IDENTIFICACIÓN Y DISEÑO EN FPGA DE LAS FUNCIONES CRÍTICAS PARA LA EJECUCIÓN EN UNA WORKSTATION DEL ALGORITMO PARA EL CÁLCULO DE COTAS EN LA TRANSICIÓN DE FASE EN EL MODELO HARD-CORE EN 2D

JAIME ANDRÉS SALAZAR POSADA

UNIVERSIDAD DEL VALLE

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

PROGRAMA ACADEMICO DE INGENIERÍA ELECTRÓNICA

CALI, VALLE DEL CAUCA

2017

IDENTIFICACIÓN Y DISEÑO EN FPGA DE LAS FUNCIONES CRÍTICAS PARA LA EJECUCIÓN EN UNA WORKSTATION DEL ALGORITMO PARA EL CÁLCULO DE COTAS EN LA TRANSICIÓN DE FASE EN EL MODELO HARD-CORE EN 2D

JAIME ANDRÉS SALAZAR POSADA

Trabajo de Grado para el título de Ingeniero Electrónico

Director: Mario Enrique Vera Lizcano M.Sc.
Codirector: Juan Carlos Vera Lizcano Ph.D.

UNIVERSIDAD DEL VALLE
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA
PROGRAMA ACADEMICO DE INGENIERÍA ELECTRÓNICA
CALI, VALLE DEL CAUCA
2017

AGRADECIMIENTOS

El trabajo realizado en este proyecto representa la consecución de muchos años de estudio, amor a la ciencia y grandes esfuerzos de mi familia. Agradezco primeramente a Dios, como es debido, por darme todas las oportunidades que me ha dado en la vida para seguir adelante. Seguidamente a mis padres Jaime F. Salazar y Gloria I. Posada por tanto sacrificio, por todo lo que tengo y todo lo que soy, por el excelente ejemplo y la constante lucha para cuidar de la familia. A mi hermana Adriana M. Salazar por darme el ejemplo de la dedicación a la familia y amor al deporte y la salud. A mis sobrinitas Mariana y Luciana Molina por dar la alegría que mi familia tanto merece. A mi abuela Ofir Granada, a mis tíos Carmen Eugenia y Carlos Julio Posada por cuidar de mi familia materna. A mi tía Chila por su cariño constante. A mis tíos Alfonso, Gilberto y Fernando Salazar que siempre con su amor a la música, las artes y la fe cuidan de mi familia paterna. A mis numerosos primos en especial a Alejandra Perafán, Diana, Margarita y Carolina Osorio, Carlos A. y Andrea Posada, Adriana y Camila Salazar por mantenerse cerca de la familia.

Agradezco infinitamente a la Universidad del Valle, profesores, trabajadores y compañeros, en especial a mi director Mario E. Vera y su hermano Juan Carlos por tanta paciencia y apoyo para lograr el grado de pregrado. También al profesor Víctor H. Sánchez por ser el mejor profesor de la escuela y por un simple consejo que hizo de este grado posible. Al Prof. Jaime Velasco por ser un ejemplo para todos los amantes de la electrónica digital. A los Prof. Quintero, Martica (QEPD) y Marlés porque nunca había visto ese gran dominio de los fenómenos eléctricos. Al Prof. Bacca por esa gran paciencia y el intento de enseñarme a programar. A los Prof. Loaiza y Asfur por liderar en muy buena manera esta gran escuela. A mi compañero Duverney Corrales por ser el mejor estudiante e investigador y seguro el mejor profesor que vaya a tener la institución en la cual este, y en general a todas esas grandes personas que enseñan e investigan en este gran campo que es la electrónica.

Finalmente, a mis amigos Lorena Gualpa, Ricardo Orejuela, Jennifer Robbins, Karina Carvajal, Melody Fajardo, Diego Maya, Laura Barona, Daniela Vaca, Juliana Cobo, Juan M. Trujillo, Natalia López y todos los demás por haberme dado los mejores momentos.

TABLA DE CONTENIDO

1. INTRODUCCIÓN 1
   1.1 DEFINICIÓN DEL PROBLEMA 1
   1.2 JUSTIFICACIÓN 2
   1.3 OBJETIVO GENERAL 2
   1.4 OBJETIVOS ESPECÍFICOS 2
   1.5 RESULTADOS ESPERADOS 3
2. MARCO TEORICO 4
   2.1 CARACTERÍSTICAS DE LOS ALGORITMOS 4
   2.2 CLASIFICACIÓN DE LOS PROBLEMAS 6
   2.3 EL MODELO HARD-CORE 8
      2.3.1 Definición del modelo hard-core 9
      2.3.2 Cobertura de vértices 9
      2.3.3 Conjunto independiente 10
   2.4 TRANSICIONES DE FASE 10
   2.5 ARTÍCULO: IMPROVED BOUNDS OF THE PHASE TRANSITION FOR THE HARD-CORE MODEL IN 2D 11
   2.6 DESCRIPCIÓN DEL ALGORTIMO COMPUTACIONAL PARA EL CÁLCULO DE VALORES CRITICOS EN EL MODELO HARD CORE 13
   2.7 CODIFICACIÓN EN MATLAB® 16
3. METODOLOGíA 19
   3.1 CLASIFICACIÓN DE LA INVESTIGACIÓN 19
   3.2 METODOLOGÍA ASOCIADA CON EL ALGORITMO 19
   3.3 METODOLOGÍA DE DISEÑO TOP DOWN 20
   3.4 METODOLOGÍA DE DISEÑO BOTTOM UP 20
   3.5 METODOLOGÍA PARA EL DESARROLLO DE LA ARQUITECTURA HARDWARE 21
   3.6 ARQUITECTURA FPGA INTEL® ARRIA II GX-125EF35C4ES 21
   3.7 METODOLOGÍA DE DISEÑO CON FPGAs 25

4. RESULTADOS OBTENIDOS 27
4.1 FUNCIONES A IMPLEMENTAR EN LA FPGA 27
4.2 FORMATO DE DATOS 28
4.3 CARACTERÍSTICAS DE LAS LISTAS 29
4.4 PARÁMETROS DEL DISEÑO 31
4.5 ARQUITECTURA GENERAL DEL SISTEMA 34
4.6 FUNCIONAMIENTO DEL COMPONENTE UNIDAD_WSM_CRIT_MAR 36
4.7 MEMORIA 37
4.8 REGISTROS DE ALMACENAMIENTO 39
4.9 PROMEDIO 41
4.10 COMPARADOR 4 42
4.11 RELOJ DE TIEMPO REAL (*Real Time Clock - RTC*) 47
4.12 UNIDAD_WSM 47
4.13 FUNCIONAMIENTO DEL COMPONENTE UNIDAD_WSM 47
4.14 COMPARADOR 1 49
4.15 COMPARADOR 2 52
4.16 UNIDAD_FREC 55
4.17 FUNCIONAMIENTO GENERAL (UNIDAD_FREC) 55
4.18 UNIDAD_FILA 57
4.19 MULTIPLICADORES 60
4.20 DIVISOR 60
4.21 CONTADORES 67
4.22 UNIDADES DE CONTROL 69
4.23 PRUEBAS DE FUNCIONAMIENTO DEL SISTEMA TOTAL PARA LA LISTA L4 75
4.24 PRUEBAS DE FUNCIONAMIENTO - LISTA L6 81
4.25 RESULTADOS DE SÍNTESIS – LISTA L8 84
4.26 COMPARACION ENTRE LAS SÍNTESIS DE LAS LISTAS L4, L6 Y L8 85
4.27 OTRAS DISCUSIONES ACERCA DE LOS RESULTADOS OBTENIDOS 90

5. CONCLUSIONES.....92
6. TRABAJOS FUTUROS 94
7. BIBLIOGRAFíA 95
8. ANEXOS.....96

LISTA DE TABLAS

Tabla 1: Lista L4 15
Tabla 2: Número y tipo de bloques DSP en la Arria II GX-125 24
Tabla 3: Ejemplos del formato punto fijo sin signo de 64 bits 29
Tabla 4: Ejemplo de listas 4a) Lista L4 4b) Lista L6..... 30
Tabla 5: Parámetros del diseño 31
Tabla 6: Comparación de divisores de 64 bits..... 67
Tabla 7: Comparación de divisores de 128 bits..... 67
Tabla 8: Características de los contadores utilizados. 68
Tabla 9: Ejecución de la Lista L4 en Matlab® - 4 cifras significativas... 78
Tabla 10: Ejecución de la Lista L4 en Matlab® - 8 cifras significativas . 79
Tabla 11: Recursos Totales – Lista L4 80
Tabla 12: Recursos por componentes – Lista L4 80
Tabla 13: Ejecución de la Lista L6 en Matlab® - 4 cifras significativas . 82
Tabla 14: Ejecución de la Lista L6 en Matlab® - 8 cifras significativas . 82
Tabla 15: Recursos Totales – Lista L6 83
Tabla 16: Recursos por componentes – Lista L6 83
Tabla 17: Recursos Totales – Lista L8 84
Tabla 18: Recursos por componentes – Lista L8 85
Tabla 19: Comparación entre listas L4, L6 y L8 86
Tabla 20: Errores y Diferencias de tiempo de ejecución. 87

LISTA DE FIGURAS

Figura 1: Diagrama de Euler de los tipos de problemas. 7
Figura 2: 2a) Cobertura de vértices. 2b) Cobertura de vértices mínima. 2c) Grafo sin cobertura de vértices. 10
Figura 3: Diagrama del flujo de datos del algoritmo 18
Figura 4: Adaptative Logic Module - ALM 22
Figura 5: Logic Array Blocks – LAB 23
Figura 6 : Configuración del DSP en modo multiplicador sumador acumulador 24
Figura 7: Flujo del diseño. 26
Figura 9: Configuración de memorias para realizar *x(L)* 32
Figura 10: Esquema general del sistema 35
Figura 11: Diagrama de componentes de UNIDAD_WSM_CRIT_MAR ... 35
Figura 12: Diagrama de componentes de UNIDAD_WSM_CRIT_MAR (detalle) 36
Figura 13: Memoria ROM. En este caso la lista L4 38
Figura 14: a) Lista L4. b) Archivo de inicialización con la lista L4. 38
Figura 15: Registros n x 64 bits (n=4=Lista L4) 40
Figura 16: Registro entrada serie – salida paralela.....41
Figura 17: Hardware de la unidad Promedio.....42
Figura 18: Hardware del Comparador 4 45
Figura 19: Prueba de funcionamiento Comparador 4 46
Figura 20: Diagrama de componentes de UNIDAD_WSM 47
Figura 21: Diagrama de componentes de UNIDAD_WSM (detalle).....48
Figura 22: Hardware del comparador 1 50
Figura 23: Prueba de funcionamiento del comparador 1 51
Figura 24: Hardware del comparador 2 53
Figura 25: Prueba de funcionamiento del comparador 2 53
Figura 26: Diagrama de componentes de UNIDAD_FREC 56
Figura 27: Diagrama de componentes de UNIDAD_FREC (detalle) 56
Figura 28. Hardware de la unidad aritmética básica UNIDAD_FILA. 58
Figura 29: Pruebas unidad aritmética básica 59
Figura 30: Arquitectura RTA del subcomponente STAGE. 66
Figura 31: Arquitectura RTB del subcomponente STAGE 66
Figura 32: Arquitectura RTC del subcomponente STAGE 66
Figura 33: Hardware de Contador ITER 68
Figura 34: Diagrama de estados de la FSM 1 71
Figura 35: Diagrama de estados de la FSM 2 73

Figura 36: Diagrama de estados de la FSM 3. 74
Figura 37: Prueba de funcionamiento – Lista L4 – Estados s0, s1, s2 .. 75
Figura 38: Prueba de funcionamiento – Lista L4 – Estados s3, s4 76
Figura 39: Prueba de funcionamiento – Lista L4 – Estado s5 77
Figura 40: Prueba de funcionamiento – Lista L4 – Estado s8 – 4 cifras significativas 78
Figura 41: Prueba de funcionamiento – Lista L4 – 8 cifras significativas 79
Figura 42: Prueba de funcionamiento – Lista L6 – 4 cifras significativas 81
Figura 43: Prueba de funcionamiento – Lista L6 – 8 cifras significativas 82
Figura 44: Crecimiento de n. 88
Figura 45: Línea de tendencia hacia el valor deseado 3.796 con $prec = 4$ 88
Figura 46: Tiempos de ejecución en PC Intel i7 con $prec = 4$ 89
Figura 47: Línea de tendencia hacia el valor deseado 3.796 con $prec = 8$ 89
Figura 48: Tiempos de ejecución en PC Intel i7 con $prec = 8$ 90

RESUMEN

El modelo hard-core utilizado en la física estadística para determinar las transiciones de estados de un material gaseoso sirve de base para la elaboración de un algoritmo definido por el Prof. J. C. Vera Ph. D., et al. en [1] que intenta mejorar otros enfoques para el cálculo de las cotas de la actividad critica (fugacidad del gas) para partículas de gas modeladas bajo el modelo hard-core. En [1] se demuestra matemáticamente que la actividad critica está en el rango $2.8 < \lambda > 3.4$. El algoritmo planteado esta codificado en lenguaje Matlab® y trabaja satisfactoriamente sin embargo el consumo en tiempo de ejecución crece significativamente cuando se debe procesar un gran número de elementos. En este trabajo de grado se implementa este algoritmo en su totalidad dentro de una FPGA de la familia ARRIA II GX de Intel® con el fin de aumentar el grado de paralelismo del hardware que procesará los datos las cuáles son unas matrices de índices.

Se diseñó una arquitectura general y otras variaciones (en reducida medida) de esta arquitectura para definir aspectos claves en el paralelismo y en el control que debe tener este hardware y funciones críticas del algoritmo, que generan cuellos de botella para llegar a un diseño que permita una aceleración de la ejecución del mismo. Al final, se lograron procesar pocas matrices, sin embargo, se identificaron plenamente los errores de diseño y las funciones más críticas.

PALABRAS CLAVES

Divisor hardware, FPGA, hard-core, punto fijo, síntesis hardware, VHDL.

ABSTRACT

The hard-cored model that is used in statistical physics to establish the state transitions of a gas is base to the elaboration of an algorithm created by Prof. J.C. Vera Ph.D. et al. in [1] that intends to enhance another approaches of calculating the bounds of a critical activity (fugacity of a gas) for gas particles modelled under the hard-core model. In [1] it shown mathematically that the critical activity is in $2.8 < \lambda > 3.4$. The proposed algorithm is codified in Matlab® and works satisfactorily, however, the consumption at runtime grows significantly when more and more elements are to be processed. In this work, this algorithm is implemented in its entirety within an FPGA of the ARRIA II GX family of Intel® in order to increase the degree of parallelism of the hardware that will process the data which are index matrices.

A general architecture and other variations were designed (in a small measure) of this architecture to define key aspects in the parallelism and the control that must have this hardware and critical functions of the algorithm, that generate bottlenecks to arrive at a design that allows an acceleration of the execution. In the end, few matrices were processed but the design errors and the most critical functions were fully identified.

KEY WORDS

Divide Hardware, FPGA, hard-core, fixed point. Hardware synthesis, VHDL.

1. INTRODUCCIÓN

1.1 DEFINICIÓN DEL PROBLEMA

Dentro del campo de la física estadística, las cotas en la transición de fase de estado de un gas se establecen con un modelo matemático llamado el modelo hard-core en Dos Dimensiones (2D) en el cual las partículas del gas se organizan formando una retícula. Para calcular estas cotas se ejecuta un algoritmo planteado por Vera et al. basado en [1], el cual verifica si una recurrencia de una función en particular converge, con sus argumentos en un determinado dominio. Lo anterior es equivalente a calcular la cota de transición de fase.

En [1] se ilustra una propuesta para optimizar el cálculo de las cotas utilizando el modelo hard-core, donde se explica matemáticamente la relación entre el cálculo de las cotas en las transiciones de fase y la determinación de la existencia de la unicidad de medidas de Gibbs usando el modelo hard-core. Weitz en [2] presenta un enfoque matemático para determinar las cotas en las transiciones de fase, Restrepo en [3] mejoró el enfoque de Weitz. Se establece en [1] que el enfoque de Weitz no es útil cuando el límite de la cota superior es mayor a 3.4. También en [1] se refina el enfoque de Restrepo mejorando el valor del límite inferior de la cota de 2.48. Existe un valor critico relacionado con el cálculo de las cotas, heurísticamente se conoce que el valor critico es 3.796.

El algoritmo para optimizar el cálculo de las cotas, basado en las demostraciones en [1], procesa un conjunto de listas compuestas por números naturales y calcula el valor crítico asociado a cada una de ellas. El algoritmo está codificado en lenguaje MATLAB®, presenta la restricción de procesar una lista por ejecución, y su complejidad computacional depende de manera significativa del valor del número de filas de la lista (n).

El algoritmo es una búsqueda binaria, el cual al realizar un mayor número de iteraciones permite obtener un resultado más preciso para una lista con un número de filas dado. Por lo tanto, entre más grande sea el número de filas, se incrementa el número de las iteraciones, obteniéndose un aumento exponencial en el tiempo de procesamiento del algoritmo gracias a la complejidad computacional antes mencionada.

Se necesita entonces una arquitectura hardware donde se ejecute el algoritmo para el cálculo de las cotas, con el fin de disminuir el tiempo de procesamiento. Por lo tanto, en este trabajo de grado se identifican y se diseñan en una FPGA el conjunto de funciones matemáticas que son críticas en la ejecución computacional del algoritmo codificado en MATLAB® y basado en [1].

1.2 JUSTIFICACIÓN

El constante avance en las arquitecturas de las FPGAs de los bloques DSP, bancos de memoria, puertos I/O, consumo de potencia y bloques de elementos lógicos, permiten a los diseñadores implementar sistemas complejos, particularmente funciones especializadas para acelerar algoritmos computacionales y así procesar señales digitales más rápidamente en comparación con un sistema de cómputo estándar. En la Universidad del Valle se dispone de estas arquitecturas hardware basadas en FPGAs y de las herramientas de desarrollo que posibilitan implementar de forma más eficiente las funciones propuestas en este proyecto.

La implementación de este trabajo de grado ayudará a mejorar el cálculo del valor crítico dado por el algoritmo basado en [1] el cual es ejecutado en procesadores de aplicación general. Se tratará de llegar a una cota c, donde $3.1 < c <= 3.796$.

1.3 OBJETIVO GENERAL

Identificar y diseñar en FPGA las funciones críticas para la ejecución en una workstation del algoritmo para el cálculo de cotas en la transición de fase en el modelo Hard-Core en 2D.

1.4 OBJETIVOS ESPECÍFICOS

- Identificar las funciones matemáticas consideradas críticas para el desempeño del algoritmo computacional en cuestión.

- Identificar los recursos disponibles claves en la arquitectura de la FPGA y en la tarjeta de desarrollo, que permitan obtener un máximo procesamiento paralelo para las funciones críticas identificadas.

- Diseñar diferentes arquitecturas a implementar en la FPGA para la ejecución de las funciones críticas identificadas.

- Analizar el rendimiento de ejecución de las funciones críticas implementadas con los diseños propuestos sobre la FPGA y la workstation de propósito general.

1.5 RESULTADOS ESPERADOS

El algoritmo en cuestión ya está implementado en lenguaje MATLAB® y se implementará en una FPGA que trabajará independientemente en una tarjeta de desarrollo. Por tanto, los resultados esperados al finalizar este trabajo serán:

- Diseño del sistema electrónico del acelerador basado en FPGA, con variaciones en su arquitectura.
- Análisis del rendimiento de ejecución de las funciones críticas implementadas con los diseños propuestos sobre la FPGA y la workstation de propósito general.

2. MARCO TEORICO

El problema principal de este trabajo es acelerar la ejecución de un algoritmo particular que calcula valores críticos de listas de números naturales. Dichas listas están relacionadas con matrices de probabilidades. El procesamiento de las listas es una programación lineal de un problema complejo clasificado como NP-difícil dentro de la teoría de grafos. La programación lineal es un método muy utilizado para resolver problemas de optimización de modelos matemáticos bajo restricciones representadas por relaciones lineales. Por ejemplo, en Finanzas se utiliza para hallar el menor costo o la máxima ganancia.

2.1 CARACTERÍSTICAS DE LOS ALGORITMOS

De acuerdo a [4], los algoritmos planteados para resolver problemas matemáticos tienen propiedades que definen su tipo con base a la complejidad de su solución. Las siguientes definiciones dan contexto al problema principal planteado en este trabajo:

- Problema matemático: es un ejercicio a resolver. Se formula por: 1) una descripción de sus parámetros o variables libres, y 2) una declaración acerca de las propiedades que debe tener su solución para ser resuelta satisfactoriamente.

- Instancia del problema: se obtiene especificando un valor a todos los parámetros del problema.

- Eficiencia del algoritmo: la noción de eficiencia involucra todos los recursos de cómputo de un algoritmo, sin embargo, en la mayoría de los casos el recurso más utilizado para medir la eficiencia de un algoritmo es el tiempo.

- Tamaño del problema: los requerimientos de tiempo de un algoritmo son convenientemente expresados por un parámetro llamado el tamaño de una instancia del problema, que tiene la finalidad de representar la cantidad de datos de entradas necesarios para describir una instancia. Comúnmente, se mide el tamaño del problema escogiendo un esquema de codificación apropiado, debido

a que para una misma instancia puede haber varias codificaciones, diversos símbolos y sistemas informáticos.

- Función de complejidad de tiempo: expresa el tiempo necesario que requiere un algoritmo para resolver una instancia del problema como función del tamaño de entrada. Esta función puede ser polinomial o exponencial.

- Algoritmos de tiempo polinomial: se dice que un algoritmo es de este tipo, cuando tiene una función de complejidad de tiempo $O(p(n))$ para una función polinomial p donde n es la longitud de entrada. Ej.: $n,\ n^2,\ n^3, n^2+2$.

- Algoritmos de tiempo exponencial: se dice que un algoritmo es de este tipo, cuando su función de complejidad de tiempo no puede ser restringida o representada por una función polinomial. Ej.: $2^n,\ 3^n$.

- Intratabilidad: un problema se dice difícil de resolver, cuando su solución es tan difícil que ningún algoritmo de tiempo polinomial puede resolverlo. Los algoritmos de tiempo polinomial son más deseados por dos razones: 1) a mayor tamaño del problema menor tiempo en comparación con los de tiempo exponencial, y 2) a medida que se mejora la tecnología donde se procesa el algoritmo el tiempo de ejecución disminuye a razón muchísimo mayor en comparación con los de tiempo exponencial.

- Reducibilidad: un problema se dice que puede ser reducido a otro problema cuando puede ser transformado o reducido en tiempo polinomial en otro problema. Un problema A puede ser reducido a un problema B cuando existe un algoritmo eficiente para solucionar B. Cuando esto es posible, resolver B no puede ser más complejo que solucionar A.

- Determinístico: significa que un sistema ante las mismas entradas produce las mismas salidas bajo una serie de etapas, instrucciones o reglas. Si un sistema es no determinístico, puede producir diferentes salidas ante las mismas entradas, pero también bajo una serie de instrucciones.

- Relajación de un problema: es una estrategia de modelamiento. Una relajación es una aproximación de un problema difícil a otro

problema de más fácil solución. La solución de una relajación de un problema provee información acerca del problema original.

2.2 CLASIFICACIÓN DE LOS PROBLEMAS

Los problemas se clasifican como tipo P (*polynomial time*), NP (*non deterministic polynomial time*), NP-completo o NP-difícil, entre otros.

Los problemas tipo P son determinísticos, como las máquinas de estado (o máquinas de Turing) y tienen las siguientes características:

- El algoritmo puede realizar un sólo cálculo computacional al tiempo.
- Dada una instancia I, el algoritmo determina o deriva una estructura (o solución S).
- La exactitud de la respuesta S depende o es inherente al algoritmo.
- Los problemas clase P pueden ser resueltos con algoritmo determinísticos de tiempo polinomial.

Los problemas NP son problemas no determinísticos y tienen las siguientes características:

- Dada una instancia I, se supone o se conjetura una estructura (o solución S).
- El algoritmo puede resolver más de una secuencia de cálculos computacionales en paralelo.
- La exactitud de la respuesta S se verifica de una manera determinística normal.
- Los problemas tipo NP pueden ser resueltos con algoritmos no determinísticos de tiempo polinomial.

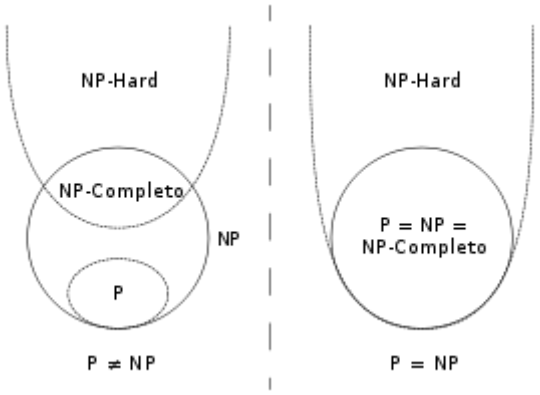
Los problemas tipo NP-completos son los más difíciles dentro del tipo NP. Para que un problema C sea NP-completo debe cumplir las dos siguientes características:

1. El problema C está en la clase NP.
2. Todos los problemas NP puede ser reducido en tiempo polinomial en C. Esto quiero decir, una transformación polinómica de L en C es un algoritmo determinista que transforma instancias de $l \in L$ en instancias de $c \in C$, tales que la respuesta a c es positiva si y solo si la respuesta a l lo es.

Los problemas NP-difícil son los problemas al menos tan complejos como NP, pero no necesariamente en NP. Un problema es NP-difícil si cumple

con la 2da propiedad de los NP-completos, pero no necesariamente con la 1ra. La clasificación y nomenclatura de los problemas NP se presenta en el diagrama de la figura 1 donde se observan dos posibilidades de la clasificación, su diferencia radica en determinar si $P = NP$ ó $P \neq NP$.

Figura 1: Diagrama de Euler de los tipos de problemas.



Fuente: Behnam Esfanbod, GNU Free License.

Encontrar un algoritmo de solución óptimo a problemas de tipo NP-difícil o NP-completo es difícil en la práctica porque en muchas ocasiones la creación de algoritmos que van tendiendo a estos tipos terminan siendo de tiempo exponencial y luego se aproxima una solución similar con tiempo polinomial.

El famoso problema del Agente viajero (*Travel Salesman Problem - TSP*) ilustra este tipo de problemas complejos pues hace parte de los 21 problemas NP-completos de Karp, los cuales son estudiados de manera extensa en matemáticas y ciencias de la computación. Dada una lista de ciudades y las distancia entre cada par de ellas, el TSP responde al interrogante de encontrar la ruta más corta posible en donde el agente viajero visite cada cuidad exactamente una vez y regrese al origen.

El TSP puede tener como parámetros: el número y posición de las ciudades, la distancia y el costo de viaje entre ellas; siendo estas variables únicas para cada instancia del problema. Este problema puede ser visto como un problema de optimización o un problema de decisión. Para ilustrar esta diferencia, un problema de optimización seria minimizar la longitud del recorrido que debe realizar el viajero para visitar todas las ciudades y regresar al origen. Un problema de decisión sería que el viajero tratara de encontrar si existe un recorrido que no sobrepase una longitud

X. Se debe recalcar la diferencia entre estos dos tipos de problemas, sin embargo, ambos enfoques tienen la misma complejidad.

La teoría de la clasificación P o NP sólo aplica para problemas de decisión en los cuales su solución es un sí o un no. Sin embargo, todo problema de optimización puede ser visto como uno de decisión si se plantea de la siguiente manera: si un problema de optimización pregunta por una estructura de un cierto tipo que tenga un mínimo costo, entonces se puede asociar a ese problema un problema de decisión, el cual incluya una restricción numérica M como un parámetro adicional, y se pregunte si existe una estructura del tipo requerido con costo no mayor a M.

Todos estos aspectos matemáticos de las características de los algoritmos son tenidos en cuenta en [1] para realizar una relajación del problema original para poder resolverlo. Es decir, en lugar de resolver el problema original se resuelve una relajación del problema usando programación lineal. Esta relajación ofrece una cota al valor buscado en el problema original.

2.3 EL MODELO HARD-CORE

El modelo hard-core (HC) es un modelo matemático que se emplea de acuerdo a [5]: en física para describir el comportamiento de la materia en estado gaseoso, en biología para la inferencia de los árboles filogenéticos, en telecomunicaciones para establecer un conjunto mínimo de sensores estacionarios para logar buena cobertura para un área dada, y en muchas otras aplicaciones. Este modelo profundamente estudiado en matemáticas y ciencias de la computación, se basa en la distribución de unos puntos o cuerpos correspondientes a los vértices de un grafo.

En física, por ejemplo, el modelo tiene la capacidad de determinar la topología de un gran número de partículas de un tamaño no insignificante, en los vértices de un grafo, evitando el contacto entre partículas vecinas. El problema, que se trata de solucionar con el modelo HC, es un problema de optimización combinacional de tipo NP-difícil.

Un enfoque natural para tratar estos problemas es diseñar un algoritmo similar que sea eficiente y que garantice una buena aproximación a la solución óptima. En las últimas dos décadas se han realizado numerosos diseños o enfoques de algoritmos de aproximación, los cuales se han

desarrollado gracias a técnicas de optimización como la programación lineal o la programación semidefinitiva como se menciona en [6].

2.3.1 Definición del modelo hard-core

El modelo hard-core (HC) se define como: Considerando un grafo G, definido por vértices V y aristas E donde $G = (V, E)$, de tamaño $N = |V|$, y a cada vértice $i \in V$ se le asocia un número o valor de ocupación $\sigma_i \in \{0, 1\}$, donde el valor 0 se refiere a un espacio libre y 1 a un espacio ocupado. La ecuación 1 dada en [5], es la medida de Gibbs, la cual permite determinar la probabilidad que tiene una configuración particular en existir, teniendo en cuenta los valores de ocupación correspondientes a cada vértice, por tanto, para cada aplicación del modelo HC, Z es una función de partición particular que hace posible que la ecuación determine una distribución de probabilidad y μ es una constante.

$$P\left(\{\sigma_i\}_{i=1,2,\dots N}\right) = \frac{1}{Z} * e^{\mu * \sum_{i \in V} \sigma_i} * \prod_{(ij) \in E} \left(1 - \sigma_i \sigma_j\right) \quad \text{Ecuación 1}$$

La medida de Gibbs representa los posibles estados de un sistema mecánico en equilibrio térmico, con un foco calórico a una temperatura fija. Las medidas de Gibbs se utilizan de manera extensa en las ciencias, debido a que son distribuciones de probabilidad, las cuales aparecen naturalmente en muchísimos problemas de mecánica estadística como se indica en [7], siendo el mejor ejemplo, la aplicación de estas medidas de Gibbs dentro del modelo de Ising para el estudio de los materiales ferromagnéticos, los cuales varían sus propiedades eléctricas en función de la temperatura.

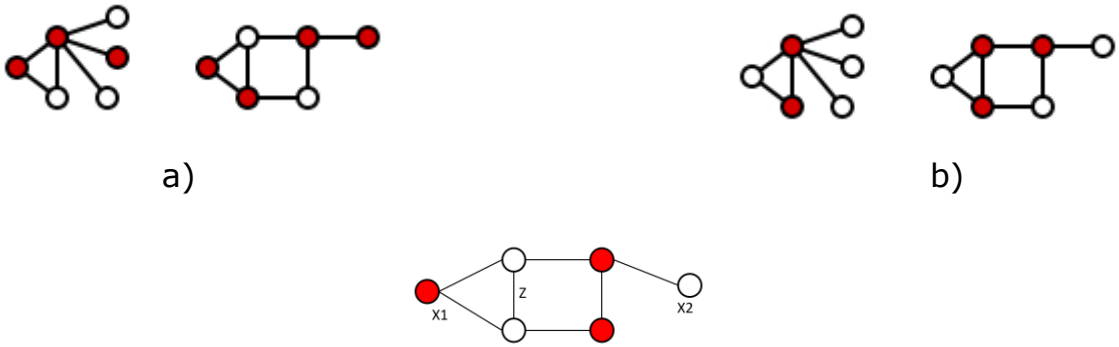
2.3.2 Cobertura de vértices

Formalmente se define la cobertura de vértices como: dado un grafo $G = (V, E)$, la cobertura de vértices V' es un subconjunto de V, tal que para toda arista $uv \in E \rightarrow u \in V' \vee v \in V'$, dicho de otra forma, la cobertura de vértices es el conjunto de vértices V', donde toda arista de G incida al menos sobre un vértice perteneciente a V'. El conjunto (trivial) de todos los vértices de un grafo es una cobertura de vértices.

En la figura 2a) se observan de color rojo la cobertura de vértices de dos grafos. En la figura 2b) se muestran los mismos grafos con su mínima cobertura de vértices. En las figuras 2a) y 2b) se observa que

independiente del número de vértices rojos todas las aristas inciden sobre al menos uno de ellos. En 2c) se observa un grafo cuyos vértices rojos no corresponden a una cobertura de vértices debido a que la arista Z no incide sobre ningún vértice rojo.

Figura 2: a) Cobertura de vértices. b) Cobertura de vértices mínima. c) Grafo sin cobertura de vértices.



2.3.3 Conjunto independiente

El conjunto independiente de un grafo G es un subconjunto de vértices, de tal manera que ningún par de estos correspondan a una arista en G. Es importante indicar, que un conjunto independiente es el complemento de una cobertura de vértices. En la figura 2a) y 2b) el conjunto independiente son los puntos blancos (los puntos rojos forman una cobertura de vértices). El número de vértices de un grafo es la suma del número de vértices de la cobertura más el número de vértices del conjunto independiente.

2.4 TRANSICIONES DE FASE

Se estudia el modelo HC aplicado principalmente en la física estadística en los gases, debido a que este modelo permite observar las diversas clases de fases que puede tener un sistema termodinámico. Se entiende como fase a un estado determinado (estable o no) del material, por ejemplo: sólido, líquido o gas. Pero también se puede dentro del modelo de Ising las fases ferromagnética y paramagnética. Así mismo como el estado de superconductividad en algunos materiales cuando están debajo

de una temperatura critica. En general, en equilibro, cada fase tiene unas propiedades termodinámicas bien definidas.

También puede existir la situación donde dos fases coexistan en el diagrama de fase de una sustancia (Presión vs Temperatura). En estos diagramas se pueden identificar, por ejemplo, puntos críticos donde se indica el valor máximo de temperatura en la cual dos fases pueden coexistir. El punto crítico dado en estos diagramas es acotado para ser calculado algorítmicamente.

2.5 ARTÍCULO: IMPROVED BOUNDS OF THE PHASE TRANSITION FOR THE HARD-CORE MODEL IN 2D

En esta sección se realiza un compendio del artículo titulado *Improved Bounds Of The Phase Transition For The Hard-Core Model In 2D* de los autores: J. C. Vera, E. Vigoda y L. Yang, en el cual se plantea una programación lineal para mejorar la cota inferior de un valor critico de transición de fase. Esta programación lineal tiene el objetivo de reducir la carga de ejecución sobre un sistema computacional que conlleva el cálculo del valor críticos cuando las listas de índices dadas son muy grandes. La determinación de este valor crítico permite calcular una constante de interés en el campo de la física de los gases, debido a que dentro del modelo hard-core aplicado a la materia gaseosa se plantea el concepto de la transición de fase entre los estados estables que pueden tomar las variables aleatorias que describen el estado (o fase) del gas.

En este artículo se establece que el modelo hard-core es un modelo de los gases que se componen de partículas no-insignificantes y consecuentemente las configuraciones de este modelo serán conjuntos independientes. Considerando un grafo finito $G = (V, E)$ de tamaño $N = |V|$ y a cada vértice $i \in V$ se le asocia un número o valor de ocupación $\sigma_i \in \{0, 1\}$, donde el valor 0 se refiere a un espacio libre y 1 a un espacio ocupado. Dada una actividad $\lambda > 0$ (correspondiente a la fugacidad del gas), las configuraciones del modelo son Ω conjuntos independientes de G donde $\sigma \in \Omega$ y tiene un peso de $w(\sigma) = \lambda^{|\sigma|}$. La medida de Gibbs que establece la probabilidad que una determinada configuración exista se define en la ecuación 2.

$$\mu(\sigma) = \frac{w(\sigma)}{Z} \qquad \text{Ecuación 2}$$

Donde $Z = \sum_{\eta \in \Omega} w(\eta)$ es la función de partición.

En este artículo se plantea que una cuestión fundamental en los modelos de física estadística, como el modelo hard-core, es si existe un único o múltiples medidas de Gibbs en $\mathbb{Z}^2$.

En este artículo se define una probabilidad condicional (o marginal) p_L^{even} (ó p_L^{odd}) y se establece que una relación dada por la ecuación 3.

$$\lim_{L \to \infty} |p_L^{odd} - (p_L^{even})| = 0 \qquad \text{Ecuación 3}$$

Donde L es una longitud, entonces si este límite es nulo existe una única medida de Gibbs en $\mathbb{Z}^2$, y si este límite es mayor a 0 entonces existen múltiples medidas de Gibbs.

En relación a la actividad crítica $\lambda_c(\mathbb{Z}^2)$ en este artículo se plantea:

- Se cree que existe una actividad crítica $\lambda_c(\mathbb{Z}^2)$ que permite establecer la siguiente relación: para una actividad λ, si $\lambda < \lambda_c(\mathbb{Z}^2)$ la unicidad de la medida de Gibbs se mantiene, y si $\lambda > \lambda_c(\mathbb{Z}^2)$ existirán múltiples medidas de Gibbs y la unicidad no se mantiene.

- Para dar una cota superior de la actividad crítica, Blanca et al. en [8] mejoró el cálculo de la actividad crítica mostrando la relación $\lambda_c(\mathbb{Z}^2) < 5.3646$.

- Weitz mostró que $\lambda_c(\mathbb{Z}^2) \geq \lambda_c(\mathbb{T}_4) = 27/16 = 1.6875$.

- Restrepo et al. en [3], mejoró el enfoque de Weitz para $\mathbb{Z}^2$ y estableció la relación $\lambda_c(\mathbb{Z}^2) > 2.388$.

- Vera et al. en [1] plantea que para cualquier conjunto independiente σ, todos los valores de la actividad $\lambda > 3.4$.

- Existen múltiples resultados heurísticos que sugieren que $\lambda_c(\mathbb{Z}^2) \approx 3.796$ como se indica en [1].

En este artículo se establece un límite a los alcances de los enfoques de Weitz o Restrepo et al.

En este artículo se proponen nuevas técnicas para mejorar las cotas inferiores de λ_c. También se prueba que $\lambda_c(T_{saw}(\mathbb{Z}^2)) < 3.4$ y por tanto el método de Weitz no es suficiente para alcanzar el limite conjeturado de 3.796. Un problema en estos enfoques es que existen condiciones que no son necesariamente realizables en $\mathbb{Z}^2$.

2.6 DESCRIPCIÓN DEL ALGORTIMO COMPUTACIONAL PARA EL CÁLCULO DE VALORES CRITICOS EN EL MODELO HARD CORE

En esta sección se explica el algoritmo que corresponde a una programación lineal de autoría de J. C. Vera y E. Vigoda, el cual sirve para acelerar la ejecución matemática del problema de optimización combinacional para el cálculo de cotas de transición de fase en el modelo hard-core. El algoritmo calcula un valor crítico c de la siguiente forma. Una función $y = fRec(x)$ con entrada $x \in [0,1]^n$ y salida $y \in [0,1]^n$ es construida, donde t es un parámetro. Vera y Vigoda demuestran que si la función $fRec(x)$ tiene un punto fijo entonces SSM (*Strong Spatial Mixing*) vale para $\mathbb{Z}^2$. Por tanto, para el máximo parámetro t para el cual la función $fRec(x)$ tiene un punto crítico, t es una cota inferior para el valor crítico. Dicho valor crítico es encontrado usando una búsqueda binaria en el intervalo [2,4].

Dado $n > 0$ y sea $L = \{L_1, L_2, \ldots, L_n\}$ una lista de conjuntos ordenada como una matriz donde cada L_i es una fila que tiene a lo más tres elementos x_j, contando repeticiones del conjunto $\{1,2,3,\ldots,\infty\}$ que pueden existir.

Sea t un parámetro fijo.

Sean y un vector de n componentes y $x \in L$.

Sea $y = fRec(x)$ una función definida por,

$$y = fRec(x) = \left(\frac{1}{1 + t * \Pi x_j}, \quad \frac{1}{1 + t * \Pi x_j}, \quad \frac{1}{1 + t * \Pi x_j}, \ldots, \frac{1}{1 + t * \Pi x_j} \right)$$

$$y_i = \frac{1}{1 + t * \Pi x_j} \qquad x_j \in L_i \quad , \quad i = 1,2 \ldots n \quad , \quad j \in \mathbb{N} \quad \text{Ecuación 4}$$

Encontrar si la función y_i tiene un (único) punto fijo en $[x, y] = [0,1]^n$ es lo que permite determinar un valor critico (la cota de la transición de fase). La notación anterior significa encontrar en el dominio $[0,1]$ en la dimensión n, si las variables x y y, relacionadas por la función $fRec()$, tienen uno o varios puntos fijos. La respuesta depende del parámetro fijo t y se sabe que existe un valor o punto crítico c tal que si $1 < t < c$ existe un único punto fijo, y si $t > c$ existen más de uno puntos fijos.

El punto fijo x de una función $fRec()$ es un punto cuya imagen y producida por la misma función tiene el mismo valor del punto.

$$fRec(x) = x \qquad \text{Ecuación 5}$$

Para encontrar el punto crítico c con cierta precisión, dada una lista L, se escoge un parámetro fijo t y se calcula:

Ecuación 6

$$x_0 = [0, 0, \dots, 0], \quad x1 = fRec(x_0),$$
$$x_2 = fRec(x_1), \quad x_3 = fRec(x_2), \dots,$$
$$x_i = fRec(x_{i-1}), \quad x_{n+1} = fRec(x_n)$$

Se puede probar que si $t > c$ este proceso diverge, donde los valores pares de n convergen a un valor y los impares a otro. Si $t < c$ la función converge. Por tanto, si n es suficientemente grande entonces basta con mirar si $|x_n - x_{n+1}|$ es suficientemente pequeño o no.

Para encontrar el valor critico c se utiliza el algoritmo *binary search*, ensayando para varios valores de t. Se inicializa con los valores límite por la izquierda y límite por la derecha dentro de la recta real, $tl = 2$ y $tr = 4$ respectivamente y gracias a estos valores iniciales toma un número determinado de iteraciones w para realizar el algoritmo de búsqueda binaria. Para cada lista L se requiere calcular la función $fRec(x)$ unas $w * iter$ veces. Se recalca que el computo de cada componente de $fRec(x)$ hace la misma operación que es de la forma $1/(1 + t * p * q * r)$ donde t esta fijo casi todo el tiempo, y por tanto, estas componentes se están calculando unas $w * iter * n$ veces, valor que puede llegar a ser del orden de miles de millones cuando n es muy grande y que causa que la ejecución del algoritmo tarde un tiempo significativo y el consumo excesivo de los recursos de un sistema de cómputo estándar.

Otra manera de explicar este algoritmo en forma de pseudocódigo es la siguiente: partiendo de un vector nulo x de n componentes, se debe iterar evaluando la función fRec con los valores pares e impares de x. Esta iteración parará cuando la diferencia entre estos valores par e impar se suficientemente pequeña. Al final se verifica una condición extra para mirar si el resultado es único ("*unique*") o múltiple ("*non-unique*").

$x\_0 = (0,\ldots,0);$
$k = 0;$
$do\{$
$\quad x\_\{2k+1\} = fRec(x\_\{2k\}).$
$\quad x\_\{2k+2\} = fRec(x\_\{2k+1\})$
$\quad k = k+1;$
$\}until\{\ norm\left(x_{\{2k\}} - x_{\{2k-2\}}\right) < epsilon\}$

$If\left(norm(x\{2k\} - x\{2k+1\} < 10 * epsilon)\right)\{$ *"unique*"$\}else\{$ *"non-unique"*$\}$

En realidad, en la recurrencia anterior se quiere calcular los límites: $x_{even} = \lim_{k\to\infty} x_{2k}$ y $x_{odd} = \lim_{k\to\infty} x_{2k+1}$

Y chequear si: $x_{even} = x_{odd}$

Para ilustrar el funcionamiento de la unidad, se presenta la lista más pequeña que se debe procesar y la función $fRec()$ asociada a dicha lista. Se poder observar que los índices de los valores x dentro de fRec() en la ecuación 7 son determinados por la lista en la tabla 1.

Tabla 1: Lista L4

L4		
3	4	5
1	3	4
1	4	5
2	2	4

Ecuación 7

$$y = fRec(x) = \left(\frac{1}{1 + \lambda x_3 x_4 x_5}, \frac{1}{1 + \lambda x_1 x_3 x_4}, \frac{1}{1 + \lambda x_1 x_4 x_5}, \frac{1}{1 + \lambda x_4 x_2^2}\right)$$

A medida que las listas son más grandes el algoritmo tiene un menor rendimiento en un computador genérico de pequeña o mediana potencia y puede demorar muchas horas para su ejecución.

2.7 CODIFICACIÓN EN MATLAB®

Todo el algoritmo explicado en la sección anterior para encontrar el valor crítico c de cada lista está codificado en lenguaje MATLAB®. El código al final de esta sección consta de tres funciones: $wsmCritMar()$, $wsm()$ y $fRec()$. Este algoritmo es una búsqueda binaria la cual se inicia llamando la función $wsmCritMar()$ (en color azul), que recibe como argumento un valor $dim \in \{4, 6, 8, 10, 12, 14, 16, 18, 20\}$ y esto determinará la lista a procesar L4, L6, L8, etc. La búsqueda binaria comienza entre los puntos $tl = 2$ y $tr = 4$ para buscar el valor crítico el cual ya se sabe heurísticamente es 3.796. En la función $wsmCritMar()$ se toma el promedio de estos puntos denominado tm para determinar si es un punto cercano al deseado. Esto se realiza hasta cumplir con la condición del mínimo intervalo establecido en el control *while()* llamado COMPARADOR 4. Este nombre es dado por el autor de este trabajo de grado y tiene relación con el posterior diseño de la unidad hardware que ejecutará esta función. El texto en color rojo no es parte del lenguaje Matlab y sirve para dar claridad de los componentes más importantes del código.

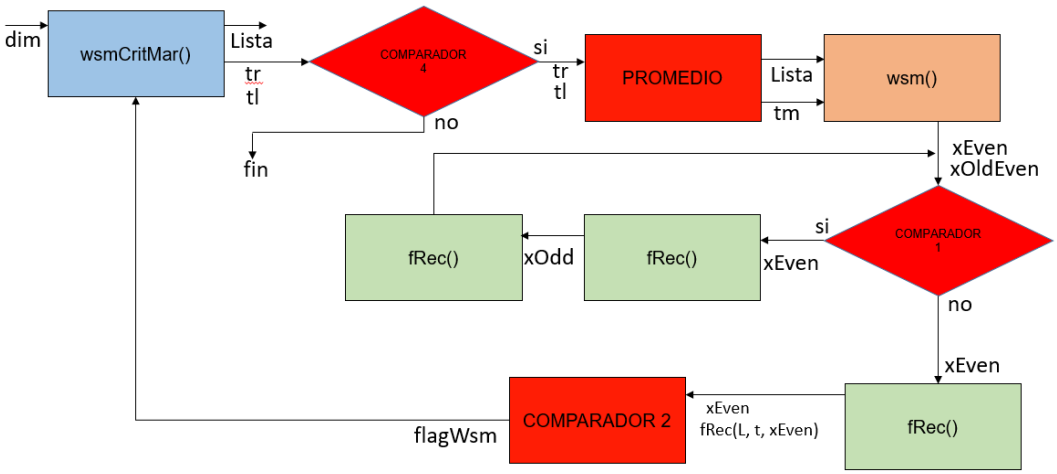
En el interior de la función $wsm()$ (en color naranja), se realiza el proceso descrito en la ecuación 6 donde se experimenta la pseudo recurrencia de la función $fRec()$ hasta que se cumpla la condición expuesta en el bucle while() nombrado como COMPARADOR 1. Cuando se sale de este bucle de control se ejecuta la función nombrada como COMPARADOR 2 para generar una señal de bandera que determinará el siguiente paso de la búsqueda binaria. Esta es la forma de hacer el cálculo de los límites anteriormente descritos.

La función $fRec()$ (en color verde) ejecuta la operación expuesta en la ecuación 4 donde se debe evaluar la función y_i, n veces, para evaluar solamente una vez $fRec()$. Es en esta evaluación donde existen más oportunidades para mejorar el paralelismo del algoritmo pues es esta función, con naturaleza recurrente, la que genera un cuello de botella en la CPU.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [tl,tr] =wsmCritMar(dim)
   %usage [t] = wsmCritMar(dim)
   %  Finds the lambda critical for WSM for list L_dim,
   L = load(['M', num2str(dim), 'red3.txt']);
   n = size(L,2);        %retorna el tamaño de la lista en la dimension 2
   prec= 4;              %es decir, el numero de columnas n=3
   tl = 2;               %critical point is between tl=2 and tr=4
   tr = 4;
   %decrease interval size using binary search
   while tr - tl > 1.1*10^(-prec) --------------------------%COMPARADOR 4 ------- ---|
      fprintf('tl = %1.3f  tr =%1.3f \n', tl, tr);                                  |
      tm = round(10^prec*(tr+tl)/2)*10^(-prec);  ←----- PROMEDIO                    |
      flagWsm = wsm(L,tm);                                                          |
      if flagWsm      % si todos los elementos de flagWsm son 1 entonces retorna 1  |
         tl = tm;                                                                   |
      else            % si algun elemento de flagWsm es 0 entonces retorna 0        |
         tr =tm;                                                                    |
      end                                                                           |
   end ------------------------------------------------------------------------------|
end

function flagWsm = wsm(L,t)
   tic
   precWSM = 12;
   [n,~] = size(L);          % n = numero de filas de L
   xEven = zeros(n, 1);
   iter = 0;
   xOldEven = ones(n, 1);
   while max(abs(xEven-xOldEven)) > 10^(-precWSM) ------%COMPARADOR 1------|
      xOldEven = xEven;                                                    |
      xOdd = fRec(L, t, xEven);                                            |
      xEven = fRec(L, t, xOdd);                                            |
      iter = iter+1;                                                       |
   end ---------------------------------------------------------------------|

   flagWsm = (abs(xEven - fRec(L,t,xEven)) < 10^(-precWSM+4) ) ← %COMPARADOR2
   fprintf('  %d iters in %f secs \n', iter, toc)
end

function y = fRec(L, t, x)
   x(end+1) = 1;
   P = prod(x(L), 2);
   y = 1./(1+t*P);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

En la figura 3 se observa el flujo de datos de la ejecución del anterior código Matlab, donde los componentes presentan los mismos colores utilizados en dicho código. Los nombres de los componentes en esta figura corresponden a los bloques hardware como se indica en el capitulo 4.

Figura 3: Diagrama del flujo de datos del algoritmo



Se observa en la figura 3, que el algoritmo comienza estableciendo un dimensión (*dim*) como parámetro de la función wsmCritMar(). Esto define una lista determinada. Los valores iniciales de la búsqueda binaria tr=4 y tl=2 pasan al componente COMPARADOR_4 que realiza el primer bucle while() en Matlab. Si la condición del bucle while es positiva los valores tr y tl pasan al componente PROMEDIO y se calcula su promedio que luego pasa como argumento a la función wsm(). Dentro de la función wsm() los vectores xEven y xOldEven entran al segundo bucle while() que corresponde al componente COMPARADOR_1. Si la condición de este bucle es positiva, entonces se implementa la función fRec() de manera pseudorecurrente, hasta no satisfacer la condición en COMP_1. Cuando sale del bucle while() del COMP_1 el valor xEven es evaluado nuevamente por la función fRec(). Finalmente, el resultado de fRec(L, t, xEven) y el vector xEven pasan al componente COMPARADOR_2. Este último da como salida la señal de sincronización flagWsm, que controla la selección de la función $if()$ implementada en el componente wsmCritMar(). El final de la ejecución del algoritmo sucede cuando la condición del while de COMP_4 es negativa.

3. METODOLOGÍA

3.1 CLASIFICACIÓN DE LA INVESTIGACIÓN

Metodológicamente los proyectos se pueden clasificar en diferentes tipos según diversos criterios como se indica en [9]. Según el control de las variables a relacionar, este trabajo de investigación se puede clasificar de tipo experimental. En este caso las variables a relacionar son el paralelismo de la arquitectura general del sistema a diseñar y su efecto sobre el tiempo de ejecución y área consumida por dicho diseño.

Según la finalidad de la investigación, este proyecto también puede clasificarse como investigación de tipo tecnológica, debido a que está orientado a demostrar la validez de la implementación de ciertas arquitecturas hardware en FPGA con el fin de acelerar la ejecución del algoritmo, el cual a su vez utiliza principios matemáticos.

Se puede establecer otro tipo de clasificación según el lugar donde se realiza la investigación. En este caso se clasifica como proyecto de laboratorio ya que para realizar la investigación se utiliza una workstation y un sistema de desarrollo basado en FPGA. Estos sistemas hardware permiten un mayor control sobre las variables en las cuales se desea experimentar.

3.2 METODOLOGÍA ASOCIADA CON EL ALGORITMO

El primer objetivo específico de este proyecto es identificar las funciones matemáticas consideradas críticas para el desempeño del algoritmo computacional en cuestión. Para abordar este objetivo se analiza con asesoría del Prof. J. C. Vera cuales son estas funciones.

El algoritmo computacional en cuestión consta de dos clases de iteraciones, la primera relacionada a la búsqueda binaria acotada por la función que ejecuta la unidad COMPARADOR_4 y la segunda iteración relacionada al proceso acotado por la función dada en la unidad COMPARADOR_1. Estas dos clases de iteraciones se puede observar como las realimentaciones vistas en la figura 3.

Para identificar estas funciones se examinó el uso de los recursos del computador (CPU, memoria, disco) durante la ejecución de dicho

algoritmo. Se observó que para listas pequeñas su ejecución es casi instantánea y consume un bajo porcentaje de los recursos. Para listas más grandes se observó que el algoritmo consume el 100% de la CPU.

Al examinar los resultados de la ejecución en Matlab® se observa que las instrucciones acotadas por el COMPARADOR_1 iteran aproximadamente $iter = 700.000$ veces y las funciones acotadas por el COMPARADOR_4 iteran aproximadamente $w = 15$ veces. Al cambiar las listas a procesar se observa que dichas iteraciones no varían significativamente. La diferencia fundamental para la evaluación del consumo del tiempo del algoritmo está en el valor del número de filas de cada lista (n), debido a que la función $fRec()$ consta de n evaluaciones de la función y_i. Se concluye entonces que $fRec()$ es la función más crítica.

Se recalca que, al variar considerablemente el tamaño del problema, el efecto de la función de complejidad de tiempo del algoritmo se hace más notable.

3.3 METODOLOGÍA DE DISEÑO TOP DOWN

La metodología Top Down comienza en un nivel superior. Como se indica en [10], las especificaciones de diseño son establecidas en términos del estado general del sistema, conociendo la interacción entre los componentes y desconociendo los detalles internos de cada componente. Puede ser visto como una rutina recursiva en donde se comienza con una especificación y descomposición, y termina con una integración y verificación. Se asocia a un sistema centralizado. Tiene como ventaja la fácil integración y por ende la verificación del sistema total. Es de fácil administración y en general se obtienen las ventajas de un sistema centralizado.

3.4 METODOLOGÍA DE DISEÑO BOTTOM UP

El diseño realizado por medio de un método Bottom Up consiste en asumir desde un inicio que el estado general de los componentes del diseño es difícil de obtener. El diseño comienza especificando los requerimientos y capacidades de cada componente individual y entonces el comportamiento colectivo emerge de la interacción entre estos componentes junto con el ambiente en los que se encuentran. Las ventajas que presenta esta metodología se basan en el hecho que no se necesita de tener todo el conocimiento del sistema global para lograr una

buena funcionalidad. No es de fácil integración y la interacción entre componentes debe ser muy bien planteada desde el inicio del proyecto.

Generalmente se combinan los dos tipos de metodologías top down y bottom up para aprovechar sus ventajas.

3.5 METODOLOGÍA PARA EL DESARROLLO DE LA ARQUITECTURA HARDWARE

El tercer objetivo específico de este proyecto es diseñar diferentes arquitecturas a implementar en la FPGA para la ejecución de las funciones críticas identificadas. Para abordar este objetivo se utiliza la metodología bottom up. Se decide iniciar el diseño de la arquitectura del sistema a partir del componente para la función $fRec()$.

Se identifica que la operación crítica para la implementación de la función $fRec()$ es el circuito de división, probándose diversas arquitecturas para un divisor punto fijo. El consumo de recursos del divisor se analizan los casos de 16, 32, 64 y 128 bits en punto fijo. Con el fin de disminuir la complejidad del hardware a diseñar no se abordó el formato de punto flotante.

Al finalizar la implementación de la función $fRec()$ y su controlador se diseñan las unidades hardware denominadas COMPARADORES 1, 2, 4 y PROMEDIO. Todos los componentes hardware diseñados son verificados de forma independiente. Con el fin de integrar todos los componentes, se plantean unidades de mayor jerarquía que corresponden a las funciones que llaman a fRec(). Se diseñan las unidades controladoras correspondientes para controlar el flujo de datos en el hardware y permitir la verificación de la ejecución del algoritmo total.

3.6 ARQUITECTURA FPGA INTEL® ARRIA II GX-125EF35C4ES

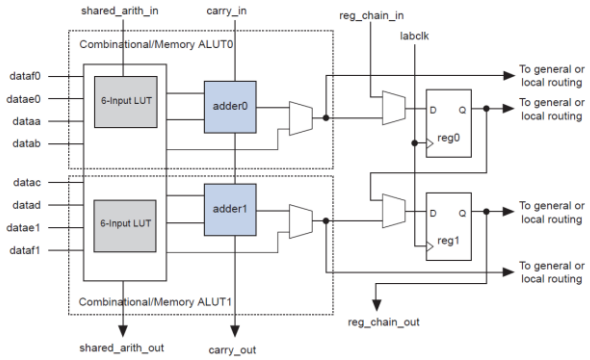
El segundo objetivo específico de este proyecto es identificar los recursos disponibles claves en la arquitectura de la FPGA y en la tarjeta de desarrollo, que permitan obtener un máximo procesamiento paralelo para las funciones críticas identificadas. Para abordar este objetivo se utiliza una FPGA de la familia Arria II GX (EP2AGX125EF35C4ES) siendo este un dispositivo con los recursos apropiados para este proyecto, los cuales son:

49640 ALMs, 8121,25 Kbits de memoria SRAM, 72 bloques DSP y 452 puertos I/O. Dicha FPGA es el componente esencial de la tarjeta de desarrollo Arria II GX (DK-DEV-2AGX125N).

3.6.1 MÓDULO DE LÓGICA ADAPTATIVO (ALM)

La arquitectura del ALM (*Adaptative Logic Module*) está compuesta por dos ALUTs (*combinational Adaptative Look Up Table*) y dos registros (figura 4), dicha arquitectura es optimizada para configurar funciones lógicas de hasta ocho variables, funciones aritméticas y registros. En la FPGA EP2AGX125 el total de ALMs es 49640.

Figura 4: Adaptative Logic Module - ALM

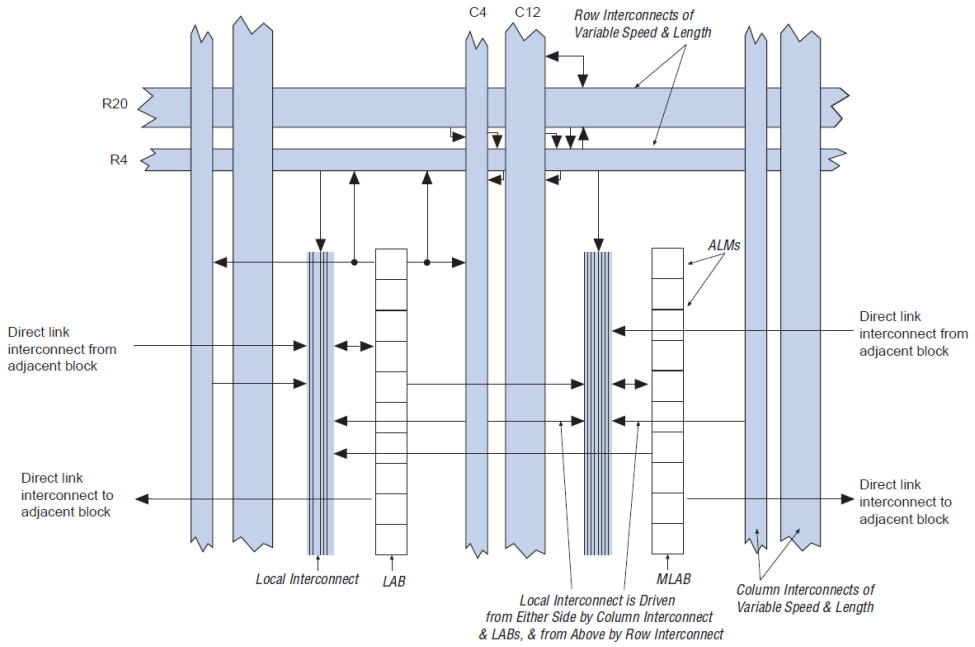


Fuente: Arria II GX Device Handbook [11].

3.6.2 BLOQUE DE MATRIZ LÓGICA (LAB)

Un LAB (*Logic Array Block*) está compuestos por diez ALMs, cadenas de carry optimizada, señales de control, entre otros elementos (Figura 5). Los LABs permiten alta flexibilidad de interconexión de sus diez ALMs, o entre ALMs de diferentes LABs. En la FPGA EP2AGX125 el total LABs es 4964.

Figura 5: Logic Array Blocks – LAB



Fuente: Arria II GX Device Handbook [11].

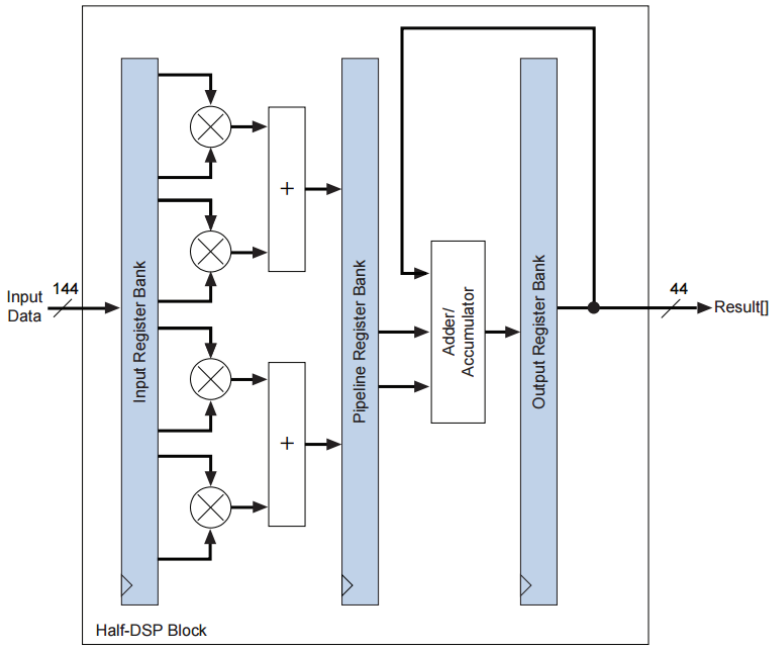
3.6.3 MEMORIA ON-CHIP

La memoria on-chip está compuesta por memoria M9K y bloques MLAB. El bloque M9K es memoria tipo SRAM de 9216 bits a 390 MHz, el cuales puede ser configurado en aspectos como: tipos de puertos (Simple, dual, dual verdadero), operación (registro de desplazamiento, RAM, ROM, FIFO), tamaño de palabra (1,2,4,8,9,16,18,32,36 bits) y paridad. Un MLAB corresponde a 640 bits de RAM a 500 MHz, el cual puede ser utilizado como LAB o como memoria propiamente dicha. En la FPGA EP2AGX125 el total de memoria on-chip es de 8121,25 Kbits, distribuida en 730 bloques de memoria tipo M9K (6570 Kbits) y 2482 MLABs de 640 bits (1551,25 Kbits).

3.6.4 BLOQUE DSP

Un modo de configuración del bloque DSP se observa en la Figura 6, se observan los elementos embebidos de la arquitectura del bloque DSP: multiplicadores paralelos embebidos, sumadores paralelos embebidos, acumulador, registros pipeline.

Figura 6 : Configuración del DSP en modo multiplicador sumador acumulador



Fuente: Arria II GX Device Handbook [11].

El bloque DSP se puede configurar en los modos: multiplicador independiente, doble multiplicador sumador, cuatro multiplicadores sumadores, multiplicador acumulador, circuito de desplazamiento, multiplicador sumador de alta precisión. También se configura para diferentes longitudes de palabra (9,12,18,36 bits). En la FPGA EP2AGX125 el total de bloques DSP es 72, los cuales pueden ser configurados como se indica en la tabla 2.

Tabla 2: Número y tipo de bloques DSP en la Arria II GX-125

	Multiplicadores paralelos independientes					**Multiplicador-sumador de alta precisión**	**Cuatro unidades de multiplicador-sumador**
Bloques DSP	**9x9**	**12x12**	**18x18**	**18x18 Complejo**	**36x36**	**18x36**	**18x18**
72	576	432	288	144	144	288	576

Fuente: Arria II GX Device Handbook [11].

3.7 METODOLOGÍA PARA LA PROGRAMACION EN FPGAs

El diseño de sistemas digitales implementados sobre FPGAs se basa en una serie etapas como se indica en la figura 7. Las herramientas EDA (*Electronic Design Automation)* que asisten y automatizan el diseño electrónico siguen dichas etapas.

Teniendo en principio un contexto de la aplicación del sistema a diseñar en la FPGA se realiza una planeación de los periféricos que se deben conectar al chip.

Se continua con la descripción del sistema a diseñar en un lenguaje de descripción de hardware VHDL o Verilog, los cuales permiten modelar los procesos concurrentes que experimentan los componentes electrónicos de un sistema digital.

Los editores para HDL asisten a la depuración de errores de sintaxis o semántica, permitiendo al compilador el inicio de la tarea de síntesis en la cual se realizan inferencias para transformar las descripciones realizadas en HDL a un modelo RTL equivalente.

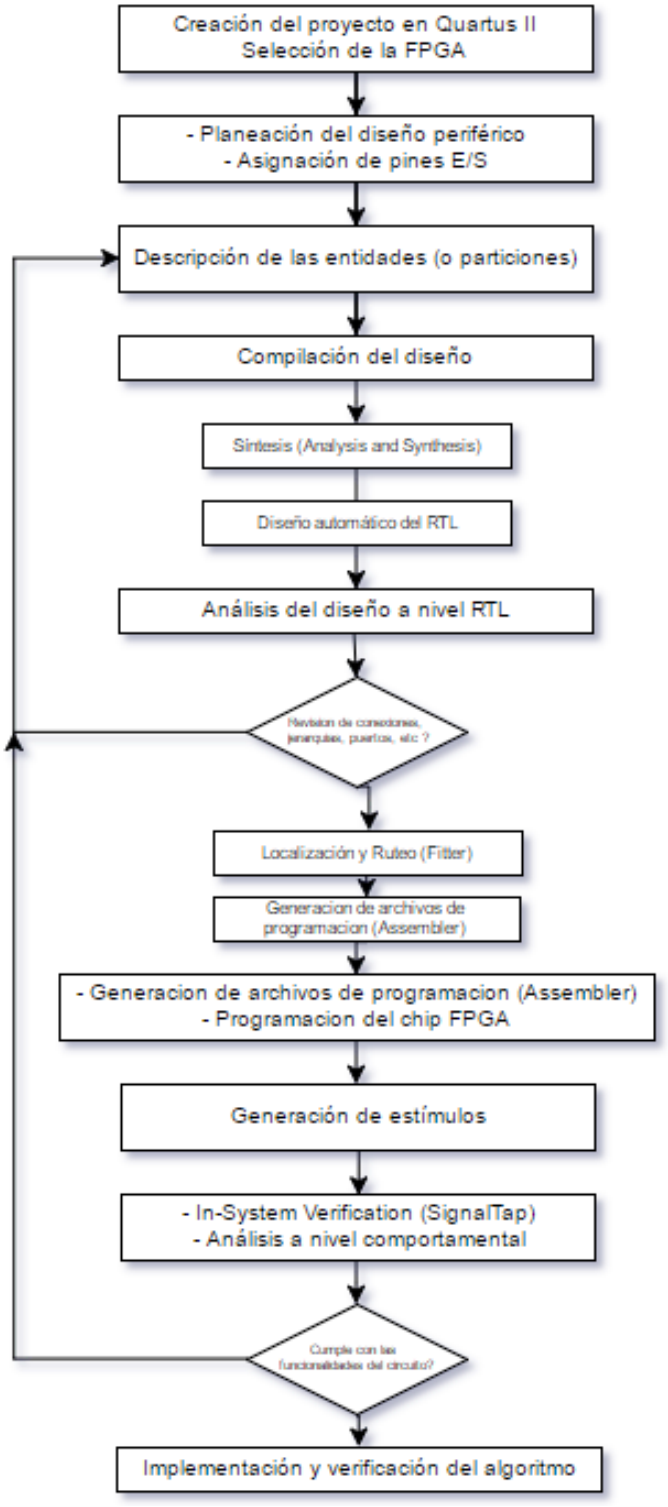
El diseñador puede verificar a nivel RTL (*Register Transfer Level*) el diseño sintetizado observando los resultados de la inferencia de la herramienta EDA para corregir errores en la descripción.

Cuando el modelo RTL es el deseado, se pasa a la etapa de mapeo de tecnología, donde la herramienta EDA decide sobre cuales recursos físicos de la FPGA van a ser utilizados para implementar el diseño.

Con el fin de establecer la ubicación exacta de: las ALUTs, los registros, bloques DSP, bloques de memoria, puertos E/S, y demás bloques especializados y sus interconexiones al interior chip, la herramienta EDA ejecuta complejos algoritmos de localización y ruteo (*Placement and Routing*).

En la última etapa se realiza la verificación comportamental, donde la herramienta SignalTap® permite al diseñador examinar y comprobar el comportamiento entre los módulos del sistema diseñado y del sistema total.

Figura 7: Flujo del diseño.



4. RESULTADOS OBTENIDOS

4.1 FUNCIONES A IMPLEMENTAR EN LA FPGA

En este proyecto se implementa en una FPGA un algoritmo que calcula el valor crítico c de la lista. El algoritmo en Matlab® está compuesto por las funciones: wsmCritMar(), wsm() y fRec() (ver sección 2.7). Para construir en la FPGA estas tres funciones, se establece que es necesario diseñar las siguiente cinco funciones.

- ***fRec()*:** La función $\boldsymbol{fRec}()$ se observa en la ecuación 8, es la principal función aritmética del algoritmo y realiza dos tareas para cada fila de la lista. La primera tarea es un redireccionamiento con punteros ejecutando el comando en Matlab® $x(L)$ donde la matriz L es la lista a procesar y x es una matriz que se actualiza en cada ciclo de ejecución La segunda tarea es la evaluación de la función $1/(1+t*p*q*r)$ donde t es un parámetro inicial y p,q,r son los tres elementos de cada fila de L. El producto $p*q*r$ se calcula con la función $prod()$ la cual devuelve un vector de n elementos resultantes de n productos sobre L . La instrucción $x(end+1)=1$ asigna una fila de unos a la matriz x.

Ecuación 8

$$function\ y\ =\ fRec(L,t,x)$$
$$x(end+1)\ =\ 1;$$
$$P\ =\ prod(x(L),2);$$
$$y\ =\ 1./(1+t*P);$$
$$end$$

- **Comparador 1**: La función llamada Comparador_1 se observa en la ecuación 9, y corresponde al valor máximo del valor absoluto de una resta de vectores, y su posterior comparación con el parámetro $10^{(-precWSM)}$. En la ecuación 9 los vectores a procesar son xEven y xOldEven cada uno con n filas. La función $max()$ calcula el máximo valor dentro del vector resultante de la resta. Se recalca que el valor $precWSM=12$ es el mínimo valor requerido por el algoritmo y por lo tanto, la longitud de la palabra en el formato en punto fijo

(escogido en la sección 4.2) para lograr esta resolución es de 64 bits.

$$max(|xEven - xOldEven|) > 10^{(-precWSM)} \ , \ precWSM = 12 \quad \text{Ecuación 9}$$

- **Comparador 2**: La función Comparador_2 se observa en la ecuación 10, corresponde a una resta, su valor absoluto, la comparación con el parámetro $10^{(-precWSM+4)}$. En la ecuación 10 los vectores a procesar son xEven y *fRec()* cada uno con n filas.

$$|xEven - fRec(L, t, xEven)| < 10^{(-precWSM+4)} \ , \ precWSM = 12 \quad \text{Ecuación 10}$$

- **Comparador 4**: La función Comparador_4 se observa en la ecuación 11, es el más simple porque no es necesario ejecutar ninguna norma, ni evaluar ninguna función, corresponde a una resta y la comparación con el parámetro $1.1 * 10^{-prec}$. En la ecuación 11 los valores a procesar tr y tl son números con el formato definido.

$$tr - tl > 1.1 * 10^{-prec} \ , \ prec = 4 \quad \text{Ecuación 11}$$

- **Comparador 3**: La función Comparador_3 (también llamada Promedio) se observa en la ecuación 12 y realiza un promedio de dos puntos tr y tl. Los factores exponenciales y la función $round()$ de la ecuación 12 definen las cifras significativas del promedio en base 10. Si se remueven los factores exponenciales y la función $round()$ así como se observa en la ecuación 13 entonces el resultado siempre se mostrará con el máximo número de cifras significativas del formato punto fijo escogido con 64 bits. La expresión que se implementa en hardware es la ecuación 13 evitando el diseño de unidades exponenciales que ocupan más recursos.

$$round\left(10^{-prec} * \frac{tr+tl}{2}\right) * 10^{-prec}, \ \ prec = 4 \quad \text{Ecuación 12}$$

$$\frac{tr+tl}{2} \quad \text{Ecuación 13}$$

4.2 FORMATO DE DATOS

Para establecer el formato de representación numérica de los datos a procesar por las unidades hardware, se debe considerar: los datos pertenecen al rango $0 \leq x \geq 4$ y la resolución que permita mayor precisión en los cálculos. Se descarta el formato IEEE-754 punto flotante binario y

decimal, con el objetivo de obtener mayor aceleración de ejecución y menor área del sistema hardware.

Se establece un formato punto fijo sin signo de 64 bits. El campo más significativo de tres bits corresponde a la parte entera y representa los cinco valores enteros 0, 1, 2, 3, 4 siendo el valor máximo $x = 4$. El campo menos significativo de 61 bits corresponde a la parte decimal que permite obtener la resolución adecuada. En la tabla 3 se presentan algunos ejemplos para el formato.

Tabla 3: Ejemplos del formato punto fijo sin signo de 64 bits

Binario	Decimal	Binario	Decimal	Binario	Decimal
000.0000 … 000	0	000.0000 … 000	0	000.0000 … 000	0
001.0000 … 000	1	000.0010 … 000	0.125	000.0000 … 001	4.336808689942017736029811203479 8e-19
010.0000 … 000	2	000.0100 … 000	0.250	000.0000 … 010	8.673617379884035472059622406959 5e-19
011.0000 … 000	3	000.0110 … 000	0.375	000.0000 … 011	1.301042606982605320808943361043 9e-18
100.0000 … 000	4	000.1000 … 000	0.500	000.0000 … 100	1.734723475976807094411924481391 9e-18
101.0000 … 000	X	000.1010 … 000	0.625	000.0000 … 101	2.168404344971008868014905601739 9e-18
110.0000 … 000	X	000.1100 … 000	0.750	000.0000 … 110	2.602085213965210641617886722087 9e-18
111.0000 … 000	X	000.1110 … 000	0.875	000.0000 … 111	3.035766082959412415220867842435 8e-18

El formato tiene las siguientes características:

- **Resolución:** diferencia entre dos números consecutivos. Se establece por medio bit menos significativo y corresponde a: $2^{-61} = 4.336808689942017736029811203479 8 * 10^{-19}$

- **Rango:** $0.0000000000000000000 \leq x \leq 4.0000000000000000000$ es el intervalo de representación.

- **Valor numérico**: se obtiene con la ecuación 14.

$$\frac{Valor\ entero\ de\ los\ 64\ bits}{2^{61}} = valor\ numérico\ del\ formato \qquad \text{Ecuación 14}$$

4.3 CARACTERÍSTICAS DE LAS LISTAS

Cada lista tiene estructura de matriz con n filas x 3 columnas y están compuestas por números naturales que representan índices o direcciones. Ejemplos de listas se observan en la tabla 4, donde 4a es la lista L4 y 4b es la lista L6.

Tabla 4: Ejemplo de listas a) Lista L4 b) Lista L6

L4		
3	4	5
1	3	4
1	4	5
2	2	4

a)

L6		
7	19	19
7	13	19
7	14	19
7	15	19
1	7	15
2	7	15
3	16	19
3	19	19
3	17	19
3	8	15
3	15	19
3	9	15
10	18	19
4	10	18
5	10	18
5	11	18
5	18	19
6	12	18

b)

A partir de las tablas 4 se establece: que el parámetro n es cuatro para L4 y dieciocho para L6, los tres valores de cada fila conforman las tres columnas de las listas, los valores de los números naturales que representan índices o direcciones para la lista L4 están en el rango {1,2, 3,4} y para lista L6 están en el rango dentro del intervalo [1,19].

Se establecen algunos criterios importantes para el procesamiento de estas listas:

1. En el caso de tener listas con elementos nulos, estos se reemplazan por el valor 1 con el fin de no anular la multiplicación en $fRec()$.
2. Para ejecutar la función $fRec()$, el número de filas en la lista es igual al número de veces que se debe procesar la función $y_i(x)$ de la forma $\frac{1}{1 + t*\Pi x_j}$.
3. El número de filas aumenta de una lista a otra.
4. El número de columnas de todas las listas es siempre tres.
5. Los valores de los números naturales que representan índices o direcciones dentro de las listas se encuentran en un rango entre [1, n+1].

4.4 PARÁMETROS DEL DISEÑO

Analizando la función $fRec()$ en el código Matlab® se puede observar que realiza dos tareas para cada fila de la lista. La primera tarea es una operación basada en punteros y puede verse como una función que recibe como argumento la lista L y devuelve una matriz del mismo tamaño de la lista con los valores comprendidos dentro del vector x en las direcciones dadas por la matriz L, su diseño se implementa con significativa complejidad en el datapath y en el control.

Con el fin de lograr esta operación se propone, como concepto inicial, la utilización de tres memorias denominadas para este proyecto ROM, RAM_X y RAM_L. La tabla 5 muestra los valores de los parámetros para sintetizar una lista determinada. Es muy importante recalcar que, para facilitar el manejo de los índices, las listas de 3 filas x n columnas, se reorganizan en las memorias como vectores de $3 * n$ filas x 1 columna.

Tabla 5: Parámetros del diseño

Lista	rango		filas	ROM			RAM_X			RAM_L		
	min	max	n	# words	addr bits	word bits	# words	addr bits	word bits	# words	addr bits	word bits
				ROM_WORDS	ROM_ADDR_BITS	ROM_Q_BITS	RAM_X_WORDS	RAM_X_ADDR_BITS	RAM_X_Q_BITS	RAM_L_WORDS	RAM_L_ADDR_BITS	RAM_L_Q_BITS
L4	1	5	4	12	4	3	5	3	64	12	4	64
L6	1	19	18	54	6	5	19	5	64	54	6	64
L8	1	53	52	156	8	6	53	6	64	156	8	64
L10	1	163	162	486	9	8	163	8	64	486	9	64
L12	1	601	600	1800	11	10	601	10	64	1800	11	64
L14	1	2372	2371	7113	13	12	2372	12	64	7113	13	64
L16	1	10018	10017	30051	15	14	10018	14	64	30051	15	64
L18	1	44476	44475	133425	18	16	44476	16	64	133425	18	64
L20	1	206690	206686	620058	20	18	2E+05	18	64	620058	20	64
en la lista L20 no se cumple => # de filas = max(L) - 1												

La descripción VHDL realizada es altamente parametrizada, donde los parámetros de ajuste directo por el diseñador son: n es el número de filas de la matriz original, w es el ancho del formato numérico establecido, ROM_ADDR_BITS es el ancho del vector de direccionamiento y ROM_Q_BITS es el ancho del dato almacenado en la memoria ROM.

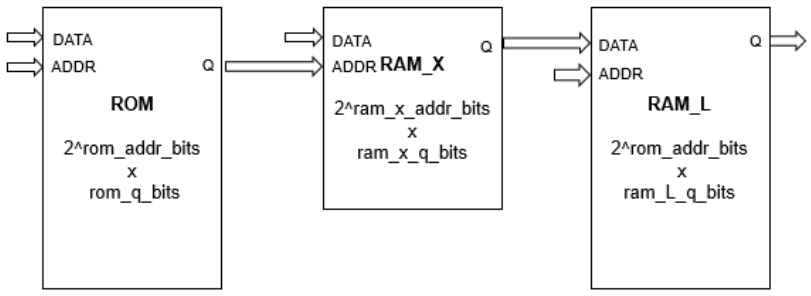
En la memoria ROM el ancho del vector de direccionamiento (ROM_ADDR_BITS) debe cumplir que $2^{ROM\_ADDR\_BITS} \geq 3 * n$. El ancho del dato almacenado (ROM_Q_BITS) debe cumplir que $2^{ROM\_Q\_BITS} \geq n$. Se debe observar en la tabla 5 que el ancho del dato almacenado en ROM (ROM_Q_BITS) es igual al ancho del vector de direccionamiento de la memoria RAM_X (RAM_X_ADDR_BITS). Se debe anotar que el ancho del vector de direccionamiento en las memorias ROM y RAM_L son iguales, debido a que almacenan matrices del mismo tamaño.

Los parámetros derivados se calculan a partir de los parámetros de ajuste directo y corresponden en la implementación de la función $fRec()$ a la cantidad de palabras de las memorias ROM y RAM_L $(3 * n)$ y la cantidad de palabras de la memoria RAM_X $(n + 1)$.

Para entender mejor la tabla 5 se explica la implementación de la operación *x(L)* en la FPGA usando tres memorias. En la figura 8 se muestran las memorias ROM, RAM_X, RAM_L. El componente ROM es cargado inicialmente con la lista, dichos valores contenidos en ROM son las direcciones de la memoria RAM_X. Se realiza la operación *x(L)* cuyo resultado es una matriz que se almacena en una tercera memoria denominada RAM_L. (RAM_L es del mismo tamaño de ROM).

La figura 8 muestra cómo se conectan las memorias para ejecutar la operación *x(L)*, se recalca que la salida Q de la memoria ROM es del mismo tamaño del puerto de direcciones de la memoria RAM_X y que la salida Q de la memoria RAM_X debe ser del mismo tamaño que el puerto de entrada de datos de la memoria RAM_L. Haciendo un barrido de toda la memoria ROM y RAM_L (debido a que son del mismo tamaño) se logra implementar la operación *x(L)*.

Figura 8: Configuración de memorias para realizar *x(L)*



Esta arquitectura diseñada para realizar la operación de punteros es también válida al reemplazar las memorias por registros paralelos, sin embargo, este remplazo modifica la unidad de control y genera diferentes resultados en la cantidad de recursos utilizados de la FPGA.

En el paquete VHDL PARAMETROS.vhd se observan: Los parámetros directos e indirectos establecido, los tipos y subtipos de datos parametrizables creados y el procedimiento *fill_array()* utilizado en el controlador_3 en el estado 3.

```
package PARAMETROS is

constant CLK_FREQ: integer:= 100000000;
-- 100 MHz = 10 ns
constant WORD_LENGTH    : integer  := 64;
constant ROWS           : integer  := 4;
constant ROM_WORDS      : integer:= ROWS*3;
constant ROM_ADDR_BITS: integer := 4;
constant ROM_Q_BITS     : integer := 3;
constant RAM_WORDS      : integer := ROWS+1;
constant RAM_ADDR_BITS: integer :=
ROM_Q_BITS;
constant RAM_Q_DATA_BITS : integer :=
WORD_LENGTH;
constant RAML_WORDS      : integer :=
ROWS*3;
constant RAML_ADDR_BITS   : integer :=
ROM_ADDR_BITS;
constant RAML_Q_DATA_BITS : integer :=
WORD_LENGTH;
constant DIEZ_A_LA_PREC_SIGNAL_COMP_1:
std_logic_vector(WORD_LENGTH-1 downto 0) :=
X"0000000000232F33";  --  = 10^(-12) =
0.000000000001
constant DIEZ_A_LA_PREC_SIGNAL_COMP_2 :
std_logic_vector(WORD_LENGTH-1 downto 0) :=
X"000000055E63B88C";  --  = 10^(-12+4) =
10^(-8) = 0.00000000001
--constant DIEZ_A_LA_PREC_SIGNAL_COMP_4 :
std_logic_vector(WORD_LENGTH-1 downto 0) :=
X"0000E6AFCCE1C388";  -- 1.1*10^(-4) =
0.00011
constant DIEZ_A_LA_PREC_SIGNAL_COMP_4 :
std_logic_vector(WORD_LENGTH-1 downto 0) :=
X"00000005E7D417CD";  -- 1.1*10^(-8) =
0.000000011

type tipo_entrada_ROM_paralelo is array
((ROWS*3)-1 downto 0) of
std_logic_vector(ROM_Q_BITS-1 downto 0);
type tipo_salida_ROM_paralelo  is array
((ROWS*3)-1 downto 0) of
std_logic_vector(ROM_Q_BITS-1 downto 0);

type tipo_entrada_RAML_paralelo is array
((ROWS*3)-1 downto 0) of
std_logic_vector(RAML_Q_DATA_BITS-1 downto
0);
type tipo_salida_RAML_paralelo  is array
((ROWS*3)-1 downto 0) of
std_logic_vector(RAML_Q_DATA_BITS-1 downto
0);

type tipo_entrada_RAMX_paralelo is array
(ROWS-1 downto 0) of
std_logic_vector(RAM_Q_DATA_BITS-1 downto
0);
type tipo_salida_RAMX_paralelo  is array (ROWS-
1 downto 0) of
std_logic_vector(RAM_Q_DATA_BITS-1 downto
0);

type tipo_entrada_RAMX_paralelo_mas1 is array
(ROWS downto 0) of
std_logic_vector(RAM_Q_DATA_BITS-1 downto
0);
type tipo_salida_RAMX_paralelo_mas1  is array
(ROWS downto 0) of
std_logic_vector(RAM_Q_DATA_BITS-1 downto
0);

subtype tipo_addr_ROM is
std_logic_vector(ROM_ADDR_BITS-1 downto 0);
subtype tipo_q_ROM    is
std_logic_vector(ROM_Q_BITS-1 downto 0);

subtype tipo_addr_RAML is
std_logic_vector(RAML_ADDR_BITS-1 downto 0);
subtype tipo_data_RAML is
std_logic_vector(RAML_Q_DATA_BITS-1 downto
0);
subtype tipo_q_RAML    is
std_logic_vector(RAML_Q_DATA_BITS-1 downto
0);

subtype tipo_addr_RAMX is
std_logic_vector(RAM_ADDR_BITS-1 downto 0);
subtype tipo_data_RAMX is
std_logic_vector(RAM_Q_DATA_BITS-1 downto
0);
subtype tipo_q_RAMX    is
std_logic_vector(RAM_Q_DATA_BITS-1 downto
0);
```

```
type tipo_comparador2 is array (ROWS-1 downto
0) of std_logic;

procedure fill_array(
signal RowIndex      : in integer;
signal myVector      : in
std_logic_vector(WORD_LENGTH-1 downto 0);
signal myArrayQ      : in
tipo_entrada_RAMX_paralelo;
signal myArrayData : out
tipo_entrada_RAMX_paralelo
);

end package;

package body PARAMETROS is

procedure fill_array(
signal RowIndex      : in integer;
signal myVector      : in
std_logic_vector(WORD_LENGTH-1 downto 0);
signal myArrayQ      : in
tipo_entrada_RAMX_paralelo;
signal myArrayData  : out
tipo_entrada_RAMX_paralelo
)is
variable myArrayDataVariable :
tipo_entrada_RAMX_paralelo;
begin
for i in 0 to ROWS-1 loop
if i = ROWIndex then
myArrayDataVariable(i)   := myVector;
else
myArrayDataVariable(i) := myArrayQ(i);
end if;
end loop;
myArrayData <= myArrayDataVariable;
end procedure;

end procedure;

end PARAMETROS;
```

4.5 ARQUITECTURA GENERAL DEL SISTEMA

Con el fin de proponer una arquitectura general del sistema se considera el algoritmo de la sección 2.7, donde se describen tres funciones: la función principal *wsmCritMar()*, la cual hace el llamado a la función *wsm(),* la que a su vez hace el llamado a la función *fRec()*.

El esquema general propuesto para la arquitectura del sistema se observa en la figura 9 que corresponde a la unidad hardware de mayor jerarquía UNIDAD_WSM_CRIT_MAR y realiza las mismas tareas de la función *wsmCritMar()*. En esta arquitectura del sistema también se observan dos entidades de menor jerarquía, UNIDAD_WSM y UNIDAD_FREC. Estas entidades realizan las mismas tareas de la funciones *wsm()* y *fRec()* respectivamente.

También se observa en la figura 9, que cada una de las tres entidades están compuestas de un datapath y unidad de control, siendo la entidad UNIDAD_WSM parte del Datapath_1 y la entidad UNIDAD_FREC parte del Datapath_2. En cuanto a las unidades de control, el controlador principal es el control_1 que además de controlar el Datapath_1, gobierna la unidad Control_2 y esta a su vez gobierna el control_3.

La descripción de la UNIDAD_WSM_CRIT_MAR es un modelo estructural VHDL, siendo la entidad de mayor jerarquía, en la figura 10 se observan sus componentes y en la figura 11 el detalle de interconexión de dicha entidad.

Figura 9: Esquema general del sistema

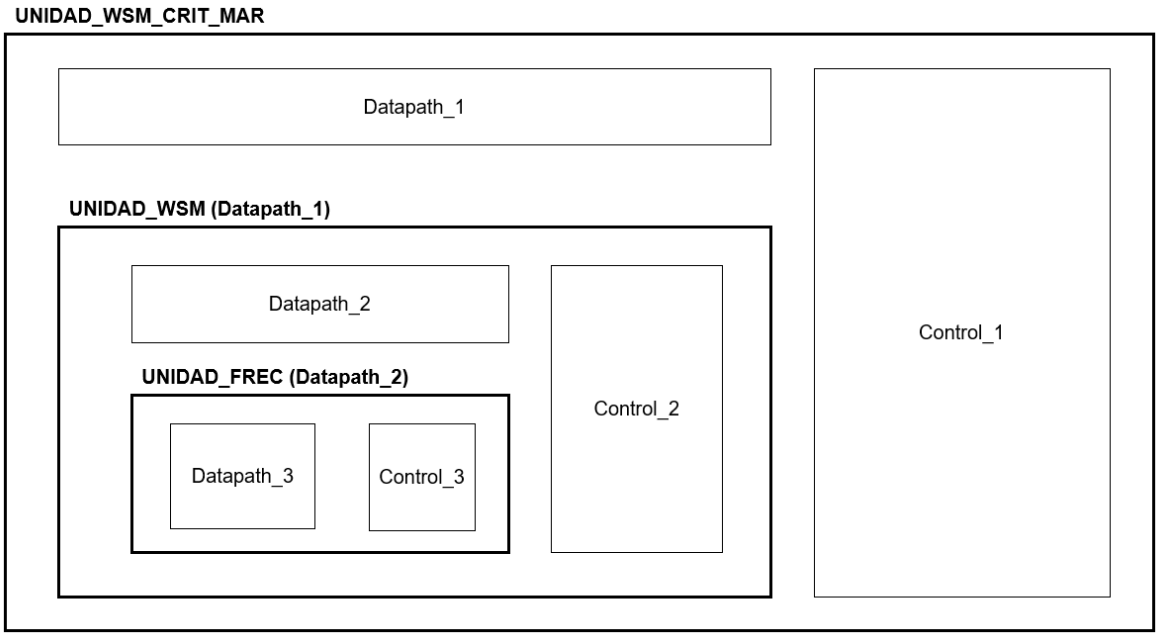


Figura 10: Diagrama de componentes de UNIDAD_WSM_CRIT_MAR

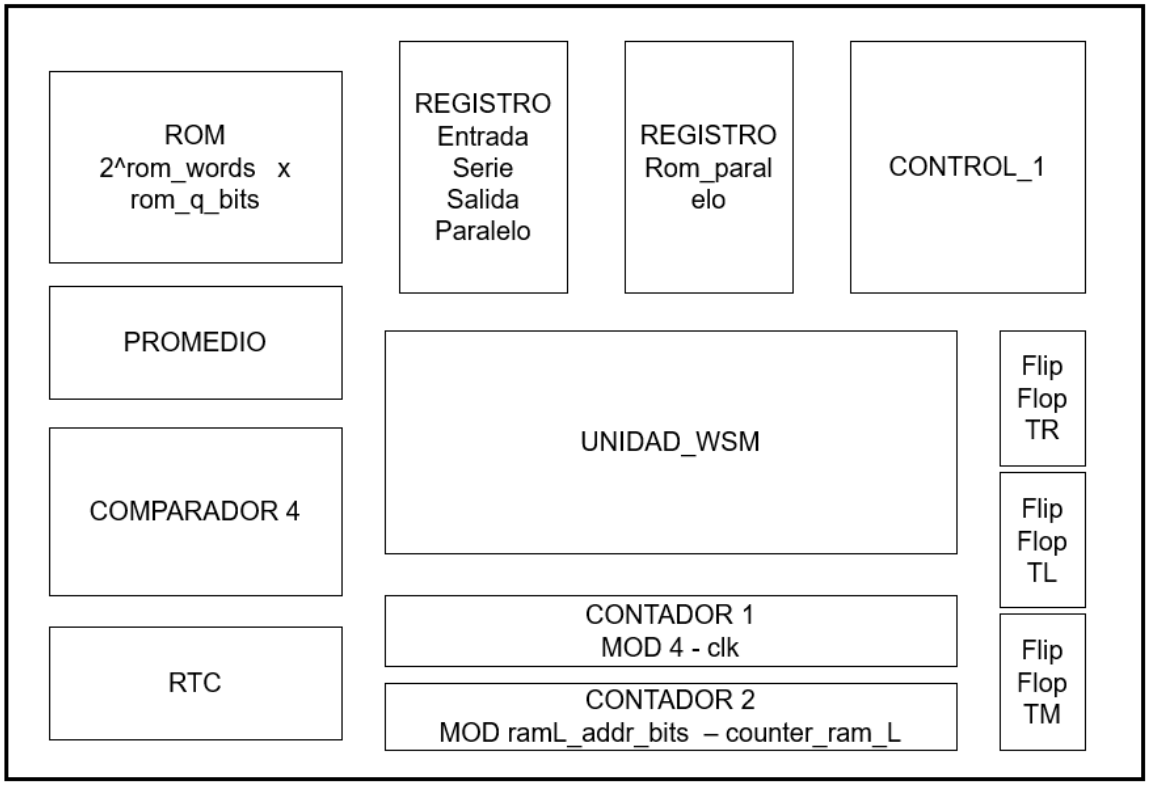
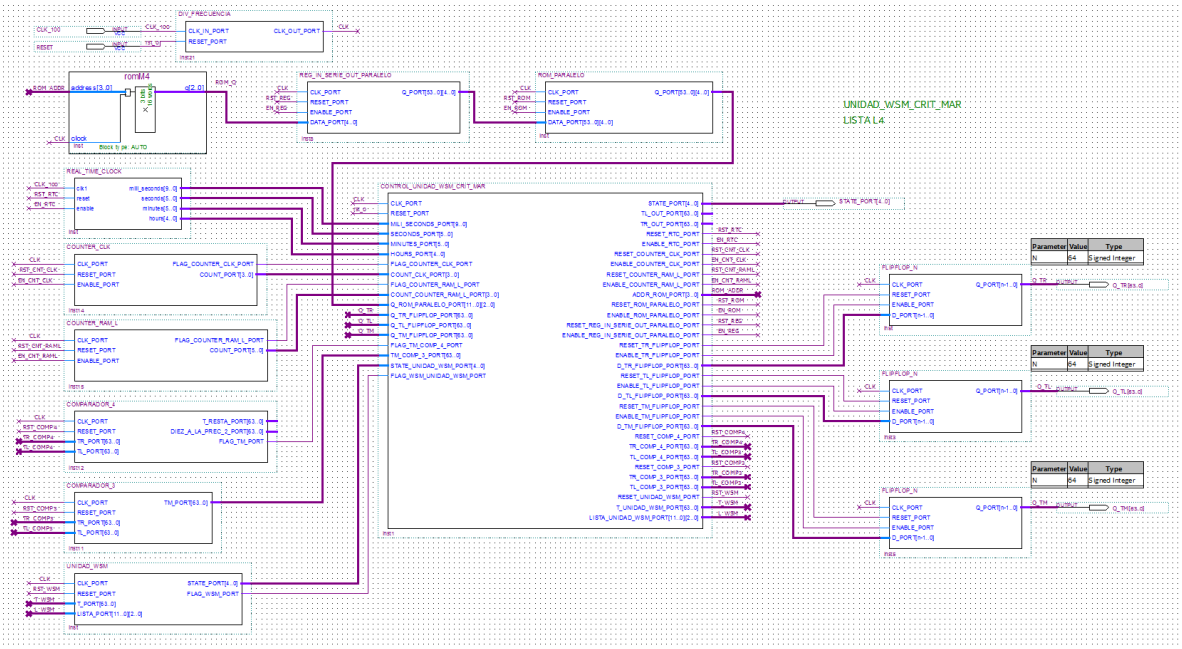


Figura 11: Diagrama de componentes de UNIDAD_WSM_CRIT_MAR (detalle)



Como se puede observar en la figura 11 la entidad UNIDAD_WSM_CRIT_MAR no recibe los valores iniciales por puertos de entrada, esos valores son parte del diseño hardware. Los valores iniciales son: la lista almacenada en la memoria ROM, los valores iniciales de tr y tl que son asignados en los primeros estados de la FSM y el valor inicial de $prec$ que es una constante al interior de la unidad COMPARADOR_4.

En la sección 4.6 se describe de manera global el flujo de datos del algoritmo en la FPGA. A partir de la sección 4.7 se especifica cada unidad de hardware implementada.

4.6 FUNCIONAMIENTO DEL COMPONENTE UNIDAD_WSM_CRIT_MAR

1. En primer lugar, se debe seleccionar la lista a procesar y se deben asignar los correspondientes parámetros de ajuste directo dados en la tabla 5 para que el diseño sea sintetizable.

2. Se debe tener la lista cargada en un archivo de inicialización de formato *.mif*.

3. Cuando se programa la tarjeta se carga automáticamente la lista en una memoria ROM y se inicializa los valores $tr = 4$ y $tl = 2$ en los primeros estados del controlador 1.

4. Con el fin de tener un manejo paralelo de la lista, se extrae la información almacenada en el bloque ROM para ser cargada en un registro denominado ROM_PARALELO, y esto se realiza por medio de otro registro llamado REG_IN_SERIE_OUT_PARALELO de un único puerto de entrada serial con un puerto de salida paralelo, donde su salida está conectada al registro paralelo ROM_PARALELO. De esta manera ya se tiene acceso a la lista en un registro y no en una memoria.

5. El algoritmo comienza cuando se cargan los valores de los registros tr y tl en la unidad COMPARADOR_4. Esta unidad se encarga de realizar una comparación y su resultado, positivo o negativo, está dado por una señal de sincronización llamada FLAG_TM_PORT que en relación al software da funcionalidad al bucle *while()* de Matlab®.

6. Si la comparación anterior es positiva o lo que es lo mismo FLAG_TM_PORT=1 entonces se cargan los registros tr y tl en la unidad PROMEDIO, la cual realiza un promedio entre tr y tl, y se carga el resultado en un tercer registro denominado tm.

7. Luego se carga la lista L y el registro TM en UNIDAD_WSM. Esta unidad retorna una señal de sincronización denominada FLAG_WSM_PORT, la cual permite un control de transición de estados para lograr una forma de implementar la instrucción de control *if()* de Matlab®.

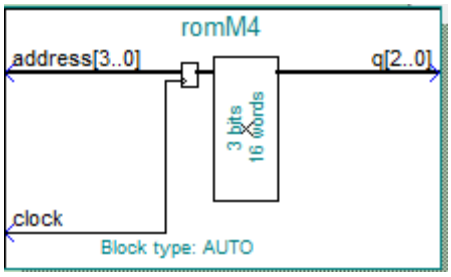
Esta es la descripción inicial del flujo de datos en hardware propuesto para la arquitectura general. A continuación, se detallará cada unidad de hardware en secciones independientes detallando sus especificaciones y realizando pruebas de funcionamiento.

4.7 MEMORIA

Para tener acceso a la información de las listas, se guardan éstas en una memoria ROM por medio de un archivo de inicialización (formato .mif – *memory initialization file*). Por medio de este archivo de inicialización se carga la memoria al realizar la programación de la tarjeta o al resetear la memoria. Se trató de cargar las listas dentro de alguna entidad directamente desde un archivo de texto, pero la herramienta EDA tiene prohibida la síntesis de comandos de lectura y escritura de archivos de texto, solamente lo permite para simulación. Como las listas son tan grandes se necesita una forma "fácil" o automática para lograr cargar las listas en señales, por este motivo se implementó la solución de cargar las listas a un componente ROM por medio de un archivo de inicialización, ya

que era fácil copiar y pegar las listas sin necesidad de realizar ninguna edición. La figura 12 muestra el tamaño del puerto de direccionamiento y el puerto de salida que corresponden con los valores de la tabla 6 de parámetros generales para la síntesis de la lista L4.

Figura 12: Memoria ROM. En este caso la lista L4



Como se observa en la figura 13, la lista en la memoria ROM se organiza como un vector columna para facilitar el manejo de índices, con lo que se definen las cantidades de bits que tendrán el bus de direcciones y el bus de salida de la memoria. Estos valores son parámetros definidos y diferentes para cada lista.

Figura 13: a) Lista L4. b) Archivo de inicialización con la lista L4.

L4		
3	4	5
1	3	4
1	4	5
2	2	4

a)

Addr	+0	ASCII
0	3	.
1	4	.
2	5	.
3	1	.
4	3	.
5	4	.
6	1	.
7	4	.
8	5	.
9	2	.
10	2	.
11	4	.

b)

4.8 REGISTROS DE ALMACENAMIENTO

Los registros de propósito específico necesarios para la implementación del algoritmo se diseñaron en virtud de la facilidad y flexibilidad del diseño. Los nombres asignados a estos registros como ROM, RAM_X o ROM_L no son porque sean memorias, sino porque el comportamiento del registro debe ser de tipo RAM o ROM. Por ejemplo, el registro RAM_L es escrito y leído muchas veces durante la ejecución del algoritmo. Además, RAM_L tiene el mismo tamaño de la matriz L. Para el caso del registro ROM este almacena las listas y no debe ser modificado.

En general los nombres asignados a componentes, parámetros o a tipos de datos relacionados con los nombres ROM y RAM_L se pueden asociar con la dimensión de los registros ROM y RAM_L que es de $3 * n \ \ x \ \ 64\, bits$. El mismo tipo de relación se puede hacer para RAM_X cuya dimensión es de $n \ \ x \ \ 64\, bits$.

El código VHDL para implementar estos registros se muestra en el anexo 1 y se basa en modelamiento estructural teniendo como componentes Flip Flops tipo D con reset asíncrono. Los resultados de síntesis de los diferentes registros diseñados se presentan a continuación.

a) Registros de tamaño n x 64 bits

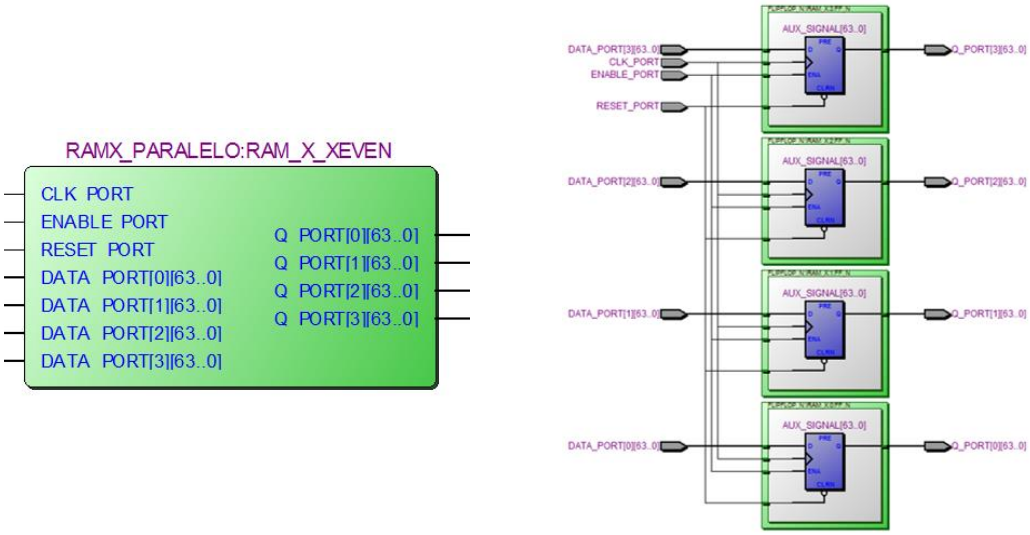
Los registros de tamaño $n \ x \ 64\, bits$, vistos en la figura 14, se relacionan con los vectores columna implementados en el algoritmo llamados X_EVEN, X_OLD_EVEN y X_ODD. Estos componentes sirven para almacenar los resultados de la unidad aritmética principal (UNIDAD_FREC) y valores de las señales con los mismos nombres.

Este registro es codificado instanciando cada flip flop y conectando sus entradas y salidas a señales de tipo *array [n-1 downto 0]* de vectores tipo *standard_logic_vector (63 downto 0)*, como se muestra en el anexo 1.

b) Registros de tamaño 3*n x 64 bits

Este registro sirve para guardar una matriz del mismo tamaño de las listas, almacenando con valores en formato de punto fijo establecido. Este registro es codificado instanciando cada flip flop y conectando sus entradas y salidas a señales de tipo *array [3*n-1 downto 0]* de vectores tipo *standard_logic_vector (63 downto 0)*.

Figura 14: Registros n x 64 bits (n=4=Lista L4)



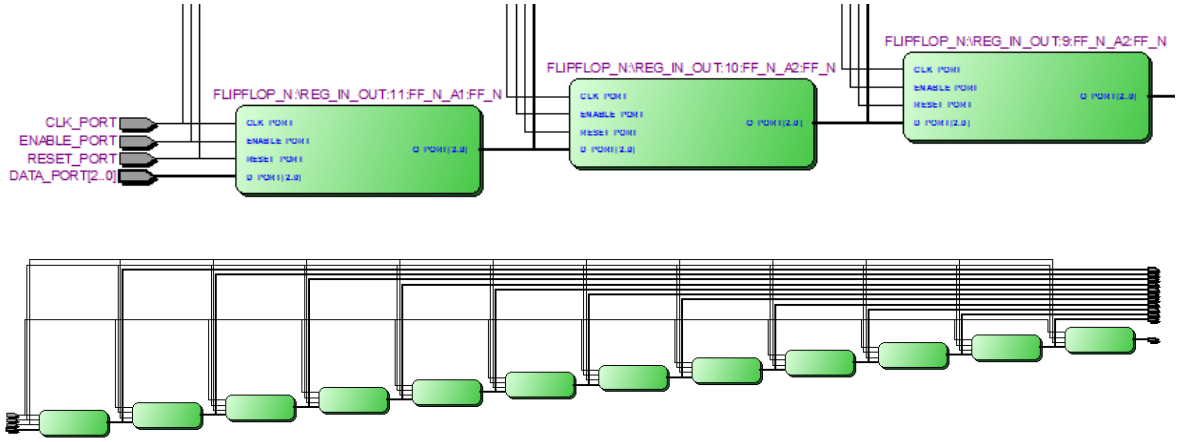
c) Registros de tamaño n+1 x 64 bits

Estos componentes serán utilizados con el fin de ejecutar la sentencia de Matlab® codificada como $x(end+1)=1$. Este comando adiciona una fila extra al final del vector x que tiene n componentes y se almacena el valor 1 en dicha fila. Este registro es codificado instanciando cada flip flop y conectando sus entradas y salidas a señales de tipo *array [(n+1)-1 downto 0]* de vectores tipo *standard_logic_vector (63 downto 0)*.

d) Registro Entrada Serie - Salida Paralela

Esta unidad de hardware llamada REG_IN_SERIE_OUT_PARALELO es un registro donde la entrada es tipo serie y la salida es paralela. Su código VHDL se puede observar en el anexo 2. Este registro se diseñó para facilitar la lectura de los datos guardados inicialmente en el archivo de inicialización de la memoria ROM hacia un registro paralelo que permite el acceso a la lista de manera paralela. Esta unidad sólo se utiliza una vez en cada ejecución del algoritmo. Este registro tiene $3*n$ registros de salida (ver figura 15).

Figura 15: Registro entrada serie – salida paralela

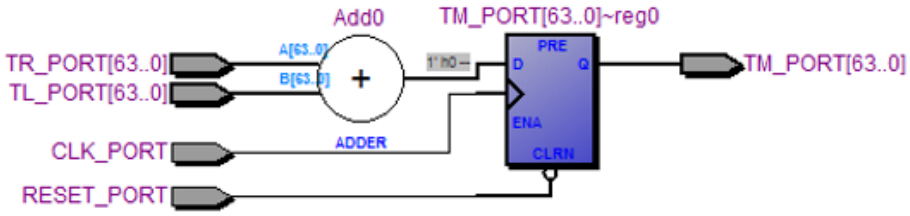


Puede interferirse que esta unidad tiene una alta latencia (=3*n), sin embargo, como se utiliza una única vez en cada procesamiento de una lista entonces no tendrá mayor efecto en el tiempo de ejecución del algoritmo. Se modela estructuralmente con ayuda de ciclos *for-generate* e *if-generate* para lograr la conexión serial flip flop por flip flop.

4.9 PROMEDIO

La unidad denominada PROMEDIO o en ocasiones Comparador_3 (figura 16), realiza la operación promedio entre dos números (ver ecuación 12). Esta operación se realiza con un sumador del mismo número de bits que el tamaño de la palabra (64) y la división por dos se implementa con un desplazamiento lógico a la derecha. En el algoritmo en Matlab® la función correspondiente a esta unidad, tiene algunos factores exponenciales y una función de redondeo para alterar la precisión del resultado, pero en este caso se ejecuta el algoritmo en otro hardware con formato personalizado y se basa únicamente en el tamaño de la palabra gracias al formato establecido. Esto no afecta la precisión de los resultados. Se concluye que el efecto de los factores exponenciales y la función de redondeo es más útil en el formato IEEE-754 debido a su precisión variable y su enorme rango logarítmico.

Figura 16: Hardware de la unidad Promedio



La descripción de la unidad PROMEDIO corresponde a un modelo comportamental implementado con dos procesos VHDL. El primer proceso se activa por los cambios en las señales de entrada TR_PORT o TL_PORT y realiza el promedio haciendo la suma y un desplazamiento lógico a la derecha para implementar eficientemente la división por dos. El segundo proceso VHDL almacena el resultado anterior en un registro. Se omite su prueba de funcionamiento por su simplicidad.

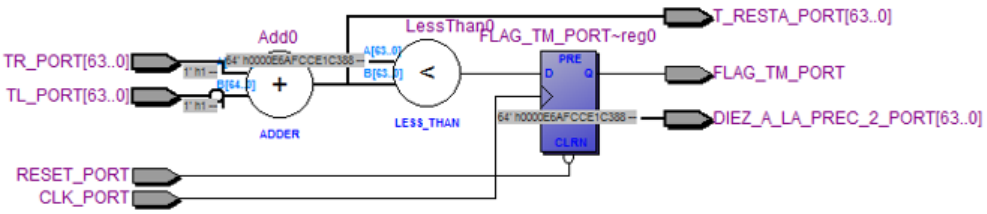
```
entity COMPARADOR_3 is
port(
CLK_PORT          : in std_logic;
RESET_PORT        : in std_logic;
TR_PORT           : in
std_logic_vector(WORD_LENGTH-1 downto 0);
TL_PORT           : in
std_logic_vector(WORD_LENGTH-1 downto 0);
TM_PORT           : out
std_logic_vector(WORD_LENGTH-1 downto 0)
  );
end COMPARADOR_3;

architecture BEHAVIORAL of COMPARADOR_3 is
signaL AUX_TM_2_SIGNAL :
std_logic_vector(WORD_LENGTH-1 downto 0) :=
(others=>'0');
signal AUX_TM_3_SIGNAL :
std_logic_vector(WORD_LENGTH-1 downto 0) :=
(others=>'0');
begin
P1: process(TR_PORT, TL_PORT)
begin
AUX_TM_2_SIGNAL <= (TR_PORT + TL_PORT);
AUX_TM_3_SIGNAL <=
('0'&(AUX_TM_2_SIGNAL(AUX_TM_2_SIGNAL'LEF
T downto 1)));
end process;

P2: process(RESET_PORT, CLK_PORT,
AUX_TM_3_SIGNAL)
begin
if(RESET_PORT = '0') then
TM_PORT <= (others => '0');
elsif(CLK_PORT'event and CLK_PORT = '1') THEN
TM_PORT <= AUX_TM_3_SIGNAL;
end if;
end process;

end BEHAVIORAL;
```

4.10 COMPARADOR 4

La unidad de hardware denominada Comparador_4, dada en la figura 17, realiza la comparación (ver ecuación 13) entre el resultado de una substracción de dos números estándar y un valor constante igual a $1.1 * 10^{-4} = 0.00011$. Todos estos tres números representados en el formato estándar de datos utilizado. Esta unidad almacena el resultado de la comparación en un flip flop para ser usado como bandera por la unidad de mayor jerarquía.

En relación al hardware exclusivamente, se debe recalcar que la estabilidad en las señales de entrada de las entidades que no tienen su entrada registrada se da por medio del controlador que las gobierne, gracias a que todas las entidades tienen su salida registrada y por ende cada cambio en las señales de entrada, por ejemplo, en este caso del Comparador_4 es gobernado por el cambio de estados en el contralador_1 y estabilizado por la salida registrada de donde provenga el dato que llega a Comparador_4. Si en optimizaciones posteriores del diseño o en trabajos posteriores se desea realizar pipeline, o particiones para una compilación incremental o un *floorplan*, se deben registrar todas las entradas (y salidas) de todos los circuitos combinacionales.

Figura 17: Hardware del Comparador 4



Se utiliza un modelamiento comportamental para este componente utilizando tres procesos VHDL. El primero realiza la resta, el segundo ejecuta la comparación y el tercero almacena el resultado de la comparación en un flip flop. La única señal de salida necesaria para la ejecución correcta del algoritmo es la señal de sincronización que es salida del flip flop llamada FLAG_TM_PORT, las otras dos señales de salida sirven para realizar pruebas de funcionamiento, pero no se conectan al momento de instanciar el componente.

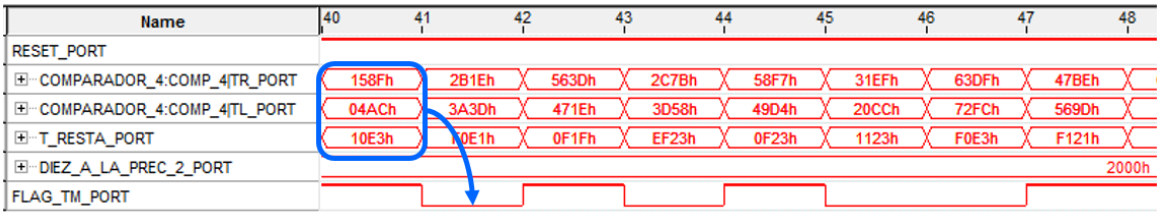
```
entity COMPARADOR_4 is
port(
CLK_PORT          : in std_logic;
RESET_PORT        : in std_logic;
TR_PORT: in std_logic_vector(WORD_LENGTH-1
downto 0);
TL_PORT: in std_logic_vector(WORD_LENGTH-1
downto 0);
T_RESTA_PORT: out
std_logic_vector(WORD_LENGTH-1 downto 0);
DIEZ_A_LA_PREC_2_PORT: out
std_logic_vector(WORD_LENGTH-1 downto 0);
FLAG_TM_PORT      : out std_logic
);
end COMPARADOR_4;

architecture BEHAVIORAL of COMPARADOR_4 is
signal TM_2_SIGNAL:
std_logic_vector(WORD_LENGTH-1 downto 0) :=
(others=>'0');
signal FLAG_TM_2_SIGNAL: std_logic := '0';
begin -- begin arch

P1: process(TR_PORT, TL_PORT, CLK_PORT)
begin
TM_2_SIGNAL <= TR_PORT - TL_PORT;
end process;
T_RESTA_PORT <= TM_2_SIGNAL;

P2: process(TM_2_SIGNAL, CLK_PORT)
```

```
begin
if (TM_2_SIGNAL >
DIEZ_A_LA_PREC_SIGNAL_COMP_4) then
FLAG_TM_2_SIGNAL <= '1';
else
FLAG_TM_2_SIGNAL <= '0';
end if;
end process NM2;
DIEZ_A_LA_PREC_2_PORT <=
DIEZ_A_LA_PREC_SIGNAL_COMP_4;
```

```
P3: process(RESET_PORT, CLK_PORT,
FLAG_TM_2_SIGNAL)
begin
if(RESET_PORT = '0') then
FLAG_TM_PORT <= '0';
elsif(CLK_PORT'event and CLK_PORT = '1') THEN
FLAG_TM_PORT <= FLAG_TM_2_SIGNAL;
end if;
end process;

end BEHAVIORAL;
```

Las pruebas para este bloque se realizaron alimentando la unidad con un generador de señales pseudo-aleatorias, donde este es sintetizado junto al propio diseño en la FPGA. La figura 18 muestra el funcionamiento de la unidad usando SignalTap®. Se observa que las pruebas se realizaron con palabras de 16 bits. Las señales TR_PORT y TL_PORT son las entradas y la señal T_RESTA es el resultado parcial de la resta entre las entradas. Para este ejemplo en particular, la señal constante es 2000h y el resultado de la comparación es un bit de señalización FLAG_TM_PORT que es igual a uno cuando T_RESTA es mayor a la constante y es cero cuando es menor. Se puede observar por ejemplo en el primer ciclo de la figura 18 que las entradas 158Fh y 04ACh generan como resultado de la resta 10E3h y este valor es menor a la constante 2000h, por lo tanto, la señal de sincronización debe ser igual a cero. Los resultados del bit de señalización FLAG_TM_PORT se obtienen un ciclo de reloj retrasado debido al uso del flip flop de salida. En el segundo ciclo se observa que el valor F0E1h en la señal T_RESTA es mayor a la constante y su resultado es uno.

Figura 18: Prueba de funcionamiento Comparador 4



4.11 RELOJ DE TIEMPO REAL (*Real Time Clock - RTC*)

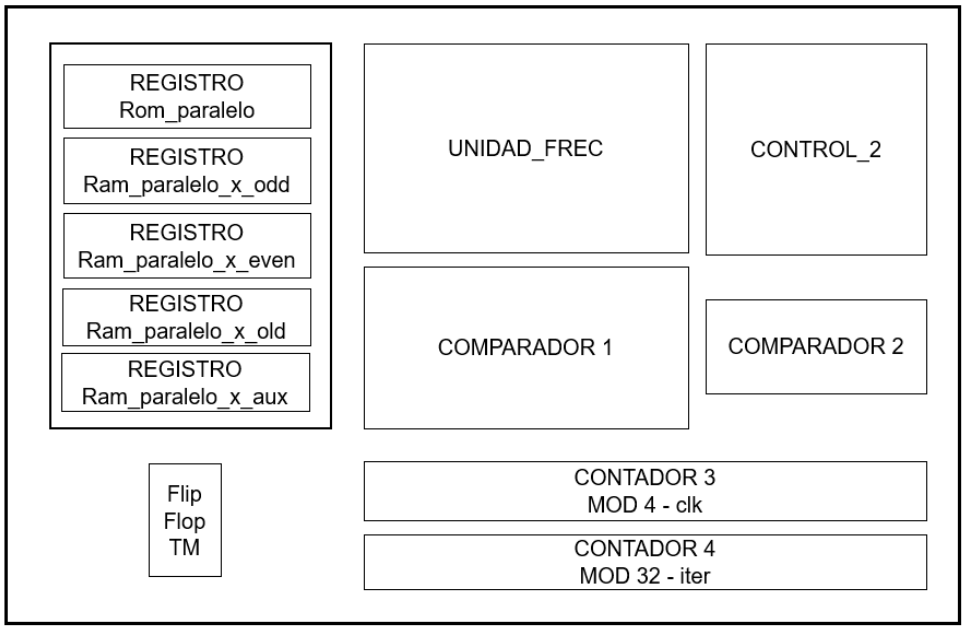
El reloj de tiempo real se implementó con el fin de obtener una medición precisa del tiempo de ejecución del algoritmo. Está basado en un contador el cual es gobernado por el reloj del sistema de 100 MHz correspondiente a 10 ns, el cual genera un reloj de 1 MHz correspondiente a 1 us. Este reloj de 1 MHz gobierna la implementación de los contadores de las unidades del RTC: microsegundos, milisegundos, segundos, minutos y horas. El RTC tiene la capacidad de contar hasta máximo 24 horas y se puede detener la cuenta y visualizarla. El siguiente código VHDL muestra los dos procesos diseñados para su implementación. Cada uno de estos son contadores, el primero trabaja con el reloj del sistema de 100 MHz y el segundo con el reloj de 1 MHz.

```
entity REAL_TIME_CLOCK is
port (
clk1            : in std_logic;
reset           : in std_logic;
enable          : in std_logic;
mili_seconds : out std_logic_vector(9 downto 0);
seconds         : out std_logic_vector(5 downto 0);
minutes         : out std_logic_vector(5 downto 0);
hours           : out std_logic_vector(4 downto 0)
);
end REAL_TIME_CLOCK;

architecture BEHAVIORAL of REAL_TIME_CLOCK is
signal micro_sec   : integer range 0 to 1000:= 0;
signal mili_sec    : integer range 0 to 1000:= 0;
signal sec,min,hour : integer range 0 to 60:= 0;
signal clk          : std_logic :='0';
begin
mili_seconds  <=  conv_std_logic_vector(mili_sec,
10);
seconds      <= conv_std_logic_vector(sec, 6);
minutes      <= conv_std_logic_vector(min, 6);
hours       <= conv_std_logic_vector(hour, 5);

process(clk1)
--constant CLK_FREQ: integer := 100000000; --
100 MHz = 10 ns
constant PULSE_FREQ: integer := 1000000; -- 1
MHz = 1 us
constant        MAX_CNT:        integer        :=
(INTEGER(CLK_FREQ/PULSE_FREQ))/2;
variable cnt : integer range 0 to MAX_CNT := 0;
begin
if (reset = '0') then
cnt := 0;
elsif(clk1'event and clk1='1') then
cnt := cnt+1;
if(cnt = MAX_CNT) then
clk <= not clk;
cnt := 0;
end if;
end if;
end process;

process(clk)
begin
if (reset = '0') then
micro_sec <= 0;
mili_sec  <= 0;
sec                <= 0;
min                <= 0;
hour               <= 0;
elsif(clk'event and clk='1') then
if enable = '1' then
micro_sec <= micro_sec + 1;
if(micro_sec = 999) then
micro_sec <= 0;
mili_sec <= mili_sec + 1 ;
if(mili_sec = 999) then
mili_sec <=0;
sec <= sec + 1;
if(sec = 59) then
sec <= 0;
min <= min + 1;
if(min = 59) then
min <= 0;
hour <= hour + 1;
if(hour = 23) then
hour <= 0;
end if;
end if;
end if;
end if;
end if;
else
micro_sec          <= micro_sec;
mili_sec           <= mili_sec;
sec                <= sec;
min                <= min;
hour               <= hour;
end if;
end if;
end process;

end BEHAVIORAL;
```

4.12 UNIDAD_WSM

El componente UNIDAD_WSM corresponde a un modelo VHDL estructural. Contiene un banco de registros de propósito específico que tienen la misma estructura que los registros explicados anteriormente. La figura 19 detalla los nombres de los componentes que pertenecen a UNIDAD_WSM. La figura 20 corresponde al detalle de UNIDAD_WSM.

Figura 19: Diagrama de componentes de UNIDAD_WSM



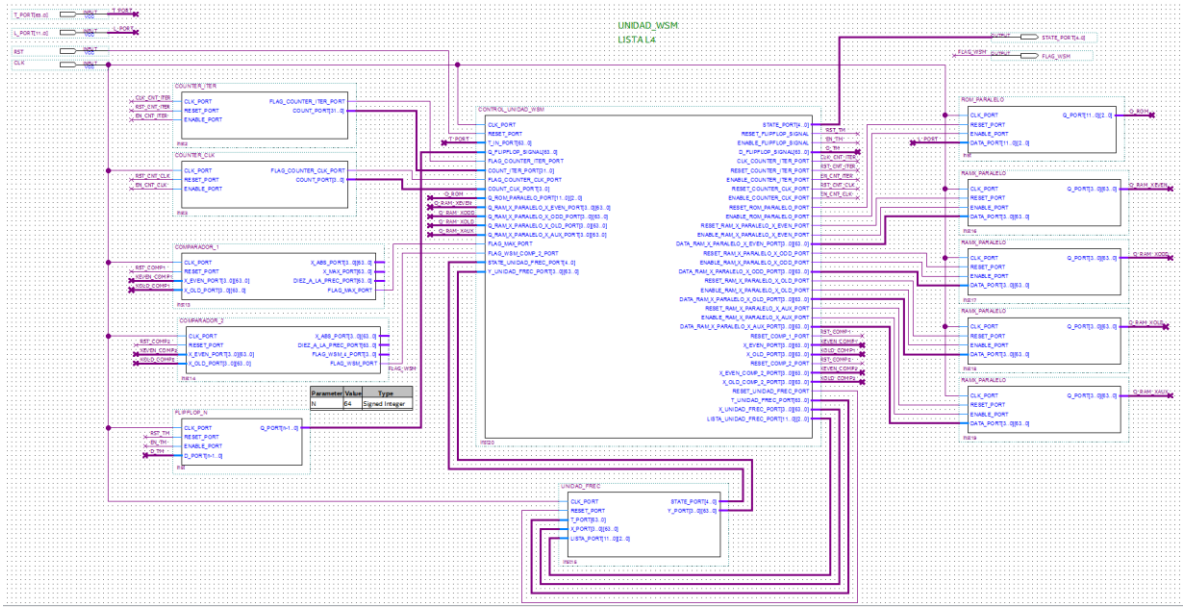
4.13 FUNCIONAMIENTO DEL COMPONENTE UNIDAD_WSM

1. Esta unidad tiene las siguientes entradas: el puerto T_PORT que es almacenado en un registro, el puerto LISTA_PORT que es almacenado en el registro ROM_PARALELO. Los puertos de salida son: el estado del controlador CONTROL_2 y la señal FLAG_WSM_PORT de sincronización que se origina en la unidad COMPARADOR_2.

2. Considerando el algoritmo en Matlab® se puede observar que al comenzar la ejecución de la función *wsm()* se define otra constante de precisión llamada PREC_WSM, la cual está implícitamente asignada por medio de una constante dentro del hardware de las unidades COMPARADOR_1 y COMPARADOR_2. La inicialización de los registros

X_EVEN={1,1,…1} y X_OLD_EVEN={0,0,…,0} se realiza en los primeros estados de la máquina de estados en CONTROL_2.

Figura 20: Diagrama de componentes de UNIDAD_WSM (detalle)



3. El paso siguiente es cargar los vectores X_EVEN y X_OLD_EVEN en el COMPARADOR_1. Esta unidad realiza una comparación y devuelve un valor binario por medio de la señal de sincronización FLAG_MAX_PORT. Esta señal permite la implementación del bucle *while* de Matlab®.

4. Si la señal de sincronización es positiva (o uno), entonces se realiza la carga de X_EVEN al registro X_OLD_EVEN, esto con el fin de guardar un valor anterior del registro X_EVEN en relación a las iteraciones dentro del bucle *while()* del comparador_1.

5. Luego se carga el componente UNIDAD_FREC con los valores de la lista L, el valor del registro TM y el valor del registro X_EVEN. La unidad devolverá un vector que se cargará en el registro X_ODD.

6. Este paso es igual al anterior con la diferencia que se cargan la lista L, el registro TM y el registro X_ODD en la UNIDAD_FREC. De igual manera la unidad devolverá un vector que se cargará en el registro X_EVEN.

7. Luego se incrementa en uno el valor del contador COUNTER_ITER que informa sobre el número de iteraciones que se presentan en la ejecución del bucle *while()*.

8. Cuando la señal de sincronización FLAG_MAX_PORT=0 entonces el flujo de datos sale del bucle de control *while()*. En relación al hardware, el paso a seguir es la carga de nuevo a la UNIDAD_FREC de la lista L, el flip flop TM y el vector X_EVEN, para que su resultado sea cargado junto el vector X_EVEN en la unidad llamada COMPARADOR_2.

9. La unidad COMPARADOR_2 realiza una nueva comparación y devuelve la bandera FLAG_WSM_PORT que se evaluará en la entidad de mayor jerarquía.

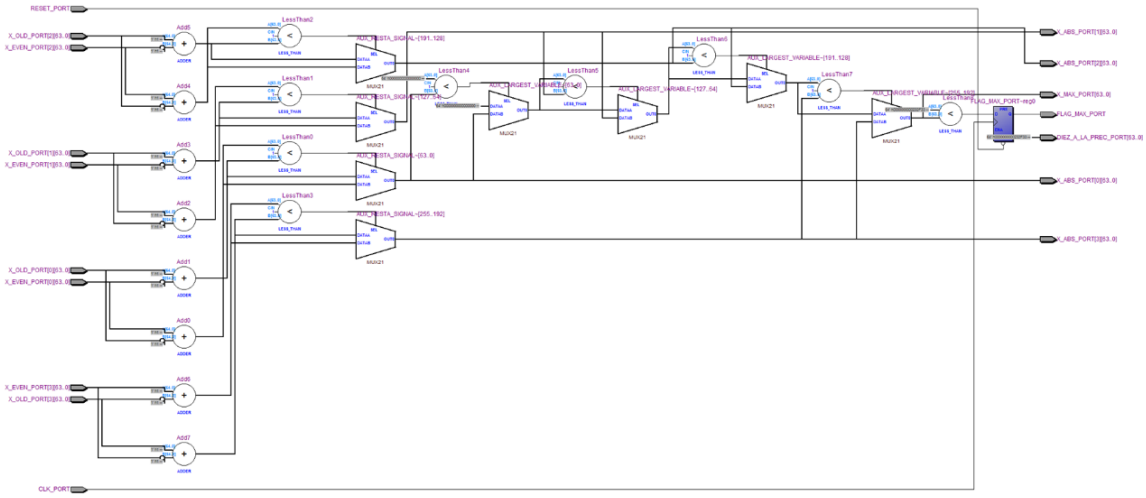
4.14 COMPARADOR 1

El bloque hardware llamado COMPARADOR_1 establece la comparación entre el máximo componente de la diferencia entre dos vectores columna y un número constante igual a 10^{-12}. El valor constante será inicializado dentro de la unidad de hardware y se guardará en un registro, el resultado de la comparación, para luego ser usado como señal de sincronización.

Esta función (ver ecuación 8) recibe dos vectores columna de los registros X_EVEN y X_OLD_EVEN que son de n componentes, se procesan con una operación de resta y se calcula el máximo componente del vector resultante. Este último valor se compara con la constante $10^{(-precWSM)}$ (10^{-12}).

La idea general del diseño del hardware de la figura 21 es que, empezando desde el primer componente del vector resultante de la substracción, esté se compare con el anterior para ir verificando si es el máximo y cuando se termine de calcular este máximo, se comparará con la constante y el resultado se guarda en el registro de la última etapa. Es un circuito combinacional. El primer proceso VHDL de esta unidad está encargado de calcular si la resta es positiva o negativa y gracias a esta distinción puede realizar correctamente la operación valor absoluto. El segundo proceso VHDL define cuál componente es el máximo comparando uno por uno con el anterior, y para realizar esto en un sólo ciclo de reloj se utiliza una variable para implementar esta asignación ya que las variables cambian sus valores cuando finaliza el tiempo delta de simulación (*delta time*) y no el tiempo de ejecución (*wall time*). El tercer proceso VHDL define la señal de sincronización binaria y ésta es almacenada en un flip flop implementado con modelamiento comportamental en el último proceso.

Figura 21: Hardware del comparador 1

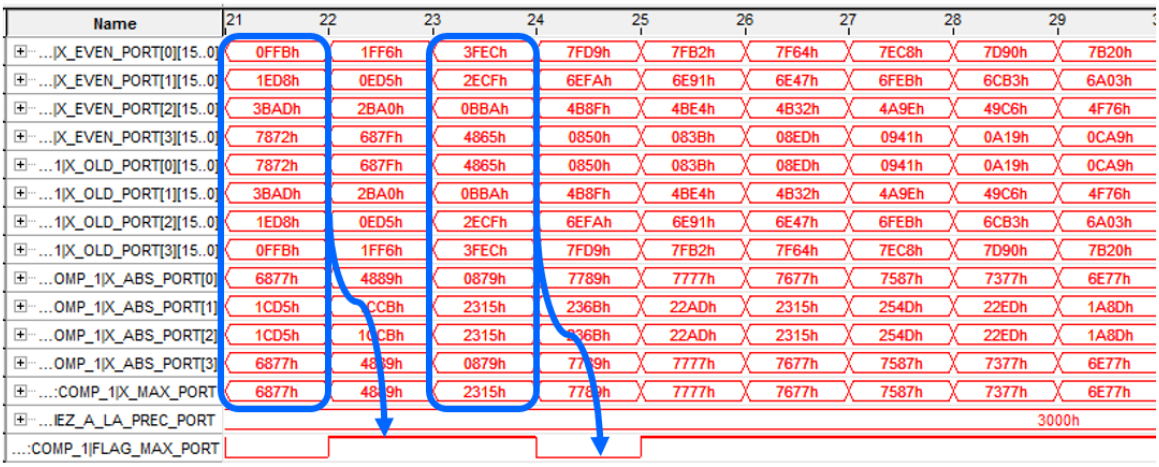


```
entity COMPARADOR_1 is
port(
CLK_PORT       : in std_logic;
RESET_PORT    : in std_logic;
X_EVEN_PORT: in tipo_entrada_RAMX_paralelo;
X_OLD_PORT  : in tipo_entrada_RAMX_paralelo;
X_ABS_PORT: out tipo_entrada_RAMX_paralelo;
X_MAX_PORT: out
std_logic_vector(WORD_LENGTH-1 downto 0);
DIEZ_A_LA_PREC_PORT: out
std_logic_vector(WORD_LENGTH-1 downto 0);
FLAG_MAX_PORT: out std_logic
);
end COMPARADOR_1;

architecture BEHAVIORAL of COMPARADOR_1 is
signal AUX_A_SIGNAL:
tipo_entrada_RAMX_paralelo := (others =>
(others=>'0'));
signal AUX_B_SIGNAL:
tipo_entrada_RAMX_paralelo := (others =>
(others=>'0'));
signal AUX_RESTA_SIGNAL:
tipo_entrada_RAMX_paralelo := (others =>
(others=>'0'));
signal AUX_Y_SIGNAL :
std_logic_vector(WORD_LENGTH-1 downto 0) :=
(others=>'0');
signal AUX_FLAG_MAX_SIGNAL: std_logic := '0';
begin -- begin arch

RESTA: PROCESS (X_EVEN_PORT, X_OLD_PORT)
BEGIN
for i in 0 to ROWS-1 loop
AUX_A_SIGNAL(i)  <=  X_EVEN_PORT(i) -
X_OLD_PORT(i);
AUX_B_SIGNAL(i)  <=  X_OLD_PORT(i)  -
X_EVEN_PORT(i);
if (AUX_A_SIGNAL(i) <= AUX_B_SIGNAL(i)) then
AUX_RESTA_SIGNAL(i) <= AUX_A_SIGNAL(i);
else
AUX_RESTA_SIGNAL(i) <= AUX_B_SIGNAL(i);
end if;
end loop;
END PROCESS RESTA;
X_ABS_PORT <= AUX_RESTA_SIGNAL;

MAX: process(AUX_RESTA_SIGNAL)
variable AUX_LARGEST_VARIABLE :
std_logic_vector(WORD_LENGTH-1 downto 0);
begin
AUX_LARGEST_VARIABLE := (others => '0');
for i in 0 to ROWS-1 loop
if(AUX_LARGEST_VARIABLE <=
AUX_RESTA_SIGNAL(i)) then
AUX_LARGEST_VARIABLE :=
AUX_RESTA_SIGNAL(i);
end if;
end loop;
AUX_Y_SIGNAL  <= AUX_LARGEST_VARIABLE;
end process MAX;
X_MAX_PORT <=  AUX_Y_SIGNAL ;

MUX: process( AUX_Y_SIGNAL )
begin
if (AUX_Y_SIGNAL >
DIEZ_A_LA_PREC_SIGNAL_COMP_1) then
AUX_FLAG_MAX_SIGNAL <= '1';
else
AUX_FLAG_MAX_SIGNAL <= '0';
end if;
end process MUX;
DIEZ_A_LA_PREC_PORT <=
DIEZ_A_LA_PREC_SIGNAL_COMP_1;
```

```
FF: process(RESET_PORT, CLK_PORT,
AUX_FLAG_MAX_SIGNAL)
begin
if(RESET_PORT = '0') then
FLAG_MAX_PORT  <= '0';
elsif(CLK_PORT'event and CLK_PORT = '1') THEN
FLAG_MAX_PORT  <=  AUX_FLAG_MAX_SIGNAL;
end if;
end process FF;

end BEHAVIORAL;
```

Las pruebas realizadas para corroborar el comportamiento funcional de la unidad se observan la figura 22 donde los vectores X_EVEN y X_OLD_EVEN son las entradas. Este ejemplo corresponde a la síntesis de la primera fila L4 donde los vectores tienen n=4 componentes. También se observan los cuatro componentes del vector resultante del cálculo del valor absoluto de la resta en la señal llamada X_ABS_PORT y el cálculo del componente máximo en la señal X_MAX_PORT. La constante para realizar la comparación es en esta prueba 3000h. Se utilizó el mismo generador de señales pseudo aleatorias para cargar la unidad y probar su comportamiento funcional cuando las entradas cambian en cada flanco.

Figura 22: Prueba de funcionamiento del comparador 1



También se observan en la figura 22 todas las señales descritas y se puede detallar en los ciclos 1 y 3 que funciona correctamente mostrando su salida por medio de un flip flop y por este motivo su salida correcta se genera un ciclo de reloj después de la carga de las entradas. En el primer ciclo vemos como sus entradas son X_EVEN=[0FFBh, 1ED8h, 3BADh, 7872h] y X_OLD_EVEN=[7872h, 7872h, 1ED8h, 0FFBh]. El cálculo del valor absoluto de la resta se ve reflejado en la señal X_ABS= [6877h, 1CD5h, 1CD5h, 6877h]. Se puede corroborar que el máximo componente del vector X_ABS es 6877h y este valor es asignado a la señal X_MAX la

cual se compara con la constante 3000h, y como es mayor entonces la señal de sincronización FLAG_MAX_PORT=1 generado un ciclo de reloj después. En el tercer ciclo se puede analizar de la misma manera para corroborar el funcionamiento cuando las entradas generan una señal de bandera igual a cero.

4.15 COMPARADOR 2

El componente COMPARADOR_2 este encargado de realizar la comparación entre la substracción de dos vectores columna y un valor igual a $10^{-12+4} = 10^{-8}$. Se debe tener en cuenta que el elemento que retorna la función $fRec()$ es un vector de la misma dimensión n que $xEven$.

La función recibe dos vectores y se ejecuta la operación resta entre ellos para luego calcular el valor absoluto al resultado (ver ecuación 9). Esto producirá otro vector en donde cada uno de sus componentes se compara con el valor constante 10^{-8} para así formar un vector resultante de n componentes que sólo pueden tomar el valor 1 ó 0 y representan el resultado de la comparación. Al final se verifica la siguiente condición: si el vector resultante está conformado de sólo unos entonces el registro de la última etapa guardará un 1, y si el vector resultante contiene al menos un cero entonces el valor del registro se asigna a 0. Este flip flop en la última etapa se utiliza como señal de sincronización y se denomina FLAG_WSM_PORT. El hardware de la unidad COMPARADOR_2 se observa en la figura 23. Los demás puertos de salida únicamente fueron utilizados para la realización de pruebas de funcionamiento y no se conectan en el momento de su instanciación.

Los ensayos de funcionamiento de la unidad se realizaron con la misma metodologia que las pruebas anteriores generando entradas pseudo aleatorias para cada ciclo de reloj utilizando una palabra de 16 bits.

En esta prueba, vista en la figura 24 se ajusta la constante a 2000h. De acuerdo a la escala en el eje X de la fig. 24 se peude observar por ejemplo en el ciclo #13 las entradas X_EVEN= [465Dh, 577Eh, 720Bh, 31D4h] y X_OLD_EVEN= [31D4h, 720Bh, 577Eh, 465Dh] dando como resultado de la resta en valor absoluto X_ABS= [1489h, 1A8Dh, 1A8Dh, 1489h]. Como todos los componentes de X_ABS son menores a la constante 2000h entonces el vector resultante es [1, 1, 1, 1] y la salida registrada de la última estapa guarda el valor 1 en el flip flop donde puede ser visualizado un ciclo de reloj después. Como se puede observar para todas las demas

entradas la señal resultante de sincronizacion es nula cuando el vector resultante tiene al menos un 0.

Figura 23: Hardware del comparador 2

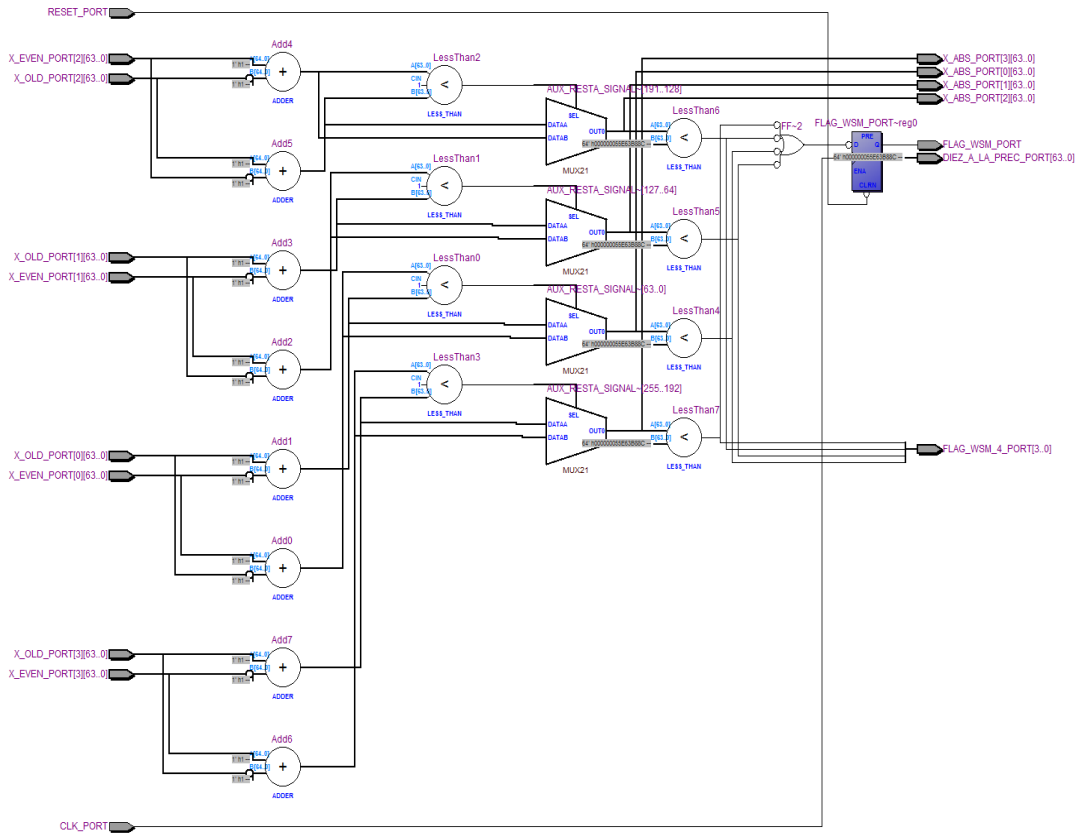


Figura 24: Prueba de funcionamiento del comparador 2

Name	6	7	8	9	10	11	12	13	14	15
...COMP_2|X_EVEN_PORT[0]	348Ch	6919h	5232h	2465h	48CBh	1197h	232Eh	465Dh	0CBBh	1976h
...COMP_2|X_EVEN_PORT[1]	25AFh	783Ah	4311h	3546h	59E8h	00B4h	320Dh	577Eh	1D98h	0855h
...COMP_2|X_EVEN_PORT[2]	00DAh	5D4Fh	6664h	1033h	7C9Dh	25C1h	1778h	720Bh	38EDh	2D20h
...COMP_2|X_EVEN_PORT[3]	4305h	1E90h	25BBh	53ECh	3F42h	661Eh	54A7h	31D4h	7B32h	6EFFh
...:COMP_2|X_OLD_PORT[0]	4305h	1E90h	25BBh	53ECh	3F42h	661Eh	54A7h	31D4h	7B32h	6EFFh
...:COMP_2|X_OLD_PORT[1]	00DAh	5D4Fh	6664h	1033h	7C9Dh	25C1h	1778h	720Bh	38EDh	2D20h
...:COMP_2|X_OLD_PORT[2]	25AFh	783Ah	4311h	3546h	59E8h	00B4h	320Dh	577Eh	1D98h	0855h
...:COMP_2|X_OLD_PORT[3]	348Ch	6919h	5232h	2465h	48CBh	1197h	232Eh	465Dh	0CBBh	1976h
...:COMP_2|X_ABS_PORT[0]	0E79h	4A89h	2C77h	2F87h	0989h	5487h	3179h	1489h	6E77h	5589h
...:COMP_2|X_ABS_PORT[1]	24D5h	1AEBh	2353h	2513h	22B5h	250Dh	1A95h	1A8Dh	1B55h	24CBh
...:COMP_2|X_ABS_PORT[2]	24D5h	1AEBh	2353h	2513h	22B5h	250Dh	1A95h	1A8Dh	1B55h	24CBh
...:COMP_2|X_ABS_PORT[3]	0E79h	4A89h	2C77h	2F87h	0989h	5487h	3179h	1489h	6E77h	5589h
...|DIEZ_A_LA_PREC_PORT								2000h		
...P_2|FLAG_WSM_4_PORT	1001b	0110b	0000b		1001b	0000b	0110b	1111b	0110b	0000b
...2:COMP_2|FLAG_WSM_PORT										

El código VHDL de la unidad COMPARADOR_2 se muestra a continuación con el fin de mostrar el modelamiento comportamental para esta entidad la cual está conformada por tres procesos. El primero realiza la resta y el valor absoluto, el segundo realiza la comparación del vector resultante en el procedimiento anterior y la constante para dar como resultado otro vector. Y el último proceso VHDL es la comprobación componente a componente para precisar si alguno de ellos es cero y entonces así, el resultado será la señal de sincronización que es registrada en un flip flop gracias a la condición de flanco del reloj en el último proceso.

```
entity COMPARADOR_2 is
port(
CLK_PORT                    : in std_logic;
RESET_PORT                  : in std_logic;
X_EVEN_PORT: in tipo_entrada_RAMX_paralelo;
X_OLD_PORT: in tipo_entrada_RAMX_paralelo;
X_ABS_PORT: out tipo_entrada_RAMX_paralelo;
DIEZ_A_LA_PREC_PORT: out
std_logic_vector(WORD_LENGTH-1 downto 0);
FLAG_WSM_4_PORT: out tipo_comparador2;
FLAG_WSM_PORT     : out std_logic
  );
end COMPARADOR_2;

architecture BEHAVIORAL of COMPARADOR_2 is
signal AUX_A_SIGNAL :
tipo_entrada_RAMX_paralelo := (others =>
(others=>'0'));
signal AUX_B_SIGNAL :
tipo_entrada_RAMX_paralelo := (others =>
(others=>'0'));
signal AUX_RESTA_SIGNAL          :
tipo_entrada_RAMX_paralelo := (others =>
(others=>'0'));
signal AUX_FLAG_WSM_2_SIGNAL  :
tipo_comparador2 := (others => '0');
signal AUX_FLAG_WSM_3_SIGNAL  :
tipo_comparador2 := (others => '0');
begin -- begin arch

RESTA2: PROCESS (X_EVEN_PORT,
X_OLD_PORT)
BEGIN
for i in 0 to ROWS-1 loop
AUX_A_SIGNAL(i) <= X_EVEN_PORT(i) -
X_OLD_PORT(i);
AUX_B_SIGNAL(i) <= X_OLD_PORT(i) -
X_EVEN_PORT(i);
if (AUX_A_SIGNAL(i) <= AUX_B_SIGNAL(i)) then
AUX_RESTA_SIGNAL(i) <= AUX_A_SIGNAL(i);
else
AUX_RESTA_SIGNAL(i) <= AUX_B_SIGNAL(i);
end if;
end loop;
END PROCESS RESTA2;
X_ABS_PORT <= AUX_RESTA_SIGNAL;

COMP: process(AUX_RESTA_SIGNAL)
begin
for i in 0 to ROWS-1 loop
if (AUX_RESTA_SIGNAL(i) <
DIEZ_A_LA_PREC_SIGNAL_COMP_2) then
AUX_FLAG_WSM_2_SIGNAL(i) <= '1';
else
AUX_FLAG_WSM_2_SIGNAL(i) <= '0';
end if;
end loop;
AUX_FLAG_WSM_3_SIGNAL <=
AUX_FLAG_WSM_2_SIGNAL;
end process COMP;
FLAG_WSM_4_PORT <=
AUX_FLAG_WSM_3_SIGNAL;
DIEZ_A_LA_PREC_PORT <=
DIEZ_A_LA_PREC_SIGNAL_COMP_2;

FF: process(RESET_PORT, CLK_PORT,
AUX_FLAG_WSM_3_SIGNAL)
begin
if(RESET_PORT = '0') then
FLAG_WSM_PORT <= '0';
elsif(CLK_PORT'event and CLK_PORT = '1') THEN
for i in 0 to ROWS-1 loop
if AUX_FLAG_WSM_3_SIGNAL(i) = '0' then
FLAG_WSM_PORT <= '0';
exit;
else
FLAG_WSM_PORT <= '1';
end if;
end loop;
end if;
end process FF;
end BEHAVIORAL;
```

4.16 UNIDAD_FREC

La función $fRec()$ es utilizada recurrentemente en este algoritmo (ecuación 10) y por tanto fue la primera implementación a realizar en FPGA para buscar su aceleración. La función devuelve un vector de n componentes y su cálculo depende de los elementos dentro de la lista y del valor del parámetro λ.

Existen diversas formas de realizar un hardware que tenga la capacidad de procesar esta función. Una forma, por ejemplo, es de tipo completamente paralelo haciendo n componentes $f_i(x)$. Sin embargo, esto se intentó y cómo n crece de manera exponencial en relación a las listas entonces, así mismo crece el consumo de recursos en la FPGA haciendo de esta implementación sólo satisfactoria para la lista L4. Otra implementación trata de hacer un sólo componente $f_i(x)$ y hacer un sacrificio en control con el fin de ejecutar $fRec()$. Sin embargo, el sacrificio en control es significativamente considerable ya que también crece el área de este control a medida que crecen las listas y sólo se lograron procesar las listas L4 y L6.

En la imagen de la figura 25 se pueden ver los componentes de UNIDAD_FREC. Se observa un banco de registros de propósito específico, tres contadores, un bloque denominado UNIDAD_FILA el cual realiza la función de la forma $1/(1 + x * y * z * w)$ y un bloque controlador denominado CONTROL_3. Las entradas de esta unidad son la lista L, el parámetro t y un vector columna x y devuelve como salidas un vector y y el estado de control. Los registros mencionados son registros paralelos los cuales de acuerdo a la figura 25 y la nomenclatura utilizada, ROM_PARALELO es un registro para almacenar la lista, RAM_L_PARALELO y RAM_X_MAS1_PARALELO sirven para realizar la operación con punteros $x(L)$ de Matlab® y finalmente un registro RAM_X_PARALELO que permite registrar el vector de salida resultante. En la figura 26, se observa el detalle del componente UNIDAD_FREC.

4.17 FUNCIONAMIENTO GENERAL (UNIDAD_FREC)

1. En primer lugar, se carga la lista L en el registro ROM_PARALELO y el vector de entrada X en el registro RAM_X_MAS1_PARALELO.

2. En segundo lugar, para realizar la operación $x(L)$, se realiza un barrido del tamaño de la lista cargando los datos provenientes de RAM_X_MAS1_PARALELO con las direcciones de que están contenidas en

el registro ROM_PARALELO y se carga el resultado en el registro RAM_L_PARALELO. En esta propuesta este barrido se implementa en un sólo ciclo de reloj produciendo un alto consumo de recursos de la FPGA.

Figura 25: Diagrama de componentes de UNIDAD_FREC

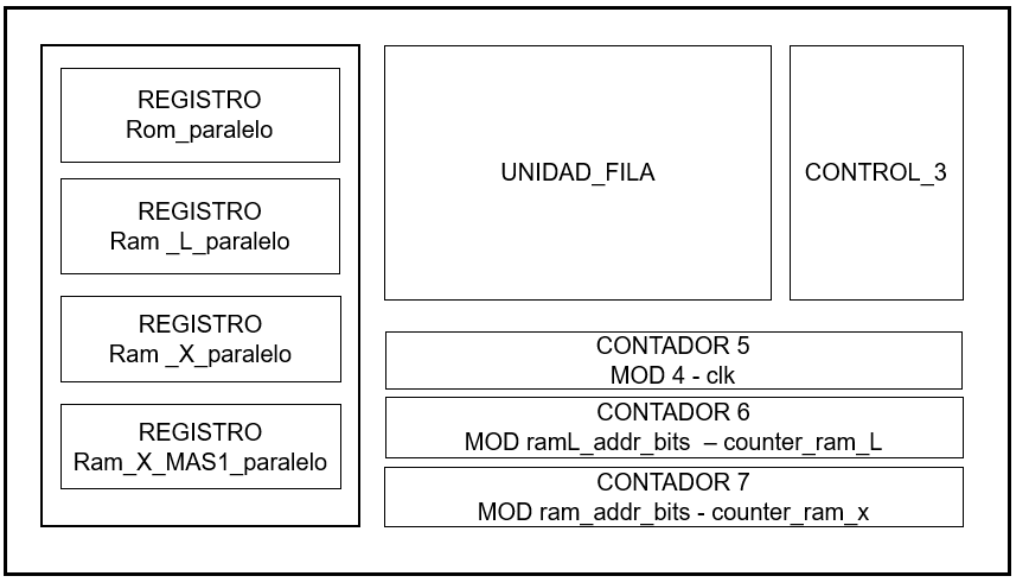
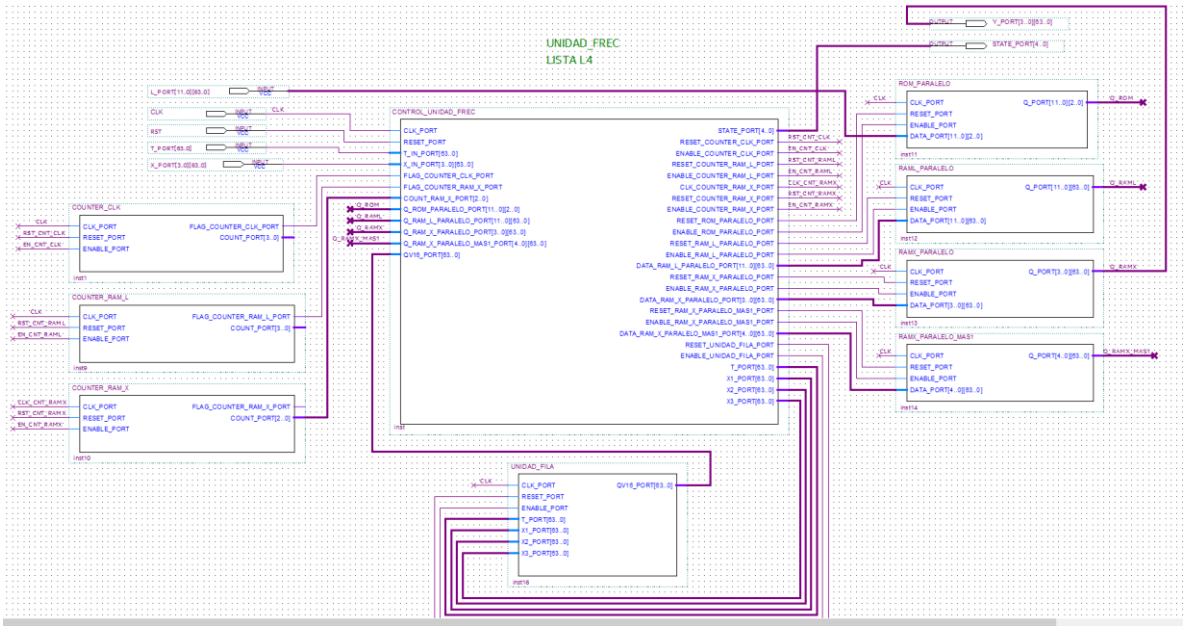


Figura 26: Diagrama de componentes de UNIDAD_FREC (detalle)



3. El siguiente paso es cargar una fila de la matriz registrada en RAM_L_PARALELO en el componente UNIDAD_FILA.

4. Se espera una señal de sincronización proveniente de un contador_5 (COUNTER_CLK) con el fin de demorar cualquier cambio en las entradas del bloque UNIDAD_FILA y así se logra solucionar el problema de latencia o retardo de esta unidad. En este paso siempre se carga la salida de UNIDAD_FILA en un componente del registro de salida RAM_X_PARALELO.

5. A continuación, se incrementa la cuenta del contador_5 y se vuelve al paso 3 donde se cargará una nueva fila de la matriz y así, componente a componente, se guardarán en el registro de salida RAM_X_PARALELO.

6. Finalmente, cuando se haya realizado todo el barrido del registro RAM_L_PARALELO y todos los componentes de RAM_X_PARALELO estén cargados entonces se entrega la salida por medio del puerto de salida y.

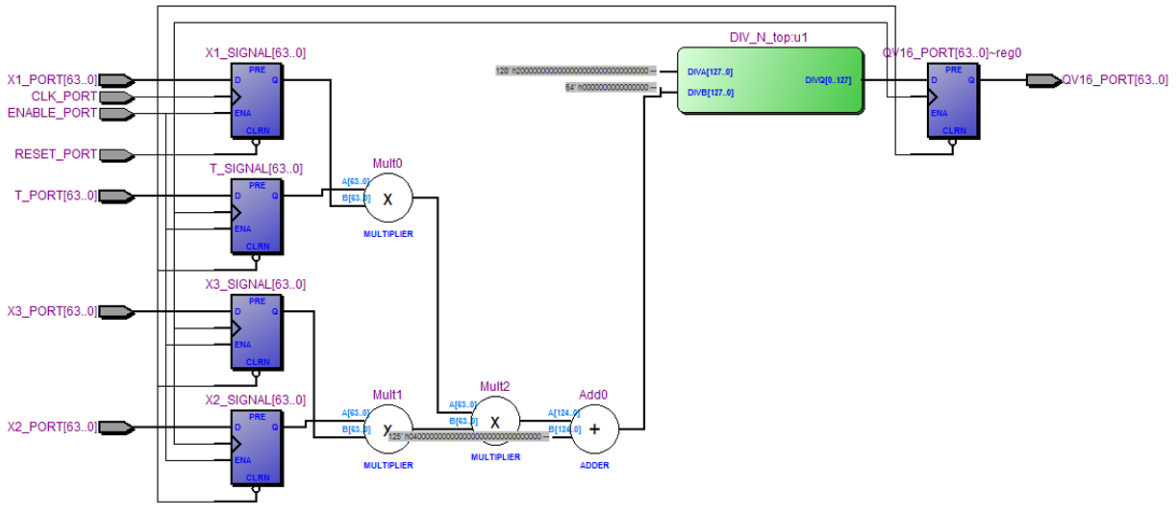
A continuación, se detallará los componentes de UNIDAD_FREC junto con algunos análisis y pruebas de funcionamiento.

4.18 UNIDAD_FILA

Para la recepción de las entradas se requiere de sincronización de datos y reset asíncrono para lograr una buena estabilidad y coherencia en la salida. El circuito propuesto para la unidad aritmética básica se observa en la figura 27. Las entradas $t,\ x1,\ x2,\ x3$ se almacenan en registros para garantizar estabilidad y sincronización en relación al retardo que generan los circuitos combinacionales de los multiplicadores, el sumador y el divisor.

Para garantizar la correcta operación de esta unidad se debe controlar la alimentación de los valores de entrada a ésta, debido a que, si no se hace correctamente teniendo en cuenta los retardos, el registro de salida $QV16$ guardará el valor incorrecto al momento en que las entradas al divisor cambian. Este problema se arregló utilizando un contador (contador_5 o COUNTER_CLK) para demorar algunos ciclos de reloj más en un estado determinado de la FSM con el fin de que las entradas del divisor no varíen de ciclo en ciclo y permitir que el registro de salida almacene el valor correcto al final del conteo de este contador auxiliar.

Figura 27. Hardware de la unidad aritmética básica UNIDAD_FILA.



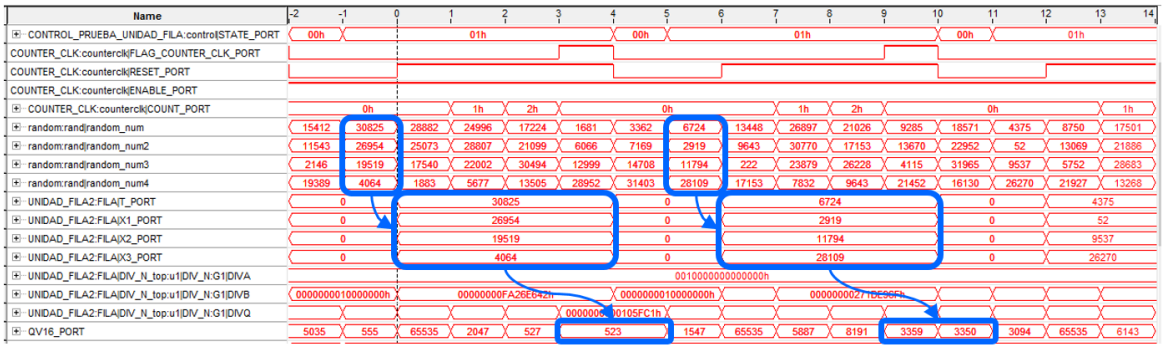
Se observa en la figura 27 que las entradas son de 64 bits y por ende los dos primeros multiplicadores son también de 64 bits. Para que el tercer multiplicador Mult2 no tuviera que ser obligatoriamente de 128 bits se toman los 64 bits más significativos de las salidas de Mult0 y Mult1. El sumador que le sigue se implementó de 128 bits para dar más bits significativos a las entradas del divisor.

El divisor se diseñó de 128 bits con el principal objetivo de tener suficientes cifras significativas, debido a que la operación a realizar es el inverso multiplicativo $1/x$, el cual se realizó haciendo un desplazamiento del denominador de la mitad de la palabra del divisor. Es decir, el numerador es asignado al valor de uno, en el formato punto fijo con 128 bits (el punto fijo estaría en el bit 125) y el denominador es la suma $(1 + t * x * y * z)$ desplazada hacia la derecha 64 bits (el punto fijo estaría en el bit 61). Para la salida se toma únicamente los 64 bits menos significativos del divisor siendo almacenados en el registro de salida QV16. De esta manera evitamos un overflow en el divisor, debido a que, si uno se divide por un valor superior que uno, el resultado siempre es inferior que uno. Entonces el divisor arrojaría resultados incorrectos, debido a que una condición importante para la correcta operación de divisores en punto fijo es que el cociente $Q \leq 2^{((n-1)-w)}$, siendo w el peso del punto fijo que en este caso es 61. Por tanto, el cociente Q siempre debe ser menor o igual a 4, como se supone es el rango máximo de los datos en estas unidades.

La figura 28 muestra una prueba básica de la UNIDAD_FILA sin la implementación del contador_5 para retardar el cambio de estado. Esto

con el fin de observar como el registro de salida toma el valor incorrecto por motivo de los retardos de los multiplicadores, el sumador y el divisor especialmente. Se emula la generación de entradas con la misma entidad pseudo aleatoria utilizada anteriormente. Se puede observar en el ciclo (-1) de la figura 28 que los valores cargados en UNIDAD_FILA son generados por las señales provenientes del generador, las cuales son T=30825, X1=26954, X2=19519, X3=4064 y dan como resultado el valor cargado en la señal QV16=523 en el ciclo 3. Utilizando la ecuación 13 para la conversión correcta de datos sin formato se puede analizar más claramente los valores de entrada T=3.762817382, X1=3.290283203, X2=2.382690429, X3=0.49609375 y el valor correcto de salida QV16= 0.0638427734. Si se analiza los valores de la señal QV16 se puede ver claramente como el valor correcto demora cuatro ciclos de reloj (para estas entradas en particular). También se puede observar cómo cambian los valores de la salida del divisor visualizados en la señal DIVQ de la figura a medida que el reloj va avanzando y por este motivo es que la implementación del contador_5 es imprescindible.

Figura 28: Pruebas unidad aritmética básica



En la figura 28 también se puede notar en el ciclo 5, que, ante otros valores diferentes, el resultado correcto no se genera en el cuarto ciclo como sucedió anteriormente. Ya que es muy complejo saber el retardo especifico (o único) de este divisor combinacional se debe considerar el módulo del contador_5, o bajar la frecuencia del reloj para evitar estos glitches. Una solución en un proyecto futuro puede ser registrar este divisor en etapas intermedias para convertir este circuito combinacional en uno secuencial, y así establecer la latencia exacta que haga independiente la operación del divisor con del reloj del sistema.

En caso de modificar la frecuencia del reloj se debe modificar el módulo del CONTADOR_CLK, para asegurar que la salida QV16 siempre sea la correcta. Se implementó el sistema con un reloj de 5 MHz para la lista L4 y un reloj de 2.5 MHz para la lista L6.

4.19 MULTIPLICADORES

Los componentes multiplicadores se realizaron utilizando las librerías *ieee.numeric_std* y *ieee.std_logic_unsigned*. Como se observa en la figura 27 se requieren de tres multiplicadores sin signo cada uno de 64 bits. La herramienta de síntesis realiza la implementación de los tres multiplicadores utilizando bloques DSP de la FPGA, con 12 bloques DSP 36-bit. De esta manera se logra un ahorro sustancial en recursos de lógica y se aumenta la velocidad y la confiabilidad del resultado de estas operaciones de multiplicación.

4.20 DIVISOR

El divisor es la operación aritmética más crucial porque hace parte de la función recurrente $fRec()$. Los operandos de este divisor son de punto fijo de 128 bits. Como no existen divisores de 128 en la mega función LPM_DIVIDE de Quartus®, se implementó este componente con una arquitectura basada en desplazamientos vista en [12]. Sin embargo, se analizaron también las arquitecturas del divisor basado en la mega función LPM_DIVIDE con 64 bits para poder compararlas y observar cuál es la que menos área ocupa. Los siguientes son los nombres de cada una de las arquitecturas estudiadas:

1. FLAT
2. HIER ó HIERG
    a. RTA
    b. RTB
    c. RTC
3. LPM_DIVIDE -> Modo DEFAULT
4. LPM_DIVIDE -> Modo AREA
5. LPM_DIVIDE -> Modo SPEED

La declaración de la entidad VHDL del divisor es:

```
entity DIV_N is
  generic(WORD_LENGTH_DIV : integer); -- WORD_LENGTH_DIV = 128
  port (
```

```
        DIVA : in std_logic_vector(WORD_LENGTH_DIV-1 downto 0);  -- Q = A/B
        DIVB : in std_logic_vector(WORD_LENGTH_DIV-1 downto 0);
        DIVO : out std_logic; --DIVO = 1 y DIVQ= 0 cuando DIVB = 0
        DIVQ : out std_logic_vector(WORD_LENGTH_DIV-1 downto 0));
end DIV_N;
```

Basado siempre en [12], se describen a continuación las arquitecturas analizadas para este divisor. La arquitectura FLAT está basada en el enfoque del lápiz y el papel. Es un divisor donde a partir de restas, comparaciones y desplazamientos se logra calcular el cociente y la señal de overflow. Dividendo y divisor pueden ser negativos o positivos dado que se utiliza una función llamada COMP2() que está definida dentro de un paquete llamado PACK.vhd. Esta función realiza el complemento a dos de un número. Gracias a esto, si un operando de la división es positivo y el otro negativo entonces el resultado debe ser tomado en complemento a dos dado que sería un resultado negativo.

Los elementos centrales de la descripción VHDL de la arquitectura FLAT corresponden a los ciclos *for-generate*. El ciclo *sh0* realiza conexiones y el ciclo *st0* realiza la asignación de los bits del cociente Q y la restauración del residuo R.

```
architecture FLAT of DIV_N is
subtype bit2n is
std_logic_vector(2*WORD_LENGTH_DIV-1
downto 0);
type vectn1 is array (WORD_LENGTH_DIV downto
0) of bit2n;
signal R, L : vectn1 := (others =>(others=>'0'));
signal Q,Q1,A1,B1 :
std_logic_vector(WORD_LENGTH_DIV-1 downto
0) := (others=>'0');
signal A2,B2,Z0 :
std_logic_vector(WORD_LENGTH_DIV-1 downto
0) := (others=>'0');
signal S, Z : std_logic := '0';
begin --begin architecture

Z0 <= (Z0'range => '0');
COMP2(DIVA, A1);
COMP2(DIVB, B1);
A2 <= DIVA when DIVA(WORD_LENGTH_DIV-1)
= '0' else A1;
B2 <= DIVB when DIVB(WORD_LENGTH_DIV-1)
= '0' else B1;
S <= DIVA(WORD_LENGTH-1) xor
DIVB(WORD_LENGTH_DIV-1);
L(0) <= Z0 & B2;
R(WORD_LENGTH_DIV) <= Z0 & A2;

shO : for j in 1 to WORD_LENGTH_DIV-1
generate
L(j) <= L(j-1)(2*WORD_LENGTH_DIV-2 downto
0) & '0';
end generate;

stO : for j in WORD_LENGTH_DIV-1 downto 0
generate
Q(j) <= '1' when (R(j+1) >= L(j)) else '0';
R(j) <= R(j+1)-L(j) when Q(j) = '1' else R(j+1);
end generate;

Z <= '1' when DIVB = Z0 else '0';
DIVO <= Z;
COMP2(Q, Q1);
DIVQ <= (DIVQ'range => '0') when Z = '1' else
Q1 when S = '1' else Q;

end FLAT;
```

La arquitectura HIER es similar a la arquitectura FLAT con la diferencia que las sentencias dentro de los ciclos *generate* donde se determinan R y Q (residuo y cociente) son agrupadas dentro de la entidad VHDL

denominada STAGE, esto con el fin de hacer la síntesis más rápida. El componente STAGE puede ser referenciado en lugar de volver a sintetizarse. En otras palabras, el componente STAGE da facilidades para la síntesis, pero el flujo de datos es igual.

La tarea de referenciar un componente que ha sido sintetizado se realiza utilizando un subprograma de Quartus® llamado *Design Partion*, el cual permite dividir el diseño en particiones, donde cualquier entidad de cualquier jerarquía puede ser definida como una nueva partición y así los archivos de síntesis, simulación, mapeo, localización y ruteo producidos en una compilación pueden ser reutilizados en otra, sin volver a ser compilados.

La tarea de referenciar un componente requiere una cuidadosa asignación de particiones y debe cumplir con importantes recomendaciones para que la herramienta de síntesis realice de manera óptima el mapeo, la localización y el ruteo. El código de la arquitectura HIER se presenta a continuación.

```
architecture HIER of DIV_N is
subtype bit2n is
std_logic_vector(2*WORD_LENGTH_DIV-1
downto 0);
type vectn1 is array (WORD_LENGTH_DIV downto
0) of bit2n;
signal R, L : vectn1 := (others =>(others=>'0'));
signal Q,Q1,A1,B1 :
std_logic_vector(WORD_LENGTH_DIV-1 downto
0) := (others=>'0');
signal A2,B2,T,Z0 :
std_logic_vector(WORD_LENGTH_DIV-1 downto
0) := (others=>'0');
signal S, Z : std_logic := '0';
begin
Z0 <= (Z0'range => '0');
COMP2(DIVA, A1);
COMP2(DIVB, B1);
A2 <= DIVA when DIVA(WORD_LENGTH_DIV-1)
= '0' else A1;
B2 <= DIVB when DIVB(WORD_LENGTH_DIV-1)
= '0' else B1;
S <= DIVA(WORD_LENGTH_DIV-1) xor
DIVB(WORD_LENGTH_DIV-1);
L(0) <= Z0 & B2; R(WORD_LENGTH_DIV) <= Z0
& A2;
Z <= '1' when DIVB = Z0 else '0';
DIVO <= Z;

shO : for j in 1 to WORD_LENGTH_DIV-1
generate
L(j) <= L(j-1)(2*WORD_LENGTH_DIV-2 downto
0) & '0';
end generate;

stO : for j in WORD_LENGTH_DIV-1 downto 0
generate
st : entity work.STAGE(RTC) -- AQUI SE ESCOGE
LA AQUITECTURA DE STAGE DE LA
ARQUITECTURA HIER
generic map(WORD_LENGTH_DIV =>
WORD_LENGTH_DIV)
port map(
R     => R(j+1),
L     => L(j),
QO    => Q(j),
RO    => R(j)
);
end generate;

COMP2(Q, Q1);
DIVQ <= (DIVQ'range => '0') when Z = '1' else
        Q1 when S = '1' else Q;

end HIER;
```

La entidad STAGE tiene tres variaciones de su arquitectura: RTA, RTB y RTC. En la arquitectura RTA (figura 29), no se logra optimización en

hardware, pero si se reduce el tiempo de compilación, debido a que se hace uso de la misma cantidad de comparadores y restadores que se utilizan en la arquitectura FLAT. La arquitectura RTB de STAGE mostrada en la figura 30 reduce un comparador e infiere un sólo restador.

La arquitectura RTC que se observa en la figura 31 sugiere otra forma de hacer el componente STAGE basándose en el hecho que las entradas de STAGE son de 256bits y que los 128 bits más significativos de la señal R (dividendo) en RTC son siempre cero. Se hace MSB=1 cuando los 128 bits más significativos de L (divisor) no son todos cero. El bit del cociente Q0 es puesto en uno cuando MSB=0 y cuando el bit más a la izquierda del resultado de la substracción es también cero. Esto significa que los 128 bits más a la derecha del R (dividendo) son mayor o igual a los 128 bits más a la derecha del divisor y los 128 bits más a la izquierda del divisor son cero. Al final se genera el residuo parcial y se pasa a la siguiente etapa.

```
entity STAGE is
generic(WORD_LENGTH_DIV : integer);
port(
R, L : in std_logic_vector(2*WORD_LENGTH_DIV-
1 downto 0); --256
QO : out std_logic;
RO : out
std_logic_vector(2*WORD_LENGTH_DIV-1
downto 0) --256
);
end STAGE;

architecture RTA of STAGE is
signal T : std_logic := '0';
begin
T <= '1' when (R >= L) else '0';
QO <= T;
RO <= R-L when (T = '1') else R;
end RTA;

architecture RTB of STAGE is
signal T : std_logic;
signal SUB :
std_logic_vector(2*WORD_LENGTH_DIV-1
downto 0);
begin
SUB <= R - L;
T <= '1' when (SUB(2*WORD_LENGTH_DIV-1) =
'0') else '0';
QO <= T;
RO <= SUB when (T = '1') else R;
end RTB;

architecture RTC of STAGE is
signal T, MSB : std_logic;
signal SUB :
std_logic_vector(WORD_LENGTH_DIV-1 downto
0);  -- 128
signal cerosN :
std_logic_vector(WORD_LENGTH_DIV-1 downto
0);  --128
begin
cerosN <= (cerosN'range => '0');
SUB <= R(WORD_LENGTH_DIV-1 downto 0) -
L(WORD_LENGTH_DIV-1 downto 0);
MSB <= '0' when L(2*WORD_LENGTH_DIV-1
downto WORD_LENGTH_DIV-1) = cerosN & B"0"
else '1'; --128 ceros
T <= '1' when (MSB = '0') and
(SUB(WORD_LENGTH_DIV-1) = '0') else '0';
QO <= T;
RO <= cerosN & SUB when (T = '1') else R; --
128 ceros

end RTC;
```

Figura 29: Arquitectura RTA del subcomponente STAGE.

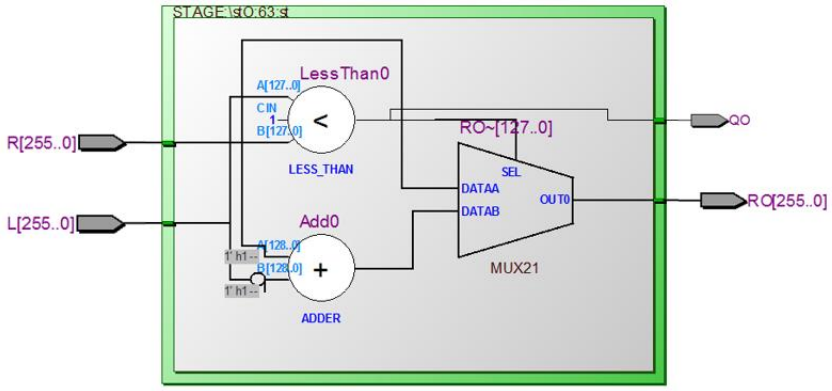


Figura 30: Arquitectura RTB del subcomponente STAGE

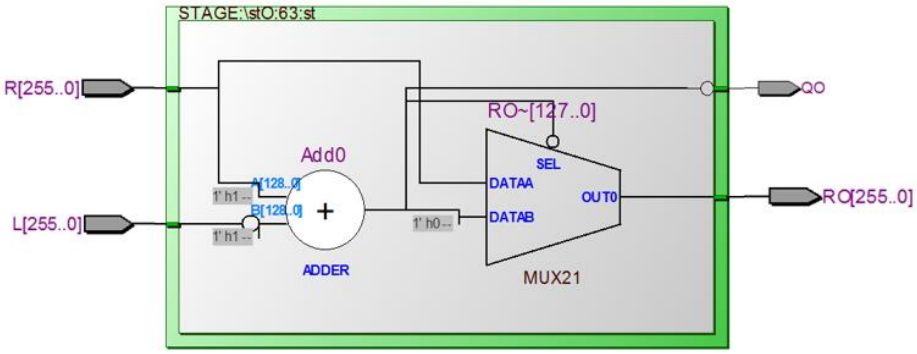
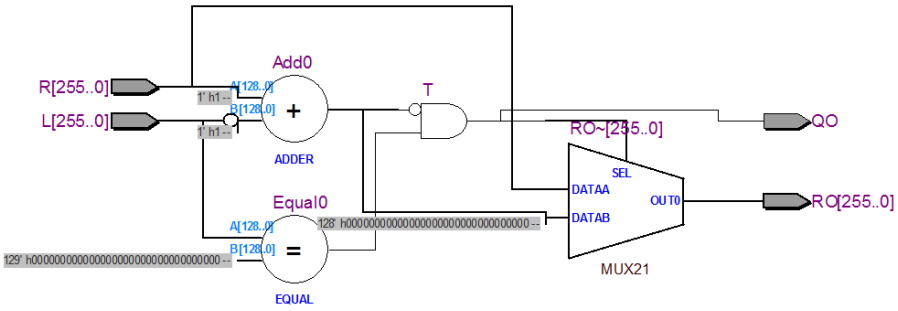


Figura 31: Arquitectura RTC del subcomponente STAGE



Como la mega función LPM_DIVIDE únicamente se puede implementar hasta de 64 bits, se analizaron varios divisores de 64 bits como se observa en la tabla 6. Se compara el área ocupada por el divisor basado en la

mega funciones y los basados en [12]. Los cuáles corresponden a las tres variaciones del subcomponente STAGE y las tres variaciones de la mega función LPM_DIVIDE que es un divisor en punto fijo predefinida por Altera (Intel®). La arquitectura RTC ocupa aprox. el 30% del área que ocupa la arquitectura RTA, lo cual muestra una significativa reducción. Se analiza también que cuando se sintetiza utilizando las variaciones del modo de optimización en síntesis (basado en área, velocidad o por defecto) en los tres casos se ocupa la misma cantidad de ALUTs y comparado con la arquitectura RTC es aún menor.

Tabla 6: Comparación de divisores de 64 bits.

Recurso	DIV 64 BITS HIERG/RTC	DIV 64 BITS - HIERG/RTB	DIV 64 BITS HIERG - HIERG/RTA	DIV 64 BITS MEGA FUNCION - DEFAULT	DIV 64 BITS MEGA FUNCION - AREA	DIV 64 BITS MEGA FUNCION - SPEED
ALUT combinacionales	7,776	15,570	26,336	4,270	4,270	4,270

La tabla 7 muestra los valores de recursos de lógica y pines de entrada y salida que se consumen en divisores de 128 bits basados en [12]. Se observa que la arquitectura HIERG/RTA y la FLAT son bastante considerables y no alcanzan los recursos disponibles en la ARRIA II GX para sintetizar estas arquitecturas (junto al diseño total) ya que esta tiene 99280 ALUTs combinacionales. La arquitectura HIERG/RTC es claramente la que se empleó.

Tabla 7: Comparación de divisores de 128 bits

	DIV 128 BITS - HIERG/RTC	DIV 128 BITS - HIERG/RTB	DIV 128 BITS - HIERG/RTA	DIV 128 BITS - FLAT
ALUT combinacionales	31,914	65,092	104,475	104,475

4.21 CONTADORES

Los contadores y/o los divisores de frecuencia son componentes codificados bajo un mismo modelo comportamental donde los todos los contadores cuentan con señal de habilitación síncrona, reset asíncrono y señal de sincronización de salida que es activada por el módulo del contador. Los contadores utilizados en esta arquitectura se presentan en la tabla 8 numerados en relación a las figuras 10, 19 y 25 que son los

diagramas de componentes de los bloques UNIDAD_WSM_CRIT_MAR, UNIDAD_WSM y UNIDAD_FREC. El parámetro RAM_L_ADDR_BITS es definido en la sección 4.4 donde se muestran los parámetros del diseño.

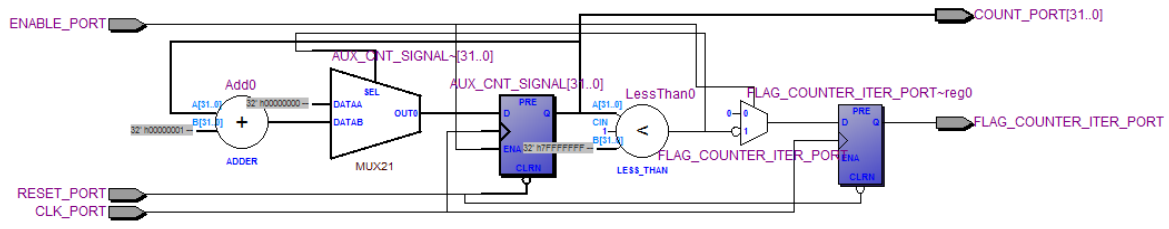
Tabla 8: Características de los contadores utilizados.

Contador	# bits	Modulo	Nombre del contador
1	4	8	COUNTER_CLK
2	RAM_L_ADDR_BITS	(FILAS*3)-1	COUNTER_RAM_L
3	4	8	COUNTER_CLK
4	32	2147483647	COUNTER_ITER
5	4	8	COUNTER_CLK
6	RAM_L_ADDR_BITS	(FILAS*3)-1	COUNTER_RAM_L
7	RAM_L_ADDR_BITS	FILAS-1	COUNTER_RAM_X

Todos los contadores tienen el mismo modelo comportamental y por lo tanto se muestra en el anexo 3, para ilustrar, únicamente el código VHDL del contador denominado COUNTER_RAM_L que tiene como señal de conteo a COUNT_PORT que es de un tipo personalizado denominado *tipo_addr_RAML* que corresponde a un tipo *std_logic_vector(RAML_ADDR_BITS downto 0)* y la señal de sincronización FLAG_COUNTER_RAML_PORT que indica cuando se cumple la cuenta. Por lo tanto, este contador es de RAML_ADDR_BITS bits con módulo (FILAS*3)-1.

En la figura 32 se observa los componentes de un contador. Los demás contadores corresponden al mismo modelo comportamental y por lo tanto son casi idénticos donde solamente se diferencian en la constante de comparación y la cantidad de bits de los componentes (sumador, multiplexor, comparador, registro y puerto de salida).

Figura 32: Hardware de Contador ITER



4.22 UNIDADES DE CONTROL

Las unidades de control están encargadas de leer señales de sincronización y datos para generar diversas señales de control (y datos) para todas las entidades en el datapath. Son circuitos secuenciales que están conformados por una lógica combinacional y una lógica de salida cada una de estas codificadas con un procedimiento en el código VHDL. La lógica secuencial es la que determina las transiciones de estado que se asignan bajo la condición de flanco de reloj, y la lógica combinacional determina las asignaciones correspondientes a las salidas que dependen únicamente del valor de la señal de estado.

Como se ilustró anteriormente, la arquitectura general del sistema se realizó con tres unidades principales: UNIDAD_WSM_CRIT_MAR de mayor jerarquía y los bloques UNIDAD_WSM y UNIDAD_FREC de menor jerarquía una de otra. Cada una de estas tres unidades tiene su unidad de control dado que es conveniente algún grado de modularización en diseños muy extensos y es de gran utilidad en la etapa de desarrollo porque cuando se modifica el datapath de algunas de estas entidades sólo hay que modificar el control correspondiente y los demás controladores siguen con su correcta funcionalidad sin cambios significativos en sus diagramas de estado.

a) Controlador de UNIDAD_WSM_CRIT_MAR: Control_1

La unidad denominada CONTROL_1 o en ocasiones CONTROL_WSM_CRIT_MAR está encargada del control de la entidad de mayor jerarquía del sistema. Es una FSM tipo Moore. Este controlador está compuesto por nueve estados que son descritos a continuación. Se puede complementar esta información con la secuencia de pasos dada para el funcionamiento del componente UNIDAD_WSM_CRIT_MAR de la sección 4.6 y con el diagrama ASM visto en el anexo 4.

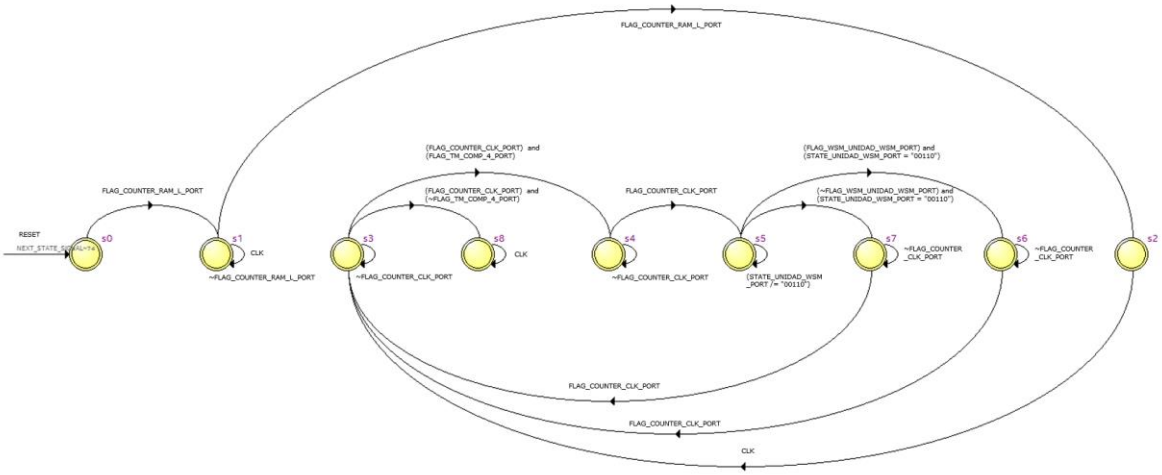
- **Estado s0:** es el estado inicial o IDLE. En este estado se resetean prácticamente todos los componentes del sistema. Al momento de resetear el componente UNIDAD_WSM se resetean también las otras dos máquinas de estado CONTROL_WSM y CONTROL_FREC, dado que la que tiene mayor jerarquía es CONTROL_WSM_CRIT_MAR. Se resetean de la misma manera los registros, el reloj de tiempo de real, la unidad COMPARADOR_4 y la unidad PROMEDIO y los contadores.

- **Estado s1:** el siguiente paso es cargar el registro REG_IN_SERIE_OUT_PARALELO con los valores almacenados en la memoria ROM. Esto se realiza secuencialmente por medio del único puerto de salida de la memoria conectada directamente al registro. El barrido de la memoria se realiza con ayuda de del contador_2 o también llamado COUNTER_RAM_L el cual se configura con un módulo igual $3 * n$.
- **Estado s2:** de modo paralelo, en un único ciclo de reloj, se carga la información almacenada en el registro REG_IN_SERIE_OUT_PARALELO en el registro ROM_PARALELO, que están conectados directamente. Al mismo tiempo se cargan los valores 2 y 4 en los flip flops TL y TR respectivamente.
- **Estado s3:** se cargan los registros TR y TL a la unidad COMPARADOR_4 y también al COMPARADOR_3 como anticipación del próximo estado. El flujo de datos no se va afectado por esta maniobra.
- **Estado s4:** el resultado o la salida del componente COMPARADOR_3 se carga en el registro TM.
- **Estado s5:** el componente UNIDAD_WSM recibe la carga de los valores en los registros TM y ROM_PARALELO. Este último ya tiene almacenado la lista correspondiente desde el estado s2.
- **Estado s6:** una vez finalizado el procesamiento de datos por parte del componente UNIDAD_WSM que demora unos cuantos ciclos de reloj (aprox. ocho ciclos), la señal resultante de esta unidad es una señal de sincronización que determina si se hace efectiva la transición al estado s6 o al s7. En el estado s6 se carga el registro TM en el registro TL.
- **Estado s7:** en este estado se carga el registro TM al registro TR.
- **Estado s8:** este es el estado final del algoritmo. Simplemente tiene la función de entregar la salida de los registros TM, TR y TL para definir el resultado satisfactorio o insatisfactorio del cálculo. Dado que se implementó un reset global asíncrono se puede reiniciar el algoritmo activando esta señal asignada a un botón en la FPGA. No es un reinicio de la FPGA donde requiera una nueva programación, es un reinicio del algoritmo activado por el usuario.

En la figura 33 se presentan el diagrama de estados de CONTROL_1 y sus transiciones de estado. En estas últimas se observan que algunas transiciones presentan la condición de flanco de reloj (CLK), esto quiere decir que no se requiere ninguna condición para la transición de estado y

esto significa que la transición se realiza bajo ninguna condición extra al flanco del reloj. La condición inicial de transición para ir al estado s0 es el reset principal proveniente o asignado a un botón de la tarjeta.

Figura 33: Diagrama de estados de la FSM 1



b) Controlador de UNIDAD_WSM: Control_2

La unidad denominada CONTROL_2 o en ocasiones CONTROL_WSM está encargada del control de la entidad UNIDAD_WSM. Es también una FSM tipo Moore y está compuesto por siete estados. Se puede complementar esta información con la secuencia de pasos dada para el funcionamiento del componente UNIDAD_WSM de la sección 4.13 y con el diagrama ASM visto en el anexo 5.

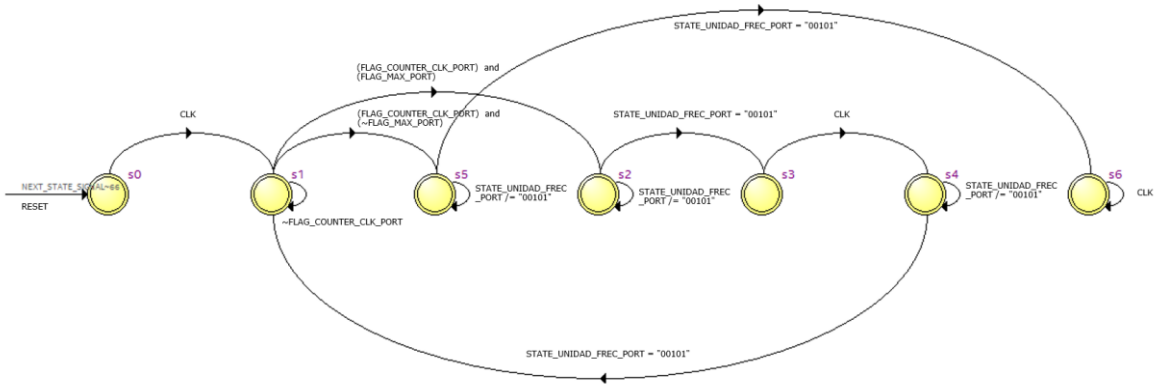
- **Estado s0:** por motivo de que esta máquina de estado es de menor jerarquía y es gobernada por el controlador_1, el estado s0 del controlador_2 no es un estado IDLE ya que durante la ejecución del algoritmo se requiere del reseteo del componente UNIDAD_WSM, que se realiza con un reset virtual. Gracias a esto hay algunos registros que no se resetean, y esto se identificó como una falla de diseño, mas, sin embargo, el algoritmo se ejecuta correctamente y se deja la corrección de la falla del diseño para un trabajo futuro. En este estado se carga la lista almacenada en el registro ROM_PARALELO que tiene el mismo nombre que el registro encontrado en UNIDAD_WSM_CRIT_MAR. Lo anterior es un proceso redundante que gasta innecesariamente recursos de memoria sin

embargo se realizó de esta manera para asegurarse que el componente UNIDAD_WSM reciba la lista y que se pueda controlar el registro ROM_PARALELO dentro de éste por medio de su controlador_2. En una situación análoga se carga el valor del registro TM encontrado en UNIDAD_WSM_CRIT_MAR en el registro TM situado en UNIDAD_WSM. Finalmente se inicializan el registro X_EVEN con valores nulos y el registro X_OLD con unos (representados en el formato punto fijo definido para este trabajo de grado). También se anticipa el próximo estado cargando X_EVEN y X_OLD en la unidad COMPARADOR_1.

- **Estado s1:** en este estado comienza el interior del bucle *while()* de Matlab® que se encuentra en la función $wsm()$. Se cargan los valores de los registros X_EVEN y X_OLD en la unidad COMPARADOR_1. Como ya se mencionó, esta carga se realiza de manera paralela y anticipada desde el anterior estado.
- **Estado s2:** en este estado, el valor del registro X_EVEN se carga en el registro X_OLD y también se cargan el valor del flip flop TM, la lista almacenada en ROM_PARALELO y el registro X_EVEN al componente UNIDAD_FREC. En este mismo estado se asigna la salida de UNIDAD_FREC al registro X_ODD.
- **Estado s3:** activación del reset del bloque UNIDAD_FREC con el fin de resetear el controlador_3 a su estado s0.
- **Estado s4:** cargar a UNIDAD_FREC los valores de X_EVEN, TM y la lista almacenada en ROM_PARALELO. La salida de UNIDAD_FREC se asigna al registro X_EVEN. También se aumenta la cuenta del contador ITER el cual realiza el conteo de iteraciones dentro del bucle de control *while()* situado en la función $wsm()$ de Matlab®. Lo último se realiza por medio del CLK de este contador.
- **Estado s5:** en este estado ya estamos afuera del bucle *while()* ya mencionado y se debe cargar nuevamente el componente UNIDAD_FREC con los valores de X_EVEN, TM y ROM_PARALELO. Su salida es cargada al COMPARADOR_2 junto con el valor del registro X_EVEN.
- **Estado s6:** este el estado final de control_2 y tiene como finalidad entregar como salida la señal de sincronización generada por la unidad COMPARADOR_2 que se da por medio de un flip flop. El reinicio del controlador_2 se realiza de manera virtual activando su señal reset asíncrona por medio del controlador_1.

El diagrama de estado del control_2 se expone en la figura 34.

Figura 34: Diagrama de estados de la FSM 2



c) Controlador de UNIDAD_FREC: Control_3

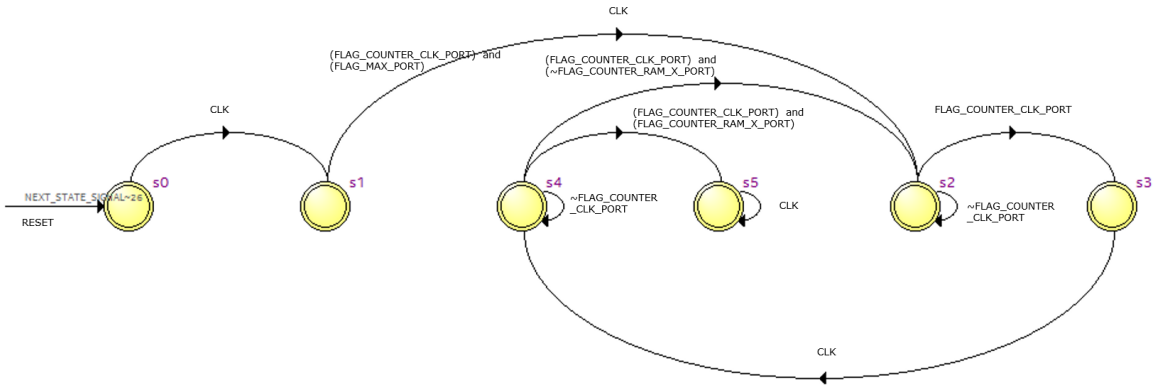
La entidad denominada CONTROL_3 o también CONTROL_FREC está encargada del control de la entidad UNIDAD_FREC. Es una máquina de estados tipo Moore compuesta por seis estados que se describen a continuación. Observamos el diagrama de estados en la figura 35. Se puede complementar esta información con la secuencia de pasos dada el funcionamiento del componente UNIDAD_WSM de la sección 4.17 y con el diagrama ASM vista en el anexo 6.

- **Estado s0:** estado inicial de la función $fRec()$ visto de la perspectiva de Matlab®. De manera similar a la redundancia impuesta en el control_2, se carga el registro ROM_PARALELO situado en UNIDAD_FREC con los valores del registro del mismo nombre situado en el componente UNIDAD_WSM. También se carga el registro RAM_X_PARALELO_MAS1 con el valor en el puerto de entrada X_PORT y su último componente se asigna con un valor 1 en el formato dado.
- **Estado 1:** En este estado se hace la operación de punteros codificada como $x(L)$ en lenguaje Matlab. Se realiza un "barrido" del registro ROM_PARALELO con el fin de cargar o llenar el registro RAM_L_PARALELO con los datos almacenados en RAM_X_PARALELO_MAS1 con las direcciones dadas por la lista en ROM_PARALELO. Este barrido en realidad es totalmente paralelo y se realiza en un sólo ciclo de reloj. Este paso se realiza por medio de un procedimiento concurrente llamado *fill_array()* fijado en el

paquete PARAMETROS.vhd por medio del cual la herramienta de síntesis gasta enormes cantidades de ALUTs combinacionales dentro del componente CONTROL_3.

- **Estado s2:** Teniendo en el registro RAM_L_PARALELO una matriz del mismo tamaño de la lista, en este estado se debe cargar una fila (los primeros tres elementos) dentro de RAM_L_PARALELO en el componente UNIDAD_FILA.
- **Estado s3:** Se carga un componente (o fila) del registro RAM_X_PARALELO ubicado en UNIDAD_FREC con el valor de salida del componente UNIDAD_FILA. Lo anterior se realiza después de unos cuantos ciclos de reloj para estabilizar la respuesta de UNIDAD_FILA haciendo que sus entradas no varíen con ayuda del contador CLK como ya ha mencionado anteriormente.
- **Estado s4:** Aumentamos la cuenta del contador COUNTER_RAM_X que tiene configurado un módulo de valor n. Como lo dice su nombre, este contador tiene como objetivo contar los elementos dentro del registro RAM_X_PARALELO.
- **Estado s5:** cuando el registro RAM_X_PARALELO ya esté completamente lleno, en este estado se entrega la salida de este registro como un vector. El reinicio del controlador_3 se realiza de manera virtual activando su señal reset asíncrona por medio del controlador_2.

Figura 35: Diagrama de estados de la FSM 3.



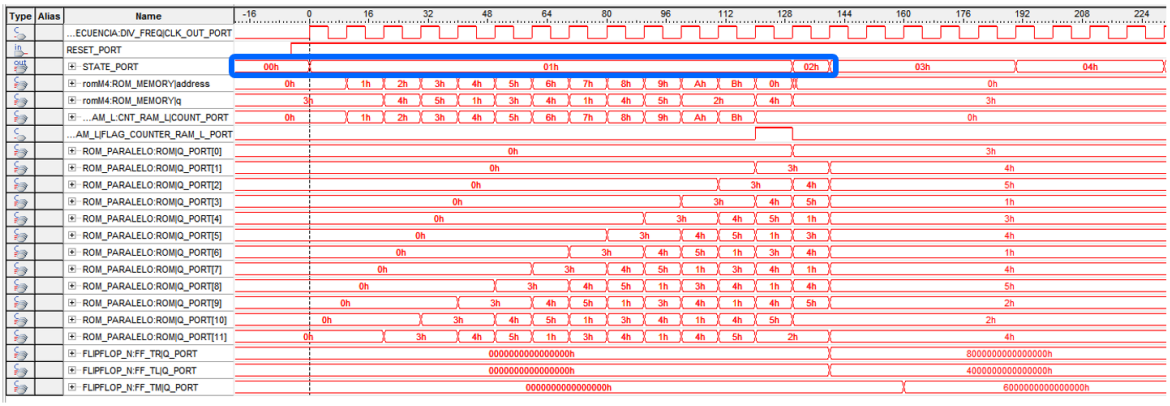
4.23 PRUEBAS DE FUNCIONAMIENTO DEL SISTEMA TOTAL PARA LA LISTA L4

Con el fin de probar el funcionamiento del procesamiento de la lista L4 se sintetiza UNIDAD_WSM_CRIT_MAR.vhd sobre la FGPA y se verifica el funcionamiento de la unidad monitoreando las señales en relación a los estados del controlador_1. Como esta es una prueba de exclusivamente de funcionamiento se configura el programa SignalTap® para tomar muestras siempre a 100 MHz y se hace trabajar al sistema con un reloj de 50 MHz para estas primeras pruebas. Más adelante se trabaja el diseño con relojes más lentos.

A continuación, se describen las pruebas de funcionamiento de los estados s0, s1, s2, s3, s4 y s5 del control_1 con la señal CLK_OUT siendo el reloj del diseño de 50 MHz. Sin embargo, para los estados s6, s7 y s8 se trabaja con un reloj de 5 MHz para la lista L4 y con uno de 2.5 MHz para la lista L6.

La figura 36 permite observar la primera señal denominada STATE_PORT, la cual es la señal del estado presente del controlador 1 se verifica estado por estado el funcionamiento de la unidad.

Figura 36: Prueba de funcionamiento – Lista L4 – Estados s0, s1, s2



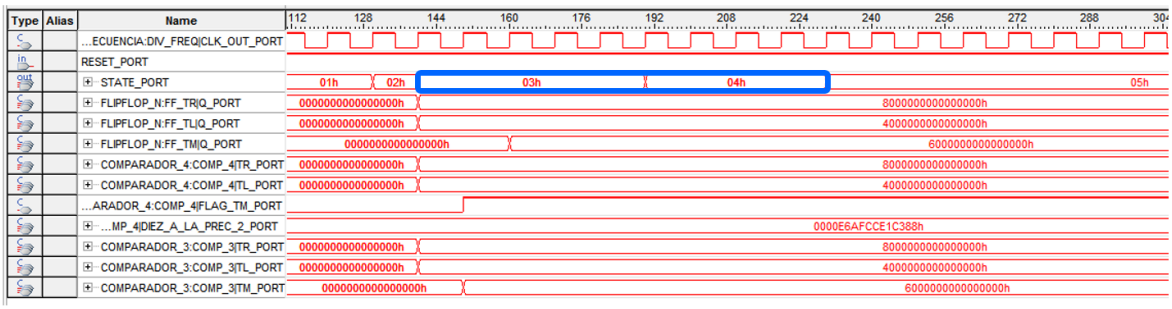
- **Estado s0:** Estado IDLE donde se activan los resets de todos los registros, los contadores y demás unidades. -> Se observa en la fig. 36 que todos los valores iniciales vistos, por lo menos, en la fig. 36, son cero y la señal RESET es la que gobierna el comienzo del algoritmo.

- **Estado s1:** Estado donde se carga en su totalidad el registro REG_IN_SERIE_OUT_PARALELO con los valores de la matriz L almacenados en la memoria ROM. -> Se evidencia en la fig. 36 que los valores de la señal Q de la memoria son cargados secuencialmente en el registro paralelo dados en la señal ROM_PARALELO [11..0].

- **Estado s2:** Estado donde se carga el registro ROM_PARALELO con los datos del registro REG_IN_SERIE_OUT_PARALELO y se cargan también TR y TL con sus valores iniciales, 4 y 2 respectivamente en el formato definido (4=8000000000000000h y 2=4000000000000000h). -> Como los registros REG_IN_SERIE_OUT_PARALELO están conectados paralelamente a los registros de ROM_PARALELO entonces al pasar al estado s2 se carga completamente la matriz L en ROM_PARALELO y se desactiva el contador COUNTER_RAM_L que gobierna el barrido de la memoria para finalizar esta carga.

- **Estado s3:** Se cargan los valores de TR y TL al COMPARADOR_4 y al COMPARADOR_3. –> En la figura 37 se observa que en el estado s3 se cargan los valores de los flip flops TR y TL en las entradas del COMPARADOR_4.

Figura 37: Prueba de funcionamiento – Lista L4 – Estados s3, s4

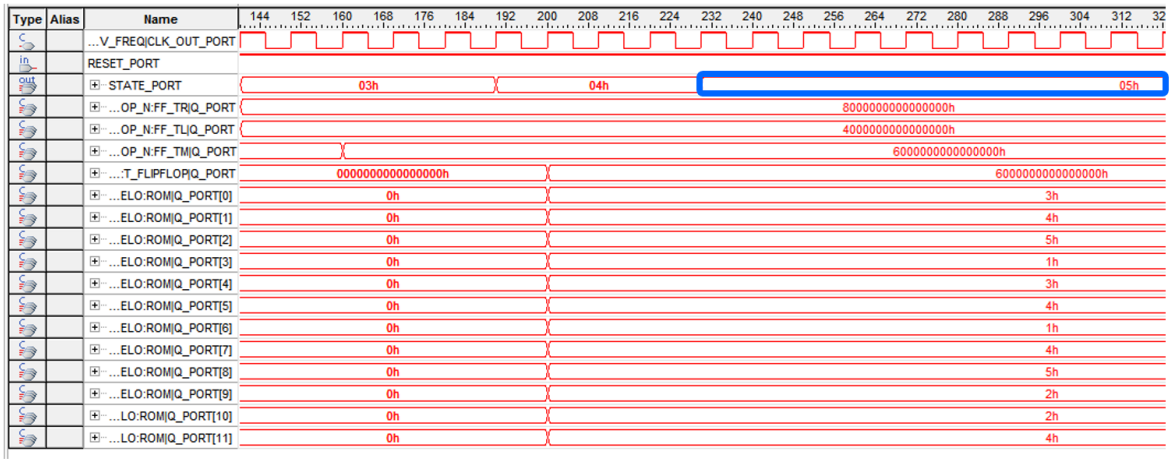


- **Estado s4:** Ya cargados con anticipación los valores de TR y TL al COMPARADOR _3, se carga la salida de esta unidad al registro TM. –> en la figura 37 se ve que los valores de TR y TL se cargan en la unidad COMPARADOR_3 en el estado s3 o hasta s2, sin embargo, esto no afecta al desarrollo del algoritmo. Esto no afecta el flujo de datos del algoritmo, pero es necesario realizar en un estado definido (s4), la mencionada acción de carga al registro TM ya que el

controlador_1 gobierna el controlador_2 y define asignaciones de señales que tienen que ser definidas en un estado determinado.

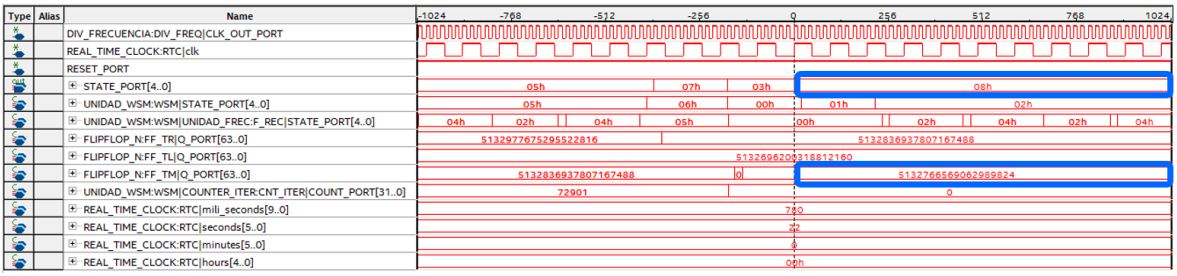
- **Estado s5:** Cargar el registro TM y la lista L en UNIDAD_WSM. –> Se observa en la figura 38 que se cargan los valores de la matriz L en el registro ROM_PARALELO que está dentro UNIDAD_WSM. Esta carga se puede anticipar en estados anteriores por eso se observa la carga en el estado s4. La carga del valor del flip flop TM también se realiza en estados anteriores al s5 en este caso en el estado s3. Esto tampoco afecta la ejecución del algoritmo, pero es necesario realizar un estado definido para realizar las mencionadas acciones de carga ya que el controlador_1 gobierna el controlador_2 y define asignaciones de señales que tienen que ser definidas en un estado determinado.

Figura 38: Prueba de funcionamiento – Lista L4 – Estado s5



En la figura 39 se observan tres señales llamadas STATE_PORT las cuales son las señales de estado presente de los controladores 1, 2 y 3. Las señales de salida de los flip flops TR, TL, y TM cada una de 64 bits se presentan como los resultados principales para corroborar el funcionamiento correcto del sistema. La señal de salida COUNT_PORT del contador COUNTER_ITER muestra el número de iteraciones que se realizan dentro de la función $wsm()$. Las señales de la entidad REAL_TIME_CLOCK sirven para medir el tiempo de ejecución del algoritmo. Se observa el reloj que gobierna a esta entidad que fue configurado con una frecuencia de 1 MHz= 1us y el reloj del diseño fue de 13 MHz en esta figura en particular.

Figura 39: Prueba de funcionamiento – Lista L4 – Estado s8 – 4 cifras significativas



- **Estado s6:** Cargar el registro TM en el registro TL. –> Este estado es una transferencia entre registros y se omite su prueba de funcionamiento.

- **Estado s7:** Cargar el registro TM en el registro TR. –> Este estado es una transferencia entre registros y se omite su prueba de funcionamiento.

- **Estado s8:** Estado final del algoritmo. Entrega la salida de los registros TR, TL y TM. –> En este estado se prueba que el algoritmo funciona correctamente para la lista L4. Se observa en la figura 39 que el resultado dado en el registro TM en el estado s8 es 5132766569062989824_{10} el cuál es el valor correcto si se realiza la conversión al formato dado en la ecuación 13 y se compara con el resultado dado por el algoritmo en Matlab® mostrado en la tabla 9.

$$\frac{5132766569062989824_{10}}{2^{61}} = 2.22598266602_{10}$$

Tabla 9: Ejecución de la Lista L4 en Matlab® - 4 cifras significativas

BINARY SEARCH	tl	tr	iters	TIEMPO (seg)
	2.2227	2.2266	28900	0.956123
	2.2247	2.2266	52971	1.750066
	2.2257	2.2266	94826	3.122401
	2.2257	2.2262	71746	2.372702
	2.2260	2.2262	81606	2.696679
Respuesta	2.2260		Tiempo Total	13.010177

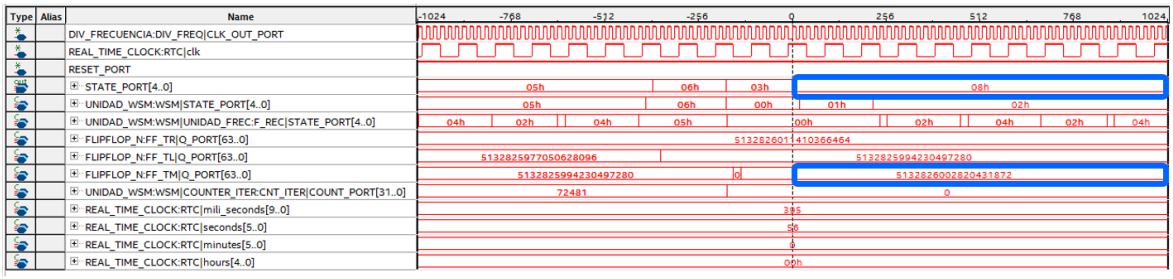
En este caso se utilizó una precisión de cuatros cifras significativas ($prec = 4$) y por tanto el valor el correcto se presenta hasta cuatro cifras significativas. Se recalca también la implementación del reloj de tiempo

de real que da la información de 00:00:22:780 segundos que demoró la ejecución del algoritmo de la lista L4 para este caso en particular.

En la tabla 9 y todas las demás tablas relacionadas con las pruebas realizadas desde Matlab® se presentan únicamente las últimas cinco iteraciones del ciclo *while()* que está dentro de la función *wsmCritMar()* ya que los datos arrojados por el programa son muy numerosos y solamente se quiere dar una idea de los cálculos parciales hechos por Matlab®, los resultados finales de TM y el tiempo total de ejecución.

Modificando el valor de la constante $prec$ que afecta únicamente a la unidad COMPARADOR_4 existe la posibilidad de aumentar la precisión del resultado. La figura 40 muestra que efectivamente el cálculo realizado con la unidad con $prec = 8$ tiene más cifras significativas. La tabla 10 muestra los resultados en Matlab con el mismo valor de cifras significativas.

Figura 40: Prueba de funcionamiento – Lista L4 – 8 cifras significativas



$$\frac{5132826002820431872_{10}}{2^{61}} = 2.2260084413_{10}$$

Tabla 10: Ejecución de la Lista L4 en Matlab® - 8 cifras significativas

Binary search	tl	tr	iters	Tiempo (seg)
	2.22600842	2.22600890	72500	2.358916
	2.22600842	2.22600866	72489	2.402638
	2.22600842	2.22600854	72484	2.364404
	2.22600842	2.22600848	72482	2.576877
	2.22600845	2.22600848	72484	2.573678
Respuesta	2.22600847		Tiempo Total	49.622669

A continuación, se expone en la tabla 11 un resumen de los recursos utilizados en la tarjeta Arria II GX. Se distingue entre los recursos del diseño por si solo y el consumo del sistema cuando se adiciona elementos que se requieren para visualizar el procesamiento como SignalTap® o una entidad que controla un LCD que se trató se implementar. Se observa que el recurso más consumido son los ALUTs combinacionales y que la parte de SignalTap® eleva bastante los recursos de memoria sin llegar nunca a consumir el 100% de la memoria disponible. Los 18 pines de E/S son dados por los pines del LCD, el CLK diferencial de 100 MHz y el reset. En la tabla 12 aparecen los detalles de los recursos consumidos diferenciando las entidades que los utilizan. Esta última tabla que la que nos permite determinar más adelante que el problema del diseño estuvo en las entidades que crecen con n: los comparadores 1 y 2, y el controlador_3.

Tabla 11: Recursos Totales – Lista L4

Recurso	Signal Tap	No Signal Tap, No LCD
Total ALUTs	21488	20687
- ALUTs combinacionales	21488	20687
- ALUTs con memoria	0	0
Total Registros	6636	2925
- Registros lógicos dedicados	6636	2925
- Registros de E/S	0	0
Total Pines E/S	18	18
Bits de memoria	275492	36
DSP 18-bit	48	48

Tabla 12: Recursos por componentes – Lista L4

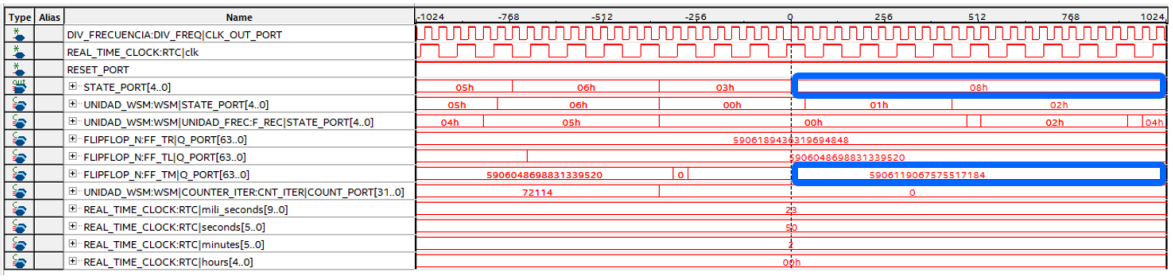
Recurso	LC combinacionales	LC registros	Bits de memoria	DSP 18-bit	DSP 36x36	Pines
UNIDAD_WSM_CRIT_MAR	21488 (1)	6636 (0)	275492	48	12	18
COMPARADOR 3	64 (64)	63 (63)	0	0	0	0
COMPARADOR 4	81 (81)	1 (1)	0	0	0	0
CONTROL UNIDAD_WSM_CRIT_MAR	215 (215)	9 (9)	0	0	0	0
COUNTER_CLK	5 (5)	5 (5)	0	0	0	0
COUNTER_RAM_L	5 (5)	5 (5)	0	0	0	0
DIV_FRECUENCIA	5 (5)	5 (5)	0	0	0	0
FLIP FLOP TL	0 (0)	63 (63)	0	0	0	0
FLIP FLOP TM	0 (0)	63 (63)	0	0	0	0
FLIP FLOP TR	0 (0)	64 (64)	0	0	0	0
REAL_TIME_CLOCK	59 (59)	45 (45)	0	0	0	0
REG_IN_SERIE_OUT_PARALELO	0 (0)	36 (0)	0	0	0	0
ROM_PARALELO	0 (0)	3 (0)	0	0	0	0
MEMORIA ROM	0 (0)	0 (0)	36	0	0	0
LCD_EXAMPLE	150 (14)	58 (12)	0	0	0	0

- LCD_CONTROLLER	136 (136)	46 (46)	0	0	0	0
UNIDAD_WSM	20252 (0)	2563 (0)	0	48	12	0
- COMPARADOR 1	1335 (1335)	1 (1)	0	0	0	0
- COMPARADOR 2	777 (777)	1 (1)	0	0	0	0
- CONTROL UNIDAD_WSM	1038 (1038)	7 (7)	0	0	0	0
- COUNTER_CLK	4 (4)	5 (5)	0	0	0	0
- COUNTER_ITER	32 (32)	32 (32)	0	0	0	0
- FLIP FLOP T	0 (0)	63 (63)	0	0	0	0
- RAM_X_ODD	0 (0)	256 (0)	0	0	0	0
- RAM_X_EVEN	0 (0)	256 (0)	0	0	0	0
- RAM_X_OLD	0 (0)	256 (0)	0	0	0	0
- ROM_PARALELO	0 (0)	36 (0)	0	0	0	0
- **UNIDAD_FREC**	17066 (0)	1650 (0)	0	48	12	
○ CONTROL_UNIDAD_FREC	1373 (1373)	6 (6)	0	0	0	0
○ COUNTER_CLK	4 (4)	5 (5)	0	0	0	0
○ COUNTER_RAM_X	2 (2)	3 (3)	0	0	0	0
○ RAM_L	0 (0)	768 (0)	0	0	0	0
○ RAM_X	0 (0)	256 (0)	0	0	0	0
○ RAM_X_MAS1	0 (0)	257 (0)	0	0	0	0
○ ROM_PARALELO	0 (0)	36 (0)	0	0	0	0
○ **UNIDAD_FILA**	15687 (4)	319 (319)	0	48	12	0
▪ MULT 0	89 (0)	0 (0)	0	16	4	0
▪ MULT 1	89 (0)	0 (0)	0	16	4	0
▪ MULT 2	89 (0)	0 (0)	0	16	4	0
▪ DIV	15416 (0)	0 (0)	0	0	0	0
SIGNAL TAP HUB	92 (1)	91 (0)	0	0	0	0
SIGNAL TAP AUTO	559 (2)	3562 (538)	275456	0	0	0

4.24 PRUEBAS DE FUNCIONAMIENTO - LISTA L6

Se realiza la misma metodología para probar el funcionamiento de la ejecución la lista L6. Observamos en la figura 41 el resultado dado con un valor de $prec = 4$. Y este resultado lo comparamos con la respuesta dada por el algoritmo en Matlab® mostrada en la tabla 13. Vemos como coinciden dentro de 4 cifras significativas. Notamos una demora de ejecución de 00:02:50:023 minutos en la FPGA.

Figura 41: Prueba de funcionamiento – Lista L6 – 4 cifras significativas



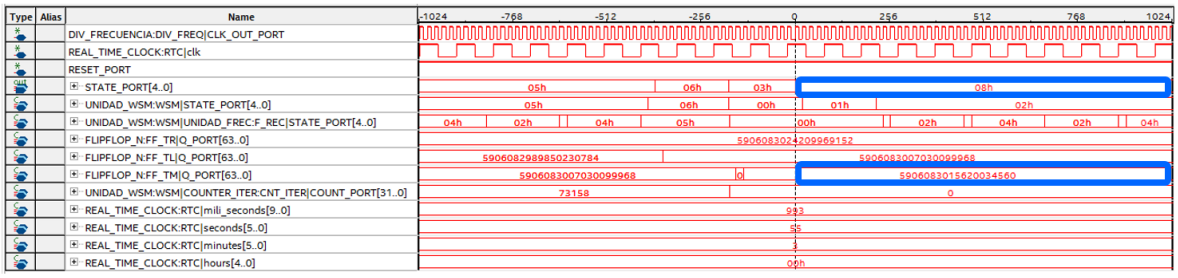
$$\frac{5906119067575517184_{10}}{2^{61}} = 2.56137084961_{10}$$

Tabla 13: Ejecución de la Lista L6 en Matlab® - 4 cifras significativas

BINARY SEARCH	tl	tr	iters	TIEMPO (seg)
	2.5586	2.5625	42700	1.486269
	2.5606	2.5625	96408	3.349985
	2.5606	2.5616	58746	2.032180
	2.5611	2.5616	76495	2.970212
	2.5611	2.5614	69439	2.680334
Respuesta	2.5613		Tiempo Total	17.955571

La figura 42 muestra la prueba de funcionamiento de la ejecución de la lista L6 con el valor de $prec = 8$ y se obtiene de esta manera más cifras significativas como se puede corroborar en la tabla 14.

Figura 42: Prueba de funcionamiento – Lista L6 – 8 cifras significativas



$$\frac{5906083007030099968_{10}}{2^{61}} = 2.56135521457_{10}$$

Tabla 14: Ejecución de la Lista L6 en Matlab® - 8 cifras significativas

BINARY SEARCH	tl	tr	iters	TIEMPO (seg)
	2.56135513	2.56135560	73167	2.744169
	2.56135513	2.56135536	73159	2.657702
	2.56135524	2.56135536	73163	2.673516
	2.56135524	2.56135530	73161	2.648423
	2.56135524	2.56135527	73161	2.712181
Respuesta	2.56135526		Tiempo Total	52.075375

Se exhibe en la tabla 15 el resumen de recursos total consumidos por el diseño para la ejecución de L6. En la tabla 16 se puede observar los recursos por componentes que permiten ver donde se ha presentado el mayor crecimiento en el consumo de recursos en comparación al diseño para la ejecución de la lista L4. Como ya se ha mencionado algunas veces, las entidades que crecen con n son las entidades que consumen los ALUTs combinacionales y no permiten sintetizar las demás listas L8, L10, etc. Se observa en la tabla 15 que el crecimiento del consumo de ALUTs es significativo respecto a la primera lista, debido a que pasa de 20687 a 54120 y el crecimiento de bits de memoria es poco ya que de 36 pasa a 270 que es insignificante frente a los 6.7 GB que existen disponibles de RAM. En la tabla 16 se puede remarcar que la entidad CONTROL_FREC con 23618 es la entidad que más modifica su consumo de ALUTs.

Tabla 15: Recursos Totales – Lista L6

Recurso	Signal Tap	No Signal Tap, No LCD
Total ALUTs	54888	54120
- ALUTs combinacionales	54888	54120
- ALUTs con memoria	0	0
Total Registros	14513	10802
- Registros lógicos dedicados	14513	10802
- Registros de E/S	0	0
Total Pines E/S	18	18
Bits de memoria	275726	270
DSP 18-bit	48	48

Tabla 16: Recursos por componentes – Lista L6

Recurso	LC combinacionales	LC registros	Bits de memoria	DSP 18-bit	DSP 36x36	Pines
UNIDAD_WSM_CRIT_MAR	54888 (1)	14513 (0)	275726	48	12	18
COMPARADOR 3	64 (64)	63 (63)	0	0	0	0
COMPARADOR 4	81 (81)	1 (1)	0	0	0	0
CONTROL UNIDAD_WSM_CRIT_MAR	217 (217)	9 (9)	0	0	0	0
COUNTER_CLK	5 (5)	5 (5)	0	0	0	0
COUNTER_RAM_L	9 (9)	7 (7)	0	0	0	0
DIV_FRECUENCIA	5 (5)	5 (5)				
FLIP FLOP TL	0 (0)	63 (63)	0	0	0	0
FLIP FLOP TM	0 (0)	63 (63)	0	0	0	0
FLIP FLOP TR	0 (0)	64 (64)	0	0	0	0
REAL_TIME_CLOCK	59 (59)	45 (45)				
REG_IN_SERIE_OUT_PARALELO	0 (0)	270 (0)	0	0	0	0
ROM_PARALELO	0 (0)	5 (0)	0	0	0	0
MEMORIA ROM	0 (0)	0 (0)	270	0	0	0
LCD_EXAMPLE	150 (14)	58 (12)	0	0	0	0
- LCD_CONTROLLER	136 (136)	46 (46)	0	0	0	0
UNIDAD_WSM	53646 (0)	10202 (0)	0	48	12	0

- COMPARADOR 1	6169 (6169)	1 (1)	0	0	0	0
- COMPARADOR 2	3503 (3503)	1 (1)	0	0	0	0
- CONTROL UNIDAD_WSM	4622 (4622)	7 (7)	0	0	0	0
- COUNTER_CLK	4 (4)	5 (5)	0	0	0	0
- COUNTER_ITER	32 (32)	32 (32)				
- FLIP FLOP T	0 (0)	63 (63)	0	0	0	0
- RAM_X_ODD	0 (0)	1152 (0)	0	0	0	0
- RAM_X_EVEN	0 (0)	1152 (0)	0	0	0	0
- RAM_X_OLD	0 (0)	1152 (0)	0	0	0	0
- ROM_PARALELO	0 (0)	270 (0)	0	0	0	0
- **UNIDAD_FREC**	39316 (0)	6367 (0)	0	48	12	
o CONTROL_UNIDAD_FREC	23618 (23618)	6 (6)	0	0	0	0
o COUNTER_CLK	4 (4)	5 (5)	0	0	0	0
o COUNTER_RAM_X	7 (7)	6 (6)	0	0	0	0
o RAM_L	0 (0)	3456 (0)	0	0	0	0
o RAM_X	0 (0)	1152 (0)	0	0	0	0
o RAM_X_MAS1	0 (0)	1153 (0)	0	0	0	0
o ROM_PARALELO	0 (0)	270 (0)	0	0	0	0
o **UNIDAD_FILA**	15687 (4)	319 (319)	0	48	12	0
▪ MULT 0	89 (0)	0 (0)	0	16	4	0
▪ MULT 1	89 (0)	0 (0)	0	16	4	0
▪ MULT 2	89 (0)	0 (0)	0	16	4	0
▪ DIV	15416 (0)	0 (0)	0	0	0	0
SIGNAL TAP HUB	92 (1)	91 (0)	0	0	0	0
SIGNAL TAP AUTO	559 (2)	3562 (538)	275456	0	0	0

4.25 RESULTADOS DE SÍNTESIS – LISTA L8

Al momento de intentar sintetizar la tercera lista, L8, la herramienta de síntesis permite ver los recursos necesarios para su síntesis, más sin embargo no permite hacer los pasos de localización y ruteo debido a que la FPGA disponible no cuenta con la cantidad de ALUTs requeridos. La tabla 17 permite ver el gran crecimiento de estos ALUTs pues pasa de 54120 a 251755 la cantidad de lógica de la lista L6 a la lista L8. También se recalca el uso casi insignificante de los bits de memoria (936). Como se hizo anteriormente, en la tabla 18 se nota el gran aumento en consumo de la entidad CONTROL_FREC de 193920 ALUTs para este hardware.

Tabla 17: Recursos Totales – Lista L8

Recurso	Signal Tap	No Signal Tap, No LCD
Total ALUTs	252553	251755
- ALUTs combinacionales	252553	251755
- ALUTs con memoria	0	0
Total Registros	33943	30213
- Registros lógicos dedicados	33943	30213
- Registros de E/S	0	0
Total Pines E/S	18	18
Bits de memoria	551848	936
DSP 18-bit	48	48

Tabla 18: Recursos por componentes – Lista L8

Recurso	LC combinacionales	LC registros	Bits de memoria	DSP 18-bit	DSP 36x36	Pines
UNIDAD_WSM_CRIT_MAR	252553 (1)	33943 (0)	551848	48	12	18
COMPARADOR 3	64 (64)	63 (63)	0	0	0	0
COMPARADOR 4	81 (81)	1 (1)	0	0	0	0
CONTROL UNIDAD_WSM_CRIT_MAR	219 (219)	9 (9)	0	0	0	0
COUNTER_CLK	5 (5)	5 (5)	0	0	0	0
COUNTER_RAM_L	11 (11)	9 (9)	0	0	0	0
DIV_FRECUENCIA	6 (6)	6 (6)				
FLIP FLOP TL	0 (0)	63 (63)	0	0	0	0
FLIP FLOP TM	0 (0)	63 (63)	0	0	0	0
FLIP FLOP TR	0 (0)	64 (64)	0	0	0	0
REAL_TIME_CLOCK	58 (58)	45 (45)				
REG_IN_SERIE_OUT_PARALELO	0 (0)	936 (0)	0	0	0	0
ROM_PARALELO	0 (0)	6 (0)	0	0	0	0
MEMORIA ROM	0 (0)	0 (0)	936	0	0	0
LCD_EXAMPLE	140 (14)	58 (12)	0	0	0	0
- LCD_CONTROLLER	126 (126)	46 (46)	0	0	0	0
UNIDAD_WSM	251310 (0)	28943 (0)	0	48	12	0
- COMPARADOR 1	18201 (18201)	1 (1)	0	0	0	0
- COMPARADOR 2	10122 (10122)	1 (1)	0	0	0	0
- CONTROL UNIDAD_WSM	13326 (13326)	7 (7)	0	0	0	0
- COUNTER_CLK	4 (4)	5 (5)	0	0	0	0
- COUNTER_ITER	32 (32)	32 (32)				
- FLIP FLOP T	0 (0)	63 (63)	0	0	0	0
- RAM_X_ODD	0 (0)	3328 (0)	0	0	0	0
- RAM_X_EVEN	0 (0)	3328 (0)	0	0	0	0
- RAM_X_OLD	0 (0)	3328 (0)	0	0	0	0
- ROM_PARALELO	0 (0)	936 (0)	0	0	0	0
- **UNIDAD_FREC**	209625 (0)	17914 (0)	0	48	12	
o CONTROL UNIDAD_FREC	193920 (193920)	6 (6)	0	0	0	0
o COUNTER_CLK	4 (4)	5 (5)	0	0	0	0
o COUNTER_RAM_X	8 (8)	7 (7)	0	0	0	0
o RAM_L	0 (0)	9984 (0)	0	0	0	0
o RAM_X	0 (0)	3328 (0)	0	0	0	0
o RAM_X_MAS1	0 (0)	3329 (0)	0	0	0	0
o ROM_PARALELO	0 (0)	936 (0)	0	0	0	0
o **UNIDAD_FILA**	15693 (4)	319 (319)	0	48	12	0
▪ MULT 0	89 (0)	0 (0)	0	16	4	0
▪ MULT 1	89 (0)	0 (0)	0	16	4	0
▪ MULT 2	89 (0)	0 (0)	0	16	4	0
▪ DIV	15422 (0)	0 (0)	0	0	0	0
SIGNAL TAP HUB	92 (1)	91 (0)	0	0	0	0
SIGNAL TAP AUTO	566 (2)	3581 (538)	550912	0	0	0

4.26 COMPARACION ENTRE LAS SÍNTESIS DE LAS LISTAS L4, L6 Y L8

En la tabla 19 se resume los resultados más relevantes de la compilación de las listas L4, L6 Y L8, enfocándose sobre los recursos requeridos para su síntesis y tiempo de ejecución en la FPGA Arria II GX. La compilación

de las listas L10, L12, L14, L16, L18 no se logró debido a que la herramienta EDA no tiene la capacidad de realizar dicho proceso, generándose un error grave en la etapa de síntesis del quartus.exe (*Analysis & Synthesis*), el cual obliga al reinicio del programa, y por ende no hay información sobre resultados de síntesis.

En la tabla 19 se observa que el número de ALUTs requeridos se incrementa más de un 200% al pasar de la lista L4 a la lista L6 y más de un 500% al pasar de la lista L6 a la lista L8. Este consumo excesivo de hardware es causado por las unidades que tienen como parámetro de síntesis el número de filas n. Se identifica que la unidad que consume más recursos es CONTROL_UNIDAD_FREC (CONTROLADOR_3) y se debe a la instrucción VHDL *fill_array()* genera un ruteo excesivo (ver el código VHDL en la sección 4.4). En la tabla 19 se recalca que las unidades hardware COMP_1 y COMP_2 también aumentan de manera considerable el consumo de recursos en relación al número de filas n.

Tabla 19: Comparación entre listas L4, L6 y L8

CARACTERISTICA	L4	L6	L8
Recursos totales (No signal tap, No LCD)			
- ALUTs combinacionales	20687	54120	252554
- ALUTs con memoria	0	0	0
- Registros lógicos dedicados	2925	10802	33943
- Pines totales	18	18	18
- Bits de memoria	36	270	936
- DSP 18-bit	48	48	48
- DSP 32-bit	12	12	12
Recursos de CONTROL_UNIDAD_FREC			
- ALUTs combinacionales	1373	23618	193920
Recursos de COMPARADOR_1			
- ALUTs combinacionales	1335	6169	18201
Recursos de COMPARADOR_2			
- ALUTs combinacionales	777	3503	10122
Tiempo de ejecución (FPGA Arria II GX 125 - L4 -> [5 Mhz], L6 -> [2,5 MHz])			
- prec = 8, precWSM=12	00:00:56:395	00:03:55:993	-
- prec = 4, precWSM=12	00:00:22:780	00:02:50:023	-
- ANS con prec =8	2.22600844	2.56135521	-
- ANS con prec =4	2.2259	2.5613	-
Tiempo de ejecución (Matlab® - Intel i7, 16GB RAM, 4GHz)			
- prec = 8, precWSM=12	00:00:05:656	00:00:07:730	00:00:07:783
- prec = 4, precWSM=12	00:00:02:337	00:00:02:547	00:00:04:111
- ANS con prec =8	2.22600847	2.56135526	2.76093594
- ANS con prec =4	2.2260	2.5613	2.7609
Tiempo de ejecución (Matlab® - AMD A10, 8GB RAM, 2.5GHz)			
- prec = 8, precWSM=12	00:00:53:951	00:00:54:356	00:00:67:899
- prec = 4, precWSM=12	00:00:22:893	00:00:18:850	00:00:34:523
- ANS con prec =8	2.22600847	2.56135526	2.76093594
- ANS con prec =4	2.2260	2.5613	2.7609

A partir de los datos de la tabla 19, se calculan los errores y las diferencias en tiempo de ejecución de las listas L4 y L6 (únicas listas sintetizadas sobre la FPGA), la comparación se realiza entre la ejecución del algoritmo hardware implementado en la FPGA vs la ejecución en Matlab utilizando dos computadores, el primero basado en Intel i7 y otro basado en AMD A10. Dichos resultados se presentan en la tabla 20.

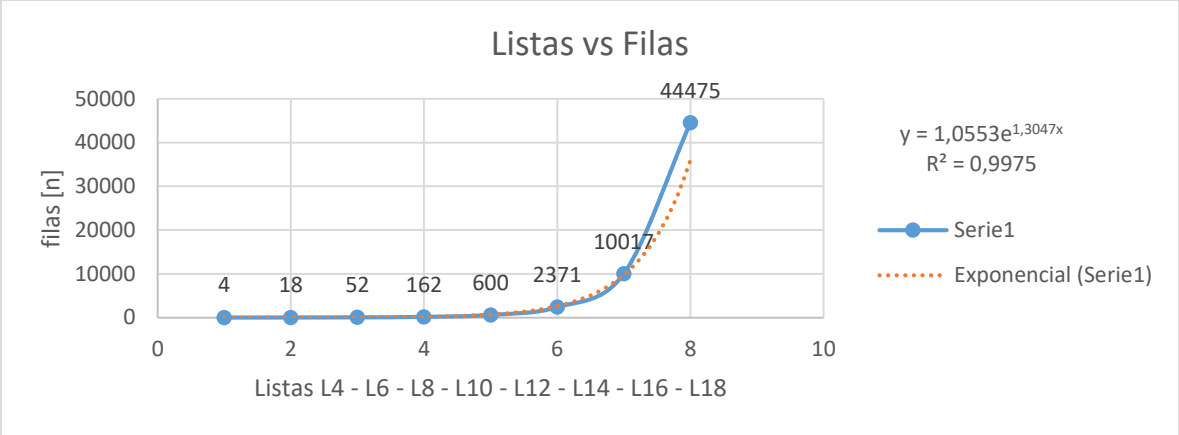
Se observa en la tabla 20 que los errores son casi nulos y se recalca que la magnitud de este error depende de la precisión del resultado dada por el valor $prec$. Como ejercicio se tomaron los valores prec=4 y prec=8. Los tiempos de ejecución de la lista L4 en el AMD A10 fueron muy similares a los de la FPGA con 0.113s y 2.444s de diferencia para los valores de prec=4 y prec=8 respectivamente. La diferencia de tiempos de la lista L6 en el AMD A10 demuestra una desaceleración de ejecución del algoritmo siendo la diferencia de 151s y 18s para los valores de prec=4 y prec=8 respectivamente. En relación a los tiempos de ejecución en el computador con Intel i7 se presentan también una desaceleración significativa para ambas listas L4 y L6. Estos resultados adversos se explican por el uso de relojes muy lentos de 5MHz para L4 y 2.5MHz para L6.

Tabla 20: Errores y Diferencias de tiempo de ejecución.

CARACTERISTICA	L4	L6
Errores y diferencias de Tiempo de ejecución FPGA Arria IIGX vs Matlab® - Intel i7, 16GB RAM, 4GHz)		
- Error con prec = 8	1.348E-6%	1.952E-6%
- Error con prec = 4	4.492E-3%	0%
- Diferencia de tiempo con prec=8 [segundos]	50.739	228.263
- Diferencia de tiempo con prec=4 [segundos]	20.443	167.476
Errores y diferencias de Tiempo de ejecución FPGA Arria IIGX vs Matlab® - AMD A10, 8GB RAM, 2.5GHz)		
- Error con prec = 8	1.348E-6%	1.952E-6%
- Error con prec = 4	4.492E-3%	0%
- Diferencia de tiempo con prec=8 [segundos]	2.444	181.637
- Diferencia de tiempo con prec=4 [segundos]	0.113	151.173

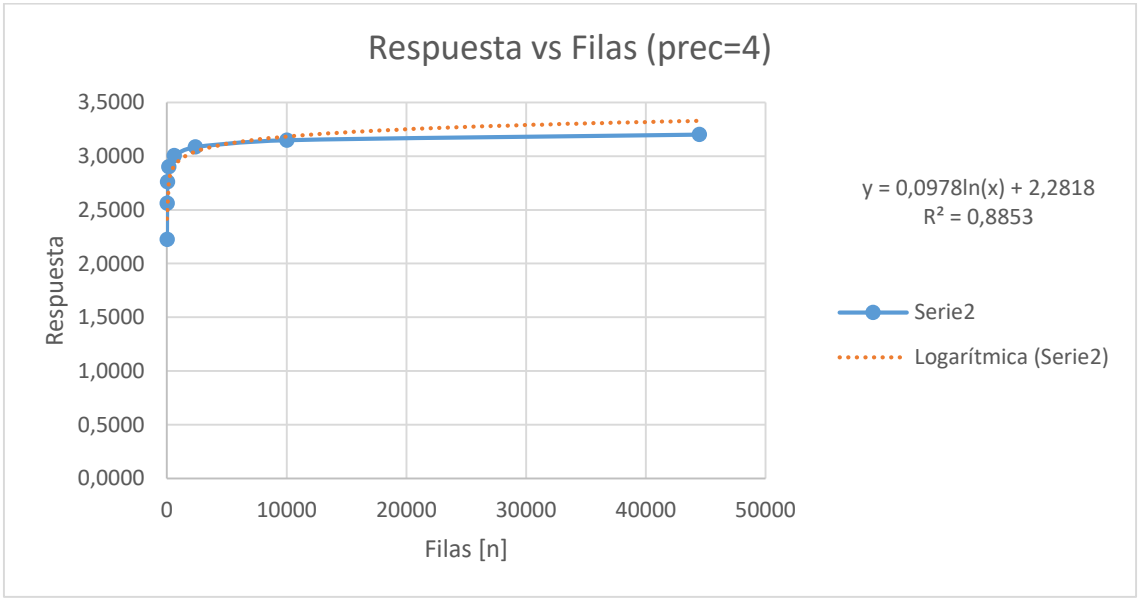
En relación a datos obtenidos por la ejecución del algoritmo en Matlab se presentan a continuación algunas graficas que corroboran información dada en secciones anteriores. En la figura 43 se observa como n efectivamente crece de manera exponencial con una línea de tendencia con un índice de correlación de 0.9975.

Figura 43: Crecimiento de n.



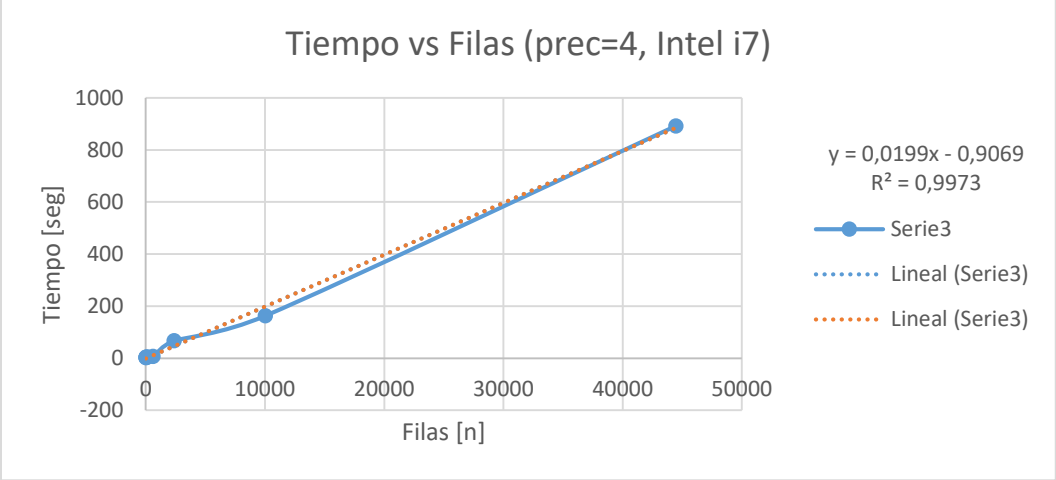
En la figura 44 se observa el comportamiento del resultado en Matlab con $prec = 4$ en relación al número de filas. A medida que la lista es más grande la precisión del resultado mejora al valor deseado de 3.796. Como solamente se tiene disponible hasta la lista L18 entonces el valor máximo alcanzado por el software Matlab® con 4 cifras significativas es 3.2010. Debido a que la correlación es 0.8853 la ecuación de la línea de tendencia logarítmica en esta figura no tiende en el infinito al valor deseado.

Figura 44: Línea de tendencia hacia el valor deseado 3.796 con $prec = 4$



En la figura 45 se observan los tiempos consumidos por el software Matlab® ejecutado sobre la workstation Intel® Core™ i7 6700k, 4 GHz, 16GB RAM, disco de estado sólido PC300 NVMe SK hy, tarjeta gráfica NVIDIA® GeForce® GTX 950. Se recalca como el comportamiento del tiempo es lineal por medio de una línea de tendencia con R igual a 0.9973. Lo anterior indica el efecto de la función de complejidad del tiempo polinomial del algoritmo.

Figura 45: Tiempos de ejecución en PC Intel i7 con $prec = 4$



Se realiza el mismo análisis con la ejecución de las listas en Matlab® para un valor de $prec = 8$. En las figuras 46 y 47 se observa las mismas relaciones que en las figuras 44 y 45.

Figura 46: Línea de tendencia hacia el valor deseado 3.796 con $prec = 8$

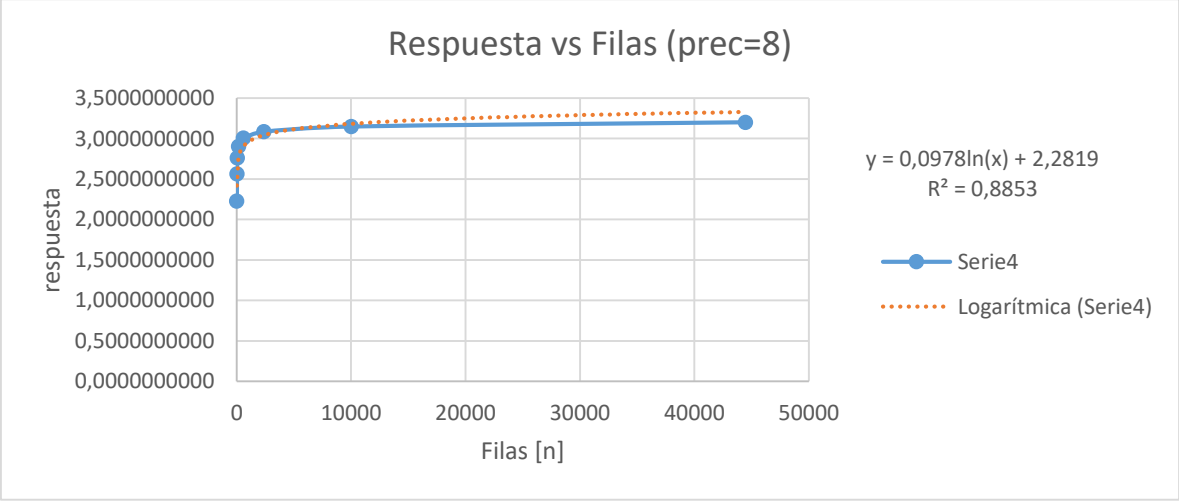
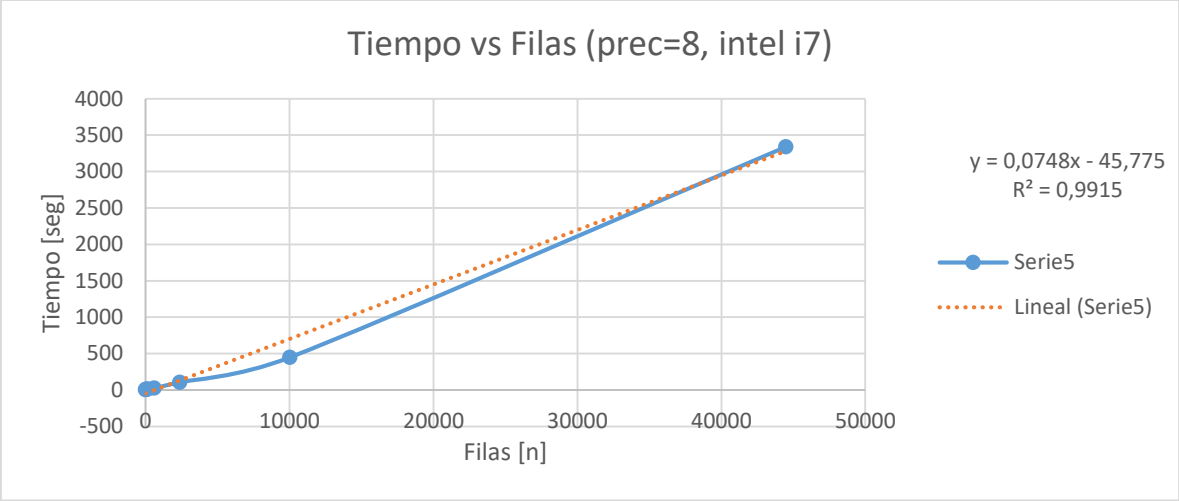


Figura 47: Tiempos de ejecución en PC Intel i7 con $prec = 8$



4.27 OTRAS DISCUSIONES ACERCA DE LOS RESULTADOS OBTENIDOS

El número de señales que se van a visualizar en las etapas de pruebas o en las etapas de validación deben ser consideradas de manera algo detallada ya que cada señal de estas y en especial la las señales que forman buses de datos, consumen significativos recursos y la compilación del diseño puede demorar un tiempo innecesario.

El reloj del sistema es un oscilador de 100 MHz disponible en la tarjeta de desarrollo. El reloj que gobierna el diseño es configurado por un divisor de frecuencia que lo reduce hasta 5 MHz para la lista L4 y 2.5 MHz para la lista L6. El uso de un reloj por medio de un divisor de frecuencia puede optimizarse debido a que cuando se implementa este divisor el ruteo de la señal de reloj se sintetiza sobre los ALUTs y esto consume más recursos y puede alterar de manera perjudicial la temporización de la señal de reloj porque esta señal, en algunas listas, es la que más fan-out tiene. Aunque en este trabajo de grado este efecto negativo no sucedió, se podría mejorar el diseño con la implementación de un PLL, y además subir así la velocidad de reloj para acelerar la ejecución del algoritmo. Se realizó este ejercicio, pero se determinó una falla en el reset del PLL que se genera una desincronización con el *trigger* de SignalTap® la cual es la señal que permite determinar el momento de capturar las señales para su visualización en pantalla, y finalmente se tuvo que trabajar con el divisor para poder finalizar el proyecto.

Si el diseño es muy extenso y requiere grandes cantidades de ALUTs la aplicación quartus.exe genera un error grave que detiene el programa entero y obliga al reinicio de Quartus®. El error STACK OVERFLOW puede deberse a que el diseño es tan grande que el computador donde se compila no tiene una paginación suficiente para procesar las instrucciones dentro del mapeo. El error STACK OVERFLOW sucede cuando se experimenta un exceso en el flujo de datos almacenados en la pila utilizada para la ejecución de una función.

Para poder implementar la compilación incremental, el diseño debe seguir todas las recomendaciones del subprograma de Quartus® - *Incremental Compilation Advisor* el cual es un asesor que da consejos para que el diseñador siga en los diseños y así la compilación incremental produzca un menor consumo de área. Por ejemplo, el diseño debe tener todas sus entidades registradas en la entrada y en la salida con el fin de que el ruteo entre particiones se realice con un menor número de LEs.

La compilación incremental es efectiva o verdadera conveniente solamente para cambios pequeños en el diseño, pero cuando se cambian parámetros que afectan a la mayoría de entidades, como en este proyecto, la compilación incremental resulta desfavorable ya que para realizar este tipo de compilación se debe compilar completamente la primera vez en cada modificación el diseño para luego si compilar incrementalmente y poder reducir el tiempo de compilación en las posteriores compilaciones.

Por medio del subprograma *Chip Planner* puede realizarse un *floorplan* anticipado o un *floorplan* tardío. El *floorplan* o plano de planta sirve para que el diseñador pueda definir la posición física de las entidades del diseño en la FPGA. Esto se realiza cuando se hace un diseño en equipo (más de dos personas) o cuando se debe reservar el espacio con los recursos necesarios para entidades que requieran tiempos de señal muy precisos o en general entidades cruciales en la aplicación del diseño. En este proyecto se podría hacer un *floorplan* tardío sin embargo sucede algo similar con la compilación incremental debido a que el diseño no puede tener muchas modificaciones dados los parámetros que afectan a casi todas las entidades del diseño. Además, debe conocerse muy bien los recursos que consumen cada entidad (partición) para que el *floorplan* este bien planeado y las entidades quepan en el espacio físico asignado. Esto último es prácticamente imposible en este proyecto debido a que en cada lista se varía en gran medida el uso de los recursos, de acuerdo al diseño propuesto.

5. CONCLUSIONES

Se logró la síntesis en hardware de la totalidad de instrucciones del algoritmo en Matlab® que posibilitaron el cálculo del valor crítico para las listas L4 y L6, también se realizaron sus respectivas verificaciones.

Se realizó un diseño altamente parametrizado para la síntesis en hardware de la totalidad de instrucciones del algoritmo en Matlab® que permitirían el cálculo del valor crítico para cualquier lista. Sin embargo, para el caso de la lista L8 solamente se obtuvo información acerca de los recursos requeridos, los cuales superan el 100% de los recursos disponibles en la FPGA. Para listas superiores no fue posible obtener resultados porque la herramienta EDA genera desbordamiento de memoria.

Aunque se utilizó una FPGA de alta complejidad como la Arria II GX 125, únicamente se logró implementar la arquitectura para el cálculo del valor crítico de las listas L4 y L6. Lo anterior se debe en gran medida a que el cálculo del valor crítico se basa en la función recurrente fRec() que depende del número de filas (n). No obstante, se logró corregir parcialmente esta limitante haciendo que una de las funciones que componen a fRec() sea independiente de n.

El error para el cálculo del valor crítico obtenido por la implementación del algoritmo en la FPGA, comparado con la ejecución del algoritmo en dos computadores es muy bajo (1.348E-6%, 1.952E-6%, 4.492E-3%, 0%)

Se obtuvo que el rendimiento de ejecución de las funciones críticas implementadas en la FPGA no presenta una mejora en comparación con la workstation de propósito general, lo cual es debido a la implementación de relojes de muy baja velocidad y los problemas de diseño en relación a las entidades que crecen con n.

Se analizaron principalmente tres arquitecturas para el divisor, la cual se estableció como una de las operaciones más demandantes en hardware, se implementó la arquitectura HIERG/RTC que corresponde aproximadamente a un 15.52 % de los recursos disponibles en la FPGA seleccionada.

Se estableció que la unidad hardware que realiza la operación basada en punteros codificada en Matlab® como $x(L)$ es también una de las operaciones más demandantes en hardware. Dado que esta operación que hace parte de la unidad hardware UNIDAD_FREC crece con n y ocupa un 6.64%, 43.64%, 76.78% del área total del diseño para la síntesis del hardware que ejecuta las listas L4, L6 y L8 respectivamente.

Al diseñar un datapath cuyos componentes presenten granularidad fina, el desarrollo de la unidad de control es más sencillo, debido a que presenta mayor facilidad la evaluación del flujo de datos entre los componentes a controlar. Pero se pueden llegar a perder ciclos de reloj en algunas transferencias entre registros.

Debido a que el diseño realizado es totalmente síncrono, se tuvo que evitar la generación de latches, los cuales se infieren en: las descripciones de las máquinas de estados con asignaciones incompletas de un bus de datos, en la incorrecta implementación de los procedimientos concurrentes y en las instrucciones de decisión incompletas.

El formato punto fijo establecido de 3 bits enteros, 61 bits decimal permitió obtener resultados satisfactorios para la ejecución del algoritmo.

6. TRABAJOS FUTUROS

Realizar unidades hardware separadas y su correspondiente controlador para las funciones que crecen con n, tales como las unidades COMPARADOR 1 y 2 y UNIDAD_FREC. Este ejercicio evitaría que el diseño realizado crezca a medida que las listas aumenten.

Implementación del formato punto flotante con el fin de lograr una compatibilidad perfecta con el formato estándar encontrado en una workstation. Sin embargo, este ejercicio es relativo debido a que en el diseño hardware se implementan diversos formatos en unidades especializadas para optimizar el consumo de recursos y acelerar procesos.

Desarrollo de la interfaz gráfica entre el sistema aceleración en FPGA y una workstation para interactuar con el software Matlab® en algoritmos que requieran de interfaces computacionales y que ejecuten algoritmos que requieran de un procesamiento en procesador de aplicación general junto con un co-procesamiento especializado para optimizar partes de dicho algoritmo.

7. BIBLIOGRAFÍA

[1] J. C. Vera, E. Vigoda y L. Yang, «Improved bounds on the phase transition for the hard-core model in 2-dimensions,» *Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques,* vol. 8096, pp. 699-713, 2013.

[2] D. Weitz., «Counting independent sets up to the tree threshold».

[3] R. Restrepo, J. Shin, P. Tetali, E. Vigoda y L. Yang., «Improved Mixing Condition on the Grid for Counting and Sampling Independent Sets.,» *Theory and Related Fields,.*

[4] M. R. Garey y D. S. Johnson, Computers and Intractability, New York: W.H. Freeman and Company.

[5] J. Barbier, F. Krzakala, L. Zdeborová y P. Zhang, «The hard-core model on random graphs revisted,» *Journal of Physics: Conference Series,* 2013.

[6] P. Raghavendra, Approximating NP-hard Problems Efficient Algorithms and their Limits, Washington: University of Washington, 2009.

[7] P. Groisman, «Medidas de Gibbs y transicion de fase,» Universidad de Buenos Aires, 2013. [En línea]. Available: http://mate.dm.uba.ar/~pgroisma/medidasdegibbs.html. [Último acceso: 01 2017].

[8] A. Blanca, D. Galvin, D. Randall y P. Tetali, «Phase Coexistence and Slow Mixing for the Hard-Core Model on Z2,» *Proceedings of the 17th International Workshop, RANDOM,* pp. 379-394, 2013.

[9] H. S. Carlessi y C. R. Meza, Metodología y diseños en investigacióncientífica, Lima, Peru: Visión Universitaria, 2006.

[10] V. Crespi, A. Galstyan y K. Lerman, «Top Down vs Bottom Up Methodologies in Multi Agen System Design,» *Autonomous Robots manuscript,* vol. 24, nº 3, pp. 303-313, 2008.

[11] Altera Corp, Arria II GX Device Handobook, San Jose, CA: Altera Corp, 2014.

[12] K. C. Chang, Digital systems design with VHDL and synthesis: an integrated approach, Los Alamitos, CA: IEEE, 1999.

8. ANEXOS

Anexo 1: Código VHDL de la entidad RAM_X_PARALELO

```
entity RAMX_PARALELO is
  port(
    CLK_PORT     : in std_logic;
          RESET_PORT     : in std_logic;
          ENABLE_PORT  : in std_logic;
          DATA_PORT      : in tipo_entrada_RAMX_paralelo;
          Q_PORT                    : out tipo_entrada_RAMX_paralelo
  );
end RAMX_PARALELO;

architecture STRUCTURAL of RAMX_paralelo is
begin
RAM_X: for k in 0 to ROWS-1 generate
          FF_N: FLIPFLOP_N
             generic map (n => RAM_Q_DATA_BITS)
                    port map(
                              CLK_PORT     => CLK_PORT,
                              RESET_PORT   => RESET_PORT,
                              ENABLE_PORT  => ENABLE_PORT,
                              D_PORT       => DATA_PORT(k),
                              Q_PORT       => Q_PORT(k)
                    );
end generate RAM_X;
end STRUCTURAL;
```

Anexo 2: Código VHDL de la entidad REG_IN_SERIE_OUT_PARALELO

```
entity REG_IN_SERIE_OUT_PARALELO is
  port(
    CLK_PORT     : in std_logic;
          RESET_PORT   : in std_logic;
          ENABLE_PORT  : in std_logic;
          DATA_PORT    : in tipo_Q_ROM;
          Q_PORT       : out tipo_salida_ROM_paralelo
  );
end REG_IN_SERIE_OUT_PARALELO;

architecture STRUCTURAL of REG_IN_SERIE_OUT_PARALELO is
signal AUX_SIGNAL : tipo_salida_ROM_paralelo := (others=>(others=>'0'));
begin  --begin arch
REG_IN_OUT: for k in (ROWS*3)-1 downto 0 generate
                    FF_N_A1: if k = (ROWS*3)-1 generate
                              FF_N: FLIPFLOP_N
                                        generic map(N => ROM_Q_BITS)
                                        port map(
                                                  CLK_PORT     => CLK_PORT,
                                                  RESET_PORT   => RESET_PORT,
                                                  ENABLE_PORT  => ENABLE_PORT,
                                                  D_PORT             => DATA_PORT,
                                                  Q_PORT             => AUX_SIGNAL(k)
                                        );
                    end generate FF_N_A1;
                    FF_N_A2: if k /= (ROWS*3)-1 generate
```

```
                        FF_N: FLIPFLOP_N
                                generic map(N => ROM_Q_BITS)
                                port map(
                                        CLK_PORT        => CLK_PORT,
                                        RESET_PORT   => RESET_PORT,
                                        ENABLE_PORT  => ENABLE_PORT,
                                        D_PORT          => AUX_SIGNAL(k+1),
                                        Q_PORT          => AUX_SIGNAL(k)
                                );
                end generate FF_N_A2;
end generate REG_IN_OUT;
Q_PORT <= AUX_SIGNAL;
end STRUCTURAL;
```

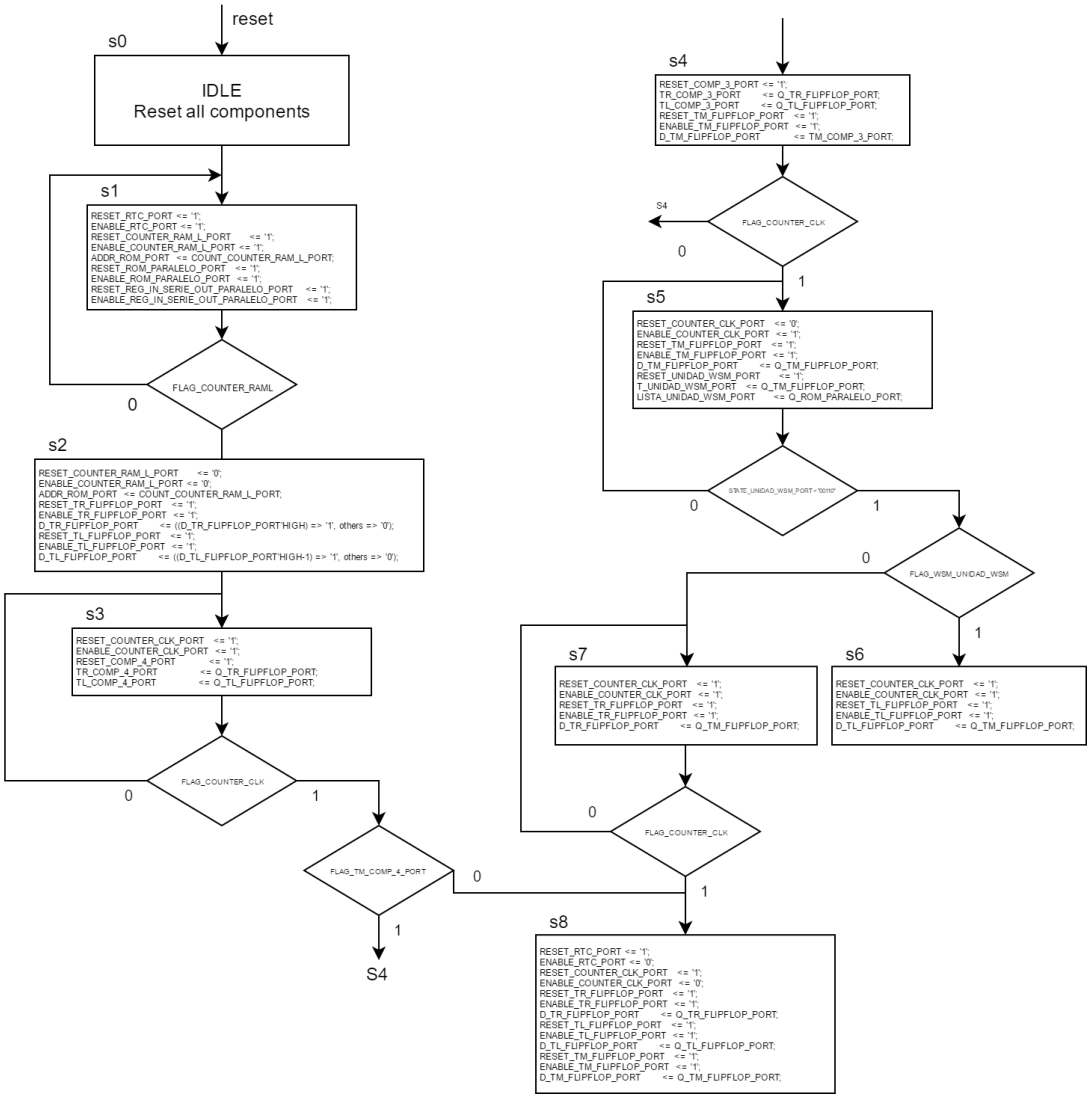
Anexo 3: Código VHDL de la entidad COUNTER_RAM_L

```
ENTITY COUNTER_RAM_L IS
        PORT (
                CLK_PORT                : IN std_logic;
                RESET_PORT      : IN std_logic;
                ENABLE_PORT     : IN std_logic;
                FLAG_COUNTER_RAM_L_PORT : out std_logic;
                COUNT_PORT              : OUT tipo_addr_RAML
        );
END COUNTER_RAM_L;

ARCHITECTURE BEHAVIORAL OF COUNTER_RAM_L IS
signal AUX_CNT_SIGNAL : UNSIGNED(RAML_ADDR_BITS-1 DOWNTO 0) := (others => '0');
begin
CNT_RAM_L: PROCESS (CLK_PORT, RESET_PORT) IS
                begin
                if RESET_PORT = '0' THEN
                        AUX_CNT_SIGNAL <= (others => '0');
                        FLAG_COUNTER_RAM_L_PORT <= '0';
                elsif CLK_PORT'event AND CLK_PORT ='1' then
                        if ENABLE_PORT = '1' then
                                        AUX_CNT_SIGNAL <= AUX_CNT_SIGNAL + 1;
                                        if AUX_CNT_SIGNAL < ((ROWS*3)-1) then
                                                FLAG_COUNTER_RAM_L_PORT<= '0';
                                        else
                                                AUX_CNT_SIGNAL <= (others => '0');
                                                FLAG_COUNTER_RAM_L_PORT<= '1';
                                        end if;
                        elsif ENABLE_PORT = '0' then
                                        FLAG_COUNTER_RAM_L_PORT <= '0';
                        end if;
                end if;
end process;
COUNT_PORT <= std_logic_vector(AUX_CNT_SIGNAL);
END BEHAVIORAL;
```

Anexo 4: Diagrama ASM de la FSM de CONTROL_UNIDAD_WSM_CRIT_MAR

DIAGRAMA ASM

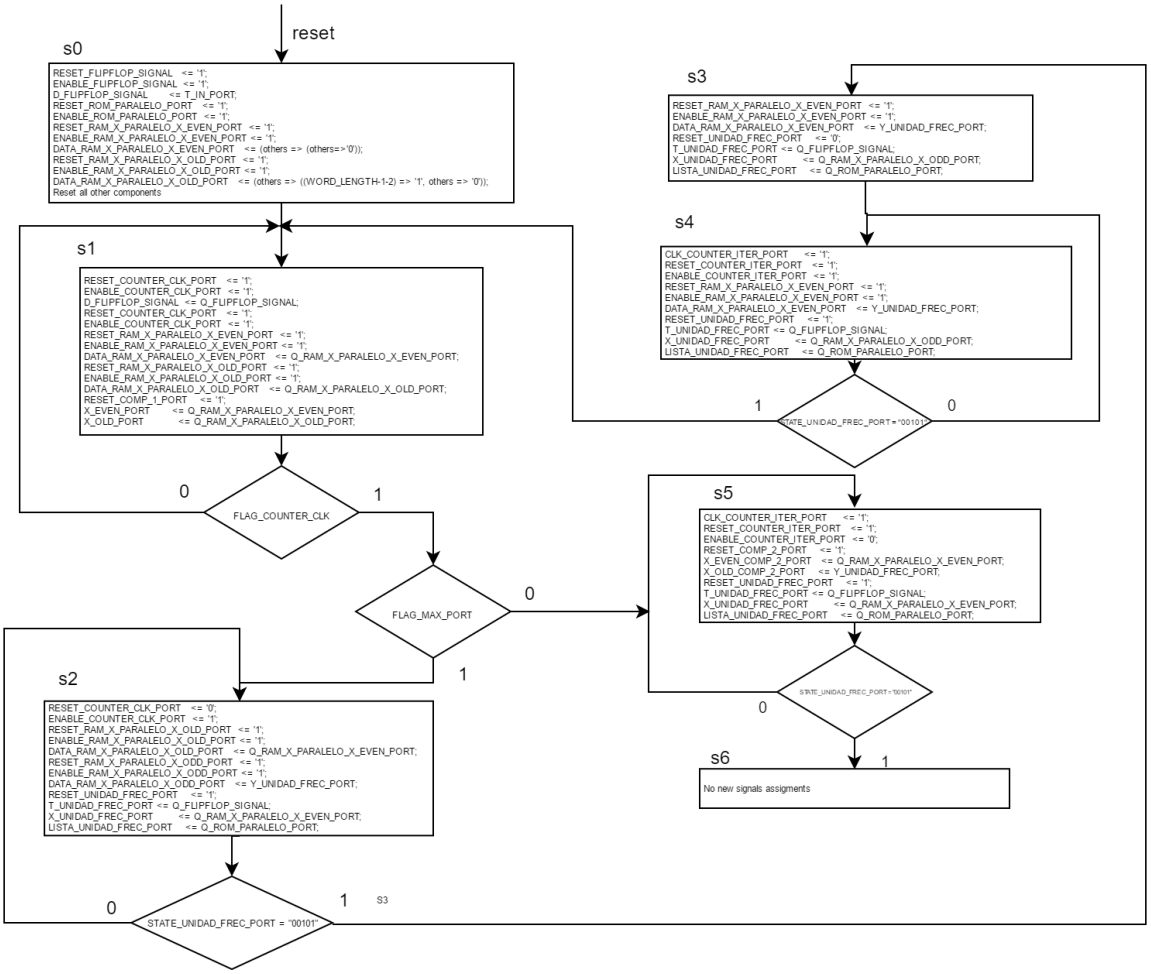
CONTROL_UNIDAD_WSM_CRIT_MAR



Anexo 5: Diagrama ASM de la FSM de CONTROL_UNIDAD_WSM

DIAGRAMA ASM

CONTROL_UNIDAD_WSM



Anexo 6: Diagrama ASM de la FSM de CONTROL_UNIDAD_FREC