

UNIVERSIDAD DE BUENOS AIRES  
FACULTAD DE INGENIERÍA  
INGENIERÍA EN ELECTRÓNICA



MEMORIA DEL TRABAJO DE TESIS

**Implementación y análisis de algoritmos  
de cálculo de Transformada Rápida de  
Fourier para su aplicación en sistemas  
OFDM**

**Autor:**  
**Andrés Dario Cassagnes**

Director:  
Dr. Ing. Ariel Lutenberg

Codirector:  
Ing. Fedeerico Giordano Zacchigna

Jurados:  
Dr. Ing. Leonardo Rey Vega  
Ing. Carlos Belaustegui Goitia  
Ing. Nicolas Álvarez

*Este trabajo fue realizado en la ciudad de San Carlos de Bariloche y en la Ciudad Autónoma de Buenos Aires, entre enero de 2016 y diciembre de 2016.*



## *Resumen*

El objetivo de este trabajo es el diseño y la implementación en hardware digital de una arquitectura de cómputo de la transformada rápida de Fourier para ser utilizada en la modulación y demodulación de un sistema de comunicaciones OFDM. El trabajo de tesis se encuadra dentro del proyecto Software Defined Radio del Laboratorio de Sistemas Embebidos.

Se presenta la teoría de base necesaria para poder explicar los desarrollos realizados. Se realiza una introducción teórica a la transformada rápida de Fourier y su utilización en los sistemas de comunicación OFDM, realizando una breve reseña de la teoría de estos sistemas de comunicación.

Se describen las arquitecturas desarrolladas durante el trabajo de tesis, explicando en cada caso el criterio utilizado para la selección de las arquitecturas y algoritmos a implementar.

Por último, se analizan los resultados obtenidos, determinando la utilidad de la o las arquitecturas implementadas para el fin para el cual se desarrollaron en base a ese análisis, y se proponen trabajos futuros.



## *Agradecimientos*

En primer lugar quiero agradecerle a mi director de tesis, el Dr. Ing. Ariel Lutenberg, mi co-director, el Ing. Federico Gioradno Zacchigna, apodado afectuosamente como el "Colo", y al Ing. Octavio Alpago, por haberme guiado, ayudado y aconsejado durante el desarrollo del trabajo de tesis. También quiero agradecerles por permitirme ser parte del proyecto Software Define Radio en el cual se enmarca el presente trabajo.

Agradezco especialmente a los miembros del jurado, el Dr. Ing. Leonardo Rey Vega, el Ing. Carlos Belaustegui Goitia y el Ing. Nicolás Alvarez por dedicar tiempo a la evaluación de este trabajo.

Quiero agradecer muy especialmente a mi esposa, Analía, por apoyarme, acompañarme, soportarme, incentivar me, y a veces marcar me el camino con cierta firmeza, a lo largo de la carrera y la realización del presente trabajo de tesis. Su apoyo y paciencia fueron fundamentales para la finalización de esta etapa de mi vida.

A mis padres, grandes responsables de la persona que soy, y de quienes aprendí el valor de la perseverancia y el trabajo duro. A mi hermano Luis y su familia, Adriana y Agustín, por aceptarme en su hogar en los primeros años de la carrera. A la familia Rodas, por aceptarme y hacerme sentir desde el principio como uno más de la familia.

A mis compañeros de la facultad con quienes desarollé una amistad más allá de las aulas. Guido Martinez y Lucho Natale, con quienes cerramos filas en los últimos años de cursada y encaramos en conjunto el arduo trabajo de la tesis de grado. A Sebastián D'Agostino y Adrián Coria, con quienes compartí el comienzo de la carrera, cursadas, charlas y una buena cantidad de cerveza.

A mi amigo de la vida Horacio Arce, apoyo incondicional en todo momento sin importar horas o distancias.

A los compañeros de facultad, a quienes valoro mucho haber conocido y compartido horas, tanto de cursada como fuera de las aulas. Sebastián Gerez, Juanjo Borges, Juan Pablo Contini, Dario Basso.

A aquellos profesores de quienes me llevo más que solo conceptos de sus materias. Fernando Barreiro, Carlos Belaustegui, Octavio Alpago, Leandro Santi, Daniel Dalmati, Juan Manuel Cruz. Compartieron no solo sus conocimientos, sino también su experiencia de vida con total generosidad.

Al Laboratorio de Sistemas Embebidos, y a su director Dr. Ing. Ariel Lutenberg, por abrirme las puertas y brindar un ámbito de trabajo excelente para poder desarrollar no solo el presente trabajo de tesis sino otros proyectos académicos.

A mis compañeros de trabajo de estos últimos años, el grupo de Instrumentación Neutrónica de INVAP SE. Darío, Emiliano, Emmanuel, Federico, Fernando y Marcos, por apoyarme e incentivar me, para que finalice el presente trabajo de tesis.



# Índice general

<b>Resumen</b>	<b>III</b>
<b>1. Introducción al Trabajo de Tesis</b>	<b>1</b>
1.1. Objetivo . . . . .	1
1.2. Alcance . . . . .	2
1.3. Organización del Trabajo . . . . .	2
<b>2. FFT y su uso en las comunicaciones</b>	<b>5</b>
2.1. La Transformada de Fourier . . . . .	5
2.1.1. Transformada de Fourier de tiempo continuo . . . . .	5
2.1.2. Transformada de Fourier en Tiempo Discreto . . . . .	6
2.2. La Transformada Rápida de Fourier . . . . .	8
2.3. Uso de la DFT en las comunicaciones digitales . . . . .	9
2.3.1. Multiplexación por división ortogonal de frecuencia (OFDM) . . . . .	9
2.3.2. Implementación de la modulación OFDM mediante DFT . . . . .	12
<b>3. Arquitecturas para el cómputo de la Transformada Rápida de Fourier</b>	<b>15</b>
3.1. Algoritmos DFT . . . . .	15
3.1.1. Algoritmo de Goertzel . . . . .	15
3.1.2. Transformación Bluestein Chirp-z . . . . .	16
3.1.3. Algoritmo DFT de Winograd . . . . .	16
3.2. Álgoritmos FFT . . . . .	17
3.2.1. Algoritmo de Cooley-Tuckey para el cálculo de FFT . . . . .	17
Algoritmos radix-r . . . . .	18
3.2.2. Algoritmo de Good-Thomas . . . . .	19
3.3. Selección del algoritmo . . . . .	20
3.3.1. Algoritmos DFT y FFT . . . . .	20
3.3.2. Arquitecturas radix-r . . . . .	21
Formas de implementar el algoritmo radix-r . . . . .	22
3.3.3. Multiplicación por los twiddle factors . . . . .	24
Algoritmo Cordic . . . . .	24
Algoritmo BKM . . . . .	26
Multiplicador complejo eficiente . . . . .	27
3.3.4. Método de redondeo o truncamiento . . . . .	28
3.3.5. Arquitecturas a implementar . . . . .	29
<b>4. Implementación a nivel de microarquitectura</b>	<b>31</b>
4.1. Arquitectura Radix-2 . . . . .	31
4.1.1. Descripción general . . . . .	31
4.1.2. Memoria . . . . .	32
4.1.3. Butterfly . . . . .	33
4.1.4. Datapath . . . . .	34
4.1.5. Unidad de control . . . . .	37

Máquinas de estados . . . . .	38
Control de la memoria . . . . .	39
Control del datapath . . . . .	39
4.1.6. Integración de la unidad de control . . . . .	39
4.2. Arquitectura Radix-4 . . . . .	40
4.2.1. Descripción general . . . . .	40
4.2.2. Memoria . . . . .	44
4.2.3. Sumador/restador . . . . .	46
4.2.4. Datapath . . . . .	47
4.2.5. Unidad de control . . . . .	51
Máquinas de estados . . . . .	52
Control de la memoria . . . . .	53
4.2.6. Integración de la unidad de control . . . . .	54
4.3. Módulos compartidos por las dos arquitecturas . . . . .	55
4.3.1. Cordic desenrollado . . . . .	55
4.3.2. Multiplicador complejo . . . . .	57
4.3.3. Unidad de escalamiento . . . . .	58
4.4. Interfaces de las arquitecturas . . . . .	59
4.5. Herramientas utilizadas para el desarrollo . . . . .	60
<b>5. Estudio, Simulación y Validación de los IP Cores Generados</b>	<b>63</b>
5.1. Estrategia de simulación, verificación y validación . . . . .	63
5.2. Simulación y verificación de los módulos individuales . . . . .	64
5.3. Simulación y validación de las arquitecturas completas . . . . .	64
5.3.1. Procesamiento de señales patrón . . . . .	64
Delta en componente '0' . . . . .	64
Delta . . . . .	65
5.3.2. Medición del error . . . . .	65
5.3.3. Medición de la distorsión total armónica . . . . .	69
Efecto de la intermodulación . . . . .	73
5.3.4. Efectos de redondear o truncar en una etapa . . . . .	73
5.3.5. Validación de las arquitecturas mediante pruebas en hardware . . . . .	75
5.4. Análisis de utilización de recursos de las arquitecturas . . . . .	76
<b>6. Conclusiones</b>	<b>79</b>
6.1. Conclusiones Generales . . . . .	79
6.2. Trabajos Futuros . . . . .	80
<b>Bibliografía</b>	<b>81</b>

# Índice de figuras

2.1. Pulso rectangular de longitud 8, continuo y muestreado con N=16 . . . . .	7
2.2. Transformada de Fourier de un pulso rectangular . . . . .	7
2.3. Dos formas de transmisión multiportadora . . . . .	10
2.4. Idea conceptual de una señal OFDM . . . . .	10
2.5. Diagrama de tiempos de un símbolo OFDM . . . . .	11
2.6. Diagrama en bloques de una transmisión punto a punto OFDM . . . . .	12
3.1. FFT Radix-2 de 8 puntos . . . . .	22
3.2. Arquitectura Radix-2 desenrollada SDF . . . . .	23
3.3. Arquitectura Radix-2 iterativa . . . . .	23
3.4. Ejemplo de rotación con algoritmo Cordic . . . . .	26
4.1. FFT Radix-2 de 8 puntos . . . . .	32
4.2. Diagrama simplificado de la arquitectura radix-2 iterativa . . . . .	33
4.3. Dual Port RAM . . . . .	33
4.4. Esquema del bloque butterfly . . . . .	34
4.5. Esquema del datapath de la arquitectura radix-2 . . . . .	34
4.6. Datapath para operaciones de transferencia en memoria . . . . .	35
4.7. Datapath para operaciones en butterfly . . . . .	36
4.8. Selección del bit del contador de puntos a evaluar . . . . .	37
4.9. Estados de la máquina de estados principal . . . . .	38
4.10. Máquina de estados operativa para modo <i>enabled</i> . . . . .	38
4.11. Datapath con las señales de control . . . . .	40
4.12. Esquema de una FFT radix-4 de 16 puntos . . . . .	43
4.13. Diagrama simplificado de la arquitectura radix-4 iterativa . . . . .	44
4.14. RAM de triple entrada y triple salida . . . . .	44
4.15. Esquema de direccionamiento de los subbloques RAM . . . . .	46
4.16. Diagrama de la unidad aritmética incluyendo los multiplexores de bypass . . . . .	47
4.17. Datapath de la arquitectura radix-4 iterativa . . . . .	48
4.18. Datapath para operaciones de transferencia en memoria . . . . .	49
4.19. Datapath para operaciones en butterfly . . . . .	50
4.20. Selección del par de bits del contador de puntos a evaluar . . . . .	52
4.21. Diagrama de estados y transiciones de la máquina de estados principal . . . . .	52
4.22. Diagrama de estados y transiciones de la máquina de estados secundaria . . . . .	53
4.23. Datapath con las señales de control . . . . .	54
4.24. Bloque de módulo de cómputo cordic . . . . .	55
4.25. Diagrama en bloques del módulo cordic . . . . .	56
4.26. Diagrama en bloques del bloque de rotaciones del módulo cordic . .	57
4.27. Bloque de módulo multiplicador complejo . . . . .	57
4.28. Diagrama en bloques de la unidad de escalamiento . . . . .	59

4.29. Señales de comunicación de las arquitecturas implementadas . . . . .	59
5.1. Respuestas a una delta en la componente 0 para las arquitecturas radix-2 y radix-4 . . . . .	65
5.2. Respuestas a una delta en la componente 7 para las arquitecturas radix-2 y radix-4 . . . . .	66
5.3. Diagrama de flujo de la simulación para la estimación del error . . . . .	67
5.4. Diagrama de flujo de la simulación para la estimación de la THD . . . . .	70
5.5. THD en función del tono de entrada, radix-2 de 1024 puntos . . . . .	71
5.6. THD en función del tono de entrada, radix-2 de 4096 puntos . . . . .	71
5.7. THD en función del tono de entrada, radix-4 de 1024 puntos . . . . .	72
5.8. THD en función del tono de entrada, radix-4 de 4096 puntos . . . . .	72
5.9. THD en función del tono de entrada, Kiss FFT . . . . .	73
5.10. THD de la respuesta a dos tonos consecutivos . . . . .	74
5.11. Comparación entre el procesamiento con escalamiento y sin escalamiento . . . . .	74
5.12. THD en función de la etapa en que se realiza es escalamiento . . . . .	75
5.13. THD en función de la etapa en que se realiza es escalamiento para una señal que provoca overflow . . . . .	75
5.14. Testbench para la validación de las arquitecturas en hardware . . . . .	76
5.15. Comparativa de tamaño de síntesis de diferentes arquitecturas para 1024 puntos en una FPGA XC5VLX110 . . . . .	77
5.16. Comparativa de tamaño de síntesis de diferentes arquitecturas para 4096 puntos en una FPGA XC5VLX110 . . . . .	77

# Índice de Tablas

3.1. Comparativa entre algoritmos FFT . . . . .	21
3.2. Cantidad de operaciones complejas para distintas longitudes de DFT	21
3.3. Comparativa entre las implementaciones paralela, desenrollada e iterativa del algoritmo radix-r . . . . .	23
3.4. Ejemplos de truncamiento y redondeo en sistema decimal . . . . .	28
4.1. Ejemplos de codificación del ángulo para un ancho de palabra del ángulo de 7 bits . . . . .	58
5.1. Métrica $E_\infty$ para 1024 realizaciones de cada arquitectura con entradas aleatorias . . . . .	67
5.2. Métrica $E_2$ para 1024 corridas de cada arquitectura con entradas aleatorias . . . . .	68



*A mis padres*



## Capítulo 1

# Introducción al Trabajo de Tesis

El presente trabajo de tesis se encuentra enfocado en el contexto del diseño de hardware digital. El mismo fue motivado por el creciente avance en las tecnologías utilizadas en los sistemas de comunicación en general y en los sistemas *Software Defined Radio*<sup>1</sup> en particular. Los sistemas *SDR* permiten implementar sistemas de comunicación mediante software o hardware digital reconfigurable, otorgándoles una gran flexibilidad.

La creciente demanda de velocidad en las telecomunicaciones lleva a la implementación de sistemas de transmisión cada vez más veloces. Uno de los sistemas de transmisión de datos más difundidos es el sistema *OFDM*, en el cual se utilizan múltiples portadora en las que se modulan los datos a trasmitir. Una forma práctica y eficiente de implementar las modulación y demodulación multiportadora requerida por este sistema es mediante el uso de la Transformada Discreta de Fourier, aprovechando los algoritmos de alta eficiencia disponibles para su implementación.

Teniendo en cuenta la complejidad de los sistemas de transmisión *OFDM* y la necesidad de poder implementarlos en forma eficiente tanto en consumo como en espacio y recursos utilizados, el presente trabajo de tesis se basa en el desarrollo de un sistema de modulación y demodulación para transmisiones *OFDM* con muy baja ocupación espacial y de recursos, de manera que pueda ser integrado fácilmente en sistemas de comunicación reducidos.

### 1.1. Objetivo

El trabajo de tesis tiene como objetivo diseñar e implementar en hardware digital distintas arquitecturas capaces de realizar el cálculo de la Transformada Discreta de Fourier (*DFT* por sus siglas en inglés), teniendo como requerimientos sobre las arquitecturas desarrolladas que realicen el cálculo en forma eficiente y sean de tamaño reducido. El diseño e implementación será realizado mediante el lenguaje de modelado de hardware Verilog, orientado a la implementación en ASIC,

---

<sup>1</sup>Software Defined Radio (SDR por sus siglas en inglés): Sistema de comunicaciones donde los componentes típicamente implementados en hardware (mezcladores, filtros, amplificadores, moduladores / demoduladores, detectores, etc) son implementados en software.

mientras que la validación se realizará en un dispositivo FPGA<sup>2</sup>.

El desarrollo de la tesis se divide en una parte teórica y una experimental. Se analizarán distintos desarrollos algorítmicos para el cálculo de la Transformada Discreta de Fourier, se seleccionarán los que se consideren más aptos de acuerdo a los requerimientos para las arquitecturas a desarrollar y se realizará un análisis para el diseño de IP cores que implementen los algoritmos seleccionados. Una vez finalizado el diseño e implementación se realizarán una serie de pruebas y ensayos de verificación de los IP cores generados, para finalizar con la validación de los mismos a través de la síntesis en FPGA utilizando un kit de desarrollo. Se analizarán los datos obtenidos de cada simulación y prueba para la determinación de las características de los diseños implementados, tales como el error y la distorsión total armónica.

## 1.2. Alcance

Como resultados a obtener de la presente tesis se tienen los siguientes:

- IP Cores codificados en el lenguaje Verilog de arquitecturas de cálculo de FFT de tamaño reducido.
- Estudio del comportamiento numérico de las arquitecturas implementadas (ruido, distorsión armónica, etc.)
- Análisis comparativos de procesamiento entre las arquitecturas desarrolladas y desarrollos de terceros.
- Proposición de trabajos futuros y/o mejoras.

## 1.3. Organización del Trabajo

En esta sección se describe la organización de la presente tesis. Con el objetivo de que la misma sea autocontenida, los primeros capítulos se ocupan de presentar las bases o conocimientos necesarios para comprender la totalidad del trabajo.

El desarrollo de la tesis se organiza de la siguiente forma:

- En el capítulo 2 se hará referencia a los conceptos requeridos para comprender el desarrollo de la Transformada Discreta de Fourier y su utilización en sistemas OFDM. Se desarrollarán los conceptos teóricos matemáticos que describen la Transformada de Fourier y se realizará una breve descripción de los sistemas de comunicación OFDM para brindar el marco teórico en el que se encuadra el trabajo de tesis.
- En el capítulo 3 se describirán los diferentes algoritmos para el cálculo de la DFT. Se elegirán aquellos que se consideren aptos para la implementación

---

<sup>2</sup>Field Programmable Gate Array: dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada ‘in situ’ mediante un lenguaje de descripción especializado.

en base a un determinado criterio. Se analizarán también las diferentes alternativas que puedan surgir para la implementación de las distintas partes componentes de los algoritmos seleccionados y se elegirán las más indicadas de acuerdo a determinados criterios que serán expuestos cuando se requieran.

- En el capítulo 4 se describirá la implementación en Verilog de los algoritmos seleccionados. Se generará un IP core en RTL para cada algoritmo implementado. Dichos RTL deberán cumplir con ciertas condiciones de portabilidad y legibilidad de código para que los mismos sean efectivamente IP Cores.
- En el capítulo 5 se expondrán los bancos de prueba de simulación y los resultados obtenidos de las mismas. También se realizará la implementación sobre dispositivos FPGA midiendo los recursos utilizados y comparándolo con implementaciones de terceros para verificar el cumplimiento de los requerimientos de diseño de las arquitecturas.
- En el capítulo 6 se extraerán las conclusiones pertinentes sobre los resultados obtenidos y se propondrán futuras mejoras de las arquitecturas a partir del análisis realizado.



## Capítulo 2

# FFT y su uso en las comunicaciones

La Transformada Rápida de Fourier (Fast Fourier Transform o FFT por sus siglas en inglés) es la denominación colectiva para un grupo de algoritmos que permiten el cálculo eficiente de la Transformada Discreta de Fourier (DFT) y posibilitan su implementación en sistemas digitales a un costo menor que el cálculo de la DFT en forma directa [3]. Cuando se habla de costo se refiere a características mensurables de la arquitectura como el tamaño, tiempo de ejecución, consumo, etc.

En este capítulo se presenta un resumen de las características de la transformada de Fourier de tiempo continuo y de su versión discreta, y de la transformada rápida de Fourier y su importancia en las comunicaciones digitales.

### 2.1. La Transformada de Fourier

La transformada de Fourier permite obtener la representación o descomposición en el dominio de las frecuencias de señales representadas en el dominio del tiempo. Esta representación permite un análisis más amplio de las señales y facilita el estudio de los efectos de los sistemas sobre las mismas y el procesamiento de señales en general.

No es la intención de este trabajo extenderse en la teoría de la Transformada de Fourier por lo que se omiten los cálculos referentes a la deducción de las ecuaciones correspondientes a las diferentes formas de la misma. Estas deducciones se pueden encontrar en [8][12].

#### 2.1.1. Transformada de Fourier de tiempo continuo

La descomposición en series de Fourier de señales periódicas permite descomponer señales con período finito en el tiempo en una suma de exponenciales complejas equiespaciadas que representan los armónicos en frecuencia que componen la señal. Teniendo en cuenta la utilidad que tiene esta representación en el análisis y procesamiento de señales periódicas (análisis de sistemas LTI, diseño de filtros, etc.) resulta de interés poder analizar de la misma manera señales no periódicas.

Pensando una señal de duración finita  $x(t)$  como un ciclo de una señal periódica  $\tilde{x}(t)$  se puede plantear la representación en serie de Fourier de la misma [9]. Una

vez expresada la sumatoria que describe la serie de Fourier se puede tomar su límite cuando el período tiende a infinito (ya que se puede pensar una señal aperiódica como una señal periódica de período infinito) convirtiendo la sumatoria en una integral. A través de una deducción matemática [9] se llega al siguiente par de ecuaciones:

$$X(j\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t} d\omega \quad (2.1)$$

$$x(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(j\omega)e^{j\omega t} d\omega \quad (2.2)$$

(2.1) y (2.2) son conocidas como el par de transformadas de Fourier, cuya función  $X(j\omega)$  es la *Transformada de Fourier* de la señal  $x(t)$  y (2.2) es la ecuación de la *transformada inversa de Fourier*.

### 2.1.2. Transformada de Fourier en Tiempo Discreto

La transformada de Fourier en tiempo continuo es una gran herramienta para el análisis de señales, pero la imposibilidad de procesar la transformada de Fourier para señales reales sumado al gran costo computacional de la aproximación numérica utilizando punto flotante lleva a la búsqueda de una alternativa que posibilite el procesamiento digital. Esto se ve potenciado además por el avance de los procesadores digitales, tanto en velocidad como en capacidad de cómputo, costo, etc. Teniendo en cuenta todo esto y para aprovechar el potencial del procesamiento digital se llega a la Transformada de Fourier en Tiempo Discreto (DFT, Discrete Fourier Transform) donde se utilizan N muestras tanto en tiempo como en frecuencia [9]. Además, como se describe en la sección 3.3.1, existen algoritmos eficientes para su cómputo lo que posibilita su implementación práctica en sistemas digitales.

Tomando como base la descomposición en series de Fourier de una secuencia periódica y su transformada de Fourier a través de sus coeficientes se llega a las ecuaciones que definen la Transformada de Fourier en Tiempo Discreto a través de un razonamiento similar al de la transformada continua de Fourier tomando una secuencia finita  $x[n]$  de longitud N como un período de una secuencia  $\tilde{x}[n]$  de período N:

$$X[k] = \sum_{n=0}^{N-1} x[n]W_N^{kn} \quad (2.3)$$

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k]W_N^{-kn} \quad (2.4)$$

donde  $W_N^{kn} = e^{-j2\pi kn/N}$ , comúnmente llamados *twiddle factors*.

(2.3) es la ecuación de la DFT mientras que (2.4) es la ecuación de la transformada inversa de Fourier en Tiempo Discreto (IDFT) también conocida como ecuación de síntesis. Esta última ecuación muestra como la señal  $x[n]$  se representa como

una combinación lineal de exponentiales complejas o desde otro punto de vista como una combinación lineal de componentes en frecuencia.

Como ejemplo se ve en la figura 2.1 un pulso rectangular continuo de longitud 8 y su versión muestreada con  $N = 16$ . Este pulso se puede tomar como un ciclo de una onda cuadrada de período 16. En la figura 2.2 se presenta la transformada continua de Fourier y la transformada en tiempo discreto de Fourier en color azul y rojo respectivamente. Se aprecia como la DFT es un muestreo de la transformada continua, con  $N = 16$ , equiespaciado en frecuencia.

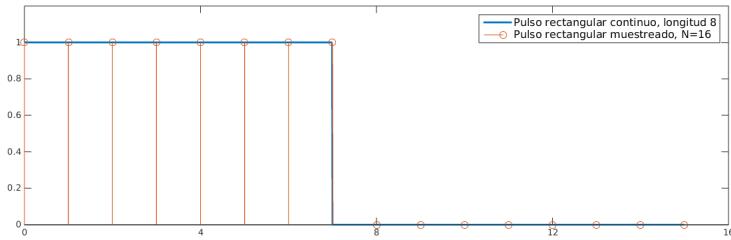


FIGURA 2.1: Pulso rectangular de longitud 8, continuo y muestreado con  $N=16$

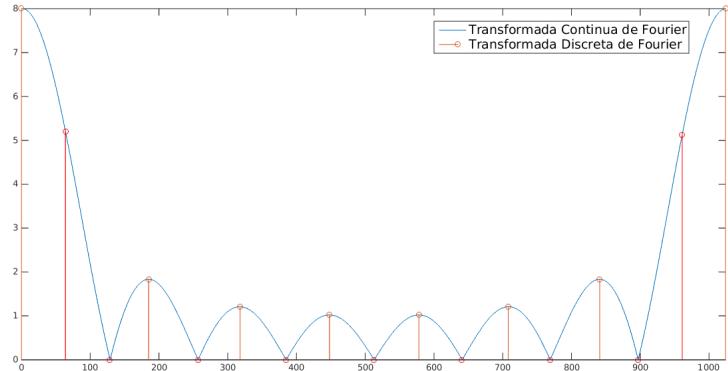


FIGURA 2.2: Transformada de Fourier de un pulso rectangular

Se debe recordar los efectos del muestreo en tiempo y en frecuencia:

- Al muestrear en tiempo se obtiene un espectro periódico con frecuencia de muestreo  $f_s$ . La aproximación de la transformada de Fourier a través de la DFT es razonable si las componentes en frecuencia de  $x(t)$  están concentradas en un rango menor a la frecuencia de Nyquist  $f_s/2$  de acuerdo con el teorema de muestreo de Shannon [7].
- Al muestrear en frecuencia la función temporal se vuelve periódica, esto es, la DFT asume a la función  $x(t)$  como periódica. Por este motivo se debe utilizar una frecuencia de muestreo y una ventana de análisis de forma que cubran un número entero de períodos de  $x(t)$  (o períodos teóricos en caso que  $x(t)$  no sea periódica) en  $N$  muestras para evitar anomalías en la transformada, tales como leakage, ripple, etc.

## 2.2. La Transformada Rápida de Fourier

La implementación directa de la DFT según (2.3) implica que para cada una de las  $N$  salidas del cálculo se requieren  $N$  operaciones aritméticas,  $N$  sumas, lo que equivale a una complejidad del orden de  $\Theta(N^2)$ , lo cual es inaceptables en sistemas escalables. Por esto históricamente se buscaron formas más eficientes de realizar el cálculo de la DFT. Estos algoritmos se conocen globalmente como Transformada Rápida de Fourier (FFT, Fast Fourier Transform), que permiten reducir la complejidad al orden de  $\Theta(N * \log(N))$  [8][3].

Los algoritmos de FFT utilizan la estrategia *divide y vencerás* a través de una transformación de una DFT de longitud  $N$  (2.3) en una representación multidimensional  $N = \prod_l N_l$ . Esto permite dividir el cálculo de una DFT de  $N$  puntos en múltiples DFT de  $N_l$  reduciendo de esta manera la complejidad computacional total.

Como ejemplo se analizará brevemente la descomposición en dos dimensiones. Se transforma el índice temporal  $n$  según

$$n = An_1 + Bn_2 \text{ mod } N \quad \begin{cases} 0 \leq n_1 \leq N_1 - 1 \\ 0 \leq n_2 \leq N_2 - 1 \end{cases} \quad (2.5)$$

Donde  $N = N_1 * N_2$  y  $A, B \in Z$  son constantes a definir. Usando esta transformación se construye un mapeo bidimensional del tipo  $f : C^N \rightarrow C^{N_1 \times N_2}$ .

Aplicando una transformación similar al índice  $k$  en el dominio de la frecuencia:

$$k = Ck_1 + Dk_2 \text{ mod } N \quad \begin{cases} 0 \leq k_1 \leq N_1 - 1 \\ 0 \leq k_2 \leq N_2 - 1 \end{cases} \quad (2.6)$$

Donde  $C, D \in Z$  son constantes a definir. Como la DFT es biyectiva se debe tener precaución en la elección de los coeficientes  $A, B, C$  y  $D$  para que la nueva representación de la transformada siga teniendo esta característica.

Esta representación implica la separación de la DFT de  $N$  puntos en dos DFT de  $N_1$  y  $N_2$  puntos aplicadas una a continuación de la otra. Esta representación se puede realizar de forma recursiva y subdividir a su vez  $N_1$  como  $N_1 = N_{11}N_{12}$  y así sucesivamente.

Los algoritmos de FFT permiten un cálculo eficiente de la DFT no solo en tiempo de cálculo sino también en complejidad de código en caso de software y en complejidad, tamaño y consumo en caso de hardware haciendo posible su implementación en circuitos integrados.

Los diferentes algoritmos de FFT se diferencian entre sí por los valores que asignan a los coeficientes  $A, B, C$  y  $D$ . Existen algoritmos que permiten aprovechar esta optimización del cálculo dependiendo de la naturaleza de la señal y el objetivo del cálculo para cualquier longitud de la DFT.

## 2.3. Uso de la DFT en las comunicaciones digitales

### 2.3.1. Multiplexación por división ortogonal de frecuencia (OFDM)

La creciente demanda de servicios de comunicación inalámbrica lleva a la necesidad de aumentar el flujo de datos transmitidos a través de radiofrecuencias. Tratar de llevar la velocidad de transmisión de símbolos para alcanzar tasas de bits del orden del Mbit o mayores requiere la utilización de ecualización adaptativa lo que eleva la complejidad y el costo de los equipos utilizados [10]. Una forma de encarar el problema es la multiplexación en frecuencia, donde los símbolos en los que se modula la información se multiplexan en múltiples portadoras y se transmiten en forma simultánea aumentando de esta forma la tasa de transferencia de bits sin necesidad de aumentar la frecuencia o la complejidad de los símbolos utilizados en la modulación. Otra ventaja de la transmisión multiportadora es su robustez frente a interferencia selectiva en frecuencia ya que solo se verían afectadas algunas de las portadoras y el error provocado se puede corregir mediante un sistema de corrección de errores.

En los sistemas tradicionales de transmisión multiportadora la banda de frecuencia total se divide en  $N$  canales sin superposición los cuales son modulados por diferentes símbolos y multiplexados en frecuencia. La transmisión simultánea de múltiples frecuencias presenta el riesgo de interferencia entre portadoras (ICI, Inter Charrier Interference) por lo que los canales deben separarse de forma de reducir la interferencia entre ellos, lo que lleva a un aprovechamiento ineficiente del espectro disponible. Para mejorar la eficiencia en el uso del ancho de banda a mediados de la década de 1960 se propuso la utilización de transmisión en paralelo, al igual que en el sistema tradicional, y la multiplexación en frecuencia sobre canales superpuestos, que llevando una velocidad de símbolo  $b$  son espaciados entre si una distancia  $b$  en frecuencia para disminuir el ruido impulsivo y distorsión por caminos múltiples además de aprovechar mejor el ancho de banda [10]. En la figura 2.3 se ve la diferencia en el aprovechamiento del ancho de banda al transmitir 8 subportadoras separadas y las mismas 8 subportadoras superpuestas.

El sistema de multiplexado en división de frecuencias ortogonales (OFDM, Orthogonal Frequency-Division Multiplexing) propone la utilización de frecuencias matemáticamente ortogonales para reducir la interferencia entre las portadoras. Con el receptor actuando como un banco de demoduladores que traslada cada subportadora a banda base, se realiza una integración sobre un período de la señal de esa subportadora para recuperar la información. Si las demás portadoras tienen un número entero de ciclos durante ese período ( $T$ ), luego de la integración la contribución de estas a la subportadora que se está procesando es nula. Entonces las portadoras serán ortogonales si están separadas un múltiplo de  $1/T$ . La figura 2.4 muestra un conjunto de subportadoras conformando una señal OFDM.

El sistema de transmisión por OFDM se utiliza en muchos tipos de comunicación actuales siendo uno de los sistemas más extendidos [10]. Este sistema está presente en las conexiones de datos tipo DSL, en las redes inalámbricas (WLAN, WPAN) de tipo Wi-Fi, WiMax, etc, en las transmisiones de video y audio digital (DAB/DVB) entre otros muchos medios de transmisión de información.

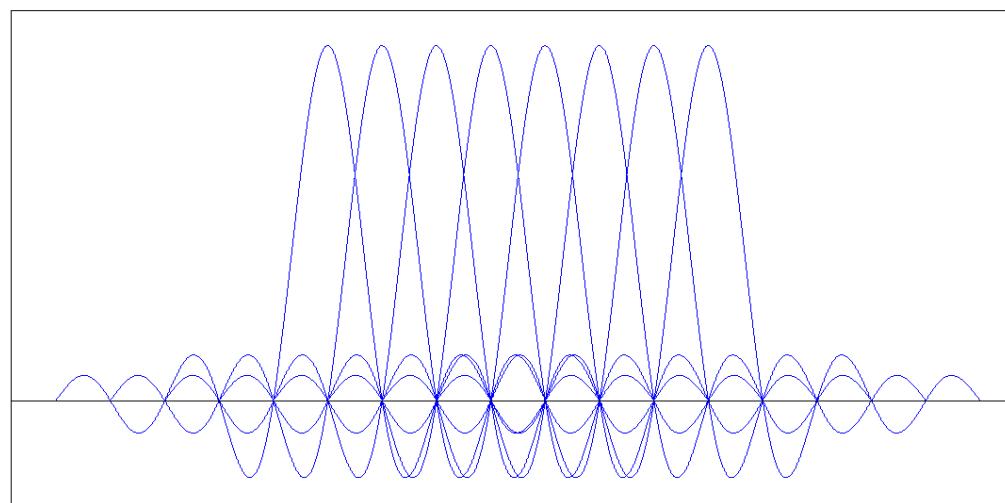
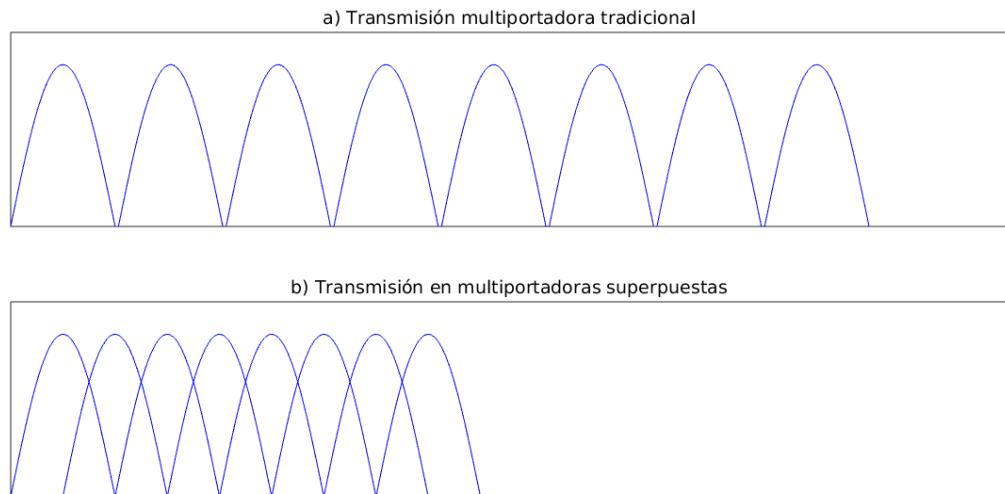


FIGURA 2.4: Idea conceptual de una señal OFDM

Si bien por si sola la modulación OFDM presenta varias ventajas en la transmisión de datos esto no alcanza para una implementación práctica ya que deben tenerse en cuenta las características distorsivas del canal.

Una de las modificaciones principales sobre el sistema teórico es el agregado de un intervalo de guarda (GI sus siglas en inglés) utilizado para reducir el efecto del delay producido por la característica multi-camino del canal. Este GI está compuesto por la repetición de la porción final del símbolo transmitido al principio del mismo, convirtiendo el símbolo en periódico. Esta porción agregada se conoce como prefijo cíclico. Gracias al prefijo cíclico el efecto dispersivo en tiempo del canal se reduce a una convolución cíclica por lo que con solo descartar el GI en el receptor se obtiene el símbolo completo. Además, gracias a las características de la convolución cíclica o circular, se preserva la ortogonalidad de las

subportadoras. La desventaja de aplicar el prefijo cíclico es la disminución en la eficiencia del uso del ancho de banda al agregar información no útil a la transmisión. La duración del GI ( $T_{guard}$ ) es seleccionada para que sea mayor al máximo delay del canal. Por lo tanto la parte efectiva de la señal recibida puede ser vista como la convolución cíclica del símbolo transmitido por la respuesta impulsiva del canal.

Por otro lado un pulso rectangular tiene un gran ancho de banda debido a los lóbulos laterales de la sinc que compone su espectro. Para reducir la potencia transmitida fuera del ancho de banda del símbolo se agrega un tiempo de ventana a la transmisión con una forma de onda que cae progresivamente, a diferencia del pulso rectangular, lo que reduce los lóbulos laterales de la sinc reduciendo a su vez el ancho de banda total del símbolo. Esta ventana se agrega al intervalo de guarda descrito anteriormente aumentando la robustez respecto de la dispersividad del canal pero reduciendo aún más la eficiencia ya que esta ventana también es descartada por el receptor.

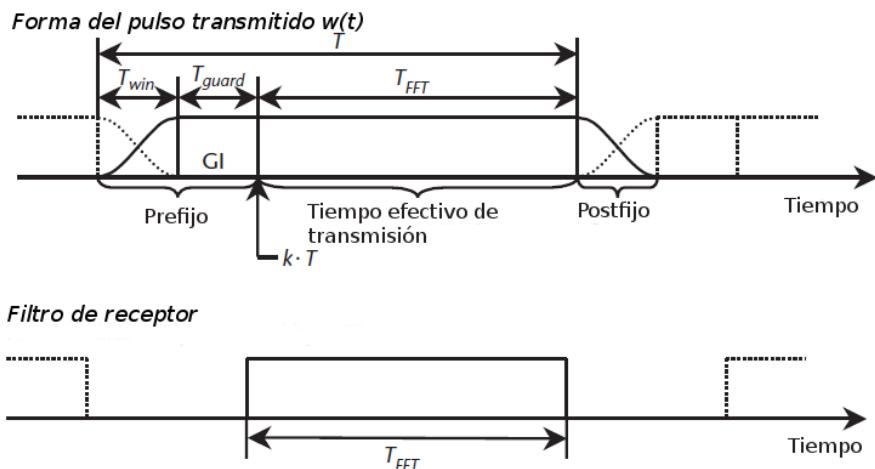


FIGURA 2.5: Diagrama de tiempos de un símbolo OFDM

Esto queda ejemplificado en la figura 2.5 donde se ve el esquema de tiempos de un símbolo OFDM. Como este procesamiento se realiza habitualmente en hardware digital estos tiempos suelen expresarse en muestras.  $T$  ( $N$ ),  $T_{FFT}$  ( $N_{FFT}$ ),  $T_{guard}$  ( $N_{guard}$ ) y  $T_{win}$  ( $N_{win}$ ) representan los tiempos (número de muestras) del símbolo transmitido, su parte efectiva, el GI y el tiempo de ventana respectivamente.

En la figura 2.6 se observa un diagrama en bloques de un transmisor OFDM. La data de origen es dividida en paquetes/canales y mapeada a los símbolos respectivos de la constelación correspondiente a la modulación seleccionada para la transmisión. Esos símbolos, que componen una constelación de símbolos complejos, son modulados en las multiportadoras ortogonales para su transmisión y se le agrega el intervalo de guarda para luego convertir la señal resultante digital en una señal analógica para ser transmitida. Una vez recibida esta señal en el receptor, es digitalizada para su procesamiento y se remueve el intervalo de guarda. La porción de información efectiva de la señal es enviada al demodulador OFDM que extrae la constelación de puntos complejos que es demapeada en la información original. El presente trabajo se enfoca en desarrollar los bloques de modulación y demodulación OFDM del diagrama de bloques expuesto.

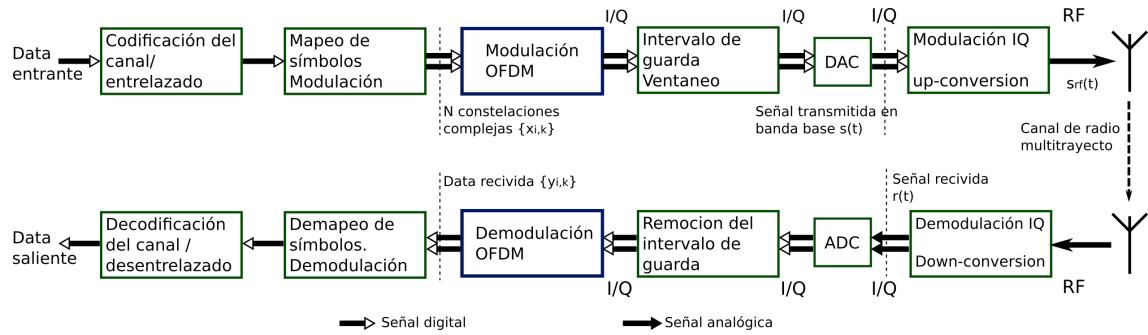


FIGURA 2.6: Diagrama en bloques de una transmisión punto a punto OFDM

### 2.3.2. Implementación de la modulación OFDM mediante DFT

Matemáticamente la OFDM se expresa como la suma de los pulsos base desplazados en tiempo y frecuencia y multiplicados por los símbolos de datos. En notación temporal, el k-ésimo símbolo de una señal OFDM se escribe como:

$$s_{RF,k}(t - kT) = \operatorname{Re} \left\{ w(t - kT) \sum_{i=-N/2}^{N/2-1} x_{i,k} e^{j2\pi(f_c + \frac{i}{T_{FFT}})(t-kT)} \right\} \quad (2.7)$$

para  $kT - T_{\text{guard}} - T_{\text{win}} \leq t \leq kT + T_{\text{FFT}} + T_{\text{win}}$  y 0 para todo otro caso.

Donde:

- $T$ : Duración de símbolo, tiempo entre dos símbolos consecutivos
- $T_{\text{FFT}}$ : Parte efectiva de un símbolo
- $T_{\text{guard}}$ : GI, duración del prefijo cíclico
- $T_{\text{win}}$ : intervalo de ventana
- $k$ : Índice del símbolo transmitido
- $i$ : Índice de subportadora,  $i \in \{-N/2, -N/2+1, \dots, -1, 0, 1, \dots, N/2-1\}$
- $x_{i,k}$ : Punto de la constelación de la señal. Símbolo complejo modulado en la i-ésima portadora del k-ésimo símbolo OFDM
- $w(t)$ : El pulso formador

Finalmente una secuencia continua de símbolos OFDM se expresa como:

$$s_{RF}(t) = \sum_{k=-\infty}^{\infty} s_{RF,k}(t - kT) \quad (2.8)$$

De estas ecuaciones se deduce la señal compleja equivalente en banda base:

$$s(t) = \sum_{k=-\infty}^{\infty} s_k(t - kT) \quad (2.9)$$

donde

$$s_k(t - kT) = w(t - kT) \sum_{i=-N/2}^{N/2-1} x_{i,k} e^{j2\pi \left(\frac{i}{T_{FFT}}\right)(t-kT)} \quad (2.10)$$

para  $kT - T_{\text{win}} - T_{\text{guard}} \leq t \leq kT + T_{\text{FFT}} + T_{\text{win}}$  y 0 para todo otro caso.

Se puede notar la similitud entre (2.10) y (2.4) de la IDFT donde  $k$  representa la subportadora. Esta similitud es sumamente importante ya que permite reemplazar los moduladores del transmisor por el cálculo de una IDFT, o su versión de mayor eficiencia IFFT, y el banco de filtros para demodular en el receptor por el cálculo de una DFT al realizar procesamiento digital. Esto representa una simplificación considerable en el diseño de sistemas OFDM y su procesamiento mediante hardware digital o incluso por software y ha motivado el creciente estudio sobre la transformada discreta de Fourier y los algoritmos de cálculo eficiente de la misma, buscando arquitecturas de procesamientos más eficiente, más pequeñas, con menor consumo de energía/recursos, etc.



## Capítulo 3

# Arquitecturas para el cómputo de la Transformada Rápida de Fourier

En el capítulo anterior fue expuesta la utilidad de la iDFT y la DFT para la modulación y demodulación de señales en un sistema OFDM y las ventajas de los algoritmos FFT para el cálculo de la transformada discreta de Fourier.

Siguiendo la terminología introducida por Burrus [2] [13] se define como *algoritmos FFT* a aquellos donde se realiza un mapeo multidimensional de los índices en sus entradas y salidas, como se explicó en la sección 2.2, dejando el término *algoritmo DFT* a aquellos algoritmos donde no se utiliza un mapeo multidimensional de los índices.

### 3.1. Algoritmos DFT

Los algoritmos DFT implementan el cálculo de la DFT aplicando optimizaciones sobre la ecuación de síntesis.

#### 3.1.1. Algoritmo de Goertzel

Cada componente del espectro de una señal  $x[n]$  en un cómputo de DFT puede escribirse como

$$X[k] = x[0] + x[1]W_N^k + x[2]W_N^{2k} + \dots + x[N-1]W_N^{(N-1)k} \quad (3.1)$$

Se pueden combinar todos los  $x[n]$  con el mismo factor común  $W_N^k$  obteniendo

$$X[k] = x[0] + W_N^k(x[1] + W_N^k(x[2] + \dots + x[N-1]W_N^k \dots)) \quad (3.2)$$

Esta ecuación resulta en una posible implementación recursiva para el cálculo de  $X[k]$ . Este cómputo es conocido como el algoritmo de Goertzel. Su mayor utilidad se encuentra para calcular un número reducido de componentes espectrales ya que para el cálculo de una DFT completa la complejidad llega a  $\Theta(N^2)$ , donde no representa ninguna mejora respecto al cálculo directo de la DFT [14].

### 3.1.2. Transformación Bluestein Chirp-z

En la transformación Bluestein Chirp-z el exponente  $kn$  en 2.3 es expandido cuadráticamente a

$$kn = -(k-n)^2/2 + n^2/2 + k^2/2 \quad (3.3)$$

El cómputo de la DFT se reduce entonces a

$$X[k] = W_N^{k^2/2} \sum_{n=0}^{N-1} (x[n]W_N^{n^2/2})W_N^{-(k-n)^2/2} \quad (3.4)$$

Entonces el cálculo de la DFT mediante este algoritmo se reduce a tres pasos:

- N multiplicaciones de  $x[n]$  por  $W_N^{n^2/2}$
- Convolución lineal de  $x[n]W_N^{n^2/2} * W_N^{n^2/2}$
- N multiplicaciones por  $W_N^{k^2/2}$

Para una transformación completa se necesita una convolución de longitud N y 2N multiplicaciones complejas. Una ventaja de este algoritmo es que puede utilizarse para cualquier valor de N, sin la limitación de que N sea par o potencia de 2 o 4 [15].

### 3.1.3. Algoritmo DFT de Winograd

El algoritmo de Winograd transforma el cómputo de la DFT en una convolución cíclica y aplica el algoritmo de convolución de Winograd [11]. La longitud de la DFT queda restringida a números primos o potencias de números primos. La transformación de la DFT en una convolución se realiza mediante el método Rader [16], en el que partiendo de la premisa que N es un número primo se puede hallar un elemento generador  $g$  que genere todos los elementos  $n$  y  $k$  [17], a excepción del 0, de forma que

$$X[k] = \sum_{n=0}^{N-1} x[n]W_N^{nk} \quad k, n \in \mathbb{Z}_N \quad (3.5)$$

Reemplazando  $n$  por  $g^n \bmod N$  y  $k$  por  $g^k \bmod N$  se obtiene el siguiente mapeo de índices:

$$X[g^k \bmod N] - x[0] = \sum_{n=0}^{N-2} x[g^n \bmod N]W_N^{g^{n+k} \bmod N} \quad (3.6)$$

para  $k \in \{1, 2, 3 \dots N - 1\}$ . Se puede notar que el lado derecho de 3.7 es una convolución cíclica, i.e.,

$$[x[g^0 \bmod N], x[g^1 \bmod N], \dots, x[g^{N-2} \bmod N] * [W_N, W_N^g, \dots, W_N^{g^{N-2} \bmod N}]] \quad (3.7)$$

Implementando la convolución con el algoritmo de Winograd se reduce la cantidad de multiplicaciones no triviales aumentando así la eficiencia del cómputo de DFT [11].

## 3.2. Algoritmos FFT

Los algoritmos FFT logran la eficiencia en el cómputo de la DFT a través de un mapeo multidimensional de los coeficientes.

### 3.2.1. Algoritmo de Cooley-Tuckey para el cálculo de FFT

El algoritmo de Cooley-Tukey es el más universal de los algoritmos para cálculo de FFT porque permite utilizar cualquier factorización de  $N$  [18]. En particular los algoritmos de Cooley-Tukey que transforman  $N$  en una potencia de base  $r$ ,  $N = r^\nu$ , son llamados *Radix-r* y son los más populares [18].

La transformación de índices propuesta por Cooley y Tukey (y por Gauss previamente) es también la más simple. Partiendo de (2.5) y (2.6) se utilizan como coeficientes  $A = N_2$ ,  $B = 1$ ,  $C = 1$  y  $D = N_1$  resultando en el mapeo:

$$n = N_2 n_1 + n_2 \quad \begin{cases} 0 \leq n_1 \leq N_1 - 1 \\ 0 \leq n_2 \leq N_2 - 1 \end{cases} \quad (3.8)$$

$$k = k_1 + N_1 k_2 \quad \begin{cases} 0 \leq k_1 \leq N_1 - 1 \\ 0 \leq k_2 \leq N_2 - 1 \end{cases} \quad (3.9)$$

Dado el intervalo que pueden tomar  $n_1$  y  $n_2$  el cálculo del módulo no necesita estar explícito.

Reemplazando  $n$  y  $k$  en  $W_N^{nk}$  de acuerdo a (3.8) y (3.9):

$$W_N^{kn} = W_N^{N_2 n_1 k_1 + N_1 N_2 n_1 k_2 + n_2 k_1 + N_1 n_2 k_2} \quad (3.10)$$

Como  $W_N^{nk}$  es de orden  $N = N_1 N_2$  se llega a que  $W_N^{N_1} = W_{N_2}$  y  $W_N^{N_2} = W_{N_1}$  que aplicado en 3.10:

$$W_N^{kn} = W_{N_1}^{n_1 k_1} W_N^{n_2 k_1} W_{N_2}^{n_2 k_2} \quad (3.11)$$

Reemplazando (3.11) en (2.3) se llega a:

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \left( W_N^{n_2 k_1} \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1} \right) \quad (3.12)$$

La sumatoria interior en (3.12) es una DFT de  $N_1$  puntos que está multiplicada por el factor  $W_N^{n_2 k_1}$ . Definiendo  $\tilde{x}[n_2, k_1] = W_N^{n_2 k_1} \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1}$  y reemplazando en (3.12) se llega a:

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \tilde{x}[n_2, k_1] \quad (3.13)$$

(3.13) muestra la DFT de  $N_2$  puntos de  $\tilde{x}$ , lo que representa la característica principal de este algoritmo, para cualquier valor de  $N_1$  y  $N_2$  tales que  $N = N_1 N_2$  la DFT de longitud  $N$  de  $x(n)$  se puede calcular siguiendo los siguientes pasos:

- Transformar los índices de la secuencia de entrada según (3.8)
- Calcular la DFT de  $N_1$  puntos de la secuencia  $x(n)$ .
- Multiplicar los puntos resultantes del paso anterior por el twiddle factor correspondiente.
- Calcular la DFT de longitud  $N_2$  de la secuencia resultante del paso anterior.
- Transformar los índices de la secuencia de salida según (3.9)

Nada impide aquí subdividir cualquiera de las DFT de la ecuación (3.13) en dos DFT de longitudes menores sucesivamente hasta obtener DFT de longitudes convenientes para realizar los cálculos.

Una ventaja del algoritmo de Cooley-Tuckey es la posibilidad del alojamiento en memoria *in-place* en el cual los resultados del cálculo de una etapa se guarda en memoria en las mismas posiciones que los valores utilizados para el cálculo, utilizando de forma eficiente la memoria ya que para el cálculo de una DFT de  $N$  puntos solo se requiere una memoria de longitud  $N$ .

### Algoritmos radix-r

Aprovechando la posibilidad que brinda el algoritmo de Cooley-Tuckey de poder factorizar  $N$  libremente se puede optar por una factorización del tipo  $N = r^\nu$ . A los algoritmos de Cooley-Tuckey de este estilo se los conoce como algoritmos *radix-r*. De esta manera el cálculo de la DFT se descompone en  $\nu$  DFTs consecutivas de  $r$  puntos cada una.

Por ejemplo para  $r = 2$  y  $\nu$  etapas,  $N = 2^\nu$ , el mapeo de índices queda como

$$n = 2^{\nu-1} n_1 + \dots + 2 n_{\nu-1} + n_\nu \quad (3.14)$$

$$k = k_1 + 2 k_2 + \dots + 2^{\nu-1} k_\nu \quad (3.15)$$

El número total de twiddle factors para un algoritmo radix-2 es:

$$\log_2(N)N/2 \quad (3.16)$$

La ventaja de esto es poder reducir el cálculo de una DFT a múltiples cálculos de DFTs de tamaño más pequeño y de cálculo más simple. Teniendo en cuenta por ejemplo que en el cómputo de una DFT de longitud 2 o 4 no es necesario realizar ninguna multiplicación no trivial a excepción del producto por el twiddle factor, eligiendo  $N$  como potencia de 2 o 4 se reduce la cantidad de multiplicaciones a realizar aumentando así la eficiencia del algoritmo.

### 3.2.2. Algoritmo de Good-Thomas

La transformación de índices sugerida por Good y Thomas transforma una DFT de longitud  $N = N_1 * N_2$  en una DFT bidimensional sin *twiddle factors*, como si hay en el algoritmo de Cooley-Tukey. El costo de la ausencia de *twiddle factors* es la necesidad de que los factores  $N_1$  y  $N_2$  deben ser coprimos (i.e.,  $\gcd(N_k, N_l) = 1$  para  $k \neq l$ ) y el mapeo se torna más complicado al tener que realizar el cálculo de los índices en el momento y no poder utilizar tablas precalculadas para ello. Si se intenta eliminar los *twiddle factors* introducidos por el mapeo de índices, según 2.5 y 2.6, se obtiene:

$$W_N^{nk} = W_N^{(An_1+Bn_2)(Ck_1+Dk_2)} = W_N^{ACn_1k_1+ADn_1k_2+BCn_2k_1+BDn_2k_2} \quad (3.17)$$

de donde surgen las siguientes condiciones necesarias que deben ser satisfechas en forma simultánea:

$$\langle AD \rangle_N = \langle BC \rangle_N = 0 \quad (3.18)$$

$$\langle AC \rangle_N = N_2 \quad (3.19)$$

$$\langle BD \rangle_N = N_1 \quad (3.20)$$

El mapeo sugerido por Good y Thomas cumple con estas condiciones y está dado como sigue:

$$A = N_2 \quad B = N_1 \quad C = N_2 \langle N_2^{-1} \rangle_{N_1} \quad D = N_1 \langle N_1^{-1} \rangle_{N_2} \quad (3.21)$$

Entonces

$$n = N_2 n_1 + N_1 n_2 \text{ mod } N \quad \begin{cases} 0 \leq n_1 \leq N_1 - 1 \\ 0 \leq n_2 \leq N_2 - 1 \end{cases} \quad (3.22)$$

$$k = N_2 \langle N_2^{-1} \rangle_{N_1} k_1 + N_1 \langle N_1^{-1} \rangle_{N_2} k_2 \bmod N \quad \begin{cases} 0 \leq k_1 \leq N_1 - 1 \\ 0 \leq k_2 \leq N_2 - 1 \end{cases} \quad (3.23)$$

Sustituyendo el mapeo de índices de Good-Thomas en la ecuación de la DFT (2.3) se obtiene

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \left( \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1} \right) \quad (3.24)$$

La sumatoria interior en (3.24) es una DFT de longitud  $N_1$ . Definiendo  $\tilde{x}[n_2, k_1] = \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1}$  y reemplazando en (3.24) se llega a:

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \tilde{x}[n_2, k_1] \quad (3.25)$$

En este caso el procedimiento para el cómputo de la DFT mediante este algoritmo es similar al procedimiento del algoritmo de Cooley-Tuckey pero sin la multiplicación intermedia por el *twiddle-factor*.

Existe una versión de este algoritmo conocido como el Algoritmo FFT de Winograd donde se utiliza el mismo mapeo de Good-Thomas pero la DFT bidimensional resultante se resuelve utilizando el producto de Kronecker [19].

### 3.3. Selección del algoritmo

#### 3.3.1. Algoritmos DFT y FFT

Una métrica habitual a la hora de comparar los algoritmos de cómputo de DFT suele ser la cantidad de multiplicaciones que deben realizarse, parámetro en que el algoritmo FFT de Winograd tiene el mejor desempeño [19], pero, dado que el dispositivo donde se va a implementar el algoritmo puede tener unidades de cálculo de productos o similares, no debe ser este el único aspecto a tener en cuenta. La primera decisión que debe tomarse es que tipo de algoritmo se utilizará, DFT o FFT. Para este trabajo de tesis se seleccionará un algoritmo FFT ya que el mapeo multidimensional de los índices permite dividir el cómputo de la DFT en cómputos más pequeños reduciendo la complejidad de la implementación y permitiendo el uso de *pipelines*<sup>1</sup> entre cada etapa de cómputo aumentando la velocidad de procesamiento.

En la Tabla 3.1 se muestra una comparativa entre los distintos algoritmos FFT que será tomada en cuenta para seleccionar cual será el algoritmo a implementar.

<sup>1</sup> El pipelining consiste en la división de un módulo combinacional en submódulos combinacionales entre los cuales se colocan registros de modo de dividir la ejecución de todo el bloque en varios ciclos de clock, lo que permite aumentar su frecuencia de trabajo

Propiedad	Cooley-Tuckey	Good-Thomas	Winograd
Restricción sobre N	No	Si. $gcd(N_k, N_l) = 1$	
Máximo orden de W	N		$\max(N_k)$
Necesidad de Twiddle Factors	Si	No	No
# Multiplicaciones	bueno	bueno	mejor
# Sumas	bueno	bueno	bueno
# Esfuerzo de cómputo de índices	mejor	bueno	malo
Data in-place	Si	Si	No

TABLA 3.1: Comparativa entre algoritmos FFT

Se puede apreciar en la tabla 3.1 que el algoritmo de Cooley-Tuckey presenta las mejores características para uso general, siendo balanceada en cuanto a la cantidad de operaciones aritméticas y la sobrecarga por el cómputo de los índices, pero ofreciendo la posibilidad de utilizarla para cualquier valor de N. Por este motivo se utilizará el algoritmo de Cooley-Tuckey para la implementación de este trabajo de tesis, ya que la aplicación final de la arquitectura será como parte de un transmisor OFDM donde no se puede limitar el número de puntos a las restricciones de los otros algoritmos mencionados. Además, utilizando un algoritmo del tipo radix-r, se puede aprovechar la descomposición en DFTs del mismo tamaño reutilizando módulos tanto en la replicación de código como en un uso más eficiente de la arquitectura una vez implementada.

### 3.3.2. Arquitecturas radix-r

Como se menciona en la sección 3.3.1 la arquitectura a implementar será de tipo radix-r. En estos algoritmos se deben calcular  $\nu$  DFTs de  $r$  puntos cada una donde  $N = r^\nu$ . El valor de  $r$  toma importancia ya que la correcta elección del mismo puede conducir a implementaciones más o menos eficientes. En la tabla 3.2 se muestra la cantidad de operaciones necesarias para bloques de DFT de distintos tamaños. Se considera como multiplicaciones triviales aquellas que son por  $\pm 1$  y  $\pm j$ .

Long. del bloque	Multiplicaciones	Multiplicaciones no triviales	sumas
2	2	0	2
3	3	2	6
4	4	0	8
5	6	5	17
7	9	8	36
8	8	2	26
9	11	10	44

TABLA 3.2: Cantidad de operaciones complejas para distintas longitudes de DFT

Se puede ver que para  $r = 2$  y  $r = 4$  no hay necesidad de realizar multiplicaciones no triviales dentro de cada bloque DFT, quedando únicamente los productos por los twiddle factors. El siguiente valor de  $r$  en orden de eficiencia es  $r = 8$  que requiere dos multiplicaciones complejas por DFT. Hay que tener en cuenta que si bien para un mismo valor de  $N$  al aumentar el valor de  $r$  disminuye la cantidad de etapas de DFT, también aumenta la complejidad de la implementación ya que

en cada bloque se deben procesar DFTs de  $r$  puntos. Teniendo en cuenta esto se decide implementar la arquitectura en dos versiones, utilizando  $N = 2$  y  $N = 4$ , ya que al ser valores pequeños permiten implementar la arquitectura para diversos valores de  $N$  manteniendo la complejidad de la implementación en niveles razonables, y se realizará un análisis comparativo del rendimiento entre ellas. Esto también implica que no es necesario agregar un multiplicador a los bloques DFT. Es por estos motivos también que la mayoría de las implementaciones comerciales de bloques de cálculo de DFT se realizan con arquitecturas radix-2 y radix-4 [20].

### Formas de implementar el algoritmo radix-r

En la figura 3.1 se muestra un esquema simplificado de un algoritmo radix-2 de 8 puntos.

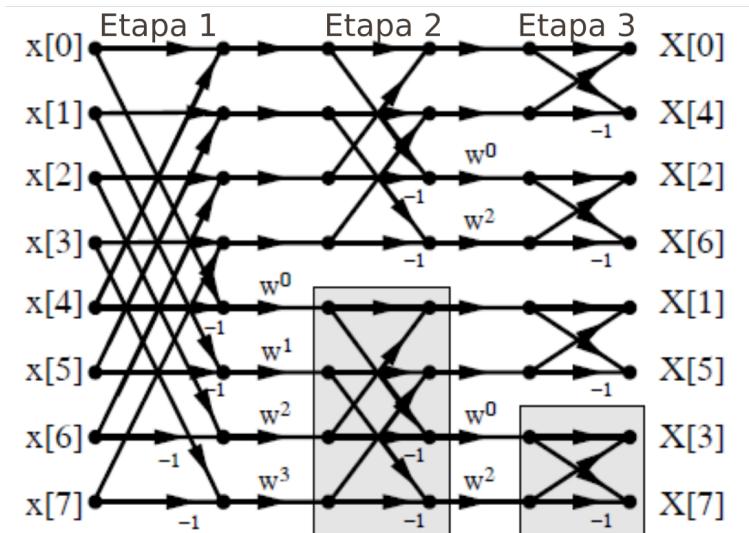


FIGURA 3.1: FFT Radix-2 de 8 puntos

En este esquema cada nodo donde llegan dos flechas representa una suma y cada flecha sobre una línea representa un producto por el factor que la acompaña. Se aprecia claramente la división en etapas en cada una de las cuales se realizan DFTs de 2 puntos. Cada DFT de dos puntos es conocida como *butterfly* por la forma que toman las dos flechas cruzadas en el esquema de la figura 3.1, en el caso de un algoritmo radix-4 el esquema es similar pero se realizan DFTs tomando cuatro puntos. Se observa que para realizar una DFT de 8 puntos se requieren 12 *butterflys*, generalizando  $\frac{N}{2} * \log_2(N)$ , y 8 multiplicadores complejos, generalizando  $\frac{N}{2} * (\log_2(N) - 1)$ . Existen diferentes variantes en la forma en que se implementa el camino de los datos (*datapath*), que implican diferencias en las cantidades de unidades de cómputo *butterfly* y de multiplicadores complejos. Las alternativas más comunes se listan a continuación:

- **Paralela** Se implementan todas las unidades *butterfly* y multiplicadores complejos necesarios, dispuesto en una arquitectura similar al esquema de la figura 3.1. Se puede implementar en forma *pipelined* colocando un banco de registros entre etapas consecutivas. La comunicación entre las etapas se realiza mediante un bus paralelo de tamaño  $N$ .

- **Desenrollada** Arquitectura SDF (Single-path Delay Feedback en inglés) [6]. Se muestra un esquema de esta arquitectura para una FFT de 8 puntos en la figura 3.2. La comunicación se realiza mediante un bus serie, admitiendo un dato de entrada en cada ciclo de *clock*. Se utiliza una unidad *butterfly* por etapa, requiriendo en total  $\log_{\nu}(N)$ , y un multiplicador complejo por etapa excepto en la última dando un total de  $\log_{\nu}(N) - 1$  multiplicadores. Se puede implementar en forma pipelined colocando un registro en medio de etapas consecutivas.
- **Iterativa** Se utiliza una única unidad *butterfly* y un único multiplicador complejo para realizar secuencialmente las operaciones de todas las etapas. En la figura 3.3 se ve un esquema de la arquitectura radix-2 iterativa para 8 puntos.

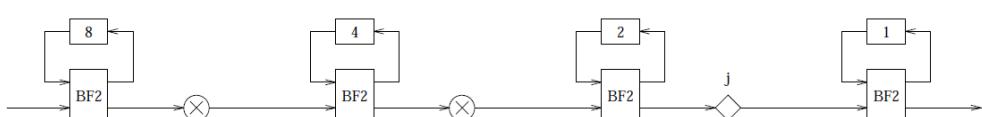


FIGURA 3.2: Arquitectura Radix-2 desenrollada SDF

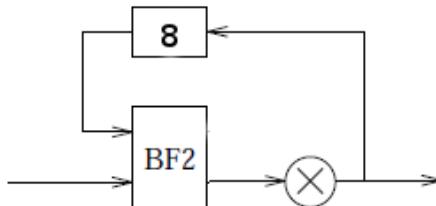


FIGURA 3.3: Arquitectura Radix-2 iterativa

En la tabla 3.3 se realiza la comparativa entre las características distintivas de cada implementación. El *throughput* se toma como la cantidad y frecuencia con que se obtienen puntos a la salida de la arquitectura. El campo *pipelined* indica si la implementación puede ser implementada de esa manera. En el caso del tamaño de memoria requerido no se consideran implementaciones *pipelined*, ya que en ese caso para la implementación paralela hay que colocar N registros por cada etapa de *pipeline* que se coloca, mientras que en la implementación desenrollada se necesita un registro por etapa de *pipeline*.

Característica	Radix paralela	Radix desenrollada	Radix iterativa
# butterfly	$\frac{N}{\nu} * \log_{\nu}(N)$	$\log_{\nu}(N)$	1
# multiplicadores	$\frac{N}{\nu} * (\log_{\nu}(N) - 1)$	$\log_{\nu}(N) - 1$	1
tamaño de memoria	0	$N - 1$	$N$
tipo de bus	Paralelo	Serie	Serie
throughput	$N$ puntos por ciclo	1 punto por ciclo	1 punto cada $\log_{\nu}(N)$ ciclos
pipeline	Si	Si	No

TABLA 3.3: Comparativa entre las implementaciones paralela, desenrollada e iterativa del algoritmo radix-r

Se puede observar que al aumentar el valor de  $N$ , la cantidad de puntos de la FFT a calcular, aumenta la cantidad de unidades *butterfly* y multiplicadores en

las implementaciones paralela, donde aumenta en forma proporcional, y en la desenrollada, donde aumenta en forma logarítmica, donde también aumenta el tamaño de la memoria. En la implementación iterativa un aumento en la cantidad de puntos a procesar solo implica un aumento en memoria.

Teniendo en cuenta que el diseño de la arquitectura está orientado a su utilización en un sistema de comunicación MIMO, no hay necesidad de un bus paralelo ya que los puntos de entrada llegan a la arquitectura en serie, y son leídos de la misma manera, por el propio funcionamiento del sistema de comunicación. En este sentido la implementación paralela queda prácticamente descartada. En cuanto al throughput, se puede operar la unidad de procesamiento FFT a una velocidad de *clock* mayor al resto del sistema para obtener el *throughput* necesario.

Se puede ver que la relación de tamaño entre la implementación desenrollada y la iterativa es del orden de  $\log_r(N)$ : 1 para los módulos de cómputo aritmético. En base a la comparativa realizada y al requerimiento de que la arquitectura sea económica en términos espaciales y de consumo se opta por la implementación iterativa, ya que al aumentar  $N$  solo se incrementa la cantidad de memoria requerida manteniendo una arquitectura de tamaño reducido y bajo consumo.

### 3.3.3. Multiplicación por los twiddle factors

La implementación de multiplicadores en lógica digital es un tema delicado en cuanto al rendimiento espacial y temporal. El cómputo de la DFT por el método de Cooley-Tuckey requiere multiplicaciones complejas por los twiddle factors por lo que la forma de implementar los multiplicadores no es un tema trivial. En una arquitectura desenrollada se necesitan  $\log_r(N) - 1$  multiplicadores, dándole principal importancia al aspecto espacial de la implementación, en tanto que en una arquitectura iterativa solo se necesita un único multiplicador por lo que el aspecto principal a tener en cuenta es la velocidad de cómputo del multiplicador para permitir una mayor frecuencia de cómputo. Se analizan tres métodos para el cómputo del producto por los *twiddle factors*.

#### Algoritmo Cordic

El producto por los twiddle factors, de la forma  $W_N^{kn} = e^{\frac{-j2\pi kn}{N}}$ , genera una rotación en el plano complejo por lo que se puede reemplazar la multiplicación por una rotación. En este sentido la primera opción que surge es la del algoritmo Cordic (Coordinate Rotation Digital Computer sus siglas en inglés) que realiza rotaciones en dos dimensiones (además de otras operaciones trigonométricas) en base únicamente a sumas/restas y desplazamientos. Este algoritmo fue presentado en 1966 por Volder [4] y es ampliamente utilizado para el cálculo de funciones trigonométricas en sistemas digitales.

El principio de funcionamiento del algoritmo es realizar microrotaciones al vector inicial hasta alcanzar una condición particular dependiente de una de las dos modalidades de funcionamiento: en modo vectorial las coordenadas  $(x_0, y_0)$  son rotadas hasta que  $y_0$  converge a cero, en modo rotacional el vector inicial  $(x_0, y_0)$

es rotado un ángulo  $\theta_n$ . Esta rotación se lleva a cabo mediante microrotaciones por un ángulo  $\theta_i$

$$x_{i+1} = x_i * \cos \theta_{i+1} - y_i * \sin \theta_{i+1} \quad (3.26)$$

$$y_{i+1} = y_i * \cos \theta_{i+1} + x_i * \sin \theta_{i+1} \quad (3.27)$$

Factorizando  $\theta_{i+1}$  en (3.26) y (3.27) se obtiene:

$$x_{i+1} = \cos \theta_{i+1}(x_i - y_i * \tan \theta_{i+1}) \quad (3.28)$$

$$y_{i+1} = \cos \theta_{i+1}(y_i + x_i * \tan \theta_{i+1}) \quad (3.29)$$

Si se restringe  $\tan \theta_{i+1}$  a  $\pm 2^{-i}$  los productos del paréntesis pueden ser reemplazados por desplazamientos aritméticos para cálculos en sistemas digitales de forma de eliminar la necesidad de multiplicaciones en todas las iteraciones. El término  $\cos \theta_{i+1}$  puede ser reemplazado por  $\cos \theta_{i+1} = \cos(\arctan 2^{-i})$  definiendo las siguientes variables:

$$K_i = \cos(\arctan 2^{-i}) = \frac{1}{\sqrt{1 + 2^{-2i}}} \quad (3.30)$$

$$d_i = \pm 1 \quad (3.31)$$

Sabiendo que el coseno es una función impar, por lo que  $\cos(\alpha) = \cos(-\alpha)$ , y reemplazando (3.30) y (3.31) en (3.28) y (3.29) se obtienen las ecuaciones finales para el algoritmo Cordic:

$$x_{i+1} = K_i(x_i - y_i * d_i * 2^{-i}) \quad (3.32)$$

$$y_{i+1} = K_i(y_i + x_i * d_i * 2^{-i}) \quad (3.33)$$

La multiplicación por  $K_i$  puede ser interpretada como una ganancia para todas las iteraciones por lo que puede ser aplicada al final como una ganancia  $K$  total del algoritmo igual a:

$$K = \prod K_i = \prod \frac{1}{\sqrt{1 + 2^{-2i}}} \quad (3.34)$$

En cada iteración se debe decidir si  $d_i = 1$  o  $d_i = -1$ , para lo cual se utiliza la diferencia entre el ángulo deseado y el ángulo actual. Para ello se define una nueva variable como

$$z_{i+1} = z_i - d_i \arctan 2^{-i} \quad (3.35)$$

Y para decidir el valor de  $d_i$  se utiliza

$$d_i = \begin{cases} -1 & \sin z_i < 0 \\ 1 & \sin z_i \geq 0 \end{cases} \quad (3.36)$$

$\arctan 2^{-i}$  puede ser calculado previamente y almacenado en tablas en memoria, al igual que el valor de  $K$ , que al ser un valor definido para una cantidad determinada de iteraciones su producto por el vector resultante puede ser calculado utilizando algoritmos eficientes para el cómputo de productos. En la figura 3.4 puede verse como se logra una rotación a través de microrotaciones a lo largo de 4 iteraciones.

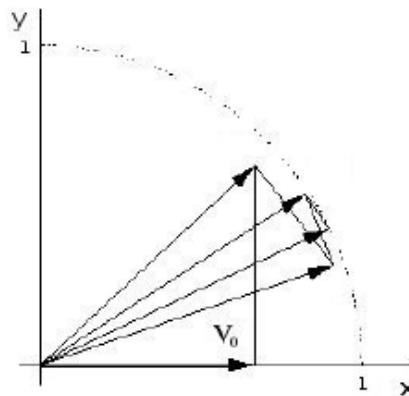


FIGURA 3.4: Ejemplo de rotación con algoritmo Cordic

Este algoritmo presenta como ventaja que solo utiliza sumas y desplazamientos para el cálculo de una rotación vectorial, y la posibilidad de implementar las etapas iterativas a través de un *pipeline* para aumentar la frecuencia de trabajo.

### Algoritmo BKM

El algoritmo BKM busca, al igual que el algoritmo Cordic, resolver funciones elementales a través de la utilización de sumas y desplazamiento de forma de poder implementarlas eficientemente en sistemas digitales. Este algoritmo se basa en las siguientes ecuaciones de recurrencia:

$$\begin{cases} L_{n+1} = L_n(1 + d_n * 2^{-n}) \\ E_{n+1} = E_n - \ln(1 + d_n * 2^{-n}) \end{cases} \quad (3.37)$$

con  $d_n \in \{-1, 0, 1, -i, i, 1 - i, 1 + i, -1 - i, -1 + i\}$ . Hay dos modos de operación del algoritmo. En el modo  $L$  se itera hasta obtener  $L_n = 1$ , entonces se obtiene  $E_n = E_1 + \ln(L_1)$ . En el modo  $E$  se itera hasta obtener  $E_n = 0$ , entonces se obtiene  $L_n = L_1 * \exp(E_1)$ . Es necesario precalcular y almacenar en memoria los valores de  $\ln(1 + d_n * 2^{-n})$  para restárselo a  $E_1$  en cada iteración. La deducción y demostración del agoritmo es compleja y se puede ver en [1] así como ejemplos de aplicación para el cálculo de varias operaciones complejas y trigonométricas. Para el cálculo del producto por los twiddle factors se utiliza la capacidad del

algoritmo de calcular la rotación de un vector  $[a, b]$  por un ángulo  $\theta$ , utilizando el modo  $E$  con  $L1 = a + ib$  y  $E1 = i\theta$ .

Comparándolo con el algoritmo Cordic, ambos brindan la posibilidad de reducir el producto por los twiddle factors a sumas y desplazamientos, con una complejidad espacial y computacional similar, aunque el BKM requiere mayor almacenamiento en memoria que el Cordic. Los dos tienen la característica de necesitar  $p$  iteraciones para obtener  $p$  bits de resolución. La principal diferencia se da en la complejidad de implementación del algoritmo, donde el BKM es más complejo que el Cordic, y en el hecho de que la principal ventaja del algoritmo BKM se da en implementaciones con sistemas numéricos redundantes [1]. Dado que la arquitectura a implementar en el presente trabajo de tesis se hará utilizando el sistema numérico conocido como *complemento a 2* (no redundante) el algoritmo BKM no provee ninguna ventaja por sobre el algoritmo Cordic, siendo por el contrario más complejo de implementar. Por este motivo de desestima su uso en este proyecto.

### Multiplicador complejo eficiente

El uso extendido del algoritmo Cordic en arquitecturas de cómputo de FFT se justifica por el costo espacial de implementar multiplicadores en sistemas digitales. Pero dado que en este caso se necesita solo un multiplicador para toda la arquitectura, la diferencia en el espacio requerido por el algoritmo Cordic y un multiplicador complejo no es significativa (caso distinto para una implementación desenrollada donde se requieren  $\log_r(N)$  multiplicaciones).

En el caso del producto por los twiddle factors se requiere realizar una multiplicación compleja de la forma:

$$R + jI = (A + jB) * (C + jD) = (A * C - B * D) + j(A * D + B * C) \quad (3.38)$$

donde  $(C + iD)$  es el twiddle factor. La implementación directa implica el uso de 4 multiplicadores. Se pueden precalcular y almacenar en una tabla determinados valores respecto a los twiddle factor obteniendo una implementación más eficiente del producto complejo, como se explica a continuación.

Se precisan y almacenan en memoria los valores  $C$ ,  $C+D$  y  $(C-D)$ . Con estos tres factores precalculados se calcula:

$$\begin{aligned} E &= A - B \\ Z &= C \times E = C \times (A - B) \end{aligned} \quad (3.39)$$

Y se computa el producto final como:

$$R = (C - D) \times B + Z \quad (3.40)$$

$$I = (C + D) \times A - Z \quad (3.41)$$

Como se indicó al principio de esta sección, al requerirse una única multiplicación compleja la diferencia entre los costos espaciales de un multiplicador complejo y el algoritmo Cordic no es significativa, por lo que se decide implementar el multiplicador complejo y evaluar su performance respecto del algoritmo Cordic para decidir cual es mejor desde el punto de vista tanto de la implementación como de la performance. Teniendo en cuenta que muchas FPGA cuentan con unidades de procesamiento de señales, incluyendo multiplicadores, la implementación de un multiplicador complejo tiene amplias ventajas sobre las demás opciones.

### 3.3.4. Método de redondeo o truncamiento

El proceso de cómputo de la DFT mediante el método Radix requiere una serie de sumas y restas, por lo que existe el riesgo de que se produzca desbordamiento (*overflow* en inglés) de la unidad aritmética. Para alamcenar el resultado de una suma entre dos operandos de  $N$  bits sin riesgo de desbordamiento se debe guardar el resultado en un número de  $N + 1$  bits. Esto implicaría incrementar en 1 el ancho de palabra de la aquitectura en cada etapa de cómputo.

Otra opción es la implmentación de un mecanismo de reducción del valor luego de una operación de suma y/o resta dividiéndolo por 2, ya que puede realizarse en forma trivial mediante un dezplazamiento hacia la derecha. Debe tenerse en cuenta que este método introduce error en el resultado ya que la dividir por 2 se pierde información.

Se decide utilizar la segunda opción por su simplicidad de implementación y porque su tamaño no depende de la cantidad de etapas a implementar. Para esto se evalua la implementación de un mecanismo de redondeo y/o truncamiento para procesar el valor resultante de la división.

El redondeo y el truncamiento se utilizan para eliminar cifras no significativas en un número. En este caso, su utilidad es la de eliminar el último bit del número que se divide por 2 para evitar el overflow.

El truncamiento consiste en eliminar las cifras no significativas descartándolas directamente. El redondeo consiste en eliminar las cifras no significativas pero se suma 1 al número resultante, en caso que la última cifra eliminada sea mayor o igual a la mitad de la base del sistema numérico, o no se suma nada en caso contrario.

En la Tabla 3.4 se muestran algunos ejemplos de la diferencia entre truncamiento y redondeo al quitar el último dígito decimal a cada número.

Número original	Truncamiento	Redondeo
2,3641	2,364	2,364
4,3156	4,315	4,316
7,6355	7,635	7,636

TABLA 3.4: Ejemplos de truncamiento y redondeo en sistema decimal

En el caso del redondeo en aritmética binaria, si el primer dígito eliminado es 1 se le suma 1 al número resultante y si es 0 no se suma.

Se puede observar que el método de truncamiento introduce un error mayor al que introduce el método de redondeo ya que el primero tiene una desviación

máxima del valor real de una unidad mientras que en el segundo la desviación máxima es de la mitad de una unidad.

Dado que la probabilidad de overflow depende de la magnitud de los puntos de entrada y que aplicar cualquier de los dos métodos de aproximación introduce error se decide implementar un mecanismo de división del resultado de la suma configurable en forma dinámica, en cuya configuración se puede habilitar y deshabilitar la opción etapa por etapa y decidir si se aplica truncamiento o redondeo en caso de habilitar la división.

### 3.3.5. Arquitecturas a implementar

De lo expuesto a lo largo del capítulo se decide implementar las siguientes arquitecturas:

- Arquitectura radix-2 iterativa
- Arquitectura radix-4 iterativa
- Algoritmo cordic para el producto por los twiddle factor para las dos arquitecturas
- Multiplicador complejo para el producto por los twiddle factor para las dos arquitecturas como alternativa al algoritmo Cordic
- Mecanismo de división por dos a la salida del sumador/restador con posibilidad de seleccionar truncamiento o redondeo en forma dinámica.

La implementación de las arquitecturas se describe en el siguiente capítulo.



## Capítulo 4

# Implementación a nivel de microarquitectura

En el capítulo 3 se analizaron diferentes tipos de arquitecturas existentes para el cómputo de la DFT y se seleccionaron dos arquitecturas para ser implementadas. A su vez fueron analizadas algunas alternativas para la implementación de diferentes aspectos del cómputo.

En este capítulo se describe la implementación a nivel de microarquitectura de los módulos que componen las dos arquitecturas seleccionadas y la forma en que esos módulos se interconectan. También se presentan las herramientas utilizadas para el diseño e implementación de las arquitecturas.

El proceso de desarrollo se dividió en tres etapas. Primero, una vez seleccionadas las arquitecturas, se conformó un diagrama en bloques identificando los módulos constitutivos, luego se implementó cada uno de esos módulos realizando pruebas de funcionamiento en cada uno en forma individual y se finalizó con la integración gradual de todos los módulos realizando las pruebas pertinentes para verificar cada etapa de integración. Una vez integrada completamente cada arquitectura se realizaron tests de verificación y validación, cuyos resultados se presentan en el capítulo 5.

Al implementarse dos arquitecturas separadas se analizará en primera instancia las particularidades de cada arquitectura, como su unidad de control, memoria y sumadores, y luego se analizarán los módulos compartidos por ambas arquitecturas.

### 4.1. Arquitectura Radix-2

#### 4.1.1. Descripción general

Como se explicó en la sección 3.3.2 al implementar un algoritmo radix-2 de  $N$  puntos en forma iterativa se utiliza el mismo módulo *butterfly* para cada una de las etapas del cómputo de la radix, debiendo esperar  $\log_2(N)$  ciclos de *clock* entre cada punto de entrada y entre cada punto de salida, durante los cuales se realiza una operación de cada etapa.

La figura 4.1 muestra el esquema presentado en la sección 3.3.2. Allí se identifican las diferentes etapas del cómputo, en cada ciclo de *clock* se ejecuta sucesivamente

una operación de cada etapa, por lo que luego de  $\log_2(N)$  ciclos se habrá ejecutado una operación de cada etapa y se vuelve a la primera.

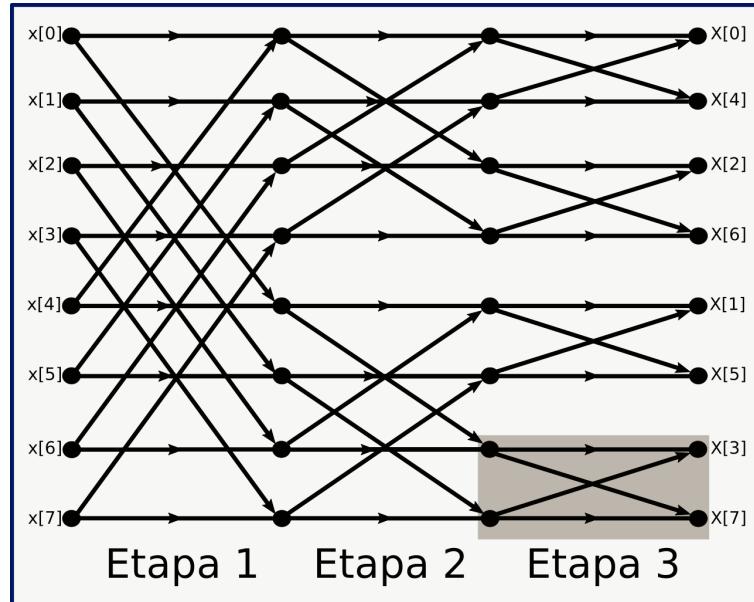


FIGURA 4.1: FFT Radix-2 de 8 puntos

Dado que los datos ingresan de a uno en la arquitectura, deben almacenarse en memoria hasta que se tiene el dato necesario para realizar una operación aritmética. Es importante notar que en cada etapa de cálculo aritmético se utilizan dos datos como entradas al *butterfly* y se obtienen dos datos como resultados, de los cuales uno se utiliza en la etapa siguiente y el otro se almacena en memoria.

En la arquitectura iterativa a implementar, se ejecuta en cada ciclo de *clock* una operación de una etapa, pasando a la siguiente etapa en el ciclo de *clock* siguiente, de forma que al transcurrir  $\log_2(N)$  ciclos de *clock* se ejecutó una operación de cada etapa volviendo a la etapa inicial.

La figura 4.2 muestra un diagrama en bloques de la arquitectura radix-2 iterativa. Se diferencian claramente tres partes: la memoria, la unidad aritmética, compuesta por el *butterfly* y el multiplicador, y la unidad de control. Los distintos colores de los bloques indican si el mismo es un circuito combinacional, un circuito secuencial o una unidad de almacenamiento. Este diagrama es a modo de esquema general y es del que se partió para el desarrollo de la arquitectura, al final de esta sección se presenta el diagrama del *datapath* detallado con las señales de control necesarias para su funcionamiento.

#### 4.1.2. Memoria

Debido al tipo de acceso a los datos se decide implementar una memoria de almacenamiento de tipo RAM de doble acceso (*dual port RAM*) permitiendo realizar simultáneamente operaciones de lectura y escritura. Las interfaces son las típicas para esta memoria y se describen en la figura 4.3.

En la figura 4.3 se muestran los buses de entrada y salida indicando su tamaño entre corchetes. El parámetro *WW* corresponde al parámetro global *WORD\_WIDTH*

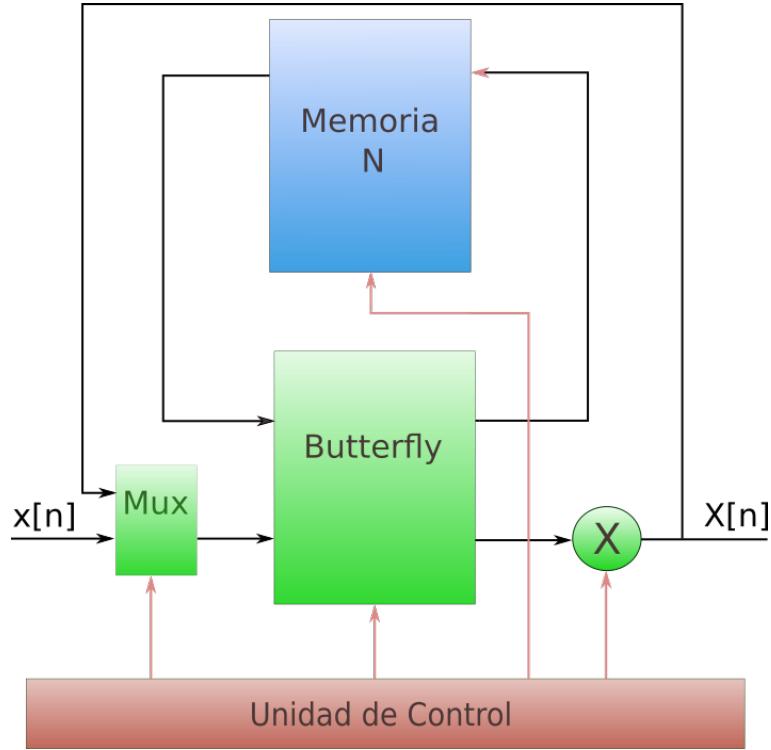


FIGURA 4.2: Diagrama simplificado de la arquitectura radix-2 iterativa

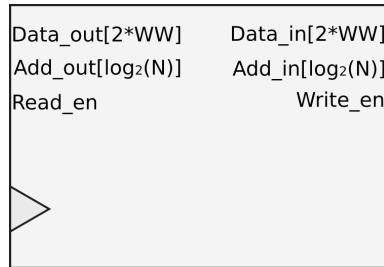


FIGURA 4.3: Dual Port RAM

$e$  indica el ancho de palabra de la arquitectura. El parámetro  $N$  indica la cantidad de puntos para la que se configura la arquitectura. Las variables a almacenar son complejas, por lo que se concatenan la parte real y la imaginaria y se guardan en memoria como un único valor. De este modo el tamaño de palabra de la memoria es el doble del tamaño de palabra de la arquitectura, teniendo también sus buses de entrada y salida el doble de ancho que los demás buses de la arquitectura.

#### 4.1.3. Butterfly

Para la arquitectura radix-2, el *butterfly* debe computar con dos operandos según la siguiente ecuación:

$$\begin{aligned} c &= a + b \\ d &= a - b \end{aligned} \tag{4.1}$$

Siendo todas las variables complejas. En la figura 4.4 se muestra el esquema del bloque *butterfly*, compuesto por un sumador y un restador complejos.

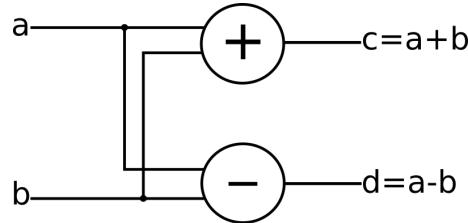


FIGURA 4.4: Esquema del bloque butterfly

#### 4.1.4. Datapath

En la figura 4.5 se muestra el *datapath* de la arquitectura radix-2. Se muestra el butterfly como un sumador y un restador, y un multiplicador genérico que puede ser implementado como un procesador cordic o como un multiplicador complejo. Dado que en las etapas intermedias y en la etapa final se debe operar con un dato de la etapa anterior se coloca un registro (*delay register*) donde se almacena ese dato durante un ciclo de clock para ser utilizado en la etapa siguiente. A la salida se le coloca un registro similar para facilitar la sincronización con otros elementos del entorno de la arquitectura.

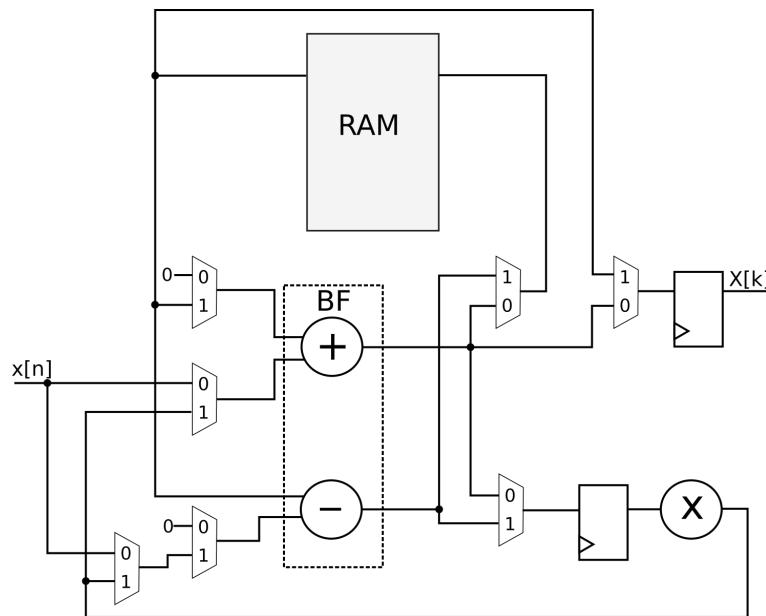


FIGURA 4.5: Esquema del datapath de la arquitectura radix-2

A través de los multiplexores se configura el *datapath* de acuerdo a la etapa en que se encuentra y el tipo de operación a realizar. De la figura 4.2 se deduce que en cada fase de cómputo se puede realizar una de dos acciones posibles, entra un dato a memoria mientras que otro dato sale de memoria hacia el multiplicador o se opera en el *butterfly* con un dato de memoria y otro dato que viene de la entrada o de la etapa anterior.

En la figura 4.6 se muestran las tres configuraciones posibles del *datapath* para las operaciones de transferencias de datos en memoria. En este tipo de operación se

lee un dato de memoria y se envía al multiplicador para realizar el producto por el *twiddle factor* o hacia la salida y se escribe un dato en memoria proveniente de la entrada o de la etapa anterior a través del *delay register*.

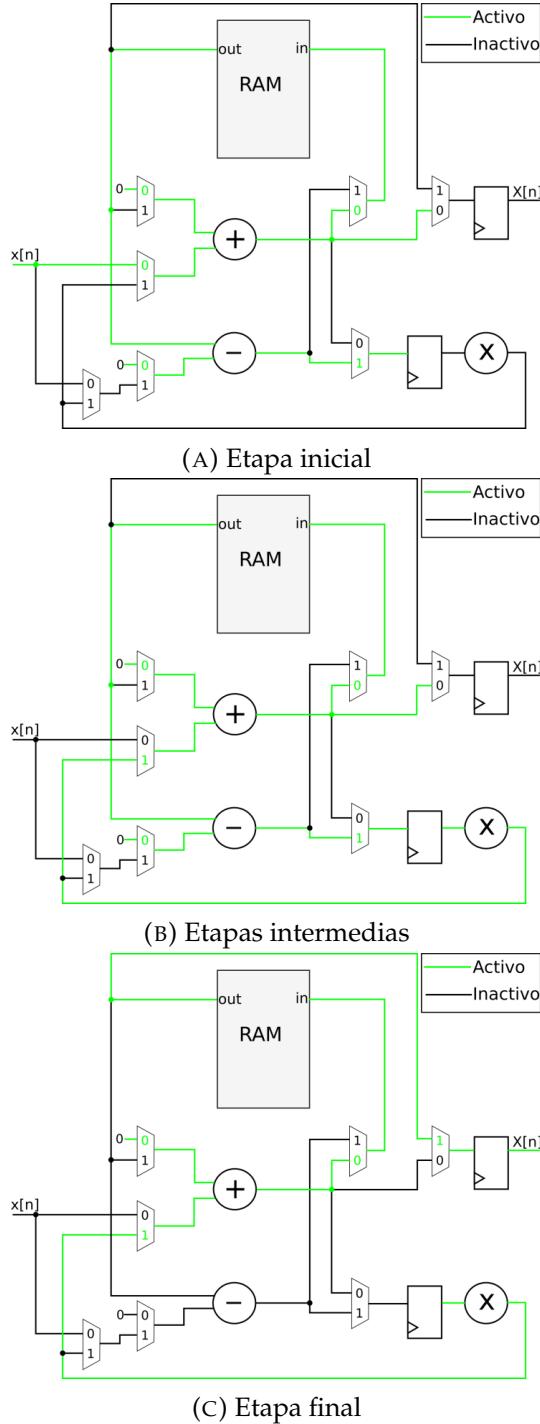


FIGURA 4.6: Datapath para operaciones de transferencia en memoria

En la figura 4.7 se muestran los posibles *datapath* para las operaciones en el butterfly. Dependiendo de si la etapa es la etapa inicial o una intermedia uno de los operandos será la entrada de la arquitectura o un dato de la entrada anterior almacenado en el *delay register* y el otro operando provendrá de la memoria. La

salida del restador del butterfly se envía directamente a memoria y la salida del sumador puede ir al *delay register* para ser utilizado en la etapa siguiente o puede ser direccionado a la salida de la arquitectura dependiendo de si la etapa es intermedia o es la etapa de salida.

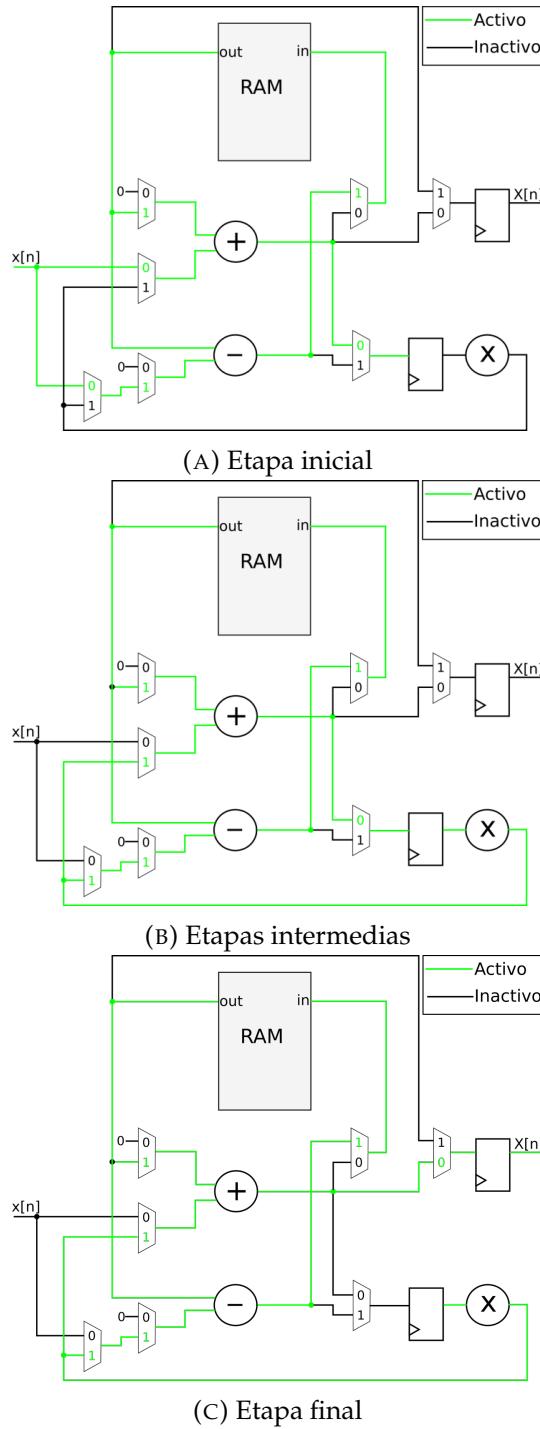


FIGURA 4.7: Datapath para operaciones en butterfly

#### 4.1.5. Unidad de control

La unidad de control debe contener la máquina de estados que controla el funcionamiento de la arquitectura, la configuración del *datapath* en cada ciclo de *clock*, el manejo de la memoria donde se almacenan los valores de cálculo y la generación de los *twiddle factors* para el multiplicador.

Teniendo en cuenta que ingresa a la arquitectura un punto nuevo cada  $\log_2(N)$  ciclos de *clock* se utiliza un contador de etapas de longitud  $\log_2(\log_2(N))$  para identificar la etapa del cómputo de la fft en que se encuentra la arquitectura. El desborde de este contador alimenta otro contador de longitud  $\log_2(N)$  que cuenta la cantidad de puntos que han ingresado a la arquitectura. Con estos dos contadores se lleva el control de la máquina de estados, que controla la configuración del *datapath* y la memoria, y la generación de los *twiddle factors*.

En la subsección 4.1.4 se describieron las distintas configuraciones del *datapath*, que deben ser gestionadas por la unidad de control de acuerdo a la etapa en que se encuentra la arquitectura y si debe realizar una transferencia a memoria o un cálculo en el *butterfly*. Tomando como ejemplo la radix-2 de 8 puntos de la figura 4.1, se ve que en la primera etapa se debe esperar la llegada del quinto punto para realizar una operación en el *butterfly*, por lo que las primeras cuatro operaciones de esa etapa serán trasnferencias de los puntos a memoria y las últimas cuatro serán operaciones en el *butterfly*. Para la segunda etapa la secuencia será de dos operaciones de transferencia a memoria y dos de operaciones aritméticas. Y para la última etapa serán operaciones de transferencia a memoria y aritméticas alteradas de a una. Entonces para una cierta etapa  $i \in \{0 \leq i \leq \log_2(N)-1\}$  la cantidad de operaciones consecutivas de cada tipo está dada por

$$\log_2\left(\frac{N}{i+1}\right) \quad (4.2)$$

Para determinar si se debe realizar una transferencia a memoria o una operación aritmética, se utiliza un solo bit del contador de puntos. El bit a evaluar se determina en función de la etapa que se está procesando, a través del *stg\_ctr*, el contador de etapas, de  $\log_2(\log_2(N))$  bits, como se muestra en la figura 4.8.

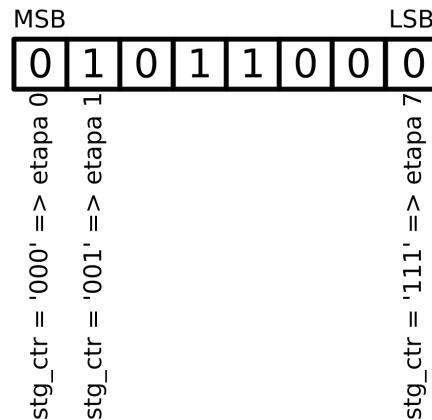


FIGURA 4.8: Selección del bit del contador de puntos a evaluar

### Máquinas de estados

La unidad de control se compone de una máquina de estados principal que controla la inicialización de los módulos de la arquitectura y una máquina de estados operativa que controla la configuración del *datapath* dependiendo de la operación que se debe realizar. En la figura 4.9 se muestran los estados y transiciones de la máquina de estados principal.

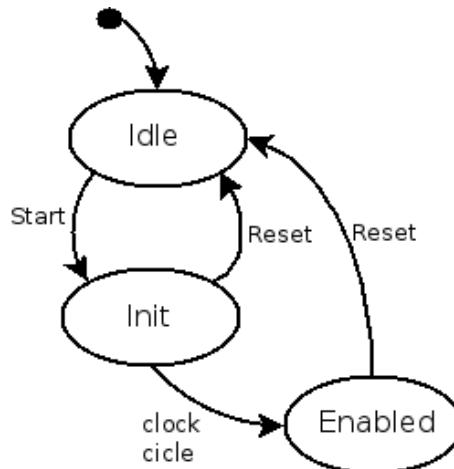


FIGURA 4.9: Estados de la máquina de estados principal

El estado inicial de la arquitectura es el estado *Idle*, donde permanece aguardando una señal para iniciar el proceso. La señal de *start* tiene la finalidad de sincronizar la arquitectura con el entorno al que está conectada, y lleva la máquina de estados al estado *Init* donde se inicializan los registros a valores conocidos y se configura el *datapath* para comenzar el procesamiento, leyendo ya el primer dato de la entrada. Un ciclo de *clock* después la máquina de estados pasa directamente al estado *Enabled* donde se realiza el procesamiento. Dentro de este estado se encuentra la máquina de estados operativa que se describe a continuación.

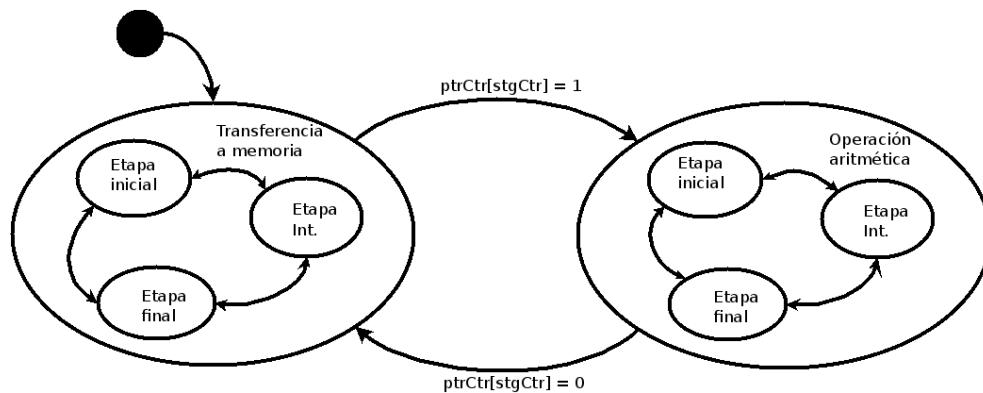


FIGURA 4.10: Máquina de estados operativa para modo *enabled*

En la figura 4.10 se observan los estados y transiciones de la máquina de estados que controla la configuración del *datapath* de acuerdo al tipo de operación a realizar. En cada uno de los estados principales funciona una máquina de estados secundaria que realiza ajustes menores de acuerdo a si la etapa actual es la etapa inicial, una intermedia o la etapa final. *ptrCtr[stgCtr]* hace referencia al bit del contador de puntos correspondiente al valor del contador de etapas, de acuerdo

a lo expuesto en la figura 4.8.

Esta máquina de estados controla además la señal de habilitación de escalamiento para la etapa actual de acuerdo al vector de escalamiento de entrada a la arquitectura. También controla, a través del contador de puntos y el de etapas, las señales de *handshaking* de salida, indicando si el dato de salida es un dato válido, señal *data\_valid* y las señales que indican si es el punto inicial (*soo*) o final (*done*) del cálculo actual.

### Control de la memoria

Dado que el algoritmo radix-2 permite el alojamiento *data in place* el manejo de la memoria es relativamente sencillo. En cada operación de una etapa se realiza siempre una escritura de datos, ya sea un dato entrante a la etapa (nuevo o de una etapa anterior) o del resultado de una operación aritmética. En cambio solo se realizan lecturas de memoria en las etapas intermedias y en la última. Al ser un algoritmo *data in place* las direcciones de memoria tanto de escritura como de lectura se calculan utilizando el contador de puntos, ya que cada en cada posición de memoria donde se alojaba un dato utilizado para una operación aritmética se alojará el resultado correspondiente a esa operación una vez realizada. Entonces en una etapa determinada se lee la posición de memoria  $k$ , se realiza un cálculo con ese dato, y se guarda nuevamente en la posición  $k$  el resultado del cálculo. Al acceder continuamente a la memoria tanto en modo escritura como lectura, las señales correspondientes de control de la memoria están siempre en modo habilitación.

### Control del datapath

El control del *datapath* se realiza mediante los multiplexores que se ven en la figura 4.5 a través de señales digitales. En cada ciclo de *clock*, de acuerdo a la máquina de estados que decide en qué etapa se encuentra el algoritmo y dependiendo de la operación a realizar se configuran los multiplexores para conformar el datapath según las figuras 4.6 y 4.7.

#### 4.1.6. Integración de la unidad de control

En la figura 4.11 se muestra el *datapath* de la figura 4.5 con el agregado de las señales de control sobre cada módulo. No se muestra la unidad de control en forma explícita para mantener la claridad del esquema. En el bloque butterfly además del sumador/restador se integra el algoritmo de escalamiento que se describe en la sección 4.3.3.

En la figura 4.11 se muestran las siguientes señales de control (se indica entre paréntesis el tamaño de las señales en caso que sea mayor a 1):

- **add\_in** ( $\log_2(N)$ ) *address in*, dirección de memoria donde se escribirá el dato
- **w\_en** *write enable*, señal de habilitación de escritura en memoria
- **add\_out** ( $\log_2(N)$ ) *address out*, dirección de memoria desde la que leerá el dato

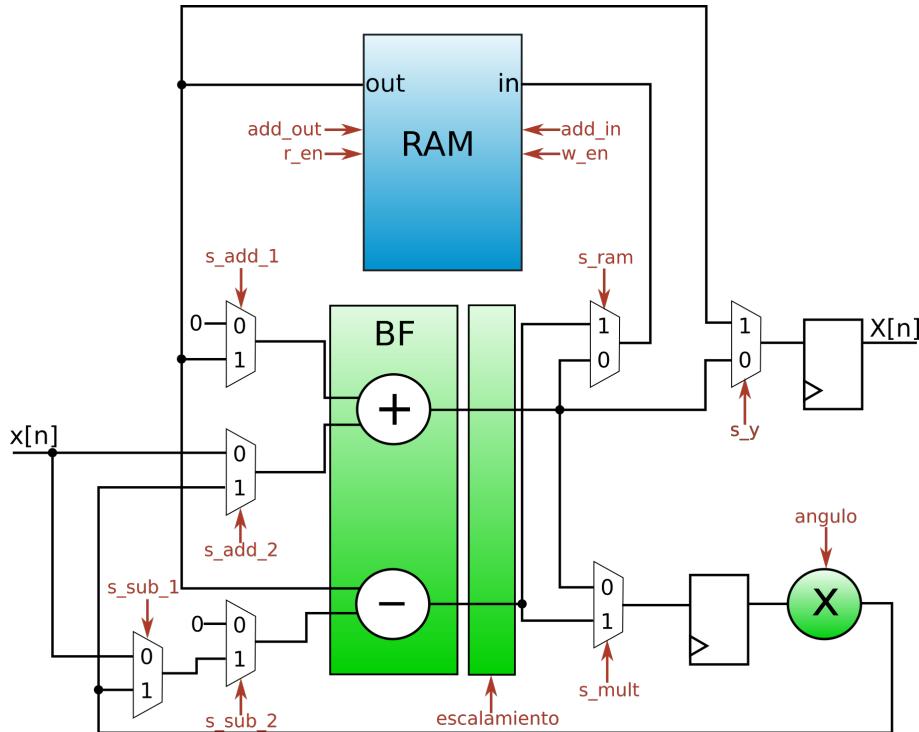


FIGURA 4.11: Datapath con las señales de control

- **r\_en** *read enable*, señal de habilitación de lectura de memoria
- **s\_add\_1** señal de control del multiplexor de la entrada 1 del sumador
- **s\_add\_2** señal de control del multiplexor de la entrada 2 del sumador
- **s\_sub\_1** señal de control del multiplexor de tipo de entrada del restador
- **s\_sub\_2** señal de control del multiplexor de entrada del restador
- **s\_mult** señal de control del multiplexor de entrada del multiplicador
- **s\_ram** señal de control del multiplexor de entrada de la memoria RAM
- **s\_y** señal de control del multiplexor de origen de la salida
- **angulo** ( $N + 1$ ) angulo de rotación para el multiplicador, ya sea cordic o multiplicador complejo.
- **escalamiento** señal de indicación de que en la etapa actual se realiza redondeo/truncamiento.

## 4.2. Arquitectura Radix-4

### 4.2.1. Descripción general

El algoritmo Radix-4 descompone el cómputo de una DFT de  $N$  puntos en  $\nu$  DFTs de 4 puntos cada una de forma que  $N = 4^\nu$ .

Partiendo de la definición de DFT:

$$\begin{aligned}
 X_k &= \sum_{n=0}^{N-1} x_n W_N^{kn} \\
 &= \sum_{n=0}^{\frac{N}{4}-1} x_n W_N^{kn} + \sum_{n=\frac{N}{4}}^{\frac{N}{2}-1} x_n W_N^{kn} + \sum_{n=\frac{N}{2}}^{\frac{3N}{4}-1} x_n W_N^{kn} + \sum_{n=\frac{3N}{4}}^{N-1} x_n W_N^{kn} \\
 &= \sum_{n=0}^{\frac{N}{4}-1} x_n W_N^{kn} + \sum_{n=0}^{\frac{N}{4}-1} x_{n+\frac{N}{4}} W_N^{k(n+\frac{N}{4})} + \sum_{n=0}^{\frac{N}{4}-1} x_{n+\frac{N}{2}} W_N^{k(n+\frac{N}{2})} + \sum_{n=0}^{\frac{N}{4}-1} x_{n+\frac{3N}{4}} W_N^{k(n+\frac{3N}{4})} \\
 &= \sum_{n=0}^{\frac{N}{4}-1} x_n W_N^{kn} + W_4^k \sum_{n=0}^{\frac{N}{4}-1} x_{n+\frac{N}{4}} W_N^{kn} + W_4^{2k} \sum_{n=0}^{\frac{N}{4}-1} x_{n+\frac{N}{2}} W_N^{kn} + W_4^{3k} \sum_{n=0}^{\frac{N}{4}-1} x_{n+\frac{3N}{4}} W_N^{kn} \\
 &= \sum_{n=0}^{\frac{N}{4}-1} (x_n + x_{n+\frac{N}{4}} W_4^k + x_{n+\frac{N}{2}} W_4^{2k} + x_{n+\frac{3N}{4}} W_4^{3k}) W_N^{kn}
 \end{aligned} \tag{4.3}$$

El resultado final en (4.3) se puede descomponer en cuatro subproblemas para los casos  $4k$ ,  $4k+1$ ,  $4k+2$  y  $4k+3$  dando lugar a:

$$\begin{aligned}
 Y_k = X_{4k} &= \sum_{n=0}^{\frac{N}{4}-1} (x_n + x_{n+\frac{N}{4}} W_4^{4k} + x_{n+\frac{N}{2}} W_4^{2*4k} + x_{n+\frac{3N}{4}} W_4^{3*4k}) W_N^{4kn} \\
 &= \sum_{n=0}^{\frac{N}{4}-1} (x_n + x_{n+\frac{N}{4}} + x_{n+\frac{N}{2}} + x_{n+\frac{3N}{4}}) W_N^{\frac{kn}{4}} \\
 &= \sum_{n=0}^{\frac{N}{4}-1} y_k W_N^{\frac{kn}{4}}, \quad k = 0, 1, \dots, \frac{N}{4} - 1
 \end{aligned} \tag{4.4}$$

$$\begin{aligned}
 Z_k = X_{4k+1} &= \sum_{n=0}^{\frac{N}{4}-1} (x_n + x_{n+\frac{N}{4}} W_4^{4k+1} + x_{n+\frac{N}{2}} W_4^{2*(4k+1)} + x_{n+\frac{3N}{4}} W_4^{3*(4k+1)}) W_N^{(4k+1)n} \\
 &= \sum_{n=0}^{\frac{N}{4}-1} ((x_n - x_{n+\frac{N}{2}}) - j(x_{n+\frac{N}{4}} - x_{n+\frac{3N}{4}})) W_N^k W_N^{\frac{kn}{4}} \\
 &= \sum_{n=0}^{\frac{N}{4}-1} z_k W_N^{\frac{kn}{4}}, \quad k = 0, 1, \dots, \frac{N}{4} - 1
 \end{aligned} \tag{4.5}$$

$$\begin{aligned}
G_k = X_{4k+2} &= \sum_{n=0}^{\frac{N}{4}-1} (x_n + x_{n+\frac{N}{4}} W_4^{4k+2} + x_{n+\frac{N}{2}} W_4^{2*(4k+2)} + x_{n+\frac{3N}{4}} W_4^{3*(4k+2)}) W_N^{(4k+2)n} \\
&= \sum_{n=0}^{\frac{N}{4}-1} ((x_n + x_{n+\frac{N}{2}}) - (x_{n+\frac{N}{4}} + x_{n+\frac{3N}{4}})) W_N^{2k} W_{\frac{N}{4}}^{kn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} g_k W_{\frac{N}{4}}^{kn}, \quad k = 0, 1, \dots, \frac{N}{4} - 1
\end{aligned} \tag{4.6}$$

$$\begin{aligned}
H_k = X_{4k+3} &= \sum_{n=0}^{\frac{N}{4}-1} (x_n + x_{n+\frac{N}{4}} W_4^{4k+3} + x_{n+\frac{N}{2}} W_4^{2*(4k+3)} + x_{n+\frac{3N}{4}} W_4^{3*(4k+3)}) W_N^{(4k+3)n} \\
&= \sum_{n=0}^{\frac{N}{4}-1} ((x_n - x_{n+\frac{N}{2}}) + j(x_{n+\frac{N}{4}} - x_{n+\frac{3N}{4}})) W_N^{3k} W_{\frac{N}{4}}^{kn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} h_k W_{\frac{N}{4}}^{kn}, \quad k = 0, 1, \dots, \frac{N}{4} - 1
\end{aligned} \tag{4.7}$$

Entonces, la unidad aritmética de la arquitectura radix-4 debe procesar cuatro puntos,  $x_n$ ,  $x_{n+\frac{l}{4}}$ ,  $x_{n+\frac{l}{2}}$  y  $x_{n+\frac{3l}{4}}$ , obteniéndose como resultado:

$$y_n = (x_n + x_{n+\frac{l}{4}} + x_{n+\frac{l}{2}} + x_{n+\frac{3l}{4}}) \quad k = 0, 1, \dots, \frac{N}{4} - 1 \tag{4.8}$$

$$z_n = ((x_n - x_{n+\frac{l}{2}}) - j(x_{n+\frac{l}{4}} - x_{n+\frac{3l}{4}})) W_N^k \quad k = 0, 1, \dots, \frac{N}{4} - 1 \tag{4.9}$$

$$g_n = ((x_n + x_{n+\frac{l}{2}}) - (x_{n+\frac{l}{4}} + x_{n+\frac{3l}{4}})) W_N^{2k} \quad k = 0, 1, \dots, \frac{N}{4} - 1 \tag{4.10}$$

$$h_n = ((x_n - x_{n+\frac{l}{2}}) + j(x_{n+\frac{l}{4}} - x_{n+\frac{3l}{4}})) W_N^{3k} \quad k = 0, 1, \dots, \frac{N}{4} - 1 \tag{4.11}$$

donde  $l$  depende de la etapa del cálculo a la que pertenece el cálculo y se define como sigue

$$\begin{aligned}
 l_1 &= N \\
 l_2 &= \log_4(N) \\
 l_3 &= \log_4(\log_4(N)) \\
 &\dots \\
 l_\nu &= 4
 \end{aligned} \tag{4.12}$$

donde  $l_i$  corresponde a la etapa  $i$ -ésima de una FFT de  $N = 4^\nu$  puntos.

En la figura 4.12 se muestra un esquema del cálculo de una radix-4 para 16 puntos, similar al esquema de la radix-2 de la figura 4.1, donde se ve el cálculo aritmético utilizando cuatro puntos como entrada y obteniendo cuatro puntos como salida.

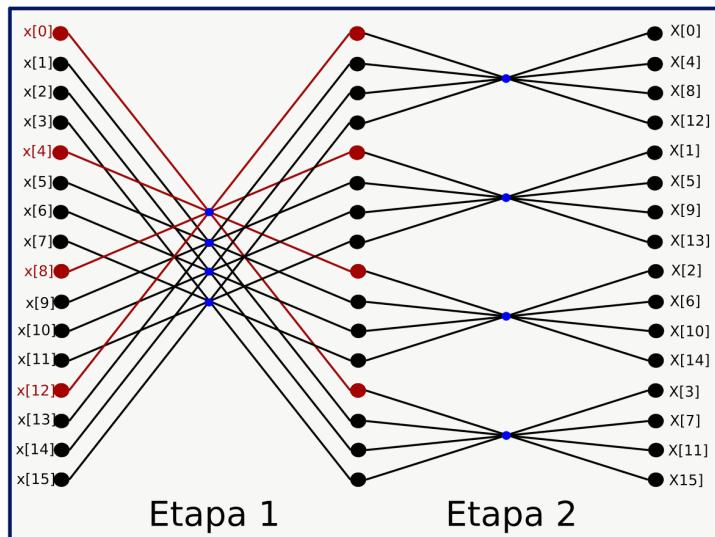


FIGURA 4.12: Esquema de una FFT radix-4 de 16 puntos

Este funcionamiento implica que para realizar un cálculo aritmético se deben tener almacenados en memoria tres puntos, que deben ser extraídos de memoria al entrar el cuarto punto para realizar el cálculo. Luego del cálculo se obtienen cuatro puntos de los cuales tres son almacenados en memoria mientras que el cuarto punto es utilizado en la etapa siguiente.

En la figura 4.13 se muestra un diagrama en bloques simplificado de la arquitectura radix-4 iterativa. Se diferencian claramente tres partes, la memoria, de tres entradas y tres salidas, la unidad aritmética conformada por el *dragonfly* (nombre que se le asigna al sumador/restador cuádruple que realiza las operaciones en la radix-4), el multiplicador, y la unidad de control. Esta arquitectura presenta un nivel superior de complejidad que la radix-2 que se traduce principalmente en una unidad de control más compleja y la unidad aritmética que puede ser un poco más lenta reduciendo así la frecuencia máxima de trabajo. Pero a su vez ofrece la ventaja de requerir la mitad de etapas que la radix-2 para realizar el cálculo de una FFT de la misma cantidad de puntos.

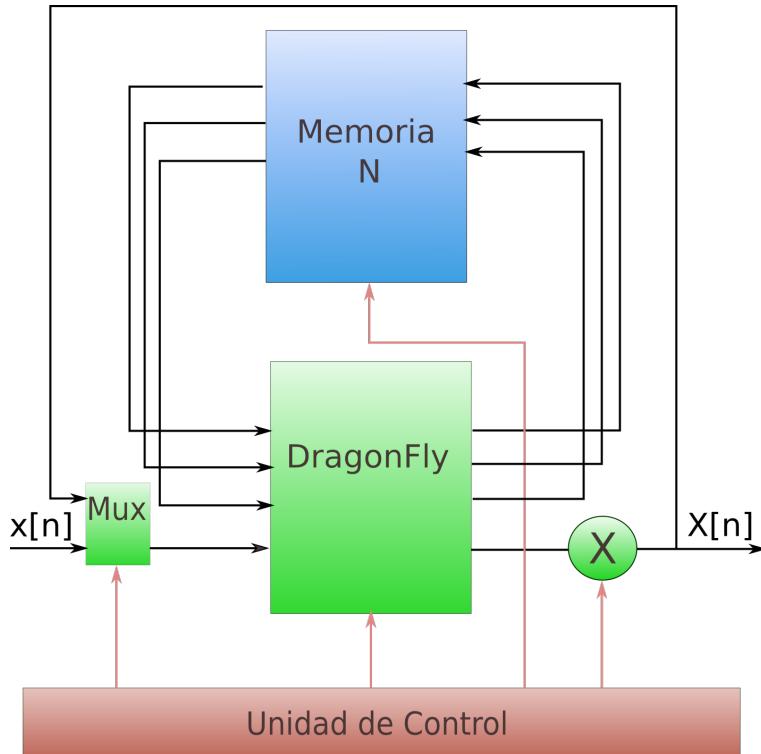


FIGURA 4.13: Diagrama simplificado de la arquitectura radix-4 iterativa

#### 4.2.2. Memoria

Como se explicó, en la arquitectura radix-4 se deben escribir y leer de memoria tres puntos en forma simultánea cada vez que se realiza una operación aritmética. En la figura 4.14 se presenta el bloque de memoria RAM de triple entrada y triple salida con sus puertos de conexión. Los valores entre corchetes indican el tamaño del bus correspondiente.

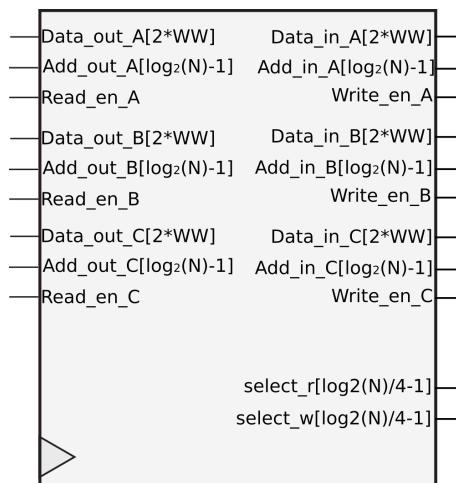


FIGURA 4.14: RAM de triple entrada y triple salida

El bloque tiene tres puertos de entrada y tres puertos de salida, cada uno con sus correspondientes señales de datos, dirección y habilitación. Las señales **select\_r**

y select\_w se utilizan para determinar, en conjunto con la señal de dirección de cada puerto, en que región de la memoria escribir.

Como es necesario poder leer y escribir en tres posiciones de memoria simultáneas se construye el bloque de memoria RAM de triple entrada y triple salida utilizando tres subbloques RAM de doble puerto igual a los utilizados en la radix-2 (subsección 4.1.2). Se hace de esta manera ya que al sintetizar en una FPGA, los bloques de memoria RAM de doble puerto se sintetizan utilizando los bloques RAM propios de la FPGA optimizando la utilización de recursos.

Cada uno de los tres subbloques RAM que componen el bloque triple tiene una capacidad de almacenamiento de  $N/3$  puntos. La distribución de los puntos en los tres subbloques se realiza de forma que en cada operación aritmética de cuatro puntos, haya un punto entrando a la unidad aritmética desde la entrada a la arquitectura o desde la etapa anterior y los restantes tres provenientes de cada uno de los subbloques RAM, para obtenerlos en forma simultánea. Del mismo modo, de los cuatro resultados de la operación aritmética, uno se reserva para la etapa siguiente y los tres restantes se almacenan cada uno en sendos subbloques RAM simultáneamente.

Como el acceso a las posiciones de memoria no es siempre a direcciones continuas, ya que al ser una arquitectura iterativa se ejecuta sucesivamente una operación de cada etapa, se direccionan los subbloques de memoria de manera que queden delimitadas las porciones de memoria correspondientes a cada etapa, como se muestra en la figura 4.15. Cada una de estas regiones es de tamaño igual a  $\frac{1}{4}$  de la longitud de la FFT de la etapa correspondiente, por ejemplo la porción de memoria correspondiente a la primer etapa será  $\frac{N}{4}$  en cada subbloque, comprendiendo en total  $\frac{3N}{4}$ , para la segunda etapa cada subbloque reservará  $\frac{\log_4(N)}{4}$  ya que cada FFT de la segunda etapa es de longitud  $\log_4(N)$ , así hasta llegar a la última etapa donde solo se reserva una posición de memoria de cada subbloque, almacenando en el bloque completo los tres puntos necesarios para realizar un cálculo aritmético cuando llegue el siguiente punto de la etapa anterior.

Para direccionar la memoria se utiliza un vector de dirección de  $\log_2(N) - 1$ , que si bien permitiría direccionar  $N/2$  posiciones de memoria aquí se lo utiliza para direccionar  $N/3$  ya que un vector de direcciones de  $\log_2(N) - 2$  solo permite direccionar  $N/4$  posiciones. Para obtener la dirección de la posición que se debe leer o escribir se utiliza como base el contenido de la entrada de dirección del subbloque correspondiente y se le aplica un enmascarado con el valor de la entrada de selección (select\_r o select\_w) que asignan valores a los bits de mayor valor de la palabra de dirección, permitiendo así leer o escribir en la posición respectiva a la dirección base en la región que corresponda a la etapa actual del cómputo de la FFT.

En la figura 4.15 se muestra esquemáticamente la división de cada subbloque en regiones a través del direccionado, y las direcciones respectivas al comienzo y final de cada región. Se ve claramente como quedan direcciones sin utilizar ya que con  $\log_2(N) - 1$  bits se podrían direccionar  $N/2$  posiciones de memoria, por lo que se utilizan únicamente  $2/3$  de las direcciones posibles.

A través de los controles de habilitación de lectura y escritura se puede acceder individualmente a cada subbloque RAM de la memoria de triple entrada y triple

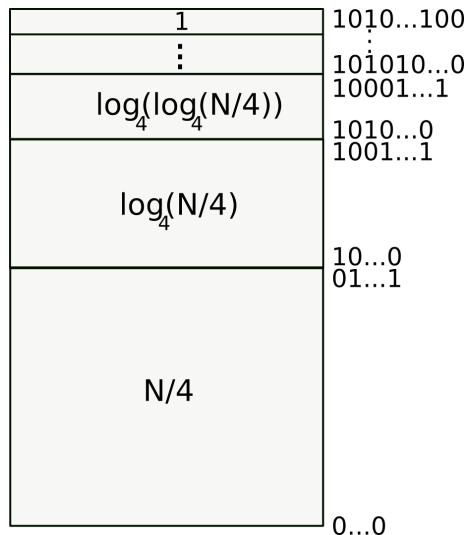


FIGURA 4.15: Esquema de direccionamiento de los subbloques RAM

salida.

#### 4.2.3. Sumador/restador

La unidad aritmética debe resolver las ecuaciones (4.8) a (4.11). Para esto se opera directamente transladando dichas ecuaciones a *verilog* teniendo en cuenta que multiplicar por  $j$  es equivalente a intercambiar las partes real e imaginaria (cambiando el signo de la última) del número complejo. Por la forma que toma cada operación de cuatro puntos en el diagrama de la figura 4.12 se le llama *dragonfly* a la unidad de cómputo aritmético de la radix-4.

En cada etapa del cómputo de la radix-4 pueden realizarse dos tipos de operaciones, una operación aritmética entre cuatro puntos o un movimiento de datos en memoria, como sucede en la radix-2. En el segundo caso, el movimiento a memoria puede ser el almacenamiento de un punto entrante a la arquitectura o proveniente de una etapa anterior, o la lectura de un punto de memoria para multiplicarlo por un twiddle factor y almacenarlo nuevamente en la región correspondiente a la etapa siguiente (corresponde al traspaso de un punto resultante de una operación aritmética de una etapa a la siguiente pasando por el multiplicador). Para las operaciones de movimientos de datos en memoria se dispone internamente en la unidad aritmética de un sistema de multiplexores que permiten direccionar la entrada del dato de la arquitectura a cualquiera de las tres salidas correspondientes a los tres subbloques de memoria para su almacenamiento, o direccionar cualquiera de las tres entradas desde memoria a la salida conectada a la entrada del multiplicador para realizar el traspaso de puntos de una etapa a la siguiente.

En la figura 4.16 se muestra el diagrama en bloque de la unidad aritmética. Se observa el arreglo de multiplexores que permite direccionar a las salidas a memoria los resultados de la operación aritmética o la entrada  $x$  del bloque. También se observa un multiplexor 4 – 1 que permite seleccionar la salida  $x$ , conectada al

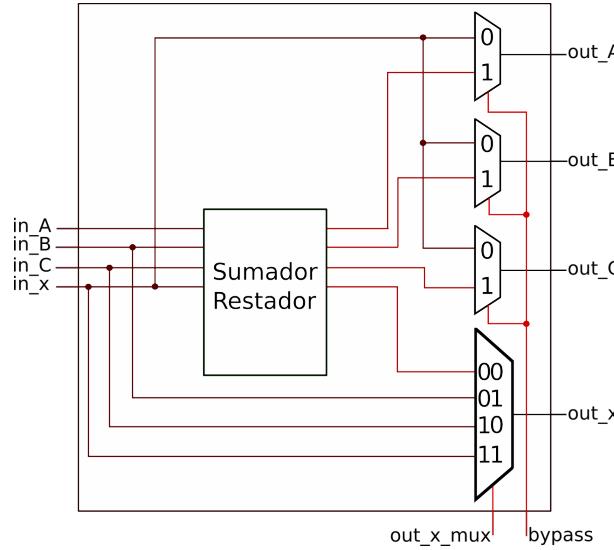


FIGURA 4.16: Diagrama de la unidad aritmética incluyendo los multiplexores de bypass

multiplicador, que permite seleccionar entre el resultado de la operación aritmética o una de las tres entradas de operandos  $A$ ,  $B$  o  $C$ . La unidad aritmética incluye los mecanismos para realizar el redondeo o truncamiento, controlados por dos señales: una señal de selección de redondeo o truncamiento y una señal de habilitación de redondeo/truncamiento individual por etapa. Este mecanismo se detalla en la subsección 4.3.3.

Para almacenar en memoria un dato entrante a la arquitectura se direcciona al subbloque de memoria correspondiente a través del módulo *dragonfly*, al igual que al extraer un dato de un subbloque de memoria para enviarlo a la salida de la arquitectura.

#### 4.2.4. Datapath

En la figura 4.17 se observa el datapath para la radix-4 iterativa. El bloque *dragonfly* concentra la unidad aritmética, el algoritmo de redondeo/truncado y el datapath interno que distribuye las entradas en las diferentes salidas. Los datos entrantes a cada subbloque de memoria pueden provenir desde la unidad aritmética como resultado de una operación o desde el *delay register*, proveniente de la etapa anterior.

El *datapath* es controlado por la unidad de control a través de los multiplexores de acceso a memoria, el multiplexor de entrada a la unidad aritmética y el *datapath* interno de la unidad aritmética por medio de sus señales de control.

En la figura 4.18 se muestran las posibles configuraciones para el *datapath* para operaciones de transferencia a memoria de acuerdo al tipo de etapa que se está procesando. Las líneas punteadas muestran caminos posibles dependiendo del subbloque de memoria donde se desea escribir o leer.

Durante estas operaciones, dependiendo de si la entrada es inicial, intermedia o final, se toma el dato de entrada de la arquitectura o de la etapa anterior y se envía a memoria, y se lee un dato de memoria y se envía al *delay register* para ser

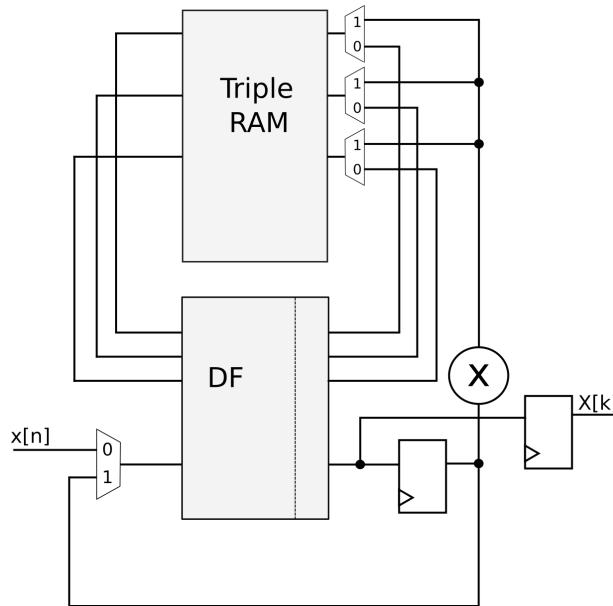


FIGURA 4.17: Datapath de la arquitectura radix-4 iterativa

luego multiplicado por el twiddle factor y ser guardado en memoria en la etapa siguiente, o enviado a la salida de la arquitectura en caso de estar ejecutándose la etapa final.

En la figura 4.19 se muestran las distintas configuraciones posibles para el *datapath* para operaciones aritméticas.

Durante estas operaciones se leen tres datos almacenados en memoria y un dato, que dependiendo de si la etapa que se está procesando es la inicial, una etapa intermedia o la etapa final, proviene de la entrada de la arquitectura o de la etapa anterior y se los procesa en la unidad aritmética. Una vez realizada la operación, tres de los resultados son almacenados en memoria y el restante se envía al *delay register* para su utilización en la etapa siguiente, en caso de estar procesando la etapa inicial o una etapa intermedia, o a la salida de la arquitectura en caso de ser la etapa final.

En la arquitectura radix-4 iterativa propuesta se pueden identificar tres tipos distintos de etapas, la etapa inicial, las intermedias y la final, en las que pueden ejecutarse una de dos posibles operaciones: una transferencia de un dato a memoria o una operación aritmética. El tipo de etapa que se está procesando y el tipo de operación que se realiza en esa etapa determinan la configuración del datapath en cada ciclo de *clock*.

A continuación se listan las posibles configuraciones de *datapath* para cada tipo de etapa y operación:

- Etapa inicial
  - Operaciones aritméticas: tres datos almacenados en memoria y el dato de entrada a la arquitectura. Tres de los resultados son almacenados en memoria mientras que el cuarto se envía al multiplicador.

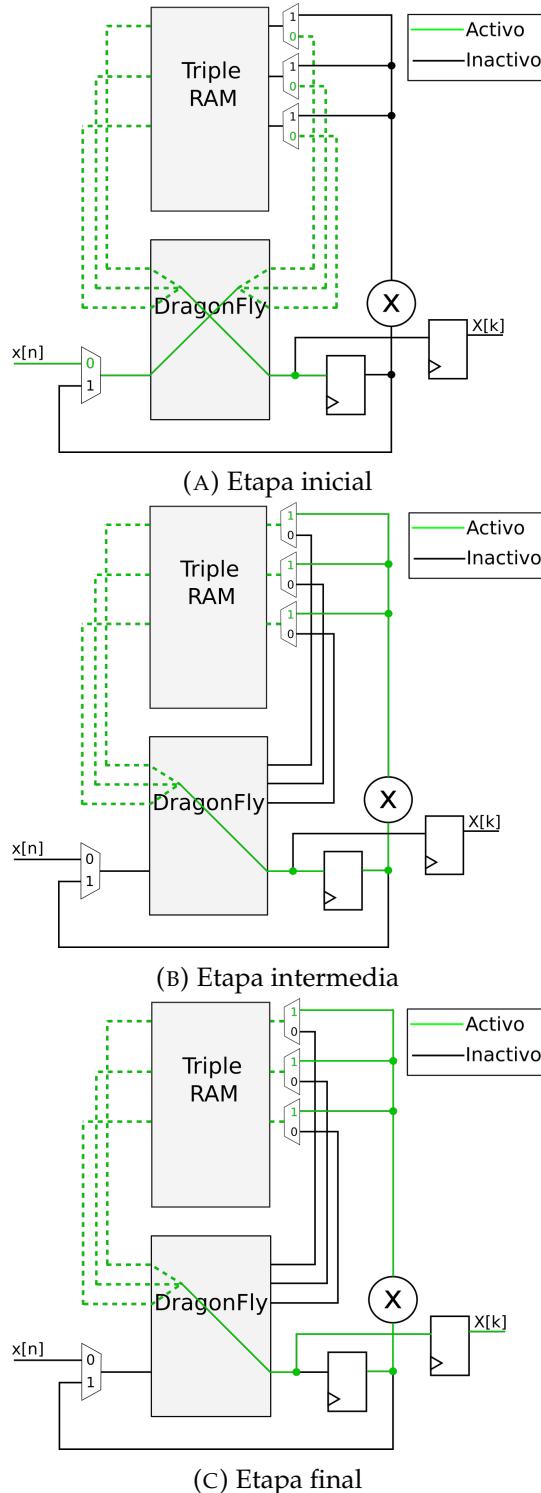


FIGURA 4.18: Datapath para operaciones de transferencia en memoria

- Transferencia a memoria: se almacena el dato entrante a la arquitectura en uno de los tres subbloques de memoria.
- Etapas intermedias
  - Operaciones aritméticas: tres datos almacenados en memoria y el dato

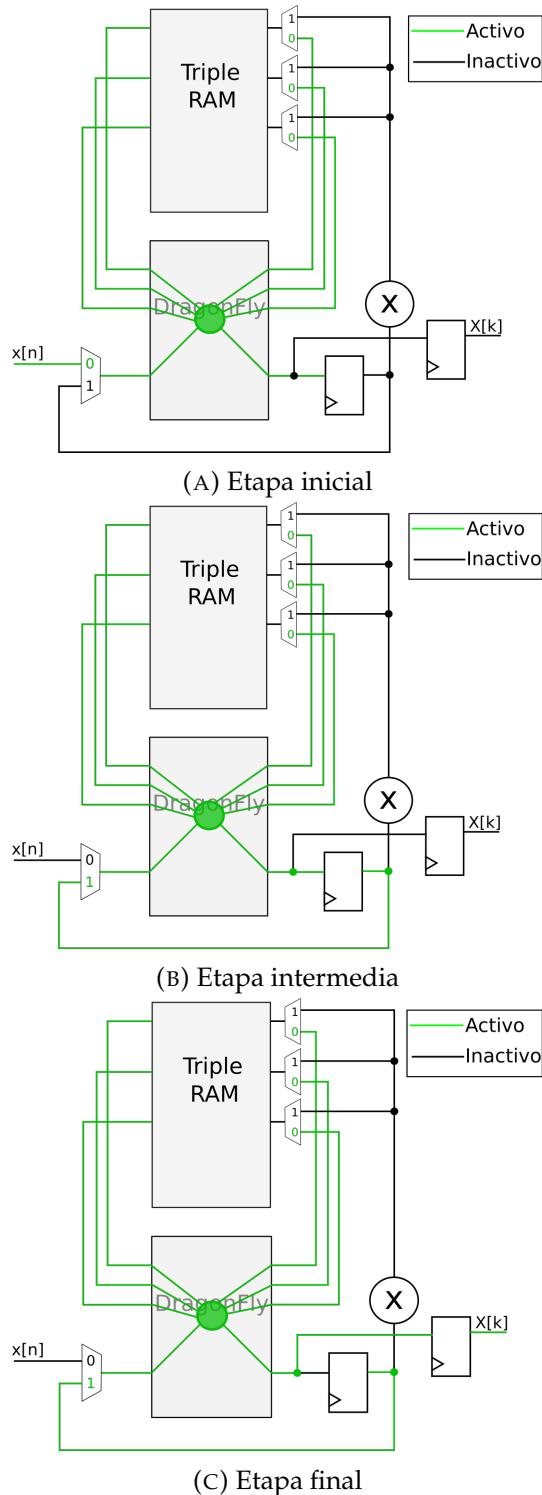


FIGURA 4.19: Datapath para operaciones en butterfly

de la etapa anterior almacenado en el *delay register*. Tres de los resultados son almacenados en memoria mientras que el cuarto se envía al multiplicador.

- Transferencia a memoria: se extrae un dato de memoria y se envía al multiplicador mientras se almacena en memoria el dato de la etapa anterior almacenado en el *delay register*.

- Etapa final

- Operaciones aritméticas: tres datos almacenados en memoria y el dato de la etapa anterior almacenado en el *delay register*. Tres de los resultados son almacenados en memoria mientras que el cuarto se envía a la salida de la arquitectura.
- Transferencia a memoria: se extrae un dato de memoria y se envía a la salida de la arquitectura mientras se almacena en memoria el dato de la etapa anterior almacenado en el *delay register*.

#### 4.2.5. Unidad de control

La unidad de control debe contener la lógica necesaria para controlar el funcionamiento de la arquitectura. Está compuesta por una máquina de estados principal que controla el funcionamiento general de la arquitectura, y una máquina de estados secundaria que configura el *datapath* de acuerdo al tipo de etapa y operación que se debe procesar.

La unidad de control además de configurar el *datapath* debe controlar el direccionamiento de la memoria, así como sus señales de control, y generar los *twiddle factors* para el multiplicador.

Al tratarse de una arquitectura radix-4, entra a la arquitectura un punto cada  $\log_4(N)$  ciclos de *clock*, y este es también el número de etapas de la arquitectura, se tiene un contador de longitud  $\log_2(\log_4(N))$  puntos para identificar el estado que se está procesando en cada ciclo de *clock*. El desborde de este contador alimenta un contador de longitud  $\log_2(N)$  que cuenta la cantidad de puntos que han ingresado a la arquitectura y permite controlar en qué estado del cómputo total se encuentra. Con estos dos contadores se lleva el control de la máquina de estados que controla el *datapath* y la memoria, y la generación de los *twiddle factors*.

Como se ve en la figura 4.12, para una radix-4 de 16 puntos, en la primer etapa hay que almacenar en memoria los primeros 12 puntos hasta realizar una operación aritmética con el decimotercer punto que entra, utilizando también el primer punto, el quinto y el noveno. En la segunda etapa se deben almacenar los primeros tres puntos que llegan y recién realizar la operación aritmética con el cuarto punto. Cada almacenamiento en memoria puede hacerse a uno de los tres subbloques, por lo que debe diferenciarse a qué subbloque pertenece cada punto que ingresa a una etapa.

Para decidir si se debe realizar una operación aritmética o una transferencia a memoria, y en este caso en qué subbloque se debe almacenar el dato, se utilizan dos bits del contador de puntos de la siguiente manera:

- [00] ->Almacenamiento en subbloque de memoria 1
- [01] ->Almacenamiento en subbloque de memoria 2
- [10] ->Almacenamiento en subbloque de memoria 3
- [11] ->Operación aritmética

Los dos bits del contador de puntos utilizados para determinar la operación a realizar dependen de la etapa, por lo que se seleccionan de acuerdo al `stg_ctr` como se muestra en la 4.20 para una arquitectura radix-4 de 256 puntos.

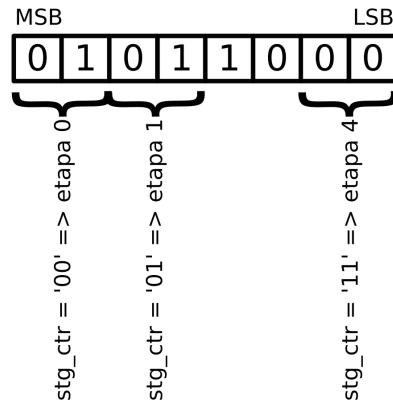


FIGURA 4.20: Selección del par de bits del contador de puntos a evaluar

### Máquinas de estados

En la figura 4.21 se muestra el diagrama de estados y transiciones la máquina de estados principal. El estado *Idle* es el estado inicial de la arquitectura. La señal *start* pasa la máquina al estado *Init* donde inicializa los parámetros de la arquitectura necesarios para comenzar a procesar y lee el primer dato de entrada a la arquitectura. Un ciclo de *clock* después la máquina pasa automáticamente al estado *enabled*.

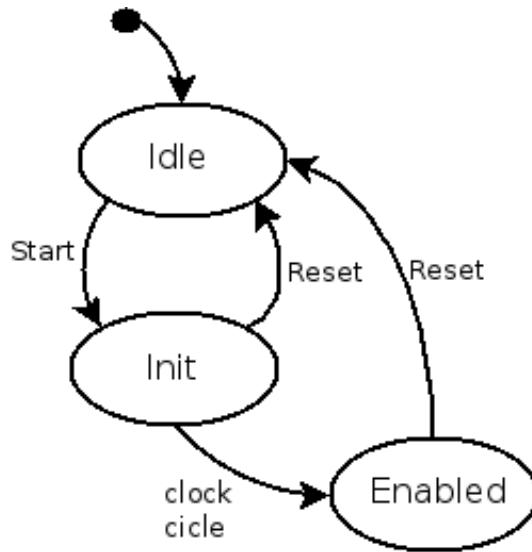


FIGURA 4.21: Diagrama de estados y transiciones de la máquina de estados principal

En el estado *Enabled* se identifica la etapa que se está procesando y se configura el *datapath* para realizar la operación correspondiente. Para esto se utiliza la máquina de estados secundaria, cuyo estado depende del valor de los bits del contador de puntos correspondientes a la etapa actual.

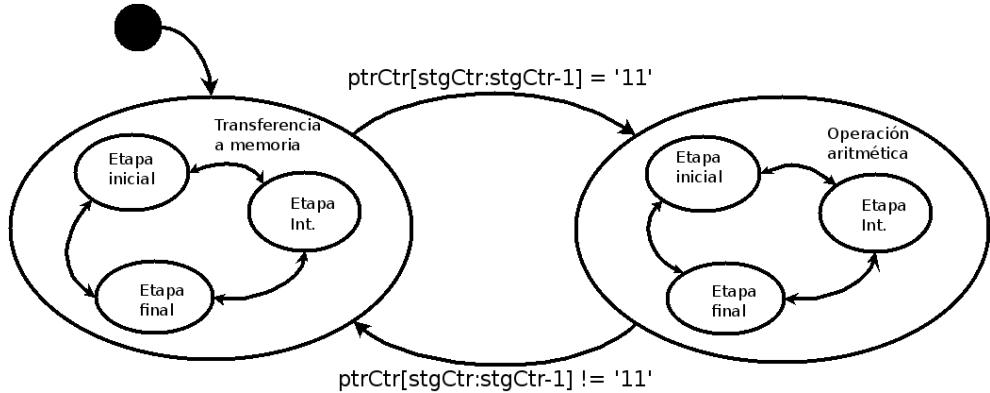


FIGURA 4.22: Diagrama de estados y transiciones de la máquina de estados secundaria

En la figura 4.22 se observa la máquina de estados secundaria. Dentro de cada uno de los estados principales se evalúa el tipo de etapa para la configuración del *datapath*. En cada uno de los estados principales funciona una submáquina de estados que realiza ajustes menores de acuerdo a si la etapa actual es la etapa inicial, una intermedia o la etapa final.  $\text{ptrCtr}[\text{stgCtr} : \text{stgCtr} - 1]$  hace referencia a los dos bits del contador de puntos correspondientes al valor del contador de etapas.

Esta máquina de estados controla además la señal de habilitación de escalamiento para la etapa actual de acuerdo al vector de escalamiento de entrada a la arquitectura. También controla, a través del contador de puntos y el de etapas, las señales de *handshaking* de salida, indicando si el dato de salida es un dato válido, señal *data\_valid* y si es el punto inicial o final de la FFT que se está procesando actualmente, señales *soo* y *done* respectivamente.

### Control de la memoria

El control de la memoria se realiza mediante los direccionamientos, las señales de habilitación de lectura y escritura, y las señales de selección de la región de memoria de cada subbloque donde se realizará la operación.

El direccionamiento de escritura se realiza directamente mapeando el valor del contador de puntos a la dirección de escritura. El direccionamiento de lectura se realiza mapeando a la dirección de lectura el valor del contador de puntos correspondiente al ciclo siguiente de *clock* ya que en cada flanco positivo del *clock* la memoria dispone a la salida el dato guardado en la dirección presente en el puerto de dirección de lectura durante el flanco.

Las señales de habilitación de lectura y escritura se controlan dependiendo del valor de los bits correspondientes del contador de puntos, habilitando el subbloque de memoria correspondiente para los casos de movimientos en memoria y habilitando los tres subbloques simultáneamente para el caso de operaciones aritméticas.

Las señales de selección de la región de memoria se controlan utilizando el contador de etapas, ya que cada región de memoria en un subbloque particular se utiliza para una etapa determinada. Las señales de selección se construyen como

una máscara para los bits superiores de la señal de direccionamiento, para ubicar la dirección indicada por el contador de puntos en la región correspondiente.

#### 4.2.6. Integración de la unidad de control

En la figura 4.23 se muestra el diagrama del *datapath* de la figura 4.17 con el agregado de las señales de la unidad de control.

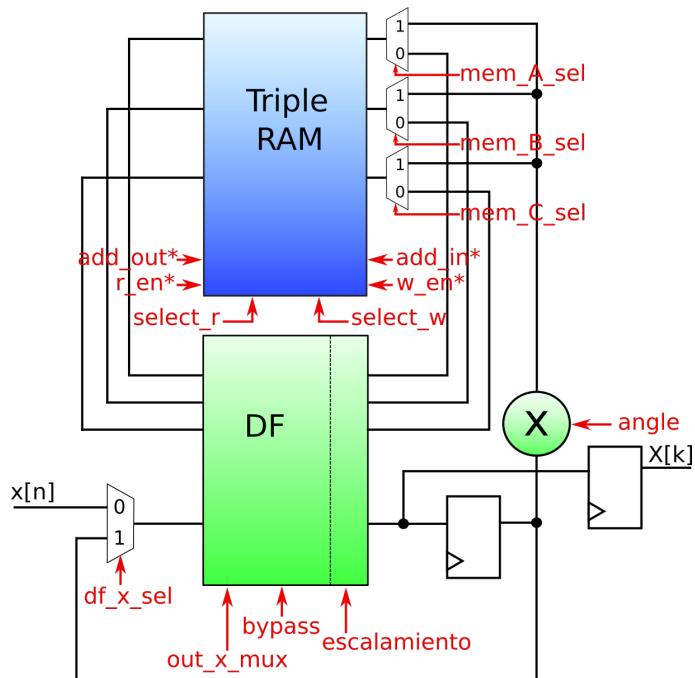


FIGURA 4.23: Datapath con las señales de control

Las señales de control de la figura 4.23 se listan a continuación, indicando entre paréntesis su tamaño si es mayor a 1:

- **`add_in*`** ( $\log_2(N)$ ) *address in*, dirección de memoria donde se escribirá el dato.
- **`w_en*`** *write enable*, señal de habilitación de escritura en memoria
- **`add_out*`** ( $\log_2(N)$ ) *address out*, dirección de memoria a la que se desea acceder.
- **`r_en*`** *read enable*, señal de habilitación de lectura de memoria
- **`select_r`** ( $\log_2(\log_4(N))$ ) Selección de la región de memoria donde se realiza la lectura.
- **`select_w`** ( $\log_2(\log_4(N))$ ) Selección de la región de memoria donde se realiza la escritura.
- **`mem_A_sel`** señal de control del multiplexor a la entrada al subbloque A de la memoria.
- **`mem_B_sel`** señal de control del multiplexor a la entrada al subbloque B de la memoria.

- **mem\_C\_sel** señal de control del multiplexor a la entrada al subbloque C de la memoria.
- **df\_x\_sel** señal de control del multiplexor de entrada a la unidad aritmética.
- **out\_x\_mux** (2) señal de control de la salida  $x$  de la unidad aritmética (ver esquema en figura 4.16).
- **out\_x\_mux** señal de control de las salidas a memoria de la unidad aritmética (ver esquema en figura 4.16).
- **angulo** ( $N + 1$ ) angulo de rotación para el multiplicador, ya sea cordic o multiplicador complejo.
- **escalamiento** señal de indicación de que en la etapa actual se realiza redondeo/ truncamiento.

Las señales correspondientes al direccionamiento de memoria, indicadas con un \*, se muestran unificadas para simplificar el diagrama, ya que cada subbloque tiene sus propias señales de control.

El bloque indicado como *DF* contiene la unidad aritmética, su *datapath* interno y la unidad de escalamiento que se describe en la sección 4.3.3.

## 4.3. Módulos compartidos por las dos arquitecturas

### 4.3.1. Cordic desenrollado

Como se explicó en la subsección 3.3.3, para el producto por los *twiddle factors* se implementan dos alternativas distintas. Una de ellas es a través de un módulo de cómputo del algoritmo cordic, según las ecuaciones desarrolladas en la subsección 3.3.3.

Se utiliza la versión desenrollada de la arquitectura cordic, ya que permite realizar el cálculo completo en un solo ciclo de *clock* y si es necesario se puede implementar en forma *pipelined* colocando registros entre las distintas etapas del cómputo cordic.

La forma de implementar el algoritmo es a través de submódulos consecutivos que realizan las microrotaciones hasta completar la rotación completa.

En la figura 4.24 se muestra el bloque de cómputo cordic con sus puertos de entradas y salidas.

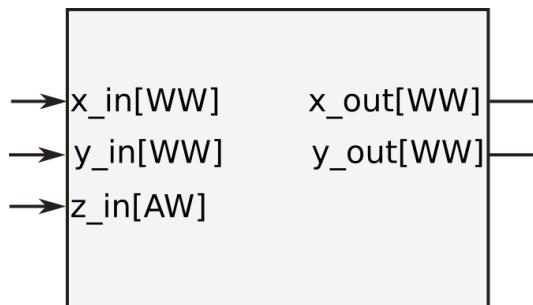


FIGURA 4.24: Bloque de módulo de cómputo cordic

Los parámetros *WW* y *AW* corresponden a los parámetros globales *Word\_Width*, el ancho de palabra de las entradas de puntos, y *Angle\_Width*, el ancho de palabra del ángulo de rotación.

Al módulo cordic entra un punto complejo en forma de vector de dos componentes, y el ángulo de rotación deseado, que representa el *twiddle factor*. Es por esto que las unidades de control de las dos arquitecturas radix implementadas generan los *twiddle factors* en forma de ángulos de rotación. La salida del módulo cordic es el vector de dos dimensiones rotado en el ángulo indicado.

Como se indica en la subsección 3.3.3 el algoritmo cordic tiene una cierta ganancia intrínseca que depende de la cantidad de iteraciones que se realizan. Como se desea que la ganancia del algoritmo sea 1 para no modificar la norma del vector de entrada se implementa un módulo de escalamiento a la salida del módulo cordic para compensar la ganancia de la arquitectura. El valor por el cual se escala la salida del módulo dependerá de la cantidad de iteraciones que se realicen, que es un parámetro global de la arquitectura cordic.

Para facilitar la rotación de los vectores de entrada, primero se analiza el ángulo de rotación para descomponerlo en un primer paso en rotaciones de  $90^\circ$  y luego se utiliza el algoritmo cordic para la rotación final menor al ángulo recto. De este modo, donde el algoritmo se aplica solo a rotaciones menores a  $90^\circ$ , se obtienen resultados más precisos con menor cantidad de rotaciones. También impacta en el tamaño total de la arquitectura, ya que se debe implementar una tabla con los valores de los  $\arctan(\alpha)$  para cada microrotación, por lo que a mayor número de iteraciones posibles mayor tamaño de la tabla de arcotangentes.

Para esto se implementa un preprocesador que analiza el ángulo de entrada y genera las primeras rotaciones de  $90^\circ$ ,  $180^\circ$  o  $270^\circ$  sobre el vector de entrada, que consisten en intercambios de las componentes y cambios de signos.

En la figura 4.25 se observa el diagrama en bloques interno del módulo de cómputo cordic. El bloque *Preprocessor* realiza el análisis del angulo y las rotaciones iniciales de  $90^\circ$  en caso de ser necesarias. El bloque *Unrolled cordic* realiza la rotación mediante el algoritmo cordic, que se muestra detallado en la figura 4.26, y los bloques *Post mult.* realizan el escalamiento final para compensar la ganancia propia del algoritmo cordic.

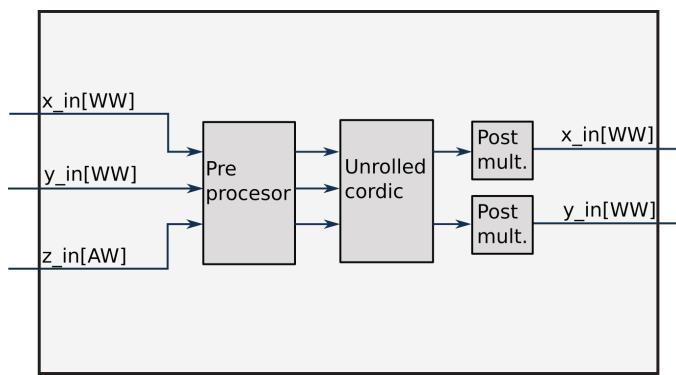


FIGURA 4.25: Diagrama en bloques del módulo cordic

Los bloques *uRot* de la figura 4.26 son los encargados de realizar las microrotaciones sucesivas de acuerdo al algoritmo cordic. Los valores de los  $\arctan(\alpha)$  se

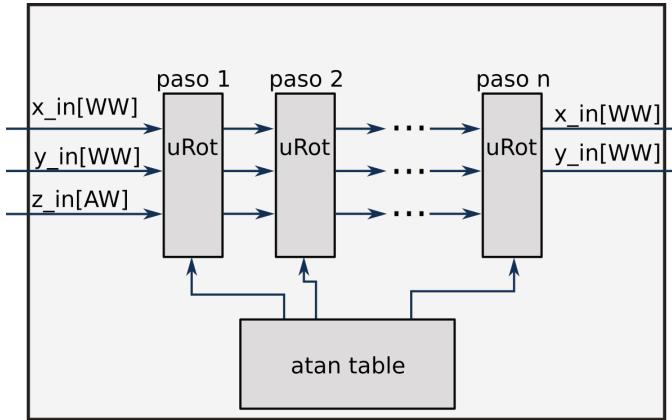


FIGURA 4.26: Diagrama en bloques del bloque de rotaciones del módulo cordic

almacenan previamente en la memoria *atan table* de acuerdo a la cantidad de iteraciones que se realizarán, y de allí son leídas por cada módulo de rotación.

### 4.3.2. Multiplicador complejo

La otra alternativa implementada para la multiplicación por los *twiddle factors* es a través de un multiplicador complejo. Para mantener la compatibilidad con el módulo cordic, y permitir que la elección entre uno u otro sea transparente para el resto de las arquitecturas, se utilizan las mismas interfaces en sus entradas y salidas, como se observa en la figura 4.27.

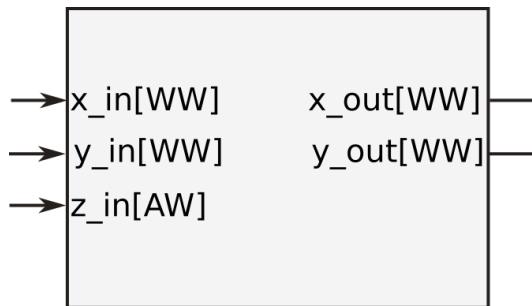


FIGURA 4.27: Bloque de módulo multiplicador complejo

El multiplicador recibe como entradas un vector de dos dimensiones,  $x_{in}$  e  $y_{in}$ , y el ángulo de rotación,  $z_{in}$ , y entrega a las salidas el vector rotado en  $x_{out}$  e  $y_{out}$ .

Para realizar la rotación a través de una multiplicación compleja, se almacenan los vectores complejos correspondientes a cada ángulo posible de rotación en una memoria. Los vectores almacenados tienen norma igual a 1 de manera que no afecten la norma del vector de entrada. Como los posibles ángulos de rotación son conocidos para una arquitectura y cantidad de puntos dados, se ordenan en memoria de forma que esta sea direccionada por el valor del ángulo generado por la unidad de control, obteniendo en cada posición el par de componentes del vector de norma 1 que corresponde al ángulo.

En la tabla 4.1 se muestran ejemplos de la codificación binaria de distintos ángulos para un ancho de palabra de ángulo de 7 bits.

Ángulo	Codificación binaria
180°	100000
90°	0100000
45°	0010000
135°	0110000

TABLA 4.1: Ejemplos de codificación del ángulo para un ancho de palabra del ángulo de 7 bits

Para reducir la cantidad de ángulos a almacenar se realiza un preprocesamiento previo donde se analiza el ángulo y se realizan rotaciones iniciales en pasos de 90° que consisten en intercambio de las componentes y cambios de signo, que son operaciones triviales, y luego el ángulo restante menor a 90° se resuelve mediante la multiplicación compleja. De esta manera, en memoria solo deben almacenarse ángulos menores a 90°.

Teniendo en cuenta que la variedad de ángulos a almacenar aumenta conforme aumenta la cantidad de puntos para la que se instancia la arquitectura, y además esa variedad aumenta añadiendo ángulos de tamaños cada vez menor, se implementa la tabla de forma que solo se crea del tamaño necesario para la cantidad de puntos específica de la arquitectura particular que se está instanciando. Por ejemplo, para una arquitectura radix-2 de 16 puntos, solo se necesita implementar la memoria con 4 valores, incluyendo el ángulo 0 en la posición de memoria 0. En cambio para una radix-2 de 64 puntos se necesitan 32 valores de ángulos distintos almacenados en memoria.

Para direccionarlos se utiliza la palabra del ángulo trasladado al primer cuadrante invertida, de modo que la posición de memoria correspondiente al ángulo 0 es la 0, la correspondiente al ángulo 45° (10...0) es la 1 (00...01), y así sucesivamente.

Al utilizar una transformación del ángulo a su vector normalizado correspondiente, no es necesario un escalamiento luego de multiplicar.

### 4.3.3. Unidad de escalamiento

La unidad de escalamiento de las unidades aritméticas de cada arquitectura permite realizar un escalamiento del resultado de la operación aritmética en una o varias etapas seleccionadas en forma dinámica, mediante una señal de entrada, para evitar posibles *overflow* a lo largo del cómputo de la FFT, ya que la aritmética de la arquitectura es de punto fijo en palabras de longitud fija.

Las opciones de escalamiento disponibles, como se explicó en la subsección 3.3.4, son redondeo y truncamiento.

En la figura 4.28 se muestra el diagrama en bloques de la unidad de escalamiento.

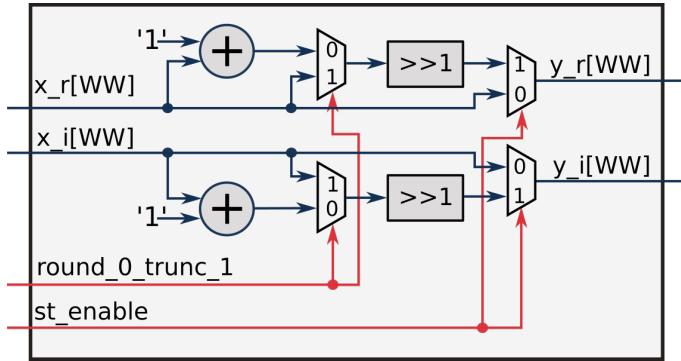


FIGURA 4.28: Diagrama en bloques de la unidad de escalamiento

La unidad le suma 1 a cada una de las entradas. Con un multiplexor controlado por la señal *round\_0\_trunc\_1* se elige si se desplaza a derecha un bit la señal original o la salida del sumador, eligiendo así entre truncamiento o redondeo respectivamente. Luego, la salida de la unidad será la señal original, en caso de no estar habilitado el escalamiento para esa etapa, o la señal desplazada, en caso de estar habilitado el escalamiento para esa etapa, realizando la selección mediante dos multiplexores controlados por la señal *st\_enable*.

#### 4.4. Interfaces de las arquitecturas

Las dos arquitecturas implementadas tienen los mismos puertos de entrada y salida mostrados en la figura 4.29.

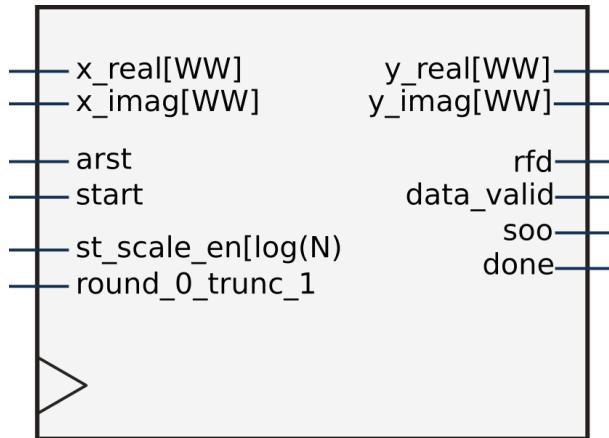


FIGURA 4.29: Señales de comunicación de las arquitecturas implementadas

Los puertos se describen en el siguiente listado:

- **x\_real** (*WORD\_WIDTH*) componente real del punto de entrada.
- **x\_imag** (*WORD\_WIDTH*) componente imaginaria del punto de entrada.
- **rst** señal de reset asincrónico. Reinicia la arquitectura.

- **start** Señal de comienzo de funcionamiento luego de un reinicio. Al detectar la señal la arquitectura lee el primer dato de entrada y comienza con el cómputo de la primer FFT.
- **st\_scale\_en** ( $(\log_2(N))$ ) Habilitación del escalamiento posterior a la operación aritmética. Un '1' en un bit habilita el escalamiento en la etapa correspondiente.
- **round\_0\_trunc\_1** Selección del tipo de escalamiento posterior a la operación aritmética, redondeo o truncamiento.
- **y\_real** (*WORD\_WIDTH*) componente real del punto de salida.
- **y\_imag** (*WORD\_WIDTH*) componente imaginaria del punto de salida.
- **rfd** señal que indica que debe colocarse un nuevo punto en la entrada de la arquitectura.
- **data\_valid** señal que indica que el dato presente en la salida es válido.
- **soo** señal que indica que el dato presente en la salida es el primero de una nueva FFT.
- **done** señal que indica que el dato presente en la salida es el último de la actual FFT.

Se listan a continuación los parámetros globales de las arquitecturas implementadas.

- **WORD\_WIDTH** Cantidad de bits de codificación de los puntos de entrada y salida.
- **CLOG2\_FFT\_POINTS** Logaritmo, en base dos, de la cantidad de puntos para la que es implementada la arquitectura. A través de este parámetro se infiere dentro de la arquitectura la cantidad de puntos de la FFT a procesar.
- **FFT\_1IFFT\_0** selección de implementación de la arquitectura para realizar FFT o IFFT.
- **ANGLE\_WIDTH** Cantidad de bits de codificación de los ángulos de representación de los *twiddle factors*.

## 4.5. Herramientas utilizadas para el desarrollo

A continuación se hace una breve reseña de las herramientas utilizadas para la implementación de las arquitecturas:

### Sublime Text 2

Sublime fue el editor de texto utilizado para codificar en lenguaje Verilog. El mismo presenta una interfaz gráfica con diferentes características de gran utilidad, marcado de palabras claves, autocompletado, etc.

### ModelSim

**ModelSim.**

ModelSim fue la herramienta utilizada para la simulación de los códigos descriptos en Verilog, con los cuales se generaron archivos de tipo wave de salida.

**gtkwave**

Gtkwave fue la herramienta utilizada para visualizar los archivos de tipo wave que fueron generados durante las simulaciones realizadas en ModelSim.

**Xilinx ISE**

Xilinx ISE es la herramienta proporcionada por la compañía Xilinx para crear los archivos de síntesis “.bit” a partir de un hardware codificado en un lenguaje HDL, para ser implementado en los dispositivos FPGA que provee. También fue utilizada para transferir los archivos “.bit” al kit de desarrollo FPGA.

**Bitbucket/Mercurial**

Para realizar el desarrollo se utilizó un repositorio hospedado en <http://www.bitbucket.org>, el cual fue utilizado a través de la herramienta de versionado Mercurial. Dicho sistema aportó diversas ventajas, entre las cuales principalmente se encuentran el mantener un registro de todas las modificaciones realizadas entre cada una de las versiones y el poder trabajar en la nube.





## Capítulo 5

# Estudio, Simulación y Validación de los IP Cores Generados

En este capítulo se describen los ensayos de verificación y validación de los IP Cores generados durante el desarrollo del presente trabajo de tesis.

### 5.1. Estrategia de simulación, verificación y validación

Durante el desarrollo de este trabajo se verificó el funcionamiento de cada módulo a través de simulaciones en verilog comprobando su correcto funcionamiento.

Dada la imposibilidad de ensayar todos los casos, por temas de tiempo, se diseñaron casos de prueba específicos que son una muestra representativa del conjunto total.

Una vez integrada cada arquitectura se realizaron ensayos de verificación y validación. Mediante bancos de prueba en verilog se procesaron señales características y se compararon con el resultado de realizar el mismo procesamiento utilizando Matlab para comprobar el correcto funcionamiento de las arquitecturas finalizadas.

Para caracterizar las diferentes arquitecturas se eligieron las siguientes métricas:

- Error máximo  $\equiv \|error\|_\infty$
- MSE  $\equiv \|error\|_2$
- Distorsión Total Armónica

Una vez verificado el funcionamiento a nivel general corroborando que transforma señales conocidas en sus transformadas y antitransformadas correspondientes, se realizaron ensayos de caracterización mediante simulaciones a través de bancos de prueba en verilog donde se analizó iterativamente el funcionamiento de las arquitecturas para obtener parámetros de caracterización como la distorsión total armónica y una métrica del error de la arquitectura tomando como referencia el procesamiento de las mismas señales en Matlab.

Se realizaron simulaciones para diferentes tamaños de palabra y cantidad de puntos. Para la cantidad de bits por palabra se utilizaron los valores de 12 y 16 bits. Estos valores fueron seleccionados teniendo en cuenta por un lado que 16 bits es un valor standard de codificación y permite la comparación directa con unidades de cómputo en otros lenguajes como C++, y 12 bits por ser un valor común

en lo que se refiere a transmisiones OFDM. En cuanto a la cantidad de puntos a procesar se realizaron simulaciones para 1024 y 4096 puntos, al ser cantidades de puntos que pueden procesarse tanto con la arquitectura radix-2 como con la radix-4.

Todas las simulaciones fueron realizadas utilizando *Modelsim* y *gtkwave*, ambos mencionados en la sección 4.5, para la visualización de las señales resultantes.

## 5.2. Simulación y verificación de los módulos individuales

Al finalizar la implementación de cada módulo se realizaron pruebas elementales de funcionamiento utilizando bancos de prueba escritos en verilog donde se buscó verificar que cada módulo cumpla con su función.

En este sentido los módulos sobre los que se realizaron mayores pruebas son los módulos multiplicadores, tanto el módulo cordic como el multiplicador complejo. También se realizaron ensayos de funcionamiento durante el desarrollo de las unidades de control para depurar la sincronización correcta de las señales de control.

Una vez finalizados los ensayos y simulaciones con resultados satisfactorios se procedió a la integración de las arquitecturas.

## 5.3. Simulación y validación de las arquitecturas complejas

Los IP cores generados como resultado del desarrollo del trabajo de tesis fueron sometidos a simulaciones y ensayos para verificar su correcto funcionamiento de acuerdo a los requerimientos planteados y obtener cotas de error y distorsión para validar el desarrollo.

### 5.3.1. Procesamiento de señales patrón

Se realizaron una serie de simulaciones utilizando señales patrón cuyas transformadas de Fourier son conocidas de forma de validar el funcionamiento de las arquitecturas. Para cada señal utilizada se presenta un gráfico con el resultado de la simulación superpuesto al resultado de realizar la transformada de Fourier sobre la misma señal utilizando Matlab. Las arquitecturas se ensayan utilizando un ancho de palabra de 16 bits, para 4096 puntos.

#### Delta en componente '0'

Utilizando como entrada una señal compuesta por un único pulso en la primer posición debe dar como resultado una señal continua de valor constante.

La entrada consistió en una señal discreta de 4096 puntos representando el muestreo de una delta en la componente 0.

En la figura 5.1 se observan las salidas obtenidas utilizando ambas arquitecturas tanto con el multiplicador complejo como con el rotador cordic. En cada gráfico se superpone la salida de la arquitectura, representada por la señal *core* en los gráficos, a la salida de realizar la misma fft utilizando precisión de punto flotante como referencia.

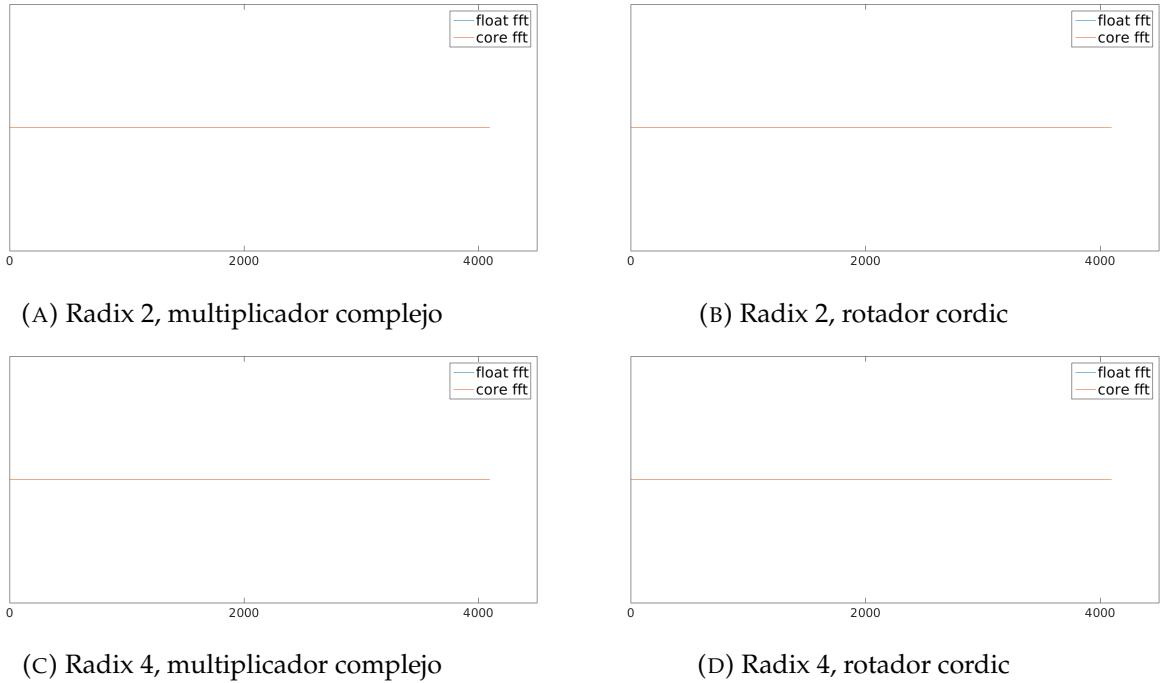


FIGURA 5.1: Respuestas a una delta en la componente 0 para las arquitecturas radix-2 y radix-4

## Delta

La transformada de Fourier de una delta debe dar un seno o un coseno cuya frecuencia está directamente relacionada con la posición del pulso.

La entrada utilizada para este ensayo consistió en una delta ubicada en la componente 6 del vector de 4094 puntos.

En la figura 5.2 se muestran las salidas obtenidas de ambas arquitecturas tanto con el multiplicador complejo como el rotador cordic. En cada gráfico se superpone la salida de la arquitectura, representada por la señal *core* en los gráficos, a la salida de realizar la misma fft utilizando precisión de punto flotante como referencia.

### 5.3.2. Medición del error

Para medir el error de procesamiento de las arquitecturas se utilizó como parámetro el resultado de realizar el mismo procesamiento mediante Matlab, ya que al utilizar precisión en punto flotante de 64 bits provee un buen contraste para la precisión en entero de 12 o 16 bits de las arquitecturas.

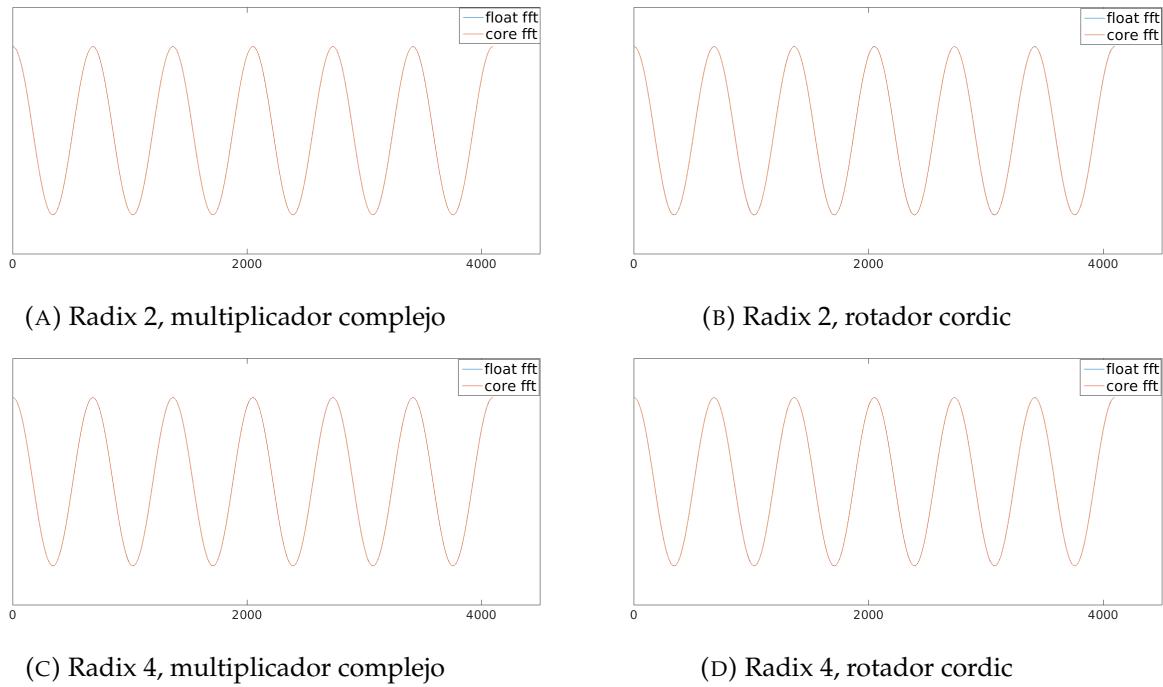


FIGURA 5.2: Respuestas a una delta en la componente 7 para las arquitecturas radix-2 y radix-4

Para obtener un parámetro comparable del error se obtienen dos métricas basadas en el cálculo de la norma de los vectores de señal y de error. Definiendo la norma  $p$  de un vector  $x$  de tamaño  $N$  como:

$$\|\vec{x}\|_p = \sqrt[p]{x[1]^p + x[2]^p + \dots + x[N]^p} \quad (5.1)$$

Se toma el error relativo en base a dos normas calculadas según (5.1), dando las métricas  $E_\infty$  y  $E_2$  como sigue:

$$E_\infty = MAX \left( \frac{|X_o[n] - X_{dut}[n]|}{X_o[n]} \right) \quad (5.2)$$

$$E_2 = \left\| \frac{X_o[n] - X_{dut}[n]}{X_o[n]} \right\|_2 \quad (5.3)$$

permitiendo estas métricas tener una medida del error ponderable y comparable.

Teniendo en cuenta que las arquitecturas implementadas se comportan como sistemas no lineales, para obtener métricas confiables de error cada simulación consistió en 1024 corridas con vectores de entrada generados aleatoriamente en cada corrida. Se calcularon las métricas  $E_\infty$  y  $E_2$  para cada corrida y luego se promedió el valor de las métricas de todas las corridas obteniendo así los valores de error de cada simulación.

En la figura 5.3 se presenta un diagrama de flujo del *script* de simulación para la estimación del error.

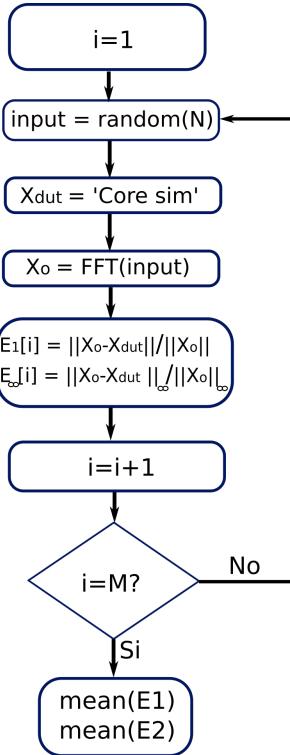


FIGURA 5.3: Diagrama de flujo de la simulación para la estimación del error

Se simularon 8 esquemas distintos para cada arquitectura, donde se alternaron los dos módulos multiplicadores, la unidad cordic y el multiplicador complejo, con dos anchos de palabra, 12 y 16 bits, para dos cantidades de puntos diferentes, 1024 y 4096. Además se realizaron dos simulaciones más utilizando una unidad de cómputo de FFT desarrollada por terceros ampliamente difundida en lenguaje C++, conocida como KISS FFT [5], con precisión de punto fijo de 16 bits.

En la tabla 5.1 se muestran los resultados de las mediciones de error utilizando la métrica  $E_\infty$ , mientras que en la tabla 5.2 se muestran las mediciones de error utilizando la métrica  $E_2$ .

	1024, 12 bits	1024, 16 bits	4096, 12 bits	4096, 16 bits
Radix-2, cordic	0,092	0,006	0,099	0,008
Radix-2, Mult.	0,232	0,003	0,340	0,108
Radix-4, cordic	0,077	0,003	0,074	0,007
Radix-4, Mult.	0,224	0,002	0,334	0,105
Kiss FFT		0,017		0,035

TABLA 5.1: Métrica  $E_\infty$  para 1024 realizaciones de cada arquitectura con entradas aleatorias

Se puede ver que el desempeño de las distintas variantes de las arquitecturas es aceptable, incluso comparado con el algoritmo FFT implementado en C++.

Se puede ver que para 12 bits el algoritmo cordic produce menor error que el multiplicador complejo. Esto se debe a que el efecto de la cuantización se hace más evidente en el multiplicador ya que al implementar los factores de multiplicación

	<b>1024, 12 bits</b>	<b>1024, 16 bits</b>	<b>4096, 12 bits</b>	<b>4096, 16 bits</b>
<b>Radix-2, cordic</b>	0,095	0,007	0,116	0,053
<b>Radix-2, Mult.</b>	0,257	0,004	0,356	0,131
<b>Radix-4, cordic</b>	0,084	0,002	0,094	0,027
<b>Radix-4, Mult.</b>	0,258	0,003	0,358	0,126
<b>Kiss FFT</b>		0,017		0,035

TABLA 5.2: Métrica  $E_2$  para 1024 corridas de cada arquitectura con entradas aleatorias

correspondientes a cada ángulo, el error de cuantización es mayor que al utilizar solo el ángulo como en el caso del cordic. Además, el hecho de realizar una multiplicación entre dos valores con precisión de punto fijo el error del resultado es mayor que en el caso de sumas y desplazamientos de una sola palabra como sucede en el cordic. En cambio, para un tamaño de palabra de 16 bits la magnitud del error de ambos sistemas de multiplicación por los twiddle factors es equivalente ya que es menos significativo el error de cuantización de los factores del multiplicador.

Para reducir el error del multiplicador complejo se debe aumentar el tamaño de palabra con el que se almacenan los factores correspondientes a cada ángulo, lo que aumenta el tamaño de memoria utilizada y el tamaño de los registros con los que se realizan las operaciones. Para reducir el error del rotador cordic se puede aumentar la cantidad de iteraciones del algoritmo, pero esto tiene como consecuencia un aumento en el tiempo consumido para realizar la rotación y un aumento en la memoria destinada a almacenar los valores de  $\arctan(2^{-i})$ .

Tambien se puede observar que el error para las arquitecturas de 4096 puntos es mayor que el error para las arquitecturas de 1024 puntos. Esto se debe a que para procesar mayor cantidad de puntos se requiere mayor cantidad de etapas, lo que lleva a mayor cantidad de operaciones aritméticas que contienen error, que se acumula y se arrastra de una etapa a la siguiente. De este modo, mientras más etapas tenga la arquitectura mayor será el error de la misma. Por este mismo motivo las arquitecturas radix-4 presentan menor error que las radix-2, ya que para la misma cantidad de puntos poseen menor número de etapas, lo que resulta en menos multiplicaciones. Esto último representa la principal ventaja de la arquitectura radix-4 por sobre la radix-2 independientemente del tipo de implementación que se utilice.

La diferencia en el error para los dos tamaños de palabras ensayados se debe al error de cuantización producto de la diferencia de precisión entre los dos tamaños. Mientras que con 12 bits la cuantización se da en pasos de  $\frac{1}{2^{12}-1} = 4,88 \times 10^{-4}$ , para 16 bits es de  $\frac{1}{2^{16}-1} = 3,05 \times 10^{-5}$ .

También se observa que para 1024 puntos el error de las arquitecturas iterativas es menor que el de la Kiss FFT, mientras que para 4096 puntos el error de las arquitecturas utilizando cordic es del orden del error de la implementación en C++. Estos resultados verifican que las arquitecturas resultantes del presente trabajo de tesis son aptas para su uso práctico, al menos en cuanto al error que producen. En cuanto a los resultados de error para las arquitecturas con multiplicador complejo de 4096 puntos, se debería aumentar el ancho de palabra de los factores de multiplicación almacenados en memoria para mejorar las métricas obtenidas.

### 5.3.3. Medición de la distorsión total armónica

La distorsión total armónica (THD por sus siglas en inglés, Total Harmonic Distortion) es la aparición de componentes espúreas en la salida de un sistema no lineal al aplicar a la entrada un tono puro. En el caso del presente trabajo de tesis el procesamiento de datos comprende el cómputo de una FFT o iFFT, por lo que la aparición de componentes espúreas tanto en frecuencia como en muestras temporales debe mantenerse en el nivel mínimo posible ya que puede distorionar completamente el resultado del procesamiento, además de ser una medida de la no linealidad del sistema.

La distorsión total armónica se calcula como:

$$THD = \frac{\sum \text{Potencia de los armónicos}}{\text{Potencia del tono fundamental}} = \frac{P_1 + P_2 + \dots + P_N}{P_0} \quad (5.4)$$

Donde  $P_0$  es la potencia del tono fundamental de frecuencia  $f_0$  y  $P_1 \dots P_N$  la potencia de los armónicos de frecuencia  $2 * f_0 \dots N * f_0$ .

Para obtener una caracterización de la THD se utilizó un banco de prueba en verilog que permite leer como entrada valores complejos listados en un archivo de texto, realizar el cómputo de la FFT mediante la arquitectura a ensayar y escribir la salida en un archivo de texto. La simulación completa se llevó a cabo mediante un *script* en Matlab que crea un archivo de texto que funciona como entrada de la arquitectura, llama a la simulación de la arquitectura mediante el banco de prueba descripto y luego procesa la salida de la simulación para hallar el valor de la THD, realizando su transformada de Fourier en un sistema de mayor precisión y analizando sus componentes. Para lograr una caracterización lo más completa posible se realizaron simulaciones sucesivas recorriendo de a uno por vez todos los tonos de entrada posibles y registrando el valor de THD para cada uno, de forma de obtener el valor para cada tono de entrada.

La forma de calcular la THD es la siguiente:

$$THD = 10 * \log\left(\frac{\sum_1^N A_i^2}{A_0^2}\right) \quad (5.5)$$

donde  $A_i$  es la amplitud de la componente de frecuencia  $f_i$ , dando el resultado del cálculo en dB.

En la figura 5.4 se muestra un diagrama de flujo de la simulación para la estimación de la THD. *Core sim* indica la simulación del IP Core mediante Modelsim, el resto de las acciones son llevadas a cabo en Matlab.

De este modo se llevaron a cabo 16 simulaciones, combinando los dos valores de tamaño de palabra y los dos valores de cantidad de puntos, para cada una de las dos arquitecturas desarrolladas utilizando como unidad de multiplicación por los *twiddle factors* el módulo cordic y el multiplicador complejo. Además se realizó el mismo análisis para una arquitectura KISS FFT utilizando tamaño de palabra de 16 bits en punto fijo.

En la figura 5.5 se muestra la THD para cada tono utilizado como entrada, para la arquitectura radix-2 iterativa para 1024 puntos, tanto para 12 bits como para 16 bits de ancho de palabra, utilizando el rotador cordic y el multiplicador complejo.

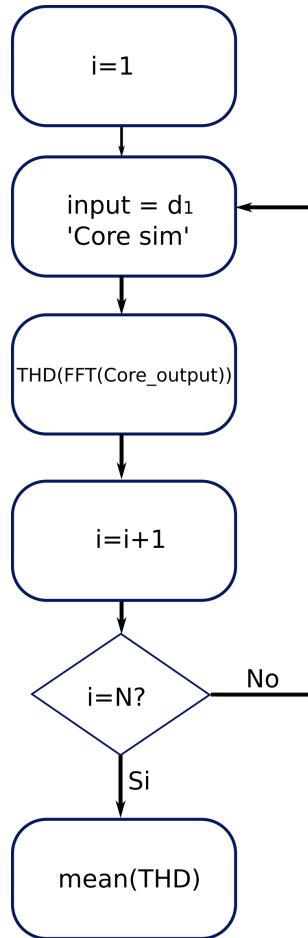


FIGURA 5.4: Diagrama de flujo de la simulación para la estimación de la THD

En la figura 5.6 se ven los gráficos de la THD para cada tono utilizado como entrada, para la arquitectura radix-2 iterativa para 4096 puntos, tanto para 12 bits como para 16 bits de ancho de palabra, utilizando el rotador cordic y el multiplicador complejo.

En la figura 5.7 se ven los gráficos de la THD para cada tono utilizado como entrada, para la arquitectura radix-4 iterativa para 1024 puntos, tanto para 12 bits como para 16 bits de ancho de palabra, utilizando el rotador cordic y el multiplicador complejo.

En la figura 5.8 se ven los gráficos de la THD para cada tono utilizado como entrada, para la arquitectura radix-4 iterativa para 4096 puntos, tanto para 12 bits como para 16 bits de ancho de palabra, utilizando el rotador cordic y el multiplicador complejo.

En la figura 5.9 se ven los gráficos de la THD para cada tono utilizado como entrada, para la arquitectura KISS FFT en C++ para 16 bits de ancho de palabra, para 1024 y 4096 puntos.

Se puede ver en todos los casos que la THD disminuye a niveles casi insignificantes para tonos en el centro del ancho de banda, subiendo luego a valores más significativos en los tonos en los límites del ancho de banda.

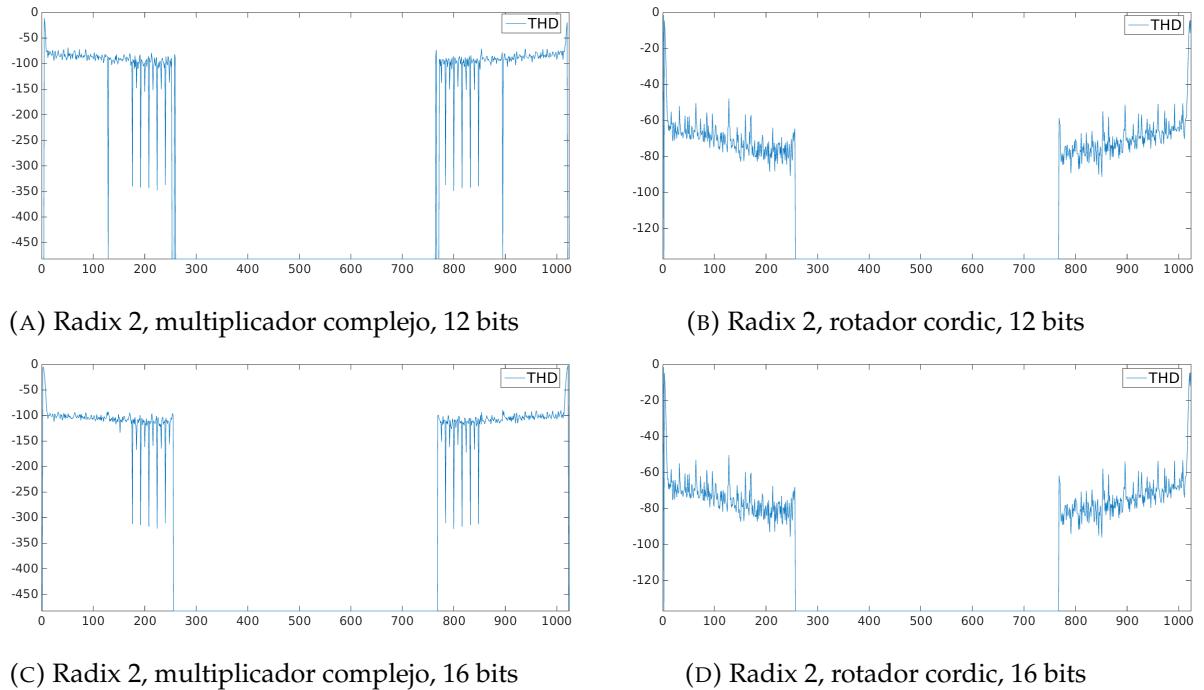


FIGURA 5.5: THD en función del tono de entrada, radix-2 de 1024 puntos

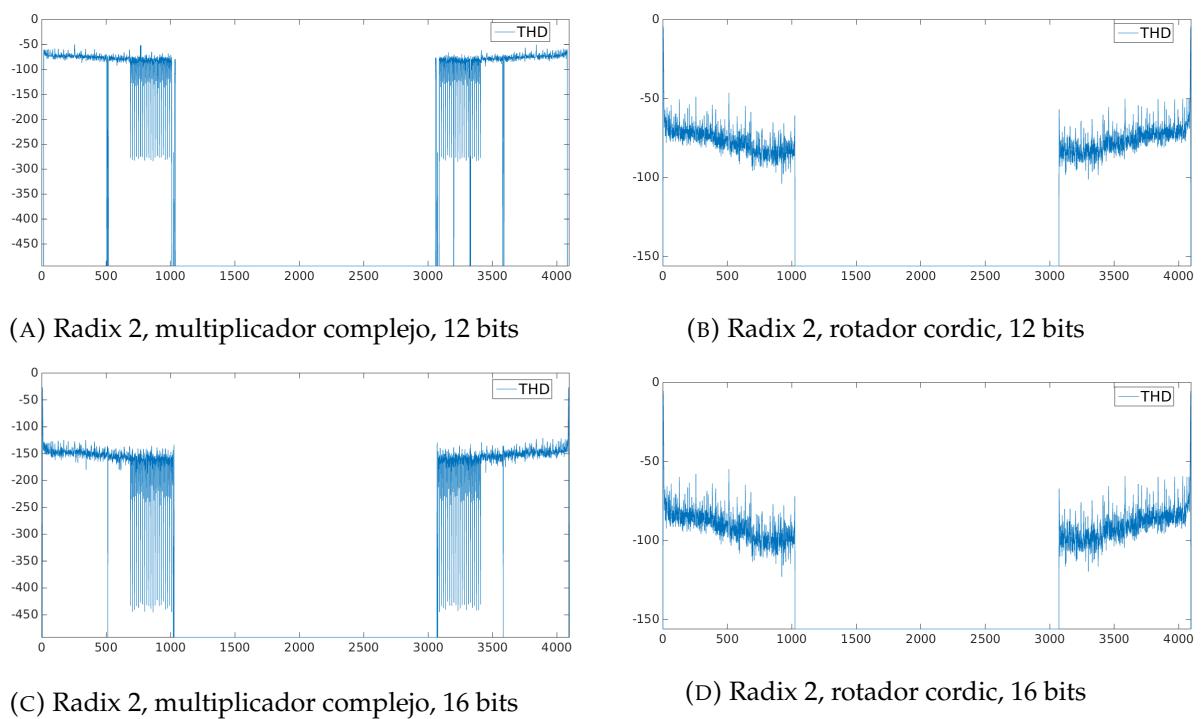


FIGURA 5.6: THD en función del tono de entrada, radix-2 de 4096 puntos

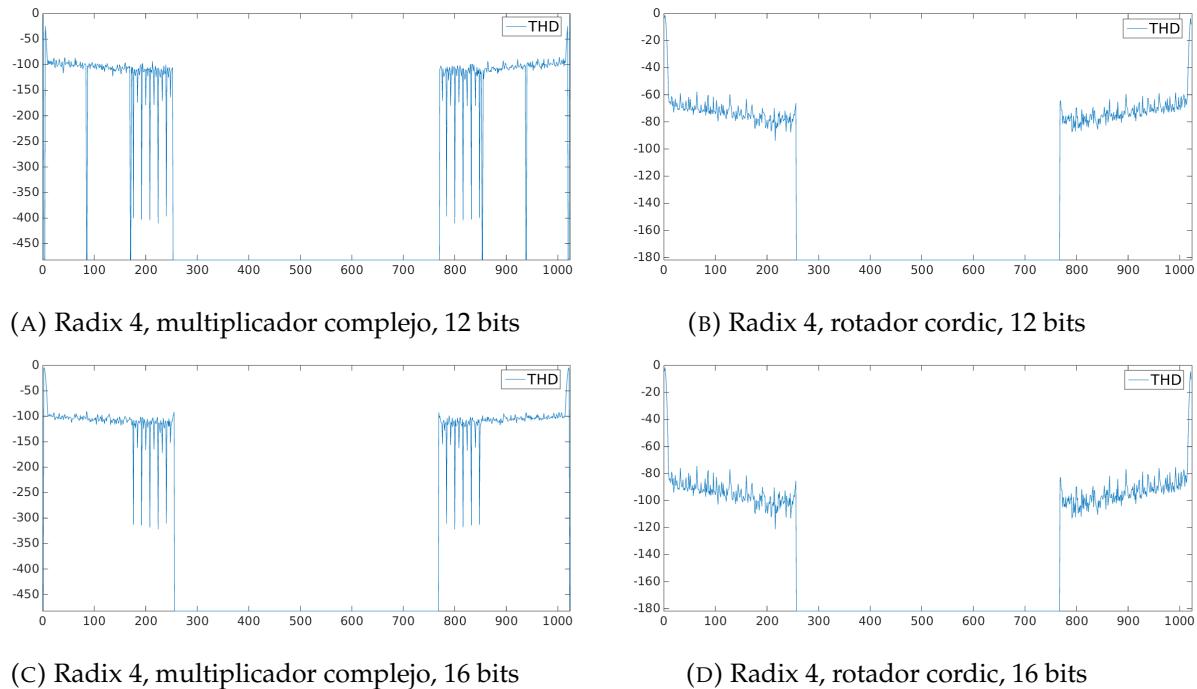


FIGURA 5.7: THD en función del tono de entrada, radix-4 de 1024 puntos

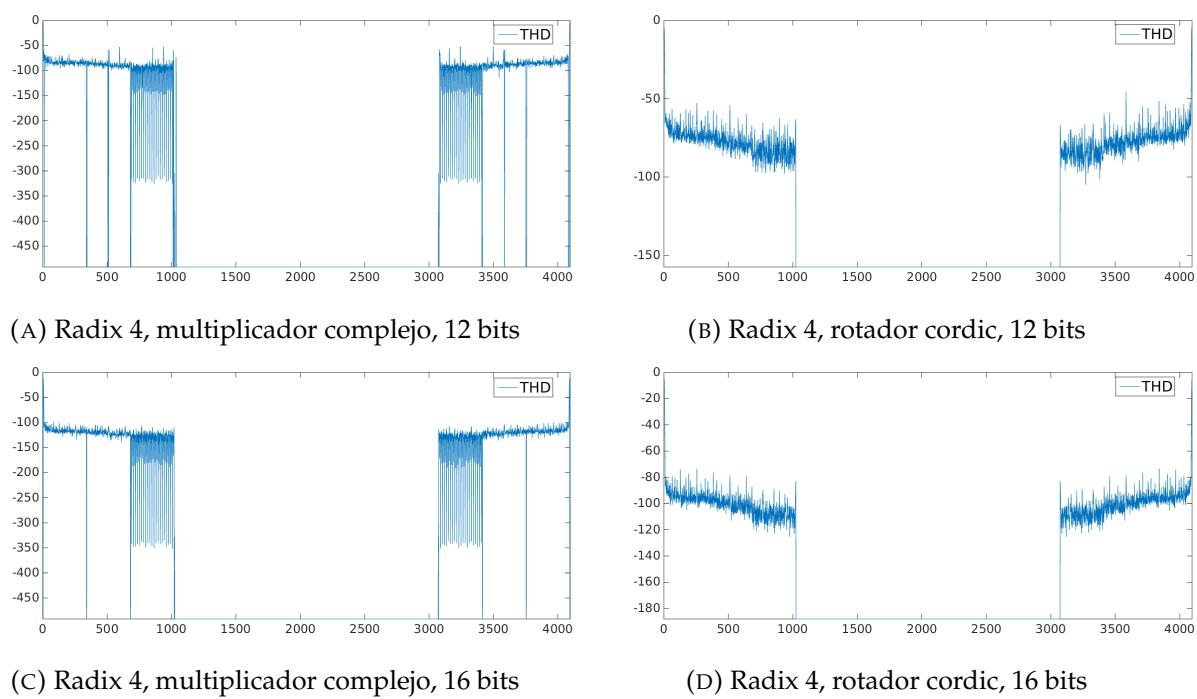
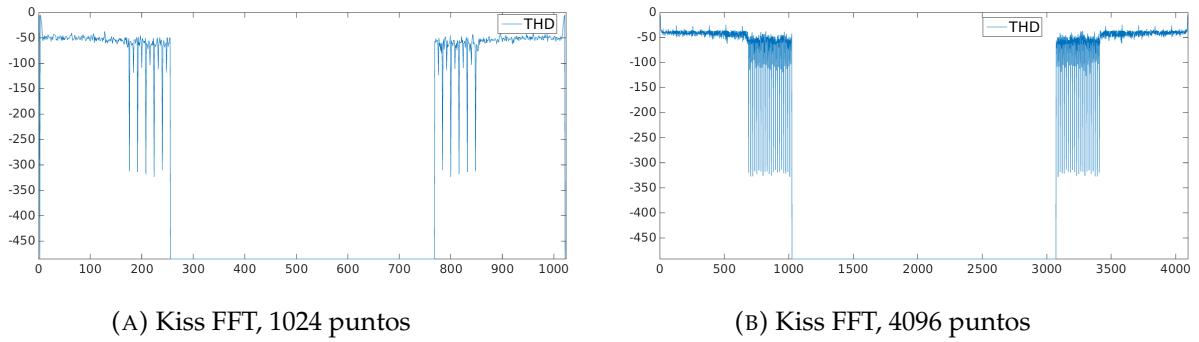


FIGURA 5.8: THD en función del tono de entrada, radix-4 de 4096 puntos



De los gráficos se puede deducir que el comportamiento de las arquitecturas desarrolladas durante el trabajo de tesis se comporta de igual manera que el algoritmo implementado en C++, ampliamente difundido su uso en aplicaciones de procesamiento de señales.

### Efecto de la intermodulación

La intermodulación es la modulación entre tonos, esto es, la distorsión que produce un tono sobre los tonos contiguos. Durante las mediciones de THD se colocó a la entrada un tono puro por vez y se midió los tonos presentes a la salida. Esta medición no permite ver los efectos de intermodulación ya que esta se produce en presencia de más de un tono a la entrada.

Para verificar la presencia de efectos significativos de intermodulación se realizaron algunas mediciones donde se utilizó como señal de entrada un tono puro primero, y luego el mismo tono junto con el tono siguiente. Comparando las salidas de ambos procesamientos se puede ver si el efecto de la intermodulación es relevante o puede ser desestimado.

En la figura 5.10 se muestran los resultados de algunas de estas mediciones. En los gráficos se superpone el resultado de utilizar un solo tono con el resultado de utilizar como entrada ese mismo tono y el tono inmediato anterior. Se puede ver que no se generan tonos armónicos de valor considerable, solo se superponen al espectro original los armónicos generados por el segundo tono, de magnitud comparable a los armónicos del primer tono.

#### 5.3.4. Efectos de redondear o truncar en una etapa

Se realizaron mediciones aplicando escalamientos, redondeo o truncamiento, aplicando primero una señal aleatoria y se comparó la salida con la respuesta a la misma entrada de la arquitectura sin escalar. En la figura 5.11 se muestra una ampliación de una zona de la salida de ambas arquitecturas, con y sin escalamiento, a la entrada aleatoria. Se puede ver el efecto del escalamiento, no solo reduciendo la amplitud de la señal, sino también reduciendo la resolución de la salida, ya que al escalar se achata la señal. Para este ensayo se utilizó la arquitectura radix-2 de 4096 puntos y 16 bits de ancho de palabra.

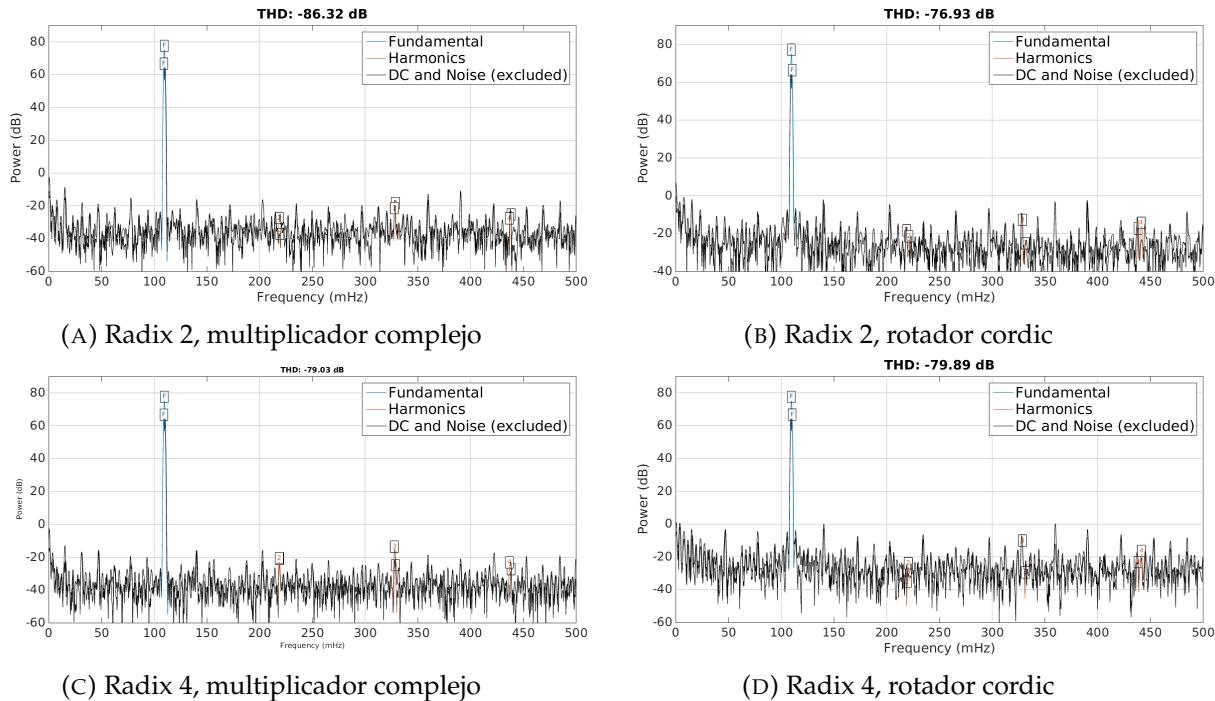


FIGURA 5.10: THD de la respuesta a dos tonos consecutivos

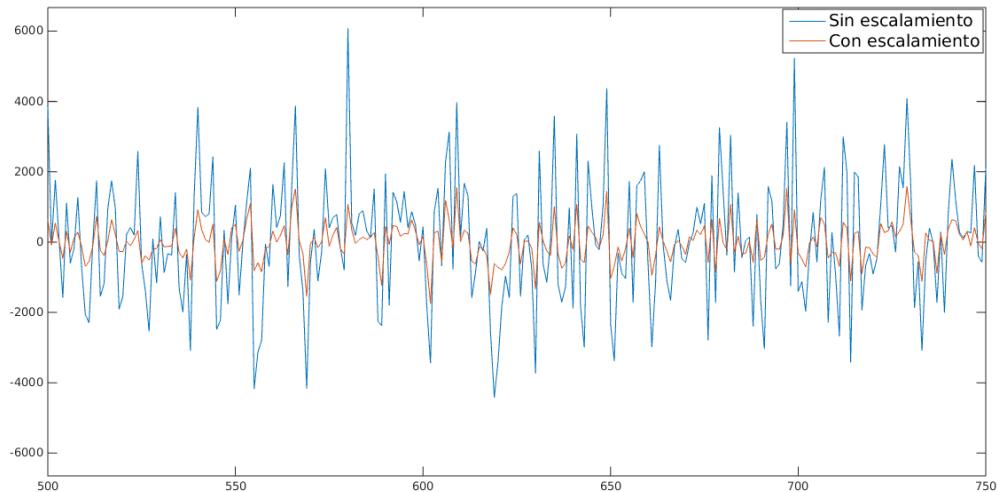
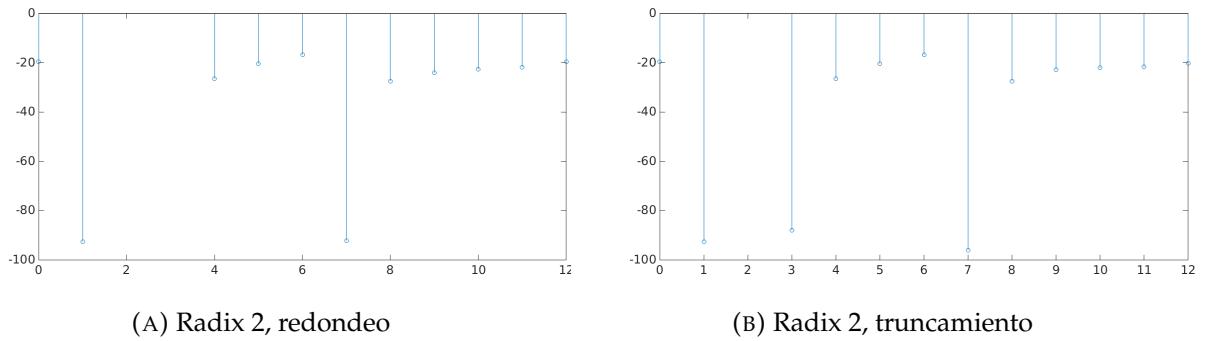
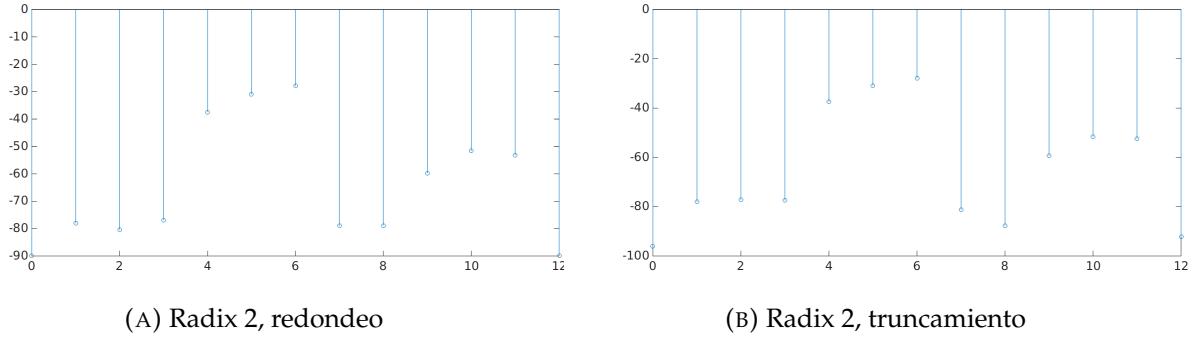


FIGURA 5.11: Comparación entre el procesamiento con escalamiento y sin escalamiento

También se realizaron pruebas para determinar el efecto del escalamiento en la distorsión. Para esto se utilizó como entrada un tono puro en la zona donde la THD para la arquitectura sin escalar es de un valor finito, y se simularon 13 esquemas distintos, utilizando en cada simulación escalamiento en una etapa distinta. En la figura 5.12 se observa la THD como resultado del escalamiento en cada etapa. En la figura 5.13 se observa el resultado del mismo experimento pero utilizando a la entrada un tono puro de una magnitud tal que cause *overflow* en caso de no utilizarse escalamiento, tanto para redondeo como para truncamiento.



En la figura 5.13 se observa como disminuye la THD al aplicar escalamiento en algunas etapas en particular, viéndose de esta manera cuales son las etapas donde se genera el *overflow* para la entrada utilizada. Esto también muestra que el mecanismo de escalamiento es útil para resolver situaciones donde se puede generar *overflow*, y al ser un mecanismo configurable dinámicamente se puede adaptar el funcionamiento de las arquitecturas a las señales particulares que se estén procesando.

### 5.3.5. Validación de las arquitecturas mediante pruebas en hardware

Para la validación de las arquitecturas en hardware se utilizó una FPGA XC5VL110, de la familia Virtex-5, fabricada por Xilinx, en una placa de desarrollo fabricada por Avnet, que provee, además del chip FPGA, otros periféricos como puerto USB, puerto serie, botones y LEDs.

Se ensayaron de esta manera las arquitecturas radix-2 y radix-4 iterativas con ancho de palabra de 12 bits, para 1024 puntos, tanto con multiplicador complejo como con el rotador cordic.

Para la síntesis de los IP Cores se utilizó el software Xilinx ISE v13.4. Para la configuración del chip se utilizó la herramienta iMpect incluida en el software mencionado.

La estrategia para los ensayos fue la generación de vectores de prueba en Matlab, que son enviados al chip a través de un puerto serie y almacenados en una memoria auxiliar para luego ser procesados por la arquitectura y comunicados a la PC a través del puerto serie, para ser analizados mediante procesamiento en Octave.

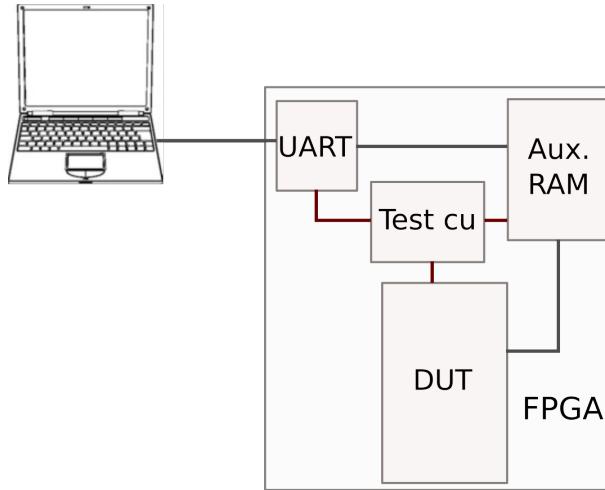


FIGURA 5.14: Testbench para la validación de las arquitecturas en hardware

Los vectores de prueba consisten tanto en las señales patrón utilizadas en los ensayos de la sección 5.3.1 como en señales aleatorias para la medición de error.

Las pruebas con señales patrón fueron positivas, obteniendo como resultado las señales esperadas para todas las entradas.

Las pruebas de medición de error con señales aleatorias dieron resultados dentro de los valores de las tablas 5.1 y 5.2.

En base a estos resultados se puede concluir en que los IP Cores desarrollados durante el presente trabajo de tesis son válidos y utilizables para los propósitos para los que fueron diseñados.

#### 5.4. Análisis de utilización de recursos de las arquitecturas

Se analiza en esta sección el tamaño de las arquitecturas implementadas y se compara con la implementación de una arquitectura radix-2 desenrollada para verificar el ahorro en el espacio ocupado por los diseños del presente trabajo de tesis.

La plataforma utilizada para este análisis es un chip XC5VLX110, marca Xilinx, de la familia Virtex-5.

Se realizó la síntesis de las diferentes arquitecturas para un tamaño de palabra de 16 bits para 1024 y 4096 puntos, utilizando el software Xilinx ISE v13.4. Los datos de implementación se obtuvieron utilizando las herramientas de análisis del software mencionado. Se realizó además la síntesis para el IP Core propietario LogiCORE FFT v.7.1 de Xilinx [21], para utilizar como una referencia extra por ser un desarrollo comercial de una empresa especializada en electrónica digital. En las figuras 5.15 y 5.16 se muestran los resultados de la síntesis de las arquitecturas mencionadas para 1024 y 4096 puntos.

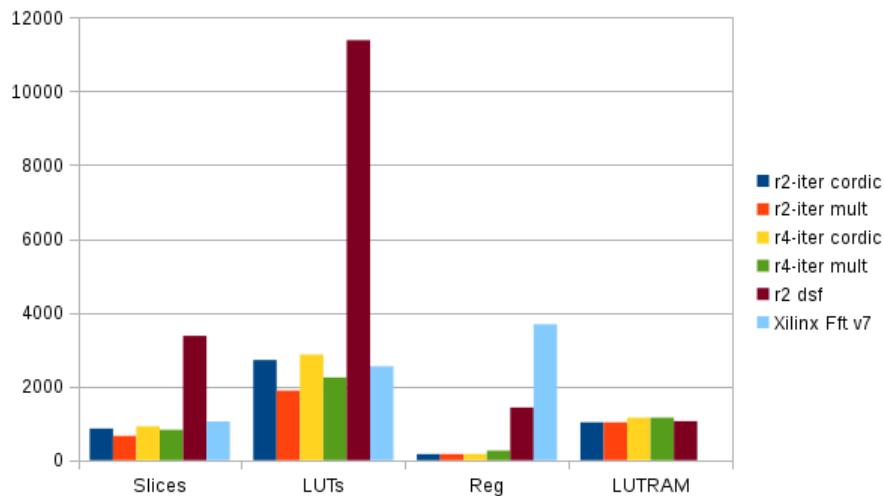


FIGURA 5.15: Comparativa de tamaño de síntesis de diferentes arquitecturas para 1024 puntos en una FPGA XC5VLX110

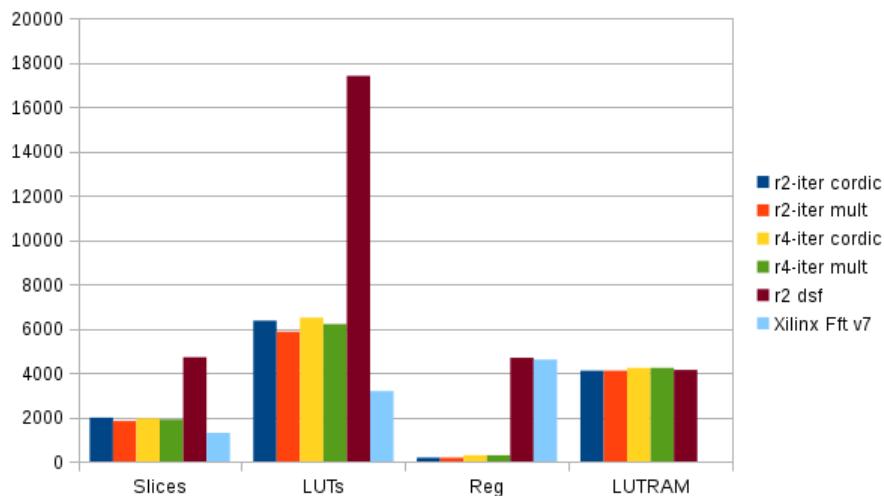


FIGURA 5.16: Comparativa de tamaño de síntesis de diferentes arquitecturas para 4096 puntos en una FPGA XC5VLX110

Se observa el ahorro en recursos que presentan las arquitecturas propuestas respecto a una arquitectura radix-2 desenrollada, e incluso frente al IP Core FFT v.7.1.

La principal diferencia se ve en el uso de LUTs, que representa la ocupación misma de recursos de la FPGA. Si bien comparado con el IP Core de Xilinx el consumo de LUTs es similar, se ve una diferencia notable en el uso de registros, mostrando una vez más que la arquitectura propuesta ocupa menor cantidad de recursos para un ancho de palabra y cantidad de puntos dados. Hay que tener en cuenta también que el IP Core FFT de Xilinx utiliza los multiplicadores dedicados de la FPGA, mientras que las arquitecturas desarrolladas en este trabajo que utilizan el rotador cordic no lo hacen.

Se puede ver también que el consumo de recursos es similar entre la radix-2 y la radix-4 iterativas, teniendo la segunda la ventaja de requerir la mitad de ciclos de *clock* para procesar la misma cantidad de puntos.

De esta manera se verifica el cumplimiento del objetivivo de desarrollar una arquitectura económica en ocupación de recursos de implementación, lo que constituye uno de los propósitos fundamentales del presente proyecto de tesis.

# Capítulo 6

## Conclusiones

### 6.1. Conclusiones Generales

El alcance propuesto para el presente trabajo se basó en los siguientes puntos:

- IP Cores codificados en el lenguaje Verilog de arquitecturas de cálculo de FFT de tamaño reducido.
- Estudio del comportamiento numérico de las arquitecturas implementadas (ruido, distorsión armónica, etc.)
- Análisis comparativos de procesamiento entre las arquitecturas desarrolladas y desarrollos de terceros.
- Proposición de trabajos futuros y/o mejoras.

Se logró implementar satisfactoriamente dos arquitecturas diferentes basadas en el algoritmo Radix para el cómputo de la FFT, ofreciendo además la posibilidad de optar por dos métodos diferentes de multiplicación por los *twiddle factors*, siendo más eficiente el multiplicador complejo en FPGAs que posean multiplicadores dedicados.

Dichas implementaciones se resumen en 20 códigos fuente para cada una, de los cuales 17 son compartidos entre ambas, que contienen la lógica y descripción del hardware en lenguaje Verilog. Adicionalmente se desarrollaron diferentes herramientas que permiten ensayar las arquitecturas en diferentes condiciones de forma automática.

Como parte de los ensayos de las arquitecturas se obtuvieron diferentes métricas que permiten caracterizarlas. Por un lado se presentó el resumen de recursos utilizados para la síntesis de las arquitecturas posibles para una FPGA y se lo comparó con los recursos utilizados por arquitecturas desarrolladas por terceros, incluyendo una comercial. De esta comparación se concluye que se cumple con el requerimiento de mínima área de chip necesaria y la economía de recursos utilizados, resultando más eficiente espacialmente que las soluciones de terceros. De este modo, resulta en una opción ventajosa para implementar en sistemas SDR con recursos limitados.

La distorsión armónica total medida se encuentra en el orden de desarrollos de terceros ampliamente utilizados, lo que indica que las arquitecturas desarrolladas en el presente trabajo de tesis son aptas para ser utilizadas en sistemas de comunicación con gran confiabilidad.

El error relativo medido, utilizando como parámetro de medición un sistema con precisión de punto flotante de 64 bits, es comparable con implementaciones de terceros utilizadas comercialmente en procesamiento de señales y sistemas de comunicación, por lo que las arquitecturas desarrolladas son aptas para su utilización en dichos sistemas.

De este modo se concluye en que se obtuvieron dos arquitecturas, con sus variantes, de baja utilización de recursos, aptas para ser utilizadas en sistemas reales de comunicación y procesamiento de señales, cumpliendo con los objetivos planteados al comienzo del trabajo de tesis.

## 6.2. Trabajos Futuros

Este trabajo se presentó la primer etapa en el desarrollo de las arquitecturas, quedando a futuro varios aspectos a mejorar y posibilidades de optimización del diseño. Como potenciales trabajos a futuro en el contexto de la presente tesis, se destacan los siguientes:

1. Estudiar posibles implementaciones de algoritmos de *dithering* para reducir el ruido generado en las arquitecturas.
2. Modificar el módulo de rotación Cordic agregando un pipeline que permita aumentar la velocidad de *clock* de las arquitecturas, sin agregar ciclos de *clock* extra al cómputo total de la FFT.
3. Modificar el multiplicador complejo agrandando el tamaño de palabra de los factores de multiplicación de los *twiddle factors* y/o optimizarlo para la utilización de los bloques de procesamiento digital de los dispositivos FPGAs.
4. Modificar el hardware para que el tamaño de palabra del ángulo de rotación sea independiente del tamaño de palabra de las señales.
5. Estudiar la posibilidad de modificar las arquitecturas de forma de poder modificar la cantidad de puntos de la FFT en forma dinámica.

# Bibliografía

- [1] Muller J.C. Bajard J.C. Kla S. «BKM: A New Hardware Algorithm for Complex Elementary Functions». En: *IEEE Transactions on Computers* 43.8 (jun. de 1994), págs. 955-962. DOI: [10.1109/ARITH.1993.378098](https://doi.org/10.1109/ARITH.1993.378098).
- [2] Burrus C. «Index Mapping for Multidimensional Formulation of the DFT Convolution». En: *IEEE Transactions on Acoustics, Speech and Signal Processing* 25.3 (jun. de 1977), págs. 239-242. ISSN: 0096-3518. DOI: [10.1109/TASSP.1977.1162938](https://doi.org/10.1109/TASSP.1977.1162938).
- [3] *Fast Fourier Transform*. Recuperado de Wikipedia, the free encyclopedia. URL: [https://en.wikipedia.org/wiki/Fast\\_Fourier\\_transform](https://en.wikipedia.org/wiki/Fast_Fourier_transform).
- [4] Volder J. *The Cordic Computer Technique*. Ed. por IRE. Trans. Elect. Comput. 1959.
- [5] *Kiss FFT is a very small, reasonably efficient, mixed radix FFT library that can use either fixed or floating point data types*. URL: <https://sourceforge.net/projects/kissfft/?source=navbar>.
- [6] Shousheng H. Torkelson M. «A New Approach to Pipeline FFT Processor». En: *International Parallel Processing Symposium* (abr. de 1996). DOI: [10.1109/IPPS.1996.508145](https://doi.org/10.1109/IPPS.1996.508145).
- [7] Manolakis D. Proakis J. *Digital Signal Processing. Principles, Algorithms and Applications*. USA: Prentice-Hall, 1997. Cap. The Sampling Theorem, págs. 29-33.
- [8] Oppenheim A. Schafer R. *Tratamiento de Señales en Tiempo Discreto*. España: Prentice Hall, 2000. Cap. Cómputo de la Transformada Discreta de Fourier, págs. 631-694.
- [9] Oppenheim A. Schafer R. *Tratamiento de Señales en Tiempo Discreto*. España: Prentice Hall, 2000. Cap. La Transformada Discreta de Fourier, págs. 543-572.
- [10] Prasad R. *OFDM for Wireless Communication Systems*. UK: Artech House, 2004. Cap. Orthogonal Frequency-Division Multiplexing, págs. 11-15.
- [11] Winograd S. «On Computing the Discrete Fourier Transform». En: *Mathematic of Computation* 32.141 (ene. de 1978), págs. 175-199. ISSN: 0025-5718.
- [12] Meyer-Baese U. *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin: Springer-Verlag, 2007. Cap. Fourier Transform, págs. 343-400.
- [13] Meyer-Baese U. *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin: Springer-Verlag, 2007. Cap. Fourier Transform, págs. 343-344.
- [14] Meyer-Baese U. *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin: Springer-Verlag, 2007. Cap. Fourier Transform, págs. 349-350.

- [15] Meyer-Baese U. *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin: Springer-Verlag, 2007. Cap. Fourier Transform, págs. 350-353.
- [16] Meyer-Baese U. *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin: Springer-Verlag, 2007. Cap. Fourier Transform, págs. 353-355.
- [17] Meyer-Baese U. *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin: Springer-Verlag, 2007. Cap. Fourier Transform, págs. 363-373.
- [18] Meyer-Baese U. *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin: Springer-Verlag, 2007. Cap. Fourier Transform, págs. 375-376.
- [19] Meyer-Baese U. *Digital Signal Processing with Field Programmable Gate Arrays*. Berlin: Springer-Verlag, 2007. Cap. Fourier Transform, págs. 365-366.
- [21] Xilinx. *LogiCORE IP Fast Fourier Transform v7.1*. 2011.