PONTIFICIA **UNIVERSIDAD CATÓLICA** DEL PERÚ

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

DISEÑO DE LA TRANSFORMADA RÁPIDA DE FOURIER CON ALGORITMO SPLIT-RADIX EN FPGA

Tesis para optar el Título de Ingeniero Electrónico, que presenta la bachiller:

Cynthia Lidia Watanabe Kanno

Asesor: Carlos Silva Cárdenas

Lima, octubre de 2009

RESUMEN

La Transformada Rápida de Fourier Split-Radix (SRFFT) es un algoritmo computacionalmente eficiente que se utiliza para calcular la Transformada Discreta de Fourier (DFT), la cual a partir de una secuencia finita de datos, obtiene otra que describe su comportamiento en el dominio de la frecuencia. Esta herramienta se utiliza en óptica, acústica, física cuántica, teorías de sistemas, tratamiento de señales, reconocimiento de voz, entre otros.

En el presente trabajo se muestra el diseño de una arquitectura para efectuar la Transformada Rápida de Fourier Split-Radix (SRFFT) en un dispositivo lógico programable como es el FPGA (Field Programmable Gate Array), en particular la Cyclone II EP2C35F672C6 de la compañía Altera. El diseño se realiza mediante el uso del lenguaje de descripción de hardware VHDL en el entorno de desarrollo Quartus II versión 9.0. Esta arquitectura es capaz de analizar $N=2^P$ datos de entrada ($P \geq 2$) , de diversos números de bits. Para las primeras pruebas se tomaron N=1024 datos de entrada de 16 bits cada uno. Luego, se registraron resultados con 256, 32 y 16 datos de entrada. Mediante el software Matlab R2006a se generó la señal de entrada de prueba (una señal de dos tonos) que se convierte a números enteros y los coeficientes que son escalados para el desarrollo de la SRFFT. La simulación ideal del sistema se realizó también en Matlab con los objetivos de comprender mejor el algoritmo y tener un punto de referencia para compararlo con las simulaciones del circuito realizadas en el software ModelSim 6.4a. Al realizar las comparaciones, se obtuvo una diferencia promedio de 0.0369 según Norma L2.

Para el diseño de la arquitectura, se analizó el algoritmo Split-Radix y se diseñaron las principales unidades funcionales de la arquitectura global para luego detallar el esquema de cada una de ellas. Así, se explican las tareas de cada etapa y la secuencia que deben seguir para su correcto funcionamiento. Después, se presenta la simulación de toda la arquitectura junto con el detalle del uso de recursos y algunas comparaciones con trabajos realizados en la misma área.

La arquitectura propuesta en la presente tesis muestra un tiempo de ejecución elevado (tiempo comprendido desde el ingreso de cada uno de los datos hasta su lectura serial en la salida), pero tiene la gran ventaja de trabajar con $N=2^P$ datos de entrada ($P \geq 2$), de cualquier número de bits, y de utilizar pocos recursos como funciones combinacionales, registros lógicos y memoria. Adicionalmente, con el diseño de este trabajo se obtuvo un consumo de potencia bajo.

A mi papi, que me sigue cuidando desde arriba, cuyos ojos son siempre bondad y amor.
A mi mami, por darme siempre lo mejor, por preocuparse por cada paso que doy, por todo su sacrificio y amor.
A mis hermanos, Juan y Gustavo, por ser mis acompañantes y soportes en la vida.
A mis padrinos, Gaby y tío Duchi, por su cariño, consejos y constante apoyo.
A mis tíos, Teresa, Rosa, Inés y Pepe, por su preocupación y afecto.
A mis dos obachan y a mi ojichan, por demostrarme lo que es coraje, fortaleza y servicio.
A mi tía Maria Rosa, por su apoyo y alegría.
A Nito, por compartir conmigo cada día.
Al Dr. Carlos Silva, por su confianza y ayuda en todo momento.
Al Dr. Paul Rodríguez, a quien le agradezco todo su tiempo y conocimiento.
A todos mis profesores y amigos, por sus enseñanzas y aliento.
A la Universidad y en especial al Grupo de Microelectrónica, por darme la oportunidad de alcanzar un óptimo desarrollo profesional y ser el lugar donde he encontrado grandes amistades.

Muchísimas Gracias a todos.

"Ya, Cynthia…déjate de tonterías…ponte a estudiar…"

Juan Watanabe Sato
Mi Papá (1946-2006)

ÍNDICE

INTRODUCCIÓN

La Transformada Rápida de Fourier (FFT) es un aporte importante al desarrollo de la tecnología de tratamiento de señales [1]. La FFT transforma una señal discreta en el dominio del tiempo a su representación en el dominio de la frecuencia de una manera más rápida y eficiente que la Transformada Discreta de Fourier (DFT) [1-2]. Es ampliamente utilizada en filtros, procesamiento de sonido, comunicaciones (modulaciones, líneas de transmisión), estadística, detección de fluctuaciones en los precios, análisis sismográfico, entre otros [2].

Dentro de las optimizaciones que presentan la FFT sobre la DFT, se encuentra la minimización del número de operaciones que se necesita en el procesamiento de la señal. En la presente tesis se trabaja con el algoritmo Split-Radix [3-4], del cual se debe estudiar su estructura interna, su complejidad operacional y averiguar cómo puede diseñarse una arquitectura eficiente que desarrolle este algoritmo con el mínimo número de recursos.

Para efectuar estas operaciones que deben realizarse de manera rápida y precisa, se puede optar por una solución software o una solución hardware. En esta última, se encuentran las soluciones de arquitectura configurable, en donde la entrada del diseño se basa en esquemáticos o lenguajes HDL (Hardware Description Language). Dentro de esta gama de dispositivos incidiremos en el FPGA (Field Programmable Gate Array) que es un dispositivo lógico programable que permite diseñar arquitecturas óptimas para aplicaciones específicas.

En el presente trabajo, se busca diseñar una arquitectura para desarrollar el algoritmo de la Transformada Rápida de Fourier Split-Radix para $N=2^P$ datos de entrada ($P \geq 2$) de cualquier número de bits, que utilice pocos recursos y consuma poca potencia. Para ello, se emplea el lenguaje VHDL (acrónimo que representa la combinación de VHSIC y HDL, donde VHSIC es el acrónimo de Very High Speed Integrated Circuit y HDL es a su vez el acrónimo de Hardware Description Language; VHDL es un lenguaje que se utiliza para diseñar circuitos digitales) con el software Quartus II de Altera para el FPGA Cyclone II del mismo fabricante.

CAPÍTULO 1
TRANSFORMADA RÁPIDA DE FOURIER SPLIT-RADIX (SRFFT) EN FPGAs

En este primer capítulo, se introducen los términos de la Transformada Rápida de Fourier o FFT y del dispositivo FPGA. También, se mencionan las características principales de cada uno de ellos para mayor comprensión del presente trabajo. Además, se detallan varias versiones de la Transformada Rápida de Fourier, en especial del algoritmo denominado Split-Radix. Al final del capítulo, se justifica el diseño del Split-Radix en un FPGA.

1.1. Alcances de la Transformada Rápida de Fourier (FFT)

Los diferentes algoritmos de la Transformada Rápida de Fourier (FFT, Fast Fourier Transform) realizan de manera eficiente el cálculo de la Transformada Discreta de Fourier (DFT). Esta última es una transformación matemática muy importante en el área de análisis de frecuencia, ya que transforma una señal discreta en el dominio del tiempo a su representación discreta en el dominio de la frecuencia [1].

La FFT tiene la ventaja de ser la versión computacionalmente eficiente de la DFT debido a su considerable ahorro en el número de operaciones matemáticas. El número de operaciones que realiza la DFT es proporcional a N^2, mientras que el número de operaciones aritméticas en un algoritmo FFT crece con relación a $N\log_2(N)$. La idea que permite esta optimización es la descomposición de la transformada en otras más simples, las cuales se ejecutan de manera sucesiva [2].

Según la descomposición de la Transformada Rápida de Fourier en DFT más simples, la FFT se clasifica como decimación en tiempo o decimación en frecuencia. En la primera, se comienza por realizar las DFTs de menor longitud, mientras que en la segunda se empieza por la de mayor longitud. Además, en la primera, donde el cómputo se realiza en tiempo, los datos de entrada son previamente ordenados vía bit-reverso de manera que luego de realizar la FFT, los datos se encuentren en el orden correcto. En cambio, en la segunda, donde el cómputo se efectúa en frecuencia, los datos ingresan a la FFT en orden y al final deben ser reordenados vía bit-reverso [1].

Adicionalmente, la FFT tiene múltiples aplicaciones en diversas áreas. Se emplea en el análisis en frecuencia de cualquier señal discreta como en el procesamiento de voz, procesamiento de señales médicas.

1.2. Alcances del Dispositivo FPGA.

Un FPGA es un Dispositivo Lógico Programable (PLD) que soporta implementaciones de circuitos lógicos de más de un millón de compuertas equivalentes en tamaño [5].

Un PLD (Programmable Logic Device) es un circuito integrado de propósito general que contiene un conjunto de compuertas lógicas e interruptores programables. Estos interruptores programables permiten la conexión entre las compuertas para obtener cualquier circuito lógico requerido [5].

El FPGA presenta una estructura general que contiene bloques lógicos, recursos de interconexión y unidades de entrada y salida; la cual puede ser configurada a necesidad, logrando una gran diversidad de aplicaciones. Los recursos de *hardware* disponibles deben ser usados correctamente, es decir, optimizar al máximo el espacio requerido para una tarea determinada, que éste sea más veloz y que tenga un menor consumo de potencia.

El FPGA a utilizar en el presente trabajo es la Cyclone II de Altera, el cual es el dispositivo disponible en el Grupo de Microelectrónica de la Pontificia Universidad Católica del Perú. Éste cuenta con las siguientes características: 33 216 elementos lógico, 438 840 bits de memoria RAM interna, 35 multiplicadores embebidos de 18x18 bits y 4 PLLs [6]. Este FPGA se encuentra en la tarjeta de desarrollo DE2 Development and Education Board de la compañía TERASIC que incluye: un ALTERA CYCLONE II 2C35, 8MBytes de memoria SDRAM, 512 KBytes de memoria SRAM, un conector RS-232, 2 osciladores (uno de 50MHz y otro de 27MHz), un conector Ethernet y un conector USB [7].

El lenguaje que se emplea para diseñar los circuitos portables y reusables en el FPGA es un Lenguaje de Descripción de Hardware (HDL: Hardware Description Language), que describe el comportamiento del sistema electrónico. Para esto, se utiliza una de las denominadas herramientas EDA (Electronic Design Automation): Quartus II de Altera, creada especialmente para las tareas de síntesis, implementación y simulación de los circuitos descritos.

1.3. Algoritmos de la Transformada Rápida de Fourier (FFT)

Todos los algoritmos de la Transformada Rápida de Fourier se basan en descomponer la Transformada Discreta de Fourier (DFT) de N datos de entrada en un número de DFTs de menos muestras, donde sus salidas son reutilizadas de forma sucesiva para obtener el resultado final de la DFT de N datos [1]. Las muestras de entrada de las DFTs de menor tamaño son números enteros y factores del número N.

Los algoritmos de FFTs más utilizados son para entradas cuyo número de muestras es igual a una potencia de dos ($N=2^P$, donde P es el exponente del número dos que da como resultado N que es el número de entradas de la FFT). Algunos ejemplos son la FFT Radix-2, FFT Radix-4 ($N=4^P$), y la FFT Split-Radix. En el caso que N no sea igual a 2^P, se puede optar por añadir ceros a la secuencia hasta conseguir un número potencia de 2, esto se conoce como zero-padding [2]; o se puede optar por utilizar FFT para números primos.

Entre los algoritmos mencionados en el párrafo anterior, existen algunos que efectúan un menor número de operaciones aritméticas. La comparación entre el número de operaciones de estos algoritmos se puede apreciar en la Tabla 1.1. En esta tabla, se puede enfatizar el ahorro del 20% de carga computacional del Split-Radix sobre la Radix-2 [8].

Tabla 1.1. Número de multiplicaciones y adiciones reales para una DFT compleja de longitud N [3].

	Multiplicaciones reales				**Adiciones reales**			
N	Radix-2	Radix-4	Radix-8	Split-Radix	Radix-2	Radix-4	Radix-8	Split-Radix
16	24	20		20	152	148		148
32	88			68	408			388
64	264	208	204	196	1032	976	972	964
128	72			516	2054			2308
256	1800	1392		1284	5896	5488		5380
512	4360		3204	3076	13566		12420	12292
1024	10248	7856		7172	30728	28336		27652

Otra clase de algoritmo es el *Prime-Factor Algorithm (PFA)* [9, Capítulo 6.2.2], donde los números de muestras de entrada de las DFTs de menor tamaño son números primos entre sí. Estas DFTs más pequeñas requieren de un mayor costo computacional que las de los algoritmos en párrafos anteriores. Sin embargo, el PFA tiene la ventaja de reutilizar los resultados intermedios y de eliminar las multiplicaciones de los factores *Twiddle* (se refiere al coeficiente trigonométrico que se multiplica con la señal). Por ello, para números de datos de entrada similares, el costo computacional es comparable entre el Algoritmo Prime Factor y los algoritmos de número de muestras de entrada de potencia de 2.

El *Algoritmo Rader* [9, Capítulo 6.1.5] se utiliza para DFTs de números de datos de entrada primos y la *Transformada Chirp Z* [9, Capítulo 6.1.4] se emplea para DFTs de números de muestras arbitrarias. Ambas convierten las DFTs en convoluciones de otras longitudes para desarrollarlas usando FFTs vía convolución rápida.

Otro algoritmo denominado *Algoritmo Goertzel* [9, Capítulo 6.1.3] se utiliza en aplicaciones que requieren de pocas muestras en frecuencia, para lo cual reduce el costo computacional con relación al cálculo directo de la DFT.

1.4. Algoritmo Split-Radix.

La FFT Split-Radix (SRFFT) es uno de los algoritmos que realiza el cálculo de la Transformada Discreta de Fourier (DFT). Este algoritmo se propuso en 1968 por R. Yavne (dato que se encuentra en el trabajo de Johnson y Frigo titulado " A Modified Split-Radix FFT with Reduced Arithmetic Complexity" [8]), el cual ha sido empleado, modificado y optimizado por diversos investigadores durante varios años. Dos de los personajes más importantes son P. Duhamel y H. Hollmann, quienes publicaron los trabajos titulados "Split-Radix FFT Algorithm" [3] e "Implementation of Split-Radix FFT Algorithms for Complex, Real, and Real Symmetric Data" [4].

Así como los algoritmos de la SRFFT propuestos por Duhamel y Hollmann, existen otros que también se encargan de calcular la DFT de potencia de 2 con el objetivo de reducir el número de operaciones aritméticas como son Xu Peng, Chen Jin Shu, Steven Johnson, Matteo Frigo, Martin Vetterli, entre otros. (Ver trabajos [3,4,8,10-13])

La FFT Split-Radix combina dos métodos computacionales para calcular la DFT: el algoritmo Radix-2 y Radix-4 [2]. Internamente, la SRFFT decimación en frecuencia aplica el Radix-2 decimación en frecuencia para las muestras de número par de la DFT de N entradas. Esto se efectúa con la siguiente fórmula:

$$X(2k) = \sum_{n=0}^{N/2-1} \left[x(n) + x(n+\frac{N}{2}) \right] W_{N/2}^{nk}, \quad k = 0,1,\ldots,\frac{N}{2}-1 \qquad \ldots (1)$$

En esta fórmula y en las siguientes W_N^n es un coeficiente trigonométrico o factor *Twiddle* igual a $W_N^n = \cos\left(\frac{2\pi n}{N}\right) - j \cdot sen\left(\frac{2\pi n}{N}\right) = e^{-j2\pi n/N}$.

Con el algoritmo Radix-4 se desarrollan las muestras de número impar de la DFT, empleando las siguientes fórmulas:

$$X(4k+1) = \sum_{n=0}^{N/4-1} \left\{ \left[x(n) - x(n+N/2) \right] - j\left[x(n+N/4) - x(n+3N/4) \right] \right\} W_N^n W_{N/4}^{kn} \qquad \ldots(2)$$

$$X(4k+3) = \sum_{n=0}^{N/4-1} \left\{ \left[x(n) - x(n+N/2) \right] + j\left[x(n+N/4) - x(n+3N/4) \right] \right\} W_N^{3n} W_{N/4}^{kn} \qquad \ldots(3)$$

Donde $k = 0,1,\ldots,\frac{N}{4}-1$.

El resultado de la DFT de N datos de entradas se obtiene gracias al cálculo sucesivo de estas descomposiciones Radix-2 y Radix-4. Así, se tiene el algoritmo SRFFT decimación en frecuencia.

A continuación, se muestra un diagrama de flujo para un algoritmo SRFFT decimación en frecuencia de sólo 32 puntos con sus respectivas mariposas.

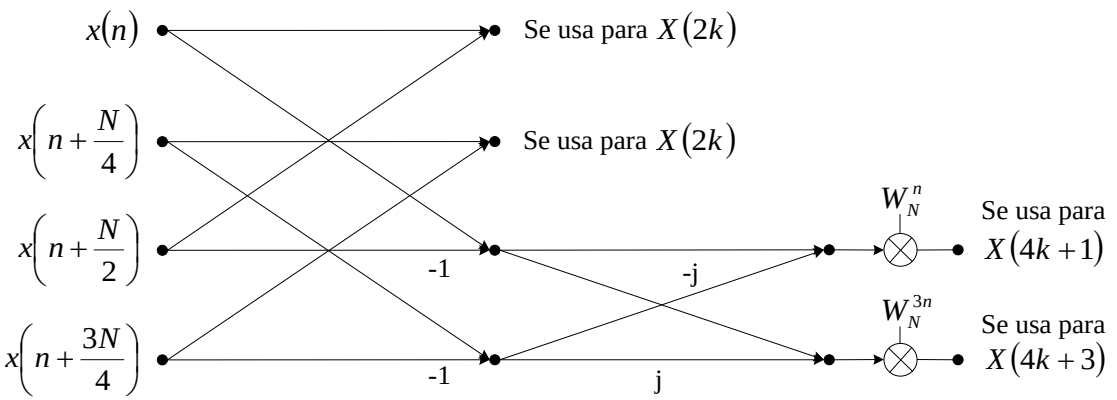


Figura 1.1. Mariposa para el algoritmo SRFFT [2].

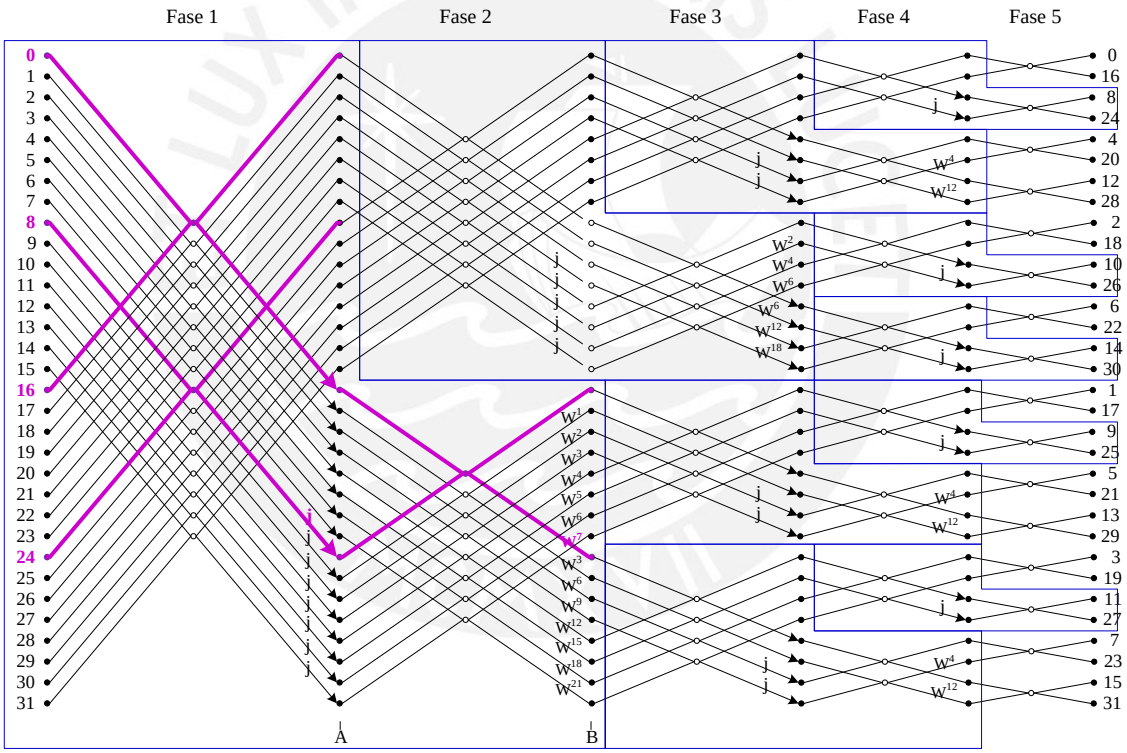


Figura 1.2. Diagrama de flujo para un algoritmo SRFFT decimación en frecuencia de longitud 32 [2].

En la Figura 1.2, se puede observar que al repetir varias veces la mariposa de la Figura 1.1 (líneas de color morado), se forman los bloques definidos en azul en forma de L que segmentan el diagrama de flujo del algoritmo SRFFT.

En la fase final de este diagrama de flujo, se debe realizar también una serie de mariposas básicas detalladas en la siguiente figura:

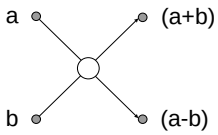


Figura 1.3. Mariposa básica.

Finalmente, se puede observar en la Figura 1.2 que las entradas de los bloques en azul en cada fase no siguen un patrón definido, por lo que su diseño en VHDL resulta complicado. Por ello, en este trabajo se desarrolla el Split-Radix simétrico [4] explicado en el Capítulo 3.

La mariposa de este Split-Radix simétrico resuelve las mismas operaciones vistas con la mariposa de la Figura 1.1, pero los resultados se encuentran en diferente orden; con lo cual se pueden predecir qué datos van a ser las entradas para la siguiente mariposa (ver Figura 3.3). Esto ayuda en gran medida a la hora de diseñar la arquitectura, pero tiene la desventaja de que la salida de la SRFFT ya no se encuentra en orden vía bit-reverso. Por ejemplo, la salida 16 que en binario es 1000 ya no se encuentra en la posición uno como lo indica su valor en binario invertido 0001. Por lo tanto, dentro de la arquitectura diseñada de la presente tesis existe una etapa que se encarga de ordenar la salida. El método de este ordenamiento se puede apreciar en la parte final del Capítulo 3.2.

1.5. Fundamentación de la SRFFT en FPGA.

La presente tesis pretende un mejor entendimiento del algoritmo Split-Radix que fue propuesta por Duhamel [3] como una herramienta de análisis de frecuencia para el procesamiento digital de señales.

Este algoritmo fue elegido ya que posee varias ventajas como un bajo número de adiciones y multiplicaciones, la misma regularidad estructural que los algoritmos Radix-4, la misma flexibilidad que los algoritmos Radix-2, no existe reordenamiento de datos dentro del algoritmo, y está numéricamente bien condicionada como la Radix-4 [4].

En el presente trabajo se diseña la SRFFT simétrica decimación en frecuencia en VHDL de acuerdo al propuesto por Pierre Duhamel en su trabajo “Implementation of Split-Radix FFT Algorithms for Complex, Real, and Real-Symmetric Data” [4]. Se escogió este algoritmo por su mayor regularidad estructural en comparación con la SRFFT original [3]. La SRFFT simétrica

se detalla en el Capítulo 3. Sin embargo, en este capítulo se explica la SRFFT decimación en frecuencia original para una mayor comprensión.

Para la primera prueba del diseño de la arquitectura SRFFT se considera 1024 entradas de 16 bits. Sin embargo, esta arquitectura tiene la ventaja de poder trabajar $N=2^P$ datos de entrada ($P \geq 2$), de diferentes números de bits, como se ve en la Tabla 4.1 donde se muestran resultados con 256, 32 y 16 entradas.

El diseño de la arquitectura de la SRFFT propuesta en este trabajo se realiza en punto fijo para mayor facilidad en la manipulación de datos. No obstante, esto genera ciertos errores al truncar los resultados de las multiplicaciones y en el desbordamiento en las adiciones y sustracciones que se tienen en cuenta a la hora de obtener los resultados.

Adicionalmente, para las diferentes aplicaciones de la SRFFT, se requiere diseñar un sistema portable, flexible, rápido y eficiente. Estas condiciones del sistema se obtienen a través del diseño físico en un dispositivo de lógica programable como es el FPGA, que proporciona numerosas ventajas como portabilidad, capacidad para modificar su programación, alta densidad de integración en un sólo circuito integrado y alta velocidad en el procesamiento de las señales de entrada.

CAPÍTULO 2
PLANTEAMIENTOS PARA LA IMPLEMENTACIÓN DE LA FFT SPLIT-RADIX

En esta parte del trabajo se plantean las hipótesis, se indican los objetivos y se señala el procedimiento a seguir para alcanzar el diseño eficiente de la SRFFT en FPGA.

2.1. Hipótesis de la Investigación.

2.1.1. Hipótesis Principal.

Dado que en la actualidad, la tecnología avanza rápidamente junto con el amplio campo de acción de la FFT, se conseguirá el diseño de una arquitectura genérica que implemente la FFT Split-Radix, y que permita variar sus parámetros y amoldar sus características para que sea posible utilizarlo en cualquier aplicación como en filtros y sonido, comunicaciones (modulaciones, líneas de transmisión), estadística, detección de fluctuaciones en los precios, análisis sismográfico, entre otros.

2.1.2. Hipótesis Secundarias.

1) Se contará con un diseño en hardware de un algoritmo FFT que requiera de un número reducido de operaciones, poca memoria, y que consuma una baja cantidad de recursos y potencia.

2) El desarrollo del algoritmo Split-Radix sobre un FPGA permitirá obtener una arquitectura flexible y adaptable a distintas aplicaciones, esto otorgará la posibilidad de expandir el diseño sobre sí mismo o integrar todo un sistema en un ASIC como diseño posterior. En otras palabras, se tendrá un gran nivel de escalaridad.

2.2. Objetivos de la Investigación.

2.2.1. Objetivo General.

Diseñar una arquitectura que realice la FFT Split-Radix decimación en frecuencia en un FPGA para $N=2^P$ datos de entrada ($P \geq 2$), de cualquier número de bits.

2.2.2. Objetivos Específicos.

1) Codificar la FFT Split-Radix en un software como MATLAB para la mejor comprensión del algoritmo.

2) Diseñar la arquitectura de la FFT Split-Radix, describirla en el FPGA utilizando VHDL en el software Quartus II de Altera, y simularlo para verificar su correcto funcionamiento.

3) Buscar en todo momento la optimización de los recursos del FPGA, así como diseños modulares de cada componente. Además, que el diseño global sea lo más flexible posible.

4) Obtener una arquitectura base con la cual sea factible realizar trabajos de investigación posteriores.

2.3. Universo y Muestra.

El universo está formado por las diferentes arquitecturas que diseñan la FFT en un FPGA y por la gran variedad de algoritmos y modificaciones existentes.

Por otra parte, la muestra está conformada por la arquitectura elegida para desarrollar la FFT Split-Radix en decimación en frecuencia para $N=2^P$ datos de entrada ($P \geq 2$) en el FPGA Cyclone II de Altera.

2.4. Metodología de la Investigación.

Para el diseño de la FFT Split-Radix, el procedimiento a seguir se divide de la siguiente manera:

Etapa 1: Investigación preliminar.

- Identificación de procesos y problemas resueltos mediante el diseño en hardware de la FFT.
- Estudio de la teoría de la FFT.
- Estudio de la FFT Split-Radix.
- Estudio de las características de las FPGAs.
- Estudio de las especificaciones de la FPGA Cyclone II de Altera.
- Investigación sobre temas específicos en descripción de sistemas con VHDL.

Etapa 2: Análisis y diseño del sistema.

• Análisis del sistema y determinación de requerimientos.

• Codificación de la FFT Split-Radix en software (MATLAB).

• Realización de pruebas y validación del código en MATLAB.

Etapa 3: Desarrollo de la solución.

• Descripción de la arquitectura de la FFT Split-Radix.

• Diseño en VHDL de la arquitectura descrita y simulación de la misma.

• Validación del diseño de la arquitectura propuesta y ajuste de sus parámetros.

• Pruebas finales.

Etapa 4: Resultados y Documentación.

• Establecimiento y medición de los resultados obtenidos.

• Elaboración de conclusiones.

• Elaboración de documentos periódicamente según el avance de las etapas.

• Elaboración del documento final.

CAPÍTULO 3
DISEÑO DE LA ARQUITECTURA DE LA FFT SPLIT-RADIX EN VHDL UTILIZANDO QUARTUS II DE ALTERA

En este capítulo, se explica el diagrama de flujo que sigue la FFT Split-Radix simétrica y el diseño de la arquitectura descrita en VHDL en el software Quartus II de Altera.

3.1. Consideraciones Preliminares.

Para algunas aplicaciones específicas de la FFT como es el diseño en hardware, donde se necesita optimizar el uso de los recursos, es muy útil incrementar la regularidad del algoritmo. Para obtener una estructura regular en el algoritmo Split-Radix, se puede emplear la mariposa de la Figura 3.2, que tiene las salidas en un orden diferente a las de la mariposa de la Figura 1.1. Para comparar ambas, la mariposa de la Figura 1.1 puede ser representada como en la Figura 3.1.

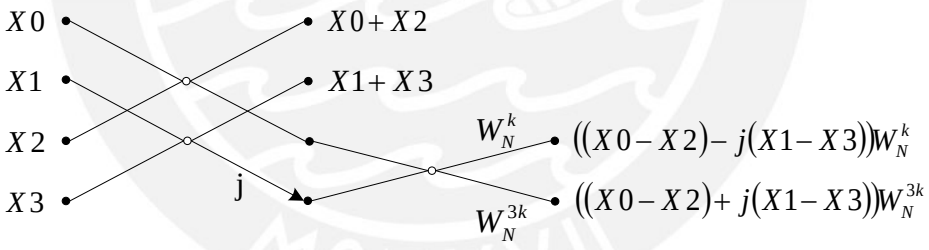


Figura 3.1. Mariposa de la FFT Split-Radix [4].

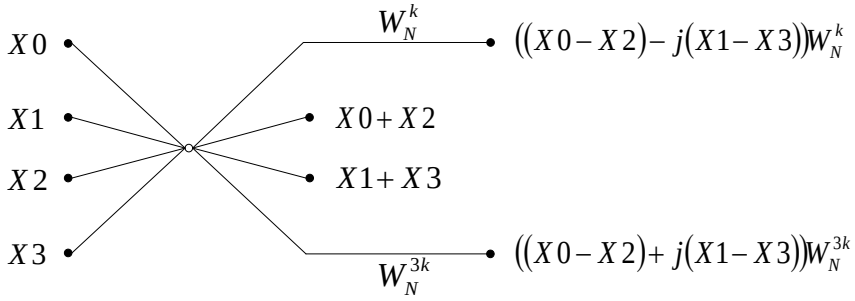


Figura 3.2. Mariposa de la FFT Split-Radix simétrica [4].

Como se puede apreciar en los diagramas, las dos mariposas FFT Split-Radix ejecutan las mismas operaciones y tienen las mismas salidas, pero en un orden distinto. El orden de las salidas de la mariposa de la Figura 3.2 brinda una gran ventaja a la SRFFT simétrica para su diseño en hardware.

El aporte de la mariposa simétrica a la regularidad estructural del algoritmo Split-Radix se puede apreciar al comparar las Figuras 1.2 y 3.3. En ésta última, los bloques azules siguen un patrón definido que ayuda a saber en qué parte de cada fase empieza un bloque de mariposas simétricas.

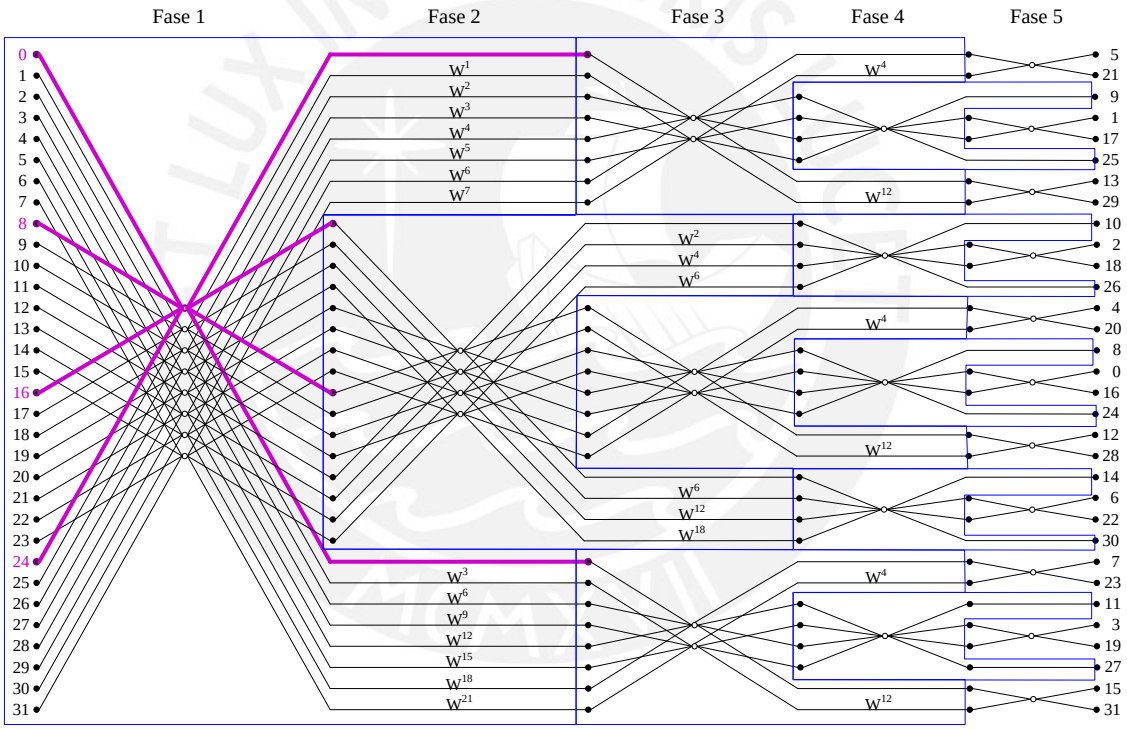


Figura 3.3. Diagrama de flujo para un algoritmo SRFFT simétrica de longitud 32 [8].

Si se aplica este esquema de la SRFFT simétrica para 32 entradas a una de 1024, los bloques se presentan como en la Figura 3.4. Estos bloques siguen un orden regular y cada tipo de bloque (diferenciado por un color diferente en cada fase) se repite $n_{-1} * 2 \pm 1$; donde n_{-1} es el resultado del número de bloques de la fase anterior, y el signo + y – se intercambia entre fases. Para la primera columna, n_{-1} es igual a 0 y se emplea el signo + como se puede observar en la parte superior de la Figura 3.4. Al final, luego de operar todas las mariposas SRFFT simétricas, se realizan 341 mariposas básicas como en la Figura 1.3.

En la Figura 3.5, se detalla el número de entradas para cada bloque de mariposas simétricas, que se representa con la letra “M”, el cual para la primera fase es igual a N y luego se va dividiendo entre dos en las siguientes fases. Además, se muestra el rango de “n” que determina cuántas veces se va a repetir la mariposa de la Figura 3.2 en cada bloque.

Cabe recordar que una mariposa tiene cuatro entradas: $x(n)$, $x\left(n+\frac{M}{4}\right)$, $x\left(n+\frac{M}{2}\right)$ y $x\left(n+\frac{3M}{4}\right)$. Por ello, “n” abarca desde 0 hasta M/4-1.

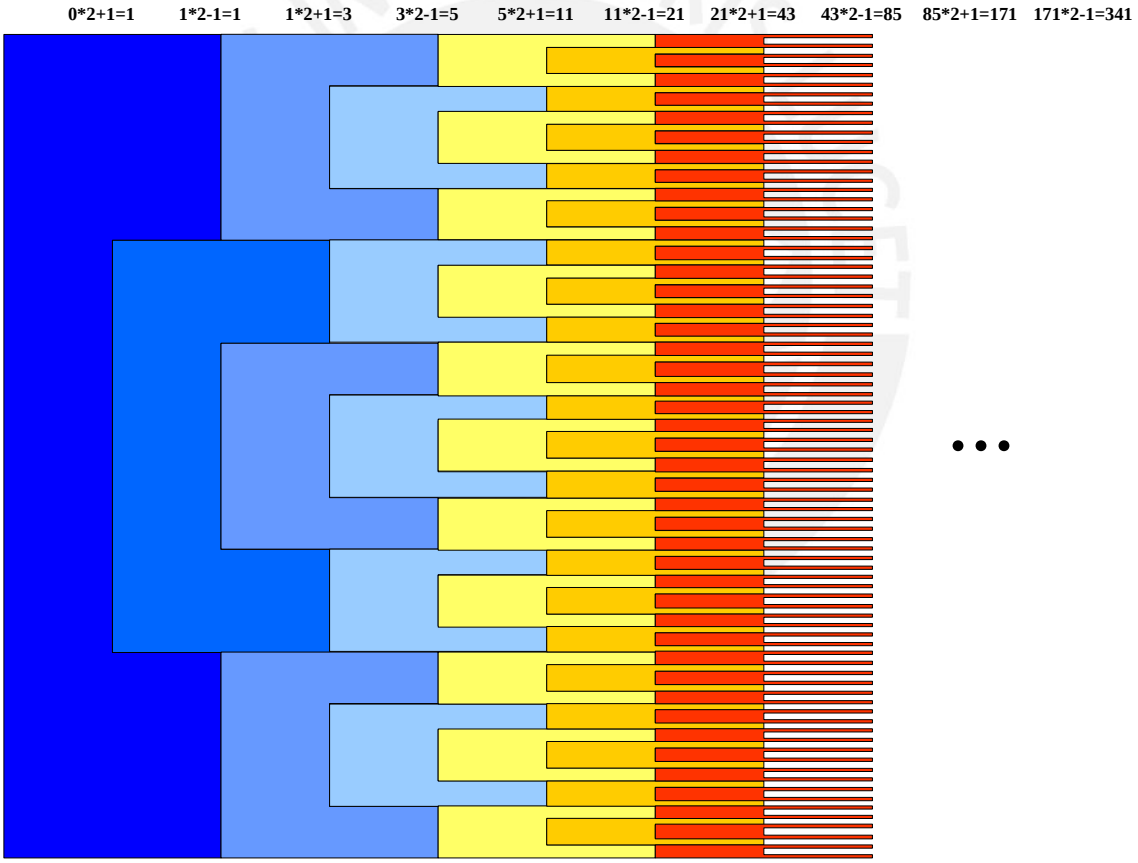


Figura 3.4. Diagrama de bloques para un algoritmo SRFFT simétrica de longitud 1024.

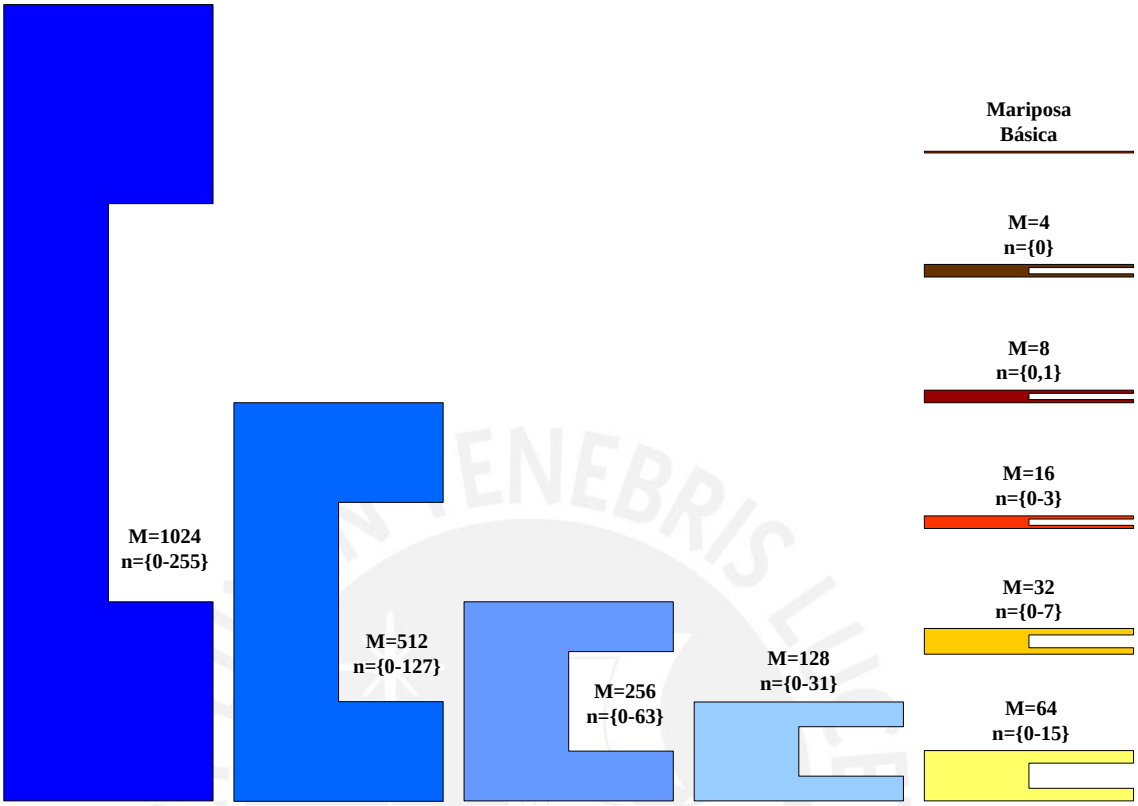


Figura 3.5. Especificación de bloques empleados en la SRFFT simétrica para 1024 entradas.

Por otro lado, los coeficientes trigonométricos o factores *Twiddle*, se generan a través del software Matlab. Estos coeficientes son números reales que para el diseño presentado en este trabajo, se van a escalar para obtener números enteros que van a operarse con los datos de entrada de la SRFFT que se requiere que sean también enteros.

Por ejemplo, si se tiene un dato con un valor de 10.35, a la entrada de la SRFFT se debe tener 1035 en binario. En el caso de los coeficientes, se escalan al multiplicarlos por $2^{(bits-1)}$ (bits es el número de bits de los datos de entrada) para luego redondearlos, y si el valor obtenido resulta $2^{(bits-1)}$ se le resta una unidad. Estos coeficientes son los elementos de las dos ROMs. Una ROM almacena la parte real de los factores *Twiddle* y la otra la parte imaginaria. Cabe resaltar, que en las ROMs no van a encontrarse los N (número de datos de entrada de la SRFFT) coeficientes trigonométricos, sino solamente la primera cuarta parte que representa el primer cuadrante del círculo unitario. Cuando se requiera un valor de otro cuadrante se calcula su equivalente y se cambia de signo si es necesario.

Los N factores *Twiddle* son los siguientes:

$$W_N^n = \cos\left(\frac{2\pi n}{N}\right) - j \cdot sen\left(\frac{2\pi n}{N}\right) = e^{-j2\pi n/N}$$

W_N^n : Coeficiente trigonométrico o factor Twiddle.

$\cos\left(\frac{2\pi n}{N}\right)$: Parte real del coeficiente.

$-sen\left(\frac{2\pi n}{N}\right)$: Parte imaginaria del coeficiente.

Donde N es el número de datos de entrada de la SRFFT y n toma valores desde 0 hasta N-1.

Otro punto importante es el rango de los números utilizados en el diseño. Tanto la cantidad de datos de entrada como el número de bits de los datos son configurables; es decir, se tiene la flexibilidad de poder escoger $N=2^P$ datos de entrada ($P \geq 2$) para la cantidad de valores que ingresan a la arquitectura propuesta y de cuántos bits son. La única restricción es la capacidad del FPGA que se está utilizando, en el caso de la Cyclone II de Altera que se emplea en esta tesis, para realizar una SRFFT de 1024 puntos de 16 bits sólo hace uso de menos del 15% de su capacidad (ver Figura 4.1), lo cual da un amplio rango de combinaciones de número de datos de entrada y de bits.

Finalmente, se recalca que se utilizan números de punto fijo para simplificar el diseño de la arquitectura. Sin embargo, esta elección trae consigo dos problemas. El primero es el truncamiento del producto en la multiplicación. Si la multiplicación se da entre dos datos de 16 bits cada uno, se obtiene una respuesta de 32 bits que va a ser reducida a sus 16 bits más significativos. El segundo problema es el desbordamiento en adiciones y sustracciones, donde se toma la acción de asignarle al resultado, el número más grande o más pequeño dentro del rango de valores que permite el número de bits de los operandos; si la respuesta no presenta desborde, la respuesta se mantiene.

3.2. Diseño de la FFT Split-Radix sobre el FPGA CYLONE II de ALTERA.

En esta sección del Capítulo 3, se presenta gráficamente la arquitectura de la FFT Split-Radix con sus partes detalladas y explicadas. Además, se describe el diseño en VHDL de esta misma arquitectura.

3.2.1. Descripción y Diseño de la Arquitectura de la FFT Split-Radix.

La arquitectura global de la FFT Split-Radix consta de cuatro etapas (Figura 3.6): la primera (*split_entrada*) corresponde a la etapa de entrada de datos, y a la escritura y lectura de la RAM; la segunda (*split_mariposa*) realiza la mariposa simétrica explicada en la sección 3.1, y la mariposa básica de la última fase de la FFT Split-Radix; la tercera (*maq_orden*) es la que se encarga de reordenar la salida, y la cuarta (*maq_split*) es la máquina de estados que controla los procesos de la primera, segunda y tercera parte a través de sus respectivas máquinas de estados.

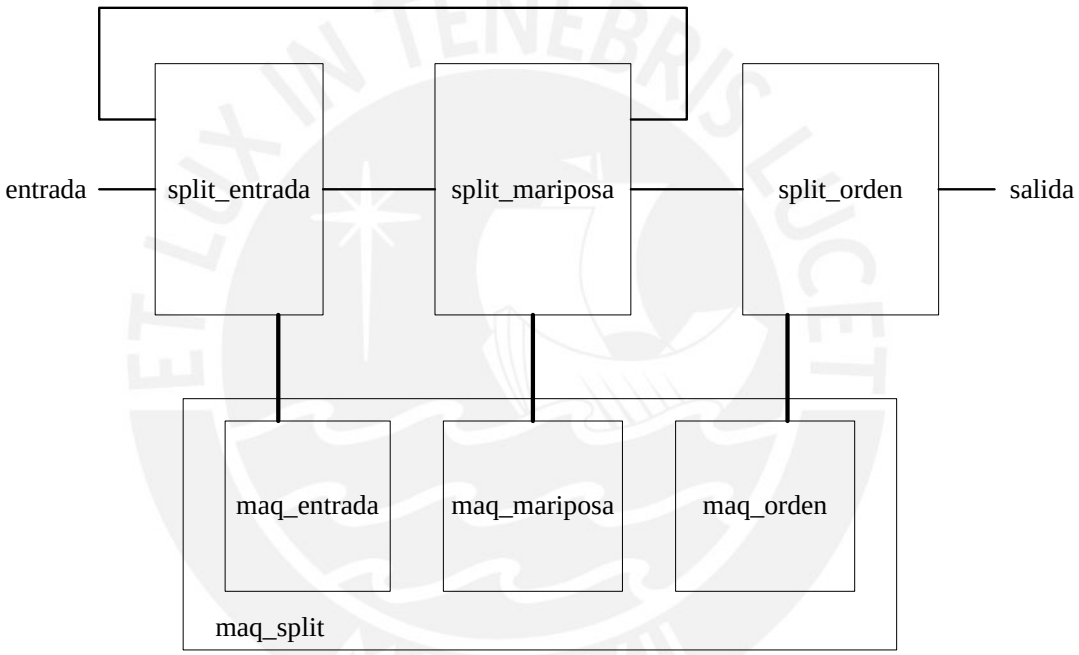


Figura 3.6. Arquitectura global de la FFT Split-Radix.

El diseño de la primera etapa, que se muestra en la Figura 3.7, se puede dividir en cuatro partes para su mejor comprensión. La primera es la entrada de datos a la RAM, representada en la Figura 3.8. En ella, se elige si lo que se escribe en memoria es la serie de datos que representan a la señal de entrada en el dominio del tiempo o los datos de salida provenientes del proceso mariposa. Estas entradas escritas en la RAM son de 2*β bits ya que están concatenadas la parte real de β bits con la imaginaria de β bits. El valor de β puede ser cualquier número entero.

En la Figura 3.9, se puede apreciar la arquitectura que se encarga de generar las direcciones para escribir y leer de la RAM. Las direcciones de escritura que proporciona son para llenar toda la memoria con los datos de entrada de la señal en tiempo. Las direcciones de lectura son las que

apuntan a los datos en la RAM que se utilizan para desarrollar la mariposa de la FFT Split-Radix.

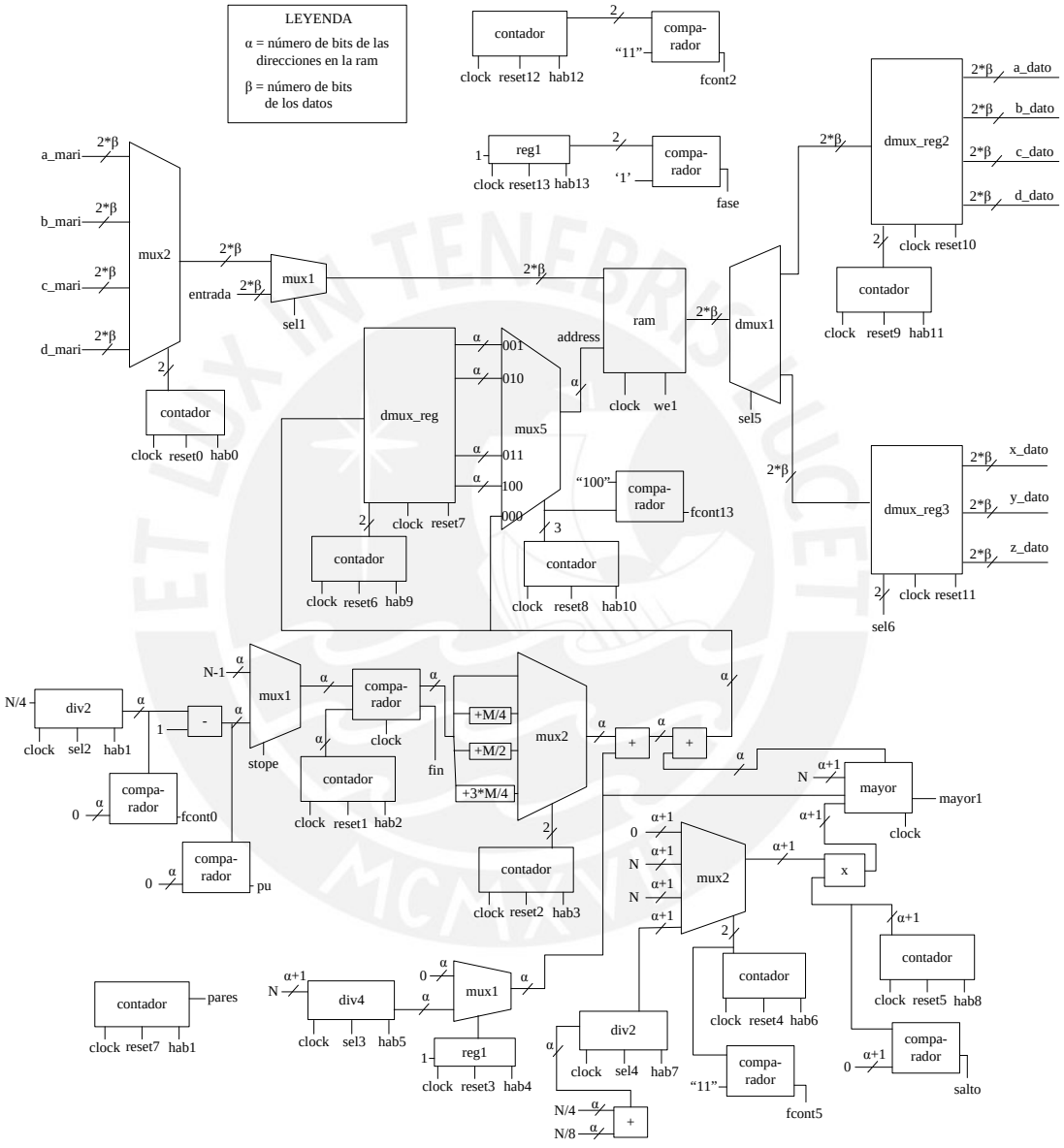


Figura 3.7. Diseño de la primera etapa de la arquitectura de la FFT Split-Radix.

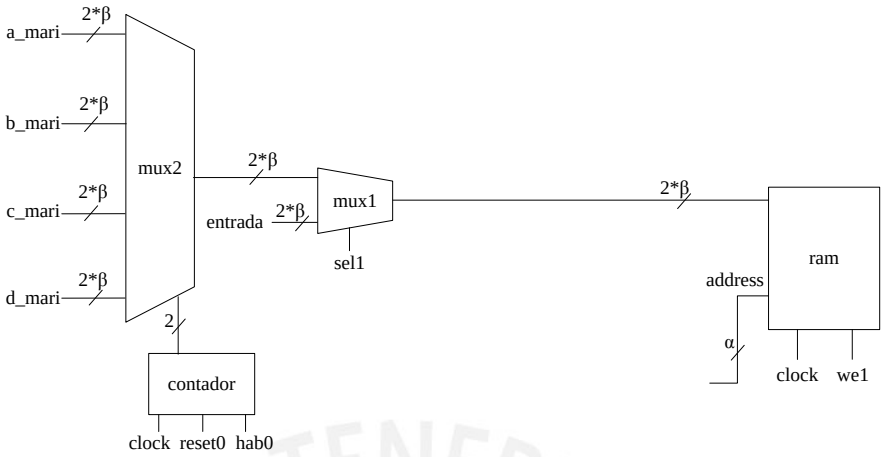


Figura 3.8. Entrada de datos a la RAM.

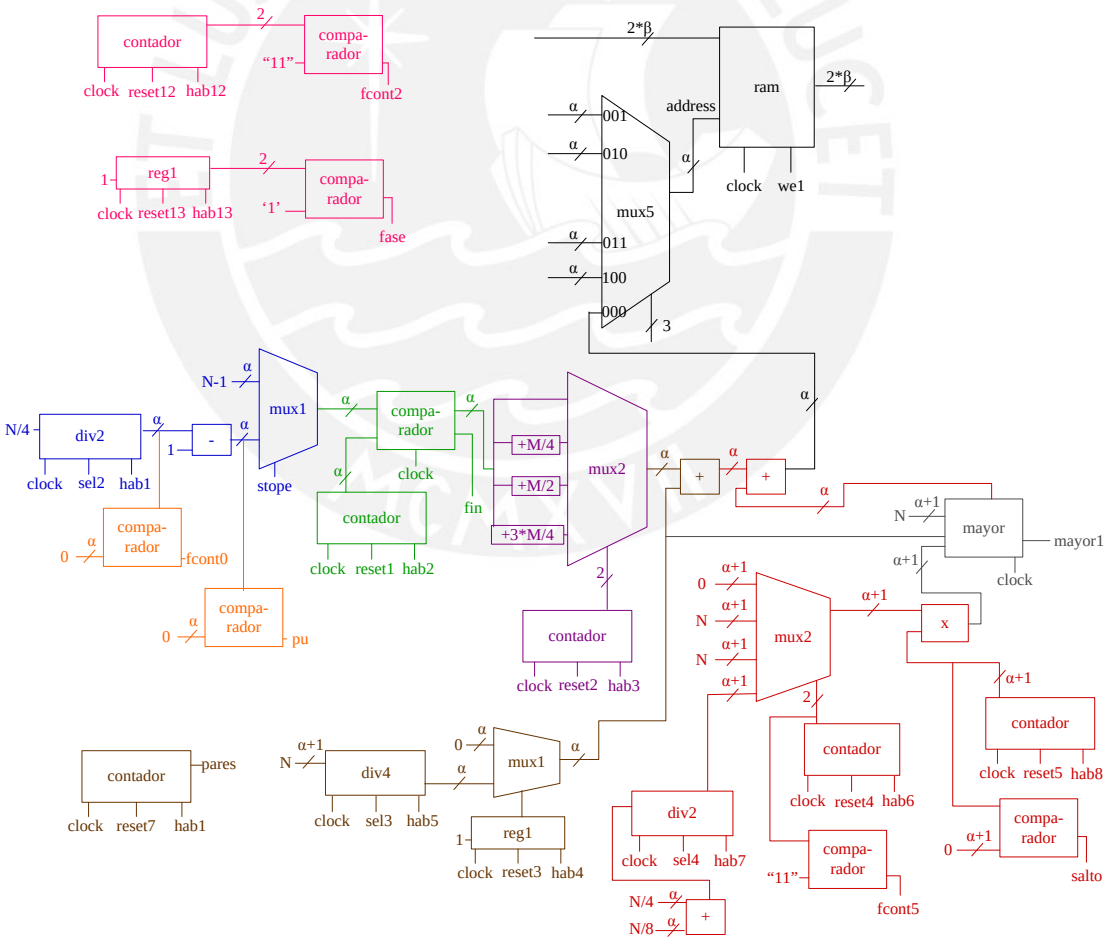


Figura 3.9. Direcciones para escritura de los datos de entrada y lectura de la RAM.

La parte de la Figura 3.9 señalada con color azul indica cuál es el último dato inicial que se escribe en la RAM, y hasta dónde se debe de leer para cada uno de los bloques conformados por las mariposas simétricas mostradas en la Figura 3.5 (los valores son los números mayores de los rangos "n"). Además, en el diagrama de color naranja, el *pu* indica cuándo se ha llegado a la antepenúltima fase para que los coeficientes trigonométricos en la mariposa simétrica sean (1+i0) y el *fcont0* señala en qué momento se llega a la penúltima fase de la SRFFT, donde se terminan de realizar todas las mariposas simétricas. La parte en color verde realiza la cuenta desde el valor cero hasta el valor tope dado por la parte azul e indica cuándo se llega a este valor. Lo delineado en morado, se encarga que se lean los datos $x(n)$, $x\left(n+\frac{M}{4}\right)$, $x\left(n+\frac{M}{2}\right)$ y $x\left(n+\frac{3M}{4}\right)$; y el *fcont2* de la parte en rosado, indica también el momento en que termina. La primera mariposa simétrica de cada fase de la FFT Split-Radix comienza en un valor distinto como se muestra en la Figura 3.4. Este valor de inicio lo establece la parte en marrón de la Figura 3.9; y se intercala entre ceros y N/4, donde el denominador de este último se multiplica por cuatro en cada fase intercalada. Adicionalmente, en la Figura 3.4, se puede apreciar que en cada fase existen varios bloques de mariposas simétricas que empiezan en distintos valores, estos valores de inicio los genera el diagrama en rojo de la Figura 3.9. Por ejemplo, para la tercera fase que tiene varias mariposas, la segunda empieza en $\frac{N}{4}+\frac{N}{8}$, y la tercera en $\left(\frac{N}{4}+\frac{N}{8}\right)\cdot 2$; para la siguiente fase, la segunda empieza en $\left(\frac{N}{4}+\frac{N}{8}\right)/2+ini$ (ini es el valor de inicio del primer bloque de mariposas de la fase, este valor es calculado por la parte de color marrón explicada anteriormente), la tercera en $\left(\left(\frac{N}{4}+\frac{N}{8}\right)/2\right)\cdot 2+ini$, la cuarta en $\left(\left(\frac{N}{4}+\frac{N}{8}\right)/2\right)\cdot 3+ini$, y así sucesivamente. Luego, para saber el momento en que termina una fase de la FFT Split-Radix, se diseñó la parte en color gris que señala si el valor generado para la dirección es mayor que el tamaño de la RAM. Para que este valor se actualice se esperan dos ciclos de reloj determinados por la variable fase de la parte en rosado.

La tercera parte de la primera etapa de la arquitectura global FFT Split-Radix es la mostrada en la Figura 3.10. Ella se encarga de mantener en cuatro registros las direcciones de lectura generadas por la segunda parte para las entradas de la mariposa simétrica. Así, cuando se tienen

los resultados de la mariposa, estos pueden ser escritos en los mismos espacios de memoria de donde han sido leídos. Además, el contador de la imagen se encargan de seleccionar las direcciones de los registros o la dirección dada por la segunda parte, y el comparador indica cuándo se terminan de leer los registros.

La entrada de direcciones de la RAM es de α bits, donde α es el $\log_2(N)$. El valor de "α" puede ser cualquier número entero mayor que dos, ya que la arquitectura propuesta en este trabajo soporta $N=2^P$ datos de entrada ($P \geq 2$).

La cuarta parte de la primera etapa se presenta en la Figura 3.11, en ella se tienen los datos leídos de la RAM que van a ser las entradas de la mariposa simétrica en el caso de las cuatro primeras salidas de esta Figura 3.11 (*a_dato,b_dato,c_dato,d_dato*) y las entradas de la mariposa básica en el caso de las tres últimas (*x_dato,y_dato,z_dato*). Para indicar en qué momento estas señales están listas para pasar a la etapa mariposa se cuenta con las variables *fcont2* para el *dmux_reg2* y el estado 18 ó 23 para el *dmux_reg3*.

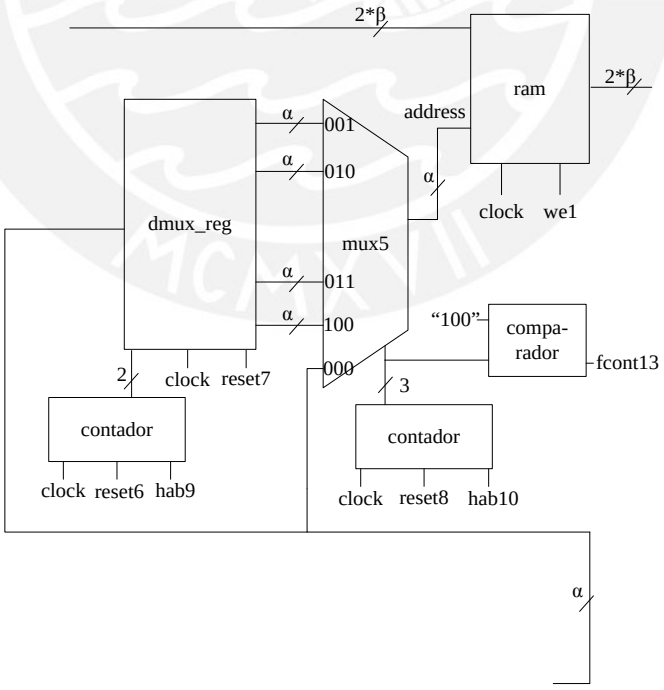


Figura 3.10. Direcciones de escritura de los resultados del proceso mariposa.

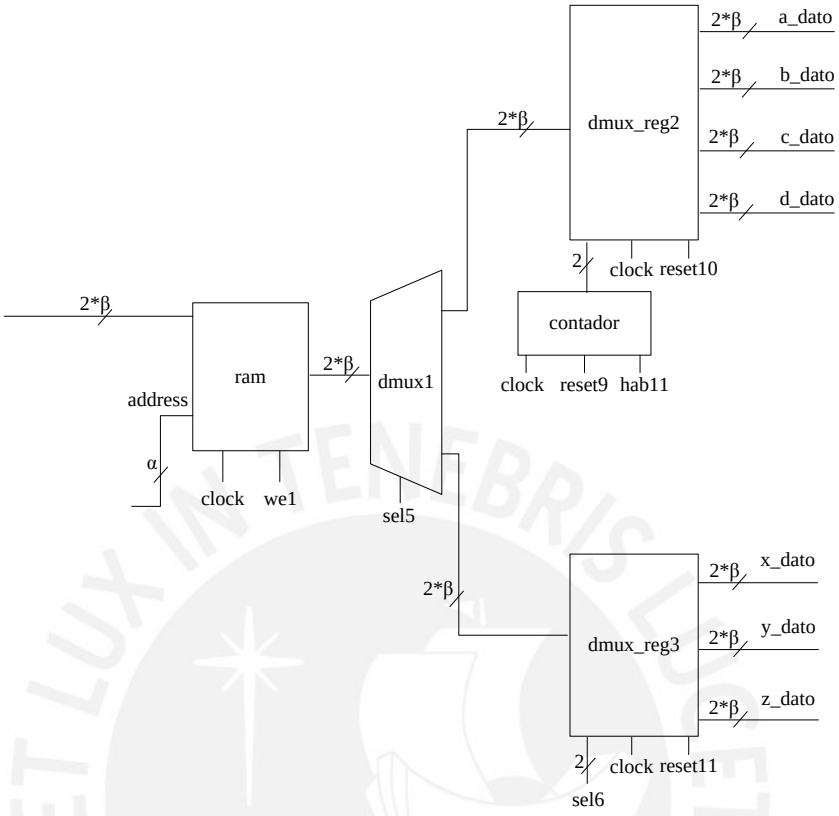


Figura 3.11. Salida de la RAM con datos para la mariposa simétrica y la mariposa básica.

La Figura 3.12 muestra el diagrama de flujo de la máquina de estados de toda esta primera etapa de la arquitectura. En ella, se tiene la señal de *reset_entrada* que lleva al estado *S0* que es un estado de reposo. Si se desea comenzar o continuar con la ejecución de esta máquina de estados, se debe activar *inicio1* con un uno. Luego, conforme se avance en los estados, se escribe en la memoria los datos de entrada iniciales, se leen los valores necesarios para efectuar las mariposas simétricas, se escribe en la RAM los resultados del proceso mariposa, y se leen todos los datos de la RAM para efectuar las mariposas básicas.

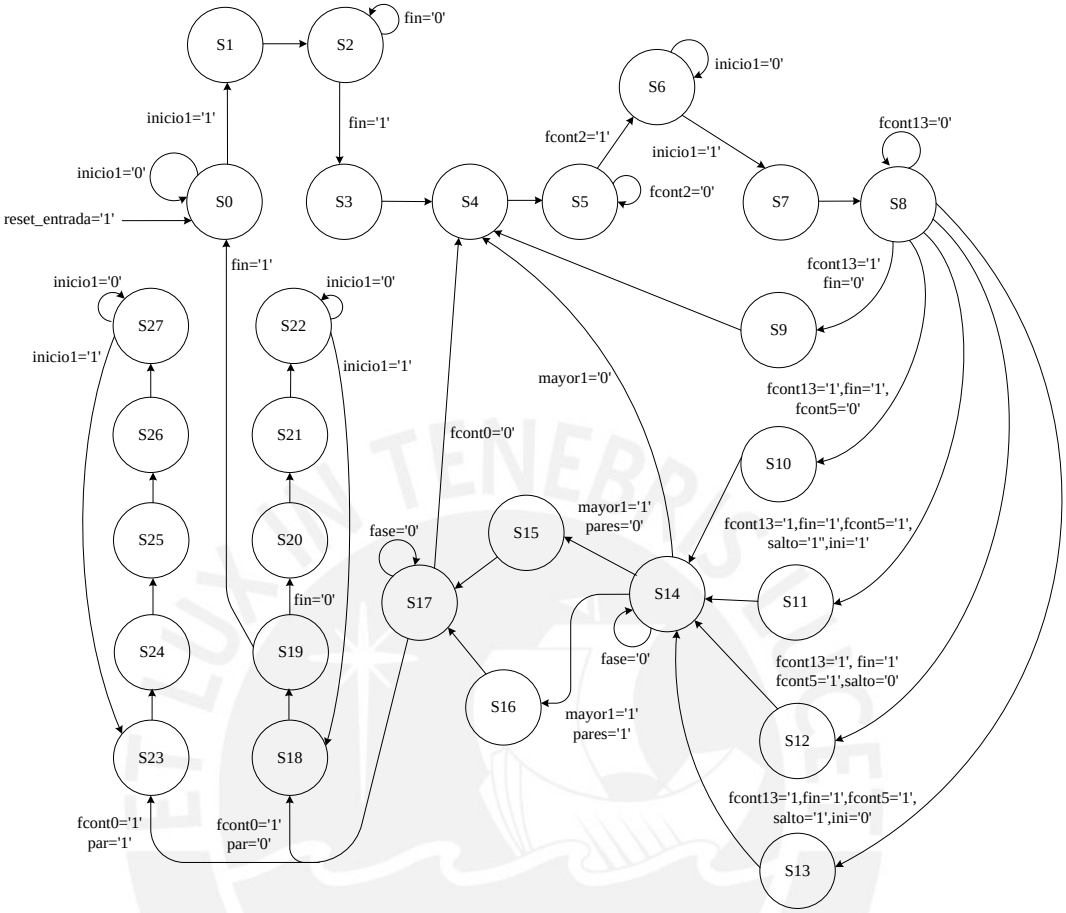


Figura 3.12. Diagrama de flujo de la máquina de estados de la primera etapa de la arquitectura FFT Split-Radix.

La acción de cada estado es la siguiente:

- El estado *S0* no realiza ninguna acción, es un estado de espera.
- El *S1* escribe el primer dato de entrada en la RAM.
- El *S2* escribe los demás datos de entrada en la RAM.
- El *S3* lee el primer dato para la mariposa simétrica.
- El *S4* espera para registrar el segundo dato.
- El *S5* lee los otros tres datos para la mariposa simétrica.
- El *S6* espera los resultados de la mariposa simétrica.
- El *S7* escribe en la RAM la primera respuesta de la mariposa simétrica.
- El *S8* escribe en la RAM los otros tres resultados de la mariposa simétrica.
- El *S9* lee el primer dato de otra mariposa.

- El *S10* ejecuta los dos primeros saltos para las dos primeras fases, con lo cual indica que sólo se realiza un bloque de mariposas en la primera fase y otra en la segunda. Además, se encarga del primer salto para la tercera fase.
- El S11 realiza los primeros saltos para las siguientes fases e inicializa el cuarto valor del *mux2* de la parte de color rojo de la Figura 3.9.
- El *S12* efectúa otros saltos de una misma fase.
- El *S13* realiza los primeros saltos para la fase tres en adelante sin inicializar el cuarto valor del *mux2* de la parte de color rojo de la Figura 3.9.
- El *S14* espera a que se actualice le valor de la variable *mayor*.
- El *S15* brinda la dirección a leer en la RAM del primer valor de una fase par.
- El *S16* brinda la dirección a leer en la RAM del primer valor de una fase impar.
- El *S17* espera un ciclo de reloj para actualizar datos.
- En el *S18* se comienza a leer para la entrada de la mariposa básica cuando el $\log_2 N$ es un número impar.
- El *S19* lee el otro valor para la mariposa básica.
- El *S20* lee el valor que no ingresa a la mariposa básica.
- El *S21* espera a que se registre el último valor.
- El *S22* espera que se efectúe la mariposa básica y se guarde en otra RAM el resultado en forma ordenada.
- En el *S23* se comienza a leer para la entrada de la mariposa básica cuando el $\log_2 N$ es un número par.
- El *S24* se lee el primer valor de la mariposa básica.
- El *S25* lee el segundo valor de la mariposa básica.
- El *S26* espera a que se registre el último valor.
- El *S27* espera que se efectúe la mariposa básica y se guarde en otra RAM el resultado en forma ordenada.

La segunda etapa de la arquitectura FFT Split-Radix se detalla en la Figura 3.13. Este diseño se puede dividir en dos partes para su explicación. La primera (color azul) se encarga de desarrollar la mariposa simétrica y la segunda (color naranja) efectúa la mariposa básica.

En la primera parte, se puede observar que en la entrada se tienen juntos los cuatro valores para realizar la mariposa simétrica. Junto con ellos, también entran en la mariposa los coeficientes trigonométricos denominados factores *Twiddle*, que se encuentran en dos memorias de sólo

lectura: una que almacena las partes reales de los coeficientes y la otra que contiene las partes imaginarias. Estos valores son generados en el software Matlab y copiados en el código en VHDL de las memorias de sólo lectura. Para escoger los dos coeficientes que entra en la mariposa simétrica, se tiene el diseño de la Figura 3.14 que da las direcciones donde se encuentran los coeficientes que se necesitan. Aquí, primero se calcula el número de factores *Twiddle* que se requieren para cada bloque de mariposas que es N/4 para la primera fase y el valor de la fase anterior entre dos para las fases posteriores. Este número se compara con los valores generados por un contador que, para la primera fase, se van a multiplicar por uno para los primeros coeficientes y por tres para los segundos coeficientes. Luego, los factores uno y tres se multiplican por dos para la siguiente fase; y los resultados de estas multiplicaciones se multiplican cada vez por dos para las fases posteriores.

Para obtener las salidas de la mariposa simétrica de la Figura 3.2, se efectúan las operaciones detalladas en la Figura 3.15. En esta última imagen, la multiplicación de (X1-X3) por –j se realiza a través de un cambio, donde el número real del resultado de (X1-X3) pasa a ser la parte imaginaria con signo negativo y el número imaginario de (X1-X3) pasa a ser la parte real del resultado. De la misma forma, la multiplicación de (X1-X3) por j se realiza al tomar la parte real del resultado (X1-X3) como parte imaginaria y la parte imaginaria de (X1-X3) como parte real con signo negativo.

Adicionalmente, la multiplicación compleja de la mariposa simétrica de la SRFFT se consigue a través de una máquina de cinco estados:

- *estado_idle* : espera dato.
- *estado_m_rr* : multiplica real con real.
- *estado_m_ii* : multiplica imaginario con imaginario; y para hallar el número real de la multiplicación compleja, resta el resultado del *estado_m_rr* con el resultado obtenido al multiplicar imaginario con imaginario.
- *estado_m_ri* : multiplica real con imaginario
- *estado_m_ir* : multiplica imaginario con real; y para hallar el número imaginario de la multiplicación compleja, suma el resultado del *estado_m_ri* con la multiplicación imaginario con real.

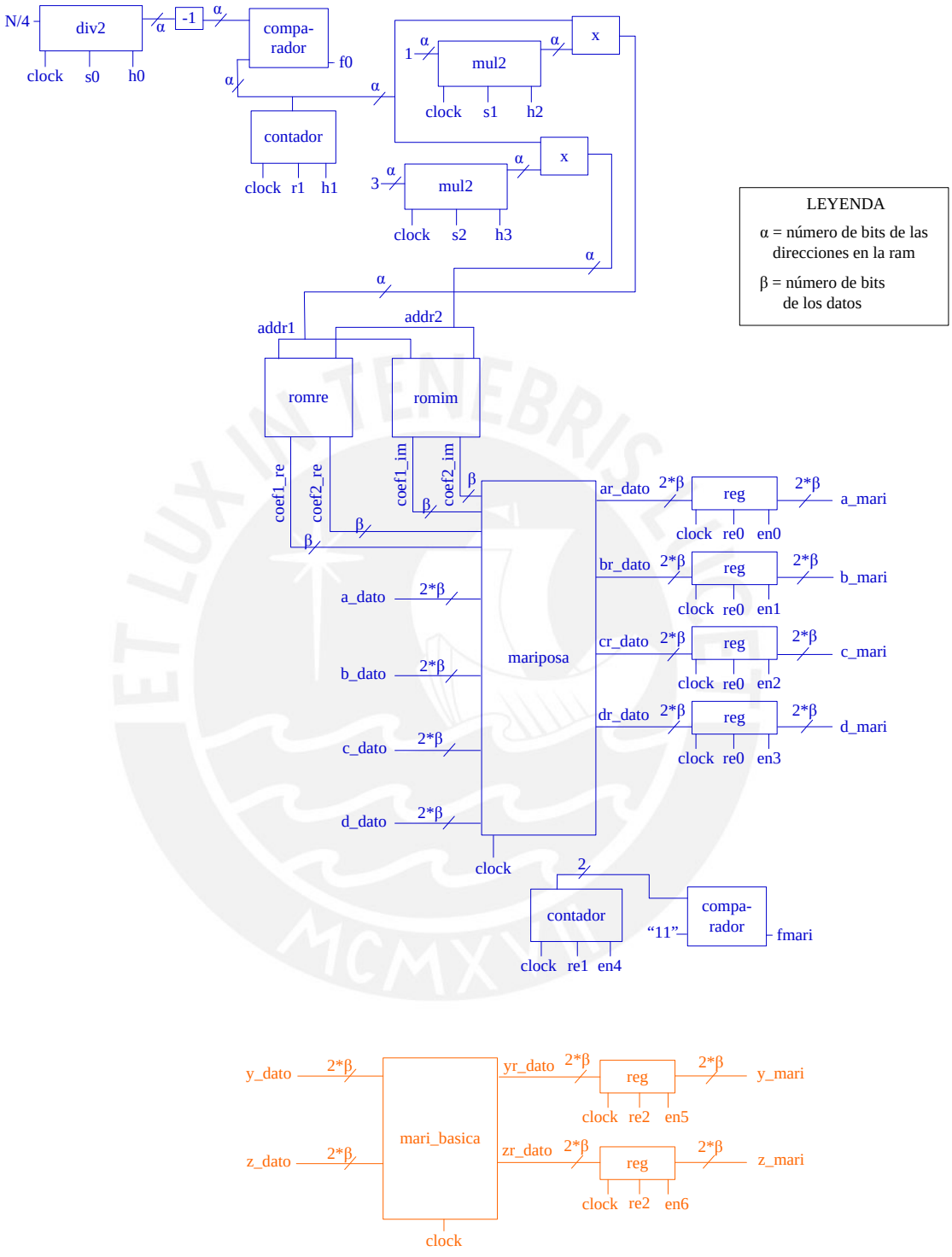


Figura 3.13. Diseño de la segunda etapa de la arquitectura global SRFFT.

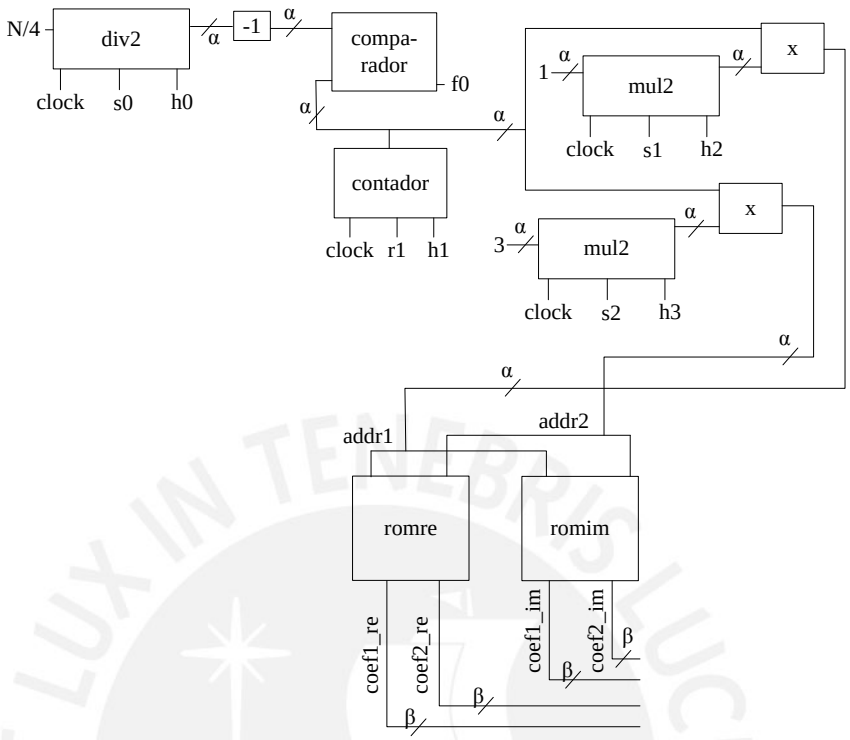


Figura 3.14. Diseño que lee los coeficientes trigonométricos.

La segunda parte de la arquitectura de la *split_mariposa* mostrada en la Figura 3.13, es más sencilla y realiza la mariposa básica que contiene las operaciones de la Figura 1.3.

La máquina de estados de la *split_mariposa*, que se encuentra en el controlador global de la SRFFT, tiene la misma función que la de la *split_entrada*: recibe señales de control y, de acuerdo a ellas, genera las entradas de reinicio y habilitadores para los componentes que maneja.

La secuencia de la máquina de estados de la *split_mariposa* se puede observar en la Figura 3.16, la cual consta de los siguientes estados:

- *e0* es un estado de espera.
- En *e1* se tienen los cuatro primeros valores de entrada para la mariposa simétrica junto con los dos primeros coeficientes trigonométricos.
- En *e2* espera a que termine de desarrollarse la mariposa simétrica.
- En *e3* se tienen los cuatro resultados de la mariposa simétrica.
- En *e4* se espera a que se escriban estos resultados en la RAM y se lean las próximas cuatro entradas de la mariposa simétrica.

- En *e5* se tienen los siguientes cuatro valores para la mariposa simétrica junto con los siguientes coeficientes trigonométricos.
- En *e6* espera a que termine de desarrollarse la mariposa simétrica.
- En *e7* se tienen los cuatro valores de la primera mariposa simétrica de la siguiente fase junto con sus respectivos coeficientes trigonométricos.
- En *e8* se realiza la mariposa básica.

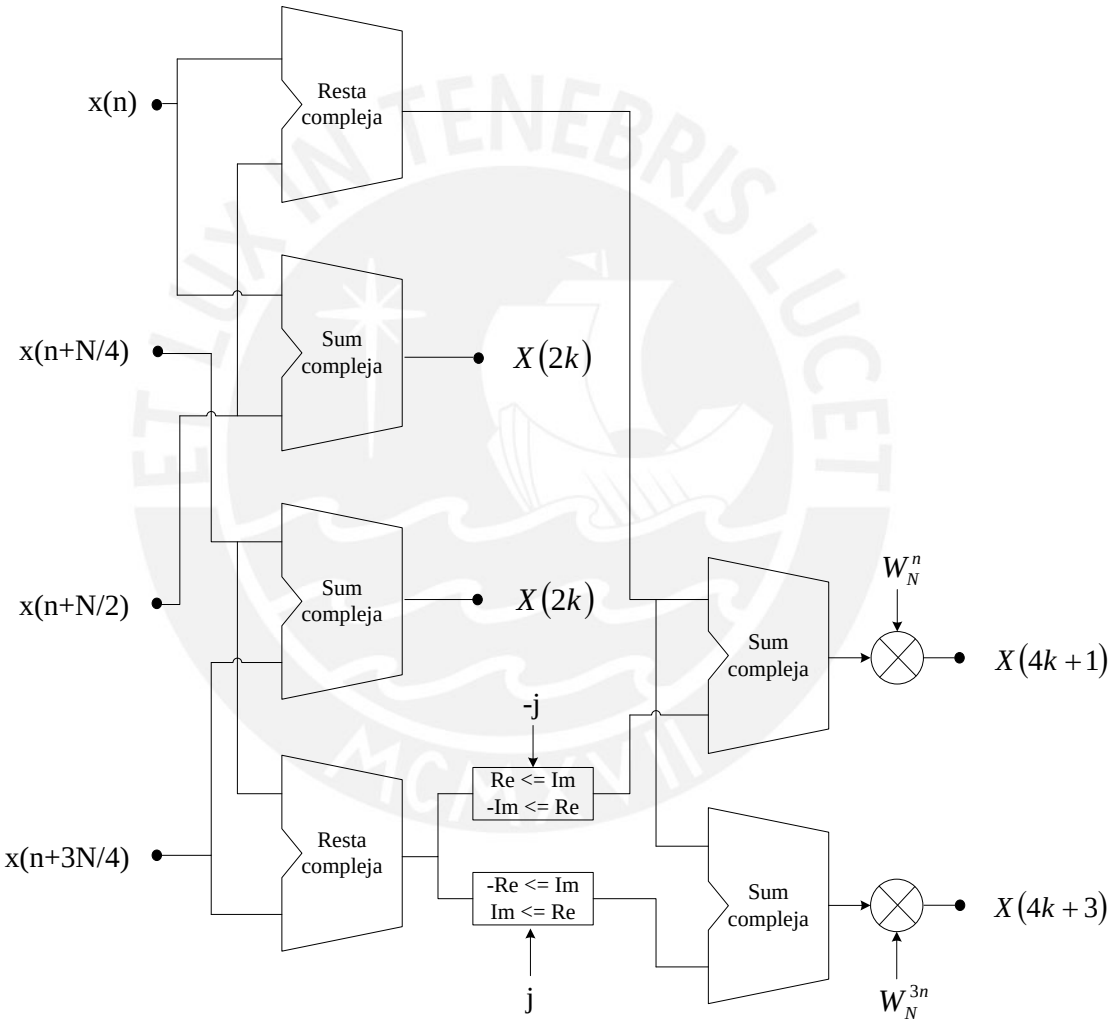


Figura 3.15. Operaciones en la mariposa simétrica de la FFT Split-Radix.

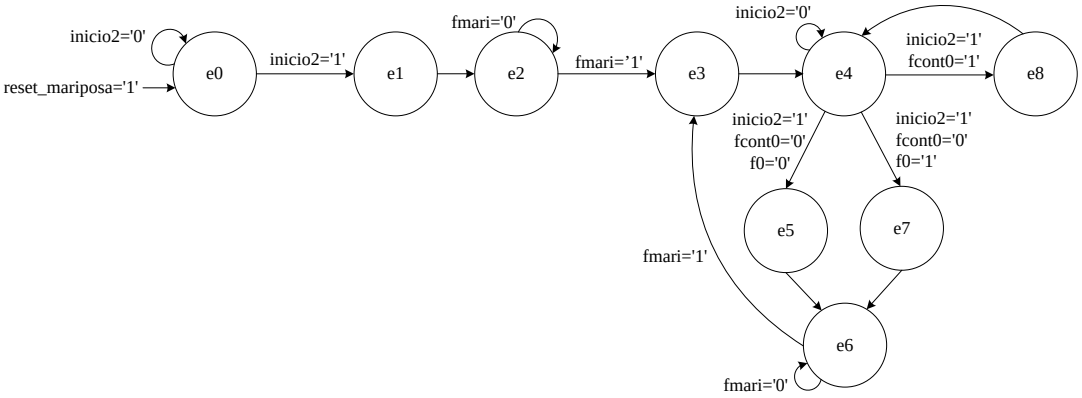


Figura 3.16. Secuencia de la máquina de estados *split_mariposa*.

Después de la etapa *split_mariposa*, sigue la etapa *split_orden* (Figura 3.17) que se encarga de reordenar las salidas que se obtienen de las señales *x_dato*,*y_mari* y *z_mari* de la Figura 3.11 y de la Figura 3.13. Esta etapa consta de una RAM de N entradas de 2*β bits con componentes dedicados para generar las direcciones de escritura y lectura.

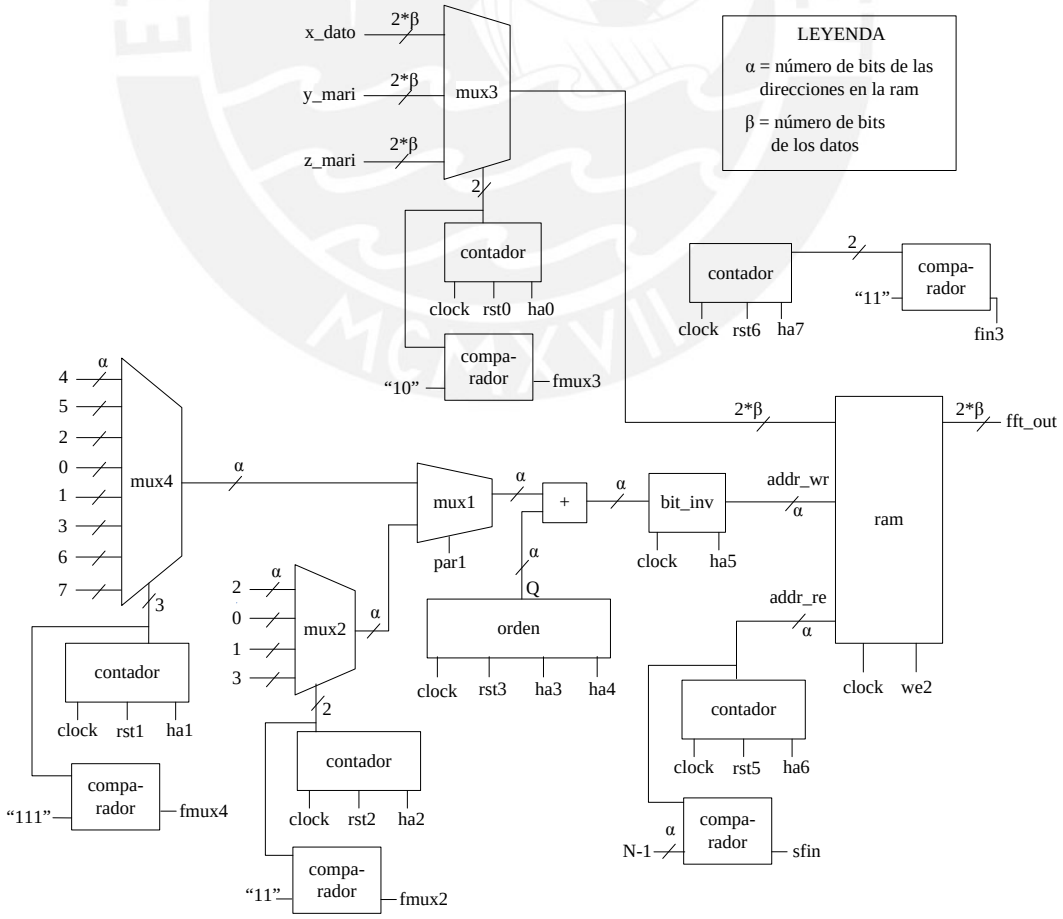
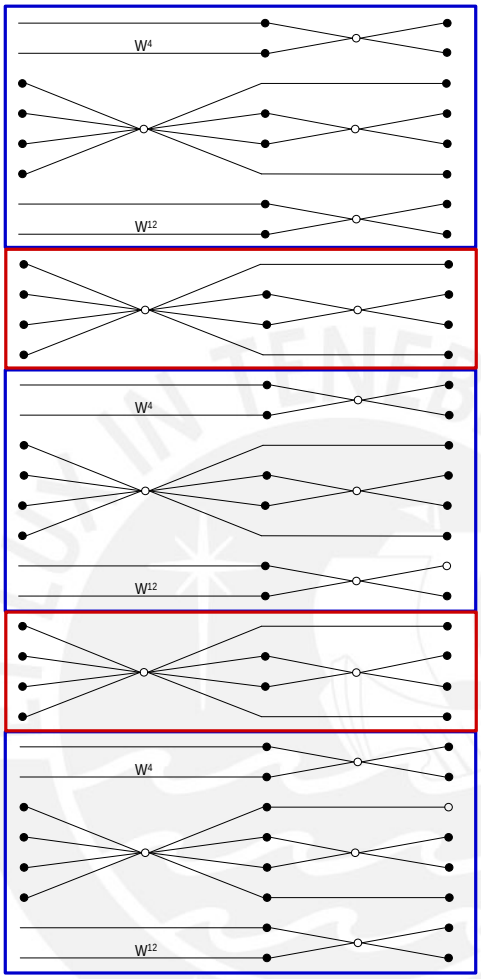


Figura 3.17. Diseño de la tercera etapa de la arquitectura global SRFFT.

Para explicar el reordenamiento que lleva a cabo esta etapa *split_orden*, se utilizará la Figura 3.18.



4	20	5
5	21	21
2	18	9
0	16	1
1	17	17
3	19	25
6	22	13
7	23	29
2	10	10
0	8	2
1	9	18
3	11	26
4	4	4
5	5	20
2	2	8
0	0	0
1	1	16
3	3	24
6	6	12
7	7	28
2	14	14
0	12	6
1	13	22
3	15	30
4	21	7
5	29	23
2	26	11
0	24	3
1	25	19
3	27	27
6	30	15
7	31	31

Figura 3.18. Bloques ocho y cuatro para el reordenamiento en la etapa *split_orden*.

En esta figura se tienen las dos últimas fases de la SRFFT simétrica para N=32, donde las salidas están agrupadas en bloques de ocho (color azul) y en bloques de cuatro (color rojo). Las salidas de los bloques de ocho tienen un orden específico, que está representado por las entradas del *mux4* de la Figura 3.17 (números de color azul en la Figura 3.18), y las salidas de los bloques de cuatro también tienen un orden representado por las entradas del *mux2* en la misma figura (números de color rojo en la Figura 3.18). Para el primer bloque de la última fase de una SRFFT de $\log_2 N$ impar que es de ocho elementos, se le suma el valor de N-16 a las entradas del *mux4* o números de color azul en la Figura 3.18. Luego, le sigue un bloque de cuatro que se le suma el número anteriormente sumado menos ocho unidades a las entradas del *mux2* o números de color rojo en la Figura 3.18. Para el siguiente bloque que es uno de ocho, se le suma el número anterior menos 16; y así sucesivamente hasta que N sea menor o igual que el número

que se le desea restar. Cuando se cumple esta condición, el orden adquiere el valor cero y a los siguientes valores de los multiplexores *mux4* y *mux2* se les suma 12 unidades más en cada bloque hasta que se terminen de escribir todos los datos en la RAM. Lo mismo se aplica para una SRFFT de $\log_2 N$ par, con la diferencia que empieza con un bloque de cuatro, al cual se le suma N-8 a las entradas del *mux2*. Estas adiciones y sustracciones las realiza el bloque orden, cuyo diseño se encuentra en la Figura 3.19, con lo cual los datos de salida estarían en el orden indicado por la columna central de números de la Figura 3.18. Para obtener el orden correcto que es el de la columna derecha de la Figura 3.18, se debe realizar la inversión de bits de la dirección obtenida del bloque orden.

Esta tercera etapa de la arquitectura global SRFFT, también cuenta con su respectiva máquina de estados (Figura 3.20) dentro del controlador global. Esta máquina de estados se encarga de controlar la *split_orden* para escribir los resultados que provienen de la etapa *split_mariposa* en el orden correcto dentro de la RAM para luego leerlos en orden desde 0 hasta N; y así, obtener las salidas que representan la señal de entrada en el dominio de la frecuencia.

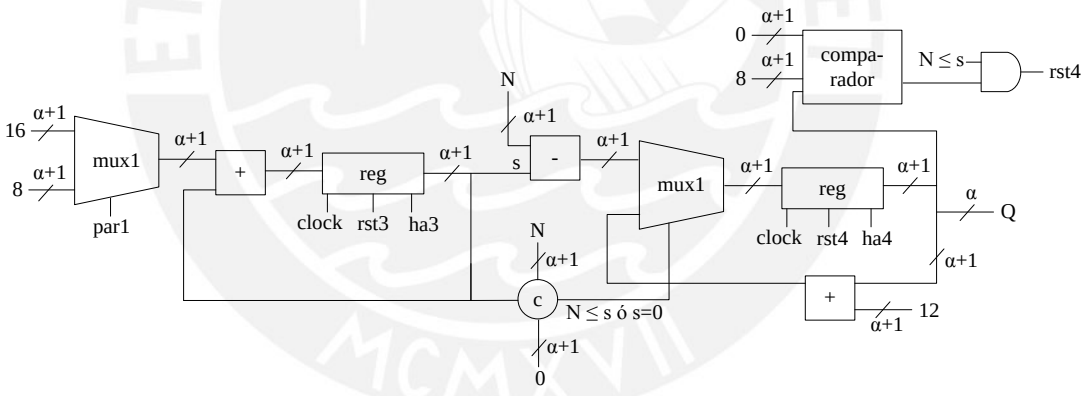


Figura 3.19. Diseño del bloque orden.

Los estados de la *split_orden* se dividen en tres partes: la primera comprende desde el estado *s1* al estado *s14*, la segunda va desde el estado *s15* al *s26*, y la última contiene a los estados *s27* y *s28*. Los estados de la primera parte se ejecutan cuando el $\log_2 N$ de la SRFFT es impar, la segunda parte corresponde cuando el $\log_2 N$ de la SRFFT es par, y la tercera se encarga de colocar en la salida de toda la SRFFT cada uno de sus resultados en forma serial.

Los estados de la *split_orden* son los que se describen a continuación:

- El estado *s0* es un estado de reposo.

- El *s1* obtiene la dirección en la RAM para el primer dato que viene de la *split_mariposa* y que es el primero de un bloque de ocho.
- El *s2* escribe en la RAM el dato que viene de la *split_mariposa* y que es el primero de un bloque de ocho.
- El *s3* espera a que culmine la escritura en la RAM.
- El *s4* escribe en la RAM los datos que vienen de la *split_mariposa* y que pertenecen al bloque de ocho ya comenzado.
- El *s5* escribe el dato que viene de *split_entrada* para el bloque de ocho.
- El *s6* espera hasta obtener otros nuevos tres datos para escribir.
- El *s7* obtiene la dirección en la RAM para el dato que viene de *split_entrada* y que empieza el bloque de cuatro.
- El *s8* espera la dirección.
- El *s9* espera la dirección.
- El *s10* escribe el dato que viene de *split_entrada* y que empieza el bloque de cuatro.
- El *s11* obtiene la dirección en la RAM para el primer dato que viene de la *split_mariposa* y que pertenece al bloque de cuatro ya comenzado.
- El *s12* espera la dirección.
- El *s13* escribe el segundo dato que viene de la *split_mariposa* para el bloque de cuatro.
- El *s14* escribe el último dato de un bloque de cuatro, el cual es proviene de *split_entrada*.
- El *s15* obtiene la dirección en la RAM para el primer dato obtenido de *split_entrada* que empieza el bloque de cuatro.
- El *s16* escribe el dato de *split_entrada* que comienza el bloque de cuatro en la RAM.
- El *s17* obtiene la dirección para el primer dato que vienen de *split_mariposa* para el bloque de cuatro.
- El *s18* escribe el segundo dato que proviene de *split_mariposa* para el bloque de cuatro.
- El *s19* espera hasta obtener otros nuevos tres datos para escribir.
- El *s20* escribe el último dato para un bloque de cuatro que proviene de *split_entrada*.
- El *s21* obtiene la dirección en la RAM para el primer dato que viene de la *split_mariposa* y que inicia un bloque de ocho.
- El *s22* espera dirección.
- El *s23* espera dirección.
- El *s24* escribe los datos que provienen de *split_mariposa* y que pertenecen al bloque de ocho.

- El *s25* escribe el dato que viene de *split_entrada* para un bloque de ocho ya comenzado.
- El *s26* espera.
- El *s27* lee el primer resultado de toda la SRFFT.
- El *s28* lee los demás resultados de toda la SRFFT.

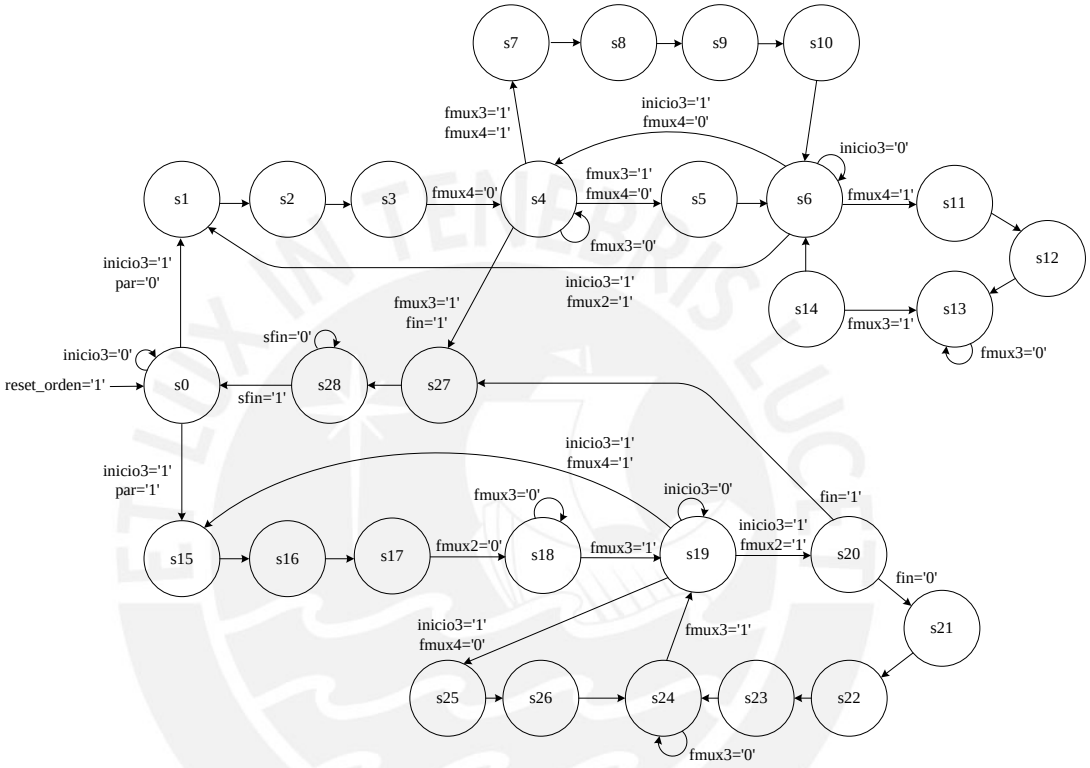


Figura 3.20. Secuencia de la máquina de estados *split_orden*.

Finalmente, se tiene la *maq_split* (Figura 3.21) para controlar el inicio, la espera y el fin de cada máquina de estados antes expuesta. Los estados de la *maq_split* ejecutan las siguientes acciones:

- *E0* marca el estado en reposo de todas las máquinas.
- *E1* inicia la *maq_entrada*.
- *E2* coloca la señal de inicio de la *maq_entrada* en cero para que en el momento indicado se quede en el estado de espera y empiece la *maq_mariposa*.
- *E3* inicia la *maq_mariposa*.
- *E4* coloca la señal de inicio de la *maq_mariposa* en cero para que en el momento indicado se quede en el estado de espera y empiece la *maq_orden*.
- *E5* inicia la *maq_orden*.

- *E6* coloca la señal de inicio de la *maq_orden* en cero para que en el momento indicado se quede en el estado de espera y empiece la *maq_entrada* o para esperar a que termine de ejecutarse toda la SRFFT.

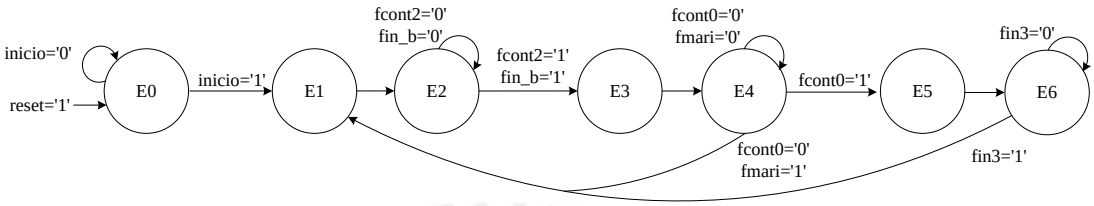


Figura 3.21. Secuencia de la máquina de estados *maq_split*.

3.2.2. Diseño en VHDL de la Arquitectura de la FFT Split-Radix.

Para el diseño de la arquitectura propuesta en el lenguaje de descripción de hardware VHDL, se siguió la metodología sugerida en el libro “Circuit Design with VHDL” [14]. La Figura 3.22 esquematiza esta metodología mediante un diagrama de flujo, donde primero se escribe el código en VHDL para luego efectuar la síntesis. En esta última, se ejecuta la compilación que se encarga de convertir lo escrito en VHDL, que describe el circuito deseado en un nivel de transferencia de registro (RTL, Register Transfer Level), a lista de conexiones (netlist) que pertenece al nivel de compuertas. Después, se procede a la optimización de la lista de conexiones dentro del nivel de compuertas para finalmente poder simular el diseño.

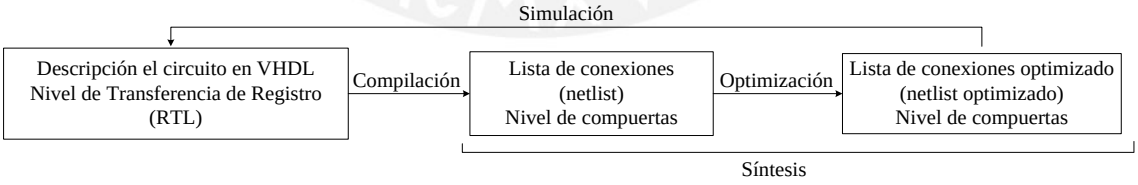


Figura 3.22. Diagrama de flujo del diseño en VHDL.

Todo el diseño se desarrolló en base a una descripción estructural de los componentes que integran el circuito completo, donde cada componente es descrito individualmente y utilizado según sea requerido en otras partes del diseño. Para ello, se empleó un único paquete de componentes para evitar confusión al momento de solicitarlos.

Además, el diseño de los circuitos es genérico, modular y portable. Genérico ya que los parámetros son referidos desde la sentencia GENERIC dentro de la entidad del código, los cuales pueden tomar diferentes valores. Modular porque el diseño del sistema se compone de módulos o bloques funcionales más pequeños. Portable porque todos los componentes del sistema son descritos en VHDL, incluso algunas las multiplicaciones y divisiones, que son obtenidas al aumentar o disminuir ceros a uno de los factores o al dividendo; ya que son operaciones donde el otro factor o el divisor son potencias de dos. La excepción se encontraría en las operaciones aritméticas de suma, resta y multiplicación que son obtenidas de la librería de Altera. Sin embargo, esto no afecta la portabilidad, ya que la mayoría de FPGAs cuenta con estas operaciones que hace que el diseño sea adaptable a otros dispositivos.

La mayoría de los circuitos presentan entradas de habilitación y reinicio. Esto permite que sean fácilmente inicializados al activar las entradas de reinicio de los componentes del circuito. En cuanto a los habilitadores, estos permiten un mejor control en el comportamiento y funcionamiento de los bloques constituyentes del sistema y también del sistema global.

CAPÍTULO 4
RESULTADOS Y VERIFICACIÓN DEL DISEÑO DE LA ARQUITECTURA

Este último capítulo comprende los resultados obtenidos y la validación del comportamiento del diseño realizado. Como se explica en el capítulo anterior, la arquitectura diseñada es descrita en el software QUARTUS II para el FPGA Cyclone II EP2C35F672C6 de ALTERA con el lenguaje VHDL.

El software Quartus II se encarga de sintetizar el código en VHDL de la SRFFT para 1024 entradas de 16 bits para obtener los datos sobre el uso de recursos del FPGA que se muestran en la Figura 4.1.

Flow Status	Successful - Thu Aug 06 19:06:47 2009
Quartus II Version	8.0 Build 231 07/10/2008 SP 1 SJ Web Edition
Revision Name	split_radix
Top-level Entity Name	split_radix
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Met timing requirements	Yes
Total logic elements	3,027 / 33,216 (9 %)
Total combinational functions	2,882 / 33,216 (9 %)
Dedicated logic registers	945 / 33,216 (3 %)
Total registers	945
Total pins	51 / 475 (11 %)
Total virtual pins	0
Total memory bits	65,536 / 483,840 (14 %)
Embedded Multiplier 9-bit elements	10 / 70 (14 %)
Total PLLs	0 / 4 (0 %)

Figura 4.1. Información y recursos utilizados en el FPGA.

Con estos resultados, se puede apreciar que se ha utilizado un total de 2882 Elementos Lógicos (EL), que representa el 9 % de toda la cantidad de EL dentro del FPGA. Además, se emplea un total de 945 Registros Lógicos Dedicados, lo cual viene a ser el 3% de uso de Registros.

En resumen, el resultado de la síntesis arroja un uso de 65536 bits de memoria (14% de recursos utilizados) y al evaluar los resultados del análisis *Timing* del Quartus II, el cual considera retardos de las señales dentro del FPGA, se pudo encontrar que la frecuencia máxima de funcionamiento del sistema es 100MHz. Del mismo modo, se pudo comprobar que la latencia

máxima del sistema, evaluada una vez ya culminada la etapa de toma de datos, es de 448.805us. Adicionalmente, la potencia máxima disipada es de 116.80mW.

Posteriormente, se hicieron las mismas pruebas pero para 256, 32 y 16 entradas de 16 bits, las cuales se pueden comparar con otros trabajos de hace cuatro años en adelante como se puede apreciar en la Tabla 4.1. Las dos columnas de color celeste de esta tabla muestran los resultados de esta tesis con 1024 datos de entrada y los de C. Chao, Z. Qin, X. Yingke y H. Chengde en el trabajo que presentaron para la IEEE en el año 2005 titulado “Design of a High performance FFT processor based on FPGA” [15]. En estas columnas como en las siguientes se puede ver que el diseño propuesto en esta tesis tiene un tiempo de ejecución muy alto debido a que todo el procesamiento se hace de manera secuencial, mientras que los demás se desarrollan de forma paralela. Sin embargo, como se nota en toda esta tabla, los recursos y la potencia utilizada por la arquitectura de esta tesis es siempre mucho menor.

La diferencia en número de recursos se puede apreciar en la comparación de las columnas de color lila de la Tabla 4.1, donde la configuración inicial es para 256 entradas de 16 bits. Aquí, se tienen los diseños de Altera en “FFT MegaCore function user guide” del año 2006 [16]; de Xilinx en “Xilinx LogiCore Fast Fourier Transform” del año 2007 [17]; y del trabajo de Jesús García, “FPGA Realization of a Split Radix FFT Processor” [18], presentado para el VLSI Circuits and Systems III en el año 2007. En estos tres, el número de funciones combinacionales es considerablemente mayor, así como el número de registros lógicos.

En las columnas verdes también se puede ver esta diferencia de recursos con la tesis de Pedro Luis Castro titulada “Desarrollo de un Módulo Digital para el Análisis Espectral de Señales de Audio” [19] de la Universidad Politécnica de Cataluña. Además, se puede observar un ahorro en el consumo de potencia del diseño propuesto en esta tesis, al compararla con el trabajo presentado por Sarkar y Kumar en “Fast 16-point FFT Core for Virtex-II FPGA” [20] de la Universidad de California, San Diego.

Así, de modo comparativo se muestra que el diseño ha sido enfocado básicamente en disminuir el consumo de potencia; pues cuanto menor sea la cantidad de recursos utilizados, menor será la potencia consumida por el dispositivo.

Tabla 4.1. Tabla comparativa de los resultados de la presente tesis con varios trabajos de hace cuatro años en adelante.

Procesador	[14]	Diseño propuesto	Altera FFT Core [15]	Xilinx FFT Core [16]	Diseño de García [17]	Diseño propuesto	Diseño propuesto	Diseño de Castro [18]	Diseño de Sarkar y Kumar [19]	Diseño propuesto
Algoritmo	SRFFT	SRFFT	SRFFT	SRFFT	SRFFT	SRFFT	SRFFT	RADIX-4	RADIX-4	SRFFT
Número de datos de entrada	1024	1024	256	256	256	256	32	16	16	16
Número de bits de las entradas	16	16	16	16	16	16	16	16	16	16
Funciones combinacionales	-	2882	2144	2027	6702	1898	1442	4469	1037	1438
Registros lógicos	-	945	3758	-	6498	910	857	2396	1432	842
Multiplicadores de 18x18	-	5	12	30	48	4	4	16	-	3
Memoria	-	64(1K)	19(9K)	3(36K)	5(4K)	16(1K)	2(1K)	-	-	1K
Tiempo de ejecución (us)	10.1	448.805	0.69	0.59	0.123	92.885	7.995	0.006744	0.0944	3.605
Frecuencia (MHz)	127	100	370	432	350	100	100	68.738	169.52	100
Potencia (mW)	-	116.80	-	-	-	116.65	116.76	-	440	116.70
Familia de FPGA	Virtex II Pro 30	Cyclone II	Stratix III	Virtex 5	Stratix II	Cyclone II	Cyclone II	Virtex II	Virtex II	Cyclone II
Dispositivo	XCV2P30	EP2C35F672C6	EP3SL70F484C2	5VSX35T	EP2S15F672C3	EP2C35F672C6	EP2C35F672C6	v300efg456-6	XC2V2000	EP2C35F672C6

Cabe resaltar que además de los beneficios mencionados en comparación con los otros trabajos, la arquitectura de esta tesis tiene la capacidad de aceptar $N=2^P$ datos de entrada ($P \geq 2$) , de diversos números de bits.

Posterior a la descripción y compilación del código, se simulan todos los bloques que conforman el sistema creado para corroborar su correcto funcionamiento frente a las diversas entradas que se puedan generar. Después, estos bloques o módulos se van asociando a entidades mayores que son también simulados para verificar la obtención de resultados adecuados. Este proceso continúa hasta formar el sistema completo que es simulado al final con la herramienta ModelSim Altera Starter Edition 6.4a.

Este software es de mucha utilidad, ya que con ella se pueden ingresar en la arquitectura señales de entrada generadas en Matlab. Para las pruebas realizadas en este trabajo, se emplea una señal sinusoidal de dos tonos. (Ver Figuras 4.2 y 4.3)

La señal de dos tonos se genera con el siguiente código:

```
N=1024;
fm1=3000;  % frecuencia de la señal
fm2=4000;
fs=20000;  % frecuencia de muestreo
T=1/fs;
s=0:N-1;
t=s*T;
x=(sin(2*pi*fm1/fs*s)+ sin(2*pi*fm2/fs*s));
```

Señal de dos tonos en Matlab

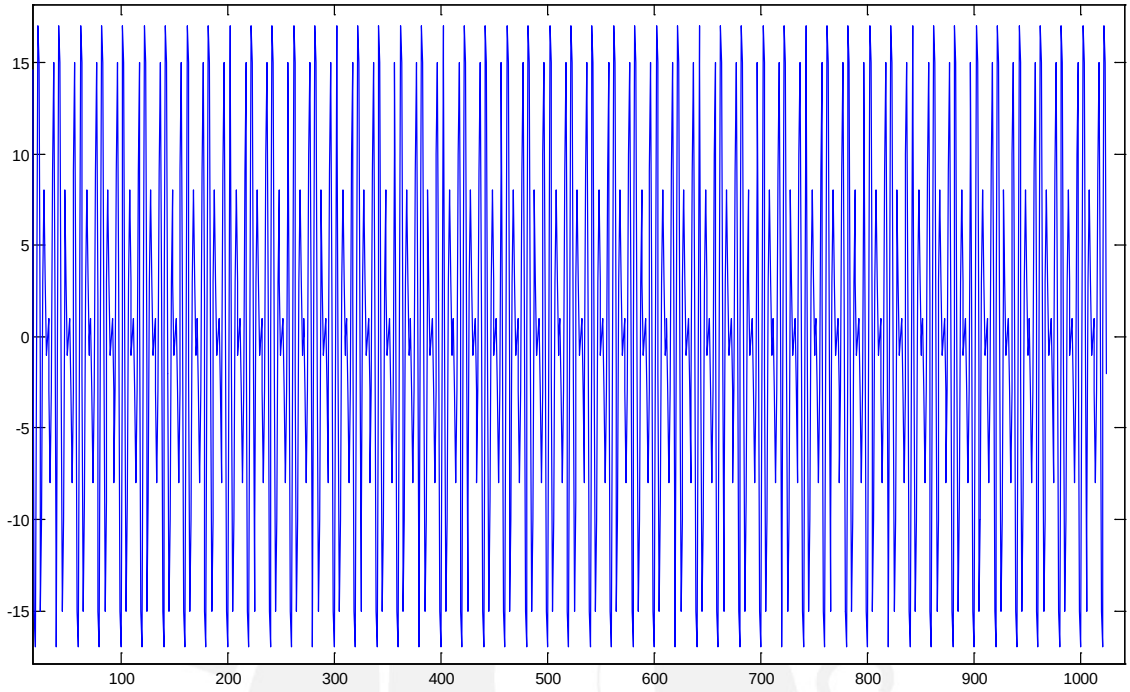


Figura 4.2. Señal de dos tonos en Matlab.

Ampliación de la señal de dos tonos en Matlab

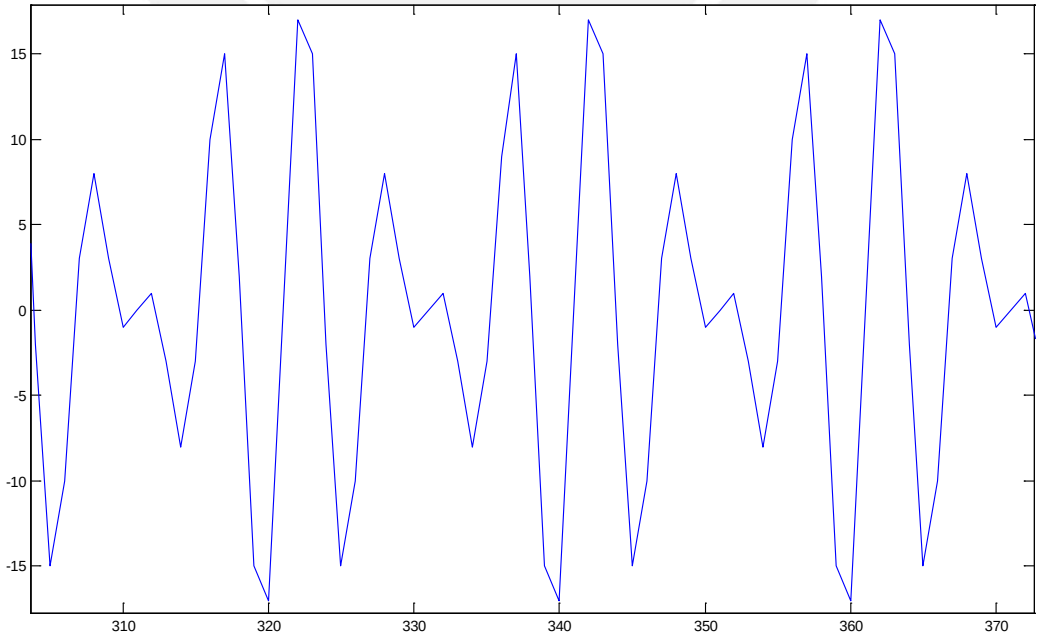


Figura 4.3. Aumento de la señal de la Figura 4.2.

Antes de mostrar las simulaciones, se tienen unas pruebas realizadas en Matlab para conseguir la FFT y la SRFFT para la señal sinusoidal de 1024 datos. Por ejemplo, en la Figura 4.4, se muestra la FFT de la señal de dos tonos dada por Matlab. Con este software, se desarrolla también la SRFFT (Figura 4.5) que da resultados similares a los de la Figura 4.4. La diferencia entre la Figura 4.4 y 4.5 se muestra en la Figura 4.6, donde dicha diferencia es de 4.0274×10^{-10} según Norma L2

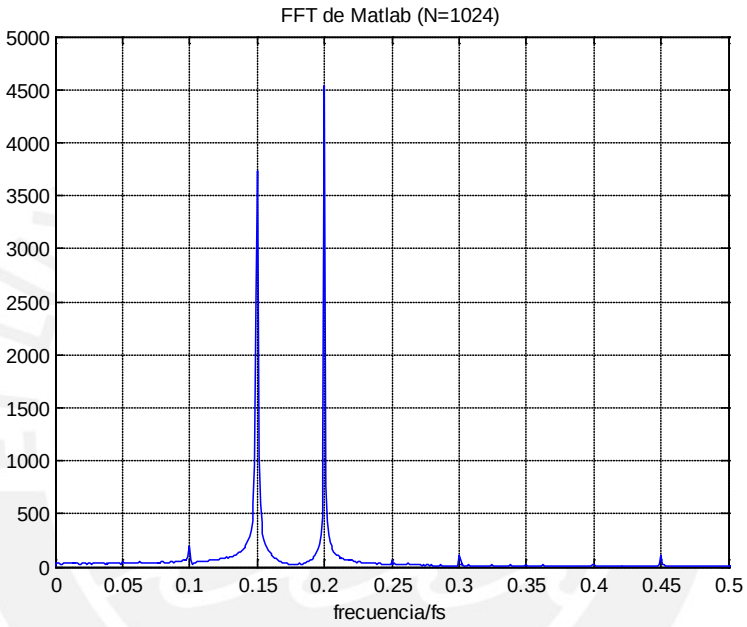


Figura 4.4. FFT de la señal de dos tonos generada en Matlab.

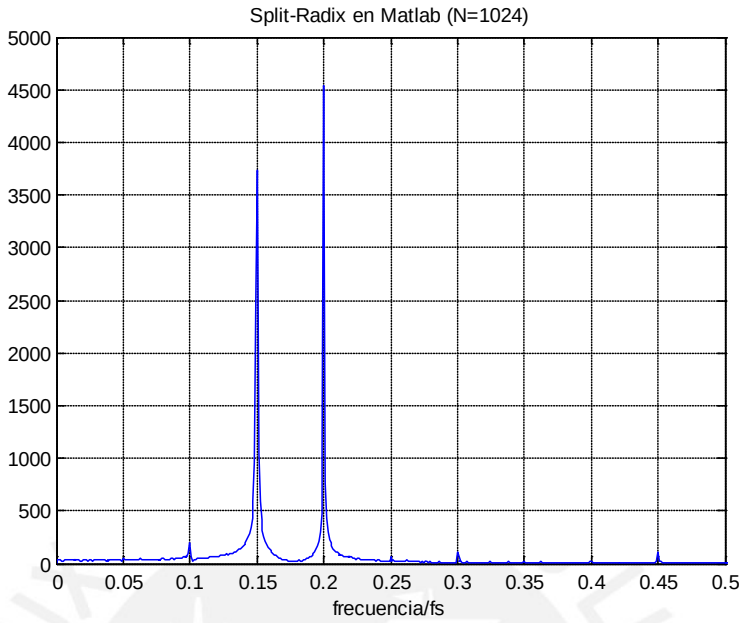


Figura 4.5. SRFFT de la señal de dos tonos generada en Matlab.

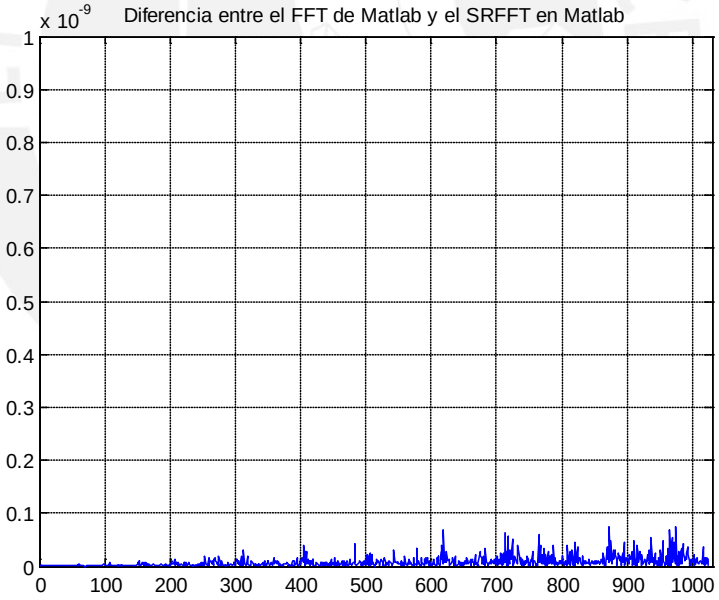


Figura 4.6. Diferencia entre el FFT de Matlab y el SRFFT en el mismo software.

En las simulaciones de las Figuras 4.7, 4.8 y 4.9, se utiliza la entrada mostrada de 1024 puntos; donde se grafican sus valores en la primera fila, el *clock* con un periodo de 10 ns; en la segunda fila se ubica la señal de inicialización; en la tercera, se encuentra el habilitador de toda la SRFFT; en la cuarta, está la señal de entrada; en la quinta, se presenta la salida de toda la SRFFT; luego, vienen las señales de las 4 máquinas de estado: entrada, mariposa, orden, Split-

Radix; después, está la cuenta para los datos que se escriben y leen de la RAM; en la siguiente, aparece la bandera que indica cuando la cuenta llega a su fin; luego vienen las seis señales que entran a la mariposa simétrica (4 entradas) y a la mariposa básica (2 entradas); y, finalmente, las señales que salen de las mismas.

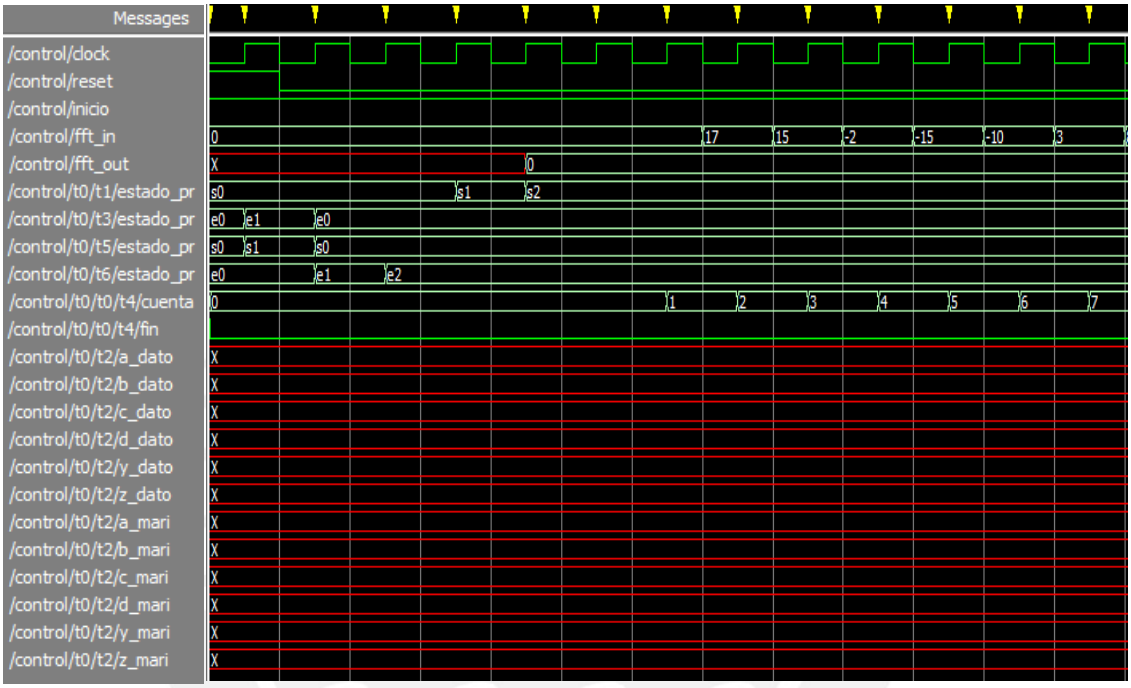


Figura 4.7. Simulación de entrada de datos.

En la Figura 4.7 se presenta la simulación de la entrada de los 10 primeros datos de los 1024 que se registran. En ella, se ve que hay un pulso de inicio (señal "*reset*") con el habilitador en '1' para que empiece a procesarse la SRFFT. Luego de cinco ciclos de reloj empiezan a escribirse los datos en la RAM, la cual se va a detener cuando la cuenta llegue a 1023.

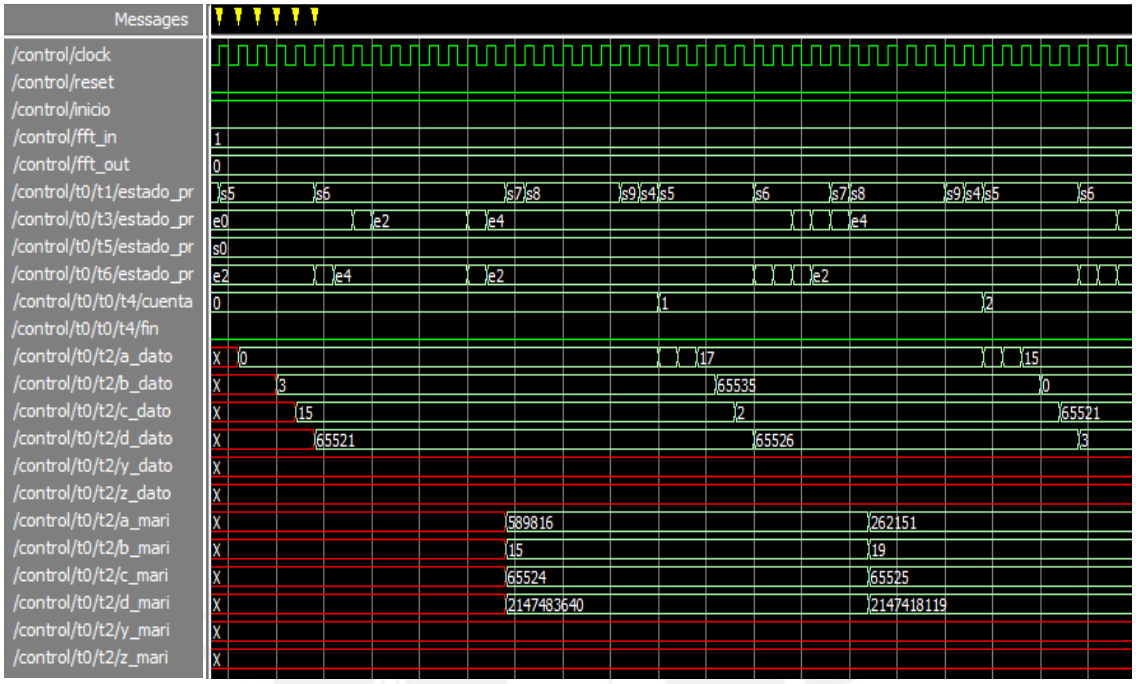


Figura 4.8. Simulación del proceso mariposa simétrica.

Después de escribirse todos los datos de entrada, se procede a leer la RAM de cuatro en cuatro para que se efectúe la mariposa simétrica como se observa en la Figura 4.8.

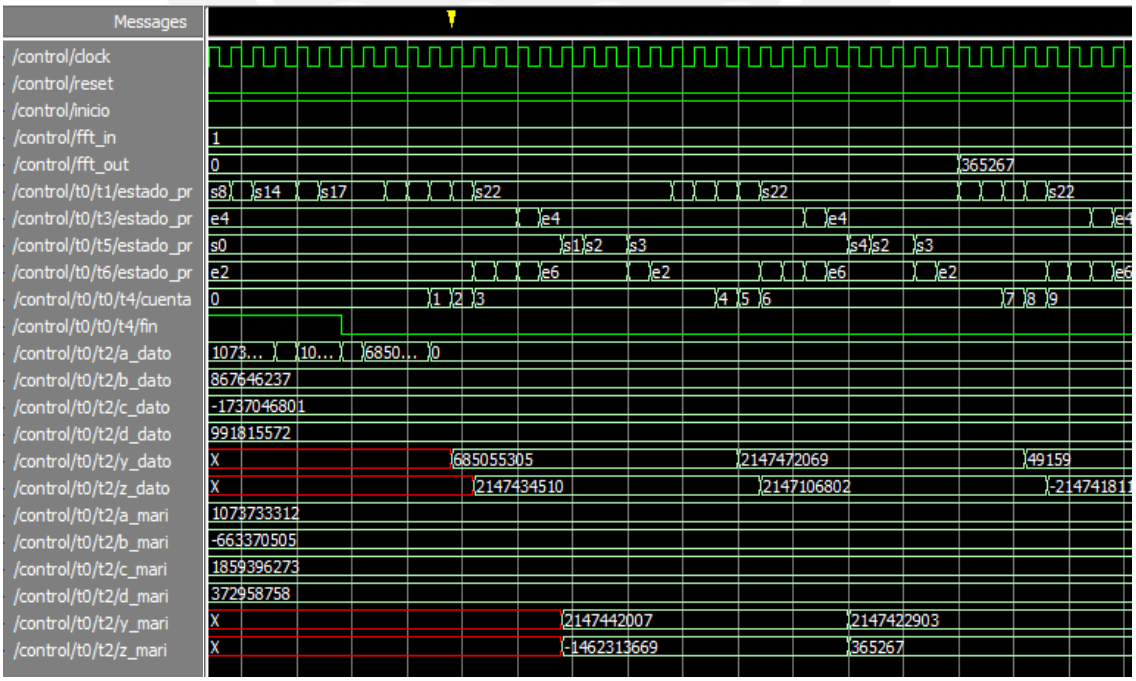


Figura 4.9. Simulación del proceso mariposa básica.

Luego de operar todas las mariposas simétricas se ejecutan las mariposas básicas como se aprecia en la Figura 4.9. A la vez que se resuelven estas mariposas, se reordenan las salidas para obtenerlas en el orden correcto al final del procesamiento de la SRFFT. Como se ve en la fila *fft_out*, ya se están registrando los últimos resultados.

Adicionalmente, en las simulaciones, se puede apreciar el correcto cambio de estados entre las tres etapas de la arquitectura global con la máquina de estados global como se detalló en el capítulo anterior.

Como se muestra en la Tabla 4.1, el diseño de la presente tesis se Para comparar la respuesta del Split-Radix codificado en Matlab con la del Split-Radix de la arquitectura desarrollada en esta tesis, se procede a escalar los resultados del presente diseño. Para ello, los valores impares que se han obtenido se multiplican por dos y los valores pares se multiplican por cuatro. Este escalamiento se debe a que el presente trabajo produce algunas distorsiones y cambios de magnitudes causadas por la aproximación de los valores de entrada, el escalamiento de los coeficientes, el truncamiento en multiplicaciones y las acciones que se toman para el desbordamiento en adiciones y sustracciones. Sin embargo, la información frecuencial de la señal es correcta como se puede apreciar en los gráficos que muestran la diferencia entre los resultados en Matlab y los de la tesis. La diferencia es en promedio de 0.0369 según Norma L2.

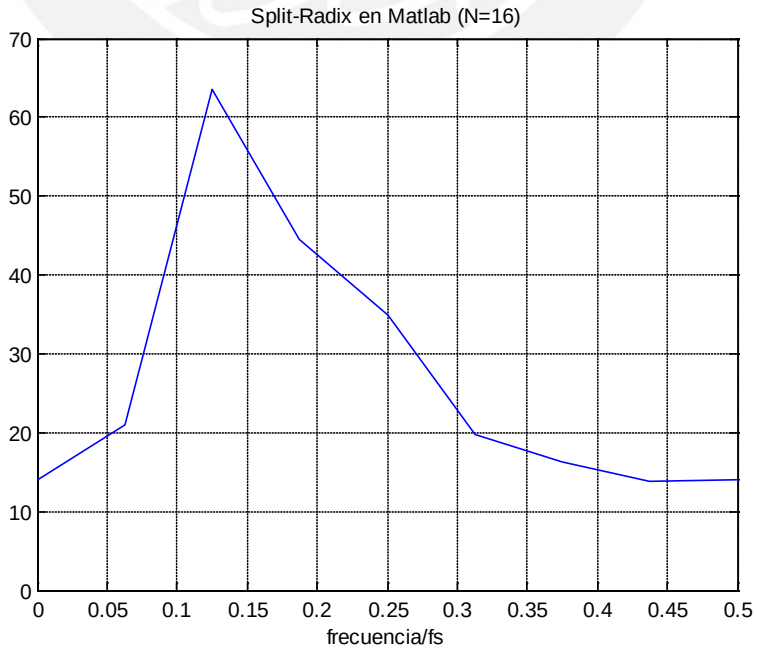


Figura 4.10. Resultados de la SRFFT en Matlab para 16 datos de entrada.

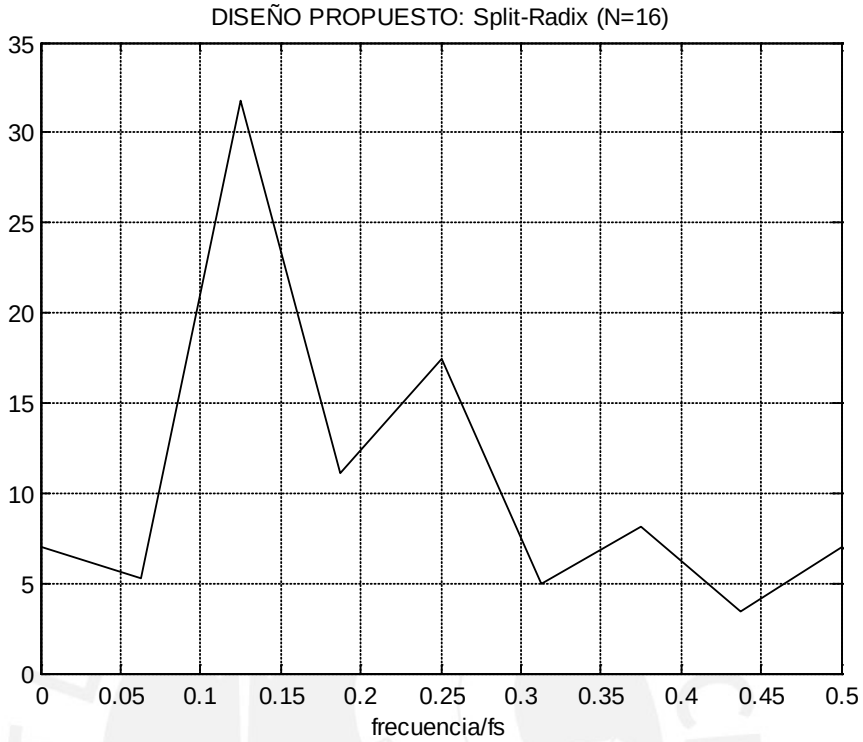


Figura 4.11. Resultados de la SRFFT del diseño propuesto para 16 datos de entrada.

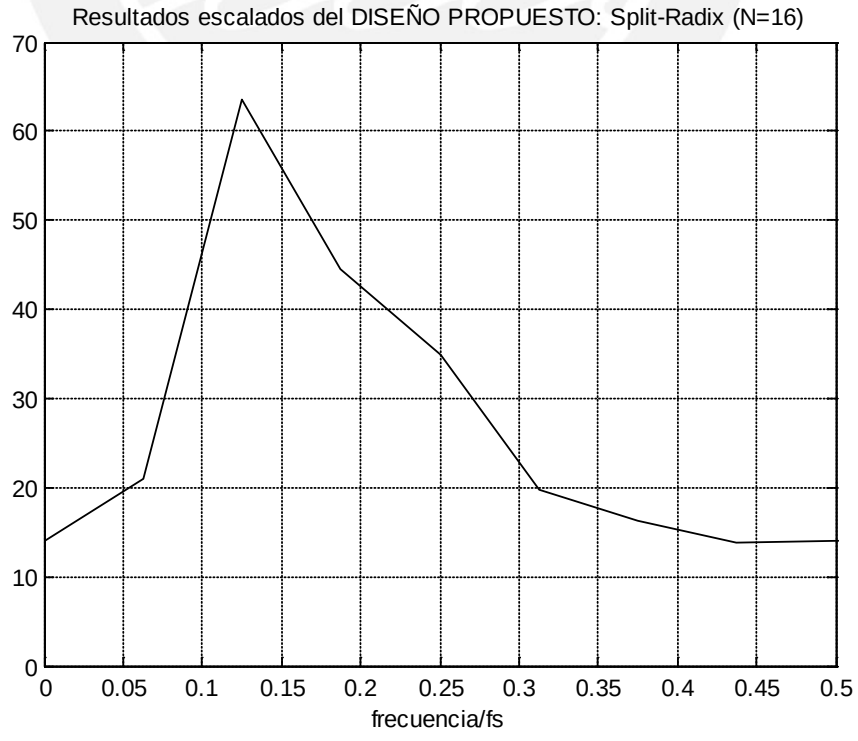
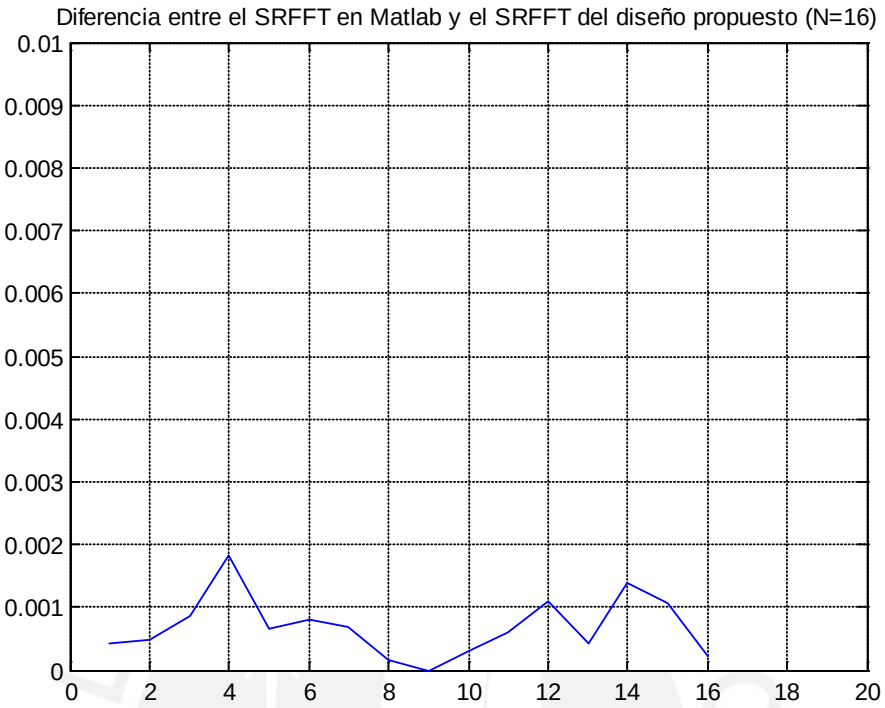


Figura 4.12. Resultados escalados de la SRFFT del diseño propuesto para 16 datos de entrada.



Diferencia: 0.0033 (Norma 2L)

Figura 4.13. Diferencia entre el SRFFT en Matlab y el SRFFT del diseño propuesto para 16 datos.

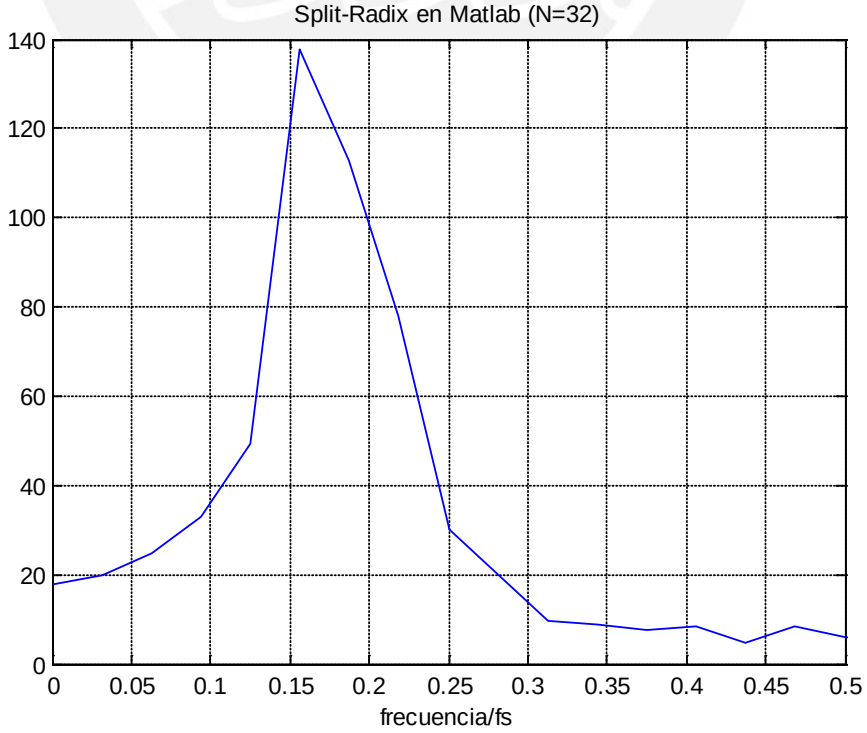


Figura 4.14. Resultados de la SRFFT en Matlab para 32 datos de entrada.

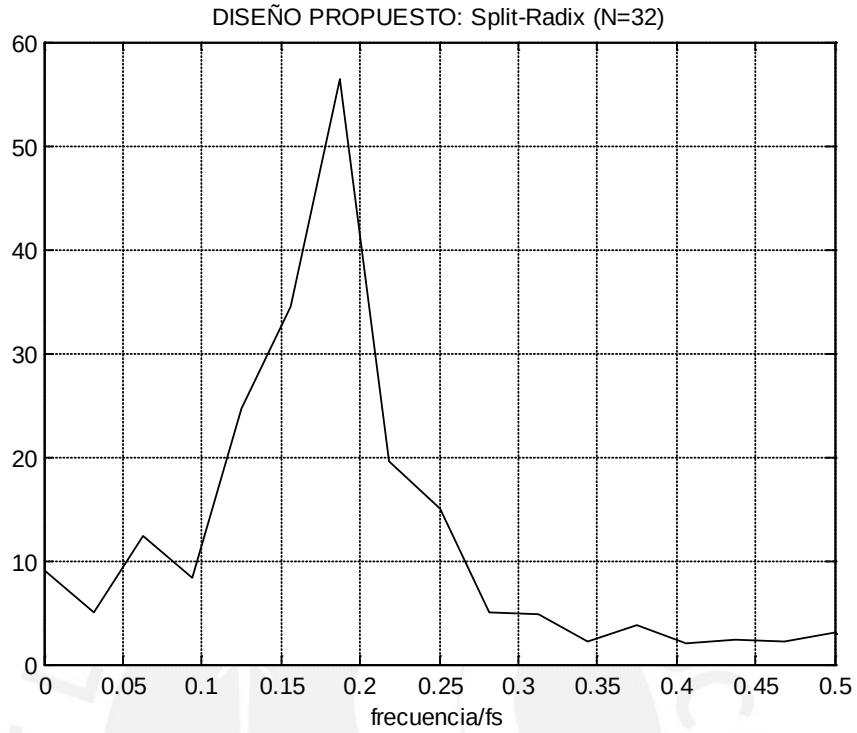


Figura 4.15. Resultados de la SRFFT del diseño propuesto para 32 datos de entrada.

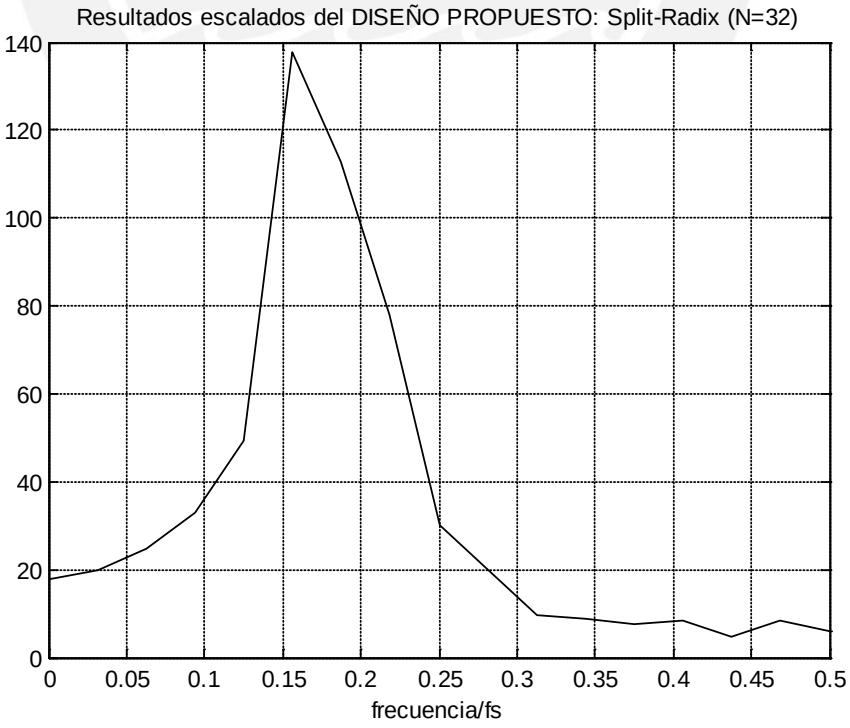
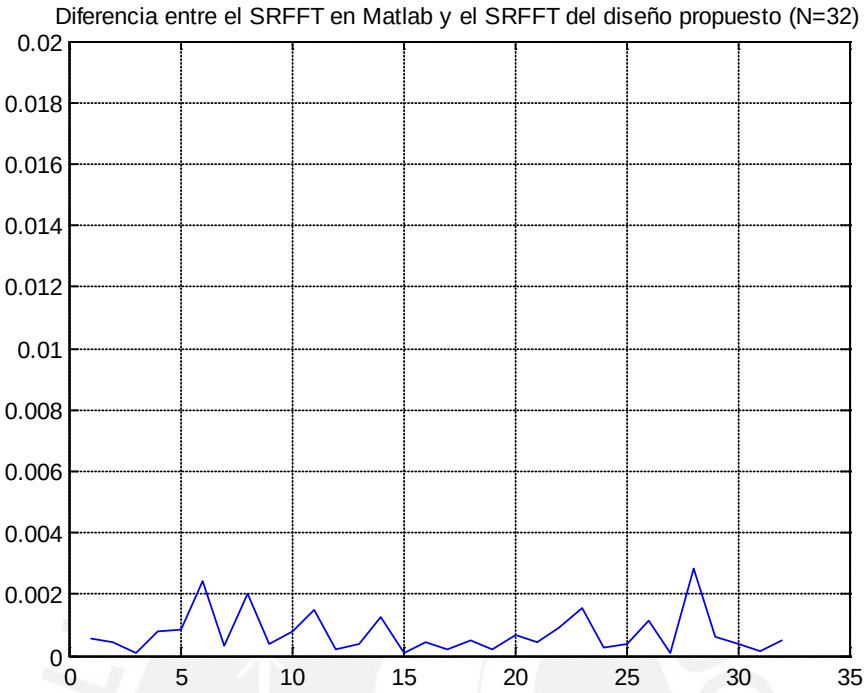


Figura 4.16. Resultados escalados de la SRFFT del diseño propuesto para 32 datos de entrada.



Diferencia: 0.0056 (Norma 2L)

Figura 4.17. Diferencia entre el SRFFT en Matlab y el SRFFT del diseño propuesto para 32 datos.

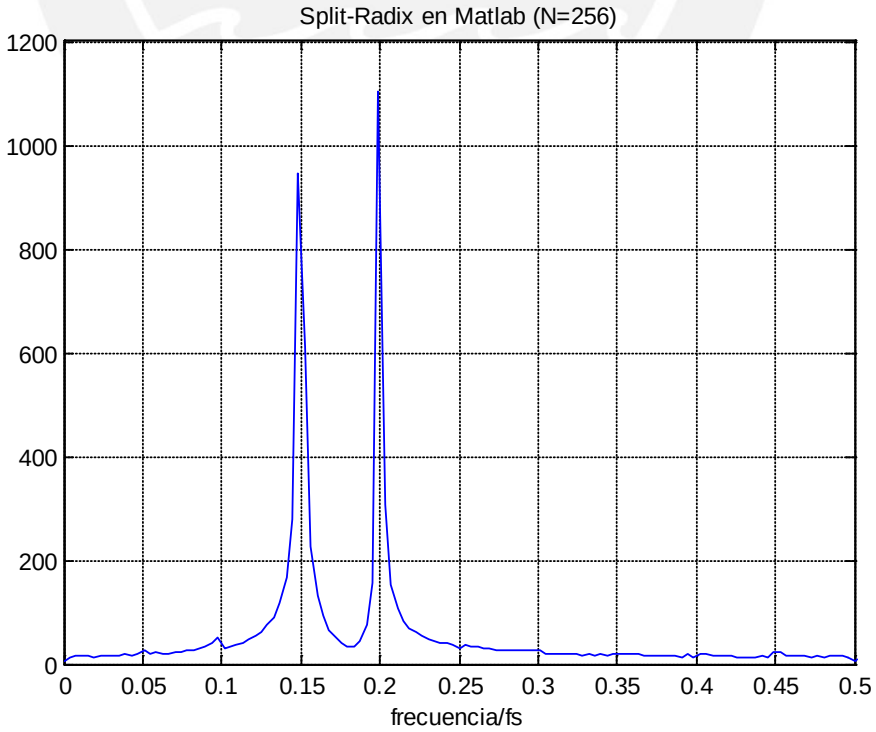


Figura 4.18. Resultados de la SRFFT en Matlab para 256 datos de entrada.

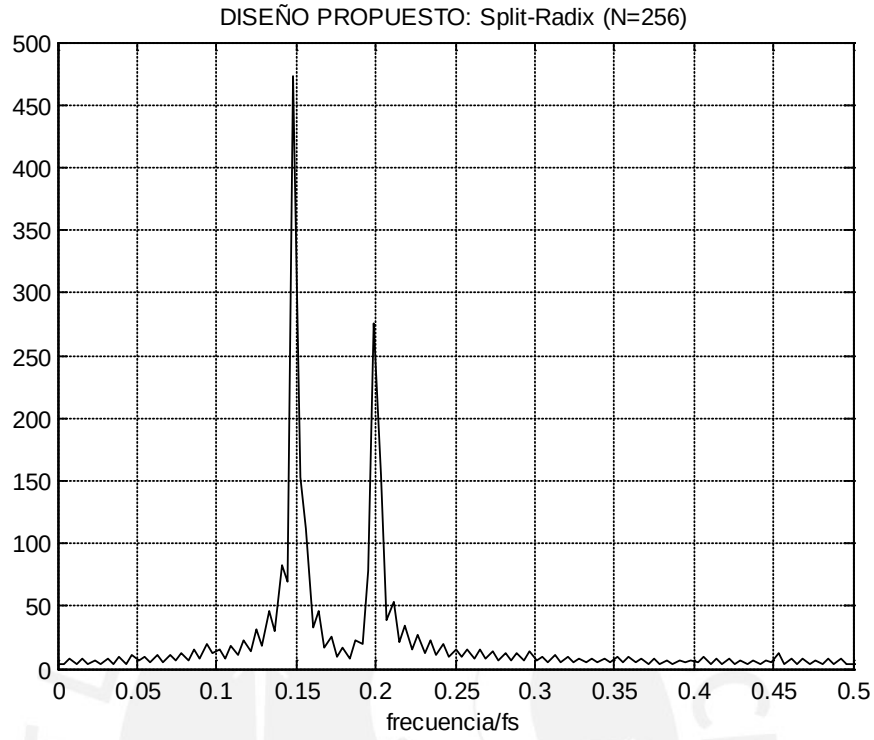


Figura 4.19. Resultados de la SRFFT del diseño propuesto para 256 datos de entrada.

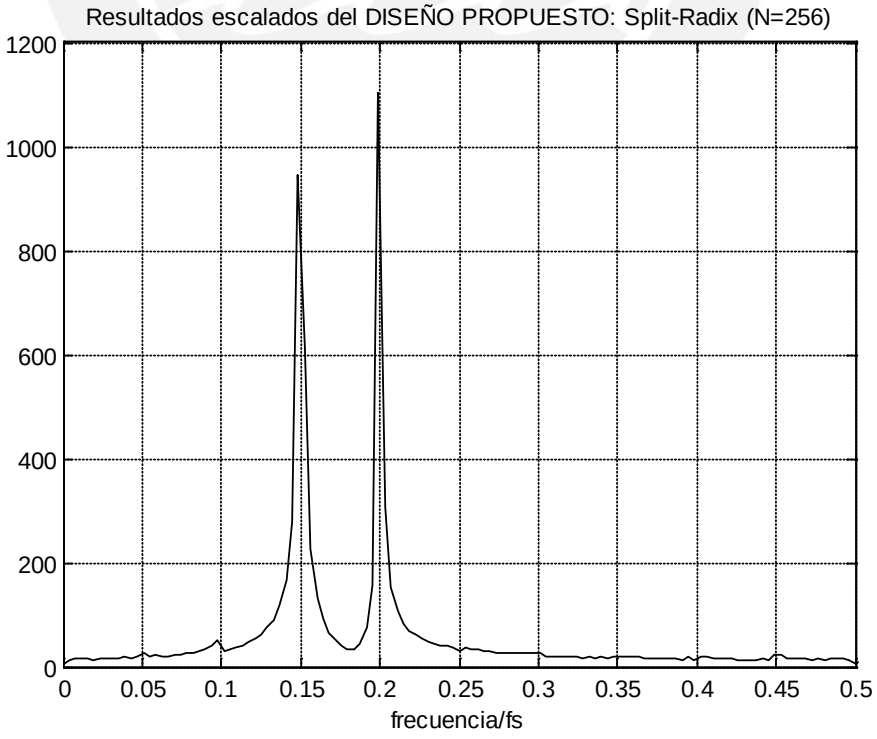
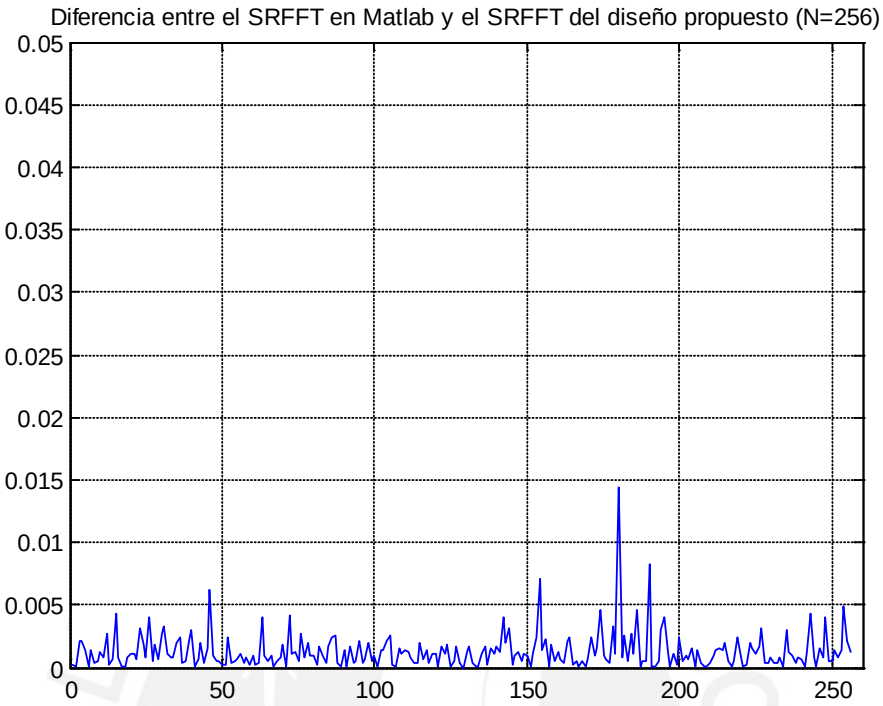


Figura 4.20. Resultados escalados de la SRFFT del diseño propuesto para 256 datos de entrada.



Diferencia: 0.0312 (Norma 2L)

Figura 4.21. Diferencia entre el SRFFT en Matlab y el SRFFT del diseño propuesto para 256 datos.

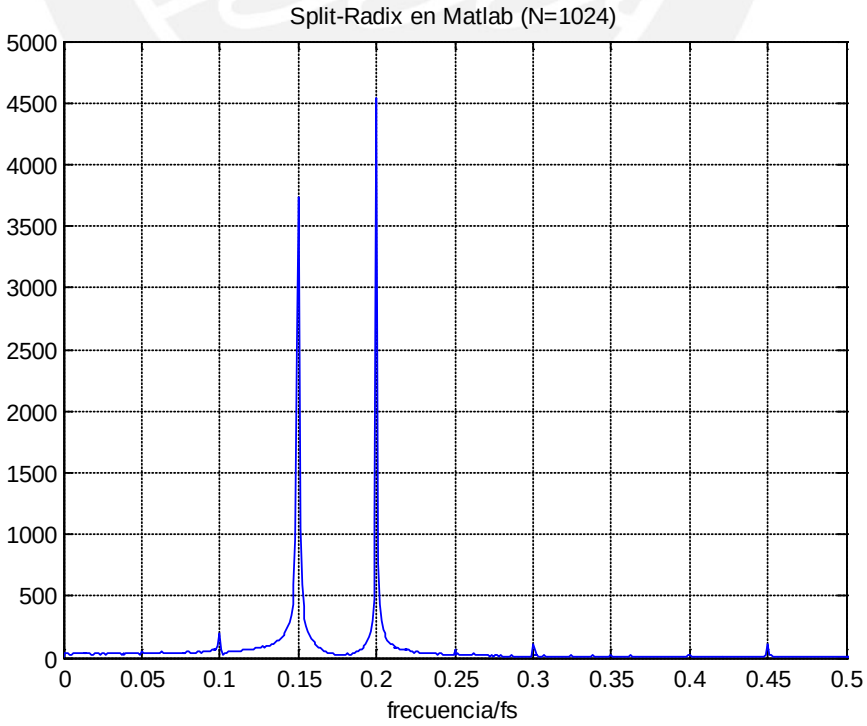


Figura 4.22. Resultados de la SRFFT en Matlab para 1024 datos de entrada.

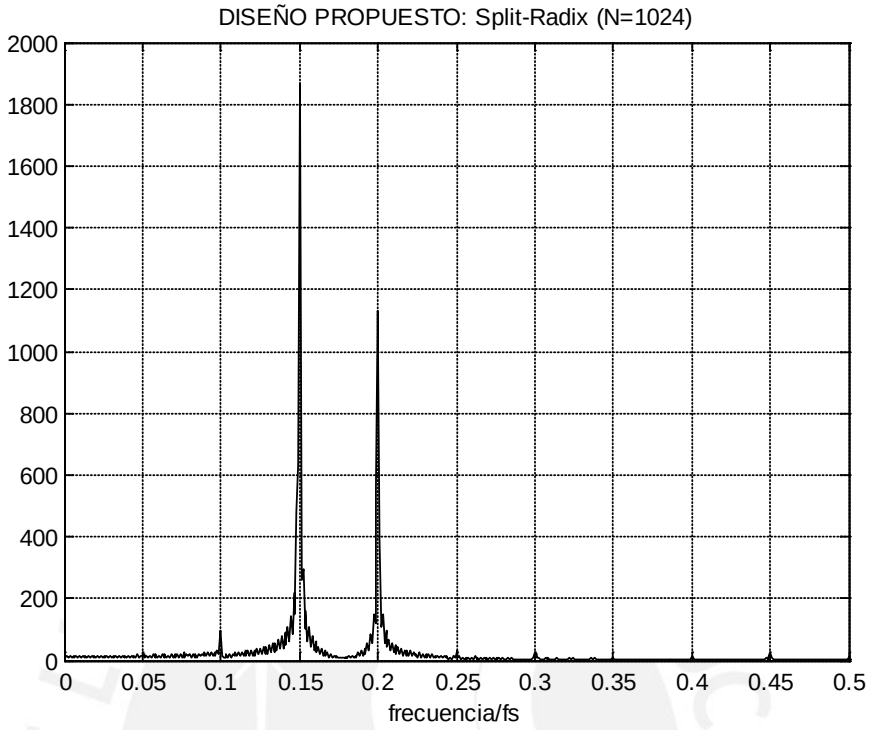


Figura 4.23. Resultados de la SRFFT del diseño propuesto para 1024 datos de entrada.

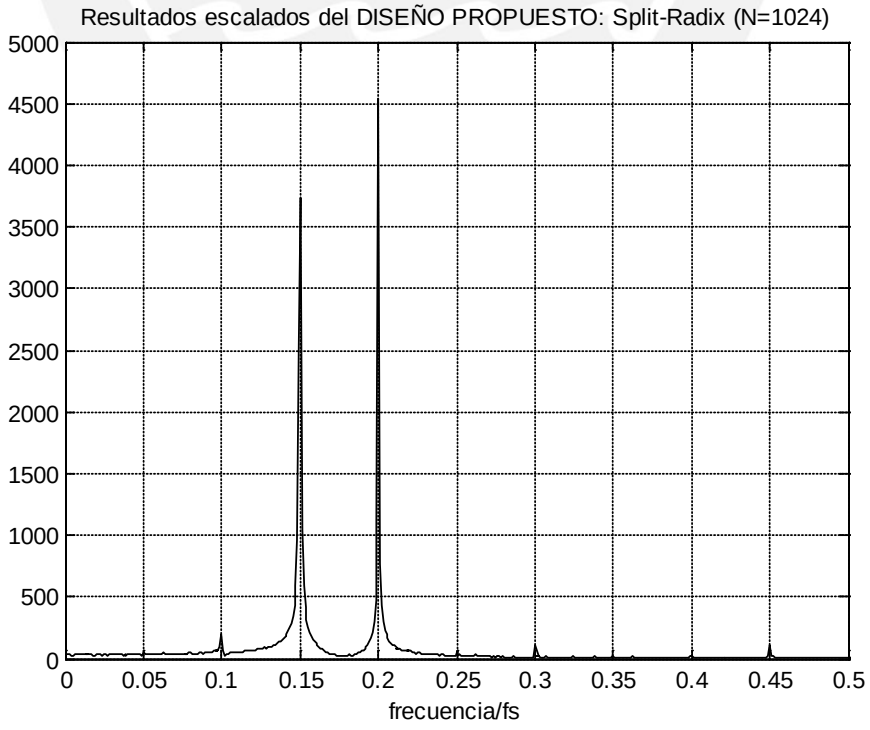
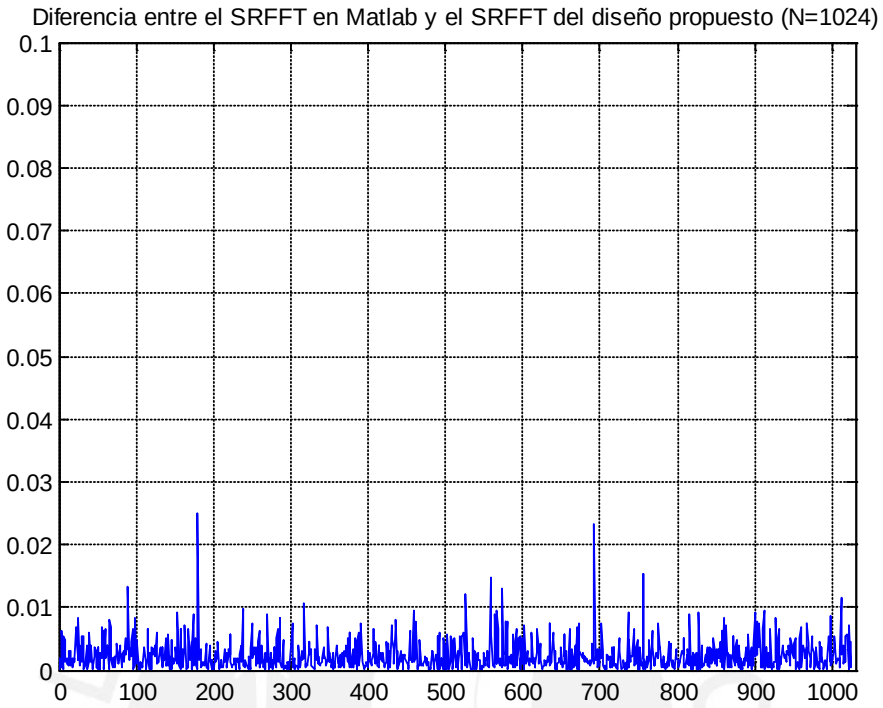


Figura 4.24. Resultados escalados de la SRFFT del diseño propuesto para 1024 datos de entrada.



Diferencia: 0.1075 (Norma 2L)

Figura 4.25. Diferencia entre el SRFFT en Matlab y el SRFFT del diseño propuesto para 1024 datos.

CONCLUSIONES

- La metodología aplicada en el diseño de la arquitectura SRFFT para la presente tesis permite una fácil configuración del sistema para $N=2^P$ datos de entrada ($P \geq 2$) , de cualquier número de bits. Esto se logra debido a la creación de bloques funcionales que son genéricos y modulares. Por lo tanto, se ha obtenido una arquitectura base que puede utilizarse en varias situaciones para diferentes requerimientos.

- Cabe resaltar que cada una de las etapas es un diseño propio que contienen una lógica y un algoritmo creado especialmente para este trabajo. Así también, la forma de reordenar los datos en la etapa final es un diseño único.

- El diseño realizado utiliza sólo algunos recursos embebidos optimizados dentro del FPGA, lo cual permite aumentar las prestaciones del sistema y no usar recursos lógicos que pueden servir para otro sub-circuito dentro de un sistema más grande; esto hace que el diseño no sea completamente portable. Sin embargo, no afecta en gran medida a la portabilidad del sistema, pues estos bloques se encuentran presentes actualmente en la mayoría de FPGAs con lo cual bastará con cambiar estos módulos por los correspondientes en el nuevo FPGA que se utilice.

- La arquitectura de esta FFT Split-Radix brinda varios beneficios como bajo número de funciones combinacionales, registros y memoria en comparación con otros trabajos realizados en este mismo tema como aquellos descritos en [15,16,17,18,19,20].

- El tiempo de operación del diseño de la SRFFT es considerable, porque las entradas y las salidas se procesan serialmente. Además, la arquitectura consta de una sola mariposa simétrica; donde la multiplicación compleja emplea una sola multiplicación, adición y sustracción, con lo cual se demora cuatro ciclos de reloj.

- Los resultados del presente diseño son comparados con los resultados del Split-Radix codificado en Matlab para 1024, 256, 32 y 16 datos de entrada de una señal sinusoidal de dos tonos y la diferencia en promedio es de 0.0369 según Norma L2.

RECOMENDACIONES

- Para el uso de esta herramienta FFT Split-Radix en cualquier aplicación, se debe de tener en cuenta las condiciones preliminares como el formato de los datos de entrada y de los coeficientes, la aproximación en adiciones y sustracciones, y el truncamiento en multiplicaciones. Con ello, se podría estimar los niveles de error introducidos por la arquitectura y así determinar si el análisis resultante es consistente con los requerimientos de la aplicación.

- Al ingresar la señal de entrada en el diseño, se debe tener en cuenta que para que se lea correctamente en la RAM, se requiere esperar 5 ciclos de reloj.

- Se pueden elaborar variaciones de la presente arquitectura de acuerdo a los requisitos de recursos y tiempo de procesamiento. Para ello, se debe tener en cuenta que existe un compromiso entra la cantidad de recursos y la velocidad: a más recursos empleados, mayor paralelismo y mayor rapidez.

- Si se desea fabricar un circuito integrado de aplicación específica de muy alta escala de integración (Very Large Scale Integrated Circuit, VLSI Circuit) a partir de la arquitectura propuesta, los bloques funcionales que hayan sido desarrollados para que se apoyen en los módulos embebidos del FPGA, deben ser rediseñados para que usen recursos lógicos en vez de estos últimos, puesto que podrán ser interpretados incorrectamente en el proceso de síntesis. Asimismo, se deben considerar otros aspectos propios de la teoría y técnicas del diseño de circuitos integrados digitales como son la tecnología a utilizar, el tiempo de respuesta de los transistores, consumo de potencia, entre otros.

BIBLIOGRAFÍA

[1] A. Oppenheim.
1999 “Discrete – Time Signal Processing”, Prentice Hall.

[2] John G. Proakis, Dimitris G. Manolakis.
1998 “Digital Signal Processing: Principles, Algorithms, and Applications”, Prentice-Hall.

[3] P. Duhamel, H. Hollmann.
1984 “Split-Radix FFT Algorithm”, Electronic Letters, volume 20, issue 1, p. 14 – 16.

[4] P. Duhamel.
1986 “Implementation of Split-Radix FFT Algorithms for Complex, Real, and Real Symmetric Data”, IEEE Transactions on Acoustics, Speech, and Signal Processing, volume ASSP-34, No 2, p. 285 – 295.

[5] S. Brown, Z. Vranesic.
2005 “Fundamentals of Digital Logic with VHDL Design”, McGraw-Hill.

[6] Altera.
2005 “Cyclone II Device Handbook”, Vol. 1.

[7] Altera.
2005 ”DE2 Development and Education Board - User Manual”, versión 1.3, Altera Corporation.

[8] S. G. Johnson, M. Frigo.
2007 “A Modified Split-Radix FFT with Fewer Arithmetic Complexity”, IEEE Transactions on Signal Processing, volume 55, issue 1, p. 111 – 119.

[9] Uwe Meyer-Baese.
2001 “Digital Signal Processing with Field Programmable Gate Arrays”, Springer-Verlag Berlin Heidelberg, Alemania, p. 2, 209-251.

[10] M. Vertteli, P. Duhamel.
1989 “Split-Radix Algorithms for Length-p^m DFT’s”, IEEE Transactions on Acoustics, Speech, and Signal Processing, volume 37, p. 57-64.

[11] Saad Bouguezel, M. Omair Ahmad, M. N. S. Swamy.
2007 “A General Class of Split-Radix FFT Algorithms for the Computation of the DFT of Length 2^m”, IEEE Transactions on Signal Processing, volume 55, p. 4127 – 4138.

[12] P. Sonna, E. Serrano, L. Castillo.
2007 “Procesador Paralelo de FFT Implementado en FPGA”, XIII Workshop Iberchip.

[13] Peng Xu, Jin Shu Chen.
2008 “FPGA Implementation of High Speed FFT algorithm Based on split-radix”, Intelligent Information Technology Application Workshops, International Symposium, p. 781-784.

[14] V.A. Pedroni.
2004 “Circuit Design with VHDL”, MIT Press, England.

[15] C. Chao, Z. Qin, X. Yingke, H. Chengde.
2005 “Design of a High performance FFT processor based on FPGA”, IEEE Proc. Asia South-Pacific Design Automation Conf., p. 920-923.

[16] Altera Inc.
2006 “FFT MegaCore function user guide”, Version 7.0.

[17] Xilinx Inc.
2007 “Xilinx LogiCore Fast Fourier Transform”, Version 4.1.

[18] J. García, J. Michell, G. Ruiz, A. Burón.
2007 “FPGA Realization of a Split Radix FFT Processor”, VLSI Circuits and Systems III, Proc. of SPIE, Vol. 6590, 65900P.

[19] P. Castro.
2005 “Desarrollo de un módulo digital para el análisis espectral de señales de audio”, Tesis de Fin de Carrera de la Universidad Politécnica de Cataluña.

[20] S. Sarkar, R. Kumar.
”Fast 16-point FFT Core for Virtex-II FPGA”, University of California, San Diego.

[21] E. Brigham.
1988 “The fast fourier transform and its applications”, Prentice-Hall.

[22] Robert W. Ramírez.
1985 “The FFT, fundamentals and concepts”, Prentice-Hall.

[23] Ronald N. Bracewell.
2000 “The Fourier transform and its applications”, McGraw-Hill.

[24] B. A. Shenoi.
2006 ”Introduction to Digital Signal Processing and Filter Design”, John Wiley & Sons, Inc., New Jersey.

[25] Communication Systems.
”FFT Tutorial”, Department of Electrical and Computer Engineering, University of Rhode Island.

[26] A. Rosado, J. Guerrero, M. Bataller, J. Espí, J. Francés.
1995 “Circuitos Programables FPGA – Fundamentos Básicos (I)”, Mundo Electrónico, mayo, p. 44 - 48.

ANEXOS

Códigos en MATLAB

Código para determinar la entrada del diseño propuesto, para graficar la FFT y SRFFT ideales dadas por MATLAB y para calcular la diferencia entre dichas gráficas.

```
clear all;
close all;
clc;

N=1024;
fm1=3000;  % Frecuencia de la señal
fm2=4000;
fs=20000;  % Frecuencia de muestreo
T=1/fs;
s=0:N-1;
t=s*T;
x=(sin(2*pi*fm1/fs*s)+ sin(2*pi*fm2/fs*s));
figure
plot(t,x)
grid on
title('Señal en el tiempo')
axis([0 0.003 -2 2])

%%%%%Entrada a enteros
for j=1:N
   y(j)=x(j)*10;
end
y=fix(y);
plot(y);
grid on
title('Señal en el tiempo')

%%%%Con FFT de matlab
espectro=abs(fft(y,N));
f=[0:N-1]/N;
figure % Nuevo cuadro para graficar
plot(f,espectro); % Gráfica de la FFT de la señal
grid on
title('FFT de Matlab (N=1024)')
axis([0 0.5 0 5000])
xlabel('frecuencia/fs')

%%Con Split-Radix
for k1 = 0:N/2-1
   A(2*k1+1) = 0;
  for n1 = 0:N/2-1
     A(2*k1+1) = A(2*k1+1)+(y(n1+1)+y(n1+1+N/2))*exp(-4*pi*i*n1*k1/N);
  end
end
```

```
for k2 = 0:N/4-1
    A(4*k2+2) = 0;
    A(4*k2+4) = 0;
   for n2 = 0:N/4-1
       A(4*k2+2) = A(4*k2+2)+((y(n2+1)-y(n2+1+N/2))-i*(y(n2+1+N/4)-y(n2+1+3*N/4)))*exp(-
2*pi*i*n2/N)*exp(-8*pi*i*n2*k2/N);
       A(4*k2+4) = A(4*k2+4)+((y(n2+1)-y(n2+1+N/2))+i*(y(n2+1+N/4)-y(n2+1+3*N/4)))*exp(-
6*pi*i*n2/N)*exp(-8*pi*i*n2*k2/N);
   end
end

% Cálculo del valor absoluto
 A = abs(A);

%%% Arreglo de frecuencias
figure
plot(f,A)
title('Split-Radix en Matlab (N=1024)')
grid on
axis([0 0.5 0 5000])
xlabel('frecuencia/fs')

%Medida de diferencia entre FFT y SRFFT en Matlab.
dif1 = abs(espectro-A);
figure
plot(dif1);
title('Diferencia entre el FFT de Matlab y el SRFFT en Matlab')
grid on
dif2 = norm(dif1);
axis([0 1030 0 0.000000001])
```

Arreglo de los datos de entrada para las simulaciones en ModelSim.

```
clc;
N=1024;
 for j=1:N
    y(j)=x(j)*10;
end
 y=fix(y);
 plot(y);
 fentrada=fopen('entrada.txt','w');
for i=1:N
fprintf(fentrada,'conv_std_logic_vector(%d,bits),\n',y(i));
end
fclose(fentrada);
```

Código que genera y escala los coeficientes trigonométricos.

```
clc;
clear all;
N=1024;
bits=16;
for n = 1:N
    a(n) = round(cos(2*pi*(n-1)/N) * 2^(bits-1));
    b(n) = round(-sin(2*pi*(n-1)/N) * 2^(bits-1));
end
for n = 1:N
  if a(n) == 2^(bits-1)
     a(n) = 2^(bits-1)-1;
  end;
  if b(n) == 2^(bits-1)
     b(n) = 2^(bits-1)-1;
 end;
end
a=int16(a);
b=int16(b);
```

Arreglo de los coeficientes trigonométricos para las simulaciones en ModelSim.

```
clc;
N=1024;
fre=fopen('coefre.txt','w');
for i=1:N/4
fprintf(fre,'conv_std_logic_vector(%d,bits),\n',a(i));
end
fclose(fre);

fim=fopen('coefim.txt','w');
for j=1:N/4
fprintf(fim,'conv_std_logic_vector(%d,bits),\n',b(j));
end
fclose(fim);
```

Código para graficar los resultados dados en las simulaciones en ModelSim.

```
clc;
clear all;
N=32;
fre=fopen('SRFFT-32','r');
 for j=1:3
 l=fgets(fre);
end
```

```
for i=1:N
l=fgets(fre);
re(i)=0;
im(i)=0;
 %Real
   if l(17)=='0'
      for j=1:16
        if l(33-j)=='1'
           re(i)=re(i)+2^(j-1);
        end
      end
   else
      for j=1:16
        if l(33-j)=='1'
           re1(33-j)=0;
        else
           re1(33-j)=1;
        end
        if re1(33-j)==1
           re(i)=re(i)+2^(j-1);
        end
      end
      re(i)=-1*(re(i)+1);
   end
%Imaginario
   if l(33)=='0'
      for j=1:16
        if l(49-j)=='1'
           im(i)=im(i)+2^(j-1);
        end
      end
   else
      for j=1:16
        if l(49-j)=='1'
           im1(49-j)=0;
        else
           im1(49-j)=1;
        end
        if im1(49-j)==1
           im(i)=im(i)+2^(j-1);
        end
      end
      im(i)=-1*(im(i)+1);
   end
end
fclose(fre);
for i=1:N
   srfft(i)=sqrt(re(i)^2+im(i)^2);
end
 fs=20000;  % Frecuencia de muestreo
```

```
 %%%SRFFT del diseño propuesto
f=[0:N-1]/N;
figure; %Nuevo cuadro para graficar
plot(f,(srfft)); % Gráfica de la FFT de la señal
grid on
title('SRFFT del diseño propuesto')
```

Código en VHDL

Diseño propuesto de la SRFFT

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity bit_inv is
  generic(bits : positive := 5);
  port (clock,hab : in std_logic;
        a : in std_logic_vector(bits-1 downto 0);
        a_inv : out std_logic_vector(bits-1 downto 0));
end bit_inv;

architecture est of bit_inv is
   begin
      process (clock,hab)
      begin
          if (clock'event and clock ='1') then
           if (hab='1') then
            for i in 0 to bits-1 loop
                    a_inv(bits-(i+1)) <= a(i);
            end loop;
          end if;
        end if;
     end process;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity comparador is
 generic (bits : integer := 10);
 port (cuenta,tope: in std_logic_vector(bits-1 downto 0);
        fin: out std_logic);
end comparador;
```

```
architecture estructura of comparador is
begin
          fin <= '1' when (cuenta = tope) else '0';
end estructura;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity contador is
  generic(bits : positive := 10); -- número de bits de los datos
  port (clk,reset,hab:in std_logic;
        Q:out std_logic_vector(bits-1 downto 0));
end contador;

architecture est of contador is
   begin
      process (clk,reset)
         variable cont:std_logic_vector(bits-1 downto 0) := (others=>'0');
      begin
         if (reset = '1') then
            cont:=(others=>'0');
         elsif rising_edge(clk) then
            if (hab='1') then
               cont:=cont+1;
               if cont=conv_std_logic_vector(0,bits) then
                  cont:=(others=>'1');
               end if;
            end if;
          end if;
      Q<=cont;
      end process;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
use work.split_package.all;

entity control is
end control;

architecture tb of control is
constant clock_period : time := 10 ns;
signal clock : std_logic := '0';
```

```
signal reset : std_logic := '1';
signal inicio : std_logic := '1';
signal fft_in : std_logic_vector (bits-1 downto 0) := (others => '0');
signal fft_out : std_logic_vector (data-1 downto 0) := (others => '0');
type sig is array (0 to N-1) of std_logic_vector (bits-1 downto 0);
constant datos: sig:= (conv_std_logic_vector(0,bits),
conv_std_logic_vector(17,bits),
conv_std_logic_vector(15,bits),
conv_std_logic_vector(-2,bits),
conv_std_logic_vector(-15,bits),
conv_std_logic_vector(-10,bits),
conv_std_logic_vector(3,bits),
conv_std_logic_vector(8,bits),
conv_std_logic_vector(3,bits),
conv_std_logic_vector(-1,bits),
conv_std_logic_vector(0,bits),
conv_std_logic_vector(1,bits),
conv_std_logic_vector(-3,bits),
conv_std_logic_vector(-8,bits),
conv_std_logic_vector(-3,bits),
conv_std_logic_vector(9,bits),
conv_std_logic_vector(15,bits),
conv_std_logic_vector(2,bits),
conv_std_logic_vector(-15,bits),
conv_std_logic_vector(-17,bits),
conv_std_logic_vector(0,bits),
conv_std_logic_vector(17,bits),
conv_std_logic_vector(15,bits),
conv_std_logic_vector(-2,bits),
conv_std_logic_vector(-15,bits),
conv_std_logic_vector(-10,bits),
conv_std_logic_vector(3,bits),
conv_std_logic_vector(8,bits),
conv_std_logic_vector(3,bits),
conv_std_logic_vector(-1,bits),
conv_std_logic_vector(0,bits),
conv_std_logic_vector(1,bits));
begin
t0: split_radix generic map (N,bits,address,data) port map (clock,reset,inicio,fft_in,fft_out);
process
begin
wait for clock_period;
reset <= '0';
end process;
process
begin
clock <= '0';
wait for clock_period/2;
clock <= '1';
wait for clock_period/2;
end process;
```

```
process
variable i:integer:=0;
begin
if i = 0 then
wait for 5*clock_period;
end if;
wait for clock_period;
if i = N then
i := N-1;
end if;
fft_in <= datos(i);
i := i + 1;
end process;
end tb;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity div2 is
  generic(bits : positive := 10); -- número de bits de los datos
  port (clk,sel,hab:in std_logic;
        y:in std_logic_vector(bits-1 downto 0);
        Q:out std_logic_vector(bits-1 downto 0));
end div2;

architecture est of div2 is
  signal y1,y1reg,y2:std_logic_vector(y'range) := (others=>'0');
 begin
  process (clk)
      begin
        if  clk'event and clk='1' then
          if (hab='1') then
             y1reg<=y1;
          end if;
        end if;
  end process;
      y1 <= y when sel='1' else y2;
      y2 <= '0' & y1reg(bits-1 downto 1);
      Q <= y1reg;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

```
entity div4 is
  generic(bits : positive := 10); -- número de bits de los datos
  port (clk,sel,hab:in std_logic;
        y:in std_logic_vector(bits-1 downto 0);
        Q:out std_logic_vector(bits-1 downto 0));
end div4;

architecture est of div4 is
  signal y1,y1reg,y2:std_logic_vector(y'range) := (others=>'0');
 begin
  process (clk)
      begin
       if  clk'event and clk='1' then
         if (hab='1') then
            y1reg<=y1;
         end if;
       end if;
  end process;
      y1 <= y when sel='1' else y2;
      y2 <= "00" & y1reg(bits-1 downto 2);
      Q <= y1reg;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity dmux is
  generic (bits : integer := 10);
  port (y: in std_logic_vector (bits-1 downto 0);
          sel: in std_logic_vector (1 downto 0);
          z: out arreglo);
end dmux;

architecture estruct of dmux is
begin
process (y,sel)
variable t : arreglo;
begin
          t := (others => (others=>'0'));
          t(conv_integer(sel)) := y;
          z <= t;
end process;
end estruct;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity dmux1 is
  generic (bits : integer := 32);
  port (y: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic;
        z: out arreglo1);
end dmux1;

architecture estruct of dmux1 is
begin
process (y,sel)
variable t : arreglo1;
begin
        t := (others => (others=>'0'));
        t(conv_integer(sel)) := y;
        z <= t;
end process;
end estruct;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity dmux2 is
  generic (bits : integer := 32);
  port (y: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (1 downto 0);
        z: out arreglo2);
end dmux2;

architecture estruct of dmux2 is
begin
process (y,sel)
variable t : arreglo2;
begin
        t := (others => (others=>'0'));
        t(conv_integer(sel)) := y;
        z <= t;
end process;
end estruct;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity dmux3 is
  generic (bits : integer := 32);
  port (y: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (1 downto 0);
        z: out arreglo3);
end dmux3;

architecture estruct of dmux3 is
begin
process (y,sel)
variable t : arreglo3;
begin
        t := (others => (others=>'0'));
        t(conv_integer(sel)) := y;
        z <= t;
end process;
end estruct;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity dmux_reg is
  generic (bits : integer := 10);
  port (clk,reset: in std_logic;
        y: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (1 downto 0);
        Q0,Q1,Q2,Q3: out std_logic_vector (bits-1 downto 0));
end dmux_reg;

architecture estruct of dmux_reg is
signal hab0,hab1,hab2,hab3: std_logic := '0';
signal z: arreglo;
begin
process (sel)
 begin
  case conv_integer(sel) is
        when 0 =>
                hab0 <= '1';
                hab1 <= '0';
                hab2 <= '0';
```

```
                hab3 <= '0';
        when 1 =>
                hab0 <= '0';
                hab1 <= '1';
                hab2 <= '0';
                hab3 <= '0';
        when 2 =>
                hab0 <= '0';
                hab1 <= '0';
                hab2 <= '1';
                hab3 <= '0';
        when others =>
                hab0 <= '0';
                hab1 <= '0';
                hab2 <= '0';
                hab3 <= '1';
  end case;
end process;
t0: dmux generic map (bits) port map (y,sel,z);
t1: reg generic map (bits) port map (clk,reset,hab0,z(0),Q0);
t2: reg generic map (bits) port map (clk,reset,hab1,z(1),Q1);
t3: reg generic map (bits) port map (clk,reset,hab2,z(2),Q2);
t4: reg generic map (bits) port map (clk,reset,hab3,z(3),Q3);
end estruct;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity dmux_reg2 is
  generic (bits : integer := 32);
  port (clk,reset: in std_logic;
        y: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (1 downto 0);
        Q0,Q1,Q2,Q3: out std_logic_vector (bits-1 downto 0));
end dmux_reg2;

architecture estruct of dmux_reg2 is
signal hab0,hab1,hab2,hab3: std_logic := '0';
signal z: arreglo2;
begin
process (sel)
 begin
  case conv_integer(sel) is
        when 0 =>
                hab0 <= '1';
                hab1 <= '0';
```

```
                hab2 <= '0';
                hab3 <= '0';
        when 1 =>
                hab1 <= '1';
                hab0 <= '0';
                hab2 <= '0';
                hab3 <= '0';
        when 2 =>
                hab2 <= '1';
                hab0 <= '0';
                hab1 <= '0';
                hab3 <= '0';
        when others =>
                hab3 <= '1';
                hab0 <= '0';
                hab1 <= '0';
                hab2 <= '0';
  end case;
end process;
t0: dmux2 generic map (bits) port map (y,sel,z);
t1: reg generic map (bits) port map (clk,reset,hab0,z(0),Q0);
t2: reg generic map (bits) port map (clk,reset,hab1,z(1),Q1);
t3: reg generic map (bits) port map (clk,reset,hab2,z(2),Q2);
t4: reg generic map (bits) port map (clk,reset,hab3,z(3),Q3);
end estruct;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity dmux_reg3 is
  generic (bits : integer := 32);
  port (clk,reset: in std_logic;
        y: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (1 downto 0);
        Q0,Q1,Q2: out std_logic_vector (bits-1 downto 0));
end dmux_reg3;

architecture estruct of dmux_reg3 is
signal sel1: std_logic_vector (1 downto 0);
signal hab0,hab1,hab2: std_logic := '0';
signal z: arreglo3;
begin
process (sel)
 begin
  case conv_integer(sel) is
        when 0 =>
```

```
                hab0 <= '1';
                hab1 <= '0';
                hab2 <= '0';
        when 1 =>
                hab0 <= '0';
                hab1 <= '1';
                hab2 <= '0';
        when 2 =>
                hab0 <= '0';
                hab1 <= '0';
                hab2 <= '1';
        when others =>
                hab0 <= '0';
                hab1 <= '0';
                hab2 <= '0';
  end case;
end process;
sel1 <= "00" when sel = "11" else sel;
t0: dmux3 generic map (bits) port map (y,sel1,z);
t1: reg generic map (bits) port map (clk,reset,hab0,z(0),Q0);
t2: reg generic map (bits) port map (clk,reset,hab1,z(1),Q1);
t3: reg generic map (bits) port map (clk,reset,hab2,z(2),Q2);
end estruct;
```

```
USE ieee.std_logic_1164.all;

ENTITY maq_entrada IS
   PORT (clock,reset_entrada,inicio1,par : IN STD_LOGIC;
pares,fcont0,fin,fcont2,fcont5,salto,mayor1,fcont13,fase,ini : in std_logic;
reset0,hab0,sel1,sel2,hab1,stope,reset1,hab2,reset2,hab3,reset3,hab4,sel3,hab5 : out std_logic;
reset4,hab6,sel4,hab7,reset5,hab8,reset6,hab9,reset7,reset8,hab10,we1 : out std_logic;
sel5,reset9,hab11,reset10,reset11,reset12,hab12,reset13,hab13 : out std_logic;
sel6: out std_logic_vector(1 downto 0));
END maq_entrada;

ARCHITECTURE estruct OF maq_entrada IS
   TYPE estado IS
(S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,S20,S21,S22,S23,
S24,S25,S26,S27);
   SIGNAL estado_pr, estado_sg : estado;
   SIGNAL temp: std_logic_vector (36 downto 0):= (others=>'0');
BEGIN
-----------------------Lógica Secuencial----------------------
   SEQ: PROCESS (reset_entrada,clock)
   BEGIN
     IF (reset_entrada = '1') THEN
      estado_pr <= S0;
     ELSIF Rising_Edge(clock) THEN
      reset0 <= temp(36);
```

```
        hab0 <= temp(35);
        sel1 <= temp(34);
        sel2 <= temp(33);
        hab1 <= temp(32);
        stope <= temp(31);
        reset1 <= temp(30);
        hab2 <= temp(29);
        reset2 <= temp(28);
        hab3 <= temp(27);
        reset3 <= temp(26);
        hab4 <= temp(25);
        sel3 <= temp(24);
        hab5 <= temp(23);
        reset4 <= temp(22);
        hab6 <= temp(21);
        sel4 <= temp(20);
        hab7 <= temp(19);
        reset5 <= temp(18);
        hab8 <= temp(17);
        reset6 <= temp(16);
        hab9 <= temp(15);
        reset7 <= temp(14);
        reset8 <= temp(13);
        hab10 <= temp(12);
        we1 <= temp(11);
        sel5 <= temp(10);
        reset9 <= temp(9);
        hab11 <= temp(8);
        reset10 <= temp(7);
        sel6 <= temp(6 downto 5);
        reset11 <= temp(4);
        reset12 <= temp(3);
        hab12 <= temp(2);
        reset13 <= temp(1);
        hab13 <= temp(0);
              estado_pr <= estado_sg;
      END IF;
    END PROCESS SEQ;
----------------------Lógica Combinacional estado siguiente---------------------
COMB: PROCESS
(estado_pr,inicio1,par,pares,fcont0,fin,fcont2,fcont5,salto,mayor1,fcont13,fase,ini)
  BEGIN
          CASE estado_pr IS
                  WHEN S0 =>
                  temp <= "00000000000000000000000000000000000000";
                  IF inicio1 = '1' THEN estado_sg <= S1;
                  ELSE estado_sg <= S0;
                  END IF;
                  WHEN S1 =>
                  temp <= "00100010101000100010000101000000000000";
                  estado_sg <= S2;
```

```
WHEN S2 =>
temp <= "00100001101000100010000101000000000000";
IF fin = '0' THEN estado_sg <= S2;
ELSE estado_sg <= S3;
END IF;
WHEN S3 =>
temp <= "00011110101011010000100100010000001000";
estado_sg <= S4;
WHEN S4 =>
temp <= "00000100010000000000010100000000000000";
estado_sg <= S5;
WHEN S5 =>
temp <= "00000100010000000000010100000100000100";
IF fcont2 = '0' THEN estado_sg <= S5;
ELSE estado_sg <= S6;
END IF;
WHEN S6 =>
temp <= "00000100000000000000000000000000001010";
IF inicio1 = '0' THEN estado_sg <= S6;
ELSE estado_sg <= S7;
END IF;
WHEN S7 =>
temp <= "10000100000000000000000001100000000000";
estado_sg <= S8;
WHEN S8 =>
temp <= "01000100000000000000000001100000000000";
IF fcont13 = '0' THEN estado_sg <= S8;
ELSIF fin = '0' THEN estado_sg <= S9;
ELSIF fcont5 = '0' THEN estado_sg <= S10;
ELSIF salto = '0' THEN estado_sg <= S12;
ELSIF ini = '1' THEN estado_sg <= S11;
ELSE estado_sg <= S13;
END IF;
WHEN S9 =>
temp <= "00000101100000000000100100010000000000";
estado_sg <= S4;
WHEN S10 =>
temp <= "00000100100000000001100100010000000001";
estado_sg <= S14;
WHEN S11 =>
temp <= "00000100100000001101100100010000000001";
estado_sg <= S14;
WHEN S12 =>
temp <= "00000100100000000011001000100000000001";
estado_sg <= S14;
 WHEN S13 =>
temp <= "00000100100000000011001000100000000001";
estado_sg <= S14;
WHEN S14 =>
temp <= "00000110100000000000000100010000000001";
If fase = '0' THEN estado_sg <= S14;
```

```
ELSIF mayor1 = '0' THEN estado_sg <= S4;
ELSIF pares = '0' THEN estado_sg <= S15;
ELSE estado_sg <= S16;
END IF;
WHEN S15 =>
temp <= "00001110100101010110100100000000000010";
estado_sg <= S17;
WHEN S16 =>
temp <= "00001110101000010110100100000000000010";
estado_sg <= S17;
WHEN S17 =>
temp <= "00000010000000000000000000000001001";
If fase = '0' THEN estado_sg <= S17;
ELSIF fcont0 = '0' THEN estado_sg <= S4;
ELSIF par = '0' THEN estado_sg <= S18;
ELSE estado_sg <= S23;
END IF;
WHEN S18 =>
temp <= "00000001101000100010000100100000000000";
estado_sg <= S19;
WHEN S19 =>
temp <= "00000001101000100010000100100000100000";
IF fin = '0' THEN estado_sg <= S20;
ELSE estado_sg <= S0;
END IF;
WHEN S20 =>
temp <= "00000001101000100010000100100001000000";
estado_sg <= S21;
WHEN S21 =>
temp <= "00000000000000000000000000010000000000";
estado_sg <= S22;
WHEN S22 =>
temp <= "00000000000000000000000000010001100000";
IF inicio1 = '0' THEN estado_sg <= S22;
ELSE estado_sg <= S18;
END IF;
WHEN S23 =>
temp <= "00000001101000100010000100100000000000";
estado_sg <= S24;
WHEN S24 =>
temp <= "00000001101000100010000100100000000000";
estado_sg <= S25;
WHEN S25 =>
temp <= "00000001101000100010000100100000100000";
estado_sg <= S26;
WHEN S26 =>
temp <= "00000000000000000000000000010001000000";
estado_sg <= S27;
WHEN S27 =>
temp <= "00000000000000000000000000000001100000";
IF inicio1 = '0' THEN estado_sg <= S27;
```

```
                ELSE estado_sg <= S23;
                END IF;
        END CASE;
END PROCESS COMB;
END estruct;
```

```
library ieee;
USE ieee.std_logic_1164.all;

ENTITY maq_mariposa IS
   PORT (clock,reset_mariposa,inicio2 : IN STD_LOGIC;
          fcont0,f0,fmari : in std_logic;
          s0,h0,r1,h1,s1,h2,s2,h3,re0,en0,en1,en2,en3,re1,en4,re2,en5,en6 : out std_logic);
END maq_mariposa;

ARCHITECTURE estruct OF maq_mariposa IS
    TYPE estado IS (e0,e1,e2,e3,e4,e5,e6,e7,e8);
   SIGNAL estado_pr, estado_sg : estado;
   SIGNAL temp: std_logic_vector (17 downto 0) := (others=>'0');
BEGIN
------------------------Lógica Secuencial-----------------------
   SEQ: PROCESS (reset_mariposa,clock)
   BEGIN
     IF (reset_mariposa = '1') THEN
      estado_pr <= e0;
     ELSIF Rising_Edge(clock) THEN
                s0 <= temp(17);
                h0 <= temp(16);
                r1 <= temp(15);
                h1 <= temp(14);
                s1 <= temp(13);
                h2 <= temp(12);
                s2 <= temp(11);
                h3 <= temp(10);
                re0 <= temp(9);
                en0 <= temp(8);
                en1 <= temp(7);
                en2 <= temp(6);
                en3 <= temp(5);
                re1 <= temp(4);
                en4 <= temp(3);
                re2 <= temp(2);
                en5 <= temp(1);
                en6 <= temp(0);
          estado_pr <= estado_sg;
     END IF;
   END PROCESS SEQ;
```

```
----------------------Lógica Combinacional estado siguiente--------------------
COMB: PROCESS (estado_pr,inicio2,fcont0,f0,fmari)
 BEGIN
        CASE estado_pr IS
                WHEN e0 =>
                temp <= "0000000000000000";
                IF inicio2 = '0' THEN estado_sg <= e0;
                ELSE estado_sg <= e1;
                END IF;

                WHEN e1 =>
                temp <= "1110111110000010000";
                estado_sg <= e2;
                WHEN e2 =>
                temp <= "0000000000000001000";
                IF fmari = '0' THEN estado_sg <= e2;
                ELSE estado_sg <= e3;
                END IF;
                WHEN e3 =>
                temp <= "0000000000111100000";
                estado_sg <= e4;
                WHEN e4 =>
                temp <= "0000000000000010000";
                IF inicio2='0' THEN estado_sg <= e4;
                ELSIF fcont0='1' THEN estado_sg <= e8;
                ELSIF f0='0' THEN estado_sg <= e5;
                ELSE estado_sg <= e7;
                END IF;
                WHEN e5 =>
                temp <= "0001000000000000000";
                estado_sg <= e6;
                WHEN e6 =>
                temp <= "0000000000000001000";
                IF fmari = '0' THEN estado_sg <= e6;
                ELSE estado_sg <= e3;
                END IF;
                WHEN e7 =>
                temp <= "0110010100000000000";
                estado_sg <= e6;
                WHEN e8 =>
                temp <= "0000000000000000011";
                estado_sg <= e4;
        END CASE;
END PROCESS COMB;
END estruct;
```

```
library ieee;
USE ieee.std_logic_1164.all;

ENTITY maq_orden IS
   PORT (clock,reset_orden,inicio3,par,fin,fmux3,fmux4,fmux2,sfin : IN STD_LOGIC;
 rst0,ha0,par1,rst1,ha1,rst2,ha2,rst3,ha3,arst4,ha4,ha5,rst5,ha6,we2,rst6,ha7 : out std_logic);
END maq_orden;

ARCHITECTURE estruct OF maq_orden IS
   TYPE estado IS
(s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,s16,s17,s18,s19,s20,s21,s22,s23,s24,s25,s26,s
27,s28);
   SIGNAL estado_pr, estado_sg : estado;
   SIGNAL temp: std_logic_vector (16 downto 0) := (others=>'0');
BEGIN
-----------------------Lógica Secuencial----------------------
   SEQ: PROCESS (reset_orden,clock)
   BEGIN
      IF (reset_orden = '1') THEN
       estado_pr <= s0;
      ELSIF Rising_Edge(clock) THEN
      rst6 <= temp(16);
      ha7 <= temp(15);
      rst0 <= temp(14);
      ha0 <= temp(13);
      par1 <= temp(12);
      rst1 <= temp(11);
      ha1 <= temp(10);
      rst2 <= temp(9);
      ha2 <= temp(8);
      rst3 <= temp(7);
      ha3 <= temp(6);
      arst4 <= temp(5);
      ha4 <= temp(4);
      ha5 <= temp(3);
      rst5 <= temp(2);
      ha6 <= temp(1);
      we2 <= temp(0);
      estado_pr <= estado_sg;
      END IF;
   END PROCESS SEQ;
-----------------------Lógica Combinacional estado siguiente----------------------
COMB: PROCESS (estado_pr,inicio3,par,fin,fmux3,fmux4,fmux2,sfin)
  BEGIN
         CASE estado_pr IS
                WHEN s0 =>
                temp <= "10100101010100100";
                IF inicio3 = '0' THEN estado_sg <= s0;
                ELSIF par = '0' THEN estado_sg <= s1;
                ELSE estado_sg <= s15;
                END IF;
```

```
WHEN s1 =>
temp <= "01100100001000000";
estado_sg <= s2;
WHEN s2 =>
temp <= "01000000000011000";
estado_sg <= s3;
WHEN s3 =>
temp <= "01000000000001000";
IF fmux4 = '0' THEN estado_sg <= s4;
ELSE estado_sg <= s6;
END IF;
WHEN s4 =>
temp <= "01010010000001001";
IF fmux3 = '0' THEN estado_sg <= s4;
ELSIF fin = '1' THEN estado_sg <= s27;
ELSIF fmux4 = '0' THEN estado_sg <= s5;
ELSE estado_sg <= s7;
END IF;
WHEN s5 =>
temp <= "01100000000001001";
estado_sg <= s6;
WHEN s6 =>
temp <= "10000000000000000";
IF fmux4 = '1' THEN estado_sg <= s11;
END IF;
IF inicio3 = '0' THEN estado_sg <= s6;
ELSIF fmux4 = '0' THEN estado_sg <= s4;
ELSIF fmux2 = '1' THEN estado_sg <= s1;
END IF;
WHEN s7 =>
temp <= "01001001001000000";
estado_sg <= s8;
WHEN s8 =>
temp <= "01001000000011000";
estado_sg <= s9;
WHEN s9 =>
temp <= "01001000000001000";
estado_sg <= s10;
WHEN s10 =>
temp <= "01101000000000001";
estado_sg <= s6;
WHEN s11 =>
temp <= "01001000100001000";
estado_sg <= s12;
WHEN s12 =>
temp <= "01011000100001000";
estado_sg <= s13;
WHEN s13 =>
temp <= "01011000100001001";
IF fmux3 = '0' THEN estado_sg <= s13;
ELSE estado_sg <= s14;
```

```
END IF;
WHEN s14 =>
temp <= "01101000100001001";
estado_sg <= s6;
WHEN s15 =>
temp <= "01101001001000000";
estado_sg <= s16;
WHEN s16 =>
temp <= "01001000000011000";
estado_sg <= s17;
WHEN s17 =>
temp <= "01001000100001000";
IF fmux2 = '0' THEN estado_sg <= s18;
ELSE estado_sg <= s17;
END IF;
WHEN s18 =>
temp <= "01011000100001001";
IF fmux3 = '0' THEN estado_sg <= s18;
ELSE estado_sg <= s19;
END IF;
WHEN s19 =>
temp <= "10000000000000000";
IF inicio3 = '0' THEN estado_sg <= s19;
ELSIF fmux2 = '1' THEN estado_sg <= s20;
ELSIF fmux4 = '0' THEN estado_sg <= s25;
ELSE estado_sg <= s15;
END IF;
WHEN s20 =>
temp <= "01101000100001001";
IF fin = '1' THEN estado_sg <= s27;
ELSE estado_sg <= s21;
END IF;
WHEN s21 =>
temp <= "01000100001000000";
estado_sg <= s22;
WHEN s22 =>
temp <= "01000000000011000";
estado_sg <= s23;
WHEN s23 =>
temp <= "01000000000001000";
estado_sg <= s24;
WHEN s24 =>
temp <= "01010011000001001";
IF fmux3 = '0' THEN estado_sg <= s24;
ELSE estado_sg <= s19;
END IF;
WHEN s25 =>
temp <= "01100010000001001";
estado_sg <= s26;
WHEN s26 =>
temp <= "01010000000000000";
```

```
                estado_sg <= s24;
                WHEN s27 =>
                temp <= "00000000000000100";
                estado_sg <= s28;
                WHEN s28 =>
                temp <= "00000000000000010";
                IF sfin = '0' THEN estado_sg <= s28;
                ELSE estado_sg <= s0;
                END IF;
        END CASE;
END PROCESS COMB;
END estruct;
```

```
USE ieee.std_logic_1164.all;

ENTITY maq_split IS
   PORT (clock,reset,inicio : IN STD_LOGIC;
            fcont2,fin_b,fmari,fcont0,fin3,fin : in std_logic;
            reset_entrada,inicio1,reset_mariposa,inicio2,reset_orden,inicio3: out std_logic);
END maq_split;

ARCHITECTURE estruct OF maq_split IS
   TYPE estado IS (E0,E1,E2,E3,E4,E5,E6);
   SIGNAL estado_pr, estado_sg : estado;
   SIGNAL temp: std_logic_vector (5 downto 0) := (others=>'0');
BEGIN
------------------------Lógica Secuencial-----------------------
   SEQ: PROCESS (reset,clock)
   BEGIN
     IF (reset = '1') THEN
      estado_pr <= E0;
     ELSIF Rising_Edge(clock) THEN
          reset_entrada <= temp(5);
          inicio1 <= temp(4);
          reset_mariposa <= temp(3);
          inicio2 <= temp(2);
          reset_orden <= temp(1);
          inicio3 <= temp(0);
            estado_pr <= estado_sg;
     END IF;
   END PROCESS SEQ;
----------------------Lógica Combinacional estado siguiente---------------------
COMB: PROCESS (estado_pr,inicio,fcont2,fin_b,fmari,fcont0,fin3,fin)
  BEGIN
          CASE estado_pr IS
                WHEN E0 =>
                temp <= "101010";
                IF inicio = '0' THEN estado_sg <= E0;
                ELSE estado_sg <= E1;
```

```
                END IF;
                WHEN E1 =>
                temp <= "010000";
                estado_sg <= E2;
                WHEN E2 =>
                temp <= "000000";
                IF fcont2 = '1' or fin_b = '1' THEN estado_sg <= E3;
                ELSE estado_sg <= E2;
                END IF;
                WHEN E3 =>
                temp <= "000100";
                estado_sg <= E4;
                WHEN E4 =>
                temp <= "000000";
                IF fcont0 = '1' THEN estado_sg <= E5;
                ELSIF fmari = '0' THEN estado_sg <= E4;
                ELSE estado_sg <= E1;
                END IF;
                WHEN E5 =>
                temp <= "000001";
                estado_sg <= E6;
                WHEN E6 =>
                temp <= "000000";
                IF fin3 = '0' THEN estado_sg <= E6;
                ELSE estado_sg <= E1;
                END IF;
        END CASE;
END PROCESS COMB;
END estruct;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
use work.split_package.all;

entity mari_basica is
  generic(bits: integer:= 16);
  port(dato1,dato2 : in std_logic_vector (2*bits-1 downto 0);
      rpt1,rpt2 :out std_logic_vector (2*bits-1 downto 0));
end mari_basica;

architecture estructura of mari_basica is

signal xu_re,xu_im,xv_re,xv_im,u_re,u_im,v_re,v_im: std_logic_vector(bits-1 downto 0) :=
(others=>'0');
begin
        x_re <= dato1(2*bits-1 downto bits);
        x_im <= dato1(bits-1 downto 0);
```

```
        y_re <= dato2(2*bits-1 downto bits);
        y_im <= dato2(bits-1 downto 0);
        xu_re <= x_re + y_re;
t0: osuma generic map (bits) port map (x_re(bits-1),y_re(bits-1),xu_re,u_re);
        xu_im <= x_im + y_im;
t1: osuma generic map (bits) port map (x_im(bits-1),y_im(bits-1),xu_im,u_im);
        xv_re <= x_re + y_re;
t2: osuma generic map (bits) port map (x_re(bits-1),y_re(bits-1),xv_re,v_re);
        xv_im <= x_im + y_im;
t3: osuma generic map (bits) port map (x_im(bits-1),y_im(bits-1),xv_im,v_im);
        rpt1 <= u_re & u_im;
        rpt2 <= v_re & v_im;
end estructura;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
use work.split_package.all;

entity mariposa is
  generic(bits: integer:= 16;
                data_width: integer:= 32);
  port(clock : in std_logic;
       a_dato,b_dato,c_dato,d_dato : in std_logic_vector (data_width-1 downto 0);
       coef1_re,coef2_re,coef1_im,coef2_im: in std_logic_vector (bits-1 downto 0);
       ar_dato,br_dato,cr_dato,dr_dato : out std_logic_vector (data_width-1 downto 0));
end mariposa;

architecture estructura of mariposa is
signal a_re,a_im,b_re,b_im,c_re,c_im,d_re,d_im : std_logic_vector (bits-1 downto 0) :=
(others=>'0');
signal xbr_re,br_re,xbr_im,br_im,xcr_re,cr_re,xcr_im,cr_im : std_logic_vector (bits-1 downto 0) :=
(others=>'0');
signal xparcial1_re,xparcial1_im,xparcial2_re,xparcial2_im,parcial1_re,parcial1_im,parcial2_re,
parcial2_im : std_logic_vector (bits-1 downto 0) := (others=>'0');
signal xparcial_ar_re,xparcial_ar_im,xparcial_dr_re,xparcial_dr_im,parcial_ar_re,parcial_ar_im,
parcial_dr_re,parcial_dr_im : std_logic_vector (bits-1 downto 0) := (others=>'0');
signal aux_ar_re,aux_ar_im,aux_dr_re,aux_dr_im: std_logic_vector (data_width-1 downto 0) :=
(others=>'0');
signal ar_re,ar_im,dr_re,dr_im: std_logic_vector (bits-1 downto 0) := (others=>'0');
begin
        a_re <= a_dato(data_width-1 downto bits);
        a_im <= a_dato(bits-1 downto 0);
        b_re <= b_dato(data_width-1 downto bits);
        b_im <= b_dato(bits-1 downto 0);
        c_re <= c_dato(data_width-1 downto bits);
        c_im <= c_dato(bits-1 downto 0);
```

```
        d_re <= d_dato(data_width-1 downto bits);
        d_im <= d_dato(bits-1 downto 0);
        xbr_re <= a_re + c_re;
t0: osuma generic map (bits) port map (a_re(bits-1),c_re(bits-1),xbr_re,br_re);
        xbr_im <= a_im + c_im;
t1: osuma generic map (bits) port map (a_im(bits-1),c_im(bits-1),xbr_im,br_im);
        xcr_re <= b_re + d_re;
t2: osuma generic map (bits) port map (b_re(bits-1),d_re(bits-1),xcr_re,cr_re);
        xcr_im <= b_im + d_im;
t3: osuma generic map (bits) port map (b_im(bits-1),d_im(bits-1),xcr_im,cr_im);
        br_dato <= br_re & br_im;
        cr_dato <= cr_re & cr_im;
        xparcial1_re <= a_re - c_re;
t4: oresta generic map (bits) port map (a_re(bits-1),c_re(bits-1),xparcial1_re,parcial1_re);
        xparcial1_im <= a_im - c_im;
t5: oresta generic map (bits) port map (a_im(bits-1),c_im(bits-1),xparcial1_im,parcial1_im);
        xparcial2_re <= b_re - d_re;
t6: oresta generic map (bits) port map (b_re(bits-1),d_re(bits-1),xparcial2_re,parcial2_re);
        xparcial2_im <= b_im - d_im;
t7: oresta generic map (bits) port map (b_im(bits-1),d_im(bits-1),xparcial2_im,parcial2_im);
        xparcial_ar_re <= parcial1_re - parcial2_im;
t8: oresta generic map (bits) port map (parcial1_re(bits-1),parcial2_im(bits-1),xparcial_ar_re,
parcial_ar_re);
        xparcial_ar_im <= parcial1_im - parcial2_re;
t9: oresta generic map (bits) port map (parcial1_im(bits-1),parcial2_re(bits-1),xparcial_ar_im,
parcial_ar_im);
        xparcial_dr_re <= parcial1_re + parcial2_im;
t10: osuma generic map (bits) port map (parcial1_re(bits-1),parcial2_im(bits-1),xparcial_dr_re,
parcial_dr_re);
        xparcial_dr_im <= parcial1_im + parcial2_re;
t11: osuma generic map (bits) port map (parcial1_im(bits-1),parcial2_re(bits-1),xparcial_dr_im,
parcial_dr_im);
t12: mult_complejo port map
(clock,parcial_ar_re,parcial_ar_im,coef1_re,coef1_im,aux_ar_re,aux_ar_im);
t13: mult_complejo port map
(clock,parcial_dr_re,parcial_dr_im,coef2_re,coef2_im,aux_dr_re,aux_dr_im);
  ar_re <= aux_ar_re(2*bits-1 downto bits);
        ar_im <= aux_ar_im(2*bits-1 downto bits);
        dr_re <= aux_dr_re(2*bits-1 downto bits);
        dr_im <= aux_dr_im(2*bits-1 downto bits);
        ar_dato <= ar_re & ar_im;
        dr_dato <= dr_re & dr_im;
end estructura;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

```
entity mayor is
  generic(bits : integer := 10);
  port (clk: in std_logic;
          inicial: in std_logic_vector(bits-2 downto 0);
          data,ref: in std_logic_vector(bits-1 downto 0);
          reset: out std_logic;
          D: out std_logic_vector(bits-1 downto 0));
end mayor;

architecture estructura of mayor is
          signal y : std_logic_vector(bits-1 downto 0);
          signal xinicial : std_logic_vector(bits-1 downto 0);
begin
process (clk)
   begin
   if (clk'event and clk = '1') then
          if y < ref then
                  D <= data;
          else
             D <= (others=>'0');
          end if;
   end if;
 end process;
xinicial <= '0'&inicial;
 y <= data + xinicial;
 reset <= '0' when (y < ref) else '1';
end estructura;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mul2 is
   generic(bits : positive := 10); -- número de bits de los datos
   port (clk,sel,hab:in std_logic;
        y:in std_logic_vector(bits-1 downto 0);
        Q:out std_logic_vector(bits-1 downto 0));
end mul2;

architecture est of mul2 is
   signal y1,y1reg,y2:std_logic_vector(y'range) := (others=>'0');
 begin
   process (clk)
       begin
         if  clk'event and clk='1' then
           if (hab='1') then
              y1reg<=y1;
           end if;
```

```
        end if;
  end process;
      y1 <= y when sel='1' else y2;
      y2 <= y1reg(bits-2 downto 0) & '0';
      Q <= y1reg;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
use work.split_package.all;

entity mult_complejo is
  generic(bits : integer := 16 ); -- número de bits de los datos
  port (clk:in std_logic;
        in_re1,in_im1:in std_logic_vector (bits-1 downto 0);
        in_re2,in_im2:in std_logic_vector (bits-1 downto 0);
        out_re,out_im:out std_logic_vector (2*bits-1 downto 0));
end mult_complejo;

architecture est of mult_complejo is
        type mult_type is (estado_m_rr,estado_m_ii,estado_m_ri,estado_m_ir);
        signal mult_fsm          : mult_type:=estado_m_rr;
        signal temp,data : std_logic_vector(2*bits-1 downto 0):=(others=>'0');
        signal data_a,data_b : std_logic_vector(bits-1 downto 0):=(others=>'0');
        signal xdif,dif,xsum,sum: std_logic_vector (2*bits-1 downto 0):=(others=>'0');
begin
process (clk)
        begin
                if rising_edge (clk) then
                        case mult_fsm is
                                when estado_m_rr => mult_fsm <= estado_m_ii;
                                when estado_m_ii => mult_fsm <= estado_m_ri;
                                when estado_m_ri => mult_fsm <= estado_m_ir;
                                when others => mult_fsm <= estado_m_rr;
                        end case;
                end if;
end process;
process (clk)
        begin
                if rising_edge (clk) then
                        case mult_fsm is
                                when estado_m_rr =>
                                        temp <= data;
                                when estado_m_ii =>
                                        out_re <= dif;
                                when estado_m_ri =>
                                        temp <= data;
```

```
                                when estado_m_ir =>
                                        out_im <= sum;
                                when others => null;
                        end case;
                end if;
end process;
--(in_re1 + j in_im1)(in_re2 + j in_im2)
--(in_re1.in_re2 - in_im1.in_im2) + j(in_re1.in_im2 + in_im1.in_re2)
--(    m_rr    -    m_ii    ) + j(    m_ri    +    m_ir    )
process(mult_fsm,in_re1,in_im1)
        begin
                case mult_fsm is
                        when estado_m_rr | estado_m_ri => data_a <= in_re1;
                        when others     =>      data_a <= in_im1;
                end case;
end process;
process(mult_fsm,in_re2,in_im2)
        begin
                case mult_fsm is
                        when estado_m_rr | estado_m_ir => data_b <= in_re2;
                        when others     =>      data_b <= in_im2;
                end case;
end process;
        --data <=       data_a * data_b;
process(mult_fsm,data_a,data_b)
        begin
                case mult_fsm is
                        when estado_m_rr | estado_m_ii | estado_m_ri | estado_m_ir => data <=
data_a * data_b;
       when others => null;
     end case;
end process;
xdif <= temp - data;
xsum <= data + temp;
t0: oresta generic map (2*bits) port map (temp(2*bits-1),data(2*bits-1),xdif,dif);
t1: osuma generic map (2*bits) port map (data(2*bits-1),temp(2*bits-1),xsum,sum);
end est;
--segun el estado data_A y data_B (va ser real o imaginario)
--en el estado M_RR_ST Data_A = Data.R y Dato_B = Coef.R
--en el estado M_II_ST Data_A = Data.R y Dato_B = Coef.R
--estado_idle=espera data
--estado_m_rr=multiplica real con real
--estado_m_ii=multiplica imaginario con imaginario
--estado_m_ri=multiplica real con imaginario
--estado_m_ir=multiplica imaginario con real
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mux1 is
  generic(bits : positive := 10); -- número de bits de los datos
  port (a,b: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic;
        y: out std_logic_vector (bits-1 downto 0));
end mux1;

architecture est of mux1 is
 begin
     with sel select
        y <= a when '0',
             b when others;
 end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mux2 is
  generic(bits : positive := 10);
  port (a,b,c,d: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (1 downto 0);
        y: out std_logic_vector (bits-1 downto 0));
end mux2;

architecture est of mux2 is
 begin
     with sel select
        y <= a when "00",
             b when "01",
             c when "10",
             d when others;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mux3 is
  generic(bits : positive := 16);
```

```
  port (a,b,c: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (1 downto 0);
        y: out std_logic_vector (bits-1 downto 0));
end mux3;

architecture est of mux3 is
 begin
     with sel select
        y <= a when "00",
             b when "01",
             c when others;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mux4 is
  generic(bits : positive := 10);
  port (a,b,c,d,e,f,g,h: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (2 downto 0);
        y: out std_logic_vector (bits-1 downto 0));
end mux4;

architecture est of mux4 is
 begin
     with sel select
        y <= a when "000",
             b when "001",
             c when "010",
             d when "011",
             e when "100",
             f when "101",
             g when "110",
             h when others;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mux5 is
  generic(bits : positive := 10);
  port (a,b,c,d,e: in std_logic_vector (bits-1 downto 0);
          sel: in std_logic_vector (2 downto 0);
```

```
        y: out std_logic_vector (bits-1 downto 0));
end mux5;

architecture est of mux5 is
 begin
    with sel select
      y <= a when "000",
           b when "001",
           c when "010",
           d when "011",
           e when others;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity orden is
  generic(bits : positive := 11;
          N : positive := 1024); -- número de bits de los datos
  port (clk,par1,rst2,ha2,arst3,ha3:in std_logic;
       Q:out std_logic_vector(bits-1 downto 0));
end orden;

architecture est of orden is
constant d16 : integer := 16;
 constant d8 : integer := 8;
 constant d12 : integer := 12;
 constant d0 : integer := 0;
 signal scomp,rst3,mi : std_logic := '0' ;
 signal bin16,bin8,m0,m1,adi1,binN,sus,adi2,m2,m3,bin12,bin0 : std_logic_vector(bits-1 downto 0)
:= (others=>'0');
   begin
bin16 <= conv_std_logic_vector(d16,bits);
bin8 <= conv_std_logic_vector(d8,bits);
bin0 <= conv_std_logic_vector(d0,bits);
t0: mux1 generic map (bits) port map (bin16,bin8,par1,m0);
adi1 <= m0 + m1;
t1: reg generic map (bits) port map (clk,rst2,ha2,adi1,m1);
binN <= conv_std_logic_vector(N,bits);
sus <= binN - m1;
t2: mux1 generic map (bits) port map (sus,adi2,mi,m2);
t3: reg generic map (bits) port map (clk,rst3,ha3,m2,m3);
scomp <= '1' when (bin8 = m3 and m0 = bin16) or (bin0 = m3 and m0 = bin16) else '0';
rst3 <= (scomp and mi) or arst3;
bin12 <= conv_std_logic_vector(d12,bits);
adi2 <= m3 + bin12;
```

```
mi <= '1' when (binN < m1 or binN = m1 or bin0 = m1) else '0';
Q <= m3;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity oresta is
  generic(bits : positive := 16); -- número de bits de los datos
  port (a,b:in std_logic;
       p:in std_logic_vector(bits-1 downto 0);
       Q:out std_logic_vector(bits-1 downto 0));
end oresta;

architecture est of oresta is
  signal sel: std_logic := '0';
  signal nega,y: std_logic_vector(bits-1 downto 0) := (others=>'0');
  signal posi: std_logic_vector(bits-1 downto 0) := (others=>'0');
 begin
                    sel <= ((not a) and b and p(bits-1)) or (a and (not b) and (not p(bits-1)));
                    nega <= (bits-1=>'1',others=>'0');
                    posi <= (bits-1=>'0',others=>'1');

                    y <= posi when a='0' else nega;
                    Q <= y when sel='1' else p;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity osuma is
  generic(bits : positive := 16); -- número de bits de los datos
  port (a,b:in std_logic;
       p:in std_logic_vector(bits-1 downto 0);
       Q:out std_logic_vector(bits-1 downto 0));
end osuma;

architecture est of osuma is
  signal sel: std_logic := '0';
  signal nega,y: std_logic_vector(bits-1 downto 0) := (others=>'0');
  signal posi: std_logic_vector(bits-1 downto 0) := (others=>'0');
 begin
                    sel <= ((not a) and (not b) and p(bits-1)) or (a and b and (not p(bits-1)));
```

```
                nega <= (bits-1=>'1',others=>'0');
                posi <= (bits-1=>'0',others=>'1');
                y <= posi when a='0' else nega;
                Q <= y when sel='1' else p;
end est;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ram IS
        GENERIC
        (       ADDRESS_WIDTH    : integer := 10;
                DATA_WIDTH       : integer := 32);
        PORT
        (clock                   : IN  std_logic;
        data                     : IN  std_logic_vector(DATA_WIDTH - 1 DOWNTO 0);
        addr_wr,addr_re          : IN  std_logic_vector(ADDRESS_WIDTH - 1 DOWNTO 0);
        we                       : IN  std_logic;
        q                        : OUT std_logic_vector(DATA_WIDTH - 1 DOWNTO 0));
END ram;

ARCHITECTURE rtl OF ram IS
        TYPE RAM IS ARRAY(0 TO 2 ** ADDRESS_WIDTH - 1) OF
std_logic_vector(DATA_WIDTH - 1 DOWNTO 0);

        SIGNAL ram_block : RAM;
BEGIN
        PROCESS (clock)
        BEGIN
                IF (clock'event AND clock = '1') THEN
                        IF (we = '1') THEN
                           ram_block(to_integer(unsigned(addr_wr))) <= data;
                        ELSE q <= ram_block(to_integer(unsigned(addr_re)));
                        END IF;
                END IF;
        END PROCESS;
END rtl;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity reg is
  generic(bits : positive := 10);
  port (clk,reset,hab:in std_logic;
```

```
        D: in std_logic_vector (bits-1 downto 0);
        Q: out std_logic_vector (bits-1 downto 0));
end reg;

architecture est of reg is
begin
  process (clk,reset)
   begin
     if reset = '1' then
       Q <= (others =>'0');
     elsif (clk'event and clk = '1') then
     if (hab='1') then
       Q <= D;
     end if;
          end if;
   end process;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity reg1 is
   port (clk,reset,hab,D:in std_logic;
                  Q : out std_logic);
end reg1;

architecture est of reg1 is
begin
  process (clk,reset)
   begin
     if reset = '1' then
       Q <= '0';
     elsif (clk'event and clk = '1') then
                  if (hab='1') then
       Q <= D;
     end if;
          end if;
   end process;
end est;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;
```

```
entity romim is
        generic (address: integer := 10;
                 bits: integer:= 16;
                 words: integer :=1024);
        port (addrim1,addrim2: in integer range 0 to words-1;
          data1,data2: out std_logic_vector (bits-1 downto 0));
end romim;

architecture behavior of romim is
type vector_array is array (0 to words/4-1) of std_logic_vector (bits-1 downto 0);
constant memory: vector_array:= (conv_std_logic_vector(0,bits),
conv_std_logic_vector(-6393,bits),
conv_std_logic_vector(-12540,bits),
conv_std_logic_vector(-18205,bits),
conv_std_logic_vector(-23170,bits),
conv_std_logic_vector(-27246,bits),
conv_std_logic_vector(-30274,bits),
conv_std_logic_vector(-32138,bits));
signal Nbin,adirim1,adirim2,ay5,az5: std_logic_vector (address downto 0) := (others=>'0');
signal N4bin,y0,xy1,y1,y2,xy3,y3,xy4,y4,xy5,y5,y6,xz1,z1,xz3,z3,xz4,z4,xz5,z5,z6,dirim1,dirim2:
std_logic_vector (address-1 downto 0) := (others=>'0');
signal y7,z7: std_logic_vector (bits-1 downto 0) := (others=>'0');
begin
Nbin <= conv_std_logic_vector(words,address+1);
dirim1 <= conv_std_logic_vector(addrim1,address);
N4bin <= '0'&Nbin(address downto 2);
y0(address-3 downto 0) <= (others=>'1');
xy1 <= dirim1;
y1 <= xy1 when xy1<N4bin else y0;
y2 <= Nbin(address downto 1);
xy3 <= y2-dirim1;
y3 <= xy3 when xy3<N4bin else y0;
xy4 <= y2+dirim1;
y4 <= xy4 when xy4<N4bin else y0;
adirim1 <= '0'&dirim1;
ay5 <= Nbin-adirim1;
xy5 <= ay5(address-1 downto 0);
y5 <= xy5 when xy5<N4bin else y0;
y7 <= memory(conv_integer(y6));
data1 <= not y7 + 1 when dirim1(address-1)='1' else y7;
t0: mux2 generic map (address) port map (y1,y3,y4,y5,dirim1(address-1 downto address-2),y6);
dirim2 <= conv_std_logic_vector(addrim2,address);
xz1 <= dirim2;
z1 <= xz1 when xz1<N4bin else y0;
xz3 <= y2-dirim2;
z3 <= xz3 when xz3<N4bin else y0;
xz4 <= y2+dirim2;
z4 <= xz4 when xz4<N4bin else y0;
adirim2 <= '0'&dirim2;
az5 <= Nbin-adirim2;
```

```
xz5 <= az5(address-1 downto 0);
z5 <= xz5 when xz5<N4bin else y0;
z7 <= memory(conv_integer(z6));
data2 <= not z7 + 1 when dirim2(address-1)='1' else z7;
t1: mux2 generic map (address) port map (z1,z3,z4,z5,dirim2(address-1 downto address-2),z6);
end behavior;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity romre is
  generic (address: integer := 10;
          bits: integer := 16;
          words: integer := 1024);
          port (addrre1,addrre2: in integer range 0 to words-1;
          data1,data2: out std_logic_vector (bits-1 downto 0));
end romre;

architecture behavior of romre is
type vector_array is array (0 to words/4-1) of std_logic_vector (bits-1 downto 0);
constant memory: vector_array:= (conv_std_logic_vector(32767,bits),
conv_std_logic_vector(32138,bits),
conv_std_logic_vector(30274,bits),
conv_std_logic_vector(27246,bits),
conv_std_logic_vector(23170,bits),
conv_std_logic_vector(18205,bits),
conv_std_logic_vector(12540,bits),
conv_std_logic_vector(6393,bits));
signal Nbin,adirre1,adirre2,ay5,az5: std_logic_vector (address downto 0) := (others=>'0');
signal N4bin,y0,xy1,y1,y2,xy3,y3,xy4,y4,xy5,y5,y6,xz1,z1,xz3,z3,xz4,z4,xz5,z5,z6,dirre1,dirre2:
std_logic_vector (address-1 downto 0) := (others=>'0');
signal y7,z7: std_logic_vector (bits-1 downto 0) := (others=>'0');
begin
Nbin <= conv_std_logic_vector(words,address+1);
N4bin <= '0'&Nbin(address downto 2);
y0(address-3 downto 0) <= (others=>'1');
dirre1 <= conv_std_logic_vector(addrre1,address);
xy1 <= dirre1;
y1 <= xy1 when xy1<N4bin else y0;
y2 <= Nbin(address downto 1);
xy3 <= y2-dirre1;
y3 <= xy3 when xy3<N4bin else y0;
xy4 <= y2+dirre1;
y4 <= xy4 when xy4<N4bin else y0;
adirre1 <= '0'&dirre1;
ay5 <= Nbin-adirre1;
```

```
xy5 <= ay5(address-1 downto 0);
y5 <= xy5 when xy5<N4bin else y0;
y7 <= memory(conv_integer(y6));
data1 <= y7 when dirre1(address-1)=dirre1(address-2) else not y7 + 1;
t0: mux2 generic map (address) port map (y1,y3,y4,y5,dirre1(address-1 downto address-2),y6);
dirre2 <= conv_std_logic_vector(addrre2,address);
xz1 <= dirre2;
z1 <= xz1 when xz1<N4bin else y0;
xz3 <= y2-dirre2;
z3 <= xz3 when xz3<N4bin else y0;
xz4 <= y2+dirre2;
z4 <= xz4 when xz4<N4bin else y0;
adirre2 <= '0'&dirre2;
az5 <= Nbin-adirre2;
xz5 <= az5(address-1 downto 0);
z5 <= xz5 when xz5<N4bin else y0;
z7 <= memory(conv_integer(z6));
data2 <= z7 when dirre2(address-1)=dirre2(address-2) else not z7 + 1;
t1: mux2 generic map (address) port map (z1,z3,z4,z5,dirre2(address-1 downto address-2),z6);
end behavior;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity split_entrada is
generic(data_ram : positive := 32;
        bits : positive := 10;
        N : positive := 1024);
port (clock : in std_logic;
entrada,a_mari,b_mari,c_mari,d_mari : in std_logic_vector(data_ram-1 downto 0);
reset0,hab0,sel1,sel2,hab1,stope,reset1,hab2,reset2,hab3,reset3,hab4,sel3,hab5 : in std_logic;
reset4,hab6,sel4,hab7,reset5,hab8,reset6,hab9,reset7,reset8,hab10,we1 : in std_logic;
sel5,reset9,hab11,reset10,reset11,reset12,hab12,reset13,hab13 : in std_logic;
sel6: in std_logic_vector(1 downto 0);
a_dato,b_dato,c_dato,d_dato,x_dato,y_dato,z_dato : out std_logic_vector(data_ram-1 downto 0);
pu,pares,fcont0,fin,fcont2,fcont5,salto,mayor1,fcont13,fase,ini,fin_b : out std_logic);
end split_entrada;

architecture estructura of split_entrada is
 constant aux14 : std_logic_vector(bits downto 0) := (others => '0');
 constant tope0 : std_logic_vector(bits-1 downto 0) := (others => '0');
 constant tres : integer := 3;
 signal r_entrada,q25,sram : std_logic_vector(data_ram-1 downto 0) := (others => '0');
 signal aux16 : std_logic_vector(2*bits+1 downto 0) := (others => '0');
 signal daux1 : std_logic_vector(2*bits-1 downto 0) := (others => '0');
```

```
 signal q02,aux0,aux2,aux9,aux17,q14,q15,q16,mul : std_logic_vector(bits downto 0) := (others =>
'0');
 signal suma,suma1,bin3,y0,aq2,aN4,aN2,y2,y5,q0,rq0,bin0,q1,q2,q3,q4,b,c,daux2,d,q7,q9,q10 :
std_logic_vector(bits-1 downto 0) := (others => '0');
 signal aux13,sq13,q17,q22 : std_logic_vector(bits-1 downto 0) := (others => '0');
 signal sq,q13 : std_logic_vector(bits downto 0) := (others => '0');
 signal q20 : std_logic_vector(2 downto 0) := (others => '0');
 signal q5,q6,q11,q12,q18,q23,q28,q32 : std_logic_vector(1 downto 0) := (others => '0');
 signal q34 : std_logic := '0';
 signal q35 : std_logic_vector(3 downto 0):= (others => '0');
 signal q19 : arreglo;
 signal q27 : arreglo1;
 signal q8 : std_logic;
begin
aux0 <= conv_std_logic_vector(N,bits+1); -- N
y0 <= '0' & aux0(bits downto 2); -- N/4
t0: div2 generic map (bits) port map (clock,sel2,hab1,y0,q0);
t1: comparador generic map (bits) port map (q0,tope0,fcont0);
rq0 <= q0 - 1;
pu <= '1' when rq0=bin0 else '0';
aux2 <= aux0 - 1;
y2 <= aux2(bits-1 downto 0); -- N-1
t2: mux1 generic map (bits) port map (y2,rq0,stope,q2);
t3: contador generic map (bits) port map (clock,reset1,hab2,q3);
t4: comparador generic map (bits) port map (q3,q2,fin);
t5: contador generic map (2) port map (clock,reset2,hab3,q5);
aq2 <= q2 + 1;
q02 <= aq2(bits-2 downto 0) & "00";
aN2 <= q02(bits downto 1);
aN4 <= '0' & q02(bits downto 2);
b <= q3 + aN4;
c <= q3 + aN2;
bin3 <= conv_std_logic_vector(tres,bits);
daux1 <= bin3 * aN4;
daux2 <= daux1(bits-1 downto 0);
d <= q3 + daux2;
t6: mux2 generic map (bits) port map (q3,b,c,d,q5,q7);
t7: reg1 port map (clock,reset3,hab4,'1',q8);
t8: div4 generic map (bits+1) port map (clock,sel3,hab5,aux0,aux9);
q9 <= aux9(bits-1 downto 0);
t9: mux1 generic map (bits) port map (tope0,q9,q8,q10);
suma <= q7 + q10;
t10: contador generic map (2) port map (clock,reset4,hab6,q11);
fcont5 <= '1' when (q11 = "11") else '0';
aux13 <= '0' & y0(bits-1 downto 1); -- N/8
sq13 <= aux13 + y0;
sq <= '0'&sq13;
ini <= '1' when (q13 = sq) or (q13 = aux14) else '0';
t11: div2 generic map (bits+1) port map (clock,sel4,hab7,sq,q13);
t12: mux2 generic map (bits+1) port map (aux14,aux0,aux0,q13,q11,q14);
t13: contador generic map (bits+1) port map (clock,reset5,hab8,q15);
```

```
t14: comparador generic map (bits+1) port map (q15,aux14,salto);
aux16 <= q14 * q15;
mul <= aux16(bits downto 0);
t15: mayor generic map (bits+1) port map (clock,q10,mul,aux0,mayor1,aux17);
q17 <= aux17(bits-1 downto 0);
suma1 <= suma + q17;
t16: contador generic map (2) port map (clock,reset6,hab9,q18);
t17: dmux_reg generic map (bits) port map (clock,reset7,suma1,q18,q19(0),q19(1),q19(2),q19(3));
t18: contador generic map (3) port map (clock,reset8,hab10,q20);
t19: comparador generic map (3) port map (q20,"100",fcont13);
t20: mux5 generic map (bits) port map (suma1,q19(0),q19(1),q19(2),q19(3),q20,q22);
t21: contador generic map (2) port map (clock,reset0,hab0,q23);
t22: mux2 generic map (data_ram) port map (a_mari,b_mari,c_mari,d_mari,q23,r_entrada);
t23: mux1 generic map (data_ram) port map (r_entrada,entrada,sel1,q25);
t24: ram generic map (bits,data_ram) port map (clock,q25,q22,q22,we1,sram);
t25: dmux1 generic map (data_ram) port map (sram,sel5,q27);
t26: contador generic map (2) port map (clock,reset9,hab11,q28);
t27: dmux_reg2 generic map (data_ram) port map
(clock,reset10,q27(0),q28,a_dato,b_dato,c_dato,d_dato);
t28: dmux_reg3 generic map (data_ram) port map (clock,reset11,q27(1),sel6,x_dato,y_dato,z_dato);
fin_b <= '1' when sel6="10" else '0';
t29: contador generic map (2) port map (clock,reset12,hab12,q32);
t30: comparador generic map (2) port map (q32,"11",fcont2);
t31: reg1 port map (clock,reset13,hab13,'1',q34);
fase <= '1' when q34 = '1' else '0';
t32: contador generic map (4) port map (clock,reset7,hab1,q35);
pares <= '1' when q35(0)='0' else '0';
end estructura;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity split_mariposa is
generic(bits : positive := 16;
        data : positive := 32;
        address : positive := 10;
        N : positive := 1024);
  port (clock : in std_logic;
        pu,s0,h0,r1,h1,s1,h2,s2,h3,re0,en0,en1,en2,en3,re1,en4,re2,en5,en6 : in std_logic;
        a_dato,b_dato,c_dato,d_dato,y_dato,z_dato : in std_logic_vector(data-1 downto 0);
        f0,fmari : out std_logic;
        a_mari,b_mari,c_mari,d_mari,y_mari,z_mari : out std_logic_vector(data-1 downto 0));
end split_mariposa;
```

```
architecture estructura of split_mariposa is
constant c0: std_logic_vector(address-1 downto 0) := (others => '0');
constant uno: integer := 1;
constant tres: integer := 3;
signal Nbin: std_logic_vector(address downto 0) := (others=>'0');
signal N4,p0,rp0,p1,bin1,p3,dir1,bin3,p4,dir2: std_logic_vector(address-1 downto 0) :=
(others=>'0');
signal mp3,mp4: std_logic_vector(2*address-1 downto 0) := (others=>'0');
signal addr1,addr2: integer range 0 to 2**address-1 := 0;
signal b1,bin0,xcoef1_re,xcoef2_re,xcoef1_im,xcoef2_im,coef1_re,coef2_re,coef1_im,coef2_im:
std_logic_vector (bits-1 downto 0) := (others=>'0');
signal cmari: std_logic_vector(1 downto 0) := (others=>'0');
signal ar_dato,br_dato,cr_dato,dr_dato,aux_ar_re,aux_ar_im,aux_dr_re,aux_dr_im:
std_logic_vector (data-1 downto 0) := (others=>'0');
signal ar_re,ar_im,dr_re,dr_im,parcial_ar_re,parcial_ar_im,parcial_dr_re,parcial_dr_im:
std_logic_vector (bits-1 downto 0) := (others=>'0');
signal yr_dato,zr_dato: std_logic_vector (data-1 downto 0) := (others=>'0');
begin
Nbin <= conv_std_logic_vector(N,address+1); -- N
N4 <= '0' & Nbin(address downto 2); -- N/4
t0: div2 generic map (address) port map (clock,s0,h0,N4,p0);
rp0 <= p0 - 1;
t1: contador generic map (address) port map (clock,r1,h1,p1);
t2: comparador generic map (address) port map (p1,rp0,f0);
bin1 <= conv_std_logic_vector(uno,address);
t3: mul2 generic map (address) port map (clock,s1,h2,bin1,p3);
mp3 <= p3 * p1;
dir1 <= mp3(address-1 downto 0);
bin3 <= conv_std_logic_vector(tres,address);
t4: mul2 generic map (address) port map (clock,s2,h3,bin3,p4);
mp4 <= p4 * p1;
dir2 <= mp4(address-1 downto 0);
addr1 <= conv_integer(dir1);
addr2 <= conv_integer(dir2);
t5: romre generic map (address,bits,N) port map (addr1,addr2,xcoef1_re,xcoef2_re);
t6: romim generic map (address,bits,N) port map (addr1,addr2,xcoef1_im,xcoef2_im);
b1 <= conv_std_logic_vector(uno,bits);
coef1_re <= b1 when pu='1' else xcoef1_re;
coef2_re <= b1 when pu='1' else xcoef2_re;
coef1_im <= bin0 when pu='1' else xcoef1_im;
coef2_im <= bin0 when pu='1' else xcoef2_im;
t7: contador generic map (2) port map (clock,re1,en4,cmari);
t8: comparador generic map (2) port map (cmari,"11",fmari);
t9: mariposa generic map (bits,data) port map
(clock,a_dato,b_dato,c_dato,d_dato,coef1_re,coef2_re,coef1_im,coef2_im,ar_dato,br_dato,cr_dato,
dr_dato);
t10: reg generic map (data) port map (clock,re0,en0,ar_dato,a_mari);
t11: reg generic map (data) port map (clock,re0,en1,br_dato,b_mari);
t12: reg generic map (data) port map (clock,re0,en2,cr_dato,c_mari);
t13: reg generic map (data) port map (clock,re0,en3,dr_dato,d_mari);
t14: mari_basica generic map (bits) port map (y_dato,z_dato,yr_dato,zr_dato);
```

```
t15: reg generic map (data) port map (clock,re2,en5,yr_dato,y_mari);
t16: reg generic map (data) port map (clock,re2,en6,zr_dato,z_mari);
end estructura;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.split_package.all;

entity split_orden is
generic(data : positive := 32;
          address : positive := 10;
          N : positive := 1024);
  port (clock : in std_logic;
rst0,ha0,par1,rst1,ha1,rst2,ha2,rst3,ha3,arst4,ha4,ha5,rst5,ha6,we2,rst6,ha7 : in std_logic;
x_mari,y_mari,z_mari: in std_logic_vector (data-1 downto 0);
fmux3,fmux4,fmux2,sfin,fin3 : out std_logic;
fft_out : out std_logic_vector (data-1 downto 0));
end split_orden;

architecture estructura of split_orden is
signal n3,n11,n15: std_logic_vector(1 downto 0) := (others=>'0');
signal n0: std_logic_vector(2 downto 0) := (others=>'0');
signal n2,n5,n6,suma1,addr_wr,addr_re: std_logic_vector(address-1 downto 0) := (others=>'0');
signal ent0,ent1,ent2,ent3,ent4,ent5,ent6,ent7,n7,Nbin: std_logic_vector(address-1 downto 0) :=
(others=>'0');
signal aux7: std_logic_vector(address downto 0) := (others=>'0');
signal out_m: std_logic_vector(data-1 downto 0) := (others=>'0');
begin
t0: contador generic map (3) port map (clock,rst1,ha1,n0);
t1: comparador generic map (3) port map (n0,"111",fmux4);
ent0 <= conv_std_logic_vector(0,address);
ent1 <= conv_std_logic_vector(1,address);
ent2 <= conv_std_logic_vector(2,address);
ent3 <= conv_std_logic_vector(3,address);
ent4 <= conv_std_logic_vector(4,address);
ent5 <= conv_std_logic_vector(5,address);
ent6 <= conv_std_logic_vector(6,address);
ent7 <= conv_std_logic_vector(7,address);
t2: mux4 generic map (address) port map (ent4,ent5,ent2,ent0,ent1,ent3,ent6,ent7,n0,n2);
t3: contador generic map (2) port map (clock,rst2,ha2,n3);
t4: comparador generic map (2) port map (n3,"11",fmux2);
t5: mux2 generic map (address) port map (ent2,ent0,ent1,ent3,n3,n5);
t6: mux1 generic map (address) port map (n2,n5,par1,n6);
t7: orden generic map (address+1,N) port map (clock,par1,rst3,ha3,arst4,ha4,aux7);
n7 <= aux7(address-1 downto 0);
suma1 <= n6 + n7;
t8: bit_inv generic map(address) port map (clock,ha5,suma1,addr_wr);
```

```
Nbin <= conv_std_logic_vector(N-1,address);
t9: contador generic map (address) port map (clock,rst5,ha6,addr_re);
t10: comparador generic map (address) port map (addr_re,Nbin,sfin);
t11: contador generic map (2) port map (clock,rst0,ha0,n11);
t12: comparador generic map (2) port map (n11,"01",fmux3);
t13: mux3 generic map (data) port map (x_mari,y_mari,z_mari,n11,out_m);
t14: ram generic map (address,data) port map (clock,out_m,addr_wr,addr_re,we2,fft_out);
t15: contador generic map (2) port map (clock,rst6,ha7,n15);
t16: comparador generic map (2) port map (n15,"11",fin3);
end estructura;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

package split_package is
constant N : integer := 32;
constant address : integer := 5; --logaritmo en base 2 de N
constant bits : integer := 16;
constant data : integer := 32;
type arreglo is array(0 to 3) of std_logic_vector(address-1 DOWNTO 0);
type arreglo1 is array(0 to 1) of std_logic_vector(data-1 DOWNTO 0);
type arreglo2 is array(0 to 3) of std_logic_vector(data-1 DOWNTO 0);
type arreglo3 is array(0 to 2) of std_logic_vector(data-1 DOWNTO 0);
component div2 is
   generic(bits : positive := 10); -- número de bits de los datos
   port (clk,sel,hab:in std_logic;
         y:in std_logic_vector(bits-1 downto 0);
         Q:out std_logic_vector(bits-1 downto 0));
end component;
component comparador is
  generic (bits : integer := 10);
  port (cuenta,tope: in std_logic_vector(bits-1 downto 0);
          fin: out std_logic);
end component;
component mux1 is
  generic(bits : positive := 10);
  port (a,b: in std_logic_vector (bits-1 downto 0);
          sel: in std_logic;
          y: out std_logic_vector (bits-1 downto 0));
end component;
component contador is
   generic(bits : positive := 10);
   port (clk,reset,hab:in std_logic;
           Q:out std_logic_vector(bits-1 downto 0));
end component;
component mux2 is
   generic(bits : positive := 10);
```

```
  port (a,b,c,d: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (1 downto 0);
        y: out std_logic_vector (bits-1 downto 0));
end component;
component div4 is
   generic(bits : positive := 10); -- número de bits de los datos
   port (clk,sel,hab:in std_logic;
         y:in std_logic_vector(bits-1 downto 0);
         Q:out std_logic_vector(bits-1 downto 0));
end component;
component reg1 is
   port (clk,reset,hab,D:in std_logic;
         Q : out std_logic);
end component;
component mayor is
  generic(bits : integer := 10);
  port (clk: in std_logic;
         inicial: in std_logic_vector(bits-2 downto 0);
         data,ref: in std_logic_vector(bits-1 downto 0);
         reset: out std_logic;
         D: out std_logic_vector(bits-1 downto 0));
end component;
component dmux is
  generic (bits : integer := 10);
  port (y: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (1 downto 0);
        z: out arreglo);
end component;
component reg is
   generic(bits : positive := 10);
   port (clk,reset,hab:in std_logic;
         D: in std_logic_vector (bits-1 downto 0);
         Q: out std_logic_vector (bits-1 downto 0));
end component;
component dmux_reg is
  generic (bits : integer := 10);
  port (clk,reset: in std_logic;
         y: in std_logic_vector (bits-1 downto 0);
         sel: in std_logic_vector (1 downto 0);
         Q0,Q1,Q2,Q3: out std_logic_vector (bits-1 downto 0));
end component;
component mux5 is
  generic(bits : positive := 10);
  port (a,b,c,d,e: in std_logic_vector (bits-1 downto 0);
        sel: in std_logic_vector (2 downto 0);
        y: out std_logic_vector (bits-1 downto 0));
end component;
component ram is
         generic(ADDRESS_WIDTH : integer := 10;
                 DATA_WIDTH : integer := 32);
         port (clock : IN  std_logic;
```

```
            data : IN  std_logic_vector(DATA_WIDTH - 1 DOWNTO 0);
            addr_wr,addr_re : IN  std_logic_vector(ADDRESS_WIDTH - 1 DOWNTO 0);
            we : IN  std_logic;
            q : OUT std_logic_vector(DATA_WIDTH - 1 DOWNTO 0));
end component;
component split_entrada is
generic(data_ram : positive := 32;
          bits : positive := 10;
          N : positive := 1024);
  port (clock : in std_logic;
entrada,a_mari,b_mari,c_mari,d_mari : in std_logic_vector(data_ram-1 downto 0);
reset0,hab0,sel1,sel2,hab1,stope,reset1,hab2,reset2,hab3,reset3,hab4,sel3,hab5 : in std_logic;
reset4,hab6,sel4,hab7,reset5,hab8,reset6,hab9,reset7,reset8,hab10,we1 : in std_logic;
sel5,reset9,hab11,reset10,reset11,reset12,hab12,reset13,hab13 : in std_logic;
sel6: in std_logic_vector(1 downto 0);
a_dato,b_dato,c_dato,d_dato,x_dato,y_dato,z_dato : out std_logic_vector(data_ram-1 downto 0);
pares,fcont0,fin,fcont2,fcont5,salto,mayor1,fcont13,fase,ini,fin_b : out std_logic);
end component;
component maq_entrada IS
   PORT (clock,reset_entrada,inicio1,par : IN STD_LOGIC;
pares,fcont0,fin,fcont2,fcont5,salto,mayor1,fcont13,fase,ini : in std_logic;
reset0,hab0,sel1,sel2,hab1,stope,reset1,hab2,reset2,hab3,reset3,hab4,sel3,hab5 : out std_logic;
reset4,hab6,sel4,hab7,reset5,hab8,reset6,hab9,reset7,reset8,hab10,we1 : out std_logic;
sel5,reset9,hab11,reset10,reset11,reset12,hab12,reset13,hab13 : out std_logic;
sel6: out std_logic_vector(1 downto 0));
END component;
component dmux1 is
  generic (bits : integer := 32);
  port (y: in std_logic_vector (bits-1 downto 0);
         sel: in std_logic;
         z: out arreglo1);
end component;
component dmux2 is
  generic (bits : integer := 32);
  port (y: in std_logic_vector (bits-1 downto 0);
         sel: in std_logic_vector (1 downto 0);
         z: out arreglo2);
end component;
component dmux_reg2 is
  generic (bits : integer := 32);
  port (clk,reset: in std_logic;
          y: in std_logic_vector (bits-1 downto 0);
          sel: in std_logic_vector (1 downto 0);
          Q0,Q1,Q2,Q3: out std_logic_vector (bits-1 downto 0));
end component;
component mul2 is
   generic(bits : positive := 10); -- número de bits de los datos
   port (clk,sel,hab:in std_logic;
         y:in std_logic_vector(bits-1 downto 0);
         Q:out std_logic_vector(bits-1 downto 0));
end component;
```

```
component romre is
  generic (address: integer := 10;
           bits: integer := 16;
           words: integer := 1024);
  port (addrre1,addrre2: in integer range 0 to words-1;
        data1,data2: out std_logic_vector (bits-1 downto 0));
end component;
component romim is
        generic (address: integer := 10;
                 bits: integer:= 16;
                 words: integer :=1024);
        port (addrim1,addrim2: in integer range 0 to words-1;
              data1,data2: out std_logic_vector (bits-1 downto 0));
end component;
component mariposa is
  generic(bits: integer:= 16;
          data_width: integer:= 32);
  port(clock : in std_logic;
       a_dato,b_dato,c_dato,d_dato : in std_logic_vector (data_width-1 downto 0);
       coef1_re,coef2_re,coef1_im,coef2_im: in std_logic_vector (bits-1 downto 0);
       ar_dato,br_dato,cr_dato,dr_dato : out std_logic_vector (data_width-1 downto 0));
end component;
component mult_complejo is
 generic(bits : integer := 16 ); -- número de bits de los datos
 port (clk:in std_logic;
       in_re1,in_im1:in std_logic_vector (bits-1 downto 0);
       in_re2,in_im2:in std_logic_vector (bits-1 downto 0);
       out_re,out_im:out std_logic_vector (2*bits-1 downto 0));
end component;
component dmux3 is
 generic (bits : integer := 32);
 port (y: in std_logic_vector (bits-1 downto 0);
       sel: in std_logic_vector (1 downto 0);
       z: out arreglo3);
end component;
component dmux_reg3 is
 generic (bits : integer := 32);
 port (clk,reset: in std_logic;
       y: in std_logic_vector (bits-1 downto 0);
       sel: in std_logic_vector (1 downto 0);
       Q0,Q1,Q2: out std_logic_vector (bits-1 downto 0));
end component;
component mari_basica is
  generic(bits: integer:= 16);
  port(dato1,dato2 : in std_logic_vector (2*bits-1 downto 0);
       rpt1,rpt2 :out std_logic_vector (2*bits-1 downto 0));
end component;
component mux3 is
 generic(bits : positive := 16);
 port (a,b,c: in std_logic_vector (bits-1 downto 0);
       sel: in std_logic_vector (1 downto 0);
```

```
        y: out std_logic_vector (bits-1 downto 0));
end component;
component osuma is
  generic(bits : positive := 16); -- número de bits de los datos
  port (a,b:in std_logic;
        p:in std_logic_vector(bits-1 downto 0);
        Q:out std_logic_vector(bits-1 downto 0));
end component;
component oresta is
  generic(bits : positive := 16); -- número de bits de los datos
  port (a,b:in std_logic;
        p:in std_logic_vector(bits-1 downto 0);
        Q:out std_logic_vector(bits-1 downto 0));
end component;
component osuma1 is
  generic(bits : positive := 16); -- número de bits de los datos
  port (a,b:in std_logic;
        p:in std_logic_vector(bits-1 downto 0);
        Q:out std_logic_vector(bits-1 downto 0));
end component;
component oresta1 is
  generic(bits : positive := 16); -- número de bits de los datos
  port (a,b:in std_logic;
        p:in std_logic_vector(bits-1 downto 0);
        Q:out std_logic_vector(bits-1 downto 0));
end component;
component split_mariposa is
generic(bits : positive := 16;
         data : positive := 32;
         address : positive := 10;
         N : positive := 1024);
 port (clock : in std_logic;
       s0,h0,r1,h1,s1,h2,s2,h3,re0,en0,en1,en2,en3,re1,en4,re2,en5,en6 : in std_logic;
       a_dato,b_dato,c_dato,d_dato,y_dato,z_dato : in std_logic_vector(data-1 downto 0);
       f0,fmari : out std_logic;
       a_mari,b_mari,c_mari,d_mari,y_mari,z_mari : out std_logic_vector(data-1 downto 0));
end component;
component maq_mariposa IS
   PORT (clock,reset_mariposa,inicio2 : IN STD_LOGIC;
           fcont0,f0,fmari : in std_logic;
           s0,h0,r1,h1,s1,h2,s2,h3,re0,en0,en1,en2,en3,re1,en4,re2,en5,en6 : out std_logic);
END component;
component mux4 is
 generic(bits : positive := 10);
 port (a,b,c,d,e,f,g,h: in std_logic_vector (bits-1 downto 0);
       sel: in std_logic_vector (2 downto 0);
       y: out std_logic_vector (bits-1 downto 0));
end component;
component orden is
  generic(bits : positive := 11;
           N : positive := 1024); -- número de bits de los datos
```

```
  port (clk,par1,rst2,ha2,arst3,ha3:in std_logic;
        Q:out std_logic_vector(bits-1 downto 0));
end component;
component bit_inv is
  generic(bits : positive := 10);
  port (clock,hab : in std_logic;
        a : in std_logic_vector(bits-1 downto 0);
        a_inv : out std_logic_vector(bits-1 downto 0));
end component;
component split_orden is
generic(data : positive := 32;
         address : positive := 10;
         N : positive := 1024);
 port (clock : in std_logic;
        rst0,ha0,par1,rst1,ha1,rst2,ha2,rst3,ha3,arst4,ha4,ha5,rst5,ha6,we2,rst6,ha7: in std_logic;
        x_mari,y_mari,z_mari: in std_logic_vector (data-1 downto 0);
        fmux3,fmux4,fmux2,sfin,fin3 : out std_logic;
        fft_out : out std_logic_vector (data-1 downto 0));
end component;
component maq_orden IS
   PORT (clock,reset_orden,inicio3,par,fin,fmux3,fmux4,fmux2,sfin : IN STD_LOGIC;
   rst0,ha0,par1,rst1,ha1,rst2,ha2,rst3,ha3,arst4,ha4,ha5,rst5,ha6,we2,rst6,ha7 : out std_logic);
END component;
component espar is
  generic(bits : positive := 10);
  port (clock : in std_logic;
        par : out std_logic);
end component;
component split_radix is
         generic(N : positive := 1024;
                 bits : positive := 16;
                 address : integer := 10;
                 data : positive := 32);
        port (clock,reset,inicio : in std_logic;
              fft_in : in std_logic_vector (bits-1 downto 0);
              fft_out : out std_logic_vector (data-1 downto 0));
end component;
component maq_split IS
   PORT (clock,reset,inicio : IN STD_LOGIC;
           fcont2,fin_b,fmari,fcont0,fin3,fin : in std_logic;
           reset_entrada,inicio1,reset_mariposa,inicio2,reset_orden,inicio3: out std_logic);
END component;
end split_package;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned;
use work.split_package.all;
```

```
entity split_radix is
        generic(N : positive := 16;
                bits : positive := 16;
                address : integer := 4;
                data : positive := 32);
        port (clock,reset,inicio : in std_logic;
              fft_in : in std_logic_vector (bits-1 downto 0);
              fft_out : out std_logic_vector (data-1 downto 0));
end split_radix;

architecture estructura of split_radix is
signal par: std_logic;
signal aux: std_logic_vector(address-1 downto 0);
signal entrada,a_mari,b_mari,c_mari,d_mari: std_logic_vector (data-1 downto 0)  := (others=>'0');
signal reset0,hab0,sel1,sel2,hab1,stope,reset1,hab2,reset2,hab3,reset3,hab4,sel3,hab5:
std_logic := '0';
signal reset4,hab6,sel4,hab7,reset5,hab8,reset6,hab9,reset7,reset8,hab10,we1: std_logic := '0';
signal sel5,reset9,hab11,reset10,reset11,reset12,hab12,reset13,hab13: std_logic := '0';
signal sel6: std_logic_vector(1 downto 0);
signal a_dato,b_dato,c_dato,d_dato,x_dato,y_dato,z_dato: std_logic_vector(data-1 downto 0) :=
(others=>'0');
signal pu,pares,fcont0,fin,fcont2,fcont5,salto,mayor1,fcont13,fase,ini,fin_b: std_logic := '0';
signal s0,h0,r1,h1,s1,h2,s2,h3,re0,en0,en1,en2,en3,re1,en4,re2,en5,en6: std_logic := '0';
signal f0,fmari: std_logic := '0';
signal y_mari,z_mari: std_logic_vector (data-1 downto 0) := (others=>'0');
signal rst0,ha0,par1,rst1,ha1,rst2,ha2,rst3,ha3,arst4,ha4,ha5,rst5,ha6,we2,rst6,ha7: std_logic := '0';
signal fmux3,fmux4,fmux2,sfin,fin3: std_logic := '0';
signal reset_entrada,inicio1,reset_mariposa,inicio2,reset_orden,inicio3: std_logic := '0';

begin
aux <= conv_std_logic_vector(address,address);
 par <= '1' when aux(0)='0' else '0';
entrada <= fft_in & "0000000000000000";
t0: split_entrada generic map (data,address,N) port map
        (clock,entrada,a_mari,b_mari,c_mari,d_mari,
        reset0,hab0,sel1,sel2,hab1,stope,reset1,hab2,reset2,hab3,reset3,hab4,sel3,hab5,
        reset4,hab6,sel4,hab7,reset5,hab8,reset6,hab9,reset7,reset8,hab10,we1,
        sel5,reset9,hab11,reset10,reset11,reset12,hab12,reset13,hab13,sel6,
        a_dato,b_dato,c_dato,d_dato,x_dato,y_dato,z_dato,
        pu,pares,fcont0,fin,fcont2,fcont5,salto,mayor1,fcont13,fase,ini,fin_b);
t1: maq_entrada port map
        (clock,reset_entrada,inicio1,par,
        pares,fcont0,fin,fcont2,fcont5,salto,mayor1,fcont13,fase,ini,
        reset0,hab0,sel1,sel2,hab1,stope,reset1,hab2,reset2,hab3,reset3,hab4,sel3,hab5,
        reset4,hab6,sel4,hab7,reset5,hab8,reset6,hab9,reset7,reset8,hab10,we1,
        sel5,reset9,hab11,reset10,reset11,reset12,hab12,reset13,hab13,sel6);
t2: split_mariposa generic map (bits,data,address,N) port map
        (clock,
        pu,s0,h0,r1,h1,s1,h2,s2,h3,re0,en0,en1,en2,en3,re1,en4,re2,en5,en6,
        a_dato,b_dato,c_dato,d_dato,y_dato,z_dato,
```

```
        f0,fmari,
        a_mari,b_mari,c_mari,d_mari,y_mari,z_mari);
t3: maq_mariposa port map
        (clock,reset_mariposa,inicio2,
        fcont0,f0,fmari,
        s0,h0,r1,h1,s1,h2,s2,h3,re0,en0,en1,en2,en3,re1,en4,re2,en5,en6);
t4: split_orden generic map (data,address,N) port map
        (clock,
        rst0,ha0,par1,rst1,ha1,rst2,ha2,rst3,ha3,arst4,ha4,ha5,rst5,ha6,we2,rst6,ha7,
        x_dato,y_mari,z_mari,
        fmux3,fmux4,fmux2,sfin,fin3,
        fft_out);
t5: maq_orden port map
        (clock,reset_orden,inicio3,par,fin,fmux3,fmux4,fmux2,sfin,
        rst0,ha0,par1,rst1,ha1,rst2,ha2,rst3,ha3,arst4,ha4,ha5,rst5,ha6,we2,rst6,ha7);
t6: maq_split port map
        (clock,reset,inicio,
        fcont2,fin_b,fmari,fcont0,fin3,fin,
        reset_entrada,inicio1,reset_mariposa,inicio2,reset_orden,inicio3);
end estructura;
```