INSTITUTO POLITECNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

DISEÑO DE UN CIRCUITO PARA LA EVALUACIÓN DE LA TRANSFORMADA RÁPIDA DE FOURIER UTILIZANDO LÓGICA PROGRAMABLE

TESIS

QUE PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS EN INGENIERIA DE CÓMPUTO

CON OPCIÓN EN SISTEMAS DIGITALES

PRESENTA

NAYELI VEGA GARCÍA

DIRECTORES DE TESIS

M. en C. OSVALDO ESPINOSA SOSA

M. en C. VICTOR HUGO GARCÍA ORTEGA

México D. F., Julio de 2013

SIP-14 bis

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 16:00 horas del día 3 del mes de junio de 2013 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Diseño de un circuito para la evaluación de la Transformada Rápida de Fourier utilizando lógica programable"

Presentada por el alumno:

VEGA	GARCÍA	NAYELI
Apellido paterno	Apellido materno	Nombre(s)

Con registro: A 1 1 0 8 7 3

aspirante de: **MAESTRÍA EN CIENCIAS EN INGENIERÍA DE CÓMPUTO CON OPCIÓN EN SISTEMAS DIGITALES**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Directores de Tesis

M. en C. Osvaldo Espinosa Sosa

M. en C. Víctor Hugo García Ortega

Dr. Sergio Suárez Guerra

Dr. Marco Antonio Ramírez Salinas

Dr. José Luis Oropeza Rodríguez

M. en C. Romeo Urbieta Parrazales

PRESIDENTE DEL COLEGIO DE PROFESORES

Dr. Luis Alfonso Villa Vargas

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de México, D. F. el día 12 del mes de Junio del año 2013, el (la) que suscribe Nayeli Vega García alumna del Programa de Maestría en Ciencias en Ingeniería de Cómputo con opción en Sistemas Digitales con número de registro A110873 adscrita al Centro de Investigación en Computación, manifiesta que es autor (a) intelectual del presente trabajo de Tesis bajo la dirección de M. en C. Osvaldo Espinosa Sosa y M. en C. Víctor Hugo García Ortega y cede los derechos del trabajo intitulado "Diseño de un circuito para la evaluación de la Transformada Rápida de Fourier utilizando lógica programable", al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso epreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección nvegag0126@gmail.com. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Nayeli Vega García

Nombre y firma

Agradecimientos

Al Consejo Nacional de Ciencia y Tecnología, CONACYT, por su apoyo durante mis estudios de maestría.

A mi alma mater:
El Instituto Politécnico Nacional y al Centro de Investigación en Computación por darme la oportunidad de ser parte de una comunidad en donde comencé mi formación como profesionista desarrollando mis estudios de maestría.

A mis asesores de tesis M. en C. Osvaldo Espinosa Sosa y M. en C. Victor Hugo García Ortega por sus acertadas sugerencias, consejos, tiempo y su invaluable ayuda durante el desarrollo de esta tesis y no con menos importancia a todos los profesores investigadores con los que tuve la oportunidad de trabajar por las enseñanzas impartidas.

Dedicatorias

A mis padres Jaime Vega Pérez y Blanca García …

Con profundo respeto por el gran ejemplo de vida que dan.

A mis hermanos Sarai, Jaime y Paulina …

Por todas las experiencias que hemos vivido, esperando dar un buen ejemplo y porque sigamos trabajando y creciendo juntos como hasta ahora.

A mis amigos…

Por el apoyo y palabras de aliento brindadas durante la realización de este trabajo.

Resumen

La Transformada discreta de Fourier es una herramienta muy importante para el procesamiento digital de señales puesto que permite obtener el espectro de frecuencias de una señal discreta en el tiempo, esto permite realizar tareas como compresión de datos, convoluciones, filtrado de señales entre otras, sin embargo, obtener el espectro de frecuencias utilizando este método requiere de una gran cantidad de recursos por lo que se utiliza el algoritmo de la transformada rápida de Fourier que permite realizar el mismo trabajo reduciendo tiempo y recursos requeridos. Contar con hardware dedicado a la evaluación de la FFT da pie a la generación de diversas aplicaciones del procesamiento digital de señales, como las ya mencionadas. Opciones para la implementación hardware de este algoritmo existen muchas, sin embargo, en años recientes el hardware reconfigurable ha mostrado tener grandes ventajas sobre otras existentes puesto que el esfuerzo de implementación es de nivel medio obteniendo buenos resultados.

Por las características del algoritmo de la FFT, la implementación hardware puede hacerse de diversas formas teniendo como ventaja el uso de hardware reconfigurable, en este trabajo se utilizó la técnica de implementación en cascada que permitió la reutilización de hardware por cada etapa del algoritmo combinado con un esquema de memoria *non-inplace* que facilitó la eliminación de dependencias de datos entre cada etapa de la arquitectura. Esta propuesta tiene la característica de ser fácilmente reconfigurable permitiendo así variar el número de puntos a evaluar haciendo pequeños cambios en las unidades de control sin modificar la ruta de datos de la arquitectura.

Finalmente se obtuvo una arquitectura que evalúa el algoritmo de la FFT para 512 puntos en un tiempo de 5.16μs a una frecuencia máxima de operación de 49.41 MHz, permitiendo un ancho de banda de 49.41 MHz con un error absoluto de 7.02 unidades en el peor de los casos, evaluando hasta 9 productos complejos por ciclo de reloj una vez que el *pipeline* se encuentra lleno. Los resultados obtenidos fueron comparados con los resultados de la evaluación realizada por Matlab y por un programa codificado en C para finalmente concluir que la arquitectura propuesta se desempeña de manera adecuada.

Abstract

The Discrete Fourier transform is a very important tool in digital signal processing since it allows to obtain the frequency spectrum of a discrete signal in time, allowing to perform tasks such as data compression, convolutions, signal filtering and many others, but the frequency spectrum obtained using this method requires a lot of computing resources, so the Fast Fourier Transform algorithm is used. The same amount of work is done with reduced time and resources. Having hardware dedicated to evaluate the Fast Fourier Transform leads to the generation of applications in digital signal processing. Options for hardware implementation of this algorithm are many and varied, as, in recent years, reconfigurable hardware has shown to have great advantages over other existing methods since the implementation effort is of medium difficulty with very good results.

Given the properties of the FFT algorithm, its hardware implementation can be done in various ways, and taking advantage of using reconfigurable hardware, the cascade implementation technique was used, that allowed the reuse of hardware for each stage of the algorithm, combined with a pattern of non-inplace memory which facilitated the removal of data dependencies between each stage of the architecture. This proposal allows the system to be readily reconfigurable, thereby enabling to vary the number of items to be evaluated by making small changes in the control units without changing the data path architecture.

An architecture was designed that evaluates FFT algorithm for 512 points in a 5.16μs time at a maximum frequency rate of 49.41 MHz, allowing bandwidth of 49.41 MHz with an absolute error of 7.02 units in the worst case, evaluating up to 9 complex products per clock cycle once the pipeline is full. The results were compared with those evaluated by programs written in Matlab and C. We conclude that the proposed architecture performs adequately.

Glosario

ASIC: Por sus siglas en inglés, *Aplicattion-specific integrated circuit.* Es un circuito integrado diseñado a la medida para realizar una función en particular. Por ejemplo, un circuito amplificador operacional es un ASIC.

DFT: Por sus siglas en inglés, *Discrete Fourier transform.* Es un tipo de transformada discreta utilizada en el análisis de Fourier. Permite la obtención del espectro de frecuencias de una señal discreta en el tiempo.

DSP: Por sus siglas en inglés, *Digital Signal Processor.* Es un sistema con base en un procesador o microprocesador que posee un conjunto de instrucciones, hardware y software optimizados para aplicaciones que requieran operaciones numéricas en tiempo real, específicamente para aplicaciones de procesamiento digital de señales.

FFT: Por sus siglas en inglés, *Fast Fourier Transform.* Es un algoritmo que permite calcular la transformada de discreta de Fourier, propuesto por Coley y Tukey en 1965 que tiene gran importancia pues permitió el ahorro de tiempo y recursos para la obtención del espectro de frecuencias de una señal.

FPGA: Por sus siglas en inglés, *Field Programable Gate Array*. Es un dispositivo lógico programable que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada mediante un lenguaje de descripción especializado.

LE: Por sus siglas en inglés, *Logic Element.* Es la unidad lógica más pequeña en la arquitectura del FPGA (Altera). Como principal característica contiene una tabla de búsqueda (LUT, *look-up table*) de cuatro entradas, que es un generador de funciones, por lo que puede implementar cualquier función de cuatro variables.

VHDL: Por sus siglas en inglés, *Very High Speed Integrated Circuit Hardware Description Lenguage*, lenguaje creado a partir de un proyecto promovido por el gobierno de Estados Unidos en 1981 y fue homologado por el IEEE en 1987 en el estándar 1076-87. VHDL utiliza una metodología de diseño de arriba hacia abajo (*top-bottom*) que permite describir el circuito de forma estructural (señales y sus

conexiones) y funcional (qué hace). Cada bloque se puede definir empleando sub-circuitos definidos en el mismo proyecto o disponibles en bibliotecas propias o de terceros.

FIFO: Tipo de memoria de rápido acceso el cual funciona bajo un esquema *first in – first out* (primero en entrar - primero en salir).

IPC: Número de instrucciones ejecutadas en un procesador por ciclo de reloj

CPI: Ciclos de reloj necesarios por cada instrucción ejecutada en un procesador

Índice de contenido

Resumen ..... **VI**

Abstract ..... **VII**

Glosario ..... **VIII**

Índice de contenido ..... **X**

Índice de figuras ..... **XIII**

Índice de tablas ..... **XVIII**

Capítulo 1. Introducción **1**

1.1 Antecedentes 1
1.2 Planteamiento del problema 5
1.3 Objetivos 6
1.3.1 Objetivo general 6
1.3.2 Objetivos específicos 6
1.4 Justificación 7
1.5 Alcances y límites 8

Capítulo 2. Marco teórico y estado del Arte **9**

2.1 Transformada rápida de Fourier 9
2.1.1 Algoritmo de la FFT Radix-2 11
2.2 Formato de punto fijo para la representación de números fraccionarios 15
2.3 Cómputo reconfigurable para el Procesamiento Digital de Señales 17
2.4 Arquitecturas para la evaluación de la FFT con base en cómputo reconfigurable 20
2.4.1 Procesador secuencial 21
2.4.2 Procesador en cascada 23
2.4.3 Procesador paralelo-iterativo 26
2.4.4 Procesador en arreglo 28

2.5 Metodología para el diseño digital 29
2.5.1 Método *buttom-up* 29
2.5.2 Método *top-down* 30
2.6 Lenguajes de descripción de hardware 32
2.6.1 Lenguaje de descripción VHDL *(VHSIC Hardware Description Language*) 33
2.6.2 Ventajas de VHDL 34
2.7 Solución propuesta 36

Capítulo 3. Diseño de la arquitectura propuesta 38

3.1 Bloque de etapa de la arquitectura para evaluar la FFT 38
3.2 Bloque de memoria de datos 41
3.2.1 Implementación de memorias FIFO para almacenamiento de datos 41
3.2.2 Implementación de memorias RAM para almacenamiento de datos 44
3.3 Bloque de *bit-reverse* 48
3.4 Bloque generador de direcciones 49
3.5 Bloque generador de coeficientes complejos 51
3.6 Bloque para evaluación de la mariposa 53
3.7 Bloque para el cálculo del módulo de resultados 58
3.8 Unidad de control por etapa 60
3.8.1 Control para la señal *we* 61
3.8.2 Control para la señal *sm* 65
3.8.3 Control para la señal *eg* y la habilitación de *we* y *sm* 69
3.9 Unidad de control general 70

Capítulo 4. Implementación y pruebas funcionales 76

4.1 Pruebas del bloque de memoria de datos 76
4.2 Pruebas del bloque *bit-reverse* 79
4.3 Pruebas del bloque generador de direcciones 81
4.4 Pruebas del bloque generador de coeficientes complejos 84
4.5 Pruebas del bloque para la evaluación de la mariposa 87
4.6 Pruebas del bloque para el cálculo del módulo de resultados 94
4.7 Pruebas de la unidad de control por etapa 98
4.7.1 Pruebas de control para la señal *we* 98
4.7.2 Pruebas de control para la señal *sm* 102

4.7.3 Pruebas de control para la señal *eg* y habilitación de *we* y *sm* 106
4.7.4 Pruebas de integración de la unidad de control por etapa 107
4.8 Pruebas de la unidad de control general 109
4.9 Pruebas del bloque de etapa de la arquitectura para la evaluación de la FFT 112
4.10 Pruebas funcionales de la arquitectura 121

Capítulo 5. Presentación de resultados 127

5.1 Arquitectura para la evaluación de 512 puntos 127
5.2 Comparativa de los resultados hardware-software de la evaluación 129
5.3 Prueba del funcionamiento de la arquitectura sobre la tarjeta de desarrollo *Altera-DE2* 143
5.4 Estadísticas temporales y comparación con el estado del arte 149
5.4.1 Propuesta de medida de rendimiento 152

Capítulo 6. Conclusiones y trabajo a futuro 156

6.1 Conclusiones 156
6.2 Trabajo a futuro 157

Referencias 159

A. Programa en C para el cálculo de la FFT 162

B. Código fuente 165

Índice de figuras

Fig. 1.1 Diagrama de bloques de un sistema de procesamiento digital de señales [1]. 2

Fig. 2.1 Mariposa para algoritmo FFT Radix-2. 12
Fig. 2.2 Algoritmo de la FFT Radix-2 [1]. 13
Fig. 2.3 Representación de números en formato de punto fijo 15
Fig. 2.4 Espectro de implementación de algoritmos de Procesamiento Digital de Señales [8]. 17
Fig. 2.5 Evaluación de la FFT para N=8. 20
Fig. 2.6 Esquema de procesamiento secuencial. 21
Fig. 2.7 Esquema de procesamiento de memoria doble. 22
Fig. 2.8 Arquitectura propuesta en [12]. 22
Fig. 2.9 Arquitectura propuesta en [14]. 23
Fig. 2.10 Esquema de procesamiento en cascada. 23
Fig. 2.11 Arquitectura propuesta en [18]. 24
Fig. 2.12 Arquitectura propuesta en [17]. 25
Fig. 2.13 Arquitectura propuesta en [19]. 25
Fig. 2.14 Esquema de procesamiento en paralelo-iterativo. 26
Fig. 2.15 Arquitectura propuesta por [20]. 27
Fig. 2.16 Arquitectura propuesta en [21]. 28
Fig. 2.17 Esquema de procesamiento en arreglo. 28
Fig. 2.18 Flujo general del diseño electrónico. 29
Fig. 2.19 Método *bottom-up* para el diseño de circuitos. 30
Fig. 2.20 Método de diseño *top-down*. 31
Fig. 2.21 Flujo de diseño de un circuito en VHDL. 33
Fig. 2.22 Diagrama general de bloques de la arquitectura propuesta. 36

Fig. 3.1 Bloque funcional para la evaluación de una etapa del algoritmo FFT. 39
Fig. 3.2 Flujo de operaciones para el algoritmo de la FFT Radix-2 con decimación en frecuencia 42
Fig. 3.3 Evaluación de la FFT Radix-2 decimación en frecuencia utilizando memorias FIFO 43
Fig. 3.4 Diagrama de bloques de la distribución de un bloque de memoria de datos. 45
Fig. 3.5 Configuración de Memoria M4K. 47
Fig. 3.6 Entidad del bloque para bit-reverse. 48
Fig. 3.7 Diagrama de componentes del bloque bit-reverse. 48
Fig. 3.8 Entidad del bloque Generador de direcciones. 50
Fig. 3.9 Diagrama de flujo para el algoritmo para la generación de direcciones. 51
Fig. 3.10 Entidad del Generador de coeficientes complejos. 52
Fig. 3.11 Diagrama de bloques del generador de coeficientes complejos. 52
Fig. 3.12 Configuración para el bloque M4K del bloque de memoria. 53
Fig. 3.13 Entidad genérica de la mariposa. 53
Fig. 3.14 Flujo de operaciones de mariposa. 54
Fig. 3.15 Definición de la operación mariposa para la FFT. 54

Índice de figuras

Fig. 3.16 Truncamiento de la multiplicación. 55
Fig. 3.17 Escalamiento inter-etapa del resultado. 56
Fig. 3.18 Definición de la mariposa para la primera etapa. 56
Fig. 3.19 Diagrama de bloques de la mariposa. 58
Fig. 3.20 Entidad del bloque para el cálculo del módulo de resultados. 58
Fig. 3.21 Flujo de operaciones para el cálculo del módulo. 59
Fig. 3.22 Entidad de la unidad de control por etapa 61
Fig. 3.23 Diagrama de bloques de la unidad de control por etapa. 61
Fig. 3.24 Carta ASM para el control de la señal *WE*. 62
Fig. 3.25 Entidad para el contador ModN/2 63
Fig. 3.26 Diagrama de bloques del componente de control de *we*. 64
Fig. 3.27 Máquina de estados para el manejo de la señal *we*. 64
Fig. 3.28 Carta ASM para el control de la señal *sm*. 66
Fig. 3.29 Diagrama de bloques de la unidad para el control de *sm*. 67
Fig. 3.31 Entidad de la Unidad de control general. 71
Fig. 3.32 Carta ASM para la unidad de control general 72
Fig. 3.33 Máquina de estados para la unidad de control general. 73
Fig. 3.34 Diagrama de bloques de la unidad de control general. 73

Fig. 4.1 Simulación de la escritura de un bloque de memoria M4K. 76
Fig. 4.2 Simulación funcional de un banco de memoria de datos. 78
Fig. 4.3 Simulación funcional del bloque *bit-reverse*. 80
Fig. 4.4 Diagrama *RTL* del bloque *bit-reverse*. 81
Fig. 4.5 Resultados de síntesis del bloque de bit-reverse para la arquitectura de 512 puntos 81
Fig. 4.6 Simulación funcional del bloque generador de direcciones. 82
Fig. 4.7 Simulación funcional para el bloque generador de direcciones de datos para la segunda etapa. .. 83
Fig. 4.8 Diagrama *RTL* de Generador de direcciones. 83
Fig. 4.9 Resultados de síntesis del bloque generador de direcciones de datos 84
Fig. 4.10 Simulación funcional del bloque generador de coeficientes complejos. 84
Fig. 4.11 Diagrama *RTL* del bloque generador de coeficientes complejos. 86
Fig. 4.12 Resultados de la síntesis para el bloque de coeficientes complejos de la tercera etapa de la arquitectura 87
Fig. 4.13 Simulación funcional del bloque para la evaluación de la mariposa de la primera etapa. 88
Fig. 4.14 Diagrama *RTL* del bloque para la evaluación de la mariposa de la primera etapa. 88
Fig. 4.15 Resultados de la síntesis de la mariposa para la evaluación de la primera etapa 89
Fig. 4.16 Simulación funcional del bloque para la evaluación de la mariposa de la segunda etapa. 89
Fig. 4.17 Diagrama *RTL* del bloque para la evaluación de la mariposa de la segunda etapa. 90
Fig. 4.18 Resultados de la síntesis del bloque para la evaluación de la mariposa de la segunda etapa 91
Fig. 4.19 Simulación funcional del bloque para la evaluación de la mariposa a partir de la tercera etapa. 92
Fig. 4.20 Diagrama RTL del bloque para la evaluación de la mariposa a partir de la tercera etapa. 93
Fig. 4.21 Resultados de síntesis del bloque de mariposa genérico 94
Fig. 4.22 Simulación funcional del módulo generador de direcciones para el cálculo del módulo. 94

Índice de figuras

Fig. 4.23 Diagrama RTL del generador de direcciones para el bloque de cálculo del módulo de resultados. 95
Fig. 4.24 Simulación funcional del cálculo del módulo de resultados. 96
Fig. 4.25 Diagrama RTL del componente para el cálculo del módulo de resultados. 97
Fig. 4.26 Resumen de síntesis del bloque de cálculo del módulo de resultados 97
Fig. 4.27 Simulación funcional del bloque contador para el control de la señal *we*. 98
Fig. 4.28 Diagrama RTL del bloque contador para el control de la señal *we*. 99
Fig. 4.29 Simulación funcional del bloque de generación de cambio de la señal *we*. 99
Fig. 4.30 Diagrama RTL del bloque de generación de cambio de la señal *we*. 100
Fig. 4.31 Simulación funcional del bloque de control para la señal *we*. 101
Fig. 4.32 Diagrama RTL del bloque para el control de la señal *we*. 102
Fig. 4.33 Simulación funcional del bloque contador para el control de la señal *sm*. 102
Fig. 4.34 Diagrama RTL del bloque contador para el control de la señal *sm*. 103
Fig. 4.35 Simulación funcional del contador *sm*. 103
Fig. 4.36 Diagrama RTL del bloque de generación de cambio de la señal *sm*. 104
Fig. 4.37 Simulación funcional del bloque de control para la señal *sm*. 104
Fig. 4.38 Diagrama RTL del circuito para el control de la señal *sm*. 106
Fig. 4.39 Simulación funcional del control para la señal *eg* y habilitación de los bloques de control de *sm* y *we*. 106
Fig. 4.40 Diagrama *RTL* del bloque de control para la señal *eg* y habilitación de *sm* y *we*. 107
Fig. 4.41 Simulación funcional de la unidad de control por etapa. 108
Fig. 4.42 Diagrama *RTL* de la unidad de control por etapa. 109
Fig. 4.43 Resultados de síntesis de la unidad de control por etapa. 109
Fig. 4.44 Simulación funcional del bloque de control general. 110
Fig. 4.45 Diagrama *RTL* de la unidad de control general. 112
Fig. 4.46 Resultados de la síntesis de la unidad de control general 112
Fig. 4.47 Simulación de bloque de etapa en condiciones ideales. 113
Fig. 4.48 Simulación de bloque de etapa con interrupción en la señal *EN_C*. 118
Fig. 4.49 Diagrama del circuito resultante de la síntesis del bloque de evaluación de una etapa de la arquitectura para evaluar la FFT (a) Unidad de control por etapa (b) Bloque generador de direcciones de datos (c) Bloque generador de coeficientes complejos (d) Multiplexores de direcciones (e) Bloques de memorias de datos (f) Multiplexores de datos (g) Registros de mariposa (h) Bloque para la evaluación de la mariposa. 120
Fig. 4.50 Resultados de la síntesis para la tercera etapa de la arquitectura 120
Fig. 4.51 Simulación funcional de la arquitectura. 123
Fig. 4.52 Simulación de la arquitectura completa para la evaluación del algoritmo con una interrupción en EN_GRAL. 124
Fig. 4.53 Verificación de la selección de las direcciones de lectura/escritura de las memorias de datos. 125
Fig. 4.54 Diagrama RTL del circuito resultante de la arquitectura para la evaluación de la FFT de 8 puntos. 126

Fig. 5.1 Diagrama RTL de la arquitectura para el cálculo de la FFT de 512 puntos. 128
Fig. 5.2 Diagrama *RTL* del circuito para la evaluación de una etapa de la FFT de 512 puntos. 128

Índice de figuras

Fig. 5.3 Resultado parte real de A de la mariposa de la última etapa de evaluación de la señal senoidal. 130
Fig. 5.4 Resultado parte imaginaria de A de la mariposa de la última etapa de evaluación de la señal senoidal. 130
Fig. 5.5 Resultado parte real de B de la mariposa de la última etapa de evaluación de la señal senoidal. 131
Fig. 5.6 Resultado parte imaginaria de B de la mariposa de la última etapa de evaluación de la señal senoidal. 131
Fig. 5.7 Comparativa de los resultados para la señal senoidal de entrada. 132
Fig. 5.8 Evolución de las diferencias para la evaluación software vs hardware con entrada de señal senoidal. 132
Fig. 5.9 Resultado parte real de A de la mariposa de la última etapa de evaluación de la señal cuadrada. 133
Fig. 5.10 Resultado parte imaginaria de A de la mariposa de la última etapa de evaluación de la señal cuadrada. 134
Fig. 5.11 Resultado parte real de B de la mariposa de la última etapa de evaluación de la señal cuadrada. 134
Fig. 5.12 Resultado parte imaginaria de B de la mariposa de la última etapa de evaluación de la señal cuadrada. 134
Fig. 5.13 Comparativa de los resultados para la señal cuadrada de entrada. 135
Fig. 5.14 Evolución de las diferencias para la evaluación software vs hardware con entrada de señal cuadrada. 136
Fig. 5.15 Resultado parte real de A de la mariposa de la última etapa de evaluación de la señal diente de sierra. 136
Fig. 5.16 Resultado parte imaginaria de A de la mariposa de la última etapa de evaluación de la señal diente de sierra. 137
Fig. 5.17 Resultado parte real de B de la mariposa de la última etapa de evaluación de la señal diente de sierra. 137
Fig. 5.18 Resultado parte imaginaria de B de la mariposa de la última etapa de evaluación de la señal diente de sierra. 137
Fig. 5.19 Comparativa de los resultados para la señal diente de sierra de entrada. 138
Fig. 5.20 Evolución de las diferencias para la evaluación software vs hardware con entrada de señal diente de sierra. 139
Fig. 5.21 Resultado parte real de A de la mariposa de la última etapa de evaluación de la señal de voz. 139
Fig. 5.22 Resultado parte imaginaria de A de la mariposa de la última etapa de evaluación de la señal de voz. 140
Fig. 5.23 Resultado parte real de B de la mariposa de la última etapa de evaluación de la señal de voz. 140
Fig. 5.24 Resultado parte imaginaria de B de la mariposa de la última etapa de evaluación de la señal de voz. 140
Fig. 5.25 Comparativa de los resultados para la señal de voz de entrada. 141
Fig. 5.26 Superposición de los espectros obtenidos para la señal de voz 142
Fig. 5.27 Evolución de las diferencias para la evaluación software vs hardware con entrada de señal de voz, fonema A. 142
Fig. 5.28 Configuración de *SignalTap II* con la arquitectura propuesta. 144
Fig. 5.29 Señal de entrada para la arquitectura propuesta. 144
Fig. 5.30 Circuito para la generación de datos de entrada a la arquitectura. 145

Índice de figuras

Fig. 5.31 Diagrama RTL de la aplicación final de prueba 145
Fig. 5.32 Resumen de síntesis de aplicación. 146
Fig. 5.33 Captura de pantalla del funcionamiento de *SignalTap II* y la aplicación de prueba. 146
Fig. 5.34 Captura de pantalla del funcionamiento de *MatLab* y la aplicación de prueba. 147
Fig. 5. 35 Captura de pantalla del funcionamiento de la aplicación de prueba para el fonema A 148
Fig. 5.36 Resumen de síntesis de la arquitectura propuesta. 149

Índice de tablas

Tabla 2.1 Reversión de bits para N = 8 puntos. 13
Tabla 2.2 Comparativa de complejidad computacional entre DFT y FFT [1]. 14
Tabla 2.3 Número de multiplicaciones y sumas reales no triviales para el cálculo de la DFT de N puntos compleja [24]. 14
Tabla 2.4 Comparación de hardware para implementación de Procesamiento Digital de Señales. 18
Tabla 2.5 Comparativa del hardware necesario para la evaluación de la FFT de 512 puntos. 37

Tabla 4.1 Vectores de prueba para la simulación del módulo de memoria M4K. 76
Tabla 4.2 Vectores de prueba para la simulación de un banco de memoria. 77
Tabla 4.3 Vectores de prueba para la simulación de bloque generador *bit-reverse*. 79
Tabla 4.4 Vectores de entrada para la simulación funcional del bloque generador de direcciones de datos. 82
Tabla 4.5 Vectores de entrada para la simulación funcional del bloque generador de direcciones de datos para la segunda etapa. 83
Tabla 4.6 Vectores de entrada para la simulación funcional del bloque generador de coeficientes complejos 85
Tabla 4.7 Contenido de la ROM para la parte real de los coeficientes. 85
Tabla 4.8 Contenido de la ROM para la parte imaginaria de los coeficientes. 85
Tabla 4.9 Vectores de prueba para la simulación del bloque de evaluación de la mariposa de la primera etapa. 87
Tabla 4.10 Flujo de operaciones para la evaluación de la mariposa de la primera etapa. 88
Tabla 4.11 Vectores de entrada para la simulación de la mariposa de la segunda etapa. 89
Tabla 4.12 Vectores de entrada para la simulación de la mariposa de las iteraciones 3 en adelante. 91
Tabla 4.13 Vectores de prueba para la simulación del bloque de cálculo del módulo de resultados. 95
Tabla 4.14 Vectores de resultados para el cálculo del módulo de resultados. 96
Tabla 4.15 Vectores de entrada para la simulación del Contador ModN/2 de la señal *we*. 98
Tabla 4.16 Vectores de entrada para la simulación del *Contador we*. 99
Tabla 4.17 Vectores de entrada para la simulación del control de la señal *we*. 100
Tabla 4.18 Vectores de entrada para la simulación del contador de periodicidad de la señal *sm*. 102
Tabla 4.19 Vectores de entrada para la simulación del contador *sm*. 104
Tabla 4.20 Vectores de prueba para la simulación funcional del control de la señal *sm*. 105
Tabla 4.21 Vectores de entrada para el control de la señal *eg* y habilitación de *sm* y *we*. 106
Tabla 4.22 Vectores de entrada para la simulación funcional de la unidad de control por etapa. 107
Tabla 4.23 Vectores de entrada para la simulación de la unidad de control general. 110
Tabla 4.24 Vectores de entrada para la simulación de una etapa de la arquitectura. 113
Tabla 4.25 Comportamiento de las señales de *we*, *sm* y *eg* con base en las entradas de control. 114
Tabla 4.26 Funcionamiento del bloque generador de direcciones. 115
Tabla 4.27 Selección de las direcciones para cada uno de los bancos de memoria mediante la señal *we*. 115
Tabla 4.28 Selección de los datos para ser evaluados por el bloque de mariposa mediante la señal *sm*. . 116

Índice de tablas

Tabla 4.29 Funcionamiento del bloque generador de los coeficientes complejos con base en el comportamiento de la señal eg. 117
Tabla 4.30 Funcionamiento de los registros de entrada al bloque para la evaluación del bloque de mariposa. 117
Tabla 4.31 Comportamiento de las señales *we, eg,* y *sm* ante la presencia de un cambio en *EN_C*. 119
Tabla 4.32 Vectores de entrada para la simulación funcional de la arquitectura para la evaluación de 8 puntos. 121
Tabla 4.33 Vectores de entrada para la simulación de la arquitectura completa con una interrupción en la señal de habilitación. 122

Tabla 5.1 Cambios realizados para la arquitectura de 512 puntos. 127
Tabla 5.2 Comparativa entre el estado del arte y la arquitectura propuesta. 150
Tabla 5.3 Comparación entre algoritmo Radix-2 vs Radix-4. 151
Tabla 5.4 Comparativa de los resultados con base en el número de operaciones por ciclo realizadas. 155

Capítulo 1. Introducción

1.1 Antecedentes

En la vida real nos encontramos rodeados de fenómenos físicos que varían en tiempo o espacio, a este tipo de variaciones se les llama señales. Una señal es definida como una cantidad física que varía con el tiempo, espacio o cualquier variable o variables independientes [1].

Una señal analógica es una función en tiempo continuo la cual toma valores de amplitud continuos, un ejemplo muy claro es la señal de voz. Las señales son un medio de comunicación entre personas y máquinas [2]. Con el fin de establecer un canal de comunicación efectivo, es necesario tratar a las señales de manera tal que se pueda obtener o destacar información de ellas, a este tratamiento se le llama procesamiento digital de señales, el cual comprende; la presentación, evaluación, transformación y manipulación de las señales.

El procesamiento digital de señales ha evolucionado rápidamente en las últimas tres décadas, su desarrollo es resultado de los avances significativos que se han dado en el area de la tecnología digital y de los circuitos integrados [1], de la misma manera que ofrece ventajas con respecto al procesamiento analógico de señales como son [2]:

- El resultado del proceso no se ve afectado por el envejecimiento de los componentes y los cambios de temperatura.
- Todos los dispositivos fabricados se comportan de manera idéntica, ya que la tolerancia de los componentes no influye en el procesamiento.
- El dispositivo en el que se está haciendo el procesamiento se puede reconfigurar de manera sencilla, no es necesario ajustar o reemplazar componentes.
- El procesamiento analógico de señales de muy baja frecuencia se dificulta debido al requerimiento de capacitores de gran capacidad y muy baja corriente de fuga.

En el caso del procesamiento digital de señales no existen limitaciones de este tipo y es capaz de desempeñar tareas en áreas como son:

- Análisis de vibraciones en máquinas para la detección temprana del desgaste de rodamientos y engranajes
- Radar: Medición a la distancia y velocidad de los contactos.

- Astronomía, detección de planetas y estrellas con base al movimiento oscilatorio que inducen en las estrellas alrededor de las cuales orbitan, entre otras.

Un sistema para el procesamiento digital de señales consta de los elementos que se observan en la Fig. 1.1.

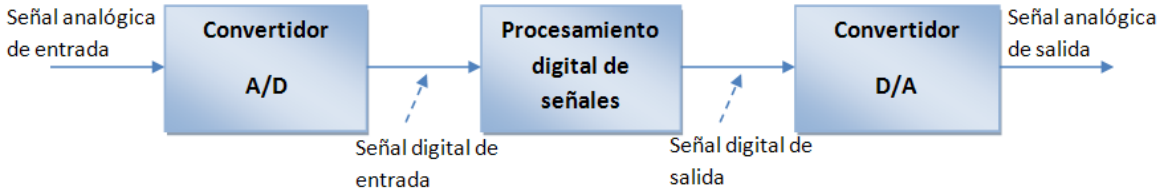


Fig. 1.1 Diagrama de bloques de un sistema de procesamiento digital de señales [1].

El bloque denominado convertidor analógico-digital es el encargado de hacer un muestreo a la señal analógica y dar a ésta una interpretación en una palabra de *n-bits*, de tal manera que sea adecuada a la entrada del bloque del procesamiento digital de señales. Dependiendo de la aplicación para la cual el sistema es requerido, el procesamiento digital de señales puede comprender la extracción, la interpretación o la supresión de información.

Para estas tareas existen diversas herramientas, tales como el diseño e implementación de filtros digitales de tipo FIR (por sus siglas en inglés, *Finite Impulse Response*) o tipo IIR (por sus siglas en inglés *Infinite Impulse Response*), el análisis en el dominio de la frecuencia de una señal por medio de la transformada rápida de Fourier, algoritmos para compresión de información, algoritmos para segmentación, filtros, convolución de imágenes, etc.

Una de las herramientas más importantes del procesamiento digital de señales es la transformada discreta de Fourier (DFT – *Discrete Fourier Transform*) debido a que mediante ella se puede obtener una representación de la señal con base en sus componentes de frecuencia.
Actualmente la DFT es utilizada para:

- Compresión de imágenes (formato JPEG)
- Compresión de Audio (formato MP3)
- Análisis espectral de señales
- Reducción de ruido en señales, como ruido blanco
- Análisis de la respuesta en frecuencia de los sistemas
- Convolución en el dominio de la frecuencia

- Correlación cruzada
- Multiplicación de números enteros grandes
- Multiplicación simbólica de polinomios

A pesar de ser una de las herramientas más utilizadas para el procesamiento digital de señales, ésta tiene una complejidad muy alta, dada por el número de multiplicaciones entre números complejos que se requieren realizar para su cálculo el cual está dado por Ec. 1.1.

$$X[m] = \sum_{n=0}^{N-1} n[n] e^{-j\frac{2\pi nm}{N}} para\ m = 0, 1, 2, \cdots\ N-1 \qquad \text{Ec. 1.1}$$

De la ecuación Ec. 1.1 se puede observar que la cantidad de productos y sumas complejas necesarios para el cálculo de la DFT de una señal discreta finita es creciente cuadráticamente. Es decir, el número de productos complejos es N^2 y el número de sumas complejas es $N(N-1)$ lo que inevitablemente aumenta la complejidad de su evaluación.

Es por esto que actualmente se utiliza el algoritmo de la transformada rápida de Fourier (FFT – *Fast Fourier Transform*), que simplifica el trabajo necesario para obtener el mismo resultado que la transformada discreta de Fourier, el algoritmo fue diseñado por J.W. Cooley y John Tukey en 1965 y permite hacer una optimización en tiempo y recursos del cálculo de la DFT.

El algoritmo de la transformada rápida de Fourier hace uso de la periodicidad del término exponencial complejo para reducir así el número de productos y sumas complejas necesarias para obtener el espectro de frecuencias de una señal. Esto deriva en un incremento en la velocidad de cómputo así como en la disminución del costo computacional.

En la actualidad se cuenta con una gran variedad de dispositivos hardware que permiten hacer la implementación de algoritmos de procesamiento digital de señales, como son los procesadores digitales de señales (DSP - *Digital Signal Processor*), los circuitos de aplicación específica (ASIC - *Application-Specific Integrated Circuit*) y los dispositivos lógicos programables (PLD -*Programmable Logic Device*) dentro de los que destacan los arreglos de compuertas de campos programables (FPGA – *Field Programmable Gate Array*), y que dependiendo de la aplicación a desarrollar es que se determina el dispositivo que mejor se adecue a las necesidades y requerimientos. En particular, los criterios de

procesamiento digital conducen a grandes requerimientos en términos de poder de procesamiento y tasa de acceso a memorias de datos [3].

Implementar el bloque de procesamiento en un circuito dedicado requiere la fabricación del dispositivo para realizar pruebas, esto resulta tener un costo elevado además de que cierra los márgenes de error del diseño, debido a que si el circuito no funciona como el usuario lo espera, se requiere realizar el proceso de diseño y fabricación nuevamente aunque tiene la ventaja de ofrecer un mejor desempeño. Al utilizar una computadora personal se tiene la flexibilidad de hacer cambios de manera sencilla pero con el sacrificio en el desempeño del procesamiento digital.

Los FPGA están a punto de revolucionar el procesamiento digital de señales de la misma manera en que los DSP lo hicieron hace casi 20 años [4]. Al considerar una alternativa para la implementación de una arquitectura que permita evaluar la transformada rápida de Fourier se deben tomar en cuenta aspectos relevantes como la velocidad de ejecución, la complejidad del hardware, la flexibilidad y precisión, ventajas que sin lugar a dudas nos brinda la implementación sobre un FPGA quienes han experimentado amplias innovaciones en los últimos años.

La tecnología ha permitido el desarrollo de dispositivos de alta densidad que están muy bien adaptados a las necesidades del procesamiento de señales en tiempo real de alto desempeño [5], por lo que un dispositivo de lógica programable como lo es un FPGA se convierte en una buena opción para la implementación de aplicaciones de este tipo.

Al finalizar el trabajo de procesamiento de la señal, dependiendo de la aplicación, es necesario añadir el bloque del convertidor digital/analógico, el cual establece una interfaz entre el resultado del procesamiento con el mundo exterior en forma de una señal analógica.

Este proyecto se enfoca al bloque identificado como procesador digital de señales el cuál es el encargado de hacer el trabajo de extracción o supresión de información de la señal a analizar, este bloque puede ser un circuito integrado dedicado a este fin o un hardware genérico como un procesador de propósito general o una computadora personal, ambas opciones tienen ventajas y desventajas como las que ya se mencionaron.

1.2 Planteamiento del problema

Como ya se ha mencionado, la importancia de la transformada rápida de Fourier radica en ser una herramienta ampliamente utilizada en el procesamiento digital de señales siendo de gran utilidad en áreas como comunicaciones, análisis de señales de voz etc.

La operación principal en la evaluación de la FFT es conocida como mariposa, como se detallará más adelante, la FFT aprovecha la periodicidad de los coeficientes complejos lo que ayuda a reducir la cantidad de productos complejos para la obtención del espectro de frecuencias. En el diseño de circuitos para la evaluación de la FFT se tienen dos enfoques que tienden a los extremos en cuanto al desempeño y al uso de recursos, el primer enfoque tiene como principio la optimización en área en el dispositivo donde va a ser implementado el diseño, lo que se ve reflejado en un mayor tiempo de ejecución ya que consiste en evaluar la FFT de manera secuencial, utilizando una sola unidad de procesamiento tal cual se haría con un DSP o una computadora personal y obteniendo como resultado una arquitectura *in – place* [18].

El segundo enfoque tiene como principio la optimización en el tiempo de ejecución, dado que evalúa la FFT por medio de la implementación de la totalidad de las unidades de procesamiento necesarias para una evaluación en paralelo de todas las mariposas, este enfoque resulta tener un costo elevado en recursos pero tiene el mejor desempeño al requerir el menor tiempo de ejecución.

Diversas propuestas en el estado del arte han optado por diseñar circuitos sobre el primer enfoque aportando en cada una de ellas alguna técnica para proponer un balance entre el tiempo de ejecución y el hardware utilizado, sin embargo, pocas son las propuestas de un diseño de arquitectura bajo un enfoque distinto.

Aunado a ello, existen otras opciones de arquitecturas diseñadas por las grandes empresas como son Xilinx y Altera, quienes proporcionan al usuario componentes de propiedad intelectual, lo que limita al usuario desde muchas perspectivas, una de ellas es el costo de las aplicaciones, por otra parte, los desarrollos de propiedad intelectual no ofrecen documentación más allá de los manuales de uso, esto deriva en un total desconocimiento de lo que se está utilizando y por lo tanto en un posible desajuste entre la aplicación para la que se requiere y la herramienta utilizada y debido a que estas arquitecturas son cerradas no se pueden implementar modificaciones.

1.3 Objetivos

1.3.1 Objetivo general

Diseñar e implementar un circuito que permita la evaluación de la transformada rápida de Fourier utilizando dispositivos de lógica programable y lenguajes de descripción de hardware tomando en cuenta un equilibrio entre el tiempo de ejecución y recursos hardware del FPGA utilizados.

1.3.2 Objetivos específicos

Del objetivo general se derivan los siguientes objetivos específicos:

- Proponer una arquitectura tomando en cuenta un equilibrio entre el tiempo de procesamiento y los recursos del FPGA para la evaluación de la FFT.
- Diseñar un circuito digital para la evaluación de la transformada rápida de Fourier utilizando un formato numérico de punto fijo con base en la arquitectura propuesta.
- Modelar y simular el diseño utilizando un lenguaje de descripción de hardware para hacer pruebas funcionales de la arquitectura propuesta.
- Utilizar y programar un dispositivo FPGA mediante un lenguaje de descripción de hardware para demostrar la obtención del espectro en frecuencia de la señal.
- Comparar el rendimiento del circuito obtenido con soluciones existentes para evaluar el funcionamiento de la arquitectura propuesta

1.4 Justificación

El contar con una arquitectura que permita la evaluación de la transformada rápida de Fourier puede ser la base de aplicaciones en telefonía celular, análisis de vibraciones, compresión de información entre otras, gracias a que puede ser implementada en un dispositivo lógico programable que tiene ventajas tales como poder reconfigurar el dispositivo el número de veces que sea necesario. La transformada discreta de Fourier juega un rol importante en el dominio del procesamiento digital de señales, y ha sido implementada en diversas áreas de las comunicaciones digitales como son redes inalámbricas, transmisión de audio/video digital, entre otras [6] en donde conviene contar con un dispositivo que calcule la FFT en tiempo real sin un gran consumo de recursos.

La introducción de los dispositivos FPGA hace aproximadamente 20 años ha dado lugar al nacimiento de lo que se conoce como cómputo reconfigurable [12]. La estructura interna de los FPGA los convierte en dispositivos perfectamente adecuados para realizar en paralelo tareas elementales del procesamiento digital de señales. Un punto clave en la tecnología FPGA es que cuenta con potentes herramientas de diseño para acortar el ciclo de implementación, proporcionar una buena utilización del dispositivo y opciones de síntesis [12]. Esta es una de las razones por las cuales el procesamiento digital de señales ha tenido un gran auge en los últimos años, sin olvidar que un FPGA permite la fácil paralelización de algoritmos que en algún otro dispositivo pocas veces se conseguiría, por ejemplo, la implementación clásica de la FFT en un procesador digital de señales requiere de un algoritmo secuencial, lo que resta velocidad de ejecución [7]. La FFT tiene la naturaleza de ser paralelizable, por lo que su implementación en un FPGA generaría grandes ventajas.

Finalmente, es conveniente contar con un diseño de arquitectura general propio para la evaluación de la FFT que permita la comprensión del algoritmo así como el diseño e implementación de la arquitectura, lo que permitiría la implementación de sistemas de procesamiento digital de señales para el filtrado o compresión de información entre otras tareas.

1.5 Alcances y límites

El trabajo está limitado a:

- El diseño y la implementación de la arquitectura se hace con base en la tarjeta de desarrollo con la que se cuenta en el laboratorio, la tarjeta *Altera DE-2*.
- Como herramienta de desarrollo se cuenta con *Quartus II* versión web para la síntesis y compilación de los proyectos y *ModelSim Altera* como herramienta de simulación.
- Aunque se hace el diseño de cada uno de los módulos de la arquitectura, se toman algunos elementos de hardware dedicado del FPGA de la tarjeta como son elementos de memoria y operadores aritméticos con el fin de obtener mejores resultados.
- Este trabajo se enfoca únicamente a la arquitectura que evalúa la FFT, cualquier otro módulo que se requiere no es implementado, se utilizan módulos ya existentes dependiendo de la tarea.
- Al finalizar el presente trabajo, se tiene el diseño y la implementación de una arquitectura capaz de hacer el cálculo de la transformada rápida de Fourier en la tarjeta de desarrollo de *Altera DE2* que cuenta con un FPGA de la familia Cyclone-2.

Capítulo 2. Marco teórico y estado del Arte

2.1 Transformada rápida de Fourier

El algoritmo de la transformada rápida de Fourier es una herramienta que permite obtener el espectro de frecuencia de una señal con una reducción en tiempo de cálculo y recursos, lo que le brinda gran relevancia dentro del procesamiento digital de señales. Partiendo de la transformada discreta de Fourier que se define en la ecuación Ec. 1.1 en donde los coeficientes complejos están definidos por Ec. 2.1.

$$W = e^{-2\pi/N} \qquad \text{Ec. 2.1}$$

El cálculo de una componente del espectro de frecuencia finalmente requiere una secuencia tal que $X(k) = x[0]W^0 + x[1]W^k + x[2]W^{2k} + \cdots + x[N-1]W^{k(N-1)}$, $k = 0,1,\dots,N-1$, esto repercute seriamente en el tiempo de ejecución así como en la cantidad de recursos a utilizar, por ejemplo, al calcular el espectro de frecuencia de una señal de ocho puntos se requieren un total de ocho multiplicaciones entre números complejos y de siete sumas para el cálculo de un solo punto, de manera general, en una etapa completa se requieren de N multiplicaciones entre números complejos y $(N-1)$ sumas complejas, dando así un total de N^2 multiplicaciones y $N(N-1)$ sumas para el espectro completo.

A sabiendas de que, de acuerdo con [1] esta herramienta se utiliza para filtrado lineal, filtrado de secuencias de larga duración, cálculo de convoluciónes, análisis de correlación y el análisis espectral entre otras aplicaciones, se puede observar que la complejidad del cálculo está dada por la gran cantidad de operaciones necesarias y que crece rápidamente en función de la cantidad de puntos a evaluar por lo que poco a poco se convierte en una herramienta poco óptima para la implementación en hardware o software, un ejemplo claro es el cálculo de la DFT de 1024 puntos en donde se evalúan aproximadamente un millón de multiplicaciones complejas convirtiéndose así, en la medida en como aumentan los puntos, en un problema intratable.

En [23] se define el algoritmo de la FFT como un método para el cálculo de la transformada de Fourier en series finitas de N números complejos en aproximadamente $Nlog_2N$ operaciones, esta disminución es debida a que se requiere mucho menor esfuerzo para cálculo ya que se reduce el número de operaciones necesarias mediante la división jerárquica en transformadas de secuencias más pequeñas [3]. El método descrito en [22] por Cooley y Tukey aborda la evaluación de la DFT por medio de un enfoque divide y vencerás, el cual consiste en hacer la descomposición de una DFT de N puntos en DFTs de menor tamaño considerando a N como un número compuesto que se puede representar con base al producto de dos

enteros [1] tal que se cumpla que $N = LM$ de tal manera que ahora los elementos de N se pueden trabajar ya no en un arreglo unidimensional, si no en una matriz indexada por l y m, donde l es índice de las filas y m de las columnas cumpliendo como condiciones que $0 \leq l \leq L-1$ y $0 \leq m \leq M-1$ quedando representados de tal manera que $x(n) = x(l,m)$ y de forma similar se trata a los elementos de la DFT de N puntos mapeados en una matriz de la forma $X(k) = X(p,q)$, si se elige una correspondencia dado que $n = l + mL$ para almacenar a los primeros L elementos en la primer columna y una correspondencia para el resultado de la DFT como $k = Mp + q$ para almacenar en la primer fila los primeros M elementos, a partir de esto, la DFT se puede expresar como una doble sumatoria de los productos entre los elementos de la matriz y los coeficientes complejos correspondientes como se muestra en la Ec. 2.3, donde se tiene que

$$X(p,q) = \sum_{m=0}^{M-1}\sum_{l=0}^{L-1} x(l,m) W_N^{(Mp+q)(mL+l)} \qquad \text{Ec. 2.2}$$

y desarrollando se tiene que

$$W_N^{(Mp+q)(mL+l)} = W_N^{MpmL} W_N^{Mpl} W_N^{qmL} W_N^{ql}$$

$$W_N^{Nmp} = 1,\ W_N^{mqL} = W_{N/L}^{mq} = W_M^{mq} \text{ y } W_N^{Mpl} = W_L^{pl}$$

expresando finalmente la doble sumatoria como en la Ec. 2.3

$$X(p,q) = \sum_{l=0}^{L-1}\left\{ W_N^{lq}\left[\sum_{m=0}^{M-1} x(l,m)\, W_M^{mq}\right]\right\} W_L^{lp} \qquad \text{Ec. 2.3}$$

Se observa que el cálculo de la DFT de N puntos implica hacer el cálculo de la DFT de M y de L resumiendo esto en tres pasos

- Cálculo de las DFTs de M puntos lo que tiene una complejidad de LM^2 multiplicaciones complejas y $LM(M-1)$ sumas complejas
- Cálculo de la nueva matriz, dada por el producto entre el resultado del paso anterior y los coeficientes complejos, esto tiene una complejidad de LM multiplicaciones complejas
- Cálculo de las DFTs de L puntos que tiene una complejidad de ML^2 multiplicaciones complejas y $ML(L-1)$ sumas complejas

Obteniendo finalmente una complejidad computacional de

- Multiplicaciones complejas: $N(M+L+1)$
- Sumas complejas: $N(M+L-2)$

Como se observa, este método redujo en gran medida la cantidad de operaciones necesarias para hacer la evaluación completa de la DFT y la mejora se puede incrementar si *N* se trata de un número compuesto que se puede expresar como $N = r_1 r_2 \cdots r_v$ ya que la descomposición anterior puede repetirse $(v-1)$ veces dando como resultado la evaluación de DFTs mas pequeñas y a su vez reduciendo más el número de operaciones requeridas.

La transformada rápida de Fourier aprovecha la periodicidad de los coeficientes complejos de la transformada discreta de Fourier y toma ventaja del hecho de que el cálculo de los coeficientes de la DFT se puede llevar a cabo iterativamente, lo que ahorra considerablemente el tiempo de cálculo. Desde la publicación de Cooley-Tukey en 1965 [22], este algoritmo ha generado cambios en las técnicas de cómputo utilizadas en el análisis espectral digital, simulación de filtros, entre otros campos relacionados es por ello que se han realizado diversas propuestas y modificaciones de este algoritmo en búsqueda de obtener un mejor resultado.

2.1.1 Algoritmo de la FFT Radix-2

El algoritmo de la FFT Radix-2 es una especificación de la propuesta hecha en [22], en donde se utiliza un valor de *N* que puede factorizarse como $N = r_1 r_2 \cdots r_v$ y $r_1 = r_2 = \cdots = r_v \equiv r$ de manera que $N = r^v$, en este caso, las DFTs resultantes de las divisiones tienen un tamaño de *r* por tal motivo, a *r* se le denomina base del algoritmo.

Para el caso específico de Radix-2, se tiene que $r = 2$ y se hace el cálculo de la DFT de $N = 2^v$ utilizando la propuesta de Cooley-Tukey implementando la técnica de divide y vencerás, para este caso en específico se escogen a *M=N/2* y a *L=2,* mediante esta selección se permite descomponer a la secuencia de *N* datos en dos mitades iguales, una para números pares y la otra para números impares de *x(n)* respectivamente y denotadas por $f_1(n)$ y $f_2(n)$, tal que

$$f_1(n) = x(2n) \qquad \text{Ec. 2.4}$$

$$f_2(n) = x(2n+1) \qquad n = 0, 1, \cdots, \frac{N}{2} - 1 \qquad \text{Ec. 2.5}$$

Si se expresa a la DFT definida en Ec. 1.1 en términos de $f_1(n)$ y $f_2(n)$ se obtiene

$$X(k) = \sum_{m=0}^{(N/2)-1} x(2m) W_N^{2mk} + \sum_{m=0}^{(N/2)-1} x(2m+1) W_N^{k(2m+1)} \qquad \text{Ec. 2.6}$$

y sabiendo que $W_N^2 = W_{N/2}$ se sustituye en Ec. 2.6 para obtener

$$X(k) = \sum_{m=0}^{(N/2)-1} f_1(m)W_{N/2}^{mk} + W_N^k \sum_{m=0}^{(N/2)-1} f_2(m)W_{N/2}^{km} \qquad \text{Ec. 2.7}$$

obteniendo finalmente una expresión como la siguiente en donde se evalúan las dos DFTs de *N/2* puntos de las secuencias iniciales de números pares e impares respectivamente

$$X(k) = F_1 + W_N^k F_2 \qquad k = 0, 1, \cdots, N-1$$

ambas funciones son periódicas en $N/2$ puntos y $W_N^{k+N/2} = -W_N^k$ por lo que finalmente se obtienen las operaciones de Ec. 2.8 y Ec. 2.9

$$X(k) = F_1 + W_N^k F_2 \qquad k = 0, 1, \cdots, \frac{N}{2} - 1 \qquad \text{Ec. 2.8}$$

$$X\left(k + \frac{N}{2}\right) = F_1 - W_N^k F_2 \qquad k = 0, 1, \cdots, \frac{N}{2} - 1 \qquad \text{Ec. 2.9}$$

Con las operaciones definidas en Ec. 2.8 y Ec. 2.9, la decimación de la entrada se puede repetir varias veces hasta que las secuencias finales sean de dos puntos, en el caso del algoritmo *Radix-2* donde $N = 2^v$ se puede hacer *v* veces obteniendo finalmente $(N/2)log_2 N$ multiplicaciones complejas y $Nlog_2 N$ sumas complejas, observando que estas cantidades crecen menos rápido en comparación con las que se requieren en la DFT.

Las ecuaciones anteriores son representadas por la operación básica denominada mariposa, que se define como se muestra en la Fig. 2.1.

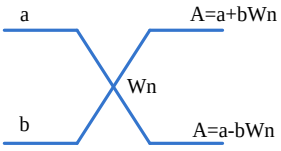


Fig. 2.1 Mariposa para algoritmo FFT Radix-2.

El flujo completo de la evaluación del algoritmo y la interacción entre el operador mariposa se pueden observar en la Fig. 2.2.

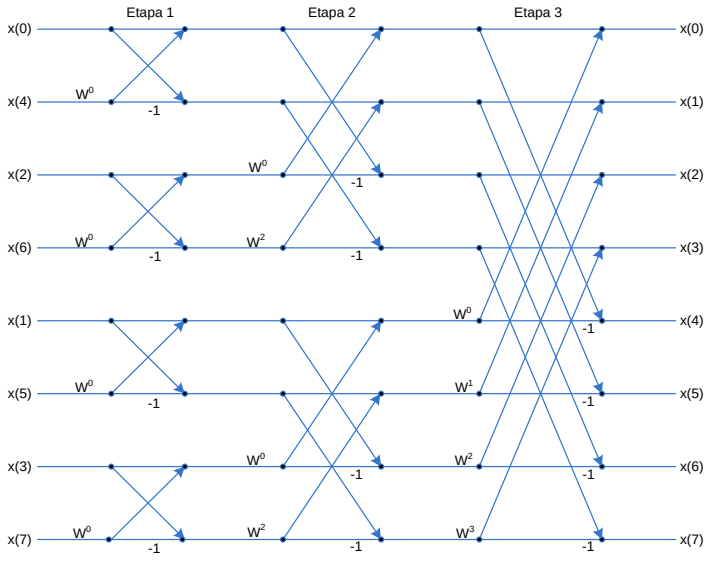


Fig. 2.2 Algoritmo de la FFT Radix-2 [1].

Como se observa en la Fig. 2.2, el orden de las entradas de los datos no es secuencial, esto es debido a la decimación hecha *(v-1)* veces de la entrada, este orden es característico del algoritmo con decimación en tiempo y está bien definido tal que se puede determinar con facilidad mediante la técnica de reversión de bits (*bit-reverse*) que consiste en representar en orden inverso la secuencia de bits que equivalen a un número tal como se muestra en la Tabla 2.1.

Tabla 2.1 Reversión de bits para N = 8 puntos.

Decimal	**Binario ($n_0n_1n_2$)**	**Reversión de bits ($n_2n_1n_0$)**	**Decimal**
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Una variante del algoritmo descrito es el algoritmo de decimación en frecuencia en donde los factores *M* y *L* se eligen al revés, permitiendo así tener una entrada de las muestras en orden pero una salida con reversión de bits siendo esta la principal diferencia entre ambas propuestas.

Con el desarrollo del algoritmo de la FFT se obtuvo una disminución con relación a la cantidad de operaciones necesarias para la obtención del espectro, a continuación, en la Tabla 2.2 se muestra una comparativa sobre los resultados que se han obtenido con la aplicación del algoritmo de la FFT.

Tabla 2.2 Comparativa de complejidad computacional entre DFT y FFT [1].

Número de puntos *N*	**Multiplicaciones complejas con la evaluación del a DFT**	**Multiplicaciones complejas con la evaluación de la FFT**	**Factor de mejora de la velocidad**
4	16	4	4.0
8	64	12	5.3
16	256	32	8.0
32	1024	80	12.8
64	4096	192	21.3
128	16384	448	36.6
256	65536	1024	64.0
512	262114	2304	113.8
1024	1048576	5120	204.8

Dadas las propiedades de la descripción general de divide y vencerás, este algoritmo tiene muchas variantes de acuerdo con los factores de *N*, todos con base en los mismos principios, sólo modificando la base del algoritmo, en la Tabla 2.3 se muestra una comparación entre los diversos algoritmos existentes.

Tabla 2.3 Número de multiplicaciones y sumas reales no triviales para el cálculo de la DFT de N puntos compleja [24].

N	**Multiplicaciones Reales**				**Sumas Reales**			
	Base 2	Base 4	Base 8	Base Partida	Base 2	Base 4	Base 8	Base Partida
16	24	20		20	125	148		148
32	88			68	408			388
64	264	208	204	196	1032	976	972	964
128	712			516	2504			2308
256	1800	1392		1284	5896	5488		5380
512	4360		3204	3076	13566		12420	12292
1024	10248	7856		7172	30728	28336		27652

La reducción en la cantidad de operaciones se hace una constante, como se observa, entre más grande sea la base del algoritmo se obtienen mejores resultados en la optimización de operaciones, sin embargo se agrega complejidad a las evaluaciones de las mariposas.

2.2 Formato de punto fijo para la representación de números fraccionarios

Dada la naturaleza del algoritmo de la FFT así como de los datos que se utilizan como coeficientes complejos, las operaciones que se realizan se hacen tomando en cuenta que se tienen operandos fraccionarios por lo que es de gran importancia contar con un sistema de representación de éstos. La representación de números fraccionarios se puede realizar utilizando formato de punto fijo o formato de punto flotante.

La representación de los números en cualquiera de los formatos obedece a las necesidades del usuario, actualmente el formato de punto flotante está especificado en el estándar IEEE 754 y entre sus ventajas están que se puede representar un rango más grande de números y se cuenta con mayor precisión al hacer operaciones aritméticas, sin embargo el manejo de este formato es más demandante en cuanto a recursos de almacenamiento de datos aunque en la actualidad es el formato más utilizado.

Por otro lado, la representación de números fraccionarios con el formato de punto fijo se utiliza para representar números fraccionarios con base 2 dentro de los sistemas digitales especialmente cuando no se tiene una unidad de punto flotante en específico siendo éste el caso, en el que el FPGA no cuenta con una unidad especial para realizar operaciones de punto flotante, se utiliza la representación de punto fijo, la cual es adecuada para procesamiento de señales de ovoz, audio en general, manejo de señales de control y algoritmos que no requieren de una alta precisión.

La característica principal de los números en punto fijo es que la cantidad de cifras que representan la parte entera y la parte decimal es siempre la misma, en la Fig. 2.3 se observa como es la representación del formato de punto fijo en un vector de bits.

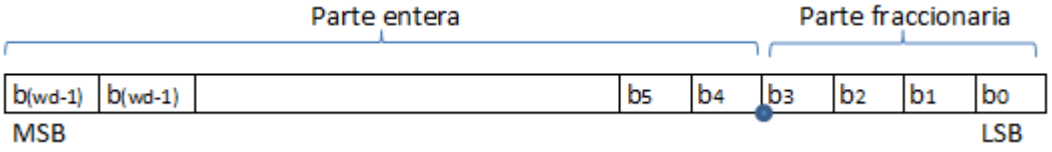


Fig. 2.3 Representación de números en formato de punto fijo

En donde *wd* es el ancho de palabra del número y llamaremos *pt* a la posición en donde se encuentra el punto decimal dentro del vector de bits. Con estos dos datos se pueden definir los siguientes conceptos.

- Precisión: Hace referencia al número total de bits que se tienen para la representación (*wd*)

- Resolución: Es el número más pequeño (que no sea cero) que se puede representar y está dado por

$$Resolución = \frac{1}{2^{pt}} \qquad \text{Ec. 2.10}$$

- Rango: Proporciona el intervalo de números que se pueden representar con un formato determinado, está dado por

$$Rango_{UFix} = \left[0, \frac{2^{wd} - 1}{2^{pt}}\right] \qquad \text{Ec. 2.11}$$

$$Rango_{Fix} = \left[-\frac{2^{wd-1}}{2^{pt}}, \frac{(2^{wd-1}) - 1}{2^{pt}}\right] \qquad \text{Ec. 2.12}$$

Para este trabajo se utiliza el formato de punto fijo para números con signo conocido como Q15, el cual se caracteriza por tener un tamaño de palabra *wd*=16 y un *pt*=15, por lo que de acuerdo con estos datos y las Ec. 2.10 y Ec. 2.12 se sabe que

$$Resolución = \frac{1}{2^{pt}} = \frac{1}{2^{15}} \cong .00003051$$

$$Rango_{Fix} = \left[-\frac{2^{wd-1}}{2^{pt}}, \frac{(2^{wd-1}) - 1}{2^{pt}}\right] = [-1, .9999]$$

Como se observa, este formato permite representar números negativos desde -1 y números positivos hasta .9999 con una resolución de hasta .00003051 lo que hace suficientemente grande el rango de valores para poder representar los coeficientes complejos requeridos para la evaluación del algoritmo, que son básicamente valores de las funciones seno y coseno.

La representación en formato de punto fijo implica la posibilidad de presentar problemas como sobre flujo y cuantización debido a que las operaciones realizadas pueden presentar resultados que requieran de una mayor cantidad de cifras por lo que para su representación es posible la pérdida de información, es por eso que en estos casos se recurre a técnicas como el truncamiento o el redondeo. En operaciones como la multiplicación, estas técnicas pueden generar una pérdida de precisión por lo que la técnica escogida depende de la aplicación que se esté realizando.

2.3 Cómputo reconfigurable para el Procesamiento Digital de Señales

Existe una amplia gama de dispositivos en los cuáles se pueden evaluar e implementar algoritmos para el procesamiento digital de señales, en los últimos años, la tendencia a implementar sistemas PDS en hardware de cómputo reconfigurable ha crecido considerablemente por las ventajas que presenta y esto ha llevado a la comunidad académica y comercial a hacer nuevos esfuerzos por optimizar energía, reducir costos y mejorar el rendimiento en tiempo y ejecución de los dispositivos reconfigurables.

El procesamiento de señales como voz, audio y video normalmente requiere de cálculos y evaluaciones de algoritmos en tiempo real y dado el inherente paralelismo que se encuentra en los algoritmos PDS los dispositivos reconfigurables se convierten en candidatos para la implementación hardware de estos [8].

El hardware reconfigurable ofrece un punto intermedio entre las ventajas de desempeño de un hardware con una función en específico y la flexibilidad de implementación tal y como se observa en la Fig. 2.4, ya que su funcionalidad lógica e interconexión pueden ser modificadas con el objetivo de adaptarse a la aplicación definida por el usuario, además de contar con recursos para fines específicos, como hardware aritmético, que pueden ser modificados fácilmente y con esto permitir el diseño a diversos parámetros de operación y conjuntos de datos diferentes. El cómputo reconfigurable para el procesamiento digital de señales mantiene activa un área de investigación como una necesidad para la integración con tecnologías más tradicionales con este fin como los DPS [8].

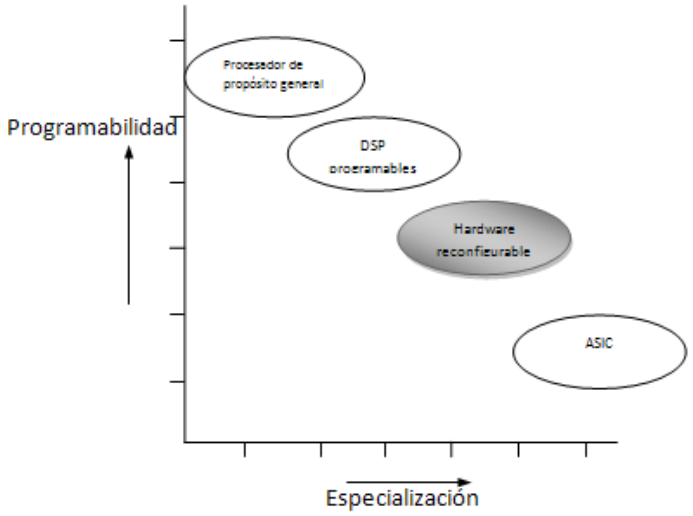


Fig. 2.4 Espectro de implementación de algoritmos de Procesamiento Digital de Señales [8].

La implementación de algoritmos PDS en hardware tiene principalmente cuatro opciones para su desarrollo que son; los circuitos integrados de aplicación específica ASIC, los procesadores digitales de

señales DSP, los procesadores de propósito general y el hardware reconfigurable. Tomando en cuenta que las aplicaciones PDS han estado enfocadas en alcanzar objetivos como el paralelismo de datos, la especialización de una aplicación en específico y la flexibilidad de la funcionalidad, el escoger un dispositivo de los antes mencionados requiere encontrar una compensación entre estos tres objetivos ya que no hay uno que cumpla con los tres al mismo tiempo, en la Tabla 2.4 se muestra una comparativa entre los cuatros dispositivos de implementación.

Tabla 2.4 Comparación de hardware para implementación de Procesamiento Digital de Señales.

	Desempeño	**Costo**	**Potencia**	**Flexibilidad**	**Esfuerzo para diseño**
ASIC	Alto	Alto	Baja	Baja	Alto
DSP	Medio	Medio	Media	Media	Medio
Procesador de propósito general	Bajo	Bajo	Media	Alta	Bajo
Hardware reconfigurable	Medio	Medio	Alta	Alta	Medio

El dispositivo ASIC brinda la oportunidad de contar con un circuito de alto desempeño, sin embargo, si se está trabajando sobre una fase de pruebas, la fabricación del mismo tiene un costo elevado además de requerir un mayor esfuerzo en el diseño, el otro extremo al ASIC es el procesador de propósito general que nos permite hacer una implementación de bajo desempeño a un costo reducido y brindándonos un alto nivel de flexibilidad. Hablando del DSP, aunque son dispositivos que fueron diseñados pensando en que pudieran cumplir con los requerimientos del procesamiento digital de señales, siguen teniendo como desventaja no explotar el paralelismo de los algoritmos y tener una flexibilidad media lo que coloca al hardware reconfigurable en ventaja con estos tres dispositivos mayormente por ser altamente flexible para el diseño y permitiendo la implementación de modificaciones rápidamente sin un costo excesivo en el esfuerzo de diseño.

Las aplicaciones PDS han colaborado con el desarrollo del hardware reconfigurable ya que han servido como casos de prueba y evaluación, esto lo ha mantenido en constante evolución, al grado de desarrollar hardware dedicado como unidades aritméticas para multiplicación y sumas eficientes, por otro lado, los beneficios directos que ofrece el hardware reconfigurable al procesamiento digital de señales pueden ser divididos en tres áreas: especialización funcional, reconfigurabilidad y paralelismo *fine-grained* [8].

- Especialización funcional: Esta característica permite hacer la especificación de valores de uso constante directamente en hardware, lo que reduce el área de implementación y por ende la potencia consumida, esto es importante en aplicaciones como son los filtros digitales.

- Reconfigurabilidad: Los sistemas PDS tienen una necesidad de reconfiguración sobre diversas restricciones como son los factores ambientales tales como los cambios en las estadísticas de ruido y señales, canal de transmisión, clima, velocidades de transmisión y estándares de comunicación.
- Paralelismo *fine-grained*: Esta característica permite hacer la implementación de un componente complejo a partir de componentes más sencillos, y es de gran utilidad para la reconfigurabilidad del sistema.

Dado que el cómputo reconfigurable en el procesamiento digital de señales es un área de investigación en desarrollo, su futuro está determinado por los mismos parámetros por los que el PDS evoluciona que son la integración de los sistemas, el cómputo reconfigurable dinámico y la compilación de alto nivel. La utilización del cómputo reconfigurable en sistemas PDS promete tener un gran auge y futuro [8].

2.4 Arquitecturas para la evaluación de la FFT con base en cómputo reconfigurable

Como se mencionó en el apartado anterior, el cómputo reconfigurable se ha convertido en una opción con muchas ventajas para la implementación de sistemas PDS, aplicaciones como la codificación de voz y procesamiento de señales de sonar entre otras requieren de un inherente procesamiento en paralelo. Una de las primeras aplicaciones en hardware de la FFT fue [10], dicha publicación señaló que la reducción de costo resultante de un hardware de propósito especial era cercanamente buena como la reducción de costo que vino con el desarrollo del algoritmo de la FFT por Cooley-Tukey en 1965.

Con la aparición de la FFT, la reducción del costo en cómputo para la obtención de la DFT se vio reflejada en la optimización de muchas aplicaciones a nivel software, a nivel hardware se obtiene una mayor ganancia con un hardware especial. Con el fin de hacer una buena implementación, se recomienda tomar en cuenta los siguientes aspectos [9].

- La razón de desarrollo del hardware
- Las opciones para su desarrollo
- El equilibrio que se debe alcanzar

En [9] se presenta una clasificación de las arquitecturas para la evaluación de la FFT en función de las unidades de cálculo utilizadas, en donde se refieren a la unidad de cálculo de la mariposa como la operación básica, sin importar qué base se utilice para el algoritmo. Considerando que se requiere calcular una FFT de ocho puntos con el algoritmo *Radix 2*, como la que se muestra en la Fig. 2.5, se puede describir la clasificación de las arquitecturas de la siguiente manera.

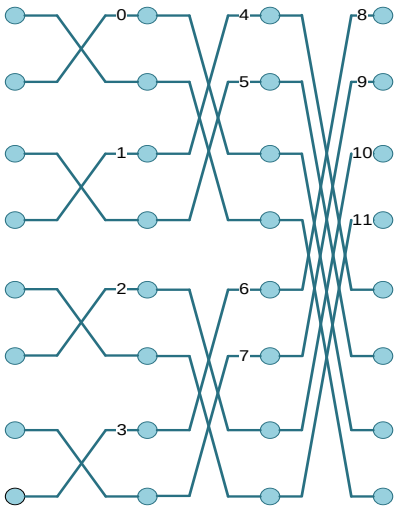


Fig. 2.5 Evaluación de la FFT para N=8.

2.4.1 Procesador secuencial

Esta arquitectura esta propuesta con base en la utilización de una sola unidad aritmética para la evaluación secuencial de la FFT completa, utilizando una memoria de *N* palabras para almacenar los datos resultantes de la iteración actual y que en la siguiente etapa serán utilizados así como se muestra en la Fig. 2.6. Este esquema de arquitectura tiene la ventaja de utilizar pocos recursos en el dispositivo, y a su vez se asemeja a la forma de evaluación de un procesador de propósito general o un DSP, sin embargo la ejecución de la FFT sobre una arquitectura de este tipo le resta velocidad a la evaluación. En [11] es identificado como una arquitectura de memoria simple o *in-place*.

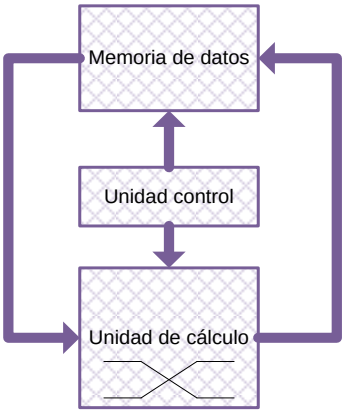


Fig. 2.6 Esquema de procesamiento secuencial.

El funcionamiento de esta arquitectura comprende el almacenamiento de los datos a procesar, la lectura de datos de la memoria, *N*/2 ejecuciones de mariposa por etapa, y la escritura de datos en la misma memoria para comenzar de nuevo hasta ejecutar la totalidad de las operaciones requeridas para la evaluación completa. Esta arquitectura se caracteriza por:

- Utilizar una unidad aritmética
- Ejecutar $(\mathrm{N}/2)\log_2\mathrm{N}$ de forma secuencial para terminar la evaluación
- Tener un tiempo aproximado de ejecución de $B(N/2)log_2N$ en donde B es el tiempo de ejecución de una mariposa.

Una variante de esta arquitectura es la que se menciona en [11] con doble memoria de datos (*dual memory)* y consiste en una sola unidad de procesamiento pero a diferencia del esquema secuencial o de memoria simple, el manejo de datos se hace entre dos memorias, como se observa en la Fig. 2.7, una memoria para lectura de datos y otra para escritura, estas memorias intercambian su función por cada etapa que se evalúa hasta que se completa la FFT.

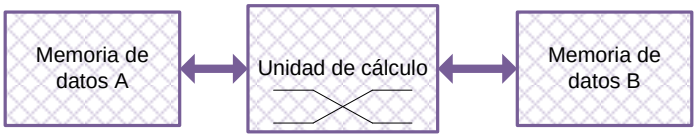


Fig. 2.7 Esquema de procesamiento de memoria doble.

La mayoría de las propuestas reportadas en el estado del arte implementan un esquema de arquitectura como éste, enfatizando en el poco consumo de recursos y poca energía utilizada, algunos ejemplos son:

- *High-Performance FFT Processing Using Reconfigurable Logic* [12]. En donde se propone una arquitectura utilizando dos unidades aritméticas para el cálculo completo de la FFT *Radix-4*, diferenciando entre la unidad que requiere hacer operaciones con coeficientes complejos y la unidad que utiliza coeficientes triviales. Es implementada en un FPGA de la familia Virtex II utilizando hardware dedicado como memorias y multiplicadores.

El almacenamiento de los datos se realiza bajo un esquema *non-in-place*, lo que significa que utilizan una memoria para lectura y otra para escritura de los datos, y con la siguiente etapa, el rol de las memorias se invierte, alcanzando una frecuencia de operación de 100MHz para un tamaño de N = 1024. En la Fig. 2.8 se muestra la arquitectura propuesta, en específico, 2593 *slices*, 12 multiplicadores y 22 bloques de RAM.

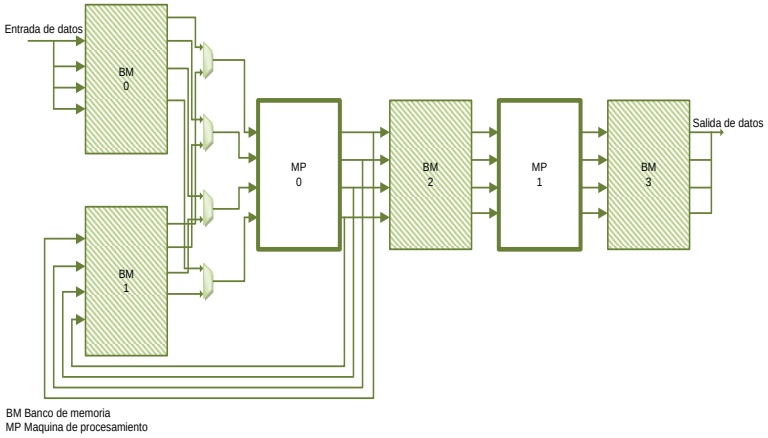


Fig. 2.8 Arquitectura propuesta en [12].

- *Design of a High Performance FFT Processor Based on FPGA* [14]. Esta arquitectura evalúa el algoritmo de la FFT Radix-4 utilizando un esquema secuencial o de memoria simple, el direccionamiento de los datos se hace sobre cuatro bancos de memorias de doble puerto, ya que se implementa un esquema de memoria *in-place,* se tienen cuatro memorias de doble puerto de donde la unidad aritmética toma los datos para procesar la operación. Adicionalmente se utiliza una memoria ROM de donde se toma el valor

de los coeficientes complejos. Esta propuesta fue implementada en un FPGA de la familia Virtex II de Xilinx, alcanzando una frecuencia de operación de 127Mhz y evaluando una FFT con N=1024 en 10.1µs, el diagrama de bloques de esta arquitectura se muestra en la Fig. 2.9.

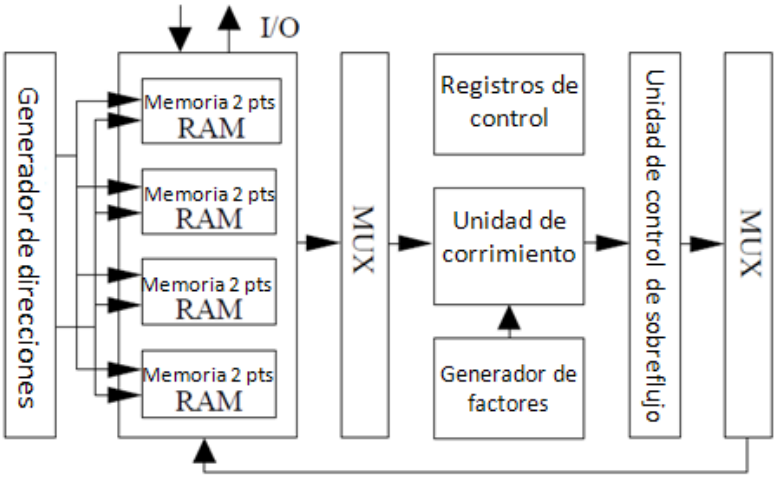


Fig. 2.9 Arquitectura propuesta en [14].

2.4.2 Procesador en cascada

Con la finalidad de aprovechar el paralelismo del algoritmo de la FFT y así incrementar el rendimiento del procesador, este esquema de arquitectura propone utilizar una unidad aritmética ejecutándose por separado al mismo tiempo en cada etapa. En esta propuesta no se utilizan memorias de N palabras debido a que el flujo de datos entre etapas es continuo para no mantener en ocio a las unidades aritméticas, el diagrama de bloques de esta arquitectura se muestra en la Fig. 2.10.

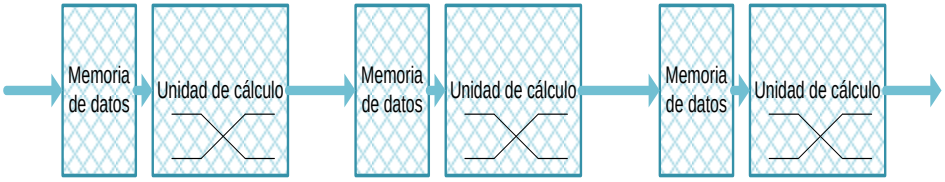


Fig. 2.10 Esquema de procesamiento en cascada.

De acuerdo con [9] la implementación de la FFT con esta arquitectura puede incrementar el rendimiento en un factor de $log_2 N$, el esquema de funcionamiento utiliza a la unidad aritmética de la etapa correspondiente para evaluar todas las operaciones de esa etapa, utilizando el diagrama de Fig. 2.10, la primera unidad evaluará las mariposas de la cero a la tres, la segunda de la cuatro a la siete y la última evaluará de la ocho a la once, este esquema tiende a disminuir el tiempo de ejecución por la razón de

utilizar varias unidades al mismo tiempo. Esta arquitectura en [11] es llamada *pipeline* y tiene como características:

- Utilizar m unidades aritméticas, donde $m = log_2 N$
- Ejecutar m operaciones en paralelo
- Ejecutar (N/2) operaciones secuenciales en cada etapa

De acuerdo con [16], las arquitecturas *pipeline* brindan un alto desempeño y un circuito de control simple, y para sistemas de alta velocidad, tales como radar de alto rendimiento, una arquitectura en cascada o *pipeline* es requerida [18]. A continuación se muestran algunas propuestas presentes en el estado del arte.

- *Radix-2 Multi-path Delay Commutator [18]:* Esta arquitectura *pipeline*, probablemente ha sido el enfoque más implementado para el algoritmo *Radix-2* de la FFT con decimación en frecuencia, consiste en implementar una unidad de procesamiento por cada etapa y dividir la trama de datos en dos tramas en paralelo. Se reporta que las unidades aritméticas trabajan al 50% de su capacidad y utiliza un esquema *in-place* para almacenar los datos en memoria. El diagrama de bloques se muestra en la Fig. 2.11.

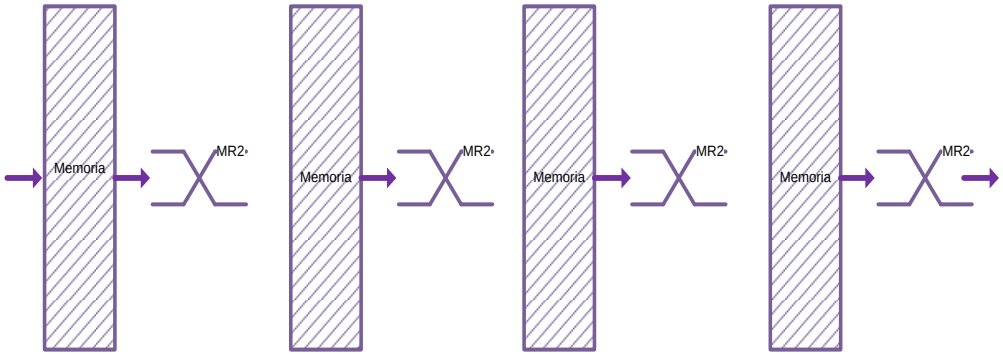


Fig. 2.11 Arquitectura propuesta en [18].

- *Efficient FPGA implementation of FFT/IFFT Processor [17].* Se propone una arquitectura para la evaluación del algoritmo *Radix-2*^{2} de la FFT con decimación en frecuencia (Fig. 2.12), esta fue implementada en un dispositivo FPGA 5VSX35T Xilinx de la familia Virtex 5. La arquitectura consta de $log_4 N$ etapas, y cada etapa está conformada por dos mariposas *Radix-2*, un registro de retroalimentación, una memoria ROM de donde se leen los coeficientes complejos y una unidad de control, esta propuesta procesa 256 muestras en .135µs a una frecuencia máxima de operación de 465 MHz utilizando 1058 LUTs de los cuales 256 fueron utilizados para memoria y 44 unidades DSP propias del dispositivo en el que se implementó.

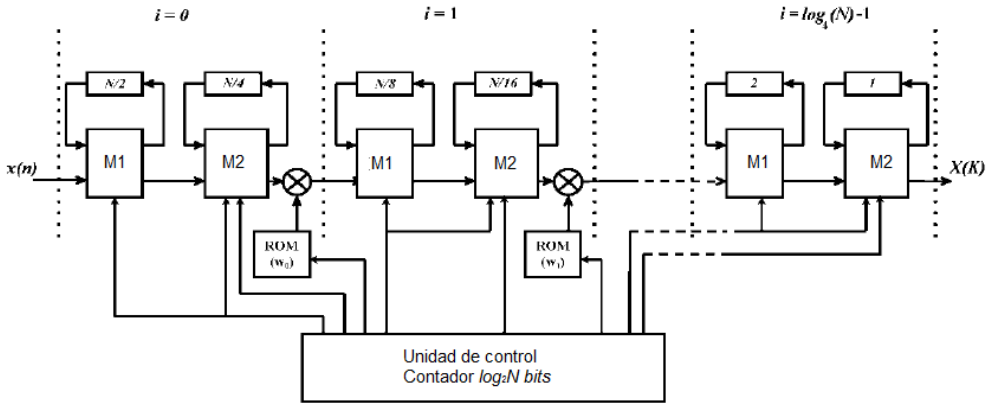


Fig. 2.12 Arquitectura propuesta en [17].

- *FPGA Implementation of a Re-Configurable FFT for Multi-standard Systems in Software Radio Context [19]*. Esta arquitectura *pipeline* implementada en un dispositivo FPGA es capaz de evaluar el algoritmo *Radix-2* decimación en tiempo de la FFT y se descompone en $log_2 N$ etapas, cada una de ellas compuesta por una mariposa *Radix-2* reconfigurable para evaluar muestras en el espacio de números imaginarios y muestras en el espacio de Galois, un banco de memoria de 3 puertos para la lectura y escritura de los datos, una memoria ROM de donde se leen los coeficientes complejos y un módulo de control por etapa y un control para toda la arquitectura. De esta propuesta se puede destacar la reconfigurabilidad de la mariposa. A continuación se muestra un diagrama de esta propuesta en la Fig. 2.13.

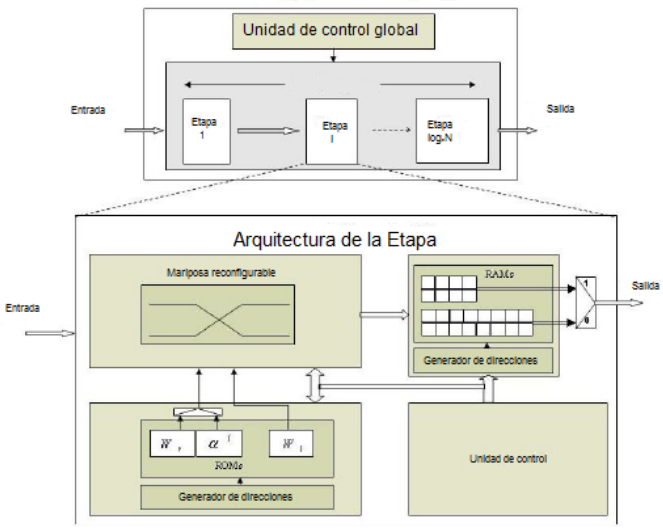


Fig. 2.13 Arquitectura propuesta en [19].

2.4.3 Procesador paralelo-iterativo

Esta arquitectura es otro enfoque de implementación de paralelismo, ya que se basa en la propuesta de utilizar varias unidades aritméticas por cada etapa de la evaluación de la FFT y hacer iterativa la evaluación, es decir, de acuerdo con la Fig. 2.4, este esquema evaluaría las mariposas cero, uno, dos y tres en la primera etapa, almacenaría los datos en una memoria para en la segunda etapa evaluar las mariposas cuatro, cinco, seis y siete al mismo tiempo y así sucesivamente, teniendo un esquema como el que se muestra en la Fig. 2.14.

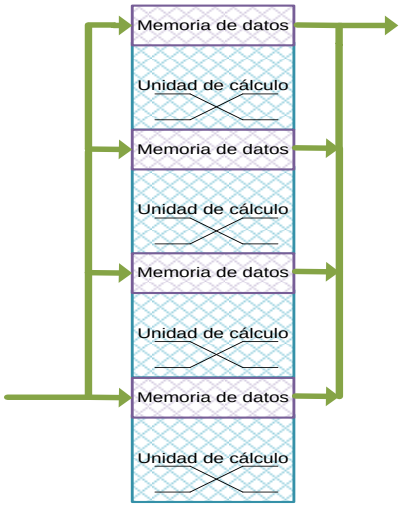


Fig. 2.14 Esquema de procesamiento en paralelo-iterativo.

Este esquema permite calcular una etapa completa con una sola evaluación, de tal manera que para obtener la evaluación de la FFT completa, se tiene que ejecutar varias veces, esta organización es la combinación de una arquitectura secuencial con un grado de paralelismo, lo que permite incrementar su rendimiento y para la FFT *radix*-2 se caracteriza por:

- Utilizar N/2 unidades aritméticas
- Se evalúan N/2 operaciones simultáneamente
- Se evalúan m etapas secuencialmente

A continuación se muestran algunas propuestas ubicadas en el estado del arte.

- *Design and implementation of a Scalable Floating-point FFT IP Core for Xilinx FPGAs [20].*

Esta propuesta evalúa el algoritmo *Radix-2* decimación en frecuencia de la FFT con un formato de datos de punto flotante utilizando el formato IEEE 754, en un FPGA Virtex 4 de Xilinx. Se hace uso de un esquema de redondeo después de las operaciones aritméticas realizadas y su arquitectura consiste en

implementar *N/2* unidades aritméticas con la finalidad de procesar una etapa al mismo tiempo, además de contar con la lógica de control y un banco de memoria de *N* palabras para almacenar los datos reportando una frecuencia máxima de operación de 115 MHz alcanzada sin especificar para que cantidad de puntos, en la Fig. 2.15 se muestra una imagen de la arquitectura presentada en donde se observa que para las pruebas implementadas se utilizó una interfaz paralelo-serie y serie-paralelo para el muestreo y la presentación de los datos, de tal manera que evalúan la FFT de 128 puntos en 1.1µs utilizando el 11.32% de los elementos DSP disponibles y el 18% de los *slices*.

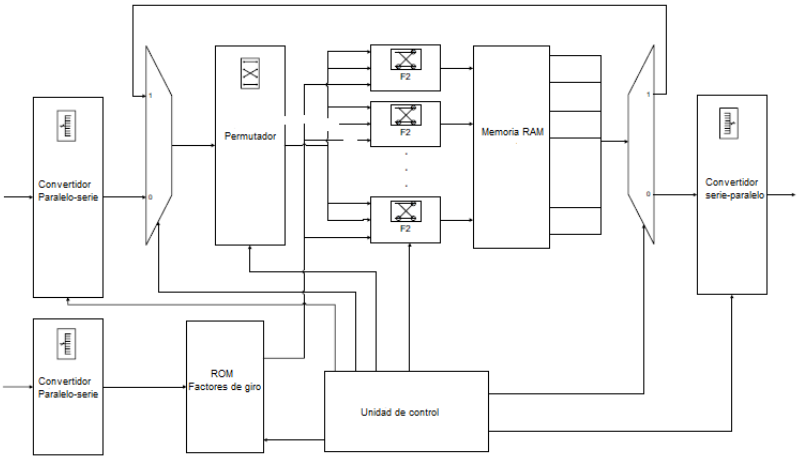


Fig. 2.15 Arquitectura propuesta por [20].

- *Design and Implementation of A Parallel Real-time FFT Processor [21]*. En esta propuesta se implementa el algoritmo *Radix-4* sobre un FPGA Xilinx de la familia Virtex II, utilizan un formato de punto flotante para realizar las operaciones y aunque no se explica a fondo el funcionamiento de la arquitectura propuesta, se menciona que se utilizan cuatro mariposas que trabajan al mismo tiempo, lo que permite agregar paralelismo a su propuesta reportando un tiempo de 40 ns para una evaluación de 16K puntos. El diagrama de bloques de la arquitectura se muestra en la Fig. 2.16, se observa que utilizan un banco de memoria para cada resultado arrojado por una mariposa, sin embargo no se menciona porque se utilizan cuatro mariposas, finalmente evalúan, de acuerdo con lo reportado en el documento 16K de puntos en 40ns con una frecuencia de reloj de 50MHz.

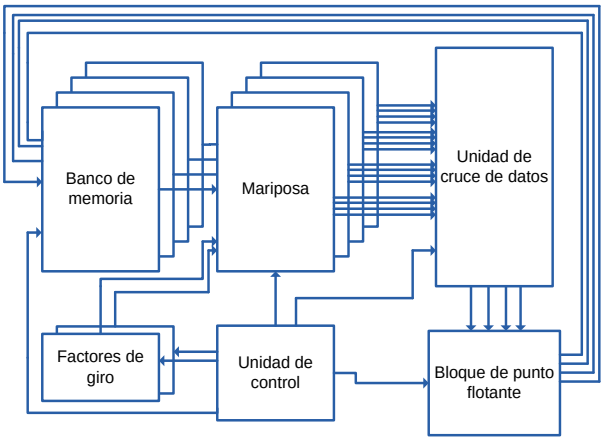


Fig. 2.16 Arquitectura propuesta en [21].

2.4.4 Procesador en arreglo

Esta arquitectura consiste en implementar todas las unidades aritméticas que se requieren para el cálculo de la FFT, como se muestra en la Fig. 2.17.

Fig. 2.17 Esquema de procesamiento en arreglo.

Este procesador presenta la ventaja de ser el más rápido de las propuestas presentadas al contar con una unidad aritmética para cada mariposa que se requiere en la FFT y evaluar todas las operaciones en paralelo al no tener que realizar accesos a memorias intermedias, sin embargo se caracteriza por requerir de una gran cantidad de hardware, lo que complica su implementación. Las características notables de esta arquitectura son:

- Utilizar $\left(\frac{N}{2}\right)(\log_2 N)$ unidades aritméticas
- Realizar $\left(\frac{N}{2}\right)(\log_2 N)$ operaciones en paralelo
- No requerir unidades de almacenamiento

2.5 Metodología para el diseño digital

De acuerdo con [26] el método es el camino adecuado o procedimiento ordenado y razonado a seguir para lograr alcanzar un objetivo, cabe destacar que el método no es único ni exclusivo, pueden existir diversos procedimientos para lograr el mismo objetivo dependiendo de variables como el tiempo o los recursos con los que se cuentan. La disciplina encargada del estudio de los métodos es la metodología.

Para este trabajo en específico, el objetivo es obtener un circuito capaz de realizar una tarea dada, como en cualquier investigación, es de suma importancia seguir un método con el fin de llegar al mejor resultado, ya que como se menciona en [28], la fase de diseño esta completa cuando una idea es transformada en una arquitectura o ruta de datos y lógica de control De forma general, el ciclo de diseño se describe en la Fig. 2.18 en donde se observa que se trata de un proceso iterativo, sin embargo existen métodos para facilitarlo, en este apartado se abordarán dos métodos para el diseño de un circuito digital, el método *bottom-up* y el método *top-down.*

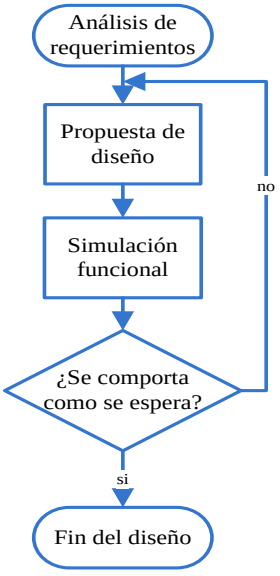


Fig. 2.18 Flujo general del diseño electrónico.

2.5.1 Método *buttom-up*

Este método tiene un enfoque incremental ya que se comienza por ubicar las unidades más pequeñas de funcionamiento, estas se agrupan en módulos que a su vez formarán un módulo más grande y así sucesivamente hasta conformar por completo el circuito. Las unidades funcionales de más bajo nivel se conocen como primitivas y comúnmente pertenecen a una biblioteca de propósito general, este método no implica una estructura jerárquica, si no que reúne componentes de bajo nivel hasta completar el diseño, la

Fig. 2.19 lo ejemplifica. De acuerdo con [27] este método comienza por elegir la tecnología de implementación y diseñando el *layout* y termina con la especificación de los requerimientos.

El utilizar este método para un diseño muy grande puede no ser lo más conveniente ya que unir miles de componentes, como lo requieren los diseños actuales, no es una tarea sencilla pues no se puede esperar que el funcionamiento sea el correcto cuando se ha perdido el control en la cantidad de elementos, para estos requerimientos, el flujo de diseño se torna ineficiente, ya que no se cuenta con una estructura jerárquica que permita hacer una separación en bloques. Esto acarrea inconvenientes en la detección de los fallos o funcionamientos inesperados.

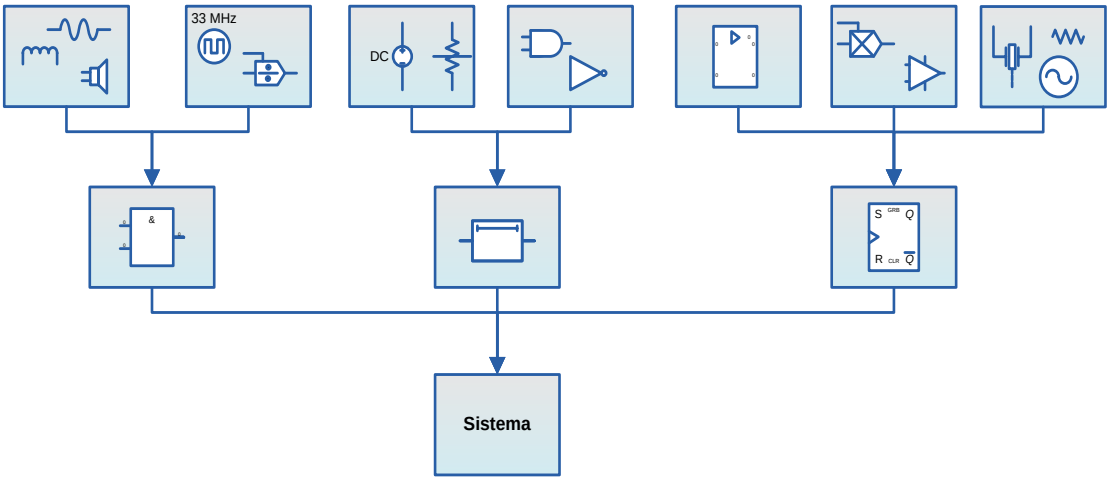


Fig. 2.19 Método *bottom-up* para el diseño de circuitos.

Este método fue uno de los primeros en surgir y tuvo gran aceptación dentro de la comunidad, a tal grado de resultar la base de algunas herramientas CAD para el diseño que permitían hacer descripciones de los diseños a bajo nivel, sin embargo, en la actualidad seguir este método resulta muy complicado.

2.5.2 Método *top-down*

Este método comienza por la descripción del sistema en donde se hace una especificación por medio de un esquema, haciendo un incremento en el nivel de detalle cuanto más se necesite, el diseño *top-down* refiere a la división recursiva del sistema en sub-componentes hasta que todos ellos sean manejables con elementos que pertenezcan a alguna librería [28], adoptando un enfoque divide y vencerás. El método se diferencia por el conocimiento y la planificación completa del sistema ya que no se puede comenzar con la implementación sino hasta que se tiene un nivel de detalle elevado, esto con el fin de prever errores o comportamientos inesperados. La Fig. 2.20 ejemplifica el uso de este método para el diseño de un circuito.

Este método presenta grandes ventajas como son:

- El diseño de sistemas complejos se vuelve manejable
- El tiempo de diseño se reduce
- La calidad del diseño se incrementa
- Es posible hacer un prototipo de manera rápida con base a un FPGA
- Los componentes del diseño se pueden reciclar, es decir, pueden servir para otros sistemas en un futuro.

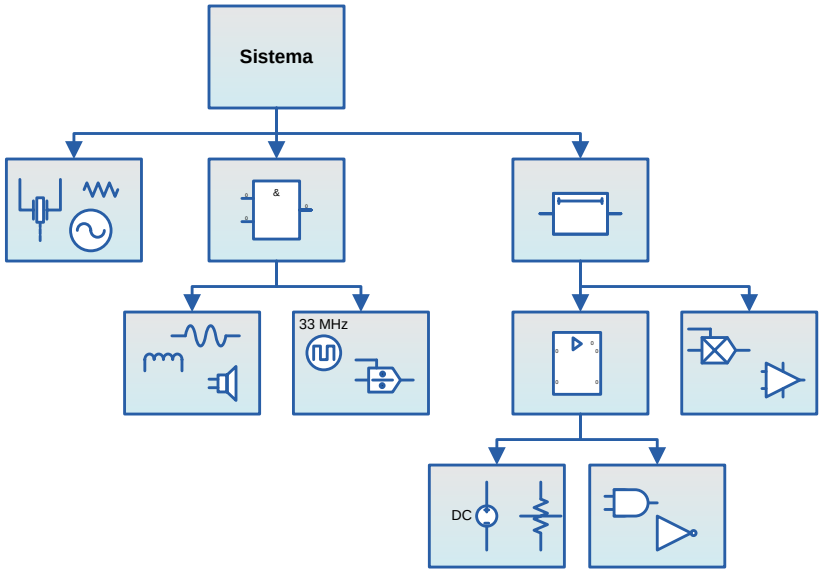


Fig. 2.20 Método de diseño *top-down.*

Utilizar un lenguaje de descripción de hardware (HDL – *Hardware Description Language*) permite hacer el diseño de un circuito con este método casi de manera natural, pues se puede hacer una descripción de la función del sistema y verificar su funcionamiento para comprobar el nivel de comprensión del objetivo mediante herramientas para verificación temprana de los diseños. Una vez que se ha comprendido por completo la función, se divide en componentes más simples y se aplica el mismo método. A diferencia del método *bottom-up* la tecnología de implementación no debe ser elegida sin antes haber comprobado el funcionamiento del circuito.

De acuerdo con [27] una de las razones por las que es conveniente la utilización de este método es porque el esfuerzo requerido para el diseño crece de forma exponencial con el número de compuertas requeridas para un circuito.

2.6 Lenguajes de descripción de hardware

Con el avance de la tecnología la posibilidad de incrementar el número de componentes en un mismo circuito ha crecido tal cual lo enuncia la Ley de Moore, esto ha generado la necesidad de contar con herramientas que auxilien en el diseño de circuitos digitales de grandes especificaciones ya que normalmente estos requieren gran inversión de ingeniería. En la década de los 50 aparecieron a nivel industrial los lenguajes de descripción de hardware que permitieron dar un salto en el diseño de circuitos complejos [30], estos tuvieron gran aceptación y 20 años después IBM y Texas Instrument desarrollaron sus propias versiones para uso exclusivo mientras que a nivel universitario también se crearon herramientas similares como lo fueron AHPL, DDL y ISPS entre otras pero que recibieron poco apoyo y mantenimiento.

Para los años 80 surgieron lenguajes como VHDL, Verilog y Abel que permitieron la implementación de un circuito a partir del conocimiento de sus entradas y salidas y el entendimiento de la función principal sin requerir conocimiento sobre cómo se ejecuta la tarea, es decir, el nivel de abstracción era mayor comparado con sus predecesores.

Un lenguaje de descripción de hardware permite la interpretación de un circuito digital en un conjunto de declaraciones que pueden describirlo en diversos niveles de abstracción como son: por comportamiento, a nivel de registros lógicos de transferencia o a nivel de componentes. Los HDL tienen por ventaja poder hacer una representación de la concurrencia que existe en los circuitos pues una vez que se ha verificado su funcionamiento por una simulación, en el hardware se hacen configuraciones que permitan que en él se mapeen los circuitos con este comportamiento, ya que no son propiamente un conjunto de instrucciones que se ejecuten secuencialmente como lo hacen los procesadores de propósito general. Los HDL han permitido un gran avance en el área del diseño electrónico pues facilitan una inversión mínima de tiempo y recursos obteniendo grandes resultados.

En los últimos años, la tendencia de los HDL ha sido captar más usuarios por medio de herramientas que tienen un alto grado de similitud con los lenguajes de programación orientados a objetos, así es como han surgido SystemC y SystemVerilog que son lenguajes que permiten hacer una descripción de los circuitos de forma 100% por comportamiento además de contar con facilidades para la verificación de estos.

2.6.1 Lenguaje de descripción VHDL *(VHSIC Hardware Description Language*)

A principios de los años 80, el departamento de defensa de Estados Unidos creó VHDL como parte de las actividades del programa *Very High Speed Integrated Circuits* que buscaba el rápido diseño de circuitos integrados. VHDL surge de la necesidad de contar con un estándar para el diseño de circuitos. Este lenguaje fue sometido a diversas revisiones realizadas por universidades, gobierno e industrias y en 1987 el IEEE hizo público el estándar 1076-1987 que fue la primera versión formalizada de VHDL. Actualmente se utiliza el estándar IEEE 1164 publicado en 1993.

VHDL hoy por hoy es uno de los lenguajes más aceptados en la comunidad para la especificación, verificación y diseño de circuitos y fue creado con la intención de sintetizar y simular funcionalmente circuitos, sin embargo una nota importante es que no todo lo que se puede verificar por una simulación es completamente sintetizable. Hacer un diseño con esta herramienta significa que se escribirá código para después ser verificado funcionalmente en una simulación y terminando con la síntesis del mismo para obtener finalmente un esquemático con compuertas lógicas y *flip-flops*, la secuencia de estas actividades se observa en la Fig. 2.21.

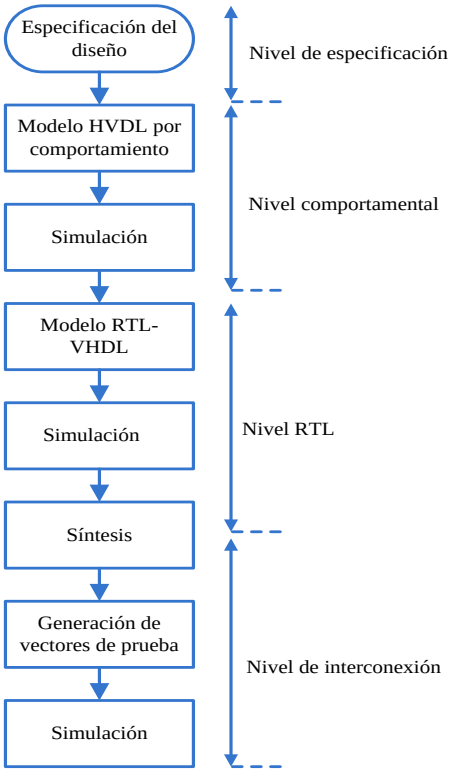


Fig. 2.21 Flujo de diseño de un circuito en VHDL.

Como se observa en la figura anterior, el diseño de un circuito está compuesto por varios pasos dentro de los que se encuentran los diferentes niveles de abstracción para la descripción, específicamente, VHDL soporta tres niveles, que son:

- Descripción por comportamiento: Este es el nivel de abstracción más alto y se caracteriza por permitir la descripción con sentencias como *if*, *case*, *for* como un lenguaje de programación, y como su nombre lo indica, describe el funcionamiento del circuito, en este nivel es en donde se encuentran los fallos entre el código que es sintetizable y el código que solamente es simulable.
- Descripción a nivel de registros lógicos de transferencia (RTL): Este es el siguiente nivel de abstracción, permite hacer descripciones por medio del conocimiento del funcionamiento del circuito.
- Descripción a nivel de componentes: Este es el nivel más bajo de abstracción, es aquí en donde se debe tener completo conocimiento del circuito y presenta la ventaja de que todo el código que se escribe bajo esta forma es 100% sintetizable.

Cabe destacar que un diseño puede ser descrito de cualquiera de las maneras antes mencionadas y si todo se ha realizado correctamente se obtendrán los mismos resultados, teniendo así una variedad de soluciones a la mano.

2.6.2 Ventajas de VHDL

Además de las ventajas que un lenguaje de descripción de hardware presentan, específicamente VHDL tiene ventajas como:

- Se encuentra disponible de forma pública, ya que es un estándar que no está sometido a una patente por lo que todos fabricantes de hardware reconfigurable y escuelas pueden hacer uso de VHDL sin restricciones.
- Al ser una estándar IEEE se garantiza la existencia de suficiente documentación y de soporte y estabilidad.
- Se puede hacer cualquier diseño ya que permite la independencia de la tecnología y del proceso de fabricación por lo que se pueden hacer diseños para tecnologías MOS, BICMOS entre otras sin que se requiera hacer alguna especificación o diferenciación.
- Se tiene una independencia de la tecnología de diseño, pues soporta diversas tecnologías como PLD, FPGA, CPLD o ASIC y circuitos secuenciales, combinacionales, síncronos o asíncronos.
- Portabilidad entre proveedores, dado que se tiene independencia entre las tecnologías de diseño, los circuitos sintetizables para hardware de la marca Altera lo son también para Xilinx y para otros

proveedores, siempre y cuando no se haga uso de las características especiales de cada proveedor de hardware.

- Reutilización de código, ya que es un estándar se permite utilizar código que ya ha generado un circuito en diversos proyectos, sin importar nuevamente la tecnología de implementación para el que haya sido utilizado puesto que se pueden crear paquetes de componentes y ser incluidos en diferentes proyectos obteniendo el mismo resultado.
- Capacidad de hacer descripción de un mismo circuito en diferentes niveles de abstracción como ya se había mencionad antes.
- Permite la modularidad en los diseños, aplicando claramente el método top-down dividiendo el diseño en bloques que pueden ser integrados más adelante

2.7 Solución propuesta

Este trabajo hace la propuesta de generar una arquitectura capaz de evaluar la FFT radix-2 con decimación en tiempo mediante un enfoque de cascada o *pipeline*, es decir, haciendo uso de una unidad de procesamiento por etapa y agregando un esquema de almacenamiento de datos *non in-place* [18] que permitirá eliminar las dependencias de datos que se pueden presentar entre cada etapa, este esquema de memoria de datos funciona también como un *pipeline* que permite tener a todas las unidades de procesamiento, a partir de un determinado momento, funcionando juntas para así agilizar el trabajo. En la Fig. 2.22 se muestra un diagrama general de la arquitectura propuesta.

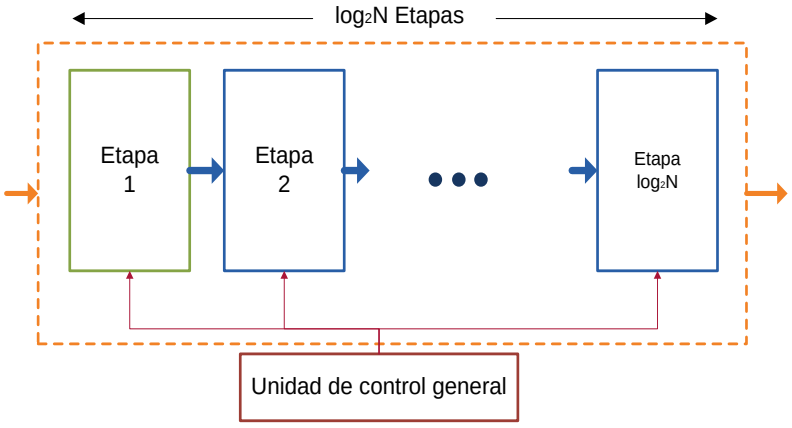


Fig. 2.22 Diagrama general de bloques de la arquitectura propuesta.

Como se observa, la arquitectura se compone de log_2N etapas, la comunicación entre etapas es el flujo de datos que se envían de manera unidireccional, es decir, cada etapa hace operaciones con los datos que le fueron enviados por la etapa anterior. De manera general, el flujo de funcionamiento de cada bloque consiste en hacer la lectura de los datos que se encuentran en las memorias, hacer la lectura del coeficiente complejo correspondiente y evaluar la mariposa para esos datos. Los resultados son enviados como entradas a la siguiente etapa en donde se evalúan de la misma manera. Este proceso se desarrolla en cada una de las etapas de la arquitectura. Es importante notar que de forma inicial, los bloques no comienzan a trabajar al mismo tiempo, estos son activados en forma gradual, por medio del bloque de unidad de control general, sin embargo, llega un momento en el cual todos los bloques se mantienen trabajando.

Una vez que se ha finalizado la evaluación del algoritmo a la salida de la última etapa se obtienen cuatro vectores de 16 bits cada uno los cuales corresponden a los resultados de la evaluación de la mariposa de dicha etapa (dos vectores para la parte real y dos vectores para la parte imaginaria) sin embargo estos

resultados todavía requieren de un procesamiento para facilitar su presentación por lo que se agrega a la arquitectura propuesta un bloque encargado de calcular el módulo de los resultados finales de la evaluación del algoritmo, este módulo ya no forma parte propiamente de la arquitectura para la evaluación de la FFT pero es importante contar con él puesto que este permite tener como salida un único vector con la magnitud de cada uno de los resultados de la evaluación y así poder construir el espectro de frecuencias de la señal de prueba. Este bloque funciona de la misma manera que los bloques de etapa, cuenta con elementos de memoria, unidad de control por etapa y demás elementos necesarios para finalizar la tarea de presentación de los resultados.

En la Tabla 2.5 se muestran datos estadísticos de la cantidad de recursos aritméticos requeridos por parte de las arquitecturas existentes en el estado del arte en comparación con los recursos que se utilizan en la arquitectura propuesta, esto con la finalidad de mostrar que se está alcanzando un nivel de balance en los recursos utilizados por la arquitectura propuesta que es capaz de ahorrar una cantidad de hardware muy grande dado que no se implementan todas las mariposas necesarias para la evaluación del algoritmo de esta forma se obtienen dos beneficios, el primero es el ahorro en el hardware con respecto a la implementación de la arquitectura 100% paralela y una disminución en el tiempo de ejecución al contar con mas unidades de trabajo en comparación con una arquitectura *in place*.

Tabla 2.5 Comparativa del hardware necesario para la evaluación de la FFT de 512 puntos.

Arquitectura	Multiplicadores	Sumadores/restadores	Total multiplicadores	Total sumadores/restadores
Arquitectura 100% iterativa	8	6	8	6
Arquitectura propuesta	8	6	72	54
Arquitectura 100% paralela	8	6	18432	13824

Capítulo 3. Diseño de la arquitectura propuesta

3.1 Bloque de etapa de la arquitectura para evaluar la FFT

El diseño de la arquitectura para la evaluación de la FFT se hizo sobre la base de utilizar un enfoque *top-down,* partiendo de un bloque funcional que se dedica a evaluar una etapa del algoritmo, cada bloque funcional está compuesto de varios elementos, en la Fig. 3.1 se muestra un diagrama de bloques de la estructura de una etapa para la evaluación de una etapa así como la interacción que existe entre los bloques que la conforman, esta es la ruta de datos para la evaluación de una etapa del algoritmo.

Cabe mencionar que el diseño de todos los módulos fue realizado tomando en cuenta que el hardware en donde se sintetizaría es un FPGA EP2C35F672C6 de la familia Cyclone-2 de Altera el cual tiene como características [30]

- 33,216 elementos lógicos
- 105 bloques de RAM del tipo M4K (4096 bits por bloque) para tener un total de 483,480 bits de memoria, cada bloque de RAM alcanza una frecuencia máxima de operación de 260 MHz y pueden ser configurados de diferentes maneras de acuerdo con las necesidades del diseñador
- 70 multiplicadores dedicados de 9 bits que alcanzan una frecuencia máxima de operación de 250 MHz
- 4 módulos PLL
- Red jerárquica de reloj para una frecuencia máxima de 402.5 MHz
- 475 pines de entrada/salida
- Soporte para mega-funciones de Altera

Algunas de las características anteriores fueron aprovechadas para la implementación de los bloques funcionales, así pues la memoria de datos de cada una de las etapas de la arquitectura así como las memorias requeridas para los bloques de coeficientes complejos fueron mapeadas en bloques de memoria RAM del tipo M4K, los multiplicadores necesarios para la evaluación de la mariposa fueron mapeados en multiplicadores dedicados de 9 bits.

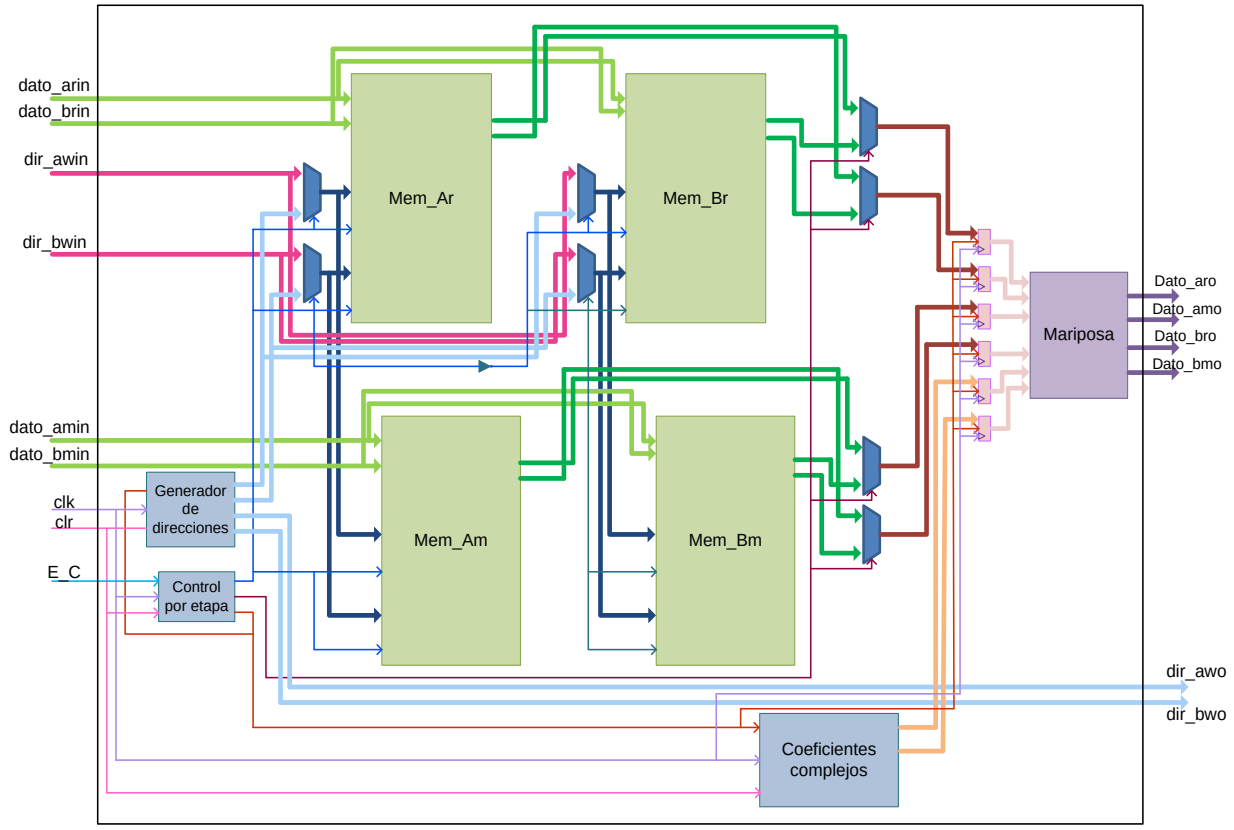


Fig. 3.1 Bloque funcional para la evaluación de una etapa del algoritmo FFT.

Cada uno de los bloques funcionales se compone de

- Dos bancos de memoria de datos en donde se almacenan los resultados de la evaluación de la mariposa de la etapa anterior.
- Generador de direcciones, este bloque se encarga de hacer el cálculo de las direcciones con las cuales se comenzará con la lectura de los datos y su posterior escritura en los bancos de memorias de la siguiente etapa.
- Generador de coeficientes complejos, este bloque se encarga de liberar los coeficientes complejos con los cuales se completan los operandos para la evaluación de la mariposa.
- Bloque de cálculo de mariposa, se encarga de evaluar la operación básica del algoritmo FFT *Radix-2* con decimación en tiempo.
- Unidad de control por etapa, se encarga de sincronizar el funcionamiento de todos los bloques de una etapa.

Y de la forma más general tiene como entradas

- *dato_arin, dato_brin, dato_amin, dato_bmin*: Estos cuatro datos son el resultado de la evaluación de la mariposa de la etapa anterior que son almacenados en los bancos de memoria de datos de la etapa actual.
- *dir_awin, dir_bwin*: Son las direcciones generadas por el bloque generador de direcciones de la etapa anterior y que permiten direccionar a las memorias de datos de la etapa actual para realizar la operación de escritura de los datos de entrada
- *clk*: Es la señal de reloj general para la arquitectura
- *clr*: Es la señal de reset general para la arquitectura
- *E_C*: Es la señal de habilitación de la etapa actual y que proviene de la unidad de control general

Para finalmente obtener como salidas

- *Dato_aro, Dato_bro, Dato_amo, Dato_bmo*: Estos datos son el resultado de la evaluación de la mariposa de la etapa actual que se utilizan como entradas para la siguiente etapa
- *dir_awo, dir_bwo*: Estas son las direcciones calculadas en la etapa actual que servirán como entradas para la siguiente etapa y así direccionar las localidades de memoria en donde se escribirán los resultados de la etapa actual.

El proceso comienza escribiendo los resultados de la etapa anterior en las memorias de datos del *Banco_A* de la etapa actual en las localidades indicadas por las direcciones de entrada, cuando la escritura termine se comienza con la lectura de esos datos para ser evaluados junto con el coeficiente complejo correspondiente, esto mientras el *Banco_B* está siendo escrito con los datos en las localidades dadas por las direcciones de entrada.

La evaluación de la FFT para N puntos requiere de log_2N etapas, por lo que se utiliza un bloque funcional como el mostrado en la Fig. 3.1 para cada una de las iteraciones por las que se compone el algoritmo, sin embargo, cabe destacar que estos bloques no son idénticos entre sí.

La primera diferencia entre ellos es el tamaño de las memorias que se utilizan para almacenar los coeficientes complejos puesto que la cantidad de ellos aumenta conforme se avanza en la evaluación. La diferencia más marcada entre los bloques funcionales se da entre la primera y la segunda etapa, dado que al comenzar con la evaluación de la FFT no se cuentan con números complejos, no son necesarios cuatro bloques de memoria si no dos y el valor de los coeficientes complejos es trivial reduciendo la operación

de la mariposa de la primera etapa a una suma y una resta por lo que este bloque no se encuentra en la primera etapa.

A partir de la segunda etapa todos los bloques conservan sus características principales, cuentan con nueve entradas (datos, direcciones, señales de control) y seis salidas (resultados y direcciones). En las siguientes secciones se describe el funcionamiento de cada uno de los componentes que son requeridos para la evaluación de una etapa.

3.2 Bloque de memoria de datos

3.2.1 Implementación de memorias FIFO para almacenamiento de datos

Una opción para la implementación de la memoria de datos es la utilización de memorias del tipo FIFO, las cuales se caracterizan por ser memorias de acceso secuencial con dos puertos de acceso, uno para lectura y otro para escritura, esto permite que las operaciones de lectura/escritura se hagan considerando que el primer dato que fue escrito debe ser el primer dato que será leído. El tiempo de realización de cualquiera de las dos operaciones para una memoria de *N* localidades es de *N* ciclos de reloj. Estas memorias tienen como ventaja el rápido acceso a la información, puesto que ya tienen un orden bien definido de cómo se harán las operaciones y no se requiere hardware adicional para la generación de las direcciones.

Para analizar el uso de las memorias FIFO para evaluar el algoritmo de la FFT con decimación en tiempo *Radix-2* observaremos el diagrama de la Fig. 2.2, en donde se ve que los datos deben de escribirse en una primera memoria FIFO en orden de reversión de bits, de acuerdo con el funcionamiento de la memoria, esto no es posible puesto que la escritura en la FIFO es secuencial.

Para poder utilizar la memoria FIFO se tendría que implementar un algoritmo para evaluar la FFT con decimación en frecuencia, de esta forma los datos serían escritos secuencialmente a la primera memoria FIFO. Partiendo de que se ha optado por implementar una arquitectura segmentada y del flujo de operaciones del algoritmo de la FFT con decimación en frecuencia que se muestra en la Fig. 3.2 se hizo el siguiente análisis.

La primera diferencia que se observa entre la Fig. 3.2 y la Fig. 2.2 es el orden de las muestras de entrada, para el algoritmo con decimación en frecuencia se requiere de una entrada en orden, lo que permite que se pueda utilizar una memoria FIFO a la entrada de la arquitectura. Un parámetro a tomarse en cuenta para

la evaluación de la mariposa es que requiere de tres operandos para su cálculo, dos datos y un coeficiente complejo, los datos provienen de la estructura de memoria que se utilice, lo que hace necesaria la lectura y escritura de ambos datos al mismo tiempo.

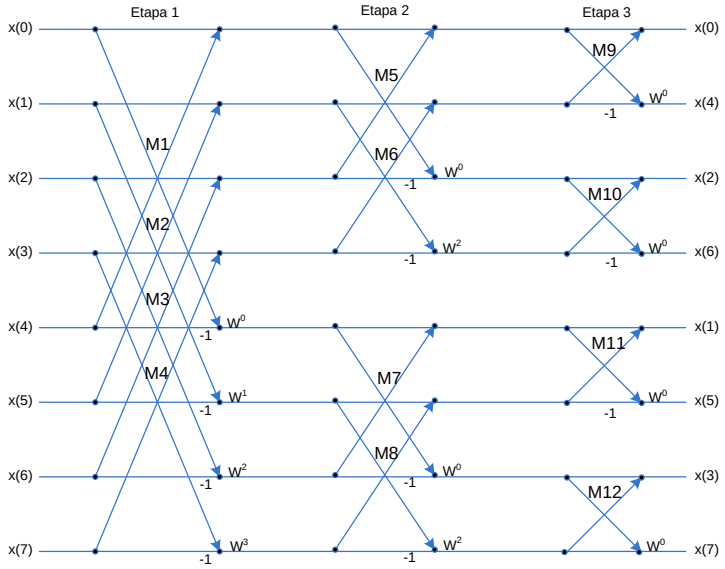


Fig. 3.2 Flujo de operaciones para el algoritmo de la FFT Radix-2 con decimación en frecuencia

Con la finalidad de poder leer dos datos al mismo tiempo para la evaluación de la mariposa sin retrasos se requerirían múltiples memorias FIFO, por ejemplo, haciendo el análisis para la Etapa 1 del algoritmo, de acuerdo con la Fig. 3.2 en donde la mariposa M1 requiere de las muestras x(0) y x(4), la mariposa M2 de las muestras x(1) y x(5), y así sucesivamente para esta etapa, se podrían utilizar dos memorias FIFO de entrada de datos.

Cada memoria almacenaría 4 datos, la primera memoria los datos correspondientes de la muestra x(0) a la muestra x(3) que equivale a almacenar los operandos *a* de cada mariposa y la segunda memoria los datos de la muestra x(4) a la muestra x(7) que corresponden al operando *b* de la mariposa. Con esta distribución de memoria, los datos de entrada completos estarían listos después de *N* ciclos de reloj, donde *N* es el número de muestras a evaluar, durante la primera escritura en las memorias FIFO, la mariposas no pueden hacer ninguna evaluación.

Para la evaluación de la primera mariposa de la Etapa 1 se haría la lectura de las primeras localidades de las FIFO (x(0)), y los resultados se almacenarían en las FIFOs de entrada de la Etapa 2, en donde se requeriría un esquema de memoria similar, utilizar una memoria FIFO de *m* localidades, donde *m* es el

número de mariposas por grupo en la etapa, para el caso de la Etapa 2 las FIFOs serían de dos palabras, y se utilizarían 4 memorias, dos por grupo de mariposas.

La primera FIFO almacenaría los operandos *a* de las mariposas M5 y M6 en tanto que la segunda FIFO del grupo almacenaría los operandos *b* de las mismas mariposas, este esquema se repetiría para las mariposas M7 y M8. Esta distribución de la memoria solucionaría el problema de hacer la lectura y escritura de ambos operandos en un mismo ciclo de reloj.

El planteamiento anterior aparentemente permitiría que la evaluación de una etapa del algoritmo se hiciera en *N/2* ciclos de reloj tomando ventaja de que una memoria de este tipo cuenta con la facilidad de utilizar señales de reloj para lectura y escritura independientes, sin embargo, aun así se tendrían ciclos de reloj muertos dado que, para la Etapa 1 una vez que se han llenado ambas FIFO al ciclo de reloj siguiente se puede comenzar con la evaluación de las mariposas, lo que tomaría *N/2* ciclos de reloj mientras que se podría llenar la primera memoria FIFO, pero el llenado de cada FIFO requiere de *N/2* ciclos, en total se utilizarían *N* ciclos de reloj para escritura y *N/2* ciclos para lectura.

Tratando de aprovechar la característica de señales de reloj diferentes para lectura y escritura, si se hace la escritura dos veces más rápido que la lectura se podría hacer la evaluación de las mariposas en tiempo pero erróneamente ya que se sobre escribirían datos que no han sido utilizados.

En La Fig. 3.3 se muestra un diagrama de la distribución de las memorias FIFO en cada etapa.

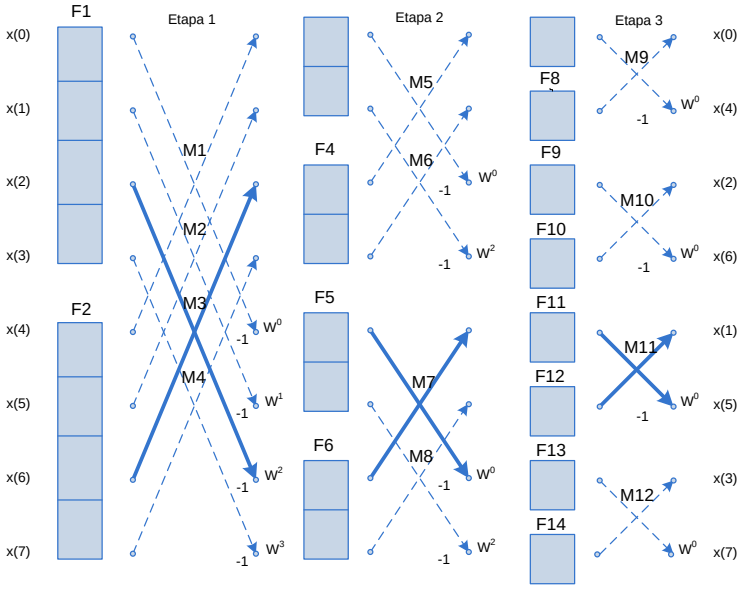


Fig. 3.3 Evaluación de la FFT Radix-2 decimación en frecuencia utilizando memorias FIFO

En la Fig. 3.3 se observa con líneas continuas y más oscuras una sola mariposa por cada etapa, esto con la finalidad de hacer referencia a una arquitectura en cascada, recordando que sólo se implementa una unidad de cálculo para cada etapa, es necesario mencionar que además de requerir de las memorias de tipo FIFO, implementar un esquema de este tipo requeriría también de hardware adicional para la selección de la memoria en donde se almacenarían los datos.

A manera de conclusión, la implementación de este esquema de memoria muestra que se reduciría la cantidad de recursos de este tipo de manera significativa ya que en cada una de las etapas del algoritmo se requerirían de 2^i memorias FIFO, donde *i* es la etapa del algoritmo, cada una de estas memorias de *m* localidades, donde *m* es el número de mariposas por grupo en cada etapa, sin embargo, habrían tiempos de evaluación muertos por la espera para completar la cantidad de operandos requeridos para la evaluación.

3.2.2 Implementación de memorias RAM para almacenamiento de datos

Una vez que se analizó la opción de utilizar memorias FIFO, se hace la propuesta de un esquema de almacenamiento de datos *non in-place*, es decir, se utilizan un par de memorias (*Mem_Ax* y *Mem_Bx*) de manera tal que mientras se escribe en la *Mem_Ax*, se realiza la lectura de los datos que se encuentran almacenados en *Mem_Bx*, este proceso toma el mismo tiempo para ambas memorias por lo que al finalizar tareas en cada memoria se puede hacer un intercambio en la función de cada una, es decir, escritura de datos en *Mem_Bx* y lectura de *Mem_Ax*.

Esta configuración de las memorias permite acelerar el proceso de evaluación debido a que elimina las dependencias de datos entre una etapa y otra, sirviendo también como una etapa de *pipeline* entre las evaluaciones de las etapas y permitiendo hacer la lectura de ambos datos para la operación en un solo ciclo de reloj por lo que se requiere únicamente *N/2* ciclos de reloj para la evaluación de una etapa del algoritmo. La Fig. 3.4 muestra un diagrama de la distribución de las memorias utilizadas.

Cada memoria tiene un tamaño de *N* palabras de 16 bits y cuenta con dos puertos de lectura/escritura (*dato_axin, dato_bxin)*, señal de habilitación de función por puerto (*we*), puertos de direcciones (*dir_a, dir_b*) y puertos de salida de datos (*dato_aX, dato_bX*), esto permite adquirir un par de datos para la operación en un sólo ciclo de reloj y efectuar las operaciones de lectura y escritura de las memorias en $N/2$ ciclos de reloj para con esto hacer la evaluación de la FFT más fluida, sin presentarse tiempos muertos entre iteraciones.

La operación de lectura/escritura entre ambas memorias es mutuamente excluyente, mientras *Mem_Ax* recibe el valor de la señal *we* tal cual la unidad de control por etapa lo envía, *Mem_Bx* recibe el valor complementario, esto permite que las memorias se complementen, mientras en una se escriben datos, de la otra memoria se está haciendo la operación de lectura de los datos previamente almacenados.

La señal de habilitación de función *we* es una señal con un periodo de N ciclos de reloj, la cual mantiene durante $N/2$ ciclos un valor uno lógico para activar escritura y en los ciclos restantes un valor de cero lógico para activar lectura, como se mencionó antes, este es el tiempo necesario para completar la función correspondiente a cada memoria.

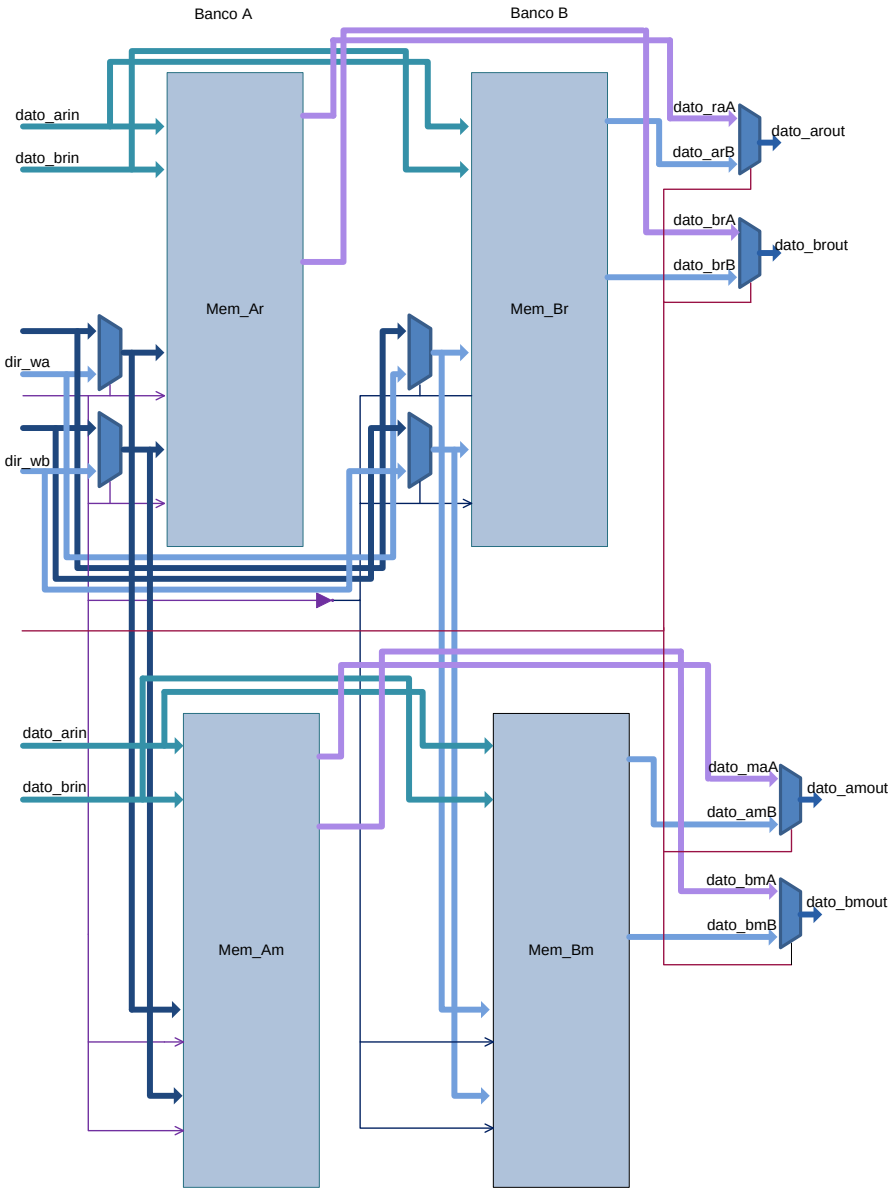


Fig. 3.4 Diagrama de bloques de la distribución de un bloque de memoria de datos.

De acuerdo con la Fig. 3.1 este bloque es el primer elemento de cada una de las etapas de la arquitectura, es aquí en donde se guardan los resultados de la evaluación de la mariposa de la etapa anterior para posteriormente ser leídos y evaluados por la mariposa de la etapa actual.

Por la naturaleza de la FFT, las direcciones de lectura y escritura no son las mismas, dependen de la operación a realizar, por lo que se requiere hacer un intercambio entre las direcciones que se utilizan dependiendo de la tarea, direcciones de escritura provenientes de la etapa anterior o direcciones de lectura que son calculadas en la misma etapa, es por eso que se utilizan bloques multiplexores que permitan hacer la selección de una dirección, para este caso, la señal de control que se utiliza es *we* puesto que las direcciones que se toman del multiplexor dependen de la tarea a realizarse.

El bus de datos de las memorias se comparte puesto que cuando se escriben datos en *Mem_Ax*, de *Mem_Bx* se lee información ya almacenada entonces no importa qué se encuentre presente en el bus de datos pues estos valores son ignorados debido a la tarea de lectura que se está ejecutando, cuando se invierten las tareas de las memorias entonces ahora quien ignora los datos presentes en el bus es *Mem_Ax*.

Los datos que se obtienen de la lectura de la memoria en turno son enviados al bloque de mariposa para ser utilizados como operandos, sin embargo, debido a la configuración de las memorias, en los buses de salida de datos siempre se tiene información, es decir, si *Mem_Ax* está bajo el modo de lectura en los buses de datos de salida se encontrarán los datos que se leen, en tanto que en los buses de datos de salida de *Mem_Bx*, que se encuentra en el modo de escritura, se tiene la información que se acaba de escribir en las localidades correspondientes, por esta razón, estos datos no pueden compartir un bus de entrada al bloque de evaluación de la mariposa pues causarían fallas.

Es por esto que se utilizan a las salidas de los datos de las memorias, dos multiplexores que permitan discriminar a uno de los datos y así continuar con la evaluación de los datos correspondientes en la mariposa. La señal de selección de los multiplexores de datos identificada como *sm* proviene de la unidad de control por etapa y tiene las mismas características que la señal de habilitación de función *we* para las memorias, el funcionamiento a detalle de ambas señales se describe en la sección de la unidad de control por etapa.

Como se observa en la Fig. 3.4 el bloque de memorias para almacenamiento de datos está formado por cuatro memorias, esto es porque las operaciones se realizan con números complejos que tienen una componente real y una componente imaginaria, un banco de memoria está conformado por una memoria

para almacenar datos reales y una memoria para almacenar los datos imaginarios (*Mem_Ar*, *Mem_Am* y *Mem_Ar*, *Mem_Bm*), cada bloque de memorias está formado por dos bancos de memorias (*Banco_A, Banco_B).*

No es necesario implementar hardware adicional para el manejo de las direcciones bajo las cuales se operan las memorias para datos imaginarios, puesto que las localidades de donde se leen los datos reales corresponden con las localidades de donde se leen los datos imaginarios y lo mismo para las localidades de escritura, sin embargo si se requieren de los multiplexores para los datos de salida por la razón ya explicada anteriormente.

Los bloques de memoria de datos fueron mapeados en bloques de memoria RAM dedicados del tipo M4K que se encuentran distribuidos en el FPGA, estos bloques permiten ser configurados dependiendo de los requerimientos del usuario, para este caso se eligió una configuración como la que se muestra en la Fig. 3.5.

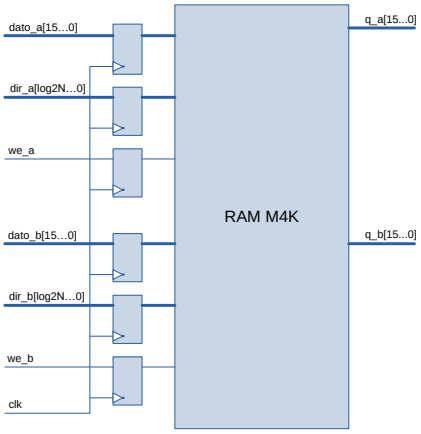


Fig. 3.5 Configuración de Memoria M4K.

Como se observa, todas las entradas de la memoria pasan primero por un registro síncrono antes de llegar a la memoria, dado que estos registros son activados en flanco de ascenso en la señal de reloj, es necesario que los datos para las operaciones se encuentren fijos y bien definidos en los buses a las entradas de los registros para que no existan problemas en las operaciones.

Para este caso en específico, los datos para las direcciones de operaciones son generados por bloques secuenciales que también dependen de una señal de reloj, aspecto que debe ser considerado en el diseño. La señal de control que utiliza este bloque es únicamente *we*, y esta señal puede tomar dos valores, cero

para indicar lectura y uno para escritura, las memorias siempre se van a encontrar en alguna de estas dos operaciones. No hay un estado en el cual las memorias se queden estáticas.

3.3 Bloque de *bit-reverse*

Como se explicó anteriormente, al tratarse del algoritmo de decimación en tiempo se requiere hacer un reordenamiento en la escritura de los datos de entrada, por lo que se diseñó un bloque que permitiera generar las direcciones de escritura para el bloque de memorias de la primera etapa. La Fig. 3.6 muestra un diagrama de la entidad generada.

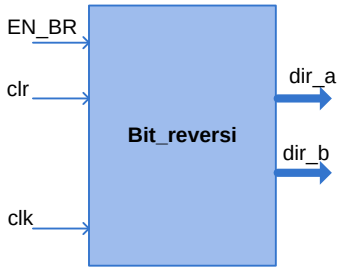


Fig. 3.6 Entidad del bloque para bit-reverse.

El bloque funciona con base en una señal de habilitación proveniente de la unidad de control general (*EN_BR*), la señal de reloj de la arquitectura (*clk*) y la señal de *reset* general (*clr*) obteniendo como salidas dos vectores con un tamaño de palabra de $log_2 N$ bits que permiten direccionar memorias de *N* palabras para almacenar las *N* muestras para la evaluación de la FFT.

Para la implementación de este bloque se utilizó un contador *modN* ascendente con *reset* asíncrono que fue descrito por comportamiento, una vez que la señal de habilitación *EN_BR* toma el valor de uno lógico, este contador genera dos números en forma de vectores de bits en cada flanco de ascenso del reloj, este proceso sintetiza un circuito secuencial, sin embargo la inversión de bits de los vectores generados por el contador se hace de manera combinatoria, en la Fig. 3.7 se muestra un diagrama de los componentes del bloque.

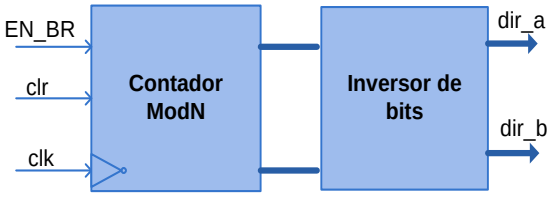


Fig. 3.7 Diagrama de componentes del bloque bit-reverse.

Si por alguna razón, el valor de *EN_BR* es interrumpido, de acuerdo con el funcionamiento del contador, la generación de las direcciones se detiene y a la salida se mantiene el último dato generado, una vez que se restablece el valor de la señal de habilitación, el bloque comienza su funcionamiento a partir del punto en donde fue detenido, dado que la generación de las direcciones depende de un circuito secuencial, es importante que la señal de habilitación para la operación esté lista antes de que sea requerida la generación de direcciones puesto que ésta es proveniente de la unidad de control general y también será resultado de un circuito secuencial, es decir, si las operaciones se realizan en un flanco de ascenso se recomienda que la señal esté lista un flanco de descenso antes, esto para evitar tiempos de *set* y *preset* y así tener las señales de entrada estables.

Este comportamiento permite detener la generación de direcciones y habilitarla tiempo después sin que haya alguna afectación en las direcciones de las localidades a escribir en las memorias de la primer etapa, este proceso se describe a detalle en el apartado de la unidad de control general.

3.4 Bloque generador de direcciones

Por la naturaleza de la FFT, el orden en que se toman los datos para la evaluación no es secuencial y tampoco se mantiene constante entre cada etapa, es por esto que se requiere un componente que permita generar las direcciones para la lectura de las localidades de memoria en donde se encuentran los datos para evaluación en la mariposa y las direcciones para la escritura de los resultados en el banco de memoria correspondiente.

Este componente se encarga de generar dichas direcciones con base en el *offset* entre cada operando por etapa, el número de mariposas por grupo (*m_d*), el número de grupos por etapa (*g_wn*) en cada una de las etapas, la señal de reloj general de la arquitectura (*clk*), la señal de *reset* general de la arquitectura (*clr*), y a una señal de habilitación proveniente de la unidad de control por etapa (*EN_GD*) como entradas y entonces obtener como salidas dos señales (*dir_ra, dir_rb*) que corresponden a las direcciones de lectura que serán enviadas al bloque de memoria de datos de la etapa actual para leer de dichas localidades los datos que serán evaluados por el bloque de mariposa, y dos señales más (*dir_wa, dir_wb*) que son enviadas al bloque de memoria de datos de la etapa siguiente para ser las direcciones de escritura. Cada una de estas señales tiene un tamaño de $log_2 N$ bits para direccionar *N* localidades del bloque de memoria de cada etapa. La Fig. 3.8 muestra la entidad a desarrollar.

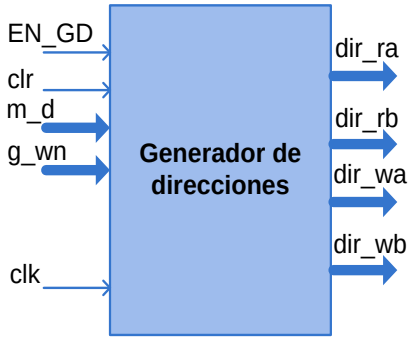


Fig. 3.8 Entidad del bloque Generador de direcciones.

Este bloque se implementó describiendo por comportamiento un algoritmo que se evalúa de manera secuencial, mediante un bloque *process* y la secuencia de ejecución es la siguiente. Se inicializan las variables internas del proceso que son *i*, el valor base de la dirección, *mar*, contador de mariposas por grupo, *grupo*, contador de grupos por etapa con cero cuando la señal *clr* toma un valor de uno lógico.

El proceso se queda en espera de un flanco de ascenso *clk* para evaluar el valor de la señal *EN_GD*, si esta tiene un valor de uno lógico se incrementa el valor de *i* y el de *mar* en una unidad, este incremento puede generar que el valor límite para el contador de mariposas sea rebasado, es por eso que se hace una comparación entre *mar* y *m_d*, si el menor de los dos es *mar* entonces los contadores permanecen con su valor, de manera contraria, *mar* toma su valor inicial, el contador de grupo *grupo* se incrementa en una unidad y la dirección base *i* se aumenta en *m_d* unidades, lo que significa que se han evaluado todas las mariposas de un grupo y se requiere hacer el cambio. De igual manera que la comparación para la cantidad de mariposas por grupo, se requiere hacer la comparación entre *grupo* y *g_wn*, si el menor de los dos es *grupo* los contadores permanecen sin cambios, de lo contrario los contadores son reiniciados lo que es indicativo de que se comenzará a evaluar nuevamente la etapa del algoritmo.

Finalmente, después de la evaluación del algoritmo, las direcciones de lectura se calculan con base en la variable *i* de tal manera que

$$dir\_ra = i \;\&\; dir\_rb = i + m\_d$$

sin embargo, dado que las direcciones de escritura no se requieren si no hasta un ciclo de reloj después que las direcciones de lectura, esto para dar tiempo al bloque de evaluación de mariposa de hacer las operaciones necesarias, en lugar de repetir el cálculo, estas son almacenadas en un *flip-flop* para ser dispuestas en el momento en que se requieran tal que

$$dir\_wa = dir\_ra \;\&\; dir\_wb = dir\_rb$$

La señal de habilitación *EN_GD* proviene de la unidad de control por etapa de cada uno de los bloques, esta señal se encarga de habilitar la generación de las direcciones, en caso de que esta sea interrumpida, a la salida se mantiene el último dato generado tanto de lectura como de escritura, esto permite reanudar las operaciones con los datos y las direcciones correctas y bajo las que se interrumpieron, evitando así errores de lectura, evaluación y escritura de los datos. El funcionamiento de este algoritmo se puede describir mediante el diagrama de flujo que se muestra en la Fig. 3.9.

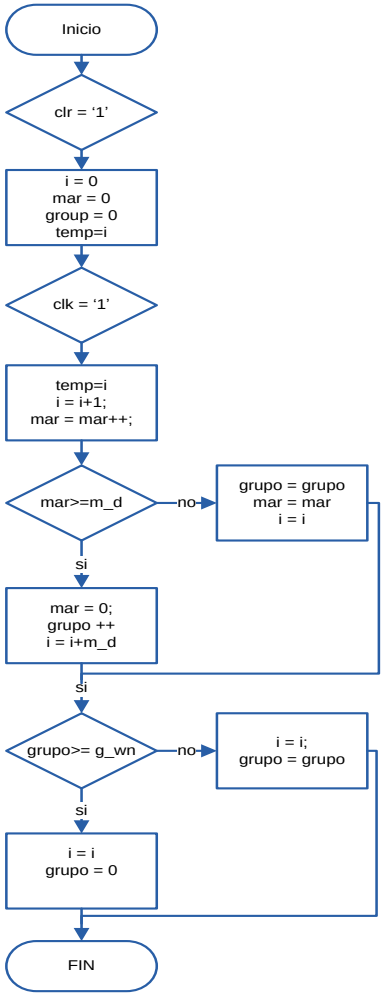


Fig. 3.9 Diagrama de flujo para el algoritmo para la generación de direcciones.

3.5 Bloque generador de coeficientes complejos

Esta unidad tiene como función la generación de los coeficientes complejos que se utilizan como operandos para la evaluación de la mariposa y se compone de dos bloques de memoria ROM de las mismas características que los bloques utilizados para el almacenamiento de datos. En la Fig. 3.10 se muestra la entidad implementada.

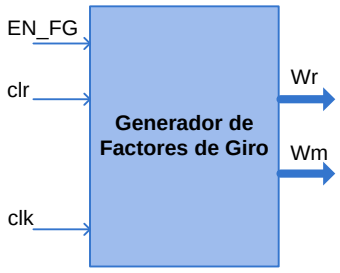


Fig. 3.10 Entidad del Generador de coeficientes complejos.

Se puede observar que el funciona a partir de en una señal de habilitación *(EN_FG)*, la señal de reloj *(clk)* y la señal de *reset* general *(clr)*, estas señales se utilizan dado que internamente la entidad está formada por un contador *modX* (el módulo del contador depende del tamaño de la memoria que se direcciona) que genera la dirección de lectura de las memorias ROM en donde se encuentran almacenados los valores calculados. Los coeficientes complejos son números complejos representados en el formato Q15 de punto fijo, es por eso que se requieren de dos memorias, una para almacenar la parte real y otra para almacenar la parte imaginaria de los coeficientes. En la Fig. 3.11 se muestra la configuración ya descrita.

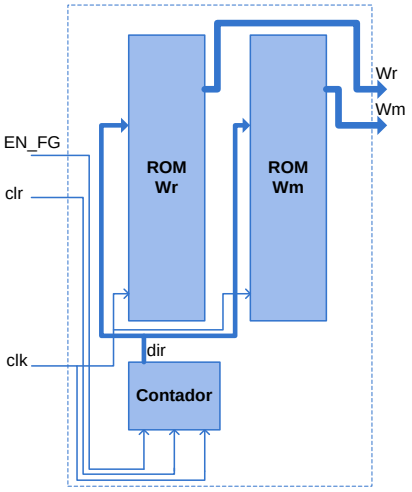


Fig. 3.11 Diagrama de bloques del generador de coeficientes complejos.

La entrada de habilitación (*EN_FG*) proviene de la unidad de control por etapa, ésta se encarga de habilitar el inicio de la cuenta y por ende el direccionamiento de las memorias ROM que al igual que los bloques anteriores de memorias, éstas se mapearon en bloques de memoria dedicados, con una configuración como la que se muestra en la Fig. 3.12.

Al ser un contador síncrono en flanco de ascenso es conveniente que la señal de habilitación cambie de valor, dependiendo de la operación que se requiere, antes de que se pregunte por ella, es por eso que el

manejo de esta señal se hace en flanco descendente de reloj. De igual manera que para los bloques generadores de direcciones, al ser interrumpida la señal de habilitación, en el bus de direcciones de las memorias se mantiene el último valor generado, lo que permite reanudar la operación de manera certera.

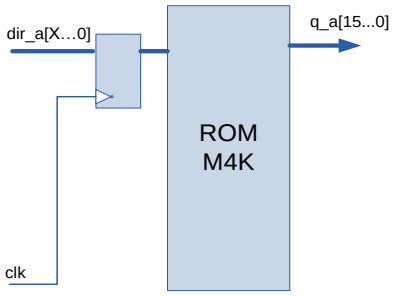


Fig. 3.12 Configuración para el bloque M4K del bloque de memoria.

El contador genera una dirección en un flanco de reloj de ascenso, ésta se envía como dirección a las memorias ROM que al tener a la entrada un registro por donde pasa la dirección se toma en tiempo de *preset*, es decir, la operación se ve reflejada hasta el siguiente ciclo de reloj en el que la dirección fue generada. Los coeficientes fueron calculados utilizando MatLab 2011 y almacenados previamente en las memorias, el tamaño de las memorias en cada etapa es diferente y va de 2 a *N/2* localidades, esto es porque se requieren más coeficientes en la medida en que la evaluación del algoritmo avanza de etapa.

3.6 Bloque para evaluación de la mariposa

Este bloque está encargado de efectuar la evaluación de la operación principal de la FFT, para esta arquitectura se utiliza una mariposa *Radix*-2 en donde se requiere de dos datos (*a* y *b*) y del coeficiente complejo (*W*) correspondiente a la mariposa de la etapa a evaluar, de forma general, estos datos son números complejos. En la Fig. 3.13 se muestra la entidad descrita.

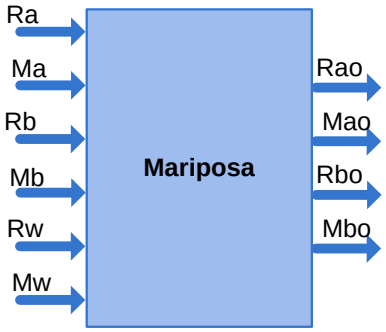


Fig. 3.13 Entidad genérica de la mariposa.

Como se observa, se tienen seis entradas, cuatro de ellas correspondientes a los datos leídos del banco de memorias de datos (*Ra*, *Ma*, *Rb*, *Mb*) y los dos restantes leídos del bloque de coeficientes complejos (*Rw*, *Mw*). Esta es la forma general de la entidad que se utiliza en la mayoría de las etapas, sin embargo, existen diferencias entre la mariposa de las primeras dos iteraciones y la mariposa de las iteraciones restantes. El flujo de las operaciones para la mariposa descrita por la entidad de la Fig. 3.13 se describe en el diagrama de la Fig. 3.14.

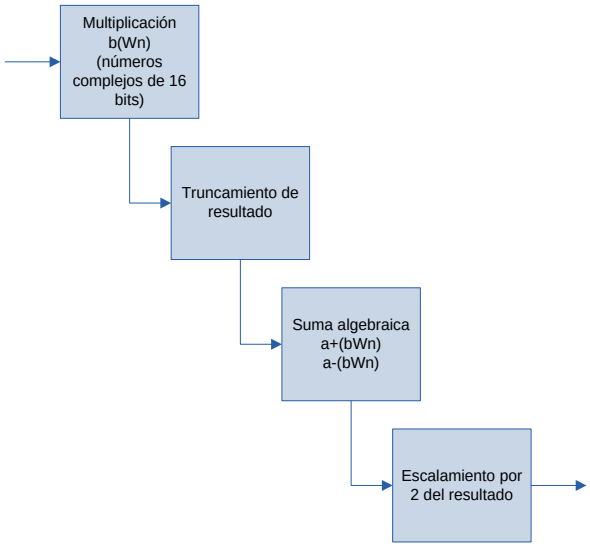


Fig. 3.14 Flujo de operaciones de mariposa.

Recordando que la mariposa está definida como

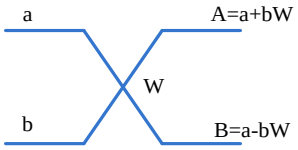


Fig. 3.15 Definición de la operación mariposa para la FFT.

La evaluación de la mariposa comienza calculando la multiplicación entre el coeficiente complejo W y el operando b proveniente de las memorias de datos, esta operación comprende para el caso más general a dos números complejos, y se descompone de la siguiente manera.

Si

$$b = (Rb + iMb)\ y\ W = (Rw + iMw)$$

Se tiene que

$$bW = (Rb + iMb)(Rw + iMw) = (RbRw) + (iBrMw) + (iMbRw) - (MbMw)$$

$$(Rb + iMb)(Rw + iMw) = (RbRw - MbMw) + i(RbMw + MbRw)$$

Finalmente, la multiplicación de dos números complejos se descompone en cuatro multiplicaciones y dos sumas entre sus factores. Al tratarse de números con un tamaño de palabra de 16 bits (dado por el factor de giro que está representado en el formato Q15), son necesarios dos multiplicadores dedicados puesto que estos son de nueve bits para cada una de las cuatro multiplicaciones requeridas, en total ocho multiplicadores, para finalmente obtener resultados de 32 bits en formato Q15.

Para completar la evaluación de la mariposa se requiere hacer un truncamiento del resultado esto con el fin de devolver el formato Q0 al resultado de la operación que es el que mantiene el operando *a* para hacer la suma y resta correspondientes. Este truncamiento consiste en tomar los bits del 15 al 30 del resultado de la multiplicación obteniendo nuevamente resultados con un tamaño de palabra de 16 bits como se muestra en la Fig. 3.16.

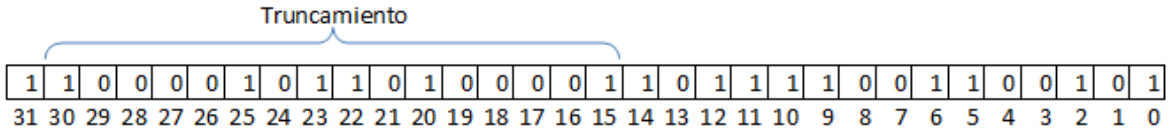


Fig. 3.16 Truncamiento de la multiplicación.

Después del truncamiento, la operación restante está dada por

Si

$$a = (Ra + iMa)\ y\ x = (Rx + iMx)$$

donde

$$Rx = (RbRw - MbMw)\ y\ Mx = (RbMw + MbRw)$$

entonces

$$A = a + x = (Ra + Rx) + i(Ma + Mx)$$
$$B = a - x = (Ra - Rx) + i(Ma - Mx)$$

Como se observa, finalmente se obtienen cuatro números correspondientes a la parte real y la parte imaginaria de datos *A* y *B* a almacenarse en las memorias, sin embargo aun queda una última operación, la cual consiste en hacer un escalamiento por dos, esto es para evitar un posible desbordamiento de la información en etapas posteriores puesto que en la medida en que se avanza entre etapas los resultados obtenidos tienden a crecer y debido a que se presenta un truncamiento en la multiplicación, se puede obtener una representación errónea de los datos.

El escalamiento previene situaciones de este tipo y consiste en hacer una división del resultado final entre dos, lo que equivale a hacer un corrimiento a la derecha de un bit y replicar el bit de signo como se observa en la Fig. 3.17.

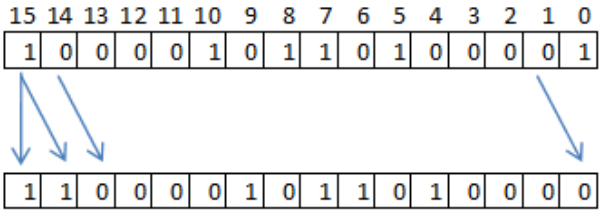


Fig. 3.17 Escalamiento inter-etapa del resultado.

La diferencia mencionada anteriormente entre la mariposa genérica y la que evalúa la primera etapa se caracteriza por la ausencia de multiplicaciones entre números complejos puesto que los datos *a* y *b* son números reales que son utilizados tal cual fueron recibidos a la entrada de la arquitectura y los coeficientes complejos en esta etapa son triviales dado que la componente real tiene un valor de 1 y la operación de las componentes imaginarias se anula puesto que la componente tiene un valor de cero, es por eso que la operación se reduce a una suma y una resta como se muestra en la Fig. 3.18.

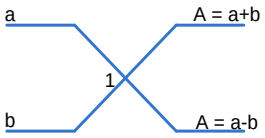


Fig. 3.18 Definición de la mariposa para la primera etapa.

La implementación de la mariposa para la primera etapa no requiere de los primeros dos pasos de la Fig. 3.14, sin embargo, para mantener una congruencia con las demás etapas, se mantiene el escalamiento descrito.

Finalmente, la mariposa que se evalúa en la segunda etapa también es diferente a la unidad genérica, puesto que recibe como entradas a los operando *a* y *b* de la primera etapa que son números reales y el factor de giro complejo, por tal motivo, las operaciones quedan definidas por

Si

$$b = (Rb + i0)\ y\ W = (Rw + iMw)$$

entonces

$$bW = (Rb + i0)(Rw + iMw) = (RbRw) + (iRbMw) + (i0Rw) - (0Mw)$$

$$(Rb + i0)(Rw + iMw) = (RbRw) + i(RbMw)$$

Que como se observa, se reducen las multiplicaciones necesarias a la mitad, a partir de este punto, las operaciones requeridas para completar la evaluación de la mariposa son las mismas que en la unidad genérica, se hace el truncamiento y después se continúa con la suma y restas correspondientes.

Si

$$a = (Ra + iMa)\ y\ x = (Rx + iMx)$$

donde

$$Rx = (RbRw)\ y\ Mx = (RbMw)$$

entonces

$$A = a + x = (Ra + Rx) + i(Mx)$$
$$B = a - x = (Ra - Rx) + i(-Mx)$$

Para finalmente aplicar el escalamiento correspondiente al resultado y obtener a partir de esta etapa números complejos que serán evaluados por la mariposa genérica descrita anteriormente.

La unidad de mariposa fue descrita por comportamiento, haciendo uso de los multiplicadores dedicados con los que cuenta el FPGA. Estos multiplicadores tienen un tamaño de palabra de 9 bits y puesto que los operandos tienen un tamaño de 16 bits son necesarios dos multiplicadores para efectuar la operación completa, por cada mariposa que se evalúa se utilizan ocho multiplicadores dedicados si se trata de la unidad genérica, para la mariposa de la segunda etapa sólo se requiere de la mitad.

Como se mencionó anteriormente, el formato de representación de los coeficientes complejos es un formato Q15 puesto que se está utilizando aritmética de punto fijo, esto obliga a hacer ajustes en las operaciones de la mariposa además del truncamiento que se requiere para la multiplicación, finalmente los resultados de la mariposa conservan un formato de representación Q0.

Esta unidad se diseñó sobre el principio de contar con un elemento completamente combinatorio puesto que los operandos *a, b* y *W* son quienes se liberan en sincronía con la señal de reloj, en tanto que la mariposa evalúa la operación en el momento en el que estos estén disponibles. Por esta razón, a las entradas de la mariposa se encuentran registrados los operandos, como se muestra en la Fig. 3.19, esto es para evitar evaluaciones a des-tiempo una vez que la señal de habilitación de lectura/escritura de la memoria sea interrumpida y para asegurar que cuando se reanuden las operaciones los resultados que se escriban sean los correctos en las localidades direccionadas, el funcionamiento de la señal de habilitación de los registros (*EN_M*) se describe en el apartado de la unidad de control por etapa, finalmente, la Fig. 3.19 muestra la el bloque de evaluación de la mariposa con los registros en la entrada.

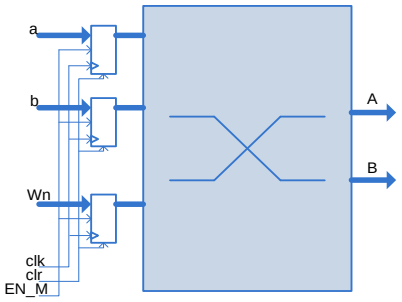


Fig. 3.19 Diagrama de bloques de la mariposa.

3.7 Bloque para el cálculo del módulo de resultados

Este es el bloque agregado al finalizar la evaluación de la FFT con el objetivo de hacer el cálculo del módulo de los resultados y de esta manera obtener un solo vector de bits como salida de la arquitectura. Este bloque puede ser considerado como una etapa más del algoritmo con la diferencia de que en él ya no se realiza la operación de de evaluación de mariposa ni se cuenta con el módulo de coeficientes complejos. La Fig. 3.20 muestra la entidad del bloque a describir.

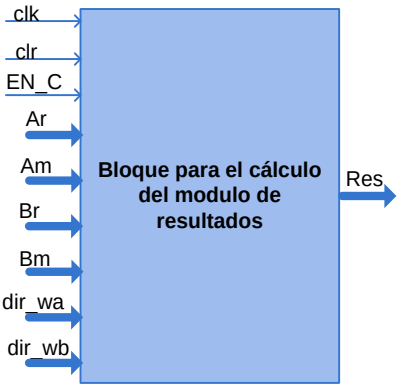


Fig. 3.20 Entidad del bloque para el cálculo del módulo de resultados.

Al ser considerado como una etapa más, este bloque se compone de dos bancos de memoria lo que permitirá optar por un esquema *non in-place*, un bloque generador de direcciones que permite direccionar las *N* localidades de las memorias, de igual manera cuenta con los multiplexores de direcciones y datos, una unidad de control por etapa y el bloque de cálculo del módulo. Como se observa, esta etapa no presenta grandes cambios en la ruta de datos por lo que se infiere que tampoco presentará cambios en la lógica de control.

- Bloque Generador de direcciones para el cálculo del módulo: El primer paso de la ruta de datos de este bloque consiste en escribir los resultados de la última etapa del algoritmo en las localidades direccionadas por los vectores generados por el bloque generador de direcciones de la última etapa, una

vez que se ha terminado de escribir las *N* palabras en las memorias correspondientes (de datos reales e imaginarios) se continúa con la lectura de esos datos mientras que en el otro banco de memoria se escriben los nuevos resultados generados.

Dado que se evalúa un algoritmo de decimación en tiempo el *bit-reverse* es aplicado al principio del algoritmo por lo que la lectura de los datos almacenados en las memorias se puede hacer en orden, es decir, comenzando por la localidad 0 e incrementando de una en una hasta llegar a la palabra *N-1*, por lo que el bloque generador de direcciones para el cálculo de módulo se reduce a un contador *modN* ascendente que incrementa su cuenta en cada flanco ascendente de reloj.

- Bloque del cálculo del módulo: Este es el bloque que sustituye al de evaluación de la mariposa, permite hacer el cálculo del módulo de los resultados leídos del banco de memorias en turno. Recordando que el módulo de un número complejo está definido como la raíz cuadrada de la suma de los cuadrados de sus componentes real e imaginaria, para esta aplicación se omite la operación de raíz cuadrada quedando definidas las operaciones realizadas en este bloque como se describen mediante el diagrama de la Fig. 3.21.

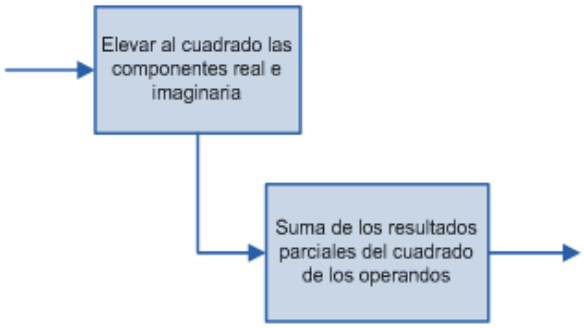


Fig. 3.21 Flujo de operaciones para el cálculo del módulo.

Este bloque recibe como entradas las componentes real e imaginaria del resultado del algoritmo, se hace el cálculo del cuadrado de las componentes por separado, utilizando dos multiplicadores para la parte real y dos multiplicadores para la parte imaginaria generando resultados con un tamaño de palabra de 32 bits, a continuación se hace la suma de estos resultados parciales, este es un componente completamente combinatorio, es decir, que evalúa las operaciones en todo momento.

Tomando en cuenta que la salida de este módulo es un vector de 32 bits y que la lectura de los bancos de memoria en cada flanco de ascenso del reloj arroja dos datos (*a* y *b*) es necesario hacer diferencia para

que ambos operandos no vayan al bloque del módulo al mismo tiempo, esto se hace mediante un multiplexor de datos que permite tomar un dato a la vez.

La señal de selección del multiplexor es el resultado de la operación *and* entre la señal de habilitación del bloque de módulo generada por la unidad de control por etapa y la señal de reloj *clk*, de tal manera que cuando *eg = 1 && clk=1* se obtiene el módulo de *a* y cuando *eg=1 && clk=0* se obtiene el módulo de *b*, esto permite dos cosas, la primera es hacer que el bloque de cálculo de módulo reciba datos mientras *eg* esté habilitada ya que cuando se interrumpa el funcionamiento de la arquitectura el valor de la señal de habilitación también será interrumpido y el resultado de la operación *and* siempre estará señalando al dato *b* de esta manera no se harán más operaciones mientras esté deshabilitado.

Cabe mencionar que la otra opción para hacer esta distinción entre los operandos es colocar *flip-flop* activado por una señal de reloj alterna a *clk* que tenga una frecuencia del doble de *clk* sin embargo, no se optó por esta solución puesto que se dio prioridad a mantener la simplicidad de la arquitectura y regir su funcionamiento por una sola señal de reloj general, con esta medida se obtienen dos resultados por cada ciclo de reloj, un resultado en el nivel alto de la señal de reloj y otro en el nivel bajo.

Finalmente, se puede observar que la ruta de datos no tiene mayores cambios en la implementación de esta etapa y que los módulos de control que a continuación se describen son compatibles con este.

3.8 Unidad de control por etapa

Este componente se encarga de generar las señales que se requieren para sincronizar el funcionamiento de las unidades descritas anteriormente con base en la señal de reloj general de la arquitectura (*clk*), la señal de *reset* (*clr*) y la señal de habilitación de la etapa (*EN_C*) proveniente de la unidad de control general, estas señales son

- *we*: Se encarga de indicar la tarea (lectura/escritura) en la que trabajarán las memorias de los bancos de esa etapa en específico, así como de seleccionar las direcciones de los multiplexores sobre las cuales se realizará la operación.
- *sm*: Esta señal se utiliza como selectora para los multiplexores de los datos de salida de las memorias.
- *eg*: Indica el momento a partir del cual se pueden generar las direcciones de una etapa, el momento en el que se comienzan a generar los coeficientes complejos y que a su vez cumple con la tarea de habilitar los registros que se encuentran a las entradas de las mariposas. Esta señal es enviada al

bloque generador de direcciones como *EN_GD*, al bloque generador de coeficientes complejos como *EN_FG* y como la señal de habilitación de los registros de la mariposa como *EN_M*.

De forma general la entidad se muestra en la Fig. 3.22.

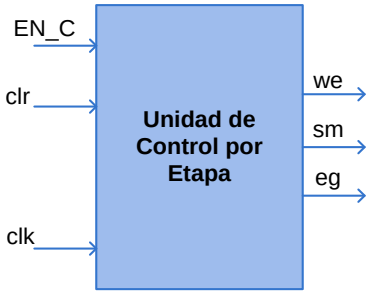


Fig. 3.22 Entidad de la unidad de control por etapa.

Dado que las señales mencionadas sirven para controlar tareas diferentes, y tomando en cuenta que se está utilizando una metodología *top-down*, se decidió diseñar un componente diferente para cada una de ellas para finalmente integrarlos y obtener la unidad de control por etapa completa, la Fig. 3.23 muestra un diagrama de bloques interno de la unidad de control por etapa.

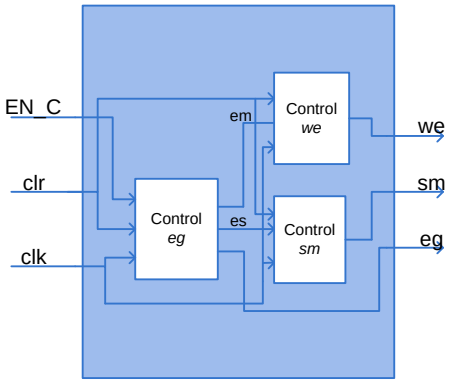


Fig. 3.23 Diagrama de bloques de la unidad de control por etapa.

3.8.1 Control para la señal *we*

Como se mencionó anteriormente, esta señal se encarga de indicar la función sobre la cual el bloque de memoria de datos trabajará durante determinado tiempo, para el diseño del circuito que genera el comportamiento de la señal se tomaron en cuenta las siguientes consideraciones.

- El primer banco de memoria en donde se escriben los datos es en el *Banco_A*
- Al habilitarse la señal *clr*, la señal *we* debe comenzar indicando la escritura de los bancos de memoria.

- La tarea de las memorias comienza con la escritura de los datos en el *Banco_A*
- La tarea de escritura de los *N* registros de un banco de memorias se hace en *N/2* ciclos de reloj debido a las características de las memorias de datos.
- Después de haber finalizado la escritura de los datos en el tiempo *t* para el *Banco_X*, en el tiempo *(t+1)* la operación del banco debe cambiar a lectura y lo mismo de lectura a escritura.
- Este intercambio de tarea se hace periódico durante tiempo indefinido siempre y cuando la señal de habilitación se encuentre activa.
- Si la señal de habilitación se interrumpe por algún motivo, la señal *we* debe permanecer en el último estado en el que se encontraba para que una vez que *em* se habilite de nuevo, la tarea continúe a partir del punto en donde se quedó.
- En consideración con los multiplexores de direcciones, tomar en cuenta que las direcciones se leen en tiempo de *preset* cada ciclo de reloj ascendente.

Con base en las características enunciadas anteriormente se puede inferir que el comportamiento de la señal *we* depende de dos contadores, uno que lleve el control del momento del cambio de operación y otro que genere el cambio. Es por eso que la máquina de estados para la señal *we* controla el comportamiento de las señales de habilitación de ambos contadores para trabajar con las siguientes características, en la Fig. 3.24 se muestra la carta ASM que describe el control de las señales de los contadores.

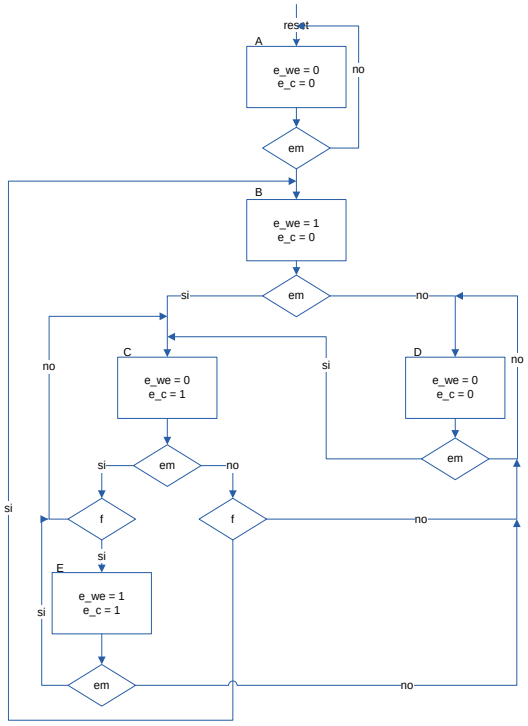


Fig. 3.24 Carta ASM para el control de la señal *WE*.

- Contador *ModN/2:* Este contador tiene la función de generar una señal de aviso para el intercambio de la operación que se realiza en las memorias, esto mediante la generación de una bandera de aviso identificada como *f,* mientras la bandera se encuentre en 0 lógico, la señal *we* se mantendrá en el último estado, de manera contraria, la señal *we* se complementará para indicar la operación opuesta. De manera general, consiste en un contador *ModN/2* ascendente que incrementa su cuenta por cada flanco descendente de reloj y siempre y cuando la señal de habilitación tenga un valor de 1 lógico de tal forma que al llegar a la cuenta máxima la bandera *f* es activada, cuando la señal de habilitación se interrumpe, la cuenta se detiene, permitiendo así reanudar las operaciones y completar el ciclo de la operación. La entidad que describe a este contador se muestra en la Fig. 3.25.

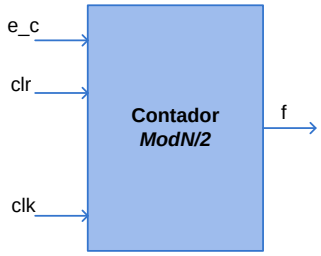


Fig. 3.25 Entidad para el contador ModN/2.

- *Contador WE:* Este contador se encarga de la generación de la señal *we* con base en la señal de habilitación de conteo *e_we* que a su vez depende de la señal de habilitación de la máquina de estados *em* y la bandera *f* generada por el contador *ModN/2*. Dado que las entradas de las memorias pasan primero por un registro y después a las memorias, y que las memorias están configuradas para realizar las operaciones en flancos de reloj ascendentes, es conveniente contar con *we* tiempo antes de que ésta sea requerida puesto que es una señal que está en sincronía con el reloj de la arquitectura, esto significa que la señal *we* es afectada en los flancos de reloj descendentes. Con el fin de cumplir con el requerimiento de comenzar la operación en un modo de escritura para el *Banco_A* de memorias, este contador tiene la característica de inicializar la señal *we* en 1 al momento de un evento en *clr*. En la Fig. 3.26 se observa un diagrama de bloques de las unidades requeridas para el control de la señal *we.*

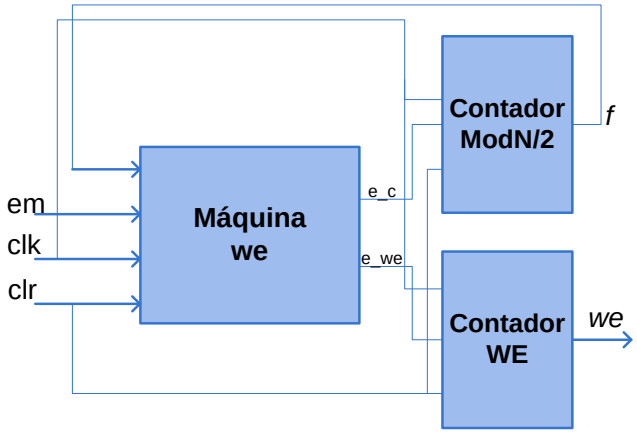


Fig. 3.26 Diagrama de bloques del componente de control de *we*.

Con base en la carta ASM de la Fig. 3.24, se generó la máquina de estados de la Fig. 3.27.

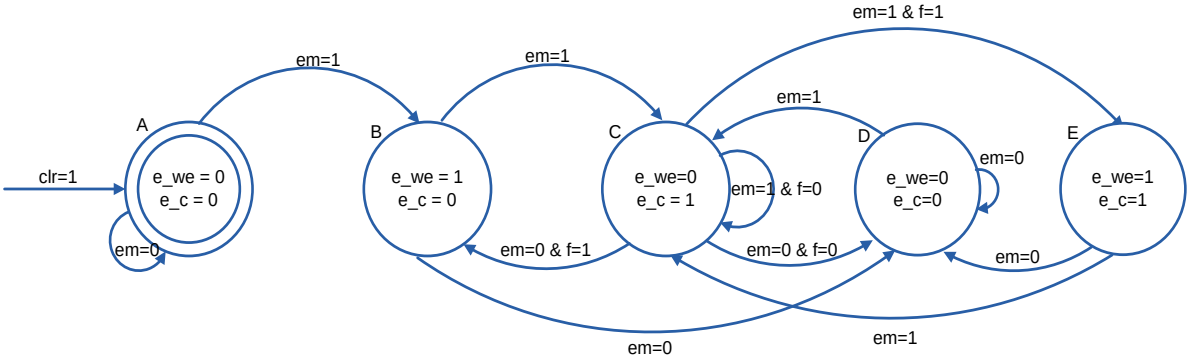


Fig. 3.27 Máquina de estados para el manejo de la señal *we*.

A continuación se hace la descripción de las operaciones realizadas en cada uno de los estados.

- Estado A: Este es el estado inicial de la máquina, se llega a él por una activación en la señal *clr* que es una entrada común para la arquitectura. Al presentarse un evento de ascenso en la señal *clr* se marca como valor inicial a los habilitadores *e_we*, y *e_c* de los contadores en 0 lógico, esto es para que *we* mantenga el valor de 1 lógico que es con el que lo inicializa el *ContadorWE* después de un evento de *reset* por tiempo indefinido hasta que exista un cambio en la señal de entrada *em.*
- Estado B: En este estado se pone en 1 lógico el valor de la señal *e_we,* esto permite habilitar el conteo en el componente *ContadorWE* lo que provoca un cambio de 1 a 0 lógico en la señal *we,* esto significa que se ha hecho un cambio de tarea de escritura a lectura en el *Banco_A*¸ y al revés para el *Banco_B.*

- Estado C: En este estado se pone en un nivel de 0 lógico el valor de *e_we* lo que permite mantener el valor de la operación en *we* y también se pone en 1 lógico el valor de la señal *e_c* para comenzar con la cuenta de los ciclos que se ha estado en la misma operación.
- Estado D: En este estado se mantiene el valor en el que esté la señal *we* mediante la inhabilitación de las señales *e_we* y *e_c*, a este estado se llega únicamente cuando la señal de habilitación *em* de la máquina para *we* se ha interrumpido, esto con la finalidad de reanudar operaciones en el mismo punto en el que fueron interrumpidas.
- Estado E: En este estado se genera un cambio en la señal *we* mediante la habilitación de la señal *e_we* y comienza la cuenta del tiempo que debe transcurrir antes de darse otro cambio con la habilitación del contador *ModN/2* por la señal *e_c*.

Finalmente, dos aspectos más a considerar, el primero es que dado que las señales de habilitación de los contadores son síncronas con la señal de reloj es importante no hacer transiciones en flancos de reloj descendentes pues como ya se mencionó anteriormente, la señal *we* requiere ser generada en esos flancos, es por eso que las transiciones de la máquina de estados se hacen en flancos de reloj ascendentes para tener listos en las señales de habilitación los valores antes de que se requiera saber de ellos en los contadores. El segundo aspecto es que observando la Fig. 3.26, la señal que habilita el funcionamiento de esta máquina no es la señal que proviene de la unidad de control general, si no la señal generada por el bloque de control de *eg*¸ esto se detallará más adelante.

3.8.2 Control para la señal *sm*

Esta señal es la entrada selectora de los multiplexores a donde llegan los buses de datos de salida de las memorias de *Banco_A* y *Banco_B*, es necesario hacer diferencia entre ellos puesto que hay datos en los buses incluso cuando se está realizando la operación de escritura. Para el diseño del circuito encargado de generar esta señal se tomaron en cuenta las siguientes consideraciones.

- Las memorias intercambian de función cada *N/2* ciclos de reloj.
- Los datos que se seleccionarán como salidas de los multiplexores, serán los que provienen del banco de memoria que se encuentre en el modo de lectura en ese momento.
- El multiplexor es un elemento completamente combinatorio.
- La lectura de las memorias se hace cada flanco de ascenso.
- Después de haber seleccionado las salidas con la señal *sm* con un valor de 1 lógico en el tiempo *t* el último dato de la lectura del *Banco_A*, en el tiempo *t+1* se seleccionan los datos de salida de las memorias del *Banco_B* con la señal *sm* con un valor de 0 lógico y viceversa.

- Esta señal se vuelve periódica mientras la señal *es* se encuentre en 1 lógico.
- Si la señal de habilitación se interrumpe por alguna razón, el valor de *sm* debe permanecer estático en el último valor y una vez que se restablezca la señal de habilitación, la operación que se estaba realizando debe comenzar desde el punto en el que se interrumpió para así completar el ciclo de la selección y evitar fallo en la selección de los datos.

De igual forma que para la señal *we*, por las características enunciadas, el diseño del circuito para control de la señal *sm* también considera dos contadores, uno que permite llevar el control del momento del cambio de la operación y otro que genere ese cambio, a continuación se describe la secuencia de activación y desactivación de las señales mediante la carta ASM correspondiente que se ve en la Fig. 3.28.

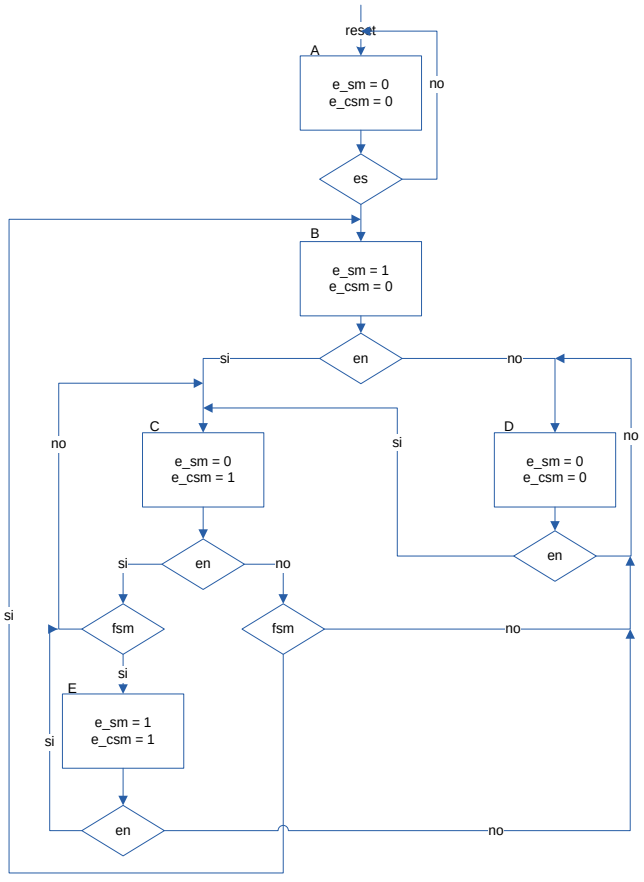


Fig. 3.28 Carta ASM para el control de la señal *sm*.

- *Contador SM_ModN/2*: Este contador tiene la función de generar una señal de aviso para el intercambio de la selección entre los datos de entrada del multiplexor, esto derivado del requerimiento en donde la señal debe ser periódica en *N* ciclos. El aviso se proporciona mediante la bandera *fsm* generada

cuando el componente llega a la cuenta máxima. Mientras *fsm* se encuentre en 0 lógico la señal *sm* mantendrá su valor, de manera contraria, la señal *sm* tomará el valor complementario, es aquí en donde se hace el intercambio entre los datos seleccionados en los multiplexores. De forma general, este componente es un contador *ModN/2* ascendente que incrementa su cuenta por cada flanco ascendente de reloj y siempre y cuando la señal de habilitación *e_csm* tenga un valor de 1 lógico, de lo contrario, la cuenta se detiene hasta que *e_csm* vuelve a ser activada.

- *Contador SM:* Este contador se encarga de la generación de la señal *sm* en función de la señal de habilitación de conteo *e_sm* y la bandera *fsm* que se genera con *Contador SM_ModN/2*. Cuando *e_sm* se encuentra activa y el valor de *fsm* es 1 lógico entonces se efectúa un cambio en la señal *sm*. Tomando en cuenta que los multiplexores son elementos completamente combinatorios, no existen retardos y a diferencia de la señal *we*, no es necesario contar con ella específicamente tiempo antes de que comience a ser utilizada puesto que sólo hace la diferenciación entre los datos de una memoria y otra, es por esto que basta con generar los cambios al mismo tiempo que se presentan los datos en los buses de salida una vez que se ha hecho la primer lectura de las localidades.

De igual manera que en la señal *we*, las señales que deben ser controladas para la generación de *sm* son las señales que permiten la habilitación de ambos contadores, esto se realiza mediante una máquina de estados. En la Fig. 3.29 se observa un diagrama de bloques de la interacción de estos componentes.

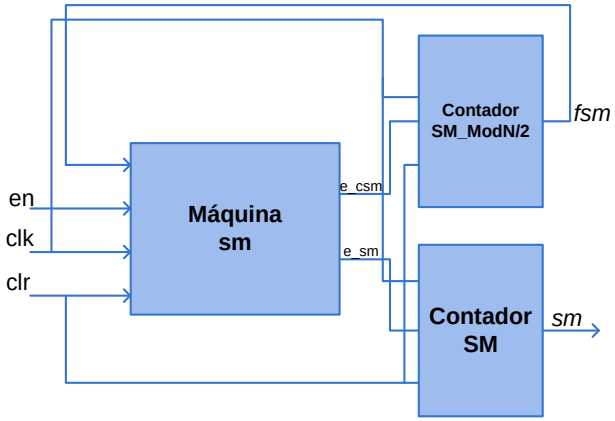


Fig. 3.29 Diagrama de bloques de la unidad para el control de *sm*.

El funcionamiento de la máquina de estados que controla la habilitación de los contadores se puede observar en la Fig. 3.30 que se derivó de la carta ASM de la Fig. 3.28.

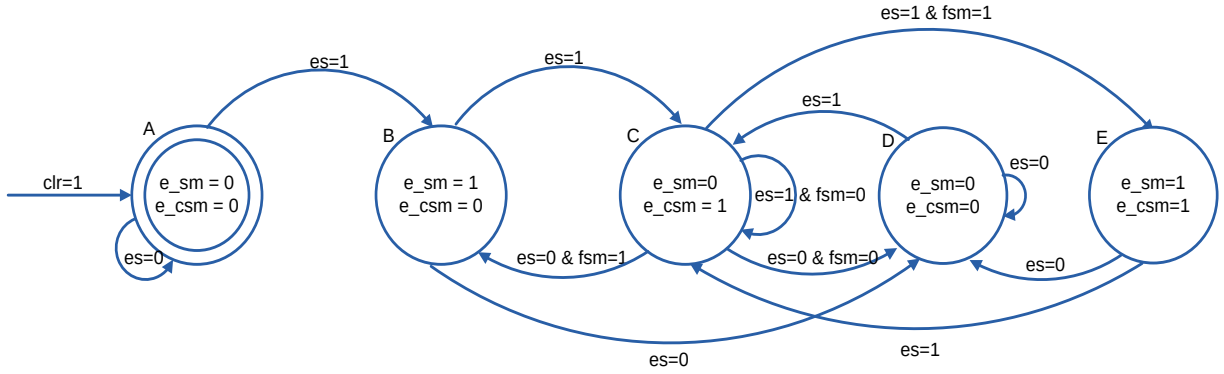


Fig. 3.30 Máquina de estados para el control de *sm.*

Describiendo la finalidad de cada uno de los estados se tiene

- Estado A: Se llega a él cuando se presenta un evento en la señal *clr* en la arquitectura, aquí se fijan como puntos de partida para las señales de habilitación de los contadores valores en 0 lógico, el *reset* provoca que el *ContadorSM* ponga en la señal *sm* el valor de 0 lógico y *SM_ContadorModN/2* inicie la cuenta en ceros.
- Estado B: En este estado se genera la habilitación de la señal *e_sm* lo que permite que se haga un cambio en la señal *sm* para la selección de los datos en los multiplexores, cambiando entre los datos del *BancoB* al *BancoA.*
- Estado C: En este estado la señal *e_sm* vuelve a ser afectada pero ahora con la finalidad de que el valor en la señal *sm* permanezca con el mismo valor, aquí mismo se habilita la señal *e_scm* para comenzar con la cuenta de la cantidad de ciclos en los que ha permanecido en la misma operación. La máquina permanece en este estado hasta que la bandera *fsm* un evento ascendente.
- Estado D: A este estado se llega si y sólo si hay una interrupción en la señal de *es*, esto significa que se ha hecho una pausa general en todas las tareas, por lo que aquí se pone en 0 lógico a la señal *e_csm* para detener la cuenta y evitar que se habilite *fsm* antes de lo requerido, y se mantiene el valor de la señal *sm* mediante la desactivación de la señal *e_sm* para el contador.
- Estado E: Este estado pone en 1 a ambas señales de habilitación de los contadores, esto con la finalidad de generar una transición en el valor de la señal *sm* y no detener la cuenta en el módulo que genera la bandera de transición, estas dos acciones provocan que *sm* se vuelva periódica en *N* ciclos.

Como ya se mencionó previamente, cada lectura de los datos de las memorias se ven reflejados en los buses de datos en flancos de ascenso de reloj, y al tratarse de elementos combinatorios, basta con que las transiciones se hagan al mismo tiempo, por lo que las señales de habilitación para los contadores deben

ser modificadas antes de que se requieran, es por eso que las transiciones para la máquina de estados se realizan en flanco de descenso del reloj. De acuerdo con la Fig. 3.29, la señal que habilita la generación de *sm* proviene del componente que genera también la señal para la habilitación de *we* y de *eg*. Este componente se describe a continuación.

3.8.3 Control para la señal *eg* y la habilitación de *we* y *sm*

Este componente se encarga de distribuir las señales de habilitación para comenzar con la generación de las señales *we* y *sm* puesto que estas unidades no son habilitadas al mismo tiempo, esto es por cómo se requiere que se habilite el funcionamiento de los componentes en función de la señal de habilitación de la etapa, el orden en que la señal de habilitación de la unidad de control por etapa afecta a los bloques internos es el siguiente.

- La primera señal en adquirir el valor de la señal de entrada es *em*, que se encarga de habilitar el funcionamiento de la señal *we* para indicar la tarea que va a realizar cada banco de memoria y con las direcciones pertinentes. Esto es porque al tratarse de una arquitectura segmentada y sobre el principio de funcionamiento de la unidad que genera las direcciones, no se requiere de contar con direcciones nuevas de lectura, cuando la escritura en los bancos de memoria de la etapa actual se está realizando con las direcciones generadas por el módulo de la etapa anterior. Esta señal se habilita en flanco de descenso de la señal de reloj puesto que se pregunta por ella en la máquina de estados en flanco de ascenso. Una vez que la señal *em* ha sido activada en flanco negativo, el cambio de escritura a lectura en las memorias tarda un ciclo de reloj más, y se da en flanco de descenso.
- La segunda señal en adquirir el valor es *es*, que se asigna leyendo la señal *em* a través de un registro activado en flanco de ascenso, esto provoca finalmente que la señal selectora de los multiplexores de datos cambie de valor medio ciclo de reloj después de que la señal *we* lo hizo, puesto que la lectura de la direcciones se hace el tiempo de *preset* y los datos se presentan a las salidas de las memorias en ese ciclo de reloj y es cuando se pueden comenzar a seleccionar.
- Finalmente, la señal *eg*, quien habilita el funcionamiento de la unidad de generación de direcciones, la generación de los coeficientes complejos, la carga y retención de los datos en los registros de entrada de la mariposa y los multiplexores de la unidad de módulo toma el valor de la señal *em* un ciclo de reloj después. La activación de la generación de direcciones se hace un ciclo de reloj después de que se ha activado *em* pues la primer dirección con la que se realiza la operación indicada por *we* ya está en el bus de direcciones por ser el valor con el que se inician después de un evento en la señal *clr* y no conviene generar direcciones antes, puesto que éstas son tomadas en

tiempo de *preset* y la primer operación que se realiza después del cambio en *we* se hace tomando la dirección presente.

El componente para la generación de los coeficientes complejos se comporta de la misma manera, es decir, en el bus de salida de datos se encuentra el coeficiente que fue direccionado después de un evento en la señal de *reset* entonces al activarse al mismo tiempo que el generador de direcciones, los coeficientes complejos se obtienen al mismo tiempo que los datos de las memorias.

Finalmente, el contar con los datos necesarios para la evaluación de la mariposa en cada flanco de ascenso del reloj obliga a que los registros que se tienen a la entrada de la mariposa sean activados mediante la misma señal pero en flanco de descenso, esto quiere decir que los datos se toman a la mitad de su periodo y permanecen como entradas de la mariposa durante un ciclo de reloj completo permitiendo así su evaluación, si esta señal se interrumpe entonces los datos se mantienen constantes hasta que se habilita de nuevo.

3.9 Unidad de control general

Este bloque, como su nombre lo indica se encarga de coordinar el funcionamiento de cada una de las etapas de la arquitectura mediante la generación de las señales de habilitación para las unidades de control por etapa. Como ya se mencionó anteriormente, esta propuesta de arquitectura se caracteriza por estar conformada de log_2N etapas, una por cada etapa de la FFT y por consecuencia una señal de habilitación por cada etapa.

Para esta propuesta en específico se requieren de $log_2N + 2$ señales de habilitación puesto que la unidad de *bit-reverse* no está dentro de una etapa pero igualmente requiere sincronización con el funcionamiento de las demás unidades y el bloque en donde se calcula el módulo de los resultados también requiere ser habilitado sobre las mimas condiciones.

La unidad de control general trabaja con base en las señales *clk*, *clr*, y *EN_GRAL* que son entradas, las primeras dos ya descritas anteriormente y que son comunes a todos los bloques, *EN_GRAL* es la señal de habilitación general de la arquitectura, cuando ésta adquiere el valor de 1 lógico comienza el proceso para la activación de cada una de las etapas en la arquitectura, estas señales de activación son mapeadas en un vector de $log_2N + 2$ bits, un bit por etapa, siendo este vector la salida de la unidad, la Fig. 3.31 muestra la entidad del componente implementado.

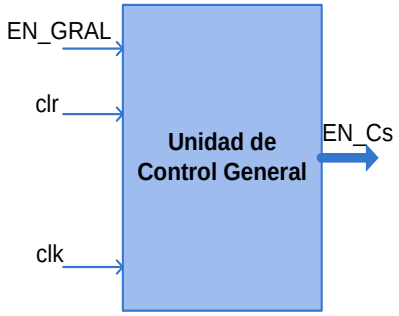


Fig. 3.31 Entidad de la Unidad de control general.

La unidad de control general es necesaria debido a que cada una de las etapas se activan en diferentes momentos haciendo referencia al nombre de la arquitectura pues éstas se habilitan en cascada, si no fuera así, los resultados obtenidos serían incorrectos simplemente por las dependencias de datos que existirían entre cada etapa.

Para el diseño de la unidad de control general se tomaron en cuenta los siguientes aspectos:

- Se requiere una señal de habilitación por cada bloque de etapa de la arquitectura
- Se requiere una señal de habilitación para la unidad de bit-reverse.
- Se requiere una señal de habilitación para el bloque de módulo de resultados.
- Las señales de habilitación por etapa deben ser síncronas a una señal de reloj.
- El tiempo transcurrido entre la habilitación de las etapas *n* y la *n+1* debe ser el mínimo necesario para completar la escritura de los bancos de memoria de la etapa *n.*
- La deshabilitación de las señales, por interrupción de la señal *EN_GRAL*, debe realizarse en el mismo instante para todas las etapas, esto con el fin de detener toda operación que se esté realizando en las diferentes etapas y así poder comenzar desde el punto en el que se interrumpieron y no afectar el flujo de las operaciones ni el resultado obtenido.
- Si ocurre una interrupción en la señal de habilitación general antes de que todas las señales de habilitación para las etapas hayan sido generadas y se mantengan constantes, se deben deshabilitar todas las etapas que ya se encuentran en funcionamiento y cuando la habilitación ocurra nuevamente, se debe reanudar la operación justo en el estado en donde se interrumpieron.
- La escritura del primer banco de memorias se hace en *N/2* ciclos de reloj a partir de que se habilita la señal para el módulo de *bit-reverse*.
- La escritura de los bancos de memorias a partir de la segunda etapa se realiza *N/2+1* ciclos puesto que se toma un ciclo de reloj más para el procesamiento de los datos a partir de la primera lectura, es decir, mientras se procesan los datos de la primer lectura, se hace la segunda lectura y así

sucesivamente, es por eso que la dirección de escritura de una etapa se genera desfasada a la dirección de lectura de la misma.

A partir de las características enumeradas anteriormente, se puede inferir que se requiere una máquina de estados que sea capaz de garantizar la correcta y completa escritura en los bancos de memoria y tomando en cuenta los últimos puntos se puede destacar que nuevamente se requieren de módulos contadores, un contador *ModN/2* y un contador *ModN/2+1*, estos componentes con la finalidad de generar señales que permitan la activación de las etapas de la arquitectura en modo de cascada. Nuevamente, se diseñó una máquina de estados encargada de controlar las señales de habilitación de los contadores, la Fig. 3.32 muestra la carta ASM para el diseño del autómata.

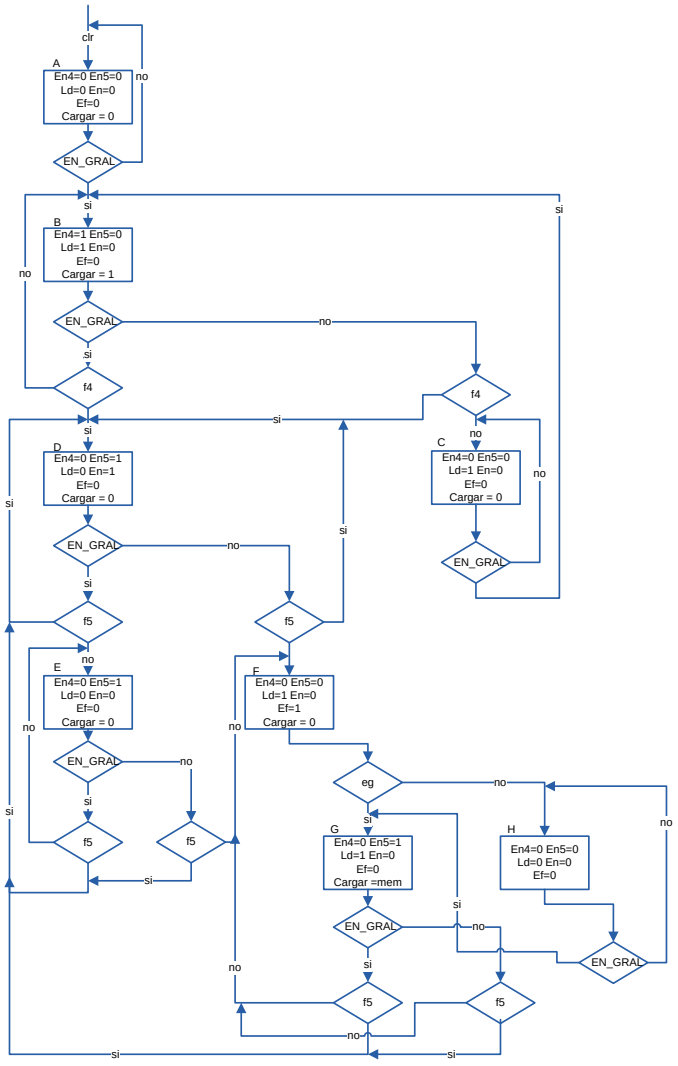


Fig. 3.32 Carta ASM para la unidad de control general.

En la Fig. 3.33 se muestra el autómata resultante que describe la lógica de control para el bloque.

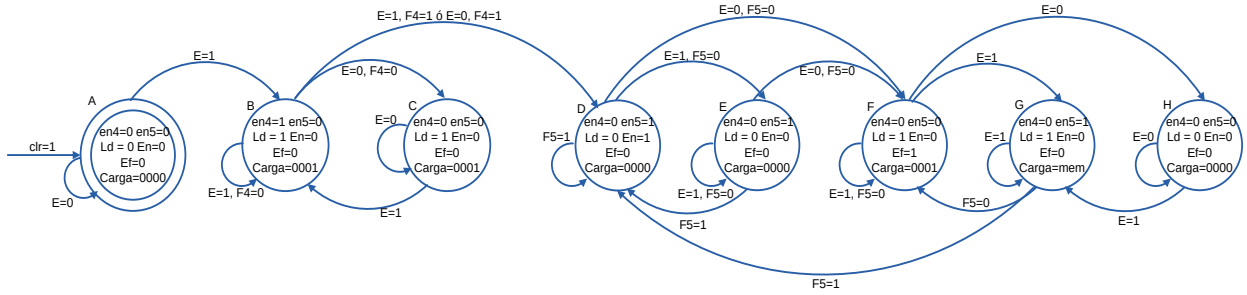


Fig. 3.33 Máquina de estados para la unidad de control general.

Se observa que en cada estado del autómata se hace la activación o desactivación de seis señales diferentes, el bloque finalmente quedó conformado como se muestra en la Fig. 3.34.

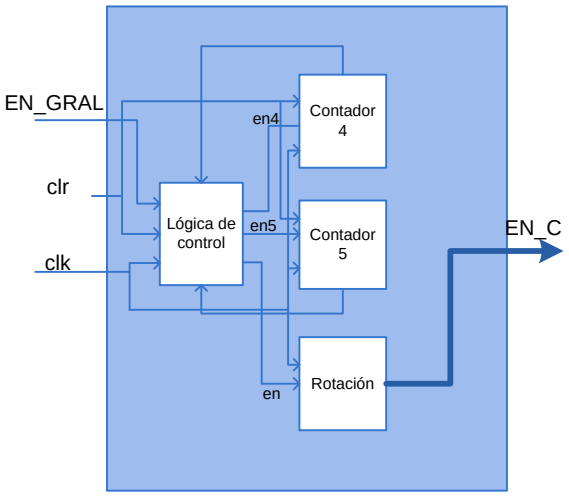


Fig. 3.34 Diagrama de bloques de la unidad de control general.

El diagrama de la Fig. 3.34 muestra como la unidad de control general se compone de cuatro bloques, el bloque modelado por el autómata de la Fig. 3.31, dos contadores y un bloque de rotación, en conjunto estos se encargan de generar el vector de bits que habilitan a cada una de las etapas de la arquitectura.

- *Contador4*: Este bloque se encarga de generar un evento positivo en una señal que sirve como bandera al llegar a la cuenta de *N/2*, esta bandera identificada como *f4* es indicadora de una transición entre estados en el autómata.
- *Contador5*: Este bloque se encarga de generar un evento positivo en una señal utilizada como bandera al llegar a la cuenta de (*N/2+1*), esta bandera identificada como *f5* es indicadora de una transición entre estados en el autómata.

- *Rotación*: Este bloque se encarga de generar el vector de bits de habilitación con base en las señales de reloj, *reset* y de habilitación producida por la lógica de control de la etapa. Habilita uno a uno los bits del vector de salida en flanco descendente de reloj mientras la señal de habilitación esté activada y cuenta con la característica de comenzar su funcionamiento a partir de la carga de un vector de entrada, esto es de utilidad cuando la señal *EN_GRAL* es interrumpida y la habilitación de las etapas debe continuar en donde se quedó por última vez.

La lógica de control en cada uno de los estados se describe a continuación:

- Estado A: Al igual que en los demás autómatas, éste es el estado inicial, al cual se llega por medio de un evento presente en *clr* y es en donde se dan valores iniciales a las señales para la interacción de los componentes que conforman la unidad de control general.
- Estado B: A este estado se llega partiendo de A siempre y cuando se presente un evento positivo en la señal de habilitación general *EN_GRAL* y es en donde se habilita la carga inicial del registro de corrimiento la cual consiste en cargar el valor correspondiente a la habilitación de la primer etapa siendo éste 1 en binario y habilitando también la señal que permite hacer la carga del registro y la señal *en4* que permite iniciar la cuenta en el *Contador4*.
- Estado C: A este estado se llega a partir del estado B y una interrupción en *EN_GRAL*, es decir, cuando se ha deshabilitado el funcionamiento de la arquitectura entonces se tiene que deshabilitar la primera etapa por lo que se hace una carga en el registro de corrimiento mediante la señal *ld=0* y *carga=0*.
- Estado D: En este estado, si la señal *EN_GRAL* se mantiene habilitada continuamente es aquí en donde se efectúa el corrimiento de los bits de la salida mediante la habilitación del registro de corrimiento por la señal *en* y la habilitación de *Contador5* que permite la periodicidad de la habilitación de una nueva etapa cada (*N/2+1*). Cada vez que se llega a este estado la señal *en* es habilitada lo que permite generar un corrimiento en el vector de salida.
- Estado E: En este estado sólo permanece activada la señal *en5* que permite que el *Contador5* de periodicidad continúe habilitado, al desactivar la señal *en* se permite que no sea activada una nueva etapa sin antes haber concluido con todas las operaciones requeridas, esto lo señala la bandera generada por *Contador5*, es por eso que *en5* permanece activada.
- Estado F: A este estado se llega una vez que se presenta una interrupción en *EN_GRAL* y es aquí en donde se hace un respaldo del valor del vector de salida puesto que cuando la señal sea nuevamente habilitada se requerirá que las operaciones comiencen tal cual se encontraban

trabajando antes del evento en *EN_GRAL*, esto se hace por medio de la activación de la señal *ef* que es quien indica que el valor del vector se almacena en una variable temporal *mem.*

- Estado G: Una vez que *EN_GRAL* ha sido habilitada de nuevo en este estado se indica que el registro de corrimiento debe hacer una carga con el valor almacenado previamente en la señal *mem* para comenzar de nuevo con la secuencia de generación.
- Estado H: En este estado se desactivan todos los procedimientos que se estén realizando en la unidad y se queda en espera de una nueva habilitación en *EN_GRAL*.

El diseño de la unidad de control general como se describió anteriormente permite realizar un escalamiento de la arquitectura rápidamente puesto que no es necesario agregar o eliminar estados de la lógica si se quiere evaluar más o menos puntos dando ventajas de reconfiguración a la arquitectura.

Capítulo 4. Implementación y pruebas funcionales

4.1 Pruebas del bloque de memoria de datos

- **Pruebas para un módulo de memoria M4K**

Como se describió en el apartado de diseño, para el bloque de memorias de datos se utilizaron los bloques de RAM dedicados del FPGA identificados como *M4K*, para comprobar su funcionamiento se realizó la simulación funcional con la herramienta ModelSim Altera ingresando como entradas los datos que se muestran en la Tabla 4.1 obteniendo así la imagen de simulación que se puede observar en la Fig. 4.1.

Tabla 4.1 Vectores de prueba para la simulación del módulo de memoria M4K.

Vectores de prueba para la simulación del modulo M4K				
WE	**dir_a**	**dir_b**	**dato_a**	**dato_b**
1	0	1	173	59
1	2	3	174	60
1	4	5	175	61
1	6	7	176	62
0	0	1	173	59
0	6	7	176	62
0	4	5	175	61
0	2	3	174	60

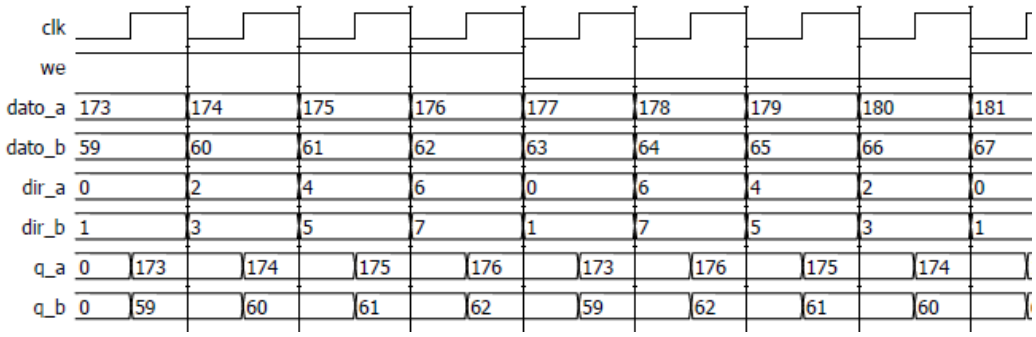


Fig. 4.1 Simulación de la escritura de un bloque de memoria M4K.

El componente de memoria simulado tiene las características descritas en el apartado de diseño y corresponde a la entidad de la Fig. 3.5 el cual describe una memoria de doble puerto de lectura y escritura, en este caso de únicamente ocho localidades. De acuerdo con la simulación y los vectores de prueba, la primera operación que se realiza sobre el elemento de memoria es la escritura de los vectores de prueba

en las localidades correspondientes para continuar con la lectura de los mismos en donde se obtienen los resultados de la segunda parte de la simulación, se puede observar que las operaciones en la memoria se realizan en flanco de ascenso de la señal de reloj.

La simulación permite ver que las operaciones en la memoria se realizan en flanco de ascenso de la señal de reloj es por eso que los vectores de las salidas están desfasados medio ciclo de reloj con las entradas dado que las entradas son modificadas cada flanco de descenso de *clk*. Mientras se está efectuando la operación de lectura de la memoria, los datos que están presentes en los buses de entrada son ignorados.

Se observa también que una vez que la señal *we* cambia de estado, al siguiente flanco de ascenso comienza la operación señalada (en este caso lectura), este aspecto fue importante en el diseño puesto que esta señal es generada por un circuito secuencial y que debe tener lista la señal de operación (sin oscilaciones) una vez que se requiera hacer el cambio de operación.

- **Pruebas para el bloque de bancos de memoria**

Una vez que se corroboró el funcionamiento y comportamiento de los módulos de memoria dedicados se implementó un banco de memorias como el que se muestra en la Fig. 3.4 y se utilizaron los vectores de prueba de la Tabla 4.2 para verificar el funcionamiento de la ruta de datos que se planteó en el diseño previo y se obtuvo la simulación de la Fig. 4.2.

Tabla 4.2 Vectores de prueba para la simulación de un banco de memoria.

Vector	we	dir_aw	dir_bw	dir_ar	dir_br	sm	dato_ain	dato_bin
1	1	0	1	6	7	0	214	113
2	1	2	3	4	5	0	216	115
3	1	4	5	2	3	0	218	117
4	1	6	7	0	1	0	220	119
5	0	0	1	6	7	1	222	121
6	0	2	3	4	5	1	224	123
7	0	4	5	2	3	1	226	125
8	0	6	7	0	1	1	228	127
9	1	0	1	6	7	0	230	129
10	1	2	3	4	5	0	232	131

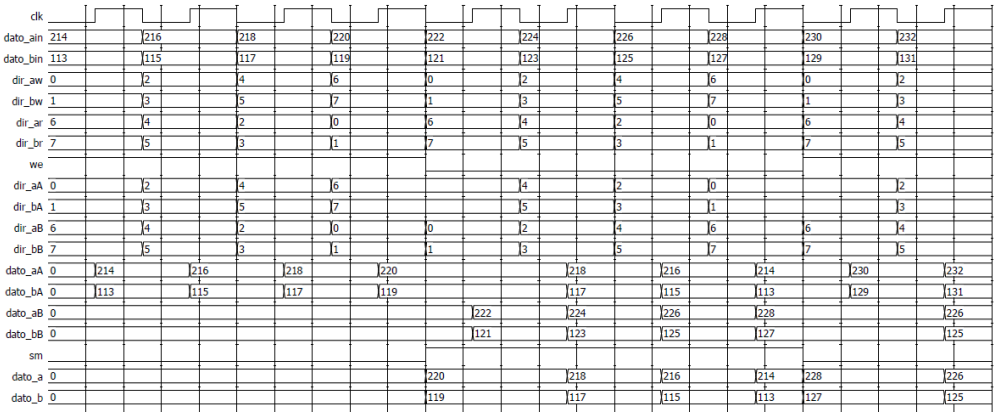


Fig. 4.2 Simulación funcional de un banco de memoria de datos.

Si se observa la simulación de la Fig. 4.2 se puede notar que sólo están incluidas dos memorias, una por cada banco de acuerdo con el diseño del capítulo 3 (en este caso Mem_Ar y Mem_Br con respecto a la Fig. 3.4), esto es debido a que una vez que se ha corroborado el correcto intercambio de funciones en estas memorias, la selección de las direcciones sobre las cuales se hacen las operaciones así como la selección de los datos de salida, el funcionamiento se puede extender a las otras dos memorias (Mem_Am y Mem_Bm) puesto que trabajan bajo el mismo principio.

Los valores de los vectores de prueba están configurados para permitir observar el comportamiento del banco de memorias haciendo el cambio entre las tareas de escritura y lectura, la simulación comienza señalando a *we* para que se ejecutará la operación de escritura en Mem_Ar y lectura de Mem_Br, esto se puede confirmar porque las señales de direcciones para Mem_Ar (*dir_aA* y *dir_bA*) toman los valores de las direcciones de escritura de entrada al banco (*dir_aw* y *dir_bw*) como lo hacen cuando el vector de entrada es el 3 de la Tabla 4.2 y *dir_aA*=4 y *dir_bA=5* en tanto que las salidas toman los valores de las señales de datos de entrada (*dato_ain* y *dato_bin*) tal que *dato_aA*=216 *y dato_bA*=117.

Por otro lado las direcciones de Mem_Br toman los valores de las direcciones de lectura (*dir_ar* y *dir_br*) tal que *dir_aB*=2 y *dir_bB*=3 y las salidas de las memoria (*dir_aB* y *dir_bB*) tienen presente un cero constante debido a que de esta forma están inicializados todos los registros de esta memoria, esta primera operación se muestra en los cuatro primeros ciclos de reloj.

Cuando *we* toma el valor de 0 lógico, como lo hace a partir de quinto ciclo de reloj¸ la tarea a ejecutarse en Mem_Ar es la lectura por lo tanto las direcciones toman el valor de las direcciones de lectura de entrada (*dir_ar* y *dir_br*) tal como lo hacen cuando el vector de prueba corresponde al 7 de la Tabla 4.2

de modo que *dir_aA*=2 y *dir_bA*=3 y en las salidas de datos se presentan los datos de lectura de acuerdo con las direcciones de entrada por lo que *dato_aA*=216 y *dato_bA*=115, mientras que para Mem_Br se está ejecutando la tarea de escritura por lo que para las direcciones de entrada toman los valores de las direcciones de escritura de entrada, entonces se tiene que *dir_aB*=4 y *dir_bB*=5 mientras que las salidas toman los valores de las entradas de datos teniendo que *dato_aB*=226 y *dato_bB*=125.

Como se explicó en el apartado de diseño, a la salida de los bancos de memorias se tienen presentes multiplexores que tienen cómo tarea hacer la selección de los datos que se van a evaluar en el bloque de mariposa mediante la señal *sm*. Esto se puede observar a lo largo de toda la simulación, tomando un vector de entrada en específico, por ejemplo el 5 de la Tabla. 4.2 en donde se indica que se hace lectura de Mem_Ar las señales de salida del banco toman los valores de las salidas de dicha memoria puesto que *sm*=1 y entonces *dato_a*=220 y *dato_b*=119 mientras que para el vector de entrada 9 *sm*=0 y *dato_a*=228 y *dato_b*=127 dado que la memoria que está en operación de lectura es Mem_Br. Con la simulación anterior se puede concluir que el bloque de memoria de datos funciona adecuadamente con base en el diseño realizado.

4.2 Pruebas del bloque *bit-reverse*

De acuerdo con el apartado de diseño, este bloque se encarga de generar las direcciones para la escritura de los datos de entrada de la arquitectura, una vez que se describió el bloque con VHDL se hicieron pruebas del diseño mediante la simulación funcional y los vectores de prueba que se muestran en la Tabla 4.3 tomando en cuenta que cada fila corresponde al valor del vector en flanco de ascenso de *clk*.

Tabla 4.3 Vectores de prueba para la simulación de bloque generador *bit-reverse.*

Vector	clr	EN_BR
1	0	0
2	1	0
3	0	1
4	0	1
5	0	1
6	0	1
7	0	0
8	0	1
9	0	1
10	0	1

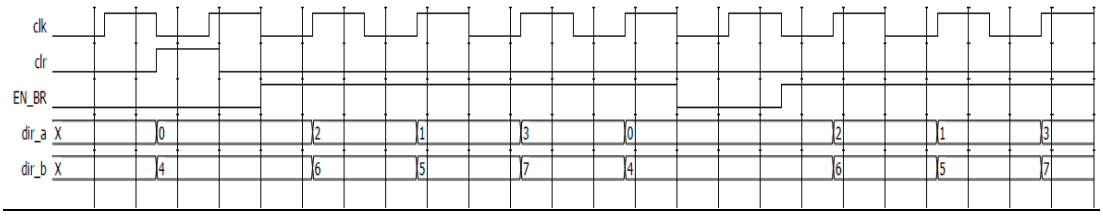


Fig. 4.3 Simulación funcional del bloque *bit-reverse.*

De acuerdo con los vectores de prueba de la Tabla 4.3, la simulación comienza con ambas señales de control (*clr, EN_BR*) deshabilitadas, lo que provoca que las salidas del componente presenten valores no definidos, esto se puede observar en la simulación durante el primer ciclo de reloj.

La primera señal de control en tomar un valor diferente es *clr* cuando el vector de entrada es el 2 de la Tabla 4.3, con esto se activa el *reset* del componente generando que la cuenta se inicialice de modo que *dir_a*=0 y *dir_b*=4, en seguida, con el vector de prueba 3 se observa que al siguiente flanco de reloj ascendente *EN_BR* toma el valor de 1 lógico y la cuenta comienza, entonces *dir_a*=2 y *dir_b*=6 y así sucesivamente mientras *EN_BR* se encuentra activada lo que se observa cuando los vectores de entrada corresponden del 3 al 7 hasta llegar al vector 8 en donde *EN_BR* se ve interrumpida durante un ciclo de reloj, esto altera las salidas puesto que la cuenta se detiene y no es hasta que *EN_BR* vuelve a ser activada que la cuenta comienza nuevamente a partir del número en que fue detenida.

Se puede observar que la primera señal requerida para que el componente comience su funcionamiento es la señal *clr* que es quien permite inicializar a las señales internas y dar un punto de partida a la cuenta, en este caso los valores de inicio son 0 y 1 que después de la inversión de bits son 0 y 4 para direccionar a una memoria de 8 palabras, una vez que se ha presentado un evento en esta señal se espera la habilitación de *EN_BR* para comenzar con la cuenta.

En la Fig. 4.4 se puede observar el diagrama RTL resultante de la síntesis del componente de entidad de la Fig. 3.6.

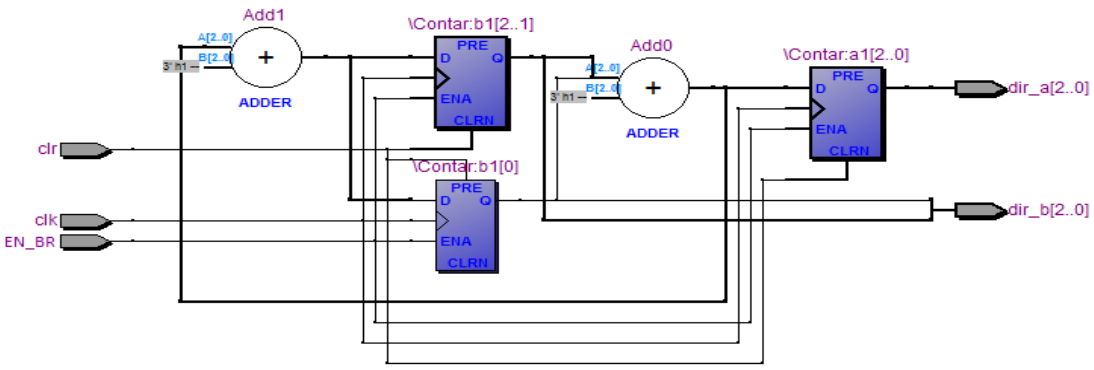


Fig. 4.4 Diagrama *RTL* del bloque *bit-reverse*.

Los resultados de síntesis del bloque para la arquitectura de 512 puntos se muestran en la Fig. 4.5.

Flow Summary	
Flow Status	Successful - Thu Jul 04 20:44:31 2013
Quartus II Version	11.0 Build 157 04/27/2011 SJ Web Edition
Revision Name	Gen_direccionesE1
Top-level Entity Name	Gen_direccionesE1
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	16 / 33,216 (< 1 %)
Total combinational functions	8 / 33,216 (< 1 %)
Dedicated logic registers	16 / 33,216 (< 1 %)
Total registers	16
Total pins	21 / 475 (4 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Fig. 4.5 Resultados de síntesis del bloque de bit-reverse para la arquitectura de 512 puntos

4.3 Pruebas del bloque generador de direcciones

Este bloque se encarga de hacer el cálculo de las direcciones sobre las cuales se hará la lectura de los datos que se encuentran guardados en las memorias de datos para su evaluación y posterior escritura en las memorias de la siguiente etapa de acuerdo con las direcciones calculadas también. Una vez que la entidad diseñada anteriormente fue descrita por comportamiento, se generó la simulación funcional con los vectores de prueba de la Tabla 4.4 obteniendo la simulación de la Fig. 4.6

Tabla 4.4 Vectores de entrada para la simulación funcional del bloque generador de direcciones de datos.

Vector	clr	EN_GD	g_wn	m_d
1	0	0	4	1
2	1	0	4	1
3	0	0	4	1
4	0	1	4	1
5	0	1	4	1
6	0	1	4	1
7	0	0	4	1
8	0	1	4	1
9	0	0	4	1
10	0	1	4	1

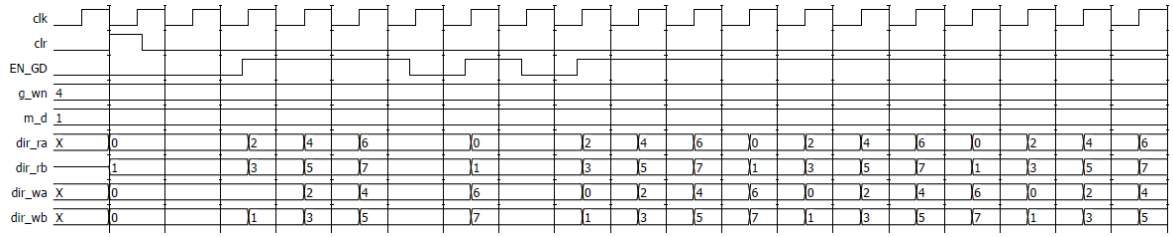


Fig. 4.6 Simulación funcional del bloque generador de direcciones.

Se observa que de igual manera que para el bloque *bit-reverse* para este bloque, el evento que genera el primer cambio en las salidas es la presencia de un valor positivo en la señal *clr* que es el *reset* asíncrono de la arquitectura en general.

Una vez que se han inicializado las salidas mediante el vector 2 se espera la habilitación de *EN_GD* que ocurre con el vector 4 para hacer el cálculo de las siguientes direcciones de lectura y al mismo tiempo liberar las direcciones de escritura con el valor anterior de las de lectura, esto se puede corroborar tomando cualquiera de los vectores de entrada del 4 al 6 en donde *EN_GD* se mantiene constante, por ejemplo, para el vector 5 *dir_ra*=4 y *dir_rb*=5 mientras que *dir_wa*=2 y *dir_wb*=3 lo que significa que las direcciones de escritura tomaron correctamente los valores que las direcciones de lectura tenían un ciclo de reloj antes y que estas fueron calculadas nuevamente.

En la simulación se observa también como una vez que la habilitación de *EN_GD* es interrumpida (vectores 7 y 9 de entrada) el cálculo de las direcciones también lo es, esto fue previsto con la finalidad de poder parar en cualquier momento el funcionamiento de la arquitectura.

Para comprobar que el algoritmo de cálculo de las direcciones se efectúa de manera correcta, la simulación funcional fue ejecutada con los vectores de entrada de la Tabla 4.5 generando la simulación de la Fig. 4.7.

Tabla 4.5 Vectores de entrada para la simulación funcional del bloque generador de direcciones de datos para la segunda etapa.

Vector	clr	EN_GD	g_wn	m_d
1	0	0	2	2
2	1	0	2	2
3	0	0	2	2
4	0	1	2	2
5	0	1	2	2
6	0	1	2	2
7	0	1	2	2

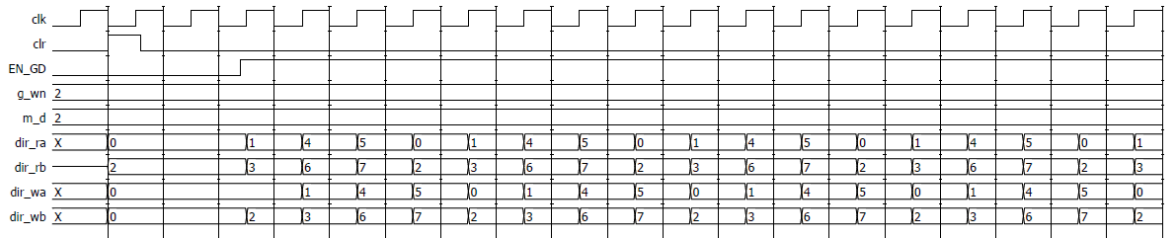


Fig. 4.7 Simulación funcional para el bloque generador de direcciones de datos para la segunda etapa.

En la Fig. 4.7 se observa que el intervalo entre las direcciones generadas por el componente incrementó en magnitud, ahora hay 2 unidades entre *dir_ra* y *dir_rb*, esto es porque los valores de las entradas *g_wn* y *m_d* han sido modificados y ambos tienen el valor de 2, esta simulación corresponde a las direcciones generadas para la segunda etapa del algoritmo FFT para ocho puntos. El circuito generado después de la síntesis de la entidad de la Fig. 3.8 se muestra en la Fig. 4.8.

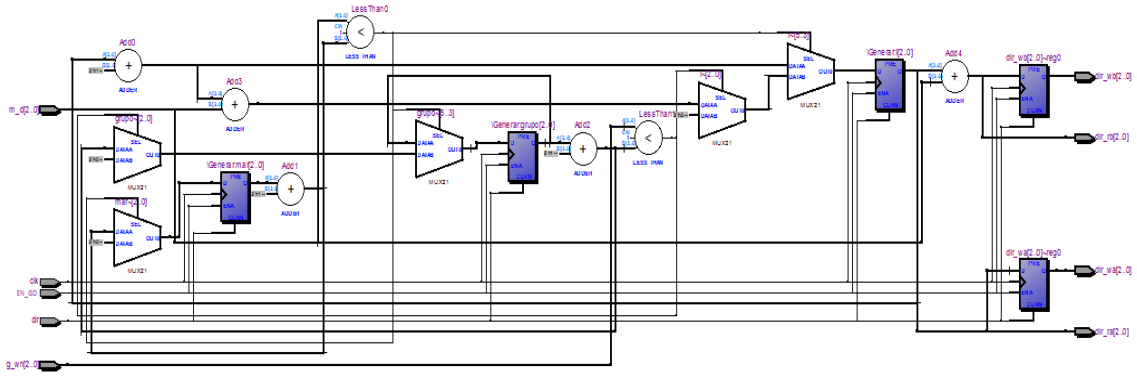


Fig. 4.8 Diagrama *RTL* de Generador de direcciones.

Los resultados de la síntesis del bloque generador de direcciones de datos se observan en la Fig. 4.9, estos corresponden a la síntesis del componente para una arquitectura de 512 puntos.

Flow Summary	
Flow Status	Successful - Thu Jul 04 20:50:11 2013
Quartus II Version	11.0 Build 157 04/27/2011 SJ Web Edition
Revision Name	Gen_direccionesEx
Top-level Entity Name	Gen_direccionesEX
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	100 / 33,216 (< 1 %)
Total combinational functions	91 / 33,216 (< 1 %)
Dedicated logic registers	45 / 33,216 (< 1 %)
Total registers	45
Total pins	57 / 475 (12 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Fig. 4.9 Resultados de síntesis del bloque generador de direcciones de datos

4.4 Pruebas del bloque generador de coeficientes complejos

El bloque está encargado de liberar uno a uno los coeficientes complejos con los que se hace la evaluación de la mariposa dependiendo de la etapa y mariposa del grupo, como se mencionó, está formado por un contador que direcciona a las memorias ROM en donde están almacenados los valores de los coeficientes complejos. La resolución del contador así como el tamaño de las memorias depende de la etapa a implementar y aumenta con cada etapa llegando hasta *N/2*.

Para esta simulación se tomó el componente de la tercera etapa de la arquitectura, el cual contiene cuatro coeficientes con sus respectivos componentes reales e imaginarios, se ingresaron como entradas los vectores que se muestran en la Tabla 4.6 obteniendo como salida la simulación de la Fig. 4.10.

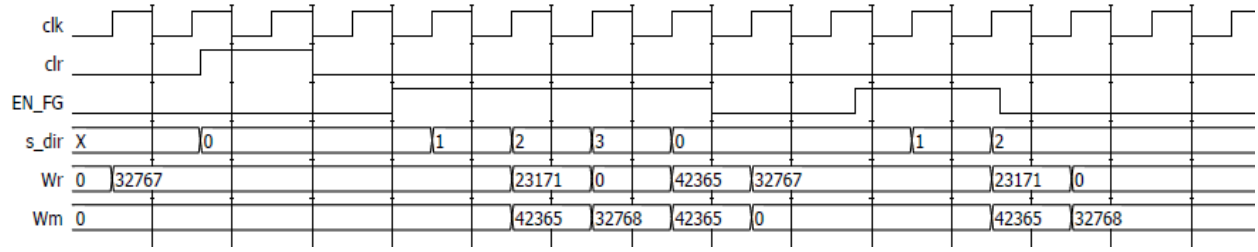


Fig. 4.10 Simulación funcional del bloque generador de coeficientes complejos.

Tabla 4.6 Vectores de entrada para la simulación funcional del bloque generador de coeficientes complejos

Vector	clr	EN_FG
1	0	0
2	0	0
3	1	0
4	0	0
5	0	1
6	0	1
7	0	1
8	0	1
9	0	0
10	0	0
11	0	1
12	0	1
13	0	0

Como se planteó en el diseño, las memorias son de tipo ROM, esto quiere decir que deben ser inicializadas desde el momento de la síntesis por lo que no se puede hacer operación de escritura sobre éstas. El contenido de las memorias se puede observar en las Tabla 4.7 y Tabla 4.8.

Tabla 4.7 Contenido de la ROM para la parte real de los coeficientes.

Memoria ROM parte real de coeficientes	
Dirección	**Dato**
0	32767
1	23171
2	0
3	42365

Tabla 4.8 Contenido de la ROM para la parte imaginaria de los coeficientes.

Memoria ROM parte imaginaria de coeficientes	
Dirección	**Dato**
0	0
1	42365
2	32768
3	42365

En la simulación se observa que la resolución del contador es de dos bits lo que permite tener una cuenta de 0 a 3, esto es suficiente para direccionar las cuatro palabras de las memorias ROM. Al ingresar el primer vector de entrada se puede observar que *s_dir* tiene un valor indefinido, esto es porque no se ha presentado un evento en el de *reset* del contador, lo que genera que *Wr*=32767 y *Wm*=0 que es lo que

contienen las primeras localidades de las memorias, los vectores de salida así como la señal de dirección de las memorias permanecen con estos valores una vez que *clr* regresa al valor de 0 lógico puesto que *EN_FG* no ha sido activada sino hasta el vector 5 que es cuando la *s_dir*=1 sin embargo los vectores *Wr* y *Wm* se mantienen con los valores de lectura anteriores y se ven alterados hasta el siguiente ciclo de reloj cuando el vector de entrada es el 6, esto se debe a que aunque la dirección de lectura de las memorias ya fue generada ésta no ha sido registrada por las entradas de las memorias lo que provoca que parezca que los datos salen un ciclo después pero realmente se debe a que las direcciones son tomadas en tiempo de *preset* es decir, cuando están haciendo el cambio al siguiente valor, por lo que cuando el vector de entrada es el 6 y *s_dir*=2 a las salidas se tiene que *Wr*=23171 y *Wm*=42365 que es el contenido de la localidad 1 de las memorias.

En la simulación se puede observar que el flujo de lectura de las memorias continúa sin ninguna alteración hasta que se presenta el vector 9 en donde se deshabilita *EN_FG* lo que provoca que *s_dir*=0 y se mantenga ahí durante dos ciclos de reloj puesto que la habilitación del contador ha sido interrumpida, teniendo que *Wr*=32767 y *Wm*=0.

Una vez que el vector de entrada es el 11, la cuenta vuelve a ser habilitada y la lectura de las memorias nuevamente cambia teniendo a las salidas *Wr* y *Wm* el contenido de la localidad 1. Se puede notar que la simulación se detiene cuando se presenta el vector 13 demostrando que la interrupción de las tareas puede hacerse en cualquier momento.

El circuito resultante de la síntesis del código se observa en el diagrama *RTL* de la Fig. 4.11 que corresponde a la entidad de la Fig. 3.10, en donde se pueden ver 3 salidas, *s_dir* solamente fue generada como una señal de prueba y monitoreo sin embargo en la implementación final esta ya no es una salida, sino una señal interna.

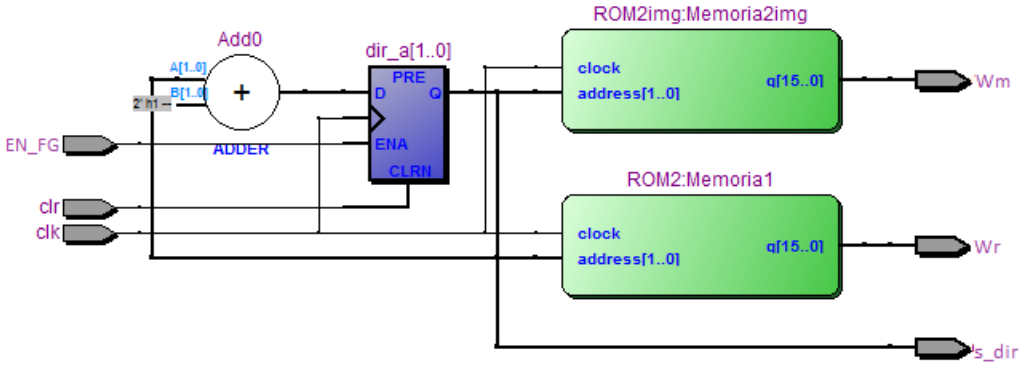


Fig. 4.11 Diagrama *RTL* del bloque generador de coeficientes complejos.

Los resultados de la síntesis para el bloque de coeficientes complejos de la tercera etapa se muestran en la Fig. 4.12, hay que recordar que la cantidad de memoria requerida entre una etapa y otra varia por las razones ya explicadas.

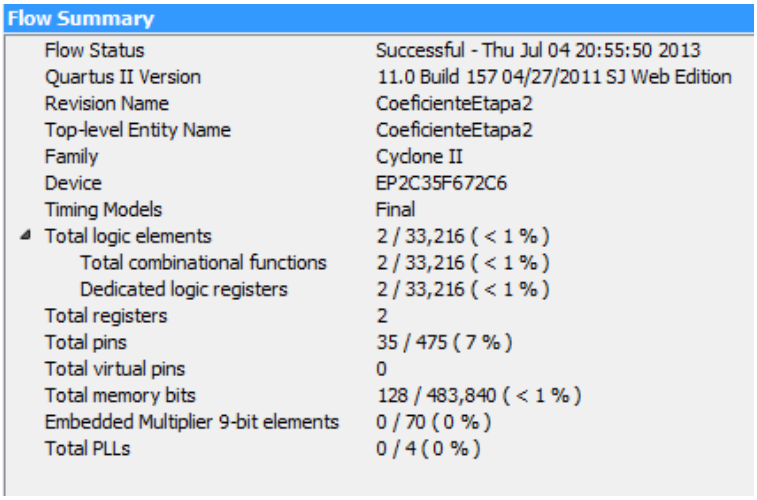


Fig. 4.12 Resultados de la síntesis para el bloque de coeficientes complejos de la tercera etapa de la arquitectura

4.5 Pruebas del bloque para la evaluación de la mariposa

Este bloque se encarga de evaluar la operación base del algoritmo de la FFT, dado que fue diseñado bajo el concepto de ser un elemento completamente combinatorio, la salida se ve afectada casi al mismo tiempo en el que los operandos de la entrada son modificados. Para esta primera prueba, los vectores de entrada utilizados se muestran en la Tabla 4.9 generando la simulación de la Fig. 4.13 en donde se muestra la simulación el funcionamiento del bloque para la evaluación de la mariposa de la primera etapa.

Tabla 4.9 Vectores de prueba para la simulación del bloque de evaluación de la mariposa de la primera etapa.

Vector	**Ar**	**Br**
1	0	1
2	2	4
3	4	7
4	6	10
5	8	13
6	10	16

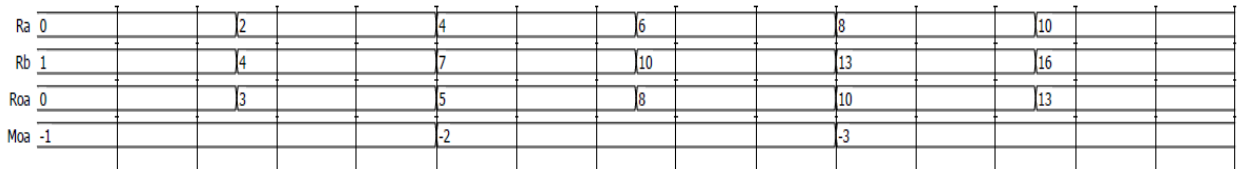


Fig. 4.13 Simulación funcional del bloque para la evaluación de la mariposa de la primera etapa.

En la Fig. 4.13 se observa que una vez que se ha evaluado la operación de acuerdo con lo descrito en el capítulo 3, se realiza una suma o resta dependiendo del caso y después se hace un escalamiento, esto para evitar un desbordamiento. Como se observa, al hacer las operaciones en formato de punto fijo el redondeo se realiza hacia el número más positivo. La Tabla 4.10 muestra el flujo de las operaciones en este bloque para los vectores de entrada de la Tabla 4.9.

Tabla 4.10 Flujo de operaciones para la evaluación de la mariposa de la primera etapa.

Operando A	Operando B	A+B	A-B	Aro = (A+B)/2	Bro = (A+B)/2
0	1	0	-1	0	-1
2	4	6	-2	3	-1
4	7	11	-3	5	-2
6	10	16	-4	8	-2
8	13	21	-5	10	-3
10	16	26	-6	13	-3

El circuito resultante de la descripción se muestra en la Fig. 4.11.

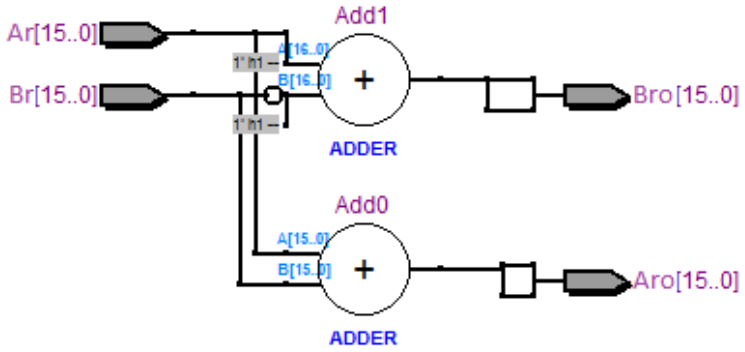


Fig. 4.14 Diagrama *RTL* del bloque para la evaluación de la mariposa de la primera etapa.

El resultado de la síntesis de este bloque se muestra en la Fig. 4.15

Flow Summary	
Flow Status	Successful - Thu Jul 04 20:59:08 2013
Quartus II Version	11.0 Build 157 04/27/2011 SJ Web Edition
Revision Name	Mariposa_op_etapa1
Top-level Entity Name	Mariposa_op_etapa1
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	32 / 33,216 (< 1 %)
Total combinational functions	32 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	64 / 475 (13 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Fig. 4.15 Resultados de la síntesis de la mariposa para la evaluación de la primera etapa

Para la segunda etapa, el diseño del bloque se diferencia del primero en que se reciben como operandos a los coeficientes complejos. Este componente fue diseñado igualmente para funcionar como un elemento completamente combinatorio, en la Tabla 4.11 se muestran los vectores de entrada bajo los cuales se ejecutó la simulación de la Fig. 4.16.

Tabla 4.11 Vectores de entrada para la simulación de la mariposa de la segunda etapa.

Vector	Ra	Rb	Rw	Mw
1	210	26	-1	0
2	212	27	0	-1
3	214	28	-1	0
4	216	29	0	-1

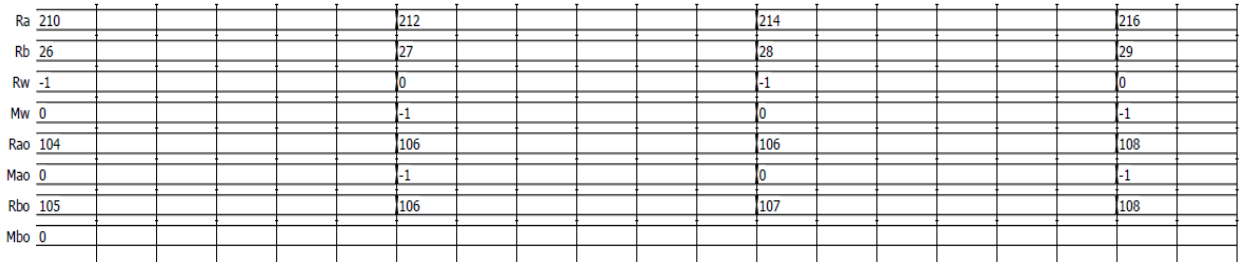


Fig. 4.16 Simulación funcional del bloque para la evaluación de la mariposa de la segunda etapa.

Para corroborar el correcto funcionamiento del bloque, se toma como ejemplo los datos

Ra = 214, Rb = 28, Rw = -1, Mw = 0 y se desarrollan las operaciones a nivel de bits como sigue

Ra = 0000000011010110,

Rb = 0000000000011100

Rw =1111111111111111

Mw =0000000000000000

$$bWn = (Rb + i0)(Rw + iMw) = (RbRw) + (iRbMw) + (i0Rw) - (0Mw)$$

$$(Rb + i0)(Rw + iMw) = (RbRw) + i(RbMw)$$

$$(RbRw) = (0000000000011100 * 1111111111111111)$$

$$= (11111111111111111111111111100100)Q_{15} = 1111111111111110Q_0$$

$$(RbMw) = (0000000000011100 * 0000000000000000)$$

$$= 00000000000000000000000000000000Q_{15} = 0000000000000000Q_0$$

$$A = a + x = (Ra + Rx) + i(Mx)$$

$$Rao = (Ra + Rx) = (0000000011010110 + 1111111111111111) = \left(\frac{0000000011010101}{2}\right)$$

$$= 0000000001101010 = 106$$

$$Mao = (Mx) = (0000000000000000) = 0$$

$$B = a - x = (Ra - Rx) + i(Mx)$$

$$Rbo = (Ra + Rx) = (0000000011010110 - 1111111111111111) = \frac{0000000011010111}{2}$$

$$= 0000000001101011 = 107$$

$$Mbo = -(Mx) = -(0000000000000000) = 0$$

Una vez que se desarrollaron cada una de las operaciones requeridas para la evaluación de la mariposa, se puede observar que los valores corresponden con los que se muestran en la simulación de la Fig. 4.12 lo que significa que el bloque funciona adecuadamente.

El circuito resultante de la descripción se muestra en la Fig. 4.17.

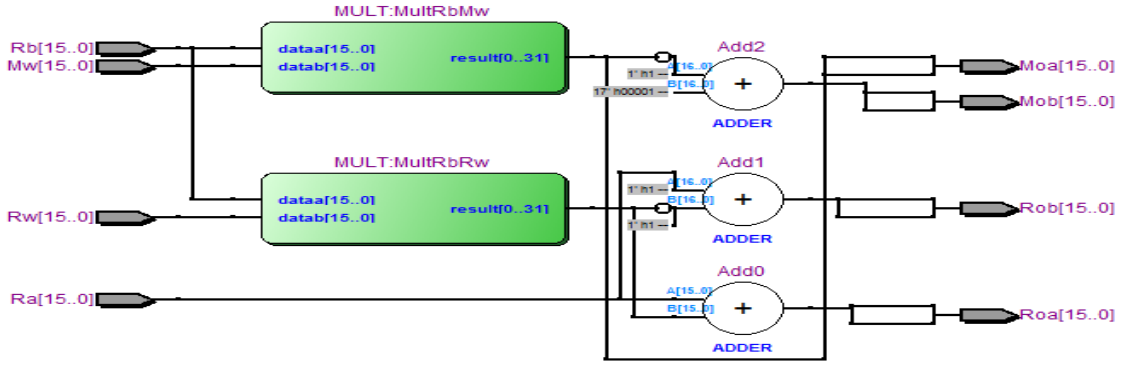


Fig. 4.17 Diagrama *RTL* del bloque para la evaluación de la mariposa de la segunda etapa.

Los resultados de la síntesis de este bloque se muestran en la Fig. 4.18 en donde se puede observar que para esta mariposa, que es la que se encarga de evaluar las operaciones de la segunda etapa de la arquitectura, se están utilizando 4 multiplicadores dedicados tal cual fue previsto en la etapa de diseño.

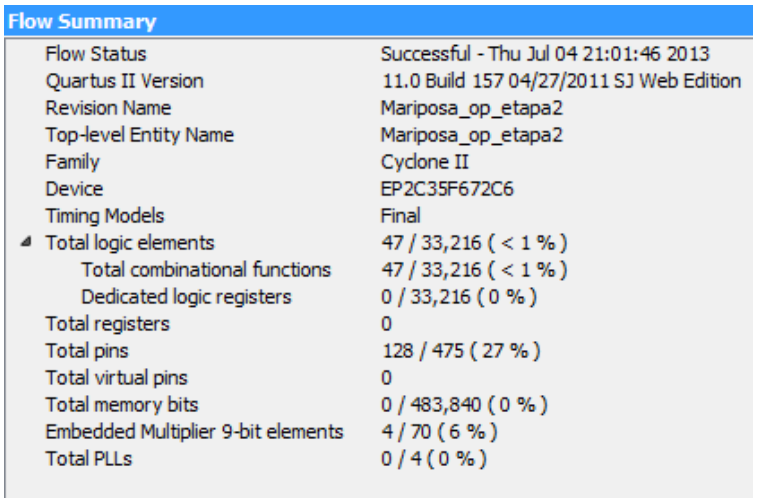


Fig. 4.18 Resultados de la síntesis del bloque para la evaluación de la mariposa de la segunda etapa

A partir de la tercera etapa, los operandos para la evaluación de la mariposa se encuentran completos, es decir, se tienen números complejos que provienen de las memorias de datos así como los coeficientes complejos. Como se describió en el diseño, este es el bloque genérico de evaluación de la mariposa, para la simulación se utilizaron los vectores de entrada de la Tabla. 4.12 con los que se obtuvo la simulación de la Fig. 4.19.

Tabla 4.12 Vectores de entrada para la simulación de la mariposa de las iteraciones 3 en adelante.

Vector	Ra	Rb	Ma	Mb	Rw	Mw
1	1	2	202	39	0	-1
2	255	1023	202	39	257	-1
3	258	1026	202	39	260	0
4	265	1033	202	39	267	0
5	270	1038	202	39	272	0

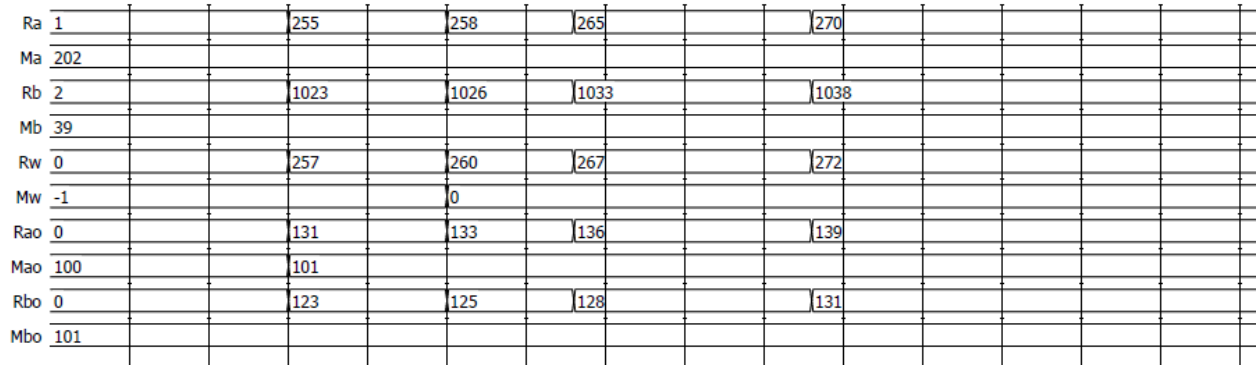


Fig. 4.19 Simulación funcional del bloque para la evaluación de la mariposa a partir de la tercera etapa.

Con el fin de corroborar el funcionamiento, así como se hizo con la mariposa de la segunda etapa se desarrollan las operaciones de acuerdo con las definidas por el diagrama de la Fig. 3.14 y se evalúan y comparan los resultados.

Con Ra = 259, Ma = 202, Rb = 1027, Mb = 39, Rw = 261, Mw = 0

Ra = 0000000100000011, Ma = 0000000011001001

Rb = 0000010000000011, Mb = 0000000000100111

Rw = 0000000100000101, Mw = 0000000000000000

Con base en las ecuaciones del capítulo 3

$$bWn = (RbRw - MbMw) + i(RbMw + MbRw)$$

$$RbRw = (0000010000000011 * 0000000100000101) =$$

$$00000000000001000001011100001111Q_{15}$$

$$MbMw = (0000000000100111 * 0000000000000000)$$

$$= 00000000000000000000000000000000Q_{15}$$

$$RbMw = (0000010000000011 * 0000000000000000)$$

$$= 00000000000000000000000000000000Q_{15}$$

$$MbRw = (0000000000100111 * 0000000100000101)$$

$$= 00000000000000000010011111000011Q_{15}$$

$$x = (RbRw - MbMw) + i(RbMw + MbRw) = Xr + iXm$$

$$Xr = (00000000000001000001011100001111 - 00000000000000000000000000000000)$$

$$= 00000000000001000001011100001111Q_{15=}0000000000001000Q_0$$

$$Xm = (00000000000000000000000000000000 + 00000000000000000010011111000011)$$

$$= 00000000000000000010011111000011Q_{15} = 0000000000000000Q_0$$

$$A = a + x = (Ra + Xr) + i(Ma + Xm)$$

$$Rao = (Ra + Xr) = (0000000100000011 + 0000000000001000) = \frac{0000000100001011}{2}$$

$$= 0000000010000101 = 133$$

$$Mao = (Ma + Xm) = (0000000011001001 + 0000000000000000) = \frac{0000000011001001}{2}$$

$$= 0000000001100101 = 101$$

$$B = a - x = (Ra - Xr) + i(Ma - Xm)$$

$$Rbo = (Ra - Xr) = (0000000100000011 - 0000000000001000) = \frac{0000000011111011}{2}$$

$$= 0000000001111101 = 125$$

$$Mbo = (Ma - Xm) = (0000000011001001 - 0000000000000000) = \frac{0000000011001001}{2}$$

$$= 0000000001100101 = 101$$

Mediante el análisis realizado anteriormente para cada una de los bloques de evaluación de la mariposa diferente se puede comprobar el correcto funcionamiento de los elementos, en la Fig. 4.20 se muestra el diagrama RTL del circuito resultante de la síntesis de la entidad en la Fig. 3.13.

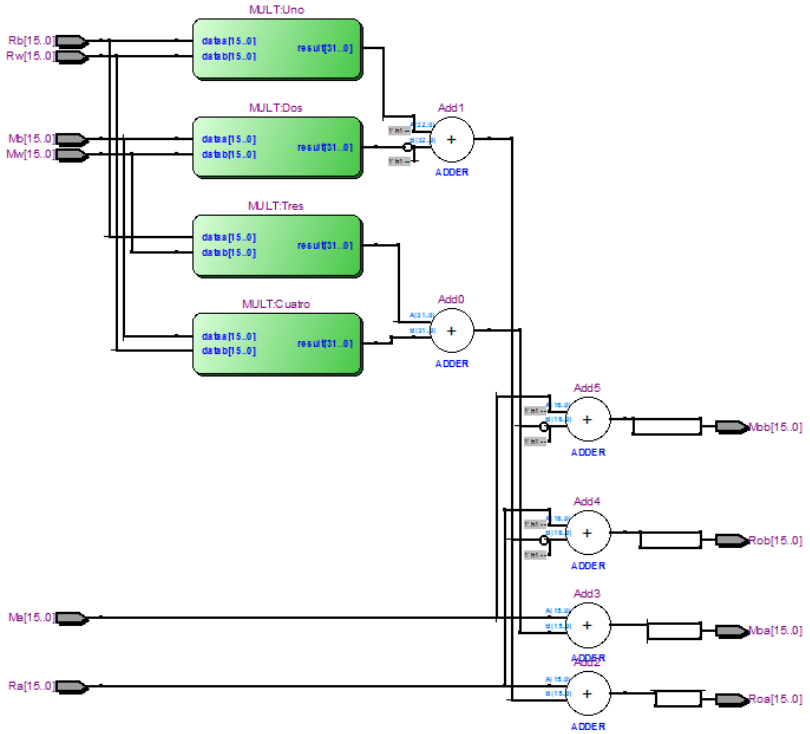
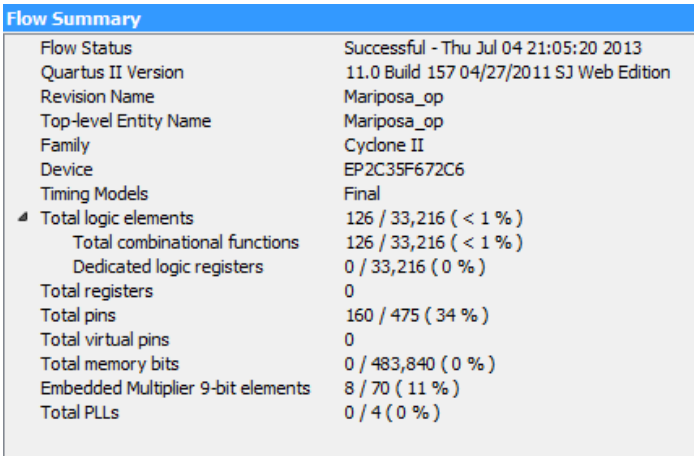


Fig. 4.20 Diagrama RTL del bloque para la evaluación de la mariposa a partir de la tercera etapa.

Los resultados de la síntesis de este bloque se muestran en la Fig. 4.21, en donde se puede observar que para esta mariposa que evalúa la operación con los datos completos se utilizan un total de 8 multiplicadores dedicados.



Flow Summary	
Flow Status	Successful - Thu Jul 04 21:05:20 2013
Quartus II Version	11.0 Build 157 04/27/2011 SJ Web Edition
Revision Name	Mariposa_op
Top-level Entity Name	Mariposa_op
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	126 / 33,216 (< 1 %)
Total combinational functions	126 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	160 / 475 (34 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	8 / 70 (11 %)
Total PLLs	0 / 4 (0 %)

Fig. 4.21 Resultados de síntesis del bloque de mariposa genérico

4.6 Pruebas del bloque para el cálculo del módulo de resultados

Como se mencionó en el capítulo de diseño, el bloque para el cálculo del módulo de los resultados es considerado en esta propuesta como una etapa más del algoritmo para fines prácticos, el cual se diferencia por los componentes de cálculo de direcciones y cálculo del módulo. En esta sección se mostrarán las pruebas a estos dos bloques mientras que la prueba de todos los componentes en conjunto será mostrada más adelante.

- **Prueba del módulo generador de direcciones para el cálculo del módulo**.

Como se describió en el diseño, este bloque es un contador ascendente *ModN* con una entrada de habilitación, una entrada de reloj y una entrada de *reset* generando como salidas dos vectores de $log_2 N$ bits de resolución que permiten direccionar las *N* localidades de las memorias en donde se almacenan los resultados de la última etapa. Este bloque fue implementado y probado con los vectores de la Tabla 4.13 obteniendo como resultado la simulación de la Fig. 4.22.

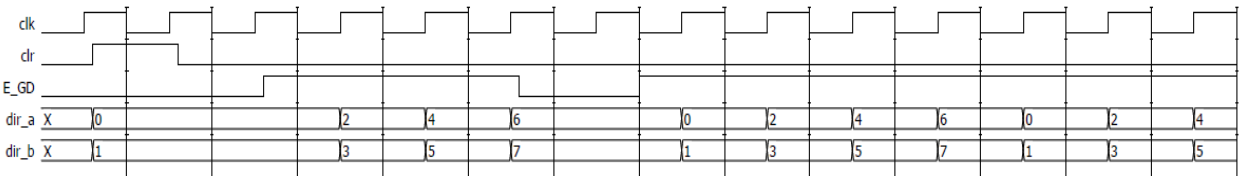


Fig. 4.22 Simulación funcional del módulo generador de direcciones para el cálculo del módulo.

Tabla 4.13 Vectores de prueba para la simulación del bloque de cálculo del módulo de resultados.

Vector	clr	E_GD
1	0	0
2	1	0
3	1	0
4	0	1
5	0	1
6	0	1
7	0	0
8	0	1

Nuevamente, este componente depende de un evento en *clr* para inicializar todas las salidas, como se muestra en los vectores de entrada 2 y 3, de no ser así, aunque la señal *E_GD* este activada, el componente no presentará variaciones en las salidas.

Una vez que se han inicializado las señales, el componente permanece en espera de la activación de la señal *E_GD* como se observa en el vector 4 en donde al siguiente flanco de ascenso del reloj la cuenta comienza a aumentar hasta llegar al valor tope máximo, este vector de entrada permanece hasta la presencia del vector 7 en donde *E_GD* es interrumpida y por tanto la cuenta es detenida hasta la activación de la señal *E_GD* como en el vector 8 en donde se retoma la cuenta en donde se quedó.

Finalmente, el circuito sintetizado se muestra en la Fig. 4.23 en donde se observa que sólo se generan dos vectores de salida, estos son las direcciones de lectura de los bancos de memoria y dado que es la última etapa de la arquitectura lo único que se requiere es el resultado aritmético del cálculo del módulo por lo que ya no se requiere de registros para almacenar las direcciones al no haber necesidad de escribir los resultados en otra memoria.

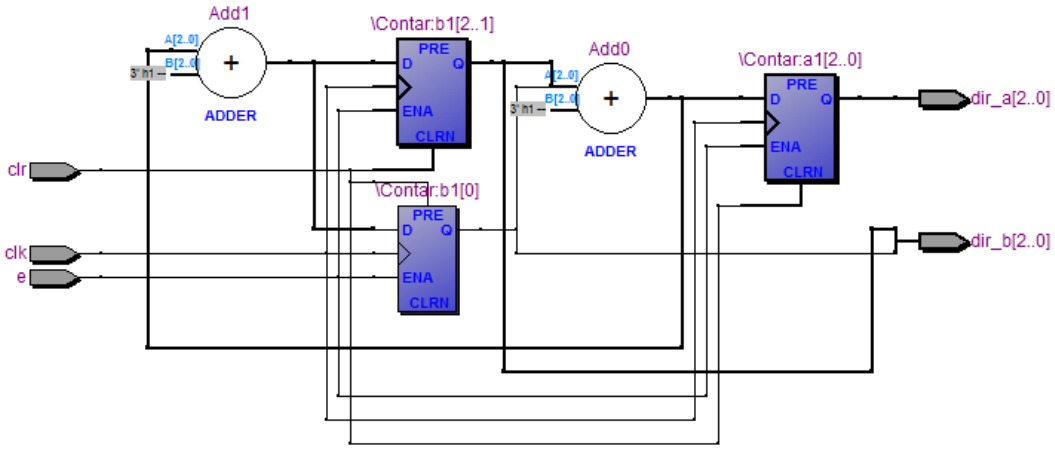


Fig. 4.23 Diagrama RTL del generador de direcciones para el bloque de cálculo del módulo de resultados.

- **Pruebas del bloque de cálculo del módulo**

Como se describió en la parte del diseño, este bloque se encarga de hacer el cálculo del módulo de los resultados del algoritmo, es quien sustituye al bloque de evaluación de la mariposa lo que significa que tiene características similares como ser un bloque completamente combinatorio y estar implementado con elementos aritméticos dedicados como son los multiplicadores. En la Tabla 4.14 se tienen los vectores utilizados como entradas a la simulación que se muestra en la Fig. 4.24.

Tabla 4.14 Vectores de resultados para el cálculo del módulo de resultados.

Vector	dato_r	dato_m
1	4404	1238
2	4886	4886
3	4926	5910
4	4864	4976

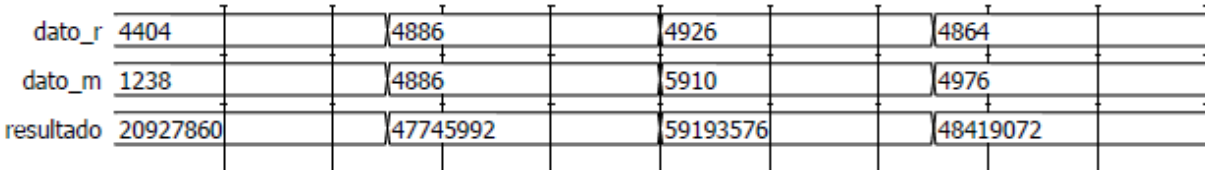


Fig. 4.24 Simulación funcional del cálculo del módulo de resultados.

Al ser un componente con base en lógica combinatoria, la evaluación de las operaciones se están realizando en todo momento, para corroborar el correcto funcionamiento queda tomar cualquiera de los vectores de entrada y evaluar manualmente las operaciones descritas en el diagrama de la Fig. 3.31, por ejemplo si se toman los datos de entrada del vector 3 en donde *dato_r*=4926 y *dato_m*=5910 se tiene que

$$dato\_r,^2 = 4926^2 \;\; y \;\; dato\_m^2 = 5910^2$$

$$dato\_r^2 = 24265476 \;\; y \; dato\_m^2 = 34928100$$

$$dato = 24265476 \; + 34928100 = 59193576$$

Del desarrollo se obtiene que *resultado=903* lo cual comprueba que para el vector de entrada 3 las operaciones se evalúan correctamente, así se puede tomar cualquier vector de entrada y verificar que el componente realiza correctamente la evaluación de las operaciones. El circuito resultante de la síntesis se muestra en la Fig. 4.25.

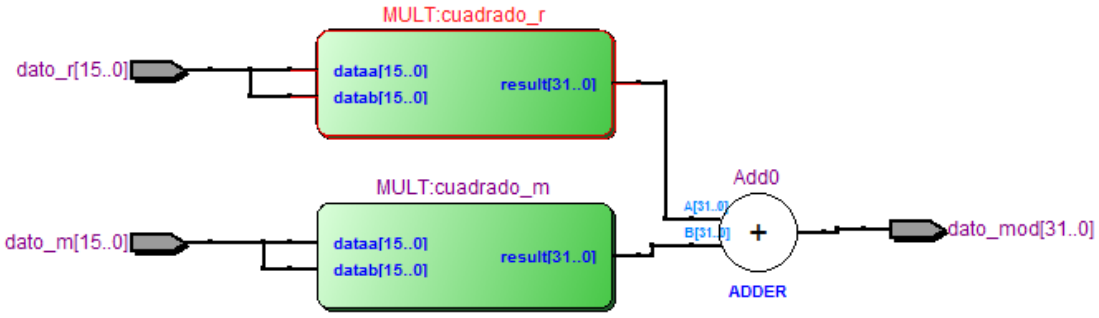


Fig. 4.25 Diagrama RTL del componente para el cálculo del módulo de resultados.

El resumen de síntesis de este bloque se muestra en la Fig. 4.26

Flow Summary	
Flow Status	Successful - Mon Jul 08 14:23:31 2013
Quartus II Version	11.0 Build 157 04/27/2011 SJ Web Edition
Revision Name	Modulo
Top-level Entity Name	Modulo
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	32 / 33,216 (< 1 %)
Total combinational functions	32 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	64 / 475 (13 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	4 / 70 (6 %)
Total PLLs	0 / 4 (0 %)

Fig. 4.26 Resumen de síntesis del bloque de cálculo del módulo de resultados

4.7 Pruebas de la unidad de control por etapa

4.7.1 Pruebas de control para la señal *we*

La señal *we* controla la operación que se realiza en cada bloque de memoria del banco correspondiente, existe una señal *we* por cada etapa de la arquitectura, como se observa en la Fig. 3.26 se observa que el comportamiento de esta señal está dado por la integración de tres componentes. Desglosando el funcionamiento, el bloque *Contador M*odN/2 se encarga de generar la bandera indicadora de que el momento de hacer un cambio en la operación se ha alcanzado, para comprobar su funcionamiento se ejecutó la simulación con los vectores de entrada de la Tabla 4.15 para obtener la Fig. 4.27.

Tabla 4.15 Vectores de entrada para la simulación del Contador ModN/2 de la señal *we.*

Vector	clr	e_c
1	0	0
2	1	0
3	0	1
4	0	1
5	0	1
6	0	0
7	0	1
8	0	1
9	0	1
10	0	1
11	0	1

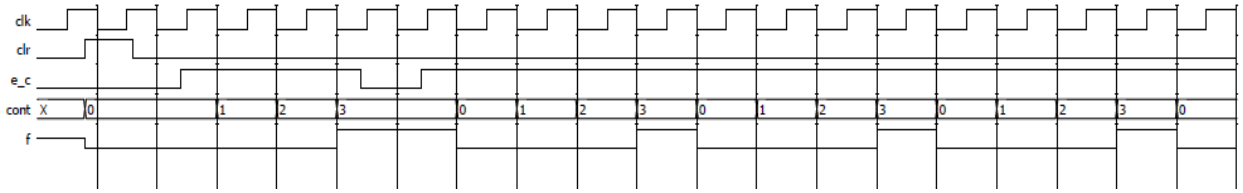


Fig. 4.27 Simulación funcional del bloque contador para el control de la señal *we.*

Al igual que los bloques anteriores que dependen de un contador, el funcionamiento de este no comienza si no hasta que se presenta un evento positivo en clr como se observa en la Fig. 4.27 cuando el vector de entrada corresponde al 2, al ser un circuito asíncrono con respecto a clr no importa el momento en el que se presente el evento ya que cont (señal interna mostrada para monitoreo) se inicializará con 0 lógico y por tanto en la bandera f habrá un 0 lógico.

El conteo ascendente comienza cuando *e_c* toma el valor de 1 lógico, como lo hace al recibir el vector 3 de entrada en cada flanco de descenso del reloj, una vez que se alcanza la cuenta máxima del contador, la bandera *f* es activada y permanece así mientras la cuenta no se modifique, este es el caso del vector 5 en

donde la cuenta llega al máximo y el vector 6 en donde la señal *e_c* es interrumpida y la cuenta ya no se modifica, es por eso que *f* mantiene el valor de 1 lógico puesto que la cuenta se detuvo en el máximo.

Una vez que *e_c* se habilita de nuevo con el vector 7 la cuenta comienza a partir del punto en el que se quedó y se puede observar que, a partir de este momento en el que ya no hay interrupciones, la señal *f* toma el valor de uno siempre y cuando *cont*=3. Esta descripción sintetiza un circuito como el que se muestra en la Fig. 4.28.

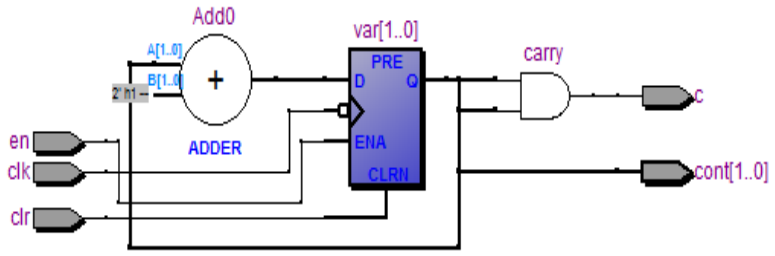


Fig. 4.28 Diagrama RTL del bloque contador para el control de la señal *we.*

En seguida, el bloque *ContadorWE* se encarga de hacer el cambio en el valor de la señal *we* es decir, de escritura a lectura y viceversa con respecto a la bandera *f* generada por *Contador ModN/2* y el estado en el que se encuentre con respecto a *Maquina we*, a continuación, en la Fig. 4.29 se muestra la simulación funcional del módulo *ContadorWE* obtenida con los vectores de entrada de la Tabla 4.28.

Tabla 4.16 Vectores de entrada para la simulación del *Contador we.*

Vector	clr	e_we
1	0	0
2	1	0
3	0	1
4	0	1
5	0	1
6	0	0
7	0	0
8	0	1
9	0	1

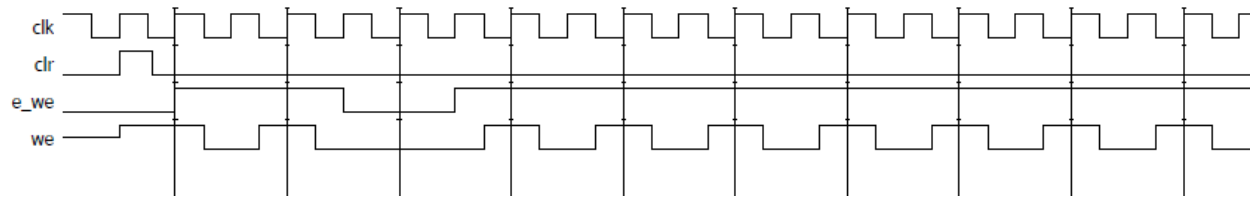


Fig. 4.29 Simulación funcional del bloque de generación de cambio de la señal *we.*

Nuevamente, el evento que dispara el funcionamiento del componente es la activación de *clr* como lo hace el vector de entrada 2 poniendo a *we* en 1 lógico que es el valor de inicialización, a partir de este momento se espera la habilitación de *e_we* como se hace con el vector 3 de entrada en donde, por cada flanco de descenso de *clk* el valor de *we* se invierte, si *e_we* es interrumpida como se hace en los vectores 6 y 7 *we* mantiene el último valor con el que se quedó y comienza con su funcionamiento cuando *e_we* nuevamente es habilitada. El circuito resultante de la síntesis se muestra en la Fig. 4.29.

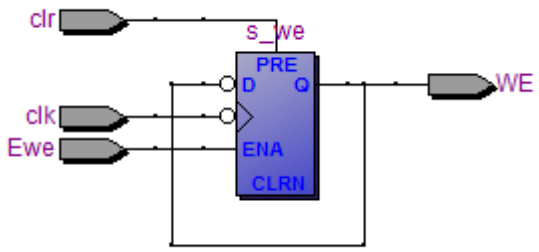


Fig. 4.30 Diagrama RTL del bloque de generación de cambio de la señal *we*.

Una vez que se han simulado los componentes por separado se hizo la integración en un solo circuito con respecto a la entidad de la Fig. 3.26 para corroborar el funcionamiento integrando la máquina de estados para la lógica de control de la señal *we*, esto se puede observar en la simulación de la Fig. 4.31 que se ejecutó con los vectores de entrada de la Tabla 4.17.

Tabla 4.17 Vectores de entrada para la simulación del control de la señal *we*.

Vector	clr	Em
1	0	0
2	1	0
3	0	0
4	0	1
5	0	1
6	0	1
7	0	1
8	0	1
9	0	0
10	0	0
11	0	1
12	0	1

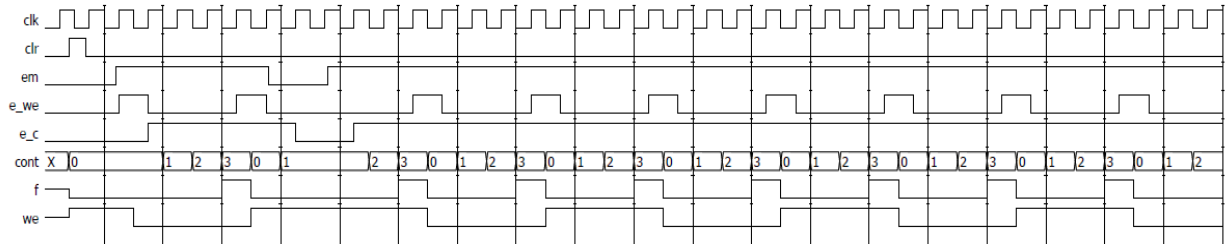


Fig. 4.31 Simulación funcional del bloque de control para la señal *we.*

En la simulación de la Fig. 4.31 se aprecia que todas las señales que interfieren en el funcionamiento de *we*, las señales generadas por la máquina de estados descrita en la Fig. 3.27 son *e_we* que activa el *Contador we*, la señal *e_c* que activa al *Contador ModN/2*. Al ingresar como entrada al vector 1 la máquina de estados entra en el estado A en donde las señales son inicializadas tal que *we* comience señalando la tarea de escritura, el circuito se mantiene en este estado hasta el momento en el que *em* es habilitado como en el vector 4 de entrada y se presenta un evento en *clk* de flanco de descenso, al ser así se hace la transición al estado B en donde *e_we*=1 lo que genera que *we* invierta su valor.

Al continuar habilitada *em*, como es el caso del vector 5, se hace una transición al estado C en donde *e_we*=0 y *e_c*=1 con lo que se activa la cuenta para indicar en qué momento se tiene que hacer un cambio en el valor de *we*, por lo que durante el vector 6 la máquina permanece en el estado C puesto que *em*=1 y *f*=0, cuando ocurre un cambio en *f* que es durante el vector de entrada 7 se hace una transición al estado E en donde se habilitan *e_we* y *e_c* generando un cambio en el valor de *we*.

Cuando se recibe como entrada el vector 8, nuevamente se hace una transición al estado C para desactivar a *e_we*, para el vector 9 de entrada en donde *em* es desactivado se hace una transición al estado D en donde se interrumpe el funcionamiento de ambos contadores permitiendo así mantener a la señal *we* en el estado anterior y a *cont* detenido en la cuenta, en este caso con un valor de 1, al presentarse como entrada el vector 10 estando en el estado D se hace una transición al estado C.

Esta transición permite que nuevamente sea activada *e_c* para así continuar con la cuenta y terminar con el ciclo de la operación en *we*. Como se observa en la simulación, cuando *em* se mantiene activada de manera constante se genera una periodicidad en la señal *we* de salida. El circuito resultante de la síntesis de este módulo se muestra en la Fig. 4.32 en donde se puede observar cómo se da la interacción entre cada uno de los componentes y que corresponde al diagrama de bloques de la Fig. 3.26.

Fig. 4.32 Diagrama RTL del bloque para el control de la señal *we*.

4.7.2 Pruebas de control para la señal *sm*

La señal *sm* controla la operación de selección de datos que serán utilizados en la evaluación de la mariposa dependiendo del banco de memorias que esté siendo leído, al igual que para la señal *we*, existe una señal *sm* por cada etapa de la arquitectura.

Como se observa en la Fig. 3.29 el comportamiento de esta señal está dado por la integración de tres componentes. Desglosando el funcionamiento del bloque de control de para *sm*, el bloque *Contador SM_ M*odN/2 se encarga de generar la bandera indicadora de que el momento de hacer un cambio en la operación se ha alcanzado, para comprobar su funcionamiento se ejecutó la simulación con los vectores de entrada de la Tabla 4.18 para obtener simulación de la Fig. 4.33.

Tabla 4.18 Vectores de entrada para la simulación del contador de periodicidad de la señal *sm*.

Vector	clr	e_csm
1	0	0
2	1	0
3	0	0
4	0	1
5	0	1
6	0	1
7	0	1
8	0	1
9	0	0
10	0	1

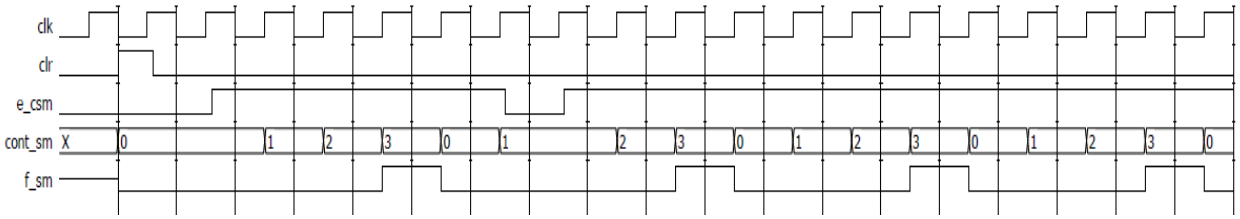


Fig. 4.33 Simulación funcional del bloque contador para el control de la señal *sm*.

Como fue planteado en el diseño, este bloque es inicializado por la señal *clr* cuando se presenta un evento positivo en ella tal como se hace con el vector en donde sin importar la señal de reloj, una vez que se presenta, la señal *cont_sm* toma el valor de cero y por tanto *f_sm* es igual a cero.

El conteo comienza cuando se presenta un evento de ascenso en la *clk* y *e_csm* está activada, esto se observa durante la entrada de los vectores del 4 al 8 en donde al momento en el que *cont_sm*=3 entonces *f_sm*=1 esto de acuerdo con el diseño del módulo, éste es el caso del vector de entrada 6. Cuando ocurre una interrupción en *e_csm* la cuenta se detiene y ésta se reanuda cuando es nuevamente es habilitada *e_csm*, esto sucede con el vector de entrada 9. A partir del vector 10 las entradas ya no son modificadas, lo que permite observar que se genera la activación de la bandera *f_sm* de manera periódica cuando el contador llega a su máximo. El circuito generado por la síntesis del componente se muestra en la Fig. 4.34.

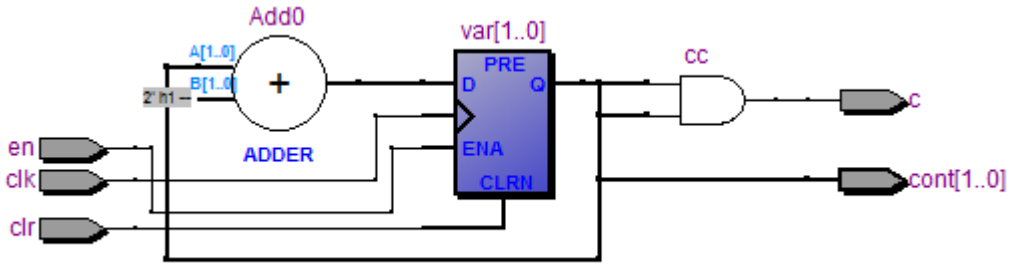


Fig. 4.34 Diagrama RTL del bloque contador para el control de la señal *sm*.

En seguida, el bloque *ContadorSM* se encarga de hacer el cambio en el valor de la señal *sm* es decir, cambiar de hacer la selección de los datos del *BancoA* a la selección de los datos del *BancoB* y viceversa con respecto a la bandera *f_sm* generada por *Contador SM_ModN/2* y el estado en el que se encuentre con respecto a *Máquina_sm*, a continuación, en la Fig. 4.35 se muestra la simulación funcional del módulo *ContadorSM* obtenida con los vectores de entrada de la Tabla 4.19.

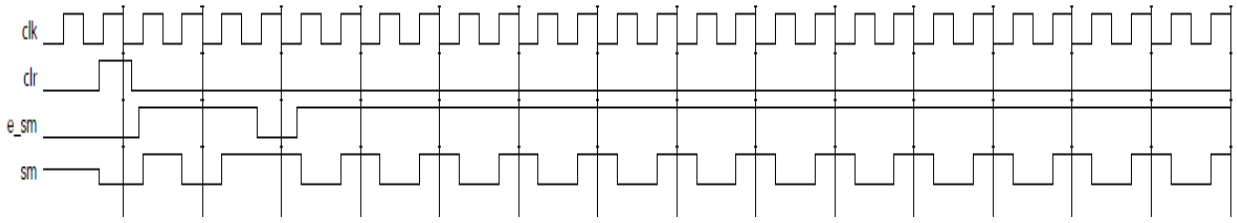


Fig. 4.35 Simulación funcional del contador *sm*.

Tabla 4.19 Vectores de entrada para la simulación del contador *sm.*

Vector	Clr	e_sm
1	0	0
2	1	0
3	0	1
4	0	1
5	0	1
6	0	0
7	0	1
8	0	1

Nuevamente, el evento que dispara el funcionamiento del componente es la activación de *clr* como lo hace el vector de entrada 2 poniendo a *sm* en 0 lógico que es el valor de inicialización, a partir de este momento se espera la habilitación de *e_sm* como se hace con el vector 3 de entrada en donde por cada flanco de ascenso de *clk* el valor de *sm* se invierte, si *e_sm* es interrumpida como se hace en el vector 7, *sm* mantiene el último valor con el que se quedó y comienza con su funcionamiento cuando *e_sm* nuevamente es habilitada. El circuito resultante de la síntesis se muestra en la Fig. 4.36.

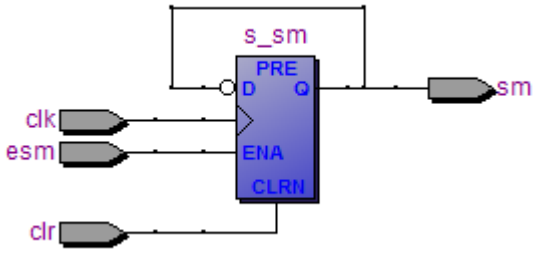


Fig. 4.36 Diagrama RTL del bloque de generación de cambio de la señal *sm.*

Una vez que se han simulado los componentes por separado se hizo la integración en un solo circuito con respecto a la entidad de la Fig. 3.29 para corroborar el funcionamiento integrando la máquina de estados para la lógica de control de la señal *sm*, esto se puede observar en la simulación de la Fig. 4.37 que se ejecutó con los vectores de entrada de la Tabla 4.20.

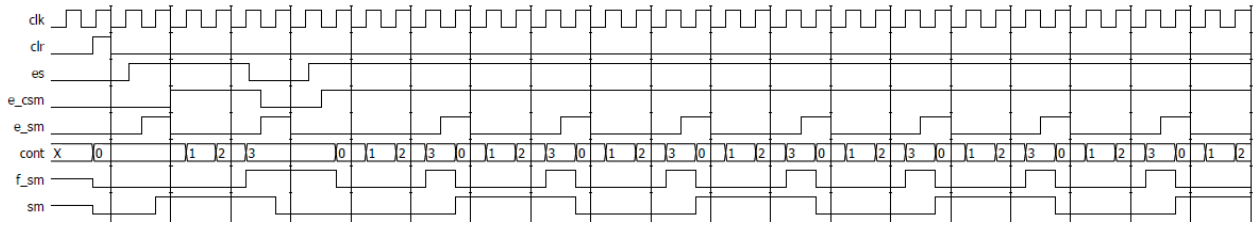


Fig. 4.37 Simulación funcional del bloque de control para la señal *sm.*

Tabla 4.20 Vectores de prueba para la simulación funcional del control de la señal *sm.*

Vector	clr	es
1	0	0
2	1	0
3	0	0
4	0	1
5	0	1
6	0	1
7	0	1
8	0	0
9	0	0
10	0	1

En la simulación de la Fig. 4.37 se pueden observar todas las señales que interfieren en el funcionamiento de *sm*, las señales generadas por la máquina de estados descrita en la Fig. 3.30 son *e_sm* que activa el *Contador sm*, la señal *e_csm* que activa al *Contador SM_ModN/2.* Al ingresar como entrada el vector 1 la máquina de estados entra en el estado A en donde las señales son inicializadas tal que *sm* comience señalando a los operandos del *BancoB* , el circuito se mantiene en este estado hasta el momento en el que *es* es habilitado como en el vector 3 de entrada y se presenta un evento en *clk* de flanco de ascenso, al ser así se hace la transición al estado B en donde *e_sm*=1.

La transición anterior genera que *sm* complemente su valor, al continuar con *es* activado como se hace con el vector 4, se hace una transición al estado C en donde *e_sm*=0 y *e_csm*=1 lo que significa que se activa el componente *Contador SM_ModN/2* el cual está encargado de generar la bandera que indica el momento en el que se requiere un cambio en la *sm*, durante los vectores 4, 5 y 6 la máquina permanece en el estado C hasta el vector 7 en donde el valor de *f_sm* ha cambiado y *es* esta activada lo que genera una transición al estado E en donde *e_sm*=1 y *e_csm*=1.

Con esto se produce un cambio en *sm* mientras continua la cuenta en *cont_sm* para permitir nuevamente generar una activación en *f_sm* cuando *cont_sm* llegue al valor máximo. De acuerdo con los vectores de entrada, al presentarse el vector 8 en donde se desactiva *en* se hace una transición al estado D para desactivar las funciones de los componentes haciendo que *sm* permanezca en el valor anterior y la cuenta se detenga, al recibir el vector 9 la máquina se mantiene en el estado D hasta recibir el vector 10 en donde se hace una transición al estado C para activar nuevamente la cuenta y mantenerse ahí hasta que *f_sm* sea activada y hacer una transición al estado E.

El vector de entrada 10 se mantiene constante a lo largo de la simulación de tal manera que se puede observar que los valores de la señal *sm* son asignados periódicamente, finalmente, el circuito resultado de la síntesis de la entidad mostrada en la Fig. 3.29 se muestra en la Fig. 4.38.

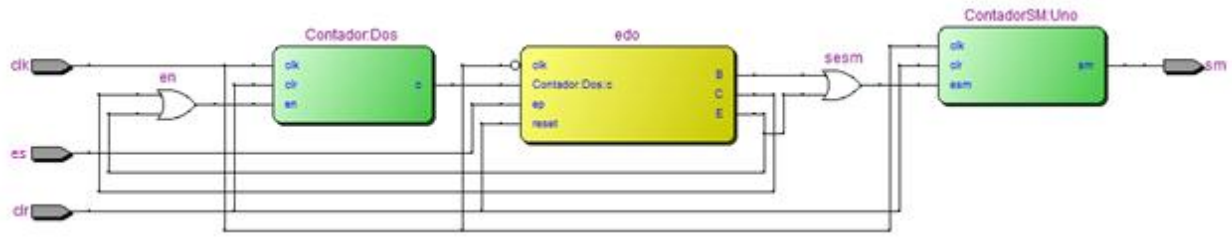


Fig. 4.38 Diagrama RTL del circuito para el control de la señal *sm.*

4.7.3 Pruebas de control para la señal *eg* y habilitación de *we* y *sm*

Como se mencionó en el capítulo 3, cada uno de los bloques de control de las señales que se requieren por etapa reciben una señal de habilitación diferente puesto que no se ponen en funcionamiento al mismo tiempo, esta habilitación depende del bloque simulado en la Fig. 4.39 con los vectores de entrada datos por la Tabla 4.21.

Tabla 4.21 Vectores de entrada para el control de la señal *eg* y habilitación de *sm* y *we.*

Vector	clr	E_C
1	0	0
2	1	0
3	0	0
4	0	0
5	0	1
6	0	1
7	0	0
8	0	1
9	0	0
10	0	0
11	0	0

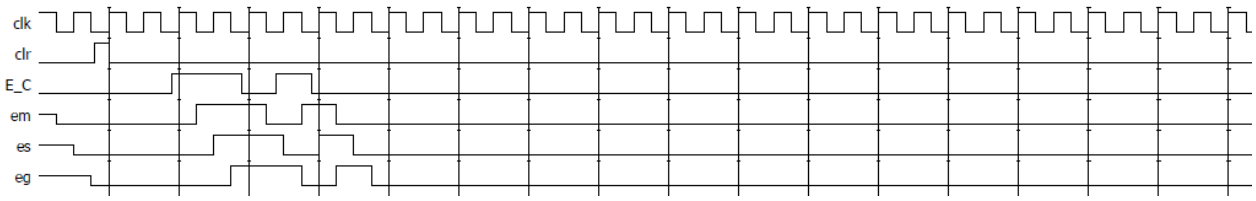


Fig. 4.39 Simulación funcional del control para la señal *eg* y habilitación de los bloques de control de *sm* y *we.*

Este bloque consiste en una cascada de *flip-flops* los cuales activan una a una las señales que servirán de control para los bloques de generación de *sm* y control de *we*, como se observa, se requiere de un evento en la señal *clr* para inicializar a todas las señales en cero, una vez que *E_C* es activada, estos valores se replican en las señales de salida de modo tal que *em* (que activa el control de la señal *we*) copia el valor de *E_C* al siguiente flanco negativo mientras que *en* (que activa el control de la señal *sm*) se activa al siguiente flanco de ascenso y *eg* un ciclo después que *em*. Cabe destacar que únicamente *em* depende directamente de *E_C*, las demás señales depende del valor de *em*. El circuito que resultante de la síntesis del diseño se muestra en la Fig. 4.40.

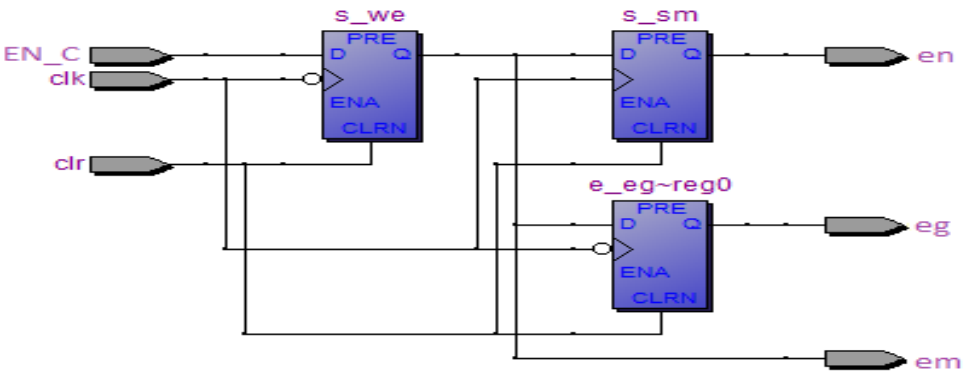


Fig. 4.40 Diagrama *RTL* del bloque de control para la señal *eg* y habilitación de *sm* y *we*.

4.7.4 Pruebas de integración de la unidad de control por etapa

Una vez que se verificó el correcto funcionamiento de los bloques que controlan a las señales *eg*, *we* y *sm* se hizo una integración de los tres bloques en un sólo componente de tal manera que se obtuviera la entidad mostrada en la Fig. 3.23, con el fin de verificar el funcionamiento de los tres componentes trabajando en conjunto se realizó la simulación del bloque con las entradas que se muestran en la Tabla 4.22 obteniendo la simulación que se muestra en la Fig. 4.41.

Tabla 4.22 Vectores de entrada para la simulación funcional de la unidad de control por etapa.

Vector	clr	E_C
1	1	0
2	0	0
3	0	0
4-16	0	1
17	0	0
18	0	0
19	0	0
20	0	1
21	0	1

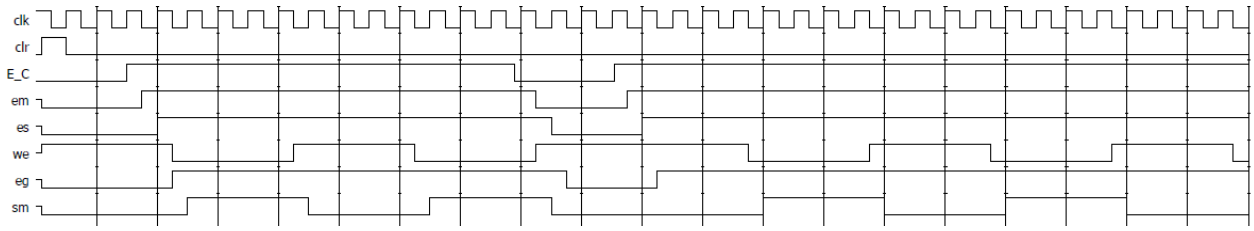


Fig. 4.41 Simulación funcional de la unidad de control por etapa.

Como se mencionó en las simulaciones de los componentes por separado, se requiere de un evento en *clr* que active la inicialización de las señales internas de las que depende el funcionamiento de las salidas, esto se hace con el vector de entrada 1 en donde se observa que *we*=1, *eg*=0 y *sm*=0.

Durante los vectores 2 y 3 el componente se mantiene en espera de la habilitación de *E_C* lo cual ocurre en con el vector 3 en donde se observa que la señal *em*=1 al siguiente flanco de descenso de *clk* y *es*=1 al siguiente flanco de ascenso lo que activa el funcionamiento de los componentes para el control de *we* y *sm*. Durante la entrada de los vectores 4 al 16 el vector de entrada permanece constante por lo que se puede observar que se generan en las salidas *we* y *sm* señales con un período de ocho ciclos de reloj (*we* con cambios en flanco de descenso y *sm* con cambios en flanco de ascenso).

Cuando se presenta una interrupción en *E_C* como se hace en los vectores del 17 al 18, las señales de habilitación internas (*em* y *es*) también se ven interrumpidas, lo que genera que para *we* y *sm* haya una extensión en la duración de período de la señal, es decir, pasan de *N/2* ciclos por nivel a *N/2+x* ciclos del nivel en donde se quedó, donde *x* es el número de flancos negativos durante los cuales *E_C* fue interrumpida, para este caso en particular se puede observar que la interrupción de *E_C* tuvo una duración de tres flancos de descenso, lo que extiende el a *we* hasta siete ciclos en nivel alto (puesto que la señal se encontraba en ese nivel cuando sucedió la interrupción) y a siete ciclos en nivel bajo a *sm* (puesto que la señal se encontraba en ese nivel cuando sucedió la interrupción).

Este funcionamiento permite hacer interrupciones desde la unidad de control general, ya que de ahí proviene *E_C* asegurando que el flujo de las operaciones continuará a partir del momento en el que se quedó. A partir del vector de entrada 20 que es en donde se reanudan las operaciones normalmente se puede observar que *we* y *sm* vuelven a generarse de manera normal. Finalmente, el circuito resultante de la síntesis correspondiente a la entidad mostrada en la Fig. 3.23 es el que se muestra en la Fig. 4.42.

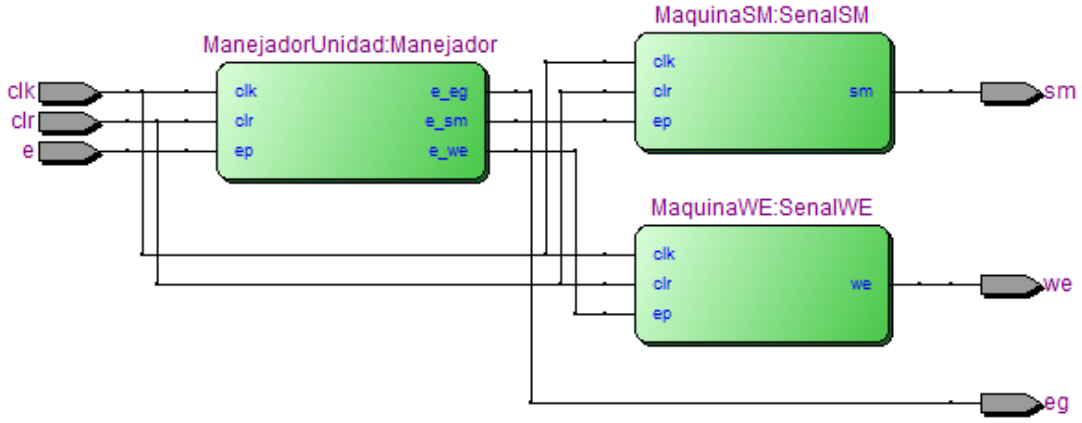


Fig. 4.42 Diagrama *RTL* de la unidad de control por etapa.

Los resultados de la síntesis de la unidad de control por etapa se muestran en la Fig. 4.43

Flow Summary	
Flow Status	Successful - Fri Jul 05 13:16:59 2013
Quartus II Version	11.0 Build 157 04/27/2011 SJ Web Edition
Revision Name	ControlLocal
Top-level Entity Name	ControlLocal
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	39 / 33,216 (< 1 %)
Total combinational functions	36 / 33,216 (< 1 %)
Dedicated logic registers	29 / 33,216 (< 1 %)
Total registers	29
Total pins	6 / 475 (1 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Fig. 4.43 Resultados de síntesis de la unidad de control por etapa.

4.8 Pruebas de la unidad de control general

Como se describió en el capítulo 3, este bloque se encarga de generar las señales de habilitación para cada uno de los bloques de control por etapa, este bloque está compuesto de dos contadores que tienen las mismas características que los ya simulados anteriormente, un bloque denominado Rotación el cual se encarga de hacer la habilitación de cada uno de los bits requeridos. La unidad de control general fue simulada con los vectores de entrada de la Tabla 4.23 obteniendo así la simulación de la Fig. 4.44.

Tabla 4.23 Vectores de entrada para la simulación de la unidad de control general.

Vector	clr	EN_GRAL
1	1	0
2-5	0	1
6-7	0	0
8-27	0	1
28-31	0	0
32-33	0	1
34	0	0
35	0	1
36	0	0
37	0	1
38	1	1
39-44	0	0
45	0	1

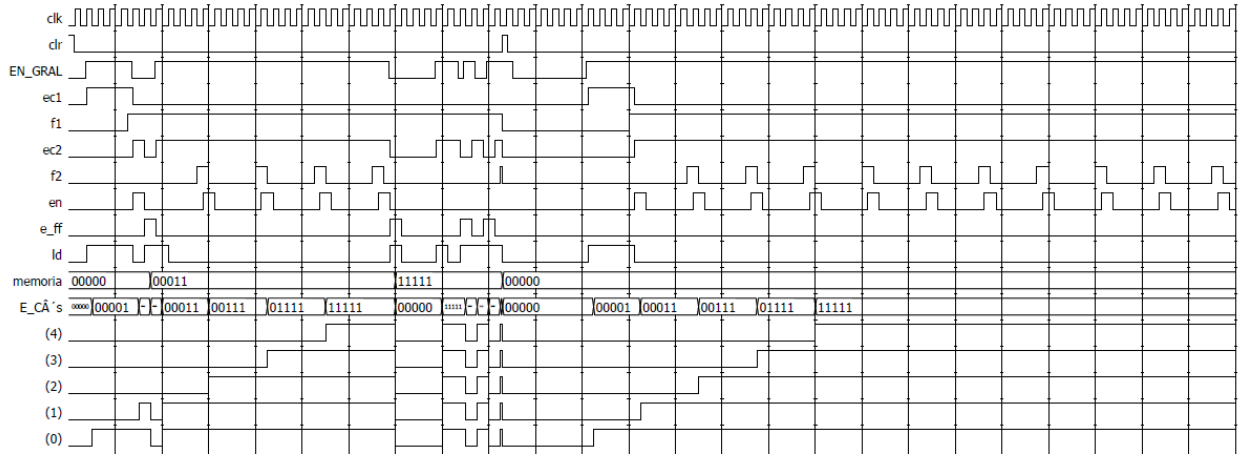


Fig. 4.44 Simulación funcional del bloque de control general.

Con el primer vector de entrada se puede observar que todas las señales intermedias se definen puesto que es la señal de inicialización de la arquitectura completa y se entra al estado A de la máquina de estados descrita en la Fig. 3.33, con el vector 2 en donde se hace la activación de *EN_GRAL* se hace la transición al estado B en donde se activa el primer contador con la señal *ec1* para generar la bandera *f1* y se activa la modalidad de carga del componente de corrimiento con el valor inicial lo que provoca que el vector *EN_C* tome dicho valor (0001 para una arquitectura de 8 puntos).

Los vectores 3,4, y 5 generan que la máquina de estados permanezca estática, es decir en el estado B para que no ocurra ninguna rotación, al presentarse el vector 6 se hace una transición al estado D aunque *EN_GRAL* haya sido desactivada, dado que se cumplen las condiciones necesarias, en este estado se deshabilita la carga, se habilita el contador c2 y se activa la rotación para activar la siguiente etapa, con el

vector 7 se hace una transición al estado F en donde se deshabilita el segundo contador y se habilita la carga con *ld*=1 cargando cero en todos los bits y activando *e_ff*=1 para guardar el vector a partir del cual se comenzará una vez que *EN_GRAL* sea activada de nuevo.

Al activarse de nuevo *EN_GRAL* como lo hace con el vector 8 se hace una transición al estado G en donde se activa nuevamente el segundo contador para continuar a partir de donde se detuvo la cuenta, se carga en el registro de corrimiento el valor almacenado en el estado F mediante la activación de *ld*, al presentarse el vector 9 sin cambios con respecto al 8 se hace la transición al estado E puesto que no se ha presentado la activación de *f2* y en este estado únicamente se deja activado el segundo contador con la finalidad de esperar a que la bandera que indica la transición se active y ahí se permanece hasta que esta señal es activada por el segundo contador.

Cuando la activación se da, hace la transición al estado D para nuevamente habilitar el corrimiento del vector y así activar una etapa más, durante los vectores subsecuentes hasta el 27 el autómata itera entre los estados D y E en donde se alcanzan a activar todas las atapas, al presentarse el vector 28 se ve interrumpida la señal *EN_GRAL* por lo que la transición se hace al estado F para almacenar el vector en donde se quedó y deshabilitar todas las etapas cargando ceros a la entrada del registro y al siguiente ciclo una transición al estado H para esperar ahí hasta la activación de *EN_GRAL*, en los vectores del 34 al 37 se da como entrada una secuencia de valores que permiten observar que todas las señales son desactivadas al mismo tiempo y durante el mismo tiempo en el que *EN_GRAL* se activa y desactiva.

Cuando se presenta un nuevo evento en *clr* como se hace con el vector 38, el autómata regresa al estado A en donde nuevamente se mantiene hasta presentarse en *EN_GRAL* una activación, lo que sucede en el vector 45 comenzando a activar una a una las etapas de la arquitectura. Con esto se puede corroborar que el funcionamiento de la unidad de control general se comporta tal cual se esperaba durante el diseño realizado. Cabe destacar que una vez que se han activado todas las etapas, la máquina permanece iterando entre los estados D y E siempre y cuando *EN_GRAL* esté activada, este aspecto no causa ningún problema puesto que aunque se sigue efectuando un corrimiento hacia la izquierda con un bit de acarreo, no se nota la diferencia entre cada operación ya que el vector en este punto está completamente activado. Finalmente, el circuito resultante de la síntesis de la entidad que se muestra en la Fig. 3.31 se puede observar en la Fig. 4.45.

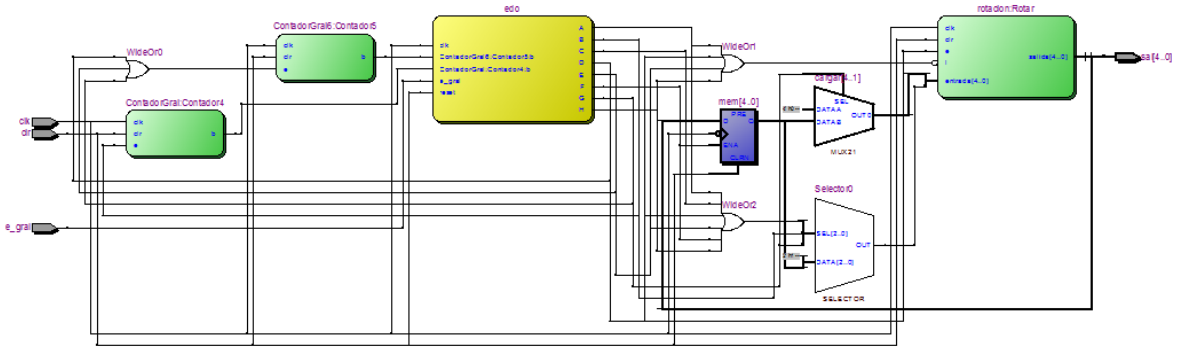


Fig. 4.45 Diagrama *RTL* de la unidad de control general.

Los resultados de la síntesis de la unidad de control general se muestran en la Fig. 4.46

Flow Summary	
Flow Status	Successful - Fri Jul 05 13:27:12 2013
Quartus II Version	11.0 Build 157 04/27/2011 SJ Web Edition
Revision Name	ControlGral
Top-level Entity Name	ControlGral
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	63 / 33,216 (< 1 %)
Total combinational functions	53 / 33,216 (< 1 %)
Dedicated logic registers	48 / 33,216 (< 1 %)
Total registers	48
Total pins	14 / 475 (3 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Fig. 4.46 Resultados de la síntesis de la unidad de control general

4.9 Pruebas del bloque de etapa de la arquitectura para la evaluación de la FFT

Siguiendo la metodología de diseño *top-down*, una vez que se han simulado por separado cada uno de los componentes que dan forma al bloque de etapa descrito en la Fig. 3.1 se dio paso a la integración en un bloque más grande para, en forma conjunta, y agregando algunos registros descritos en el capítulo 3, comprobar que cada una de las iteraciones es capaz de hacer correctamente el cálculo de la mariposa y de seguir la ruta de datos que se planteó en un inicio. La Fig. 4.47 muestra la simulación de una etapa de la evaluación del a FFT ejecutada con las entradas que se muestran en la Tabla. 4.24.

Tabla 4.24 Vectores de entrada para la simulación de una etapa de la arquitectura.

Vector	clr	E_C	m_d	g_wn	dato_aw	dato_bw	dir_aw	dir_bw
1	1	0	2	2	43760	1386	7	2
2	0	0	2	2	43760	1386	7	2
3	0	0	2	2	43760	1386	7	2
4	0	1	2	2	43760	1386	7	2
5	0	1	2	2	43760	1386	7	2

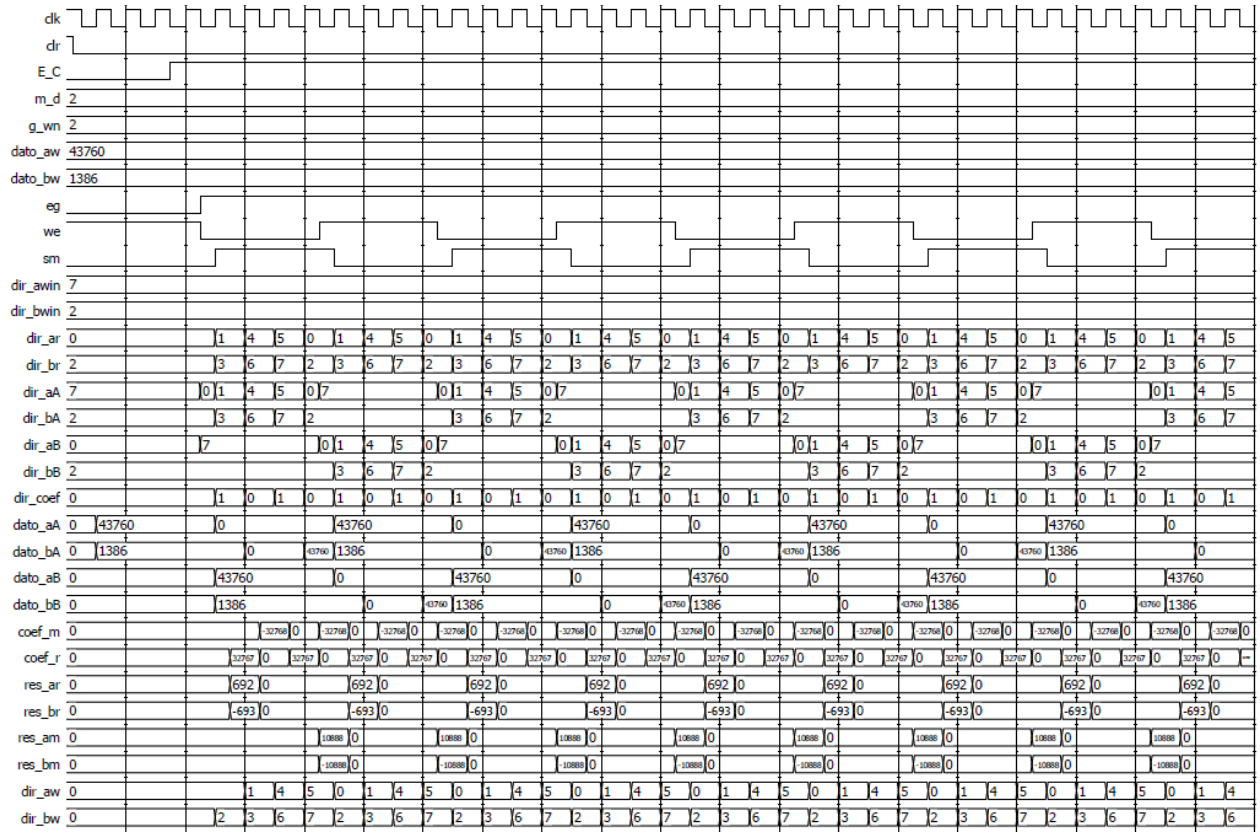


Fig. 4.47 Simulación de bloque de etapa en condiciones ideales.

La simulación anterior corresponde al bloque de la segunda etapa del algoritmo, esto es porque se parte del hecho de que todas las etapas de la arquitectura están conformadas por el mismo tipo de elementos y la ruta de datos es la misma, entonces se puede inferir que si funciona una etapa las demás están correctamente implementadas, sin embargo hay que hacer hincapié en la correcta inicialización de las memorias ROM del bloque generador de coeficientes complejos ya que un error dentro de los coeficientes provocaría resultados erróneos en la evaluación de la mariposa.

A continuación se hace un desarrollo de la ruta de datos que se sigue con base en los datos de entrada de la Tabla 4.24. Para fines de la verificación de la ruta de datos y control de una etapa se utilizan parámetros de entrada constantes en las direcciones de escritura y los datos de entrada, dado que estos provienen del bloque de la etapa anterior.

Se presenta la segunda etapa de la FFT en donde el número de mariposas por grupo es dos y el número de grupos por etapa igualmente (tomando en cuenta que está simulación se realizó para una FFT de 8 puntos con la finalidad de poder ser monitoreada más fácilmente). En la Tabla 4.25 se muestran los valores que, de acuerdo con diseño, adquieren las señales internas de control con base en las entradas *clk*, *clr* y *EN_C*.

Tabla 4.25 Comportamiento de las señales de *we, sm* y *eg* con base en las entradas de control.

Vector	***clk'event***	***Clr***	***EN_C***	***we***	***sm***	***eg***
1	1	1	0	1	0	0
2	0	0	0	1	0	0
3	1	0	0	1	0	0
4	0	0	0	1	0	0
5	1	0	0	1	0	0
6	0	0	0	1	0	0
7	1	0	0	1	0	0
8	0	0	1	1	0	0
9	1	0	1	1	0	0
10	0	0	1	**0**	0	**1**
11	1	0	1	0	**1**	1
12	0	0	1	0	1	1
13	1	0	1	0	1	1
14	0	0	1	0	1	1
15	1	0	1	0	1	1
16	0	0	1	0	1	1
17	1	0	1	1	1	1

Una vez que se ha presentado un evento en la señal *clr* como en el vector de entrada 1, las señales de control para la etapa se inicializan como se observa en la Tabla 4.25 en donde *we* indica que las operaciones comienzan por escribir en el *Banco_A* direccionado por *dir_awin* y *dir_bwin* los datos *dato_ain* y *dato_bin*, esta operación tarda cuatro ciclos de reloj y se realizan en flanco ascendente de reloj, mientras del *Banco_B* se realiza la operación de lectura direccionado por *dir_ar* y *dir_br*.

Una vez que se terminó de escribir los datos en *Banco_A* y que en *EN_C* (que proviene de la unidad de control general) ocurre un evento, en el próximo flanco descendente se genera un cambio en el valor de

las señales *we*, y *eg*, ahora *we* indica lectura para el *Banco_A* y escritura para *Banco_B*. La lectura de los datos se hace direccionada por los valores que se generan al haber un cambio en *eg*.

La tabla 4.26 indica el comportamiento de las direcciones ante los cambios en el valor de *eg*, se observa que cuando la señal *eg* adquiere el valor de 1 lógico comienza el cálculo de las direcciones de lectura y escritura lo que fue verificado en las simulaciones del bloque respectivo, mientras que en la Tabla 4.27 se observa la selección de direcciones por parte de *we* que es una entrada al multiplexor de direcciones; mientras *we* tiene el valor de uno, las direcciones seleccionadas son las de escritura, de lo contrario son las de lectura. Cada uno de los bancos de memoria en el bloque de la etapa tiene un par de multiplexores en donde la señal *we* llega complementada al *Banco_B*.

Tabla 4.26 Funcionamiento del bloque generador de direcciones.

eg	***dir_ar***	***dir_br***	***dir_aw***	***dir_bw***
0	0	2	0	2
0	0	2	0	2
0	0	2	0	2
0	0	2	0	2
1	1	3	0	2
1	4	6	1	3
1	5	7	4	6
1	0	2	5	7
1	1	3	0	2
1	4	6	1	3
1	5	7	4	6
1	0	2	5	7
1	1	3	0	2

Tabla 4.27 Selección de las direcciones para cada uno de los bancos de memoria mediante la señal *we*.

we	***dir_awin***	***dir_bwin***	***dir_ar***	***dir_br***	***dir_aa***	***dir_ba***	***dir_ab***	***dir_bb***
1	7	2	0	2	7	2	0	2
1	7	2	0	2	7	2	0	2
1	7	2	0	2	7	2	0	2
1	7	2	0	2	7	2	0	2
0	7	2	0	2	0	2	7	2
0	7	2	1	3	1	3	7	2
0	7	2	4	6	4	6	7	2
0	7	2	5	7	5	7	7	2
1	7	2	0	2	7	2	0	2
1	7	2	1	3	7	2	1	3
1	7	2	4	6	7	2	4	6
1	7	2	5	7	7	2	5	7

Mientras se escribían los primeros datos en *Banco_A* la señal *sm* se encontraba seleccionando los datos de salida de *Banco_B* y dado que las memorias fueron inicializadas con ceros en todas sus localidades los datos seleccionados eran ceros, pero una vez que se comienza a hacer lectura de *Banco_A* ahora *sm* debe seleccionar los datos de salida de éste banco, en la Tabla 4.28 se observa el comportamiento de dichas señales.

Tabla 4.28 Selección de los datos para ser evaluados por el bloque de mariposa mediante la señal *sm*.

sm	***dato_aa***	***dato_ba***	***dato_ab***	***dato_bb***	***dato_am***	***dato_bm***
0	43760	1386	0	0	0	0
0	43760	1386	0	0	0	0
0	43760	1386	0	0	0	0
0	43760	1386	0	0	0	0
0	43760	1386	0	0	0	0
1	0	1386	43760	1386	0	1386
1	0	0	43760	1386	0	0
1	0	0	43760	1386	0	0
1	0	43760	43760	1386	0	43760
0	43760	1386	0	1386	0	1386
0	43760	1386	0	0	0	0
0	43760	1386	0	0	0	0
0	43760	1386	0	43760	0	43760

Al tener los datos de los operandos disponibles para la evaluación de la mariposa se requieren también los coeficientes complejos que son generados a partir de la activación de la señal *eg*, la Tabla 4.29 muestra el comportamiento del bloque en donde se puede observar como la señal *eg* toma el valor de uno de los coeficientes complejos que comienzan a ser generados.

Tabla 4.29 Funcionamiento del bloque generador de los coeficientes complejos con base en el comportamiento de la señal eg.

Eg	*coef_r*	*coef_m*
0	0	0
0	0	0
0	0	0
0	0	0
1	32767	0
1	0	-32768
1	32767	0
1	0	-32768
1	32767	0
1	0	-32768
1	32767	0
1	0	-32768
1	32767	0
1	0	-32768

De acuerdo con el diseño, el bloque para la evaluación de la mariposa es un elemento completamente combinatorio, sin embargo a las entradas se encuentran registros que dan sincronía al elemento de cálculo. El comportamiento de la mariposa depende indirectamente del control de los registros, el cual se hace mediante las señales *eg* y *clk* en cada flanco de descenso. La Tabla 4.30 muestra el comportamiento del bloque.

Tabla 4.30 Funcionamiento de los registros de entrada al bloque para la evaluación del bloque de mariposa.

Eg	*dato_ar*	*dato_br*
0	0	0
0	0	0
0	0	0
0	0	0
1	0	1386
1	0	0
1	0	0
1	0	43760
1	0	1386
1	0	0
1	0	0
1	0	43760
1	0	1386

Ya que se tienen todos los datos necesarios para la evaluación de la mariposa, los datos son evaluados y pasan a ser las entradas de la siguiente etapa al igual que las direcciones de escritura calculadas anteriormente. En esta primera simulación se muestran condiciones ideales para las señales de entrada, en

la Fig. 4.48 se muestra la simulación para condiciones de entrada diferentes a la primera en la señal *EN_C*.

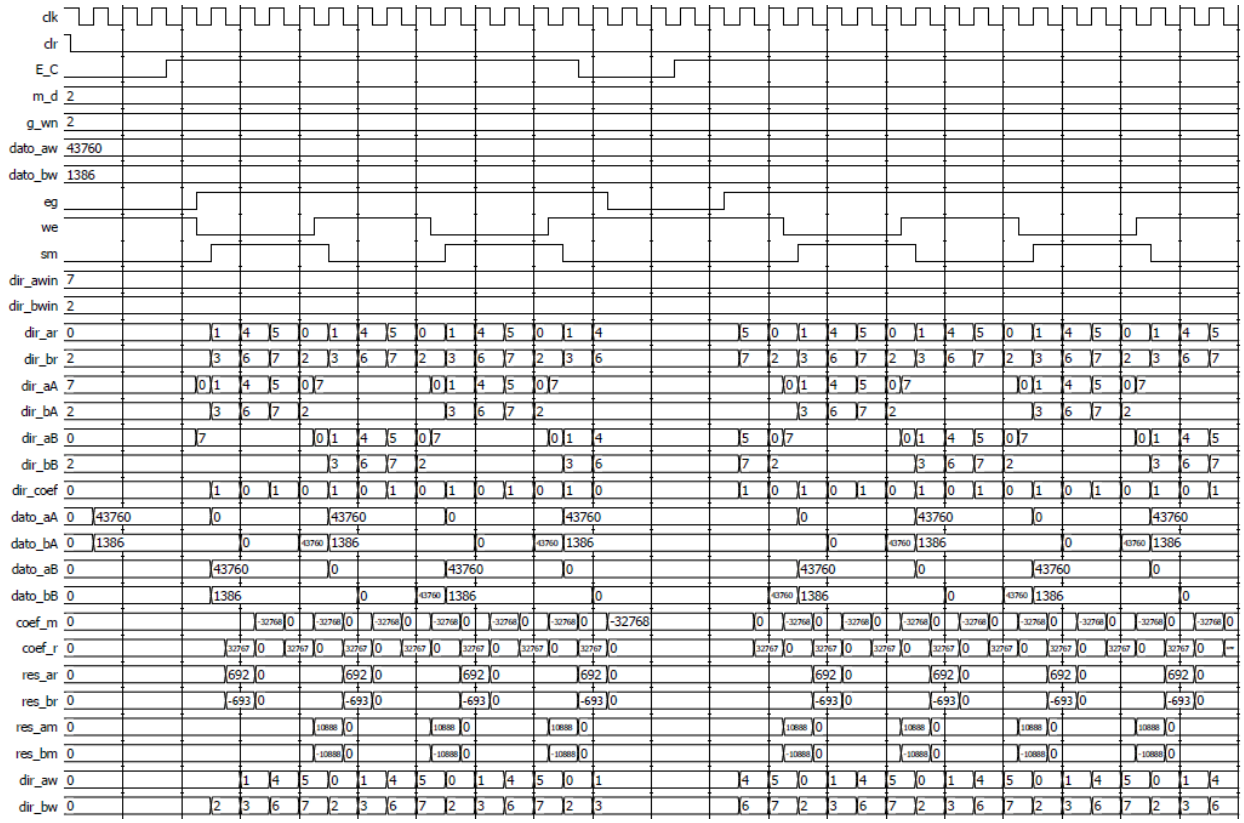


Fig. 4.48 Simulación de bloque de etapa con interrupción en la señal *EN_C*.

En la Fig.4.48 se observa que las operaciones en cada uno de los bloques se interrumpen de acuerdo con las entradas de habilitación de cada uno siendo estas *eg*, *we*, y *sm*. En la Tabla 4.31 se muestra el comportamiento de éstas ante la interrupción.

Tabla 4.31 Comportamiento de las señales *we*, *eg*, y *sm* ante la presencia de un cambio en *EN_C*.

clk´event	***EN_C***	***we***	***eg***	***sm***
1	1	1	1	1
0	1	1	1	1
1	1	1	1	0
0	0	1	1	0
1	0	1	1	0
0	0	1	0	0
1	0	1	0	0
0	0	1	0	0
1	0	1	0	0
0	0	1	0	0
1	1	1	0	0
0	1	1	0	0
1	1	1	0	0
0	1	1	1	0
1	1	1	1	0
0	1	1	1	0
1	1	1	1	0
0	1	0	1	0
1	1	0	1	0
0	1	0	1	1
1	1	0	1	1
0	1	0	1	1

La Tabla 4.31 muestra el comportamiento de las señales de control, gráficamente se puede observar que el tiempo que *EN_C* es deshabilitada es de tres ciclos de reloj completos, sin embargo las señales de control generadas con base en ella se deshabilitan por cuatro ciclos completos, esto es porque recordando el diseño de la unidad de control por etapa, el funcionamiento de las señales de control dependen del valor de la señal de habilitación del bloque de control de la señal *we* y éste se registra cada flanco negativo de reloj y dado que la habilitación de la *EN_C* se da en el nivel bajo de la señal de reloj éste no se registra sino hasta el siguiente flanco de descenso.

El que ocurra este pequeño retraso en la habilitación de las señales de control no afecta el flujo de datos ni el funcionamiento de la arquitectura puesto que este será a lo más de un ciclo de reloj. Para el caso especial de la última etapa, que es en donde se hace el cálculo del módulo de los resultados, los elementos que se diferencian son la unidad de mariposa que es reemplazada por el bloque de cálculo del módulo, se elimina el bloque generador de coeficientes complejos y se cambia el bloque generador de direcciones, la ruta de datos así como la lógica de control siguen iguales, lo que permite continuar con la conclusión de que al probar una etapa se puede inferir que el funcionamiento de las demás es correcto.

En la Fig. 4.49 se muestra el resultado de la síntesis del bloque para una etapa, es importante recordar que todos los bloques son similares, las diferencias se centran en la dimensión de las memorias del bloque generador de coeficientes complejos.

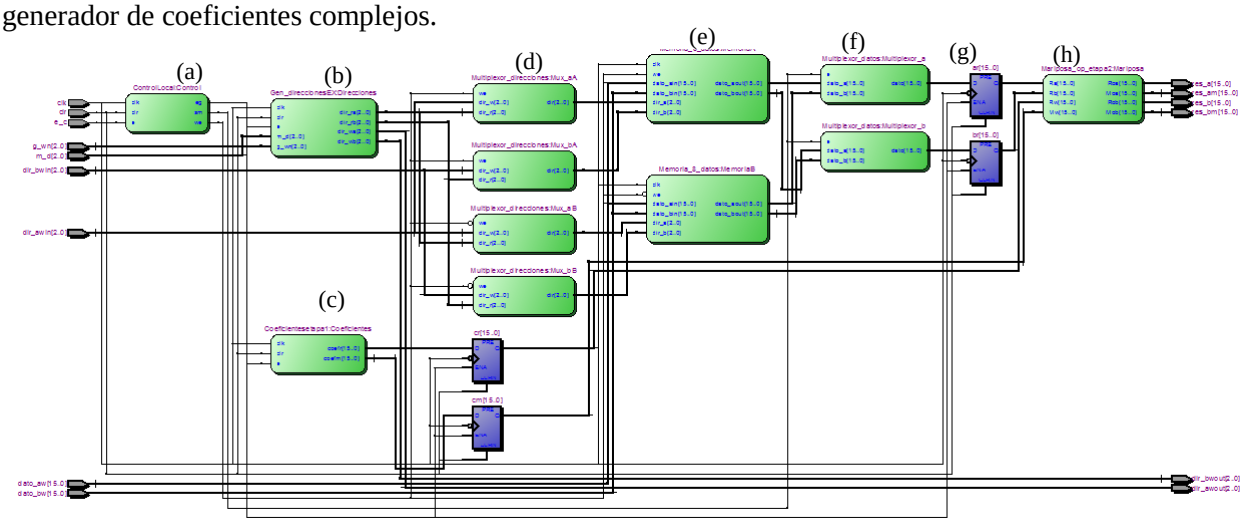


Fig. 4.49 Diagrama del circuito resultante de la síntesis del bloque de evaluación de una etapa de la arquitectura para evaluar la FFT (a) Unidad de control por etapa (b) Bloque generador de direcciones de datos (c) Bloque generador de coeficientes complejos (d) Multiplexores de direcciones (e) Bloques de memorias de datos (f) Multiplexores de datos (g) Registros de mariposa (h) Bloque para la evaluación de la mariposa.

Los resultados de la síntesis del bloque para la tercera etapa de la arquitectura se muestran en la Fig. 4.49, cabe señalar que estos resultados son parciales y diferentes para cada uno de los bloques de etapa puesto que existen diferencias de componentes entre ellos, como son la cantidad de memoria requerida para el bloque de coeficientes complejos y el numero de multiplicadores para las mariposas de la primera y segunda etapa.

Flow Summary

Flow Status	Successful - Fri Jul 05 13:35:57 2013
Quartus II Version	11.0 Build 157 04/27/2011 SJ Web Edition
Revision Name	ModuloC
Top-level Entity Name	ModuloC
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	364 / 33,216 (1 %)
Total combinational functions	353 / 33,216 (1 %)
Dedicated logic registers	140 / 33,216 (< 1 %)
Total registers	140
Total pins	185 / 475 (39 %)
Total virtual pins	0
Total memory bits	32,896 / 483,840 (7 %)
Embedded Multiplier 9-bit elements	8 / 70 (11 %)
Total PLLs	0 / 4 (0 %)

Fig. 4.50 Resultados de la síntesis para la tercera etapa de la arquitectura

4.10 Pruebas funcionales de la arquitectura

Una vez que se ha probado el funcionamiento de la ruta de datos del bloque que evalúa una etapa del algoritmo el siguiente paso fue integrar todos los bloques de etapa (incluyendo el bloque de cálculo del módulo de resultados) en una sola arquitectura para obtener como resultado la evaluación completa del algoritmo. En la Fig. 4.51 se observa una simulación funcional del algoritmo para una entrada de ocho puntos ejecutada con las entradas de los vectores de la Tabla 4.32.

Tabla 4.32 Vectores de entrada para la simulación funcional de la arquitectura para la evaluación de 8 puntos.

Vector	clr	EN_GRAL
1	1	0
2	0	0
3	0	1
4	0	0

En la simulación de la Fig. 4.50 se puede observar la secuencia de activación en cascada de cada una de las señales de control para los bloques de evaluación de las etapas de la arquitectura, para el caso de la evaluación de ocho puntos, se tienen cinco señales de activación. La Tabla 4.32 muestra que las entradas a la simulación son únicamente cuatro vectores, se inicia con un evento en la *clr* que permite inicializar todas las señales internas de la arquitectura, una vez hecho esto se mantiene en espera de la activación de la señal *EN_GRAL* para comenzar con la cascada de activación en el vector *EN_C*.

La tabla describe cómo es que las unidades de control por etapa se activan en forma secuencial, es decir, una detrás de la otra dando el tiempo suficiente para que se den las condiciones necesarias para el correcto funcionamiento de las iteraciones siguientes. Hacer una evaluación de la ruta de datos con una simulación de estas dimensiones se torna tedioso, en este caso, *EN_GRAL* no presenta alguna interrupción, lo que permite que la activación de todos los componentes se lleve a cabo de manera normal, la última señal mostrada en la simulación arroja las magnitudes del espectro de frecuencias de la señal tal como fue previsto en el diseño.

Una vez que se corroboró el funcionamiento en condiciones ideales, se ejecutó nuevamente la simulación de la arquitectura pero con los vectores de entrada de la Tabla. 4.33 en donde se presenta una interrupción a lo largo del desarrollo de la evaluación, la simulación resultante se muestra en la Fig. 4.52.

Tabla 4.33 Vectores de entrada para la simulación de la arquitectura completa con una interrupción en la señal de habilitación.

Vector	clr	EN_GRAL
1	1	0
2	0	0
3-35	0	1
36-38	0	0
38-	0	1

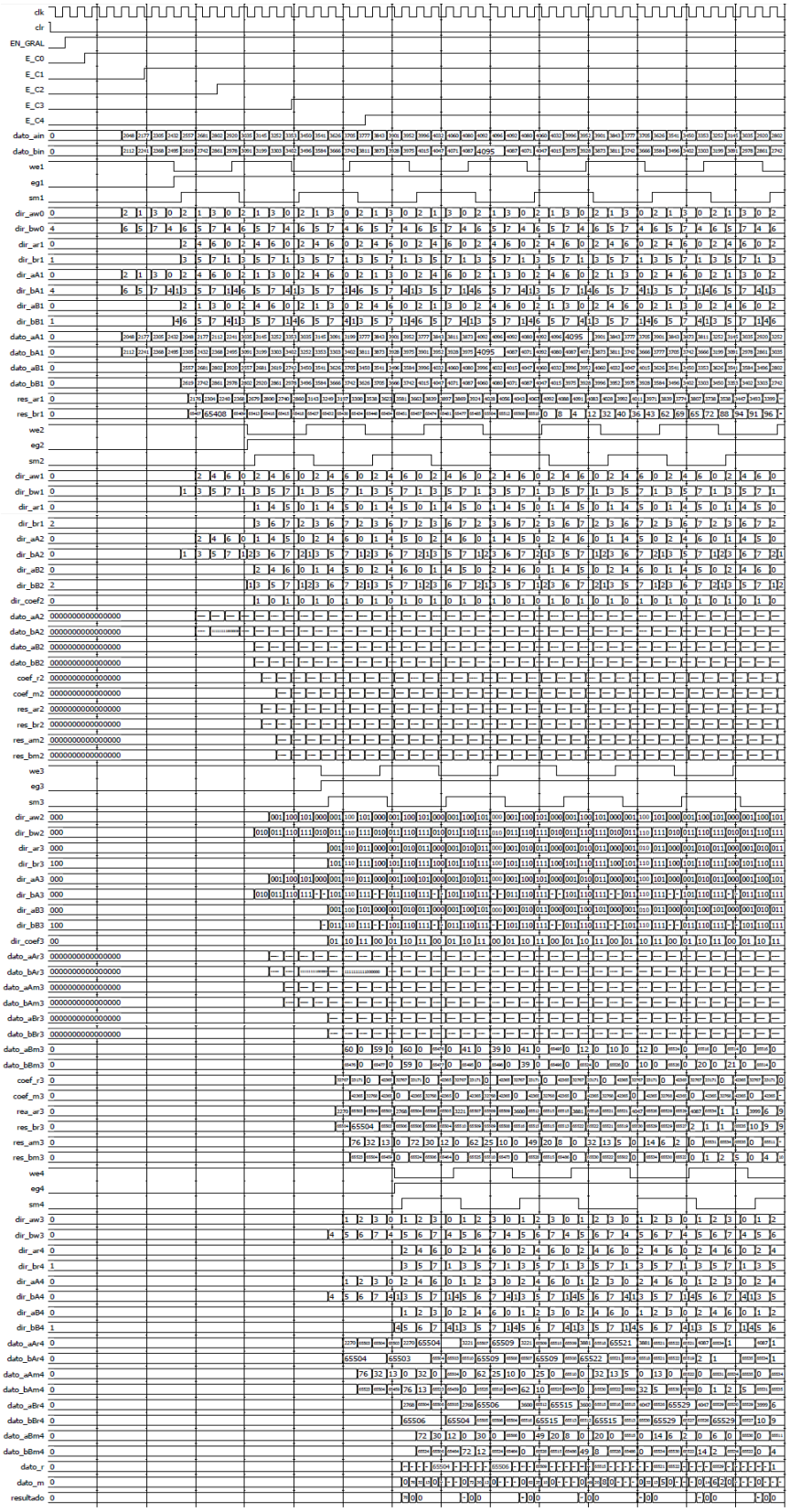


Fig. 4.51 Simulación funcional de la arquitectura.

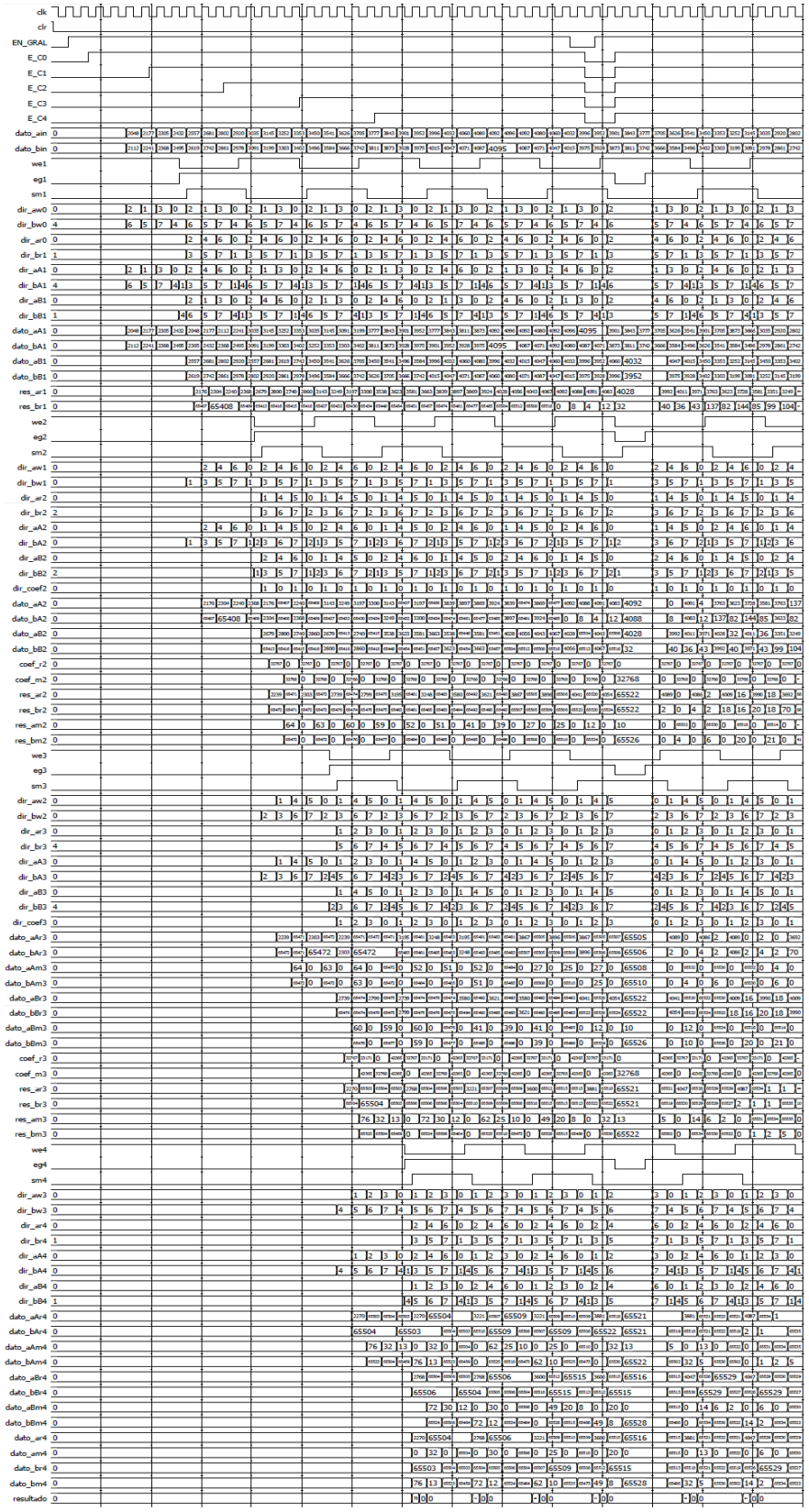


Fig. 4.52 Simulación de la arquitectura completa para la evaluación del algoritmo con una interrupción en EN_GRAL.

Se puede observar que *EN_GRAL* fue interrumpida durante dos ciclos de reloj completos, lo que de acuerdo con las condiciones del diseño genera una extensión en las señales *we* y *sm* pasando de cuatro a seis ciclos de duración. Se presenta un retraso en la desactivación de las actividades, esto es con la intención de asegurar que las condiciones previas a la activación sean correctamente respaldadas y así poder continuar con el desarrollo del algoritmo una vez que *EN_GRAL* vuelva a activarse.

Una manera superficial de corroborar que la ejecución de la ruta de datos se lleva de forma adecuada es observando la selección de las direcciones de operaciones en cada etapa una vez que se reactivó el funcionamiento de la arquitectura, la Fig. 4.52 muestra lo antes mencionado.

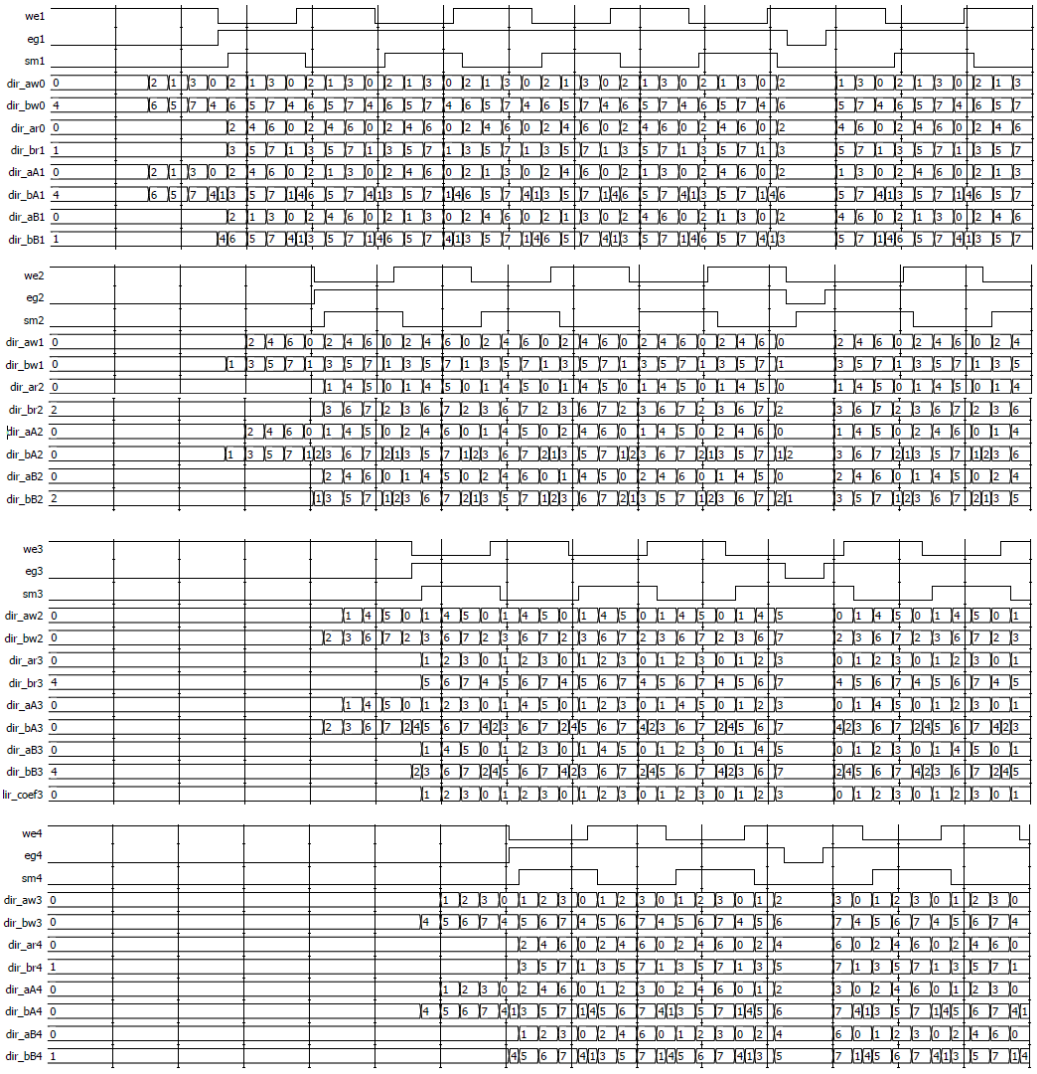


Fig. 4.53 Verificación de la selección de las direcciones de lectura/escritura de las memorias de datos.

Se puede observar que la selección de las direcciones en cada una de las iteraciones va desfasada una con respecto de otra, sin embargo, cuando ocurre la interrupción de la señal *EN_GRAL* todas las señales *eg's* son interrumpidas al mismo tiempo, lo que permite ya no generar más direcciones a partir de ese punto y mantener la última en memoria.

Las señales *we* y *sm* se mantienen en el estado anterior y las operaciones se detienen, entonces cuando se restablecen las condiciones se activan de nuevo los bloques y si se observan las señales *dir_aa1* y *dir_ba1* que indican las direcciones para operaciones en el *Banco_A* de la primera etapa en donde se escogen entre las direcciones de *bit-reverse* y las direcciones generadas dentro del bloque, una vez que se restablece el funcionamiento, cuando hay un cambio de función de escritura a lectura la selección de las direcciones cambia de 0,4 a 0,1 indicando que los bloques fueron puestos en funcionamiento correctamente. Lo mismo se puede observar en las señales *dir_aa2* y *dir_ba2* para la segunda etapa en donde se pasa de lectura a escritura seleccionando 0,1 en lugar de 0,2 y en las señales *dir_aa3* y *dir_ba3* para la tercera etapa pasando de lectura a escritura con las direcciones 0,2 en lugar de 0,4.

Finalmente, la Fig. 4.54 muestra el diagrama *RTL* que se obtiene después de la síntesis de la arquitectura para la evaluación de ocho puntos en donde se puede observar que hay un bloque por cada una de las etapas (en este caso 4 contando el cálculo del modulo de resultados), un bloque para la unidad de control general y otro para el bloque para el cálculo de direcciones *bit-reverse*. Como se planteó en el diseño de la arquitectura se observan dos señales de salida, una corresponde a un vector de 32 bits que contiene el resultado del cálculo de las magnitudes del espectro de frecuencias, como se mencionó en el apartado del diseño, este vector da como salida a dos resultados por cada ciclo de reloj de entrada, es decir, un resultado en el flanco de ascenso y otro en el flanco de descenso, la otra señal de salida corresponde a la habilitación del bloque de cálculo del módulo de resultados que indica el momento en que comienzan a salir los resultados, esta señal es interna pero se consideró necesaria únicamente como indicativo de la salida de resultados.

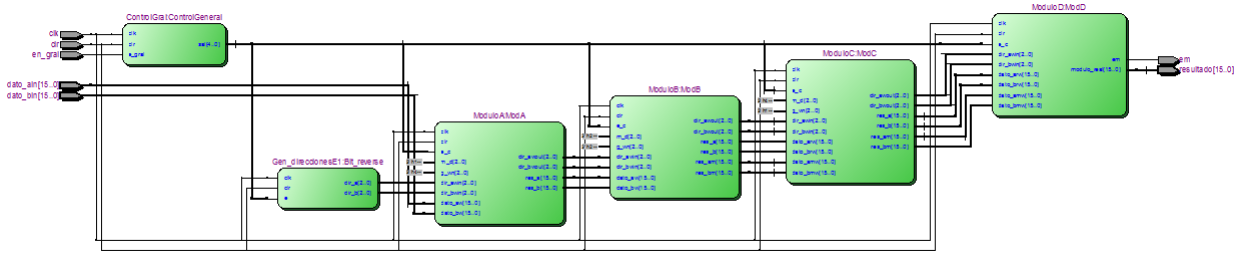


Fig. 4.54 Diagrama RTL del circuito resultante de la arquitectura para la evaluación de la FFT de 8 puntos.

Capítulo 5. Presentación de resultados

5.1 Arquitectura para la evaluación de 512 puntos

Como se observó en el capítulo anterior, todas las pruebas fueron hechas sobre una arquitectura para la evaluación de ocho puntos partiendo del entendido de que sería más sencillo verificar estado por estado el comportamiento de los diversos componentes. Una vez que se ha verificado que la ruta de datos así como la lógica de control de la arquitectura funcionan adecuadamente se hizo una extensión de la arquitectura de tal modo que ahora se pueden evaluar 512 puntos.

En la Tabla 5.1 se describen los cambios necesarios que se hicieron a los bloques haciendo hincapié en que la estructura del bloque para la evaluación de una etapa conserva la misma forma, es decir, contiene exactamente los mismos elementos que los que una arquitectura para la evaluación de ocho puntos contiene.

Tabla 5.1 Cambios realizados para la arquitectura de 512 puntos.

Componente	**Cambios realizados**
Bancos de memoria de datos	Se hizo una extensión en la capacidad de la memoria, de ocho a 512 palabras de 16 bits, por tanto, la resolución de los vectores de direcciones cambia de tres a nueve bits.
Bloque de *bit-reverse*	Se aumentó la resolución del contador de tres a nueve bits para ser capaz de direccionar a los bancos de memoria de datos.
Bloque generador de direcciones	Se aumentó la resolución del contador de tres a nueve bits para ser capaz de direccionar a los bancos de memoria de datos.
Bloque generador de coeficientes complejos	Se añadieron seis bloques más, cada uno del doble de elementos que el anterior llegando a tener un bloque con 256 coeficientes puesto que el número de iteraciones necesarias para la evaluación del algoritmo aumentó y por ende la cantidad de coeficientes.
Bloque para la evaluación de la mariposa	Se mantiene igual
Bloque de cálculo del módulo de resultados	Se mantiene igual
Elemento de control de la señal *we*	Se aumentó la resolución del contador de periodicidad (*Contador ModN/2*), de cuatro a 256 ciclos para aumentar el tiempo de lectura y escritura de los bancos de memoria de datos.

Elemento de control para la señal *sm*	Se aumentó la resolución del contador de periodicidad (*Contador ModN/2*) de cuatro a 256 ciclos para seleccionar correctamente los operandos para la evaluación de la mariposa.
Unidad de control general	Se aumentó la resolución de los contadores de transición, de cuatro a 256 y de cinco a 257 que se utilizan como habilitadores de los estados.

La Tabla 5.1 muestra que los cambios realizados a los elementos no son mayores, estos se reducen a la expansión de las memorias de datos y al aumento de la resolución de los contadores para el control de las señales, esto permite que un rediseño de la ruta de datos así como de la lógica de control no sea necesario.

El circuito resultante de la síntesis del proyecto completo para la evaluación de 512 puntos se muestra en la Fig. 5.1 en donde se ve que se cuentan con 10 bloques, cada uno para la evaluación de una etapa del algoritmo (nueve para el algoritmo y una para el cálculo del módulo), un bloque para la unidad de control general y uno para el generador *bit-reverse*. La Fig. 5.2 muestra la estructura interna de cada bloque de etapa.

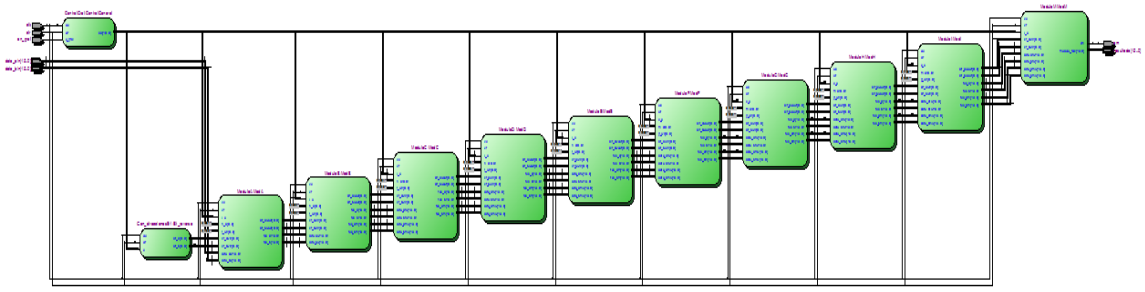


Fig. 5.1 Diagrama RTL de la arquitectura para el cálculo de la FFT de 512 puntos.

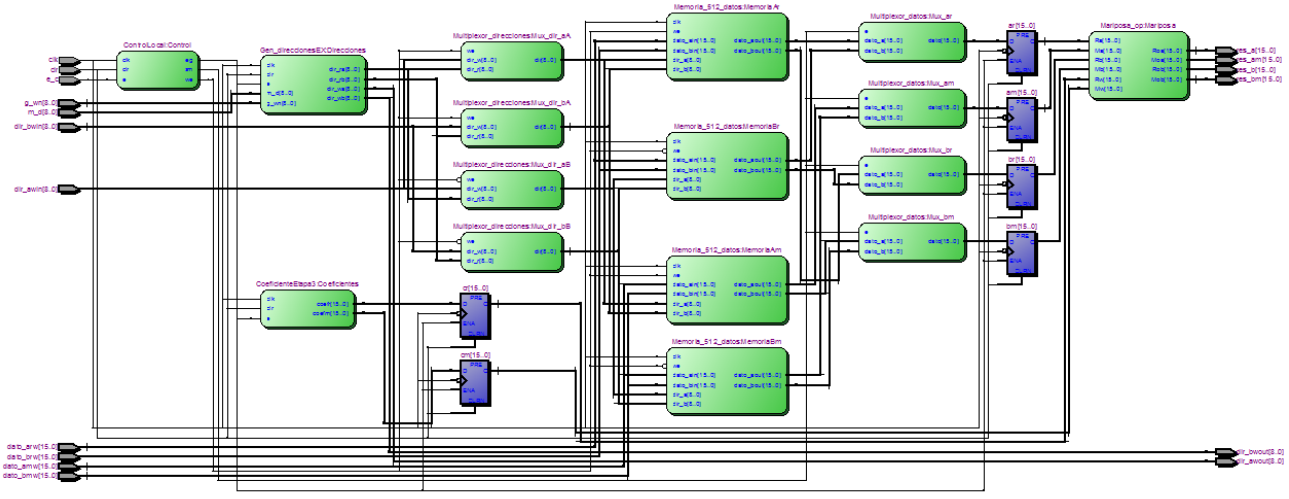


Fig. 5.2 Diagrama *RTL* del circuito para la evaluación de una etapa de la FFT de 512 puntos.

Como se observa a simple vista, el circuito se sintetiza de igual manera que para una arquitectura de ocho puntos mostrada en la Fig. 4.49 sin embargo las diferencias se encuentran como ya se describió en la Tabla 5.1, además de que cada uno de los bloques de etapa contiene un componente generador de coeficientes complejos diferente, recordando que conforme se avanza en la evaluación del algoritmo se requiere de una cantidad mayor de coeficientes complejos.

5.2 Comparativa de los resultados hardware-software de la evaluación

En el capítulo 4 se realizaron las pruebas funcionales de cada uno de los bloques por separado, sin embargo, la arquitectura no había sido probada con una señal bien definida, esto se hace en esta sección en donde se ejecuta la simulación de la arquitectura con datos de prueba los cuales fueron obtenidos a partir de Matlab.

Se utilizaron tres señales de prueba, senoidal, cuadrada y diente de sierra, estas fueron generadas por rutinas en Matlab, las cuales tienen una frecuencia fundamental de 100 Hz y fueron muestreadas a 10KHz, de estos se obtuvieron dos archivos de texto por cada forma de onda.

- Datos_decimales.txt: Contiene 512 muestras de la señal en un formato de número decimal el cual fue utilizado como entrada para el algoritmo programado en C que evalúa la FFT en formato de punto flotante y de manera iterativa, el código del programa se muestra en el Anexo A.
- Datos_binarios.txt: Contiene 512 muestras de la señal en formato de número binario con tamaño de palabra de 16 bits, este archivo fue utilizado como entrada de datos para las simulaciones realizadas con *ModelSim-Altera*

La evaluación del algoritmo con el programa en C permitió hacer una comparación etapa por etapa y mariposa por mariposa con los datos resultantes de la simulación de la arquitectura propuesta, dichas comparaciones sirvieron para observar las diferencias existentes entre la evaluación software (hecha con el programa en C) y la evaluación hardware (hecha con la arquitectura propuesta).

- **Comparativa de resultados para la señal de entrada senoidal**

Se utilizaron 512 muestras de una señal senoidal de frecuencia fundamental de 100 Hz tomadas a una frecuencia de muestreo de 10 KHz con una resolución de 14 bits y una amplitud máxima de 16382 unidades, las Fig. 5.3, Fig. 5.4, Fig. 5.5 y Fig. 5.6 muestran las gráficas comparativas de los resultados arrojados por la mariposa de la última etapa del algoritmo, una gráfica por cada resultado (Ar, Br, Am,

Bm) en donde se pueden observas las diferencias existentes entre una y otra evaluación, cada imagen contiene dos gráficas, la parte superior representa los resultados completos de la última evaluación mientras que la parte de abajo corresponde a un acercamiento en la zona en donde hay mayor concentración de información con la finalidad de hacer más notorias las diferencias entre uno y otro.

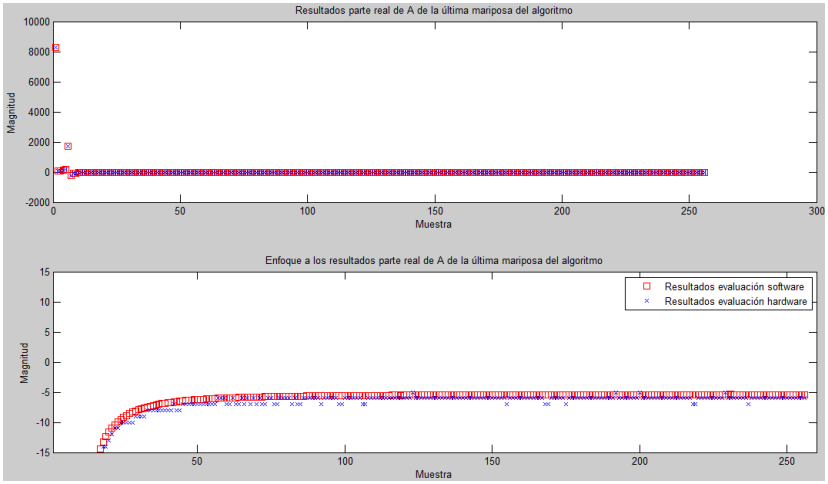


Fig. 5.3 Resultado parte real de A de la mariposa de la última etapa de evaluación de la señal senoidal.

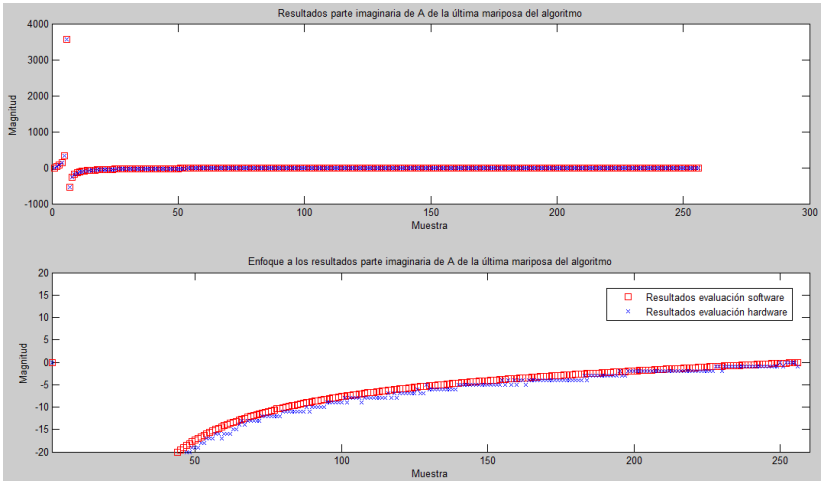


Fig. 5.4 Resultado parte imaginaria de A de la mariposa de la última etapa de evaluación de la señal senoidal.

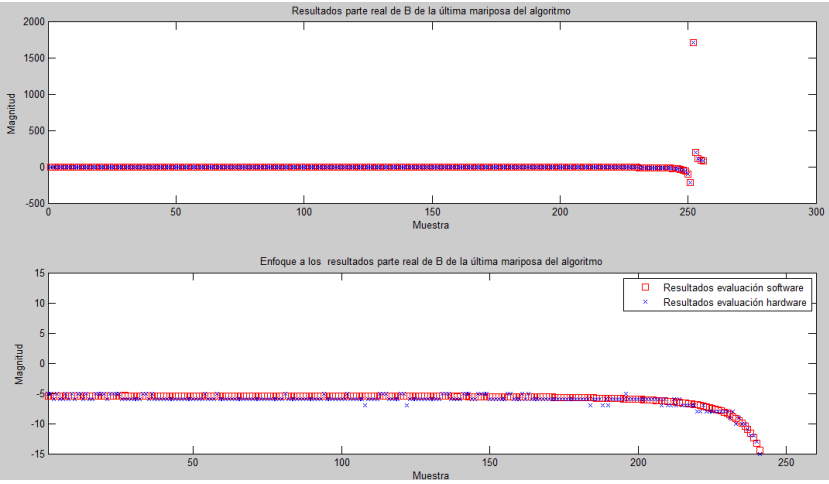


Fig. 5.5 Resultado parte real de B de la mariposa de la última etapa de evaluación de la señal senoidal.

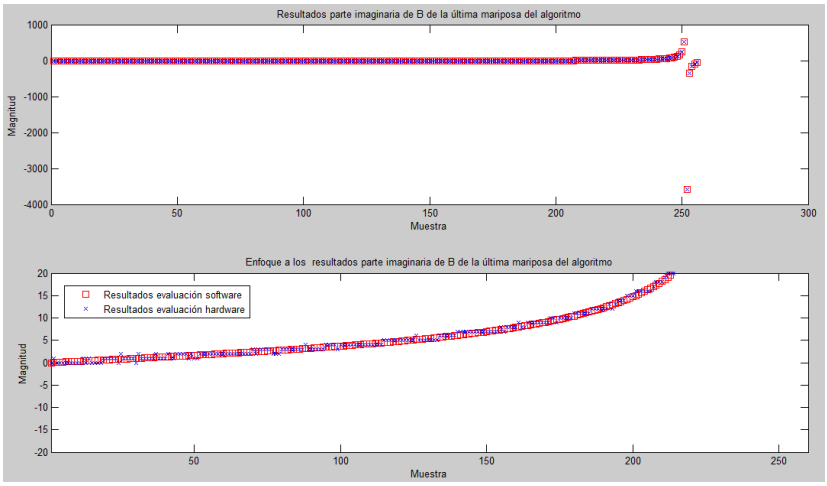


Fig. 5.6 Resultado parte imaginaria de B de la mariposa de la última etapa de evaluación de la señal senoidal.

En cada una de las gráficas se muestra el comportamiento de dos curvas de valores, la curva representada por los cuadros corresponde a los valores arrojados por la evaluación en C (evaluación software), mientras que la curva representada por las cruces corresponde a los resultados dados por la arquitectura propuesta (evaluación hardware). Es notable que no exista gran diferencia entre una curva y otra, es decir, las magnitudes de los resultados son muy similares, recordando que estos resultados son los correspondientes a la evaluación de la mariposa de la última etapa. La Fig. 5.7 muestra las gráficas correspondientes a la señal de entrada, el espectro de frecuencia obtenido por MatLab, el espectro de frecuencia obtenido por la evaluación software y el espectro de frecuencia obtenido por la evaluación hardware, en donde se aprecia que las componentes de frecuencia se encuentran en la misma muestra para todos los casos, sin embargo, es importante mencionar que se hizo una normalización de los resultados obtenidos con la evaluación por

Matlab para tener una mejor presentación, recordando que las evaluaciones con C y con la arquitectura propuesta presentan truncamientos y escalamientos a lo largo de la evaluación del algoritmo.

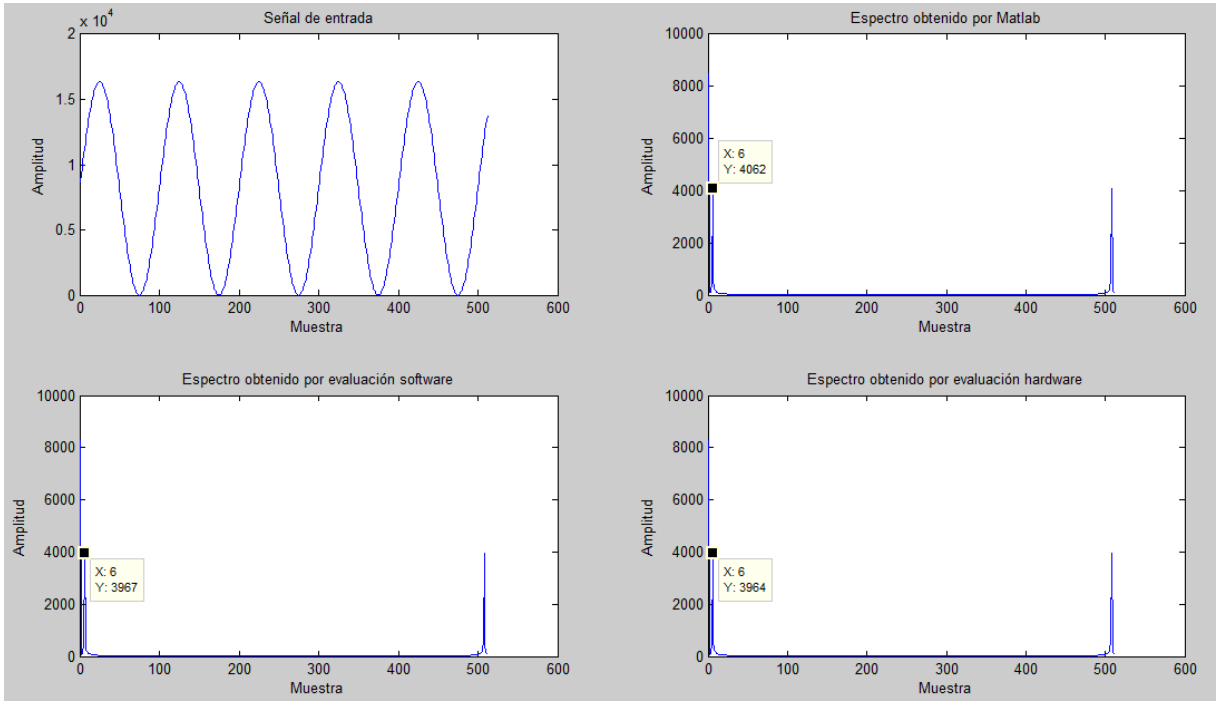


Fig. 5.7 Comparativa de los resultados para la señal senoidal de entrada.

Al comparar las evaluaciones software y hardware se pudo observar que existen diferencias entre los resultados de estas dos, por cada etapa fueron calculadas las diferencias entre cada dato, después se obtuvo el máximo valor en cada etapa de tal manera que en la gráfica de la Fig. 5.8 se muestra la evolución de las diferencias a lo largo de la evaluación del algoritmo para la señal de entrada senoidal.

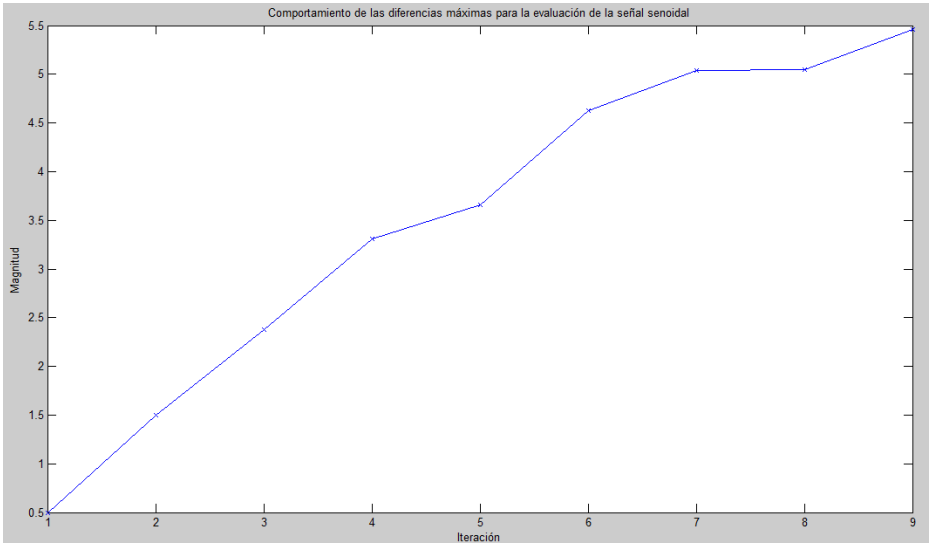


Fig. 5.8 Evolución de las diferencias para la evaluación software vs hardware con entrada de señal senoidal.

Con cada etapa se obtiene un aumento en la diferencia máxima de los resultados, para este caso en particular, se puede observar que la diferencia en las primeras etapas del algoritmo crece más rápido y que a partir de la etapa 5 las diferencias crecen más lentamente, sin embargo se llega al punto máximo de 5.46 unidades valor absoluto, cabe resaltar que este error está dado en la primera componente del espectro, o sea en la componente de corriente directa del espectro correspondiente.

- **Comparativa de resultados para la señal de entrada cuadrada**

Se utilizaron 512 muestras de una señal cuadrada con una frecuencia fundamental de 100 Hz muestreada a 10 KHz con una resolución de 14 bits teniendo una amplitud máxima de 16300 unidades, las Fig. 5.9, Fig. 5.10, Fig. 5.11 y Fig. 5.12 muestran las gráficas comparativas de los resultados arrojados por la mariposa de la última etapa del algoritmo, una gráfica por cada resultado (Ar, Br, Am, Bm) en donde se pueden observar las diferencias existentes entre una y otra evaluación, de igual manera, cada grafica está compuesta de dos graficas, la superior muestra los resultados tal cual fueron calculados mientras que la segunda muestra un acercamiento a la zona de mayor concentración de información en donde las diferencias pueden hacerse más notorias.

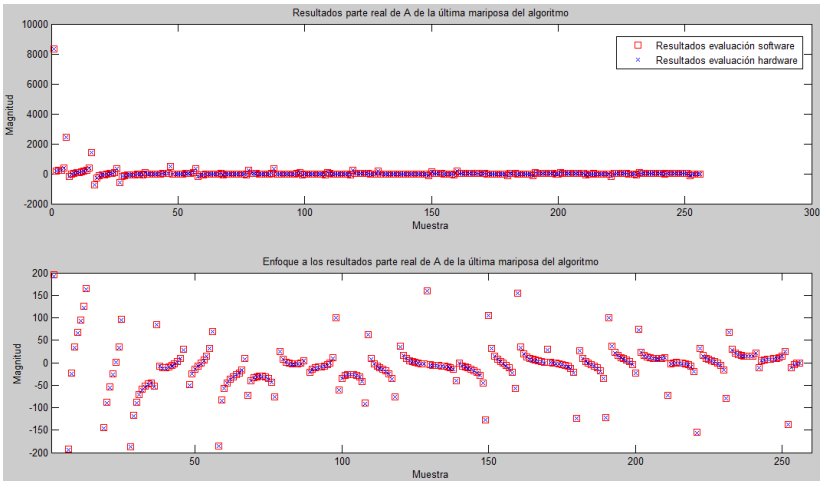


Fig. 5.9 Resultado parte real de A de la mariposa de la última etapa de evaluación de la señal cuadrada.

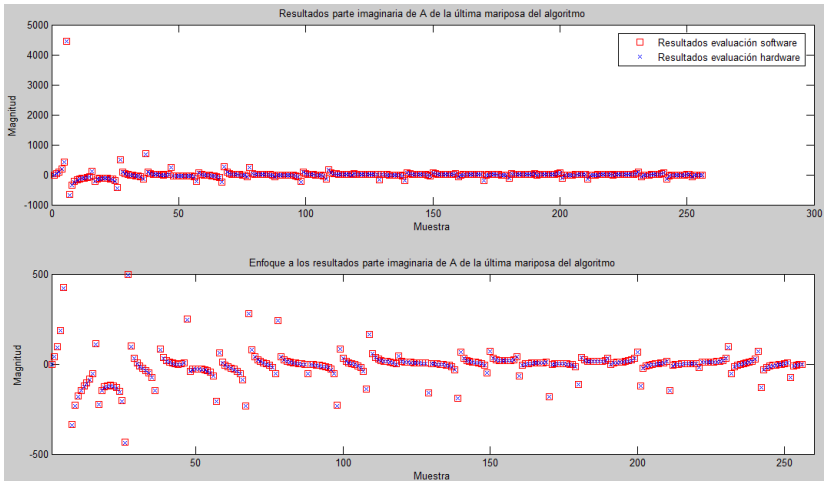


Fig. 5.10 Resultado parte imaginaria de A de la mariposa de la última etapa de evaluación de la señal cuadrada.

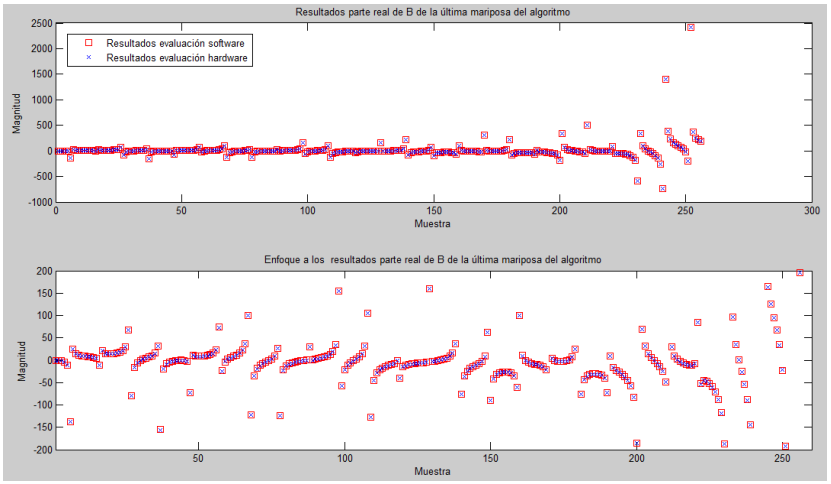


Fig. 5.11 Resultado parte real de B de la mariposa de la última etapa de evaluación de la señal cuadrada.

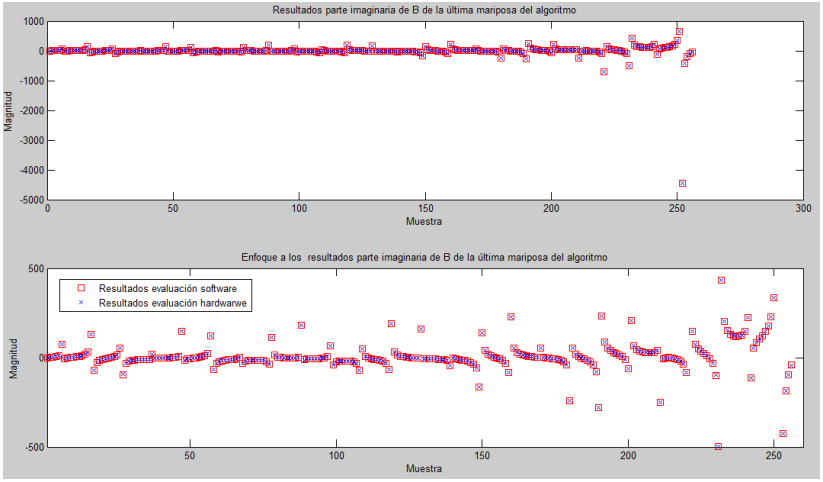


Fig. 5.12 Resultado parte imaginaria de B de la mariposa de la última etapa de evaluación de la señal cuadrada.

En cada una de las gráficas se muestra el comportamiento de dos curvas de valores, la curva representada por los cuadros rojos corresponde a los valores arrojados por la evaluación software hecha con el programa en C, mientras que la curva representada por las cruces corresponde a los resultados dados por la arquitectura propuesta.

Es notable que no exista gran diferencia entre una curva y otra, es decir, las magnitudes de los resultados son muy similares, recordando que estos resultados son los correspondientes a la evaluación de la mariposa de la última etapa. En la Fig. 5.13 se muestran las gráficas correspondientes a la señal de entrada, el espectro de frecuencia obtenido por Matlab, el espectro de frecuencia obtenido por la evaluación en C y el espectro de frecuencia generado por la arquitectura en el FPGA, en donde se puede apreciar que al igual que en el caso de la señal senoidal, la forma de los espectros calculados es similar ubicando las componentes de frecuencia en la misma muestra, aunque hubo la necesidad de hacer una normalización en los resultados obtenidos por MatLab.

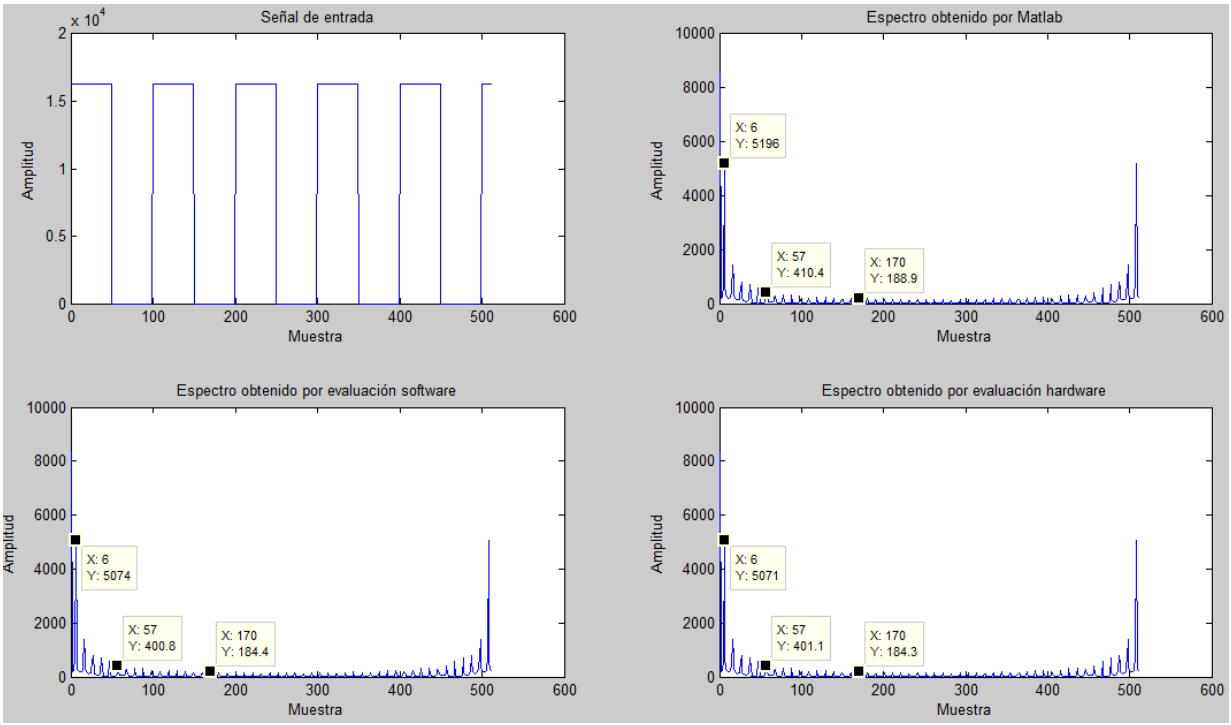


Fig. 5.13 Comparativa de los resultados para la señal cuadrada de entrada.

Se hizo también un análisis del comportamiento de las diferencias existentes a lo largo de la evaluación del algoritmo, la Fig. 5.14 presenta la curva del comportamiento de estas donde se observa que en la última etapa es en donde se encuentra la diferencia máxima con un valor de 7.02 unidades. De manera similar que para la señal senoidal, se observa que la diferencia máxima entres los resultados de las

evaluaciones tiende a incrementarse con respecto a la etapa que se esté calculando y que el aumento se hace constante en una unidad en la mayoría de las etapas evaluadas.

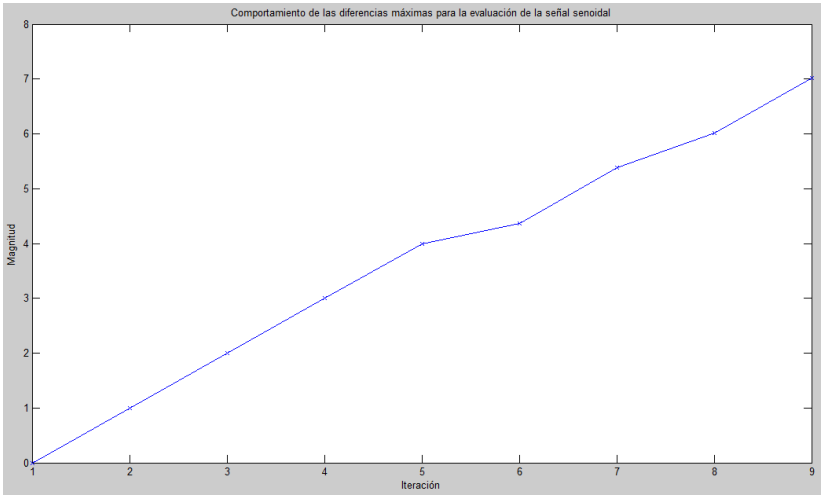


Fig. 5.14 Evolución de las diferencias para la evaluación software vs hardware con entrada de señal cuadrada.

- **Comparativa de resultados para la señal de entrada diente de sierra**

Se utilizaron 512 muestras de una señal diente de sierra con una frecuencia fundamental de 100 Hz muestreada a 10 KHz con una resolución de 14 bits teniendo una amplitud máxima de 16300 unidades, las Fig. 5.15, Fig. 5.16, Fig. 5.17 y Fig. 5.18 muestran las gráficas comparativas de los resultados arrojados por la mariposa de la última etapa del algoritmo, una gráfica por cada resultado (Ar, Am, Br Bm) en donde se pueden observar las diferencias existentes entre una y otra evaluación para una señal siente de sierra, nuevamente se muestran las gráficas tal cual fueron calculadas y un acercamiento a la zona de mayor concentración de información.

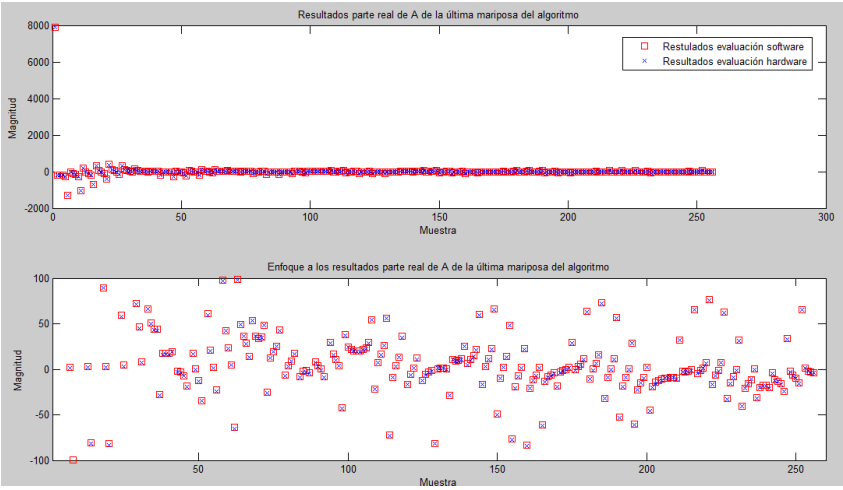


Fig. 5.15 Resultado parte real de A de la mariposa de la última etapa de evaluación de la señal diente de sierra.

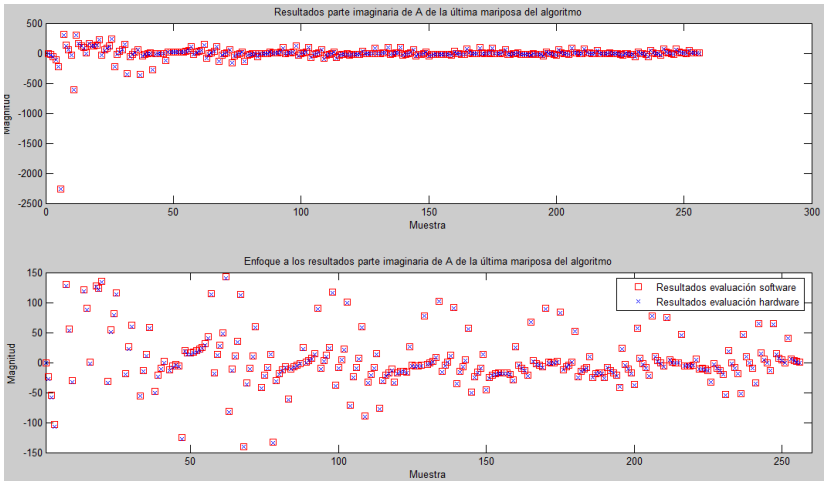


Fig. 5.16 Resultado parte imaginaria de A de la mariposa de la última etapa de evaluación de la señal diente de sierra.

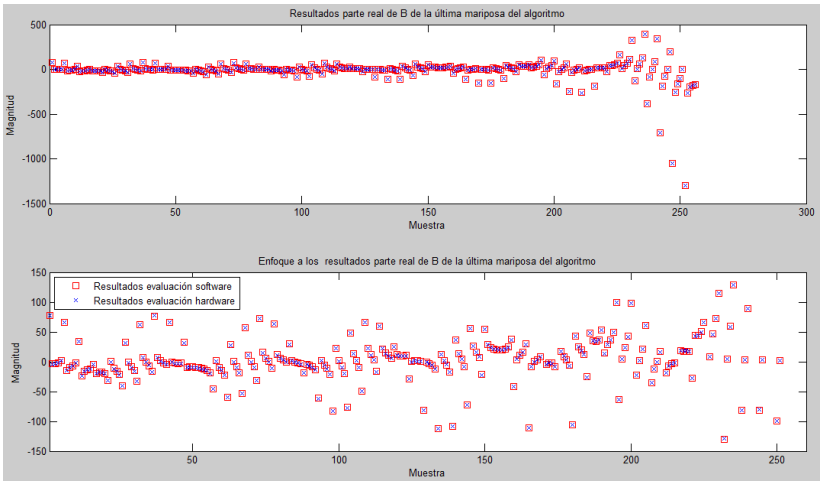


Fig. 5.17 Resultado parte real de B de la mariposa de la última etapa de evaluación de la señal diente de sierra.

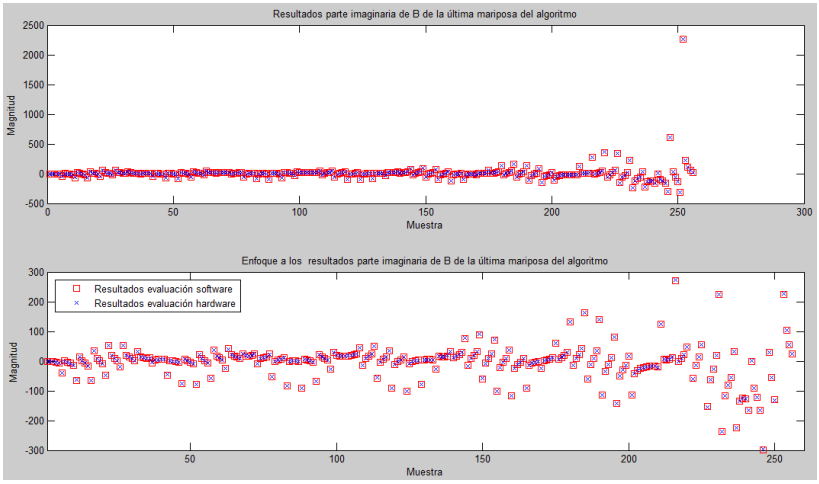


Fig. 5.18 Resultado parte imaginaria de B de la mariposa de la última etapa de evaluación de la señal diente de sierra.

Como se describió para los dos casos anteriores, en cada una de las gráficas se muestra el comportamiento de dos curvas de valores, la curva representada por los cuadros corresponde a los valores arrojados por la evaluación en C, mientras que la curva representada por las cruces corresponde a los resultados dados por la arquitectura propuesta. Las diferencias que existen entre cada una de la evaluaciones siguen siendo casi transparentes para la graficación, esto no quiere decir que no existan, sino que no son tan grandes como para ser perceptibles en las gráficas. En la Fig. 5.19 se muestran las gráficas correspondientes a la señal de entrada, el espectro de frecuencia obtenido por Matlab, el espectro de frecuencia obtenido por la evaluación en C y el espectro de frecuencia generado por la arquitectura en el FPGA.

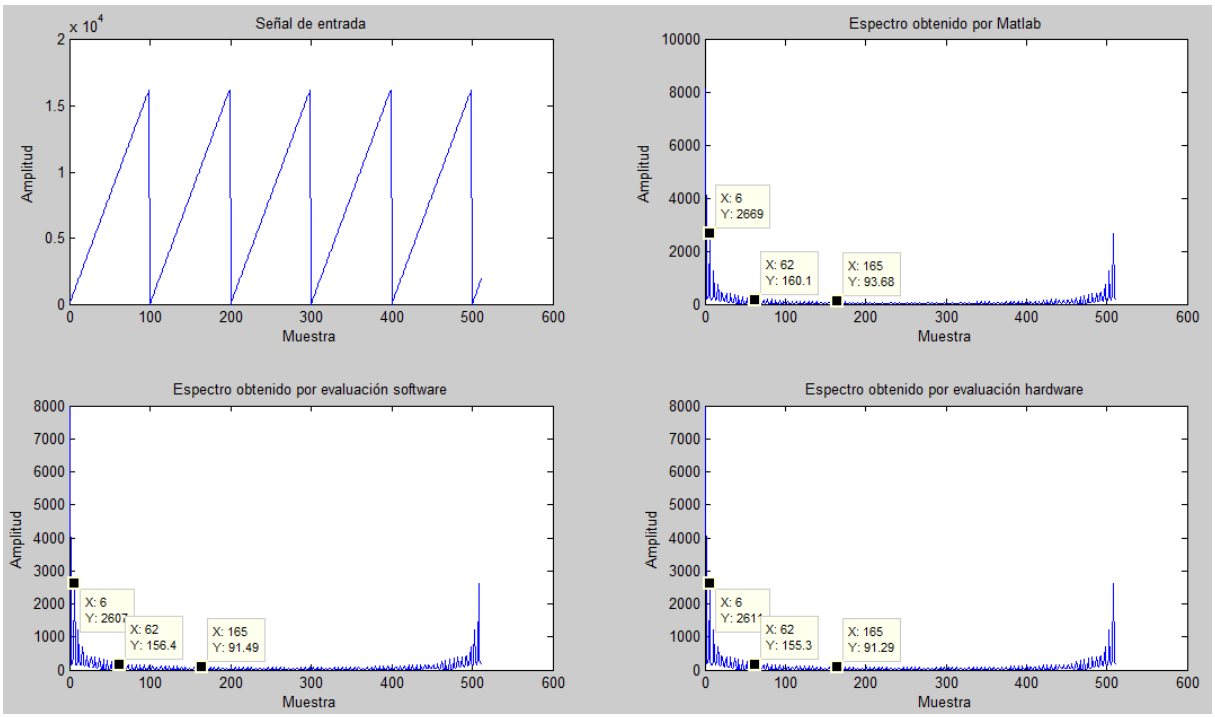


Fig. 5.19 Comparativa de los resultados para la señal diente de sierra de entrada.

Al igual que los casos anteriores, aunque las gráficas de los espectros son muy parecidas, existe la presencia de diferencias entre los resultados, la gráfica de la Fig. 5.20 muestra el comportamiento de las diferencias máximas a lo largo de las etapas de la arquitectura para la evaluación del algoritmo en donde se tiene que la diferencia más grande es la que se presenta en la última etapa con un valor de 6.36 unidades valor absoluto.

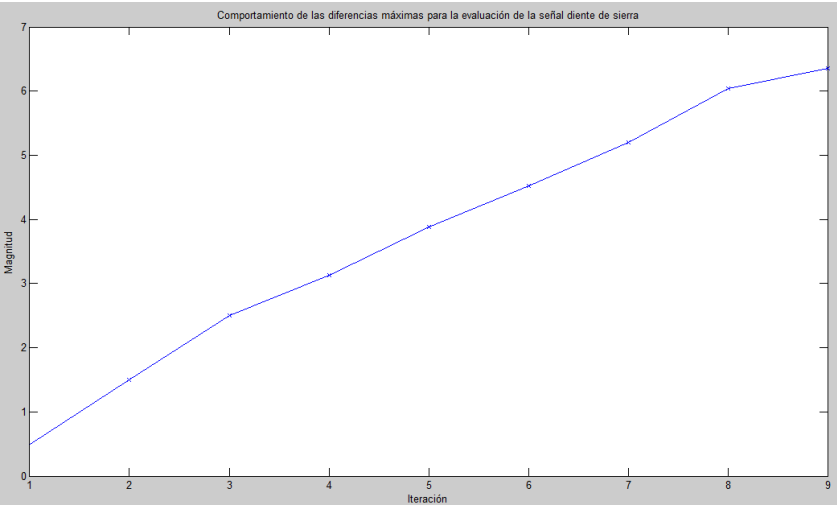


Fig. 5.20 Evolución de las diferencias para la evaluación software vs hardware con entrada de señal diente de sierra.

- **Análisis de resultados para una señal de voz**

Después de haber observado el comportamiento de la arquitectura propuesta con señales que se podría decir, son ideales como las que se utilizaron, se decidió utilizar datos provenientes de una señal de voz correspondiente al fonema A, esta señal fue adquirida tomando muestras a una frecuencia de 44 KHz con una resolución de 13 bits. Las Fig. 5.21, Fig. 5.22, Fig. 5.23 y Fig. 5.24 muestran las graficas comparativas de los resultados arrojados por la mariposa de la última etapa del algoritmo, una grafica por cada resultado (Ar, Am, Br Bm).

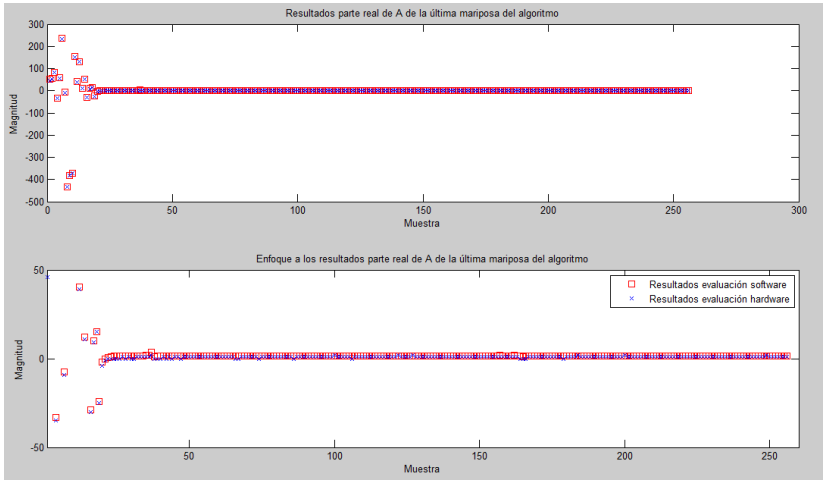


Fig. 5.21 Resultado parte real de A de la mariposa de la última etapa de evaluación de la señal de voz.

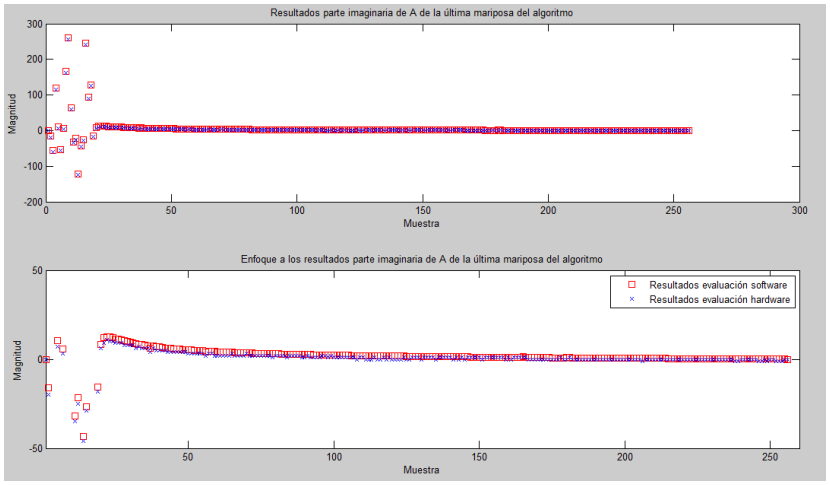


Fig. 5.22 Resultado parte imaginaria de A de la mariposa de la última etapa de evaluación de la señal de voz.

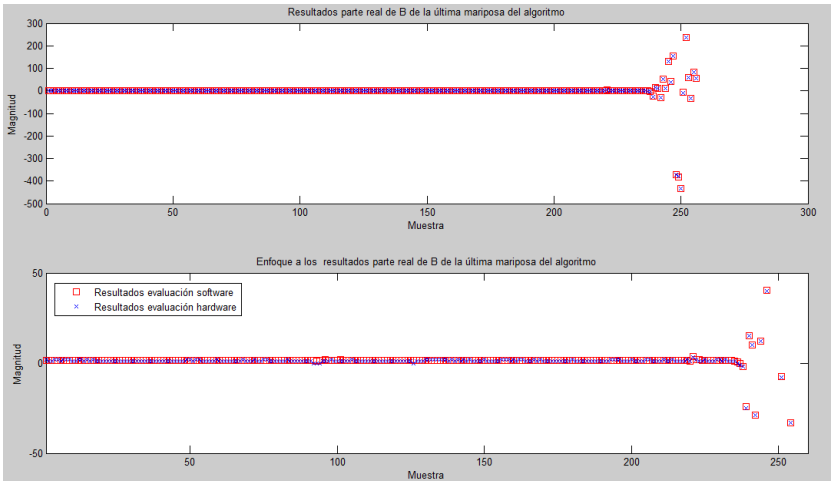


Fig. 5.23 Resultado parte real de B de la mariposa de la última etapa de evaluación de la señal de voz.

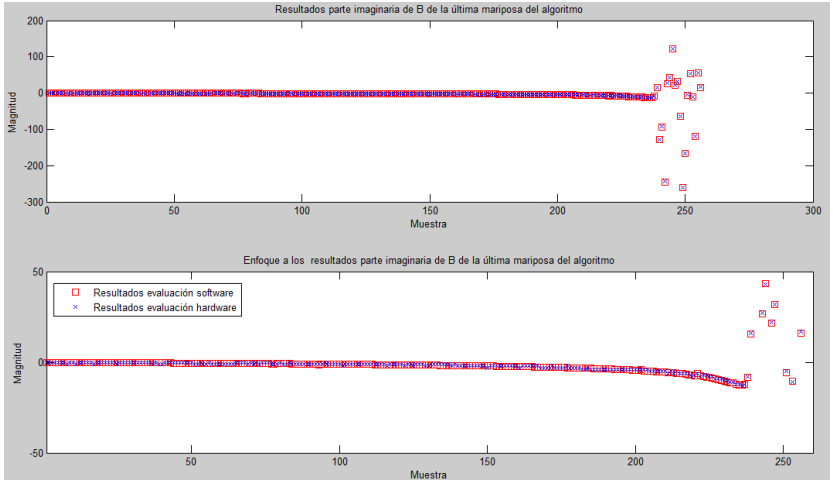


Fig. 5.24 Resultado parte imaginaria de B de la mariposa de la última etapa de evaluación de la señal de voz.

De igual manera que en las pruebas anteriores, la curva representada por los cuadros corresponde a los valores arrojados por la evaluación en C, mientras que la curva representada por las cruces corresponde a los resultados dados por la arquitectura propuesta. Las diferencias que existen entre ambas evaluaciones siguen siendo imperceptibles para las gráficas. En la Fig. 5.25 se muestran las gráficas correspondientes a la señal de entrada, el espectro de frecuencia obtenido por Matlab, el espectro de frecuencia obtenido por la evaluación en C y el espectro de frecuencia generado por la arquitectura en el FPGA.

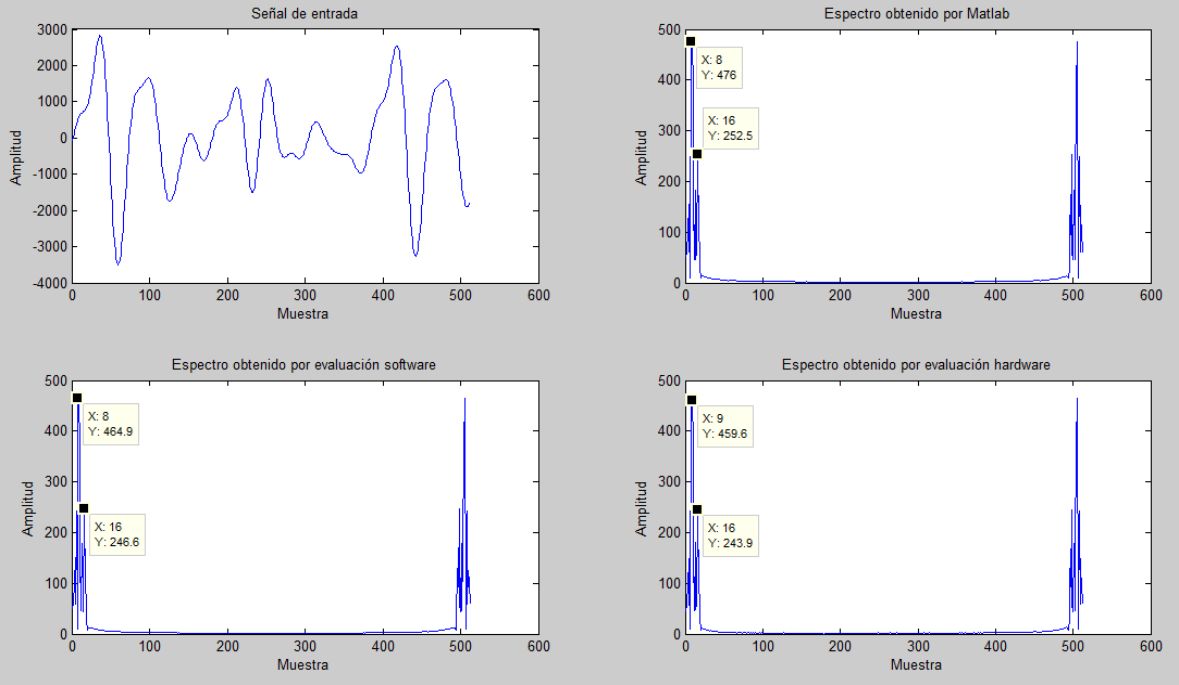


Fig. 5.25 Comparativa de los resultados para la señal de voz de entrada.

A simple vista observando el espectro completo de los tres resultados se puede notar que al menos en forma, los tres son muy similares, la Fig. 5.26 muestra un acercamiento y superposición de los resultados para observar con mayor detalle el espectro de frecuencias de la señal, las diferencias que existen entre estos resultados son básicamente de magnitud sin embargo se puede observar que los tres resultados muestran las componentes de frecuencia en las mismas muestras a lo largo del espectro.

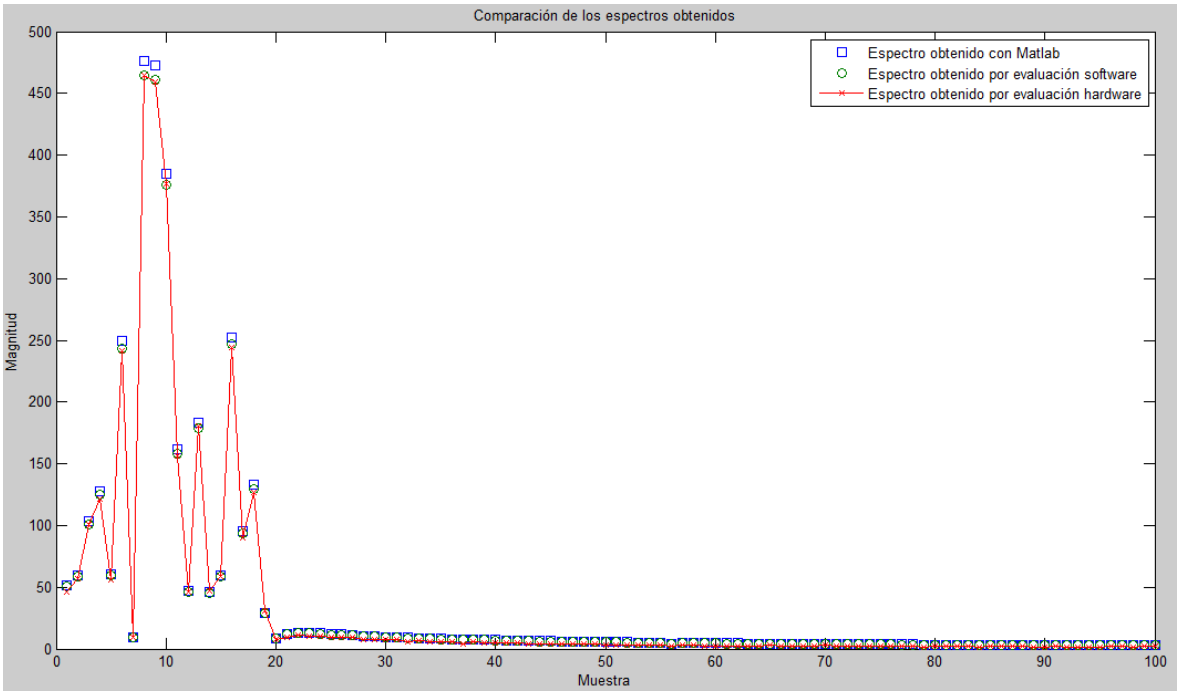


Fig. 5.26 Superposición de los espectros obtenidos para la señal de voz

En la gráfica anterior se puede observar que las diferencias entre los espectros mostrados son casi imperceptibles, la Fig. 5.27 muestra la evolución de las diferencias en la evaluación de cada etapa del algoritmo para esta señal de entrada en donde la diferencia más grande se presenta en la evaluación de la última etapa del algoritmo con un valor de 4.52 unidades.

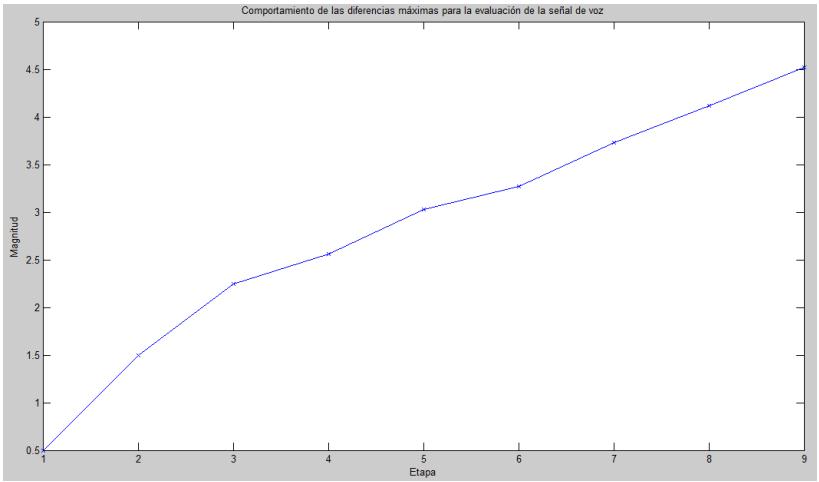


Fig. 5.27 Evolución de las diferencias para la evaluación software vs hardware con entrada de señal de voz, fonema A.

De manera general, en las cuatro pruebas realizadas se puede observar que el comportamiento de las diferencias entre cada etapa es creciente, ésta aumenta con cada nueva etapa que se evalúa en el algoritmo

y que no aumenta más allá de una unidad por etapa. El crecimiento más rápido de las diferencias se puede observar en las pruebas hechas para la señal cuadrada puesto que las diferencias comienzan en cero y llegan hasta 7.02 unidades en la última etapa siendo ésta la diferencia más grande para las cuatro señales de tal manera que podría inferirse que la magnitud de las diferencias crece a mayor cantidad de puntos sean evaluados, es decir, a mayor número de etapas en el algoritmo se tengan.

Cabe mencionar que la menor diferencia se obtuvo con la evaluación de la señal de voz, esto se debió a que esta señal contiene componentes positivas y negativas y que las señales anteriores de prueba (senoidal, cuadrada y diente de sierra) tiene valores únicamente positivos mayores a cero.

Es natural que estas diferencias existan entre la evaluación hardware y la evaluación software debido a que una está realizada con aritmética de punto flotante (evaluación software) y otra con aritmética de punto fijo (evaluación hardware) lo que genera una pérdida en la precisión aunado a los presentes truncamientos de los resultados de las multiplicaciones que existen en el componente de evaluación de mariposa, sin embargo, se puede observar que estas diferencias no son significantes pues en el peor de los casos equivale a 7.02 unidades de error absoluto en las comparaciones para estas señales de entrada, el cual es pequeño considerando que se está utilizando aritmética de punto fijo en donde además se hacen truncamientos en los resultados.

5.3 Prueba del funcionamiento de la arquitectura sobre la tarjeta de desarrollo *Altera-DE2*

Los resultados que se presentaron en el apartado anterior son producto de las simulaciones realizadas con la herramienta *ModelSim – Altera*, sin embargo, para corroborar el correcto funcionamiento de la arquitectura propuesta sobre el dispositivo se utilizó la herramienta *SignalTap II Logic Analyzer* que es un analizador de estados lógicos por software, esta herramienta permite configurar a la arquitectura de tal manera que los datos resultado de su funcionamiento pueden ser enviados a la computadora para ser observados en la interfaz de *SignalTap II* o con la interfaz de Matlab, en la Fig. 5.28 se muestra un diagrama de bloques de la configuración del diseño con esta herramienta.

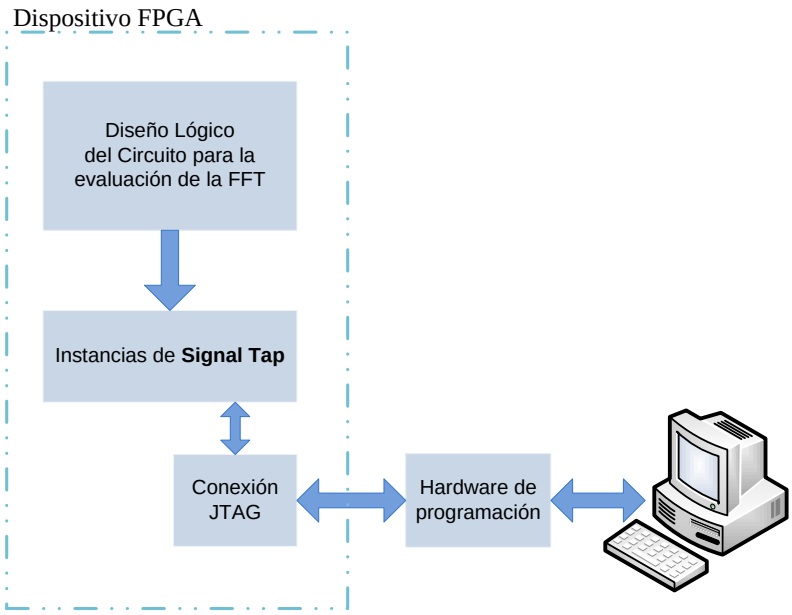


Fig. 5.28 Configuración de *SignalTap II* con la arquitectura propuesta.

Para observar el funcionamiento de la arquitectura se requirió de un componente que permitiera la entrada de los datos a procesar, este elemento consta de una memoria ROM de doble puerto y un contador para generar las direcciones de lectura de la memoria, en esta se almacenaron 512 muestras de una señal senoidal con frecuencia fundamental de 1KHz tomadas a una frecuencia de muestreo de 5KHz que fueron generadas con *MatLab 2011*. La Fig. 5.29 muestra la forma de onda resultante de este procedimiento.

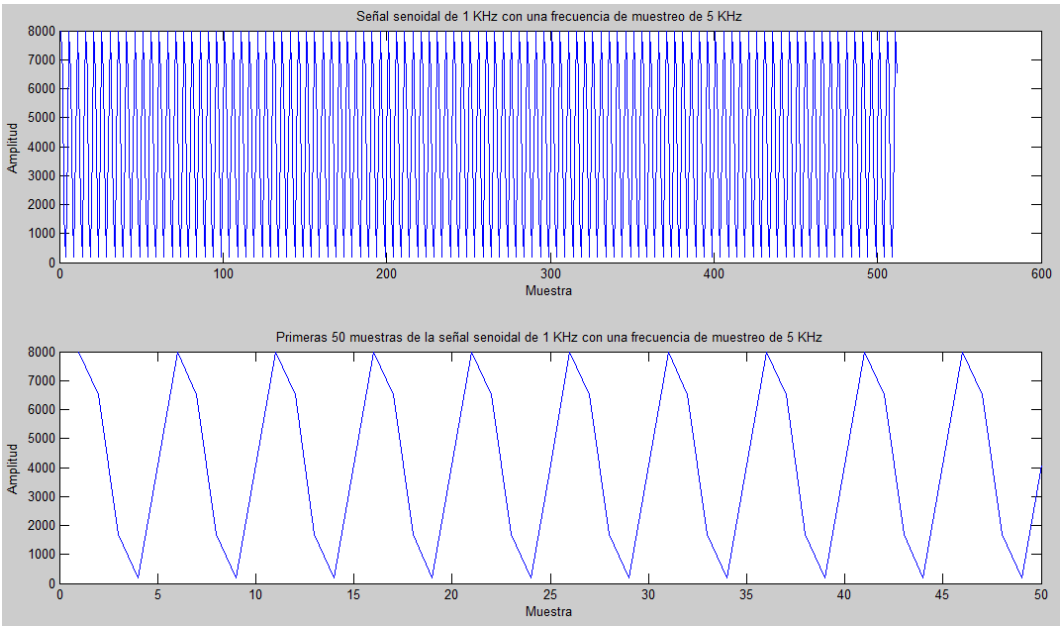


Fig. 5.29 Señal de entrada para la arquitectura propuesta.

La primera gráfica corresponde a las 512 muestras que se almacenaron en la memoria ROM, como se observa, la señal tiene una amplitud máxima cercana a 8192 y mínima a 0, lo que significa que la señal se encuentra montada sobre una señal de corriente directa y los valores de la señal son únicamente positivos.

Por cada flanco ascendente de reloj los datos son leídos y mandados como entradas a la arquitectura dos datos correspondientes a dos muestras de la señal generada, la lectura de los datos se realiza cada flanco de ascenso y únicamente se detiene cuando la señal *clr* esta activada. En la Fig. 5.30 se observa el diagrama RTL resultante de la síntesis del circuito para la generación de datos.

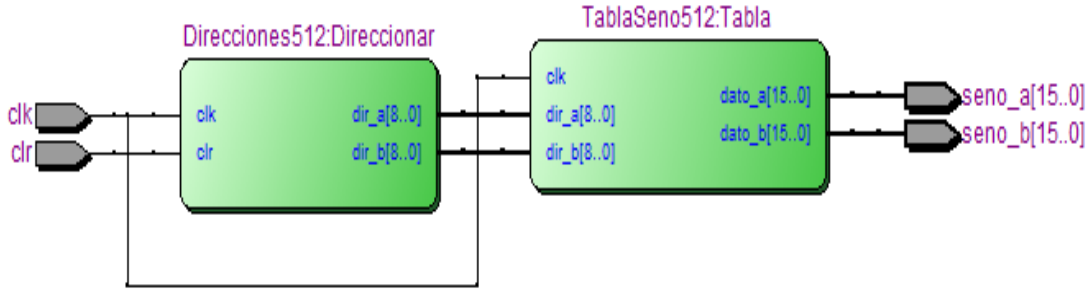


Fig. 5.30 Circuito para la generación de datos de entrada a la arquitectura

Este circuito se une a la arquitectura y como resultado se obtiene el circuito de la Fig. 5.31 en donde se observa el diagrama RTL de la aplicación completa.

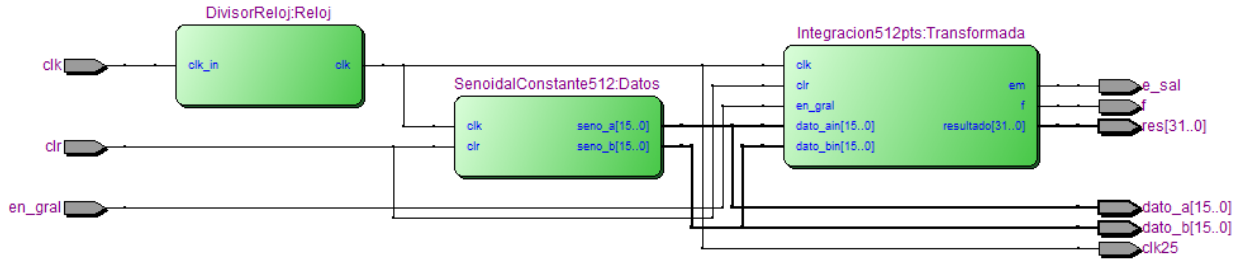


Fig. 5.31 Diagrama RTL de la aplicación final de prueba

El resumen de la síntesis del proyecto completo se puede observar en la Fig. 5.32.

Flow Summary	
Flow Status	Successful - Wed Jul 10 16:19:48 2013
Quartus II Version	11.0 Build 157 04/27/2011 SJ Web Edition
Revision Name	Datos_FFT_Fuentes
Top-level Entity Name	Datos_FFT_Fuentes
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	4,031 / 33,216 (12 %)
Total combinational functions	3,265 / 33,216 (10 %)
Dedicated logic registers	2,099 / 33,216 (6 %)
Total registers	2099
Total pins	70 / 475 (15 %)
Total virtual pins	0
Total memory bits	354,752 / 483,840 (73 %)
Embedded Multiplier 9-bit elements	64 / 70 (91 %)
Total PLLs	0 / 4 (0 %)

Fig. 5.32 Resumen de síntesis de aplicación.

Una vez que fueron asignadas las terminales de entrada y salida en la tarjeta se realizó la primera prueba de funcionamiento, dado que la frecuencia de reloj para el funcionamiento es de 25 MHz, es muy difícil observar algún cambio en el comportamiento de las salidas físicas en la tarjeta *Altera DE-2* por lo que se hizo uso de *SignalTap II Logic Analyzer*. El procedimiento para la configuración de *SignalTap II* fue realizado con base en el manual del software [29], y una vez que se ha puesto a funcionar, se puede observar una pantalla como la que se muestra en la Fig. 5.33 en donde se muestra como las señales monitoreadas son la salida de datos, la señal de reloj, la señal de habilitación, y el vector de resultados de salida. Para esta prueba, el archivo de configuración de *SignalTap II* permite la captura de 512 datos en cualquier instante y siempre sin importar lo que pase en las señales, esta configuración permite observar cómo se comporta la aplicación bajo diversos cambios en diferentes instantes de tiempo.

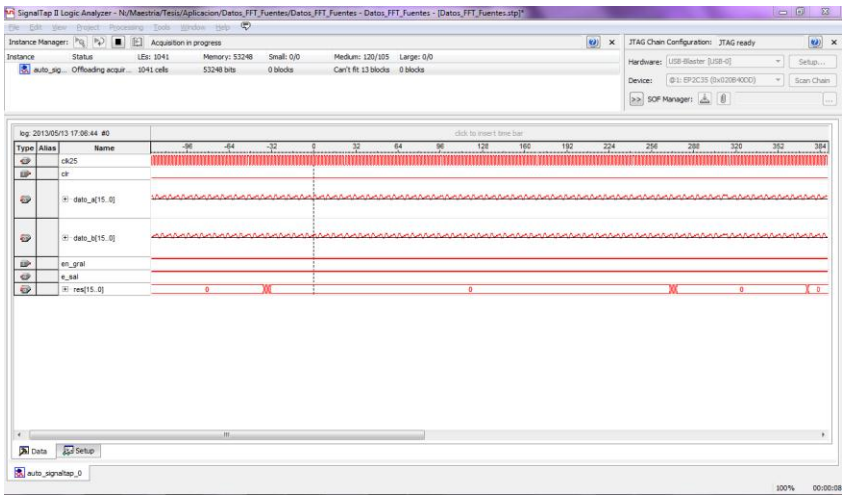


Fig. 5.33 Captura de pantalla del funcionamiento de *SignalTap II* y la aplicación de prueba.

Una vez que se corroboró que la arquitectura responde correctamente a la configuración del archivo generado por *SignalTap II* se hizo una conexión con *MatLab*, esto con la finalidad de observar de mejor manera el comportamiento de la aplicación, en la Fig. 5.34 se muestra el resultado final de dicha configuración.

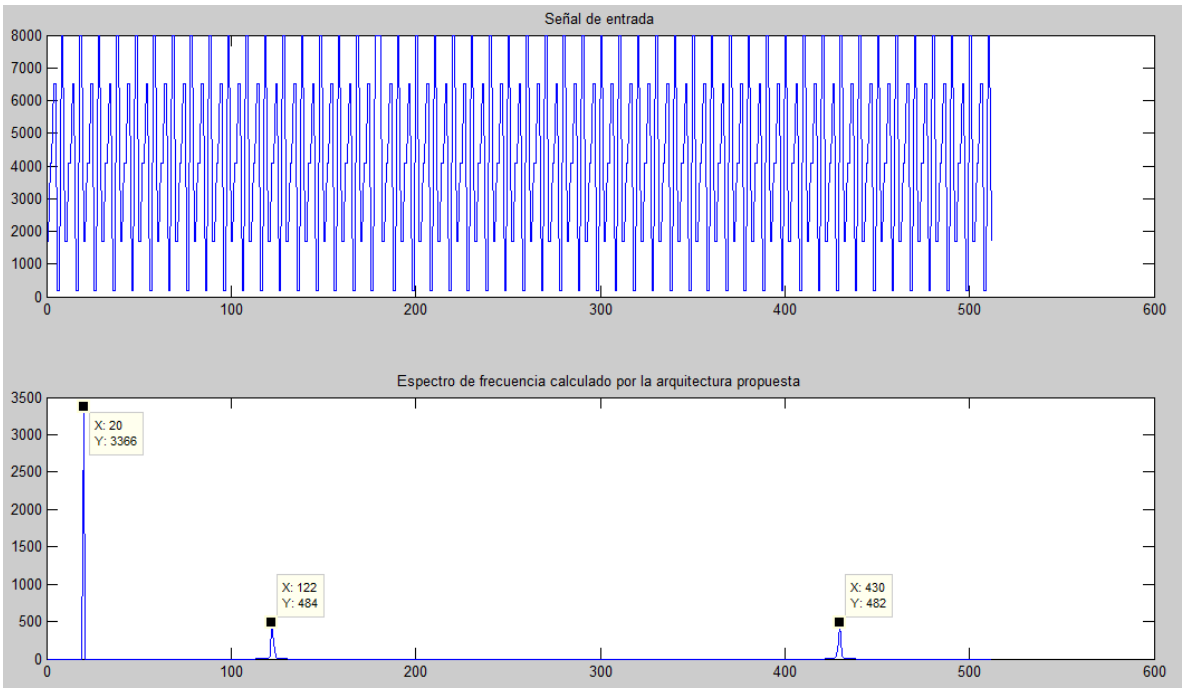


Fig. 5.34 Captura de pantalla del funcionamiento de *MatLab* y la aplicación de prueba.

En la primera grafica se puede observar la componente a de la señal de entrada, es por esto que se percibe distorsionada dado que la señal de entrada realmente es una composición de dos datos por cada ciclo de reloj. En la segunda gráfica se muestra el espectro de frecuencias obtenido por la arquitectura propuesta.

El espectro de frecuencias muestra la presencia de tres picos a lo largo de las 512 muestras, el primero que está etiquetado con (20,3366) corresponde a la componente de corriente directa sobre la cual se encuentra montada la señal senoidal mientras que las muestras etiquetadas con (122,484) y (430,482) corresponden a la componente de 1 KHz de la señal. Esto se puede saber recordando que tratándose de una FFT de 512 puntos, la muestra 512 del espectro de frecuencias resultante corresponde a la frecuencia de muestreo, en este caso, a 5 KHz y que la primera muestra es correspondiente a la componente directa de la señal.

Aplicando una regla de 3 se obtiene que para este caso, en la muestra 102 debería estar presente el pulso indicativo de la frecuencia fundamental de la señal de prueba.

Para este caso en específico, se tiene que el espectro de frecuencias esta defasado por 20 muestras, esto es por el instante en el que fue capturada la trama de datos desde la tarjeta, por lo que la muestra correspondiente a la frecuencia fundamental se debe encontrar 102 muestras adelante de la correspondiente a la componente de directa, esto se puede comprobar con el pico etiquetado por (122,3366) el cual cumple con lo requerido para representar la frecuencia fundamental de la señal de entrada.

Como parte de las pruebas funcionales de la arquitectura propuesta, se utilizó la señal de voz correspondiente al fonema *A* tal como se explicó anteriormente, esta señal fue también almacenada en el componente de generación de datos de entrada para corroborar el correcto funcionamiento de la arquitectura trabajando sobre la tarjeta de desarrollo, en la Fig. 5.35 se observa una captura de pantalla de los resultados obtenidos.

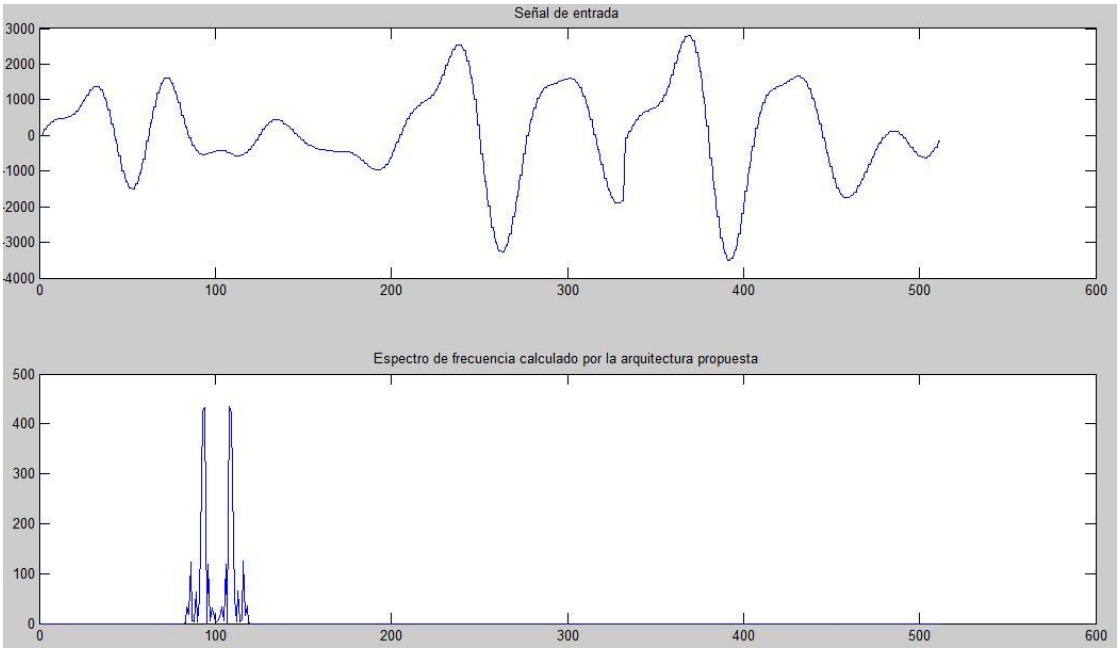
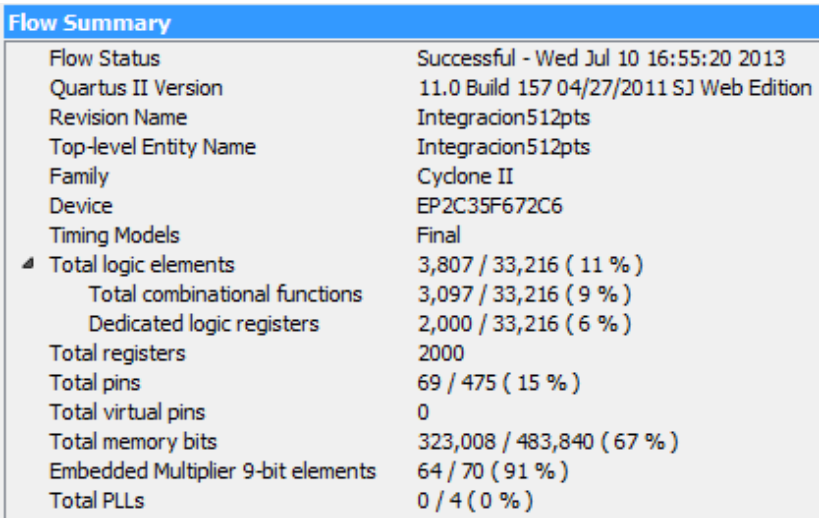


Fig. 5. 35 Captura de pantalla del funcionamiento de la aplicación de prueba para el fonema A

En la parte superior de la gráfica se observa la señal de entrada a la arquitectura y en la parte inferior se observa el espectro de frecuencias obtenido, ambos resultados muestran gran semejanza con los resultados mostrados en la Fig. 5.25 aunque el espectro de frecuencias se encuentra recorrido y comienza aproximadamente en la muestra 100, esto se debe al momento de la captura de la trama de datos por parte de *SignalTap II*.

5.4 Estadísticas temporales y comparación con el estado del arte

Una vez que se simuló funcionalmente la implementación de la arquitectura, se realizó la síntesis del proyecto completo para obtener los resultados de los recursos utilizados como elementos de memoria, multiplicadores dedicados y elementos lógicos y se obtuvo el resultado mostrado en la Fig. 5.36.



Flow Summary	
Flow Status	Successful - Wed Jul 10 16:55:20 2013
Quartus II Version	11.0 Build 157 04/27/2011 SJ Web Edition
Revision Name	Integracion512pts
Top-level Entity Name	Integracion512pts
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	3,807 / 33,216 (11 %)
Total combinational functions	3,097 / 33,216 (9 %)
Dedicated logic registers	2,000 / 33,216 (6 %)
Total registers	2000
Total pins	69 / 475 (15 %)
Total virtual pins	0
Total memory bits	323,008 / 483,840 (67 %)
Embedded Multiplier 9-bit elements	64 / 70 (91 %)
Total PLLs	0 / 4 (0 %)

Fig. 5.36 Resumen de síntesis de la arquitectura propuesta.

La arquitectura está compuesta de dos elementos fundamentales, la memoria de datos y el módulo de evaluación de la mariposa, esto se ve directamente reflejado en los recursos consumidos por la implementación en donde se utiliza el 91% de los multiplicadores dedicados debido a que sobre estos elementos son mapeados los multiplicadores que se utilizan para la evaluación de la mariposa y se utiliza el 67% de los recursos de memoria dedicada puesto que ahí es en donde son mapeadas las memorias de datos y de acuerdo con la estructura de la arquitectura se requieren de 19 bloques de memorias de 512 palabras cada uno, así como también son mapeadas las memorias en donde se almacenan los coeficientes complejos.

Así mismo, después de la síntesis y compilación del proyecto se obtuvo una frecuencia máxima de operación de la arquitectura de 49.41 MHz, éste es el primer parámetro de comparación contra las arquitecturas reportadas en el estado del arte. La arquitectura propuesta toma 2317 ciclos de reloj desde el momento en el que la señal *EN_GRAL* es activada hasta que el resultado de la evaluación de la mariposa de la última etapa se obtiene en condiciones iniciales, es decir, cuando la evaluación del algoritmo se realiza por primera vez, esto se traduce en 46.89 µs para la evaluación inicial del algoritmo, y 2573 ciclos de reloj con la obtención del módulo de los resultados igualmente en condiciones iniciales lo que se

traduce a 52.07 µs. Estos tiempos naturalmente se deben a que las unidades comienzan a ser activadas y el *pipeline* aún no se encuentra lleno de datos para mantener en funcionamiento constante a la arquitectura.

Una vez que todas las unidades comienzan a trabajar, la arquitectura tarda 256 ciclos de reloj que equivale a 5.18 µs, el tiempo por etapa se ve disminuido en una razón de dos, gracias a la configuración de las memorias que tienen doble puerto de lectura y escritura lo que permite que la evaluación de la mariposa se realice en un solo ciclo de reloj.

En la Tabla 5.2 se observa una breve comparativa de los resultados obtenidos de la arquitectura propuesta contra las arquitecturas que se tomaron para el estado del arte.

Tabla 5.2 Comparativa entre el estado del arte y la arquitectura propuesta.

Arquitectura	**Algoritmo implementado**	**Cantidad de puntos a evaluar**	**Frecuencia máxima de operación**	**Tiempo de ejecución**	**Dispositivo de Implementación**
High-Performance FFT Processing Using Reconfigurable Logic [12]	Radix-4	1024	100 MHz	10 µs	Virtex-II de Xilinx XC2V2000BF957 -5
Design a High Performance FFT Processor Based on FPGA [14]	Radix-4	1024	127 MHz	10.1 µs	Xilinx XCV2P30
Efficient FPGA implementation of FFT/IFFT Processor [17]	Radix-2^2	256	465 MHz	.135 µs	Virtex-5 de Xilinx XC5VSH35T
FPGA Implementation of a Re-configurable FFT for Multi-standard system in Software radio [19]	Radix-2	--	--	--	
Design and implementation of a Scalable Floating FFT-IP Core for Xilinx FPGAs [20]	Radix-2	128	115 MHz	1.11µs	Xilinx XC4VSX55-11
Design and Implementation of a Parallel Real-time FFT Processor [21]	Radix-4	16K	50MHz-	40 ns por mariposa	VirtexII Xilinx
Arquitectura propuesta	Radix-2	512	49.41 MHz	5.18 µs	Cyclone-2 Altera EP2C35F672C6

De la Tabla 5.2, las casillas que tienen dos guiones representan datos que no fueron publicados por lo que no se tiene acceso a ellos, se hizo la comparación con seis arquitecturas, es relevante mencionar que ninguna de estas arquitecturas fue evaluada bajo las mismas condiciones de trabajo, pero que se escogieron por considerarse referentes de las categorías en las que se encuentran. Estas observaciones complican el proceso para una comparación objetiva y concreta.

Para el caso de las arquitecturas [12] y [14] que se tratan de enfoques secuenciales, es decir, solamente utilizan una unidad de mariposa para la evaluación del algoritmo, donde se mitiga impacto en el tiempo de implementación utilizando el algoritmo R*adix-4* que por naturaleza es más rápido puesto que se utilizan cuatro operandos para la evaluación de la mariposa del algoritmo, ambas arquitecturas evalúan 1024 puntos (se requiere una potencia de 4 para el algoritmo), el doble que la arquitectura propuesta por lo que no se pueden comparar en primera instancia, sin embargo, partiendo del hecho de que la arquitectura propuesta tarda (*N/2*) ciclos de reloj a la frecuencia máxima reportada anteriormente, se tendría que la arquitectura evaluaría la misma cantidad de muestras en 10.36 µs casi igualando el tiempo reportado para [12] y [14]. De acuerdo con [1], el algoritmo *Radix-4* tiene una complejidad de $^{3N}/_{8}\, log_2 N$ de donde se puede inferir la información de la Tabla. 5.3.

Tabla 5.3 Comparación entre algoritmo Radix-2 vs Radix-4.

Cantidad de puntos (N)	**Radix-2** $^{N}/_{2}\, log_2 N$	**Radix-4** $^{3N}/_{8}\, log_2 N$	**Factor de mejora de velocidad**
4	4	3	1.333
16	32	24	1.333
64	192	144	1.333
256	1024	768	1.333
1024	5120	3840	1.333

Se puede notar que el aumento de la velocidad de cálculo del algoritmo *Radix-2* al algoritmo *Radix-4* es constante, siempre de 1.33 unidades. Al haber obtenido un tiempo de ejecución de 10.36µs las evidencias parecen indicar que la arquitectura propuesta tuvo una ganancia en la velocidad de evaluación de casi 1.333 veces respecto a la tendencia entre ambos algoritmos al haber obtenido casi el mismo tiempo en la evaluación con una diferencia de .25µs con respecto a [14].

Las arquitecturas [17] y [19] corresponden al enfoque de cascada, es decir, utilizan una mariposa por cada etapa del algoritmo, esto aumenta el nivel de paralelismo y no consume recursos innecesarios, el caso de comparación es la arquitectura propuesta en [17] en donde se evalúan 256 muestras en un tiempo de

0.135µs. Tomando en cuenta que se evalúa con el algoritmo *Radix-2*^{2} que se caracteriza por resolver el algoritmo *Radix-4* utilizando dos unidades de cálculo *Radix-2* se puede decir que este tiene también un factor de incremento de velocidad de 1.33 mencionado anteriormente. Partiendo del mismo principio de comparación para [12] y [14], la arquitectura propuesta tomaría 2.5µs en la evaluación de 256 muestras, es notorio que el tiempo de ejecución obtenido es más grande, sin embargo se estaría utilizando menos hardware para la evaluación del algoritmo con una diferencia en el tiempo de aproximadamente 2.4 unidades de tiempo (µs).

Finalmente, haciendo una comparación con la arquitectura propuesta en [20] en donde un caso de prueba fue la evaluación de 128 puntos en 1.11µs se puede recurrir nuevamente al recurso de la estimación en donde la arquitectura presentada tomaría un tiempo de 1.29µs bajo la frecuencia máxima de 49.41 MHz. Ambas arquitecturas utilizan más de una unidad de mariposa para realizar la evaluación del algoritmo, sin embargo lo hacen bajo diferentes enfoques, [20] lo hace bajo el enfoque paralelo-iterativo, las evidencias muestran que ambos enfoques evaluarían el algoritmo *Radix-2* en tiempos no muy distantes uno del otro.

5.4.1 Propuesta de medida de rendimiento

De acuerdo con [31], el rendimiento del CPU depende de tres características, el periodo de ciclo de reloj o frecuencia de operación, la cantidad de ciclos de reloj por instrucción y la cantidad de instrucciones por ejecutar, para este caso en específico, las instrucciones corresponden a la cantidad total de productos complejos que se requiere evaluar por cada arquitectura.

En este apartado se hace una propuesta de la cantidad de productos complejos evaluados por ciclo de reloj, lo que en [31] está definido como la cantidad de instrucciones por ciclo de reloj (IPC), esto hecho con base en el siguiente análisis.

En [31] se define al CPU_{time} como

$$CPU_{time} = \frac{CiclosCPU}{Fmax} \qquad \text{Ec. 5.1}$$

De donde se puede obtener que

$$CiclosCPU = FmaxCPU_{time} \qquad \text{Ec. 5.2}$$

Dónde:

CPU_{time}= Tiempo total de ejecución.

Fmax = Frecuencia de operación del reloj

CiclosCPU = Ciclos de reloj totales de ejecución

Con la expresión anterior se puede estimar el CPU_{time} que corresponde al tiempo total de ejecución de cierta cantidad de instrucciones dependiendo de la frecuencia de operación del reloj y de la cantidad total de ciclos de reloj, aunado a ello se puede contar el número de instrucciones ejecutadas las cuales se identifican como *IC,* con estos dos datos se puede obtener el promedio de ciclos de reloj por instrucción, de modo que

$$CPI = \frac{CiclosCPU}{IC} \qquad \text{Ec. 5.3}$$

De forma inversa, se pueden obtener la cantidad de instrucciones ejecutadas por ciclo de reloj puesto que

$$IPC = \frac{IC}{CiclosCPU} \qquad \text{Ec. 5.4}$$

Con las cuatro expresiones anteriores se puede generar una tabla en donde se obtengan como medidas de rendimiento el *IPC* y el *CPI* haciendo una analogía de los datos con los que se cuenta y los datos que se requieren, es decir, el *IC* que para un procesador está definido como el número total de instrucciones a ejecutar, para esta arquitectura en específico se convierte en el número total de productos complejos necesarios para evaluar el algoritmo dependiendo de la base.

El CPU_{time}, para este caso, corresponde al tiempo reportado por cada propuesta del estado del arte en el que evalúan *N* puntos con el algoritmo a la *Fmax* que es la frecuencia de operación del reloj, ambos datos fueron reportados en los resultados de cada una de las de las arquitecturas del estado del arte.

Por ejemplo, para el caso de la arquitectura propuesta, la frecuencia máxima reportada *Fmax* es de 49.41MHz, el tiempo total de evaluación CPU_{time} es de 5.18µs y el número de productos complejos *IC* es de 2304. Con esta información procederíamos a realizar el cálculo del número de operaciones por ciclo *IPC* y del número de ciclos de reloj por instrucción *CPI* de la siguiente forma

$$CiclosCPU = FmaxCPU_{time} = 49.41MHz(5.18\mu s) \cong 256$$

$$CPI = \frac{CiclosCPU}{IC} = \frac{256}{2304} = .111$$

$$IPC = \frac{IC}{CiclosCPU} = \frac{2304}{256} = 9$$

Con la arquitectura propuesta se realizan 9 productos complejos por cada ciclo de reloj, para las arquitecturas reportadas en el estado del arte los resultados se muestran en la tabla 5.4.

Con los resultados de los cálculos hechos para cada una de las arquitecturas se puede observar que la arquitectura propuesta, una vez que el *pipeline* está lleno puede evaluar 9 productos complejos por cada ciclo de reloj, este dato rebasa por mucho a las arquitecturas secuenciales [12] y [14] que evalúan 3.48 y 2.99 productos complejos respectivamente mientras que la propuesta de [17] que se trata de una arquitectura segmentada realiza 12.19 productos complejos por cada ciclo de reloj, estas tres arquitecturas evalúan el algoritmo *Radix-4* que tiene un factor de incremento en la velocidad de 1.33 veces.

En la arquitectura propuesta se evalúa un algoritmo Radix-2 por lo que si tomamos en cuenta el incremento de velocidad de 1.33 del algoritmo Radix-4, se tiene que la arquitectura propuesta evaluaría 11.97 cálculos por ciclo de reloj, dejando claro que se siguen evaluando más productos complejos que las arquitecturas secuenciales y que se encuentra muy cerca de la propuesta segmentada de [17] y superando el número de operaciones realizadas para 128 puntos con la propuesta de [20].

Tabla 5.4 Comparativa de los resultados con base en el número de operaciones por ciclo realizadas.

Arquitectura	Algoritmo implementado	Cantidad de puntos a evaluar	Total de Operaciones (IC)	Frecuencia máxima de operación (Fmax)	CPU_{time}	$CiclosCPU$	IPC	CPI
High-Performance FFT Processing Using Reconfigurable Logic [12]	Radix-4	1024	3840	100 MHz	10µs	1000	≅3.84	≅.2604
Design a High Performance FFT Processor Based on FPGA [14]	Radix-4	1024	3840	127 MHz	10.1µs	1282	≅2.99	≅.3338
Efficient FPGA implementation of FFT/IFFT Processor [17]	Radix-2^2	256	768	465 MHz	.135µs	63	≅12.19	≅.0820
FPGA Implementation of a Re-configurable FFT for Multi-standard system in Software radio [19]	Radix-2	--	--	--	--		--	--
Design and implementation of a Scalable Floating FFT-IP Core for Xilinx FPGAs [20]	Radix-2	128	448	115 MHz	1.11µs	127	≅ 3.52	≅.2834
Design and Implementation of a Parallel Real-time FFT Processor [21]	Radix-4	16K	86016	50MHz-	--	--	--	--
Arquitectura propuesta	Radix-2	512	2304	49.41 MHz	5.18µs	256	9	≅.1111

Capítulo 6. Conclusiones y trabajo a futuro

6.1 Conclusiones

Se realizó una propuesta de arquitectura para la evaluación de la transformada rápida de Fourier con base en lógica programable y habiendo hecho un estudio del estado del arte se concluyó utilizar un enfoque de ejecución hardware en cascada con la finalidad de aumentar el paralelismo en el cálculo de los productos complejos realizados, la reutilización de hardware para la evaluación del algoritmo y un esquema de memoria *non-inplace* con el propósito de eliminar todas las dependencias de datos y los tiempos muertos entre las iteraciones, pensando en obtener un circuito equilibrado en cuanto a recursos utilizados y tiempo de ejecución, el resultado de esta propuesta se ve plasmado en el diseño de la misma.

Se realizó el diseño de una arquitectura para calcular la transformada rápida de Fourier utilizando el método de diseño *top-down* con base en lógica programable, en este caso en específico, utilizando el FPGA EP2C35F672C6 de la familia Cyclone 2 con el que cuenta la tarjeta de desarrollo Altera DE2, todos los elementos para realizar operaciones aritméticas fueron diseñados partiendo del hecho de que se sintetizarían en elementos de hardware dedicado del FPGA para aumentar el aprovechamiento de los recursos de la tarjeta de desarrollo y así lograr una disminución en la cantidad de hardware utilizado. De igual manera, los elementos de memoria requeridos para sintetizar los bancos en cada etapa son elementos de memoria dedicados, los cuales se eligieron con el fin de no utilizar elementos lógicos y poder obtener memorias de doble puerto de lectura y escritura para no perder tiempo en el manejo de los datos, con estos dos aspectos considerados en el diseño se logró utilizar el 11% de elementos lógicos del dispositivo.

La simulación funcional de la arquitectura se realizó utilizando el método *buttom-up* es decir, de forma modular a manera de garantizar que los errores de diseño e integración fueran fácilmente localizados, las pruebas realizadas a cada uno de los bloques ya fueron descritas en el capítulo 4 en donde se comprobó que todos los elementos funcionan de acuerdo con las características tomadas en cuenta para el diseño obteniendo finalmente una arquitectura capaz de evaluar el algoritmo de la FFT.

Una vez que se probó el diseño realizado mediante simulaciones funcionales, se integró al proyecto un componente para la generación de los datos de entrada a la arquitectura y se sintetizó completamente configurando su funcionamiento para ser monitoreado mediante la utilización de *SignalTap II* y MatLab permitiendo así descargar el diseño en la tarjeta de desarrollo para observar el comportamiento de la arquitectura evaluando el algoritmo sobre el hardware.

Finalmente, se puede decir que todos los objetivos fueron cumplidos en tiempo y forma ya que se obtuvo una arquitectura capaz de evaluar el algoritmo de la transformada rápida de Fourier por medio de la utilización de una unidad de cálculo por cada etapa y un esquema de memoria que permite eliminar las dependencias de datos de la evaluación. La arquitectura obtenida reportó una frecuencia máxima de operación de 49.41 MHz lo que permite evaluar la FFT de 512 muestras en 5.16μs, con estos resultados se logra igualar a otras arquitecturas que implementan algoritmos teóricamente más rápidos y con un error absoluto máximo de 7.02 unidades lo que permite concluir que la arquitectura tiene un buen desempeño tanto en tiempo como en aritmética.

Aunque en primera instancia no se pudieron comparar los resultados de la arquitectura propuesta con las existentes, dado que ninguna de ellas fue probada bajo las mismas condiciones, con la medida de desempeño propuesta se comparó el número de operaciones por ciclo de cada arquitectura y se verificó que la arquitectura propuesta puede calcular 9 productos complejos por ciclo de reloj. Con este resultado se muestra una mejora del desempeño con respecto a las otras arquitecturas reportadas.

Cabe resaltar que la arquitectura propuesta cuenta con la posibilidad de ser reconfigurada (con la necesidad de hacer una compilación completa) a manera de evaluar más o menos puntos de manera sencilla sin hacer grandes cambios en la lógica de control puesto que la ruta de datos permanece igual. De manera general se puede concluir que todos los objetivos fueron alcanzados correcta y completamente.

6.2 Trabajo a futuro

Como se mencionó al principio de este trabajo, el contar con un componente dedicado expresamente a la evaluación de la transformada rápida de Fourier es de gran ayuda e importancia puesto que muchas de las aplicaciones del procesamiento digital de señales requieren del espectro de frecuencias de la señal para diversas tareas como filtrado y compresión de datos. Este trabajo ha aportado una arquitectura reconfigurable que realiza esta tarea sin compromiso en hardware ni en tiempo sin embargo aun se puede trabajar en la optimización de la arquitectura con el fin de aumentar la frecuencia de operación y reducir la cantidad de memoria requerida con actividades como:

- Implementación de un nuevo esquema de almacenamiento de los coeficientes complejos con la finalidad de reducir la cantidad de recursos utilizados al mínimo. Esto se puede lograr al utilizar los complementos de los coeficientes.

- Trabajar en el diseño de la arquitectura tomando en cuenta parámetros propios del hardware de implementación como redes de distribución de reloj (PLL) entre otros con el fin de maximizar la frecuencia de operación de la arquitectura y así reducir el tiempo de ejecución del algoritmo.
- Proponer un esquema de arquitectura para la evaluación del algoritmo utilizando aritmética de punto flotante agregando los respectivos módulos en la arquitectura.
- Realizar el diseño VLSI para la fabricación de la arquitectura

Referencias.

[1] Proakis, J. & Manolakis, D. *Digital Signal Processing Principles, Algorithms and Applications.* México: Prentice-Hall, 2005.

[2] Bachiesi, J. V. *Señales, Introduccion al Procesamiento Digital de Señales.* Valparaiso Chile: Ediciones Universitarias de Valparaiso, 2008.

[3] Pirsch, P. *Achitectures for Digital Signal Processing.* New York: Wiley, 1998.

[4] Mayer-Base, U. *Digital Signal Processing with Field Programmable Gate Arrays.* New York: Springer

[5] Zhao, Y., Erdogan, A. & Arslan, T. "A low-power and domain-specific reconfigurable FFT fabric for system-on-chip applications", in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium,* Abr.-2005.

[6] Shiqun Zhang; Dunshan Yu., "Design and implementation of a parallel real-time FFT processor", in *Proceedings of the 7th International Conference on Solid-State and Integrated Circuits Technology,* Oct.- 2004, pp. 1665-1668.

[7] Ahmed Saeed, M. E. "Efficient FPGA implementation of FFT/IFFT". *International Journal of Circuits, Systems and Digital Signal Processing*, 2009, pp. 103-110.

[8] Tessier, R. & Burleson, W. "Reconfigurable Computing for Digital Signal Processing: A Survey", *Journal of VLSI Signal Processing*, Vol. 28, No. 1, May.-Jun., 2007, pp. 7-27.

[9] Bergland, G. "Fast Fourier Transform Hardware Implementations-an Overview", *IEEE Transactions on audio and Electracoustics*, Vol. 17, No. 2, 1969, pp. 104-108.

[10] R. Klahn, R. R. Shively, E. Gomez & M. J. Gilmartin, "The time-saver: FFT hardware", *Electronics*, 1968, pp. 92-97.

[11] Baas, B., "A Low-Power, High-Performance 1024-Point FFT Processor", *IEEE Journal of Solid-State Circuits,* Vol.34, No. 3, Mar.-1999, pp. 380-387.

[12] Szedo, G.; Yang, V.; Dick, C., "High-Performance FFT Processing Using Reconfigurable Logic", in *Conference Record of the Thirty-Fifth Asilomar Conference on Signals, Systems and Computers*, Vol. 2, Nov.-2001, pp. 1353-1356.

[13] Zhao, Y.; Erdogan, A.; Arslan, T., "A low-power and domain-specific reconfigurable FFT fabric for system-on-chip applications", in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, Abr.-2005.

[14] Chao, C.; Qin, Z.; Yingke, X.; Chengde, H, "Design of a high performance FFT processor based on FPGA", in *Proceedings of the Design Automation Conference*, Vol. 2, Ene.-2005, pp. 920-923.

[15] He, S.; Torkelson, M., "Design and implementation of a 1024-point pipeline FFT processor", in *Proceedings of the Custom Integrated Circuits Conference*, May.-1998, pp. 131-134.

[16] Wu, G.; Liu, Y., "Radix-2^2 Based Low Power Reconfigurable FFT Processor", in *IEEE International Symposium on Industrial Electronics*, Jul.-2009, pp. 1134-1138.

[17] Saeed, A.; Elbably, M.; Eladawy, M., "Efficient FPGA implementation of FFT/IFFT Processor", *International Journal of circuits Systems and Signal Processing*, 2009, pp. 103-109.

[18] Rabiner, L.; Gold, B. "Theory and application of digital signal processing", New Jersey, 1975, Prentice-Hall.

[19] Ghouwayel, A.; Louët, Y., (2009), "FPGA Implementation of a Re-configurable FFT for Multi-Standard Systems in Software Radio Context", IEEE Transactions on Consumer Electronics, Vol. 55, No. 2, May.-2009, pp.950-958.

[20] Montaño, V., & Jiménez, M., (2010), "Design and implementation of a Scalable Floating-point FFT IP Core for Xilinx FPGA", in *53rd IEEE International Midwest Symposium on Circuits and Systems*, Seattle, WA, Ago.-2010, pp. 533-536.

[21] Zhang, S.; Yu, C., (2004), "Design and Implementation of a Parallel Real-time FFT Processor", in *Proceedings of the 7th International Conference on Solid-State and Integrated Circuits Technology*, Vol. 3, Oct.-2004, pp.1665-1668.

[22] Cooley, J.; Tukey, J "An Algorithm for the Machine Calculation of Complex Fourier Series", *IEEE Transactions on Electronic Computer*, Vol. EC-15, No. 4, 1966, pp. 680-681.

[23] Cooley, J., Lewis, P., Welch, P.,"Historical Notes on the Fast Fourier Transform", *IEEE Transactions on Audio and Electroacoustics*, Vol. 15, No. 2, Jun.-1967, pp. 75-79.

[24] Duhamel, P., "Implementation of Split-Radix FFT Algorithm", *Electron. Lett*, Vol. 20, 1986, pp. 14-16.

[25] F. Rodríguez-Henríquez, N. A.; Saqib, A.; Díaz-Pérez, C. K.; Koc. "Cryptographic Algorithms on Reconfigurable Hardware" *Signals and Communication Technology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[26] Vega Pérez J., *Método de diseño de sistemas fotovoltaicos: una propuesta*, M. S. Teshis, Proyecto de estudios sociales tecnológicos y científicos, IPN., 1998.

[27] Sjoholm, S. & Lindh, L., "VHDL for designers", USA, Prentice Hall, 1997.

[28] Navabi, Z., "VHDL analysis and modeling of digital systems", New York, Mc Graw Hill, 1998.

[29] Altera, "Introduction to the Quartus II Software Version 10.0" [en linea]. Disponible en: http://www.altera.com/literature/manual/archives/intro_to_quartus2.pdf

[30] Altera, "Cyclone II Device Family Data Sheet" [en linea] Disponible en: http://www.altera.com/literature/manual/archives/intro_to_quartus2.pdf

[31] Hennessy, J. & Patterson, D. "Computer Architecture a Quantitative Approach". USA: Morgan Kaufmann Publishers, 2007.

A. Programa en C para el cálculo de la FFT

Se presenta el código del programa en C utilizado para el cálculo de la FFT de manera iterativa y que sirvió para verificar el funcionamiento etapa por etapa de la arquitectura propuesta

```
#include <stdio.h>

int main(void)
{
   int NI = 512,E= 9, m=0;
   int n, e, i, grupo, mari, g_Wn, m_d, deng = 2, denm = NI;
   char texto[30], indice[30],c_seno[30], c_coseno[30];
   float bWn_R, bWn_I, auxr, auxi;
   float R[NI], I[NI],num, Coseno[NI/2], Seno[NI/2],coseno,seno;
   int TipoT = -1, p_array, f;
   FILE *fd, *fres, *fi, *fc,*fs, *fmodulo;
   int j;

//inicializar los arreglos de las muestras
   fd = fopen("N:\datos.txt","r");
   fi = fopen ("N:\indices.txt", "r");

   if (fd == NULL)
   {
      printf("Lo siento!\n");
   }
   else if (fi == NULL)
   {
      printf ("Archivo de indices no se pudo abrir");
   }
   else {
      for (j = 0; j <NI; j++)
      {
         fgets(texto,30,fd);
         sscanf(texto, "%f",&num);
         fgets (indice,30,fi);
         sscanf(indice,"%d", &p_array);
         R[p_array] = num;
         I[j] = 0;
         //printf ("array[%d] = %f\n",p_array,R[p_array]);
      }
   }
   fclose(fd);
   fclose (fi);

  //llenado de las tablas de coeficientes complejos
   fs = fopen("N:\senos.txt","r");
   fc = fopen ("N:\cosenos.txt", "r");
```

```
if (fc == NULL)
{
   printf("Lo siento, no se abrio COSENOS!\n");
}
else if (fs == NULL)
{
   printf ("Lo siento, no se abrio SENOS");
}
else {
   for (f=0; f<(NI/2);f++)
   {
      fgets(c_seno,30,fs);
      fgets(c_coseno,30,fc);
      sscanf(c_seno,"%f",&seno);
      sscanf(c_coseno,"%f",&coseno);
      Coseno[f] = coseno;
      Seno[f] = seno;
   }
}


   fres = fopen("resultados.txt","w");
   if (fres == NULL)
   {
      printf("No se abrio archivo para escribir resultados!");
   }
   else {

   for( e = 0; e < 9; e++ )        //etapa
   {
      fprintf(fres,"\t\tEtapa %d\n",e+1);
      fprintf(fres,"ra \t ma \t rb \t mb \t wr \t wm \t ar \t am \t br \t bm \n\n");
      printf ("Etapa %d\n", e);       //impresión del número de etapa
      i = 0;
      g_Wn = NI / deng;               //número de coeficientes
      for( grupo = 0; grupo < g_Wn; grupo++ )     //grupo
      {
        n = 0;
        m_d = NI / denm;
        for( mari = 0; mari < m_d; mari++ )      //mariposa
        {
          auxr  = R[ i ];
          auxi = I[ i ];
          bWn_R = R[ i+m_d ] * Coseno[ n ] + (TipoT * I[ i+m_d ] * Seno[ n ]);
          bWn_I = I[ i+m_d ] * Coseno[ n ] - (TipoT * R[ i+m_d ] * Seno[ n ]);
          printf("Ar = %.2f \t Br = %.2f \t Wr %d = %.5f\n", R[i], R[i+m_d], n, Coseno[n]);
          printf("Am = %.2f \t Bm = %.2f \t Wm %d = %.5f\n", I[i], I[i+m_d], n, Seno[n]);
```

```
                    //fprintf(fres,"%5.2f \t %5.2f\t %5.2f\t %5.2f\t %.5f\t %.5f\t ", R[i], I[i], R[i+m_d],
        I[i+m_d], Coseno[n], Seno[n]);
                    R[ i ]     = ((R[ i ] + bWn_R)/2);
                    I[ i ]     = ((I[ i ] + bWn_I)/2);
                    R[ i+m_d ] = ((auxr - bWn_R)/2);
                    I[ i+m_d ] = ((auxi - bWn_I)/2);
                    printf ("\t ar = %.3f \t\t am = %.3f\n", R[i], I[i]);
                    printf ("\t br = %.3f \t\t bm = %.3f\n\n", R[i+m_d], I[i+m_d]);
                    //fprintf(fres,"%5.2f \t %5.2f \t %5.2f \t %5.2f \n", R[i], I[i], R[i+m_d],I[i+m_d]);
                    fprintf(fres,"%5.2f \t %5.2f \n %5.2f \t %5.2f \n", R[i], I[i], R[i+m_d],I[i+m_d]);
                    i++;
                    n = n + g_Wn;
                  }
                  i = i + m_d;
                }
                deng = deng * 2;
                denm = denm / 2;
              }
}
```

B. Código fuente

- **Bloque de etapa de la arquitectura**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.diseno_512_pts.all;

entity ModuloC is
port (
        clk         : in std_logic;
        clr         : in std_logic;
        e_c         : in std_logic;
        m_d         : in std_logic_vector (8 downto 0);
        g_wn        : in std_logic_vector (8 downto 0);

        dir_awin : in std_logic_vector (8 downto 0);
        dir_bwin : in std_logic_vector (8 downto 0);
        dato_arw : in std_logic_vector (15 downto 0);
        dato_brw : in std_logic_vector (15 downto 0);
        dato_amw : in std_logic_vector (15 downto 0);
        dato_bmw : in std_logic_vector (15 downto 0);

        dir_awout: out std_logic_vector (8 downto 0);
        dir_bwout: out std_logic_vector (8 downto 0);
        res_a       : out std_logic_vector (15 downto 0);
        res_b       : out std_logic_vector (15 downto 0);
        res_am      : out std_logic_vector (15 downto 0);
        res_bm      : out std_logic_vector (15 downto 0)
);
end ModuloC;

architecture A_ModuloC of ModuloC is

signal seg              : std_logic;
signal swe, sweb : std_logic;
signal ssm              : std_logic;

signal address_ar   : std_logic_vector (8 downto 0);
signal address_br   : std_logic_vector (8 downto 0);
signal address_awout            : std_logic_vector (8 downto 0);
signal address_bwout            : std_logic_vector (8 downto 0);
signal address_aA  : std_logic_vector (8 downto 0);
signal address_bA  : std_logic_vector (8 downto 0);
signal address_aB  : std_logic_vector (8 downto 0);
signal address_bB  : std_logic_vector (8 downto 0);

signal s_coefr                  : std_logic_vector (15 downto 0);
signal s_coefm                  : std_logic_vector (15 downto 0);
signal data_aAr     : std_logic_vector (15 downto 0);
signal data_bAr     : std_logic_vector (15 downto 0);
signal data_aBr     : std_logic_vector (15 downto 0);
signal data_bBr     : std_logic_vector (15 downto 0);
signal data_ar                  : std_logic_vector (15 downto 0);
signal data_br                  : std_logic_vector (15 downto 0);

signal data_aAm    : std_logic_vector (15 downto 0);
signal data_bAm    : std_logic_vector (15 downto 0);
```

```
signal data_aBm    : std_logic_vector (15 downto 0);
signal data_bBm    : std_logic_vector (15 downto 0);
signal data_am             : std_logic_vector (15 downto 0);
signal data_bm             : std_logic_vector (15 downto 0);

signal ar,br,am,bm,cr,cm       : std_logic_vector (15 downto 0);

begin

--Unidad de control
Control : ControlLocal port map(
        clk     => clk,
        clr     => clr,
        e       => e_c,
        we      => swe,
        sm      => ssm,
        eg      => seg
);

sweb    <= not swe;
--Componente para generar direcciones
Direcciones : Gen_direccionesEX port map(
        clr     => clr,
        clk     => clk,
        e       => seg,
        m_d     => m_d,
        g_wn    => g_wn,
        dir_ra  => address_ar,
        dir_rb  => address_br,
        dir_wa  => address_awout,
        dir_wb  => address_bwout
);

--Componenetes para multiplexión de direcciones
Mux_dir_aA : Multiplexor_direcciones port map(
        dir_w   => dir_awin,
        dir_r   => address_ar,
        we      => swe,
        dir     => address_aA
);

Mux_dir_bA : Multiplexor_direcciones port map(
        dir_w   => dir_bwin,
        dir_r   => address_br,
        we      => swe,
        dir     => address_bA
);

Mux_dir_aB : Multiplexor_direcciones port map(
        dir_w   => dir_awin,
        dir_r   => address_ar,
        we      => sweb,
        dir     => address_aB
);

Mux_dir_bB : Multiplexor_direcciones port map(
        dir_w   => dir_bwin,
        dir_r   => address_br,
        we      => sweb,
        dir     => address_bB
);
```

```
--Banco de memorias (2 para reales y 2 para imaginarios)
MemoriaAr : Memoria_512_datos port map(
        clk             => clk,
        we              => swe,
        dato_ain  => dato_arw,
        dato_bin  => dato_brw,
        dir_a           => address_aA,
        dir_b           => address_bA,
        dato_aout       => data_aAr,
        dato_bout       => data_bAr
);

MemoriaBr : Memoria_512_datos port map(
        clk             => clk,
        we              => sweb,
        dato_ain  => dato_arw,
        dato_bin  => dato_brw,
        dir_a           => address_aB,
        dir_b           => address_bB,
        dato_aout       => data_aBr,
        dato_bout       => data_bBr
);

MemoriaAm : Memoria_512_datos port map(
        clk             => clk,
        dato_ain  => dato_amw,
        dato_bin  => dato_bmw,
        dir_a           => address_aA,
        dir_b           => address_bA,
        we              => swe,
        dato_aout       => data_aAm,
        dato_bout       => data_bAm
);

MemoriaBm : Memoria_512_datos port map(
        clk             => clk,
        dato_ain  => dato_amw,
        dato_bin  => dato_bmw,
        dir_a           => address_aB,
        dir_b           => address_bB,
        we              => sweb,
        dato_aout       => data_aBm,
        dato_bout       => data_bBm
);

--Generador de coeficientes
Coeficientes : CoeficienteEtapa2 port map(
        clk     => clk,
        clr     => clr,
        e       => seg,
        coefr   => s_coefr,
        coefm   => s_coefm
);

--Multiplexores de datos para mariposa
Mux_ar : Multiplexor_datos port map(
        e       => ssm,
        dato_a  => data_aAr,
        dato_b  => data_aBr,
        dato    => data_ar
);
```

```
Mux_br : Multiplexor_datos port map(
        e       => ssm,
        dato_a  => data_bAr,
        dato_b  => data_bBr,
        dato    => data_br
);

Mux_am : Multiplexor_datos port map(
        e       => ssm,
        dato_a  => data_aAm,
        dato_b  => data_aBm,
        dato    => data_am
);

Mux_bm : Multiplexor_datos port map(
        e       => ssm,
        dato_a  => data_bAm,
        dato_b  => data_bBm,
        dato    => data_bm
);

-----------------------------------------------------------------
Datos_Mariposa : process (clk, clr)
begin
        if (clr = '1') then
                ar      <= (others => '0');
                br      <= (others => '0');
                am      <= (others => '0');
                bm      <= (others => '0');
                cr <= (others => '0');
                cm <= (others => '0');
        elsif (clk'event and clk = '0') then
                if (seg = '1') then
                        ar      <= data_ar;
                        br      <= data_br;
                        am      <= data_am;
                        bm      <= data_bm;
                        cr      <= s_coefr;
                        cm      <= s_coefm;
                end if;
        end if;
end process;

-----------------------------------------------------------------

--Componente para el cálculo de la mariposa
Mariposa : Mariposa_op port map(
        Ra      => ar,
        Ma      => am,
        Rb      => br,
        Mb      => bm,
        Rw      => cr,
        Mw      => cm,
        Roa     => res_a,
        Moa     => res_am,
        Rob     => res_b,
        Mob     => res_bm
);

--Asignación de salidas
        dir_awout       <= address_awout;
        dir_bwout       <= address_bwout;
```

```
end A_ModuloC;
```

- **Bloque de memoria M4K**

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

entity Memoria_512_datos is
port (
        clk         : in std_logic;
        we          : in std_logic;
        dato_ain : in std_logic_vector (15 downto 0);
        dato_bin : in std_logic_vector (15 downto 0);
        dir_a       : in std_logic_vector (8 downto 0);
        dir_b       : in std_logic_vector (8 downto 0);
        dato_aout : out std_logic_vector (15 downto 0);
        dato_bout : out std_logic_vector (15 downto 0)
);
end Memoria_512_datos;

architecture A_Memoria_512_datos of Memoria_512_datos is
component Mem_512_datos IS
        PORT
        (
                address_a       : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
                address_b       : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
                clock           : IN STD_LOGIC  := '1';
                data_a          : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
                data_b          : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
                wren_a          : IN STD_LOGIC  := '0';
                wren_b          : IN STD_LOGIC  := '0';
                q_a             : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
                q_b             : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
        );
END component;

begin

        Memoria : Mem_512_datos port map
        (
                address_a       => dir_a,
                address_b       => dir_b,
                clock           => clk,
                data_a          => dato_ain,
                data_b          => dato_bin,
                wren_a          => we,
                wren_b          => we,
                q_a             => dato_aout,
                q_b             => dato_bout
        );

end A_Memoria_512_datos;
```

- **Bloque multiplexor de direcciones**

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

entity Multiplexor_direcciones is
```

```
port (
        dir_w     : in std_logic_vector (8 downto 0);
        dir_r     : in std_logic_vector (8 downto 0);
        we        : in std_logic;
        dir       : out std_logic_vector (8 downto 0)
);
end Multiplexor_direcciones;

architecture A_Multiplexor_direcciones of Multiplexor_direcciones is
begin

 dir <= dir_w when (we= '1') else dir_r;

end A_Multiplexor_direcciones;
```

- **Bloque multiplexor de datos**

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

entity Multiplexor_datos is
port (
        e         : in std_logic;
        dato_a    : in std_logic_vector (15 downto 0);
        dato_b    : in std_logic_vector (15 downto 0);
        dato      : out std_logic_vector (15 downto 0)
);

end Multiplexor_datos;

architecture A_Multiplexor_datos of Multiplexor_datos is
begin

dato <= dato_a when (e= '1') else dato_b;

end A_Multiplexor_datos;
```

- **Bloque de *bit-reverse***

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

entity Gen_direccionesE1 is
port (
        clk       : in std_logic;
        clr       : in std_logic;
        e         : in std_logic;
        dir_a     : out std_logic_vector (8 downto 0);
        dir_b     : out std_logic_vector (8 downto 0)
);
end Gen_direccionesE1;

architecture A_Gen_direccionesE1 of Gen_direccionesE1 is
signal a : std_logic_vector (8 downto 0);
signal b : std_logic_vector (8 downto 0);

signal en : std_logic;
begin

        Contar : process (clk, clr)
```

```
        variable a1         : std_logic_vector (8 downto 0);
        variable b1         : std_logic_vector (8 downto 0);
        begin
                if (clr = '1')then
                        a1 := "000000000";
                        b1 := "000000001";
                elsif (clk'event and clk = '1')then
                        if (e = '1') then
                                a1 := b1 + 1;
                                b1 := a1 + 1;
                        end if;
                end if;
                a <= a1;
                b <= b1;
        end process;


        dir_a(0)  <= a(8);          dir_b(0)  <= b(8);
        dir_a(1)  <= a(7);          dir_b(1)  <= b(7);
        dir_a(2)  <= a(6);          dir_b(2)  <= b(6);
        dir_a(3)  <= a(5);          dir_b(3)  <= b(5);
        dir_a(4)  <= a(4);          dir_b(4)  <= b(4);
        dir_a(5)  <= a(3);          dir_b(5)  <= b(3);
        dir_a(6)  <= a(2);          dir_b(6)  <= b(2);
        dir_a(7)  <= a(1);          dir_b(7)  <= b(1);
        dir_a(8)  <= a(0);          dir_b(8)  <= b(0);

end A_Gen_direccionesE1;
```

- **Bloque generador de direcciones**

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

entity Gen_direccionesEX is
port (
        clr             : in std_logic;
        clk             : in std_logic;
        e               : in std_logic;
        m_d             : in std_logic_vector (8 downto 0);
        g_wn            : in std_logic_vector (8 downto 0);
        dir_ra      : out std_logic_vector (8 downto 0);
        dir_rb      : out std_logic_vector (8 downto 0);
        dir_wa      : out std_logic_vector (8 downto 0);
        dir_wb      : out std_logic_vector (8 downto 0)
);
end Gen_direccionesEX;

architecture A_Gen_direccionesEX of Gen_direccionesEX is
signal dir_a, dir_b  : std_logic_vector(8 downto 0);
begin

        Generar   : process (clk, clr, m_d)
        Variable mar        : std_logic_vector(8 downto 0);
        variable i : std_logic_vector(8 downto 0);
        variable grupo : std_logic_vector (8 downto 0);
        begin
                if (clr = '1')then
                        i           := "000000000";
                        mar         := "000000000";
```

```
                grupo  := "000000000";
        elsif (clk'event and clk = '1')then
                if (e = '1') then
                        i       := i + '1';
                        mar := mar + '1';
                        if (mar >= m_d) then
                                mar := "000000000";
                                grupo := grupo + '1';
                                i := i + m_d;
                                if (grupo >= g_wn) then
                                        i       := "000000000";
                                        grupo := "000000000";
                                else
                                        i := i;
                                end if;
                        else
                                grupo := grupo;
                        end if;
                end if;
        end if;
        dir_a           <= i;
        dir_b           <= i + m_d;
end process;

dir_ra  <= dir_a;
dir_rb  <= dir_b;

Registrar : process (clk, clr)
begin
        if (clr = '1') then
                dir_wa  <= "000000000";
                dir_wb  <= "000000000";
        elsif(clk'event and clk='1')then
                if (e = '1') then
                dir_wa  <= dir_a;
                dir_wb  <= dir_b;
                end if;
        end if;
end process;
end A_Gen_direccionesEX;
```

- **Bloque generador de coeficientes complejos para la segunda etapa**

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

entity Coeficientesetapa1 is
port (
        clk     : in std_logic;
        clr     : in std_logic;
        e       : in std_logic;
        coefr   : out std_logic_vector (15 downto 0);
        coefm   : out std_logic_vector (15 downto 0)
);
end Coeficientesetapa1;

architecture A_Coeficientesetapa1 of coeficientesetapa1 is
signal dir_a        : std_logic_vector (0 downto 0);
component ROM1 IS
        PORT
```

```
	(
		address		: IN STD_LOGIC_VECTOR (0 DOWNTO 0);
		clock		: IN STD_LOGIC  := '1';
		q		: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
	);
END component;

component ROM1m IS
	PORT
	(
		address		: IN STD_LOGIC_VECTOR (0 DOWNTO 0);
		clock		: IN STD_LOGIC  := '1';
		q		: OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
	);
END component;

begin

	Generar : process (clk, clr, e)
	variable ddir : std_logic_vector (0 downto 0);
	begin
		if (clr = '1')then
			ddir	:= "0";
		elsif (clk'event and clk = '1')then
			if (e = '1')then
				ddir	:= ddir + '1';
			else
				ddir := ddir;
			end if;
		end if;
		dir_a <= ddir;
	end process;

Memoriar	: ROM1  port map
	(
		address => dir_a,
		clock	=> clk,
		q		=> coefr
	);

Memoriam	: ROM1m		port map
	(
		address => dir_a,
		clock	=> clk,
		q		=> coefm
	);

end A_Coeficientesetapa1 ;
```

- **Bloque de evaluación de la mariposa para la primera etapa**

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

entity Mariposa_op_etapa1 is
port (
	Ra	: in std_logic_vector (15 downto 0);
	Rb	: in std_logic_vector (15 downto 0);
	Roa	: out std_logic_vector (15 downto 0);
	Rob	: out std_logic_vector (15 downto 0)
```

```
);
end Mariposa_op_etapa1;

architecture A_Mariposa_op_etapa1 of Mariposa_op_etapa1 is
signal	Roaround		: std_logic_vector (15 downto 0);
signal	Robround		: std_logic_vector (15 downto 0);
signal	aux_a	: std_logic_vector (15 downto 0);
signal	aux_b	: std_logic_vector (15 downto 0);

begin
		aux_a	<= Ra + Rb;
		aux_b	<= Ra - Rb;

		Roaround (14 downto 0) <= aux_a (15 downto 1);
		Robround (14 downto 0) <= aux_b (15 downto 1);
		Roaround (15) <= aux_a(15);
		Robround (15) <= aux_b(15);

Roa <= Roaround;
Rob <= Robround;
end A_Mariposa_op_etapa1;
```

- **Bloque de evaluación de la mariposa para la segunda etapa**

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

entity Mariposa_op_etapa2 is
port
(
		Ra		: in std_logic_vector (15 downto 0);
		Rb		: in std_logic_vector (15 downto 0);
		Rw		: in std_logic_vector (15 downto 0);
		Mw		: in std_logic_vector (15 downto 0);

		Roa: out std_logic_vector (15 downto 0);
		Moa: out std_logic_vector (15 downto 0);
		Rob: out std_logic_vector (15 downto 0);
		Mob: out std_logic_vector (15 downto 0)
);
end Mariposa_op_etapa2;

architecture A_Mariposa_op_etapa2 of Mariposa_op_etapa2 is
signal RbRw, RbMw		: std_logic_vector (31 downto 0);
signal Rn, Mn		: std_logic_vector (15 downto 0);
signal Roaa, Moaa, Robb, Mobb		: std_logic_vector (15 downto 0);
signal aux_ar, aux_br, aux_am, aux_bm : std_logic_vector (15 downto 0);

begin

MultRbRw :  MULT port map
			(
			dataa	=> Rb,
			datab	=> Rw,
			result	=> RbRw
			);
MultRbMw :  MULT port map
			(
			dataa	=> Rb,
```

```
                datab     => Mw,
                result    => RbMw
                );

        Rn      <= RbRw (30 downto 15);
        Mn      <= RbMw (30 downto 15);

        aux_ar  <= Ra + Rn;
        aux_am  <= Mn;
        aux_br  <= Ra - Rn;
        aux_bm  <= "0000000000000000" - Mn;

        Roaa (14 downto 0) <= aux_ar(15 downto 1);
        Roaa (15) <= aux_ar(15);

        Moaa (14 downto 0) <= aux_am(15 downto 1);
        Moaa (15) <= aux_am(15);

        Robb (14 downto 0) <= aux_br(15 downto 1);
        Robb (15) <= aux_br(15);

        Mobb (14 downto 0) <= aux_bm(15 downto 1);
        Mobb (15) <= aux_bm(15);

--ASIGNACION DE SALIDAS
        Roa     <= Roaa;
        Rob     <= Robb;
        Moa     <= Moaa;
        Mob     <= Mobb;

end A_Mariposa_op_etapa2;
```

- **Bloque de evaluación de la mariposa genérico**

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

entity Mariposa_op is
port (
        Ra, Ma   : in      std_logic_vector(15 downto 0);
        Rb, Mb   : in      std_logic_vector(15 downto 0);
        Rw, Mw   : in      std_logic_vector(15 downto 0);
        Roa, Moa: out      std_logic_vector(15 downto 0);
        Rob, Mob :out      std_logic_vector(15 downto 0)
);
end Mariposa_op;

architecture A_Mariposa_op of Mariposa_op is

signal RbRw     :std_logic_vector (31 downto 0);
signal MbMw     :std_logic_vector (31 downto 0);
signal RbMw     :std_logic_vector (31 downto 0);
signal MbRw     :std_logic_vector (31 downto 0);

signal suma32   :std_logic_vector (31 downto 0);
signal resta32  :std_logic_vector (31 downto 0);

signal Mn       :std_logic_vector (15 downto 0);
signal Rn :std_logic_vector (15 downto 0);
```

```
signal Roaa, Robb  : std_logic_vector (15 downto 0);
signal Moaa, Mobb: std_logic_vector (15 downto 0);
signal aux_ar, aux_am, aux_br, aux_bm : std_logic_vector (15 downto 0);

begin

                Uno :  MULT port map
                (
                        dataa           => Rb,
                        datab           => Rw,
                        result      => RbRw
                );

                Dos :  MULT port map
                (
                        dataa           => Mb,
                        datab           => Mw,
                        result      => MbMw
                );

                Tres :  MULT port map
                (
                        dataa           => Rb,
                        datab           => Mw,
                        result      => RbMw
                );

                Cuatro :  MULT port map
                (
                        dataa           => Mb,
                        datab           => Rw,
                        result      => MbRw
                );


                        suma32   <= RbMw + MbRw;
                        resta32   <= RbRw - MbMw;

                        Rn        <= resta32(30 downto 15);
                        Mn        <= suma32(30 downto 15) ;

                        aux_ar    <= Ra + Rn;
                        aux_am   <= Ma + Mn;
                        aux_br    <= Ra - Rn;
                        aux_bm   <= Ma - Mn;

                        Roaa (14 downto 0) <= aux_ar(15 downto 1);
                        Roaa (15) <= aux_ar(15);

                        Moaa (14 downto 0) <= aux_am(15 downto 1);
                        Moaa (15) <= aux_am(15);

                        Robb (14 downto 0) <= aux_br(15 downto 1);
                        Robb (15) <= aux_br(15);

                        Mobb (14 downto 0) <= aux_bm(15 downto 1);
                        Mobb (15) <= aux_bm(15);

--ASIGNACION DE SALIDAS
                        Roa       <= Roaa;
                        Rob       <= Robb;
```

```
                    Moa     <= Moaa;
                    Mob     <= Mobb;

end A_Mariposa_op;
```

- **Bloque para el cálculo del módulo de resultados**

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

entity Modulo is
port (
        dato_r    : in std_logic_vector (15 downto 0);
        dato_m    : in std_logic_vector (15 downto 0);
        dato_mod: out std_logic_vector (31 downto 0)
);
end Modulo;

architecture A_Modulo of Modulo is
signal res_r : std_logic_vector (31 downto 0);
signal res_m : std_logic_vector (31 downto 0);
signal aux : std_logic_vector (31 downto 0);

begin

        cuadrado_r : MULT PORT map
        (
                dataa           => dato_r,
                datab           => dato_r,
                result    => res_r
        );

        cuadrado_m : MULT PORT map
        (
                dataa           => dato_m,
                datab           => dato_m,
                result    => res_m
        );

        aux <= res_r + res_m;

        dato_mod <= aux;
end A_Modulo;
```

- **Unidad de control por etapa**

```
library ieee;
use ieee.std_logic_1164.all;
use work.ControlLocalPack.all;

entity ControlLocal is
port (
        clk       : in std_logic;
        clr       : in std_logic;
        e         : in std_logic;
        we        : out std_logic;
        sm        : out std_logic;
        eg        : out std_logic
```

```
);
end ControlLocal;

architecture A_ControlLocal of ControlLocal is
signal se_we, se_sm             : std_logic;

begin

Manejador : ManejadorUnidad port map(
            clk         => clk,
            clr         => clr,
            ep          => e,
            e_we        => se_we,
            e_eg        => eg,
            e_sm        => se_sm
);

SenalWE : MaquinaWE port map (
            clk         => clk,
            clr         => clr,
            ep          => se_we,
            we          => we
);

SenalSM : MaquinaSM port map(
            clk         => clk,
            clr         => clr,
            ep          => se_sm,
            sm          => sm
);
end A_ControlLocal;
```

- **Bloque para el control de la señal *we***

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

entity MaquinaWE is
port (
            clk         : in std_logic;
            clr         : in std_logic;
            ep          : in std_logic;
            we          : out std_logic
);
end MaquinaWE;

architecture A_MaquinaWE of MaquinaWE is
type state_type is (A, B, C, D, E);
signal edo, edo_sig   : state_type;

signal en, ewe, flg : std_logic;

component ContadorWE is
port (
            clk         : in std_logic;
            clr         : in std_logic;
            Ewe         : in std_logic;
            WE          : out std_logic
);
end component;

component Contador_neg is
```

```
port (
        clk     : in std_logic;
        clr     : in std_logic;
        en      : in std_logic;
        c       : out std_logic
);
end component;

begin

Uno : Contador_neg port map (
        clk     => clk,
        clr     => clr,
        en      => en,
        c       => flg
        );

Dos : ContadorWE port map(
        clk     => clk,
        clr     => clr,
        Ewe     => ewe,
        WE      => we
);

Transicion : process (clk, clr)
begin
        if (clr = '1') then
                edo <= A;
        elsif (clk'event and clk = '1')then
                edo <= edo_sig;
        end if;
end process;

FSM     : process (edo, ep, flg)
begin
        --lwe <= '0';
        ewe <= '0';
        en      <= '0';

        case edo is
                when A =>
                        if (ep = '1') then
                                edo_sig <= B;
                        else
                                edo_sig <= A;
                        end if;
                when B =>
                        ewe <= '1';
                        en  <= '0';
                        --lwe <= '0';
                        if (ep = '1') then
                                edo_sig <= C;
                        else
                                edo_sig <= D;
                        end if;
                when C =>
                        ewe <= '0';
                        --lwe <= '0';
                        en  <= '1';
                        if (ep = '1') then
                                if (flg = '1') then
                                        edo_sig <= E;
```

```
                                        else
                                                edo_sig <= C;
                                        end if;
                                else
                                        if (flg = '1') then
                                                edo_sig <= B;
                                        else
                                                edo_sig <= D;
                                        end if;
                                end if;
                        when D =>
                                ewe <= '0';
                                en <= '0';
                                --lwe <= '0';
                                if (ep = '1') then
                                        edo_sig <= C;
                                else
                                        edo_sig <= D;
                                end if;
                        when E =>
                                ewe <= '1';
                                en <= '1';
                                --lwe <= '0';
                                if (ep = '1') then
                                        edo_sig <= C;
                                else
                                        edo_sig <= D;
                                end if;
        end case;
end process;
end A_MaquinaWE;
```

- **Contador_we**

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

entity ContadorWE is
port (
        clk     : in std_logic;
        clr     : in std_logic;
        Ewe     : in std_logic;
        WE      : out std_logic
);
end ContadorWE;

architecture A_ContadorWE of ContadorWE is
signal s_we     : std_logic;
begin
        Conteo : process (clk, clr)
        begin
                if (clr = '1') then
                        s_we <= '1';
                elsif (clk'event and clk = '0') then
                        if (Ewe = '1') then
                                s_we    <= not s_we;
                        end if;
                end if;
        end process;

        WE <= s_we;
```

```
end A_ContadorWE;
```

- **Contador_neg**

```
entity Contador_neg is
port (
        clk     : in std_logic;
        clr     : in std_logic;
        en      : in std_logic;
        c       : out std_logic
);
end Contador_neg;

architecture A_Contador_neg of Contador_neg is
signal var      : std_logic_vector(7 downto 0);
signal carry    : std_logic;
begin

Conteo_neg      : process (clr,clk)
begin
        if (clr = '1')then
                var <= "00000000";
        elsif (clk'event and clk = '0')then
                if (en = '1')then
                        var <= var + '1';
                end if;
        end if;
end process;

        carry <= var(0) and var(1) and var(2) and var(3) and var(4) and var(5) and var(6) and var(7);
        c               <= carry;
end A_Contador_neg;
```

- **Bloque para el control de la señal *sm***

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

entity MaquinaSM is
port (
        clk     : in std_logic;
        clr     : in std_logic;
        ep      : in std_logic;
        sm      : out std_logic
);
end MaquinaSM;

architecture A_MaquinaSM of MaquinaSM is
component ContadorSM is
port (
        clk     : in std_logic;
        clr     : in std_logic;
        esm     : in std_logic;
        sm      : out std_logic
);
end component;

component Contador is
```

```
port (
        clk     : in std_logic;
        clr     : in std_logic;
        en      : in std_logic;
        c       : out std_logic
);
end component;

type state_type is (A, B, C, D, E);
signal edo, edo_sig   : state_type;
signal en, f, sesm : std_logic;

begin

Uno : ContadorSM port map(
        clk     => clk,
        clr     => clr,
        esm     => sesm,
        sm      => sm
);

Dos : Contador port map(
        clk     => clk,
        clr     => clr,
        en      => en,
        c       => f
);

Trancision : process (clk,clr,edo_sig)
begin
        if (clr = '1') then
                edo <= A;
        elsif (clk'event and clk = '0') then
                edo <= edo_sig;
        end if;
end process;

FSM : process (edo, ep, f)
begin
        sesm <= '0';
        en   <= '0';
        case edo is
                when A =>
                        sesm  <= '0';
                        en                      <= '0';
                        if (ep = '1') then
                                edo_sig <= B;
                        else
                                edo_sig <= A;
                        end if;
                when B =>
                        sesm    <= '1';
                        en                      <= '0';
                        if (ep = '1') then
                                edo_sig <= C;
                        else
                                edo_sig <= D;
                        end if;
                when C =>
                        sesm    <= '0';
                        en      <= '1';
                        if (ep = '1') then
```

```
                                        if (f = '1') then
                                                edo_sig <= E;
                                        else
                                                edo_sig <= C;
                                        end if;
                                else
                                        if (f = '1') then
                                                edo_sig <= B;
                                        else
                                                edo_sig <= D;
                                        end if;
                                end if;
                        when D =>
                                sesm    <= '0';
                                en      <= '0';
                                if (ep = '1') then
                                        edo_sig <= C;
                                else
                                        edo_sig <= D;
                                end if;
                        when E =>
                                sesm <= '1';
                                en <= '1';
                                if (ep = '1') then
                                        edo_sig <= C;
                                else
                                        edo_sig <= D;
                                end if;
        end case;
end process;
end A_MaquinaSM;
```

- **Contador_sm**

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

entity ContadorSM is
port (
        clk     : in std_logic;
        clr     : in std_logic;
        esm     : in std_logic;
        sm      : out std_logic
);
end ContadorSM;

architecture A_ContadorSM of ContadorSM is
signal s_sm : std_logic;
begin

Cambio : process (clk,clr)
begin
        if (clr = '1') then
                s_sm    <= '0';
        elsif (clk'event and clk = '1') then
                if (esm = '1') then
                        s_sm <= not s_sm;
                end if;
```

```
		end if;
end process;
sm <= s_sm;
end A_ContadorSM;
```

- **Contador**

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

entity Contador is
port (
		clk		: in std_logic;
		clr		: in std_logic;
		en		: in std_logic;
		c		: out std_logic
);
end Contador;

architecture A_Contador of Contador is
signal var		: std_logic_vector (7 downto 0);
signal cc	: std_logic;
begin

		Contar	: process (clk, clr,en, var)
		begin
			if (clr = '1')then
				var		<= "00000000";
			elsif (clk'event and clk = '1')then
				if (en = '1')then
					var <= var + '1';
				end if;
			end if;
		end process Contar;

		cc		<= var(0) and var (1) and var(2) and var(3) and var(4) and var(5) and var(6) and var(7);
		c		<= cc;

end A_Contador;
```

- **Control para la señal *eg* y la habilitación de *we* y *sm***

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

entity ManejadorUnidad is
port (
		clk		: in std_logic;
		clr		: in std_logic;
		ep		: in std_logic;
		e_we		: out std_logic;
		e_eg		: out std_logic;
		e_sm		: out std_logic
		);
end ManejadorUnidad;

architecture A_ManejadorUnidad of ManejadorUnidad is
signal s_we : std_logic;
```

```
signal s_sm : std_logic;
begin
Activar_we          : process (clk,clr)
begin
        if (clr = '1') then
                s_we    <= '0';
        elsif (clk'event and clk = '0') then
                s_we    <= ep;
        end if;
end process;
e_we    <= s_we;

Activar_eg          : process (clk,clr)
begin
        if (clr = '1') then
                e_eg    <= '0';
        elsif (clk'event and clk = '0') then
                e_eg    <= s_we;
        end if;
end process;

Activar_sm          : process (clk,clr)
begin
        if (clr = '1') then
                s_sm    <= '0';
        elsif (clk'event and clk = '1') then
                s_sm    <= s_we;
        end if;
end process;
e_sm    <= s_sm;
end A_ManejadorUnidad;
```

- **Unidad de control general**

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

entity ControlGral is
port (
        clk         : in std_logic;
        clr         : in std_logic;
        e_gral: in std_logic;
        sal         : out std_logic_vector (10 downto 0)
);
end ControlGral;

architecture A_ControlGral of ControlGral is


component rotacion is
port (
        clk         : in std_logic;
        clr         : in std_logic;
        l, e        : in std_logic;
        entrada : in std_logic_vector (10 downto 0);
        salida      : out std_logic_vector (10 downto 0)
);
end component;
component ContadorGral is
port (
        clk         : in std_logic;
        clr         : in std_logic;
        e                   : in std_logic;
```

```
		b			: out std_logic
);
end component;

component ContadorGral6 is
port (
		clk		: in std_logic;
		clr		: in std_logic;
		e			: in std_logic;
		b			: out std_logic
);
end component;

signal cont4, cont5 : std_logic_vector (8 downto 0);
signal f4, f5, en4, en5, ef : std_logic;
signal ld, en : std_logic;
signal cargar : std_logic_vector (10 downto 0);
signal aux, mem : std_logic_vector (10 downto 0);


type state_type is (A, B, C, D, E, F, G, H);
signal edo, edo_sig   : state_type;


begin

Contador4 : ContadorGral port map(
		clk		=> clk,
		clr		=> clr,
		e			=> en4,
		b			=> f4
);

Contador5 : ContadorGral6 port map(
		clk		=> clk,
		clr		=> clr,
		e			=> en5,
		b			=> f5
);

Rotar : rotacion port map(
		clk		=> clk,
		clr		=> clr,
		l			=> ld,
		e			=> en,
		entrada => cargar,
		salida	=> aux
);

Transicion : process (clk, clr)
begin
		if (clr = '1') then
			edo <= A;
		elsif (clk'event and clk = '1')then
			edo <= edo_sig;
		end if;
end process;

Maquina : process (edo, f4, f5, e_gral, mem)
begin
		en4 <= '0';
		en5 <= '0';
```

```
ld <= '0';
en <= '0';
ef <= '0';
cargar <= (others => '0');
case edo is
	when A =>
		en4 <= '0';
		en5 <= '0';
		ld <= '0';
		en <= '0';
		ef <= '0';
		cargar <= (others => '0');
		if (e_gral = '1') then
			edo_sig <= B;
		else
			edo_sig <= A;
		end if;
	when B =>
		en4	<= '1';
		en5	<= '0';
		ld			<= '1';
		en			<= '0';
		ef <= '0';
		cargar(10 downto 1) <=(others => '0');
		cargar (0) <= '1';
		if (e_gral = '1') then
			if (f4 = '1') then
				edo_sig <= D;
			else
				edo_sig <= B;
			end if;
		else
			if (f4 = '1') then
				edo_sig <= D;
			else
				edo_sig <= C;
			end if;
		end if;
	when C =>
		en4	<= '0';
		en5	<= '0';
		ld			<= '1';
		en			<= '0';
		ef <= '0';
		cargar <= (others => '0');
		if (e_gral = '1') then
			edo_sig <= B;
		else
			edo_sig <= C;
		end if;
	when D =>
		en4	<= '0';
		en5	<= '1';
		ld			<= '0';
		en			<= '1';
		ef <='0';
		cargar <= (others => '0');
		if (e_gral = '1') then
			if (f5 = '1') then
				edo_sig <= D;
			else
				edo_sig <= E;
```

```
                end if;
        else
                if (f5 = '1') then
                        edo_sig <= D;
                else
                        edo_sig <= F;
                end if;
        end if;
when E =>
        en4     <= '0';
        en5     <= '1';
        ld              <= '0';
        en              <= '0';
        ef      <='0';
        cargar <= (others => '0');
        if (e_gral = '1') then
                if (f5 = '1') then
                        edo_sig <= D;
                else
                        edo_sig <= E;
                end if;
        else
                if (f5 = '1') then
                        edo_sig <= D;
                else
                        edo_sig <= F;
                end if;
        end if;
when F =>
        en4     <= '0';
        en5     <= '0';
        ld              <= '1';
        en              <= '0';
        ef      <= '1';
        cargar <= (others => '0');
        if (e_gral = '1') then
                edo_sig <= G;
        else
                edo_sig <= H;
        end if;
when G =>
        en4     <= '0';
        en5     <= '1';
        ld              <= '1';
        en              <= '0';
        ef      <= '0';
        cargar <= mem;
        if (e_gral = '1') then
                if (f5 = '1') then
                        edo_sig <= D;
                else
                        edo_sig <= E;
                end if;
        else
                if (f5 = '1') then
                        edo_sig <= D;
                else
                        edo_sig <= F;
                end if;
        end if;
when H  =>
        en4     <= '0';
```

```
                        en5     <= '0';
                        ld      <= '0';
                        en      <= '0';
                        ef      <= '0';
                                if (e_gral = '1') then
                                        edo_sig <= G;
                                else
                                        edo_sig <= H;
                                end if;
                end case;
end process;

registro : process (clk, clr)
begin
        if (clr     = '1') then
                mem <= (others => '0');
        elsif (clk'event and clk = '0') then
                if (ef = '1') then
                        mem <= aux;
                end if;
        end if;
end process;

sal <= aux;
end A_ControlGral;
```

- **Bloque de rotación**

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

entity rotacion is
port (
        clk         : in std_logic;
        clr         : in std_logic;
        l, e        : in std_logic;
        entrada : in std_logic_vector (10 downto 0);
        salida      : out std_logic_vector (10 downto 0)
);
end rotacion;

architecture a_rotacion of rotacion is
signal aux          : bit_vector (10 downto 0);
begin

        process (clk, clr)
        begin
                if (clr = '1') then
                        aux <= (others => '0');
                elsif (clk'event and clk = '0') then
                        if (l = '1') then
                                aux <= to_bitvector (entrada);
                        elsif (e = '1') then
                                aux <= aux sla 1;
                        end if;
                end if;
        end process;
        salida <= to_stdlogicvector (aux);

end a_rotacion;
```

- **Contador_N/2**

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

entity ContadorGral is
port (
	clk	: in std_logic;
	clr	: in std_logic;
	e		: in std_logic;
	b		: out std_logic
);
end ContadorGral;

architecture A_ContadorGral of ContadorGral is
signal conteo : std_logic_vector (8 downto 0);
begin

contador : process (clk,clr)
begin
	if (clr = '1')then
		conteo <= "000000000";
	elsif (clk'event and clk = '0')then
		if (e = '1')then
			if (conteo = "100000000") then
				conteo <= "000000000";
			else
				conteo <= conteo + '1';
			end if;
		end if;
end process;
b <= conteo(8) and not conteo(7) and not conteo(6) and not conteo(5) and not conteo(4) and not conteo(3) and not conteo(2) and
not conteo(1) and not conteo(0);
end A_ContadorGral;
```

- **Contador_N/2+1**

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

entity ContadorGral6 is
port (
	clk	: in std_logic;
	clr	: in std_logic;
	e		: in std_logic;
	b		: out std_logic
);
end ContadorGral6;

architecture A_ContadorGral6 of ContadorGral6 is
signal conteo : std_logic_vector (8 downto 0);
begin

contador : process (clk,clr)
begin
	if (clr = '1')then
		conteo <= "000000000";
```

```
        elsif (clk'event and clk = '0')then
                if (e = '1')then
                        if (conteo = "100000001") then
                                conteo <= "000000001";
                        else
                                conteo <= conteo + '1';
                        end if;
                end if;
        end if;
end process;
b <= conteo(8) and not conteo(7) and not conteo(6) and not conteo(5) and not conteo(4) and not conteo(3) and not conteo(2) and
not conteo(1) and conteo(0);
end A_ContadorGral6;
```