UNIVERSIDAD NACIONAL DE INGENIERÍA
FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

TESIS:

"ADAPTACIÓN DE UN ALGORITMO USANDO LA FFT PARA INCREMENTAR LA VELOCIDAD DE TRANSMISIÓN EN SISTEMAS DE MULTIPLEXACIÓN FRECUENCIAL"

PARA OBTENER EL GRADO DE MAESTRO EN CIENCIAS CON MENCIÓN EN TELEMÁTICA

ELABORADO POR:

LEANDRO TEODORO ARIAS ANTONIO

ASESOR:

M. Sc. Ing. JUAN CARLOS ALVAREZ SALAZAR

LIMA – PERÚ

2020

DEDICATORIA

A mis Padres Tobías y María Julia que ya no están, pero sus ejemplos y dedicación, fueron los que guiaron para conseguir los objetivos planteados.

AGRADECIMIENTOS

Quiero agradecer póstumamente el Dr. Manuel Chang Ching mi primer asesor, sus sabias recomendaciones que aún recuerdo.

Quiero agradecer al Ing. Juan Carlos Álvarez Salazar profesor de los cursos de Tesis I y Tesis II que plasmaron la culminación de la Tesis, así mismo a todos los profesores de la Escuela de Post Grado de la FIEE, en las dos etapas del proceso de la Maestría en Ciencias con mención en Telemática

INDICE DE CONTENIDO

CAPITULO I 3

ANTECEDENTES Y DESCRIPCION DEL PROBLEMA 3

1.1 Antecedentes bibliograficos 3
1.2 Descripcion de la realidad problematica 6
1.3 Formulacion del problema 7
1.4 Justificación e importancia de la investigación 7
1.5 Objetivos 8
1.5.1 Objetivo general 8
1.5.2 Objetivos específicos 8
1.6 Hipótesis 8
1.6.1 Hipótesis general 8
1.6.2 Hipótesis secundarias 8
1.7 Variables e Indicadores 8
1.7.1 Variable independiente e indicadores 8
1.8 Unidad de análisis 8
1.9 Tipo y nivel de investigación 8
1.10 Periodo de análisis 8
1.11 Fuentes de información e instrumentos utilizados 9
1.12 Técnicas de recolección y procesamiento de datos 9
1.13 UML – Diagrama de Casos de Uso 9

CAPITULO II 11

MARCO TEORICO Y CONCEPTUAL 11

2.1 Algoritmos DFT 11
2.2 Álgoritmos FFT 11
2.2.1 Algoritmo de *Cooley-Tuckey* para el cálculo de FFT 11
2.2.2 Algoritmos radix-r 13
2.2.3 Transformada de Fourier en Tiempo Discreto 14
2.2.4 La Transformada Rápida de Fourier 14
2.2.5 Radix-2 en decimación de frecuencia 15
2.2.6 Radix-2 en decimación de tiempo 16
2.3 Uso de la DFT en las comunicaciones digitales 20
2.3.1 Multiplexación por división ortogonal de frecuencia (OFDM) 20
2.3.2 Implementación de la modulación OFDM mediante DFT 25
2.3.3 Arquitecturas radix-r 27
2.3.4 Algoritmo *Cordic* 28
2.3.5 Multiplicador complejo eficiente 30
2.3.6 Método de redondeo o truncamiento 30
2.3.7 Arquitecturas a adaptar 32

CAPITULO III 33

DESARROLLO DEL TRABAJO DE LA TESIS 33

3.1 Desarrollo de la adaptación del algoritmo Radix-2 33
3.1.1 Algoritmos DFT y FFT 33
3.1.2 Estructura Radix-2 34
3.1.2.1 Descripción General 34
3.1.3 Formas de adaptar el algoritmo radix-r 35
3.1.4 Multiplicación por los *twiddle factors* 35
3.3 Herramientas utilizadas para el desarrollo 37

CAPITULO IV 38

ANALISIS Y PRESENTACION DE RESULTADOS 38
4.1 Simulación y analisis de resultados 38
4.1.1 Dispositivo Lógico Programable FPGA 40
4.1.2 Validación de las arquitecturas mediante pruebas en hardware 40
4.2 Análisis de utilización de recursos de las arquitecturas 41
CONCLUSIONES Y RECOMENDACIONES 44

GLOSARIO 46

BIBLIOGRAFIA 47

ANEXO A 49

a.1.1 Descripción General 49
a.1.2 Memoria 50
a.1.3 Sumador/restador 53
a.1.4 Datapath 54
a.1.5 Unidad de control 58
a.1.6 Máquinas de estados 59
a.1.7 Control de la memoria 60
a.1.8 Integración de la unidad de control 61

ANEXO B 63

b.1.1 Delta en componente '0' 63
b.1.2 Bloque Transformada Inversa Rápida de Fourier (IFFT). 63
b.1.3 Bloque Transformada Rápida de Fourier (FFT) 66
b.1.4 Delta 69
b.1.5 Medición del error 69
b.1.6 Lenguaje de Descripción de Hardware VHDL 73
b.1.7 Distorsión Total Armónica 73
b.2 Simulación y analisis de los módulos individuales 74
b.3 Simulación y analisis de las arquitecturas completas 74
b.3.1 Procesamiento de señales patrón 74
b.3.2 Efecto de la intermodulación 75
b.3.3 Efectos de redondear o truncar en una etapa 75

ANEXO C 77

c.1.1 Memoria 77
c.1.2 Butterfly 78

c.1.3 Datapath 78
c.1.4 Unidad de control 82
c.1.5 Máquinas de Estados 83
c.1.6 Control de la memoria 84
c.1.7 Control del datapath 85
c.1.8 Integración de la unidad de control 85
c.2 Módulos compartidos por las dos arquitecturas 86
c.2.1 *Cordic* desenrollado 86
c.2.2 Multiplicador complejo 88
c.2.3 Unidad de escalamiento 90

ANEXO D 92

Código fuente Complex.java 92
Código fuente FFT.java 95
Código fuente Radix2FFT.java 98

INDICE DE ILUSTRACIONES

Figura 1.1: UML - Diagrama de Adaptación de un Algoritmo usando FFT para OFDM. 9
Figura 1.2: UML - Diagrama de Adaptación de radix-2 para OFDM. 10
Figura 2.1: Diagrama de Clases de la Adaptación de radix-2 FFT para OFDM.....13
(Fuente: Referencia [18]) 13
Figura 2.2: Diagrama de mariposa para la obtención de la FFT en decimación en frecuencia.....16
(Fuente: Referencia [4]) 16
FIGURA 2.3: Diagrama de mariposa del algoritmo radix-2 en decimación en el tiempo para la obtención de la FFT con N = 8 17
(Fuente: Referencia [4]) 17
FIGURA 2.4: FFT Radix-2 de 8 puntos 18
(Fuente: Referencia [4]) 18
FIGURA 2.5: Arquitectura radix-2 desenrollada SDF.....19
(Fuente: Referencia [2]) 19
FIGURA 2.6: Arquitectura Radix-2 iterativa 19
(Fuente: Referencia [2]) 19
FIGURA 2.7: Dos formas de transmisión multiportadora.....22
FIGURA 2.8: Muestra espectral de una señal OFDM.....22
FIGURA 2.9: Diagrama de tiempos de un símbolo OFDM 23
(Fuente: Referencia [11]) 23
.....24
FIGURA 2.10: Diagrama en bloques de una transmisión punto a punto OFDM 24
(Fuente: Referencia [11]) 24
FIGURA 2.11: Diagrama de bloques conceptual de un sistema comunicación OFDM. 25
(Fuente: Referencia [11]) 25
FIGURA 2.12: Técnica de Modulación OFDM 25
(Fuente: Referencia [12]) 25
FIGURA 2.13: Ejemplo de rotación con algoritmo *Cordic*.....29
(Fuente: Referencia [19]) 29
FIGURA 3.1: FFT radix-2 de 8 puntos.....34
(Fuente: Referencia [11]) 34
FIGURA 3.29: Señales de comunicación de las estructuras implementadas.....36
(Fuente: Referencia [14]) 36
FIGURA 4.1: Pruebas con FPGA 38
FIGURA 4.2: Resultados con FPGA 39
(Fuente: Referencia [21]) 39
FIGURA 4.3: Estructura conceptual de un dispositivo FPGA 40
(Fuente: Referencia [9]) 40
FIGURA 4.4: Test bench para la validación de las arquitecturas en hardware 41
FIGURA 4.5: Comparativa de tamaño de síntesis de diferentes arquitecturas para 2^{10} puntos en una FPGA XC5VLX110 42

(Fuente: Referencia [2]) 42
FIGURA 4.6: Comparativa de tamaño de síntesis de diferentes arquitecturas para 2^{12} puntos en una FPGA XC5VLX110 42
(Fuente: Referencia [2]) 42
FIGURA A.1: Esquema de una FFT radix-4 de 16 puntos 49
(Fuente: Referencia [11]) 49
FIGURA A.2: Diagrama simplificado de la arquitectura radix-4 iterativa 50
(Fuente: Referencia [2]) 50
FIGURA A.3: RAM de triple entrada y triple salida 51
(Fuente: Referencia [2]) 51
FIGURA A.4: Esquema de direccionamiento de los sub bloques RAM 52
(Fuente: Referencia [2]) 52
FIGURA A.5: Diagrama de la unidad aritmética incluyendo los multiplexores de bypass 54
(Fuente: Referencia [2]) 54
FIGURA A.6: Datapath de la arquitectura radix-4 iterativa 55
(Fuente: Referencia [2]) 55
FIGURA A.7: Datapath para operaciones de transferencia en memoria 56
(Fuente: Referencia [2]) 56
FIGURA A.8: Datapath para operaciones en butterfly 57
(Fuente: Referencia [2]) 57
FIGURA A.9: Selección del par de bits del contador de puntos a evaluar 59
(Fuente: Referencia [2]) 59
FIGURA A.10: Diagrama de estados y transiciones de la máquina de estados principal 59
(Fuente: Referencia [2]) 59
FIGURA A.11: Diagrama de estados y transiciones de la máquina de estados secundaria 60
(Fuente: Referencia [2]) 60
FIGURA A.12: Datapath con las señales de control 61
(Fuente: Referencia [2]) 61
Figura B.1: Diagrama esquemático del bloque IFFT 66
(Fuente: Referencia [11]) 66
Figura B.2: Diagrama esquemático del bloque FFT 68
(Fuente: Referencia [11]) 68
FIGURA B.3: Respuestas a una delta en la componente 0 para las arquitecturas radix-2 y radix-4 69
(Fuente: Referencia [2]) 69
FIGURA B.4: Respuestas a una delta en la componente 7 para las arquitecturas radix-2 y radix-4 70
(Fuente: Referencia [2]) 70
FIGURA B.5: Diagrama de flujo de la simulación para la estimación del error 71
(Fuente: Referencia [2]) 71
FIGURA B.6: THD en función de la etapa en que se realiza es escalamiento 76
FIGURA B.7: THD en función de la etapa en que se realiza es escalamiento para una señal que provoca overflow 76
(Fuente: Referencia [2]) 76
FIGURA C.1: Diagrama simplificado de la arquitectura radix-2 iterativa 77
(Fuente: Referencia [2]) 77
FIGURA C.2: Dual Port RAM 78

(Fuente: Referencia [2]) 78
FIGURA C.3: Esquema del bloque butterfly 78
(Fuente: Referencia [2]) 78
FIGURA C.4: Esquema del datapath de la arquitectura radix-2 79
(Fuente: Referencia [2]) 79
FIGURA C.5: Datapath para operaciones de transferencia en memoria 80
FIGURA C.6: Datapath para operaciones en butterfly 81
(Fuente: Referencia [2]) 81
FIGURA C.7: Selección del bit del contador de puntos a evaluar 83
(Fuente: Referencia [2]) 83
FIGURA C.8: Estados de la máquina de estados principal 83
(Fuente: Referencia [2]) 83
FIGURA C.9: Máquina de estados operativa para modo *enabled* 84
(Fuente: Referencia [2]) 84
FIGURA C.10: Datapath con las señales de control 85
(Fuente: Referencia [2]) 85
FIGURA C.11: Bloque de módulo de cómputo *cordic* 86
(Fuente: Referencia [14]) 86
FIGURA C.12: Diagrama en bloques del módulo cordic 88
(Fuente: Referencia [14]) 88
FIGURA C.13: Diagrama en bloques del bloque de rotaciones del módulo *cordic* 88
(Fuente: Referencia [14]) 88
FIGURA C.14: Bloque de módulo multiplicador complejo 89
(Fuente: Referencia [14]) 89
FIGURA C.15: Diagrama en bloques de la unidad de escalamiento 90
(Fuente: Referencia [14]) 90

INDICE DE TABLAS

TABLA 2.1: COMPARACIÓN DEL ESFUERZO COMPUTACIONAL DE LA DFT Y LA FFT 14

TABLA 2.2: COMPARATIVA ENTRE LAS IMPLEMENTACIONES PARALELA, DESENROLLADA E ITERATIVA DEL ALGORITMO RADIX-R 19

TABLA 2.3: CANTIDAD DE OPERACIONES COMPLEJAS PARA DISTINTAS LONGITUDES DE DFT 27

TABLA 2.4: EJEMPLOS DE TRUNCAMIENTO Y REDONDEO EN SISTEMA DECIMAL.. 31

TABLA B.1: MÉTRICA E_∞ PARA 1024 REALIZACIONES DE CADA ARQUITECTURA CON ENTRADAS ALEATORIAS 72

TABLA B.2: MÉTRICA E2 PARA 1024 CORRIDAS DE CADA ARQUITECTURA CON ENTRADAS ALEATORIAS 72

TABLA C.1: EJEMPLOS DE CODIFICACIÓN DEL ÁNGULO PARA UN ANCHO DE PALABRA DEL ÁNGULO DE 7 BITS 89

RESUMEN

El objetivo principal del trabajo de investigacion es la adaptación de un algoritmo basado en la transformada rápida de Fourier (FFT-Fast Fourier Transformer) y su aplicación en un sistema OFDM mediante el diseño y la implementación en hardware de una arquitectura de cómputo de la transformada rápida de Fourier (FFT) basada en el análisis y la mejora sustancial del algoritmo *Cooley-Tukey*. Esto se logra con el algoritmo radix-2, que es una variante de la generación de algoritmos radix-r, para ser utilizada en la modulación y demodulación de un sistema de comunicaciones digitales OFDM. En primera instancia se presenta el algoritmo *Cooley-Tukey* que es el más universal de los algoritmos para el cálculo de FFT. Una mejora sustancial de este algoritmo, son los llamados radix-r y lo que se adaptará es el algoritmo radix-2, logrando que el DFT se descomponga en varios DFTs consecutivos, de modo que los cálculos serán más sencillos. A continuación, se presenta la estructura de los algoritmos radix-r y las formas de su adaptación en modo general en tres clases: radix paralelo, radix descompuesto y radix iterativo y para el caso particular del radix-2 se utilizará la forma iterativa en cada una de las etapas del cómputo, logrando que la transmisión de datos sea más rápida y eficiente cuando se adaptan en los sistemas OFDM.

ABSTRACT

The main objective of this research work is the adaptation of an algorithm based on the Fast Fourier Transform (FFT) fast the aplication in an OFDM system an design and implementation in digital hardware of a computing architcture of the FFT base on an excellent improvement of the Cooley-Tukey algorithm. It is achieved with the Radix-2 algorithm wich is a variety of the Radix-r algorithm generation that is used in a modulation and demodulation in an OFDM communication system.First, it is perfomed the Cooley-Tukey algorithm is the most common algorithms in order to calculate an FFT, an excellent improvement of this algorithm are called Radix-r, which is adapted as the Radix-2 algorithm, achieving that DFTs computing are descomposed in several consecutive DFT whit an advantage in the computing as the easiest.Then the Radix-r algorithm architecture are perfomed in a general mode of the implementation shapes in three kinds parallel, descomposed and iterative. And particular case, of the Radix-2, we will utilize en iterative form in each stage of the Radix computing. So achieving the data transmission is faster and more efficient when they are implemented in the OFDM systems.

INTRODUCCION

La creciente demanda de velocidad en las telecomunicaciones lleva a la adaptación 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 utiliza múltiple portadoras en las que se modulan los datos a transmitir. Una forma práctica y eficiente de adaptar la modulación y la demodulación multiportadora requerida por este sistema es mediante el uso de la transformada rápida de Fourier, aprovechando los algoritmos de alta eficiencia disponibles para su adaptación.

El presente trabajo de investigación se encuentra enfocado en el contexto de la adaptación de un algoritmo basado en FFT en hardware digital, y fue motivado por el creciente avance en las tecnologías utilizadas en los sistemas de comunicación en general y en los sistemas de radio definido por software (SDR), en particular, los sistemas SDR permiten implementar sistemas de comunicación mediante software o hardware digital reconfigurable, otorgándoles una gran flexibilidad.

Teniendo en cuenta la complejidad de los sistemas de transmisión OFDM y la necesidad de poder adaptarlos en forma eficiente tanto en consumo como en espacio y recursos utilizados, el presente trabajo de investigación 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

De los antecedentes bibliográficos revisados, se describe la realidad programática del presente trabajo de investigación, los aportes de cada tesis permiten analizar y discutir que aspectos teóricos y prácticos se deben considerar para la adaptación del algoritmo Radix-2, basado en FFT para su aplicación en un sistema OFDM; que es un tipo de modulación descrito con amplitud en el marco teórico como parte de los distintos tipos de modulación que existen actualmente y ios resultados que se pueden obtener con la aplicación del algoritmo Radix-2 basados en FFT. También se resalta el análisis de los diferentes algoritmos basados en FFT para diferentes aplicaciones y en particular para cubrir los objetivos del presente trabajo de investigación.

En el capitulo I están detallados los antecedentes bibliográficos de los distintos autores, entre tesis, trabajos de investigación y artículos que se analizaron y son referencias en los distintos capítulos del trabajo de investigación, como los algoritmos y su descripción detallada.

El capitulo II corresponde al marco teorico y el eje central es la adaptación del algoritmo radix-2, en base a la mejora sustancial del algoritmo de referencia el Cooley-Tukey. Se analizaron otros algoritmos como el de Winograd, Cordic, Goertzel, los radix-r, DFT y FFT. También se describe el modelo de comunicación en OFDM, la selección de las técnicas, de modulación en OFDM, y las señales subportadoras que transmite OFDM.

Asimismo estan descritas las ecuaciones de las transformadas DFT y FFT, en base a la serie de Fourier discreto que, en modo frecuencial se denomina transformada rápida de Fourier FFT y están detallados los algoritmos radix-2, en decimacion de tiempo y frecuencia.

En el capitulo III, se muestra al detalle la adaptación del algoritmo radix-2, explicando su arquitectura. El proceso tuvo tres etapas: la primera fué la selección del algoritmo en base a su estructura y los módulos que forman parte del proceso; donde se realizaron las pruebas de eficiencia, rapidez de transmisión de datos, reducción del tiempo de ejecución y el número de multiplicaciones por etapas; en la segunda se implementó cada uno de los módulos, realizando pruebas de funcionamiento en forma individual y en la tercera se realizó la integración gradual de todos los módulos con cada arquitectura y se realizaron las pruebas de verificación de los resultados y se presentan en el capitulo IV.

En el capitulo IV se presenta en forma directa los resultados obtenidos las que comprueban la hipotesis general H_0, las hipótesis secundarias H_1 y H_2, con lo cual se cumplen los objetivos general y especificos de la presente tesis.

Para obtener estos resultados, se verificó el funcionamiento de cada módulo a través de simulaciones en verilog comprobando su correcto funcionamiento.

Finalmente, se presentan las conclusiones que verifican la hipótesis principal y secundaria, de la adaptación del algoritmo radix-2

CAPITULO I

ANTECEDENTES Y DESCRIPCION DEL PROBLEMA

1.1 Antecedentes bibliograficos

En la tesis de Cassagnes [2], que se ha usado como referencia base, se describe cuatro aspectos fundamentales.

1.1.1.- Descripción de los algoritmos, *Radix-r, Cordic,Radix-2, Radix-4* y *Cooley-Tukey*, sus ecuaciones y su implementación en FFT.

1.1.2.- Se detallan las técnicas de codificación de OFDM, básicamente tres, QPSK, BPSK y QAM, comparando sus ventajas y desventajas, en la codificación de cada una de las sub portadoras, en los procesos de modulación y demodulación, aumentado la velocidad de la transmisión de datos en OFDM

1.1.3.- Se compara al detalle las arquitecturas de los algoritmos Radix-2 y Radix-4, su implementación y adaptación del algoritmo Radix-2, que es el tema del trabajo de investigación. se describe las configuraciones de la Memoria, *Butterfly*, *Datapath*, unidad de control y máquinas de estado, mediante diagrama de bloques, diagramas de flujo y sus códigos de funcionamiento.

1.1.4.- Finalmente se efectua la simulación y validación de los IP *cores,* mediante los programas de las funciones de *Verilog* y *Matlab*, se presentan los resultados de THD con gráficos de comparación entre los algoritmos Radix-2 y Radix-4. Se presentan las herramientas usados para obtener de estos resultados

En la tesis de Cruz [3]. se presenta el desarrollo e implementación de un sistema embebido en FPGA (*Field Programable Gate Array*) para el procesamiento de datos en 2D y sus aplicaciones. Se muestran varios métodos para realizar la recuperación del frente de onda, uno de éstos se realiza resolviendo la ecuación diferencial producida por los datos de un sensor conocido como "sensor de curvatura", utilizando como recurso la FFT. Debido al alto consumo de recursos que se necesitan para resolver la FFT y además de que el cálculo se debe realizar en tiempo real, es necesario implementar un dispositivo embebido que lleve a cabo esta función. Para aplicaciones de procesamiento de señales hay dos que

se emplean con mayor frecuencia: los DSPs (*Digital Signal Processor*) y los FPGAs (*Field Programable Gate Array*) estos últimos han evolucionado muy rápidamente, tanto que un FPGA puede sustituir a varios DSPs. En el caso de la FFT en dos dimensiones. es posible reducir el tiempo de procesamiento de la FFT al aumentar la frecuencia base, por medio de los bloques DCM con los que cuenta el FPGA

Se validó completamente que un FPGA cumple con los requisitos para implementar un sistema que nos permite realizar el cálculo de la FFT.

El sistema de comunicación OFDM óptico cristográfico está descrito en el artículo de Ferrin y León del año 2010 [5]. En donde se afirma que el usuario de las tecnologías de información y comunicación (TIC's) cada día demanda mayor velocidad en transferencia de datos (principalmente multimedia) y a su vez la seguridad suficiente para proteger su contenido. En estos dos contextos Colombia ha tomado dos grandes decisiones que marcarán el futuro de la sociedad: por un lado, en 2008, la Comisión Nacional de Televisión (CNTV) seleccionó la norma de origen europeo DVB-T como el estándar de la Televisión Digital Terrestre (TDT) que se emitirá en el país. Y, por otro lado, en 2011, establece una política de ciberseguridad y ciberdefensa detallada en el CONPES 3701 (Lineamientos de Polìticas para Ciberseguridad y Ciberdefensa) que vigilará y protegerá la actividad de los usuarios en esta era de la información.

El esquema de modulación FFT-OFDM basado en la capa física del estándar IEEE 1901, es analizado en la tesis de Mitacc [11]. La arquitectura fue descrita mediante lenguaje de descripción de hardware VHDL y su implementación en un dispositivo FPGA.

El sistema diseñado consta de dos módulos principales, el transmisor y receptor. El primero se encarga de generar una señal OFDM a partir de una trama de entrada de 2^{12} bits, mientras que el segundo realiza el proceso inverso, es decir decodifica una trama de 2^{12} bits a partir de una señal de OFDM recibida. Para los procesos de modulación y demodulación, se emplean núcleos de IFFT y FFT de 2^{12} puntos y se utiliza el esquema QPSK para la codificación de cada una de las sub portadoras. Asimismo, ambos módulos transmisor y receptor, cuentan con mecanismos de codificación y corrección de errores, a fin de reducir la propagación de los mismos en los paquetes de datos transmitidos. La descripción en VHDL del sistema de transmisión-recepción diseñado fue sintetizada, utilizando las herramientas del software ISE 14.4 de Xilinx®, para el dispositivo FPGA Spartan-6/ XC6SLX45. Entre los resultados obtenidos, destaca que la máxima frecuencia de operación alcanzada por el sistema es de 107,68 MHz. Asimismo, en simulaciones realizadas de la operación del sistema en presencia de modelos de ruido periódico síncrono

y ruido periódico asíncrono, se obtuvieron cero errores para valores de SNR mayores a 11 DB.

En la tesis de Montoya [12], se analizan los diferentes tipos de modulación para sistemas PLC empleados en las redes energéticas inteligentes y se describen las características de los diferentes tipos de modulación. Las frecuencias utilizadas y el esquema de modulación son dos factores principales que tienen una influencia significativa en la eficiencia del sistema y también la velocidad del servicio PLC. La mejor técnica de modulación para esta tecnología es el OFDM, un esquema de modulación de múltiples operadores en los que un flujo de datos de alta velocidad se divide en múltiples corrientes de datos que son modulados por el uso de sub portadoras que son ortogonales entre si. Al evaluar el desempeño de OFDM para PLC se determina que puede solucionar los problemas de interferencias en un canal interior. Se evaluaron tres técnicas de codificación basadas en OFDM y la de mejor desempeño es la QSPK que se basa en fase y frecuencia. Los análisis fueron realizados hasta 30 MHz a través de un programa Matlab y en las simulaciones se utilizó un modelo de ruido blanco aditivo gaussiano distribuido en todo el espectro de frecuencias. En BPSK con el filtro IFFT la muestra de 2^{10} bits se divide por 4 y da una trama paralela de 2^8 bits transmitidos, tiene 2 dB de VER/SNR con mejor desempeño que las diversas técnicas QAM. En cambio, el QSPK presenta mejores resultados en la relación VER/SNR en comparación con BPSK. Los resultados indican que QPSK soporta las características aditivas del ruido impulsivo. En las simulaciones puede interpretarse que los 3 dB de QSPK la hacen inmune al ruido, lo que permite la rápida sincronización y re-sincronización para recuperar información.

En la tesis de Diaz [4], se presenta el algoritmo radix-2 con diezmado en frecuencia y reordenamiento en la salida de bits aplicado a transmisión de datos en un sistema OFDM y se observa un incremento significativo en la velocidad de transmisión de datos obtenidos en el laboratorio. Este algoritmo es conocido como decimación en el tiempo porque los datos que se reordenan son las muestras en el tiempo y radix-2 porque son dos grupos los que se forman (pares e impares). Las operaciones que se realizan en este algoritmo se conocen como operaciones de mariposa. La TDF y otras trasformadas en el espacio frecuencial en dos dimensiones tienen aplicación en el análisis de imágenes por diversas razones, algunas de ellas tienen que ver con los propósitos de mejorar las imágenes, o seleccionar ciertas características o estructuras de interés para hacer mediciones sobre la imagen a tratar. La extensión de 1 dimensión a 2 dimensiones para la TDF es sencilla. También se analiza el tiempo de ejecución de la FFT, con la cual se logra mayor velocidad

en la trasmisión de datos y eficiencia. El análisis del tiempo de ejecución del algoritmo indica que el proceso de recuperación es rápido para todas las pruebas.

En la revista de investigación de Pedraza [15], se ilustra la implementación de la FFT en hardware aplicada a recepción en OFDM

En esta investigación se ilustra la forma de implementar la transformada rápida de Fourier FFT usando el algoritmo de *Cooley -Tukey*, aplicado a los sistemas que requieren de recepción en OFDM. La investigación está basada en una arquitectura FPGA para obtener más rendimiento que un PC con un lenguaje de alto nivel; además, se desarrollan las diferentes partes de hardware necesarias para el cálculo de la FFT con ejemplos de aplicación de 64 y 128 puntos, con la posibilidad de ampliarla a 256 y 512 puntos. Muchos estándares alámbricos o inalámbricos han adaptado OFDM por su variedad de aplicaciones. Por ejemplo, es la base del estándar para ADSL (*Asymmetric Digital Subscriber Line*) y para DAB (*Digital Audio Broadcasting*) en el mercado europeo. En el entorno de las redes inalámbricas, OFDM es el corazón del estándar IEEE 802.11a y *HiperLAN/2*, el cual implementa OFDM de una forma similar. Luego de realizar simulaciones y pruebas de laboratorio resulta prometedor el desempeño del procesador FFT que se ha diseñado. Inicialmente se cuenta con un oscilador de 50MHz, lo que lleva al sistema a realizar un cálculo completo de la FFT de 64 puntos en 7.92µs, para un total de 126262 transformadas por segundo. Adicionalmente, se encuentra abierta la posibilidad de aumentar la frecuencia de la señal de reloj y adicionar un nuevo pipelining en el cual se puedan calcular etapas de la FFT en paralelo, lo que requeriría más memoria y recursos del FPGA, que actualmente no se encuentran disponibles. En el desarrollo de sistemas con procesamiento digital, los multiplicadores son los bloques que más recursos abarcan. Es posible reducir el hardware sacrificando velocidad de procesamiento, lo cual no es deseable para sistemas con restricciones temporales y de espacio. Reduce significativamente el consumo de potencia el espacio empleado para el sistema.

La simulación de ODFM con Matlab está descrito en el artículo de Lin del año 2010 [7]. OFDM aplicado a la evolución de 4G está descrito en el paper de Wang del año 2011[20].

1.2 Descripcion de la realidad problematica

Se necesitan dispositivos de alto rendimiento según el estándar IEEE802.11a (2013) del cual OFDM es la parte esencial y el procesamiento digital de señales es una de las áreas de mayor desarrollo en la industria de la computación. Este concepto ha causado un auge de los sistemas en chip (*SoC*), que permiten que se generen bloques funcionales de hardware, listos para ser sintetizados para dispositivos lógicos programables o para sistemas en VLSI, usados en el desarrollo de sistemas de comunicaciones. Uno de estos

bloques usados a menudo es el de la transformada rápida de Fourier (FFT), que permite reducir notablemente el área y el consumo de los sistemas basados en OFDM. Muchos estándares alámbricos o inalámbricos han adaptado OFDM por su variedad de aplicaciones. Por ejemplo, es la base del estándar para ADSL (*Asymmetric Digital Subscriber Line*) y para DAB (*Digital Audio Broadcasting*) en el mercado europeo. En el entorno de las redes inalámbricas, OFDM es el corazón del estándar IEEE 802.11a y *HiperLAN/2*, el cual implementa OFDM de una forma similar. Uno de los bloques usados a menudo es el de la transformada rápida de Fourier FTT, que permite reducir notablemente el área y el consumo de los sistemas basados en OFDM.

1.3 Formulacion del problema

EL desarrollo actual de la tecnolgia requiere incrementar la velocidad de transmisión de datos y se debe lograr mediante ia adaptación de un algoritmo de cálculo basado en FFT, que es una demanda de los tiempos actuales, en un sistema OFDM

1.4 Justificación e importancia de la investigación

La sociedad actual requiere de mayores velocidades de transmisión de datos y el presente trabajo de investigación se encuentra enfocado en el contexto de la adaptación de un algoritmo basado en FFT mediante el hardware digital para aplicar a un sistema OFDM, en los seis antecedentes revisados se muestra que es posible incrementar la velocidad de transmisión de datos y el avance en las tecnologías utilizadas en los sistemas de comunicación en general y en los sistemas Software Defined Radio1 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 portadoras en las que se modulan los datos a trasmitir. Una forma práctica y eficiente de implementar la modulación y la 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 implementar en forma eficiente tanto en consumo como en espacio y recursos utilizados, en 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.

1.5 Objetivos

1.5.1 Objetivo general

Mediante la adaptación de un algoritmo de cálculo basado en FFT se incrementará la velocidad de transmisión de datos en un sistema OFDM.

1.5.2 Objetivos específicos

Reducir el número de multiplicaciones por etapas, en el algoritmo radix-2

Reduccion del tiempo de ejecución de la transmisión de datos en el sistema OFDM

1.6 Hipótesis

1.6.1 Hipótesis general

H_0.- La velocidad de transmisión de datos en un sistema OFDM puede aumentarse mediante la adaptación de un algoritmo basado en FFT.

1.6.2 Hipótesis secundarias

H_1.- La adaptaciónde un algoritmo basado en FFT genera la reducción del número de multiplicaciones por etapas en un sistema OFDM.

H_2.- La adaptación del algoritmo basado en FFT permite la reducción del tiempo de ejecución en un sistema OFDM.

1.7 Variables e Indicadores

1.7.1 Variable independiente e indicadores

Algoritmo basado en FFT

a.- Tiempo de ejecución del algoritmo basado en FFT

b.- Velocidad de transmisión de datos en un sistema OFDM

1.7.3 Variable dependiente e indicador

Velocidad de transmisión de datos

Indicador

Velocidad de transmisión de datos

1.8 Unidad de análisis

Sistema OFDM implementado en una FPGAXC5VLX10

1.9 Tipo y nivel de investigación

La investigación es aplicada según Bunge al desarrollo que emplea los conocimientos adquiridos en la Maestria para resolver la adaptación e implementación del algoritmo basado en FFT para su aplicación a un sistema de comunicacion OFDM.

1.10 Periodo de análisis

Junio del 2017 hasta agosto del 2019

1.11 Fuentes de información e instrumentos utilizados

Las fuentes de información se han obtenido de revistas indexadas, tesis de maestría, libros especializados y artículos de investigación. los instrumentos utilizados son software y hardware.

1.12 Técnicas de recolección y procesamiento de datos

Mediante la revisión de revistas de investigación, tesis de maestría, libros especializados, se obtuvo la recolección y procesamiento de datos.

1.13 UML – Diagrama de Casos de Uso

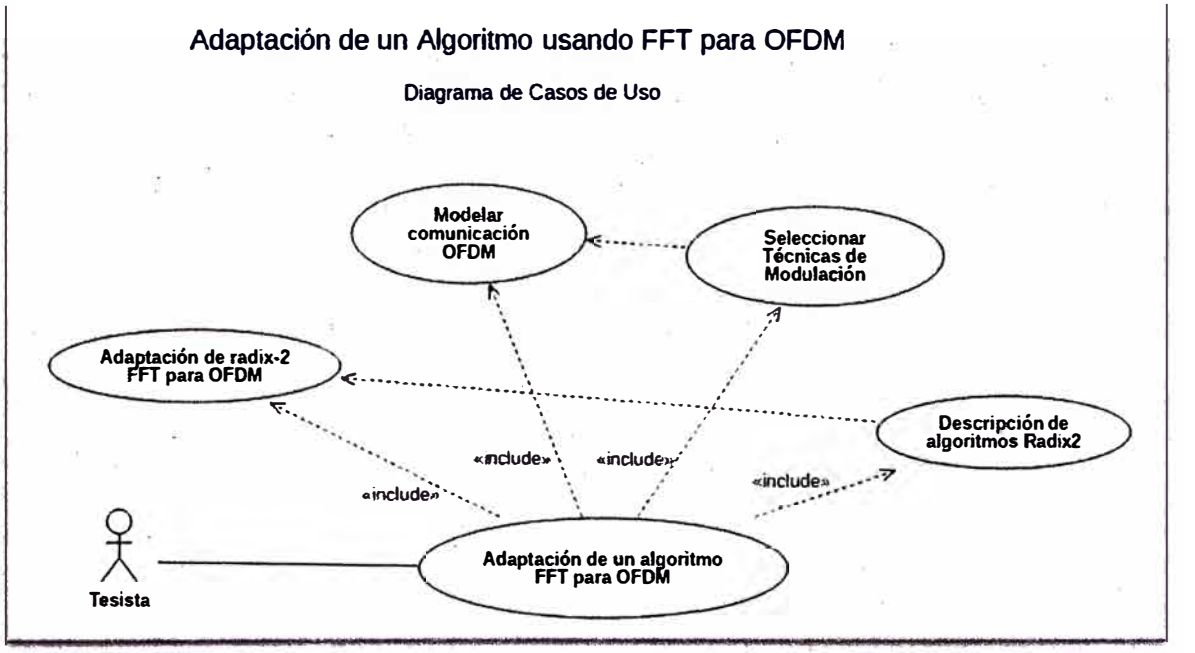


Figura 1.1: UML - Diagrama de Adaptación de un Algoritmo usando FFT para OFDM.

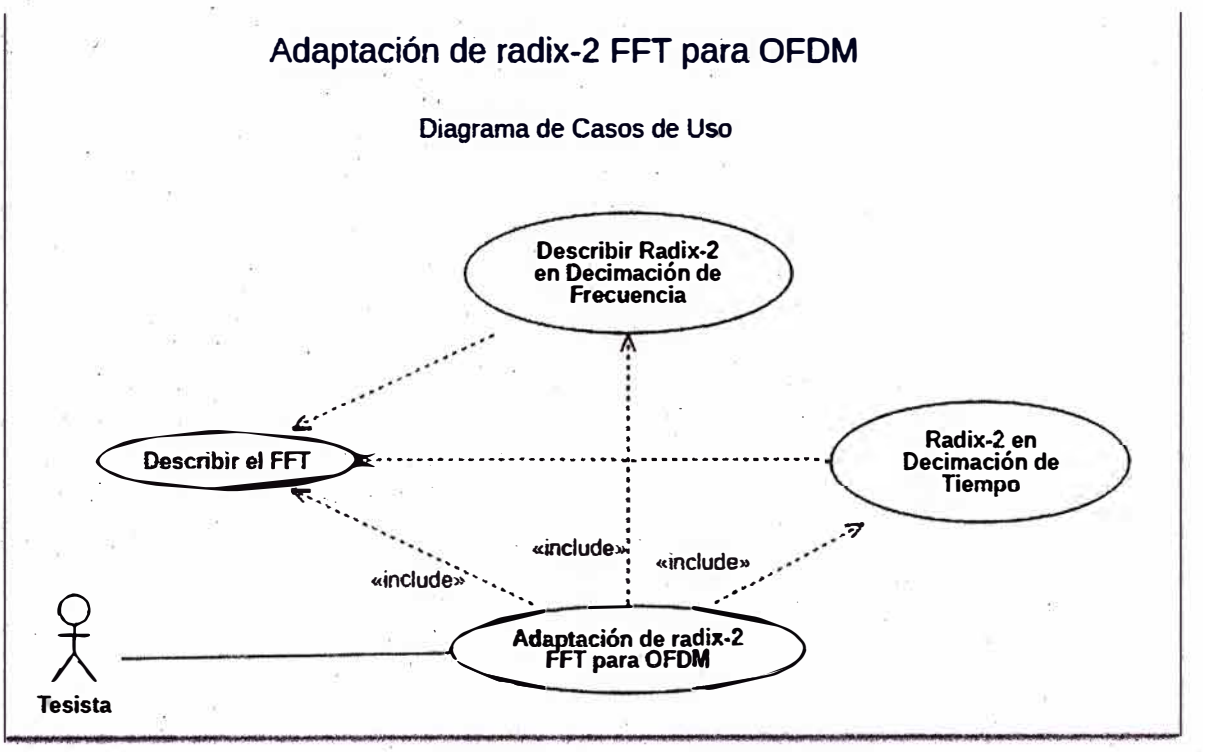


Figura 1.2: UML - Diagrama de Adaptación de radix-2 para OFDM.

CAPITULO II
MARCO TEORICO Y CONCEPTUAL

2.1 Algoritmos DFT

Los algoritmos FFT logran la eficiencia en el cómputo de la DFT (*Discret Fourier Transform)* a través de un mapeo multidimensional de los coeficientes.

Teniendo como base la descomposición en series de Fourier de una secuencia periódica y su transformada de Fourier através de sus coeficientes se llega a las ecuaciones (2.1) y (2.2) que definen la DFT, considerando una secuencia finita x(n) de longitud N como periodo, donde $W_N^{kn} = e^{-\frac{j2k\pi n}{N}}$ llamados los *twiddle factors*

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \tag{2.1}$$

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-kn} \tag{2.2}$$

2.2 Álgoritmos FFT

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

2.2.1 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 [17] y [18]. De la referencia [2], en particular los algoritmos de *Cooley-Tukey* que transforman N en una potencia de base r, $N = r^v$, son llamados radix-r y son los más populares.

La transformación de índices propuesta por *Cooley - Tukey* (y por Gauss previamente) es también la más simple, como se indica

$A = N_2$, $B = 1$, $C = 1$ y $D = N_1$ resultando en el mapeo:

$$n = N_2 n_1 + n_2 \qquad \begin{cases} 0 \le n_1 \le N_1 - 1 \\ 0 \le n_2 \le N_2 - 1 \end{cases} \tag{2.3}$$

$$k = k_1 + N_1 k_2 \qquad \begin{cases} 0 \le k_1 \le N_1 - 1 \\ 0 \le k_2 \le N_2 - 1 \end{cases} \tag{2.4}$$

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 (2.1) y (2.2):

$$W_N^{kn} = W_N^{N_2 n_1 k_1 + N_1 N_2 n_1 k_2 + n_2 k_1 + N_1 n_2 k_2} \tag{2.5}$$

Como W_N^{nk} es de orden $N = N_1 N_2$ se llega a que $W_N^{N_1} = W_{N_2}$ y $W_N^{N_2} = W_{N_1}$ aplicado en (2.5): se obtiene

$$W_N^{kn} = W_{N_1}^{n_1 k_1} W_N^{n_2 k_1} W_{N_2}^{n_2 k_2} \tag{2.6}$$

Reemplazando (2.1) en (2.2) se llega a:

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \left(W_N^{n_2 k_1} \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1} \right) \tag{2.7}$$

La sumatoria interior en (2.7) 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 (2.7) se llega a:

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \tilde{x}[n_2, k_1] \tag{2.8}$$

(2.8) 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 (2.3)
- 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 (2.4)

Nada impide aquí subdividir cualquiera de las DFT de la ecuación (2.8) en dos DFT de longitud menor sucesivamente hasta obtener DFT de longitud conveniente para realizar os 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 guardan en memoria en las mismas posiciones que los valores utilizados para el cálculo, utilizando en forma

eficiente la memoria ya que para el cálculo de una DFT de *N* puntos solo se requiere una memoria de longitud *N*.

2.2.2 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 $N = r^v$. Como se vió en la referencia [4] a los algoritmos de este estilo se los conoce como algoritmos radix-r. De esta manera el cálculo de la DFT se descompone en *v* DFTs consecutivas de *r* puntos cada una. Por ejemplo, para $r = 2$ y *v* etapas, $N = 2^v$, el mapeo de índices queda como:

$$n = 2^{v-1} n_1 + \ldots + 2n_{v-1} + n_v \tag{2.9}$$

$$k = k_1 + 2k_2 + \ldots + 2^{v-1} k_v \tag{2.10}$$

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

$$\log_2(N) N / 2 \tag{2.11}$$

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 ó 4 no es necesario realizar ninguna multiplicación no trivial a excepción del producto por el *twiddle factor*, eligiendo *N* como potencia de 2 ó 4 se reduce la cantidad de multiplicaciones a realizar aumentando así la eficiencia del algoritmo.

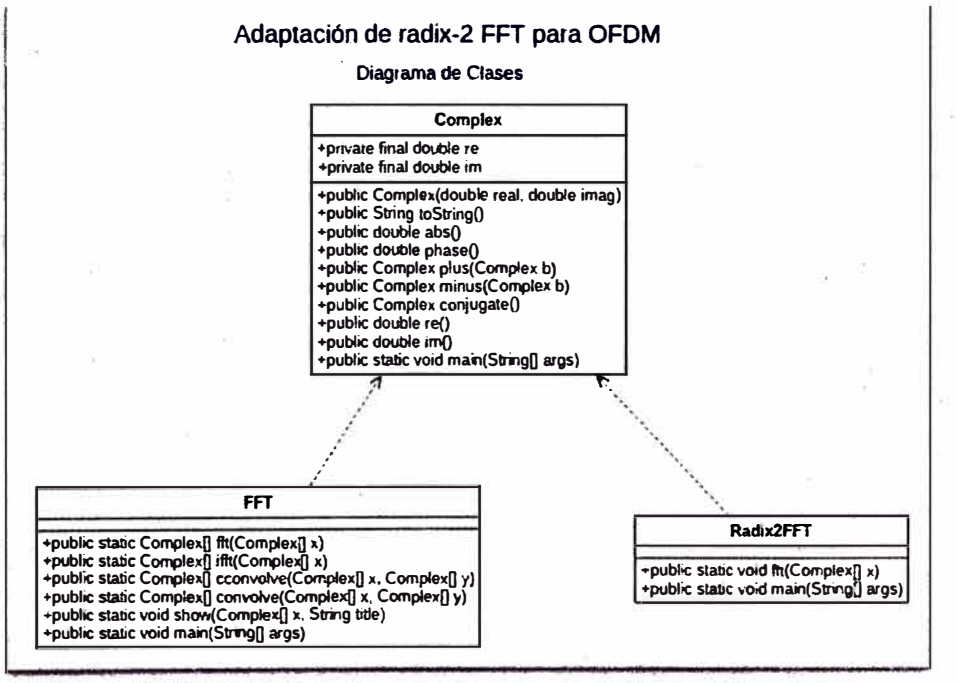


Figura 2.1: Diagrama de Clases de la Adaptación de radix-2 FFT para OFDM (Fuente: Referencia [18])

2.2.3 Transformada de Fourier en Tiempo Discreto

La TDF es una de las herramientas más importantes en el procesamiento digital de señales. Visto en las referencias [2], [11] el cálculo de la TDF requiere un gran número de multiplicaciones y adiciones complejas. Para N puntos, la TDF requiere N^2 multiplicaciones complejas y $(N - 1)$ adiciones complejas. Por ejemplo, para el cálculo de 8 puntos se requieren $8^2 = 64$ multiplicaciones complejas y $8(8 - 1) = 56$ sumas complejas. Si $N = 1024$, entonces se requieren $(1024)^2$ multiplicaciones y 1024x1023 sumas, aproximadamente 1 millón de multiplicaciones complejas y sumas. Los algoritmos de la FFT (Fast *Fourier Transform*) reducen la carga computacional para calcular la TDF. Los algoritmos más populares para la obtención de la FFT están basados en el algoritmo de Cooley-Tukey y son los de radix-4 y radix-2, de los cuales el último es el más empleado. En la tabla 2.1 se muestra una comparación en la eficiencia del cálculo de la FFT.

N	DFT – N^2 Multiplicaciones complejas y operaciones de Suma	FFT - $N\log_2 N$ Multiplicaciones complejas y operación de suma	Esfuerzo computacional de la FFT comparada con la DFT (%)
8	64	24	37.50
32	1024	160	15.62
256	65536	1048	3.12
1024	1048576	10240	0.98
4096	16777216	49152	0.29

Tabla 2.1: Comparación del esfuerzo computacional de la DFT y la FFT

2.2.4 La Transformada Rápida de Fourier

La implementación directa de la DFT según 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 inaceptable en sistemas escalables. Por esto históricamente se buscaron formas más eficientes de realizar el cálculo de la DFT. Estos algoritmos se conocen globalmente como la transformada rápida de Fourier (FFT), que permiten reducir la complejidad al orden de $\Theta(N * \log(N))$ [13][11]. Los algoritmos de FFT utilizan la estrategia divide y vencerás a través de la transformación de una DFT de longitud N en una representación multidimensional $N = \pi_i N_i$. Esto permite dividir el cálculo de una DFT de N puntos en múltiples DFT de N_i reduciendo de esta manera la complejidad computacional total. De la referencia [11] como ejemplo se analizará brevemente la descomposición en dos dimensiones. Se transforma el índice temporal n según

$$n = (An_1 + Bn_2) \bmod N \qquad \begin{cases} 0 \le n_1 \le N_1 - 1 \\ 0 \le n_2 \le N_2 - 1 \end{cases} \tag{2.12}$$

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 $f : C^N \rightarrow C^{N_1 . N_2}$. Aplicando una transformación similar al índice k en el dominio de la frecuencia:

$$k = (Ck_1 + Dk_2) \bmod N \qquad \begin{cases} 0 \le k_1 \le N_1 - 1 \\ 0 \le k_2 \le N_2 - 1 \end{cases} \tag{2.13}$$

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.

2.2.5 Radix-2 en decimación de frecuencia

El algoritmo de decimación en frecuencia para la obtención de la FFT está basado en la descomposición de la TDF de N-puntos, en la cual se va dividiendo la secuencia de salida hasta obtener una TDF de dos puntos. El reordenamiento de la TDF se realiza separando los valores con índice par de los datos discretos de la frecuencia $[k]$ = [0,2,4, …, N − 2] y los valores con índice impar de los datos discretos de la frecuencia $X[k]$ = [1,3,5, …, N−1]. El diagrama de la figura 2.2 muestra este algoritmo para 8 puntos.

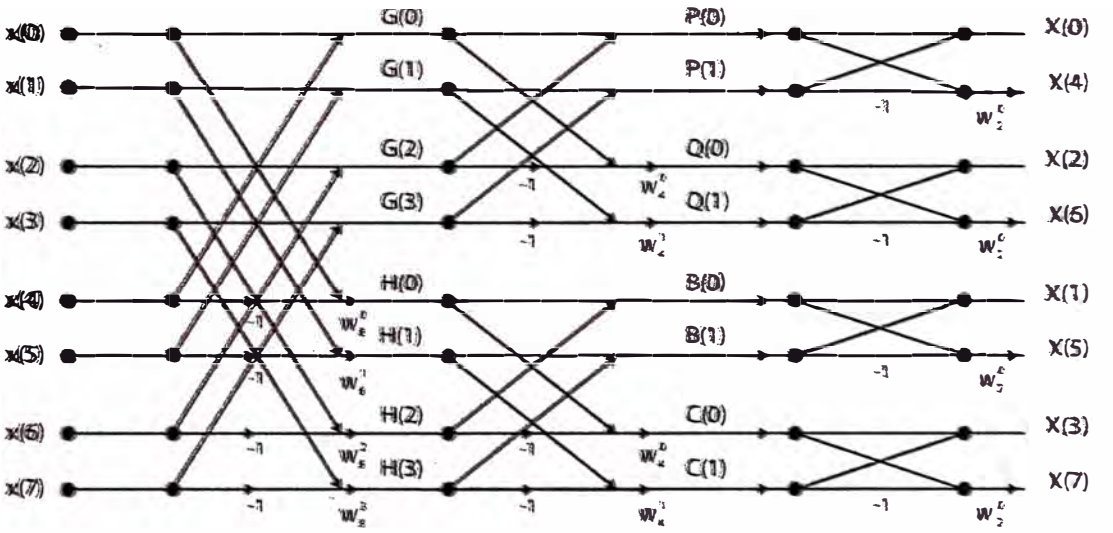


Figura 2.2: Diagrama de mariposa para la obtención de la FFT en decimación en frecuencia

(Fuente: Referencia [4])

Este algoritmo es llamado *decimación en frecuencia* porque las muestras que se separan en grupos de dos, son los datos discretos de la frecuencia. El número total de multiplicaciones para el algoritmo radix-2 en decimación en frecuencia es de $\frac{N}{2}log_2 N$ multiplicaciones complejas.

2.2.6 Radix-2 en decimación de tiempo

Este algoritmo reordena la ecuación de la TDF en dos partes: la suma de los valores discretos con índices pares $n = [0, 2, 4, \ldots, N-2]$ y la suma de los valores discretos con índices impares $n = [1, 3, 5, \ldots, N-1]$ como se muestra en la siguiente ecuación.

$$X[k] = \sum_{N=0}^{N-1} x[n]e^{-j2\pi(nk/N)} \tag{2.18}$$

$$X[k] = \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-\left(j\frac{2\pi(2n)k}{N}\right)} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-\left(j\frac{2\pi(2n+1)k}{N}\right)} \tag{2.19}$$

$$X[k] = \sum_{n=0}^{\frac{N}{2}-1} x(2n)e^{-\left(j\frac{2\pi nk}{\frac{N}{2}}\right)} + e^{-\left(j\frac{2\pi k}{N}\right)} \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)e^{-\left(j\frac{2\pi nk}{\frac{N}{2}}\right)} \tag{2.20}$$

$$X[k] = TDF_{\frac{N}{2}}[x(0), x(2), \ldots, x(N-2)] + W_N^k TDF_{\frac{N}{2}}[x(1), x(3), \ldots, x(N-1)]$$

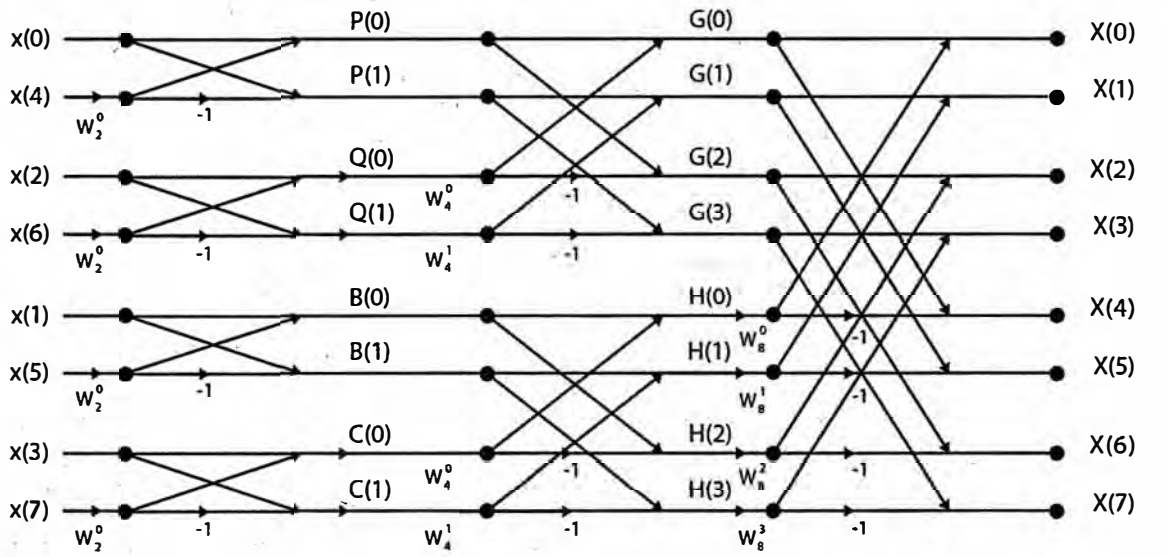


FIGURA 2.3: Diagrama de mariposa del algoritmo radix-2 en decimación en el tiempo para la obtención de la FFT con N = 8

(Fuente: Referencia [4])

Este algoritmo es conocido como decimación en el tiempo porque los datos que se reordenan son las muestras en el tiempo y radix-2 porque son dos grupos los que se forman (pares e impares). Las operaciones que se realizan en este algoritmo se conocen como *operaciones de mariposa* y el diagrama de la figura 2.4 muestra este algoritmo para 8 puntos.

Cada proceso de decimación requiere $\frac{N}{2}$ multiplicaciones complejas. El número total de multiplicaciones complejas requeridas para completar todo el proceso es de $\frac{N}{2}\log_2 N$, por lo tanto el numero de operaciones se reduce de N^2 a $\frac{N}{2}\log_2 N$.

En la figura 2.3 se muestra un esquema simplificado de un algoritmo radix-2 de 8 puntos.

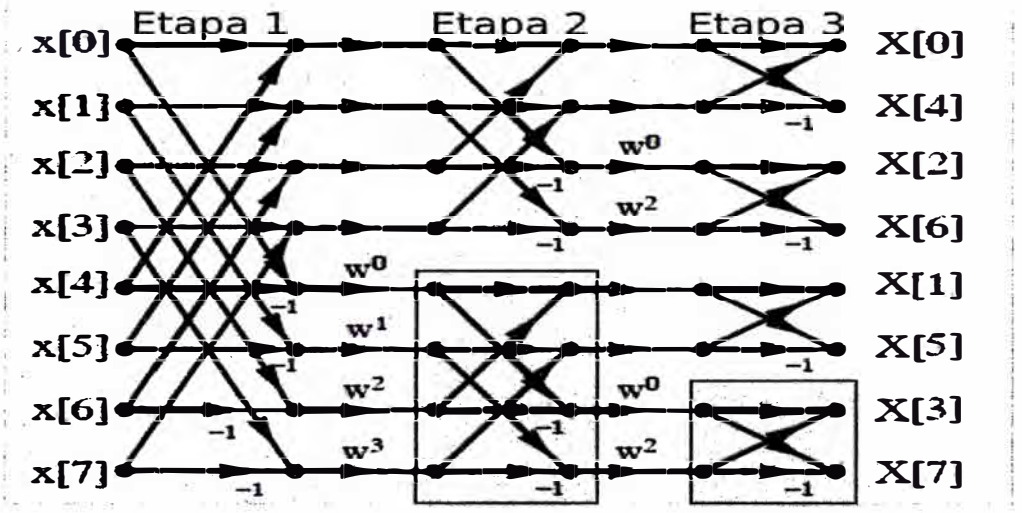


FIGURA 2.4: FFT Radix-2 de 8 puntos
(Fuente: Referencia [4])

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 mariposas por la forma que toman las dos flechas cruzadas en el esquema de la figura 2.2, 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 *mariposas,* generalizando $(N/2)*\log_2(N)$, y 8 multiplicadores complejos, generalizando $(N/2)*(\log_2(N) - 1)$. Existen diferentes variantes en la forma en que se adapta 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 adaptan todas las unidades mariposa y multiplicadores complejos necesarios, dispuesto en una arquitectura similar al esquema de la figura 2.2. Se puede adaptar en forma pipeline 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) [15]. Se muestra un esquema de esta arquitectura para una FFT de 8 puntos en la figura 2.5. La comunicación se realiza mediante un bus en serie, admitiendo un dato de entrada en cada ciclo de reloj. Se utiliza una unidad butterfly por etapa, requiriendo en total $\log_v(N)$, y un multiplicador complejo por etapa excepto en la última dando un total de $\log_v(N) - 1$ multiplicadores. Se puede adaptar en forma pipeline colocando un registro en medio de etapas consecutivas.

- **Iterativa** Se utiliza una única unidad mariposa y un único multiplicador complejo para realizar secuencialmente las operaciones de todas las etapas. En la figura 2.6 se ve un esquema de la arquitectura radix-2 iterativa para 8 puntos.

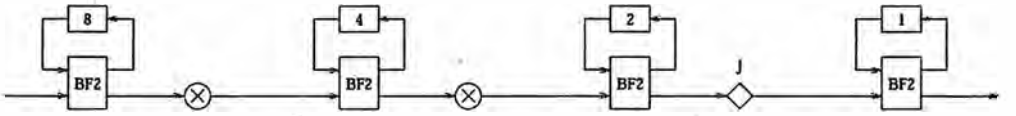


FIGURA 2.5: Arquitectura radix-2 desenrollada SDF
(Fuente: Referencia [2])

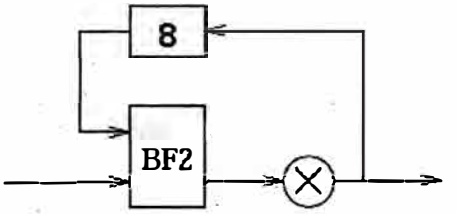


FIGURA 2.6: Arquitectura Radix-2 iterativa
(Fuente: Referencia [2])

En la tabla 2.2 se realiza la comparación entre las características distintivas de cada adaptación. El *throughput* se toma como la cantidad y frecuencia con que se obtienen puntos a la salida de la arquitectura. El campo *pipeline* indica si la adaptación puede ser implementada de esa manera. En el caso del tamaño de memoria requerido no se consideran implementaciones *pipeline*, ya que en ese caso para la adaptación paralela hay que colocar N registros por cada etapa de *pipeline* que se coloca, mientras que en la adaptación desenrollada se necesita un registro por etapa de *pipeline*.

Característica	Radix paralela	Radix desenrrollada	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	N	$N - 1$	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 2.2: Comparación entre las adaptaciones 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 mariposa y multiplicadores en las adaptaciones 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 adaptación 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 rendimiento, se puede operar la unidad de procesamiento FFT a una velocidad de reloj mayor al resto del sistema para obtener el rendimiento necesario.

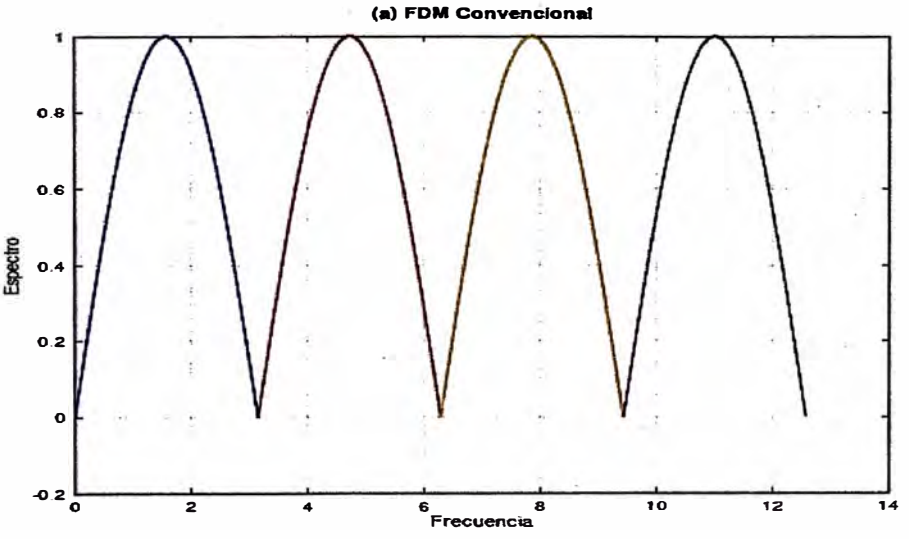
Se puede ver que la relación de tamaño entre la adaptación desenrollada y la iterativa es del orden de $log_2(N)$: 1 para los módulos de cómputo aritmético. En base a la comparativa realizada y al requerimiento de que la arquitectura sea económica en términos espaciales y de consumo se opta por la adaptación iterativa, ya que al aumentar N solo se incrementa la cantidad de memoria requerida manteniendo una arquitectura de tamaño reducido y bajo consumo.

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. Como se muestra en la referencia [11] y [4]. Tratar de llevar la velocidad de transmisión de símbolos para alcanzar tasas de bits del orden del Mbit o mayor se requiere la utilización de ecualización adaptativa lo que eleva la complejidad y el costo de los equipos utilizados [16]. 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 como indica la referencia [12]. Otra ventaja de la transmisión multiportadora es su robustez frente a interferencia selectiva en frecuencia ya que solo se verían afectadas alguna 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 [16]. En la figura 2.6 se ve la diferencia en el aprovechamiento del ancho de banda al transmitir 4 subportadoras separadas y las mismas 4 subportadoras superpuestas. El sistema de multiplexado en división de frecuencias ortogonales (*OFDM, Orthogonal Frecuency-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 (2.7) muestra un conjunto de subportadora 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 [16]. 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 medios de transmisión de información.



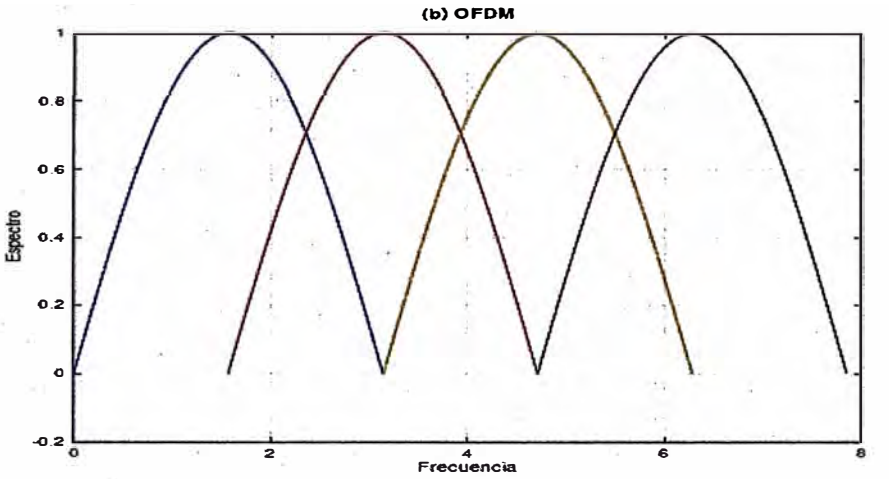


FIGURA 2.7: Dos formas de transmisión multiportadora

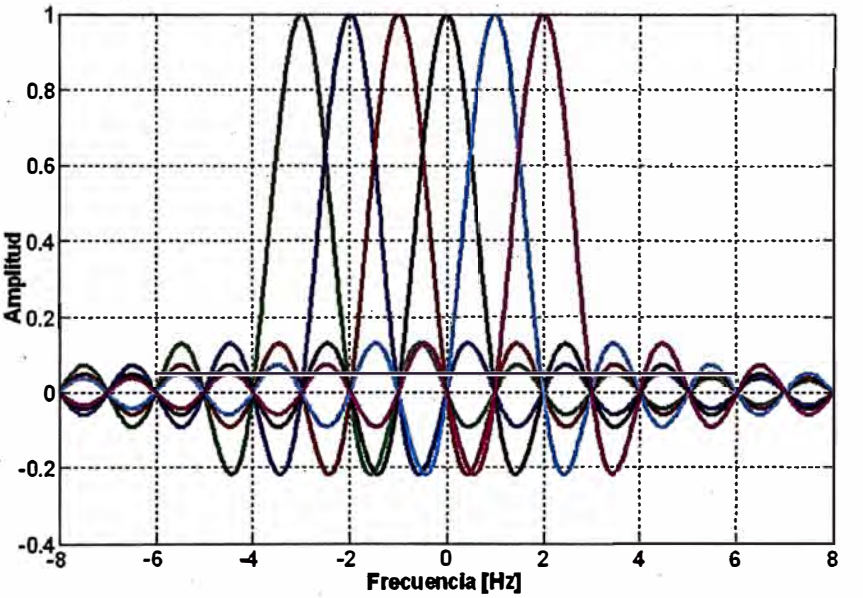


FIGURA 2.8: Muestra espectral de una señal OFDM

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*) utilizado para reducir el efecto del retrazo 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 (*Tguard*) 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.

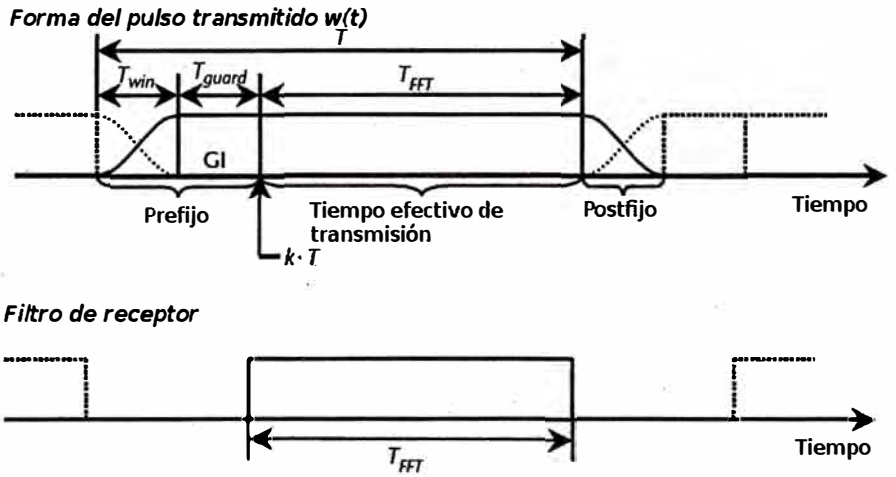


FIGURA 2.9: Diagrama de tiempos de un símbolo OFDM
(Fuente: Referencia [11])

Esto queda mostrado en la figura (2.9) 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 (2.9) se observa un diagrama en bloques de un transmisor OFDM. La data de origen es dividida en paquetes/canales y mapeada a lo símbolos respectivos de la constelación correspondiente

a la modulación seleccionada para la transmisión. Esos símbolos, que componen una constelación de símbolos complejos, es modulada 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 (2.10) [8]

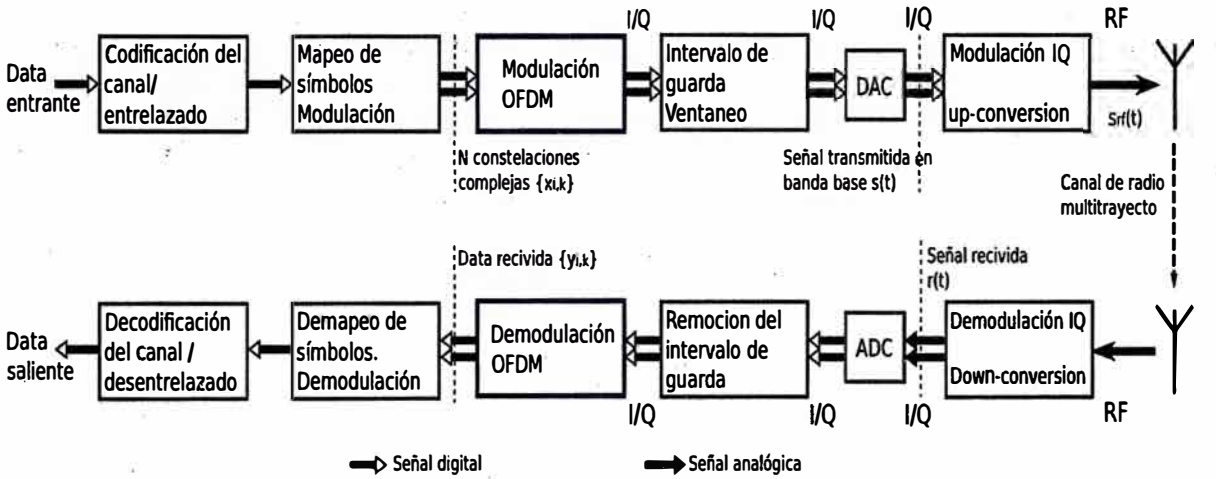


FIGURA 2.10: Diagrama en bloques de una transmisión punto a punto OFDM (Fuente: Referencia [11])

Adicionalmente al diagrama de puntos y de bloques de transmisión de punto a punto en OFDM se presentan el diagrama de bloques conceptual de un sistema de comunicación OFDM en la figura 2.10 y el diagrama de bloques de la técnica de modulación OFDM en la figura (2.11) y (2.12)

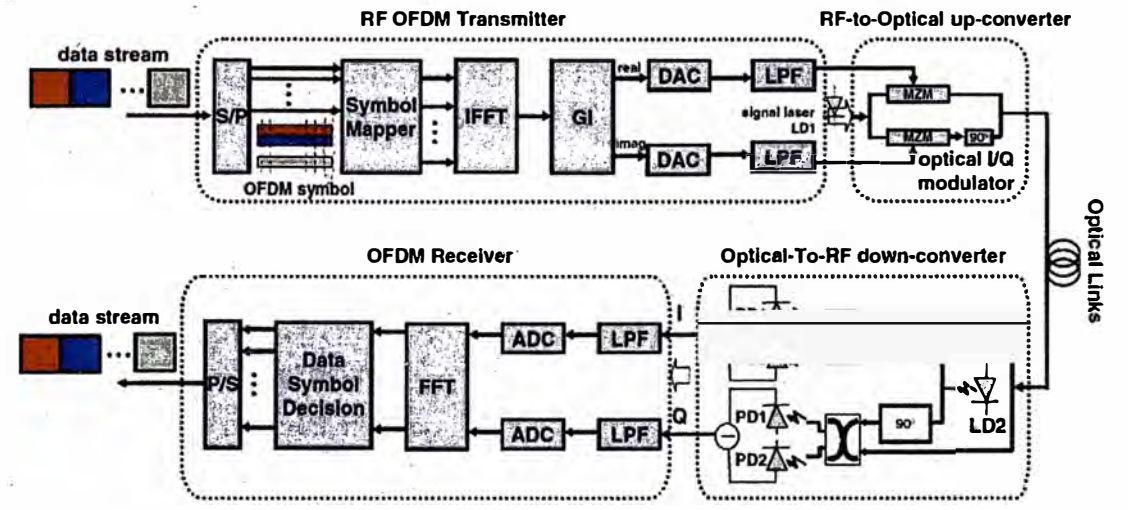


FIGURA 2.11: Diagrama de bloques conceptual de un sistema comunicación OFDM.
(Fuente: Referencia [11])

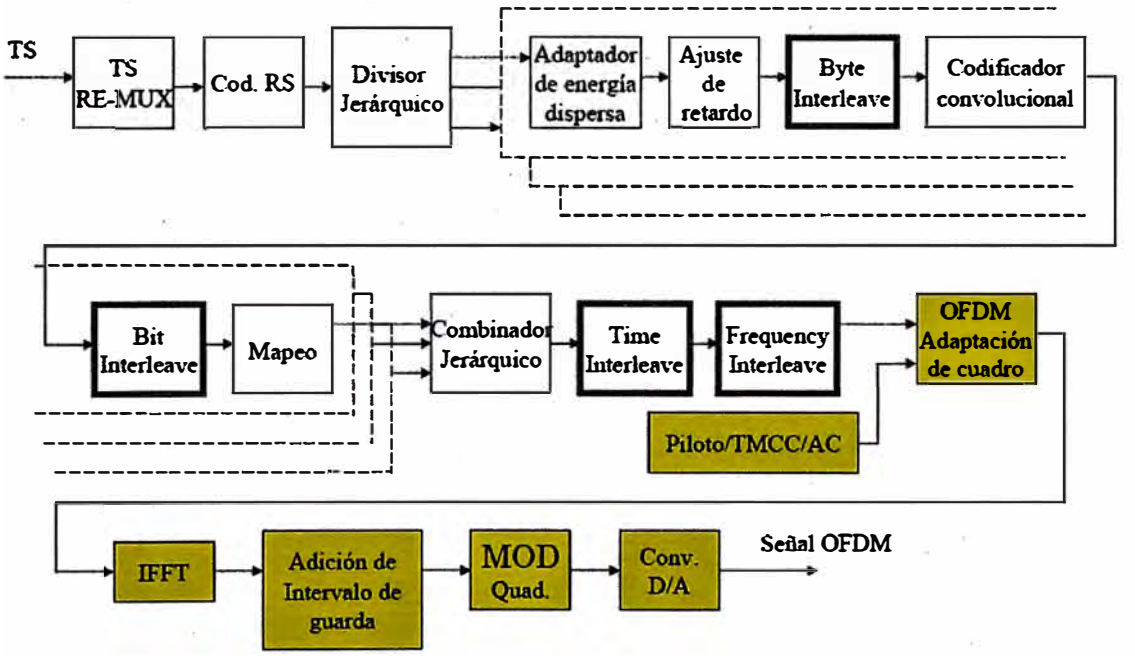


FIGURA 2.12: Técnica de Modulación OFDM
(Fuente: Referencia [12])

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. Referencias [11] y [12]. En notación temporal, se expresa el k-ésimo símbolo de una señal OFDM se escribe como:

$$s_{RF,k}(t-kT) = \mathrm{Re}\left\{ \omega(t-kT) \sum_{i=-\frac{N}{2}}^{\frac{N}{2}-1} x_{i,k} e^{\left(f_c + \frac{i}{T_{FFT}}\right)(t-kT)} \right\} \quad (2.14)$$

para $kT - T_{win} - T_{guard} \le t \le kT + T_{FFT} + T_{win}$ *k y 0* para todo otro caso. Donde:

T: Duración de símbolo, tiempo entre dos símbolos consecutivos T_{FFT}:

T_{FFT}: Parte efectiva de un símbolo

T_{guard}: GI; duración del prefijo cíclico

T_{win}: intervalo de ventana

k: Índice del símbolo transmitido

i: Índice de subportadora, *i ∈{Ñ/2, Ñ/2 + 1, ... ¨1},{ 0, 1, ...Ñ/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:

$$s_{RF}(t) = \sum_{k=-\infty}^{\infty} s_{RF,k}(t-kT) \quad (2.15)$$

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

$$s(t) = \sum_{k=-\infty}^{\infty} s_k(t-kT) \quad (2.16)$$

donde

$$s_k(t-kT) = \omega(t-kT) \sum_{i=-\frac{N}{2}}^{\frac{N}{2}-1} x_{i,k} e^{j2\pi\left(\frac{i}{T_{FFT}}\right)(t-kT)} \quad (2.17)$$

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

Se puede notar la similitud entre (2.15) y (2.16) 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 y recursos

2.3.3 Arquitecturas radix-r

Como se menciona en la sección 2.3.1 y la referencia [2], la arquitectura a adaptar será de tipo radix-r. En estos algoritmos se deben calcular v DFTs de r puntos cada una donde $N = r^v$. El valor de r toma importancia ya que la correcta elección del mismo puede conducir a adaptaciones más eficientes. En la tabla 2.3 se muestra la cantidad de operaciones necesarias para bloques de DFT de distintos tamaños. Se considera como multiplicaciones triviales aquellas que son por±1 y ±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 2.3: Cantidad de operaciones complejas para distintas longitudes de DFT (Fuente: Referencia [4])

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 adaptación ya que en cada bloque se deben procesar DFTs de r puntos. Teniendo en cuenta esto se decide adaptar la arquitectura en dos versiones, utilizando N = 2 y N = 4, ya que al ser valores pequeños permiten adaptar la arquitectura para diversos valores de N manteniendo la complejidad de la adaptación 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 adaptaciones comerciales de bloques de cálculo de DFT se realizan con arquitecturas radix-2 y radix-4.

2.3.4 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) que realiza rotaciones en dos dimensiones (además de otras operaciones trigonométricas) en base únicamente a sumas/restas y desplazamientos [2] este algoritmo fue presentado en 1966 por *Volder* [19] y es ampliamente utilizado para el cálculo de funciones trigonométricas en sistemas digitales.

El principio de funcionamiento del algoritmo es realizar micro rotaciones 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 θ_n. Esta rotación se lleva a cabo mediante micro rotaciones por un ángulo θ_i

$$x_{i+1} = x_i * \cos\theta_{i+1} - y_i * \sin\theta_{i+1} \quad (2.21)$$

$$y_{i+1} = y_i * \cos\theta_{i+1} + x_i * \sin\theta_{i+1} \quad (2.22)$$

Factorizando θ_{i+1} en (2.21) y (2.22) se obtiene:

$$x_{i+1} = \cos\theta_{i+1}(x_i - y_i * \tan\theta_{i+1}) \quad (2.23)$$

$$y_{i+1} = \cos\theta_{i+1}(y_i + x_i * \tan\theta_{i+1}) \quad (2.24)$$

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:

$$K_i = \cos(\arctan 2^{-i}) = \frac{1}{\sqrt{1+2^{-2i}}} \quad (2.25)$$

$$d_i = \pm 1 \quad (2.26)$$

Sabiendo que el coseno es una función impar, por lo que $\cos(\alpha) = \cos(-\alpha)$, y reemplazando (2.25) y (2.26) en (2.23) y (2.24) se obtienen las ecuaciones finales para el algoritmo *Cordic*:

$$x_{i+1} = K_i(x_i - y_i * d_i * 2^{-i}) \quad (2.27)$$

$$y_{i+1} = K_i(y_i + x_i * d_i * 2^{-i}) \quad (2.28)$$

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:

$$K = \prod K_i = \prod \frac{1}{\sqrt{1+2^{-2i}}}$$
$$i = 1,2,...n \qquad (2.29)$$

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

$$z_{i+1} = z_i - d_i \arctan 2^{-i} \qquad (2.30)$$

Y para decidir el valor de d_i se utiliza

$$d_i = \begin{cases} -1 & \sin z_i < 0 \\ 1 & \sin z_i \geq 0 \end{cases} \qquad (2.31)$$

$\arctan 2^{-i}$ 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 2.13 puede verse como se logra una rotación a través de micro rotaciones a lo largo de 4 iteraciones.

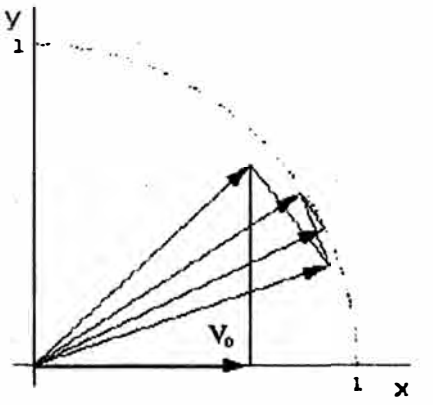


FIGURA 2.13: Ejemplo de rotación con algoritmo *Cordic*
(Fuente: Referencia [19])

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

2.3.5 Multiplicador complejo eficiente

El uso extendido del algoritmo *cordic* en arquitecturas de cómputo de FFT se justifica por el costo espacial de adaptar 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 adaptación 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:

$$R + jI = (A + jB) * (C + jD) = (A * C - B * D) + j(A * D + B * C) \qquad (2.32)$$

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

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

$$E = A - B$$
$$Z = C \times E = C \times (A - B) \qquad (2.33)$$

Y se computa el producto final como:

$$R = (C - D) \times B + Z \qquad (2.34)$$
$$I = (C + D) \times A - Z \qquad (2.35)$$

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 el desempeño. Teniendo en cuenta que muchas FPGA [10] cuentan con unidades de procesamiento de señales, incluyendo multiplicadores, la adaptación de un multiplicador complejo tiene amplias ventajas tales como otros algoritmos.

2.3.6 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)* de la unidad aritmética. Para almacenar 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 arquitectura en cada etapa de cómputo.

Otra opción es la adaptación 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 desplazamiento hacia la derecha. Debe tenerse en cuenta que este método puede introducir error en el resultado ya que al dividir por 2 se puede perder información. Se decide utilizar la segunda opción por su simplicidad de adaptación y por que su tamaño no depende de la cantidad de etapas a adaptar. Para esto se evalúa la adaptación 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 2.4 se muestran algunos ejemplos de la diferencia entre truncamiento y redondeo al quitar el último digito 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 2.4: Ejemplos de truncamiento y redondeo en sistema decimal

(Fuente: Referencia [11])

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 al aplicar cualquiera 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.

2.3.7 Arquitecturas a adaptar

De lo expuesto a lo largo del capítulo se decide adaptar 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.

CAPITULO III
DESARROLLO DEL TRABAJO DE LA TESIS

En este capítulo se describe la adaptación del algoritmo radix-2 y los módulos que componen las dos estructuras seleccionadas y la forma en que esos módulos se interconectan. También se presentan las herramientas utilizadas para el diseño y adaptación de los algoritmos.

El proceso de desarrollo se dividió en tres etapas, para verificar las hipótesis planteadas Primero, la reducción del número de multiplicaciones por etapa, aumentará la velocidad de transmisión de datos en un sistema OFDM, segundo al reducir el tiempo de ejecución por etapas, se logra aumentar la velocidad de transmisón de datos en un sistema OFDM y tercero al verificarse las dos hipótesis anteriores; entonces, la adaptación del algoritmo radix-2 cumplirá satisfactoriamente el objetivo de la presente tesis.

Una vez seleccionados los algoritmos, se conformó un diagrama en bloques identificando los módulos constitutivos, luego se adaptó 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 estructura se realizaron pruebas de verificación, cuyos resultados se presentan en el capítulo 4.

3.1 Desarrollo de la adaptación del algoritmo Radix-2

3.1.1 Algoritmos DFT y FFT

La primera métrica habitual para 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; 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 qué tipo de algoritmo se utilizará, DFT o FFT. Para este trabajo de tesis se ha seleccionado un algoritmo FFT ya que el mapeo multidimensional de los índices expresados en las ecuaciones (2,12) y (2,13) del capituo II, permite dividir el cómputo de

la DFT en cómputos más pequeños reduciendo la complejidad de la implementación y permitiendo el uso de pipelines entre cada etapa de cómputo aumentando la velocidad de procesamiento.

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, expresados en las ecuaciones (2,7) y (2.8); pero, ofreciendo la posibilidad de utilizarla para cualquier valor de N. Esta es la razón por lo que se utilizará una variante del algoritmo *Cooley-Tuckey* el radix-2 para la adaptación de este trabajo de investigación; 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 adaptada.

3.1.2 Estructura Radix-2

3.1.2.1 Descripción General

Como se explicó en la sección 2.3.2 ,al adaptar 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-2, debiendo esperar $N=2^v$ ciclos de reloj entre cada punto de entrada y entre cada punto de salida, durante los cuales se realiza una operación de cada etapa.

La figura 3.1 muestra el esquema presentado en la sección 2.2.6. 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.

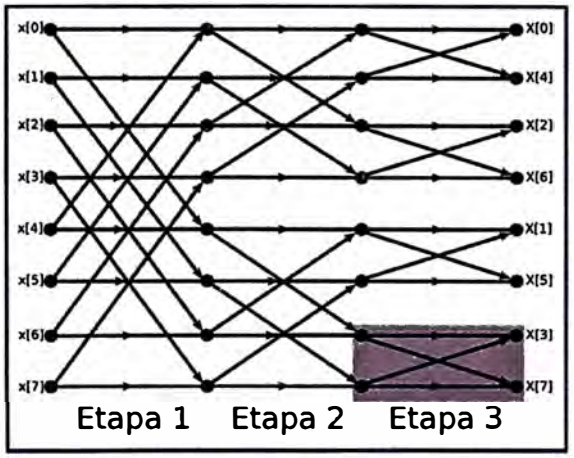


FIGURA 3.1: FFT radix-2 de 8 puntos

(Fuente: Referencia [11])

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 a la mariposa 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 adaptar, se ejecuta en cada ciclo de *clock* una operación de una etapa, pasando a la siguiente etapa en el ciclo de reloj siguiente, de forma que al transcurrir $\log_2(N)$ ciclos de reloj se ejecutó una operación de cada etapa volviendo a la etapa inicial

La figura 3.1 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 valores 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.

3.1.3 Formas de adaptar el algoritmo radix-r

Para adaptar el algoritmo radix-r, en particular el algoritmo radix-2, para este trabajo de investigacion con $N=2^3$, se representan los diagramas para la obtención de la FFT en decimacion en frecuencia figura 3.1 y decimacion en tiempo figura 3.2 y la referencia [4].

3.1.4 Multiplicación por los *twiddle factors*

La adaptación 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 adaptar los multiplicadores no es un tema trivial. En una arquitectura desenrollada se necesitan $\log_2(N) - 1$ multiplicadores, dándole principal importancia al aspecto espacial de la adaptación, 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*.

3.2 Interfaces de las estructuras

Las dos estructuras implementadas tienen los mismos puertos de entrada y salida mostrados en la figura 3.29.

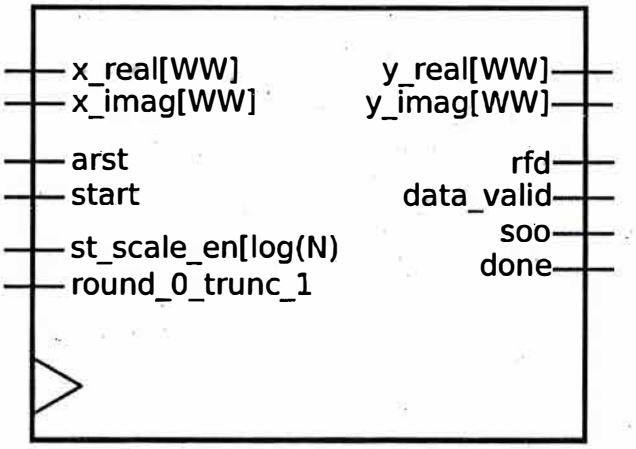


FIGURA 3.29: Señales de comunicación de las estructuras implementadas (Fuente: Referencia [14])

Los puertos se describen en el siguiente listado:

- **x_real** (W ORD_W IDT H) componente real del punto de entrada.
- **x_imag** (W ORD_W IDT H) componente imaginaria del punto de entrada.
- **arst** 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 primera FFT.
- **st_scale_en** ($\log_v(N)$) Habilitación del escalamiento posterior a la operación aritmética. Un t_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_1_IFFT_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.obs/ ddf/pch

3.3 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 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 [21] 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 situado 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.

CAPITULO IV

ANALISIS Y PRESENTACION DE RESULTADOS

En este capítulo se describen los resultados, verificación y validación del algoritmo Radix- 2 adaptado.

4.1 Simulación y analisis de resultados

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 simulacion y análisis 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.

Como el objetivo es aumentar la velocidad de transmision de datos, reduciendo el numero de multiplicaciones por etapas que efectua el algoritmo radix-2 y la reducción del tiempo de ejecución, se comprobó que se aumenta la velocidad de transmisión de datos en un 10%, disminuyendo el tiempo de ejecución en 10% del algoritmo Cooley-Tukey al algoritmo radix-2, al reducir el tiempo de ejecución de 12,5 us a 7,5 us, se aumenta la Transmision de datos en forma significativa. A continuación, se muestran las tablas comparativas

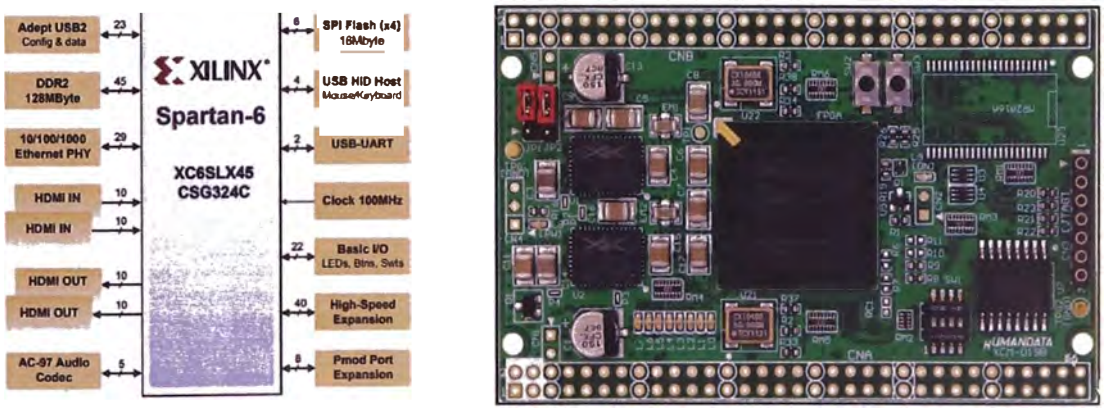


FIGURA 4.1: Pruebas con FPGA

(Fuente: Referencia [21])

La tabla (4.1) es la comprobación de la hipótesis 1 (al reducirse el número de

multiplicaciones por etapa aumentará la velocidad de transmisión de datos Bits/seg).

N° Multiplic.	1048576	4096	3072	1024
Cooley-Tukey	16us	14,5us	12us	8,5us
Radix-2	14,04us	13,05us	10,6us	7,5us

Tabla 4.1. Numero de multiplicaciones y velocidad de transmisión de datos

A continuación, se muestran en la tabla (4.2) los cuadros del algoritmo Cooley –Tukey, para entradas de 1024 puntos, 12 bits y 16 bits; 4096 punto, 12 bits y 16 bits, en este caso se usó como comparativa el algoritmo Kiss FFT que es una variante mejorada del FFT.

Entradas,Tiempo	**1024, 12**	**1024, 16**	**4096, 12**	**4096, 16**
Cooley-Tukey	14bit/s	14,5bit/us	15bit/us	16,5bit/us
Kiss FFT	13bit/us	14bit/us	16bit/us	15bit/us

Tabla 4.2. Comparativa de pruebas de entrada en algoritmo Cooley-Tukey

En la tabla (4.3) se muestra que al mejorar el algoritmo Cooley- Tukey con la adaptación del algoritmo radix-2 y comparado con los algoritmos radix-4 y Kiss FFT en dos versiones Cordic y Multicanal, se observa que, al aumentar el número de puntos de entrada con diferentes bits, los tiempos de ejecucion disminuyen considerablemente y aumenta la velocidad de transmisión de datos, cumpliéndose las hipótesis propuestas en este trabajo de tesis.

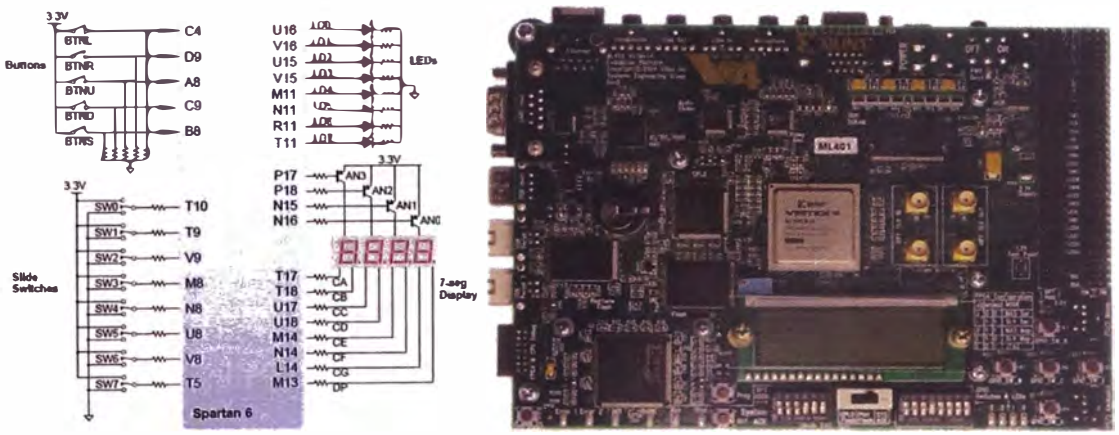


FIGURA 4.2: Resultados con FPGA
(Fuente: Referencia [21])

Entradas,Nbits,Tiempo	1024, 12, 8.3	1024,16, 8	4096, 12, 8.3	4096, 16, 8
Radix-2, *cordic*	9,2Mbit/s	8,15 Mbit/s	8,25 Mbit/s	10,42 Mbit/s
Radix-2, Mult.	9,12	8,03 Mbit/s	8,34 Mbit/s	11,08 Mbit/s
Radix-4, *cordic*	8,76	8,36 Mbit/s	8,74 Mbit/s	10,07 Mbit/s
Radix-4, Mult.	8,45	8,35 Mbit/s	9,13 Mbit/s	11,24 Mbit/s
Kiss FFT	9,06	8,54 Mbit/s	9,86 Mbit/s	12,4 Mbit/s

Tabla 4.3, comparativa del algoritmo radix-2 con radix-4 y Kiss FFT

Para caracterizar las diferentes arquitecturas a continuación se hace una descripción detallada de las partes de como se realizo la simulación y los análisis de los resultados y como una consecuencia adicional de este proceso se incluye la distorsion total armonica (THD)

4.1.1 Dispositivo Lógico Programable FPGA

Un *field-programmable gate array* (FPGA) es un dispositivo lógico que contiene un arreglo bidimensional de celdas lógicas genéricas e interruptores programables [9], como se muestra en la figura 4.3.

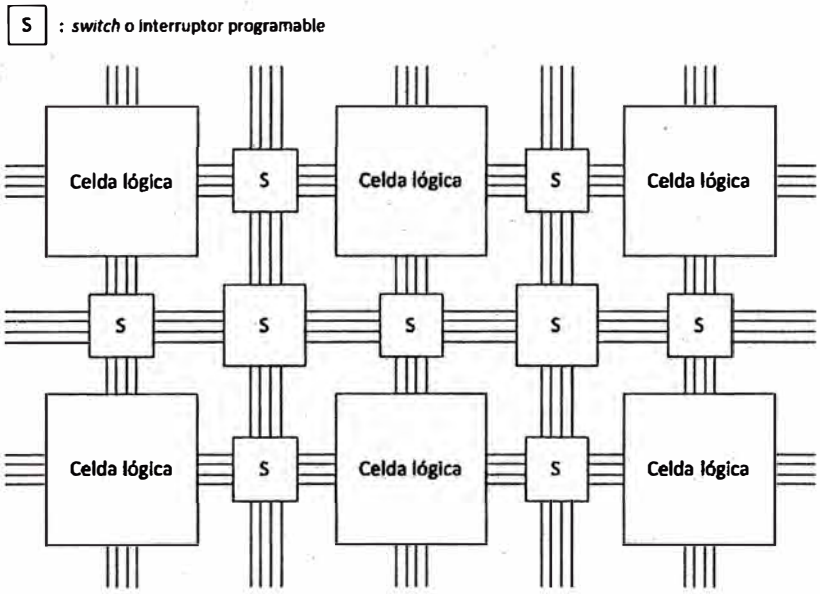


FIGURA 4.3: Estructura conceptual de un dispositivo FPGA
(Fuente: Referencia [9])

Una celda lógica puede ser configurada para realizar una determinada función, y un interruptor programable puede ser empleado para proveer interconexiones entre las celdas lógicas. Para la implementación de un diseño determinado, se especifican las funciones de cada celda lógica y se activan o desactivan las conexiones de cada interruptor programable. Dado que la configuración del dispositivo es realizada por el usuario, se dice que es "programable en el campo" o *field-programmable*.

4.1.2 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 serial, botones y LEDs.

Se ensayaron de esta manera las arquitecturas radix-2 y radix-4 iterativas con ancho de palabra de 12 bits, para 2^{10} 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 iMpact 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 serial y almacenados en una memoria auxiliar para luego ser procesados por la arquitectura y comunicados a la PC a través del puerto serial, para ser analizados mediante procesamiento en Octave.

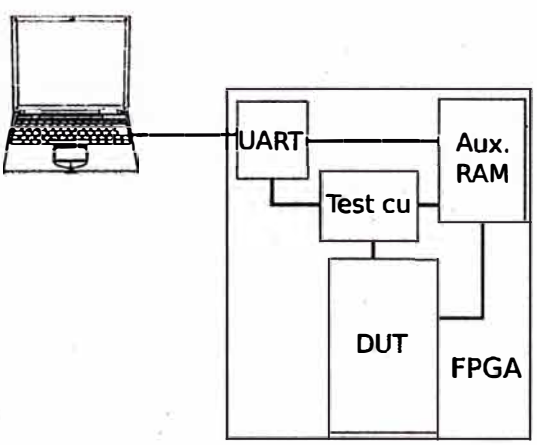


FIGURA 4.4: Test bench para la validación de las arquitecturas en hardware
(Fuente: Referencia [1])

Los vectores de prueba consisten tanto en las señales patrón utilizadas en los ensayos de la sección 4.3.1 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 la tabla 2.4. Estan en el anexo D.

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

4.2 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 investigación.

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 4.5 y 4.6 se muestran los resultados de la síntesis de las arquitecturas mencionadas para 1024 y 4096 puntos.

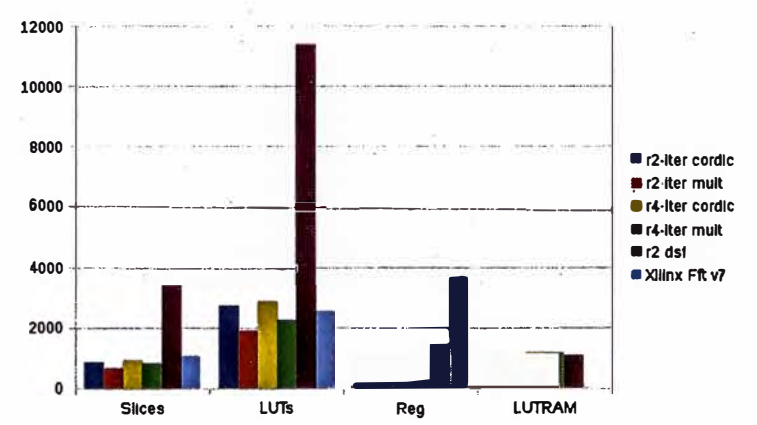


FIGURA 4.5: Comparativa de tamaño de síntesis de diferentes arquitecturas para 2^{10} puntos en una FPGA XC5VLX110
(Fuente: Referencia [2])

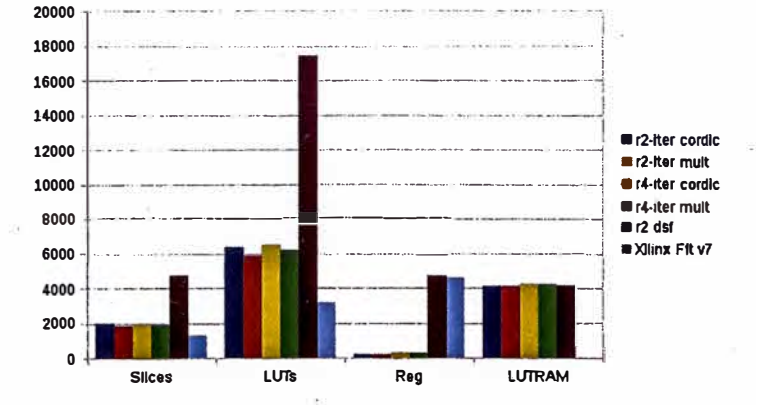


FIGURA 4.6: Comparativa de tamaño de síntesis de diferentes arquitecturas para 2^{12} puntos en una FPGA XC5VLX110
(Fuente: Referencia [2])

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 objetivo de desarrollar una arquitectura económica en ocupación de recursos de implementación, lo que constituye uno de los propósitos fundamentales del presente trabajo de investigación.

CONCLUSIONES Y RECOMENDACIONES

Conclusiones y recomendaciones

Conclusiones

1.- De acuerdo a la hipótesis H_1 se verifico que la reducción del tiempo de ejecución que se realizo con el algoritmo radix-2 en decimacion en tiempo y frecuencia indicados en el capitulo 3, logrando aumentar la velocidad de transmisión de datos en el sistema OFDM, tanto en la entrada con FFT y salida IFFT, aumentando la transmisión de data en aproximadamente 10% y la reducción del tiempo de ejecución en promedio de 3,5 us

2.- De acuerdo a la hipótesis H_2 al reducir el número de multiplicaciones por etapas y al pasar de la arquitectura del algoritmo de Cooly-Tukey a radix-2, se logra aumentar la transmisión de Data y se reduce significativamente el tiempo de ejecucion en ambas arquitecturas para el cómputo de la FFT 2,5 us en promedio

3.- Como consecuencia de H_1 y H_2 se verifica que la adaptación del algoritmo radix-2 cumple con la hipótesis H_0 de aumentar la velocidad de trasmisión de datos en un rango de tiempo menor para todas las entradas y salidas que están especificados en la figura 4.2, por ejemplo, el bloque de entrada en FFT para 16 bits y 4096 puntos de muestra en el tiempo, se genera la misma cantidad de componentes en frecuencia, ratificando el aumento en la transmisión de datos y reducción en el tiempo de ejecución.

Recomendaciones

1.- La adaptación del algoritmo 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.

2.- 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.

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

4.- 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 investigación.

GLOSARIO

ADSL	Asymetric Digital Suscriber Line
CORDIC	Coordinate Rotation Digital Computer
DAB	Digital Audio Broadcasting
DFT	Discrete Fourier Transform
DSP	Digital Signal Processor
DVB-T	Digital Video Broadcasting-Terrestrial
FFT	Fast Fourier Transform
FPGA	Field Programable Gate Array
ICI	Inter Charrier Interference
IFFT	Inverse Fast Fourier Transform
OFDM	Ortogonal Frecuency Division Multiplexing
PLC	Programable Logic Controller
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase- Shift Keying
SDF	Single-path Delay Feedback
SDR	Software Defined Radio
TDD	Test-Driven Development
TDT	Television Digital Terrestrial
THD	Total Harmonic Distortion
TIC / ICT	Information and Communication Tecnology
VHDL / HDL	Hardware Description Language
VLSI	Very Large Scale Integration
VER/SNR	Signal/Noise
BPSK	Binary Phase Shift Keying
DCM	Digital Clock Manager
MIMO	Multiple Input Multiple Ouput
GI	Guard Interval
DAC	Digital Analog Convertión
ADC	Analog to Digital Converter
LPF	Low Pass Filter
MZM	Electróptico Amplitude Modulator

BIBLIOGRAFIA

[1] **Borgerding, M.** (2005). Kiss FFT is a very small, reasonably efficient, mixed radix FFT library that can use either fixed or floating point data types.

https://sourceforge.net/projects/kissfft/?source=navbar

[2] **Cassagnes, A.** (2016). Implementacion y Analisis de Algoritmos de Calculo de FFT para sus aplicaciones en Sistemas OFDM Tesis de Maestria UBA, Buenos Aires

[3] **Cruz, W.** (2013). Desarrollo y puesta en marcha de un Sistema Embebido en FPGA para el procesamiento de datos en 2D. Tesis de Maestria UNAM, Mexico DF

[4] **Diaz, J.** (2010). Recuperacion de información en Textos hablados.Tesis de Maestria, Universidad Nacional de Trujillo.

[5] **Ferrin, C., León, J.** (2010). Sistema de Comunicatión OFDM óptico cristográfico. https://www.unab.edu.co/sites/default/files/MemoriasGrabadas/papers/capitulo8_paper_19.pdf

[6] **IEEE802.11A** Standard Perfomance in Mobile Environment

[7] **Lin, P.** (2010) OFDM Simulation in Matlab. California Polytechnic State University,

[8] **Manolakis, D.** (1997). Proakis J. Digital Signal Processing. Principles, Algorithms and Applications. USA: Prentice-Hall, Cap. The Sampling Theorem, págs. 29-33.

[9] **Meyer-Baese, U.** (2007). Digital Signal Processing with Field Programmable Gate Arrays. Berlin: Springer-Verlag, Cap. Fourier Transform, págs. 349-350.

[10] **Meyer-Baese, U.** (2007). Digital Signal Processing with Field Programmable Gate Arrays. Berlin: Springer-Verlag, Cap. Fourier Transform, págs. 363-373.

[11] **Mitacc, M.** (2013). Diseño de un Sistema de Transmision y Recepcion Basado en OFDM para Comunicaciones PLC Banda Ancha.Tesis de Maestria PUCP, Lima.

[12] **Montoya, M.** (2017). Evaluación de los distintos tipos de Modulacion para Sistemas PLC empleados en Redes Energeticas Inteligentes. Tesis de Maestria, Universidad Catolica de Santiago Guayaquil.

[13] **Oppenheim, A.** (2000). Schafer R. Tratamiento de Señales en Tiempo Discreto. España: Prentice Hall, Cap. Cómputo de la Transformada Discreta de Fourier, págs. 631-694.

[14] **Oppenheim, A.** (2000). Schafer R. Tratamiento de Señales en Tiempo Discreto. España: Prentice Hall, Cap. La Transformada Discreta de Fourier, págs. 543-572.

[15] **Pedraza, C.** (2006). Revista Hallazgos Implementacion de un Algoritmo FFT en Hardware aplicado a recuperación en OFDM - http://usta.edu.co/index php/

[16] **Prasad, R.** (2004). OFDM for Wireless Communication Systems. UK: Artech House, Cap. Orthogonal Frequency-Division Multiplexing, págs. 11-15.

[17] **Sedgewick, R.** (2017). Fast Fourier Transform. Universidad de Princeton

https://www.cs.princeton.edu/~wayne/cs423/lectures/fft-4up.pdf

[18] **Sedgewick, R., Wayne, K.** (2017). Princeton University
https://introcs.cs.princeton.edu/java/97data/FFT.java.html

[19] **Volder, J.** (1959). The Cordic Computer Technique. Ed. por IRE. Trans. Elect. Comput.

[20] **Wang, X.** (2011). OFDM and Its Application 4G. 2011
https://www.researchgate.net/publication/224327440_OFDM_and_its_applications_A_survey

[21] **Xilinx.** LogiCORE IP Fast Fourier Transform v7.1. 2011.

ANEXO A

a.1 Arquitectura Radix-4

a.1.1 Descripción General

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

En la figura A.1 se muestra un esquema del cómputo de una radix-4 para 16 puntos, similar al esquema de la radix-2 de la figura 3.1, donde se ve el cómputo aritmético utilizando cuatro puntos como entrada y obteniendo cuatro puntos como salida.

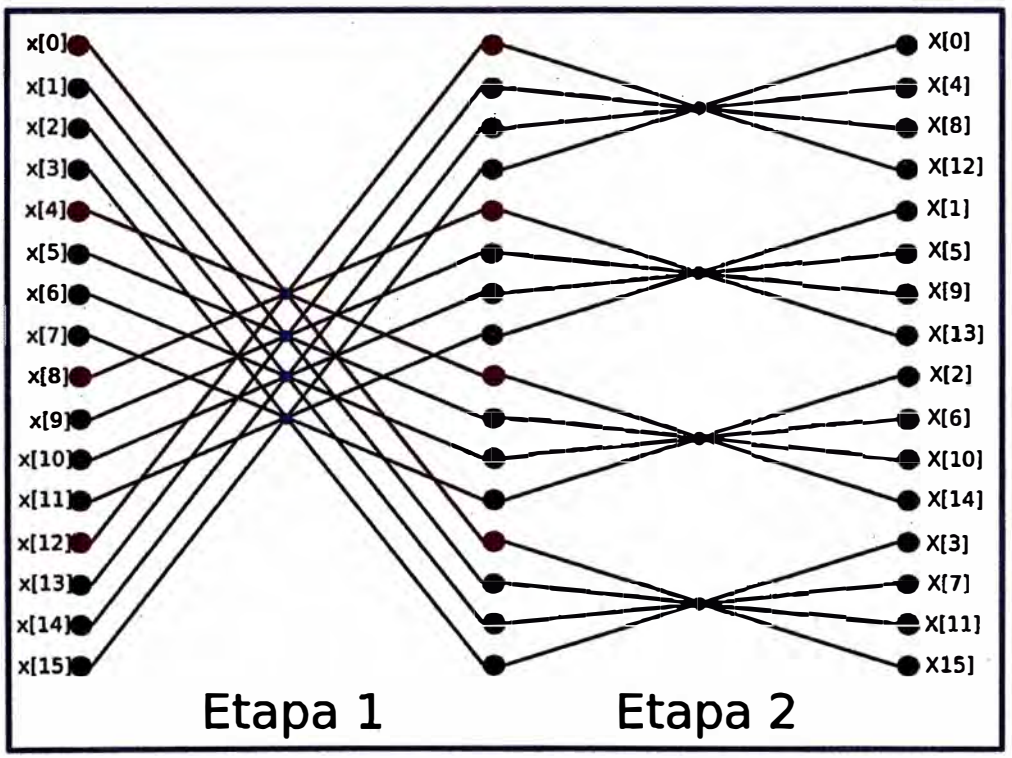


FIGURA A.1: Esquema de una FFT radix-4 de 16 puntos
(Fuente: Referencia [11])

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 A.2 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 cómputo de una FFT de la misma cantidad de puntos.

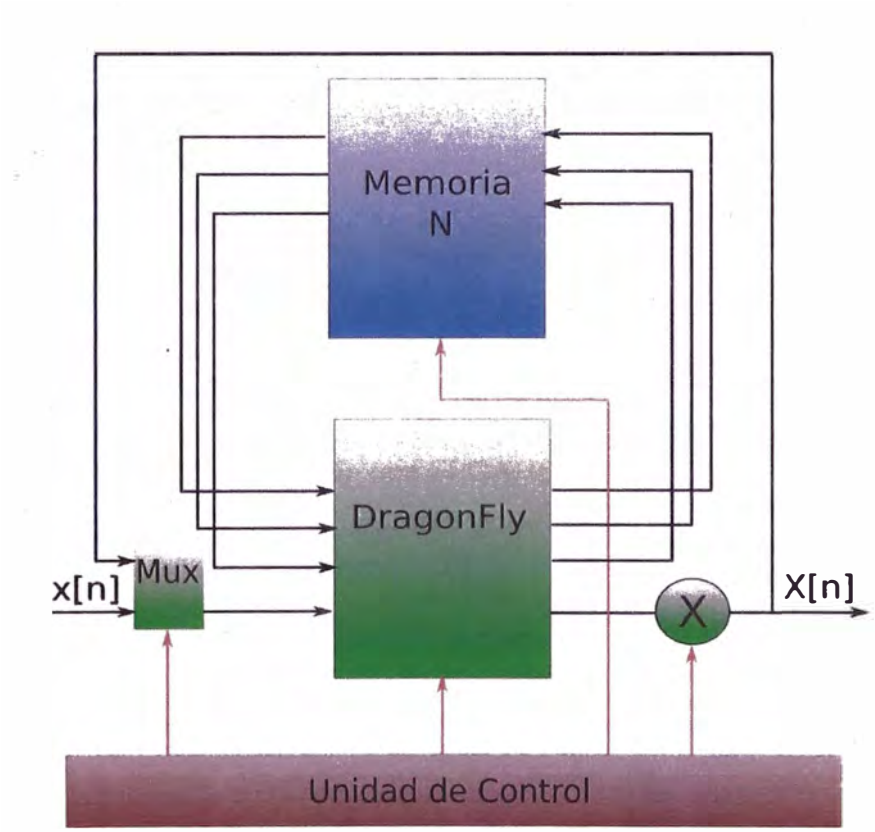


FIGURA A.2: Diagrama simplificado de la arquitectura radix-4 iterativa (Fuente: Referencia [2])

a.1.2 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 A.3 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.

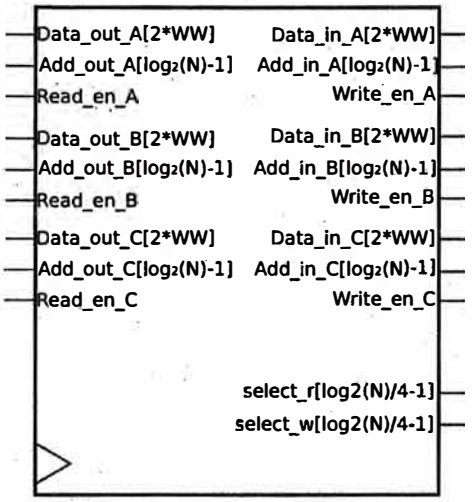


FIGURA A.3: RAM de triple entrada y triple salida
(Fuente: Referencia [2])

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 sub bloques RAM de doble puerto igual a los utilizados en la radix- 2 (subsección 3.1.2). 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 sub bloques RAM que componen el bloque triple tiene una capacidad de almacenamiento de $N/3$ puntos. La distribución de los puntos en los tres sub bloques 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 sub bloques 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 sub bloques 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 sub bloques de memoria de manera que queden delimitadas las porciones de memoria correspondientes a cada etapa, como se muestra en la figura A.4. Cada una de

estas regiones es de tamaño igual a ¼ de la longitud de la FFT de la etapa correspondiente, por ejemplo la porción de memoria correspondiente a la primer etapa será N/4 en cada sub bloque, comprendiendo en total 3N/4, para la segunda etapa cada sub bloque reservará $(\log_4(N))/4$ ya que cada FFT de la segunda etapa es de longitud $\log_4(N)$, así hasta llegar a la ultima etapa donde solo se reserva una posición de memoria de cada sub bloque, 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 utilizan 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 sub bloque 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 A.4 se muestra esquemáticamente la división de cada sub bloque 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 sub bloque RAM de la memoria de triple entrada y triple salida.

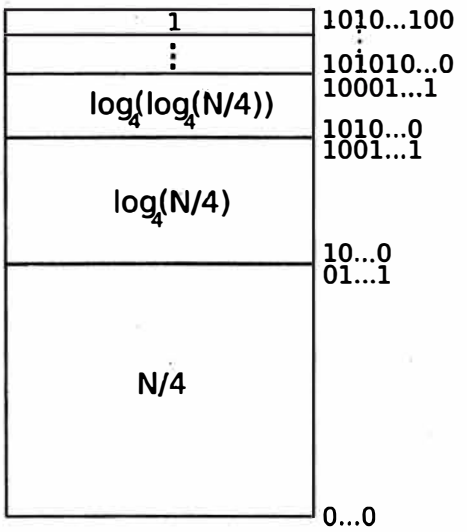


FIGURA A.4: Esquema de direccionamiento de los sub bloques RAM
(Fuente: Referencia [2])

a.1.3 Sumador/restador

La unidad aritmética debe resolver las ecuaciones (2,8) a (2.11). Para esto se opera directamente trasladando 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 A.1 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 sub bloques 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 A.5 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 multiplicador, que permite seleccionar entre el resultado de la operación aritmética o una de las tres entradas de operando 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 3.1.4.

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

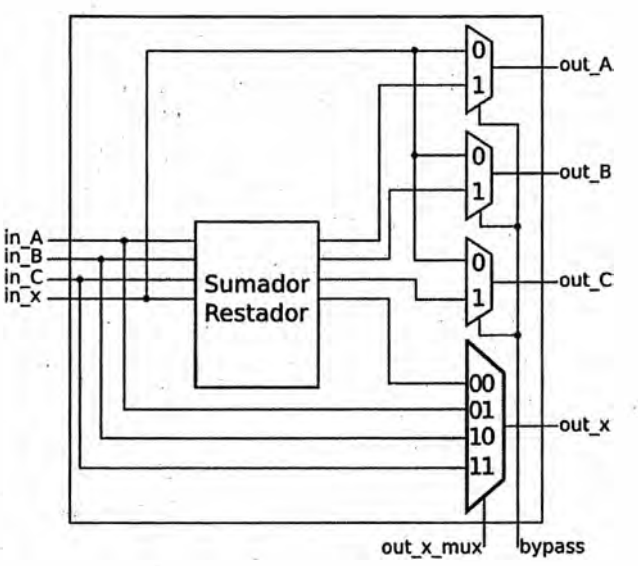


FIGURA A.5: Diagrama de la unidad aritmética incluyendo los multiplexores de bypass (Fuente: Referencia [2])

a.1.4 Datapath

En la figura A.6 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 sub bloque 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 A.7 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 sub bloque 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 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 A.8 se muestran las distintas configuraciones posibles para el *datapath* para operaciones aritméticas.

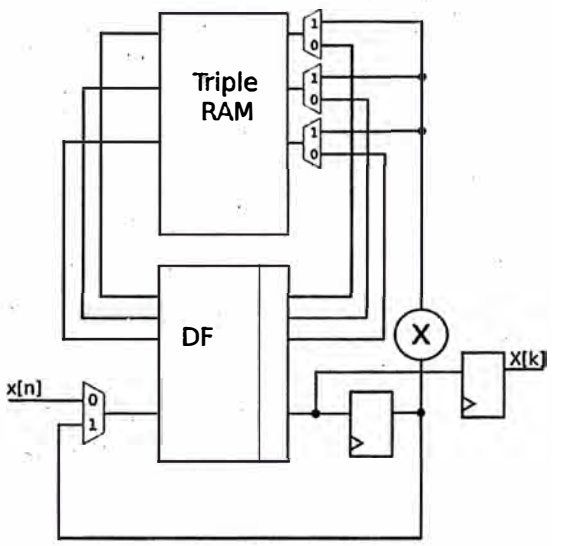


FIGURA A.6: Datapath de la arquitectura radix-4 iterativa
(Fuente: Referencia [2])

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.

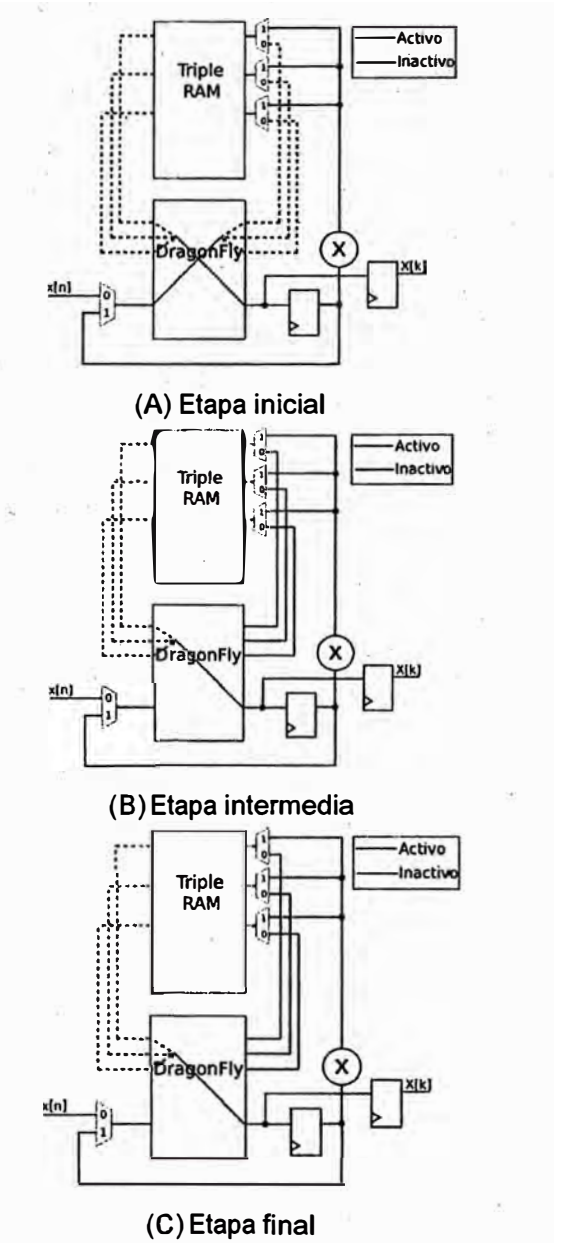


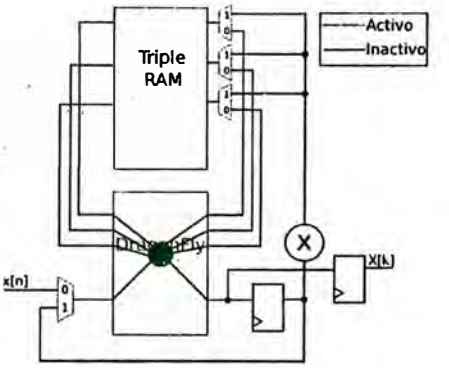
FIGURA A.7: Datapath para operaciones de transferencia en memoria

(Fuente: Referencia [2])

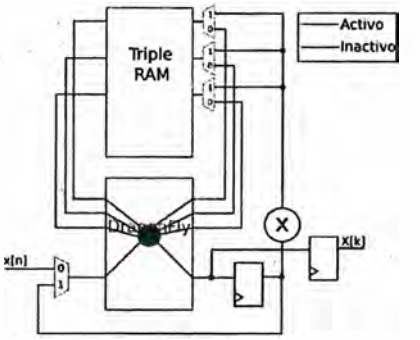
Transferencia a memoria: se almacena el dato entrante a la arquitectura en uno de los tres sub bloques de memoria.

- **Etapas intermedias**

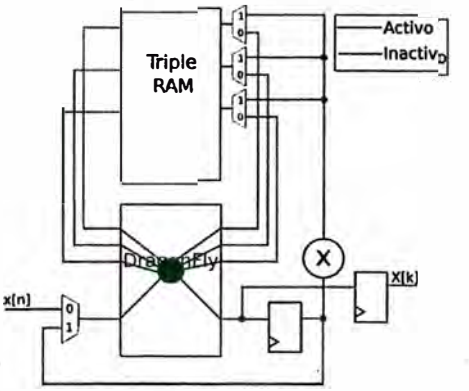
Operaciones aritméticas: tres datos almacenados en memoria y el dato de la etapa anterior almacenado en el *delay register*. Tres de los resultados son almacenados en memoria mientras que el cuarto se envía al multiplicador.



(A) Etapa inicial



(B) Etapa intermedia



(C) Etapa final

(D)

FIGURA A.8: Datapath para operaciones en butterfly

(Fuente: Referencia [2])

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*.

a.1.5 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 secundarios 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 que 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 A.1, para una radix-4 de 16 puntos, en la primera 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 sub bloques, por lo que debe diferenciarse a que sub bloque 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 que se sub bloque se debe almacenar el dato, se utilizan dos bits del contador de puntos de la siguiente manera:

- [00] ->Almacenamiento en sub bloque de memoria 1
- [01] ->Almacenamiento en sub bloque de memoria 2
- [16] ->Almacenamiento en sub bloque de memoria 3
- [17] ->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 A.9 para una arquitectura radix-4 de 256 puntos.

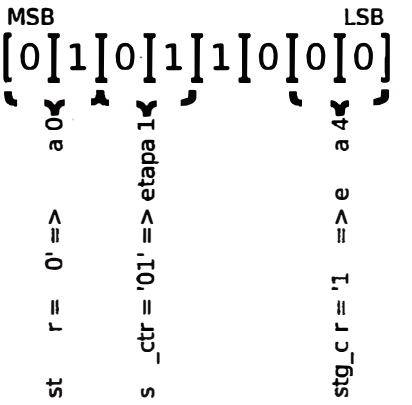


FIGURA A.9: Selección del par de bits del contador de puntos a evaluar
(Fuente: Referencia [2])

a.1.6 Máquinas de estados

En la figura A.10 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*.

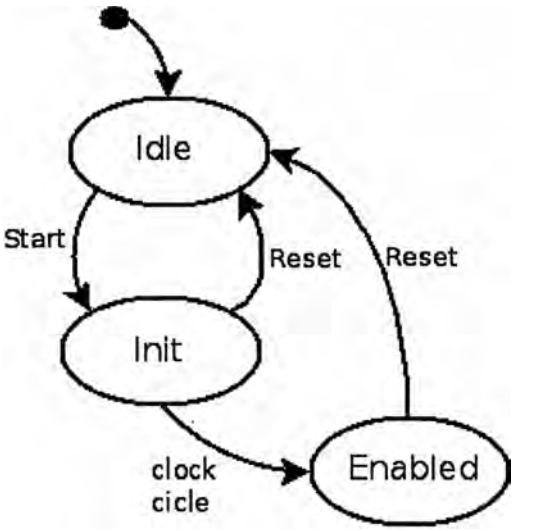


FIGURA A.10: Diagrama de estados y transiciones de la máquina de estados principal
(Fuente: Referencia [2])

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.

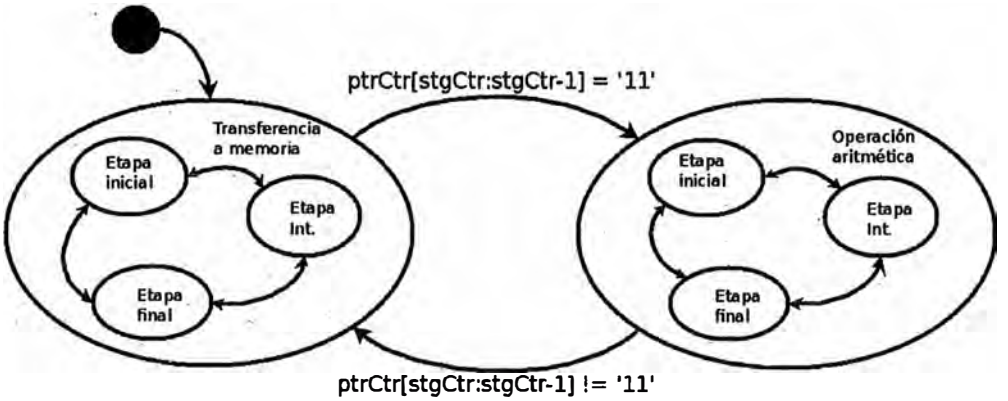


FIGURA A.11: Diagrama de estados y transiciones de la máquina de estados secundaria (Fuente: Referencia [2])

En la figura A.11 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. *ptrCtr*[*stgCtr* : *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.

a.1.7 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 sub bloque 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 sub bloque de memoria correspondiente para los casos de movimientos en memoria y habilitando los tres sub bloques 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 sub bloque particular se utiliza para una etapa determinada. Las señales de selección se construyen como una máscara para lo bits superiores de la señal de direccionamiento, para ubicar la dirección indicada por el contador de puntos en la región correspondiente.

a.1.8 Integración de la unidad de control

En la figura A.12 se muestra el diagrama del *datapath* de la figura A.6 con el agregado de las señales de la unidad de control.

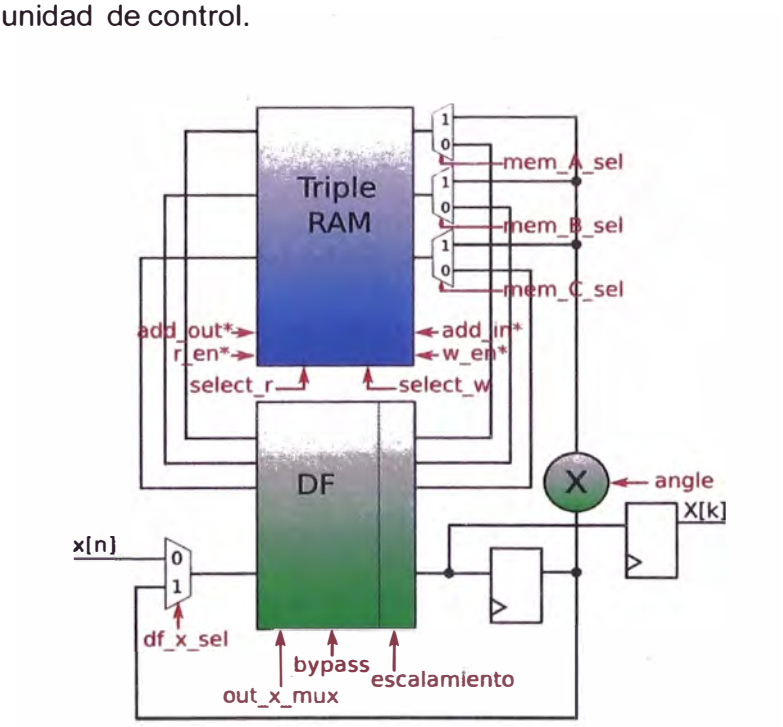


FIGURA A.12: Datapath con las señales de control
(Fuente: Referencia [2])

Las señales de control de la figura A.12 se listan a continuación, indicando entre paréntesis su tamaño si es mayor a 1:

- **add_in*** (log2(N)) address in, dirección de memoria donde se escribirá el dato.
- **w_en*** write enable, señal de habilitación de escritura en memoria
- **add_out*** (log2(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** (log2(log4(N)))) Selección de la región de memoria donde se realiza la lectura.

- **select_r** (log2(log4(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 sub bloque B de la memoria.
- **mem_C_sel** señal de control del multiplexor a la entrada al sub bloque 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 A.5).
- **out_x_mux** señal de control de las salidas a memoria de la unidad aritmética (ver esquema en figura A.5).
- **angulo (N + 1)** ángulo 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 sub bloque 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 3.1.4.

ANEXO B

b.1.1 Delta en componente '0'

Utilizando como entrada una señal compuesta por un único pulso en la primera 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.

b.1.2 Bloque Transformada Inversa Rápida de Fourier (IFFT).

La IFFT empleada es de 4096 puntos, lo que significa que recibe 4096 componentes de frecuencia, y genera la misma cantidad de muestras en el tiempo. Por tanto, el término N toma el valor de 4096 en la ecuación 2.2 presentada en el capítulo 2. De este modo, la operación realizada por la IFFT implementada ($x(n)$) está descrita por la ecuación 3.2, donde n es el número de muestra en el tiempo, k el número del componente frecuencial y $X(k)$ el valor del k-ésimo componente frecuencial.

$$x(n) = \frac{1}{4096} \sum_{k=0}^{4095} X(k) e^{\frac{jnk2\pi}{4096}} \quad , \; n = 0, \dots, 4095 \tag{B.1}$$

Para la implementación de la IFFT se emplea el núcleo de propiedad intelectual (*IP core*) Fast Fourier Transform v8.0 de Xilinx®, cuya hoja de datos se presenta en [21]. Este *IP* core no sólo efectúa el cálculo de la IFFT, sino que también realiza la conversión serial-paralela para los datos de entrada y la conversión paralela-serial para los datos de salida. A continuación, se presentan los parámetros seleccionados para su configuración:

- Tipo de implementación: *Pipelined Streaming I/O*. Esta arquitectura permite un procesamiento continuo de datos, y por tanto es la que permite la mayor tasa de datos entre los diferentes modos de implementación. Sin embargo, es la arquitectura que requiere la mayor cantidad de recursos.
- Tamaño de la transformada: 4096 puntos.

- Formato de datos: punto fijo. Tanto los datos de entrada como los de salida se expresan en formato de punto fijo.
- Tamaño de bus de datos de entrada: 16. Tanto los componentes de fase como los de cuadratura son de 16 bits.
- Tamaño de factor de fase: 16. A mayor tamaño del factor de fase, mayor inmunidad al ruido y mayor cantidad de recursos utilizados.
- Tipo de escalamiento: *unscaled* (no escalado). Permite una mayor precisión de los datos en la salida que el modo *scaled* (escalado).
- Modo de redondeo: *convergent rounding* (redondeo convergente). Permite una mayor precisión en los datos que el modo *truncation* (truncado).
- Ordenamiento de las salidas: orden natural. Esto implica que los datos en la salida son mostrados en orden consecutivo desde el primero hasta el último.
- Inserción de prefijo cíclico: sí.
- Cantidad de etapas que usan *Block RAM*: 7. Se utiliza dicha cantidad de bloques de memoria dedicados.

En las siguientes líneas, se presentan las entradas y salidas empleadas del núcleo de propiedad intelectual FFT v8.0 (no se muestran las señales que no han sido utilizadas), así como los valores o señales que se les asignan.

- *s_axis_config_tdata*: especifica el tamaño del prefijo cíclico y el tipo de transformada (FFT o IFFT). Se le asigna el valor decimal de 1252, el cual indica que el prefijo cíclico es de dicha longitud y que se selecciona el modo IFFT.
- *s_axis_config_tvalid*: indica que la señal *s_axis_config_tdata* es válida. Se le asigna el valor constante '1'.
- *s_axis_data_tdata*: señal de datos de entrada. Se le asigna la señal mostrada en la figura A.5.
- *s_axis_data_tvalid*: señal de entrada que indica que la trama de datos ingresada es válida. Se le asigna la señal *valid_data_in*, como se observa en la figura A.5.
- *s_axis_data_tlast*: señal de entrada que se activa externamente cuando se envía la última muestra de una trama. Esta señal no se emplea, y por tanto se le asigna el valor constante '0'.
- *m_axis_data_tready*: señal de entrada que indica que el circuito conectado en la salida (circuito esclavo) está listo para recibir datos. Se le asigna el valor constante
- '1'.

- *aclk*: señal de entrada de reloj. Se le asigna la señal *clock*, la cual es la señal de reloj del sistema.
- *m_axis_data_tdata*: señal de datos de salida. Se asigna a la señal mostrada en la figura B.1.
- *m_axis_data_tvalid*: señal que indica que la trama de datos de salida es válida. Se asigna a la señal *valid_data_out*, como se observa en la figura B.1.

En la figura 4.2 se muestra un diagrama esquemático del bloque IFFT implementado. Se observa que las señales de entrada *I_comp* y *Q_comp* ingresan inicialmente a bloques denominados *Shift*, para luego ser concatenadas a una sola señal que es ingresada a la entrada *s_axis_data_tdata* del *IP Core* FFT v8.0. Estos bloques *Shift* realizan un desplazamiento de 10 bits hacia la izquierda de sus valores de entrada, a fin de que los últimos bits que contienen información se desplacen a posiciones más significativas. Además, se aprecia que la señal de salida *m_axis_data_tdata* del *IP Core* es inicialmente dividida en dos señales, las cuales representan las componentes real e imaginaria de la muestra de salida. Estas señales son procesadas por bloques denominados *Division*, los cuales dividen sus valores de entrada entre 4096 (desplazamiento de 12 bits a la izquierda) y toman los 16 bits menos significativos del resultado. La finalidad de este bloque es realizar la división de las muestras entre el número 4096, como se muestra en la ecuación 4.1, puesto que el *IP Core* no realiza la división internamente. Finalmente, las salidas de los bloques *Division* son concatenadas para formar la señal *data_out* de salida.

Las entradas y salidas generales del circuito de la figura B.1 son:

- *Clock*: señal de reloj, común a todo el sistema.
- *I_comp*: señal de datos de entrada de 16 bits, correspondiente a las componentes en fase de los símbolos QPSK.
- *Q_comp*: señal de datos de entrada de 16 bits, correspondiente a las componentes en cuadratura de los símbolos QPSK.
- *Valid_data_in*: señal que indica cuándo se recibe una trama válida de datos.
- *Data_out*: señal de datos de salida de 32 bits, correspondiente a las muestras temporales que genera la IFFT.
- *Valid_data_out*: señal que indica cuándo los datos en la salida son válidos.

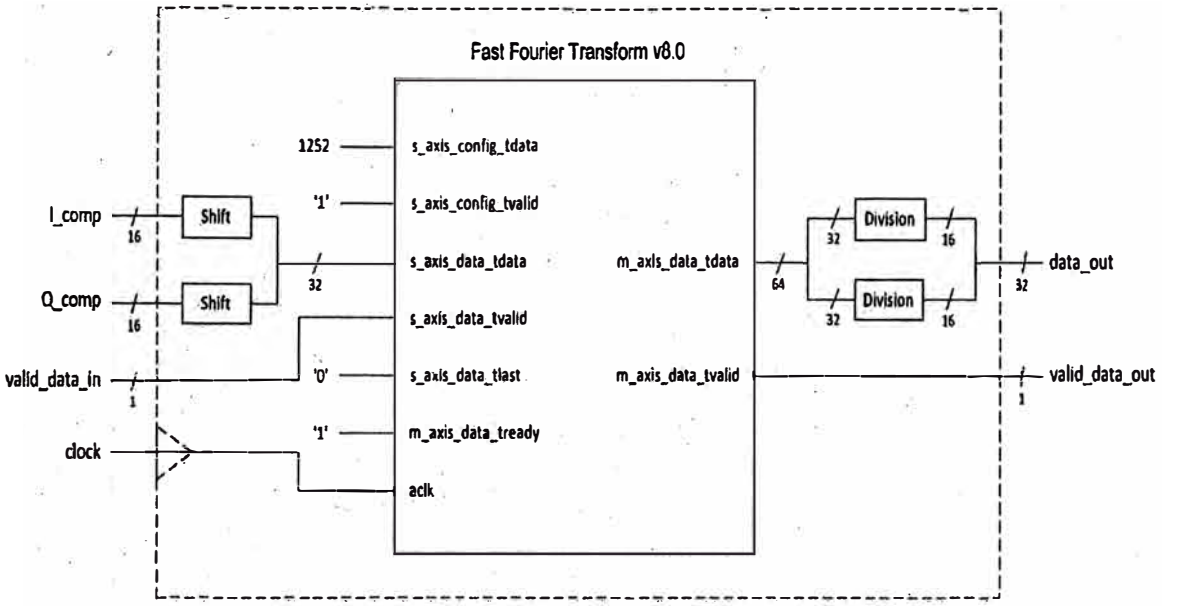


Figura B.1: Diagrama esquemático del bloque IFFT

(Fuente: Referencia [11])

Es importante mencionar que el bloque IFFT no emplea los ángulos de fase que propone el estándar IEEE 1901 para cada una de las subportadoras durante la transmisión de un símbolo OFDM, debido a que el *IP Core* empleado no permite la introducción de los mismos. Por tanto, todas las subportadoras se consideran en fase, es decir, presentan un ángulo de fase de 0°.

b.1.3 Bloque Transformada Rápida de Fourier (FFT)

La FFT empleada es de 4096 puntos, lo que significa que recibe 4096 muestras en el tiempo, y genera la misma cantidad de componentes de frecuencia. Por tanto, el término *N* toma el valor de 4096 en la ecuación 2.1 presentada en el capítulo 2. De esta manera, la operación realizada por la FFT implementada ($X(k)$) está descrita por la ecuación 3.3, donde k es el número del componente frecuencial, n el número de muestra en el tiempo y $x(n)$ el valor de la n-ésima muestra en el tiempo.

$$X(k) = \sum_{n=0}^{4095} x(n) e^{-\frac{jnk2\pi}{4096}} \quad , \; k = 0, \ldots, 4095 \tag{B.2}$$

Para la implementación de la FFT se emplea el núcleo de propiedad intelectual (*IP core*) Fast Fourier Transform v8.0 de Xilinx®, cuya hoja de datos se presenta en [21]. Este *IP core* no sólo efectúa el cálculo de la FFT, sino que también realiza la conversión serial-paralela para los datos de entrada y la conversión paralela-serial para los datos de salida.

Los parámetros seleccionados para la configuración del *IP core* son los mismos que los empleados para el bloque IFFT en el transmisor, con la excepción de que no se habilita la inserción de prefijo cíclico.

A continuación, se presentan las entradas y salidas empleadas del núcleo de propiedad intelectual FFT v8.0 (no se muestran las señales que no han sido utilizadas), así como los valores o señales que se les asignan.

- *s_axis_config_tdata*: especifica el tipo de transformada (FFT o IFFT). Se le asigna el valor decimal de 1, el cual indica que se selecciona el modo FFT.
- *s_axis_config_tvalid*: indica que la señal *s_axis_config_tdata* es válida. Se le asigna el valor constante '1'.
- *s_axis_data_tdata*: señal de datos de entrada. Se le asigna la señal *data_in,* como se muestra en la figura B.1
- *s_axis_data_tvalid*: señal de entrada que indica que la trama de datos ingresada es válida. Se le asigna la señal *valid_data_in*, como se observa en la figura B.1.
- *s_axis_data_tlast*: señal de entrada que se activa externamente cuando se envía la última muestra de una trama. Esta señal no se emplea, y por tanto se le asigna el valor constante '0'.
- *m_axis_data_tready*: señal de entrada que indica que el circuito conectado en la salida (circuito esclavo) está listo para recibir datos. Se le asigna el valor constante
- '1'.
- *aclk*: señal de entrada de reloj. Se le asigna la señal *clock*, la cual es la señal de reloj del sistema.
- *m_axis_data_tdata*: señal de datos de salida. Se asigna a la señal mostrada en la figura B.1.
- *m_axis_data_tvalid*: señal que indica que la trama de datos de salida es válida. Se asigna a la señal *valid_data_out*, como se observa en la figura B.1.

En la figura B.2 se muestra un diagrama esquemático del bloque FFT implementado. Se observa que la señal de salida *m_axis_data_tdata* del *IP Core* es inicialmente dividida en dos señales, las cuales representan las componentes en fase (I) y en cuadratura (Q) del símbolo QPSK generado. Estas señales son procesadas con el fin de reducir su longitud de 32 bits a 16 bits, de modo inverso al procedimiento que se realizó en el bloque IFFT del transmisor para la adaptación de las señales *I_comp* y *Q_comp* de 16 a 32 bits. Para ello, se emplean inicialmente los bloques denominados *Redondeo*, los cuales toman los 16 bits menos significativos y redondean la palabra resultante a sus 6 bits más

significativos, de modo que los bits restantes son ceros. Luego, los bloques *Shift* realizan un desplazamiento de 10 bits hacia la derecha de los valores de cada señal, de modo que finalmente genera las salidas *I_comp* y *Q_comp* de 16 bits.

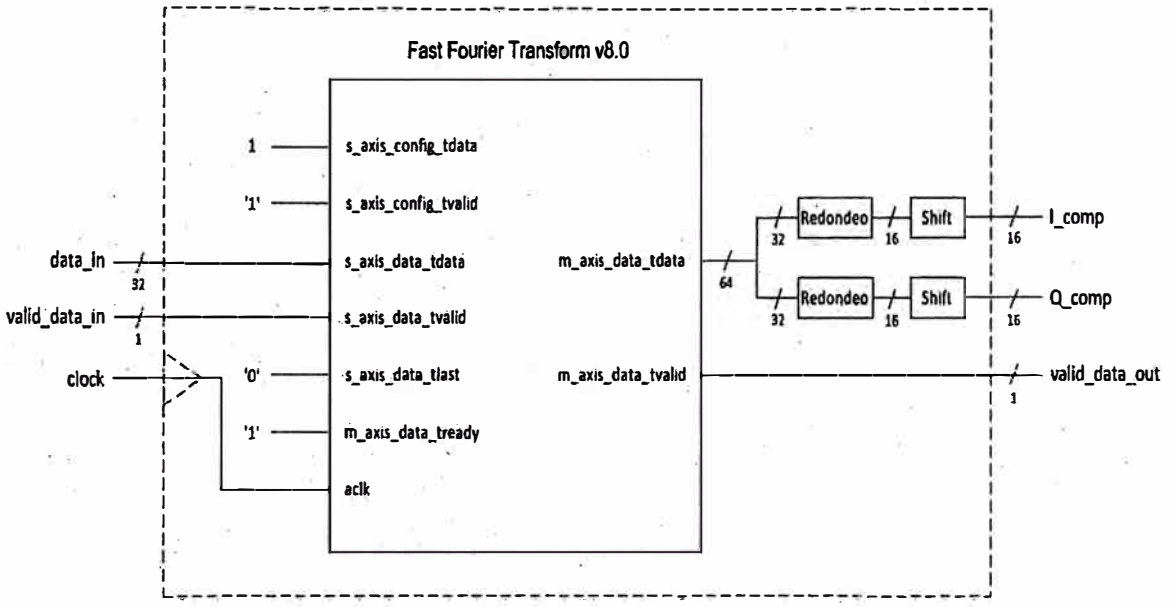
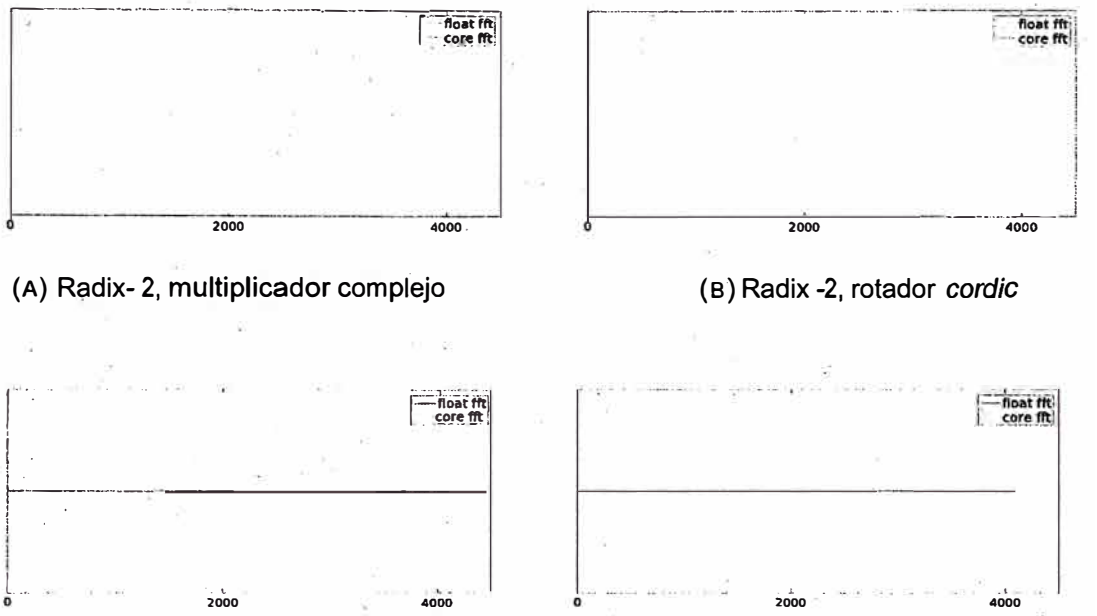


Figura B.2: Diagrama esquemático del bloque FFT
(Fuente: Referencia [11])

- Las entradas y salidas generales del circuito mostrado son:
- *Clock*: señal de reloj, común a todo el sistema.
- *Data_in*: señal de datos de entrada de 32 bits, correspondiente a las muestras temporales del símbolo OFDM recibido.
- *Valid_data_in*: señal que indica cuándo se recibe una trama válida de datos.
- *I_comp*: señal de datos de salida de 16 bits, correspondiente a las componentes en fase de los símbolos QPSK generados.
- *Q_comp*: señal de datos de salida de 16 bits, correspondiente a las componentes en cuadratura de los símbolos QPSK generados.
- *Valid_data_out*: señal que indica cuándo los datos en la salida son válidos.

En la figura B.3 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.



(A) Radix- 2, multiplicador complejo

(B) Radix -2, rotador *cordic*

(C) Radix-4, multiplicador complejo

(D) Radix- 4, rotador *cordic*

FIGURA B.3: Respuestas a una delta en la componente 0 para las arquitecturas radix-2 y radix-4

(Fuente: Referencia [2])

b.1.4 Delta

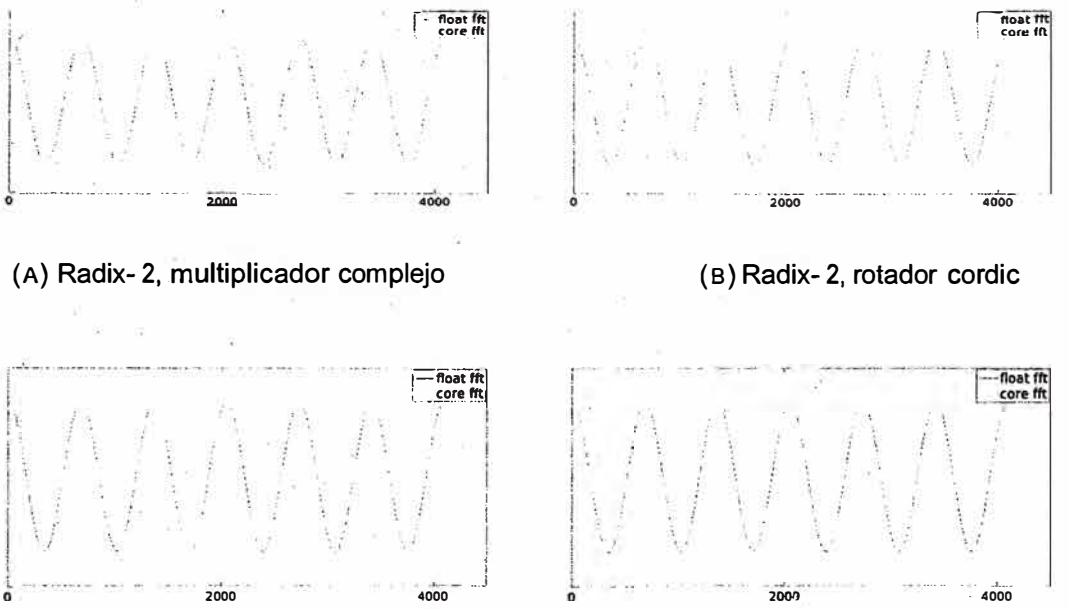
La transformada de Fourier de una delta debe dar un seno o un coseno cuya frecuencia está directamente relacionada con la posición del pulso.

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

En la figura B.3 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.

b.1.5 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.



(A) Radix- 2, multiplicador complejo

(B) Radix- 2, rotador cordic

(C) Radix- 4, multiplicador complejo

(D) Radix -4, rotador cordic

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

(Fuente: Referencia [2])

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:

$$\|\vec{x}\|_p = \sqrt[p]{x[1]^p + x[2]^p + \ldots + x[N]^p} \tag{B.3}$$

Se toma el error relativo en base a dos normas calculadas según (4.1), dando las métricas E_∞ y E_2 como sigue:

$$E_\infty = MAX\left(\frac{X_o[n] - X_{dut}[n]}{X_o[n]}\right) \tag{B.4}$$

$$E_2 = \left\| \frac{X_o[n] - X_{dut}[n]}{X_o[n]} \right\|_2 \tag{B.5}$$

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_∞ 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 B.5 se presenta un diagrama de flujo del *script* de simulación para la estimación del error.

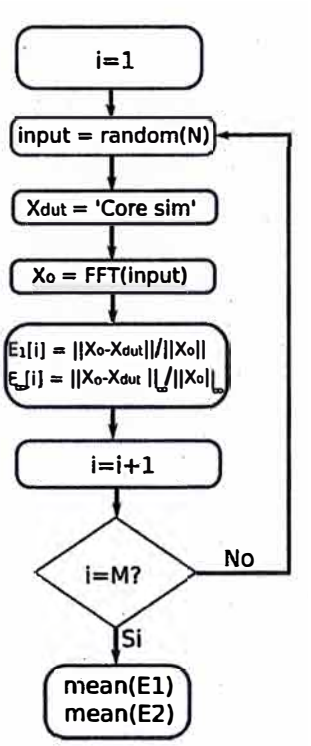


FIGURA B.5: Diagrama de flujo de la simulación para la estimación del error (Fuente: Referencia [2])

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 [21], con precisión de punto fijo de 16 bits.

En la tabla 4.3 se muestran los resultados de las mediciones de error utilizando la métrica E_∞, mientras que en la tabla 4.4 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 B.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

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

TABLA B.2: Métrica $E2$ 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 $\arctan(2^{-i})$.

También 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 $1/12^{12-1} = 4{,}88 * 10^{-4}$ Para 16 bits es $1/2^{16-1} = 3{,}05 * 10^{-5}$

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 investigación 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.

b.1.6 Lenguaje de Descripción de Hardware VHDL

Los lenguajes de descripción de hardware (*hardware description languages* o HDLs) son empleados para modelar o describir sistemas digitales en un nivel de abstracción deseado, ya sea desde el punto de vista estructural o comportamental [9]. A diferencia de los lenguajes de programación, donde las instrucciones se ejecutan en forma secuencial, los HDLs cuentan con sentencias concurrentes, es decir, que se ejecutan en simultáneo.

Los HDLs más empleados a nivel mundial son VHDL y Verilog, los cuales se han establecido como estándares industriales. Ambos son utilizados principalmente para implementar circuitos digitales en dispositivos lógicos programables como FPGAs o CPLDs, o en circuitos integrados de aplicación específica (ASICs).

b.1.7 Distorsión Total Armónica

Una vez verificado el funcionamiento a nivel general corroborando que transforma señales conocidas en sus transformadas y transformadas inversas 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 3.5, para la visualización de las señales resultantes.

b.2 Simulación y analisis 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 procedieron a la integración de las arquitecturas.

b.3 Simulación y analisis de las arquitecturas completas

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.

b.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.

b.3.2 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

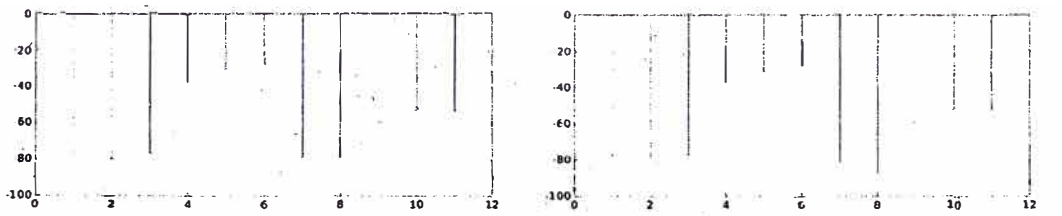
Para verificar la presencia de efectos significativos de intermodulación se realizaron algunas mediciones donde se utilizó como señal de entrada un tono puro primero, y luego el mismo tono junto con el tono siguiente. Comparando las salidas de ambos procesamientos se puede ver si el efecto de la intermodulación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

En la figura B.7 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.

b.3.3 Efectos de redondear o truncar en una etapa

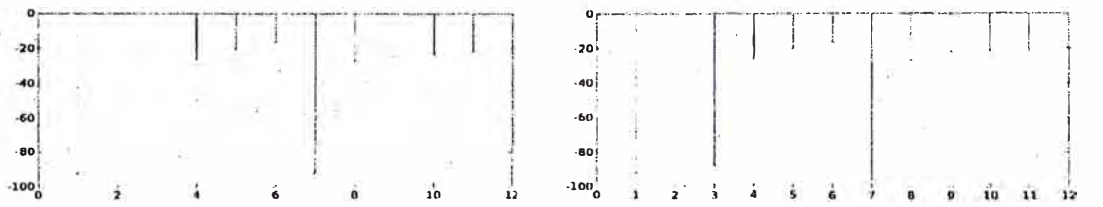
Se realizaron mediciones aplicando escalamientos, redondeo o truncamiento, aplicando primero una señal aleatoria y se comparó la salida con la respuesta a la misma entrada de la arquitectura sin escalar. En la figura B.7 se muestra una ampliación de una zona de la salida de ambas arquitecturas, con y 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.

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 B.7 se observa la THD como resultado del escalamiento en cada etapa. En la figura B.6 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.



(A) Radix- 2, redondeo (B) Radix- 2, truncamiento

FIGURA B.6: THD en función de la etapa en que se realiza es escalamiento (Fuente: Referencia [2])



(A) Radix- 2, redondeo (B) Radix- 2, truncamiento

FIGURA B.7: THD en función de la etapa en que se realiza es escalamiento para una señal que provoca overflow

(Fuente: Referencia [2])

En la figura B.7 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.

ANEXO C

c.1.1 Memoria

Debido al tipo de acceso a los datos se decide adaptar 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 C.1.

En la figura C.1 se muestran los buses de entrada y salida indicando su tamaño entre corchetes. El parámetro *WW* corresponde al parámetro global *WORD_WIDTH*

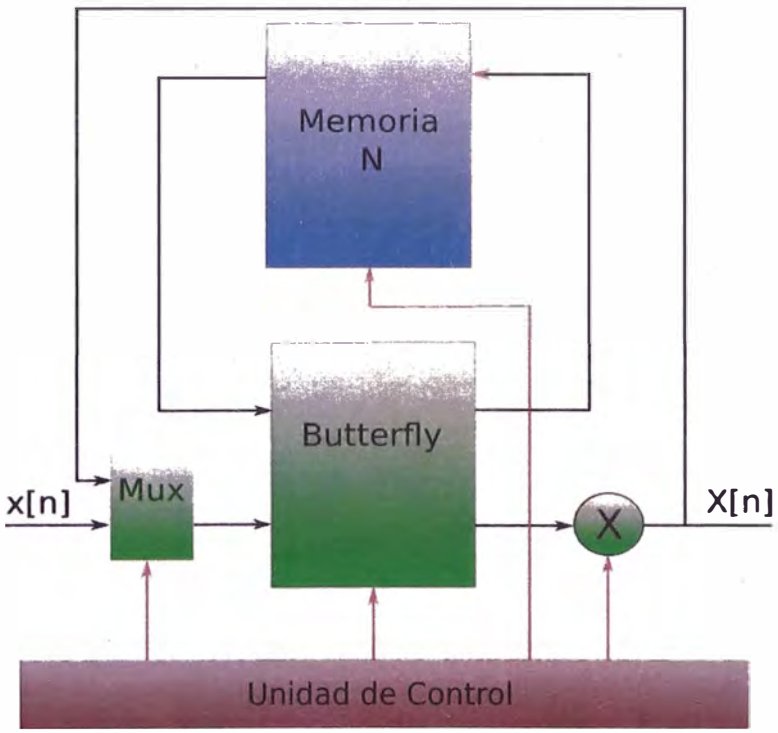


FIGURA C.1: Diagrama simplificado de la arquitectura radix-2 iterativa
(Fuente: Referencia [2])

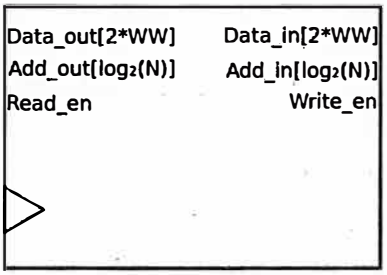


FIGURA C.2: Dual Port RAM
(Fuente: Referencia [2])

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.

c.1.2 Butterfly

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

$$c = a + b$$
$$d = a - b \qquad \text{(C.1)}$$

Siendo todas las variables complejas. En la figura 3.9 se muestra el esquema del bloque *butterfly*, compuesto por un sumador y un restador complejos.

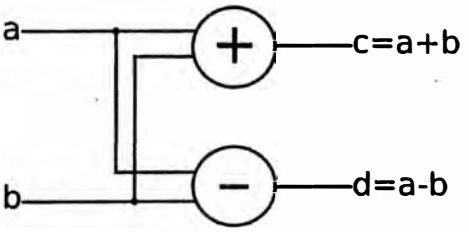


FIGURA C.3: Esquema del bloque butterfly
(Fuente: Referencia [2])

c.1.3 Datapath

En la figura 3.10 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 adaptado 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.

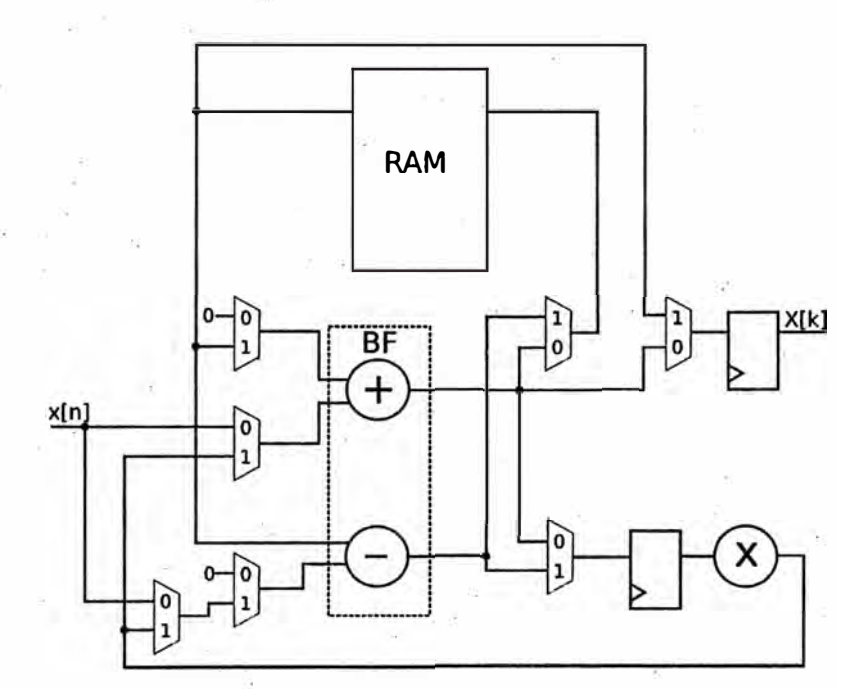


FIGURA C.4: Esquema del datapath de la arquitectura radix-2
(Fuente: Referencia [2])

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 C.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 C.5 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*.

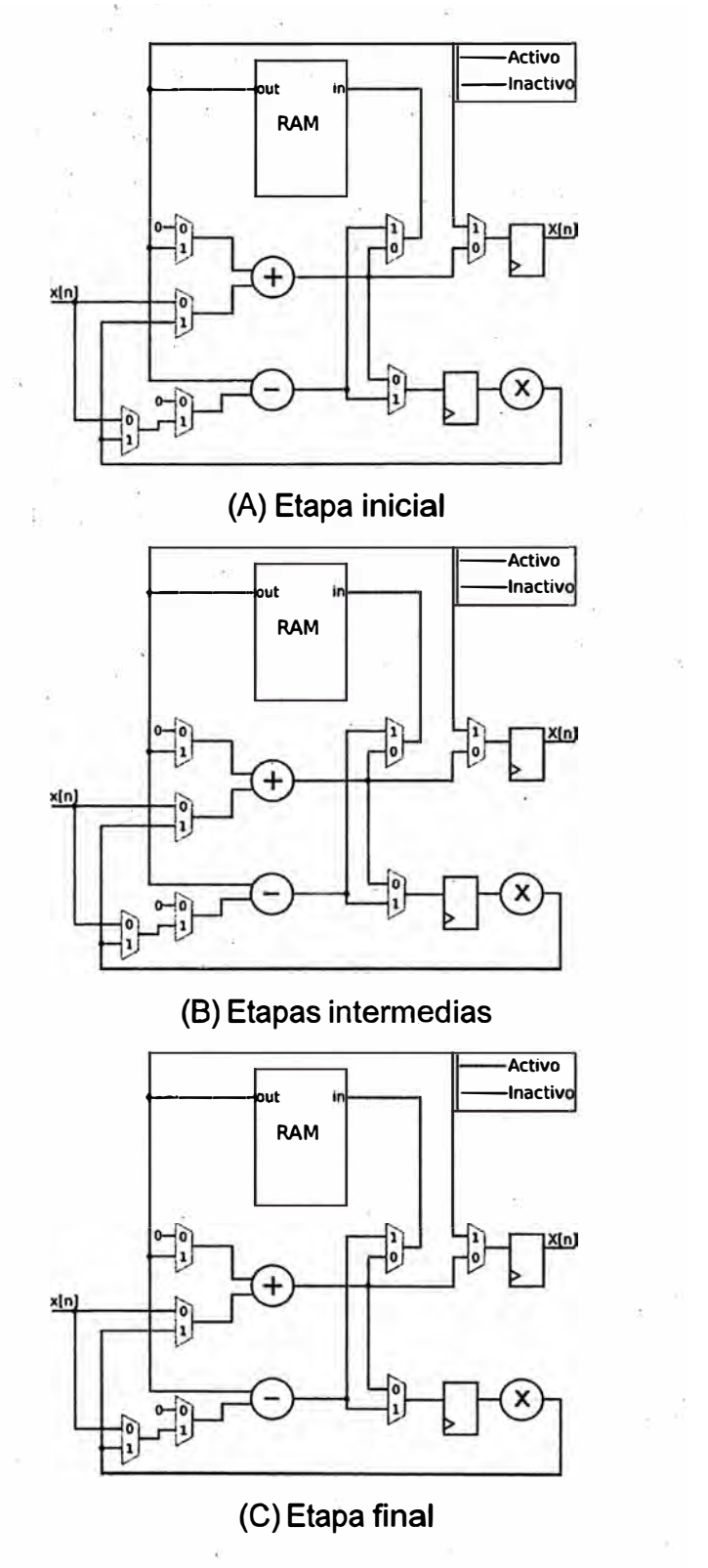
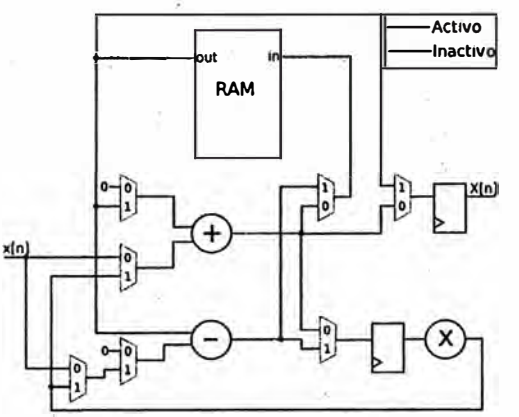


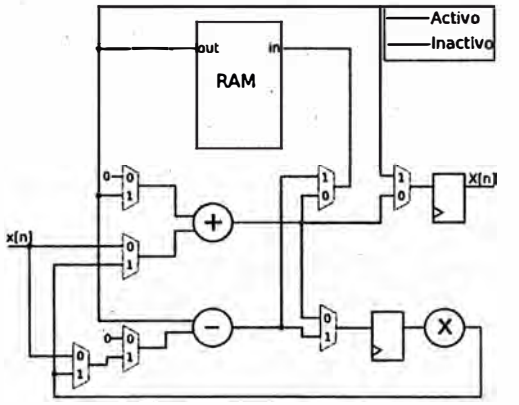
FIGURA C.5: Datapath para operaciones de transferencia en memoria

En la figura C.6 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*

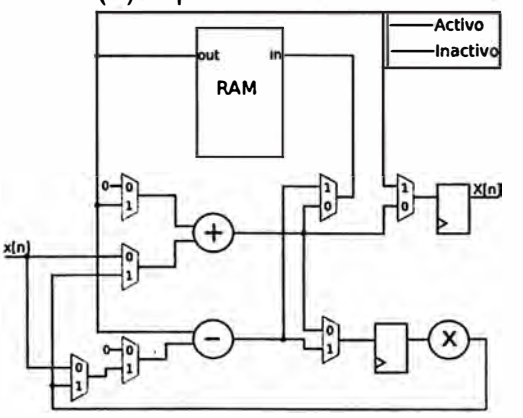
y el otro operando provendrá de la memoria. La salida del restador del butterfly se envía directamente a memoria y la salida del sumador puede ir al *delay register* para ser utilizado en la etapa siguiente o puede ser direccionado a la salida de la arquitectura dependiendo de si la etapa es intermedia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



(A) Etapa inicial



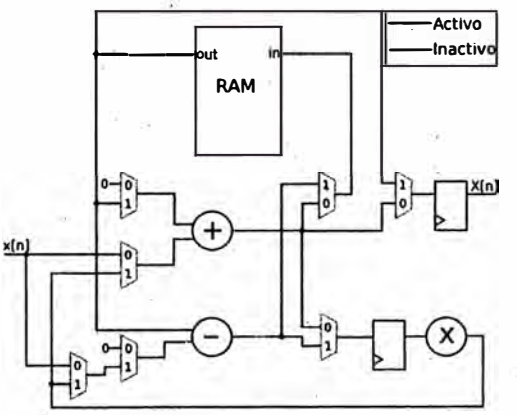
(B) Etapas intermedias



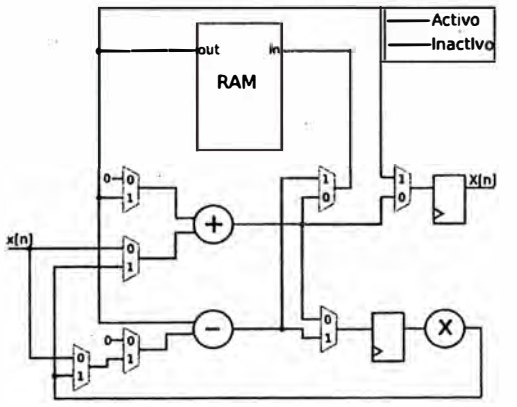
(C) Etapa final

FIGURA C.6: Datapath para operaciones en butterfly
(Fuente: Referencia [2])

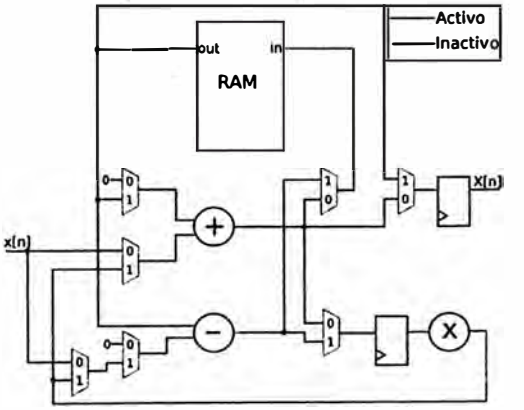
y el otro operando provendrá de la memoria. La salida del restador del butterfly se envía directamente a memoria y la salida del sumador puede ir al *delay register* para ser utilizado en la etapa siguiente o puede ser direccionado a la salida de la arquitectura dependiendo de si la etapa es intermedia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



(A) Etapa inicial



(B) Etapas intermedias



(C) Etapa final

FIGURA C.6: Datapath para operaciones en butterfly
(Fuente: Referencia [2])

c.1.4 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 3.1.4 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 C.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 trasferencias 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 alternadas 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

$$log_2\left(\frac{N}{i+1}\right) \tag{C.2}$$

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 3.8.

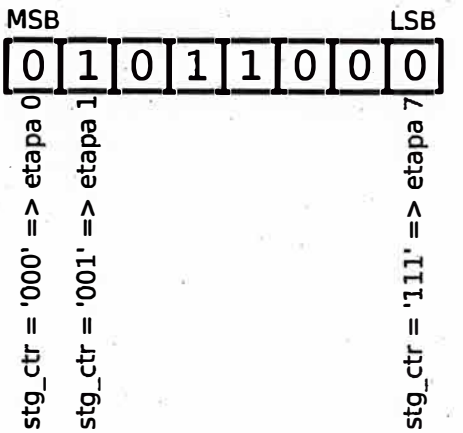


FIGURA C.7: Selección del bit del contador de puntos a evaluar
(Fuente: Referencia [2])

c.1.5 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 C.8 se muestran los estados y transiciones de la máquina de estados principal.

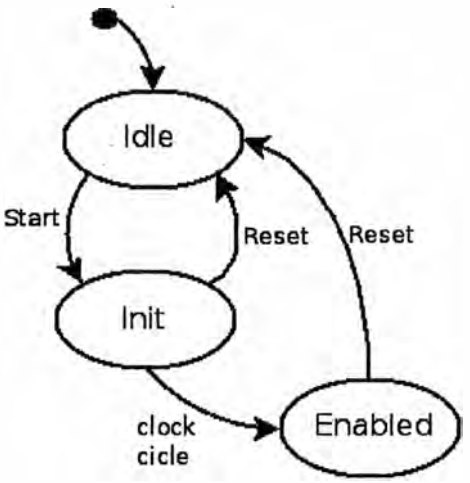


FIGURA C.8: Estados de la máquina de estados principal
(Fuente: Referencia [2])

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

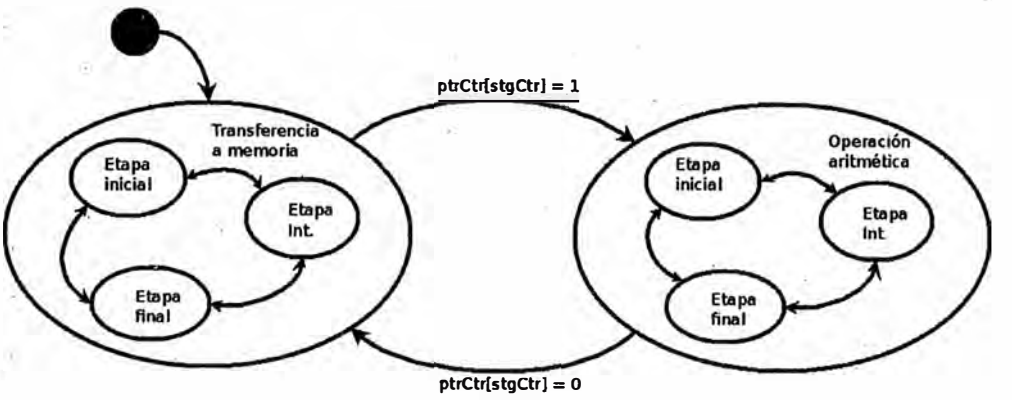


FIGURA C.9: Máquina de estados operativa para modo *enabled*
(Fuente: Referencia [2])

En la figura C.9 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 C.7.

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 cómputo actual.

c.1.6 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.

c.1.7 Control del datapath

El control del *datapath* se realiza mediante los multiplexores que se ven en la figura C.5 a través de señales digitales. En cada ciclo de *clock*, de acuerdo a la máquina de estados que decide en que etapa se encuentra el algoritmo y dependiendo de la operación a realizar se configuran los multiplexores para conformar el datapath según las figuras C.6 y C.7.

c.1.8 Integración de la unidad de control

En la figura C.10 se muestra el *datapath* de la figura C.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 3.1.4.

En la figura C.10 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** (log2(N)) address in, dirección de memoria donde se escribirá el dato
- **w_en** write enable, señal de habilitación de escritura en memoria
- **add_out** (log2(N)) address out, dirección de memoria desde la que leerá el dato

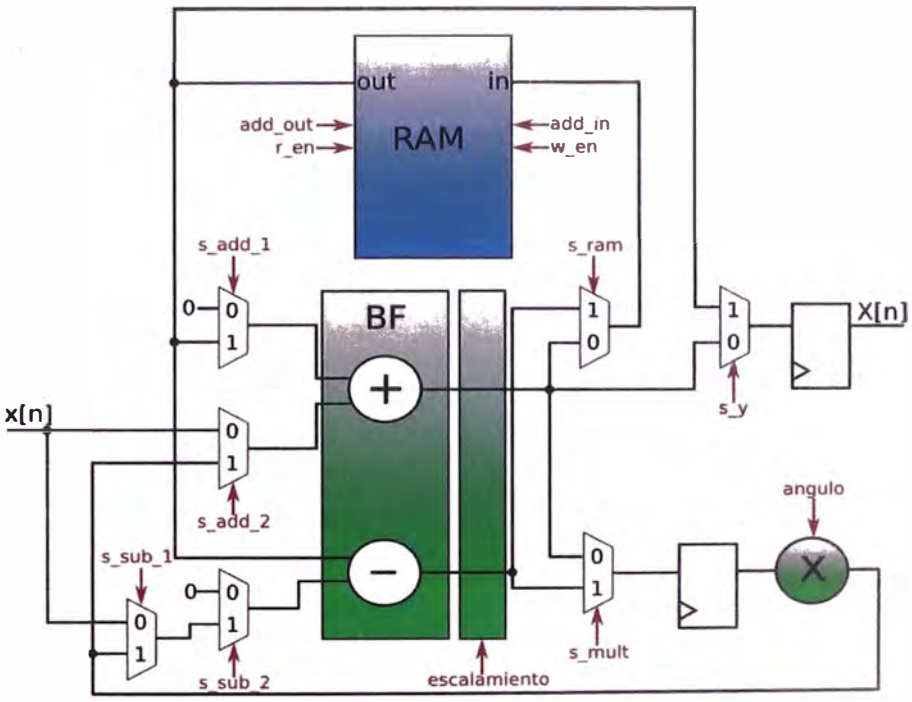


FIGURA C.10: Datapath con las señales de control

(Fuente: Referencia [2])

- **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.

c.2 Módulos compartidos por las dos arquitecturas

c.2.1 *Cordic* desenrollado

Como se explicó en la subsección 3.1.4, 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 3.1.4.

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 *pipeline* colocando registros entre las distintas etapas del cómputo *cordic*.

La forma de implementar el algoritmo es a través de sub módulos consecutivos que realizan micro rotaciones hasta completar la rotación completa.

En la figura C.11 se muestra el bloque de cómputo cordic con sus puertos de entradas y salidas.

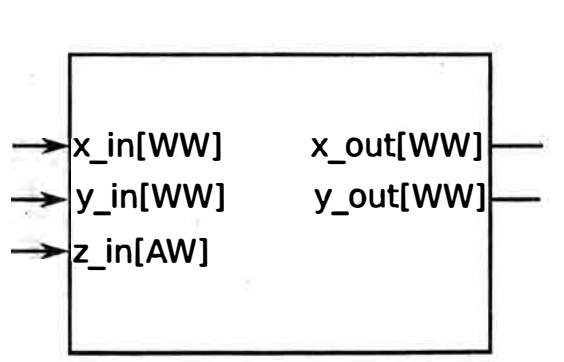


FIGURA C.11: Bloque de módulo de cómputo *cordic*
(Fuente: Referencia [14])

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 3.1.4, 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° 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°, 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(α) para cada micro rotación, por lo que a mayor número de iteraciones posibles mayor tamaño de la tabla de arco tangentes.

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

En la figura C.12 se observa el diagrama en bloques interno del módulo de cómputo cordic. El bloque *Preprocessor* realiza el análisis del ángulo y las rotaciones iniciales de 90° en caso de ser necesarias. El bloque *Unrolled cordic* realiza la rotación mediante el algoritmo cordic, que se muestra detallado en la figura C.12, y los bloques *Post mult.* realizan el escalamiento final para compensar la ganancia propia del algoritmo cordic.

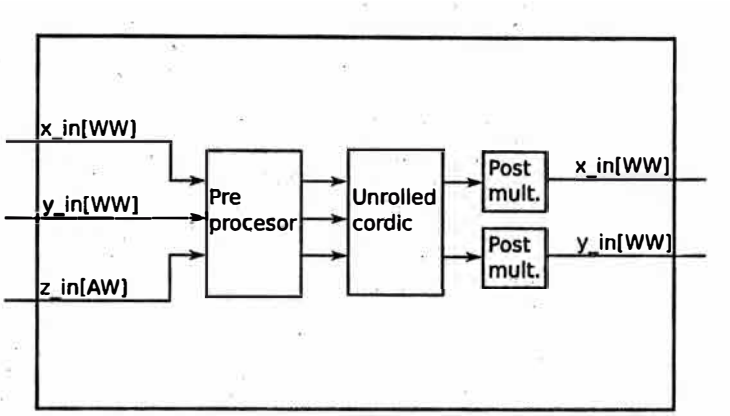


FIGURA C.12: Diagrama en bloques del módulo cordic
(Fuente: Referencia [14])

Los bloques *uRot* de la figura C.13 son los encargados de realizar las micro rotaciones sucesivas de acuerdo al algoritmo cordic. Los valores de los arctan(α) se 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.

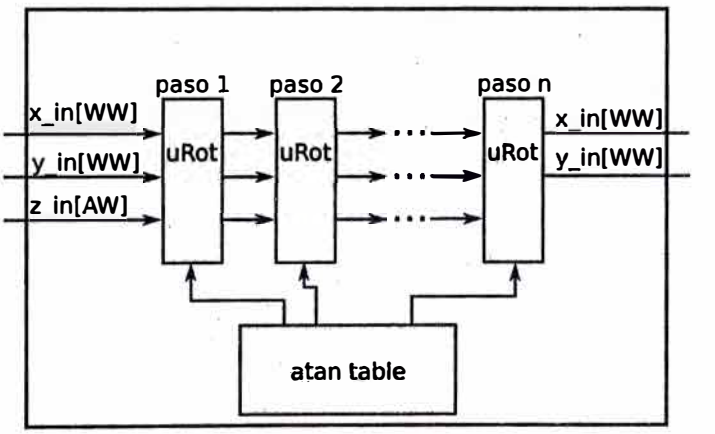


FIGURA C.13: Diagrama en bloques del bloque de rotaciones del módulo *cordic*
(Fuente: Referencia [14])

c.2.2 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 C.14.

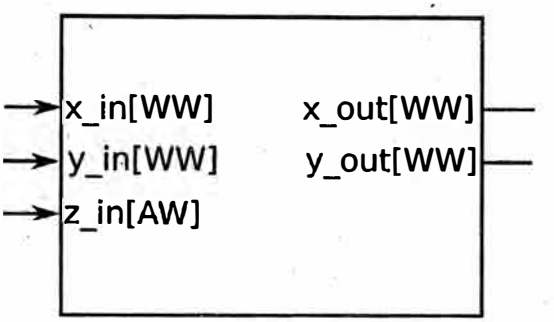


FIGURA C.14: Bloque de módulo multiplicador complejo
(Fuente: Referencia [14])

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

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

TABLA C.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 pre procesamiento 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 imple- menta 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.

c.2.3 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 3.1.4, son redondeo y truncamiento.

En la figura C.15 se muestra el diagrama en bloques de la unidad de escalamiento.

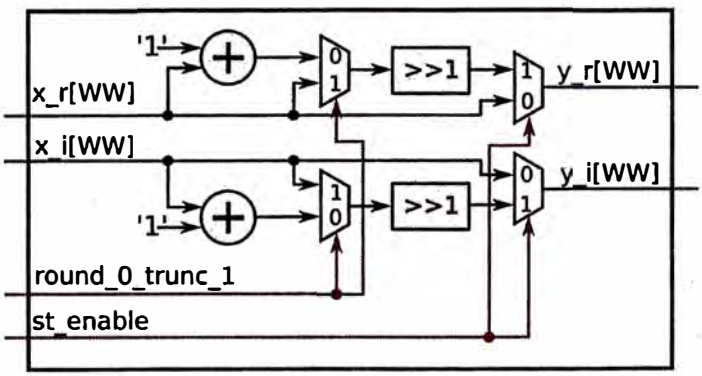


FIGURA C.15: Diagrama en bloques de la unidad de escalamiento
(Fuente: Referencia [14])

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 seña 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*.

ANEXO D

Código fuente Complex.java

```
/*************************************************************************
 *  Compilation:  javac Complex.java
 *  Execution:    java Complex
 *
 *  Data type for complex numbers.
 *
 *  The data type is "immutable" so once you create and initialize
 *  a Complex object, you cannot change it. The "final" keyword
 *  when declaring re and im enforces this rule, making it a
 *  compile-time error to change the .re or .im instance variables after
 *  they've been initialized.
 *
 *  % java Complex
 *  a            = 5.0 + 6.0i
 *  b            = -3.0 + 4.0i
 *  Re(a)        = 5.0
 *  Im(a)        = 6.0
 *  b + a        = 2.0 + 10.0i
 *  a - b        = 8.0 + 2.0i
 *  a * b        = -39.0 + 2.0i
 *  b * a        = -39.0 + 2.0i
 *  a / b        = 0.36 - 1.52i
 *  (a / b) * b  = 5.0 + 6.0i
 *  conj(a)      = 5.0 - 6.0i
 *  |a|          = 7.810249675906654
 *  tan(a)       = -6.685231390246571E-6 + 1.0000103108981198i
 *
 *************************************************************************/

import java.util.Objects;

public class Complex {
    private final double re;   // the real part
    private final double im;   // the imaginary part

    // create a new object with the given real and imaginary parts
    public Complex(double real, double imag) {
        re = real;
        im = imag;
    }

    // return a string representation of the invoking Complex object
    public String toString() {
        if (im == 0) return re + "";
        if (re == 0) return im + "i";
        if (im <  0) return re + " - " + (-im) + "i";
        return re + " + " + im + "i";
    }

    // return abs/modulus/magnitude
    public double abs() {
        return Math.hypot(re, im);
    }

    // return angle/phase/argument, normalized to be between -pi and pi
    public double phase() {
        return Math.atan2(im, re);
    }

    // return a new Complex object whose value is (this + b)
```

```
    public Complex plus(Complex b) {
        Complex a = this;             // invoking object
        double real = a.re + b.re;
        double imag = a.im + b.im;
        return new Complex(real, imag);
    }

    // return a new Complex object whose value is (this - b)
    public Complex minus(Complex b) {
        Complex a = this;
        double real = a.re - b.re;
        double imag = a.im - b.im;
        return new Complex(real, imag);
    }

    // return a new Complex object whose value is (this * b)
    public Complex times(Complex b) {
        Complex a = this;
        double real = a.re * b.re - a.im * b.im;
        double imag = a.re * b.im + a.im * b.re;
        return new Complex(real, imag);
    }

    // return a new object whose value is (this * alpha)
    public Complex scale(double alpha) {
        return new Complex(alpha * re, alpha * im);
    }

    // return a new Complex object whose value is the conjugate of this
    public Complex conjugate() {
        return new Complex(re, -im);
    }

    // return a new Complex object whose value is the reciprocal of this
    public Complex reciprocal() {
        double scale = re*re + im*im;
        return new Complex(re / scale, -im / scale);
    }

    // return the real or imaginary part
    public double re() { return re; }
    public double im() { return im; }

    // return a / b
    public Complex divides(Complex b) {
        Complex a = this;
        return a.times(b.reciprocal());
    }

    // return a new Complex object whose value is the complex exponential of this
    public Complex exp() {
        return new Complex(Math.exp(re) * Math.cos(im), Math.exp(re) * Math.sin(im));
    }

    // return a new Complex object whose value is the complex sine of this
    public Complex sin() {
        return new Complex(Math.sin(re) * Math.cosh(im), Math.cos(re) * Math.sinh(im));
    }

    // return a new Complex object whose value is the complex cosine of this
    public Complex cos() {
        return new Complex(Math.cos(re) * Math.cosh(im), -Math.sin(re) * Math.sinh(im));
    }

    // return a new Complex object whose value is the complex tangent of this
    public Complex tan() {
        return sin().divides(cos());
    }



    // a static version of plus
    public static Complex plus(Complex a, Complex b) {
        double real = a.re + b.re;
        double imag = a.im + b.im;
        Complex sum = new Complex(real, imag);
```

```
        return sum;
    }

    // See Section 3.3.
    public boolean equals(Object x) {
        if (x == null) return false;
        if (this.getClass() != x.getClass()) return false;
        Complex that = (Complex) x;
        return (this.re == that.re) && (this.im == that.im);
    }

    // See Section 3.3.
    public int hashCode() {
        return Objects.hash(re, im);
    }

    // sample client for testing
    public static void main(String[] args) {
        Complex a = new Complex(5.0, 6.0);
        Complex b = new Complex(-3.0, 4.0);

        StdOut.println("a            = " + a);
        StdOut.println("b            = " + b);
        StdOut.println("Re(a)        = " + a.re());
        StdOut.println("Im(a)        = " + a.im());
        StdOut.println("b + a        = " + b.plus(a));
        StdOut.println("a - b        = " + a.minus(b));
        StdOut.println("a * b        = " + a.times(b));
        StdOut.println("b * a        = " + b.times(a));
        StdOut.println("a / b        = " + a.divides(b));
        StdOut.println("(a / b) * b  = " + a.divides(b).times(b));
        StdOut.println("conj(a)      = " + a.conjugate());
        StdOut.println("|a|          = " + a.abs());
        StdOut.println("tan(a)       = " + a.tan());
    }

}
```

Código fuente FFT.java

```
/******************************************************************************
 *  Compilation:  javac FFT.java
 *  Execution:    java FFT n
 *  Dependencies: Complex.java
 *
 *  Compute the FFT and inverse FFT of a length n complex sequence
 *  using the radix 2 Cooley-Tukey algorithm.

 *  Bare bones implementation that runs in O(n log n) time. Our goal
 *  is to optimize the clarity of the code, rather than performance.
 *
 *  Limitations
 *  -----------
 *    - assumes n is a power of 2
 *
 *    - not the most memory efficient algorithm (because it uses
 *      an object type for representing complex numbers and because
 *      it re-allocates memory for the subarray, instead of doing
 *      in-place or reusing a single temporary array)
 *
 *  For an in-place radix 2 Cooley-Tukey FFT, see Radix2FFT.java
 *  *
 ******************************************************************************/

public class FFT {

    // compute the FFT of x[], assuming its length is a power of 2
    public static Complex[] fft(Complex[] x) {
        int n = x.length;

        // base case
        if (n == 1) return new Complex[] { x[0] };

        // radix 2 Cooley-Tukey FFT
        if (n % 2 != 0) {
            throw new IllegalArgumentException("n is not a power of 2");
        }

        // fft of even terms
        Complex[] even = new Complex[n/2];
        for (int k = 0; k < n/2; k++) {
            even[k] = x[2*k];
        }
        Complex[] q = fft(even);

        // fft of odd terms
        Complex[] odd  = even;  // reuse the array
        for (int k = 0; k < n/2; k++) {
            odd[k] = x[2*k + 1];
        }
        Complex[] r = fft(odd);

        // combine
        Complex[] y = new Complex[n];
        for (int k = 0; k < n/2; k++) {
            double kth = -2 * k * Math.PI / n;
            Complex wk = new Complex(Math.cos(kth), Math.sin(kth));
            y[k]       = q[k].plus(wk.times(r[k]));
            y[k + n/2] = q[k].minus(wk.times(r[k]));
        }
        return y;
    }


    // compute the inverse FFT of x[], assuming its length is a power of 2
    public static Complex[] ifft(Complex[] x) {
        int n = x.length;
        Complex[] y = new Complex[n];

        // take conjugate
        for (int i = 0; i < n; i++) {
            y[i] = x[i].conjugate();
        }
```

```
        // compute forward FFT
        y = fft(y);

        // take conjugate again
        for (int i = 0; i < n; i++) {
            y[i] = y[i].conjugate();
        }

        // divide by n
        for (int i = 0; i < n; i++) {
            y[i] = y[i].scale(1.0 / n);
        }

        return y;

    }

    // compute the circular convolution of x and y
    public static Complex[] cconvolve(Complex[] x, Complex[] y) {

        // should probably pad x and y with 0s so that they have same length
        // and are powers of 2
        if (x.length != y.length) {
            throw new IllegalArgumentException("Dimensions don't agree");
        }

        int n = x.length;

        // compute FFT of each sequence
        Complex[] a = fft(x);
        Complex[] b = fft(y);

        // point-wise multiply
        Complex[] c = new Complex[n];
        for (int i = 0; i < n; i++) {
            c[i] = a[i].times(b[i]);
        }

        // compute inverse FFT
        return ifft(c);
    }


    // compute the linear convolution of x and y
    public static Complex[] convolve(Complex[] x, Complex[] y) {
        Complex ZERO = new Complex(0, 0);

        Complex[] a = new Complex[2*x.length];
        for (int i = 0;        i <   x.length; i++) a[i] = x[i];
        for (int i = x.length; i < 2*x.length; i++) a[i] = ZERO;

        Complex[] b = new Complex[2*y.length];
        for (int i = 0;        i <   y.length; i++) b[i] = y[i];
        for (int i = y.length; i < 2*y.length; i++) b[i] = ZERO;

        return cconvolve(a, b);
    }

    // display an array of Complex numbers to standard output
    public static void show(Complex[] x, String title) {
        StdOut.println(title);
        StdOut.println("-------------------");
        for (int i = 0; i < x.length; i++) {
            StdOut.println(x[i]);
        }
        StdOut.println();
    }


/*************************************************************************
 *  Test client and sample execution
 *
 *  % java FFT 4
 *  x
 *  -------------------
 *  -0.03480425839330703
```

```
 *  0.07910192950176387
 *  0.7233322451735928
 *  0.1659819820667019
 *
 *  y = fft(x)
 *  -------------------
 *  0.9336118983487516
 *  -0.7581365035668999 + 0.08688005256493803i
 *  0.44344407521182005
 *  -0.7581365035668999 - 0.08688005256493803i
 *
 *  z = ifft(y)
 *  -------------------
 *  -0.03480425839330703
 *  0.07910192950176387 + 2.6599344570851287E-18i
 *  0.7233322451735928
 *  0.1659819820667019 - 2.6599344570851287E-18i
 *
 *  c = cconvolve(x, x)
 *  -------------------
 *  0.5506798633981853
 *  0.23461407150576394 - 4.033186818023279E-18i
 *  -0.016542951108772352
 *  0.10288019294318276 + 4.033186818023279E-18i
 *
 *  d = convolve(x, x)
 *  -------------------
 *  0.001211336402308083 - 3.122502256758253E-17i
 *  -0.005506167987577068 - 5.058885073636224E-17i
 *  -0.044092969479563274 + 2.1934338938072244E-18i
 *  0.10288019294318276 - 3.614732306247811SE-17i
 *  0.5494685269958772 + 3.122502256758253E-17i
 *  0.240120239493341 + 4.655566391833896E-17i
 *  0.02755001837079092 - 2.1934338938072244E-18i
 *  4.01805098805014E-17i
 *
 ******************************************************************************/

public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    Complex[] x = new Complex[n];

    // original data
    for (int i = 0; i < n; i++) {
        x[i] = new Complex(i, 0);
        x[i] = new Complex(-2*Math.random() + 1, 0);
    }
    show(x, "x");

    // FFT of original data
    Complex[] y = fft(x);
    show(y, "y = fft(x)");

    // take inverse FFT
    Complex[] z = ifft(y);
    show(z, "z = ifft(y)");

    // circular convolution of x with itself
    Complex[] c = cconvolve(x, x);
    show(c, "c = cconvolve(x, x)");

    // linear convolution of x with itself
    Complex[] d = convolve(x, x);
    show(d, "d = convolve(x, x)");
}

}
```

Código fuente Radix2FFT.java

```
/*************************************************************************
 *  Compilation:  javac Radix2FFT.java
 *  Execution:    java Radix2FFT n
 *  Dependencies: Complex.java
 *
 *  Compute the FFT of a length n complex sequence in-place.
 *  Uses a non-recursive version of the Cooley-Tukey FFT.
 *  Runs in O(n log n) time.
 *
 *  Reference:  Algorithm 1.6.1 in Computational Frameworks for the
 *  Fast Fourier Transform by Charles Van Loan.
 *
 *
 *  Limitations
 *  -----------
 *   - assumes n is a power of 2
 *
 *
 *************************************************************************/

public class Radix2FFT {

    // compute the FFT of x[], assuming its length is a power of 2
    public static void fft(Complex[] x) {

        // check that length is a power of 2
        int n = x.length;
        if (Integer.highestOneBit(n) != n) {
            throw new RuntimeException("n is not a power of 2");
        }

        // bit reversal permutation
        int shift = 1 + Integer.numberOfLeadingZeros(n);
        for (int k = 0; k < n; k++) {
            int j = Integer.reverse(k) >>> shift;
            if (j > k) {
                Complex temp = x[j];
                x[j] = x[k];
                x[k] = temp;
            }
        }

        // butterfly updates
        for (int L = 2; L <= n; L = L+L) {
            for (int k = 0; k < L/2; k++) {
                double kth = -2 * k * Math.PI / L;
                Complex w = new Complex(Math.cos(kth), Math.sin(kth));
                for (int j = 0; j < n/L; j++) {
                    Complex tao = w.times(x[j*L + k + L/2]);
                    x[j*L + k + L/2] = x[j*L + k].minus(tao);
                    x[j*L + k]       = x[j*L + k].plus(tao);
                }
            }
        }
    }


    // test client
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        Complex[] x = new Complex[n];

        // original data
        for (int i = 0; i < n; i++) {
            x[i] = new Complex(i, 0);
            // x[i] = new Complex(-2*Math.random() + 1, 0);
        }
        for (int i = 0; i < n; i++)
            StdOut.println(x[i]);
        StdOut.println();

        // FFT of original data
        fft(x);
        for (int i = 0; i < n; i++)
```

```
            StdOut.println(x[i]);
        StdOut.println();

    }

}
```