} = (v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$, la aplicación de $\mathbf{B}$ produce el vector $\mathbf{v}_{br} = (v_0, v_4, v_2, v_6, v_1, v_5, v_3, v_7)$. Esta permutación es importante, ya que al aplicarse en la salida del codificador (o en la entrada al decodificador) permite estimar los bits de $\mathbf{u}$ en orden natural.

En base a lo anterior, se expresa gráficamente el codificador polar para $N = 8$ (Fig. 2.4). En el grafo los $s_{i,j} \in \{0, 1\}$ son nodos correspondientes a codificaciones parciales, y se conocen como sumas parciales. En un codificador polar hay $(1 + log_2 N) \times N$ nodos de este tipo, y su localización es importante para el proceso de decodificación, como se verá en la próxima sección.

2.2.4. Decodificación – el algoritmo de cancelaciones sucesivas

Tras ser modulado y transmitido por el canal ruidoso, el mensaje codificado $\mathbf{x}$ llega al sistema receptor. Éste es demodulado y entregado al decodificador como un mensaje $\mathbf{y}$. El decodificador polar tiene la tarea de estimar los N bits del mensaje original $\mathbf{u}$, que se encuentran ordenados en función de los frozen bits. Existen distintas formas para lograrlo. La forma clásica es aplicar el algoritmo de decodificación por cancelaciones sucesivas (SCD), presentado por Arikan. Se trata de un procedimiento que secuencialmente obtiene estimaciones $\hat{u}_i$ de los bits del mensaje original. Habiendo utilizado la permutación bit-reverse durante la codificación, estos bits son obtenidos en orden natural, comenzando por $\hat{u}_0$ y terminando por $\hat{u}_{N-1}$.

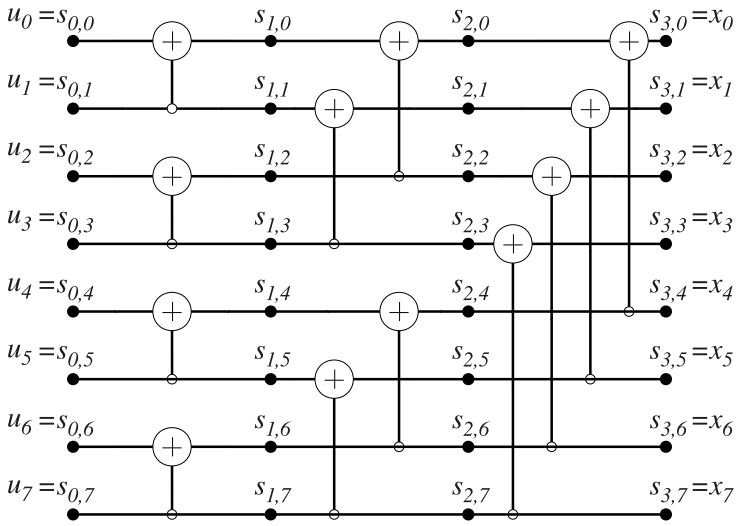


Figura 2.4: Codificador polar para $N = 8$

Para estimar cada $\hat{u}_i$ el decodificador utiliza la siguiente regla de decisión:

$$\hat{u}_i = \begin{cases} 0, & \text{si } i \in \mathbb{F} \\ 0, & \text{si } \frac{\Pr(\mathbf{y}, \hat{u}_0^{i-1} | u_i = 0)}{\Pr(\mathbf{y}, \hat{u}_0^{i-1} | u_i = 1)} > 1 \\ 1, & \text{en otro caso} \end{cases} \tag{2.3}$$

En la expresión anterior $\Pr(\mathbf{y}, \hat{u}_0^{i-1} | u_i = b)$ es la probabilidad de haber recibido $\mathbf{y}$, haber estimado los bits anteriores como $\hat{u}_0^{i-1} = (\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{i-1})$, y tener $b \in \{0, 1\}$ como valor actual del bit decodificado. El cociente de probabilidades encontrado es conocido como relación de similitud o LR del bit $\hat{u}_i$. Es claro, a partir de la expresión anterior, que el decodificador debe conocer los frozen bits utilizados durante el proceso de codificación.

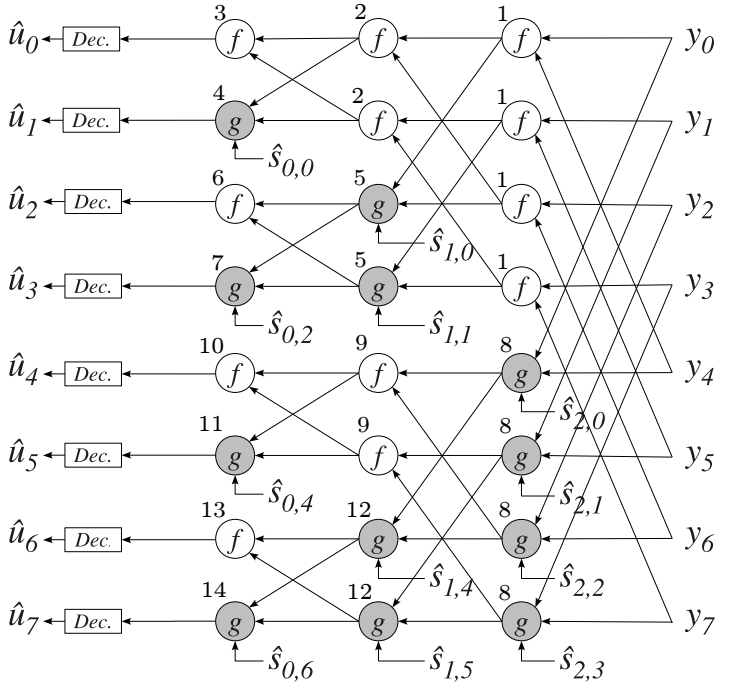


Figura 2.5: Grafo del algoritmo de cancelaciones sucesivas para $N = 8$

Para llegar al nivel de decisión, el algoritmo calcula los LR con un grafo semejante al utilizado para calcular la transformada rápida de Fourier (FFT), cuando se usa una estructura tipo

butterfly. A modo de ejemplo, se presenta el grafo de la decodificación para el caso $N = 8$ en la Fig. 2.5 [5].

Puede simplificarse el proceso de decodificación, observando que se busca seguir el camino inverso en el grafo del codificador de la Fig. 2.4. Antes de ingresar al decodificador, a partir de los y_i se obtienen los primeros LR. Éstos se propagan entonces, siguiendo las aristas de derecha a izquierda, y son procesados secuencialmente en el orden indicado en la figura, en distintos nodos f y g hasta ingresar en el extremo a los nodos de decisión. Las estimaciones finales de los $\hat{u}_i$ van apareciendo a medida que cada LR en la Ec. 2.3 está disponible. Cuando se encuentran, estas estimaciones se propagan de izquierda a derecha como sumas parciales $\hat{s}_{i,j}$. Estas sumas se corresponden por sus índices con los nodos $s_{i,j}$ que se pueden observar en la estructura del codificador.

Las funciones f y g toman las expresiones:

$$f(a,b) = \frac{1+ab}{a+b} \tag{2.4}$$

$$g(a,b,\hat{s}) = a^{1-2\hat{s}}b \tag{2.5}$$

Donde a y b son LR que llegan a los nodos siguiendo las aristas superior e inferior, respectivamente. La función g tiene además como argumento la suma parcial $\hat{s}$. La existencia de este argumento implica un límite en la secuencia de decodificación, y es uno de los factores más relevantes durante la fase de diseño de la implementación.

Capítulo 3

Implementación práctica

3.1. Tecnología de FPGA

3.1.1. Arquitectura

En contraposición con los ASIC, cuya disposición interna no puede ser modificada tras su fabricación, la versatilidad de las FPGA se apoya en la posibilidad de reconfiguración de cada chip, haciéndolo adaptable una y otra vez, a diferentes necesidades. Para lograr esta posibilidad, las FPGA cuentan con una matriz de miles de celdas lógicas de un mismo tipo, y una gran capacidad de conectarlas con otros componentes y entre sí.

Las celdas configurables permiten implementar diversas funciones lógicas mediante la 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 para registrar las salidas, multiplexores y circuitos aritméticos rápidos. Para ilustrar lo anterior, en la Fig. 3.1 se muestra la estructura de un módulo de lógica adaptable (ALM), tipo de celda lógica encontrada en las FPGA del fabricante Intel.

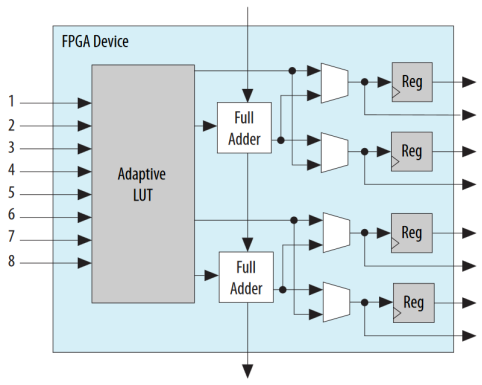


Figura 3.1: Componentes dentro de los ALM en FPGA de la familia Cyclone V

Las celdas no son los únicos componentes de las FPGA. Éstas generalmente integran además 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, minimizando el skew. Este conjunto de tecnologías permite al diseñador lograr diversas

aplicaciones, aún de alta velocidad.

3.1.2. Lenguajes de descripción de hardware

Para obtener aplicaciones en FPGA se utiliza un lenguaje de descripción de hardware, o HDL, de alto nivel, 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 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 varios de los estilos anteriores.

3.1.3. Flujo de trabajo

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

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

Los IDE tienen además 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 los mismos lenguajes y procesos de desarrollo semejantes. Los tiempos de desarrollo para FPGA se suelen medir en meses en lugar de años, y los costos de desarrollo son también inferiores que para ASIC. En definitiva, es una tecnología apropiada para el desarrollo de sistemas como los de control de errores. Por ello su aparición es habitual en la bibliografía que trata los códigos polares, como se verá en la siguiente sección.

3.2. Estudio de diseño

Los códigos polares son códigos FEC modernos. Los últimos años han aparecido numerosas publicaciones, en búsqueda de implementaciones eficientes y competitivas con las existentes para códigos ya más establecidos, como los LDPC.

Para definir las arquitecturas de codificación y decodificación a implementar durante este proyecto se estudiaron numerosas publicaciones, principalmente referidas a la implementación en hardware de códigos polares. Debido a la mayor complejidad de la tarea de decodificación respecto a la de codificación, existe una abundante bibliografía al respecto, realizándose un análisis exhaustivo de más de cuarenta algoritmos e implementaciones de decodificación. Algunas de ellas se destacan a continuación por su relevancia en el ámbito de los códigos polares o su influencia en la implementación práctica en este proyecto.

Arikan propone en un principio dos arquitecturas pipeline para codificación o decodificación SCD. Se destacan el uso de una estructura regular de bloques apropiada para el hardware, la versatilidad y la simplicidad de las mismas [6].

Leroux estudia distintas implementaciones en hardware de SCD, poniendo el foco en los niveles de paralelización del procesamiento. Propone la aproximación *min-sum* para las funciones f y g para implementaciones en hardware [7]. Finalmente, implementa una arquitectura semi-paralela que busca la reutilización de elementos de procesamiento (PE) y propone un elemento combinado (MPE) que puede efectuar las funciones f o g en base a una señal de control. Para reducir la complejidad del hardware utiliza representación en formato signo-magnitud de los LR, cuantificando en punto fijo, y trabaja en dominio logarítmico con LLR. Esta arquitectura presenta relativa complejidad para el manejo de los bloques de memoria, y requiere de una lógica de control especial [5].

Mishra publica la primera implementación del algoritmo SCD en ASIC. Con la arquitectura semi-paralela alcanza para $N = 1024$ velocidades de 49 Mbps [8].

Raymond encuentra que representaciones en punto fijo con reducidas cantidades de bits presentan similares resultados de corrección que las de punto flotante [9].

Pamuk implementa en FPGA una arquitectura flexible en cuanto al consumo de hardware versus velocidad,usando un algoritmo de propagación de creencias o *belief propagation*, con ciertas similitudes con SCD [10]. Finalmente, este autor implementa también en hardware una arquitectura que descompone mensajes de longitud N en mensajes de longitud $\sqrt{N}$ [11]. Park implementa este algoritmo en ASIC, utilizando una arquitectura paralela y alcanzando velocidades de 4,68 Gbps para $N = 1024$ [12]. Una modificación de este algoritmo, con técnicas empleadas en SCD, mejora la performance de decodificación y el throughput (TP) [13].

Berhault propone utilizar un registro de desplazamiento con retroalimentación lineal para calcular las sumas parciales en la arquitectura semi-paralela. Con el registro de desplazamiento es posible además implementar el codificador [14] [15]. Fan implementa un generador de sumas parciales cuya latencia es independiente de la longitud de mensaje [16].

Zhang publica una arquitectura SCD con la intención de reducir la latencia de decodificación. Son destacables el uso de pre-cómputo para minimizar latencias dentro de los PE, el uso de MPE y un sumador-restador que reduce la utilización de hardware de los PE para la función g. Utiliza la representación complemento a dos para los LLR [17].

Tal y Vardy presentan una variante de SCD usando listas y verificación de redundancia cíclica (CRC). Muestran una importante mejora en la capacidad correctora respecto a SCD tradicional, a cambio de una mayor complejidad algorítmica [18]. En 2014 se publica una arquitectura para ASIC del algoritmo de listas, sin empleo de CRC, que muestra consumo de hardware y latencia altos. La performance correctora es proporcional al número de listas almacenables [19]. Lin agrega CRC a una implementación en ASIC, y logra además reducir la latencia de decodificación [20]. Xiong implementa el algoritmo de listas decodificando los bits por conjuntos [21]. Éste propone posteriormente una variante del algoritmo de listas, en búsqueda de una reducción de la complejidad de procesamiento [22]. Fan duplica el throughput alcanzable con el algoritmo de

listas [23]. Piao diseña un elemento de procesamiento combinado para reducir el requerimiento de hardware de arquitecturas de decodificación con listas [24]. Yuan desarrolla para este algoritmo una arquitectura que permite estimar simultáneamente múltiples bits [25].

Para alcanzar mayores velocidades Sarkis y Giard trabajan con SCD simplificado, probando arquitecturas totalmente paralelas e integrando pipelines. Pueden interpretarse dichas arquitecturas como microprocesadores dedicados para la decodificación. Se destaca también el uso de codificación sistemática para mejorar la performance correctora, y que el throughput depende de la construcción del código [26]. Se alcanzan TP de 237 Gbps [27].

Dizdar y Arikan implementan en ASIC y en FPGA una arquitectura SCD simple y versátil, completamente combinacional. Se alcanzan muy elevadas velocidades y eficiencias de energía y de área. La arquitectura puede incorporar pipelines para mayores velocidades. El consumo de los recursos de los chips es muy elevado [28].

Del análisis bibliográfico se pueden hacer algunas observaciones. En la mayoría de las publicaciones se emplean variantes del algoritmo de cancelaciones sucesivas. El algoritmo SCD no es óptimo en cuanto a performance de decodificación para longitudes de código menores a 2^{20}. A longitudes bajas, la performance es mejor en códigos correctores más estudiados. La decodificación SCD tiene además la desventaja de ser secuencial y presentar alta latencia. El empleo de codificación sistemática mejora la performance de SCD a cambio de mayor complejidad del codificador. Lo mismo ocurre al utilizar listas, con un incremento en la latencia proporcional al tamaño de las listas empleadas. El mejor algoritmo de decodificación polar integra codificación sistemática, SCD con uso de listas y de CRC. Esta combinación logra superar la performance los códigos LDPC para $N = 2^{11}$. Continúa el desafío de encontrar implementaciones más rápidas y eficientes en el uso de hardware, y otros algoritmos y formas de extender la ventaja de los códigos polares a longitudes mayores.

Al implementar códigos correctores en hardware existe un compromiso entre la velocidad, la performance correctora, el consumo de potencia y la cantidad de recursos de hardware utilizados. Decodificadores que cometen menos errores necesitan de hardware más complejo, y la velocidad de decodificación es más baja. Algoritmos más simples pueden implementarse logrando mayores velocidades y eficiencias en el uso del hardware. Estos compromisos se tienen en cuenta en la siguiente sección, en la que se detallan las arquitecturas empleadas para la implementación de códigos polares en este proyecto.

3.3. Criterios de diseño de arquitectura

Para implementar el sistema de control de errores para códigos polares en FPGA, se programaron en lenguaje VHDL un codificador y un decodificador polares. Para hacerlo se seleccionaron de la bibliografía arquitecturas de alto nivel, y se diseñaron módulos necesarios para su implementación. Los criterios de selección o diseño se derivaron a partir de un conjunto de características deseadas y de requerimientos para el sistema, que se enumeran en esta sección.

Las principales características deseadas para la implementación son:

- Performance correctora demostrable, comparable a la encontrada en bibliografía.
- Capacidad de ser utilizado en canales AWGN, u otros si fuera posible.
- Posibilidad de cambiar el conjunto de los frozen bits para cada mensaje, de acuerdo al estado del canal.
- Código fuente escrito en lenguaje VHDL, paramétrico respecto a la longitud de los mensajes codificados N, la tasa de código R y la cantidad de bits de cuantificación Q asociados a los valores de los LLR.
- Código fuente que sea portable a distintas familias de FPGA.

- Utilización de las placas de desarrollo Terasic DE-10 Standard (integrando FPGA Intel Cyclone V modelo 5CSXFC6D6F31C6), disponibles en el Laboratorio de Comunicaciones (LAC) de la Facultad de Ingeniería de la Universidad Nacional de Mar del Plata.

Además de las características anteriores, se establece que en la entrada del codificador debe encontrarse el mensaje que incluye los frozen bits. A la entrada del decodificador deben llegar LLR calculados a partir de los mensajes recibidos, y a la salida debe ser entregado un mensaje decodificado que incluye los frozen bits.

3.4. Codificador

Para programar el codificador se seleccionó una de las arquitecturas propuestas por Arikan [6]. Esta arquitectura es independiente del código a utilizar y de la tasa de código, y puede usarse para cualquier longitud de código.

La arquitectura permite obtener la matriz generadora $\mathbf{F}^{\otimes \mathbf{n}}$ a partir de copias de dos tipos de operadores básicos, lo cual es ventajoso para una implementación en hardware.

El primero de estos operadores es un mezclador o *reverse-shuffler*, que cumple la función ya vista en el capítulo anterior. Si se coloca a la entrada de dicho operador un vector $\mathbf{v} = (v_0, v_1, \ldots, v_{N-1})$, a la salida se obtiene un vector $\mathbf{v}_{rs} = (v_0, v_2, \ldots, v_{N-2}, v_1, v_3, \ldots, v_{N-1})$.

El segundo operador efectúa aplicaciones del kernel polar $\mathbf{F}^{\otimes \mathbf{1}}$. Para un código de longitud N, el operador realiza $N/2$ aplicaciones del kernel. Partiendo de un vector $\mathbf{v} = (v_0, v_1, \ldots, v_{N-1})$ el operador produce un vector $\mathbf{v}_{\oplus} = (v_0 \oplus v_1, v_1, v_2 \oplus v_3, v_3, \ldots, v_{N-2} \oplus v_{N-1}, v_{N-1})$.

La estructura del codificador está formada por varias etapas en cascada, cada una compuesta a su vez por un módulo mezclador en cascada con un módulo sumador. Para un codificador de longitud de mensaje N se tienen $log_2 N$ etapas. Se muestra como ejemplo una de estas etapas, obtenida durante el desarrollo con el IDE Intel Quartus Prime, en la Fig. 3.2.

La estructura permite también que el codificador, si fuera requerido por el esquema decodificador, incorpore un módulo que efectúe la operación *bit-reverse*, a la entrada o a la salida del mismo. Esta capacidad es necesitada para la implementación del sistema completo, como se verá más adelante.

La arquitectura de Arikan es versátil, ya que con el agregado de hasta $log_2 N - 1$ registros entre los módulos permite implementaciones pipelined de hasta $log_2 N - 1$ niveles. Estas implementaciones permiten al codificador procesar varios mensajes al mismo tiempo, logrando multiplicar el throughput por un factor de hasta casi $log_2 N$.

Otra variante que se deriva de esta arquitectura es utilizar un búcle que conecte la salida del codificador a su entrada, intercalando registros. Así se pueden procesar los mensajes de a uno, haciendo $log_2 N$ usos de una única etapa. Esta variante sería más apropiada para longitudes de código muy altas, ya que el consumo de hardware a dichas longitudes sería mayoritariamente el que corresponde a una sola etapa, más el de los registros y la lógica de control.

En base a esta arquitectura general de Arikan se implementó finalmente un codificador combinacional, en el que se codifica un mensaje por ciclo de reloj. Éste implementa la matriz generadora de forma directa, sin registros internos ni lógica de control. Su elección se basó en su simpleza y en que su throughput es suficiente para alimentar al decodificador utilizado. Los resultados de su síntesis se exponen más adelante en el informe. Otro motivo para su elección es que, como se mostrará en la próxima sección, el decodificador implementado está compuesto en parte por varios codificadores combinacionales.

3.5. Decodificador

3.5.1. Arquitectura de alto nivel

Para implementar el decodificador polar se seleccionó la arquitectura propuesta por Dizdar y Arikan recientemente, en el año 2016 [28]. Esta arquitectura es combinacional, recursiva y

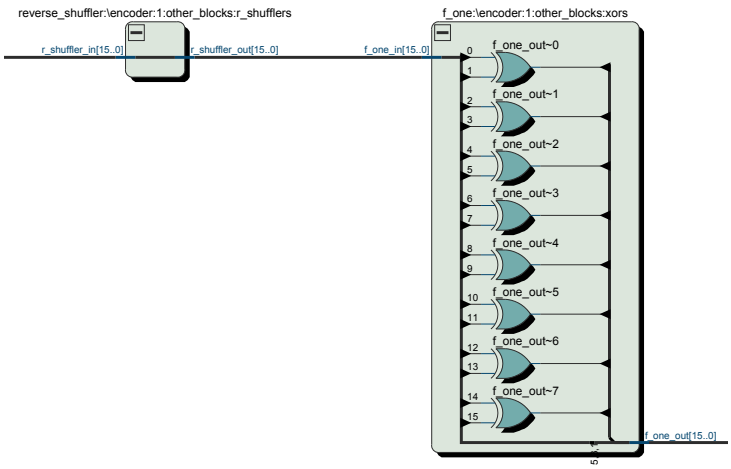


Figura 3.2: Visualización RTL de una etapa del codificador para $N = 16$

simple, y aplica el algoritmo SCD. En el trabajo de Dizdar se presenta esta arquitectura de alto throughput, alta eficiencia energética y alto throughput por área. Es una arquitectura completamente paralela, y ejemplifica el estado del arte en las implementaciones veloces del algoritmo SCD, pudiendo también incorporar pipelines. Estas características la hacen atractiva para que la FPGA funcione como acelerador de simulaciones de canales de comunicaciones, posibilidad que permite acortar los tiempos de simulación por el método Monte Carlo, principalmente para probabilidades de error muy bajas, que requieren de pruebas con grandes cantidades de mensajes. La implementación de esta arquitectura se describe a continuación.

Los códigos polares pertenecen a la familia de los códigos concatenados generalizados. Un código de longitud de mensaje N y tasa de código R está conformado por dos códigos de longitud $N/2$ y tasas de código R' y R'' respectivamente. Esta característica puede aprovecharse para construir de forma recursiva decodificadores para N en base a dos decodificadores para $N/2$. Esto se aprecia en la Fig. 3.3, en la que las líneas de colores encuadran los decodificadores componentes del decodificador para $N = 8$.

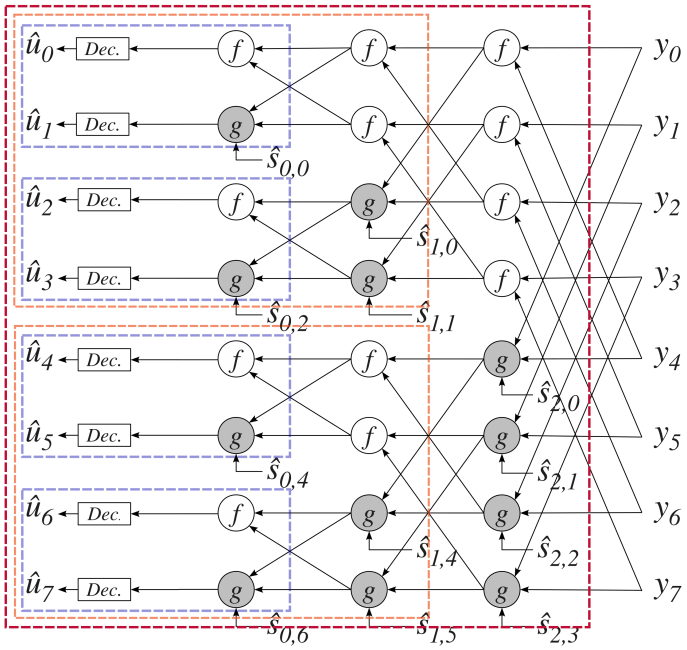


Figura 3.3: Composición recursiva del decodificador para $N = 8$

Para visualizar la implementación utilizada para la estructura recursiva se presenta el

diagrama del decodificador para $N = 16$ y sus bloques componentes en la Fig. 3.4.

En la Fig. 3.4 aparecen varios tipos de bloques:

- **Registro de frozen bits**: Almacenamiento de los frozen bits, que corresponden a los calculados para codificar el mensaje actual.
- **Decodificador base**: Decodificador para $N = 4$ en el que se toma la decisión de estimación de los bits del mensaje original. Cada bloque tiene cuatro LLR y cuatro frozen bits como entradas.
- **Codificador N**: Codificador polar combinacional y paramétrico, semejante al empleado en el sistema transmisor. Propaga los bits estimados como sumas parciales a los bloques de procesamiento que las requieran.
- **M PE Gen N**: Bloque paramétrico que genera $^{N}/_{2}$ bloques combinados de procesamiento MPE que implementan los operadores f y g. Cada bloque tiene dos LLRs como entradas.

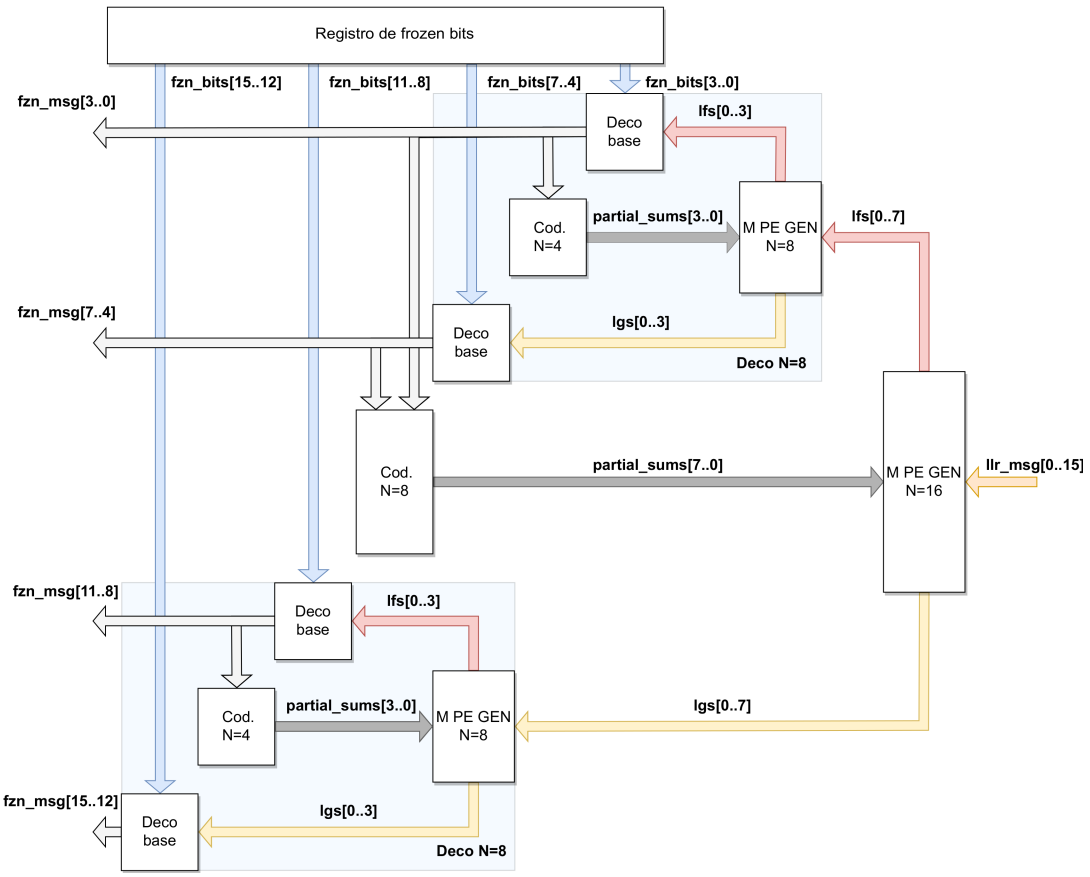


Figura 3.4: Entidades y señales en el decodificador para $N = 16$

En la implementación del decodificador, los bloques decodificadores para $N = 8$ pertenecen al caso base de la recursión, que es el nivel más bajo para el que se define la topología general que une todos los bloques componentes. Se aprecia fácilmente en la Fig. 3.4 cómo se replicaría la estructura para mayores valores de N. El decodificador para $N = 16$ corresponde al nivel de recursión siguiente al caso base, y es parte del caso general de la recursión. El codificador más básico encontrado en la estructura es el que corresponde a $N = 4$. Estos componentes serán tratados en detalle más adelante. Se hace notar finalmente en la Fig. 3.4 que las señales correspondientes a los LLR, provenientes del canal y de las funciones f y g, se propagan hacia la izquierda, mientras las sumas parciales se propagan desde los bloques de decisión hacia la derecha.

La estructura del decodificador combinacional es repetitiva y emplea un gran número de componentes básicos. Un decodificador para longitud N requiere:

- ${}^{N}/_{4}$ decodificadores básicos.
- $\frac{N}{2} \times log_2 \frac{N}{4}$ unidades MPE, sin contar las pertenecientes a los decodificadores básicos.
- $\sum_{i=2}^{log_2 {}^{N}/_{2}} \frac{N}{2^{i+1}} e_{2^i}$ codificadores, e_x representando un codificador para mensajes de longitud x. No tiene en cuenta este número los codificadores e_2 dentro de los decodificadores básicos.
- Un registro de $N \times 1$ bits para almacenar los frozen bits a la entrada del decodificador.
- Un registro de $N \times Q$ bits para almacenar los LLRs iniciales en la entrada del decodificador.
- Un registro de $N \times 1$ bits para almacenar los bits del mensaje decodificado a la salida.

El codificador para $N = 16$ de la Fig. 3.4 contiene:

- 4 decodificadores básicos.
- 16 MPE, sin contar los pertenecientes a los decodificadores básicos.
- Codificadores: $1 \times e_8$ y $2 \times e_4$.
- Dos registros de 16 bits cada uno.
- Un registro de $16 \times Q$ bits.

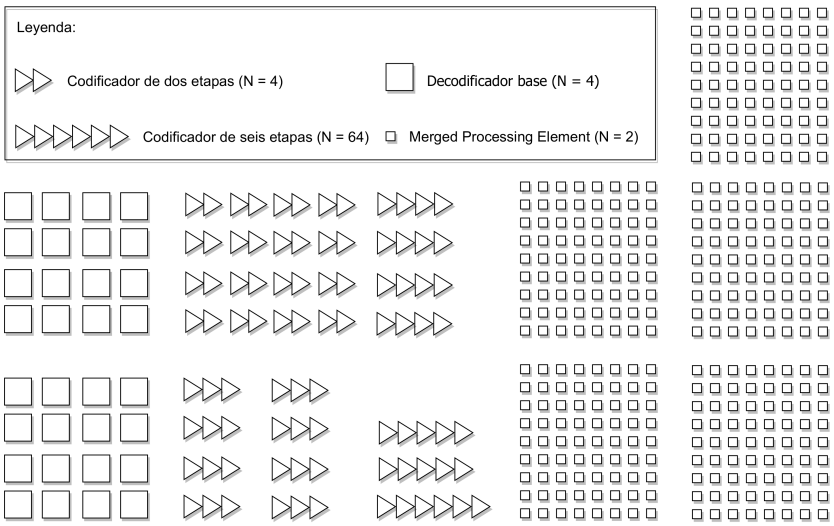


Figura 3.5: Bloques componentes del decodificador para $N = 128$

Tomando como ejemplo un decodificador más complejo, para $N = 128$, se calcula que requiere:

- 32 decodificadores básicos.
- 320 MPE, sin contar los pertenecientes a los decodificadores básicos.
- Codificadores: $1 \times e_{64}$, $2 \times e_{32}$, $4 \times e_{16}$, $8 \times e_8$ y $16 \times e_4$.
- Dos registros de 128 bits cada uno.
- Un registro de $128 \times Q$ bits.

Todas las entidades que realizan operaciones en el decodificador para $N = 128$ se muestran en la Fig. 3.5. No se incluyen los registros de almacenamiento de entradas y salidas.

La programación en VHDL de una gran estructura repetitiva y recursiva, y compuesta por numerosas entidades, puede simplificarse considerablemente si se aprovecha la característica recursiva de la misma, como se trata a continuación.

3.5.2. Programación recursiva en VHDL

Teniendo en cuenta las características anteriores, para programar el decodificador en VHDL se decidió utilizar una variante recursiva del estilo de programación estructural. Esta técnica puede simplificar y facilitar la declaración de estructuras recursivas [29].

En general puede emplearse para describir una estructura recursiva y parametrizable de hardware una entidad $E_t = (t, p_1, \ldots, p_n, E, S)$. Para esta entidad $t \in \mathbb{N}$ es un parámetro que define el tamaño de la estructura, $p_1, \ldots, p_n$ son parámetros adicionales que determinan la implementación de la estructura, E y S son conjuntos de puertos de entrada y de salida, respectivamente. El número de puertos de entrada y de salida puede depender de los otros parámetros. Además, la implementación de E_t contiene una o más entidades E_{t-1}, con $t > b$, y la implementación de E_b no contiene instancia alguna del tipo E_i. La entidad E_b pertenece al caso base de la recursión, que es el caso para el cual la recursión termina.

La programación en estilo recursivo en VHDL es poco habitual y la bibliografía encontrada al respecto es escasa. A pesar de esto, se trabajó sobre el reporte técnico de Ashenden sobre modelos recursivos, partiendo de las técnicas y plantillas en versión IEEE 1076-1987 del lenguaje para obtener una sintaxis apropiada para la versión IEEE 1076-1993, utilizada en este trabajo.

En VHDL las estructuras recursivas se pueden describir, usando el estilo estructural de programación, mediante la declaración de entidades, usando la palabra reservada **entity**. Los parámetros t y $p_1, \ldots, p_n$ deben declararse como constantes con la palabra **generic** y se pueden comunicar entre entidades usando la sentencia **generic map**. Los puertos pueden parametrizarse usando arreglos o registros, cuyo tamaño depende de los parámetros anteriores.

La entidad E_t debe ser declarada dos veces. Inicialmente debe ser declarada como **entity** al comienzo del código VHDL, y luego re-declarada como componente con la palabra **component**, antes del comienzo de la arquitectura de la entidad. Por otro lado, deben declararse paramétricamente en la arquitectura las señales que conectan a la entidad E_t con las entidades E_{t-1}. Éstas deben ser luego conectadas usando mapeos de puertos o **port map**.

Finalmente, debe definirse en la implementación de la arquitectura la topología para el caso base de la recursión con la sentencia **if** $t = b$ **generate**, y para el caso general con la sentencia **if** $t > b$ **generate**.

Para programar el decodificador se tomó como parámetro de tamaño la longitud del código, es decir $t = N$. El caso base de la recursión se programó para $N = 8$, y el caso general para valores de $N > 8$. Por lo tanto, el código fuente sólo permite sintetizar decodificadores para valores de $N \geq 8$. Los componentes utilizados fueron los que aparecen en la Fig. 3.2, con excepción de los registros. La declaración de señales se hizo utilizando el tipo **record** y otros tipos declarados dentro de un paquete común a todo el sistema implementado. El código VHDL del decodificador polar está disponible en los anexos del informe.

En la Fig. 3.6 se presenta la visualización RTL obtenida en Quartus tras sintetizar un decodificador para $N = 16$. Se destaca que el sintetizador interpreta correctamente las entidades que componen la estructura recursiva y sus conexiones, con gran similitud a la Fig. 3.2. La estructura se repite también en el interior uno de los dos decodificadores para $N = 8$ que lo componen, como se aprecia en la Fig. 3.7.

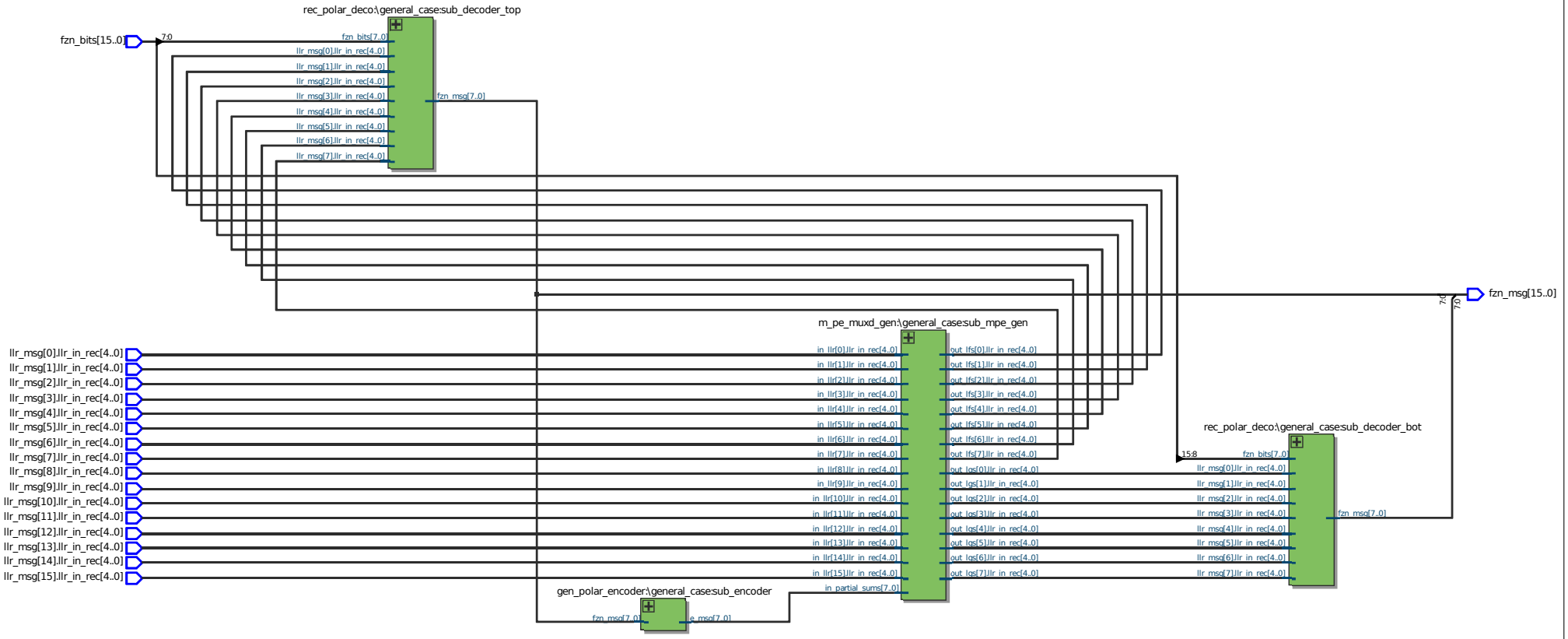


Figura 3.6: Diagrama RTL del decodificador para $N = 16$

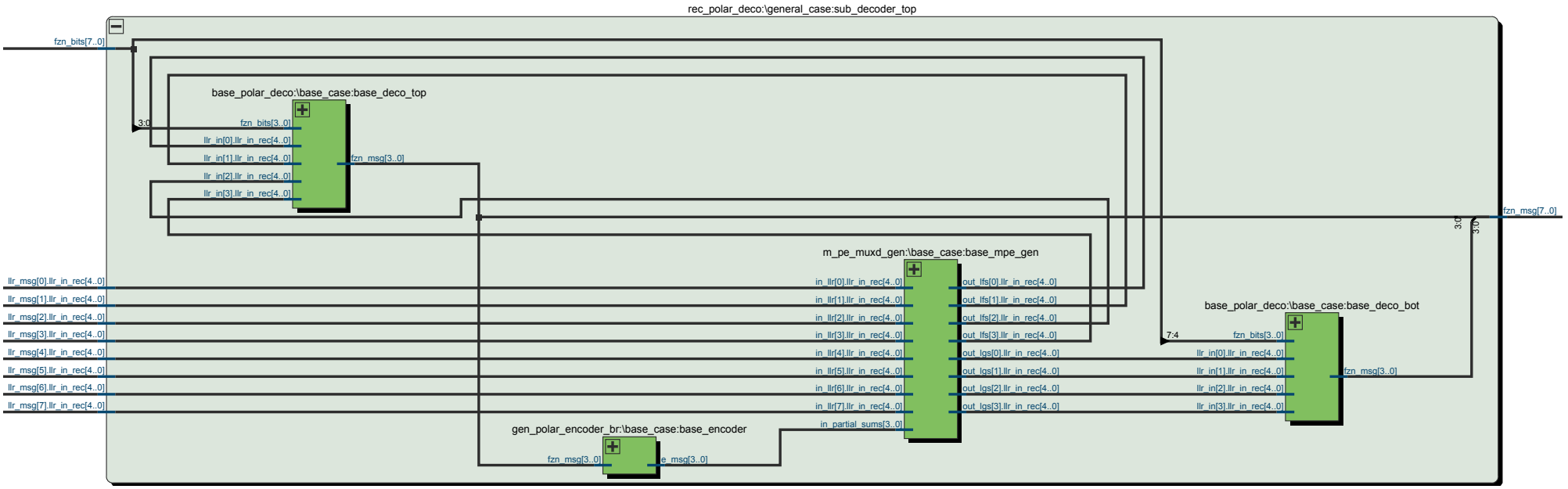


Figura 3.7: Diagrama RTL del decodificador $N = 8$, componente del decodificador para $N = 16$

3.5.3. Aproximaciones para el algoritmo de decodificación por cancelaciones sucesivas

En el algoritmo de decodificación por cancelaciones sucesivas las funciones f y g adoptan las expresiones:

$$f(a,b) = \frac{1+ab}{a+b} \tag{3.1}$$

$$g(a,b,\hat{s}) = a^{1-2\hat{s}}b \tag{3.2}$$

Las expresiones anteriores operan sobre el dominio de los LR y requieren de la aplicación de multiplicaciones y divisiones. Si, en cambio, se trabaja en el dominio logarítmico, se reduce la complejidad de los bloques destinados al cálculo de dichas expresiones [7]. En el dominio logarítmico pueden utilizarse expresiones alternativas para las funciones f y g, operando sobre LLRs simbolizados como L_i:

$$f(L_a, L_b) = 2\tanh^{-1}\left(\tanh(\frac{L_a}{2})\ \tanh(\frac{L_b}{2})\right) \tag{3.3}$$

$$g(L_a, L_b, \hat{s}) = L_b + (-1)^{\hat{s}} L_a \tag{3.4}$$

La nueva ecuación de g implica el uso de sumas y restas, que son fáciles de implementar en hardware; la ecuación de f es todavía compleja para implementar. Puede emplearse la aproximación denominada *min-sum*, que se utiliza en el algoritmo de propagación de creencias para decodificación de códigos LDPC [7]. Con esta aproximación quedan las ecuaciones:

$$f(L_a, L_b) \approx signo(L_a)\ signo(L_b)\ min(|L_a|, |L_b|) \tag{3.5}$$

$$g(L_a, L_b, \hat{s}) = L_b + (-1)^{\hat{s}} L_a \tag{3.6}$$

En la Ec. 3.6 es necesario para f comparar las magnitudes de los L_i. También es necesario calcular el signo de los L_i según:

$$signo(L_i) = \begin{cases} 1, & \text{si } L_i \geq 0 \\ -1, & \text{en otro caso} \end{cases} \tag{3.7}$$

La utilización de la aproximación *min-sum* ocasiona una degradación de la performance de decodificación despreciable, a cambio de una gran simplificación en el hardware requerido para su implementación [5].

Para implementar en hardware las funciones anteriores, es necesario cuantificar los LLRs con un número fijo Q de bits. La utilización de un número finito de bits de cuantificación puede degradar la performance de decodificación en comparación a representaciones de punto flotante. Si se simula el algoritmo para distintos valores de Q, se encuentra que con seis bits la performance es casi idéntica a la obtenida en punto flotante; con cinco bits la performance es levemente peor, pero es un compromiso aceptable para reducir el requerimiento de hardware [5]. Estas características se mantienen para longitudes de código menores [28].

En los sistemas basados en hardware se representan generalmente los LLRs en formatos de punto fijo, ya sea como complemento a dos o como signo y magnitud (SM) [5]. Ambas representaciones destinan un bit para el signo y $Q-1$ bits para la magnitud. La representación en formato SM puede simplificar el hardware para implementar la Ec. 3.6, y por eso es utilizada en los elementos de procesamiento.

3.5.4. Elementos de procesamiento

Consideraciones iniciales

Los elementos de procesamiento son fundamentales para la operación del decodificador. El entrelazado de un gran número de estos elementos forma una red que opera sobre los LLR para lograr finalmente la mejor estimación posible de los bits del mensaje original. Debido a su presencia numerosa en la mayor parte de la estructura del decodificador, el diseño óptimo de los mismos es particularmente importante. Por ello, cualquier simplificación en los recursos que requieren podría lograr un ahorro importante en el requerimiento de recursos de la totalidad del decodificador. De manera similar, cualquier mejora en la velocidad de los mismos debería otorgar un efecto nada despreciable en la velocidad del decodificador como un conjunto.

En la bibliografía, se encuentran varios diseños completos para los elementos de procesamiento, como los elementos combinados (MPE) de Leroux para implementaciones en formato signo y magnitud de los LLR, y los de Zhang para decodificadores que trabajen en complemento a dos [5] [17]. La implementación de Leroux se diseñó con un criterio de reutilización de los MPE, por lo que su estructura no es óptima desde el punto de vista de la velocidad o el requerimiento reducido de recursos de hardware. La implementación de Zhang presenta bajas latencias, pero trabaja en signo-magnitud solamente de forma interna y requiere de módulos de conversión de formatos de cuantificación; los módulos de conversión agregan complejidad y extienden el camino crítico del diseño. El camino crítico es el recorrido más largo dentro de un módulo de la FPGA y el tiempo de propagación de una señal por el mismo determina la máxima frecuencia a la que puede operar.

Teniendo en cuenta que la arquitectura combinacional seleccionada para la implementación en este trabajo es altamente demandante en recursos de hardware, y que busca maximizar la velocidad de procesamiento, se decidió realizar un diseño original de los elementos de procesamiento. La cuantificación SM de los LLR permite que las implementaciones de los operadores f y g compartan recursos. El uso compartido de recursos implica la utilización de un elemento de procesamiento que combine ambas funciones, denominado elemento de procesamiento combinado. Por lo tanto, se decidió que el diseño se destinaría a elementos de procesamiento combinados, con una cuantificación en formato signo y magnitud.

La implementación de la función f requiere la comparación de las magnitudes y de los signos de dos LLR. El operador g implica efectuar una suma de dos LLR, si la suma parcial asociada es 0, y la resta de los LLR cuando la suma parcial es 1 (Ec. 3.6).

Algoritmo de adición y sustracción en formato signo-magnitud y pre-cómputo

Para encarar el diseño se estudió el algoritmo de adición y sustracción para números representados en un formato de signo y magnitud, que se presenta en la Fig. 3.8 [30].

En la Fig. 3.8 se destaca inicialmente la necesidad de hacer una comparación de signos y una comparación de magnitudes. Ésto es lo que necesita la función f, por lo tanto se comprueba que se pueden compartir recursos entre las dos funciones y que es conveniente seguir el desarrollo en combinado del elemento de procesamiento.

En segundo lugar, se observa que al efectuar una suma o una resta se deben seguir caminos separados, en función de las comparaciones de signos y magnitudes. Esto abre la posibilidad de efectuar en paralelo la suma y la resta de los LLR.

La técnica de pre-cómputo es aplicada por Zhang con el objetivo de reducir la latencia de los PE [17]. Para una función general $F = (a_1, a_2, \ldots, a_n)$ que depende de múltiples argumentos a_i, la técnica implica el cálculo inicial de todos los resultados posibles dependientes de algunos de los a_i, seleccionando finalmente el resultado final en base a los argumentos restantes y en

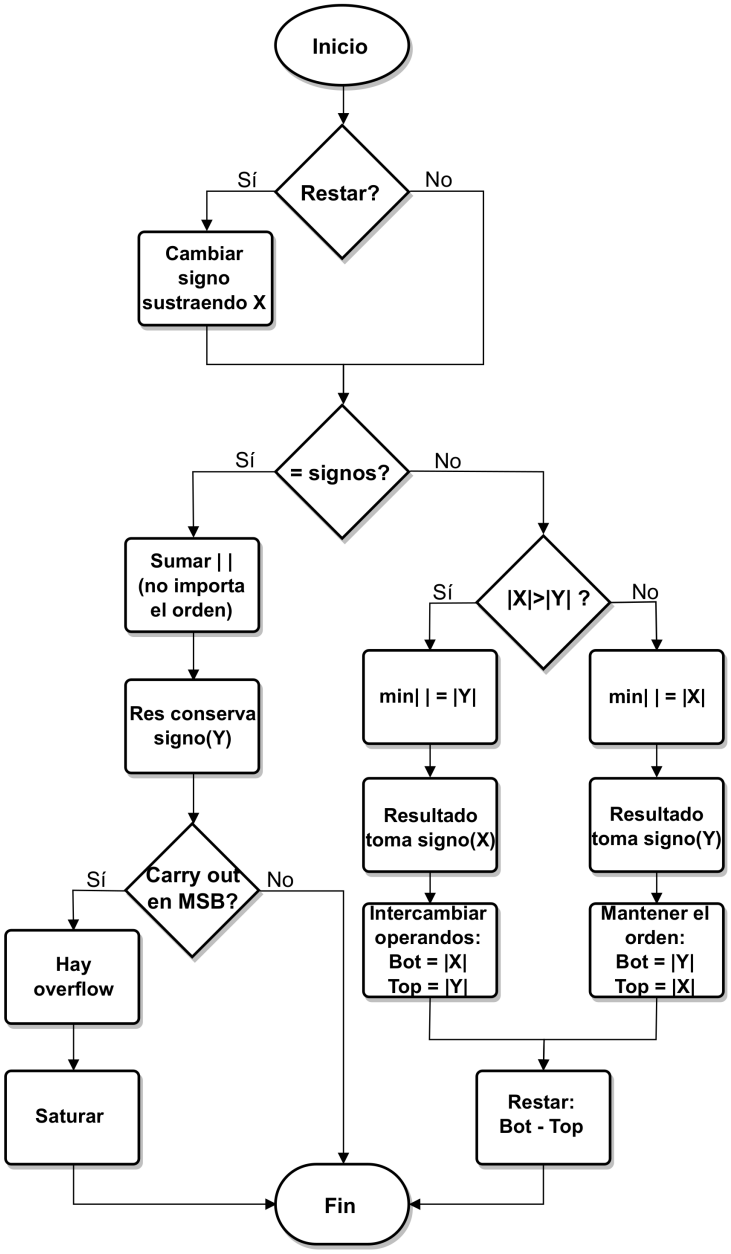


Figura 3.8: Algoritmo para suma y resta con representación signo-magnitud

consecuencia reduciendo la latencia efectiva de la función.

Para la función g la técnica de pre-cómputo sería aplicable, ya que se podría calcular en paralelo la suma y la resta de los LLR, relegando la decisión final del resultado a la disponibilidad de la suma parcial. Esta técnica es especialmente relevante considerando que se trabaja con el algoritmo de decodificación por cancelaciones sucesivas. Un MPE sería activado inicialmente para calcular la función f y comenzar el cálculo de los dos resultados posibles de g. El resultado de f, estando disponible al inicio del pre-cálculo de g, podría continuar su propagación a lo largo del decodificador; mientras tanto, la función g finalizaría el pre-cálculo de sus dos posibles resultados. Finalmente, el MPE sería activado al arribar al mismo la suma parcial relacionada, que seleccionaría el LLR resultante apropiado de la función g.

Efectuar en paralelo la suma y la resta implicaría a priori integrar bloques sumadores y restadores separados en el MPE. Zhang propone un sumador-restador paralelo que comparte recursos a nivel de las compuertas lógicas, reduce el consumo de hardware un 57 % respecto al hardware separado, y presenta la misma prestación en cuanto a velocidad [17]. El bloque podría ser implementado en el MPE para lograr los beneficios de reducción de latencia y consumo de hardware que se buscan desde un principio.

El algoritmo de suma y resta, presenta un problema en cuanto a la ocurrencia de desbordes u *overflow* de los resultados de la suma. Para evitar la pérdida de precisión entre los distintos MPE es entonces necesario ir aumentando de a un bit la cantidad de bits Q para cuantificar los LLR con un gran aumento de la complejidad del sistema. Sin embargo, la utilización de una cuantificación uniforme a lo largo del decodificador, con $Q = 5$ degrada la performance de decodificación marginalmente respecto a una implementación de gran precisión con punto flotante [5] [28]. Con estas consideraciones, para simplificar la implementación del decodificador se decidió adoptar una cuantificación uniforme con $Q = 5$ bits. En caso de existir overflow, se opta por saturar el resultado, estableciendo un límite máximo a la magnitud de los LLR en la entrada y el interior del decodificador.

Elemento de procesamiento combinado

Siguiendo las consideraciones anteriores se diseñó el elemento de procesamiento combinado cuyo diagrama con los sub-módulos, señales y anchos de buses se presenta en la Fig. 3.9.

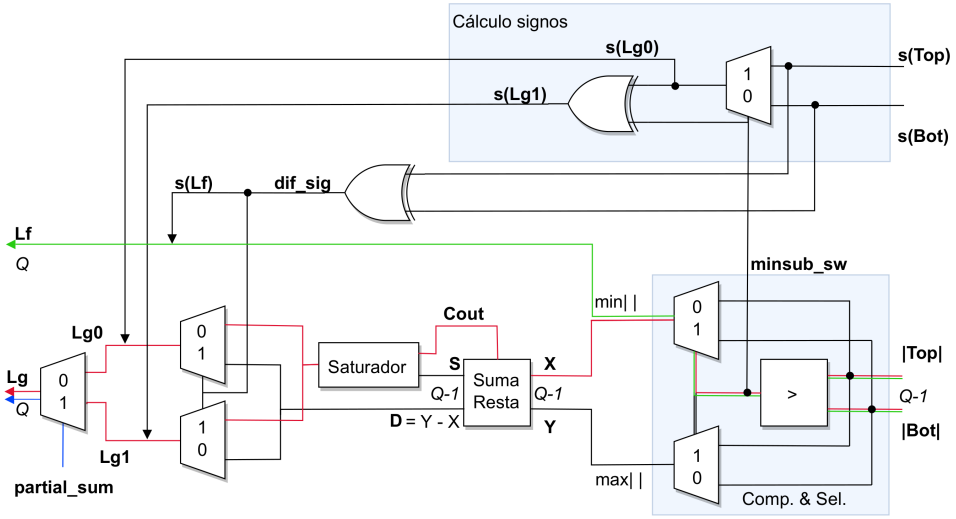


Figura 3.9: Bloques y señales dentro del elemento de procesamiento combinado

Módulo de comparación y selección de magnitudes

En la Fig. 3.9 se destaca inicialmente un bloque de comparación y selección en función de las magnitudes de los LLR. Este bloque cumple cuatro funciones:

- Encontrar la menor de las magnitudes de los dos LLR en la entrada.
- Entregar al sumador-restador la menor de las magnitudes como sustraendo X y la mayor como minuendo Y para poder restar adecuadamente.
- Entregar a la salida del MPE la menor de las magnitudes como la magnitud del resultado del operador f.
- Señalar al bloque de cálculo de signos si el encontrar la menor de las magnitudes implicó conmutar las entradas.

Módulo de cálculo de signos

El módulo de cálculo de signos determina el signo de los dos resultados posibles del operador g, denominados $s(Lg)_{\hat{s}=i}$ según el valor que adopta la suma parcial $\hat{s}$. En el formato de signo y magnitud un 0 en el bit de signo identifica a un número positivo, y un 1 representa un

número negativo. El cero puede ser representado con el signo positivo o negativo, y presenta magnitud nula. Para encontrar la lógica que implemente este cálculo se evaluaron las posibles combinaciones de signos y magnitudes de las entradas al MPE, y se encontraron los valores correctos que deberían adoptar los signos de los resultados. En base a este procedimiento se produjo la Tabla 3.1 con las siguientes entradas:

- El signo del LLR de la entrada superior al MPE, **s**(**Top**).
- El signo del LLR de la entrada inferior, **s**(**Bot**).
- El bit que indica la conmutación de las magnitudes de las entradas, obtenido en el bloque de comparación y selección, **SW**.

Salidas		Entradas		
$s(Lg)_{\hat{s}=1}$	$s(Lg)_{\hat{s}=0}$	SW	$s(Top) = s(X)$	$s(Bot) = s(Y)$
0	0	0	0	0
1	1	0	0	1
0	0	0	1	0
1	1	0	1	1
1	0	1	0	0
1	0	1	0	1
0	1	1	1	0
0	1	1	1	1

Cuadro 3.1: Tabla de verdad para cálculo de los signos del resultado de g

A partir de la tabla de verdad anterior se compuso el mapa de Karnaugh asociado y al simplificar se encontraron las siguientes ecuaciones lógicas:

$$s(Lg)_{\hat{s}=1} = SW \cdot \overline{s(X)} + \overline{SW} \cdot s(Y) \tag{3.8}$$

$$s(Lg)_{\hat{s}=0} = SW \cdot s(X) + \overline{SW} \cdot s(Y) \tag{3.9}$$

Las ecuaciones anteriores pueden implementarse con dos multiplexores (MUX), pero uno de ellos puede ser simplificado, tomando $s(Lg)_{\hat{s}=0}$ y observando que:

$$s(Lg)_{\hat{s}=1} = s(Lg)_{\hat{s}=0} \oplus SW \tag{3.10}$$

Sumador-restador combinado

Este módulo se implementa con el hardware propuesto por Zhang y opera solamente sobre los $Q-1$ bits de magnitud de los LLR [17]. Lleva a cabo las operaciones $Y+X$ e $Y-X$. Se compone paramétricamente por un medio sumador-restador de un bit para el bit menos significativo (LSB), y una cascada de $Q-2$ sumadores-restadores completos de un bit. La salida carry-out del sumador-restador correspondiente al bit más significativo (MSB) se usa para señalar la existencia de overflow en la suma.

Módulo saturador

Este bloque es simplemente un MUX cuya entrada de selección corresponde al bit de carry-out del módulo sumador-restador. Cuando el valor del carry-out es 1 se reemplaza el resultado de la suma por la mayor magnitud representable con $Q-1$ bits (igual a $2^{Q-1}-1$).

Conmutador de resultados

Si se observa el algoritmo de la Fig. 3.8 se puede apreciar que hay un nodo de decisión basado en la igualdad de los signos de las entradas. A partir de este nodo el camino se bifurca según se desee sumar o restar, y las operaciones se completan en paralelo. Para determinar las salidas correctas del sumador-restador es necesario conmutar las mismas en base a una señal denominada **dif_sig**. Cundo su valor es 1, esta señal indica que existe una desigualdad de los signos de las entradas al MPE. La señal es también el bit con el signo del LLR que resulta del operador f.

Salidas del elemento de procesamiento

En cuanto las operaciones se completan, el resultado de la operación f se dispone en la salida del MPE. En la Fig. 3.9 se representa con trazos verdes el camino crítico de esta operación. Se observa que el mismo coincide con el camino crítico del módulo de comparación y selección de magnitudes. Para la función g el camino crítico real se muestra con los trazos rojos. Se extrae de inmediato que es considerablemente más largo que el camino crítico de f. Los resultados de g se encuentran disponibles en la entrada de un multiplexor, y debido a la característica secuencial del algoritmo SCD el camino crítico *efectivo* de g es el representado por el trazo de color azul. Gracias al diseño con la técnica de pre-cómputo el camino crítico de una relativamente extensa cadena de módulos de procesamiento es entonces reemplazado por el de un único multiplexor.

Elemento de procesamiento con salida simplificada

En el diseño de sistemas digitales en FPGA suele hacerse un compromiso o *trade-off* entre la velocidad y el consumo de recursos de hardware. Para el MPE diseñado puede lograrse una reducción en el consumo de recursos, a cambio de un leve incremento en el camino crítico de la implementación de g. Este leve rediseño se presenta en la Fig. 3.10, en la que el camino crítico efectivo de g se representa con los trazos azules.

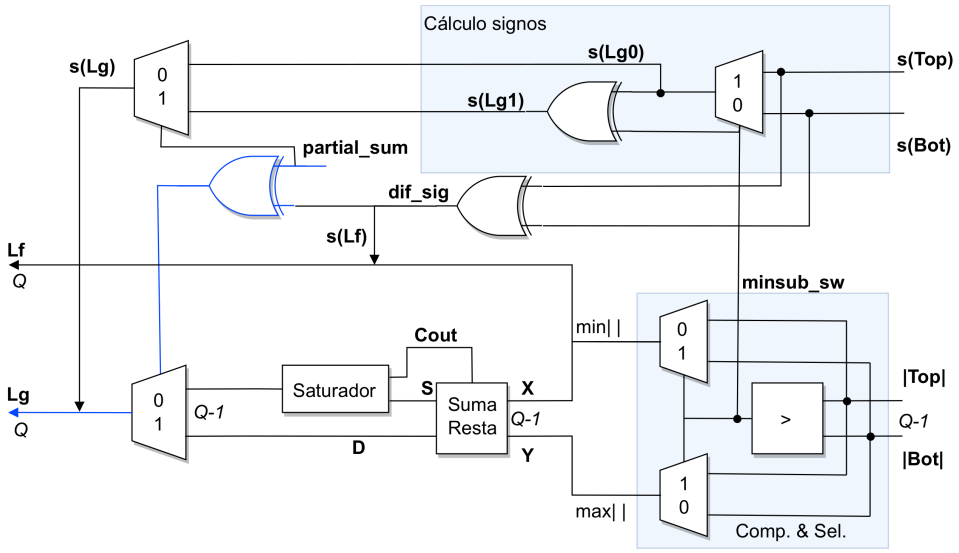


Figura 3.10: Bloques y señales dentro del MPE con salida simplificada

Para lograr la simplificación se partió con la intención de combinar en un solo multiplexor los tres multiplexores en la salida del diseño original, que en conjunto establecen el la magnitud del resultado de g. Se elaboró la Tabla 3.2 especificando el valor que debería tomar la señal de selección *SEL* del nuevo multiplexor.

Con la tabla de verdad anterior, es trivial encontrar que se cumple la relación:

$$SEL = \hat{s} \oplus dif\_sig \tag{3.11}$$

Salida	Entradas	
SEL	$\hat{s}$	dif_sig
0	0	0
1	0	1
1	1	0
0	1	1

Cuadro 3.2: Tabla de verdad para simplificar la salida del operador g

Es por lo tanto posible tener sólo un MUX para seleccionar la magnitud correcta. Debe tenerse en cuenta que al hacer este cambio debe agregarse además un MUX con $SEL = \hat{s}$ para poner a la salida el signo de g correcto.

Con estas modificaciones se logra reemplazar un MUX con entradas de ancho Q y dos MUX con entradas de ancho $Q-1$ por un MUX con entradas de ancho $Q-1$, un MUX de un bit y una compuerta *XOR*, requiriendo una menor complejidad de consumo y un camino crítico *real* más corto. Sin embargo, al implementar estos cambios el camino crítico efectivo es más largo, habiéndosele agregado el camino crítico correspondiente a una compuerta *XOR*. Si bien este incremento en el camino crítico parece despreciable, debe tenerse en consideración que el MPE forma parte de una red que procesa secuencialmente los LLR. Por lo tanto, este incremento puede interpretarse, para el decodificador combinacional de este trabajo, como la conexión en serie de $N-1$ compuertas *XOR*. El efecto de la simplificación sobre el camino crítico deja entonces de ser despreciable y debe ser tenido en consideración al momento de escoger un modelo de elemento de procesamiento.

En este trabajo se sigue un criterio de maximización de la velocidad, por lo tanto para la implementación del decodificador se decidió continuar con el MPE diseñado originalmente, sin la reducción de consumo de recursos de hardware presentada en estos últimos párrafos.

Corrección de ceros

La representación de signo y magnitud tiene dos representaciones posibles para el cero. En las primeras simulaciones del MPE de la Fig. 3.9 se observó que el operador g podía generar ceros con el bit de signo negativo, y que también podía propagarlos a partir de los LLR de entrada. Para poder determinar si estos efectos producirían degradación en la performance correctora, se diseñó un simple bloque de corrección para estos ceros *negativos*, cuya disposición interna se muestra en la Fig. 3.11.

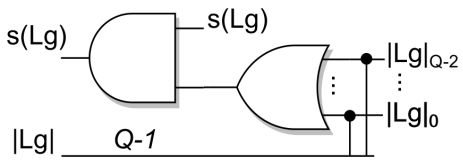


Figura 3.11: Bloque para corregir ceros negativos en la función g

El bloque corrector está compuesto por una compuerta *OR* ancha que toma los $Q-2$ bits de la magnitud del resultado de g. El resultado de esta compuerta con una operación *AND* con el bit de signo fuerza a la salida del MPE un signo positivo cuando los bits de la magnitud de g son todos ceros. El bloque es transparente para las señales cuya magnitud es distinta de cero.

Se presenta en la Fig. 3.12 el MPE que incorpora los bloques correctores de ceros negativos. La ubicación de los mismos se hace en la entrada del MUX que define la salida de g, conservando así la longitud del camino crítico original y manteniendo el criterio de diseño fundamental adoptado en este trabajo.

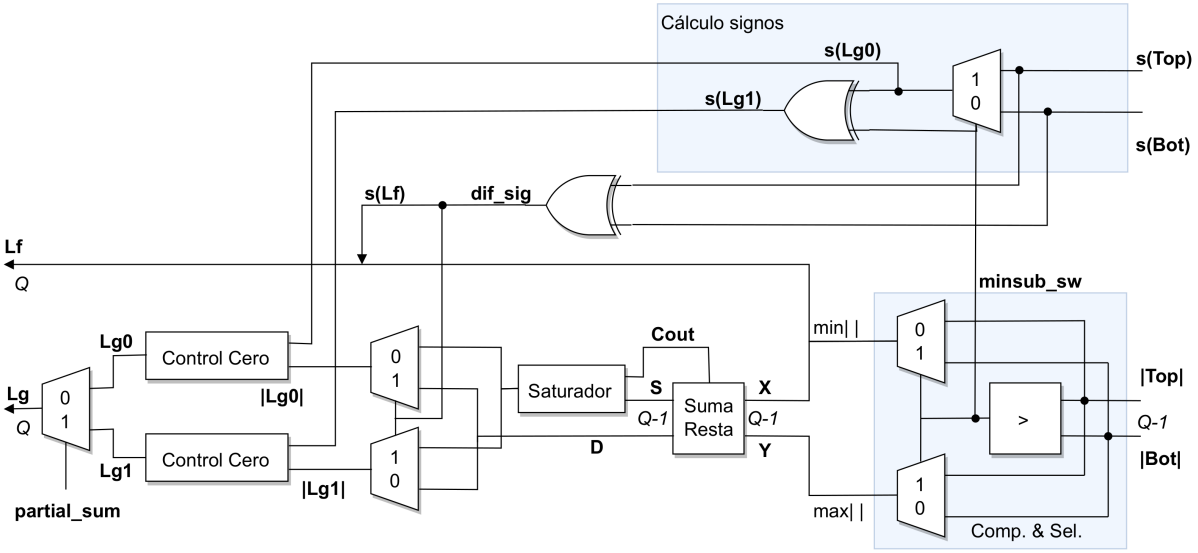


Figura 3.12: Bloques y señales dentro del MPE con control de ceros para g

3.5.5. Decodificador base

El decodificador básico, un decodificador combinacional para el caso especial $N = 4$, podría implementarse trivialmente con cuatro elementos combinados de procesamiento, una compuerta *XOR* para determinar una de las sumas parciales, y las conexiones adecuadas de estos componentes.

En su trabajo sobre la arquitectura combinacional, Dizdar propone efectuar simplificaciones en la etapa de cálculo de las funciones f y g previas a la decisión de estimación final [28]. Las simplificaciones involucran principalmente operar con los signos de los LLR resultantes de la etapa previa, y efectuar comparaciones de magnitud para obtener los signos de los LLR que alcanzan el nivel de decisión. Así se evita efectuar las sumas y restas en la etapa final, en conjunto con las operaciones auxiliares que éstas requieren. Las estimaciones finales se implementan para cada índice con la operación *AND* con el frozen bit correspondiente. Las simplificaciones permiten reducir la latencia de decodificación y el consumo de recursos de hardware del decodificador básico.

El decodificador base se implementa con dos MPE y las simplificaciones anteriores en la etapa final. Su diagrama se muestra en la Fig. 3.13.

3.5.6. Generación de sumas parciales

Para generar las sumas parciales en el interior del decodificador se utilizan codificadores combinacionales para distintas longitudes de mensaje. Los mismos respetan el esquema de codificación con reorden bit-reverse, de las entradas o de las salidas, que se debe emplear en el codificador ubicado en el sistema transmisor.

Los generadores de sumas parciales menos complejos requieren de una etapa y una sola compuerta *XOR*, y se encuentran solamente en los decodificadores básicos. El generador de sumas parciales más complejo necesario para el funcionamiento de este decodificador es único y corresponde a una longitud de mensaje $^{N}/_{2}$.

3.5.7. Registros de almacenamiento

El decodificador opera sin señales de reloj ni registros de almacenamiento internos. Sin embargo, durante el todo el tiempo que lleva la operación de decodificación el sistema debe disponer a la entrada de registros sincrónicos en los que se dispongan los bits de los LLR del canal, y de un registro en el que se encuentren los frozen bits. El contenido de los registros con las entradas no puede cambiar mientras el sistema esté decodificando un mensaje, de lo contrario la

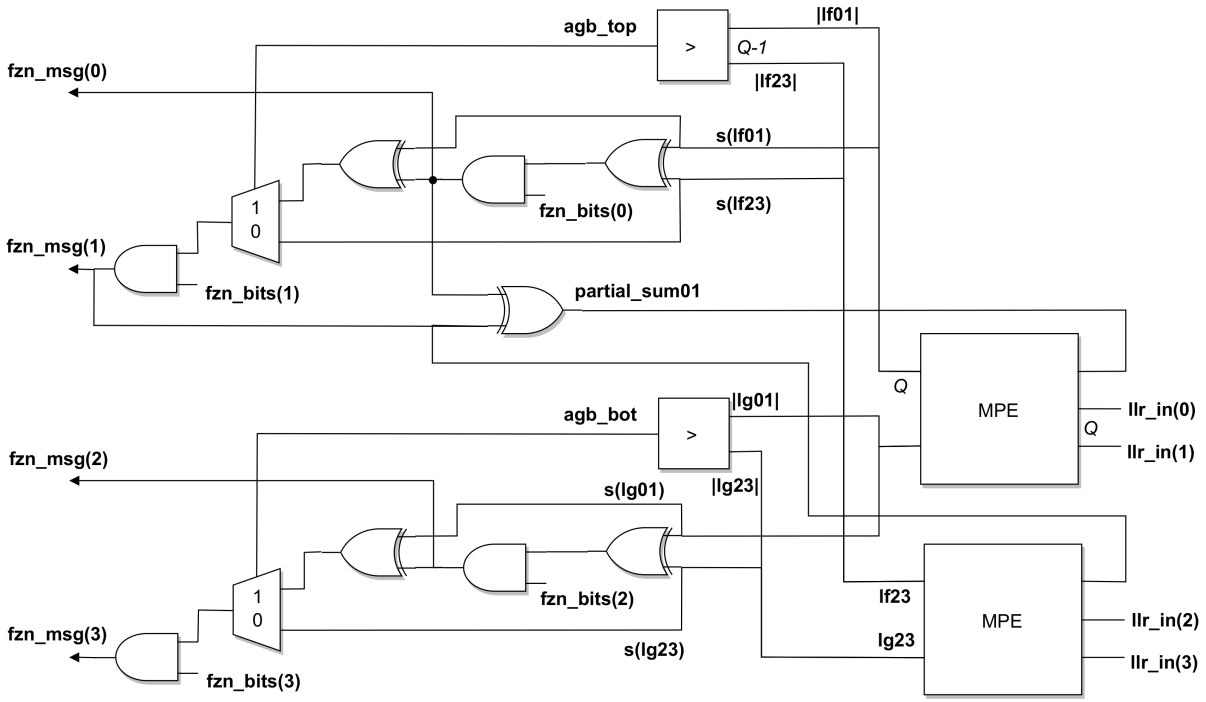


Figura 3.13: Bloques y señales dentro del decodificador básico

performance de decodificación empeorará. En la salida del decodificador debe existir un registro para almacenar los bits que son estimados por el decodificador.

Para implementar los registros se utilizan N flip-flops para los LLR, de capacidad de Q bits cada uno, y para los registros restantes se utilizan flip-flops de capacidad unitaria.

El control de dichos registros de almacenamiento es externo al decodificador, y es efectuado por una máquina de estado, parte del sistema de pruebas que se describirá en el siguiente capítulo.

3.6. Desarrollo del sistema

3.6.1. Diagramas RTL y simulaciones funcionales

El desarrollo de un sistema complejo requiere de un procedimiento de depuración de errores en varios niveles jerárquicos. Éste implica ciclos de pruebas unitarias y de integración de complejidad creciente, que involucran la detección y corrección de errores en el diseño y la programación. Para facilitar este procedimiento, se programó desde un principio siguiendo el estilo de programación estructural, y se modularizó lo más posible el código en VHDL. El código para todas las entidades del codificador y decodificador se hizo de forma genérica, sin utilizar bibliotecas de Intel o Altera, para permitir al usuario final la portabilidad del mismo sin modificaciones a FPGAs de otos fabricantes. Los ciclos de prueba se llevaron a cabo desde los niveles de entidades simples de VHDL hasta el nivel de integración total del sistema, cuyos resultados serán el foco del capítulo quinto del informe.

La programación de los distintos módulos o entidades en VHDL se hizo con el entorno de programación integrado Intel Quartus Prime Lite 17.1, ofrecido de forma gratuita por el fabricante de la FPGA. Este IDE permite realizar todos los pasos necesarios para pasar desde un archivo de código fuente en lenguaje de descripción de hardware a un archivo binario de configuración de FPGA.

El primer paso tras realizar una compilación en Quartus es corregir los eventuales errores de sintaxis que éste detecte. Una vez completada exitosamente la operación de análisis y

síntesis del diseño, este IDE ofrece una herramienta para visualizar la interpretación RTL que el compilador hizo del mismo. Con esta herramienta pueden visualizarse los puertos, señales internas y entidades involucradas. También existe en el visualizador la posibilidad de alcanzar el nivel de los elementos lógicos básicos, como las compuertas, los multiplexores y los comparadores. Otra posibilidad es la de visualizar los estados y transiciones de las FSM que el compilador pudiera encontrar.

A lo largo del desarrollo se utilizó esta herramienta visual con todas las entidades programadas, para comprobar que Quartus interpretaba correctamente el código de las mismas. En carácter ilustrativo, se expone en la Fig. 3.14 la visualización RTL de Quartus para uno de los MPE que componen el decodificador en la versión final del sistema.

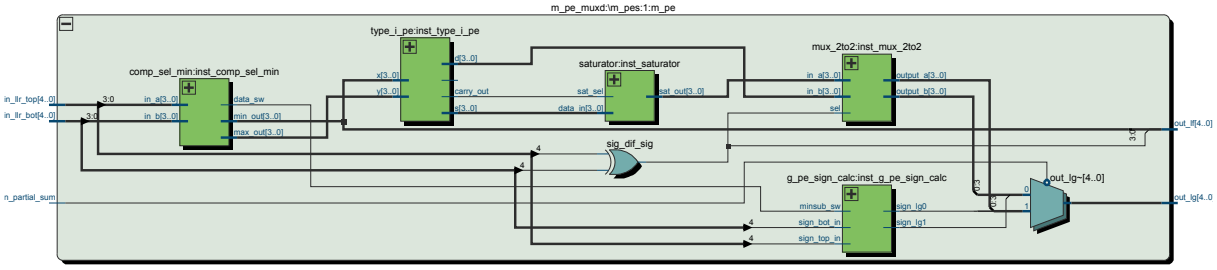


Figura 3.14: Visualización RTL en Quartus de un elemento de procesamiento combinado

Otra característica destacable de Quartus es que ofrece un simulador funcional de la lógica implementada. El mismo permite, tras completar la etapa de análisis y síntesis de la compilación, probar el funcionamiento del diseño en VHDL para distintas entradas. Además, permite especificar gráficamente las formas de onda de las entradas, y obtener tras la simulación las formas de onda de las salidas. Se ejemplifica en la Fig. 3.15 el simulador, con el resultado de la prueba funcional del bloque corrector de ceros negativos.

El entorno de pruebas se utilizó para depurar la mayoría de las entidades del proyecto. Todas las entidades de jerarquías bajas fueron simuladas, y para entidades más complejas como el decodificador, se hicieron pruebas con algunas entradas para valores bajos de longitud de mensaje, como 8 y 16. Para las pruebas de integración final fue necesario implementar un sistema de pruebas automático de alta velocidad, que procesara miles de mensajes por segundo. Este sistema de pruebas se trata en el próximo capítulo.

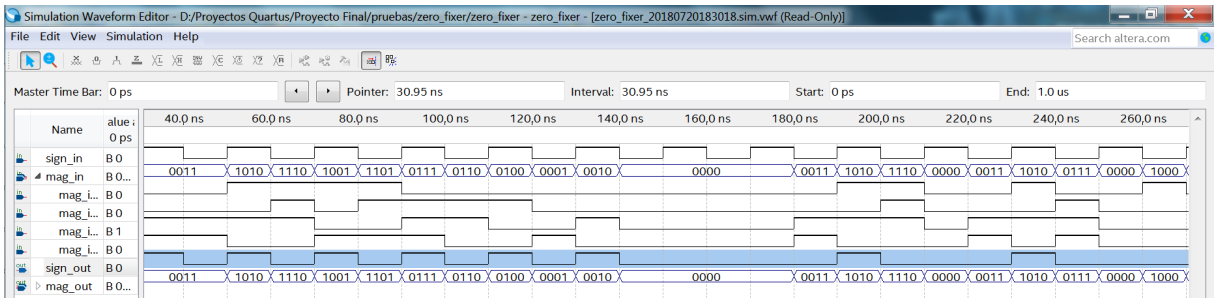


Figura 3.15: Resultado de la simulación funcional del bloque corrector de ceros

3.6.2. Resultados de síntesis

Se presentan a continuación tablas con los resultados de síntesis obtenidos con Quartus con la opción de compilación *Performance – High effort*, para maximizar el throughput sin aumentar el consumo de recursos de hardware. El IDE ofrece también opciones de compilación que favorecen otros resultados de síntesis. Éstas opciones permiten compilaciones balanceadas, optimizadas para reducir el consumo de recursos, o para reducir la potencia requerida. Algunas de estas opciones pueden incrementar considerablemente los tiempos de compilación.

Los resultados de esta sección son tomados del reporte del fitter de Quartus. Las compilaciones se realizan utilizando pines virtuales para lograr resultados cuando el valor de N es elevado. Para estos valores la cantidad real de pines del dispositivo utilizado impone un límite a la compilación.

Las frecuencias máximas de operación son las estimadas con un modelo de temporización lento, una tensión de alimentación de la FPGA de 1100 mV y una temperatura de operación de 85 ºC.

Los valores de throughput son teóricos y calculados a partir de las frecuencias estimadas; las estimaciones corresponden al procesamiento de N bits. El throughput efectivo de la información sin procesar depende de la tasa de código empleada, y su valor es $TP_{ef} = TP \times Rc$.

Codificadores

Longitud de mensaje	ALMs	LUTs	Registros	f_{MAX} (MHz)	Throughput (Gbps)
4	4	5	8	717,36	2,86
8	13	8	16	717,36	5,73
16	27	22	32	524,38	8,39
32	56	49	64	445,63	14,26
64	118	107	128	341,88	20,15
128	252	239	256	316,36	44,33
256	583	567	512	247,65	63,39
512	1.242	1.283	1.024	217,3	111,25
1024	2.656	2.825	2.048	190,48	195,05
2048	5,244	5.696	4.096	151,54	310,35
4096	11.227	12.451	8.192	140,13	573,97
Máximo disponible	41.910	110.000	83.820	–	–

Cuadro 3.3: Resultados de síntesis del codificador para distintas longitudes de mensaje

En la Tabla 3.3 se encuentran los resultados de síntesis obtenidos para codificadores combinacionales para distintas longitudes de mensajes. Se aprecia en la tabla que hay incrementos cuasi lineales de ALMs y tablas de búsqueda. Los registros, de un bit cada uno, corresponden exactamente con las necesidades de almacenamiento de la entrada y de la salida de los codificadores.

La máxima frecuencia de operación estimada es inversamente proporcional a la complejidad del codificador. Con el aumento lineal en la complejidad, las frecuencias decaen en menor medida. Por último, las velocidades escalan de manera aproximadamente lineal, ya que el incremento en N compensa siempre la caída en la frecuencia máxima. Son destacables los muy elevados valores de throughput que los codificadores combinacionales alcanzan, especialmente a valores mayores de N.

El codificador más complejo que se logró sintetizar para el dispositivo utilizado en el proyecto corresponde a $N = 4096$. Si bien el número de recursos disponibles en la FPGA parece permitir una eventual síntesis para $N = 8192$, debe tenerse en cuenta la gran necesidad de ruteo de señales que implica un codificador de semejante longitud, que implica la imposibilidad de lograrla prácticamente.

Elementos de procesamiento combinado

En la Tabla 3.4 se presentan los resultados de síntesis para las tres variantes de elementos de procesamiento diseñadas en este proyecto. Los resultados se presentan para distintas cantidades de bits de cuantificación para los LLR.

Puede extraerse de la Tabla 3.4 que el MPE original y el simplificado requieren el mismo número de LUTs. La variante con corrección de ceros negativos, inesperadamente, requiere menos LUTs que las otras. Una posible explicación de esto es que el agregado de los bloques

correctores permita al compilador hacer simplificaciones en la lógica implementada, requiriendo menos LUTs o permitiendo fraccionar las tablas de una forma más eficiente. A pesar de esto, el requerimiento de ALMs de esta variante es superior en el caso para $Q = 6$.

Si se comparan los consumos de ALMs de las variantes original y simplificada, se encuentra que los mismos varían en función de Q. Para cinco bits es menor el consumo para el MPE original, mientras para seis, la simplificación de los multiplexores permite un ahorro de dos módulos configurables.

Al analizar las máximas frecuencias, se encuentra que todas están en el mismo orden para los distintos valores de Q. La reducción del camino critico real de g en la variante simplificada, no se traduce en un incremento de la frecuencia máxima apreciable. La variante con corrección de ceros negativos no muestra frecuencias menores que sus contrapartidas. Ésto se debe a las simplificaciones que logra el compilador, que se traducen también en el acortamiento del camino crítico estimado.

Los resultados anteriores son en definitiva dependientes de la tecnología del dispositivo que los implementa. Los requerimientos de celdas lógicas y lookup tables, así como también las frecuencias máximas de operación, variarán en función de la estructura interna de los módulos lógicos configurables y de la lógica de ruteo disponible en aquellos dispositivos.

	Original			Simplificado			Corrector de ceros		
Q	ALMs	LUTs	f_{MAX}(MHz)	ALMs	LUTs	f_{MAX}	ALMs	LUTs	f_{MAX}
4	6	14	535,62	6	14	514,67	6	11	535,33
5	7	19	347,22	8	19	359,97	8	17	399,2
6	16	34	311,04	14	34	308,93	16	30	295,77

Cuadro 3.4: Resultados de síntesis de los elementos de procesamiento combinado diseñados

Los elementos de procesamiento diseñados se comparan en la Tabla 3.5 con un diseño de referencia. Esta referencia se toma en consideración ya que utiliza también la representación en signo y magnitud para los LLR, y presenta sus resultados para distintas cantidades de bits de cuantificación [31]. Adicionalmente, implementa el MPE en una FPGA Intel Stratix IV, cuyas ALMs son idénticas a las usadas en este trabajo y se expusieron esquemáticamente en la Fig. 3.1. Este diseño de MPE es el mismo que se utiliza en la arquitectura semi-paralela, también para el algoritmo SCD [5]. Se omite en la comparación la variante con corrección de ceros, ya que simulaciones funcionales mostraron diferencias entre ésta y la referencia. Estas diferencias correspondieron, para las mismas entradas, a la existencia ceros negativos en las salidas de la referencia, y a la inexistencia de éstos en la variante correctora.

	Original		Simplificado		Referencia	
Q	LUTs	f_{MAX}(MHz)	LUTs	f_{MAX} (MHz)	LUTs	f_{MAX} (MHz)
5	19	347,22	19	359,97	23	381
6	34	311,04	34	308,93	27	368
7	36	357,4	39	273,07	31	361
8	38	289,52	39	299,85	35	355
20	118	139,94	119	145,88	100	269

Cuadro 3.5: Resultados de síntesis de los elementos de procesamiento combinado diseñados y de la referencia

Observando la tabla, es claro que el MPE de referencia requiere un consumo de LUTs inferior, y que sus frecuencias máximas de operación son mayores en todos los casos. Sin embargo, el diseño de referencia no utiliza pre-cómputo, y su camino crítico efectivo para la implementación de g es mayor.

Las diferencias en los consumos de recursos pueden entenderse al considerar que la referencia utiliza un diseño específico para un dispositivo. Por lo tanto, puede hacer uso de las bibliotecas

del fabricante, que permiten implementar las sumas y restas con los sumadores completos que se encuentran en las ALMs. En oposición, este proyecto sigue un criterio de código VHDL genérico y no utiliza bibliotecas del fabricante. En consecuencia, los MPE diseñados en este trabajo requieren de LUTs para efectuar las sumas y restas que resultan en un consumo de recursos de hardware mayor.

Decodificadores

Es importante comparar también el elemento singularmente más complejo del sistema de corrección de errores con alguna de las implementaciones publicadas. Se comparan a continuación los resultados de síntesis del decodificador con los del trabajo que propuso originalmente la arquitectura combinacional [28]. La referencia presenta resultados obtenidos en una FPGA del fabricante Xilinx.

El decodificador de referencia comparte varias características con el utilizado en este proyecto. Ambos tienen la misma arquitectura combinacional, que aprovecha la recursividad en la decodificación utilizando códigos polares. Además, los dos trabajos utilizan la misma cuantificación de formato signo y magnitud con $Q = 5$ bits.

Los resultados de síntesis se muestran en la Tabla 3.6.

	Este proyecto				Referencia		
N	LUTs	RAM (bits)	f_{MAX} (MHz)	TP (Mbps)	LUTs	RAM (bits)	TP (Mbps)
8	209	56	88,77	710,1	–	–	–
16	625	112	39,04	626,2	1.479	112	1.050
32	1.682	224	17,68	565,7	1.918	224	880
64	4.308	448	8,32	532,4	5.126	448	850
128	9.595	896	3,7	473,6	14.517	896	820
256	22.782	1.792	1,68	430	35.152	1.792	750

Cuadro 3.6: Resultados de síntesis del decodificador diseñado y del decodificador de referencia

En la Tabla 3.6 se presentan por un lado, todos los decodificadores que se pudieron sintetizar en el proyecto, con el código fuente en VHDL desarrollado. La referencia no presenta resultados para $N = 8$. Cabe aclarar que los valores de throughput correspondientes a este proyecto se obtuvieron a partir de las máximas frecuencias estimadas por Quartus. En cambio, los de la referencia son valores reales obtenidos de la implementación.

En primer lugar, es destacable que el consumo de recursos de hardware del decodificador diseñado en este proyecto es notablemente inferior al de la referencia. En segundo lugar, resalta el hecho de que la implementación de referencia logra alcanzar velocidades considerablemente más altas. Por último, es apreciable que ambas muestran idénticos requerimientos de registros, ya que comparten la arquitectura y sólo utilizan registros en sus interfaces externas.

Para determinar si la discrepancia se debía a la opción de compilación en Quartus, se repitió la síntesis con la opción *Performance – Agressive*, que además de aumentar el tiempo de compilación incrementa el consumo de recursos en pos de lograr mayores velocidades. El resultado obtenido con esta opción para $N = 128$ dio una velocidad de 478,7 Mbps y un consumo de 9.708 LUTs. Por lo tanto, la opción de compilación no explica las diferencias encontradas, al menos para ese valor de N.

Otra posible explicación de la diferencia en consumo de hardware puede encontrarse en los elementos de procesamiento, ya que representan una gran parte de los bloques dentro de los decodificadores combinacional. La referencia no explicita el diseño de sus elementos de procesamiento, por lo que no es posible compararlos con los de este trabajo.

Como se verá en el capítulo próximo, el decodificador diseñado muestra en la práctica velocidades superiores a las estimadas con Quartus y a las mostradas en la referencia.

Por último, hace notarse que los resultados de ambos trabajos reflejan una ventaja importante de la arquitectura combinacional frente a las arquitecturas sincrónicas. La operación a relativamente bajas frecuencias y la ausencia de registros internos permiten al decodificador operar con consumos reducidos de potencia.

Capítulo 4

Sistema de pruebas

4.1. Implementación System On Chip

4.1.1. Consideraciones iniciales

Para desarrollar el sistema de codificación y decodificación se utilizó una placa de desarrollo Terasic modelo DE-10 Standard. Dos unidades de esta placa estuvieron disponibles desde el comienzo del proyecto en el Laboratorio de Comunicaciones del Departamento de Ingeniería Electrónica y Computación. La placa está basada en una FPGA Intel modelo Cyclone V 5CSXFC6D6F31C6N, del tipo *System On Chip* (SOC). El encapsulado de la FPGA integra un procesador dedicado tipo HPS, de dos núcleos y arquitectura ARM de 32 bits. Además del circuito integrado principal, la placa contiene diversos periféricos, interfaces y puertos, que se muestran en la Fig. 4.1.

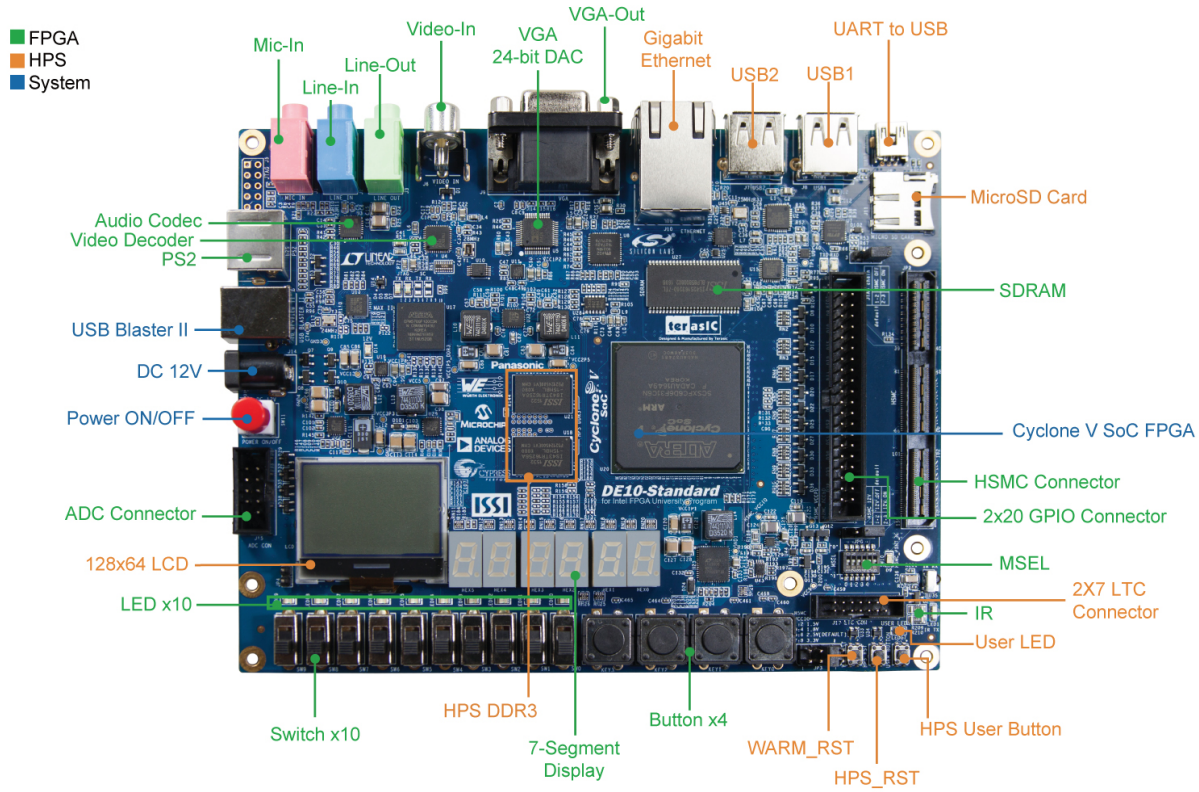


Figura 4.1: Placa de desarrollo y sus principales componentes

Como se aprecia en la Fig. 4.1, la FPGA no cuenta con un acceso directo a puertos para comunicación dúplex con el exterior. Una forma de replicar la función de comunicación sería utilizando los pines de propósito general (GPIO) en conjunto con lógica de control sintetizada en FPGA. Otro modo sería hacer uso de los puertos en la placa como USB o Ethernet, utilizando

el procesador como intermediario, ya que tiene las conexiones a los periféricos que controlan dichos puertos.

Para comprobar el funcionamiento de sistemas correctores de errores suele utilizarse el método Monte Carlo. Éste implica, para esta implementación, enviar a la FPGA un número grande de mensajes para su procesamiento y posterior análisis estadístico de resultados.

Inicialmente se pensó en hacer uso del HPS como simple intermediario y conectar la placa a una computadora mediante comunicación por USB. En este caso la computadora estaría entonces encargada de controlar un esquema de comunicaciones similar al ya visto en el primer capítulo, en la Fig. 2.1. Un programa ejecutándose en dicha computadora, como MATLAB, debería trabajar con múltiples repeticiones del siguiente ciclo:

1. Generación de un mensaje aleatorio en la computadora.
2. Envío por puerto serie del mensaje a la FPGA para codificación.
3. Recuperación del mensaje codificado.
4. Modulación del mensaje y agregado de ruido presente en una simulación de un canal de comunicaciones.
5. Cálculo y envío por puerto serie de los LLR necesarios para el decodificador.
6. Recuperación del mensaje decodificado.
7. Comparación del mensaje decodificado con el mensaje inicialmente generado, para determinar la existencia y el número de posibles diferencias.

La repetición del ciclo anterior, un número lo suficientemente grande de veces, permitiría estimar las probabilidades de que el decodificador cometiera errores. Comprobar probabilidades de error de decodificación muy bajas requerirían el procesamiento de millones de mensajes. Si se utilizara el esquema de pruebas mediante computadora, sería necesario efectuar cuatro comunicaciones serie por cada mensaje a probar.

La placa de desarrollo utilizada cuenta con el integrado FT232R para control de comunicación USB. Este integrado implementa un Transmisor-Receptor Asíncrono Universal (UART) que soporta velocidades de comunicación no estándar de hasta 3 Gbaud. Sin embargo, el UART está limitado por una conexión USB 2.0 que soporta velocidades bajas, de hasta 12 Mbps [32]. El throughput de este puerto es muy bajo en comparación a lo que puede alcanzar el sistema codificador–decodificador.

Otra opción que ofrece la placa es el puerto Gigabit Ethernet. El throughput de este puerto es mucho mayor que el del puerto USB. Sin embargo, para una comunicación con éste, es necesario sintetizar en la FPGA un controlador de la comunicación, y es además necesario idear un protocolo de comunicación a nivel de aplicación. Los paquetes que se intercambian deben ser procesados en ambos extremos de la comunicación para poder extraer la información relevante que contienen.

Teniendo en consideración lo anterior, se decidió implementar el esquema de pruebas directamente en el HPS. Esta opción tiene tres ventajas principales.

En primer lugar, la FPGA se comunica con el HPS de forma interna mediante puentes configurables de muy alta velocidad [33]. Estos puentes permiten el movimiento rápido y directo de datos, sin *overhead*, y evitando el desarrollo de protocolos a nivel de aplicación. Con ésto se pueden reducir los intervalos de tiempo ociosos para la FPGA.

En segundo lugar, el sistema de pruebas permite independizarse de la computadora. En un caso de uso, sólo sería necesario establecer inicialmente, desde la computadora, los parámetros de la prueba y correr un programa sobre un sistema operativo embebido. La placa podría entonces desconectarse y mantenerse en funcionamiento durante horas o días, trabajando de forma continua a alta velocidad. Una vez finalizadas las pruebas, se podría recuperar del sistema

un archivo en el que se encuentren almacenados los resultados de la prueba y notificar al usuario el fin de la misma mediante periféricos como LEDs.

Finalmente, el procesador 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.

Una posibilidad para integrar el sistema final era sintetizar el codificador en una placa de desarrollo, usar la computadora para simular el canal de comunicaciones, e implementar el decodificador en una segunda placa. Los resultados de síntesis del codificador y del decodificador mostraron que el consumo de hardware era predominantemente debido al decodificador. En base a ésto se estimó que el sistema completo para una longitud de mensaje de hasta 256 bits podría ser implementado en una sola placa de desarrollo. La utilización de dos placas interconectadas era por lo tanto innecesaria y hubiese probablemente requerido una mayor complejidad de desarrollo. Además, el uso de una sola placa era deseable teniendo en cuenta la disponibilidad de sólo dos unidades en el Laboratorio de Comunicaciones. El desarrollo de un sistema que intercomunicara dos placas debería ser modificado frente a eventualidades que dejaran a una de las placas fuera de servicio o indisponibles para el uso. En contrapartida, el desarrollo de un sistema que requiriera una sola placa permitiría seguir trabajando en el proyecto sin retrasos, gracias a la disponibilidad de una unidad idéntica de repuesto ante eventuales inconvenientes con una de las placas.

Por último, una implementación que hiciera uso intensivo 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 y parecen ser una tendencia en el mercado de los sistemas embebidos.

4.1.2. Arquitectura del sistema de pruebas

En base a las consideraciones hechas hasta aquí, en la placa se implementó finalmente un sistema de pruebas cuya arquitectura se encuentra en la Fig. 4.3. La arquitectura se diseñó pensando en la utilización por parte un usuario final experto como, por ejemplo, un diseñador que integra sistemas de comunicaciones, o un investigador en las materias de códigos de control de errores y comunicaciones digitales. Las posibles interacciones de los destinatarios del sistema con el mismo se exponen en la Fig. 4.2.

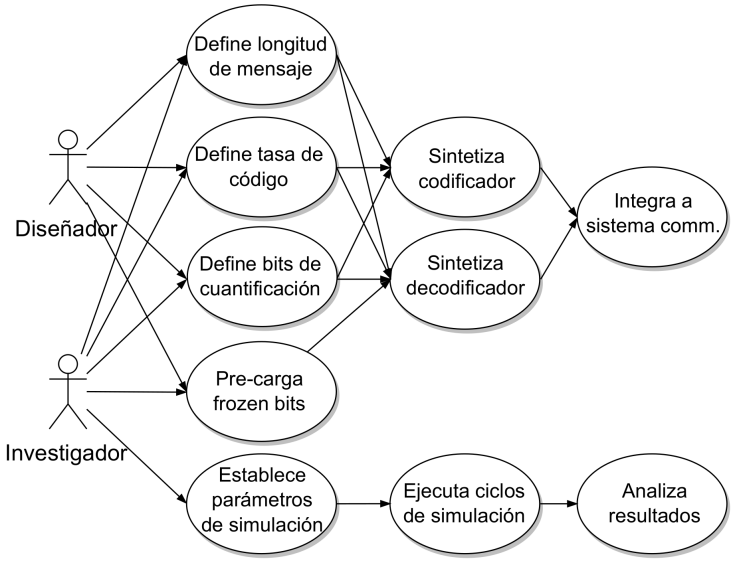


Figura 4.2: Diagrama de usuarios y de uso de los componentes del sistema

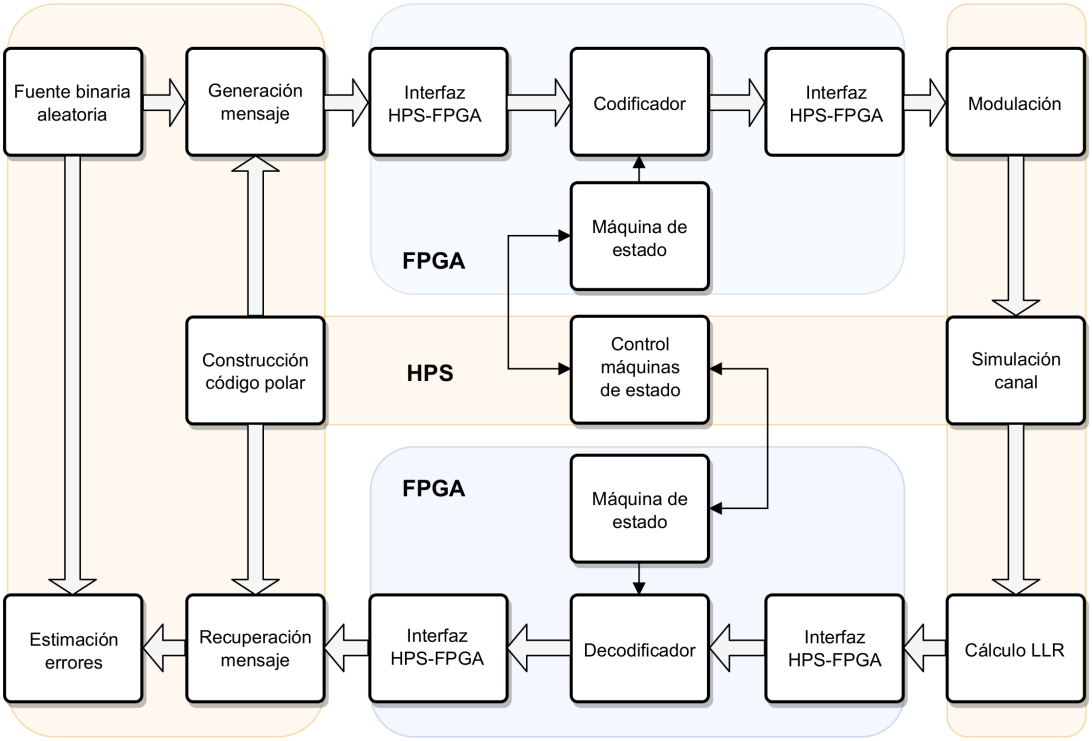


Figura 4.3: Arquitectura del sistema de pruebas implementado

El sistema de la Fig. 4.3 permite implementar el ciclo de pruebas presentado en la subsección anterior, en el menor tiempo posible. Dicho sistema cuenta de tres componentes principales. Por un lado, el codificador y el decodificador se encuentran sintetizados en la FPGA junto a sus interfaces, elementos de memoria y sistemas de control. Por otra parte, el HPS provee el soporte físico para comunicarse con la FPGA y correr el sistema operativo en conjunto con el programa que efectúa las pruebas.

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) [33]. 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.

En este sistema el procesador es el encargado del control general de todo el conjunto. Para ello se implementan en total tres 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 dos sistemas como máquinas de estado finito (FSM), esclavos del maestro en el 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 comunicación entre el maestro y los esclavos se efectúa con cuatro registros dedicados, conectados a través del puente ligero HPS-FPGA. Cada FSM lee en cada ciclo de reloj el valor almacenado en un registro, y establece la transición adecuada entre los estados en base a dicho valor y al estado actual. Simultáneamente responde al HPS escribiendo el estado actual en el registro correspondiente. El sistema maestro funciona de forma similar, con sus registros

correspondientes, pero se diferencia como maestro teniendo la capacidad de iniciar y finalizar las comunicaciones, teniendo el control absoluto sobre lo que ocurre en la FPGA.

Para el almacenamiento de las entradas y salidas del codificador y del decodificador, el sistema utiliza los mismos tipos de RAM, generados a partir de bloques de memoria dedicados en la FPGA. Dichos bloques no ocupan espacio en la lógica configurable de la placa pero su número es limitado. Permiten, sin embargo, ser configurados para distintos anchos y cantidades de palabras. Para la implementación se utilizan memorias de dos puertos y dos señales independientes de reloj; en todas ellas un puerto de 64 bits se comunica de forma nativa a través del bus rápido con el procesador. El otro puerto, de 128 bits, se conecta con los registros que forman parte de la lógica del codificador o del decodificador. Las FSM se encargan del control de las señales que manejan las memorias, respetando las formas de onda de la Fig. 4.4. A través de la comunicación mediante los registros de control, se asegura que la FPGA no escriba sobre una dirección de memoria que está siendo leída por el HPS y viceversa. Así el sistema evita corromper los datos almacenados y las consecuencias catastróficas que esto tendría para la performance de corrección del sistema de control de errores.

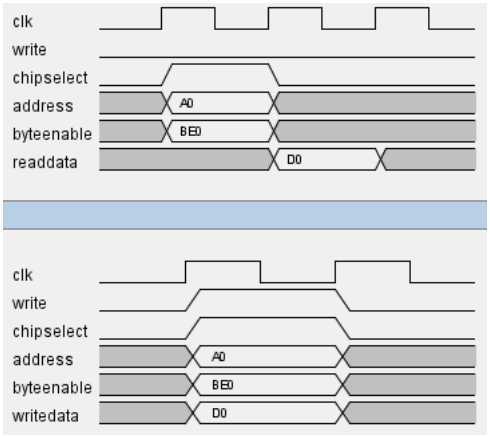


Figura 4.4: Formas de onda de las interfaces Avalon para lectura y escritura de memoria

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. Para el decodificador, la latencia de decodificación 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 un nuevo mensaje codificado está disponible a la entrada y la orden de decodificación es recibida; mientras el conteo se lleva a cabo, la memoria de salida se encuentra deshabilitada. Una vez finalizado el conteo, la FSM habilita la escritura de las memorias de salida, y notifica al programa la disponibilidad de un nuevo mensaje decodificado, listo para ser enviado a través del bus rápido. Para el codificador, la latencia es generalmente menor que el período de reloj con el que trabaja la FSM, por lo tanto, el sistema trabaja sin el conteo para el mismo; el conteo puede sin embargo habilitarse accediendo al código fuente VHDL, cambiando el valor de una constante en el paquete de configuración.

Las máquinas de estado del codificador y del decodificador son semejantes. Éstas consisten, en forma resumida, de un estado inicial de espera de órdenes, uno de espera de codificación o decodificación, uno de finalización de tareas y un estado especial que se alcanza al fallar en la interpretación de órdenes. La coordinación del procesador y de la FPGA a través de este sistema de control combinado permite 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.

4.1.3. Desarrollo del software

Para diseñar el sistema completo se trabajó en paralelo con el procesador y con la FPGA. Del lado del procesador, se decidió elaborar un programa que implementara mediante funciones las tareas de los módulos de la Fig. 4.3, corriendo sobre un sistema operativo OS. Si bien la operación del OS consume recursos del procesador, la programación es más simple que la realizada 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, e implican generalmente tiempos de desarrollo más largos [34].

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. Ésto abre la posibilidad de acceder de forma remota al sistema, desde otras localizaciones geográficas. La versión de Ångstrom 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 [34] [35] [36]. En este proyecto se utilizó también la posibilidad de cargar la configuración de la FPGA desde la tarjeta microSD. Para esto, es antes necesario ejecutar un script, provisto por Intel, que permite pasar de un archivo de extensión *.sof*, generado en Quartus con cada compilación, a un archivo binario *.rbf* de configuración. Esto facilita el procedimiento de configuración de la FPGA, y abre la posibilidad de cargar configuraciones desde el sistema operativo de forma dinámica, según los requerimientos de la aplicación.

El programa que corre sobre el sistema operativo se compuso en lenguaje C utilizando dos entornos de desarrollo separados. El programa principal se desarrolló en el IDE Eclipse ARM DS-5 versión 5.27.1. Esta versión del IDE Eclipse es una modificación de la empresa ARM para el desarrollo de sistemas embebidos basados en sus microprocesadores. Para el desarrollo se solicitó a ARM por correo electrónico una licencia gratuita, con la limitación de sólo poder hacer depuración de programas para Linux a través de una conexión Ethernet.

En segundo lugar, se programaron y testearon, con el IDE gratuito Code::Blocks versión 17.12, funciones necesarias para el sistema de pruebas. Las pruebas individuales de estas funciones se hicieron en este IDE. A medida que fueron superadas las pruebas, las versiones finales de estas funciones fueron integradas, progresivamente, al programa principal en Eclipse. Finalmente se probaron las mismas ejecutándolas en la placa DE-10 y observando los resultados en el entorno de depuración Ethernet de Eclipse.

Para desarrollar el programa se trabajó además con el código en lenguaje C++ de un sistema de simulación de códigos polares, desarrollado en el Laboratorio de Comunicaciones. En base a este código se programaron también funciones portables al programa principal en Eclipse, y se depuraron errores en el sistema corrector mediante comparaciones entre decodificación por hardware y por software. Finalmente, se hicieron modificaciones a este código para efectuar simulaciones que se presentan en el siguiente capítulo.

Por último, se trabajó con un paquete de simulación de códigos polares en MATLAB versión R2015a [37]. Con este paquete se efectuaron simulaciones y pruebas necesarias para el desarrollo y depuración del programa principal.

El programa desarrollado implementa los módulos correspondientes al HPS en la arquitectura que se expone en la Fig. 4.3, ofreciendo las siguientes características funcionales:

- Control de un sistema implementado en la FPGA.
- Control de los frozen bits almacenados en la FPGA.
- Interfaz gráfica implementada en texto para interacción mediante terminal en Linux, desde una computadora. Ingreso de comandos por teclado.
- Posibilidad de mostrar todos los códigos polares que se encuentren almacenados en la FPGA.
- Posibilidad de simular codificación y decodificación con códigos polares con la configuración por parte del usuario, en tiempo de ejecución, de los siguientes parámetros: $16 \leq N \leq 128$, $K \leq N$, parte entera Q_i y parte fraccionaria Q_f para una cuantificación tipo punto fijo SM con $Q = 5$ bits.
- Posibilidad de establecer un canal de comunicaciones tipo AWGN o BSC con un rango de estados asociado.
- Posibilidad de establecer un número máximo de mensajes a probar por estado de canal.
- Cálculo de los LLR, cuantificación SM de los mismos y manipulación de las palabras para comunicación de datos desde y hacia la FPGA.
- Construcción de códigos polares mediante límites de Bhattacharyya con parámetros ilimitados N y K, para un tipo de canal y rango de estados.
- Construcción y transmisión a la FPGA de códigos polares de forma automática, si éstos no se encontraran inicialmente en memoria.
- Posibilidad de verificar que en la FPGA se encuentre programado, a partir del archivo binario de configuración, el sistema corrector de errores.
- Almacenamiento de parámetros, probabilidad binaria de error, probabilidad de error por bloque y otras notificaciones en archivo de texto.
- Cálculo de tiempos totales y promedio por mensaje para los ciclos de pruebas.
- Notificación al usuario de la cantidad de mensajes ya procesados para el estado actual del canal.
- Notificación al usuario de eventuales errores en la comunicación con las máquinas de estado en la FPGA.

El programa permitiría también ser utilizado con un sistema corrector de errores con códigos polares para $N = 256$ en la FPGA. Con leves modificaciones a los códigos en C y en VHDL, y en el entorno de integración del sistema, puede soportar cuantificaciones tipo punto fijo SM con cualquier número Q de bits. El sistema implementado en FPGA debería ser modificado para poder usar estas características.

4.1.4. Integración del sistema

Para desarrollar los sistemas de control e interfaces alrededor del sistema de corrección de errores en la FPGA, se hizo uso de la herramienta denominada *Platform designer*, disponible en el IDE de licencia gratuita Quartus Prime Lite versión 17.1. Esta herramienta presenta un entorno gráfico que permite integrar el HPS a un sistema que utiliza una FPGA. A través de dicho entorno se pueden crear y configurar las interfaces entre el HPS y la FPGA. El mismo

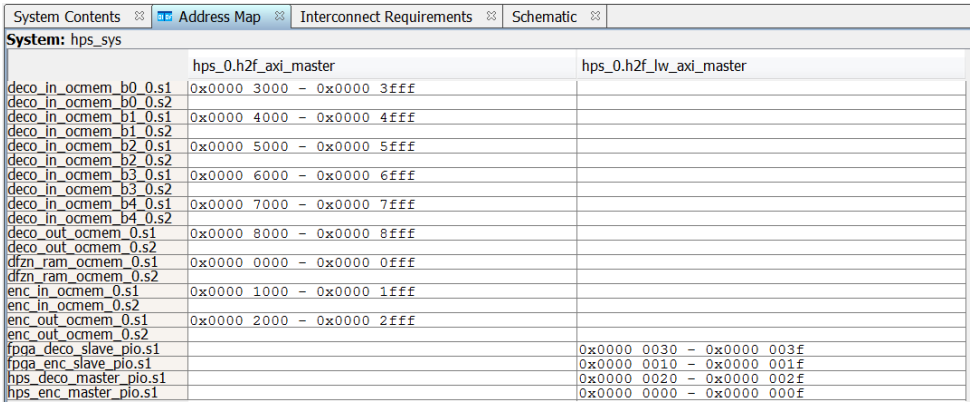
permite también integrar bloques parametrizables, implementados con propiedad intelectual del fabricante de la FPGA y sintetizados a partir de código VHDL o Verilog generado de forma automática. Platform designer se ocupa de generar el código de módulos de propiedad intelectual sintetizables que se encargan del manejo de las interfaces con el procesador, utilizando los protocolos AMBA AXI y Avalon. Ésto permite al diseñador abstraerse de los aspectos más complicados de la comunicación interna en un sistema sofisticado como el de la FPGA SOC.

Para el desarrollo del sistema completo, en este trabajo se partió de un nuevo proyecto en Platform designer. En primer lugar se configuraron características como el ancho de los buses y las frecuencias y *timings* de las memorias, estos últimos obtenidos del proyecto de referencia *Golden Reference Hardware Design* (GHRD) provisto por Terasic en el paquete de soporte.

A continuación se crearon los registros del sistema de control y los módulos de RAM y se conectaron a los buses correspondientes. Para la memoria que almacena los frozen bits, se creó un archivo de configuración en formato Intel HEX. Dicho archivo pre-carga ciertos códigos polares en la memoria, al momento de configurar la FPGA a partir del archivo binario en la tarjeta microSD, emulando el funcionamiento de una ROM. Los códigos se encuentran así disponibles de forma inmediata en el decodificador.

A continuación se creó una señal de reloj de referencia de 50 MHz para las FSM, los registros en el sistema corrector de errores y los puertos de RAM de 128 bits visibles desde la FPGA. Con un módulo PLL se generó una frecuencia de 100 MHz para las interfaces AXI lentas y rápidas, y los puertos de RAM de 64 bits visibles desde el HPS.

El último paso fue establecer el mapa de direcciones de memorias y registros de estado visibles desde el procesador, tomando la forma que se aprecia en la Fig. 4.5. Estas direcciones son las que el programa utiliza como offsets, a partir de las direcciones de base de los buses, para el mapeo de memoria. Intel provee un script que permite, a partir del sistema diseñado en Platform designer, generar un archivo de encabezado con los datos necesarios para un mapeo de memoria correcto.


System Contents | Address Map | Interconnect Requirements | Schematic

System: hps_sys

	hps_0.h2f_axi_master	hps_0.h2f_lw_axi_master
deco_in_ocmem_b0_0.s1	0x0000 3000 - 0x0000 3fff	
deco_in_ocmem_b0_0.s2		
deco_in_ocmem_b1_0.s1	0x0000 4000 - 0x0000 4fff	
deco_in_ocmem_b1_0.s2		
deco_in_ocmem_b2_0.s1	0x0000 5000 - 0x0000 5fff	
deco_in_ocmem_b2_0.s2		
deco_in_ocmem_b3_0.s1	0x0000 6000 - 0x0000 6fff	
deco_in_ocmem_b3_0.s2		
deco_in_ocmem_b4_0.s1	0x0000 7000 - 0x0000 7fff	
deco_in_ocmem_b4_0.s2		
deco_out_ocmem_0.s1	0x0000 8000 - 0x0000 8fff	
deco_out_ocmem_0.s2		
dfzn_ram_ocmem_0.s1	0x0000 0000 - 0x0000 0fff	
dfzn_ram_ocmem_0.s2		
enc_in_ocmem_0.s1	0x0000 1000 - 0x0000 1fff	
enc_in_ocmem_0.s2		
enc_out_ocmem_0.s1	0x0000 2000 - 0x0000 2fff	
enc_out_ocmem_0.s2		
fpga_deco_slave_pio.s1		0x0000 0030 - 0x0000 003f
fpga_enc_slave_pio.s1		0x0000 0010 - 0x0000 001f
hps_deco_master_pio.s1		0x0000 0020 - 0x0000 002f
hps_enc_master_pio.s1		0x0000 0000 - 0x0000 000f

Figura 4.5: Mapa de direcciones de registros y memorias usados en el sistema de pruebas

4.1.5. Interfaces de usuario

El sistema completo cuenta con dos interfaces de usuario. La primera es la interfaz gráfica de texto que permite al usuario realizar las funciones enumeradas anteriormente en la sección "Desarrollo del software".

La segunda interfaz es provista por la placa de desarrollo, y utiliza algunos de los variados dispositivos de visualización y conexión que ésta ofrece. Esta interfaz es implementada directamente por el sistema en FPGA y hace uso de algunos de los displays de siete segmentos (denominados HEX), LEDs, switches y botones que se encuentran en la Fig. 4.1.

El primer display HEX a la izquierda muestra al usuario la orden actual recibida por la FSM del codificador. Las posibilidades son:

1. **0** $\implies$ **Reset**: orden de pasar al estado de reset.

2. **E** $\Longrightarrow$ **Encode**: orden de codificar el mensaje, que ya se encuentra disponible en memoria de entrada.

Los siguientes tres displays HEX muestran la dirección de memoria con el código polar utilizado actualmente. Las direcciones entre 0x000 y 0x07F son dedicadas a frozen bits pre-cargados en el momento de síntesis. Las direcciones entre 0x080 y 0x0FF pueden ser destinadas a códigos generados en el procesador durante el tiempo de ejecución. La memoria, de un tamaño de 4096 bytes, permite pre-cargar hasta 128 códigos, para $N = 128$, y tener almacenados en totalidad 256 conjuntos de frozen bits. El display dedicado al nibble más significativo de la dirección se reserva para eventuales incrementos en el tamaño de esta memoria. Por lo tanto, su valor se fija en 0x0. El quinto display HEX muestra la orden actual recibida por la FSM del decodificador. Los valores encontrados son:

1. **0** $\Longrightarrow$ **Reset**: orden de pasar al estado de reset.

2. **d** $\Longrightarrow$ **Decode**: orden de decodificar el mensaje, que ya se encuentra disponible en la memoria de entrada.

El último display muestra el estado con el que la FSM del decodificador responde al HPS y de acuerdo al estado actual:

1. **0** $\Longrightarrow$ **Reset**: la FSM se encuentra en el estado de espera inicial.

2. **d** $\Longrightarrow$ **Decoding**: la FSM está aguardando la decodificación de un mensaje.

3. **F** $\Longrightarrow$ **Fin**: se terminó de codificar un mensaje, y el resultado se encuentra disponible en la memoria de salida para ser leído desde el microprocesador.

4. **b** $\Longrightarrow$ **Bad command**: se recibió del HPS un comando no reconocido.

Por otro lado, se utilizaron los diez LEDs de la placa para notificar al usuario el estado actual de las FSM. Cinco de estos diodos se dedicaron a cada una de las máquinas de estado con un esquema de notificación semejante, expresado en formato binario para el codificador, con 1≡*encendido* y 0≡*apagado*:

1. **10000** $\Longrightarrow$ **Reset**: el estado actual es el de reset y espera de órdenes.

2. **01000** $\Longrightarrow$ **Procesando mensaje**: se está procesando un mensaje.

3. **00100** $\Longrightarrow$ **Fin del procesamiento**: se terminó de procesar un mensaje.

4. **00011** $\Longrightarrow$ **Error**: se encontró un error de interpretación del comando.

Para notificar los estados del decodificador se utilizaron los mismos códigos binarios, pero en disposición espejada.

Estos dispositivos visuales trabajan con una frecuencia de refresco igual a la frecuencia de reloj de las FSM (50 MHz), por lo tanto, el ojo humano no puede seguir en detalle las distintas notificaciones. Sin embargo, sí puede apreciar la intensidad luminosa de los distintos dispositivos, que es directamente proporcional al tiempo promedio que las FSM permanecen en cada estado. Así puede determinar, sin acceder a la computadora, y de forma aproximada, el funcionamiento correcto del sistema. En caso de un error crítico, que congelara las máquinas de estado o el programa de pruebas, los displays ayudan en la eventual depuración de las causas de error. Finalmente, en operación normal el usuario puede determinar visualmente el estado general de espera en reset observando en los displays HEX la aparición de todos ceros, y en los LEDs la aparición del código binario 1000000001. Por otro lado, el usuario puede generalmente estimar la dirección de memoria de frozen bits actualmente utilizada y determinar el estado del canal de comunicaciones siendo actualmente simulado.

El usuario puede también interactuar con el sistema de forma directa, utilizando la placa de desarrollo. Un botón permite hacer un reset general a la parte del sistema implementada en

FPGA y a las interfaces. Otro permite reiniciar el HPS de forma independiente a la FPGA. Un tercer botón permite cortar la alimentación de toda la placa de desarrollo.

Por último, se dispone de ocho switches que el usuario debe configurar para establecer la cuenta máxima del contador en el decodificador, que determina el fin del procesamiento. Ésto permite al usuario determinar experimentalmente el retardo de propagación en el sistema y deducir a partir de éste la frecuencia máxima de operación.

Este procedimiento, aplicado al sistema decodificador, implica iniciar una serie de pruebas con algunos centenares de miles de mensajes, con un canal y un estado que garanticen probabilidades de error de mensaje extremadamente bajas (por ejemplo AWGN con $SNR = 8dB$). Estos switches no pueden ser manipulados durante la ejecución de un ciclo de pruebas, ya que no cuentan con un sistema anti-rebotes y pueden inducir errores; el estado de los switches debe modificarse antes del inicio o después del fin de un ciclo de pruebas. Comenzando con la cuenta máxima disponible, el usuario puede ir progresivamente reduciendo el valor de la cuenta, hasta observar en pantalla la existencia de errores de decodificación. Con los ocho switches el contador puede contar entre 0 y 255 ciclos de 50 MHz. La cuenta en cero equivale a dedicar un solo ciclo de reloj a la espera. Por lo tanto, el sistema soporta decodificadores cuyos retardos de propagación se encuentren entre los 40 nanosegundos y los 5,12 microsegundos, equivalentes a frecuencias de operación entre los 195,3 KHz y los 25 MHz.

Los switches se operan con la lógica: $sw\uparrow\equiv 1$; $sw\downarrow\equiv 0$. La configuración de su operación es fácilmente modificable en el código fuente, y pueden también destinarse switches para limitar la velocidad de operación del codificador.

4.2. Resultados de síntesis

Para el desarrollo de sistemas en FPGA o ASIC es importante determinar el consumo de recursos de hardware disponibles. Estos recursos son limitados y deben a veces ser compartidos con otros sistemas, cuando el desarrollo se ubica dentro de cierta jerarquía.

El sistema completo se sintetizó para distintos valores de N con Quartus Prime Lite 17.1, utilizando la opción de compilación *Performance (High effort)* para maximizar la velocidad. Esta elección se hizo en concordancia con el criterio de maximización de velocidad de operación, utilizado durante el diseño de todo el sistema. Los consumos de recursos de hardware necesarios para la implementación en la FPGA Intel Cyclone V 5CSXFC6D6F31C6N se exponen en la Tabla 4.1. Los resultados se extraen de los reportes del fitter de Quartus, generado automáticamente tras las compilaciones completas del sistema.

Longitud de mensaje	ALMs	LUTs	Registros	Pines I/O	Bloques M10K	f_{MAX} (MHz)
8	2.800	4.191	4.067	177	72	82,29
16	3.081	5.289	4.125	177	72	97,06
32	3.751	6.414	4.251	177	72	126,81
64	5.266	8.901	4.469	177	72	96,51
128	8.793	14.691	4.936	177	72	104,31
Máximo disponible	41.910	110.000	83.820	499	553	–

Cuadro 4.1: Resultados de síntesis del sistema para distintas longitudes de mensaje

En la Tabla 4.1 se aprecia que los ALMs, LUTs y registros tienen un requerimiento de recursos inicial cercano a los expresados para $N = 8$. Este requerimiento depende en gran medida de las FSM e interfaces de comunicación HPS–FPGA, implementadas con la lógica de la FPGA. El crecimiento en el requerimiento de recursos no es lineal respecto a la longitud de mensaje para ALMs y LUTs. Para los registros, su requerimiento está predominantemente determinado por las interfaces de comunicación con el HPS, por lo que éste no escala linealmente con el incremento de N. Las cantidades de pines y bloques de memoria M10K son independientes de la longitud de mensaje utilizada.

En la tabla anterior las frecuencias son las máximas estimadas por Quartus, para la señal

de reloj de base en la FPGA, con el modelo lento, una tensión de 1100 mV y una temperatura de 85 °C. Todas las estimaciones se encuentran dentro de un mismo orden y concuerdan con la utilización de la frecuencia fijada en 50 MHz para las implementaciones.

En este trabajo no pudo lograrse la compilación y síntesis del sistema para $N = 256$. Para éste se elaboró una implementación duplicando el número de módulos de memoria, ofreciendo al sistema de corrección de errores un acceso a las memorias rápido y totalmente paralelo, con puertos de 256 bits de ancho, y evitando modificar las máquinas de estado ya programadas. Los reportes del compilador indicaron que la FPGA poseía los recursos necesarios para sintetizar la totalidad de la lógica, pero el Fitter no podía rutear el diseño para el dispositivo seleccionado. El diseño sea probablemente sintetizable en modelos de FPGA que dispongan de mayores recursos.

Los valores de las máximas frecuencias en la Tabla 4.1 son estimaciones, hechas por el entorno de desarrollo, para la señal de reloj maestra que maneja las FSM y registros. Esta frecuencia de reloj no es la frecuencia real a la que operan el codificador y el decodificador.

En el laboratorio se establecieron las frecuencias reales utilizando el procedimiento presentado ya en la sub-sección "Interfaces de usuario". Es interesante contrastar los valores encontrados con los estimados en la Tabla 3.6. A tal efecto se presenta la Tabla 4.2.

	f_{MAX} est. (MHz)	TP est. (Mbps)	f_{MAX} real (MHz)	TP real (Mbps)
Codificador	316,36	44.330	50	6.400
Decodificador	3,7	473,6	7,14	913,9

Cuadro 4.2: Comparación de frecuencias y velocidades estimadas y reales para $N = 128$

Se observa en la Tabla 4.2 que para el codificador hay una gran diferencia entre las frecuencias y velocidades estimadas y reales. Para entender ésto, debe considerarse que en la implementación del sistema final, el codificador debió operar a una frecuencia baja, impuesta por el sistema de control de los registros de almacenamiento. Esta limitación no es importante para el funcionamiento del sistema de pruebas. El retardo de 20 nanosegundos, debido a la espera de la codificación, es despreciable frente a los aproximadamente 300 microsegundos que lleva en promedio cada ciclo de prueba por mensaje.

En contrapartida, para el decodificador se encontró una velocidad de operación real prácticamente el doble de alta; la estimación del IDE Quartus es en definitiva muy conservadora. Es destacable, en conclusión, la importancia de verificar en el laboratorio las frecuencias de operación, para aprovechar así al máximo las capacidades del diseño sintetizado, y también evitar posibles errores.

Como conclusión de esta sección, se presenta una comparación de la implementación final en FPGA del sistema completo de pruebas con la implementación del decodificador de la referencia para $N = 128$ [28].

Se destaca en la Tabla 4.3 que la implementación de este proyecto alcanza velocidades reales mayores que las de la referencia, con un consumo de recursos similar.

Debe tenerse en cuenta que las LUTs requeridas por el sistema de este proyecto soportan no solamente al decodificador, sino también al codificador, las dos máquinas de estado y a la lógica correspondiente a las interfaces con el HPS. Del reporte sobre consumo de recursos por entidad, se obtuvo que al decodificador se destinan 9.731 LUTs, que representan un 67 % de lo requerido por la referencia. En ésta no se aclara qué elementos agrega en su implementación en torno al decodificador.

Por otro lado, en dicho reporte se encontró que la lógica dedicada a comunicar el HPS con la FPGA implica el uso de 4.642 LUTs, que representa el 31,6 % del total requerido por el sistema.

Este trabajo		Referencia	
LUTs	Throughput (Mbps)	LUTs	Throughput (Mbps)
14.691	913,9	14.517	820

Cuadro 4.3: Resultados de implementación del sistema completo y de la referencia para $N = 128$

4.3. Consideraciones finales de la implementación

El código fuente de la implementación de todo el sistema fue desarrollado pensando en la posibilidad de futuras modificaciones por parte de un usuario experto. El código se escribió buscando hacerlo mantenible y reutilizable, utilizando comentarios y modularidad cuando fuera posible.

Para desarrollar las entidades en VHDL se siguió el estilo de programación estructural, dividiendo entidades importantes en varias entidades más pequeñas, siguiendo un orden jerárquico que facilitara la modularidad, la adaptabilidad y la accesibilidad a las simulaciones funcionales. El código es fácilmente adaptable, permitiendo reutilizar módulos en otras implementaciones o intercambiarlos por otros. El código fuente del codificador y del decodificador se logró totalmente parametrizable para $8 \leq N$, K, y Q. El código del sistema que integra los módulos de corrección de errores, la lógica de control y de las interfaces HPS–FPGA, es parametrizable para $8 \leq N \leq 128$ y K, con la limitación de variaciones de Q requiriendo leves modificaciones de la integración del sistema y de algunas entidades VHDL. El sistema completo requirió la programación específica de 28 entidades en VHDL. El código del programa en C se logró también parametrizable para $8 \leq N \leq 128$ y K. El código es también altamente modularizado y comentado. El sistema cuenta con varias interfaces de usuario que facilitan la utilización del sistema y la depuración de eventuales errores.

Las características anteriores abren la posibilidad de utilizar el sistema de FPGA y HPS construido alrededor del sistema corrector para facilitar el desarrollo de otras aplicaciones. Con leves modificaciones en las interfaces y en el programa en C debería ser posible probar el funcionamiento de módulos implementados en FPGA para ejecutar tareas fuera del ámbito de los códigos correctores de errores. Un hipotético diseñador de dichas aplicaciones debería agregar al código en C sus propias funciones de prueba, o adaptar las ya existentes. Este diseñador evitaría entonces dedicar demasiado tiempo al desarrollo de las interfaces, los sistemas de control y de depuración de su sistema.

Capítulo 5

Resultados experimentales

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. Con 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 diversos objetivos: determinar la mejor forma de cuantificar los LLR, verificar funcionalidades del sistema y efectuar comparaciones con resultados de referencia. En las pruebas la FPGA se encargó de codificar y decodificar los mensajes de prueba. El programa corriendo en el HPS se ocupó de simular e implementar los demás elementos del sistema de comunicaciones.

En todas las pruebas se calcularon los frozen bits específicamente para cada tipo y estado de canal utilizado. El cálculo de los mismos se hizo con el algoritmo recursivo basado en los límites de Bhattacharyya [2]. Todos los códigos utilizados se pueden encontrar en el anexo de este informe.

Los mensajes fueron modulados en software usando la señalización binaria por desplazamiento de fase (BPSK) con las correspondencias de bits a niveles: $0 \to 1$; $1 \to -1$.

Para obtener los LLRs de los canales AWGN, con relaciones de señal a ruido SNR en unidades de dB, para cada bit recibido y_i se utilizaron las expresiones:

$$L_i(y_i) = \frac{2 \times y_i}{\sigma^2} \tag{5.1}$$

$$\sigma = \frac{1}{\sqrt{2 \times Rc \times {}^{E_b}/_{N_0}}} \tag{5.2}$$

$$\frac{E_b}{N_0} = 10^{\frac{SNR}{10}} \tag{5.3}$$

Los LLR se representaron saturando sus magnitudes con los máximos valores representables con Q bits de cuantificación signo–magnitud, como propone Leroux [5].

Para determinar la performance correctora de la implementación se calcularon las probabilidades o tasas binaria de error (BER) y las tasas de error por bloque (FER). Las estimaciones de las tasas se obtuvieron para mensajes generados aleatoriamente. Estas relaciones dependen en parte del tipo y del estado del canal de comunicaciones empleado, y son interpretadas con mayor facilidad con la ayuda de gráficos.

5.1. Cuantificación

Para encontrar la mejor forma de representar los LLR con $Q = 5$ bits, se ejecutaron pruebas con distintos arreglos de bits para la parte entera y para la parte fraccionaria. Un LLR representado en formato de signo y magnitud destina siempre un bit al signo, y los restantes a la magnitud. Las representaciones de magnitud se expresan como $Q_{i,f}$, con i bits para la parte

entera y f bits para la parte fraccionaria.

Se compararon los resultados con los obtenidos en un decodificador implementado en software en lenguaje C++, facilitado por el Laboratorio de Comunicaciones. A dicho programa se lo modificó para operar con las limitaciones impuestas por la cuantificación en punto fijo SM y obtener de este modo una comparación más precisa.

Para los parámetros $N = 128$, $K = 64$ se obtuvieron gráficos para el canal AWGN. En la Fig. 5.1 se presentan las curvas con las tasas de error por bloque. En la Fig. 5.2 se encuentran las correspondientes a las tasas de error por bit.

Se aprecia de los dos conjuntos de curvas, en primer lugar, que la cuantificación $Q_{3,1}$ es la que ofrece la mejor performance para BER y FER. Existe un orden de magnitud de diferencia entre las probabilidades de dicha representación y la correspondiente a $Q_{1,3}$. Por lo tanto, se utilizará de modo regular la cuantificación $Q_{3,1}$ para obtener los resultados de las pruebas restantes. En segundo lugar, para las mismas disposiciones de bits de cuantificación, las curvas correspondientes al sistema en FPGA y al sistema en software son casi idénticas para todo el rango de prueba. Por lo tanto, el decodificador en hardware opera como lo esperado. Por último, en ambos gráficos se cumple la mejora en la performance a medida que la SNR aumenta.

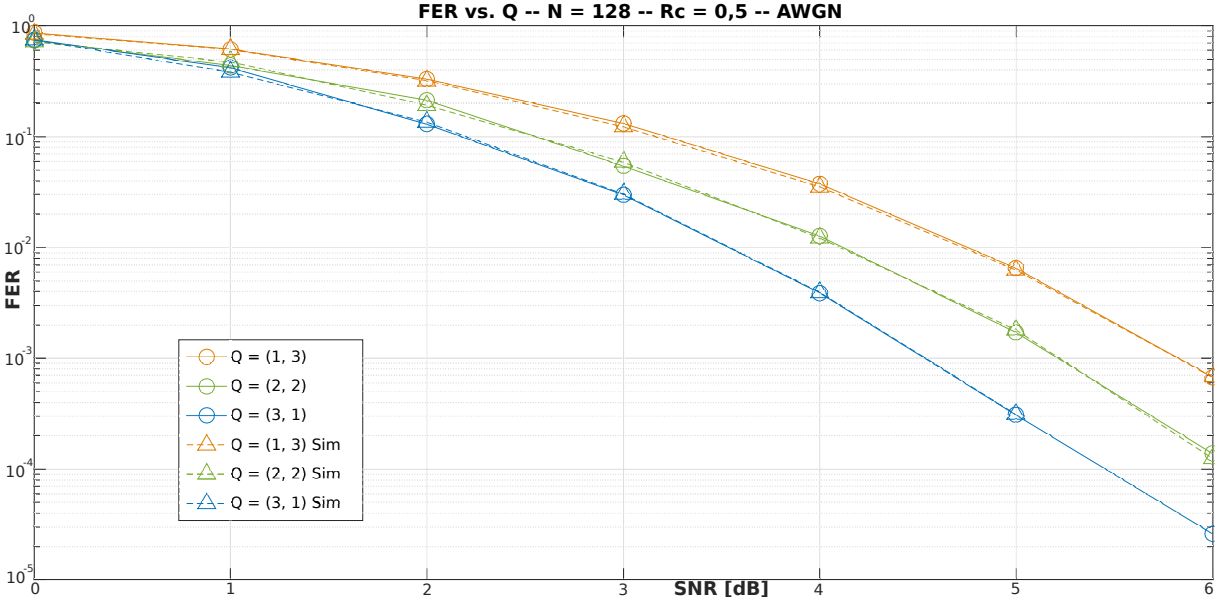


Figura 5.1: FER – $N = 128$, $Rc = 0,5$ – $Q_{1,3}$ vs. $Q_{2,2}$ vs. $Q_{3,1}$ – AWGN
FPGA vs. Simulación SM

5.2. Cambio de tasa de código

Para verificar la capacidad del sistema de adaptarse a variaciones de la tasa de código, cambiando el conjunto de frozen bits asociados a la misma, se realizó una prueba con $Rc = 0,75$. Como referencia se tomó un paquete de MATLAB para simulación de códigos polares [37].

Para $N = 128$, $K = 96$ y $Q_{3,1}$ se obtuvieron las curvas de la Fig. 5.3 para el canal AWGN:

Observando la Fig. 5.3 se puede comprobar que la performance de la FPGA es en todos los casos cercana a la de la referencia. La degradación de performance por utilizar la representación de punto fijo es mínima. No sería entonces justificable incrementar la complejidad del hardware al utilizar un número mayor de bits de cuantificación. Esto va en concordancia con lo hallado en la bibliografía [5] [28].

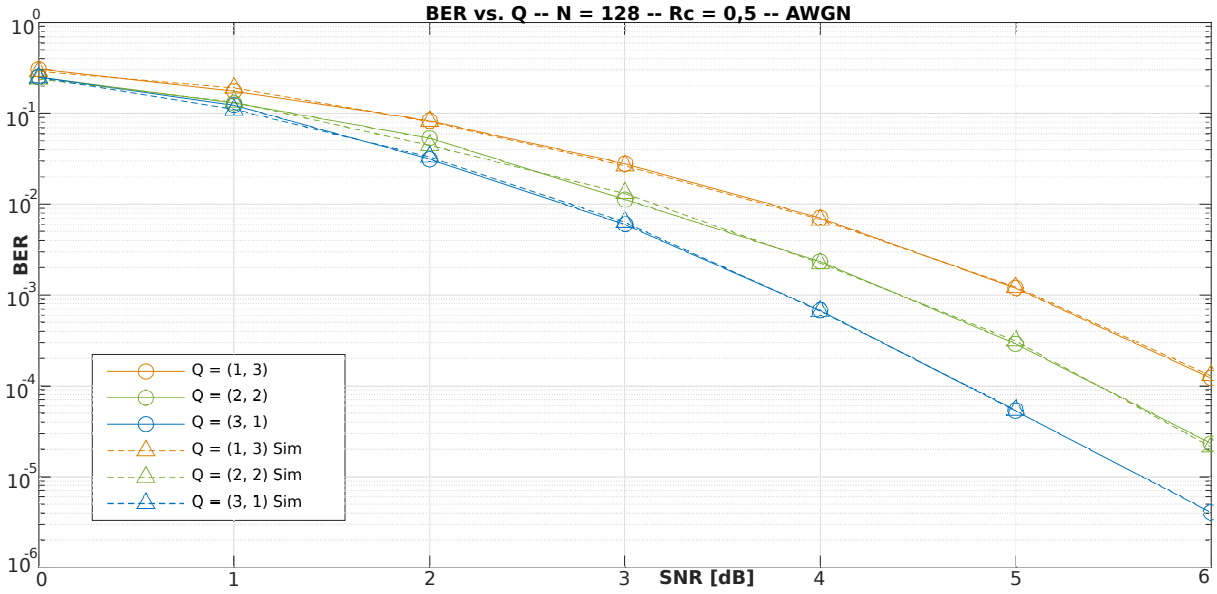


Figura 5.2: BER – $N = 128$, $Rc = 0,5$ – $Q_{1,3}$ vs. $Q_{2,2}$ vs. $Q_{3,1}$ – AWGN
FPGA vs. Simulación SM

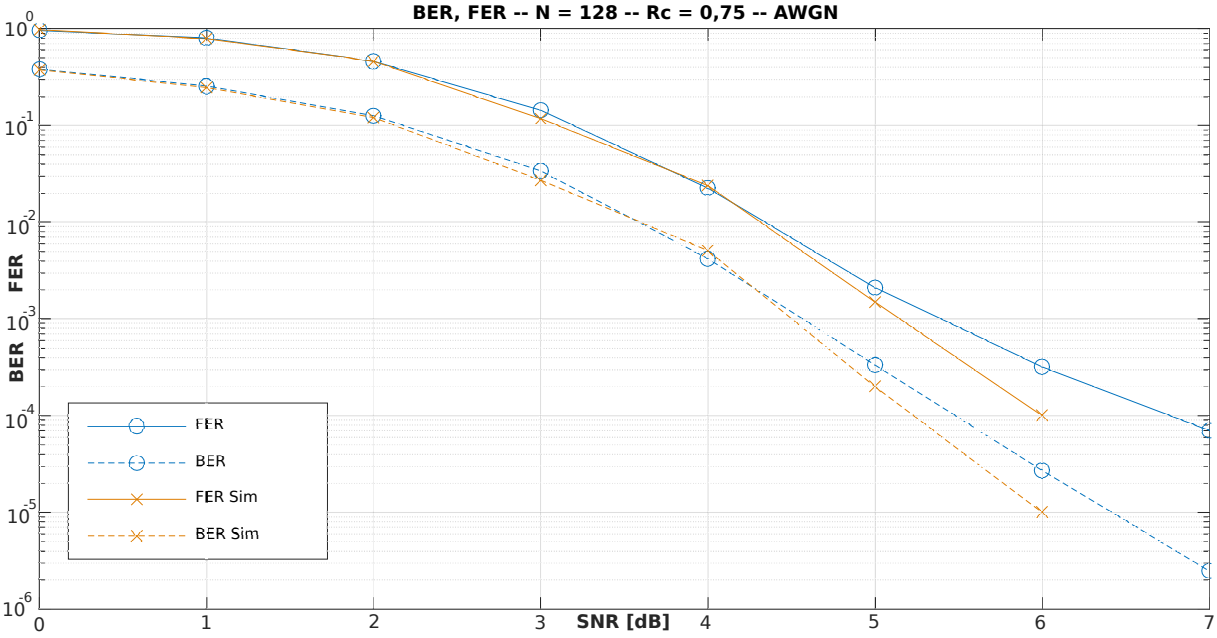


Figura 5.3: BER, FER – $N = 128$ – $Rc = 0,75$ – $Q_{3,1}$ – AWGN
FPGA vs. Simulación en punto flotante

Si se comparan estos resultados con los valores de FER y BER para $Q_{3,1}$, en la Fig. 5.1 y la Fig. 5.2 respectivamente, se encuentra que el incremento en la tasa de código degrada la performance.

Por último, en todas las curvas se cumple que en igualdad de condiciones la tasa de error binaria es inferior a la tasa de error de bloque. Estas conclusiones concuerdan con lo expresado en la teoría de codificación para control de errores.

5.3. Escalabilidad

El código fuente desarrollado en este proyecto permite implementar sistemas escalables para longitudes de mensaje $8 \leq N \leq 128$. Para verificar esta característica del código, se sintetizaron sistemas con los valores $N = 64$ y $N = 128$, manteniendo en ambos casos una tasa de código

$Rc = 0,5$.

En la Fig. 5.4 se presentan los resultados de la prueba de escalabilidad, utilizando el canal de ruido gaussiano, para la FPGA y la referencia en punto flotante.

Se observa, inicialmente, que la performance del sistema es en todos los casos próxima a la de la referencia. En segundo lugar, tanto FER y BER muestran mejores valores para las longitudes de código mayores. Finalmente, se mantiene en todos los casos que FER > BER.

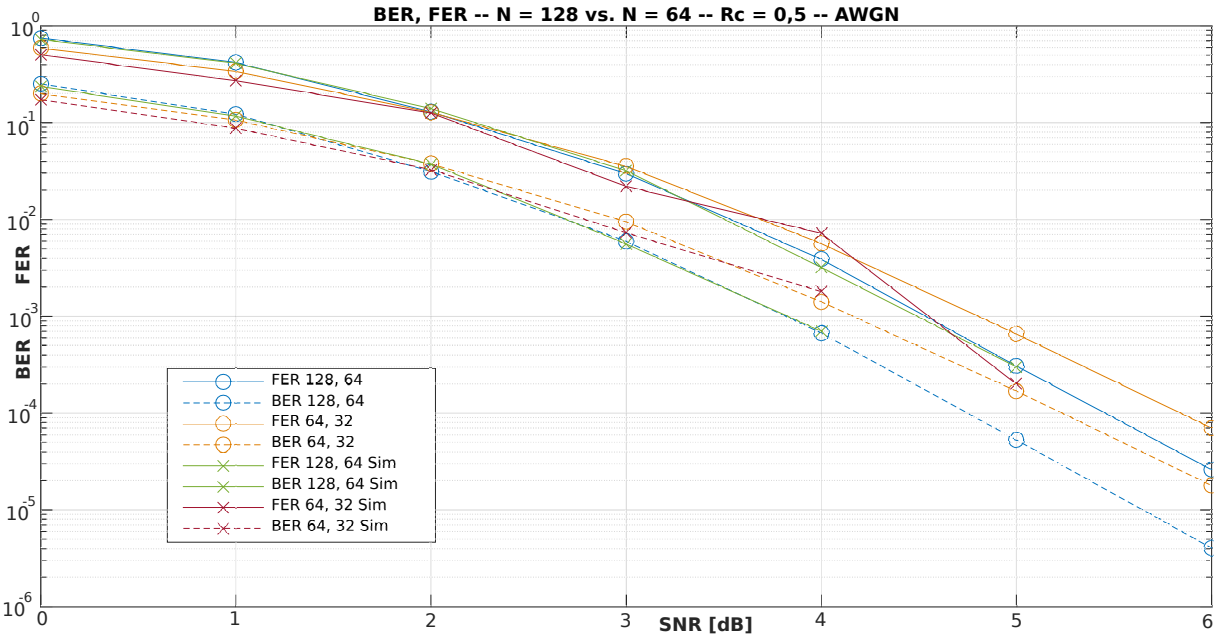


Figura 5.4: BER, FER – $N = 128$ vs. $N = 64$ – $Rc = 0,5$ – $Q_{3,1}$ – AWGN
FPGA vs. Simulación en punto flotante

5.4. Adaptabilidad al canal

El sistema de codificación y decodificación con códigos polares puede adaptarse a distintos tipos de canales. Para verificar esta capacidad, se realizó una prueba seleccionando el canal BSC como uno de los parámetros.

En la Fig. 5.5 se encuentran los resultados de la prueba. En el canal BSC, probabilidades de transición de símbolo más altas representan peores estados. Por ello, en contrapartida con el canal gaussiano, las curvas son monótonamente crecientes. Las curvas son muy cercanas a las determinadas con la referencia en punto flotante. Se comprueba entonces la adaptabilidad del sistema a los tipos y estados de canales de comunicaciones.

5.5. Corrección de ceros

En el capítulo 3 se diseñó una variante del elemento de procesamiento que corregía los ceros negativos en la salida de la función g. Para determinar el efecto de esta corrección sobre la performance, se hizo una comparación entre los sistemas empleando el MPE original y el modificado.

En la Fig. 5.6 se encuentran los resultados de las pruebas. Es evidente a partir de éstos que los diferentes MPE no implican una diferencia apreciable en la performance de decodificación.

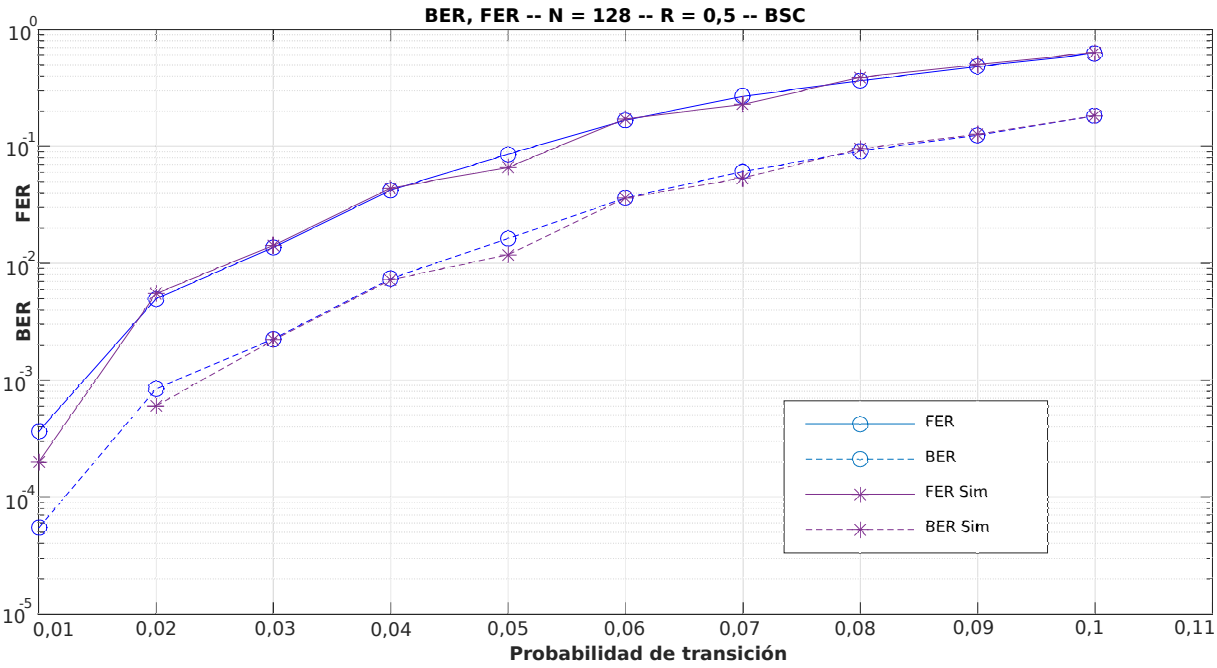


Figura 5.5: BER, FER – $N = 128$ – $Rc = 0,5$ – $Q_{3,1}$ – BSC
FPGA vs. Simulación en punto flotante

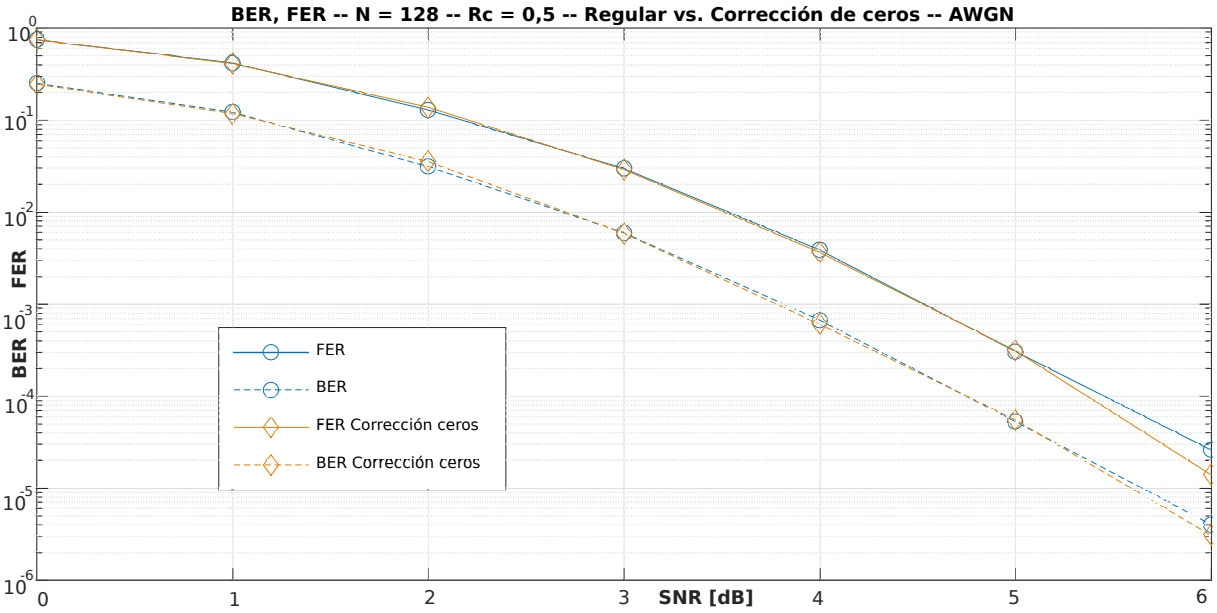


Figura 5.6: BER, FER – $N = 128$ – $Rc = 0,5$ – $Q_{3,1}$ – AWGN
FPGA Regular vs. FPGA Con corrección de ceros negativos

5.6. Simplificación de los LLR

Hasta aquí se hicieron pruebas calculando las relaciones de similitud en función de las muestras del canal y de las potencias de ruido. Existe una forma simplificada de calcular los LLR para el canal AWGN, que es independiente del estado del mismo [38]. Con ésta, los LLR se calculan como:

$$L_i(y_i) = 4 \times y_i \tag{5.4}$$

En la Fig. 5.7 se exponen los resultados en función de las dos formas de calcular los LLRs. Se observa, en primer lugar, que el cálculo tradicional de los LLR con $Q_{1,3}$ presenta una performance marcadamente inferior a las demás. Cuando se cuantifica con $Q_{3,1}$ la performance se aproxima

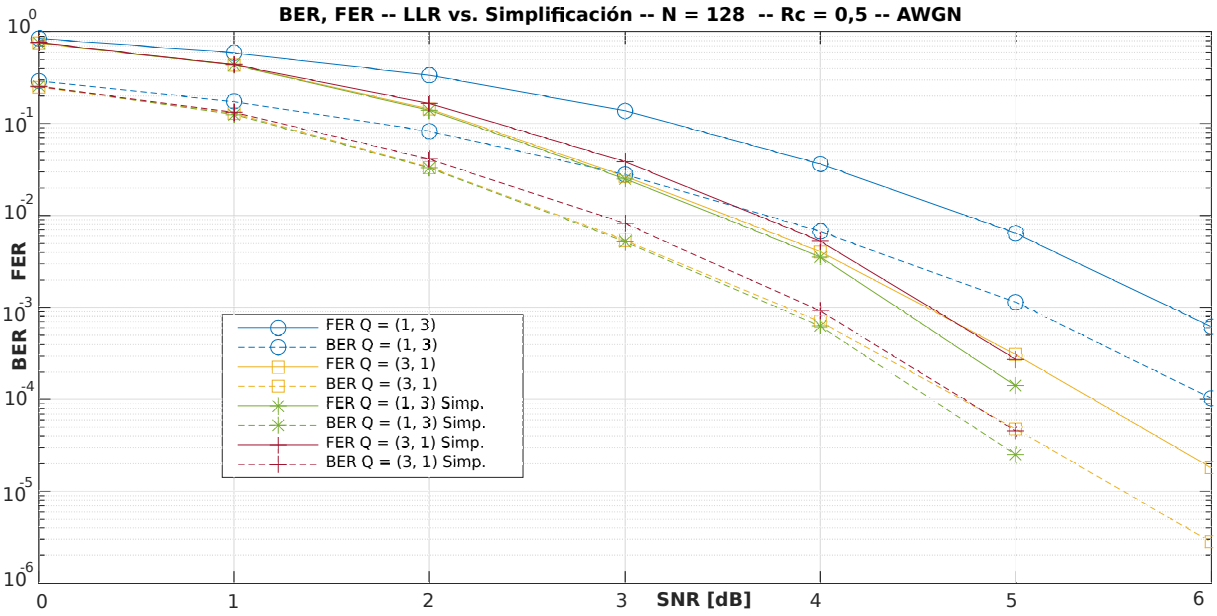


Figura 5.7: BER, FER – $N = 128$ – $Rc = 0,5$ – $Q_{3,1}$ – AWGN
FPGA Regular vs. FPGA Con simplificación de LLRs

a la de la simplificación.

La combinación de simplificación de LLRs y representación $Q_{1,3}$ exhibe en todos los casos la mejor performance. Esta mejora parece acentuarse a mayores relaciones de señal a ruido. Esto se debe a que la forma regular de calcular los LLRs con formato signo–magnitud satura las máximas magnitudes. Se produce entonces un recorte de las crecientes magnitudes que se encuentran a mayores valores de SNR, y se pierde información para la estimación correcta de los LLRs.

Capítulo 6

Conclusiones y consideraciones a futuro

En este trabajo se diseñó y se implementó en una placa de desarrollo FPGA un sistema rápido de corrección de errores para comunicaciones digitales que utiliza códigos polares como técnica de control. Con el objetivo de facilitar el desarrollo del proyecto, asegurando su portabilidad a otras placas de desarrollo y su facilidad de uso y posibilidad de modificaciones, se siguieron criterios de modularidad, propios del lenguaje VHDL. Teniendo en consideración el propósito de su utilización como banco de pruebas, capaz de procesar y analizar el comportamiento de millones de mensajes sobre diferentes canales, se adoptó también el objetivo de lograr un diseño de alta velocidad y performance de error comparable a las implementaciones conocidas sobre este tema.

Esta tarea demandó, en un principio, un exhaustivo estudio de más de treinta publicaciones referidas a la implementación en hardware de códigos polares. Se estudiaron diferentes arquitecturas de alto nivel para la construcción de codificadores y decodificadores.

En cuanto al decodificador, tras una evaluación de las diferentes posibilidades , en general de tipo sincrónico, con control de máquinas de estado y enfocadas a resultados eficientes en cuanto al área ocupada, se seleccionó una novedosa arquitectura, de tipo combinacional, energéticamente eficiente y con velocidad de procesamiento elevada.

Esta arquitectura se caracteriza por implementar de manera recursiva el algoritmo de decodificación de cancelaciones sucesivas, sencillo y originalmente propuesto por Arikan, siendo fundamental para las investigaciones actuales sobre códigos polares. Hoy en día, la mayoría de los estudios sobre estos códigos se mantienen centrados en la implementación de este algoritmo y de sus posibles modificaciones para lograr mejor performance en control de errores, a bajo costo en términos de área y velocidad, resultando en general un delicado equilibrio entre estos aspectos.

Con el propósito de optimizar el consumo de recursos y la velocidad,se trabajó sobre un diseño original de un componente central de la arquitectura de decodificación: el elemento de procesamiento.

Las características propias de las operaciones requeridas por el decodificador, sumado a la cuantificación seleccionada, permitieron obtener un diseño eficiente en cuanto al requerimiento de recursos. Por otra parte, al adoptar el algoritmo de adición y sustracción para representación signo-magnitud, se pudo emplear una técnica de pre-cómputo que ayudó a mitigar problemas de latencia propios del algoritmo de decodificación. También, partiendo del diseño original del elemento de procesamiento, se derivaron dos variantes del mismo. Una de ellas contempló aspectos relacionados con el objetivo de reducir la complejidad del hardware. La otra variante, pretendió estudiar el efecto de los ceros negativos en la representación signo-magnitud y su efecto en la performance de error del decodificador.

Las entidades necesarias para ambos esquemas, codificación y decodificación, se crearon siguiendo el estilo de programación estructural, característico del lenguaje VHDL. Por otra

parte, dada la estructura de alto nivel del decodificador, se empleó una variante de programación recursiva, poco habitual en este lenguaje de programación de hardware, pero elegante y eficaz a la hora de la implementación.

Las entidades de menor jerarquía se desarrollaron con un procedimiento de programación estructural y modular, simulaciones individuales y de integración, depuración y corrección buscando evitar futuros errores en un sistema de por sí complejo.

Siguiendo estas pautas de desarrollo, se escribió un código fuente, completamente genérico y parametrizado, tanto para el codificador como para el decodificador. De este modo, se persigue un doble objetivo: facilidad de la implementación en cuanto a su posible modificación y portabilidad a placas de desarrollo FPGA de otros fabricantes, o incluso a circuitos integrados dedicados como los ASIC.

Con el objetivo de investigar la performance de control de errores del conjunto codificador-canal-decodificador se desarrolló un sistema de pruebas automatizado. En general, estos bancos de prueba se implementan de tal manera que el codificador y el decodificador residen en la placa FPGA y el canal, por su parte, se simula aparte, en una PC, con la ayuda de algún programa, generalmente MATLAB. De este modo, se puede simular la comunicación de mensajes aleatorios mediante algún software apropiado y utilizar la FPGA para acelerar las tareas computacionamete más complejas de codificación y decodificación.

Considerando que la placa de desarrollo disponible integraba un microprocesador en la propia placa, se diseñó un sistema totalmente integrado en la misma. En este caso, es el propio microprocesador el encargado de generar mensajes aleatorios, entregárselos al codificador, recibir del mismo los mensajes codificados, contaminar dichos mensajes con ruido siguiendo las características del canal programado, entregar estos mensajes al decodificador, recibir los mensajes decodificados y comparar con el mensaje original para evaluar la performance de error. La utilización de interfaces internas de alta capacidad permite una rápida comunicación entre el procesador y la lógica sintetizada. Así se logra un banco de pruebas mucho más compacto y eficiente en términos de velocidad, que además no requiere conectar una computadora externa para las tareas de procesamiento. El propio microprocesador, con sus memorias y periféricos, es funcionalmente una computadora en el interior de la FPGA.

Con el agregado de un sistema operativo, y la utilización de los dispositivos disponibles en la placa de desarrollo, se implementaron interfaces con el usuario que facilitarán el empleo del sistema de pruebas. El sistema operativo permite al usuario configurar, con ayuda de una computadora, los parámetros de las pruebas de comunicaciones, ejecutar las mismas a altas velocidades, y almacenar de forma permanente los resultados. Con la configuración ideada, los displays y botones en la placa permiten comprobar y controlar el funcionamiento de los componentes sintetizados en FPGA.

En definitiva, se implementó un sistema embebido de pruebas, que podría ser fácilmente adaptado para la evaluación de otros componentes sintetizados en FPGA. Debido a la modularidad del código del programa, con leves modificaciones, podrían enviarse datos a dichos componentes, analizar sus salidas, y comprobar su funcionamiento. Las interfaces entre el procesador y la FPGA, y la lógica de control ya existentes facilitarían la puesta en marcha de dichas modificaciones.

La integración completa del sistema permitió llevar a cabo un conjunto masivo de pruebas de funcionalidad y performance. Se pudo así comprobar que la implementación es escalable y adaptable a distintos tipos y estados del canal de comunicaciones. La comparación con otros trabajos de referencia confirmó la elección correcta de la cantidad de bits de cuantificación para las entradas del decodificador. Se pudo además optimizar la distribución de dichos bits. Por último, el sistema permitió optimizar las frecuencias de operación del decodificador en FPGA, reduciendo los tiempos de espera de decodificación a la mitad respecto a los estimados por las herramientas de desarrollo.

Los resultados de síntesis permitieron obtener los requerimientos de hardware reales del

sistema para el modelo de FPGA empleado. La comparación de éstos con la referencia de arquitectura combinacional mostró que, con un consumo casi idéntico de recursos de hardware, el diseño en este proyecto es un 11 % más rápido, para el decodificador de 128 bits [28]. Estos resultados en término de área se obtuvieron sintetizando para este trabajo sintetizando para este trabajo no sólo el decodificador, sino además el codificador, lógica de control e interfaces con un microprocesador. Si se considera solamente la cantidad de recursos destinados al decodificador, se obtuvo además que esta implementación requiere un 33 % menos de tablas de búsqueda que la implementación de referencia.

En conclusión, se cumplió el objetivo principal del proyecto de implementar un sistema de comunicaciones con códigos polares. Efectuando pruebas de decodificación y comparando con simulaciones de referencia, se validó la capacidad de corrección de errores de esta implementación. Por último, los resultados de la implementación superaron a la referencia tanto en requerimiento de recursos como en velocidad, demostrando la utilidad de los criterios de diseño mantenidos durante todo el proceso de desarrollo.

Teniendo en cuenta lo experimentado a lo largo de este proyecto, se pueden hacer algunas consideraciones a tener en cuenta en futuras ampliaciones de este proyecto, o en proyectos similares. Como una primera consideración, se podría estudiar alguna forma de llevar a cabo esta implementación para longitudes de mensajes superiores a 128 bits. Dado que el código VHDL de esta implementación es completamente genérico y parametrizable, podría probarse en placas más potentes y aún de otros fabricantes. Si se decidiera continuar con la FPGA utilizada en este proyecto, podría buscarse inicialmente reducir el número de bloques de memoria cuyos puertos de datos se disponen en paralelo. Ésto podría reducir el requerimiento de ruteo de señales entre la FPGA y el HPS, aunque probablemente reduzca las velocidades de comunicación entre éstos. Una segunda opción sería usar codificación por software, liberando los recursos correspondientes y, en contrapartida, ralentizando los ciclos de prueba de mensajes y simulaciones.

En referencia a los LLRs, se podría ahondar en el estudio de su cálculo, y buscar las formas óptimas de representarlos en función de distintos estados de canales de comunicaciones.

Si se emplearan otros entornos de desarrollo para FPGA, o se dispusiera de dispositivos de otros fabricantes, sería interesante obtener resultados de síntesis para los distintos elementos de procesamiento, o del sistema en su totalidad. Otros fabricantes utilizan otras disposiciones internas en las celdas lógicas configurables de sus FPGA, por lo tanto, los resultados de síntesis podrían diferir respecto a los encontrados en este trabajo. De este modo, se podría comprobar si se mantiene el menor requisito de LUTs del elemento de procesamiento con control de ceros negativos.

En relación al último punto, también se podrían hacer más pruebas para determinar el efecto de la corrección de ceros negativos en la performance de decodificación para mayores valores de N. En estos casos, la aparición de ceros negativos en el interior del decodificador sería a priori más probable, ya que las funciones f siempre propagan ceros y existirían además más posibilidades de que éstos se generen con las funciones g.

Por último, se podría aprovechar la modularidad del código y reutilizar las entidades sintetizadas en este proyecto en otras arquitecturas. Se podrían ensayar con éstas diseños sincrónicos que las activaran repetidamente durante el procesamiento de cada mensaje, requiriendo en consecuencia menos recursos. En sentido opuesto, se podrían buscar mayores valores de throughput implementando pipelines. Aprovechando las características del código fuente, se podrían agregar registros en varios niveles dentro del decodificador para multiplicar la cantidad de mensajes que pueden ser procesados por cada ciclo de reloj, en contrapartida se incrementaría el requerimiento de hardware. Otra posibilidad podría ser aprovechar algunos módulos programados para experimentar con alguna implementación híbrida.

Bibliografía

[1] C. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, nro. 5, pp. 379–423, 623–656, Oct. 1948.

[2] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Transactions on Information Theory*, vol. 55, nro. 7, pp. 3051–3073, Jul. 2009.

[3] ——, "A Performance Comparison of Polar Codes and Reed-Muller Codes," *IEEE Communications Letters*, vol. 12, nro. 6, pp. 447–449, Jun. 2008.

[4] H. Vangala, E. Viterbo, y Y. Hong, "A Comparative Study of Polar Code Constructions for the AWGN Channel," *CoRR*, vol. abs/1501.02473, nro. 6, Jun. 2015. [Online]. Disponible: http://arxiv.org/abs/1501.02473

[5] C. Leroux, A. Raymond, G. Sarkis, y W. Gross, "A Performance Comparison of Polar Codes and Reed-Muller Codes," *IEEE Transactions on Signal Processing*, vol. 61, nro. 2, pp. 289–299, Ene. 2013.

[6] E. Arikan, "Polar codes: A pipelined implementation," *ISBC 2010*, Jul. 2010.

[7] C. Leroux, A. Raymond, G. Sarkis, I. Tal, A. Vardy, y W. Gross, "Hardware Implementation of Successive Cancellation Decoders for Polar Codes," *CoRR*, vol. abs/1111.4362, pp. 289–299, Nov. 2011. [Online]. Disponible: http://arxiv.org/abs/1111.4362

[8] A. Mishra, A. Raymond, L. Amaru, G. Sarkis, C. Leroux, P. Meinerzhagen, A. Burg, y W. Gross, "A Successive Cancellation Decoder ASIC for a 1024-bit Polar Code in 180nm CMOS," *IEEE Asian Solid State Circuits Conference (A-SSCC)*, pp. 205–208, Nov. 2012.

[9] A.Raymond y W. Gross, "Scalable Successive-Cancellation Hardware Decoder for Polar Codes," *IEEE Transactions on Signal Processing*, vol. 62, nro. 20, pp. 5339–5347, Ago. 2014.

[10] A. Pamuk, "An FPGA implementation architecture for decoding of polar codes," *ISWCS 2011*, Nov. 2011.

[11] A. Pamuk y E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE International Symposium on Information Theory*, pp. 957–961, Jul. 2013.

[12] Y. Park, Y. Tao, S. Sun, y Z. Zhang, "A 4.68Gb/s Belief Propagation Polar Decoder with Bit-Splitting Register File," *Symposium on VLSI Circuits Digest of Technical Papers*, pp. 1–2, Jun. 2014.

[13] G. Berhault, C. Leroux, C. Jego, y D. Dallet, "Hardware Implementation of a Soft Cancellation Decoder for Polar Codes," *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Sep. 2015.

[14] A. Pamuk y E. Arikan, "Partial sums generation architecture for successive cancellation decoding of polar codes," *SiPS 2013 Proceedings*, pp. 407–412, Oct. 2013.

[15] G. Berhault, C. Leroux, C. Jego, y D. Dallet, "Partial Sums Computation In Polar Codes Decoding," *IEEE International Symposium on Circuits and Systems (ISCAS)*, Mayo 2015.

[16] Y. Fan y C. Tsui, "An Efficient Partial-Sum Network Architecture for Semi-Parallel Polar Codes Decoder Implementation," *IEEE Transactions on Signal Processing*, vol. 62, nro. 12, pp. 3165–3179, Jun. 2014.

[17] C. Zhang, B. Yuan, y K. Parhi, "Reduced-latency SC polar decoder architectures," *IEEE International Conference on Communications*, pp. 3471–3475, Dic. 2012.

[18] I. Tal y A. Vardy, "List Decoding of Polar Codes," *CoRR*, vol. abs/1206.0050, Feb. 2014. [Online]. Disponible: http://arxiv.org/abs/1206.0050

[19] A. Balatsoukas-Stimming, A. Raymond, W. Gross, y A. Burg, "Hardware Architecture for List SC Decoding of Polar Codes," *ArXiv e-prints*, vol. abs/1303.7127, Feb. 2014. [Online]. Disponible: http://arxiv.org/abs/1303.7127

[20] J. Lin, C. Xiong, y Z. Yan, "A Reduced Latency List Decoding Algorithm for Polar Codes," *IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 56–61, Oct. 2014.

[21] C. Xiong, J. Lin, y Z. Yan, "Symbol-Decision Successive Cancellation List Decoder for Polar Codes," *IEEE Transactions on Signal Processing*, vol. 64, nro. 3, pp. 675–687, Oct. 2015.

[22] C. Xiong, J. Lin, y Z. Yan., "A Low-Latency List Successive-Cancellation Decoding Implementation for Polar Codes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, nro. 12, pp. 3499–3512, Dic. 2016.

[23] Y. Fan, C. Xia, J. Chen, C. Tsui, J. Jin, H. Shen, y B. Li, "A Low-Latency List Successive-Cancellation Decoding Implementation for Polar Codes," *IEEE Journal on Selected Areas in Communications*, vol. 34, nro. 2, pp. 303–317, Feb. 2016.

[24] Z. Piao, C. Kim, y J. Chung, "An Efficient List Successive Cancellation Decoder for Polar Codes," *Journal of Semiconductor Technology and Science*, vol. 16, nro. 5, pp. 550–556, Oct. 2016.

[25] B. Yuan y K. Parhi, "LLR-Based Successive-Cancellation List Decoder for Polar Codes With Multibit Decision," *IEEE Transactions on Circuits and Systems*, vol. 64, nro. 1, pp. 21–25, Ene. 2017.

[26] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, y W. Gross, "Fast Polar Decoders: Algorithm and Implementation," *IEEE Journal on Selected Areas in Communications*, vol. 32, nro. 5, pp. 946–957, Jul. 2013.

[27] P. Giard, G. Sarkis, C. Thibeault, y W. Gross, "A 237 Gbps Unrolled Hardware Polar Decoder," *Electronics Letters*, vol. 51, nro. 10, pp. 762–763, 2014.

[28] O. Dizdar y E. Arikan, "A High-Throughput Energy-Efficient Implementation of Successive-Cancellation Decoder for Polar Codes Using Combinational Logic," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, nro. 3, pp. 436—-447, Dic. 2016.

[29] P. J. Ashenden, "Recursive and Repetitive Hardware Models in VHDL," Departamento de Ingeniería Eléctrica y Computación, Universidad de Cincinnati, Reporte técnico TR 160/12/93/ECE, Dic. 1993.

[30] J. Kaps, *Sign Magnitude Addition - Subtraction Algorithm*, ECE 332 Digital Electronics and Logic Design Lab, George Mason University, consultada en julio de 2018. [Online]. Disponible: ece.gmu.edu/~jkaps/courses/ece331-s07/resources/signedinteger.pdf

[31] A. Raymond, "Design and Hardware Implementation of Decoder Architectures for Polar Codes," Tesis de Maestría, McGill University, Montréal, Canada, Ago. 2013.

[32] Future Technology Devices International, *FT232R USB UART IC*, Mar. 2012, ver. 2.10.

[33] Intel, *Cyclone V Technnical Reference Manual*, Jul. 2018.

[34] Processor architecture laboratory – École Polytechnique Fédérale de Lausanne, *SoC-FPGA Design Guide DE0-Nano-SoC Edition*, Abr. 2018, ver. 1.32.

[35] Rocketboards.org, *Embedded Linux Beginners Guide*, consultada en agosto de 2018. [Online]. Disponible: rocketboards.org/foswiki/Documentation/EmbeddedLinuxBeginnerSGuide

[36] Oguz Meteer, *Building embedded Linux for the Terasic DE10-Nano*, Ago. 2017, consultada en agosto de 2018. [Online]. Disponible: bitlog.it/hardware/building-embedded-linux-for-the-terasic-de10-nano-and-other-cyclone-v-soc-fpgas/

[37] H. Vangala, *Polar Codes in MATLAB*, consultada en septiembre de 2017. [Online]. Disponible: www.polarcodes.com

[38] L. Arnone, M. Liberatori, L. Rabioglio, C. Gayoso, y J. C. Moreira, "Signal Amplitude Limiter based Soft Distance Decoding Algorithm for Polar Codes over Impulsive Noise Channels," *XVII Workshop on Information Processing and Control (RPIC)*, Sep. 2017.

Apéndice A

Códigos polares utilizados

Se listan los códigos utilizados durante el trabajo. Los mismos fueron calculados a partir de los límites de Bhattacharyya. Se especifican junto a los canales y estados asociados con los parámetros (N, K, F), donde aparecen:

- La cantidad de bits del mensaje codificado, N.
- La cantidad de bits del mensaje sin codificar, K.
- El conjunto $\mathbb{F}$ de los $N - K$ índices de los canales menos capaces, denominados *frozen bits*. Expresados en forma hexadecimal con la posición de cada bit asociada a un índice de canal, y considerando: 0 → índice frozen ; 1 → índice con información.

Códigos polares:

1. AWGN 0 dB – 4 dB → (128, 64, FFFFFFFEFFF8E880 FEE8E00080000000).
2. AWGN 5 dB → (128, 64, FFFFFFFCFFE8E880 FEE8E800C0000000).
3. AWGN 6 dB → (128, 64, FFFFFFF8FEE8E880 FEE8E880E0000000).
4. AWGN 0 dB → (128, 96, FFFFFFFFFFFFFFE8 FFFEFEE8FEC08000).
5. AWGN 1 dB – 2 dB → (128, 96, FFFFFFFFFFFFFFE8 FFFFFEE8FE808000).
6. AWGN 3 dB – 4 dB → (128, 96, FFFFFFFFFFFFFFE8 FFFEFEE8FEC08000).
7. AWGN 5 dB → (128, 96, FFFFFFFFFFFFFEE8 FFFEFEE8FEE08000).
8. AWGN 6 dB → (128, 96, FFFFFFFFFFFEFEE8 FFFEFEE8FEE88000).
9. AWGN 7 dB → (128, 96, FFFFFFFEFFFEFEE8 FFFEFEE8FEE8C000).
10. BSC 0,01 – 0,10 → (128, 64, FFFFFFFEFFF8E880 FEE8E00080000000).
11. BSC 0,02 – 0,15 → (128, 64, FFFFFFFEFFF8E880 FEE8E00080000000).
12. AWGN 0 dB – 4 dB → (64, 32, FFFEFEC0 FC808000).
13. AWGN 5 dB – 6 dB → (64, 32, FFFEFEE0 F8808000).

Apéndice B

Código VHDL

Se presenta en esta sección el código fuente de algunas de las entidades particularmente relevantes para la síntesis del sistema de corrección de errores con códigos polares.

B.1. Paquete de configuración

Entidad con constantes, tipos y funciones requerido por la mayoría de las entidades del sistema.

Modificando solamente los valores de las constantes N y Q se pueden sintetizar codificadores y decodificadores para distintas longitudes de mensajes y cantidades de bits de cuantificación de los LLR.

Modificando N se modifica la longitud de mensaje utilizada por el sistema de pruebas.

```
1  --*****************************************************************************
2  --! Nombre archivo: pkg_polar_codec.vhd
3  --!
4  --! Descripción: Package con utilidades para el codec polar.
5  --!
6  --! Generics: -
7  --!
8  --! Entradas: -
9  --!
10 --! Salidas: -
11 --!
12 --! Display: -
13 --!
14 --! Archivos requeridos: -
15 --!
16 --! Copyright&copy - Federico K.
17 --*****************************************************************************
18 library ieee;
19 use ieee.std_logic_1164.all;
20 use ieee.numeric_std.all;
21
22 --! Local libraries
23 library work;
24
25 --! Entity/Package Description
26 package pkg_polar_codec is
27
28     -- Declaración de constantes:
29     constant N : natural := 128; -- Longitud en bits del mensaje codificado.
30     constant K : natural := 64; -- Longitud en bits del mensaje sin codificar.
31     constant N_BASE : natural := 4; -- Longitud en bits del mensaje codificado
```

```
32      -- del deco base en decodificador recursivo.
33      constant Q_BITS : natural := 5; -- Cantidad de bits de cuantificación
34      -- (signo + magnitud) de los LLR.
35      constant SIGN_BIT : natural := Q_BITS - 1; -- Posición del bit de signo
36      -- de los LLR.
37      constant MAG_BITS : natural := Q_BITS - 1; -- Cantidad de bits para
38      -- representar magnitud de LLR.
39      constant ENC_MAX_CNT : natural := 0; -- Máximo valor a contar por FSM
40      -- para esperar codificación.
41      constant DECO_MAX_CNT : natural := 48; -- Máximo valor a contar por FSM
42      -- para esperar decodificación.
43
44      -- Declaración de tipos:
45      subtype llr is std_logic_vector(Q_BITS - 1 downto 0); -- LLR con mag y
46      -- signo.
47      subtype llr_mag is std_logic_vector(Q_BITS - 2 downto 0); -- Parte con
48      -- la magnitud de los LLR.
49      type llr_base_set is array(0 to N_BASE - 1) of llr;
50    type llr_set is array(0 to N - 1) of llr;
51      type llr_record is record
52              llr_in_rec : llr; --std_logic_vector(Q_BITS - 1 downto 0);
53          end record;
54      type llr_rec_set is array(natural range <>) of llr_record;
55
56      -- Devuelve el logaritmo en base 2 de enteros hasta 2147483647 = 2**31 - 1
57      -- (enteros de 32 bits)
58      function pc_log2(arg : in natural) return natural;
59
60      -- Devuelve un std_logic_vector con sus bits en orden BR. Función por
61      -- Jonathan Bromley en Google groups: comp.lang.vhdl
62      function reverse_any_vector(a : in std_logic_vector)
63         return std_logic_vector;
64
65      -- Devuelve el valor bit-reversed de un índice
66      function bitrev(index : in natural; len : in natural) return natural;
67
68  end package pkg_polar_codec;
69
70  --! Descripción del cuerpo del package
71  package body pkg_polar_codec is
72
73      function pc_log2(arg : in natural) return natural is
74      variable i : natural;
75      begin
76          i := 0;
77          while (2**i < arg) and i < 31 loop
78              i := i + 1;
79          end loop;
80          return i;
81      end function pc_log2;
82
83      function reverse_any_vector(a : in std_logic_vector)
84          return std_logic_vector is
85          variable result : std_logic_vector(a'range);
86          alias aa : std_logic_vector(a'reverse_range) is a;
87      begin
88          for i in aa'range loop
89              result(i) := aa(i);
90          end loop;
91          return result;
92      end function reverse_any_vector;
```

```
93
94      function bitrev(index : in natural; len : in natural) return natural is
95          variable auxi_slv : std_logic_vector(pc_log2(len) - 1 downto 0);
96          variable br_index : natural;
97      begin
98      auxi_slv := std_logic_vector(to_unsigned(index, pc_log2(len)));
99          br_index := to_integer(unsigned(reverse_any_vector(auxi_slv)));
100         return br_index;
101     end function bitrev;
102
103 end package body pkg_polar_codec;
```

B.2. Entidades del codificador

B.2.1. Codificador

Entidad que construye el codificador a partir de la matriz **F**. La aplicación del reorden bit-reverse de entradas o salidas se efectúa en una entidad de jerarquía superior, instanciando esta entidad.

```
1   --*****************************************************************************
2   --! Nombre archivo: gen_polar_encoder.vhd
3   --!
4   --! Descripcion:    Aplica codificación polar a un mensaje que fue ordenado
5   --!                 según las posiciones de los fzn bits. Bit-reverse a la salida.
6   --!
7   --! Generics:    GENC_N : longitud en bits del bloque codificado.
8   --!
9   --! Entradas:    fzn_msg(GENC_N) : mensaje frozen.
10  --!
11  --! Salidas:    e_msg(GENC_N) : mensaje codificado.
12  --!
13  --! Display: -
14  --!
15  --! Archivos requeridos: f_one.vhd, reverse_shuffler.vhd, pkg_polar_codec.vhd
16  --!
17  --! Copyright&copy - Federico K.
18  --*****************************************************************************
19  library ieee;
20  use ieee.std_logic_1164.all;
21
22  --! Local libraries
23  library work;
24  use work.pkg_polar_codec.all;
25
26  --! Entity/Package Description
27  entity gen_polar_encoder is
28      generic(
29          GENC_N : natural := 128
30      );
31      port(
32          fzn_msg : in std_logic_vector(GENC_N - 1 downto 0);
33          e_msg : out std_logic_vector(GENC_N - 1 downto 0)
34      );
35          end entity gen_polar_encoder;
36
37  architecture rtl of gen_polar_encoder is
38
39      constant TOTAL_STAGES : natural := pc_log2(GENC_N);
40      type conn_record is record -- Declara registro y arreglo para conectar
```

```
41      -- bloques del generate.
42              rs_to_fone : std_logic_vector(GENC_N - 1 downto 0);
43              fone_to_rs : std_logic_vector(GENC_N - 1 downto 0);
44           end record;
45      type conn_total_stages is array(0 to TOTAL_STAGES - 1) of conn_record;
46      signal conn_array : conn_total_stages;
47      signal sig_fzn_msg : std_logic_vector(GENC_N - 1 downto 0);
48
49      -- Declaraciones de componentes: ----------------------------------
50      component f_one is
51      generic(
52          N : natural := 128
53      );
54      port(
55          f_one_in : in std_logic_vector(N - 1 downto 0);
56          f_one_out : out std_logic_vector(N - 1 downto 0)
57      );
58      end component f_one;
59
60      component reverse_shuffler is
61      generic(
62          N : natural := 128
63      );
64      port(
65          r_shuffler_in : in std_logic_vector(N - 1 downto 0);
66          r_shuffler_out : out std_logic_vector(N - 1 downto 0)
67      );
68      end component reverse_shuffler;
69          ------------------------------------------------------------------------
70
71  begin
72
73      sig_fzn_msg <= fzn_msg;
74
75      encoder : for i in 0 to (TOTAL_STAGES - 1) generate
76
77      begin
78          first_blocks : if i = 0 generate -- Genera bloques de la
79          -- primera etapa.
80
81              first_shuffler : reverse_shuffler
82              generic map(
83                  N => GENC_N
84              )
85              port map(
86                  r_shuffler_in => sig_fzn_msg,
87                  r_shuffler_out => conn_array(i).rs_to_fone
88              );
89
90              first_xors : f_one
91              generic map(
92                  N => GENC_N
93              )
94              port map(
95                  f_one_in => conn_array(i).rs_to_fone,
96                  f_one_out => conn_array(i).fone_to_rs
97              );
98          end generate first_blocks;
99
100         other_blocks : if i > 0 generate -- Genera bloques de las otras
101         -- etapas.
```

```
102
103             r_shufflers : reverse_shuffler
104             generic map(
105                 N => GENC_N
106             )
107             port map(
108                 r_shuffler_in => conn_array(i-1).fone_to_rs,
109                 r_shuffler_out => conn_array(i).rs_to_fone
110             );
111
112             xors : f_one
113             generic map(
114                 N => GENC_N
115             )
116             port map(
117                 f_one_in => conn_array(i).rs_to_fone,
118                 f_one_out => conn_array(i).fone_to_rs
119             );
120         end generate other_blocks;
121
122     end generate encoder;
123
124     e_msg <= conn_array(TOTAL_STAGES - 1).fone_to_rs; -- Asigna señales a
125     -- puertos.
126
127 end architecture rtl;
```

B.2.2. Kernel de la matriz generadora

```
1  --*****************************************************************************
2  --! Nombre archivo: f_one.vhd
3  --!
4  --! Descripcion: Aplica el operador F^1 a los N bits de la entrada, tomados de
5  --!              a pares.
6  --!
7  --! Generics: N : longitud en bits del bloque codificado.
8  --!
9  --! Entradas: f_one_in(N)
10 --!
11 --! Salidas: f_one_out(N)
12 --!
13 --! Display: -
14 --!
15 --! Archivos requeridos: -
16 --!
17 --! Copyright&copy - Federico K.
18 --*****************************************************************************
19 library ieee;
20 use ieee.std_logic_1164.all;
21 use ieee.numeric_std.all;
22
23 --! Local libraries
24 library work;
25
26 --! Entity/Package Description
27 entity f_one is
28     generic(
29         N : natural := 16
30     );
31     port(
```

```
        f_one_in : in std_logic_vector(N - 1 downto 0);
        f_one_out : out std_logic_vector(N - 1 downto 0)
    );
end entity f_one;

architecture rtl of f_one is

begin
    process(f_one_in)
    begin
        for i in 0 to N/2 - 1 loop -- N=16 -> 0 to 7
            f_one_out(2*i) <= f_one_in(2*i) xor f_one_in(2*i + 1);
            f_one_out(2*i + 1) <= f_one_in(2*i + 1);
        end loop;
    end process;
end architecture rtl;
```

B.2.3. Mezclador para matriz generadora

```
--*******************************************************************************
--! Nombre archivo: reverse_shuffler.vhd
--!
--! Descripcion: Implementa mezclado de los bits del mensaje de entrada para
--!              utilización en codificador polar.
--!
--! Generics: N : longitud en bits del bloque codificado.
--!
--! Entradas: r_shuffler_in(N)
--!
--! Salidas: r_shuffler_out(N)
--!
--! Display: -
--!
--! Archivos requeridos: -
--!
--! Copyright&copy - Federico K.
--*******************************************************************************
library ieee;
use ieee.std_logic_1164.all;

--! Local libraries
library work;

--! Entity/Package Description
entity reverse_shuffler is
    generic(
        N : natural := 16
    );
    port(
        r_shuffler_in : in std_logic_vector(N - 1 downto 0);
        r_shuffler_out : out std_logic_vector(N - 1 downto 0)
    );
end entity reverse_shuffler;

architecture rtl of reverse_shuffler is

begin
    process(r_shuffler_in)
    begin
        -- Ej.: N=16 -> (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
```

```
        -- <= (0,2,4,6,8,10,12,14,1,3,5,7,9,11,13,15)
        for i in 0 to N/2 - 1 loop -- N=16 -> 0 to 7
            r_shuffler_out(i) <= r_shuffler_in(2*i);
            r_shuffler_out(i + N/2) <= r_shuffler_in(2*i + 1);
        end loop;
    end process;
end architecture rtl;
```

B.3. Entidades del decodificador

B.3.1. Decodificador recursivo

Entidad programada de forma recursiva que implementa el decodificador combinacional.

```
--*****************************************************************************
--! Nombre archivo: rec_polar_deco.vhd
--!
--! Descripcion: Deco recursivo para códigos polares. Usa cancelaciones sucesivas.
--!
--! Generics: Constantes en pkg_polar_codec.
--!
--! Entradas: llr_msg :  record con los LLR del canal.
--!           fzn_bits :  vector con las posiciones de los fzn bits (0 = fzn).
--!
--! Salidas: fzn_msg : vector con mensaje frozen estimado.
--!
--! Display: -
--!
--! Archivos requeridos: base_polar_deco.vhd, gen_polar_encoder_brout.vhd,
--!                      pkg_polar_codec.vhd, m_pe_muxd_gen.vhd
--!
--! Copyright&copy - Federico K.
--*****************************************************************************
library ieee;
use ieee.std_logic_1164.all;

--! Local libraries
library work;
use work.pkg_polar_codec.all;

--! Entity/Package Description
entity rec_polar_deco is
        generic(
                RDECO_N : natural := N
        );
    port(
                llr_msg : in llr_rec_set(0 to RDECO_N - 1);
                fzn_bits : in std_logic_vector(RDECO_N - 1 downto 0);
                fzn_msg : out std_logic_vector(RDECO_N - 1 downto 0)
    );
end entity rec_polar_deco;

architecture rec of rec_polar_deco is

        -- Declaración de señales
        signal sig_lfs, sig_lgs : llr_rec_set(0 to RDECO_N/2 - 1);
        signal sig_partial_sums : std_logic_vector(RDECO_N/2 - 1 downto 0);
        signal sig_fzn_msg : std_logic_vector(RDECO_N - 1 downto 0);
        signal sig_fzn_msg_top, sig_fzn_msg_bot :
        std_logic_vector(RDECO_N/2 - 1 downto 0);
```

```
47
48          -- Declaración de componentes
49          component base_polar_deco is
50          generic(
51                  BDECO_N : natural
52          );
53      port(
54                  llr_in : in llr_rec_set(0 to BDECO_N - 1);
55                  fzn_bits : in std_logic_vector(BDECO_N - 1 downto 0);
56                  fzn_msg : out std_logic_vector(BDECO_N - 1 downto 0)
57      );
58          end component base_polar_deco;
59
60          component m_pe_muxd_gen is
61          generic(
62                  MPEGEN_N : natural
63          );
64     port(
65                  in_llr : in llr_rec_set(0 to MPEGEN_N - 1);
66                  in_partial_sums : in std_logic_vector(MPEGEN_N/2 - 1 downto 0);
67                  out_lfs : out llr_rec_set(0 to MPEGEN_N/2 - 1);
68                  out_lgs : out llr_rec_set(0 to MPEGEN_N/2 - 1)
69    );
70          end component m_pe_muxd_gen;
71
72          component gen_polar_encoder_brout is
73          generic(
74                  GENC_N : natural
75          );
76          port(
77                  fzn_msg : in std_logic_vector(GENC_N - 1 downto 0);
78                  e_msg : out std_logic_vector(GENC_N - 1 downto 0)
79          );
80          end component gen_polar_encoder_brout;
81
82          component rec_polar_deco is
83          generic(
84                  RDECO_N : natural
85          );
86      port(
87                  llr_msg : in llr_rec_set(0 to RDECO_N - 1);
88                  fzn_bits : in std_logic_vector(RDECO_N - 1 downto 0);
89                  fzn_msg : out std_logic_vector(RDECO_N - 1 downto 0)
90      );
91      end component rec_polar_deco;
92          ------------------------------------------------------------
93
94  begin
95
96          base_case : if RDECO_N = 8 generate
97          begin
98                  base_mpe_gen : m_pe_muxd_gen
99                          generic map(
100                                 MPEGEN_N => RDECO_N
101                         )
102                         port map(
103                                 in_llr(0 to RDECO_N - 1) =>
104                     llr_msg(0 to RDECO_N - 1),
105                                 in_partial_sums =>
106                     sig_partial_sums(RDECO_N/2 - 1 downto 0),
107                                 out_lfs => sig_lfs(0 to RDECO_N/2 - 1),
```

```
108                             out_lgs => sig_lgs(0 to RDECO_N/2 - 1)
109                     );
110
111             base_deco_top : base_polar_deco
112                     generic map(
113                             BDECO_N => RDECO_N/2
114                     )
115                     port map(
116                             llr_in(0 to RDECO_N/2 - 1) =>
117                 sig_lfs(0 to RDECO_N/2 - 1),
118                             fzn_bits => fzn_bits(RDECO_N/2 - 1 downto 0),
119                             fzn_msg => sig_fzn_msg(RDECO_N/2 - 1 downto 0)
120                     );
121
122             base_encoder : gen_polar_encoder_brout
123                     generic map(
124                             GENC_N => RDECO_N/2
125                     )
126                     port map(
127                             fzn_msg => sig_fzn_msg(RDECO_N/2 - 1 downto 0),
128                             e_msg => sig_partial_sums(RDECO_N/2 - 1 downto 0)
129                     );
130
131             base_deco_bot : base_polar_deco
132                     generic map(
133                             BDECO_N => RDECO_N/2
134                     )
135                     port map(
136                             llr_in(0 to RDECO_N/2 - 1) =>
137                 sig_lgs(0 to RDECO_N/2 - 1),
138                             fzn_bits => fzn_bits(RDECO_N - 1 downto RDECO_N/2),
139                             fzn_msg => sig_fzn_msg(RDECO_N - 1 downto RDECO_N/2)
140                     );
141         end generate base_case;
142
143         general_case : if RDECO_N > 8 generate
144         begin
145                 sub_mpe_gen : m_pe_muxd_gen
146                         generic map(
147                                 MPEGEN_N => RDECO_N
148                         )
149                         port map(
150                                 in_llr => llr_msg(0 to RDECO_N - 1),
151                                 in_partial_sums =>
152                     sig_partial_sums(RDECO_N/2 - 1 downto 0),
153                                 out_lfs(0 to RDECO_N/2 - 1) =>
154                     sig_lfs(0 to RDECO_N/2 - 1),
155                                 out_lgs => sig_lgs(0 to RDECO_N/2 - 1)
156                         );
157
158                 sub_decoder_top : rec_polar_deco
159                         generic map(
160                                 RDECO_N => RDECO_N/2
161                         )
162                         port map(
163                                 llr_msg => sig_lfs(0 to RDECO_N/2 - 1),
164                                 fzn_bits => fzn_bits(RDECO_N/2 - 1 downto 0),
165                                 fzn_msg(RDECO_N/2 - 1 downto 0) =>
166                     sig_fzn_msg(RDECO_N/2 - 1 downto 0)
167                         );
168
```

```
169                 sub_encoder : gen_polar_encoder_brout
170                         generic map(
171                                 GENC_N => RDECO_N/2
172                         )
173                         port map(
174                                 fzn_msg => sig_fzn_msg(RDECO_N/2 - 1 downto 0),
175                                 e_msg => sig_partial_sums(RDECO_N/2 - 1 downto 0)
176                         );
177
178                 sub_decoder_bot : rec_polar_deco
179                         generic map(
180                                 RDECO_N => RDECO_N/2
181                         )
182                         port map(
183                                 llr_msg => sig_lgs(0 to RDECO_N/2 - 1),
184                                 fzn_bits => fzn_bits(RDECO_N - 1 downto RDECO_N/2),
185                                 fzn_msg => sig_fzn_msg(RDECO_N - 1 downto RDECO_N/2)
186                         );
187         end generate general_case;
188
189         fzn_msg <= sig_fzn_msg;
190
191 end architecture rec;
```

B.3.2. Elemento de procesamiento combinado

```
1  --*****************************************************************************
2  --! Nombre archivo: m_pe_muxd.vhd
3  --!
4  --! Descripcion: Processing element que implementa las funciones f y g del deco.
5  --!              out_lg0 = llr_bot + llr_top, out_lg1 = llr_bot - llr_top
6  --!
7  --! Generics: Q_BITS : nro. de bits de cuantificación para representación sig + mag.
8  --!           SIGN_BIT : posición del bit de signo
9  --!
10 --! Entradas: in_llr_top : llr superior
11 --!           in_llr_bot : llr inferior
12 --!           in_partial_sum : suma parcial para decisión
13 --!
14 --! Salidas:  out_lg : llr g superior (suma si suma p. = 0, resta si suma p. = 1)
15 --!           out_lf : llr f
16 --!
17 --! Display: -
18 --!
19 --! Archivos requeridos: comp_sel_min.vhd, full_add_sub.vhd, half_add_sub.vhd,
20 --!                      mux_2to2.vhd, parametric_and.vhd, parametric_or.vhd,
21 --!                      pkg_polar_codec.vhd,saturator.vhd, type_i_pe.vhd,
22 --!                      us_compare_agb.vhd
23 --!
24 --! Copyright&copy - Federico K.
25 --*****************************************************************************
26 library ieee;
27 use ieee.std_logic_1164.all;
28
29 --! Local libraries
30 library work;
31
32 use work.pkg_polar_codec.all;
33
34 --! Entity/Package Description
```

```
35 entity m_pe_muxd is
36         port(
37                 in_llr_top : in std_logic_vector(Q_BITS - 1 downto 0);
38                 in_llr_bot : in std_logic_vector(Q_BITS - 1 downto 0);
39                 in_partial_sum : in std_logic;
40                 out_lf : out std_logic_vector(Q_BITS - 1 downto 0);
41                 out_lg :        out std_logic_vector(Q_BITS - 1 downto 0)
42         );
43 end entity m_pe_muxd;
44
45 architecture rtl of m_pe_muxd is
46
47         -- Declaraciones de señales:
48         signal sig_minsub_sw, sig_dif_sig: std_logic;
49         signal sig_x, sig_y : std_logic_vector(Q_BITS - 2 downto 0);
50         signal sig_s, sig_d : std_logic_vector(Q_BITS - 2 downto 0);
51         signal sig_cout : std_logic; -- Carry out del type_i_pe
52         signal sig_saturator_out : std_logic_vector(Q_BITS - 2 downto 0);
53         signal sig_slg0, sig_slg1 : std_logic; -- Signos de los Lg sin corregir cero
54         signal sig_maglg0, sig_maglg1 : std_logic_vector(Q_BITS - 2 downto 0);
55         signal sig_lg0, sig_lg1 : std_logic_vector(Q_BITS - 1 downto 0);
56         signal sig_out_slf : std_logic; -- PARA USAR ZF CON F
57         signal sig_out_maglf : std_logic_vector(Q_BITS - 2 downto 0);
58
59         -- Señales test
60         signal sig_out_slg0, sig_out_slg1 : std_logic;
61         signal sig_out_maglg0, sig_out_maglg1 : std_logic_vector(Q_BITS - 2 downto 0);
62         signal sig_test_in_magtop, sig_test_in_magbot :
63                 std_logic_vector(Q_BITS - 2 downto 0);
64         signal sig_test_in_stop, sig_test_in_sbot : std_logic;
65
66         --------------------------------------------------------------------------
67
68 begin
69         sig_dif_sig <= in_llr_top(SIGN_BIT) xor in_llr_bot(SIGN_BIT); -- Signo de F
70
71         -- Instanciaciones de componentes:
72         inst_mux_2to2 : entity work.mux_2to2
73                 generic map(
74                         DATA_WIDTH => MAG_BITS
75                 )
76                 port map(
77                         in_a => sig_saturator_out,
78                         in_b => sig_d,
79                         sel => sig_dif_sig,
80                         output_a => sig_maglg0,
81                         output_b => sig_maglg1
82                 );
83
84         inst_saturator : entity work.saturator
85                 generic map(
86                         DATA_WIDTH => MAG_BITS
87                 )
88                 port map(
89                         sat_sel => sig_cout,
90                         data_in => sig_s,
91                         sat_out => sig_saturator_out
92                 );
93
94         inst_type_i_pe : entity work.type_i_pe
95                 generic map(
```

```
96                      DATA_WIDTH => MAG_BITS
97              )
98              port map(
99                      x => sig_x,
100                     y => sig_y,
101                     s => sig_s,
102                     d => sig_d,
103                     borrow_out => OPEN,
104                     carry_out => sig_cout
105             );
106
107     inst_comp_sel_min : entity work.comp_sel_min
108             generic map(
109                     DATA_WIDTH => MAG_BITS
110             )
111             port map(
112                     in_a => in_llr_top(Q_BITS - 2 downto 0),
113                     in_b => in_llr_bot(Q_BITS - 2 downto 0),
114                     min_out => sig_x,
115                     max_out => sig_y,
116                     data_sw => sig_minsub_sw
117             );
118
119     inst_g_pe_sign_calc : entity work.g_pe_sign_calc
120             port map(
121                     minsub_sw => sig_minsub_sw,
122                     sign_top_in => in_llr_top(SIGN_BIT),
123                     sign_bot_in => in_llr_bot(SIGN_BIT),
124                     sign_lg0 => sig_slg0,
125                     sign_lg1 => sig_slg1
126             );
127
128     ------------------------------------------------------------
129     sig_lg0(SIGN_BIT) <= sig_slg0;
130     sig_lg0(Q_BITS - 2 downto 0) <= sig_maglg0;
131     sig_lg1(SIGN_BIT) <= sig_slg1;
132     sig_lg1(Q_BITS - 2 downto 0) <= sig_maglg1;
133
134     out_lf(SIGN_BIT) <= sig_dif_sig; -- Asignaciones a puertos de salida
135     out_lf(Q_BITS - 2 downto 0) <= sig_x;
136
137     with in_partial_sum select -- MUX para seleccionar la salida de g adecuada
138             out_lg <= sig_lg0 when '0',
139               sig_lg1 when others;
140
141 end architecture rtl;
```

B.3.3. Decodificador básico

```
1  --*****************************************************************************
2  --! Nombre archivo: base_polar_deco.vhd
3  --!
4  --! Descripcion: Decodificador polar combinacional para N = 4. Base del
5  --!              deco recursivo.
6  --!
7  --! Generics: Constantes en pkg_codec_polar.
8  --!
9  --! Entradas:  llr_in : array de record con los LLR.
10 --!            fzn_bits : vector con las posiciones de los fzn bits (0 = fzn)
11 --!
```

```
12 --! Salidas: fzn_msg : mensaje frozen estimado.
13 --!
14 --! Display: -
15 --!
16 --! Archivos requeridos: us_compare_agb.vhd, m_pe_muxd.vhd, mux_2to2.vhd,
17 --!                       mux_2to1.vhd, type_i_pe, parametric_and.vhd,
18 --!                       full_add_sub.vhd, half_add_sub.vhd, pkg_codec_polar.vhd
19 --!
20 --! Copyright&copy - Federico K.
21 --*****************************************************************************
22 library ieee;
23 use ieee.std_logic_1164.all;
24
25 --! Local libraries
26 library work;
27 use work.pkg_polar_codec.all;
28
29 --! Entity/Package Description
30
31 entity base_polar_deco is
32         generic(
33                 BDECO_N : natural := 4
34         );
35     port(
36                 llr_in : in llr_rec_set(0 to BDECO_N - 1);
37                 fzn_bits : in std_logic_vector(BDECO_N - 1 downto 0);
38                 fzn_msg : out std_logic_vector(BDECO_N - 1 downto 0)
39     );
40 end entity base_polar_deco;
41
42 architecture rtl of base_polar_deco is
43     -- Señales:
44         signal sig_mux_2to1_TOP_in_a, sig_mux_2to1_TOP_in_b, sig_mux_2to1_TOP_output :
45             std_logic_vector(0 downto 0); -- Para poder trabajar con los puertos de
46             -- mux_2to1 paramétrico.
47         signal sig_mux_2to1_TOP_sel : std_logic; -- Para poder trabajar con los
48         -- puertos de mux_2to1 paramétrico.
49         signal sig_mux_2to1_BOT_in_a, sig_mux_2to1_BOT_in_b, sig_mux_2to1_BOT_output :
50             std_logic_vector(0 downto 0); -- Para poder trabajar con los puertos de
51             -- mux_2to1 paramétrico.
52         signal sig_mux_2to1_BOT_sel : std_logic;
53
54         signal sig_us_compare_agb_TOP_in_a, sig_us_compare_agb_TOP_in_b :
55             std_logic_vector(Q_BITS - 2 downto 0);
56         signal sig_us_compare_agb_TOP_agb : std_logic;
57         signal sig_us_compare_agb_BOT_in_a, sig_us_compare_agb_BOT_in_b :
58             std_logic_vector(Q_BITS - 2 downto 0);
59         signal sig_us_compare_agb_BOT_agb : std_logic;
60
61         signal sig_out_lf01, sig_out_lf23 : std_logic_vector(Q_BITS - 1 downto 0);
62         signal sig_out_lg01, sig_out_lg23 : std_logic_vector(Q_BITS - 1 downto 0);
63
64         signal sig_xor_sl0_sl1, sig_xor_sl2_sl3 : std_logic;
65         signal sig_fzn_msg : std_logic_vector(BDECO_N - 1 downto 0);
66         signal sig_slg01, sig_slg23 : std_logic;
67         signal sig_xor_slg01_slg23 : std_logic;
68         signal sig_partial_sum01, sig_partial_sum23 : std_logic;
69
70     -- Declaraciones de componentes:
71     component m_pe_muxd is
72         port (
```

```
73              in_llr_top : in std_logic_vector(Q_BITS - 1 downto 0);
74              in_llr_bot : in std_logic_vector(Q_BITS - 1 downto 0);
75              in_partial_sum : in std_logic;
76              out_lf : out std_logic_vector(Q_BITS - 1 downto 0);
77              out_lg : out std_logic_vector(Q_BITS - 1 downto 0)
78          );
79          end component m_pe_muxd;
80
81      component mux_2to1 is
82          generic(
83              DATA_WIDTH : natural
84          );
85          port(
86              in_a : in std_logic_vector(DATA_WIDTH - 1 downto 0);
87              in_b : in std_logic_vector(DATA_WIDTH - 1 downto 0);
88              sel : in std_logic;
89              output : out std_logic_vector(DATA_WIDTH - 1 downto 0)
90          );
91      end component mux_2to1;
92
93      component us_compare_agb is
94          generic(
95              DATA_WIDTH : natural
96          );
97          port(
98              in_a : in std_logic_vector(DATA_WIDTH - 1 downto 0);
99              in_b : in std_logic_vector(DATA_WIDTH - 1 downto 0);
100             agb : out std_logic
101         );
102         end component us_compare_agb;
103
104 begin
105     -- Instanciación de componentes:
106     inst_us_compare_agb_TOP : us_compare_agb
107         generic map(
108             DATA_WIDTH => MAG_BITS
109         )
110         port map(
111             in_a => sig_us_compare_agb_TOP_in_a,
112             in_b => sig_us_compare_agb_TOP_in_b,
113             agb => sig_us_compare_agb_TOP_agb
114     );
115
116     inst_mux_2to1_TOP : mux_2to1
117         generic map(
118             DATA_WIDTH => 1
119         )
120         port map(
121             in_a => sig_mux_2to1_TOP_in_a,
122             in_b => sig_mux_2to1_TOP_in_b,
123             sel => sig_mux_2to1_TOP_sel,
124             output => sig_mux_2to1_TOP_output
125     );
126
127     inst_m_pe_muxd_01 : m_pe_muxd
128         port map(
129             in_llr_top => llr_in(0).llr_in_rec,
130             in_llr_bot => llr_in(1).llr_in_rec,
131             in_partial_sum => sig_partial_sum01,
132             out_lf => sig_out_lf01,
133             out_lg => sig_out_lg01
```

```
134     );
135
136     inst_m_pe_muxd_23 : m_pe_muxd
137         port map(
138             in_llr_top => llr_in(2).llr_in_rec,
139             in_llr_bot => llr_in(3).llr_in_rec,
140             in_partial_sum => sig_partial_sum23,
141             out_lf => sig_out_lf23,
142             out_lg => sig_out_lg23
143     );
144
145     inst_us_compare_agb_BOT : us_compare_agb
146         generic map(
147             DATA_WIDTH => MAG_BITS
148         )
149         port map(
150             in_a => sig_us_compare_agb_BOT_in_a,
151             in_b => sig_us_compare_agb_BOT_in_b,
152             agb => sig_us_compare_agb_BOT_agb
153     );
154
155     inst_mux_2to1_BOT : mux_2to1
156         generic map(
157             DATA_WIDTH => 1
158         )
159         port map(
160             in_a => sig_mux_2to1_BOT_in_a,
161             in_b => sig_mux_2to1_BOT_in_b,
162             sel => sig_mux_2to1_BOT_sel,
163             output => sig_mux_2to1_BOT_output
164     );
165         ------------------------------------------------------------------------------------
166
167         -- Asignaciones a señales:
168         ------- Mitad superior del deco:
169         sig_us_compare_agb_TOP_in_a <= sig_out_lf01(Q_BITS - 2 downto 0);
170         sig_us_compare_agb_TOP_in_b <= sig_out_lf23(Q_BITS - 2 downto 0);
171         sig_mux_2to1_TOP_sel <= sig_us_compare_agb_TOP_agb;
172
173         sig_xor_sl0_sl1 <= sig_out_lf01(SIGN_BIT);
174         sig_xor_sl2_sl3 <= sig_out_lf23(SIGN_BIT);
175
176         sig_fzn_msg(0) <= (sig_xor_sl0_sl1 xor sig_xor_sl2_sl3) and fzn_bits(0);
177
178         sig_mux_2to1_TOP_in_a(0) <= sig_xor_sl2_sl3; -- Invierte entradas porque este
179         -- MUX opera con sel invertido.
180         sig_mux_2to1_TOP_in_b(0) <= sig_xor_sl0_sl1 xor sig_fzn_msg(0);
181
182         sig_fzn_msg(1) <= sig_mux_2to1_TOP_output(0) and fzn_bits(1); -- Para poder
183         -- trabajar con los puertos de mux_2to1 paramétrico.
184
185         ------- Mitad inferior del deco:
186         sig_partial_sum01 <= sig_fzn_msg(0) xor sig_fzn_msg(1);
187         sig_partial_sum23 <= sig_fzn_msg(1);
188
189         sig_us_compare_agb_BOT_in_a <= sig_out_lg01(Q_BITS - 2 downto 0);
190         sig_us_compare_agb_BOT_in_b <= sig_out_lg23(Q_BITS - 2 downto 0);
191         sig_mux_2to1_BOT_sel <= sig_us_compare_agb_BOT_agb;
192
193         sig_slg01 <= sig_out_lg01(SIGN_BIT);
194         sig_slg23 <= sig_out_lg23(SIGN_BIT);
```

```
195
196         sig_xor_slg01_slg23 <= sig_slg01 xor sig_slg23;
197         sig_fzn_msg(2) <= sig_xor_slg01_slg23 and fzn_bits(2);
198
199         sig_mux_2to1_BOT_in_a(0) <= sig_slg23; -- Invierte entradas porque
200         -- este MUX opera con sel invertido.
201         sig_mux_2to1_BOT_in_b(0) <= sig_slg01 xor sig_fzn_msg(2);
202
203         sig_fzn_msg(3) <= sig_mux_2to1_BOT_output(0) and fzn_bits(3);
204         -- Para poder trabajar con los puertos de mux_2to1 paramétricos.
205
206         ------- Asignaciones a puertos de salida
207         fzn_msg(0) <= sig_fzn_msg(0);
208         fzn_msg(1) <= sig_fzn_msg(1);
209         fzn_msg(2) <= sig_fzn_msg(2);
210         fzn_msg(3) <= sig_fzn_msg(3);
211
212 end architecture rtl;
```