Diseño e Implementación de un Coprocesador basado en FPGA para el reconocedor de voz SPHINX

por

Ing. Guillermo Aníbal Marcus Martínez

Tesis

Presentada al Programa de Graduados en Electrónica, Computación, Información y Comunicaciones

como requisito parcial para obtener el grado académico de

Maestro en Ciencias en Ingeniería Electrónica

especialidad en

Sistemas Electrónicos

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

Diciembre de 2003

© Guillermo Aníbal Marcus Martínez, 2003

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

División de Electrónica, Computación, Información y Comunicaciones

Programa de Graduados

Los miembros del comité de tesis recomendamos que la presente tesis de Guillermo Aníbal Marcus Martínez sea aceptada como requisito parcial para obtener el grado académico de **Maestro en Ciencias en Ingeniería Electrónica**, especialidad en:

Sistemas Electrónicos

Comité de tesis:

Dra. Patricia Hinojosa Flores
Asesor de la tesis

Dr. Alfonso Ávila Ortega
Sinodal

Dr. Juan Nolazco Flores
Sinodal

Dr. David Garza Salazar
Director del Programa de Graduados

Diciembre de 2003

Índice general

Diseño e Implementación de un Coprocesador basado en FPGA para el reconocedor de voz SPHINX

Guillermo Aníbal Marcus Martínez, M.C.
Instituto Tecnológico y de Estudios Superiores de Monterrey, 2003

Asesor de la tesis: Dra. Patricia Hinojosa Flores

Se presenta el diseño y la implementación de un coprocesador basado en FPGA para el reconocedor de voz SPHINX. Se implementaron un sumador y un multiplicador de punto flotante de precisión sencilla completamente segmentado capaces de operar a 40 y 50MHz respectivamente, así como las partes de apoyo necesarias para la comunicación con la tarjeta de desarrollo RACE-1. El diseño con 6 unidades de cómputo consta de 12 sumadores y 12 multiplicadores, opera a una frecuencia de 33MHz y operando sin antememorias de datos ofrece una ganancia marginal comparado con los tiempos de cómputo de una PC a 1.8GHz para vectores de mas 256 elementos.

Capítulo 1

Introducción

En la investigación en reconocimiento de voz, muchos progresos se han hecho en los últimos años, en el desarrollo de nuevos algoritmos y aprovechando el incremento en capacidad computacional de las computadoras, habilitando aplicaciones que eran imposibles técnicamente o prohibitivamente costosas. Una de estas ramas de investigación, que anteriormente era muy poco explorada dados sus requerimientos de cómputo, es el uso de HMM continuos en vez de discretos. Esto trae más exactitud en el reconocimiento, pero involucra cálculos más complejos. En particular, dado el tipo y la cantidad de cálculos y dependiendo de las características del reconocedor deseado, generar los modelos correspondientes para la decodificación (mediante un proceso conocido como entrenamiento del reconocedor) puede consumir varias horas de tiempo dedicado en un arreglo de estaciones de trabajo.

Por lo tanto, se buscan soluciones para realizar los cálculos más rápidamente. Una manera es agregar hardware dedicado que permita disminuir el tiempo de cómputo, asistiendo al programa en puntos claves que se detectan como de uso intensivo. Este hardware pueden ser un ASIC (Application Specific Integrated Circuit), semiconductores especializados diseñados y fabricados específicamente para realizar esta tarea; DSPs (Digital Signal Processors), procesadores genéricos orientados al manejo de señales y programados para asistir en la tarea; ó FPGAs (Field Programmable Gate Arrays), semiconductores de lógica reprogramable que permiten implementar diseños específicos sin involucrar procesos costosos de fabricación.

Dado que el diseño e implementación con ASICs es muy costoso e involucra mucha experiencia en el área, la cual actualmente no poseemos; y dado que el uso de DSPs no pre-

sentaba una mejora significativa con el uso de los procesadores de las estaciones de trabajo los cuales ya cuentan con unidades de cómputo vectorial (SIMD), entonces se optó por diseñar un coprocesador basado en FPGA, que aprovecha las ventajas de los ASIC en términos de recursos disponibles con la facilidad de diseño y reprogramación de sistemas embebidos y DSPs.

1.1. Antecedentes

La mayoría de los trabajos relacionados en esta área están orientados a la decodificación con HMM discretos. Pocos se concentran en HMM continuos, y aún menos en el proceso de entrenamiento. Las publicaciones [Sil97] [Bar01] son ejemplos de implementaciones combinadas de hardware y software para el entrenamiento de HMM discretos. Los artículos [Rus00][Rus02a][Rus02b] documentan los avances realizados por un grupo de investigación de la Universidad de Birmingham en el entrenamiento de HMM continuos y su uso posterior en decodificación, y que representa el nivel al que nos gustaría llegar al final de este trabajo de investigación.

1.2. Objetivos

Nuestro objetivo principal es la propuesta e implementación de un diseño de coprocesador que apoye en los cálculos necesarios para el entrenamiento de HMM continuos utilizados por el reconocedor SPHINX. Para ello, se plantean los siguientes objetivos secundarios:

1. Elección, instalación, configuración y uso de las diferentes herramientas involucradas en el diseño, como son la herramienta de captura (Mentor Graphics HDL Designer Pro), el sintetizador (Mentor Graphics Leonardo Spectrum), el simulador (Modeltech ModelSim) y el place and route (Xilinx ISE 4.1).

2. Implementación de partes de soporte para el uso de las tarjetas de desarrollo RACE-1 y sus periféricos.

3. Implementación de un sumador y un multiplicador de punto flotante, que como se verá más adelante, son parte esencial para el funcionamiento de HAVOC.

1.3. Organización

El documento está organizado en 7 capítulos y 3 apéndices. El capítulo 2 cubre el marco teórico, donde se presentan los conocimientos básicos en arquitectura y aritmética computacional necesarios para la exposición del trabajo realizado. El capitulo 3 presenta y explica la arquitectura del coprocesador, presenta sus partes constitutivas y su interfaz con los programas de la PC. El capítulo 4 detalla el diseño y el funcionamiento de partes de soporte para el coprocesador, como son el Cliente de Bus Local y el Manejador de Memorias ZBT. El capítulo 5 detalla la estructura e implementación del multiplicador y el sumador de punto flotante. En el capítulo 6 se presentan y analizan los resultados de las pruebas realizadas con el coprocesador. Por último, en el capítulo 7 se presentan las conclusiones y las posibilidades de trabajos futuros.

En el apéndice A se da un resumen de la teoría de reconocedores de voz y de los algoritmos asociados. En el apéndice B se tratan varios temas de instalación y configuración de los programas y herramientas utilizados. En el apéndice C se muestran diagramas de tiempo adicionales que complementan la documentación de los resultados. Se incluye un CDROM anexo que contiene los códigos de prueba, el código fuente y los repositorios de diseño del HDL Designer, así como una versión electrónica de este documento.

Capítulo 2

Fundamentos de Arquitectura y Aritmética Computacional

Presentamos a continuación unas bases muy concisas de los temas de arquitectura y aritmética computacional que son necesarios para los desarrollos que se explican en los siguientes capítulos de este documento. Para mayor profundidad sobre alguno de los temas aquí planteados, se sugiere la consulta de las obras referidas en la bibliografía, [Pat03][Par00][Xil02].

Este capítulo se ha dividido en tres secciones principales. La primera sección cubre temas de Arquitectura Computacional, en particular la Segmentación (*Pipelining*) y sus Riesgos (*Pipeline Hazards*). La segunda sección presenta los conceptos de lógica reprogramable y la arquitectura Virtex-II. La tercera sección cubre temas de Aritmética Computacional y operaciones en Punto Flotante.

2.1. Arquitectura Computacional

La arquitectura computacional se basa en la medición y análisis de tiempos de ejecución, y en la manera de mejorar los diseños de los circuitos que realizan las tareas en cuestión para mejorar dichos tiempos. Por lo tanto, una de las leyes básicas, conocida como la *Ley de Amdahl*, es la que define la mejora o speedup como:

$$\text{Speedup} = \frac{\text{Tiempo de Ejecución}_{\text{viejo}}}{\text{Tiempo de Ejecución}_{\text{nuevo}}}$$

2.1.1. Segmentación

La segmentación o *Pipelining* consiste en dividir un circuito combinacional en n etapas separadas mediante el uso de registros. Como en una linea de producción, las etapas van realizando las operaciones por pasos, guiadas por el reloj del sistema, y cada etapa puede estar operando sobre un dato diferente. Si cada etapa puede estar trabajando en un dato independiente, entonces el primer dato procesado tarda n ciclos, pero a partir de ahí se obtiene un dato nuevo por cada ciclo. Por lo tanto, el hecho de dividir el circuito permite una ganancia máxima teórica de n. Por lo tanto, podemos escribir:

$$\text{Speedup}_{\text{pipeline}} = \frac{\text{Ciclo de reloj sin pipeline}}{\text{Ciclo de reloj con pipeline}} \approx n$$

Sin embargo, pueden ocurrir diferentes situaciones por las cuales un *pipeline* no puede avanzar, por ejemplo, que la operación a realizar dependa de un dato que todavía no ha terminado de ser procesado. En estos casos, es necesario bloquear (*stall*) el *pipeline*, con uno o más ciclos sin operación. Al tomar esto en cuenta en las ecuación anterior, obtenemos que:

$$\begin{aligned} \text{Speedup}_{\text{pipeline}} &= \frac{1}{1 + \text{Ciclos bloqueados por instrucción}} \times \frac{\text{Ciclo de reloj sin pipeline}}{\text{Ciclo de reloj con pipeline}} \\ &\approx \frac{1}{1 + \text{Ciclos bloqueados por instrucción}} \times n \end{aligned}$$

Los riesgos en la segmentación, o *Pipeline Hazards*, son aquellas condiciones que producen un bloqueo del *pipeline*. Estas condiciones se dividen en 3 categorías:

- **Riesgos Estructurales**, Son aquellos que aparecen cuando existe un conflicto de recursos, por ejemplo, cuando dos etapas intentan utilizar el mismo recurso al mismo tiempo, y éste no lo permite.
- **Riesgos de Datos**, Son aquellos que aparecen cuando una operación depende de un dato que todavía está siendo procesado por otra etapa del *pipeline*.
- **Riesgos de Control**, Son aquellos que aparecen cuando una operación modifica el flujo

de un programa.

A su vez, los riesgos de datos pueden clasificarse de la siguiente manera, dependiendo del tipo de operación que realizan. Estos riesgos dependen de la situación y de la arquitectura en cuestión:

- **RAR**, Read After Read, Representa una lectura de una localidad de memoria mientras una operación de lectura está pendiente en el pipeline. En prácticamente la totalidad de las arquitecturas, esto no representa un riesgo.
- **RAW**, Read After Write, Representa una lectura de una localidad de memoria mientras una operación de escritura está pendiente en el pipeline, por lo tanto, leyendo el valor anterior.
- **WAR**, Write After Read, Representa una escritura en una localidad de memoria mientras una lectura está pendiente en el pipeline, recibiendo la lectura el nuevo valor.
- **WAW**, Write after Write, Representa una escritura en una localidad de memoria mientras otra escritura a la misma dirección está pendiente, ejecutando las escrituras en el order incorrecto.

En general, podemos decir que la intención en el diseño de arquitecturas computacionales es minimizar los bloqueos de un *pipeline*, independientemente de su causa. Los riesgos de datos más comunes son los tipo RAW y las WAR, y en determinadas situaciones, pueden usarse técnicas como el adelanto de datos (*Data Forwarding*) para proveer a las etapas anteriores con los resultados de operaciones que no han finalizado en el *pipeline*. Los riesgos estructurales y de control son muy dependientes de la arquitectura y de los recursos usados. Un análisis más en detalle de estos temas puede verse en [Pat03].

2.2. Lógica Programable

La lógica programable engloba todos los circuitos que permiten reemplazar lógica discreta, combinatoria y/o secuencial, mediante un dispositivo semiconductor que puede ser programado para implementar la función lógica deseada. Dentro de estos dispositivos se pueden

agrupar a los PAL (Programmable Array Logic), GAL (Generic Array Logic) y PLD (Programmable Logic Device) como circuitos integrados que consisten de bloques o unidades lógicas (llamados macroceldas) interconectados entre si y con los pines de entrada (tipicamente en una matriz de todos contra todos), donde cada macrocelda consiste típicamente de una tabla o función lógica y de un flip-flop (opcional). Normalmente estos dispositivos constan de unos pocos bloques (de 4 a 22 macroceldas) y son utilizados para reemplazar lógica discreta que de otra manera ocuparía mucho espacio en un circuito. Los CPLD (Complex PLD) se basan en el mismo concepto, pero poseen muchas más macroceldas, de 10 a unos cuantos cientos, donde las macroceldas están agrupadas jerárquicamente en conjuntos de unas 16 macroceldas cada uno, en los cuales todas las macroceldas del mismo conjunto se encuentran interconectadas entre si, mientras que las conexiones entre los conjuntos son limitadas.

Los FPGAs (Field Programmable Logic Devices) corresponden a una familia diferente de dispositivos, ya que consisten de un gran número de bloques lógicos (desde unos pocos cientos a decenas de miles) con abundancia de flipflops y organizados típicamente como una matriz, rodeados de bloques de IO e interconectados mediante una matriz de enrutamiento. Como la matriz para conectarlos todos entre sí sería prohibitivamente grande, diferentes esquemas de interconexión han sido propuestos por los fabricantes. Cada bloque lógico, llamado CLB (Configurable Logic Block) suele consistir de una cantidad considerable de lógica, con una o más tablas de función, uno o más flipflops, multiplexores, y circuiteria adicional. Actualmente existe la tendencia a dividir los bloques en unidades más pequeñas, llamadas *Slices*, donde cada uno consiste típicamente en una tabla lógica, uno o dos flipflops y algo más de lógica, y que pueden interconectarse entre si completamente dentro del CLB para formar funciones más complejas.

2.2.1. Arquitectura Virtex-II

La organización básica de la arquitectura Virtex II, [Xil02], puede verse en la figura 2.1. En un Virtex-II, cada CLB consta de 4 SLICEs, y están organizados como una matriz de CLBs. Alrededor se encuentran los bloques de entrada/salida, IOBs, que se encargan

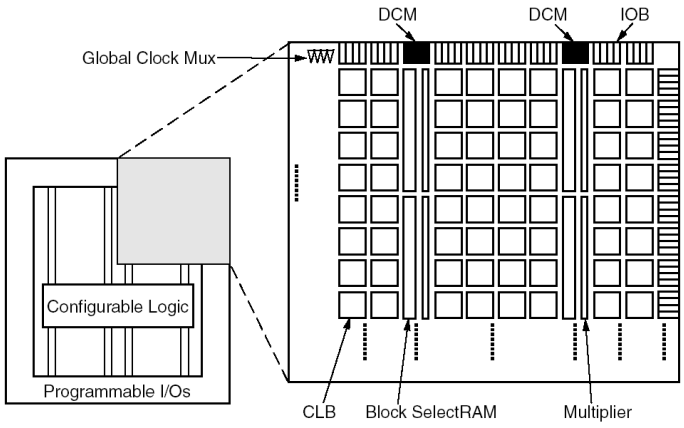

Figura 2.1: Arquitectura Virtex-II

de las diferentes configuraciones de señal de los pines, así como también poseen registros para interfaz con señales DDR. Dentro de la matriz de CLB se pueden ver columnas con bloques especializados, que consisten en un par de memoria de bloque (BlockRAM) y un multiplicador por hardware de 18x18 bits. Entre los IOB también pueden apreciarse bloques de manejo de reloj (DCM, *Distributed Clock Managers*), que permiten manipular señales de reloj.

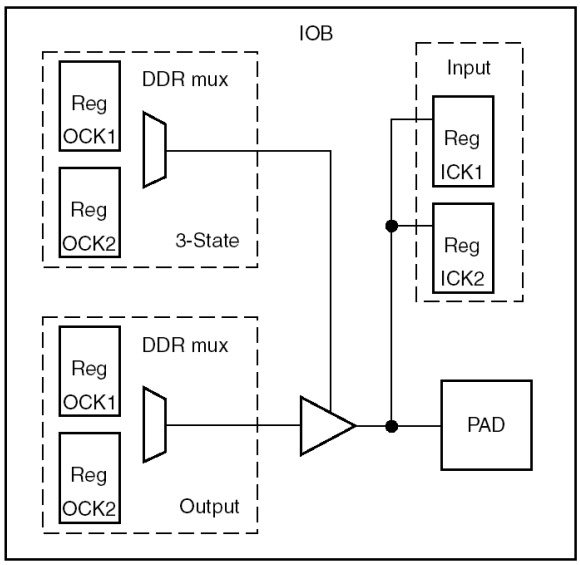

Figura 2.2: IOB

En la figura 2.2 se muestra la configuración interna de un IOB. Aparte de los registros asociados con la capacidad de usar señalización DDR, el bloque IOB maneja muchos estándares configurables al momento de la configuración del bloque, incluidos esquemas de impedancia controlada para señalización de alta velocidad.

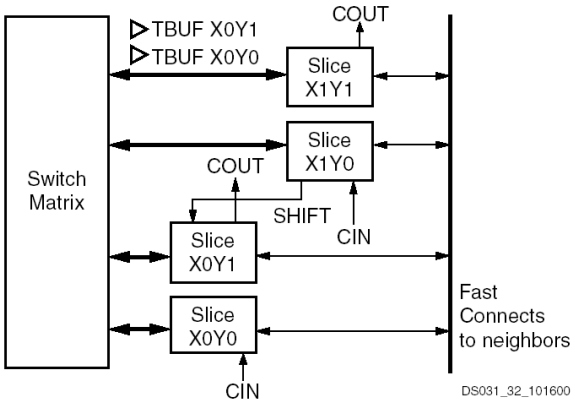

Figura 2.3: CLB

En la figura 2.3 se puede ver el arreglo de los SLICEs dentro de un CLB, junto con 2 buffers de 3 estados. Xilinx introdujo en la arquitectura Virtex-II un nuevo arreglo de los SLICEs dentro del CLB, integrando 4 SLICEs por CLB, pero donde los SLICEs pueden asociarse por pares para crear funciones más complejas predefinidas, como cadenas rápidas de suma o registros de corrimiento muy largos.

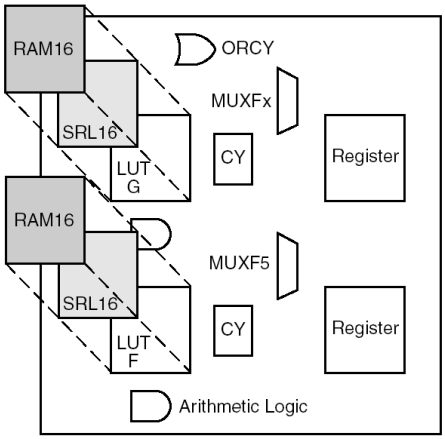

Figura 2.4: SLICE

En la figura 2.4 se puede ver a grandes rasgos que cada SLICE consiste de dos generadores de función, dos mulplixeores, dos flipflops y lógica de aritmética. Cada generador de función puede configurarse como una memoria de 16 bits, como una tabla de búsqueda (LUT, Look Up Table) de 4 entradas, o como un registro de corrimiento de 16 posiciones x 1 bit.

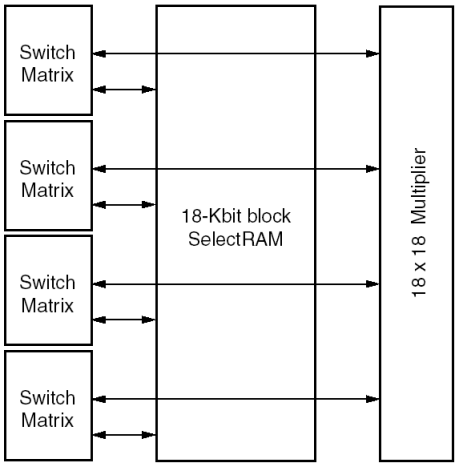

Figura 2.5: Memoria de Bloque y Multiplicador

Los multiplicadores son de 18x18 bits con signo en complemento a dos, y producen un resultado de 36 bits. Los bloques de memoria BlockRAM de 18Kbit pueden configurarse para un ancho de palabra variable, con palabras de 1,2,4,9,18 y 36 bits. Los multiplicadores y las memorias de bloque comparten la lógica de interconexión con los demás elementos, tal como se muestra en la figura 2.5, por lo que si ambos se usan, la memoria debe configurarse como de 18 bits de ancho de palabra. La interconexión está optimizada para que el bloque de RAM alimente al multiplicador contiguo.

El Virtex-II posee bloques dedicados para la alimentación de los relojes del sistema, y éstos están localizados en los bordes superior e inferior, en el centro. Internamente, estas señales son llevadas hasta el centro y distribuidas por redes especializadas de bajo retraso a todo el FPGA. Sin embargo, estas redes están localizadas por cuadrantes, por lo que en circuitos con muchos relojes, esto debe ser cuidado. Los bloques de DCM permiten la inversión, desfase, multiplicación y división de una señal de reloj con retrasos mínimos. Para

mayor información de la arquitectura y de parámetros funcionales, refiérase a [Xil02].

2.3. Aritmética Computacional

En esta sección presentamos varias maneras de representar números no enteros, para luego concentrarnos en representaciones de puntos flotante y en particular en la representación IEEE754 así como en la manera de realizar operaciones de suma y multiplicación en ella.

En la representación binaria de números enteros sin signo, un número x de n bits donde $x = b_{n-1}b_{n-2}...b_1b_0$ representa al número decimal $d = b_{n-1}2^{n-1}+b_{n-1}2^{n-2}+...+b_12^1+b_02^0$, por lo que $x \in [0, 2^n - 1]$.

Para representar números enteros con signo la forma más usada es mediante la representación de complemento a dos, donde el negativo de un número se obtiene negando todos los bits y sumándole una unidad. Por lo tanto, si definimos a y como el complemento de x, $y = -x = x_{compl} = x_{neg} + 1$. La representación de complemento a dos permite representar números enteros con signo en el rango $[2^{n-1}, 2^{n-1} - 1]$, y el valor decimal de un número en complemento a dos de n bits es $d = -b_{n-1}2^{n-1} + b_{n-1}2^{n-2} + ... + b_12^1 + b_02^0$.

Para representar números no enteros, pueden usarse representaciones de punto fijo o de punto flotante. En la representación de punto fijo más sencilla, conocida como base fraccional o base Q, se cambian los pesos de los bits, de manera de que los exponentes de las potencias de dos puedan ser positivos o negativos, produciendo valores por encima y por debajo de uno. Si se toma Q como la posición del punto decimal, un número en base fraccional de n bits en complemento a dos representa a un número decimal de valor $d = -b_{n-1}2^{n-Q-1} + b_{n-1}2^{n-Q-2} + ... + b_12^{-Q+1} + b_02^{-Q}$, o $d = -b_{n-1}2^{n-Q-1} + \sum_{i=0}^{n-2} b_i2^{i-Q}$.

2.3.1. Representaciones en Punto Flotante

En las representaciones mencionadas hasta el momento, cada número binario representa un número decimal, y la diferencia entre cada uno de los números decimales es fija para todo número dentro de esa representación. En cambio, en los números de punto flotante la precisión varía dependiendo del rango, por lo que los números en determinados rangos se

representan con mayor precisión que otros.

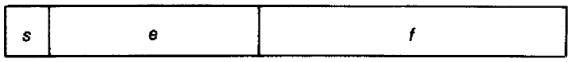

Figura 2.6: Número en Punto Flotante

Una manera tradicional de representar un número de punto flotante es determinando un número de bits para un significando (o mantissa), para un exponente y para un signo, tal como se muestra en la figura 2.6. Por lo tanto, el número a representar viene dado por la fórmula $d = (-1)^s \cdot f \cdot 2^e$, donde f es un número de representación fraccional y e es un entero con signo. Esta es la base de la representación IEEE754.

Otras representaciones existen para números de punto flotante, un ejemplo es el uso de bases logarítmicas en vez de exponenciales, donde f es un número entero, y e es un número fraccional. Esta representación no es muy usada, pero tiene ciertas ventajas para algunas aplicaciones. Para un estudio más completo de las diferentes representaciones númericas, se puede consultar [Par00].

2.3.2. Estándar IEEE754

El estándar IEEE754, [Boa85], usa una representación similar a la presentada en la figura 2.6. En el estándar existende dos formatos, los cuales difieren en el número de bits utilizados para la representación, y por lo tanto, influye en la precisión. Por ello, son conocidos como el formato de precisión simple, que representa los números mediante una palabra de 32 bits, y el formato de precisión doble, que utiliza 64 bits. A continuación se detallan:

- **S**, Un bit para el signo del número a representar.
- **e**, Un exponente desplazado, de manera que $e = E + desplazamiento$. En precisión sencilla e son 8 bits y $e \in [-126, +127]$, por lo que el desplazamiento es de 127, mientras que en precisión doble e son 11 bits y $e \in [-1022, +1023]$, por lo que el desplazamiento es de 1023.

- **f**, La parte fraccional del número $p = 1.f$, donde f es un número entre 0 y 1 representado en base Q. En precisión sencilla f son 24 bits, por lo que Q=24, mientras que en precisión doble f son 53 bits y Q=53.

Por lo tanto, un número en representación IEEE754 viene dado de la siguiente forma:

$$d = (-1)^S \cdot p \cdot 2^E = (-1)^S \cdot (1.f) \cdot 2^{e-desplazamiento}$$

A esto se suman las representaciones de números especiales, que son las siguientes:

- **Cero**,(Z), Se representa con $e = 0$ y $f = 0$.
- **Infinito**,(INF), Se representa con $e = 255$ y $f = 0$. El signo determina si es infinito positivo o negativo.
- **Not a Number**,(NaN), Se representa con $e = 255$ y $f \neq 0$. El signo no es importante.
- **Números denormalizados**,(denormals), Se representan con $e = 0$ y $f \neq 0$. En este caso, $d = (-1)^S \cdot (0.f) \cdot 2^{-126}$.

2.3.3. Operaciones básicas en el estándar IEEE754

En esta sección describiremos como realizar multiplicaciones y sumas con números en punto flotante.

Multiplicación

Dados dos números $a = S_a \cdot p_a \cdot 2^{e_a}$ y $b = S_b \cdot p_b \cdot 2^{e_b}$ en representación IEEE754, que es una representación exponencial, el producto $c = a \times b$ consiste en tres partes separadas:

- **Signo**, Dado los signos de S_a y S_b, si ambos son iguales el signo del resultado es positivo, y si son diferentes el signo del resultado es negativo.

- **Exponente**, Dado los exponentes de e_a y e_b, el exponente resultado es la suma de los exponentes. Sin embargo, hay que recordar que los exponentes tienen un desplazamiento, por lo que es necesario quitar el desplazamiento, sumar los exponentes y volver a sumar el desplazamiento para obtener la representación correcta.

- **Significando**, Dadas los siginificandos de p_a y p_b, el significado resultado es el producto de los significandos. Sin embargo, dado que $p_a \in [1, 2)$ y $p_b \in [1, 2)$, el producto de ambos se encontrará en $[1, 4)$, por lo que puede ser necesario ajustar el valor del resultado y del exponente para que se represente de manera correcta en el formato IEEE754.

A esto hay que agregar el trato de los números especiales:

- Si uno o ambos de los operandos es cero, el resultado es cero.

- Si uno o ambos de los operandos es NaN, el resultado es NaN.

- Si uno o ambos de los operandos es INF, el resultado es INF.

Suma

El cálculo de la suma de números en punto flotante es más complicado que la multiplicación. Dados dos números $a = S_a \cdot p_a \cdot 2^{e_a}$ y $b = S_b \cdot p_b \cdot 2^{e_b}$ en representación IEEE754, para sumar de manera directa los significandos es necesario alinear los números, esto es, llevar el exponente del número menor al del número mayor. Viendo como determianr cada una de las partes del resultado tenemos:

- **Signo**, Depende de los signos S_a, S_b, y del signo de la suma de p_a^* y p_b^*.

- **Exponente**, El exponente del resultado se obtiene al alinear los números a y b. Se determina el mayor entre e_a y e_b, obteniendo e_c, y se divide el significando del menor entre 2^{diff}, donde diff es la diferencia entre ambos exponentes. De esta manera, ambos números se encuentran en el mismo exponente, y se pierden bits de precisión en el número que contribuye menos con el resultado.

- **Significando**, Una vez alineados, es necesario pasar los significandosa complemento a dos si los operandos son negativos para poder sumarlos directamente. Una vez más, tomando en cuenta el rango, vemos que si $p_a^* \in [1, 2)$ y $p_b^* \in [1, 2)$, el resultado $p_c \in [0, 4)$, por lo que una vez más puede ser necesario ajustar el significado y el exponente del resultado para representarlos de manera correcta en el formato IEEE754.

Agregando las siguientes consideraciones para los números especiales:

- Si uno o ambos de los operandos es NaN, el resultado es NaN.
- Si ambos operandos son INF y del mismo signo, el resultado es INF con el mismo signo.
- Si ambos operandos son INF pero de signo diferente, el resultado es NaN.
- Si uno de los operandos es INF, el resultado es INF.

Capítulo 3

HAVOC: Arquitectura y Descripción Funcional

HAVOC, HArdware VOice Coprocessor, es un coprocesador basado en FPGA que apoya al reconocedor de voz SPHINX en el cálculo de las densidades gaussianas. En este capítulo se describirá el cálculo a realizar, los bloques constitutivos del procesador, su mapa de memoria y la manera de accesar y utilizar la funcionalidad del coprocesador implementado desde la PC.

3.1. Cálculo a realizar

Nos interesa realizar el cálculo de la siguiente distribución gaussiana:

$$b_j(o_t, \mu_j, \sigma_j) = \frac{1}{\prod_{i=1}^{d} \sqrt{2\pi\sigma_{ji}}} e^{-\frac{1}{2}\sum_{i=1}^{d}\left(\frac{o_{ti}-\mu_{ji}}{\sigma_{ji}}\right)^2}$$

donde o_t es el vector de observaciones, μ_j la media y σ_j la varianza. Esta fórmula se usa muchas veces como parte del proceso iterativo de cálculo de los parámetros del reconocedor de voz. Para calcular b_j más eficientemente, se hace en espacio logarítmico, y luego se devuelve el resultado. Esto permite deshacernos de la exponenecial y simplificar las operaciones a realizar. Entonces tenemos que:

$$\log b_j = -\frac{1}{2} d \log 2\pi - \frac{1}{2}\sum_{i=1}^{d} \log \sigma_{ji} - \frac{1}{2}\sum_{i=1}^{d}\left(\frac{o_{ti}-\mu_{ji}}{\sigma_{ji}}\right)^2$$

donde el primer término es una constante, el segundo termino no depende de t, por lo

que no cambia con el tiempo sino que puede calcularse en avance para cada distribución, y el último término puede reescribirse como:

$$\frac{1}{2}\sum_{i=1}^{d}\left(\frac{o_{ti}-\mu_{ji}}{\sigma_{ji}}\right)^2 = \sum_{i=1}^{d}\frac{1}{2\sigma_{ji}^2}(o_{ti}-\mu_{ji})^2 = \sum_{i=1}^{d}K_{ji}(o_{ti}-\mu_{ji})^2$$

y esta última expresión es la que a continuación vamos a detallar cómo se realiza en SPHINX y luego en el coprocesador.

3.1.1. En SPHINX

En SPHINX el cálculo de las densidades gaussianas se realiza en la librería `modinv`, en el archivo `gauden`. Las funciones principales para ello son `gauden_compute`, `log_full_densities`, `log_topn_densities` y `log_diag_eval`.

La función `log_full_densities` calcula todas las densidades para una determinada mezcla gaussiana, llamando a la función `log_diag_eval`, como se puede ver a continuación:

```
void
log_full_densities(float64 *den,
   uint32  *den_idx,/* the indices of the component densities */
   uint32   n_density,/* The number of component densities of the mixture */
   uint32   veclen,/* the length of the feature vector */
   vector_t obs,/* A feature vector observed at some time */
   vector_t *mean,/* means of the mixture density */
   vector_t *var,/* variances of the mixture density */
   float32  *log_norm) /* normalization factor for density */
{
    uint32 i;

    for (i = 0; i < n_density; i++) {
      den[i] = log_diag_eval(obs, log_norm[i], mean[i], var[i], veclen);
      den_idx[i] = i;
    }
}
```

La función `log_topn_densities` es una modificación de `log_full_densities` que sólo calcula las n densidades más significativas, por lo que no es necesario presentarla.

La función `log_diag_eval`, mostrada a continuación, calcula la fórmula antes descrita, donde el valor de `norm` ha sido calculado con antelación:

```
float64
log_diag_eval(vector_t obs,
      float32 norm,
      vector_t mean,
      vector_t var_fact,
      uint32 veclen)
{
    float64 d, diff;
    uint32 l;

    d = norm; /* log (1 / 2 pi |sigma^2|) */

    for (l = 0; l < veclen; l++) {
      diff = obs[l] - mean[l];
      d -= var_fact[l] * diff * diff; /* compute -1 / (2 sigma ^2) * (x - m) ^ 2 terms */
    }

    return d;
}
```

3.1.2. En el coprocesador

Observando el código presentado en la sección anterior, se puede obervar que el peso de los cálculos radica en la ejecución de la función `log_diag_eval` muchas veces, en particular, las operaciones dentro del bloque `for`. Por lo tanto, el trabajo del coprocesador se concentra en realizar este mismo trabajo de una manera más rápida. Para ello, se implementa la siguiente operación en el coprocesador:

$$\sum_{i=1}^{d} K_{ji}(o_{ti} - \mu_{ji})^2 \tag{3.1}$$

Por lo tanto, el coprocesador recibe lun vector de observación o_t, un vector de medias μ_t, y un vector de constantes precalculadas K_j, y devuelve un valor b_j que corresponde a una densidad gaussiana. Cada una de las operaciones se realiza con números de punto flotante de precisión sencilla, aún cuando en SPHINX para el resultado de la operación se utiliza un número de punto flotante de precisión doble. Esto se hace así dado que de otra manera

implicaría el manejo de unidades de precisión sencilla y precisión doble, así como el manejo de conversiones entre una y otra.

3.2. Implementación

Implementándose como un coprocesador, HAVOC responde a comandos y solicitudes de la PC en la que está instalada. Esto es, HAVOC es un periférico que funciona en modo esclavo, donde el programa en la PC escribe los datos de entrada, instruye al coprocesador a calcular, verifica si HAVOC a terminado de calcular y a continuación lee los resultados de los cálculos.

El diseño se implementó en una tarjeta RACE-1 diseñada y proporcionada por la Universidad de Mannheim, Alemania. Dicha tarjeta consta de un FPGA Virtex-II de la compañía Xilinx, con una capacidad de 3 millones de compuertas y grado 4 (x2v3000bf957-4), un procesador PLX9656 dedicado a la comunicación con un bus PCI de 32/64 bits y 33/66 MHz, 4 bancos de memorias ZBT de 512k*32 cada uno (2MB) con buses independientes, y un banco de memoria SDRAM de 32MB*32 (128MB). También cuenta con dos relojes independientes, uno con una frecuencia máxima de 64MHz que se utiliza para la comunicación entre el PLX y el FPGA, y otro con una frecuencia máxima de 125MHz que se utiliza para los bancos de memorias y el FPGA.

En las próximas secciones se muestra la configuración de entrada salida y el mapa de memoria para el coprocesador sin utilizar las memorias ZBT como caché de datos. Luego se presenta una descripción rápida de las unidades de cómputo, y a continuación los cambios hechos para añadir caché de datos al diseño.

3.2.1. Comunicación con la PC

Toda la comunicación entre la PC y el FPGA se realiza a través del bus PCI, por intermedio del PLX. El PLX presenta a la PC un rango de direcciones accesible a través del bus PCI, y permite realizar diversas operaciones de lectura y escritura sobre dicho rango. A su vez, el PLX presenta al FPGA dichas operaciones mediante un bus local, mediante una

interfaz simplificada que libera al FPGA del manejo de transacciones del bus PCI, a la vez que permite frecuencias de operación separadas.

El bus PCI es un bus de 32 ó 64 bits y 33 ó 66 MHz, dependiendo de la funcionalidad disponible en el bus de la PC. En nuestro caso se utilizó un bus de 32 bits y 33MHz. El bus local es de ancho y frecuencia configurable, en este diseño se utiliza como un bus de 32 bits y 33MHz. Por lo tanto, las direcciones son direcciones de palabras de 32 bits, aunque para accesarlas a través del bus PCI, es necesario usar direcciones de byte.

3.2.2. Diagrama Base

En las figuras 3.1 y 3.2 se muestra la entidad principal del coprocesador HAVOC. Consta de un Cliente de Local Bus (*LBC*) para la comunicación con la PC a través del PLX, el cual será explicado en detalle en el capítulo 4.1; de un bloque de configuración de los relojes del sistema (*HAVOC_clk*) y de un número configurable de Unidades de Cómputo (*GaussComputeUnit*).

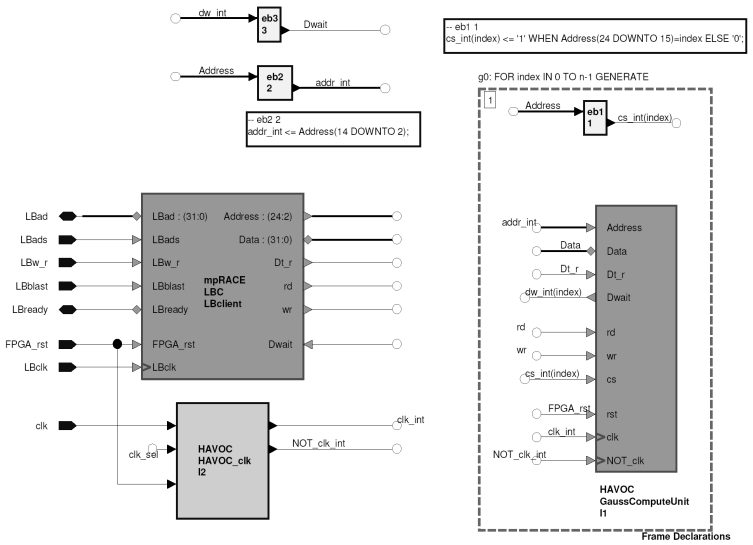

Figura 3.1: HAVOC, parte 1

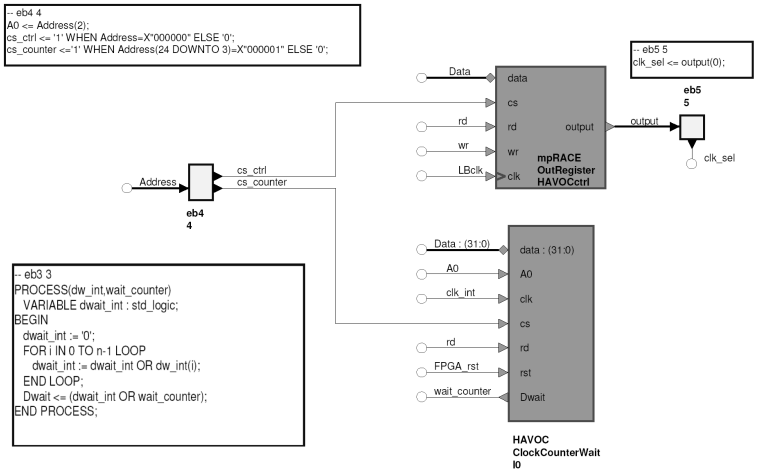

Figura 3.2: HAVOC, parte 2

3.2.3. Mapa de Memoria

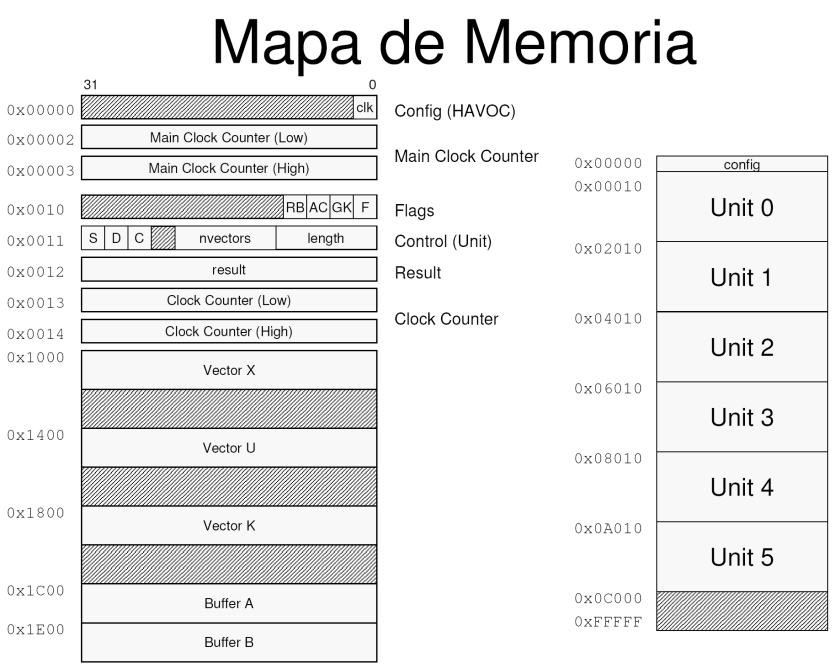

Figura 3.3: Mapa de Memoria General

El tamaño del FPGA permitió la incorporación de 6 unidades de cómputo independientes en paralelo. Estas unidades se mapean tal como se muestra en la parte derecha de la figura 3.3. Aqui se pueden distinguir dos áreas principales, un área para registros de configuración y estado globales, y un área para cada unidad de cómputo. El área global tiene disponible actualmente lo siguiente:

- **Un registro de Configuración**, que posee actualmente un único bit que permite seleccionar la relación entre el reloj del sistema y el reloj interno, alimentando directamente el reloj de entrada, o dividiéndolo por un factor de 2.5x. Este reloj es independiente del reloj que se utiliza para el bus local. Por los momentos, el diseño no soporta relojes asíncronos, por lo que no debe de usarse la opción del divisor.
- **Un contador de Reloj**, con un registro de 64 bits. Cuenta los pulsos de reloj desde el último reset del FPGA, y se puede utilizar como una referencia aproximada de tiempo.

El área de cada unidad de cómputo presenta el siguiente mapa de memoria puede verse en la parte izquierda de la figura 3.3, y consta de las siguientes partes:

- **Flags**, Un registro de banderas de estado, que contiene 4 banderas:
  - **F-Finished**, la unidad de cómputo ha finalizado los cálculos.
  - **GK-GaussKernel**, la unidad ha terminado la etapa de cálculo del Gauss Kernel, y espera para continuar con la acumulación. Sólo se activa si la unidad está en modo de debugging.
  - **AC-Accumulate**, la unidad ha terminado una iteración de la acumulación, y espera para continuar con la siguiente iteración. Sólo se activa si la unidad está en modo de debugging.
  - **RB-ResultBuffer**, indica el búffer (A ó B) del cual hay que leer los resultados de la acumulación cuando se están calculando más de un vector por búffer. Si RB=0, el resultado está en el Búffer A, si RB=1, el resultado está en el Búffer B.
- **Control**, Un registro de configuración y control de la unidad. Contiene los siguientes parametros para configurar y controlar la unidad de cómputo:
  - **length**, (bits 0-8). Aquí la unidad recibe la longitud del vector, menos uno. Este valor tiene un rango de 1 a 511, y siempre debe ser un número impar, de manera que el número de elementos en el búffer sea siempre un número par y además potencia de 2. Esto es, debe ser uno de los siguientes valores: 1,3,7,15,31,63,127,255,511.
  - **nvectors**, (bits 9-17). Aquí la unidad recibe el número de vectores por búffer, menos uno. Este valor tiene un rango de 0 a 511. El producto de la longitud del vector y el número de vectores no puede superar el tamaño de búffer, que es de 512 valores.
  - **S-Start**, (bit 31). Este bit controla el inicio de la unidad. La unidad comienza a calcular cuando detecta una transición de 0 a 1 en este bit. Debe limpiarse por software una vez que se han completado los cálculos.

- **C-Continue**, (bit 30). Este bit controla el avance entre etapas y/o iteraciones cuando la unidad se encuentra en modo de debugging.
- **D-Debug**, (bit 29). Este bit habilita el modo de debugging. Este modo es usado principalmente para confirmar la correcta operación de la unidad de cómputo y permitir la lectura de valores intermedios de los cálculos.

- **Acc**, Un registro de resultado del sumador. este registro contiene el valor de resultado cuando se opera con un sólo vector por búffer. Si se opera con más de un vector por búffer contiene el resultado para el último vector.
- **Un contador de reloj**. Consta de dos registros, la parte alta y la parte baja del contador. Se utiliza principalmente para benchmarking de la unidad de cómputo, y cuenta el número de ciclos de reloj del sistema desde que se recibe el bit de Start hasta que se activa el bit de Finished.
- **Tres buffers de entrada**. Estos búffers contienen los valores correspondientes a los vectores X, U y K, donde X corresponde a los valores de o_t, U a los valores de μ_t y K corresponde a $K_j i$ de las ecuaciones definidas al comienzo del capítulo. Cada búffer puede contener un máximo de 512 valores, correspondientes a uno o más vectores de datos, de acuerdo a los valores de los registros *length* y *nvectors*.
- **Dos buffers de salida**. Estos búffers, identificados con las letras A y B, son utilizados por el proceso de acumulación de los resultados del GaussKernel. El resultado final se encuentra en el búffer indicado por el bit *RB* del registro *Flags*.

3.2.4. Unidad de Cómputo

La unidad de Cómputo, implementada como la entidad *GaussComputeUnit*y que se puede ver en la figura 3.4, encapsula los registos de estatus y configuración, el contador de reloj, y una entidad *GKvector*. Esta última entidad, mostrada en la figura 3.5, es la que realiza actualmente el trabajo de cómputo y consta de 4 partes, a saber:

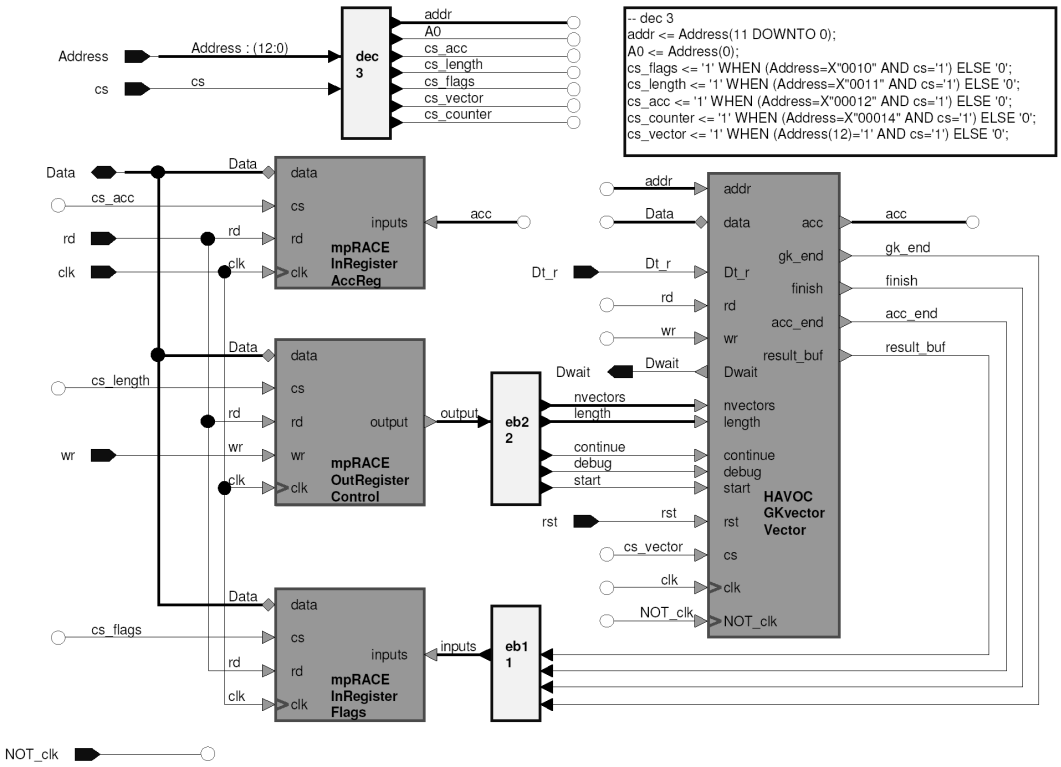

Figura 3.4: Entidad GaussComputeUnit

- **La Unidad de Control**. Representada por el bloque *GKvector_Control_pipeline*, contiene la máquina de estados que controla la operación de las etapas de cómputo.

- **La etapa del Gauss Kernel**. Representada por el bloque *GKvector_kernel*, calcula cada uno de los valores internos de la sumatoria. Contiene los búffers X, U y K.

- **La etapa de Acumulación**. Representada por el bloque *GKvector_accumulator*, realiza la suma de los valores calculados por la etapa anterior. Contiene los búffers A y B.

- **El Manejo de IO**. Representada por el bloque *GKvector_IO*, controla las señales de lectura, escritura y tristate de los búffers desde el exterior, y genera las señales de espera necesarias.

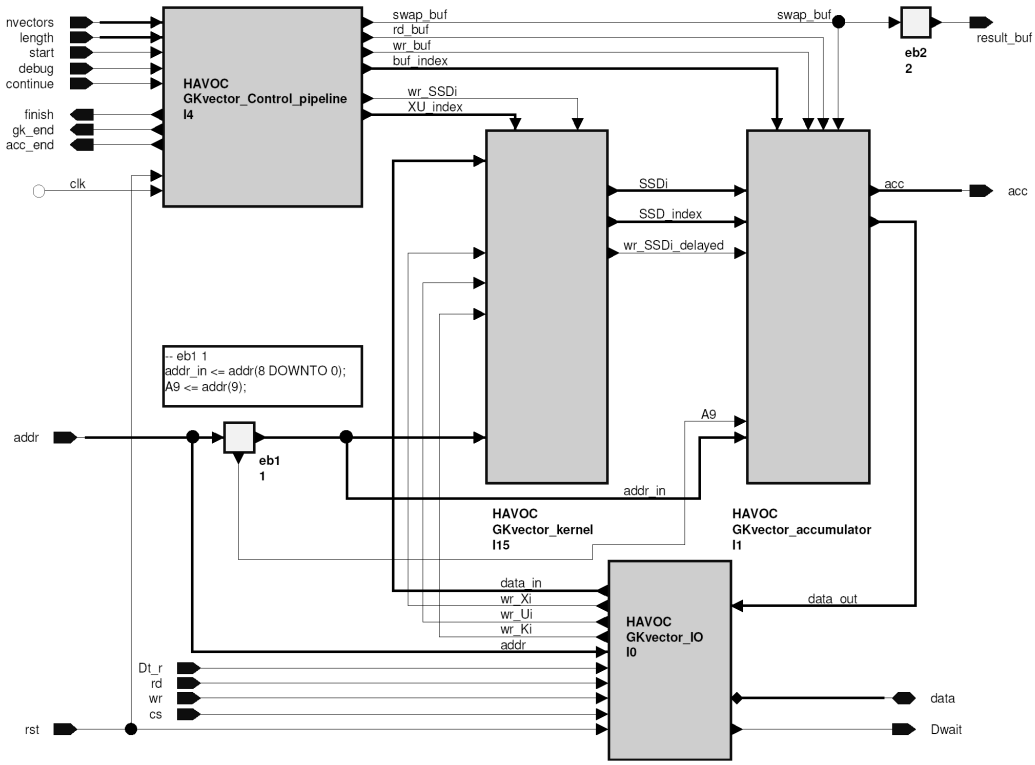

Figura 3.5: Entidad GKvector

Etapa Gauss Kernel

El bloque de GaussKernel del *GKvector*, que se muestra en las figuras 3.6 y 3.7 consta de los búffers de entrada X, U y K, de la entidad de cálculo *GaussKernel*, así como de elementos de retraso que permiten alinear los datos con las señales de control, ya que la entidad *GaussKernel* es segmentada y tarda 14 ciclos de reloj en producir el primer resultado. Por la misma razón, las señales de control del búffer K se retrasan 10 ciclos, para alinear los datos con los de los búffers X y U al momento de calcular.

La entidad *GaussKernel*, mostrada en la figura 3.8, es la que realiza el cálculo de la parte interna de la sumatoria de la ecuación 3.1. Consta de dos multiplicadores y un sumador de punto flotante de precisión simple completamente segmentados, los cuales serán explicados en detalle en los capítulos 5.1 y 5.2.

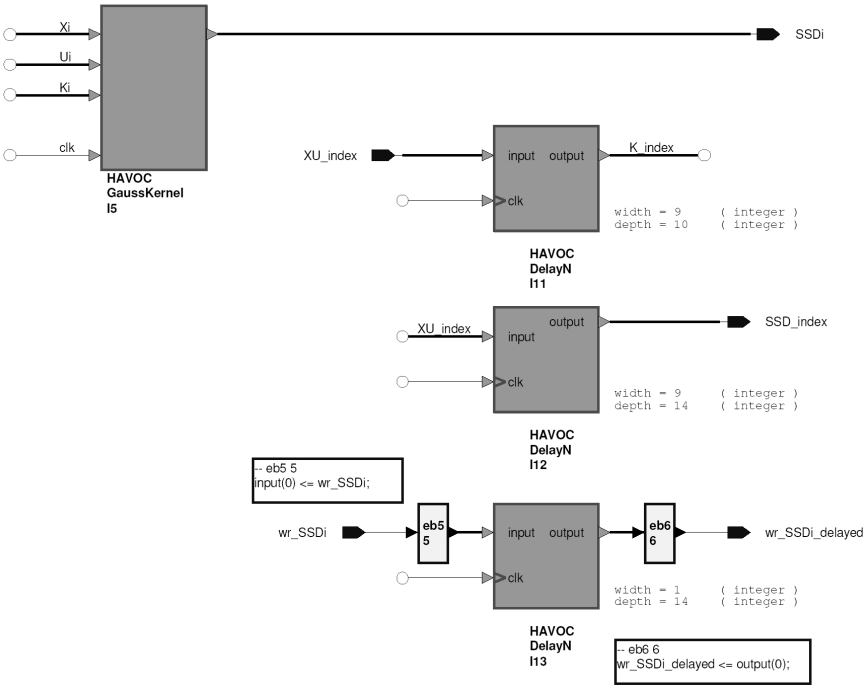

Figura 3.6: Bloque GKvector Kernel, parte 1

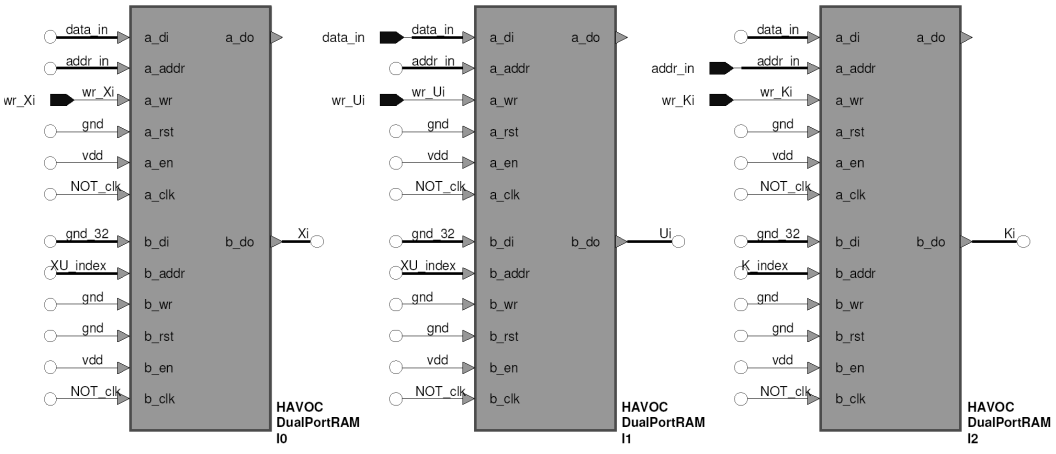

Figura 3.7: Bloque GKvector Kernel, parte 2

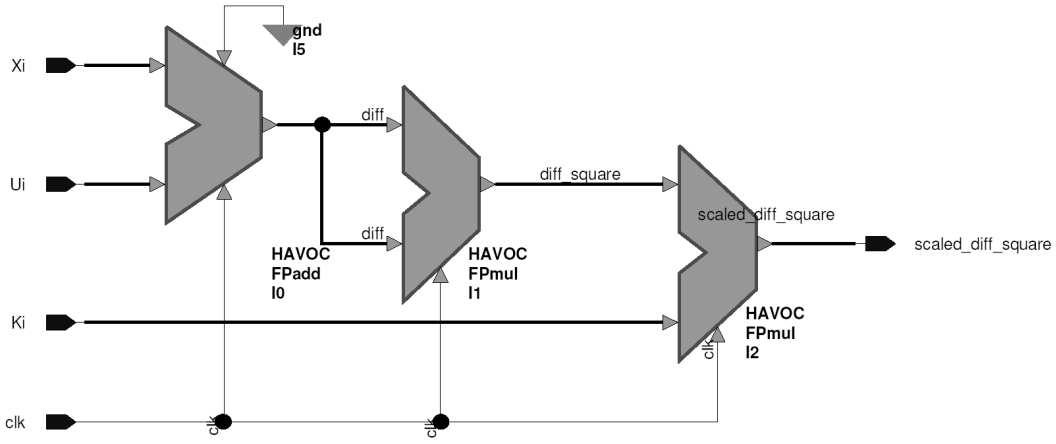

Figura 3.8: Entidad Gauss Kernel

Etapa de Acumulación

Esta etapa, que se muestra en las figuras 3.9, 3.10 y3.11, es la encargada de realizar la sumatoria de la ecuación 3.1. Consta de los búffers A y B, de un sumador de punto flotante de precisión simple, y de la lógica necesaria para conmutar las entradas y las salidas entre los dos búffers, así como para retrasar determinadas señales de control dado el retraso de la segmentación del sumador. Los elementos fueron calculados por la etapa anterior y almacenados en el búffer A. El algoritmo de suma será explicado en detalle en la descripción de la etapa de suma de la sección de control de la entidad *GKvector*. Esta etapa permite accesar los búffers A y B externamente, conmutarlos entre la entrada y la salida del sumador para realizar un proceso iterativo, y direccionar por separado los elementos pares e impares de un mismo búffer, gracias a que se utilizan memorias de doble puerto.

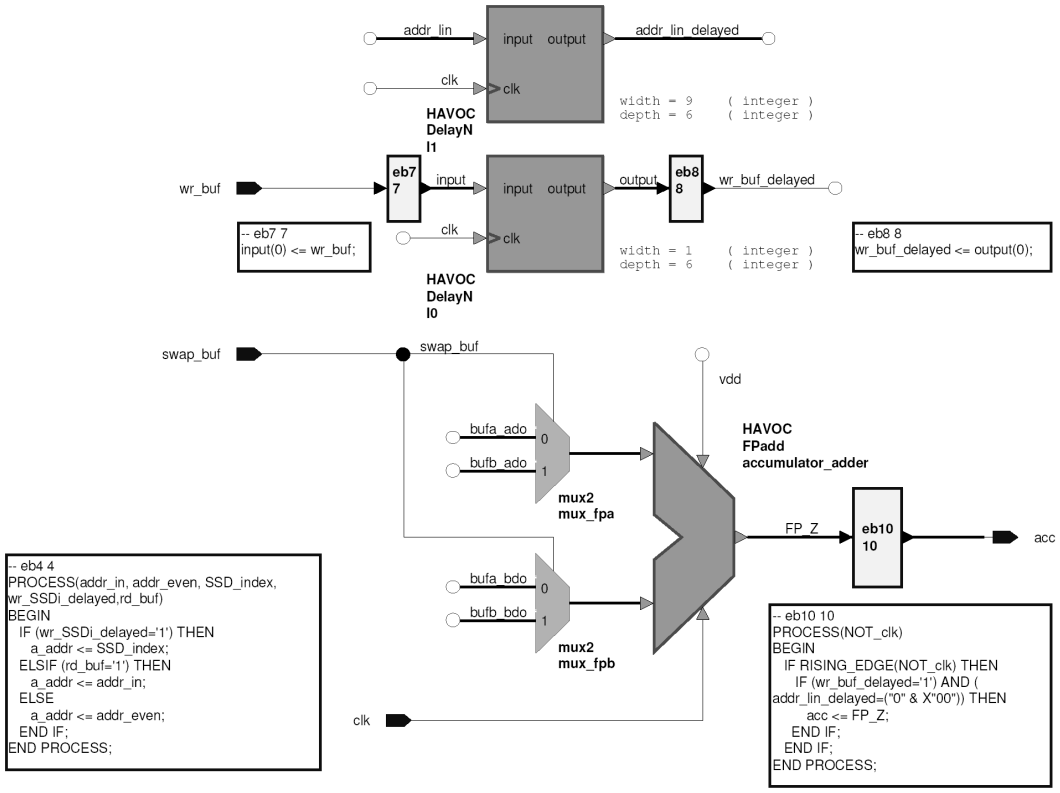

Figura 3.9: Bloque GKvector Accumulator, parte 1

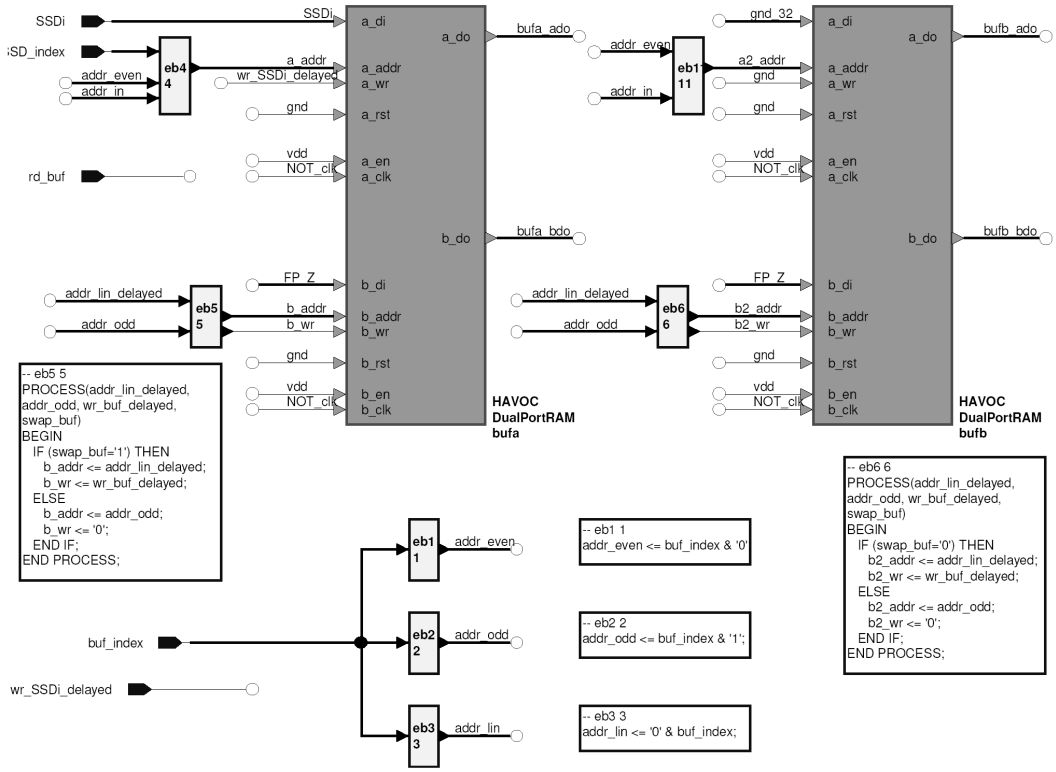

Figura 3.10: Bloque GKvector Accumulator, parte 2

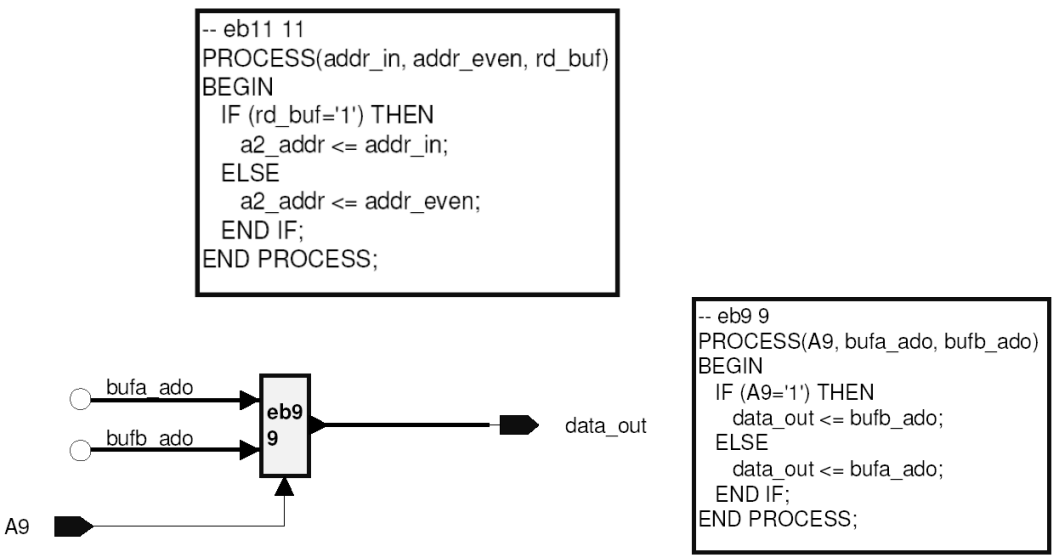

Figura 3.11: Bloque GKvector Accumulator, parte 3

IO

Este bloque, que se puede apreciar en la figura 3.12, genera las señales de control para accesar los contenidos de los búffers (X, U, K, A y B) desde el exterior de la entidad *Gaussvector*. En particular, dichas señales están habilitadas solamente cuando la Unidad de Cómputo no se encuentra activa.

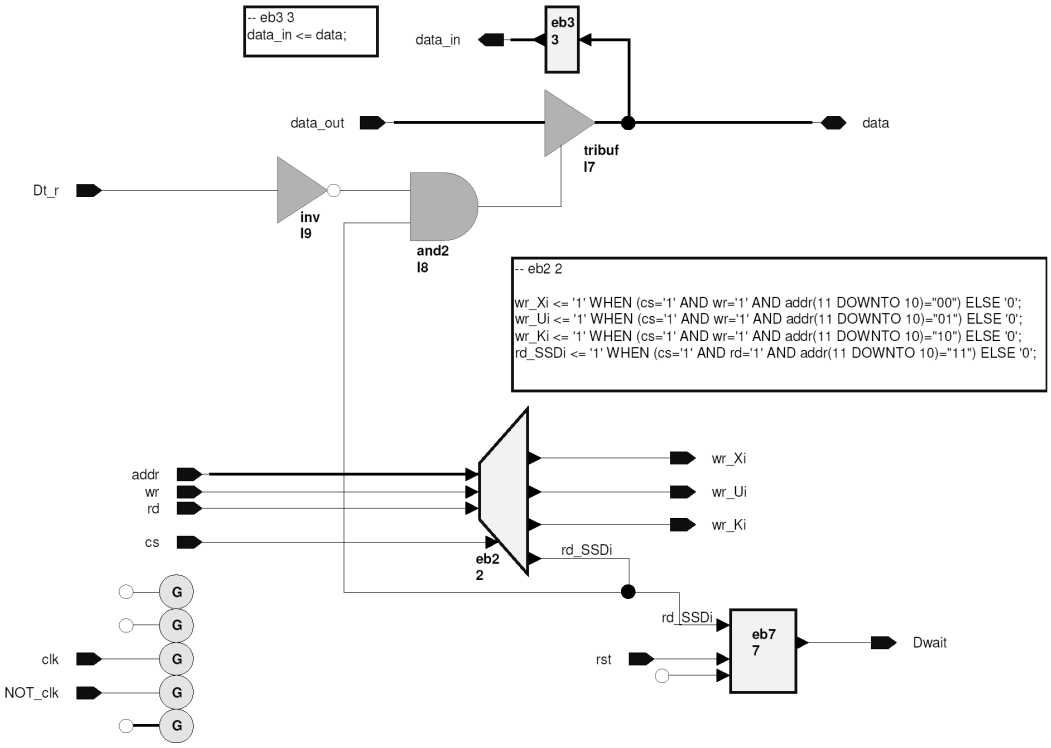

Figura 3.12: Bloque GKvector IO

Unidad de Control

El bloque de la Unidad de Control es una máquina de estados que gobierna el comportamiento de la Unidad de Cómputo. En particular, responde a las señales de entrada generando las señales de control necesarias para realizar el cómputo de la ecuación 3.1 de una manera eficiente. Las máquinas de estados correspondientes se detallan en las figuras 3.13 y 3.14. En 3.13 se detalla la máquina principal, mientras que en 3.14 se detalla la máquina asociado con la etapa de acumulación, correspondiente al estado *Add* de la figura anterior.

Inicialmente, el unidad de cómputo se encuentra en reposo esperando la señal de inicio (**S-Start**), que comienza la primera etapa, el cómputo de la parte interna de la sumatoria por el GaussKernel. La máquina de estados recorre todos los elementos del búffer, determinados por los valores de configuración **length** y **nvectors**, y coloca el resultado en el búffer A en la misma dirección de los datos de origen. Una vez finalizada con la cuenta, la unidad de control añade 14 ciclos de espera para permitir que se vacíen las etapas segmentadas.

Una vez completada esta etapa, la unidad de control verifica si se ha solicitado el modo de debugging (**D-Debug**), en cuyo caso habilita la lectura externa de los búffers y espera por la señal de continuar (**C-Continue**) para pasar a la etapa de acumulación. Si no se ha solicitado el modo de debugging, apsa directamente a la etapa de acumulación.

La etapa de acumulación calcula la suma de los valores de cada vector. Como el sumador es segmentado, si quisieramos sumar secuencialmente todos los números tendríamos que vaciar el pipeline por cada suma, por lo que el cálculo de N vectores de longitud L tomaría $N(L-1)$ sumas y $6N(L-1)$ ciclos de reloj, lo cual es bastante ineficiente ya que no aprevacha la capacidad de segmentación del sumador. Por lo tanto, la suma se realiza iterativamente, sumando todos los números impares con todos los pares de un búffer, y guardándolos en el otro búffer, para luego conmutar los búffers y repetir la operación, hasta obtener un único valor por cada vector. La máquina de estados de la figura 3.14 implementa el siguiente algoritmo para realizar este cálculo:

```
length=length/2

while length <> 0
{
   addr=0
   for i=0 to nvectors-1 do
      for j=0 to length-1 do
       addr=addr+1
   flush_adder
   length=length/2
     swap_buffers

     if is_debug
        wait_for_continue
}
```

dada la restricción de que la longitud del vector **length** sea par y que sea una potencia de 2, entonces siempre es un número par de elementos a sumar en cada iteración. Este algoritmo permite calcular la suma de N vectores de longitud L en $\log_2 L$ iteraciones en un total de $N(L-1)$ sumas y $N(L-1) + 6 \log L$ ciclos de reloj. De esta manera, la segmentación del sumador es utilizada de una manera mas eficiente.

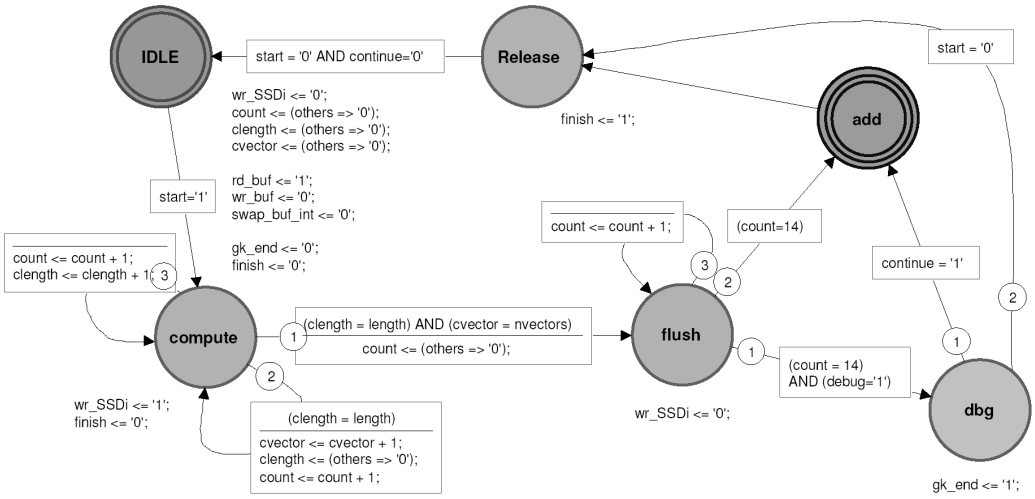

Figura 3.13: Bloque GKvector Control pipeline

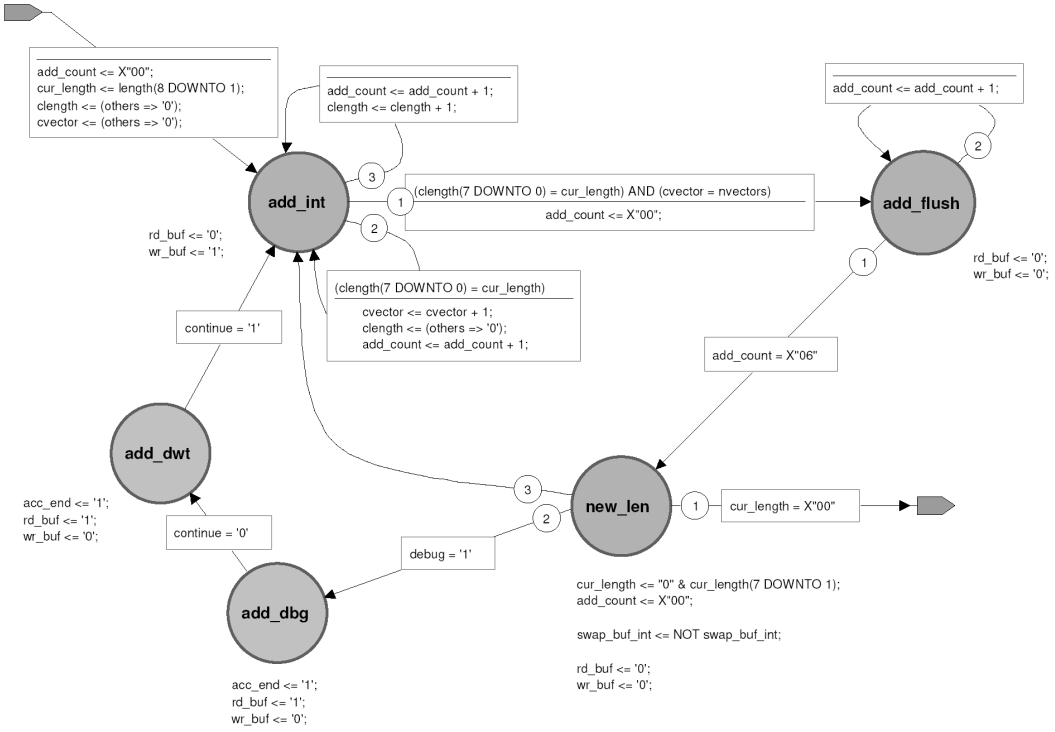

Figura 3.14: Estado Add del bloque GKvector Control pipeline

3.3. Caching de Datos

El manejo de antememorias para este diseño aumentó bastante la complejidad de su arquitectura, ya que implica escribir los vectores a calcular en las antememorias, utilizando para ello los bancos de memoria ZBT (un banco para cada conjunto de datos de entrada X, U, y K, y otro para el conjunto de datos de salida), para luego leer de ellos los datos para alimentar a las Unidades de cómputo y posteriormente escribir los resultados en la antememoria de salida. Adicionalmente es necesario generar las señales de control adecuadas para el funcionamiento de las Unidades de Cómputo. En principio, se optó por manejar ”páginas de memoria”, donde cada página correspondía a un búffer de 512 elementos, de manera de realizar una interfaz sencilla con el diseño existente.

Sin embargo, dicho esquema no pudo ser completamente validado por limitaciones de tiempo, además de que durante su implementación se hizo evidente que otras modificaciones de la arquitectura proveerían una solución más elegante y eficiente. El manejador de memorias ZBT desarrollado para esta etapa se documenta en 4.2, mientras que las modificaciones hechas a la arquitectura para añadir las antememorias no se incluyen en este documento, pero pueden consultarse directamente en los archivos de diseños contenidos en el CDROM adjunto.

Capítulo 4

Unidades Periféricas y de Soporte

4.1. Cliente de Bus Local

El Cliente de Bus Local, (*LBC*), permite comunicar al diseño que se implementa dentro del FPGA con la PC mediante el bus PCI, por medio del circuito integrado PLX9656 [PLX03]. A continuación se explicarán las capacidades de este chip, el modo como se comunica con el FPGA, y más adelante, cuando se detalle la implementación del cliente se explicará la manera como el cliente se comunica con el diseño interno en el FPGA, así como sus limitantes.

4.1.1. Interfaz PLX9656

El PLX9656 maneja toda la lógica de transacciones del bus PCI, y convierte dichas operaciones en accesos sobre un rango de direcciones de 32 bits y 8-32 bits de ancho (32 bits en nuestro caso) con un protocolo de comunicación simple, al que llama bus local. En particular, maneja asíncronamente los relojes del bus PCI y del bus local, de manera que si se desea, se puede tener un diseño de baja velocidad del lado del bus local sin necesidad de afectar el funcionamiento del bus PCI, que operará a su velocidad nominal (33 o 66MHz son las velocidades soportadas por el PLX), a la vez de que provee con la máxima tasa de transferencia entre ambos buses, y realiza automáticamente operaciones de alineación y conversión de ëndianess.̈ Del lado del bus local, el PLX provee señales de control y de datos direcciones, que dependen del modo de operación especificado. Los posibles modos determinan si los

datos y las direcciones usan buses separados o si comparten un mismo bus multiplexado. El modo M corresponde a un bus no multiplexado para conexión con microprocesadores Motorola. El modo C corresponde a un bus no multiplexado para microprocesadores Intel u otros dispositivos, mientras que el modo J corresponde a un bus multiplexado para microprocesadores Intel u otros dispositivos. El PLX9656 de la tarjeta RACE1 se encuentra configurado por diseño en modo J, utilizando un único bus para direcciones y datos.

Adicionalmente, el PLX permite el inicio de transacciones tanto desde el bus PCI como desde el bus Local. Si el bus PCI inicia una transacción, el PLX es un esclavo en el bus PCI y el maestro en el bus local. Si el bus local inicia la transacción, el PLX es un esclavo en el bus local y el maestro en el bus PCI. Como ya se ha comentado, el diseño se ha restringido a sólo comenzar operaciones desde la PC, esto es, todas las transacciones son iniciadas por el bus PCI, por lo que el PLX es siempre el maestro del bus local y el FPGA es siempre un esclavo en el bus local, esperando y respondiendo a transacciones iniciadas por el PLX.

Las señales de importancia entre ambos se describen a continuación.

- **LBad[31:0]**,*(Address/Data Bus)*. Es el bus bidireccional, multiplexado de direcciones y de datos. Las señales de control determinan el contenido del bus y la dirección de los datos.

- **LBads**,*Address Data Strobe)*. Determina si el valor presente en el bus *LBad* es una dirección o un dato. Si *LBads*=0, el bus contiene una dirección, en otro caso contiene un dato. Esta señal también marca el comienzo de un nuevo acceso al bus.

- **LBw_r**,*(Write/Read)*. Determina si se está realizando una lectura o una escritura. Si *LBw_r*=1, el PLX está escribiendo en el bus local en la dirección especificada, en otro caso está leyendo el valor de la dirección.

- **LBlast**,*(Burst Last Word)*. Especifica si es el último dato en una transferencia de ráfaga. Cuando *LBlast*=0, el dato en el bus es el último de una ráfaga.

- **LBready**, Permite al PLX saber si el dato presente en el bus ha sido procesado correctamente. En una lectura, le dice al PLX que el datos es válido. En una escritura,

le dice que al PLX que el valor ha sido escrito correctamente y que puede proceder. *LBready*=0 significa válido, otro valor añade un ciclo de espera a la transacción.

A continuación listamos otras señales de interés pero que no son indispensables para el funcionamiento básico de las transacciones en el bus local. Para más información de estas y de otras señales no listadas, favor de consultar el manual del PLX.

- **LBE[3:0]**, *(Byte Enable)*. Codifican el ancho del bus de datos y representan los bytes habilitados según la configuración. Se usa cuando hay buses de menos de 32 bits, para habilitar sólo el ancho correspondiente.

- **LHOLD**, *(Hold Request)*. Salida del PLX. Cuando el PLX activa esta señal, significa que solicita ser el maestro del local bus. Se usa cuando hay más de un dispositivo con capacidad de ser maestro del bus.

- **LHOLDA**, *(Hold Acknowledge)*. Entrada al PLX. Cuando esta señal se activa, significa que el control del bus le ha sido otorgado al PLX. Se usa para completar el protocolo de solicitud de bus cuando hay más de un maestro. El control no debe de ser asignado a menos que sea solicitado por el PLX mediante la señal *LHOLD*.

- **BTERM**, *(Burst Terminate)*. Cuando el PLX es maestro del bus local, instruye al PLX a cancelar una transferencia de ráfaga. Si todavía hay datos pendiente por transferir, el PLX cancela la ráfaga he inicia un nuevo ciclo a partir de la dirección pendiente.

- **Dt_t**, *(Data Transmit/Receive)*. Se utiliza para determinar la dirección de los transceivers del bus de datos. Cuando *Dt_r*=1, el PLX está colocando datos en el bus, sino, el PLX está leyendo datos del bus.

4.1.2. Transferencias Sencillas

Una transferencia sencilla requiere de dos ciclos del bus local para completarse. En el primer ciclo, el PLX coloca la dirección en el bus *LBad*, y marca el valor como dirección e inicio de acceso con la señal *LBads*=0. También especifica si la operación es una lectura o

una escritura con la señal *LBw_r*. En el siguiente ciclo, el PLX vuelve el valor de la señal *LBad*=1, para indicar que es un dato, y coloca el valor de *LBlast*=0, para indicar que sólo un valor va a ser transferido. Si la operación es una escritura, en este ciclo el PLX coloca el dato a escribir. Si es una lectura, el PLX coloca el bus en alta impedancia y espera recibir un valor válido. Durante el primer ciclo, la señal de *LBready* debe estar en alta impedancia, por requisito de operación del PLX; y durante el segundo ciclo *LBready*=0, para indicar que la tranferencia se realizó correctamente.

4.1.3. Transferencias en Ráfaga

Una tranferencia en ráfaga es muy similar a una sencilla. Consiste en colocar la dirección inicial y el tipo de transacción, y a continuación los datos secuencialmente, asumiendo que representan un bloque de memoria que comienza en la dirección especificada, y donde la última palabra de ese bloque se marca con la señal *LBlast*.

Por lo tanto, en el primer ciclo de una transferencia de ráfaga, el PLX coloca la dirección en el bus *LBad*, y marca el valor como dirección e inicio de acceso con la señal *LBads*=0. También especifica si la operación es una lectura o una escritura con la señal *LBw_r*. En el siguiente ciclo, el PLX vuelve el valor de la señal *LBad*=1, para indicar que es un dato, y coloca el valor de *LBlast*=1, para indicar que mas de un valor va a ser transferido. Si la operación es una escritura, en este ciclo el PLX coloca el primer dato a escribir, y si es una lectura, coloca el bus en alta impedancia y espera recibir el primer dato válido. En los ciclos sucesivos se repite esta transferencia de palabras, y se asume que cada dato corresponde a una dirección inmediatamente siguiente a la del dato anterior. Cuando el PLX llega al último dato que desea transferir, coloca la señal *LBlast*=0 para indicar que es el fin de la ráfaga y que en el siguiente ciclo el bus está libre. Durante el primer ciclo, la señal de *LBready* debe estar en alta impedancia, por requisito de operación del PLX; y durante los demás ciclos *LBready*=0, para indicar que la palabra correspondiente se transfirió correctamente.

Como se puede observar, una tranferencia sencilla es en realidad una transferencia de ráfaga de un solo dato.

4.1.4. Transferencias con ciclos de Espera

En las transferencias descritas hasta el momento, se asume que los datos están listos o pueden ser procesados en el mismo ciclo cuando son presentados. Sin embargo, la mayoría de los circuitos no responde tan rápidamente, por lo que es necesario incluir ciclos de espera en las transacciones. Para ello, en el ciclo del dato correspondiente se coloca la señal *LBready*=1. Esto hace que el PLX genere a su vez ciclos de espera en el bus PCI. Cuando el dato es finalmente válido o ha sido procesado correctamente, se coloca la señal *LBready*=0. Dependiendo de la señal *LBlast*, la transacción termina en ese ciclo o se espera un nuevo dato en el siguiente.

4.1.5. Implementación

La implementación actual del cliente de bus local que se muestra en la figura 4.1 le presenta al PLX un espacio de memoria contiguo de 22 bits. Las direcciones superiores del espacio de memoria están reservadas para otras operaciones de la tarjeta, en particular, para la programación del CPLD y del FPGA, así como para la configuración de los relojes. El LBC realiza las operaciones necesarias en el bus local y presenta al diseño interno con una interfaz simple de acceso de memoria, consistente de un bus de direcciones (*Address*), un bus de datos (*Data*), señales de lectura (*rd*) y escritura (*wr*), un control de transceivers (*Dt_r*) que permite saber la dirección del bus de datos, y una señal de entrada (*Dwait*) para instruir al LBC de añadir ciclos de espera cuando un dato no está listo.

En la figura 4.2 se puede observar el diseño del LBC. Consta de una máquina de estados que realiza las secuencias de las transacciones del bus local, un contador para ir incrementando automáticamente las direcciones en el bus interno durante las transferencias de ráfaga, y los buses de tres estados necesarios para manejar los buses bidireccionales. En la figura 4.3 se puede observar la máquina de estados mencionada anteriormente.

Es importante mencionar en este punto que la señal *LBready* cambia asíncronamente, y es necesario que ocurra así. Si se va a generar un ciclo de espera, debe generarse la señal en el mismo ciclo, sino el PLX interpreta que el dato ha sido procesado correctamente. Por lo

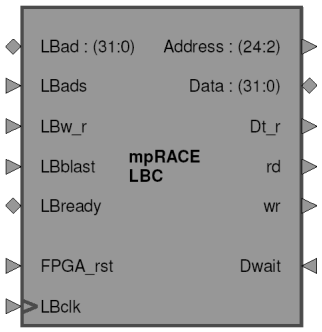

Figura 4.1: Puertos de la entidad LBC

tanto, la señal de *Dwait* en el diseño interno también debe ser generada asíncronamente. El hecho de no cumplir con los tiempos de la señal *LBready* bloqueará la transferencia en el bus PCI y muy probablemente inhabilitará a la computadora, ya que el PLX no liberará nunca el bus.

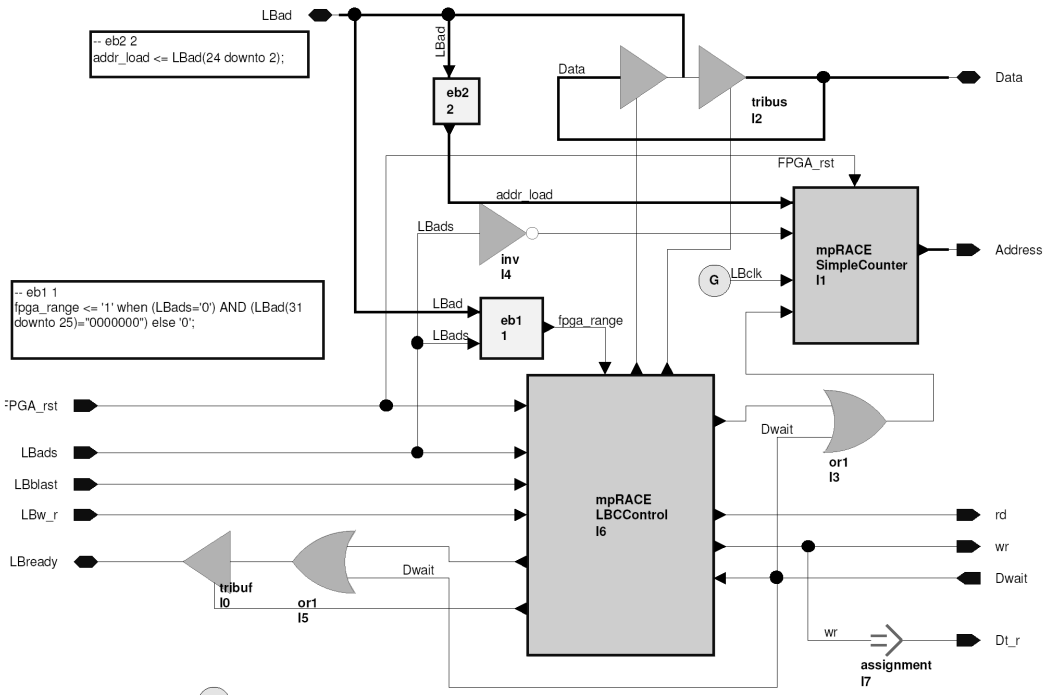

Figura 4.2: Local Bus Client

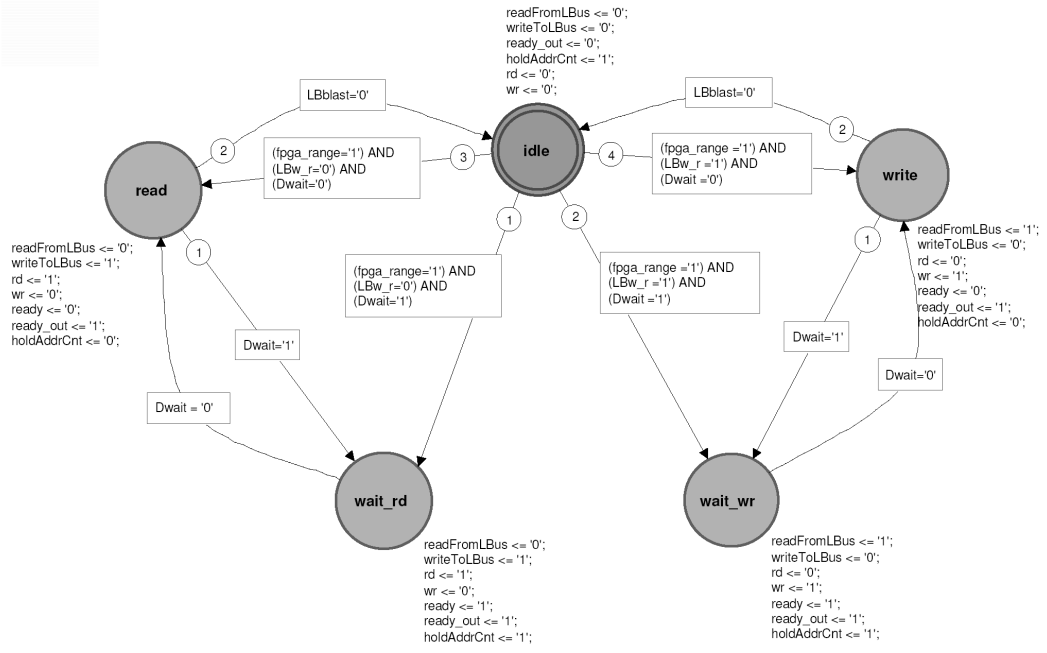

Figura 4.3: Máquina de Estados del LBC

4.1.6. Resultados

En esta sección mostramos los resultados del modelo de comportamiento del cliente de bus local realizando diferentes transacciones (figura 4.4), con retrasos en los accesos (figura 4.5) y en ráfagas (figura 4.6). Por último se muestran los diagramas de tiempo del modelo ya sintetizado para operar a 33MHz, en las figuras 4.7 y 4.8.

En la figura 4.4 puede verse una serie de trasacciones entre el bus local y las señales internas del bus de datos del LBC. Se realizan dos transacciones dentro del rango asignado al FPGA, una escritura fuera del rango, y posteriormente dos lecturas de las direcciones anteriores.

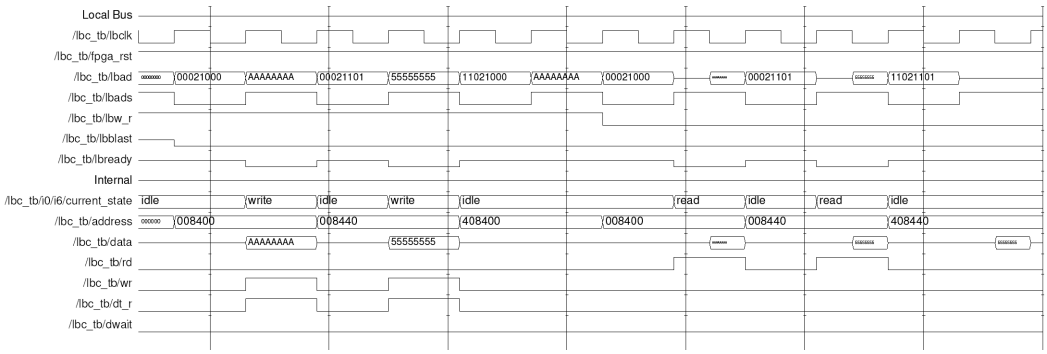

Figura 4.4: Escritura / Lectura Sencilla del LBC

En la figura 4.5 puede verse una escritura seguida de una lectura, donde ocurren retrasos en los accesos. Estos retrasos son generados desde la parte interna del diseño (bus de dirección y datos), mediante la señal *Dwait*. Obsérvese como el cambio de esta señal es asíncrono en su transferencia a la señal *LBready* para la subida, pero síncrono con el reloj en el flanco de bajada, esto es por un requisito del PLX, donde la maquina de estados garantiza que la señal esté en ready”durante un ciclo completo para la finalización correcta de la transacción por parte del PLX.

En la figura 4.6 se ve una escritura de ráfaga seguida de una lectura de ráfaga, sin ciclos de espera. En este diagrama la diferencia es la señal de *LBblast*, que es la que indica el final de una ráfaga. El cleinte de bus local permite que cada escreitura o lectura de una ráfaga tenga un número arbitrario de ciclos de espera. Sin embargo, esa situación no se simuló en

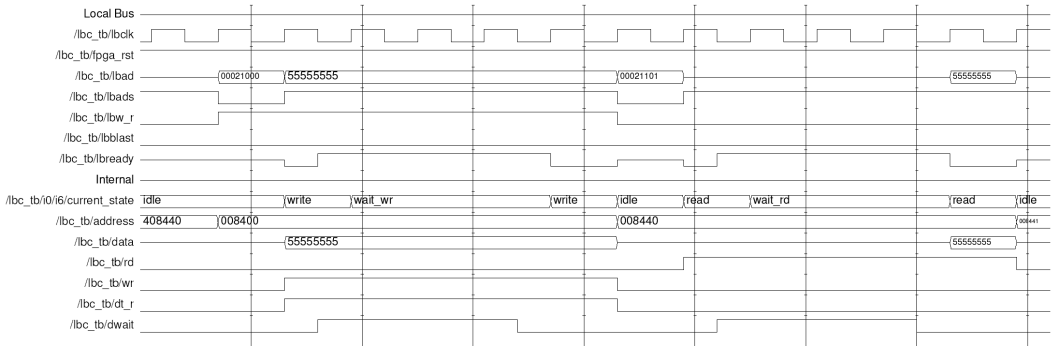

Figura 4.5: Escritura / Lectura Sencilla con esperas del LBC

los diagramas anexos.

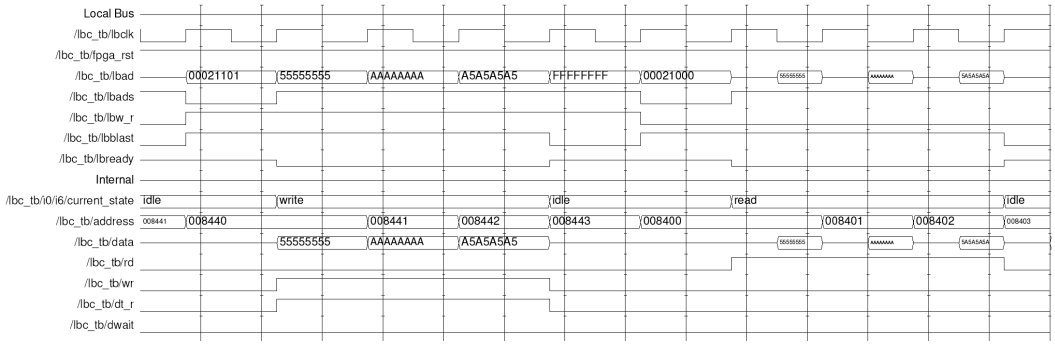

Figura 4.6: Escritura / Lectura en Ráfagas del LBC

En las figuras 4.7 y 4.8 se puede observar la simulación del comportamiento del circuito sintetizado a 33MHz, realizando las operaciones equivalentes a las figuras 4.4 y 4.6. En estas nuevas figuras pueden observarse la diferencia en los tiempos de propagación de las señales y su diferencia con los tiempos ideales de la simulación de comportamiento. En estos diagramas algunas señales no pueden observarse dado que fueron fusionadas en el diseño sintetizado por el sintetizador y el P&R.

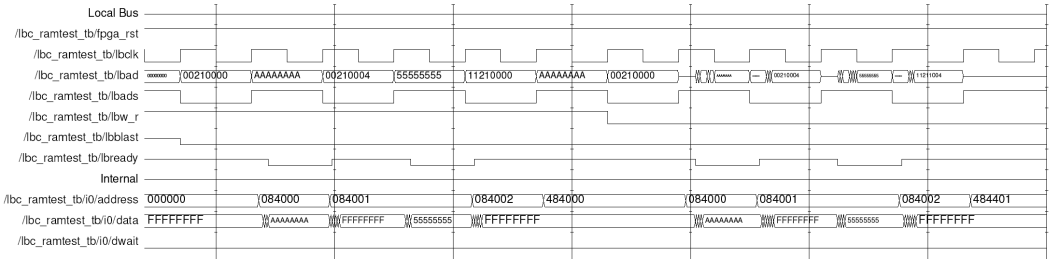

Figura 4.7: Escritura / Lectura Sencilla del LBC, Sintetizado a 33MHz

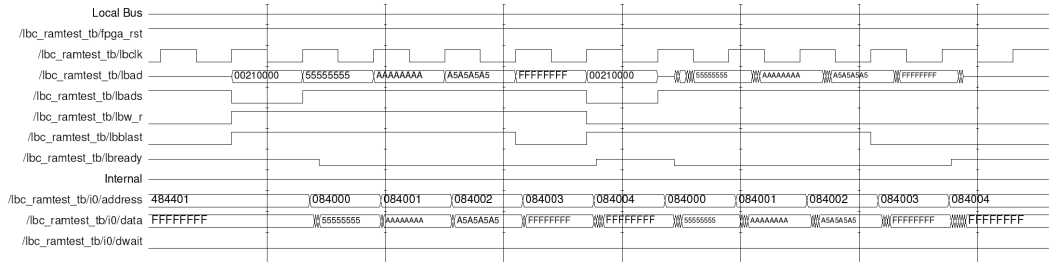

Figura 4.8: Escritura / Lectura en Ráfagas del LBC, Sintetizado a 33MHz

4.2. Manejador de Memorias ZBT

El manejador de memorias ZBT (ZBT Manager) permite el acceso a los bancos de las memorias ZBT por parte del LBC y del diseño interno en el FPGA mediante buses dedicados a cada uno de ellos. El manejador de memorias ZBT añade los ciclos de espera necesarios para el LBC, y le permite un acceso directo a las memorias al diseño interno. Un diagrama de los puertos se puede observar en la figura 4.9. En la tarjeta RACE1 existen cuatro bancos de memorias ZBT completamente independiente, y cada uno de ellos consiste de una memoria Samsung K7N163601A de 512k x 36. A continuación se detalla el funcionamiento de estas memorias y los detalles de la implementación del manejador. Cada banco de memoria utiliza un manejador propio independiente.

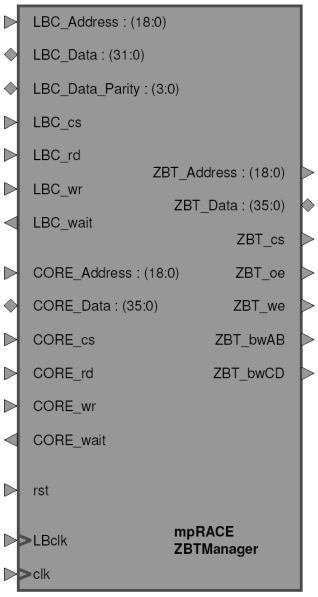

Figura 4.9: Puertos de la Entidad ZBTManager

4.2.1. Memorias Samsung K7N163601A

Las memorias Samsung K7N163601A [Sam02] son memorias estáticas de 512k x 36 con un diseño segmentado NtRAM (no turnaround RAM), tambien llamado de ZBT (Zero Burst Time). Estas memorias se caracterizan por tener un diseño interno segmentado, y añadir

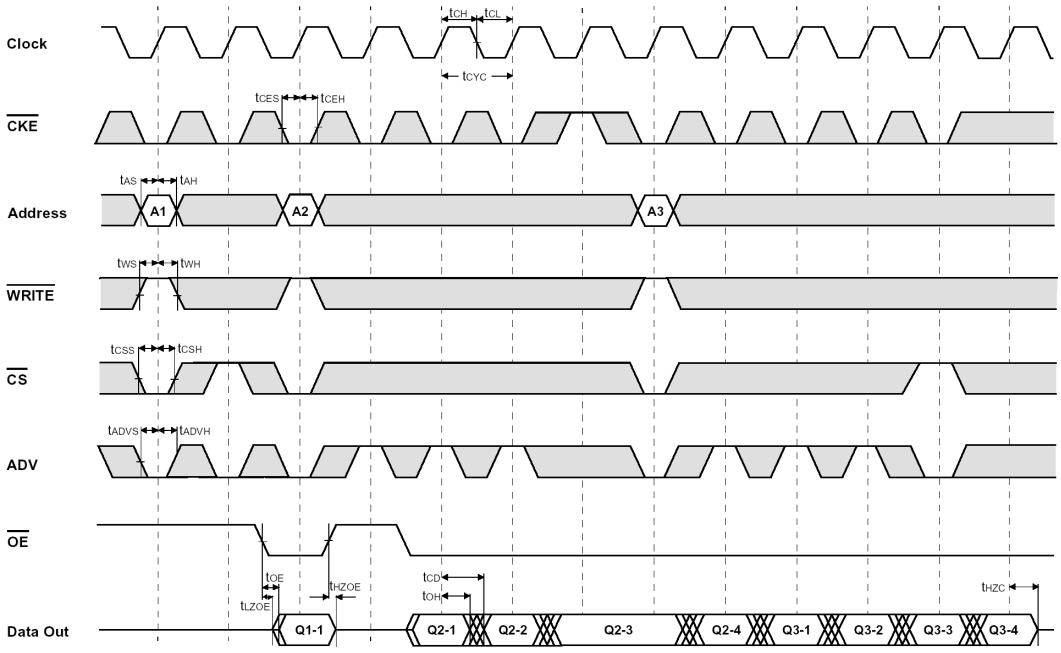

Figura 4.10: Ciclos de lectura de Memorias ZBT

dos ciclos de diferencia entre la entrada de la dirección y de la dirección. Al ser segmentado, durante cada ciclo se puede acceder una dirección diferente, por lo que en acceso de ráfagas, sólo existe un retraso inicial de dos ciclos, a partir de entonces los datos aparecen en cada ciclo de reloj, desfasados en dos ciclos con respecto al bus de direcciones.

Las señales de control *(CS,WE)* van registradas junto con la dirección, mientras que la señal de control de *OE* va registrada con el dato correspondiente, ya que controla la dirección del bus de datos. En las figuras 4.10 y 4.11 se muestran diagramas de tiempo de lecturas y escrituras de la memoria.

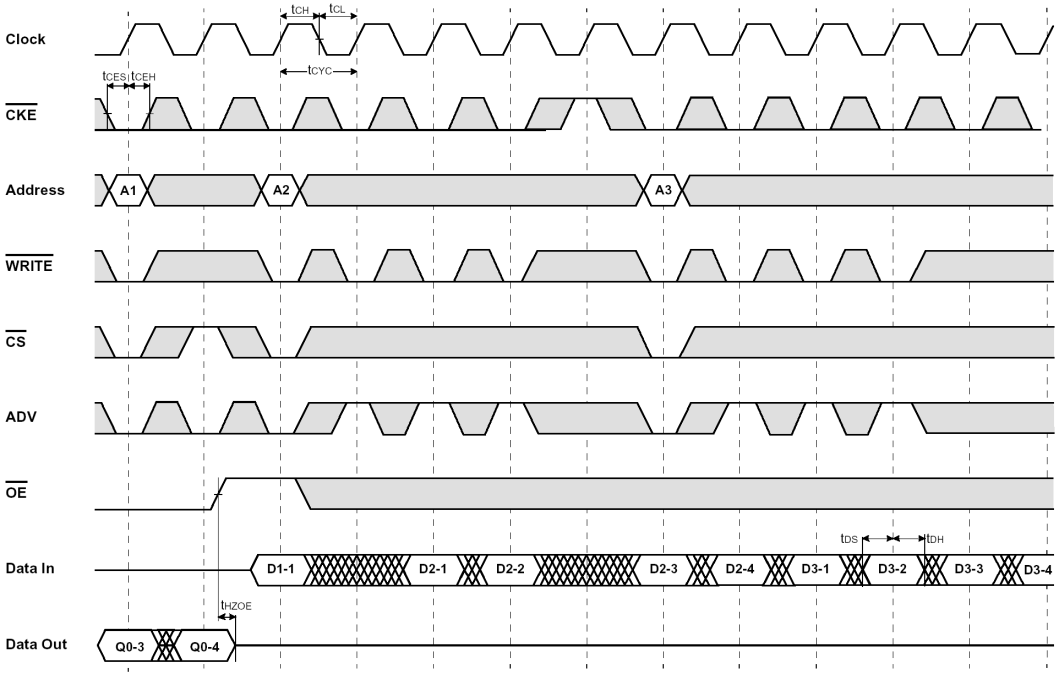

Figura 4.11: Ciclos de escritura de Memorias ZBT

4.2.2. Implementación

En las figuras 4.12 a 4.15 se muestra el circuito esquemático del Manejador de Memorias ZBT. El diseño es principalmente un multiplexor entre los dos canales de la entidad, permitiendo que el canal LBC o el canal CORE tenga acceso a las conexiones de la memoria. Si el puerto LBC realiza un acceso, una máquina de estados se encarga de generar los ciclos de espera y los retrasos de señales correspondientes, incluido un ciclo de espera para el puerto CORE, ciclo durante el cual el LBC está haciendo su acceso a la memoria. El manejador se encarga de generar las señales de control apropiadas cuando el LBC realiza su acceso, pero las señales de control del puerto CORE son pasadas directamente, ya que si CORE es un diseño segmentado, el desfase de las señales puede añadirse directamente (y de una manera mas eficiente) en la estructura del pipeline. La figura 4.16 muestra la maquina de estados del manejador.

Si bien el diseño es funcional y ha sido probado con frecuencias de reloj de 33MHz, este diseño presenta varias desventajas. En primer lugar, no opera correctamente si el reloj de las memorias ZBT no es síncrono con el reloj del bus local. En segundo lugar, no permite transferencias de ráfaga entre el LBC y las memorias, que podría realizarse si ambos estuviesen más integrados y no se considerara cada acceso como independiente. En última instancia, el diseño no es muy práctico al momento de interconectarse con el CORE. Probablemente una nueva implementación que integre el LBC y los manejadores de las memorias ZBT proveería un mejor diseño, mas compacto y eficiente.

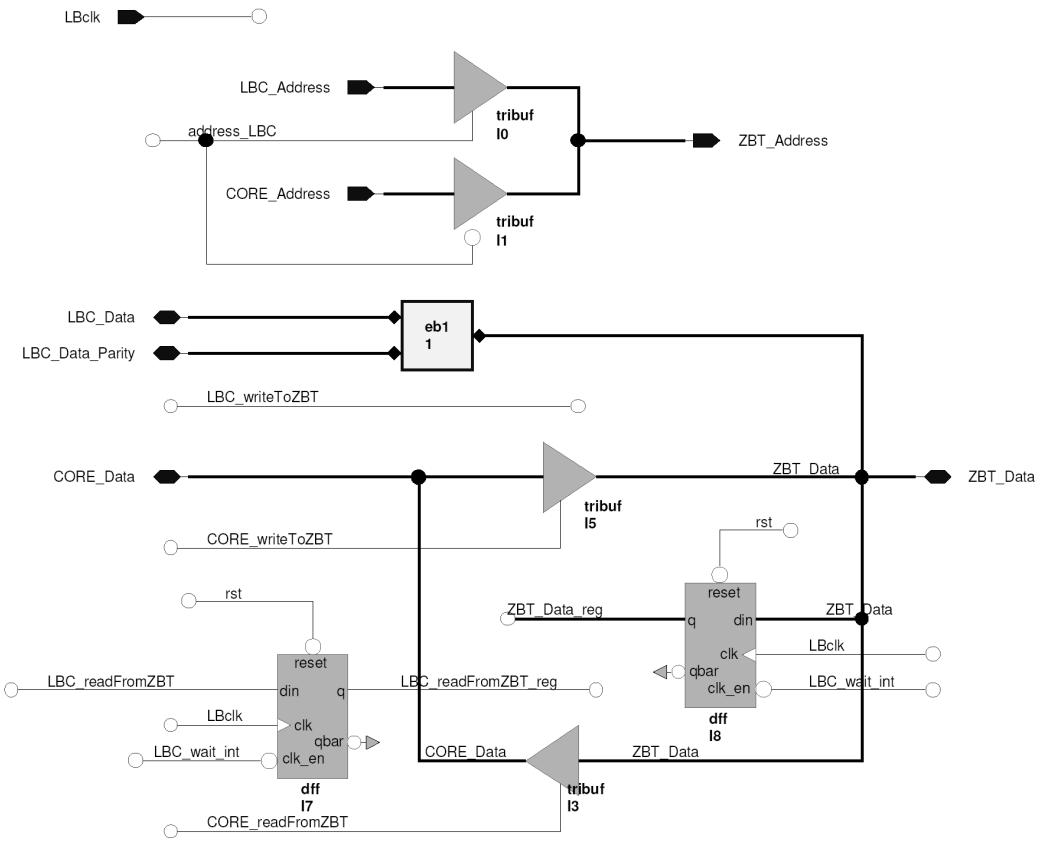

Figura 4.12: Entidad ZBT Manager, parte 1

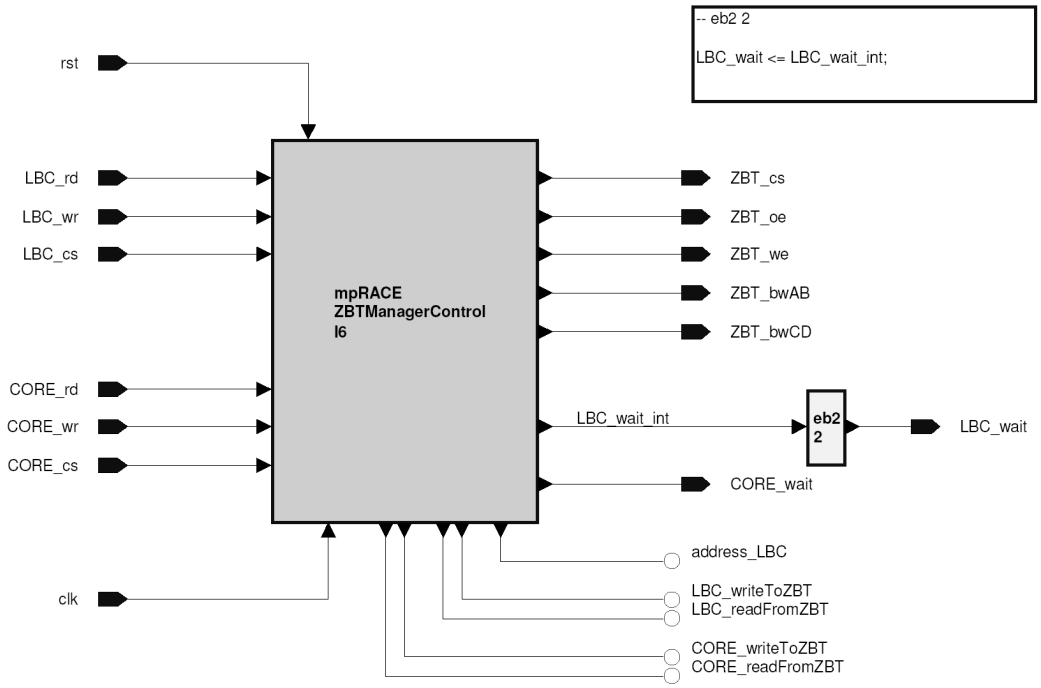

Figura 4.13: Entidad ZBT Manager, parte 2

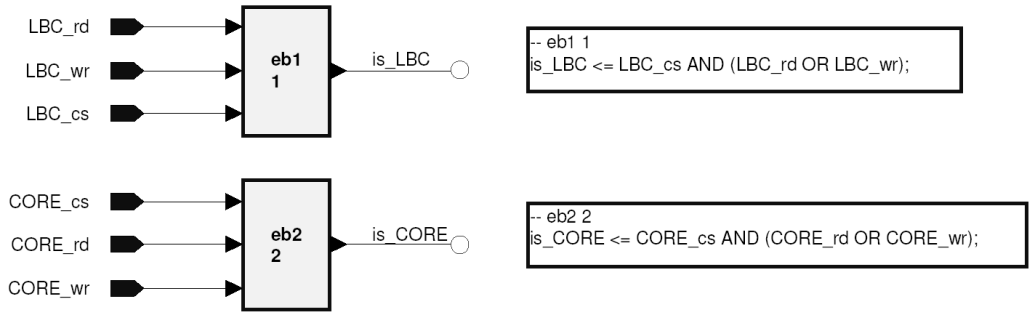

Figura 4.14: Bloque de Control del ZBT Manager, parte 1

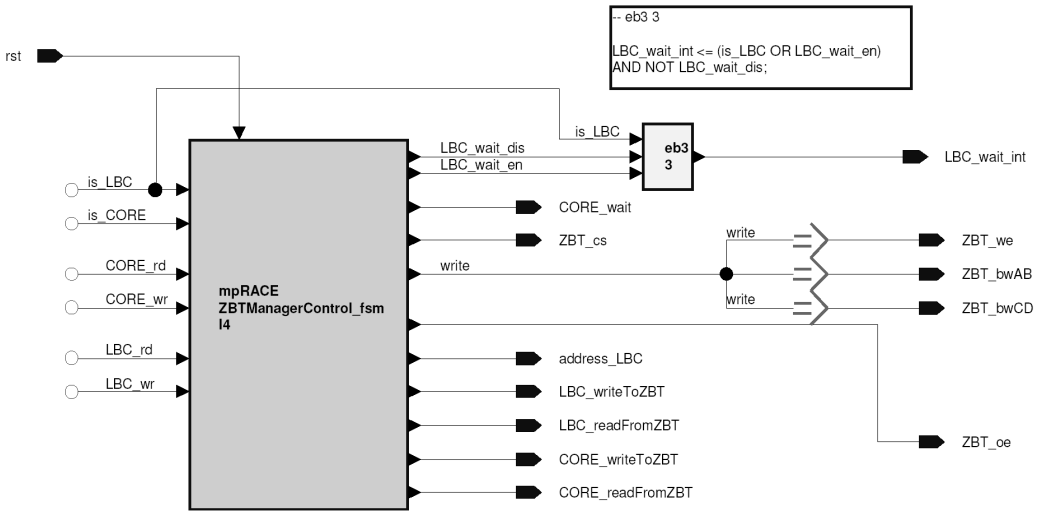

Figura 4.15: Bloque de Control del ZBT Manager, parte 2

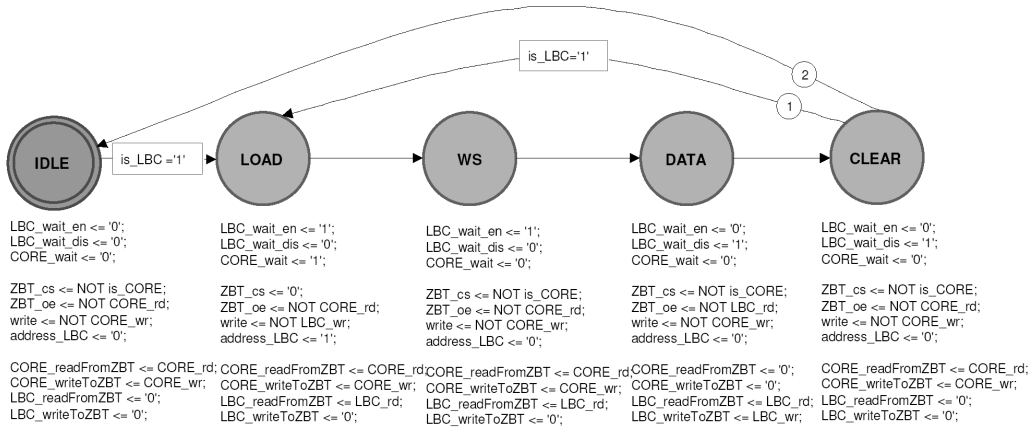

Figura 4.16: Maquina de Estados del FSM

4.2.3. Resultados

En esta sección se muestran los diagramas de tiempo del Manejador de Memorias ZBT. Se muestran ciclos de escritura, lectura y escritura en ráfaga para el modelo de comportamiento (figuras 4.17, 4.18 y 4.19, y luego para el modelo ya sintetizado operando a 33MHz (figuras 4.20, 4.21 y 4.22). Es de notar que en estos diagramas no se simula el comportamiento de la memoria ZBT, solo el comportamiento del controlador. Los diagramas están divididos en 3 secciones, agrupando señales correspondientes al LBC, señales correspondientes a la memoria ZBT, y señales internas del manejador de ZBT, que permiten entre otras cosas, ver el estado actual de la maquina de estados.

En la figura 4.17 puede verse un par de escrituras desde el LBC hacia la memoria ZBT. Es importante observar dos aspectos principales. Primero, el funcionamiento de la señal *Dwait* del LBC, y segundo, el desfase entre la dirección y los datos que el manejador de memoria inserta entre la dirección y los datos del bus interno del LBC.

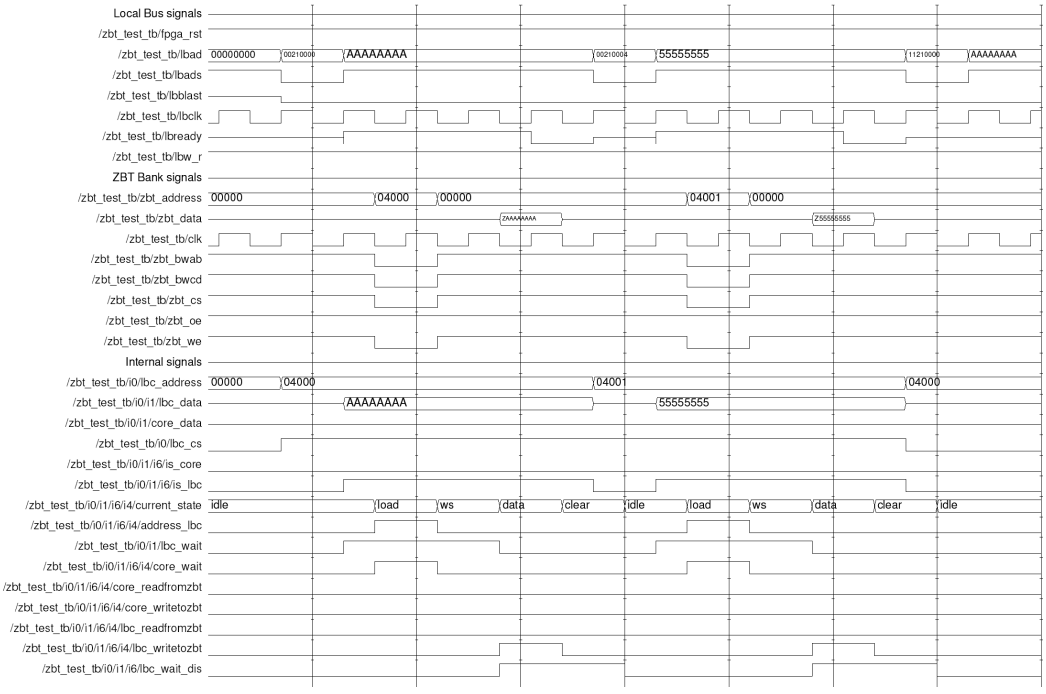

Figura 4.17: Escritura del LBC en un banco de memoria ZBT

En la figura 4.18 pueden verse dos lecturas desde el LBC de la memoria ZBT. La difer-

encia con respecto a la figura 4.17 radica en la señal de control de dirección de la memoria (textitzbt_oe), que en este diagrama se ve más claro que está retrasada y que va con el dato. Ademas, puede verse el retraso de medio ciclo de la transferencia del dato desde la ZBT hacia el bus interno del LBC.

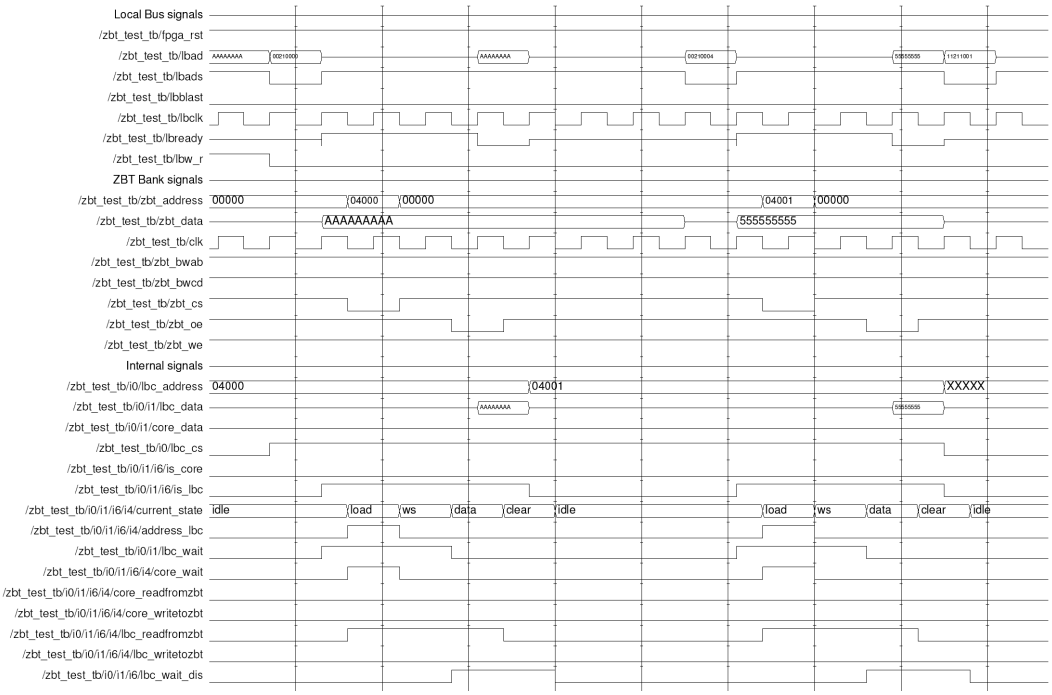

Figura 4.18: Lectura del LBC de un banco de memoria ZBT

En la figura 4.19 puede verse una escritura en ráfaga a la memoria ZBT. Como puede observarse, el manejador hace un uso mucho menos que eficiente de las capacidades de la memoria ya que inserta 3 retrasos por cada dato a escribir. La dirección en este caso es generada por el contador de direcciones del LBC. Si se integraran el LBC y el manejador de ZBT, estas transferencias pueden ahcerse mucho más rápidamente, ya que sólo existiría retraso en la primera transacción, y las demás serían a la velocidad completa de la ZBT.

En las siguientes tres figuras, 4.20, 4.21 y 4.22, se muestran las mismas pruebas de las figuras 4.17, 4.18 y 4.19 utilizando el diseño sintetizado con una frecuencia de reloj de 33MHz. Nótense los retrasos de las señales, donde la mayoria de las señales en los diagramas ideales cambian con flancos de bajada (para las señales de la ZBT), y en el caso sintetizado ocurren un tiempo despues, algunas justo antes del flanco de subida. Esto es precisamente lo

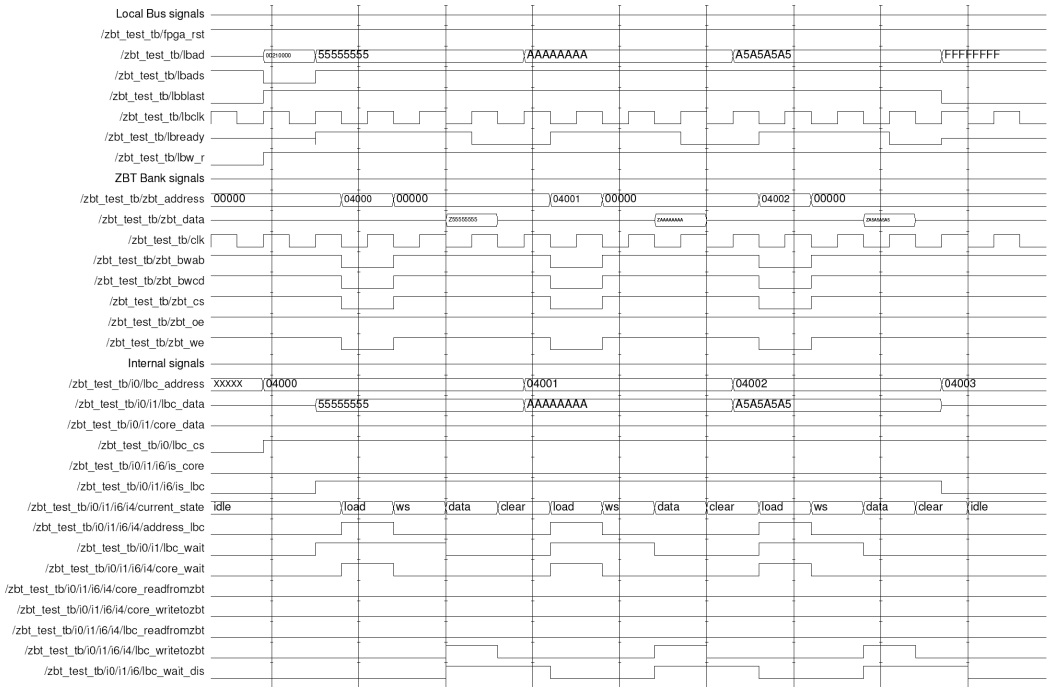

Figura 4.19: Escritura del LBC en ráfaga en un banco de memoria ZBT

que se desea, se hacen los cambios en el flanco de bajada para que las señales estén estables en el flanco de subida, que es cuando la memoria ZBT registra las señales.

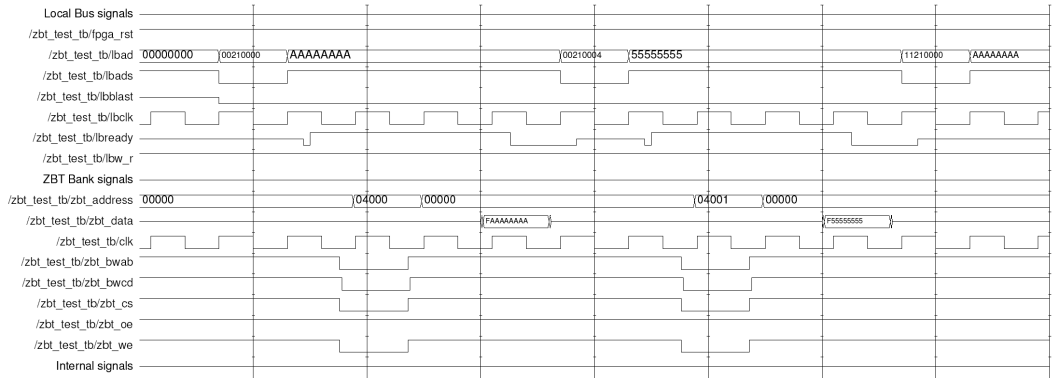

Figura 4.20: Escritura del LBC en un banco de memoria ZBT, Sintetizado a 33MHz

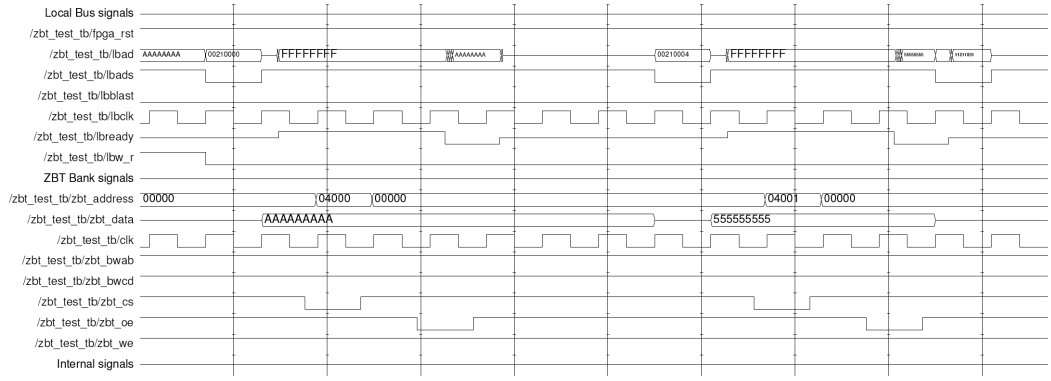

Figura 4.21: Lectura del LBC de un banco de memoria ZBT, Sintetizado a 33MHz

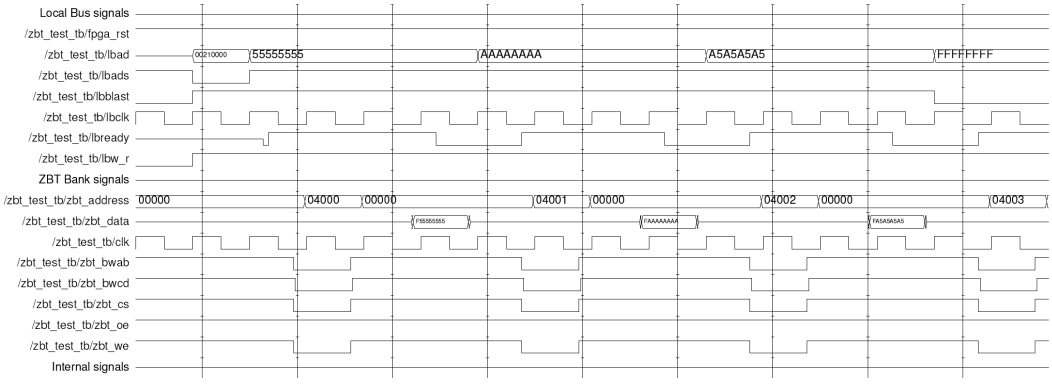

Figura 4.22: Escritura del LBC en ráfaga en un banco de memoria ZBT, Sintetizado a 33 MHz

Capítulo 5

Unidades de Punto Flotante

5.1. Multiplicador de Punto Flotante

La unidad del multiplicador de punto flotante implementada opera sobre dos números de precisión sencilla en representación IEEE754 y genera un resultado de precisión sencilla también en representación IEEE754. Opera correctamente sobre valores normalizados, denormalizados y valores especiales (NaN, INF, Zero). Realiza un solo tipo redondeo, a saber, redondeo al número más próximo, y no genera banderas de excepciones, aunque son tomadas en cuenta para las operaciones.

En primera instancia se explicará la implementación de ciclo único (no segmentada), que permite ver de una manera mas clara las diferentes partes que componen al multiplicador. En la sección siguiente se presenta la implementación segmentada y cada una de las etapas. Por último se explica cada una de las partes que componen al multiplicador.

5.1.1. Implementación de Ciclo Único

En la figura 5.1 se muestra el diagrama esquemático del multiplicador. Este diseño no depende del reloj, por lo que es en principio un gran circuito combinatorio. Se pueden distinguir 7 secciones principales, a saber: el desempaquetado de los números, el cálculo del signo, el cálculo del exponente, el cálculo del significando, los casos especiales, el ajuste y el empaquetado del resultado.

A grandes rasgos, se separan los operandos en sus diferentes partes, las que se calculan

por separado (el signo, el exponente y la mantissa). Una vez calculados, se procede a realizar los ajustes necesarios para su correcta representación, modificando tentativamente el significando y el exponente resultante. Este ajuste se traduce en una normalización, un redondeo y una nueva normalización, tal como se describió en 2.3.3 y como se detalla más adelante. Por último, se empaqueta de nuevo el resultado. Los casos especiales son manejados aparte mediante una tabla.

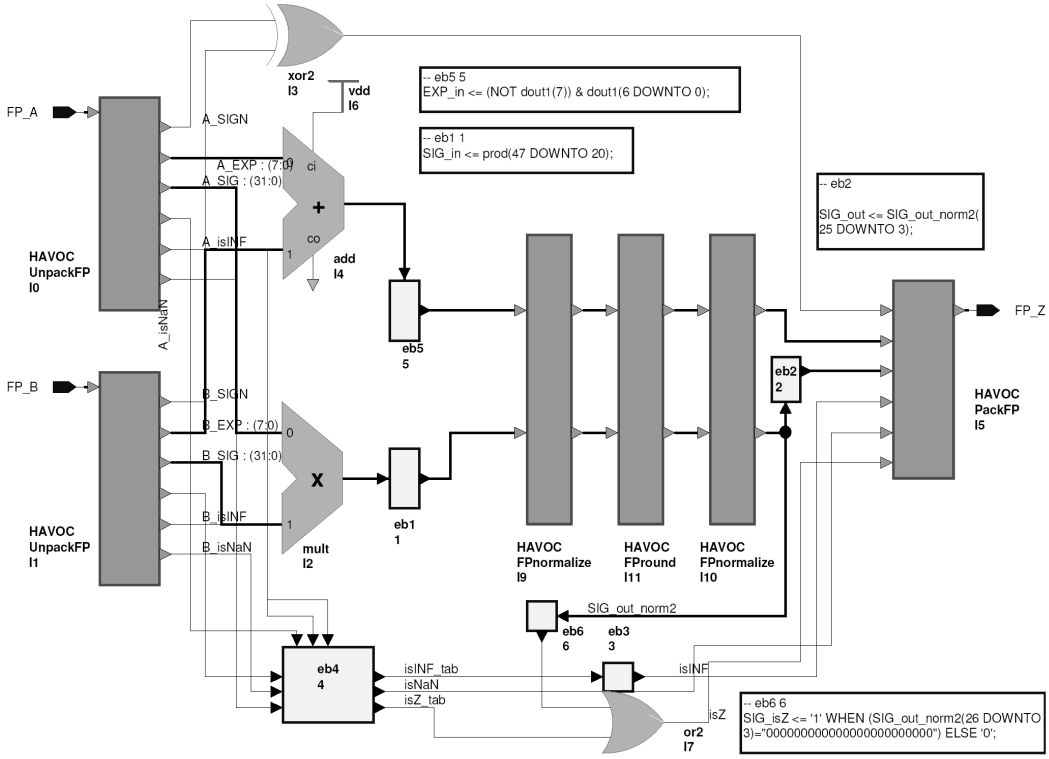

Figura 5.1: FPmul, implementación de ciclo único

5.1.2. Implementación por Etapas

Para migrar la implementación de ciclo único a un diseño segmentado, se dividió el circuito en etapas o bloques que balancearan aproximadamente el tiempo de cómputo de cada etapa. Registros a la salida de cada etapa controlan el flujo de datos en cada flanco de subida del reloj. El circuito se dividió en 4 etapas, donde la primera etapa (figura 5.3) consiste del desempaquetado, el cálculo del signo y la identificación de excepciones, la segunda etapa

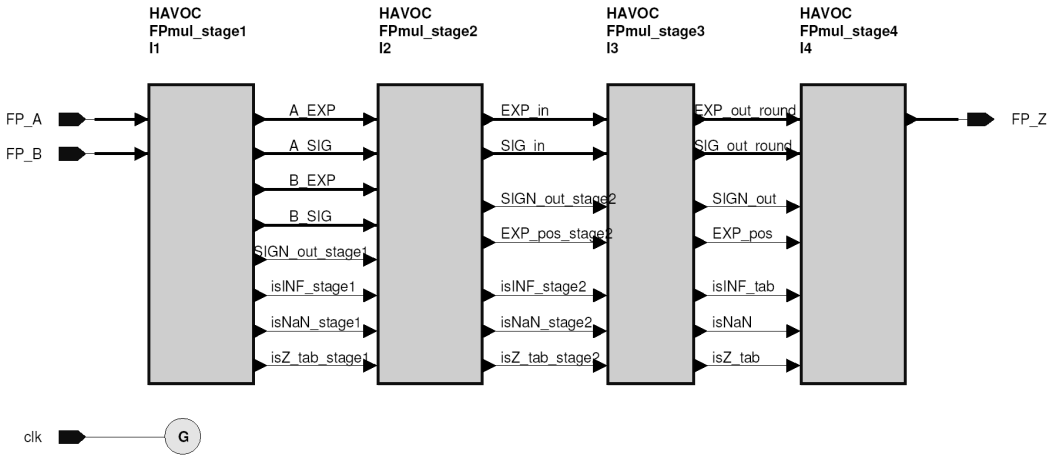

Figura 5.2: Etapas del FPmul segmentado

(figura 5.4) consiste del cálculo del exponente y la mantissa, la tercera etapa (figura 5.5) abarca la primera normalización y el redondeo, y la cuarta y última etapa (figura 5.6) consiste de la última normalización, la detección de condiciones finales de excepción y el empaquetado del resultado.

5.1.3. Cálculo del Signo Resultante

Dado dos números, el signo del producto de ellos será positivo si el signo de los operandos es el mismo, o negativo si son diferentes. Esto corresponde a la lógica de una compuerta XOR, por lo que el signo del resultado se determina fácilmente.

5.1.4. Cálculo del Exponente

Dado los exponentes de los operandos, el exponente del resultado es la suma de los exponentes. Como en la representación IEEE754 los exponentes tienen un desplazamiento de 127, hacer el cálculo implicaría restar 127 a cada exponente, realizar la operación de suma y volver a sumar el desplazamiento para obtener el exponente adecuado. Una manera fácil de simplificar este cálculo es sumar uno a la operación y restarle 128 al resultado. La unidad se puede añadir como un carry de entrada al sumador, y la resta de 128 consiste en negar el bit más significativo del resultado.

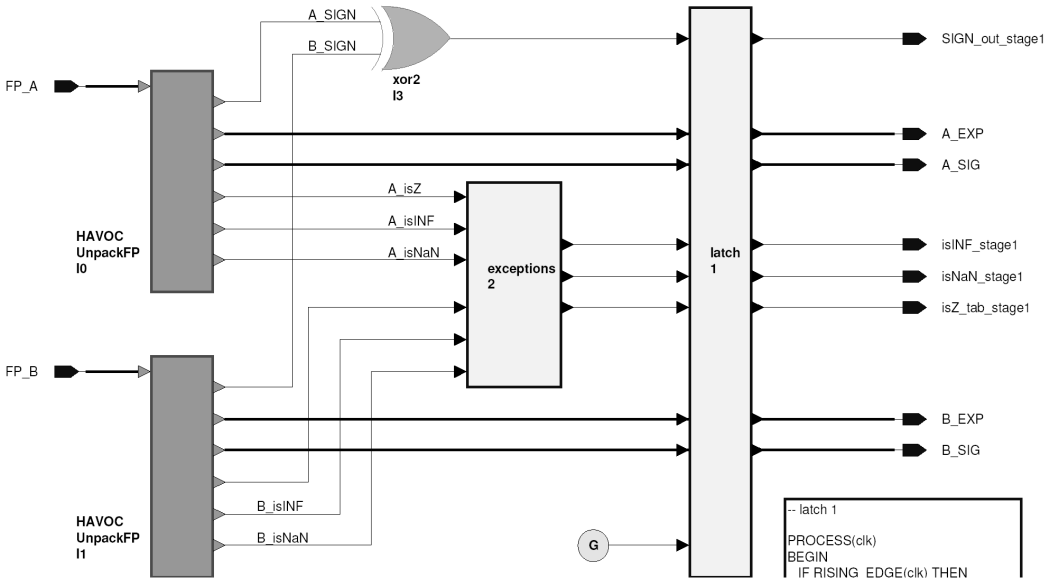

Figura 5.3: Etapa 1 del FPmul segmentado

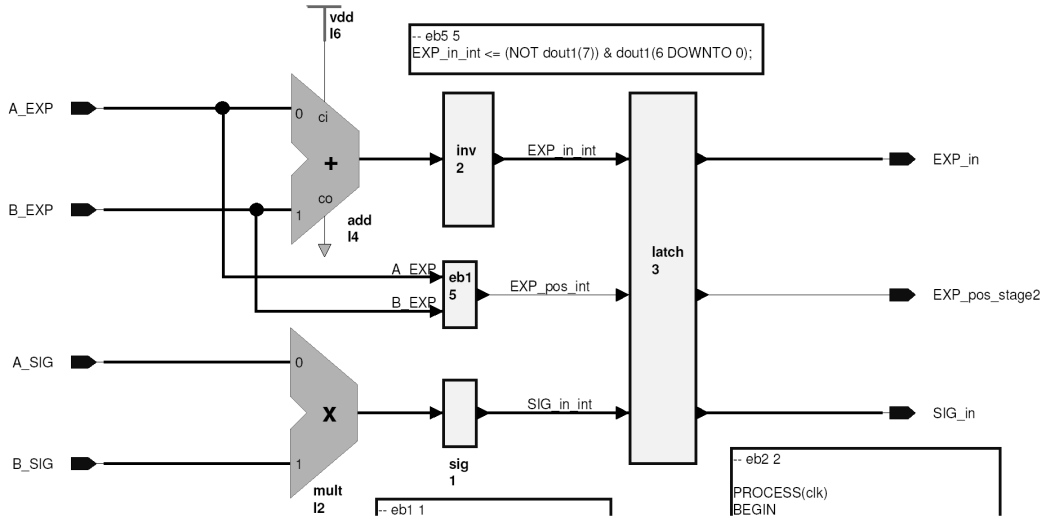

Figura 5.4: Etapa 2 del FPmul segmentado

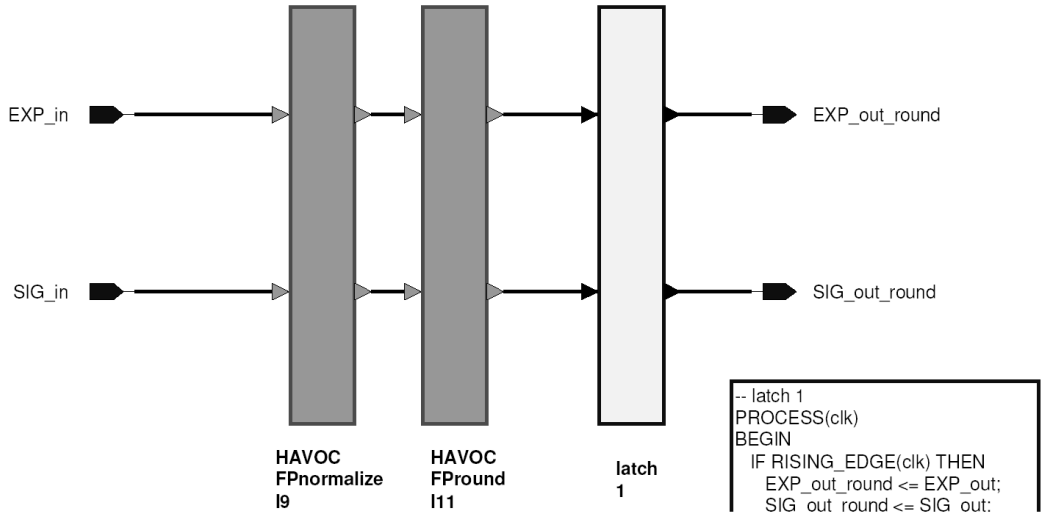

Figura 5.5: Etapa 3 del FPmul segmentado

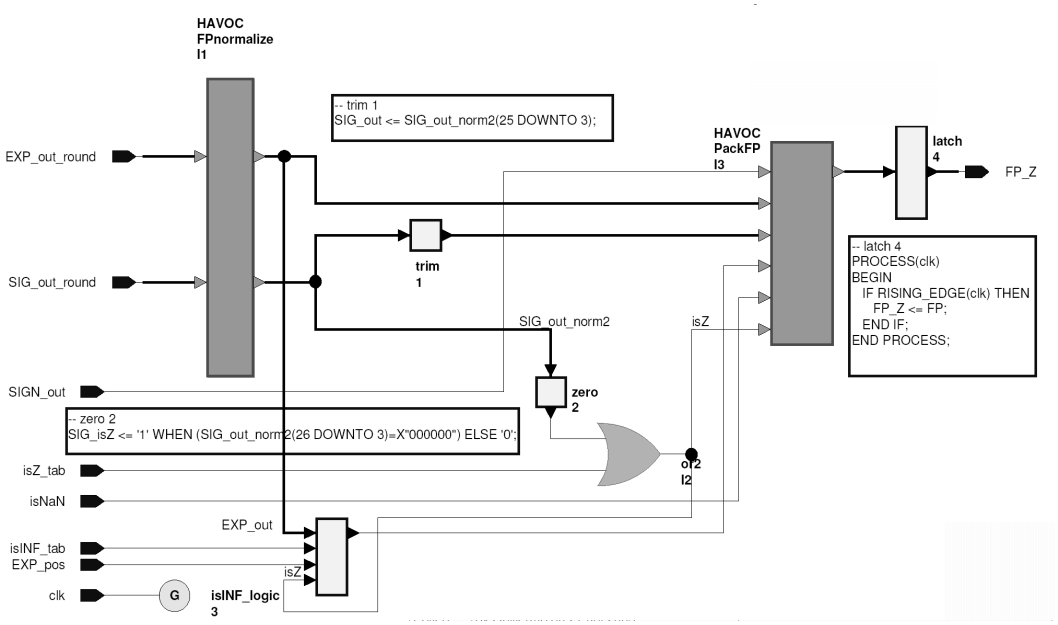

Figura 5.6: Etapa 4 del FPmul segmentado

5.1.5. Cálculo de la Mantissa

El significando del resultado requiere la multiplicación de los significandos de los operandos, y realizar ajustes posteriores sobre el resultado y el exponente para adecuarlos a la representación IEEE754. En principio, consiste en un multiplicador de enteros. Dada la representación en base fraccional del significando, hay que recordar que se toman los bits más significativos del resultado, no los menos.

5.1.6. Unpacking

Este bloque recibe un vector de 32 bits que contiene un número de punto flotante en representación IEEE754 de precisión sencilla, y lo separa en signo, exponente, mantissa, y detecta números especiales: Cero, Infinito (INF) y No-Es-Un-Numero (NaN). Esta operación se realiza para cada uno de los operandos de entrada.

5.1.7. Normalización

Dado que los significandos representan números en el rango $[1, 2)$, el resultado de la multiplicación se encontrará en el rango $[1, 4)$. Por lo tanto, puede ser necesario ajustar (normalizar) el resultado rotándolo una posición a la derecha e incrementando el exponente. Esta etapa realiza este ajuste.

5.1.8. Redondeo

Dados los bits extras de precisión que se producen al momento de la multiplicación de los significandos, el resultado dependerá del esquema de redondeo elegido para producir el significando correcto del resultado. Por simplicidad, esta unidad de multiplicación implementa un solo esquema de redondeo, el de redondeo al número mas cercano. Para ello, basta utilizar 2 bits extras de la multiplicación para determinar si el resultado se aproxima hacia arriba o hacia abajo; 1 bit para saber si el resto es mayor o menor que la mitad, y otro de guardia en caso de que en la etapa anterior se realizara un corrimiento. Este redondeo tentativamente le suma una unidad al significando del resultado, por lo que existe la posibilidad de

que el resultado desborde nuevamente, por lo que es necesario otro paso de Normalización para asegurar que el significando se encuentra en el rango correcto de $[1, 2)$.

5.1.9. Packing

En este bloque se toman el signo, el exponente y el significando del resultado y se produce un arreglo de 32 bits que es el número de punto flotante de precisión sencilla en representación IEEE754. Si se activó alguna de las banderas de Zero, INF o NaN, este bloque ignora los valores de entrada y produce la salida acorde.

5.1.10. Resultados

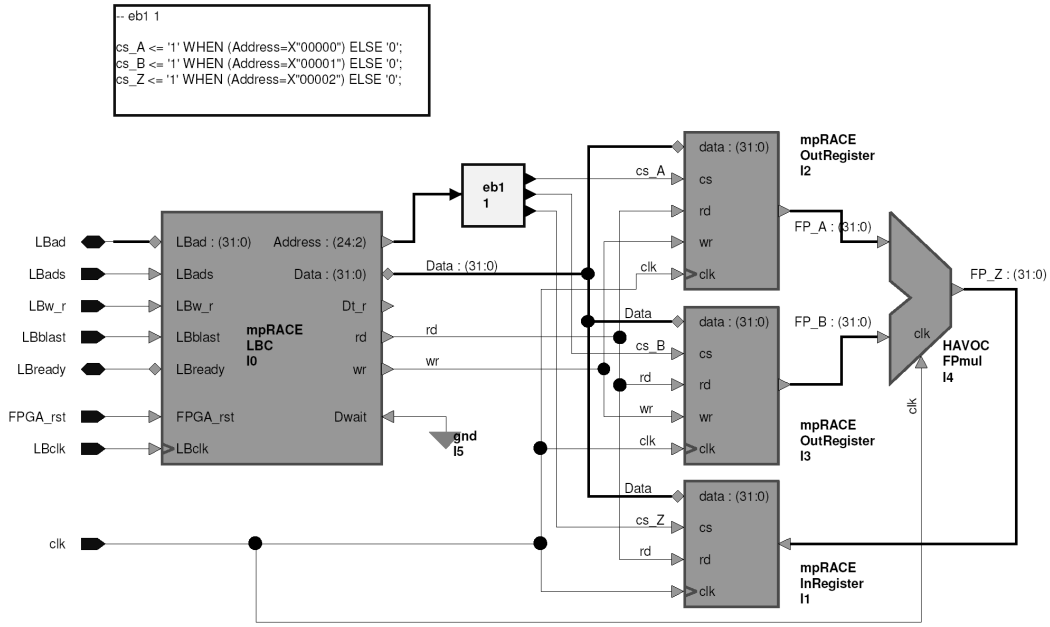

Figura 5.7: Diseño de prueba del FPmul

En esta sección se muestran las pruebas realizadas sobre el multiplicador de punto flotante. El diseño de prueba implementado puede verse en la figura 5.7. A continuación se muestra el resultado de un programa de prueba, que realiza operaciones sobre el diseño implementado en la tarjeta funcionando a 33MHz y las compara con el mismo cálculo realizado por la PC (cuyo valor se muestra entre paréntesis).

El listado presenta los dos número a multiplicar, el resultado de la ejecución en HAVOC en decimal y en hexadecimal, y a continuación el resultado de la ejecución de la misma operación con el procesador de la PC, mostrado también en decimal y hexadecimal. Como puede observarse, todos los resultados coinciden, por lo que no se detectaron anormalidades.

Las primeras operaciones de esta secuencia permiten probar el funcionamiento con valores y condiciones especiales, como INF, Cero y Overflow. Luego se prueban valores normales, y valores que produzcan pérdida de precisión. Una prueba más exhaustiva requeriría el uso de una suite como *SoftFloat*, que establece patrones específicos y detallados de prueba de unidades de punto flotante.

```
1.#INF       * 1.#INF       = 1.#INF 7f800000 (1.#INF) 7f800000
3.40282e+038 * 3.40282e+038 = 1.#INF 7f800000 (1.#INF) 7f800000
3.40282e+038 * 2            = 1.#INF 7f800000 (1.#INF) 7f800000
-3.40282e+038 * 1.1         = -1.#INF ff800000 (-1.#INF) ff800000
0            * 0            = 0 0             (0) 0
0            * 0.5          = 0 0             (0) 0
55           * 1            = 55 425c0000     (55) 425c0000
0.5          * 0.5          = 0.25 3e800000   (0.25) 3e800000
1.01         * 2.02         = 2.0402 400292a3 (2.0402) 400292a3
11           * 5            = 55 425c0000     (55) 425c0000
1e+013       * 5e+011       = 5e+024 68845951 (5e+024) 68845951
1.01         * 22340        = 22563.4 46b046cd (22563.4) 46b046cd
-1.01        * 2.02         = -2.0402 c00292a3 (-2.0402) c00292a3
1.01         * -2.02        = -2.0402 c00292a3 (-2.0402) c00292a3
-15.34       * 8.5          = -130.39 c30263d7 (-130.39) c30263d7
1.00001      * 2            = 2.00002 4000005c (2.00002) 4000005c
1.01         * 2.0002       = 2.0202 40014afd (2.0202) 40014afd
```

En el listado C.1 de los anexos se puede observar los resultados de la síntesis del diseño de prueba. El diseño ocupa 157 Slices, 4 multiplicadores en hardware, y puede operar a una frecuencia máxima de 50.9 MHz. Puede observarse cuantos componentes en detalle utiliza cada una de las etapas del multiplicador, y que éstas se encuentran bien balanceadas, ya que cada una ocupa aproximadamente la misma área. Es de notar que el speedup producto de la segmentación es mayor que n, tomando en cuenta que la frecuencia máxima de operación del circuito de ciclo único es 8 MHz, ya que sp $= 50/8 = 6{,}25 > 4$. esto es debido a que el sintetizador y el P&R pueden optimizar más eficientemente los circuitos de cada etapa por separado que el circuito combinatorio completo.

Por último, en el anexo C.2 pueden verse los diagramas de tiempo correspondientes para la ejecución de este patrón de pruebas en el multiplicador de ciclo único y en el multiplicador segmentado, utilizando el modelo de comportamiento. El modelo sintetizado fué probado con el programa de prueba *FPmul_test.cxx*, incluido en el CDROM.

5.2. Sumador de Punto Flotante

La unidad del sumador de punto flotante implementada opera sobre dos números de precisión sencilla en representación IEEE754 y genera un resultado de precisión sencilla también en representación IEEE754. Opera correctamente sobre valores normalizados, denormalizados y valores especiales (NaN, INF, Zero), con la excepción de un error de redondeo en algunas situaciones, y que se encuentra bajo estudio. Realiza un solo tipo redondeo, a saber, redondeo al número más próximo, y no genera banderas de excepciones, aunque son tomadas en cuenta para las operaciones.

En primera instancia se explicará la implementación de ciclo único (no segmentada), que permite ver de una manera mas clara las diferentes partes que componen al sumador. En la sección siguiente se presenta la implementación segmentada y cada una de las etapas. Por último se explica cada una de las partes que componen al sumador.

5.2.1. Implementación de Ciclo Único

En la figura 5.8 se muestra el diagrama esquemático del multiplicador. Este diseño no depende del reloj, por lo que es en principio un gran circuito combinatorio. Se pueden distinguir 7 secciones principales, a saber: el desempaquetado de los números, el cálculo del signo, el cálculo del exponente, el cálculo del significando, los casos especiales, el ajuste y el empaquetado del resultado.

A grandes rasgos, tal como se describió en 2.3.3 y como se detalla más adelante, se separan los operandos en sus diferentes partes, y se procede a alinear los operandos para luego calcular el signo, el exponente y la mantissa por separado. Una vez calculados, se procede a realizar los ajustes necesarios para su correcta representación, modificando tentativamente el significando y el exponente resultante. Este ajuste corresponde en una normalización, un redondeo y una nueva normalización. Por último, se empaqueta de nuevo el resultado. Los casos especiales son manejados aparte mediante una tabla. Como se permiten operaciones de suma y de resta, durante el proceso de alineación se realizan ajustes para complementar los operadores cuando sea necesario, ajuste que es de todas maneras necesario cuando uno o

ambos de los operadores son negativos.

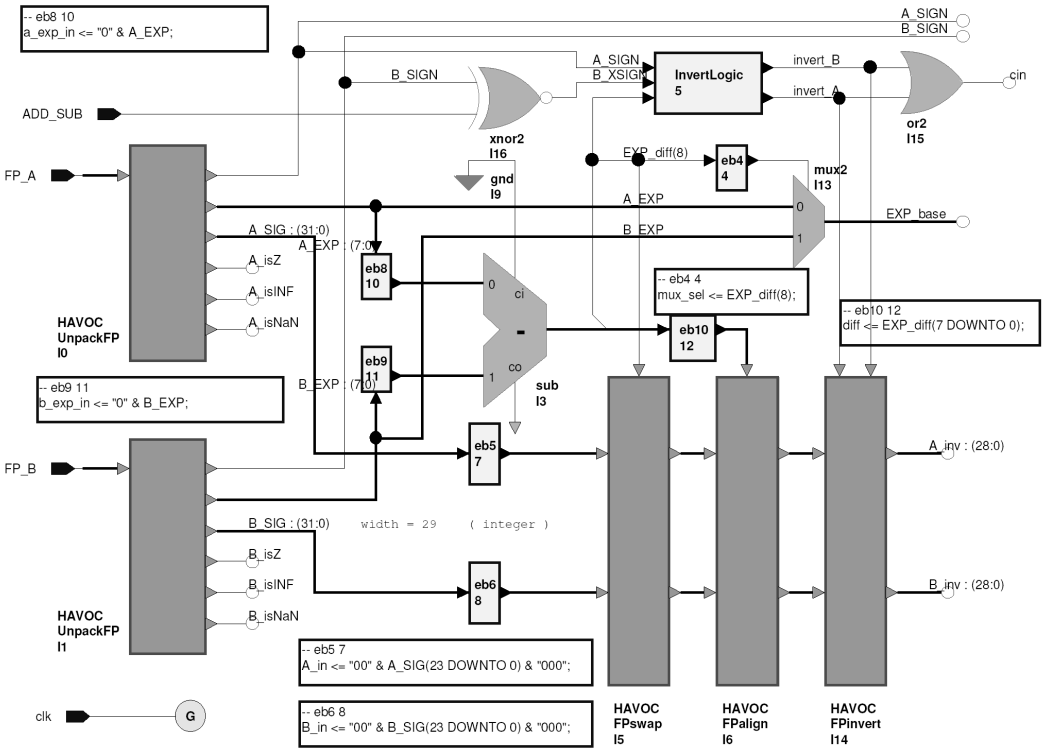

Figura 5.8: FPadd en ciclo único, parte 1

5.2.2. Implementación por Etapas

Para migrar la implementación de ciclo único a un diseño segmentado, se dividió el circuito en etapas o bloques que balancearan aproximadamente el tiempo de cómputo de cada etapa. Registros a la salida de cada etapa controlan el flujo de datos en cada flanco de subida del reloj. El circuito se dividió en 6 etapas,tal como se muestra en la figura 5.10. La primera etapa (figura 5.11) consiste del desempaquetado, el cálculo de la diferencia de los exponentes y el ajuste del signo del operando B con el tipo de operación a realizar (suma/resta); la segunda etapa (figura 5.12) consiste de la selección del exponente, de la alineación de los significandos, de la lógica de inversión de los operandos y de la tabla de manejo de excepciones; la tercera etapa (figura 5.13) abarca la inversión de los operandos y de la suma de los significandos; la cuarta etapa (figura 5.14) consta de la lógica del signo del resultado, la com-

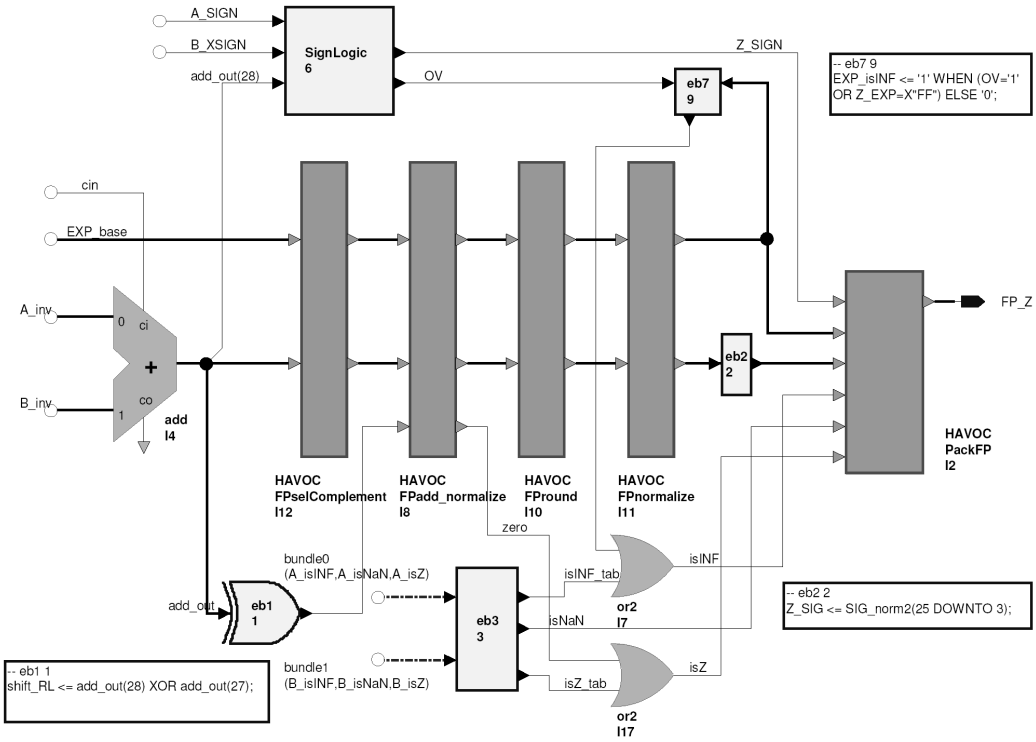

Figura 5.9: FPadd en ciclo único, parte 2

plementación selectiva del resultado (en caso que sea negativo) y la primera normalización; la quinta etapa (figura 5.15) consta del redondeo y la última normalización; y la sexta y última etapa (figura 5.16) consiste del empaquetado y lógica de soporte para las condiciones especiales.

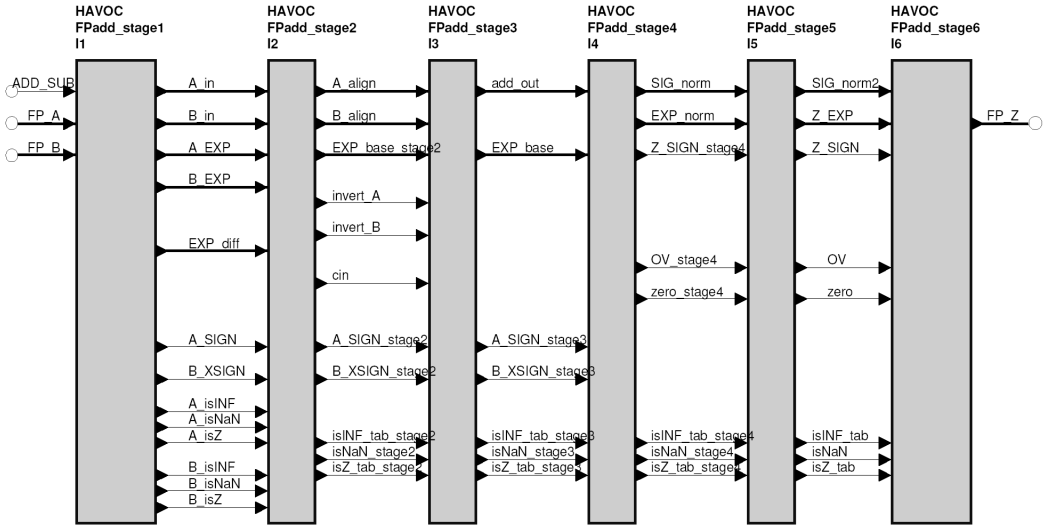

Figura 5.10: FPadd segmentado

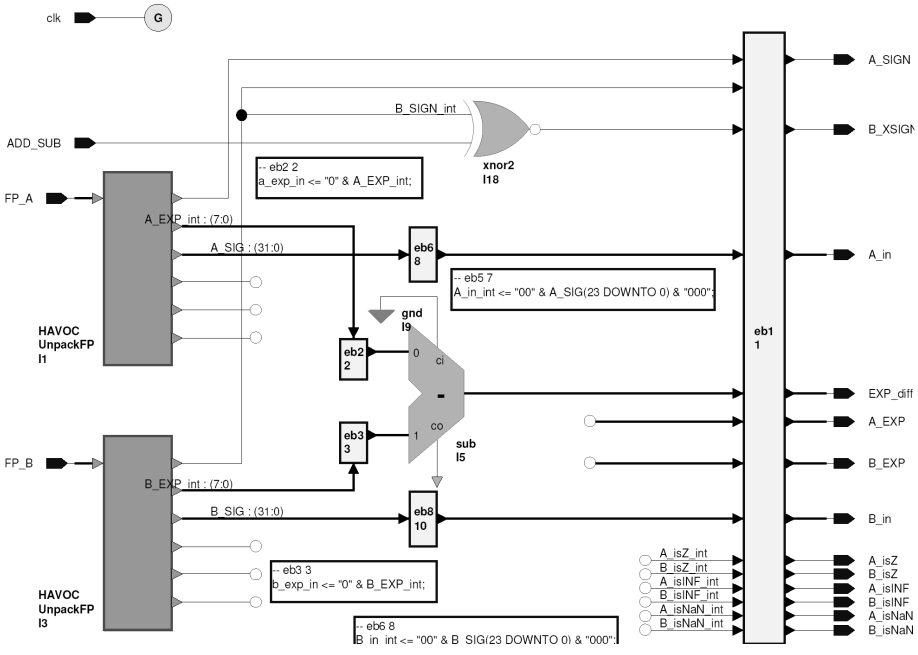

Figura 5.11: FPadd segmentado, Etapa 1

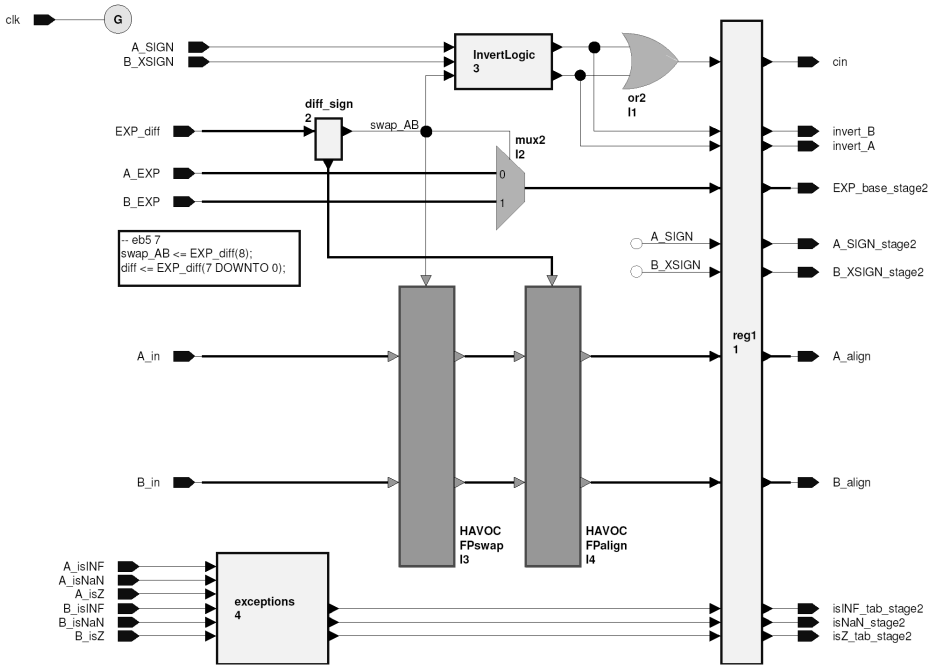

Figura 5.12: FPadd segmentado, Etapa 2

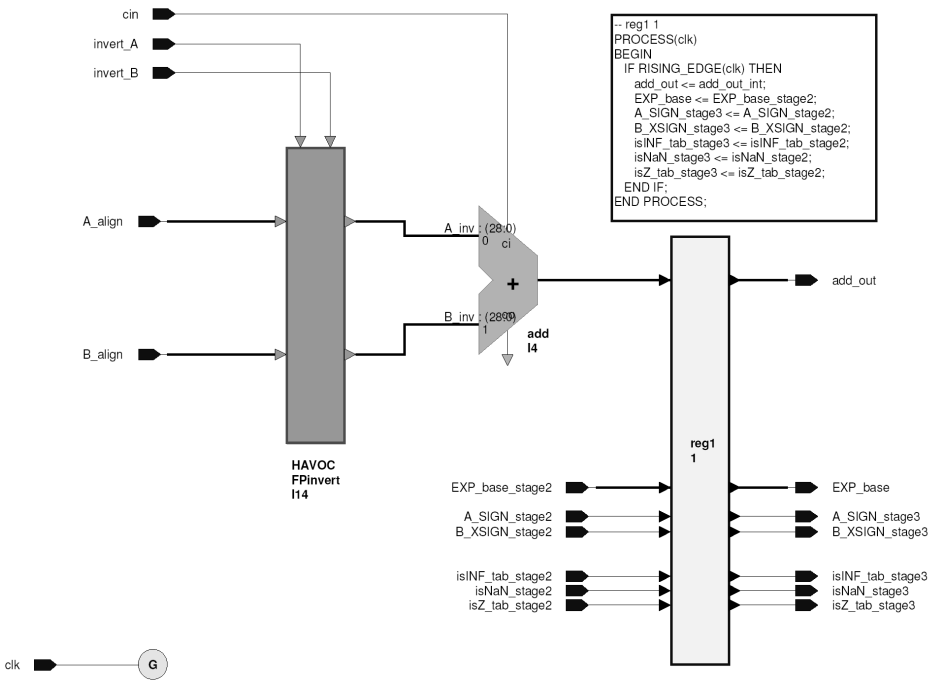

Figura 5.13: FPadd segmentado, Etapa 3

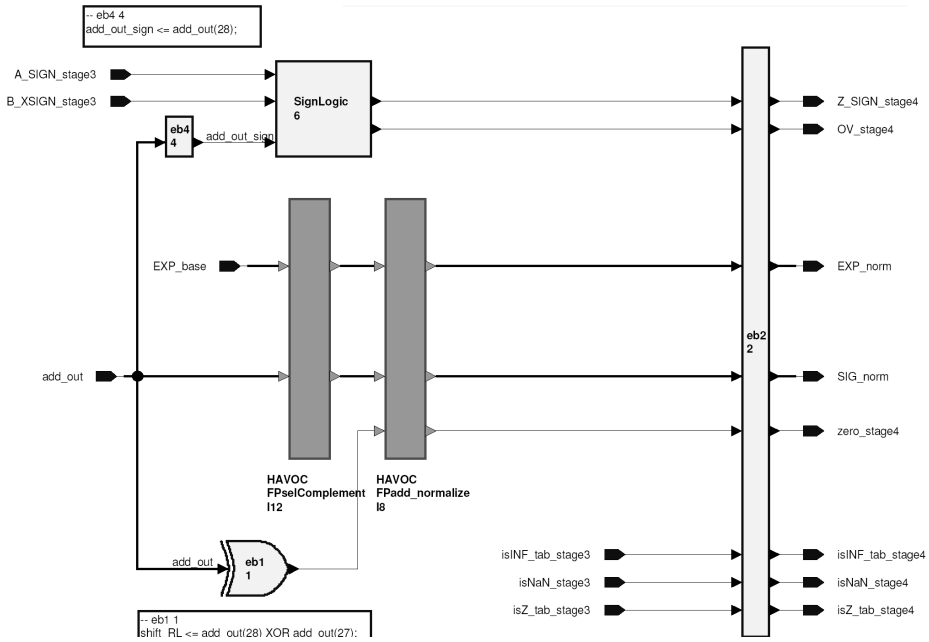

Figura 5.14: FPadd segmentado, Etapa 4

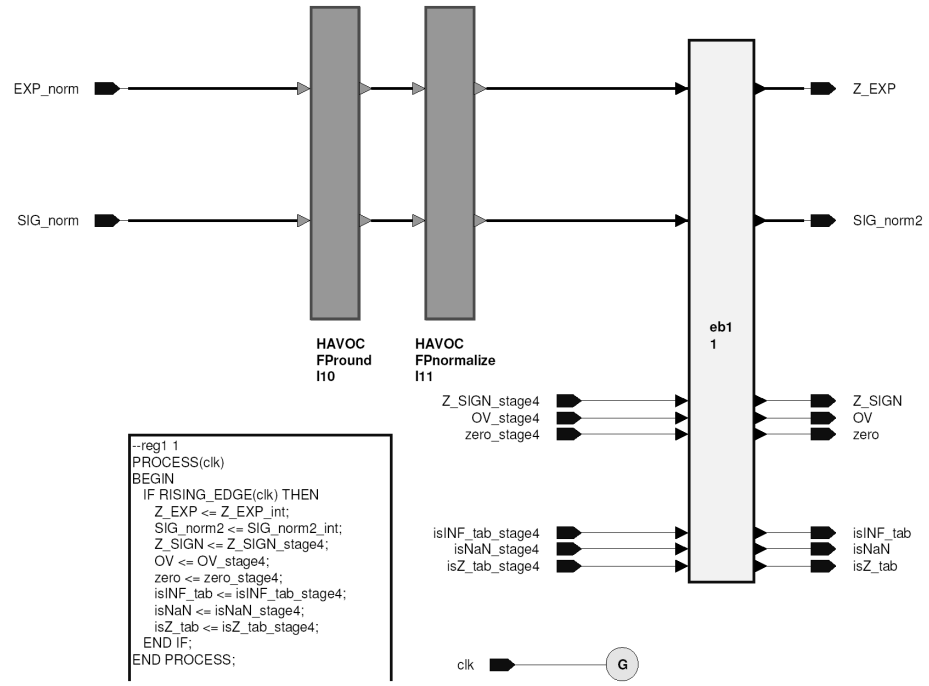

Figura 5.15: FPadd segmentado, Etapa 5

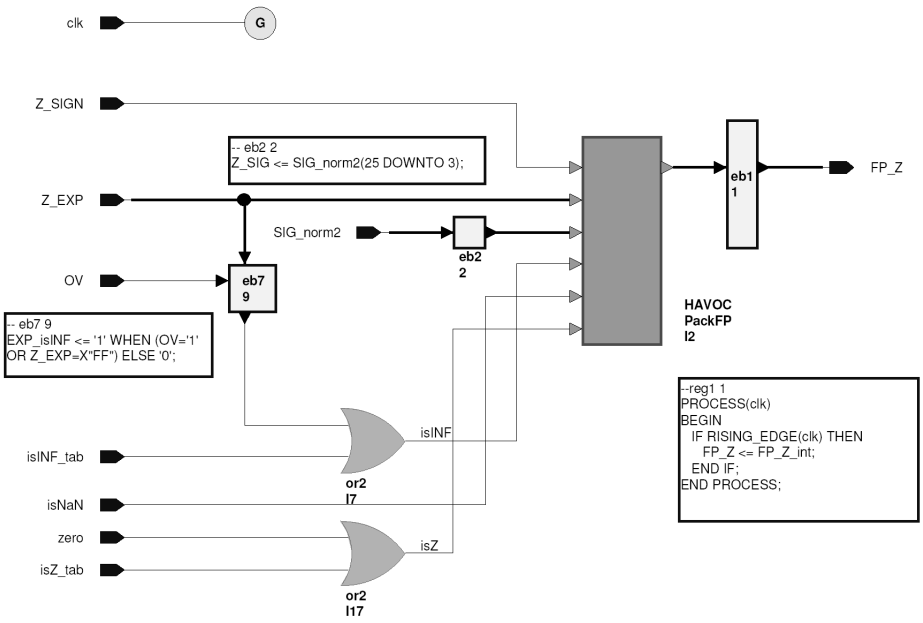

Figura 5.16: FPadd segmentado, Etapa 6

5.2.3. Cálculo del Signo Resultante

El signo del resultado depende de los signos de los operandos, de la operación a realizar, y del signo del resultado de la suma de los significandos. El signo de la operación y del operador pueden integrarse en uno solo mediante una compuerta XOR. Estos signos se usan para direccionar una tabla que produce la bandera de Overflow y el signo resultado. La tabla se muestra en la figura 5.17

	A	B	C	D	E
1	A_SIGN_stage3	B_XSIGN_stage3	add_out_sign	OV	Z_SIGN
2	'0'	'0'	'0'	'0'	'0'
3	'0'	'0'	'1'	'1'	'0'
4	'0'	'1'	'0'	'0'	'0'
5	'0'	'1'	'1'	'0'	'1'
6	'1'	'0'	'0'	'0'	'0'
7	'1'	'0'	'1'	'0'	'1'
8	'1'	'1'	'0'	'0'	'1'
9	'1'	'1'	'1'	'1'	'1'
10				'0'	'0'

Figura 5.17: Tabla de Resolución de Signos

5.2.4. Cálculo del Exponente

El exponente base a utilziar para los cálculos es el mayor exponente de los dos operandos, y se determina mediante un multiplexor controlado por el signo de la diferencia de ambos (valor que será usado para alinear los significandos). El exponente puede modificarse luego de la suma dependiendo del proceso de ajuste del resultado.

5.2.5. Cálculo de la Mantissa

La mayor complejidad del sumador se encuentra en la manipulación de los significandos. Antes de realizar la suma de los operandos, es necesario alinearlos y puede ser necesario complementarlos. El proceso de alinearlos es costoso en términos de recursos, y dado que solo es necesario alinear uno de los operandos, sólo se implementa un bloque de alineación y se añade un bloque que permite intercambiarlos de lugar, para alinear el correcto. Si uno de los operandos es negativo, es necesario complementarlo para realizar la suma en complemento a dos de manera correcta.

Una vez sumados los significandos, si el resultado es negativo es necesario complementarlo, ya que en el formato IEEE754 el significado es siempre positivo. A continuación se normaliza el resultado, ya que se encuentra en el rango $[0, 4)$, se redondea y se vuelve a normalizar, en caso de que el redondeo lo sacara una vez más fuera del rango de representación.

5.2.6. Unpacking

Este bloque recibe un vector de 32 bits que contiene un número de punto flotante en representación IEEE754 de precisión sencilla, y lo separa en signo, exponente, mantissa, y detecta números especiales: Cero, Infinito (INF) y No-Es-Un-Numero (NaN). Esta operación se realiza para cada uno de los operandos de entrada.

5.2.7. Intercambio Selectivo

Dependiendo del signo de la diferencia de los exponentes se elige qué operando se debe alinear. Para ello, se usa este signo para determinar si el bloque *SelectiveSwap* intercambia de posición los significandos, en preparación para el bloque *Align*.

5.2.8. Alineación de las Mantissas

La Alineación consiste en rotar la mantissa de un operando n veces, donde n es la diferencia entre sus exponentes ($n \in [-28, 28]$), de manera de que ambos significandos se

encuentren representados en el exponente base seleccionado. Realizar esto en un ciclo de reloj implica un desplazamiento en paralelo, donde dependiendo del parámetro se elige la salida correcta, y esto es muy costoso en recursos del FPGA. Por lo tanto, sólo se rota el significando de la entrada *B*.

5.2.9. Negación Selectiva

Si uno de los significandos es negativo, y dependiendo del signo de la operación a realizar, Puede ser necesario complementar uno de los significandos para realizar la suma en complemento a dos. Si es necesario, este bloque realiza la negación, y externamente se añade un acarreo de entrada al sumador de significandos.

5.2.10. Complemento Selectivo

Si el resultado de la suma de significandos es negativo, es necesario complementarla para que el significando del resultado sea positivo, como lo requiere el formato IEEE754. Este blqoue realzia esta operación.

5.2.11. Normalización

El sumador utiliza dos bloques de normalización diferentes. El primero, *Add_Normalize*, normaliza el resultado rotando el significando en cualquiera de las dos direcciones, dado que el rango del resultado no normalizado es $[0, 4)$. Esto representa o 24 rotaciones hacia la izquierda o una rotación hacia la derecha. Este bloque es uno de los más costosos en términos de recursos, ya que además de la lógica para rotación, es necesario un bloque que cuente el número de zeros iniciales (Leading Zero Counter), que es un circuito bastante lento. El segundo bloque es el mismo descrito en el multiplicador, y involucra a lo máximo una rotación hacia la derecha.

5.2.12. Redondeo

Es el mismo bloque utilizado en el multiplicador, y sólo realiza redondeo hacia el número más próximo. En este caso, los bits adicionales aparecen del proceso de alineación de los significandos, así como del proceso de normalización previo al redondeo.

5.2.13. Packing

En este bloque se toman el signo, el exponente y el significando del resultado y se produce un arreglo de 32 bits que es el número de punto flotante de precisión sencilla en representación IEEE754. Si se activó alguna de las banderas de Zero, INF o NaN, este bloque ignora los valores de entrada y produce la salida acorde.

5.2.14. Resultados

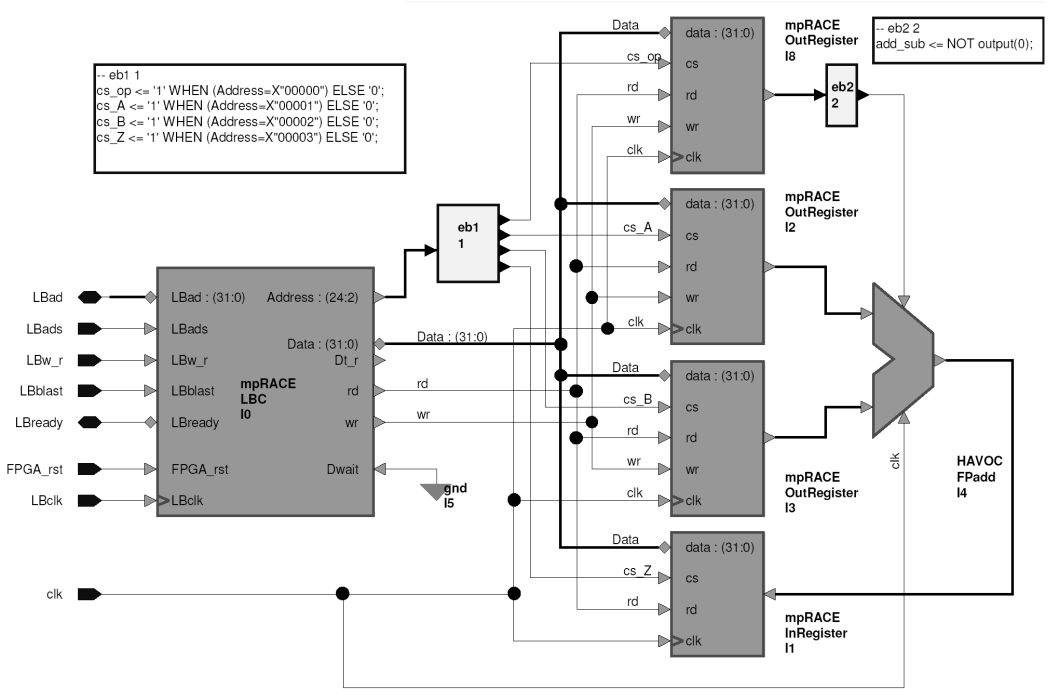

Figura 5.18: Diseño de prueba del FPadd

En esta sección se muestran las pruebas realizadas sobre el sumador de punto flotante.

El diseño de prueba implementado puede verse en la figura 5.18. A continuación se muestra el resultado de un programa de prueba, que realiza operaciones sobre el diseño implementado en la tarjeta funcionando a 33MHz y las compara con el mismo cálculo realizado por la PC (cuyo valor se muestra entre paréntesis).

El siguiente listado está dividido en 2 partes. La primera realiza operaciones de suma, la segunda, operaciones de resta sobre el mismo conjunto de datos. Los listados muestran los operandos, el resultado de realizar la operación con el diseño sintetizado operando a 33 MHz ejecutando en la tarjeta de desarrollo, en decimal y en hexadecimal; y por último, la misma operación realizada por el procesador de la PC, también en decimal y hexadecimal.

Se prueban valores especiales, resultados especiales, y operaciones que afecten la precisión. En particular, se destaca el resultado de la suma $-1{,}0111 + 200$ (línea 11), que produce un resultado incorrecto, con una variación en el bit menos significativo de la palabra hexadecimal. Este es producto del bug que se mencionó anteriormente en el sumador. En analisis detallados de los diagramas de tiempo, pudo observarse que este error se produce luego de sumar los significandos, en las etapas de redondeo.

```
-----------------------------------------
  FPadd

1            + 0            = 1 3f800000            (1) 3f800000
0            + 1            = 1 3f800000            (1) 3f800000
1            + 2            = 3 40400000            (3) 40400000
1.011        + 0.00123      = 1.01223 3f8190c1      (1.01223) 3f8190c1
0.45         + 0.0056       = 0.4556 3ee94467       (0.4556) 3ee94467
-1           + -2           = -3 c0400000           (-3) c0400000
-1.011       + -0.00123     = -1.01223 bf8190c1     (-1.01223) bf8190c1
-0.45        + -0.0056      = -0.4556 bee94467      (-0.4556) bee94467
1            + -2           = -1 bf800000           (-1) bf800000
-1           + 2            = 1 3f800000            (1) 3f800000
-1.0111      + 200          = 198.989 4346fd28      (198.989) 4346fd29
1            + -0.002345    = 0.997655 3f7f6651     (0.997655) 3f7f6651
1            + 3.40282e+038 = 3.40282e+038 7f7fffff (3.40282e+038) 7f7fffff
3.40282e+038 + 3.40282e+038 = 1.#INF 7f800000       (1.#INF) 7f800000
-1           + 1.17549e-038 = -1 bf800000           (-1) bf800000
-----------------------------------------
  FPsub

1            - 0            = 1 3f800000            (1) 3f800000
```

```
1            - 1            = 0 0                   (0) 0
0            - 1            = -1 bf800000           (-1) bf800000
1            - 2            = -1 bf800000           (-1) bf800000
1.011        - 0.00123      = 1.00977 3f814025      (1.00977) 3f814025
0.45         - 0.0056       = 0.4444 3ee38865       (0.4444) 3ee38865
-1           - -2           = 1 3f800000            (1) 3f800000
-1.011       - -0.00123     = -1.00977 bf814025     (-1.00977) bf814025
-0.45        - -0.0056      = -0.4444 bee38865      (-0.4444) bee38865
1            - -2           = 3 40400000            (3) 40400000
-1           - 2            = -3 c0400000           (-3) c0400000
-1.0111      - 200          = -201.011 c34902d7     (-201.011) c34902d7
1            - -0.002345    = 1.00234 3f804cd7      (1.00234) 3f804cd7
1            - 3.40282e+038 = -3.40282e+038 ff7fffff (-3.40282e+038) ff7fffff
3.40282e+038 - 3.40282e+038 = 0 0                   (0) 0
-1           - 1.17549e-038 = -1 bf800000           (-1) bf800000
```

En el anexo C.3 se muestra el reporte de síntesis del diseño de prueba, que utiliza 376 Slices y opera a una frecuencia máxima de 40.3 MHz. En este caso, puede verse que las etapas no están completamente balanceadas, ya que las etapas 2 y 4 ocupan un área mucho mayor que las demás, por lo que son candidatas a subdividirse en un futuro para obtener un mejor desempeño. Nótese que el sumador no utilzia recursos especializados, sólo CLBs. El speedup de la segmentación, comparado con la frecuencia de operación de 6 MHz del sumador de ciclo único, es de sp $= 40/6 = 6{,}66 > 6$, una vez más esto es producto de una mejor optimización de diseños pequeños por parte del sintetizador y del P&R.

En el anexo C.4 se muestran los diagramas de tiempo correspondientes a los patrones de prueba con los modelos de comportamiento del sumador de ciclo único y segmentado. El diseño sintetizado se probó utilizando el programa *FPadd_test.cxx* incluido en el CDROM.

Capítulo 6

Resultados del Coprocesador

En este capítulo mostramos los resultados del funcionamiento y el desempeño de la implementación del coprocesador HAVOC, en su versión sin antememoria y con capacidad de múltiples vectores por búffer. Las pruebas se realizaron en una computadora PC con un procesador P4 de 1.8GHz, 384MB de RAM y un bus PCI de 32bits a 33MHz, ejecutando Windows XP Service Pack 1, y en la cual se instaló la tarjeta de desarrollo RACE-1 y todas las herramientas de desarrollo. En primera instancia se presentan los listados de salida del programa de prueba, para luego proceder a graficar e interpretar los resultados.

6.0.15. Sumario de la Síntesis

En el listado del anexo C.5 puede verse la utilización de los recursos del FPGA. En particular, el límite de cuántas unidades de cómputo se pueden implementar en el diseño no viene dado por el número de CLBs, sino por la combinación de multiplicadores en hardware y bloques de memoria usados. Como puede verse en el sumario, cada unidad usa 5 BlockRAMs y 8 Multiplicadores los cuales no se pueden superponer ya que los multiplicadores usan buses de 18 bits y los BlockRAM de 36 bits, por lo que cada unidad usa en conjunto 13 recursos. Por lo tanto, si el máximo disponible son 96, esto nos da un máximo de $7{,}38 \approx 7$ unidades, que es el reporte anexo y que representa un uso del 94 % de este subconjunto de recursos disponibles. Dado que Leonardo recomienda no sobrepasar el 80 % de la utilización de los recursos dado que esto actúa en detrimento del desempeño al aumentar el tiempo de ruteo de las señales, se decidió disminuir la cuenta a 6 unidades de cómputo.

Del diseño con 7 unidades de cómputo, puede observarse que se utilizan 94 % de la

combinación Mulplicadores/BlockRAM, pero sólo 56 % de los Slices disponibles. La señal *clk_int* es el reloj de salida del bloque de manejo de reloj, y es la que alimenta a las unidades de cómputo. Por lo tanto, dado este reporte, no es posible operar este diseño con un reloj de 33 MHz. El diseño con 6 unidades de cómputo da mayor holgura al P&R, permitiendo operar a una frecuencia de 33 MHz sin problemas, que es el que se utilizó para las pruebas.

6.0.16. Listado de Salida, Operación

El listado de salida, mostrado en el anexo C.6, se divide en tres secciones. La primera parte tiene por objetivo verificar la confiabilidad de los resultados producidos por el coprocesador, dado el bug presente en el sumador. El criterio es:

- Si el resultado de la operación sobre un vector es igual al resultado producido por la PC para dicho vector, el resultado es correcto.
- Si el resultado de la operación sobre un vector diferente del resultado producido por la PC en menos de 16 ulp ($abs(HAVOC - PC) < 16$, tomando los valores en hexadecimal), el resultado se considera inexacto.
- Si el resultado difiere en más de 16 ulp, se considera erróneo.

La segunda parte del listado mide el número de ciclos utilizados por el coprocesador para calcular un vector, y calcula el tiempo basado en la frecuencia de reloj de 33MHz. Al contar el número de ciclos de reloj y utilizar un circuito dedicado para ello, no hay overhead asociado a esta medición.

En la gráfica 6.1 se puede observar como el número de ciclos aumenta linealmente con un incremento exponencial de la longitud de los vectores calculados. En la gráfica 6.2 se puede observar la mejora producto de usar múltiples vectores por búffer, al compararlo con los ciclos usados por un solo vector por búffer. Por último, en la gráfica 6.3 se presenta la ganancia de velocidad asociada con esta mejora. En particular es importante destacar la ganancia de 5.4 para vectores de 64 elementos, dado que SPHINX calcula vectores de 37 elementos.

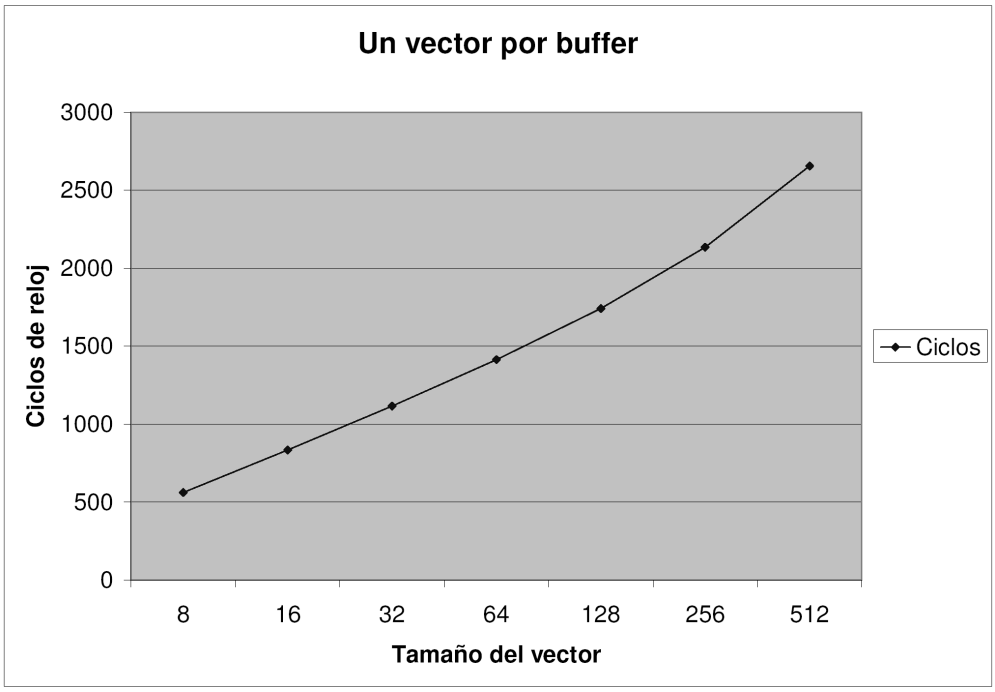

Figura 6.1: Un vector por buffer

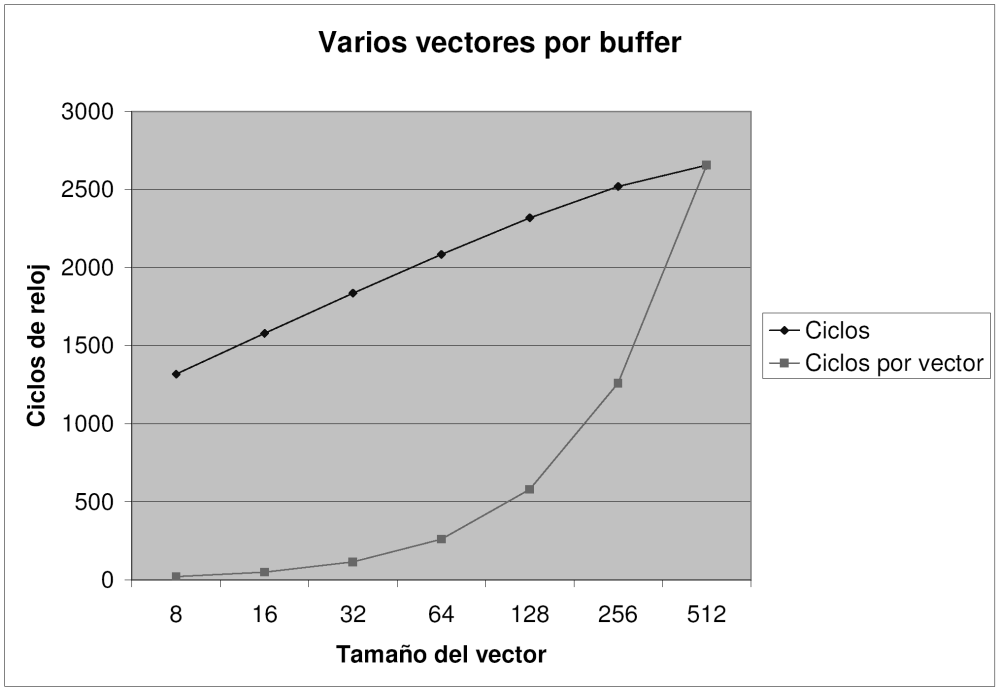

Figura 6.2: Varios vectores por buffer

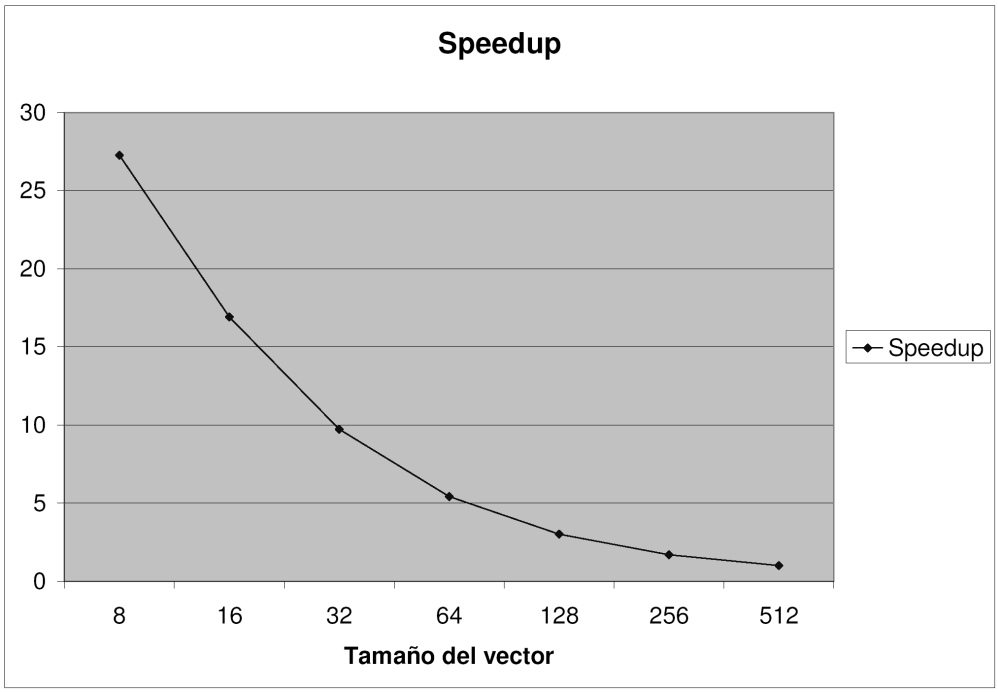

Figura 6.3: Speedup de usar varios vectores por búffer

6.0.17. Listado de Salida, Desempeño

En esta tercera sección del listado se presentan tiempos de ejecución. Se utilizan dos métodos diferentes para cargar los datos al coprocesador. En el método 1, primero se cargan todos los buffers y luego se activan todas las unidades en paralelo. En el método 2, se van cargando las unidades a medida que se liberan de trabajo, y se activan apenas se terminan de cargar. Por lo tanto, las unidades no van funcionando de manera síncrona. A continuación se detallan las columans de las tablas:—

- **Units**, Número de unidades de cómputo utilizadas para el cálculo en el coprocesador.
- **Length**, Longitud de los vectores utilizados para la prueba. Como se está probando el coprocesador con múltiple vectores por búffer, éstos se llenaron a su máxima capacidad para el tamaño del vector, para aprovechar el speedup asociado.
- **PC**, Tiempo consumido por la PC en realizar el mismo cálculo del vector, medido utilizando funciones del sistema.
- **HAVOC**, Tiempo consumido por el coprocesador en realizar el cálculo del vector, medido utilizando funciones del sistema.
- **Speedup**, Ganancia de tiempo, calculada al dividir el tiempo de la PC / tiempo de HAVOC.
- **HAVOC_T**, Tiempo consumido por el coprocesador en realizar el cálculo del vector, medido utilizando el contador de ciclos de reloj presente en HAVOC.
- **HAVOC_L**, Tiempo consumido por la PC en realizar la carga de los datos en los búffers del coprocesador. Medido utilizando el contador de ciclos de reloj presente en HAVOC.
- **HAVOC_C**, Tiempo utilizado por el coprocesador en realizar los cálculos. Se mide como el tiempo desde que la PC envia el comando de activar la unidad hasta que la PC detecta que el coprocesador ha concluido. Medido utilizando el contador de ciclos de reloj presente en HAVOC.

- **overhead**, Es la diferencia entre el tiempo total medido en el coprocesador (HAVOC_T), y el tiempo utilizado para realizar los cálculos (HAVOC_C). Representa la suma de todos los tiempos que no implican tiempo actual de cómputo.

Siendo los resultados muy similares entre el método 1 y el 2, analizaremos los resultados solo para el método 1, ya que el método 2 no da un beneficio siginificativo y hace los análisis mas complejos. En las gráficas 6.4 y 6.5 se muestra la composición de los tiempos totales para cada tamaño de vector, con una y con seis unidades de cómputo, respectivamente. En estas gráficas se puede apreciar que el tiempo total es principalmente overhead, y que ese overhead es casi en su totalidad correspondiente al tiempo de carga de los búffers previos al inicio del cómputo. Asimismo, se puede observar que esta tendencia se hace más notable a medida que se aumentan las unidades de cómputo.

Por lo tanto, dado que la PC ya posee los datos en su memoria y usa un canal dedicado para comunicarse con ella, en la gráfica 6.6 comparamos el tiempo de computo de la PC con los tiempos de cómputo usando de 1-6 unidades de cómputo, para diferentes tamaños de vector. En esta gráfica es importante notar que los tiempo de la PC presentan dos caracterisitcas: primero, algunos tiempos son reportados como cero, dado que son muy pequeños para que el sistema los pueda medir, y seguidamente, están ̈paquetizados, esto quiere decir que el sistema miden en múltiplos de una cantidad fija, por lo que con tiempos pequeños el error es muy significativo. Para paliar esta situación, se realizan 10000 iteraciones de cada cálculo. De esta gráfica pueden extraerse varios datos importantes. Primeramente, puede verse que la mejora de utilizar más unidades de cómputo es lineal, esto es, duplicar el número de unidades de cómputo disminuye el tiempo de computo a la mitad. Luego, puede verse que el aumento del tiempo de cómputo con respecto al tamaño del vector es casi lineal, lo que quiere decir que duplicar el tamaño del vector equivalente a duplicar el tiempo de cómputo. Por último, comparando los tiempos de cómputo con el tiempo de la PC, puede verse que con 6 unidades de cómputo (en algunos casos con menos), para tamaños de vectores grandes, el coprocesador es mas rápido que la PC. Para tiempos menores, el método de medición en la PC no permite hacer una comparación.

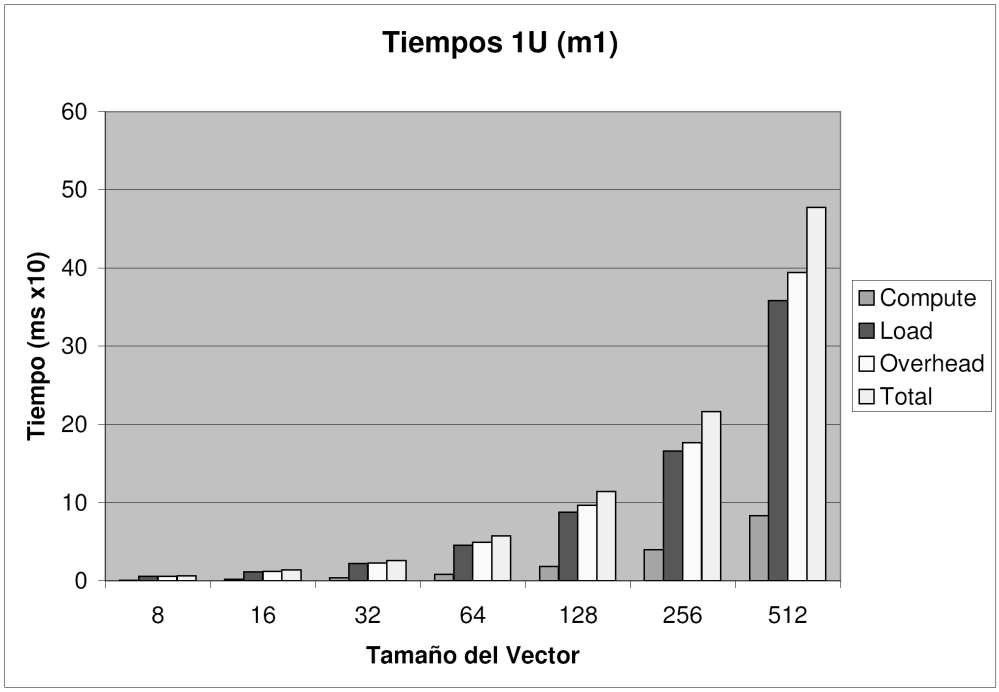

Figura 6.4: Tiempos de 1 Unidad, método 1

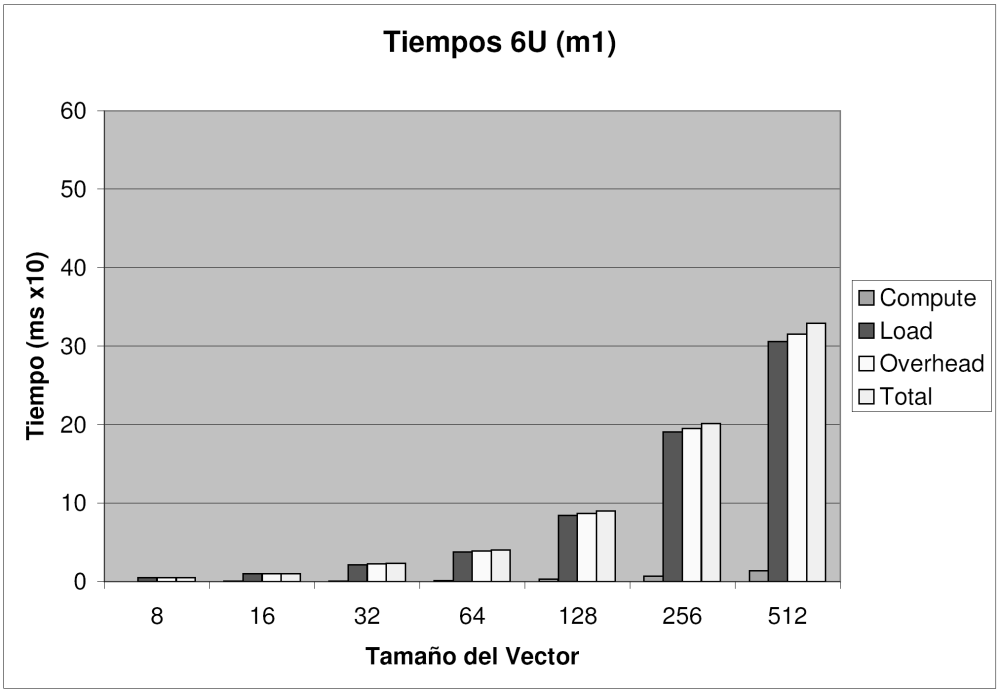

Figura 6.5: Tiempos de 6 Unidades, método 1

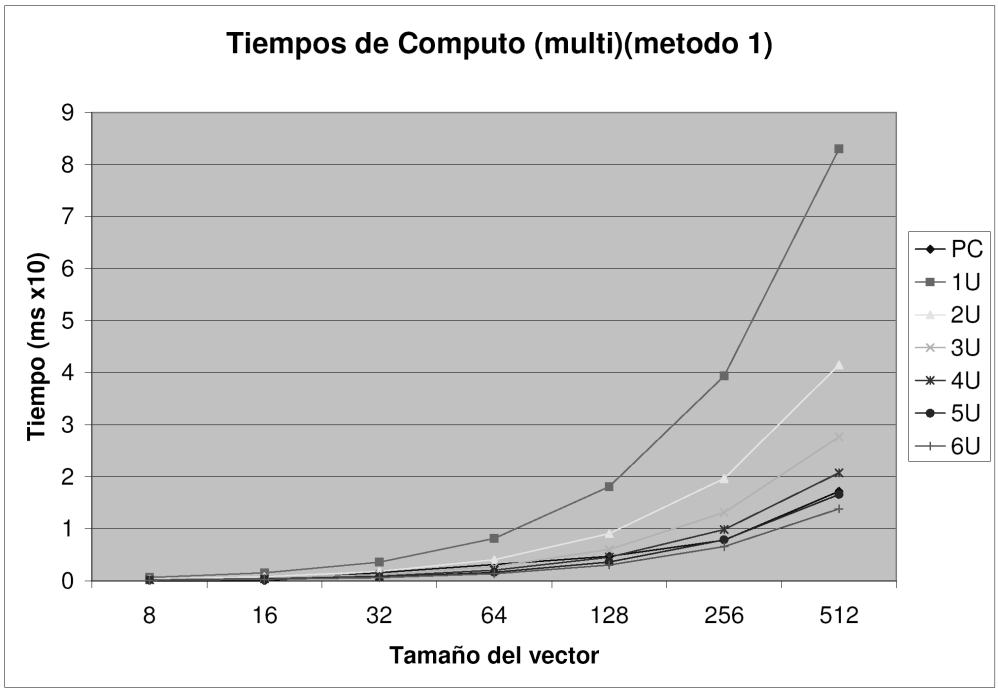

Figura 6.6: Comparativa de Tiempos de Cómputo, método 1

Capítulo 7

Conclusiones

Como resultados de este trabajo de investigación y basados en los objetivos propuestos en la Introducción de este documento, podemos presentar los siguientes puntos:

- Se completó el diseño e implementación de un multiplicador de punto flotante de precisión sencilla. Se presenta un diseño de ciclo único y uno segmentado de 4 etapas, que ocupa 157 SLICEs y es capaz de operar a 50 MHz.
- Se completó el diseño e implementación de un sumador de punto flotante de precisión sencilla, tambien en ciclo único y segmentado de 6 etapas, que ocupa 376 SLICEs y es capaz de operar a 40 MHz.
- Se completó el desarrollo de una primera generación de partes de interfaz, en particular del Cliente de Bus Local y del Manejador de Memorias ZBT, para la utilización de los componentes de la tarjeta de desarrollo.
- Se completó el desarrollo de una primera generación del coprocesador HAVOC que realiza el trabajo deseado aunque no con el nivel de desempeño esperado. Opera a 33MHz y con 2 unidades de cómputo ocupa 2360 SLICEs. Esta versión posee varias optimizaciones ya implementadas, como el manejo de búffers de entrada, manejo de varios vectores por búffer, tamaño de vector configurable, cálculo eficiente de la sumatoria de valores y un modo de debugging para la detección y análisis de errores.

Las unidades de punto flotante son un logro bastante significativo, ya que no se encontraron unidades disponibles para uso académico, por lo que es una contribución que puede hacerse de dominio público para el uso de otros centros educativos y de investigación. A la vez, pueden usarse en cursos de aritmética computacional como apoyo en la explicación de operaciones con punto flotante, y seguir trabajndo sobre ellas para tener unidades con funcionalidad completa.

El coprocesador para entrenamiento de voz puede compararse con trabajos similares, pero la mayoría de las publicaciones se concentran en algoritmos para HMM discretos. Sin embargo, en el trabajo realizado con HMM continuos publicado por Stephen Melnikoff y Martin Russell en [Rus00][Rus02a][Rus02b], donde se hace un desarrollo muy similar al que hacemos aquí, los autores presentan un circuito que opera a 44MHz y se implementa en un Virtex XCV1000 en una tarjeta con 8MB de memoria SRAM, que genera una ganancia de 40 veces con respecto a la implementación en software ejecutándose en un Pentium III a 450MHz. Este coprocesador produce un vector de observación cada 40 ciclos (para un vector de 39 elementos), lo que equivale a una observación cada $0{,}9\mu s$, mientras que HAVOC produce 8 vectores de 64 elementos cada 2084 ciclos, o 260 ciclos por vector de observación, que es un vector cada $7{,}8\mu s$ a 33MHz. La diferencia principal entre ambos diseños viene dada por la manera como realizan la acumulación de los valores, ya que los autores aparentemente cuentan con un acumulador de ciclo único capaz de operar a 40 MHz, o más probablemente, convierten los resultados a punto fijo para usar un acumulador sencillo, mientras que HAVOC realiza varias iteraciones de un sumador multietapas para obtener la acumulación de resultado.

Aun cuando el enfoque usado en este trabajo es diferente y por el momento no aporta ganancia de desempeño al sistema, es importante destacar que hay pocas publicaciones en el área de HMM continuos en hardware y es un trabajo que nos acerca a los desarrollos de punta. Como primera iteración de un proceso de desarrollo, es un avance enorme, que permite establecer comparaciones entre diferentes enfoques al abordar un mismo problema.

7.1. Trabajos Futuros

Este trabajo abre muchas opciones para trabajos futuros. Por un lado, es necesario corregir el bug del sumador, ademas de otras mejoras, en particular en la unidad de ZLC, que puede reemplazarse por un predictor de ceros. Ambas unidades pueden someterse a pruebas más profundas medificando y utilizando la suite *SoftFloat* de pruebas de unidades de punto flotante. Ambas unidades de punto flotante pueden extenderse para manejar doble precisión, pueden añadirse las banderas de excepción y varios modos de redondeo, así como realizar optimizaciones de posición a nivel de SLICE / CLB en el FPGA para optimizar su desempeño. Esto puede ser muy provechoso en el futuro, ya que Xilinx está extendiendo la cobertura de la arquitectura Virtex-II hacia la familia Spartan-3.

La arquitectura del coprocesador puede modificarse para añadir de una manera más directa el acceso a las memorias ZBT, disminuyendo con ello, y por mucho, el tiempo asociado a la carga de los datos. Puede flexibilizarse el tamaño de los vectores para evitar desperdiciar ciclos en el cálculo de las suma (se necesitan 39 pero se calculan 64, por lo que 25 valores del vector son calculados y no son usados, un desperdicio de 39 %). Puede agregarse cierto nivel de programación, de manera de aprovechar los recursos del coprocesador en otros procesos, aparte de SPHINX.

Apéndice A

Reconocimiento de Voz

En este apéndice se presentan los conceptos básicos de los algoritmos de reconocimiento de voz basados en modelos escondidos de Markov. Exposiciones más detalladas pueden encontrarse en [Hon01] y [Rab89].

A.1. Cadenas de Markov

Una cadena de Markov es una maquina de estados finita donde los estados representan las observaciones, y cada transición de un estado a otro viene representado por una probabilidad. En este sistema, la salida es el estado en el que se encuentra. Por lo tanto, si se desea conocer la probabilidad de que una secuencia determinada de observaciones (o de estados), basta con multiplicar las probabilidades de que los estados sigan la secuencia deseada.

Así, si se tiene una secuencia de observaciones cualquiera $X = \{X_1, X_2, X_3, ..., X_N\}$, donde cada observación corresponde con un elemento de un alfabeto $O = \{O_1, O_2, O_3, ..., O_M\}$, se puede escribir la probabilidad de una secuencia como:

$$P(X) = P(X_1, X_2, X_3, ..., X_N) = P(X_1) \prod_{i=2}^{N} P(X_i | X_1^{i-1})$$

donde $X_1^{i-1} = X_1, ..., X_{i-1}$. Si asumimos que:

$$P(X_i | X_1^{i-1}) = P(X_i | X_{i-1})$$

que es una restricción conocida como la asunción de Markov. Esta quiere decir que la probabilidad de pasar a un estado solo depende del estado en que se encontraba el sistema en el tiempo inmediatamente anterior, y esto define un sistema de primer orden.

Por lo tanto, si definimos como a_{ij} la probabilidad de pasar del estado i al estado j; y a π_i como la probabilidad de que la cadena inicie en el estado i, entonces podemos definir una matriz $A = \{a_{ij}\}$, donde todas las columnas deben de sumar 1; y un vector inicial $\Pi = \{\pi_{ij}\}$ que tambien debe sumar 1, ya que la cadena debe de estar obligatoriamente en un estado definido en todo momento, por lo que la probabilidad de que el estado sea indeterminado es cero.

A.2. Modelos Escondidos de Markov

En los modelos escondidos de Markov (Hidden Markov Models, HMM),la secuencia de observaciones ya no corresponde con los estados. En este caso, cada estado representa un conjunto de probabilidades de que sea una observación determinada. Por lo tanto, no se puede decir que un estado genera únicamente una observación. Para este caso, podemos definir:

- Un alfabeto de observaciones $O = \{o_1, o_2, ..., o_M\}$. Estas corresponden con las salidas que se están modelando y definen los simbolos posibles que se pueden emitir.
- Un conjunto de estados posibles $\Omega = \{s_1, s_2, ..., s_N\}$, donde s_t es el estado del sistema en el tiempo t.
- Una matriz de probabilidad de transición $A = \{a_{ij}\}$, donde a_{ij} representa la probabilidad de pasar del estado i al estado j. Por lo tanto, A es una matriz de NxN.
- Una matriz de probabilidad de salida $B = \{b_i(k)\}$, donde $b_i(k)$ es la probabilidad de emitir el simbolo o_k cuando se encuentra en el estado i. Por lo tanto, se tienen N vectores, uno por estado, cada uno con M probabilidades, una por simbolo. Por lo tanto, B es una matriz de MxN.
- Una distribución inicial de estados $\Pi = \{\pi_i\}$, donde π_i es la probabilidad de que i sea el estado inicial.

Por lo tanto, un HMM se puede escribir como una función parametrizada por:

$$\Phi = (A, B, \Pi)$$

Basandonos en esta notación, se pueden identificar tres incógnitas principales a resolver para los HMM:

- **Evaluación**, que es, Dado un modelo Φ y una secuencia de observaciones $X = \{X_1, X_2, ..., X_T\}$, cual es la probabilidad de que el modelo generara dichas observaciones?, esto es, $P(X|\Phi)$. Esto nos permite evaluar que tán bien un modelo se ajusta a secuencia determinada.

- **Decodificación**. Dado un modelo Φ y una secuencia de observaciones $X = \{X_1, X_2, ..., X_T\}$, cual es la secuencia de estados $S = \{s_1, s_2, ..., s_T\}$ más probable en dicho modelo que genere dicha secuencia? Con esto podemos obtener la secuencia de estados que mejor se adapta a las observaciones dado el modelo.

- **Aprendizaje** Dado un modelo Φ y un conjunto de observaciones, cómo podemos ajustar los parámetros del modelo para maximizar la probabilidad $P(X|\Phi)$. Esto nos permite obtener los parámetros del modelo a partir de un conjunto de observaciones conocidas.

A.3. Programación Dinámica

En la programación dinámica básicamente se calcula la distancia mínima entre dos conjuntos de muestras. Para ello, se usa $D(i, j) = \text{mín}_k[D(i-1, k) + d(k, j)]$, donde $d(i, j)$ es la distancia entre dos vectores de muestras x_i e y_j. Como se puede ver, se descartan los caminos que no son óptimos al depender del camino óptimo de la muestra anterior. De esta manera, no es necesario evaluar todos los posibles caminos. El camino óptimo puede construirse una vez evaluadas todas las muestras mediante backtracking.

A.4. Evaluación de los HMM - Algoritmo hacia delante

Para calcular la probabilidad de una secuencia de observaciones X dado un modelo Φ, la forma más directa es sumar todas las probabilidades de todas las secuencias de estados $S = s_1, s_2, ..., s_T$ que pueden generarla, esto es, la probabilidad de cada una de las secuencias de estados es el producto de la probabilidad inicial y la probabilidad conjunta de salida de los estados restantes:

$$P(X|\Phi) = \sum_S P(S|\Phi)P(X|S,\Phi)$$

Por lo tanto, usando la asunción de Markov, $P(S|\Phi)$ puede reeescribirse como:

$$P(S|\Phi) = P(s_1|\Phi)\prod_{t=2}^{T} P(s_t|s_{t-1},\Phi) = \pi_{s_1}a_{s_1s_2}...a_{s_{T-1}s_T}$$

Para la misma secuencia de estados S puede reescribirse $P(X|S,\Phi)$ como:

$$P(X|S,\Phi) = P(X_1^T|S_1^T,\Phi) = \prod_{t=1}^{T} P(X_t|s_t,\Phi) = b_{s_1}(X_1)b_{s_2}(X_2)...b_{s_T}(X_T)$$

Y sustituyendo ambos desarrollos tenemos:

$$P(X|\Phi) = \sum_S P(S|\Phi)P(X|S,\Phi) = \sum_S a_{s_0s_1}b_{s_1}(X_1)a_{s_1s_2}b_{s_2}(X_2)...a_{s_{T-1}s_T}b_{s_T}(X_T)$$

Este resultado se puede calcular de una manera síncrona, organizando una malla de estados, donde se tienen en columnas los estados y en filas la secuencia de observaciones deseada, por lo que horizontalmente se avanza en t. Cada una de las celdas contiene la probabilidad acumulada (la probabilidad hacia adelante) de que ese estado emita la observación deseada en ese instante de tiempo, dadas las anteriores. La suma de las probabilidades de la última columna es la probabilidad de que el modelo genere esta secuencia particular de observaciones. El algoritmo de probabilidad hacia adelante se puede escribir como:

1. **Inicialización**

$$\alpha_1(i) = \pi_i b_i(X_1) \qquad 1 \leq i \leq N$$

2. **Inducción**

$$\alpha_t(j) = \left[\sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij}\right] b_j(X_t) \qquad 2 \leq t \leq T; \quad 1 \leq j \leq N$$

3. **Terminación**

$$P(X|\Phi) = \sum_{i=1}^{N} \alpha_T(i)$$

A.5. Decodificación - Algoritmo Viterbi

El algoritmo hacia adelante calcula la probabilidad de que un modelo produzca una secuencia de observaciones, pero no da la secuencia de estados que ofrece la mejor aproximación. Para ello, se utiliza un algoritmo similar al hacia adelante, modificado para recordar la secuencia de estados que tiene la probabilidad más alta. Para ello, se utiliza al igual que en programación dinámica un backtracking para recuperar la mejor secuencia.

A.6. Estimación de los parámetros - Algoritmo Baum Welch

La estimación de los parámetros del modelo $\Phi(A, B, \Pi)$ es la tarea más compleja de las tres, ya que no existe un procedimiento analítico que maximice los parámetros del modelo para un conjunto de datos de entrenamiento dado. Aqui se describe el procedimiento iterativo conocido como el algoritmo Baum-Welch, o el algoritmo adelante/atras.

La probabilidad hacia adelante se define como:

$$\alpha_t(i) = P(X_1^t, s_t = i|\Phi)$$

Similarmente, la probabilidad hacia atras se define como:

$$\beta_t(i) = P(X_{t+1}^T | s_t = i, \Phi)$$

Y para el procedimiento hacia atrás tenemos:

1. **Inicialización**

$$\beta_T(i) = 1/N \qquad 1 \leq i \leq N$$

2. **Inducción**

$$\beta_t(i) = \left[\sum_{j=1}^{N} a_{ij} b_j(X_{t+1}) \beta_{t+1}(j) \right] \qquad t = T-1, ..., 1; \quad 1 \leq i \leq N$$

En base a estas dos definiciones podemos definir $\gamma_t(i,j)$, que es la probabilidad de cambiar del estado i al estado j en el momento t, esto es:

$$\gamma_t(i,j) = P(s_{t-1} = i, s_t = j | X_1^T, \Phi) = \frac{\alpha_{t-1}(i) a_{ij} b_j(X_t) \beta_t(j)}{\sum_{k=1}^{N} \alpha_T(k)}$$

Los parámetros se pueden ir refinando iterativamente, maximizando $P(X|\Phi)$ en cada iteración. Esto produce un nuevo juego de parametros que llamamos $\hat{\Phi}$. Entonces tenemos que maximizar la función:

$$Q(\Phi, \hat{\Phi}) = \sum_{S} \frac{P(X, S|\Phi)}{P(X|\Phi)} log P(X, S|\hat{\Phi})$$

Luego de trabajar las ecuaciones, se puede llegar a que:

$$\hat{a_{ij}} = \frac{\sum_{t=1}^{T} \gamma_t(i,j)}{\sum_{t=1}^{T} \sum_{k=1}^{N} \gamma_t(i,k)}$$

$$\hat{b_j}(k) = \frac{\sum_{t \in X_t = o_k} \sum_i \gamma_t(i,j)}{\sum_{t=1}^{T} \sum_i \gamma_t(i,j)}$$

Procedimiento para Baum-Welch:

1. **Estimación inicial** de Φ.

2. **Paso-E**. Calcular $Q(\Phi, \hat{\Phi})$ basado en Q.

3. **Paso-M**. Calcular $\hat{\Phi}$ de acuerdo a las fórmulas de reestimación de parámetros para maximizar la función Q.

4. **Iterar**. Hacer $\Phi = \hat{\Phi}$ y repetir desde el paso 2 hasta cumplir un criterio de convergencia.

Este procedimiento es para una sola secuencia de observaciones. Estos valores pueden consolidarse a través de M secuencias de datos, por lo que se pueden extender las fórmulas para incluir a otros datos.

A.7. HMM continuos

En este caso, las observaciones no son parte de un conjunto finito, sino de un espacio. Esto implica una función de probabilidad de salida diferente, y ya no se requiere el cuantificador, lo que elimina ademas el error de cuantificación.

Ahora, $b_j(x)$ representa funciones de densidad de probabilidad de salida continua, que se puede representar genéricamente como una mezcla de funciones Gausianas. Por lo tanto, si utilizamos M gausianas para aproximar el valor, tenemos que:

$$b_j(x) = \sum_{k=1}^{M} c_{jk} N(x, \mu_{jk}, U_{jk}) = \sum_{k=1}^{M} c_{jk} b_{jk}(x)$$

donde $N(x, \mu_{jk}, U_{jk})$ representa una gausiana simple con vector promedio de μ_{jk} y una matriz de covarianza U_{jk} para el estado j. M representa el numero de componentes de la mezcla, y c_{jk} es el peso de la k-ésima gausiana dentro de la mezcla.

Realizando un desarrollo similar a los HMM discretos para la reestimación de los nuevos parámetros, obtenemos que:

$$\hat{\mu}_{jk} = \frac{\sum_{t=1}^{T} \zeta_t(j,k) x_t}{\sum_{t=1}^{T} \zeta_t(j,k)}$$

$$\hat{U}_{jk} = \frac{\sum_{t=1}^{T} \zeta_t(j,k)(x_t - \hat{\mu}_{jk})(x_t - \hat{\mu}_{jk})^t}{\sum_{t=1}^{T} \zeta_t(j,k)}$$

donde $\zeta_t(j,k)$ se calcula como:

$$\zeta_t(j,k) = \frac{p(X, s_t = j, k_t = k|\Phi)}{p(X|\Phi)} = \frac{\sum_i \alpha_{i-1}(i) a_{ij} c_{jk} b_{jk}(x_t) \beta_t(j)}{\sum_{i=1}^{N} \alpha_T(i)}$$

y el peso dentro de la mezcla se puede reestimar como:

$$\hat{c_{jk}} = \frac{\sum_{t=1}^{T} \zeta_t(j,k)}{\sum_{t=1}^{T} \sum_k \zeta_t(j,k)}$$

Apéndice B

Programas Utilizados

Como plataforma de desarrollo se utilizó un sistema de inicio dual con los sistemas operativos Linux y Windows. Esto es dado que SPHINX se ejecuta en ambiente Linux, pero las herramientas de desarrollo para FPGA utilizadas ejecutaban, al momento de comenzar el desarrollo, únicamente en Windows. Por lo tanto, utilizamos las siguientes configuraciones:

B.1. En Linux

- Linux Mandrake 9.0
- Manejadores PLX
- Librería uelib
- gcc
- SPHINX 3

Para la instalación del manejador PLX y de la libreria uelib en linux, se siguió el siguiente procedimiento, transcrito de las comunicaciones con el grupo de desarrollo en Alemania:

```
Hi Matthias,

I finally was able to get the mpRace board working under Linux Mandrake 9.0. Thanks for the help in previous
```

emails with this issue. I had to do some configuration, so I send you this email mainly to document the installation process. This was done with the pcidriver_mexico and uelib_mexico you sent me earlier.

To install the pciDriver:

* Uncompress the pciDriver to a directory (in my case, to /usr/src/pciDriver)
* change dir to /usr/src/pciDriver
* change user to root
* execute ’./configure’
* execute ’make’
* execute ’make install’

This will produce a warning about the kernel being tainted. This has to do with the way the shl module is being installed. At this point, the driver should be functional.

* execute ’./testDriver’ in the directory /usr/src/pciDriver/src/driver/test.

However, the shl will not reload if the machine is rebooted, forcing to reinstall it again. To work around this, edit the file ’/etc/rc.local’ and add at the end the line: ’insmod shl’. This will install the module at each boot. We are currently looking for a better way to do this.

To install the uelib:

* Uncompress the uelib to a directory (in my case, /usr/uelib)

The configure script does not work properly in Mandrake 9. It does not replace some variables at the moment of creating the makefiles. (In particular, @am_include@, @am_quote@ and @install_sh@ are not replaced).

For this reason, It is necessary to update the automake and autoconf utilities, which are managed by rpm’s in Mandrake. Both versions are included in the Mandrake CDs but are not installed by default, or can be downloaded from update servers on the internet. I installed automake 1.63 and autoconf 2.53, to match as close as possible the versions you use in Germany. Install both using the Mandrake control center. May be neccesary to temporarily remove some packages, like ’build’. It can be reinstalled after the update.

The new versions can be checked using ’automake --version’ and ’autoconf --version’.

In my case, autoconf keep both the older and the new version installed, so it is necessary to manually update the symbolic links at /usr/bin. For this, do:

* change user to root
* go to directory /usr/bin
* remove autom4te
* remove autoconf
* create the new link for autom4te with: ’ln -s autom4te-2.5x autom4te’
* create the new link for autoconf with: ’ln -s autoconf-2.5x autoconf’

This should do the trick. Check again the versions to be sure.

Now, to the uelib:

* change directory to /usr/uelib.
* do ’aclocal’
* do ’libtoolize --force’. Will send a warning about the m4 file. Ignore it.
* do ’autoconf’
* do ’automake’
* execute ’./configure’
* execute ’make’
* execute ’make install’

```
To test it was installed properly, run ’./testuelib’ in /usr/uelib/src/libtest.

I attach the outputs from the testDriver, testuelib and Race1test.

Best Regards,

Guillermo
```

B.2. En Windows

- Windows XP Service Pack 1
- Manejadores PLX
- Librería uelib
- Microsoft Visual Studio 6 Service Pack 5
- Xilinx ISE 4.1
- Mentor Graphics HDL Designer Pro v2002.1b
- Mentor Graphics Leonardo Spectrum v2002e16
- Mentor Graphics Modelsim SE 5.7c

La instalación del manejador PLX en Windows puede requerir una ligera manipulación del archivo INF asociado, y darle la ruta correcta al manejador de dónde debe encontrar los archivos. En particular, la modificación consiste en descomentar la linea que solicita la copia de archivos para el chip 9656, que aparece como:

```
[DDInstall_9656.NT]
AddReg    = AddRegistry_NT_9656
;CopyFiles = CopyFiles_ApiDll, CopyFiles_9656
```

Las herramientas de Mentor Graphics requieren configuración adicional para funcionar con el servidor de licencias de Mentor del ITESM. En particular, hay que configurar la siguiente variable de entorno con el puerto y el servidor de licencias del instituto:

```
LM_LICENSE_FILE=port@server.mty.itesm.mx
```

Dado que se usa la versión profesional de Modelsim en vez de la que viene con el ISE, es necesario crear las librerias precompiladas. Para ello se utiliza el siguiente script, que crea las librerias de primitivas de simulación (simprims), la libreria de bloques lógicos del ISE (logiblox), la libreria de primitivas universales de todos los componentes de Xilinx (uniprim), y los bloques precompilados de la libreria CORE (del CORE Generator), (xilinxCorelib). Fué necesario ejecutar esto manualmente dado que el asistente disponible desde el HDL Designer no funcionó para nuestra versión del ISE.

```
set MODEL_TECH [file join $env(MODEL_TECH)/..]
set XILINX $env(XILINX)
set VHDL_out [file join $MODEL_TECH/xilinx/vhdl]

vlib $VHDL_out/simprim
vmap simprim $VHDL_out/simprim
vcom -explicit -work simprim $XILINX/vhdl/src/simprims/simprim_Vpackage.vhd
vcom -explicit -work simprim $XILINX/vhdl/src/simprims/simprim_Vcomponents.vhd
vcom -explicit -work simprim $XILINX/vhdl/src/simprims/simprim_VITAL.vhd

vlib $VHDL_out/logiblox
vmap logiblox $VHDL_out/logiblox
vcom -explicit -work logiblox $XILINX/vhdl/src/logiblox/mvlutil.vhd
vcom -explicit -work logiblox $XILINX/vhdl/src/logiblox/mvlarith.vhd
vcom -explicit -work logiblox $XILINX/vhdl/src/logiblox/logiblox.vhd

vlib $VHDL_out/unisim
vmap unisim $VHDL_out/unisim
vcom -explicit -work unisim $XILINX/vhdl/src/unisims/unisim_VPKG.vhd
vcom -explicit -work unisim $XILINX/vhdl/src/unisims/unisim_VCOMP.vhd
vcom -explicit -work unisim $XILINX/vhdl/src/unisims/unisim_VITAL.vhd
vcom -explicit -work unisim $XILINX/vhdl/src/unisims/unisim_VCFG4K.vhd

vlib $VHDL_out/xilinxcorelib
vmap xilinxcorelib $VHDL_out/xilinxcorelib


vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/ul_utils.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft_utils.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfftv2_utils.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft1024v2.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft1024v2_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft16v2.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft16v2_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft256v2.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft256v2_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft64v2.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft64v2_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_constants_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_utils_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_fd_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_fd_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_v2_0.vhd
```

```
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mem_init_file_pack.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_ram_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_ram_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bus_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bus_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mem_dp_block_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mem_dp_block_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_addsub_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_addsub_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_compare_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_compare_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_counter_binary_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_counter_binary_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bit_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bit_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_dist_mem_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_dist_mem_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_twos_comp_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_twos_comp_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_constants_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_comps_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_utils_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_fd_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_fd_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_vgen_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_vgen_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_fd_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_fd_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sincos_v2_0_pack.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sincos_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sincos_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft32_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft32_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft1024.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft1024_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft16.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft16_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft256.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft256_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft64.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/vfft64_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mulVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mulVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/acc2sVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/acc2sVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/addsVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/addsVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/adreVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/adreVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/adrleVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/adrleVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/cmpsVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/cmpsVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mux2VHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mux2VHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mux3VHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mux3VHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mux4VHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mux4VHT_comp.vhd
```

```
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/pscVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/pscVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/regceVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/regceVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/saddceVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/saddceVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/subreVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/subreVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/subrleVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/subrleVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/tsb16sVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/tsb16sVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/tsb16xVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/tsb16xVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/tsb32xVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/tsb32xVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sync_fifo_pkg_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sync_fifo_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sync_fifo_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sync_fifo_pkg.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sync_fifo_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sync_fifo_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sqrootVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sqrootVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/trigtabl_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/trigtabl_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dividervht.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dividervht_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sdafirVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sdafirVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/pdafirVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/pdafirVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/ncoiq_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/nco_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_constants_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_const_pkg_behav_v3_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_pkg_v3_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_gen_v3_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_gen_v3_1_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_const_pkg_behav_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_pkg_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_gen_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_gen_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_const_pkg_behav.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_pkg_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_gen_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_gen_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dpramVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dpramVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/romrVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/romrVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/syncramVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/syncramVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_comps_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mac_pack_v1_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_utils_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_fd_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_fd_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_addsub_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_addsub_v3_0_comp.vhd
```

```
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_accum_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_accum_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_vgen_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mult_vgen_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mac_v1_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mac_v1_1_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mac_pack_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mac_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mac_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/kdcm_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/kdcm_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/kdcm_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/kdcm_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/kcmpipeVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/kcmpipeVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/kcmVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/kcmVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/integVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/integVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/fifosyncVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/fifosyncVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/encode_8b10b_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/encode_8b10b_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/delayVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/delayVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/decode_8b10b_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/decode_8b10b_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/decode_8b10b_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/decode_8b10b_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sincos_v3_0_pack.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_constants_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_utils_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_fd_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_fd_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bit_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bit_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_fd_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_fd_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/pipe_bhv_v2_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/pipe_bhv_v2_1_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sincos_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sincos_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_addsub_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_addsub_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_accum_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_accum_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dds_v3_0_pack.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dds_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dds_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sincos_v2_1_pack.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sincos_v2_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/sincos_v2_1_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_accum_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_accum_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dds_v2_0_pack.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dds_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dds_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dafir_pack_v5_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/da_fir_v5_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/da_fir_v5_0_comp.vhd
```

```
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dafir_pack_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/da_fir_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/da_fir_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dafir_pack_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/da_fir_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/da_fir_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/dafir_pack.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/da_fir_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/da_fir_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_dist_mem_v4_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_dist_mem_v4_1_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_dist_mem_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_dist_mem_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/combfiltVHT.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/combfiltVHT_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/cic_pack_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/cic_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/cic_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/cam_pkg_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/cam_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/cam_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mem_sp_block_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mem_sp_block_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemv2_pkg.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_decode_binary_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_decode_binary_v2_0_comp.vhd
#vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemv2sp_v2_0.vhd
#vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemv2sp_v2_0_comp.vhd
#vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemv2dp_pkg.vhd
#vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemv2dp_v2_0.vhd
#vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemv2dp_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemsp_pkg_v3_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mem_init_file_pack_v3_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemsp_v3_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemsp_v3_1_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mem_init_file_pack_sp_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemsp_pkg_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemsp_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemsp_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemdp_pkg_v3_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemdp_v3_1.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemdp_v3_1_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/bit_correlator_comp_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/bit_correlator_pack_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/bit_correlator_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_compare_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_compare_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bus_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bus_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_counter_binary_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_counter_binary_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_decode_binary_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_decode_binary_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_bus_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_bus_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bus_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bus_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_bufe_v4_0.vhd
```

```
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_bufe_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_buft_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_buft_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_ld_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_ld_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_ram_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_ram_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_twos_comp_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_twos_comp_v4_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_comps_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_sim_arch_v4_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_decode_binary_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_decode_binary_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_bus_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_bus_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_bufe_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_bufe_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_buft_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_buft_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_ld_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_ld_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bit_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bit_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_fd_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_fd_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_ram_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_ram_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_twos_comp_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_twos_comp_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_comps_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_sim_arch_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_bus_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_bus_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_bufe_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_bufe_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_buft_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_buft_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_ld_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_ld_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_sim_arch_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_addsub_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_addsub_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_accum_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_accum_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_decode_binary_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_decode_binary_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_bus_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_bus_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_bufe_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_bufe_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_buft_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_slice_buft_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_ld_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_reg_ld_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bit_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bit_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_fd_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_fd_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_ram_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_shift_ram_v1_0_comp.vhd
```

```
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_twos_comp_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_twos_comp_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/pipeline.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_constants.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_comps.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_utils.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_sim_arch.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/prims_sim_arch_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/async_fifo_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_dist_mem_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_dist_mem_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemdp_pkg_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/mem_init_file_pack_dp_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemdp_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/blkmemdp_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_compare_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_compare_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bus_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bus_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_counter_binary_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_counter_binary_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bus_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bus_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_v3_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/async_fifo_v3_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bus_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bus_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/async_fifo_v2_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/async_fifo_v2_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_dist_mem_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_dist_mem_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_compare_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_compare_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bus_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_mux_bus_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_counter_binary_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_counter_binary_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bus_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bus_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/c_gate_bit_v1_0_comp.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/async_fifo_pkg.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/async_fifo_v1_0.vhd
vcom -explicit -work xilinxcorelib $XILINX/vhdl/src/xilinxcorelib/async_fifo_v1_0_comp.vhd
```

Apéndice C

Diagramas y Listados Adicionales

C.1. Reporte de Síntesis del Multiplicador

```
*****************************************************

Cell: FPmul_test    View: struct    Library: tests

*****************************************************


 Cell           Library  References     Total Area

BUFGP           xcv2      2 x
BUFT            xcv2    128 x      1    128 LUTs
FDC             xcv2     25 x      1     25 Dffs or Latches
FDE             xcv2     64 x      1     64 Dffs or Latches
FDP             xcv2      1 x      1      1 Dffs or Latches
FPmul_stage1    havoc     1 x     39     39 gates
                                  39     39 Function Generators
                                   1      1 MUXF5
                                  68     68 Dffs or Latches
FPmul_stage2    havoc     1 x     79     79 gates
                                  39     39 Dffs or Latches
                                  73     73 MUX CARRYs
                                  79     79 Function Generators
                                   4      4 Block Multipliers
FPmul_stage3    havoc     1 x     74     74 gates
                                  38     38 Dffs or Latches
                                   7      7 MUX CARRYs
                                  74     74 Function Generators
FPmul_stage4    havoc     1 x      1      1 MUXF5
                                  32     32 Dffs or Latches
```

```
                                    7      7 MUX CARRYs
                                   74     74 Function Generators
                                   74     74 gates
GND             xcv2      1 x
IBUF            xcv2      4 x
IOBUF           xcv2     32 x
LUT1            xcv2      3 x      1      3 Function Generators
LUT2            xcv2      4 x      1      4 Function Generators
LUT3            xcv2      2 x      1      2 Function Generators
LUT3_L          xcv2     23 x      1     23 Function Generators
LUT4            xcv2     16 x      1     16 Function Generators
MUXCY_L         xcv2     22 x      1     22 MUX CARRYs
OBUFT           xcv2      1 x
XORCY           xcv2     23 x

 Number of ports :                      39
 Number of nets :                      470
 Number of instances :                 355
 Number of references to this view :     0

Total accumulated area :
 Number of Block Multipliers :           4
 Number of Dffs or Latches :           267
 Number of Function Generators :       314
 Number of LUTs :                      128
 Number of MUX CARRYs :                109
 Number of MUXF5 :                       2
 Number of gates :                     314
 Number of accumulated instances :     982
*********************************************
Device Utilization for 2V3000bf957
*********************************************
Resource                Used    Avail   Utilization
-----------------------------------------------
IOs                     39      684       5.70%
Function Generators     314     28672     1.10%
CLB Slices              157     14336     1.10%
Dffs or Latches         267     30724     0.87%
Block RAMs              0       96        0.00%
Block Multipliers       4       96        4.17%

-----------------------------------------------
Using wire table: xcv2-3000-4_wc
```

```
                        Clock Frequency Report

Clock                : Frequency
      -----------------------------------

LBclk                : 76.6 MHz
clk                  : 50.9 MHz
```

C.2. Diagramas de tiempo del Multiplicador

En las figuras C.1, C.2 y C.3 se observa las operaciones del multiplicador de ciclo único usando el modelo de comportamiento en situación ideal. El diagrama se segmentó en 3 figuras separadas dado su tamaño. El patrón de prueba aplicado se simula mediante escrituras en las direcciones correspondientes en el bus local, tal como lo realiza el programa con el modelo sintetizado ejecutando en la tarjeta de desarrollo. El patrón aplicado es el mismo que se usa en el programa.

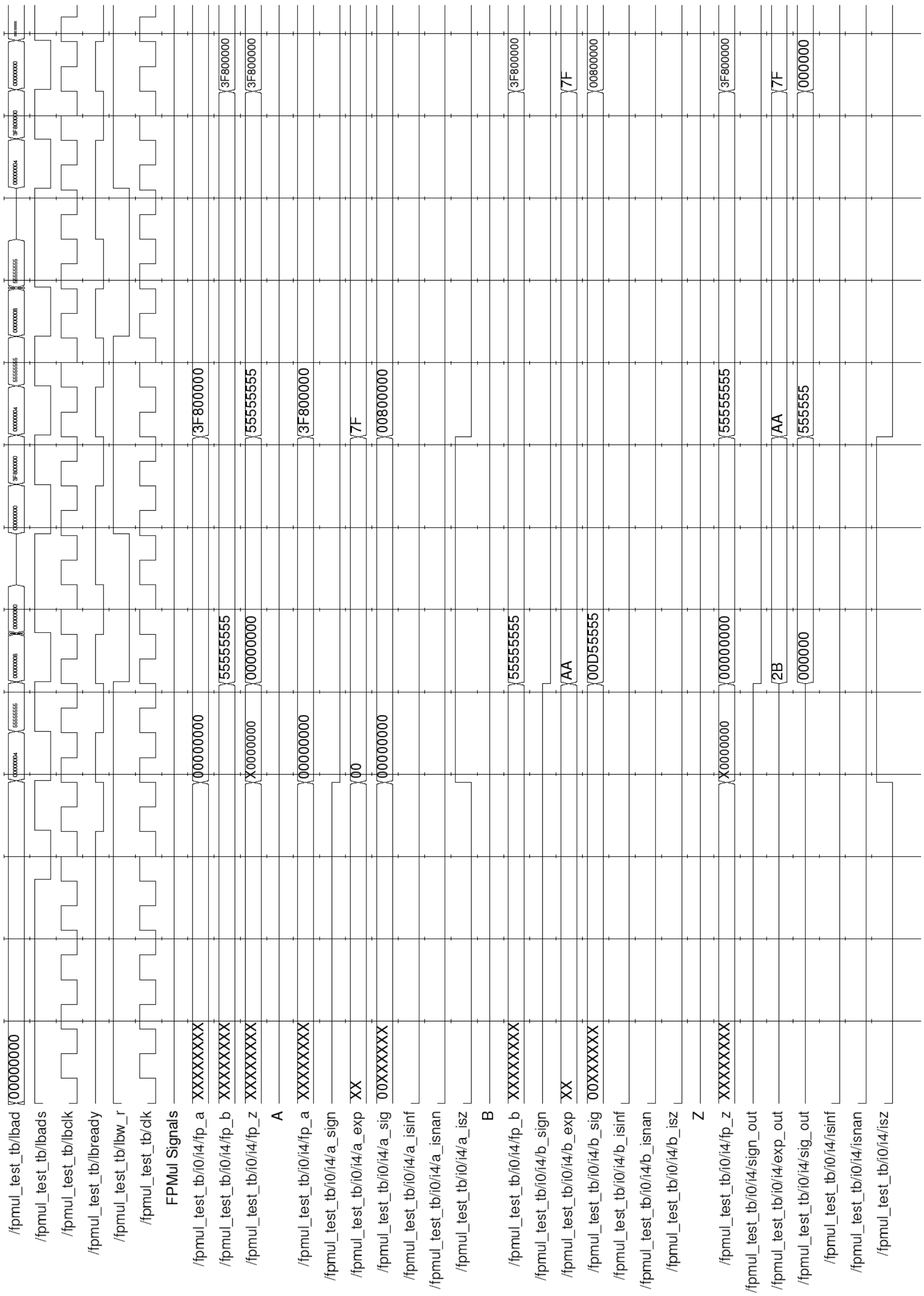

Figura C.1: Multiplicador de Punto Flotante, 1 / 3

Figura C.2: Multiplicador de Punto Flotante, 2 / 3

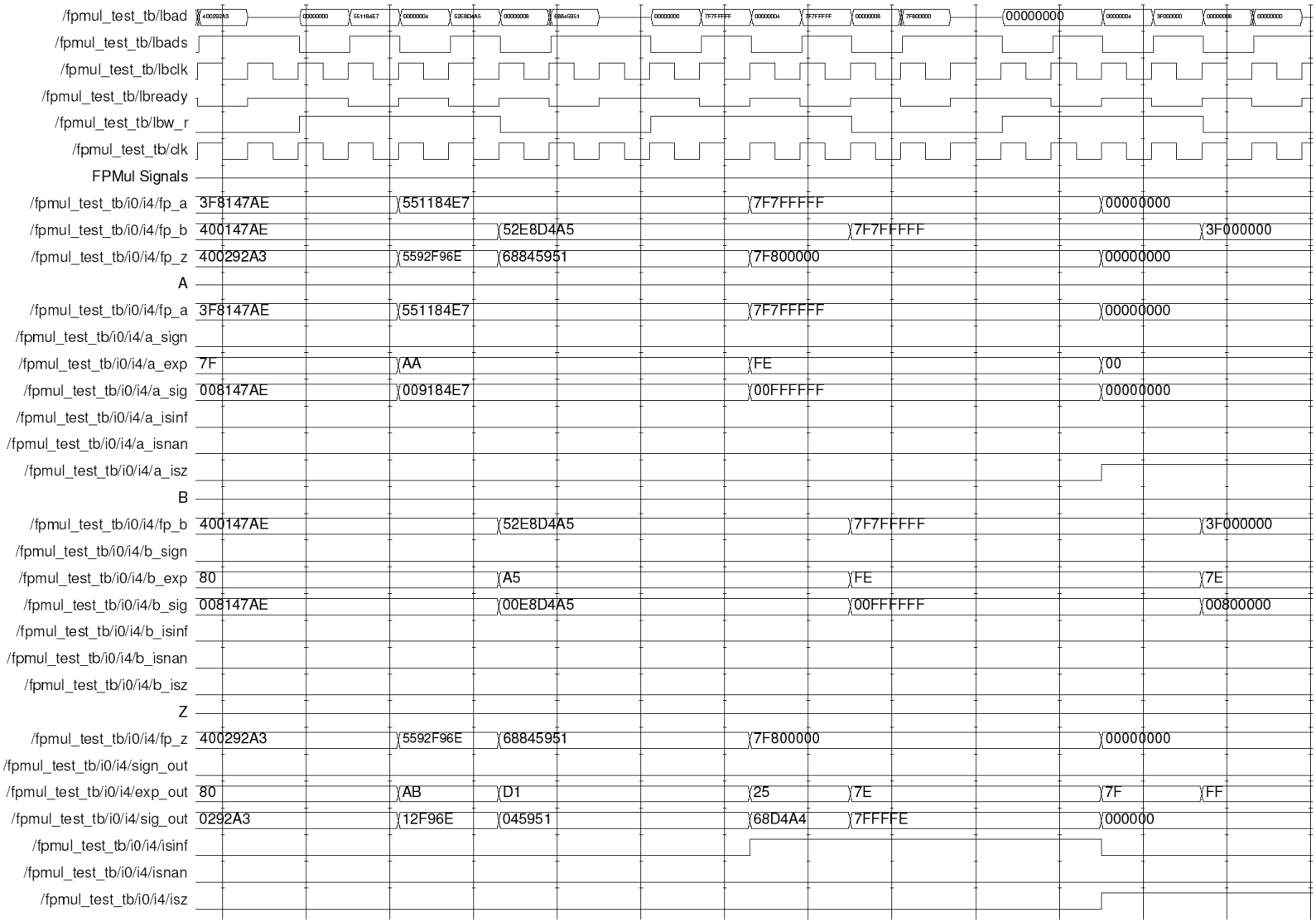

Figura C.3: Multiplicador de Punto Flotante, 3 / 3

En las figuras C.4, C.5 y C.6, se muestra un diagrama similar al anterior, pero para el multiplicador segmentado. En este caso, al multiplicador le toma 4 ciclos producir el resultado correcto. Dado que las entradas se mantienen estables por mucho tiempo, 4 ciclos después de un cambio de valor en las entradas, la salida también se vuelve estable. Como se puede observar, los resultados son equivalentes a los del multiplicador de ciclo único, con la ventaja de que el circuito sintetizado puede operar a una frecuencia mucho mayor (50 MHz, comparado con los 8 MHz del circuito de ciclo único).

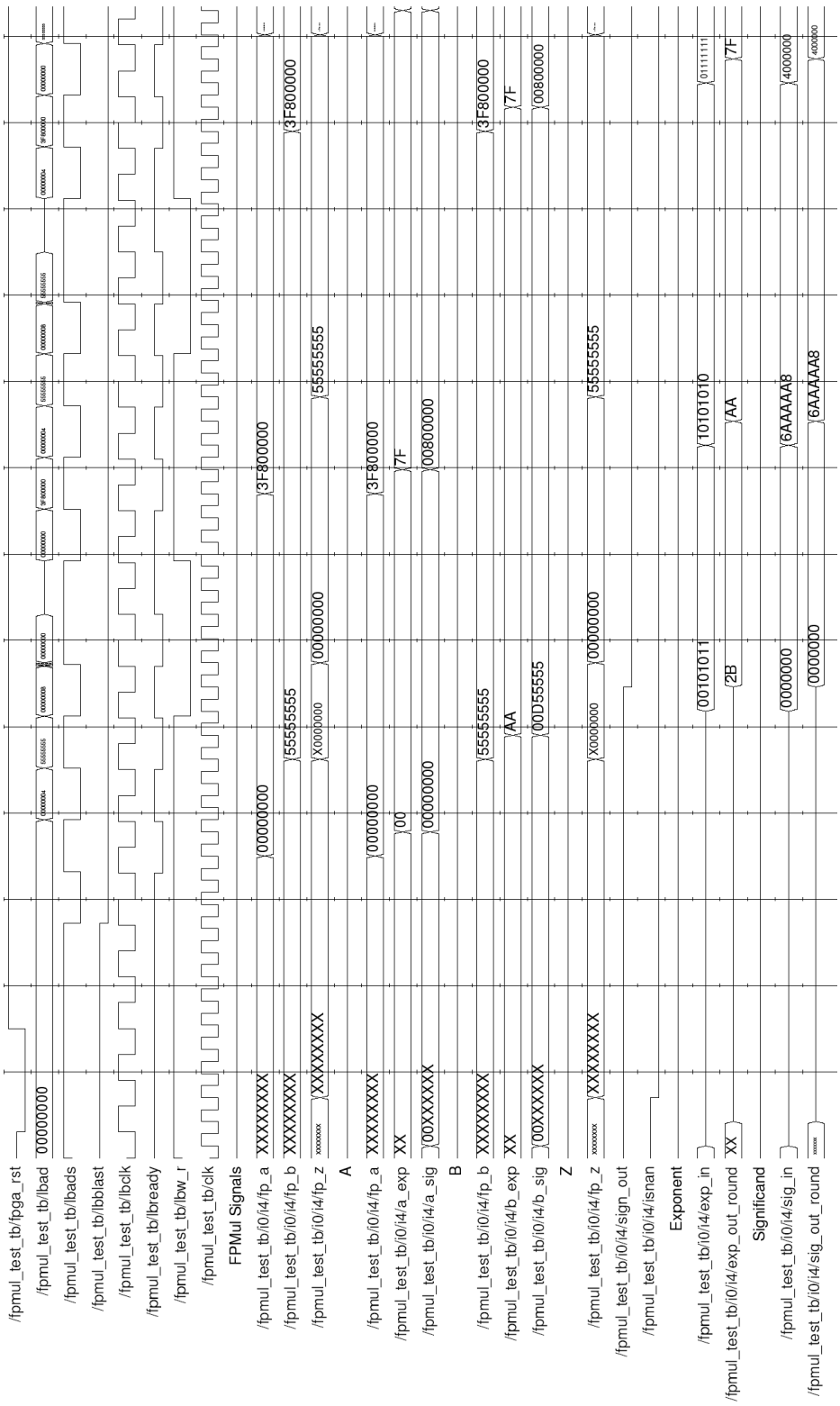

Figura C.4: Multiplicador de Punto Flotante, segmentado, 1 / 3

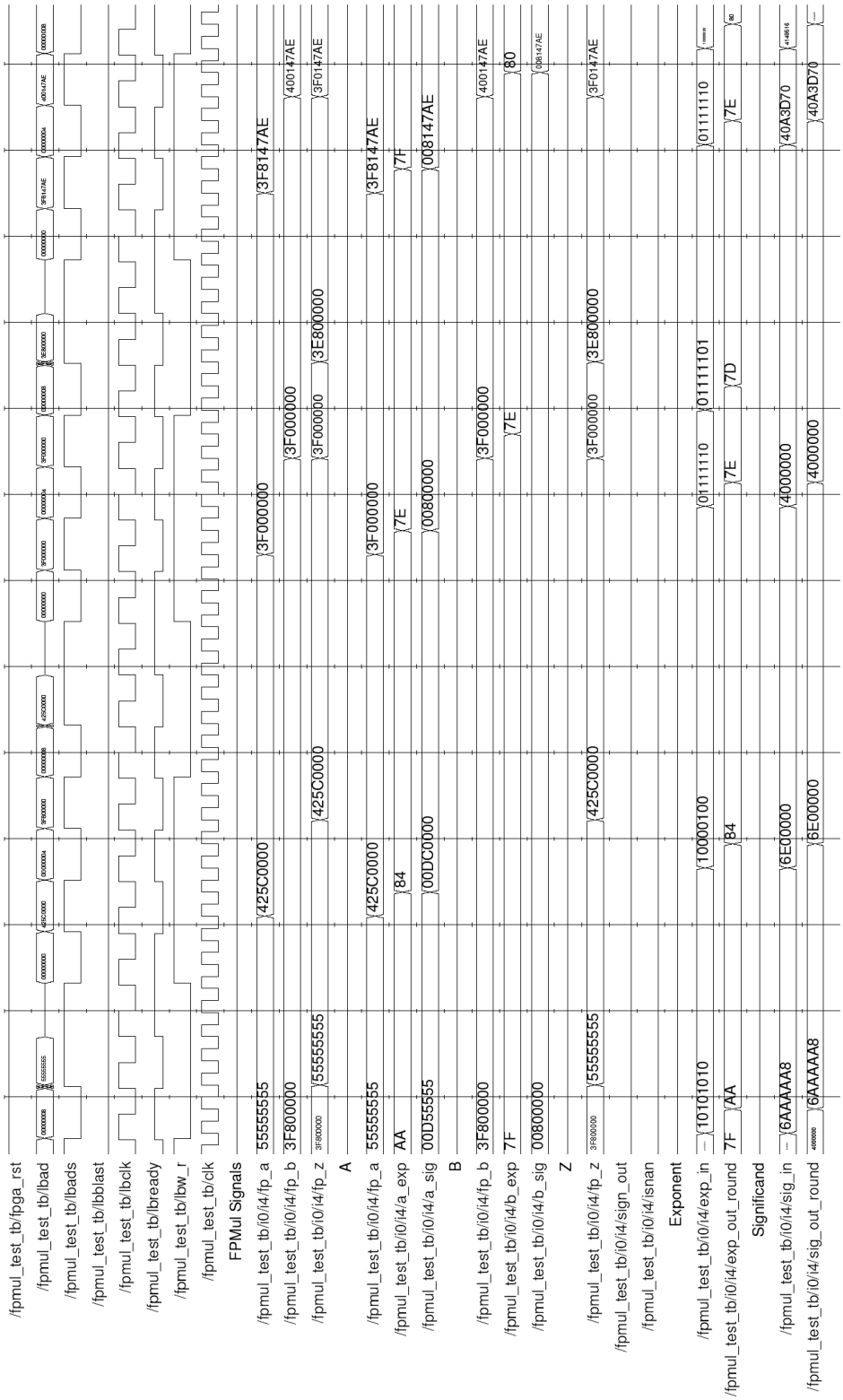

Figura C.5: Multiplicador de Punto Flotante, segmentado, 2 / 3

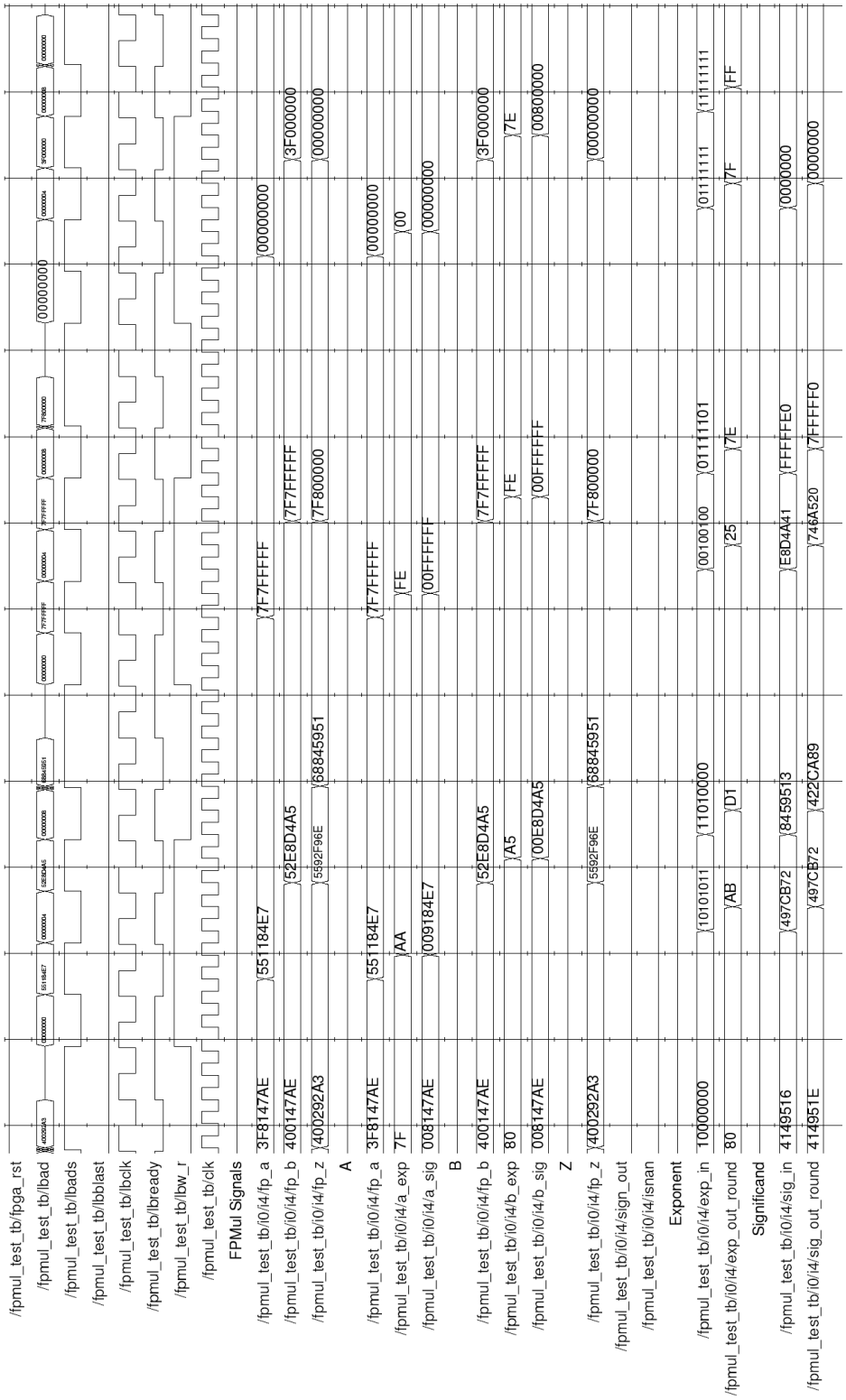

Figura C.6: Multiplicador de Punto Flotante, segmentado, 3 / 3

C.3. Reporte de Síntesis del Sumador

```
*****************************************************

Cell: FPadd_test    View: struct    Library: tests

*****************************************************


 Cell           Library  References     Total Area

BUFGP           xcv2      2 x
BUFT            xcv2    160 x      1    160 LUTs
FD              xcv2     32 x      1     32 Dffs or Latches
FDC             xcv2     25 x      1     25 Dffs or Latches
FDE             xcv2     96 x      1     96 Dffs or Latches
FDP             xcv2      1 x      1      1 Dffs or Latches
FPadd_stage1    havoc     1 x     39     39 gates
                                  80     80 Dffs or Latches
                                   7      7 MUX CARRYs
                                  39     39 Function Generators
FPadd_stage2    havoc     1 x     67     67 Dffs or Latches
                                 191    191 Function Generators
                                   4      4 MUX CARRYs
                                  31     31 MUXF5
                                   8      8 MUXF6
                                 190    190 gates
FPadd_stage3    havoc     1 x     42     42 Dffs or Latches
                                  28     28 MUX CARRYs
                                  80     80 Function Generators
                                  80     80 gates
FPadd_stage4    havoc     1 x     40     40 Dffs or Latches
                                 264    264 gates
                                 264    264 Function Generators
                                  62     62 MUXF5
                                  34     34 MUX CARRYs
                                  10     10 MUXF6
FPadd_stage5    havoc     1 x     91     91 gates
                                  24     24 MUXF5
                                  37     37 Dffs or Latches
                                   7      7 MUX CARRYs
                                  91     91 Function Generators
GND             xcv2      1 x
IBUF            xcv2      4 x
```

```
IOBUF           xcv2     32 x
LUT1            xcv2      4 x      1      4 Function Generators
LUT2            xcv2      5 x      1      5 Function Generators
LUT3            xcv2      2 x      1      2 Function Generators
LUT3_L          xcv2     23 x      1     23 Function Generators
LUT4            xcv2     21 x      1     21 Function Generators
MUXCY_L         xcv2     22 x      1     22 MUX CARRYs
OBUFT           xcv2      1 x
PackFP          havoc     1 x     31     31 gates
                                  31     31 Function Generators
XORCY           xcv2     23 x

 Number of ports :                      39
 Number of nets :                      661
 Number of instances :                 460
 Number of references to this view :     0

Total accumulated area :
 Number of Dffs or Latches :           420
 Number of Function Generators :       751
 Number of LUTs :                      160
 Number of MUX CARRYs :                102
 Number of MUXF5 :                     117
 Number of MUXF6 :                      18
 Number of gates :                     750
 Number of accumulated instances :    1758
*********************************************
Device Utilization for 2V3000bf957
*********************************************
Resource                Used    Avail   Utilization
-----------------------------------------------
IOs                     39      684       5.70%
Function Generators     751     28672     2.62%
CLB Slices              376     14336     2.62%
Dffs or Latches         420     30724     1.37%
Block RAMs              0       96        0.00%
Block Multipliers       0       96        0.00%

-----------------------------------------------
Using wire table: xcv2-3000-4_wc


                      Clock Frequency Report
```

```
Clock                : Frequency
      ------------------------------------

LBclk                : 70.4 MHz
clk                  : 40.3 MHz
```

C.4. Diagramas de tiempo del Sumador

En las figuras C.7, C.8, C.9 y C.10 se muestra el diagrama de tiempos del diseño utilizando el sumador de ciclo único. El diagrama fué dividido en 4 figuras dado su tamaño, y presenta las mismas operaciones que se aplican al diseño sintetizado en el programa de prueba, simulando transacciones en el bus local.

Figura C.7: Sumador de Punto Flotante, 1 / 4

Figura C.8: Sumador de Punto Flotante, 2 / 4

Figura C.9: Sumador de Punto Flotante, 3 / 4

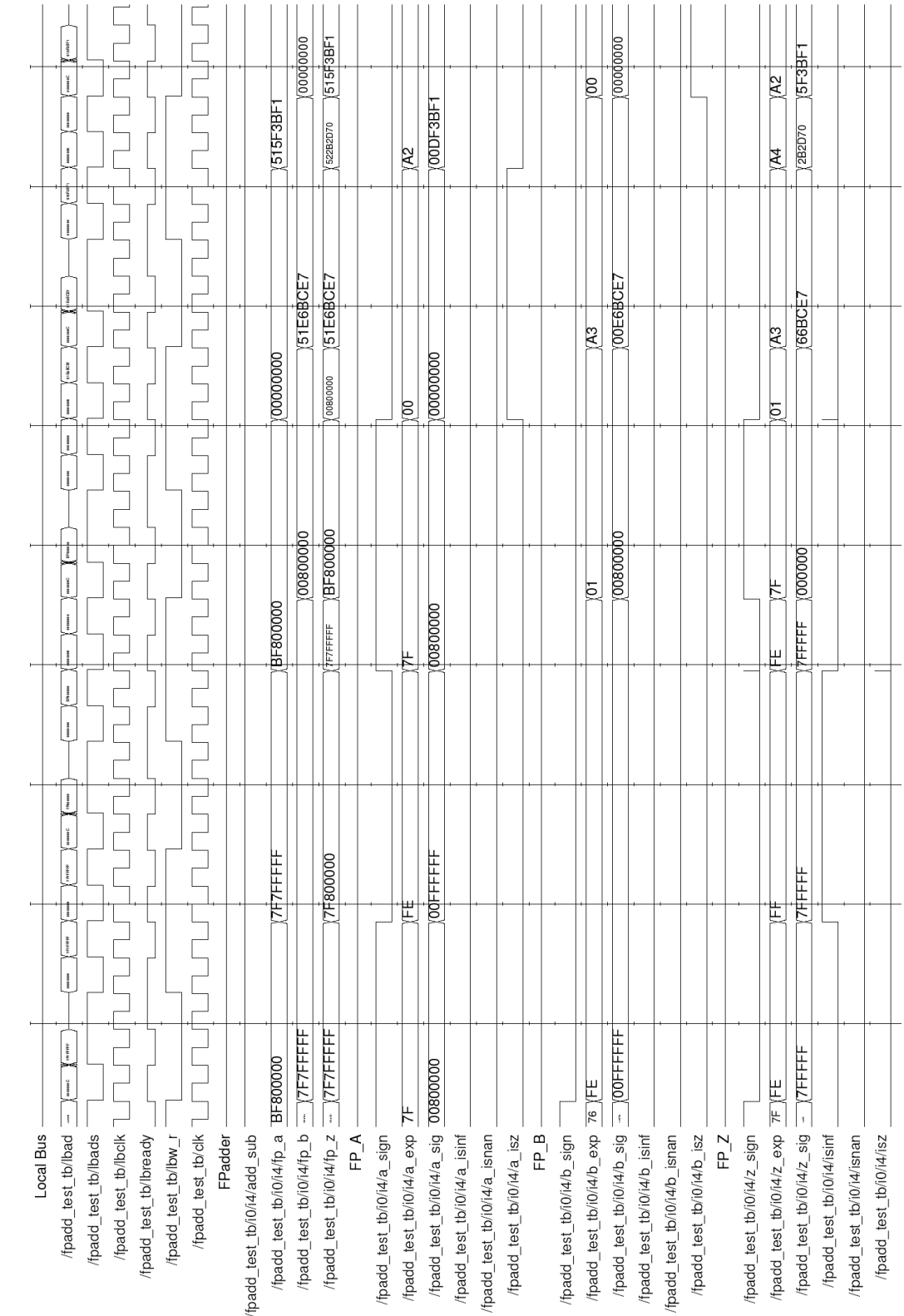

Figura C.10: Sumador de Punto Flotante, 4 / 4

En las figuras C.11, C.12, C.13 y C.14 se presentan los diagramas de tiempo para el diseño de prueba utilizando el sumador segmentado. Aquí se puede observar que el diseño requiere de 6 ciclos para estabilizar la salida una vez que las entradas están estables.

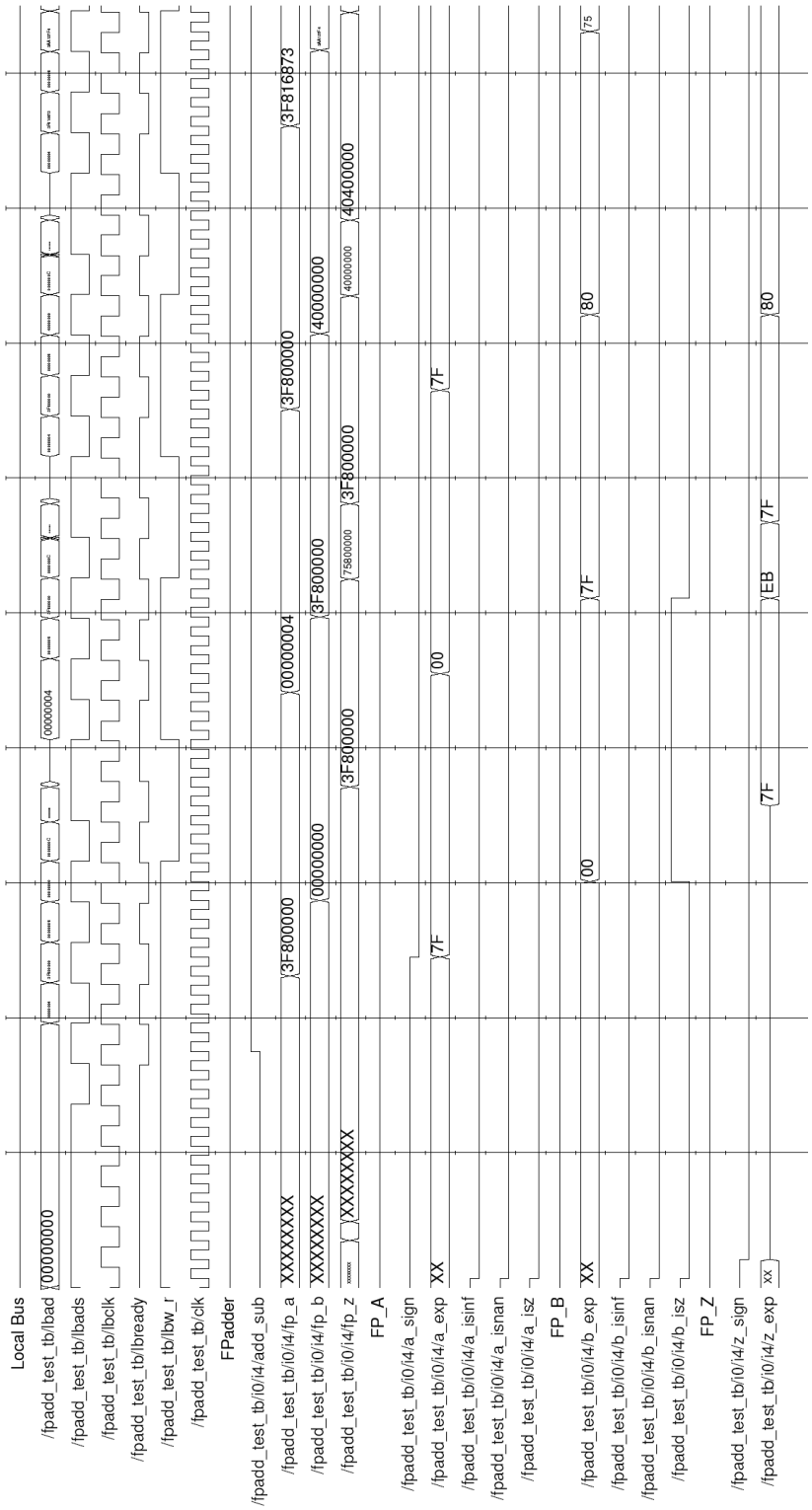

Figura C.11: Sumador de Punto Flotante, segmentado, 1 / 4

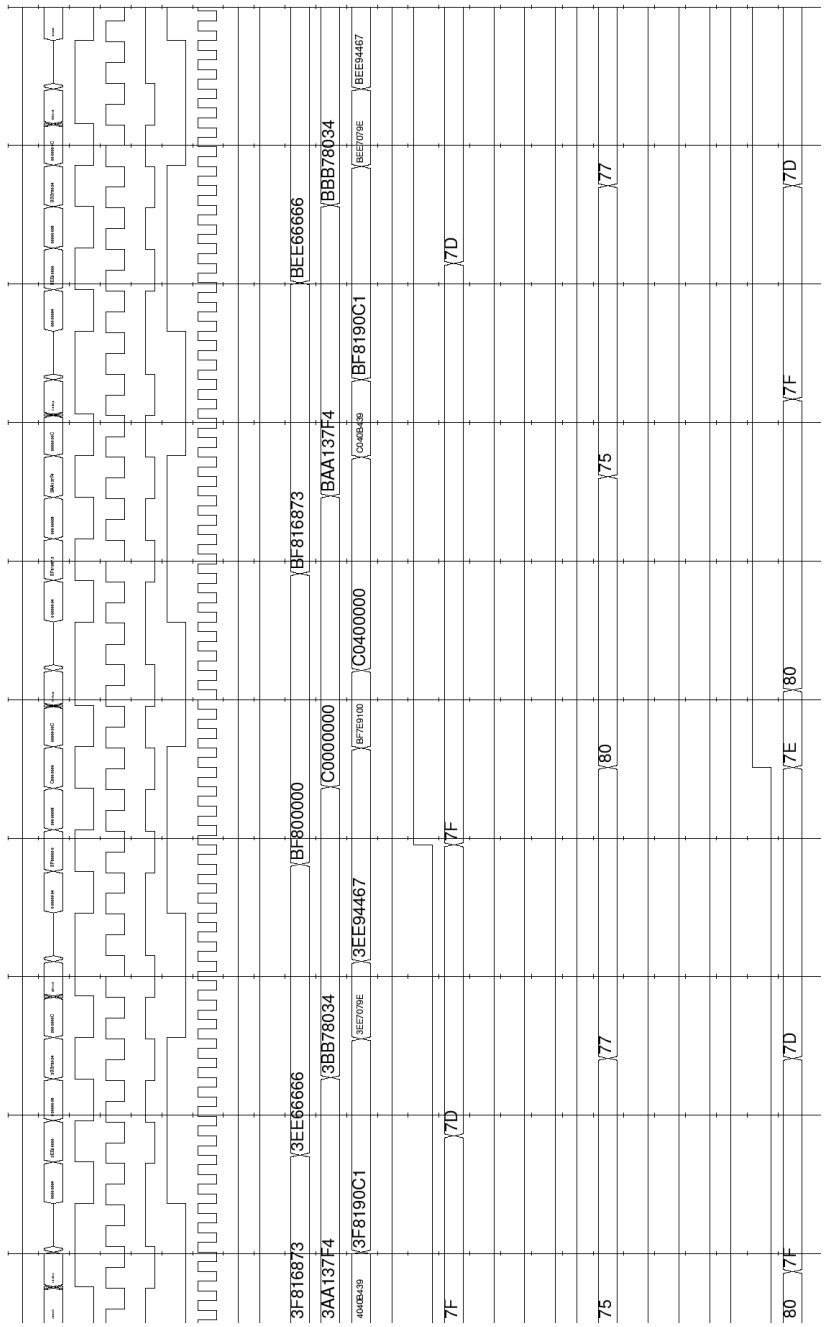

Figura C.12: Sumador de Punto Flotante, segmentado, 2 / 4

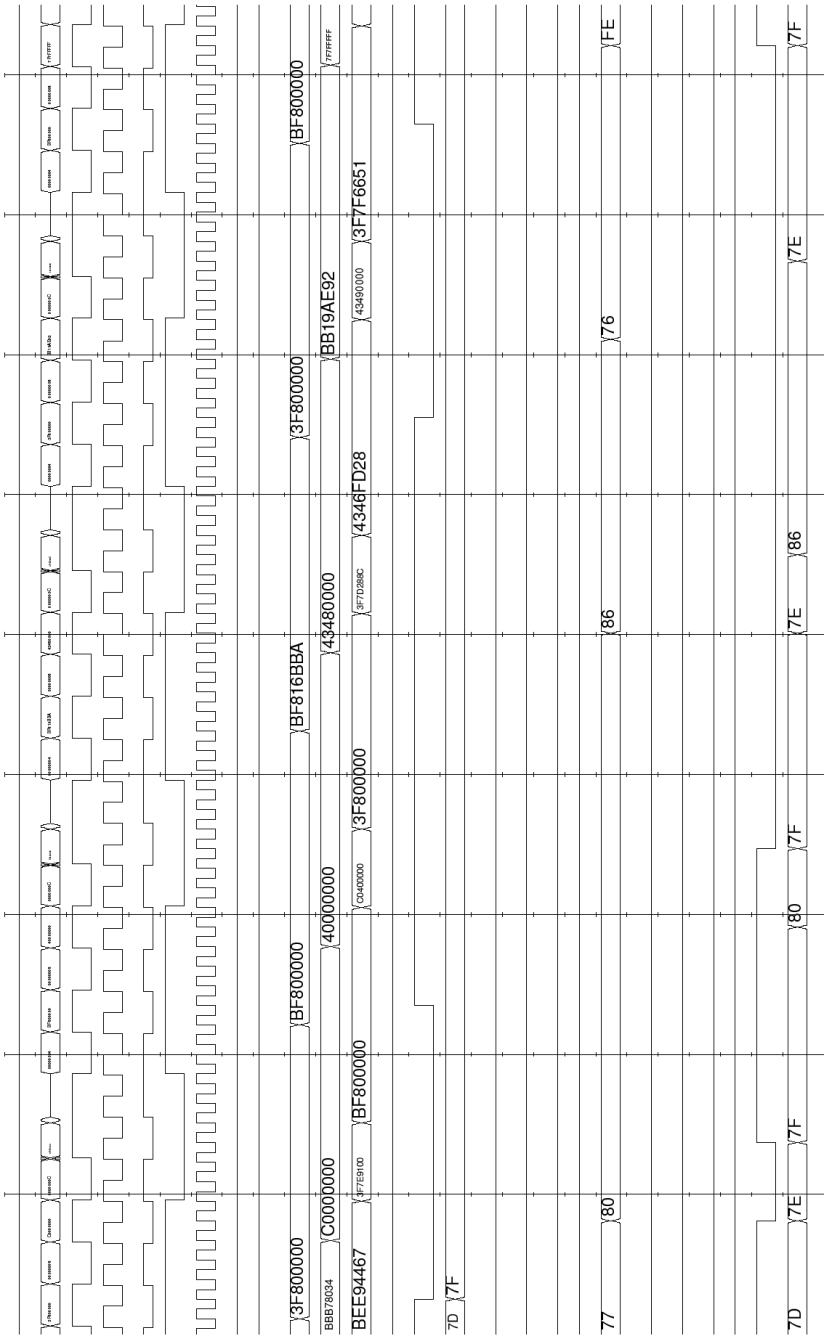

Figura C.13: Sumador de Punto Flotante, segmentado, 3 / 4

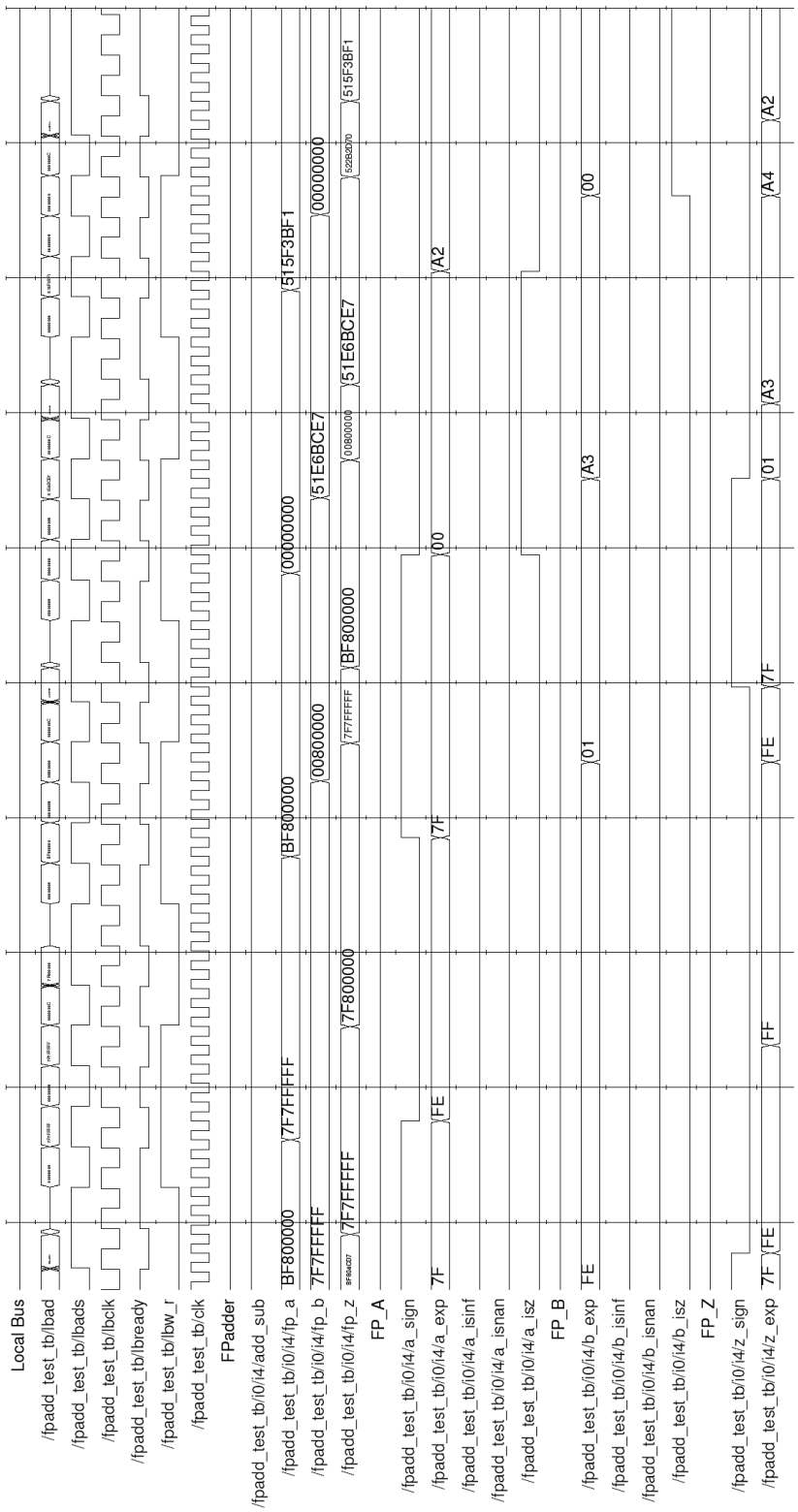

Figura C.14: Sumador de Punto Flotante, segmentado, 4 / 4

C.5. Reporte de Síntesis de HAVOC

```
*****************************************************

Cell: HAVOC    View: struct    Library: havoc

*****************************************************
```

Cell	Library	References			Total Area
BUFG	xcv2	2	x		
BUFGMUX	xcv2	1	x		
BUFGP	xcv2	1	x		
BUFT	xcv2	1216	x	1	1216 LUTs
ClockCounter_notri	mprace	1	x	98	98 gates
				64	64 Dffs or Latches
				63	63 MUX CARRYs
				98	98 Function Generators
DCM	xcv2	2	x		
FDC	xcv2	38	x	1	38 Dffs or Latches
FDE	xcv2	32	x	1	32 Dffs or Latches
FDP	xcv2	1	x	1	1 Dffs or Latches
GND	xcv2	1	x		
GaussComputeUnit_notri	havoc	7	x	8	56 Block Multipliers
				5	35 Block RAMs
				207	1449 MUXF5
				22	154 MUXF6
				2274	15918 Function Generators
				430	3010 MUX CARRYs
				1166	8162 Dffs or Latches
				2208	15456 gates
IBUF	xcv2	5	x		
IOBUF	xcv2	32	x		
LUT1	xcv2	37	x	1	37 Function Generators
LUT2	xcv2	6	x	1	6 Function Generators
LUT3	xcv2	4	x	1	4 Function Generators
LUT3_L	xcv2	23	x	1	23 Function Generators
LUT4	xcv2	30	x	1	30 Function Generators
MUXCY_L	xcv2	22	x	1	22 MUX CARRYs
OBUFT	xcv2	1	x		
XORCY	xcv2	23	x		

```
 Number of ports :                   39
```

```
 Number of nets :                     1332
 Number of instances :                1485
 Number of references to this view :     0

Total accumulated area :
 Number of Block Multipliers :          56
 Number of Block RAMs :                 35
 Number of Dffs or Latches :          8297
 Number of Function Generators :     16116
 Number of LUTs :                     1216
 Number of MUX CARRYs :               3095
 Number of MUXF5 :                    1449
 Number of MUXF6 :                     154
 Number of gates :                   15654
 Number of accumulated instances :   34402
*********************************************
Device Utilization for 2V3000bf957
*********************************************
Resource               Used    Avail   Utilization
-----------------------------------------------
IOs                    39      684       5.70%
Function Generators    16116   28672    56.21%
CLB Slices             8058    14336    56.21%
Dffs or Latches        8297    30724    27.00%
Block RAMs             35      96       36.46%
Block Multipliers      56      96       58.33%

-----------------------------------------------
Using wire table: xcv2-3000-4_wc



                      Clock Frequency Report

Clock                : Frequency
      ----------------------------------

LBclk                : 34.5 MHz
clk                  : N/A
clk_int              : 28.9 MHz
I2_ClkDivider_CLKDIVN_int : N/A
I2_I1_CLK180_int     : 34.5 MHz
```

C.6. Listado de Salida Completo de HAVOC

```
Id is: 4320d19  must be: 4b20839
Found 1 Boards  opening board with logical number 0
Serial: 12
Parallel configuration
Using DMA config
Config FPGA with HAVOC.bit ...ok

Compute Testing
************************************************


Doing testing for 10000 iterations...ok.

 Length   Correct    Inexact        Bad     TOTAL
 -------- ---------- -------------- ------- -----
    8       5069       4931          0     10000
   16       4420       5579          1     10000
   32       3526       6474          0     10000
   64       2699       7300          1     10000
  128       1934       8062          4     10000
  256       1396       8601          3     10000
  512        941       9051          8     10000

Multiple vectors per buffer

Doing testing for 10000 iterations...ok.

 Length   Correct    Inexact        Bad     TOTAL
 -------- ---------- -------------- ------- -----
    8     328255     311733         12    640000
   16     141573     178417         10    320000
   32      56598     103394          8    160000
   64      21483      58507         10     80000
  128       7779      32217          4     40000
  256       2776      17218          6     20000
  512        971       9020          9     10000

Compute Times
****************************************


Single Vector per Buffer

 Length   VPB    Cycles   Time
 -------- ------ -------- -----
     8     1       561      17
    16     1       834      26
    32     1      1115      34
    64     1      1412      44
   128     1      1741      54
   256     1      2134      66
   512     1      2655      82

Multiple Vectors per Buffer

 Length   VPB    Cycles   Time
 -------- ------ -------- -----
     8    64      1317      41
```

16	32	1578	49
32	16	1835	57
64	8	2084	65
128	4	2317	72
256	2	2518	78
512	1	2655	82

Benchmarking
****

Method 1

Units	Length	PC	HAVOC	Speedup	HAVOC_T	HAVOC_L	HAVOC_C	overhead
1	8	0	625000	0	632710	552880	64610	568100
1	16	0	1093750	0	1081810	886090	154340	927470
1	32	156250	2187500	0.0714286	2236130	1761200	358390	1877740
1	64	156250	4687500	0.0333333	4575710	3629130	814060	3761650
1	128	468750	9062500	0.0517241	9289200	7188340	1810150	7479050
1	256	781250	19062500	0.0409836	19164850	14500620	3934370	15230480
1	512	1406250	36718750	0.0382979	36866360	27518570	8296870	28569490
2	8	0	468750	0	504040	459620	32510	471530
2	16	0	937500	0	971390	867610	77420	893970
2	32	0	2031250	0	1941260	1713510	179480	1761780
2	64	156250	4062500	0.0384615	3933010	3435200	407030	3525980
2	128	312500	8437500	0.037037	8343830	7243250	905070	7438760
2	256	781250	17187500	0.0454545	17089000	14579950	1967180	15121820
2	512	1406250	33125000	0.0424528	33218120	28228160	4148430	29069690
3	8	0	468750	0	476460	443700	21810	454650
3	16	0	937500	0	943370	868270	51770	891600
3	32	0	1875000	0	1883690	1719330	119840	1763850
3	64	156250	3750000	0.0416667	3813240	3429340	271570	3541670
3	128	312500	7812500	0.04	7943210	7143880	603860	7339350
3	256	781250	15781250	0.049505	15875650	14161430	1311720	14563930
3	512	1406250	31718750	0.044335	31919840	28373850	2766170	29153670
4	8	0	468750	0	471400	444360	16460	454940
4	16	0	1093750	0	979680	912760	38950	940730
4	32	156250	1562500	0.1	1937290	1800570	90020	1847270
4	64	156250	3750000	0.0416667	3864540	3567090	203840	3660700
4	128	312500	7656250	0.0408163	7603300	6964690	452530	7150770
4	256	781250	15625000	0.05	15655360	14198450	983590	14671770
4	512	1562500	32031250	0.0487805	32298480	29344280	2074210	30224270
5	8	0	625000	0	496040	472770	13170	482870
5	16	0	937500	0	951720	895430	31060	920660
5	32	156250	1875000	0.0833333	1923470	1795770	71670	1851800
5	64	156250	3750000	0.0416667	3722610	3478810	162810	3559800
5	128	312500	7500000	0.0416667	7430160	6882220	362030	7068130
5	256	625000	15000000	0.0416667	14906390	13743980	786870	14119520
5	512	1562500	30781250	0.0507614	30874760	28354120	1659370	29215390
6	8	0	468750	0	488150	466810	11110	477040
6	16	0	937500	0	936720	881470	26130	910590
6	32	156250	1875000	0.0833333	1920070	1786380	60210	1859860
6	64	312500	3750000	0.0833333	3846160	3613320	136110	3710050
6	128	312500	7968750	0.0392157	7829220	7331200	301930	7527290
6	256	625000	15156250	0.0412371	15046180	14057880	656250	14389930
6	512	1562500	29687500	0.0526316	29930010	27785260	1383080	28546930

Method 2

Units	Length	PC	HAVOC	Speedup	HAVOC_T	HAVOC_L	HAVOC_C	overhead
1	8	0	468750	0	541370	458100	64610	476760
1	16	156250	937500	0.166667	1074640	891750	154340	920300
1	32	156250	2187500	0.0714286	2255050	1839380	358390	1896660
1	64	312500	4375000	0.0714286	4491800	3568780	814060	3677740
1	128	468750	9531250	0.0491803	9521190	7348980	1810150	7711040
1	256	781250	18750000	0.0416667	19007280	14499770	3934370	15072910
1	512	1562500	37031250	0.0421941	37724560	28337770	8296870	29427690
2	8	0	468750	0	441380	430400	32510	408870
2	16	156250	781250	0.2	888630	866490	77420	811210
2	32	156250	1718750	0.0909091	1759440	1715680	179480	1579960
2	64	156250	3593750	0.0434783	3510330	3422270	407680	3102650
2	128	468750	7343750	0.0638298	7445590	7269220	905800	6539790
2	256	781250	15000000	0.0520833	15961260	15610620	1967970	13993290
2	512	1562500	30000000	0.0520833	30665900	29853020	4149260	26516640
3	8	0	468750	0	447650	436570	21810	425840
3	16	0	937500	0	881650	859220	51770	829880
3	32	156250	1718750	0.0909091	1841500	1796840	119840	1721660
3	64	156250	3593750	0.0434783	3567000	3479960	271570	3295430
3	128	312500	7031250	0.0444444	7269990	7088220	603860	6666130
3	256	1093750	15781250	0.0693069	19407060	18976790	1311720	18095340
3	512	1406250	28906250	0.0486486	29543090	28773810	2766170	26776920
4	8	0	468750	0	444460	433410	16460	428000
4	16	0	937500	0	882560	860570	38950	843610
4	32	156250	1718750	0.0909091	1754860	1711260	90020	1664840
4	64	156250	3750000	0.0416667	3608520	3511380	203840	3404680
4	128	312500	7031250	0.0444444	7233790	7058830	453260	6780530
4	256	781250	14218750	0.0549451	14285830	13936100	984380	13301450
4	512	1562500	29687500	0.0526316	30475210	29710980	2075040	28400170
5	8	0	468750	0	445860	434780	13170	432690
5	16	0	781250	0	924540	902650	31060	893480
5	32	0	1875000	0	1866100	1821820	72250	1793850
5	64	156250	3593750	0.0434783	3565000	3477480	163460	3401540
5	128	312500	7031250	0.0444444	7129110	6862870	362750	6766360
5	256	781250	13906250	0.0561798	15874690	15385950	787660	15087030
5	512	1875000	31718750	0.0591133	36230030	35467620	1660200	34569830
6	8	0	468750	0	467220	455990	11110	456110
6	16	0	1093750	0	1543870	1521870	26130	1517740
6	32	156250	1875000	0.0833333	1907470	1863800	60210	1847260
6	64	156250	3906250	0.04	3964970	3877250	136110	3828860
6	128	312500	7812500	0.04	8319520	8143160	301930	8017590
6	256	625000	14843750	0.0421053	15785760	15298800	656250	15129510
6	512	1406250	28593750	0.0491803	29508290	28786790	1383080	28125210

Bibliografía

[Bar01] Vargas, F.L.; Ribeiro Fagundes, R.D.; Barros Junior, D. A fpga-based viterbi algorithm implementation for speech recognition systems. In *IEEE*, 2001.

[Boa85] IEEE Standards Board, editor. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[Hon01] Acero, Alex; Huang, Xuedong; Hon, Hsiao-Wuen. *Spoken language processing : a guide to theory, algorithm, and system development*. Prentice Hall, 2001.

[Men02a] Mentor Graphics. *HDL Designer Series Tutorials*, 2002.

[Men02b] Mentor Graphics. *LeonardoSpectrum HDL Synthesis Manual*, 2002.

[Men02c] Mentor Graphics. *LeonardoSpectrum User Manual*, 2002.

[Men02d] Mentor Graphics. *Predicting the Output of Finite State Machines*, 2002.

[Par00] Parhami, Behrooz. *Computer Aritmethic, Algorithms and Hardware Designs*. Oxford University Press, 2000.

[Pat03] Hennessy, John L.; Patterson, David A. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd. edition, 2003.

[PLX03] PLX Technology. *PCI 9656BA Data Book*, 2003.

[Rab89] Rabiner, Lawrence R. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, volume 77, pages 257–274, 1989.

[Rus00] Melnikoff, S.J.; James-Roxby, P.B.; Quiqley, S.F.; Russell, M.J. Reconfigurable computing for speech recognition: Preliminary findings. In *Lecture Notes in Computer Science*, number 1896, pages 495–504. Springer-Verlag, 2000.

[Rus02a] Melnikoff, S.J.; Quiqley, S.F.; Russell, M.J. Implementing a simple continuous speech recognition system on an fpga. In *Proceedings of the 10 th Annual IEEE Symposium on Field–Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, 2002.

[Rus02b] Melnikoff, S.J.; Quiqley, S.F.; Russell, M.J. Speech recognition on an fpga using discrete and continuous hidden markov models. In *FPL*, 2002.

[Sam02] Samsung Electronics. *512Kx36/32 & 1Mx18 Pipelined NtRAM*, 2002.

[Sil97] Yun, Hyun-Kyu; Smith, Aaron; Silverman, Harvey. Speech recognition hmm training on reconfigurable parallel processor. In *IEEE*, pages 242–243, 1997.

[Vil98] Terés, Lluís; Olcoz, Serafín; Torroja, Yago; Villar, Eugenio. *VHDL, Lenguaje Estándar de Diseño Electrónico*. McGraw-Hill, 1998.

[Xil00] Xilinx. *Libraries Guide 3.3.06i*, 2000.

[Xil02] Xilinx. *Virtex-II 1.5V Field-Programmable Gate Arrays*, 2002.