Universidad Autnóma Metropolitana
Unidad Azcapotzalco

División de Ciencias Básicas e Ingeniería

Proyecto Terminal en Ingeníera en Computación

Co-Procesador Matemático de Funciones Trigonométricas en un Sistema Embebido Basado en FPGA

Proyecto que presenta:
David García Hernández

para obtener el titulo de:
Ingeniero en Computación

Director de Proyecto:
M. en C. Oscar Alvarado Nava

México, D.F. **Trimestre 10P, julio de 2010**

Resumen

Un coprocesador matemático es un circuito que se añade a un sistema, con el objetivo de asignarle cálculos de alto costo computacional y aumentar la velocidad del procesamiento de datos, éstos representados bajo un determinado estándar, normalmente se utiliza la representación de simple precisión para números de punto flotante llamada en el estÃ¡ndar IEEE 754. El circuito coprocesador es activado por medio de una señal y es colocado en sus puertos de entrada los valores numéricos a procesar, cuando éste termina emite una señal de fin junto con el resultado.

Actualmente se cuenta con bastante información relacionada con los sistemas embebidos también conocidos como sistemas empotrados. Como es de saberse este tipo de sistemas cuenta con un reducido nÃºmero de recursos, (CPU, RAM y sistema de buses), es por eso que se busca implementar algoritmos que requieran de la menor cantidad de recursos computacionales, que sean eficientes y con una precisión aceptable.

El cálculo de funciones trigonométricas es un caso claro de operaciones de alto costo, es por eso que la búsqueda de algoritmos, capaces de calcular dichas funciones, se ha hecho sobre literatura relacionada con el diseño de hardware, aritmética computacional, arquitectura de computadoras y diseño lógico. El algoritmo seleccionado para este proyecto es el algoritmo de CORDIC (Coordinate Rotation Digital Computer), basado en operaciones básicas como la suma y resta, desplazamientos y búsqueda en tablas de datos.

Antes de comenzar directamente con la descripción del coprocesador matemático en hardware, se hace una implementación en lenguaje C, aprovechando las funciones matemáticas que tiene en biblioteca. De a cuerdo con lo observado se describieron circuitos en VHDL (*Very High Speed Integrated Circuit Hardware Description Language*). Finalmente es llevado a un sistema basado en un FPGA y para corroborar su funcionamiento es implementado un algoritmo por software que requiera de las funciones seno y coseno, estas últimas calculadas por hardware.

Agradecimientos

- A la División de Ciencias Básicas e Ingeniería de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco.

- Al Departamento de Electrónica por las facilidades dadas para la realización del proyecto.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivaciones

A medida que se han incrementado las funciones de ciertos dispositivos electrónicos utilizados para el procesamiento digital de señales, éstos se hacen más complejos. Dispositivos que en sus orígenes eran sencillos periféricos con funciones menores a las computadoras en las que se encontraban conectados, ahora son potentes módulos con mayor grado de integración: microprocesadores, memoria, sistema de buses, tablas de datos y hasta su propio sistema operativo.

Con el fin de aligerar la carga de trabajo hecha por un microprocesador central, los dispositivos actuales trabajan en conjunto con el microprocesador ocupándose de ciertas actividades para los que fueron diseñados. Es por eso que ha crecido la demanda de coprocesadores matemáticos para el procesamiento de imágenes, audio, video, simulación y en áreas como la inteligencia artificial y la robótica. Las aplicaciones anteriormente mencionadas requieren de operaciones aritméticas y trigonométricas sobre números de punto flotante; estas últimas son importantes en dichas aplicaciones por que son periódicas y por lo tanto modelan varios procesos naturales con el mismo comportamiento[1] . Además el rango de valores que son utilizados por las aplicaciones multimedia así como sus precisiones, son específicas de la aplicación, siendo necesario desarrollar circuitos coprocesadores especializados, con el propósito de mejorar el desempeño y el ahorro de recursos.

A través de dispositivos programables como los FPGA's[1], es posible implementar circuitos que lleven a cabo el cálculo de operaciones aritméticas y trigonométricas de manera eficiente ya que pueden aprovechar el paralelismo inherente del hardware. Actualmente se cuenta con lenguajes lo suficientemente potentes para descripción de hardware y dotados con funciones almacenadas en biblioteca como los lenguajes de alto nivel, pero hacer uso desmedido de estas funciones, no afecta el hecho de que un

[1] *Field-Programmable Gate Array*, Arreglo de Compuertas Programables en Campo

circuito sea capaz de cumplir la función para lo que fue pensado, pero si incrementa considerablemente la complejidad y el costo del mismo, por eso es necesario de un buen diseño y de la implementación de un buen algoritmo para la resolución de un determinado cálculo.

Existe un conjunto de algoritmos pensados para su desarrollo en hardware, con las características de emplear operaciones sencillas y compuertas lógicas, basados en patrones y reutilización de módulos, así como el manejo de información a nivel de bits. Todos ellos bastante eficientes para la multiplicación, la división, suma, sustracción, raíces, etcétera. Para el cálculo de las funciones seno y coseno se encuentra el llamado algoritmo de CORDIC (*Coordinate Rotation Digital Computer*), basado en operaciones básicas como la suma y resta, desplazamientos y búsqueda en tablas de datos; ampliamente utilizado en trabajos dedicados al desarrollo de coprocesadores matemáticos, en varias ocasiones nombrados Unidades de Punto Flotante, incorporados como periférico de un procesador programado[2][3].

1.2. Objetivos

En el presente trabajo se implementa a nivel de hardware el algoritmo de CORDIC, para que sea ocupado como un periférico coprocesador sobre un FPGA XC2VP30 de la familia Virtex II Pro; con el fin de acelerar el proceso de calcular funciones trigonométricas a nivel de software, sin que se vea afectada la exactitud ofrecida por los lenguajes de alto nivel.

1.2.1. Objetivos particulares

- **Selección de algoritmos adecuados para hardware y para calcular funciones trigonométricas.** De los distintos algoritmos que se encuentran en la literatura, se hizo un análisis de su complejidad matemática, partiendo del entendido que a mayor complejidad, son más los recursos computacionales empleados. Así fue como se logró discernir y optar por la elección del adecuado.

- **Implementación y Análisis del algoritmo en lenguaje C.** Aprovechando el potencial del lenguaje y las funciones almacenadas en biblioteca, se modeló el algoritmo seleccionado para obtener una visión de su estructura y comportamiento así como corroborar la veracidad de los resultados que ofrece, se tomó como punto de comparación los resultados que se consiguen al ocupar las funciones sin() y cos() de la biblioteca math.h del mismo lenguaje. Posteriormente se hizo un análisis de precisión y la exactitud requerida contra la obtenida, también se realizó un pequeño acercamiento al modo en el que opera el mismo algoritmo a nivel de hardware.

- **Desarrollar módulo en hardware de las funciones seno y coseno.** Una de las principales razones por las que es conocido el método de CORDIC, es porque

obtiene simultáneamente los valores de las funciones seno y coseno. Entonces se comenzó con el desarrollo del módulo, que a su vez se sub-dividió en otros más; cada uno diseñado, desarrollado y probado (en simulación) por separado, finalmente se conjuntaron todos en uno sólo, adicionando un bloque más, para controlar y coordinar las funciones de los demás elementos.

- **Incorporar módulos en un sistema mínimo.** Aquí se comenzó a implementar el circuito la tarjeta FPGA.

- **Determinar validez de los resultados.** En primera instancia se corroboró que verdaderamente se hayan obtenido los valores correctos, y que se haya cumplido con el objetivo de acelerar el proceso de cálculo.

- **Desarrollar interfaz Hardware-Software.** Son las adecuaciones tanto en la parte hardware como en la parte software para que haya comunicación y coordinación entre ellas, así como la elaboración de funciones con el mismo fin, también llamadas drivers.

- **Realizar pruebas con algoritmos implementados en software.** Para mostrar el beneficio del coprocesador se implementó un programa por software que requiera en repetidas ocasiones de los cálculos de las funciones seno y coseno para que el coprocesador se encargue de obtener los resultados y disminuir la carga de trabajo que se hace por software y aumentar la velocidad del procesamiento.

El perfil de rendimiento...

1.3. Organización del proyecto

El presente proyecto se encuentra dividido en seis secciones. En el capítulo uno se muestra como funciona el algoritmo de CORDIC y en base al fundamento matemático que se encuentra detrás de éste, se plantean algunas adecuaciones que no irrumpen con los cálculos pero simplifican el modo de operar. En el segundo capítulo se hace un modelado del algoritmo en lenguaje C, a fin de analizarlo y comprobar lo mencionado en el capítulo uno, para comenzar a proponer su diseño orientado a hardware.

En el tercer capitulo se construye el módulo en hardware capaz de obtener las funciones trigonométricas seno y coseno, mostrando las aportaciones de cada uno de sus componentes y cómo es que interactúan entre ellos. Una vez corroborado en simulación de señales, se continúa con la implementación sobre un FPGA y después de volver a corroborar la veracidad de los resultados se realiza la interfaz hardware-software para tener la comunicación entre ambas partes. Dentro del cuarto capítulo se pone a interactuar un programa en software con el coprocesador matemático.

El capitulo cinco consiste exclusivamente de los resultados obtenidos, junto con su respectivo análisis, comentarios y observaciones; en términos de los valores obtenidos de las funciones trigonométricas y los valores obtenidos por el algoritmo que se ocupó como prueba de la interacción hardware-software. También son analizadas otras cantidades como el tiempo de ejecución, velocidad, pulsos de reloj, entre otros. Y en el capitulo 6 están asentadas las colusiones del proyecto y algunas propuestas de continuidad del mismo proyecto y su incorporación en otros a fines relacionados.

Capítulo 2

El Algoritmo de CORDIC

2.1. Introducción

Originalmente fue desarrollado por Jack E. Volder en 1959[4] como una solución digital a los problemas de navegación en tiempo real. John Stephen Walther, en Hewlett-Packard, realizó una generalización del algoritmo aumentando su capacidad para el cálculo de más funciones. Básicamente el algoritmo se basa en la rotación de un vector unitario en un plano cartesiano y la evaluación de la longitud y el ángulo del mismo.

Este algoritmo se basa únicamente realizar sumas, desplazamientos y consultas a tablas para calcular funciones trigonométricas circulares, hiperbólicas y lineales[2]. De acuerdo con el cálculo que se desee obtener es necesario de una pequeña variante del método general, por eso al conjunto de algoritmos orientados a la estimación de una función en particular se les conoce como algoritmos Cordicos[5]; la simplicidad de éstos los hace convenientes para su descripción en hardware. En el presente proyecto se realizarán los cálculos para obtener las funciones trigonométricas circulares seno y coseno.

2.2. Fundamento Matemático

El comportamiento del algoritmo de CORDIC es análogo al de una balanza de aguja, cuando se desea conocer el peso de un objeto la aguja inicialmente se mueve por encima y por de bajo del peso "real" aproximándose cada vez más a éste.Figura[2.1]

Sea v un vector en el plano y suponer que v se ha colocado de tal manera que su punto inicial se encuentra en el origen de un sistema de coordenadas rectangulares, es decir, $x_0, y_0 = (0, 0)$. Las coordenadas (x_1, y_1) del punto terminal de v (denominadas componentes de v) son igual a (0,1), dicho de otro modo $v = (0, 1)$[6].

Dentro del grupo de transformaciones lineales se tiene el siguiente sistema de ecuaciones (2.1) para la rotación de un vector en sentido contrario a las manecillas del reloj a través de un ángulo θ respecto al eje X positivo[6].

Figura 2.1: Balanza de aguja

$$\begin{aligned} x^{'} &= x\cos\theta - y\,\mathrm{sen}\,\theta \\ y^{'} &= y\cos\theta + x\,\mathrm{sen}\,\theta \end{aligned} \tag{2.1}$$

De (2.1) se factoriza $\cos\theta \neq 0$ y ahora se tiene:

$$\begin{aligned} x^{'} &= \cos\theta(x - y\tan\theta) \\ y^{'} &= \cos\theta(y + x\tan\theta) \end{aligned} \tag{2.2}$$

dada la siguiente relación:

$$\cos\theta = \frac{1}{\sqrt{1 + \tan^2\theta}} \tag{2.3}$$

sustituyendo (2.3) en (2.2):

$$\begin{aligned} x' &= \tfrac{1}{\sqrt{1+\tan^2\theta}}(x - y\tan\theta) \\ y' &= \tfrac{1}{\sqrt{1+\tan^2\theta}}(y + x\tan\theta) \end{aligned} \tag{2.4}$$

Para el algoritmo de CORDIC, las rotaciones son remplazadas por pseudorotaciones[7], por la siguiente razón: en una rotación no se ve alterada la magnitud del vector v que se mueve, en cambio una pseudorotación afecta lo afecta en un factor de$\sqrt{1+\tan^2\theta}$, respecto al valor original.

Asumiendo que $x = x_0$, $y = y_0$ y $z = z_0$, para n iteraciones de las rotaciones, se tiene la siguiente expresión (2.5):

$$\begin{aligned} x_n &= x\cos\left(\sum_{i=0}^{n-1}\theta_i\right) - y\,\mathrm{sen}\left(\sum_{i=0}^{n-1}\theta_i\right) \\ y_n &= y\cos\left(\sum_{i=0}^{n-1}\theta_i\right) + x\,\mathrm{sen}\left(\sum_{i=0}^{n-1}\theta_i\right) \\ z_n &= z - \left(\sum_{i=0}^{n-1}\theta_i\right) \end{aligned} \tag{2.5}$$

En cambio una pseudorotación se ve como en (2.6):

$$\begin{aligned} x''_n &= \left(x\cos\left(\sum_{i=0}^{n-1}\theta_i\right) - y\,\mathrm{sen}\left(\sum_{i=0}^{n-1}\theta_i\right)\right)\prod_{i=0}^{n-1}\sqrt{1+\tan^2\theta_i} \\ y''_n &= \left(y\cos\left(\sum_{i=0}^{n-1}\theta_i\right) + x\,\mathrm{sen}\left(\sum_{i=0}^{n-1}\theta_i\right)\right)\prod_{i=0}^{n-1}\sqrt{1+\tan^2\theta_i} \\ z''_n &= z - \left(\sum_{i=0}^{n-1}\theta_i\right) \end{aligned} \tag{2.6}$$

A las ecuaciones (2.5) y (2.6) se les ha incluido una componente z, no se interprete como una dimensión más del espacio vectorial; z es más bien un acumulador angular, es éste donde se carga el valor del ángulo del cuál se desea conocer las funciones.

Entonces, retomando que una rotación se expresa como en (2.4) y a ésta se le multiplica por un factor de$\sqrt{1+\tan^2\theta}$ para obtener una pseudorotación, entonces es posible expresarla en términos de las ecuaciones.

$$\begin{aligned} x' &= x - y\tan\theta \\ y' &= y + x\tan\theta \end{aligned} \tag{2.7}$$

Dada la siguiente relación

$$\tan\theta = \pm 2^{-i}, i \in N \tag{2.8}$$

Entonces la multiplicación por la tangente ahora se reduce a una operación de desplazamientos a la derecha. Véase el ejemplo de la tabla 2.1. El objetivo del algorit-

	base 10		base 2	
	numero = 14		numero = 1110	
i	2^{-i}	numero*2^{-i}	$2^{-}i$	numero*2^{-i}
0	1.000	14.000	1.000	1110.000
1	0.500	7.000	0.100	111.000
2	0.250	3.500	0.010	11.100
3	0.125	1.750	0.001	1.110

Tabla 2.1: Desplazamientos a la derecha de un número multiplicano por 2^{-i}

mo es realizar un número n de pseudorotaciones de ángulos cada vez menores, con el propósito de que las componentes del vector v se aproximen al valor del seno (para la componente en y) y el coseno (para la componente en x), utilizando las coordenadas obtenidas en la pseudorotación $n-1$.Figura[2.2]

Ahora, consideando que que $\cos\theta = \cos(-\theta)$, (por ser una función par[1]) y sustitutendo (2.8) en (2.2), una iteración pude verse de la siguiente manera:

$$\begin{aligned} x_{i+1} &= k_i(x_i - y_i d_i 2^{-i}) \\ y_{i+1} &= k_i(y_i + x_i d_i 2^{-i}) \end{aligned} \tag{2.9}$$

de donde k se obtiene al sustituir (2.8) en (2.3). veáse en (2.10) y $d = \pm 1$, dependiendo del sentido de la rotación.

$$k = \frac{1}{\sqrt{1+2^{-2i}}} \tag{2.10}$$

El factor k_i puede aplicarse al final del proceso como una constante k_n, definida como sigue:

$$k_n = \prod_{i=0}^{n+1} k_i \tag{2.11}$$

Para determinar el sentido de la pseudorotación (como el movimiento de la aguja en la balanza), el algoritmo se auxilia del acumulador angular z en cada paso de la

[1]Decimos que una función es par si $f(-x) = f(x)$. En este caso la gráfica es simétrica respecto al eje Y.[1]

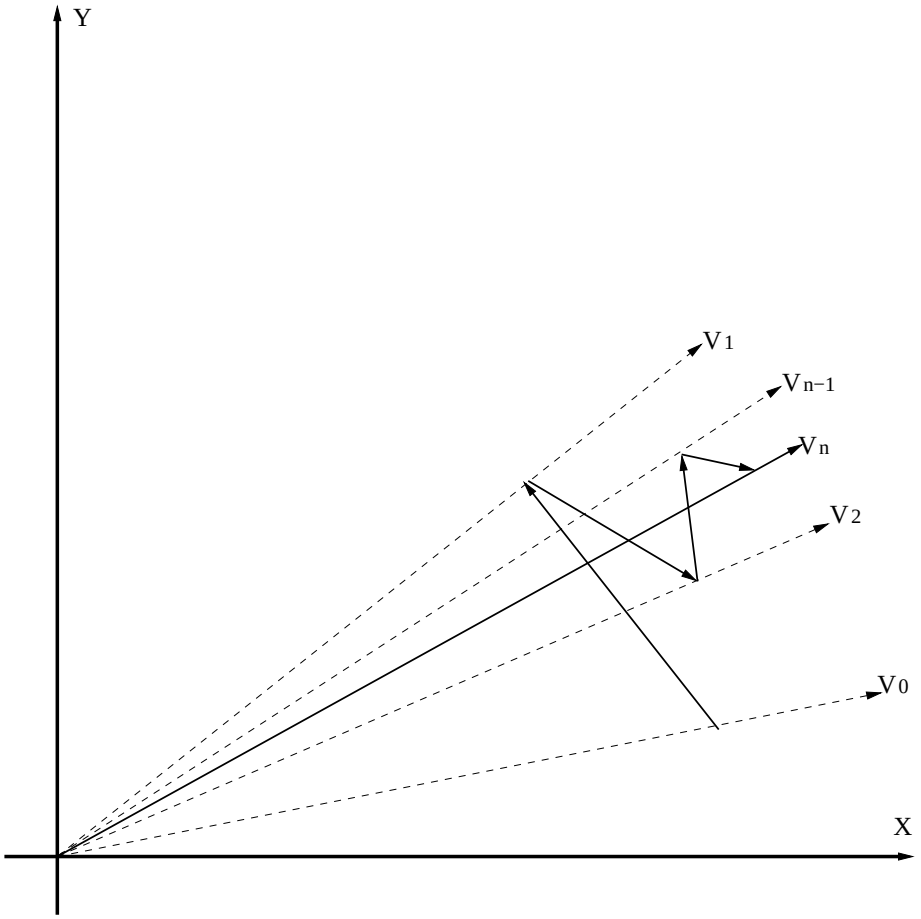

Figura 2.2: Rotaciones del vector v

iteración, forzando a que z se haga cero; entonces d será 1 si z es mayor que cero, y -1 cuando z sea mayor que cero. Finalmente z queda expresado como sigue:(2.12):

$$z_{i+1} = z_i - d_i \arctg(2^{-i}) \tag{2.12}$$

Los valores dados por $\arctg(2^{-i})$ se pueden almacenar en una tabla, como éstos son obtenidos de la función inversa[2] a la tangente (arco tangente), en consecuencia su rango (dominio de la tangente) es de $-\frac{\Pi}{2}$ a $\frac{\Pi}{2}$ y el alcance del algoritmo de CORDIC se restringe al mismo rango.

A esta forma de operar del algoritmo se le conoce como *modo de rotación*. En resumen en el modo de rotación la componente z se inicializa con el ángulo del cual se desea conocer el seno y el coseno; el sentido de la rotación se determina con el propósito de que minimizar la magnitud del ángulo acumulado en z, aproximándose éste a cero en cada iteración.El sistema de ecuaciones para este modo es el siguiente:

[2]La función inversa de f denotada f^{-1},se define como: $f^{-1}(y) = x <=> y = f(x)$. El dominio de f^{-1} es el rango de f y el rango de f^{-1} es el dominio de f[1]

$$\begin{aligned} x_{i+1} &= x_i - y_i d_i 2^{-i} \\ y_{i+1} &= y_i + x_i d_i 2^{-i} \\ z_{i+1} &= z_i - d_i \operatorname{arc\,tg}(2^{-i}) \end{aligned} \tag{2.13}$$

donde:

$$d = \begin{cases} -1 & , si \quad z_i < 0 \\ 1 & , si \quad z_i \geq 0 \end{cases}$$

Partiendo del sistema de ecuaciones (2.6) y de que z tiende a cero, entonces $\sum_{i=0}^{n-1} \theta_i = z$ para $x = x_0$, $y = y_0$ y $z = z_0$ [8] se tiene (2.14):

$$\begin{aligned} x_n &= A_n(x_0 \cos z_0 - y_0 \operatorname{sen} z_0) \\ y_n &= A_n(y_0 \cos z_0 + x_0 \operatorname{sen} z_0) \\ z_n &= 0 \\ A_n &= \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} \end{aligned} \tag{2.14}$$

2.3. Cálculo del Seno y el Coseno

Si se inicializa $y_0 = 0$, del sistema de ecuaciones (2.14) entonces se obtiene (2.14)[8]:

$$\begin{aligned} x_n &= A_n x_0 \cos z_0 \\ y_n &= A_n x_0 \operatorname{sen} z_0 \\ z_n &= 0 \\ A_n &= \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} \end{aligned} \tag{2.15}$$

Si además $x_0 = \frac{1}{A_n}$ de (2.15) entonces(2.16):

$$\begin{aligned} x_n &= \cos z_0 \\ y_n &= \operatorname{sen} z_0 \\ z_n &= 0 \end{aligned} \tag{2.16}$$

Sea z_0 igual al ángulo inicial, véase que para n iteraciones se obtiene simultáneamente los valores del coseno y del seno para las componente x y y respectivamente, y z "igual a cero".

2.4. Conclusiones

De este algoritmo y en general para los métodos multipaso, se puede aprovechar la característica de que las funciones para calcular los valores de las componentes x, y y z son evaluadas solamente una vez por cada iteración, los valores iniciales son reemplazados por los que se obtuvieron en el actual paso y serán tomados para volver a evaluar las funciones en el siguiente, por lo tanto el ahorro en tiempo es bastante [9].

Las consideraciones a tener son la propagación del error de redondeo, la cantidad de iteraciones necesarias para obtener resultados aceptables, la codificación y programación del método. De las anteriores se derivan algunas otras que se muestran a lo largo del proyecto, sin embargo es necesario tener una visión general para realizar las adecuaciones necesarias.

En los capítulos posteriores se realiza la implementación y el análisis tanto a nivel de software como en hardware, basándose en el fundamento matemático que aquí se da. El capitulo 3 es importante para corroborar lo que la teoría indica y también es útil para mostrar un ejemplo del algoritmo de CORDIC aplicado para un ángulo en específico.

Capítulo 3

Análisis del algoritmo de CORDIC en lenguaje C

3.1. Introducción

Cuando se trabaja por primera vez con un determinado algoritmo, es importante hacer pruebas de escritorio, con el objetivo de corroborar su forma de operar y la veracidad de los resultados que arroja. En este capítulo se muestra la codificación del algoritmo de CORDIC, escrito en lenguaje C y compilado con gcc; para registrar el tiempo de ejecución se ejecutó el programa fuera de un ambiente gráfico, y con el mínimo de procesos activos, para minimizar el uso de recursos en el sistema.

Después de que se ha comprendido la estructura del método, es válido comenzar a jugar con sus componentes, a fin de observar variaciones, comprobar expectativas y comprender aún más su funcionamiento. Esto ayuda ha encontrar nuevas alternativas para solucionar un mismo problema y proponer mejoras.

En este caso, se busca minimizar la complejidad de las operaciones matemáticas involucradas con los cálculos, las adaptaciones mostradas requieren de un gasto mayor a nivel de software pero el benefició es notable cuando se hace la implementación en hardware.

3.2. Análisis funcional

Inicialmente se muestra la codificación del algoritmo en lenguaje C aprovechando las funciones contenidas en la biblioteca *math.h*, el código completo se muestra en el apéndice A.

Los miembros de la estructura *vector* son las componentes x, y y z. como se indica en [3] el vector se coloca inicialmente sobre el eje X y la componente z se inicializa con el valor del ángulo dado (líneas 28 y 31).

El siguiente bloque pertenece directamente al comportamiento del algoritmo:

```
36 for (i = 0; i < iteraciones; i++)
37 {
38 if (vecRotar.z >= 0)
39 d = 1;
40 else d = -1;
41
42 k = 1/(sqrt(1+pow(2,-2*i)));
43 Pik *= k;
44
45 x = vecRotar.x;
46 vecRotar.x = vecRotar.x - vecRotar.y*d*pow(2,-i);
47 vecRotar.y = vecRotar.y + x*d*pow(2,-i);
48 vecRotar.z = vecRotar.z - d*atan(pow(2,-i));
49
50 printf("\n | %2d | X: %+-2.10f | Y: %+-2.10f | Z: %+-2.10f |
51    k: %+-2.10f | Pik: %+-2.10f |"
52 ,i, vecRotar.x*Pik, vecRotar.y*Pik, vecRotar.z, k, Pik);
53 }
```

De acuerdo con el sistema de ecuaciones (2.13), d se define en función de z, esto se hace con la estructura de control *if* de las líneas 38 - 40.

Y las componentes del vector se calculan tal como en (2.13) en las líneas (46 - 48), nótese que en la línea 45 se respalda el valor de x, eso es necesario porque en cada iteración y necesita ser calculada en función del valor de x calculado en el paso *i-1*.

Finalmente, es necesario calcular k y la sucesión de sus productos (Pik) como se indica en (2.11) (líneas 42 - 43).

Dentro de la función *prinf* se hace el producto que caracteriza a (2.9) como una pseudorotación.

grados	0^o	30^o	45^o	60^o	75^o	90^o
radianes	0	$\Pi/6$	$\Pi/4$	$\Pi/3$	$5\Pi/12$	$\Pi/2$
	0	0.52360	0.78540	1.04720	1.30900	1.57080

Tabla 3.1: Ángulos en grados y radianes

Obsérvese que el tipo de dato que se maneja es *float*, esto es por que el coprocesador opera con el estándar IEEE754 (32 bits).

A continuación se prueba el desempeño del algoritmo. Los ángulos dados en la tabla son utilizados, en adelante, para realizar pruebas. Estos valores se han calculado con un sencillo programa en lenguaje C, dada la ecuación (3.1)[1] [11], y corroborados con [11].

$$1^o = \frac{\Pi}{180} rad \tag{3.1}$$

Evaluar con $5\pi/12$, para 40 iteraciones. De lo anterior, se muestra en la tabla3.2 los resultados obtenidos. Para corroborar los resultados obtenidos se hace el cálculo del mismo ángulo con un sencillo programa en lenguaje C que utiliza las funciones sin() y cos() de la biblioteca *math.h* y se consiguieron las cantidades de 0.2588160634 para el coseno y 0.2588160634 para el seno en un tiempo real de 0.004s.

Obsérvese lo siguiente:

- El algoritmo converge para x = cos a partir de la iteración 36 y para y = sen a partir de la iteración 32.
- tomando como valor real lo calculado con *math.h* y como aproximado el obtenido en la iteración 40 de CORDIC, de la ecuación (3.2)[9], el error absoluto es de 0.000000052 y 0.000000046 para el coseno y el seno respectivamente.
- Se ha comprobado que al ángulo inicialmente almacenado en la componente z se aproxima a cero en cada paso.
- El valor de Pik se vuelve una constante a partir de la iteración 11.
- El tiempo real de ejecución oscila entre los 0.010s y 0.012s, tres veces más lento que el programa que se sirve de las funciones trigonométricas de biblioteca.

$$|valor \quad real - aproximado| \tag{3.2}$$

Resultados similares se observaron para los ángulos de prueba de la Tabla 3.1, pero ninguno rebasa el converger en menos de 40 iteraciones.

	x	y	z	k	Pik
i	+1.0000000000	+0.0000000000	+1.3090000153		
0	+0.7071067691	+0.7071067691	+0.5236018519	+0.7071067691	+0.7071067691
1	+0.3162277639	+0.9486832917	+0.0599542429	+0.8944271803	+0.6324555278
2	+0.0766965002	+0.9970545024	-0.1850244203	+0.9701424837	+0.6135720015
3	+0.1997736264	+0.9798420724	-0.0606694257	+0.9922778606	+0.6088339090
4	+0.2605054524	+0.9654723714	+0.0017493843	+0.9980525970	+0.6076482534
5	+0.2302220599	+0.9731381379	-0.0294904492	+0.9995120764	+0.6073517799
6	+0.2453973804	+0.9694225524	-0.0138667205	+0.9998779297	+0.6072776318
7	+0.2529632722	+0.9674758524	-0.0060543795	+0.9999694824	+0.6072590947
8	+0.2567405092	+0.9664803152	-0.0021481493	+0.9999923706	+0.6072544456
9	+0.2586276837	+0.9659770675	-0.0001950268	+0.9999980927	+0.6072533131
10	+0.2595708933	+0.9657240274	+0.0007815354	+0.9999995232	+0.6072530150
11	+0.2590993229	+0.9658506762	+0.0002932542	+0.9999998808	+0.6072529554
12	+0.2588635196	+0.9659139329	+0.0000491135	+1.0000000000	+0.6072529554
13	+0.2587456101	+0.9659455324	-0.0000729568	+1.0000000000	+0.6072529554
14	+0.2588045668	+0.9659297398	-0.0000119216	+1.0000000000	+0.6072529554
15	+0.2588340446	+0.9659218418	+0.0000185960	+1.0000000000	+0.6072529554
16	+0.2588193058	+0.9659257912	+0.0000033372	+1.0000000000	+0.6072529554
17	+0.2588119364	+0.9659277659	-0.0000042922	+1.0000000000	+0.6072529554
18	+0.2588156211	+0.9659267786	-0.0000004775	+1.0000000000	+0.6072529554
19	+0.2588174635	+0.9659262849	+0.0000014298	+1.0000000000	+0.6072529554
20	+0.2588165423	+0.9659265318	+0.0000004761	+1.0000000000	+0.6072529554
21	+0.2588160817	+0.9659266552	-0.0000000007	+1.0000000000	+0.6072529554
22	+0.2588163120	+0.9659265935	+0.0000002377	+1.0000000000	+0.6072529554
23	+0.2588161968	+0.9659266243	+0.0000001185	+1.0000000000	+0.6072529554
24	+0.2588161393	+0.9659266398	+0.0000000589	+1.0000000000	+0.6072529554
25	+0.2588161105	+0.9659266475	+0.0000000291	+1.0000000000	+0.6072529554
26	+0.2588160961	+0.9659266513	+0.0000000142	+1.0000000000	+0.6072529554
27	+0.2588160889	+0.9659266533	+0.0000000068	+1.0000000000	+0.6072529554
28	+0.2588160853	+0.9659266542	+0.0000000030	+1.0000000000	+0.6072529554
29	+0.2588160835	+0.9659266547	+0.0000000012	+1.0000000000	+0.6072529554
30	+0.2588160826	+0.9659266549	+0.0000000002	+1.0000000000	+0.6072529554
31	+0.2588160821	+0.9659266551	-0.0000000002	+1.0000000000	+0.6072529554
32	+0.2588160824	+0.9659266550	+0.0000000000	+1.0000000000	+0.6072529554
33	+0.2588160823	+0.9659266550	-0.0000000001	+1.0000000000	+0.6072529554
34	+0.2588160823	+0.9659266550	-0.0000000001	+1.0000000000	+0.6072529554
35	+0.2588160823	+0.9659266550	-0.0000000000	+1.0000000000	+0.6072529554
36	+0.2588160824	+0.9659266550	-0.0000000000	+1.0000000000	+0.6072529554
37	+0.2588160824	+0.9659266550	-0.0000000000	+1.0000000000	+0.6072529554
38	+0.2588160824	+0.9659266550	+0.0000000000	+1.0000000000	+0.6072529554
39	+0.2588160824	+0.9659266550	-0.0000000000	+1.0000000000	+0.6072529554

Tabla 3.2: iteraciones para calcular el coseno y el seno del ángulo: 1.30900

i	$\arctg(2^{-i})$:	i	$\arctg(2^{-i})$:
0	0.7853981633974	20	0.0000009536743
1	0.4636476090008	21	0.0000004768372
2	0.2449786631269	22	0.0000002384186
3	0.1243549945468	23	0.0000001192093
4	0.0624188099960	24	0.0000000596046
5	0.0312398334303	25	0.0000000298023
6	0.0156237286205	26	0.0000000149012
7	0.0078123410601	27	0.0000000074506
8	0.0039062301320	28	0.0000000037253
9	0.0019531225165	29	0.0000000018626
10	0.0009765621896	30	0.0000000009313
11	0.0004882812112	31	0.0000000004657
12	0.0002441406201	32	0.0000000002328
13	0.0001220703119	33	0.0000000001164
14	0.0000610351562	34	0.0000000000582
15	0.0000305175781	35	0.0000000000291
16	0.0000152587891	36	0.0000000000146
17	0.0000076293945	37	0.0000000000073
18	0.0000038146973	38	0.0000000000036
19	0.0000019073486	39	0.0000000000018

Tabla 3.3: Valores de $\arctg(2^{-i})$: para 40 iteraciones con 13 cifras decimales

3.3. Adecuaciones y análisis de resultados

Acorde con la implementación hecha en la sección anterior y con el fundamento matemático del capitulo2, se hicieron una serie de modificaciones al código original, y fueron probadas una a una. A continuación se enlistan:

- Se toma como una constante a PiK = 0.6072529554, obtenida a partir de la onceava iteración (tabla3.2), ahora los cálculos hechos en las líneas 42-43 se suprimen, véase lo importante que es este cambio, el ahorro de las operaciones raíz cuadrada, división y multiplicación se pueden entender como un ahorro de recursos computacionales.

- Del sistema de ecuaciones (2.16) si se inicializa x con Pik constante, y $y = 0$, entonces se han eliminado los productos hechos en la línea 52.

- El cálculo de $atan(2^{-i})$ de la línea 48, se hizo por separado con una precisión de 13 dígitos decimales y los valores fueron almacenados en un arreglo de tipo *float* con cuarenta localidades para la misma cantidad de iteraciones (Tabla3.3), con el propósito de simular la tabla de búsqueda .

3.3.1. Observaciones generales

El error absoluto respecto a los valores obtenidos con la biblioteca *math.h*, ahora es de 0.000000089 para el coseno y de 0.000000119 para el seno, con un tiempo de ejecución que oscila entre los 0.009s y 0.010s, éste es más rapido que la codificación anterior (Tabla3.4).

Con los resientes cambios es posible notar que el error a cambiado muy poco, además de que la precisión sigue oscilando entre los seis y siete números decimales.

El cálculo de 2^{-i} de las líneas 46-47 se ha dejado para la implementación en hardware, como ya se había dicho, el producto de este número con otro, implica el desplazamiento de bits a la derecha, a pesar de ser ésto cierto, dado el estándar IEEE754, esta afirmación se interpreta de manera distinta. En el capítulo siguiente se demuestra como se resuelve esta situación así como la manera de realiza la multiplicación con d (líneas 46-48).

Nótese en el arreglo de arco-tangentes que el valor de los números almacenados decrecen conforme se va recorriendo la tabla, y el dato almacenado en la posición 40, es apenas de dos decimales distintos de cero, es por eso que se escogió la longitud de 13 dígitos decimales, para que se tuvieran un efecto significativo en cada iteración.

Finalmente el código queda como se muestra en el apéndice B.

3.4. Conclusiones

Prácticamente se eliminaron las operaciones costosas y se demostró que la eficiencia del método no se ha perdido, pensar entre esta última implementación y la descripción en hardware es prácticamente un paso.

Lo siguiente a este capitulo de análisis, consiste en desmembrar la estructura del programa para comenzar a modular los procesos que lo componen, hacer su correspondiente descripción en hardware y una vez definidas cada una de las entidades, se realiza la integración en un solo circuito.

El proceso del presente capitulo fue de gran ayuda para tener una visión del algoritmo y generar una perspectiva del diseño en hardware. Además los resultados son de utilidad para continuar con las estimaciones de error absoluto.

	x	y	z
i	+0.6072529554	+0.0000000000	+1.3090000153
0	+0.6072529554	+0.6072529554	+0.5236018300
1	+0.3036264777	+0.9108794332	+0.0599542223
2	+0.0759066194	+0.9867860675	-0.1850244403
3	+0.1992548704	+0.9772977233	-0.0606694445
4	+0.2603359818	+0.9648442864	+0.0017493655
5	+0.2301845998	+0.9729797840	-0.0294904672
6	+0.2453874052	+0.9693831205	-0.0138667384
7	+0.2529607117	+0.9674660563	-0.0060543972
8	+0.2567398846	+0.9664779305	-0.0021481670
9	+0.2586275339	+0.9659764767	-0.0001950445
10	+0.2595708668	+0.9657239318	+0.0007815177
11	+0.2590993345	+0.9658506513	+0.0002932365
12	+0.2588635385	+0.9659138918	+0.0000490959
13	+0.2587456405	+0.9659454823	-0.0000729744
14	+0.2588045895	+0.9659296870	-0.0000119393
15	+0.2588340640	+0.9659217596	+0.0000185783
16	+0.2588193119	+0.9659256935	+0.0000033195
17	+0.2588119507	+0.9659276605	-0.0000043099
18	+0.2588156462	+0.9659266472	-0.0000004952
19	+0.2588174939	+0.9659261703	+0.0000014122
20	+0.2588165700	+0.9659264088	+0.0000004585
21	+0.2588161230	+0.9659265280	-0.0000000184
22	+0.2588163614	+0.9659264684	+0.0000002201
23	+0.2588162422	+0.9659265280	+0.0000001009
24	+0.2588161826	+0.9659265280	+0.0000000412
25	+0.2588161528	+0.9659265280	+0.0000000114
26	+0.2588161528	+0.9659265280	-0.0000000035
27	+0.2588161528	+0.9659265280	+0.0000000040
28	+0.2588161528	+0.9659265280	+0.0000000003
29	+0.2588161528	+0.9659265280	-0.0000000016
30	+0.2588161528	+0.9659265280	-0.0000000007
31	+0.2588161528	+0.9659265280	-0.0000000002
32	+0.2588161528	+0.9659265280	+0.0000000000
33	+0.2588161528	+0.9659265280	-0.0000000001
34	+0.2588161528	+0.9659265280	-0.0000000000
35	+0.2588161528	+0.9659265280	+0.0000000000
36	+0.2588161528	+0.9659265280	-0.0000000000
37	+0.2588161528	+0.9659265280	+0.0000000000
38	+0.2588161528	+0.9659265280	-0.0000000000
39	+0.2588161528	+0.9659265280	-0.0000000000

Tabla 3.4: coseno y el seno del ángulo: 1.30900

Capítulo 4

Descripción en hardware del algoritmo de CORDIC

4.1. Introducción

El algoritmo de CORDIC se expresa por un sistema de tres ecuaciones, a pesar de su simplicidad, la descripción del método se ha hecho en distintos módulos independientes, capaces de realizar operaciones menores contenidas dentro de la suma final de dos parámetros por cada ecuación.

En el presente capítulo se muestra la integración de dichos componentes así como la coordinación de los mismos. Se comienza con una descripción muy superficial del circuito y posteriormente se va detallando los componentes con sus respectivas señales de control y procesos; algunos secuenciales y otros combinacionales.

Como ya se ha mencionado el estándar para representar números de punto flotante que se utiliza en el presente proyecto es *IEEE754*, por tal motivo la los datos son separados en signo, exponente y mantiza. Para tener mayor exactitud la mantiza se extendió de 23 bits a 42, únicamente dentro del módulo encargado de realizar la suma.

4.2. Descripción general

El circuito coprocesador matemático esta compuesto por módulos secuenciales y combinacionales; el pulso de reloj (**clk**) requerido es un puerto de entrada del circuito, una señal de reset **rst** para reiniciar contador y borrar registros, para seleccionar la función trigonométrica se ocupa la señal **func** (0 para seleccionar el coseno y 1 para el seno) y la señal de inicio **ini** para que el circuito comience a calcular. Finalmente se tiene un puerto de entrada de 32 bits para ingresar el ángulo en radianes con el estándar ieee754, y las salidas son dos, un resultado con el mismo estándar y una señal de terminado **end**.Figura4.1

La arquitectura utilizada para este circuito es *Bit-Pralela Iterativa*, por la principal característica que esta tiene de reutilizar el mismo hardware para n iteraciones, para ello es requerido un módulo de control, que coordine el flujo de la información y el comportamiento del resto de los módulos. El modelo de diseño es *Top-Dow*, consiste en comenzar de una entidad vista como una caja negra capaz de recibir, procesar y generar datos, después ésta será dividida en otros módulos independientes de menor jerarquía, también es posible continuar la división de los módulos en otros más sencillos tanto como sea necesario.

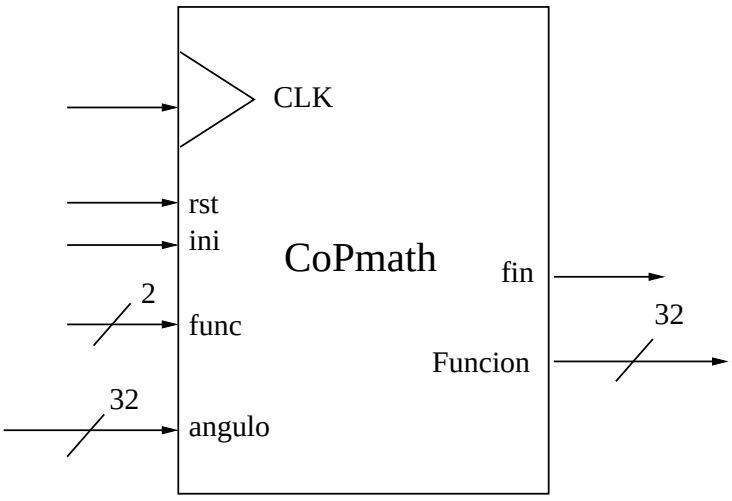

Figura 4.1: Circuito coprocesador matemático

4.2.1. Trayectoria de datos

La ruta de datos se encuentra dividida en tres bloques: multiplexor, memoria y componentes Figura4.2. Posteriormente se describe cada uno de los componentes del coprocesador; el menos complejo es el **MUX PRINCIPAL** con la función de permitir que sea leído el ángulo inicial sólo una vez, junto con las constantes almacenadas en los registros **xini** y **yini**.

Dado que el módulo de **componentes** tiene un retardo de cálculo, éste le avisará al multiplexor el momento en que los datos que le envía son los finales, por medio de la señal **done** para que éstos sean retenidos, por parte del CONTROL recibirá la indicación del momento en el pueda leer los datos generados a partir de la primera iteración **(sel)**. La unidad de **memoria** recibe los bits de cada componente sin proceso alguno, por parte del **mux principalL**.

En la Figura4.3 se detalla el bloque de **memoria** y **componentes**.

Memoria. Aquí se tiene encapsulado una memoria ROM con las constantes de $atan(2^{-i})$ para cuarenta iteraciones. El **CONTORL** se encarga de llevar el conteo de cada iteración, ese mismo dato es aprovechado para extraer de la **ROM** el valor almacenado correspondiente a la iteración actual.

Como se refleja en las líneas 64 - 67 del código fuente del apéndice B, las ecuaciones requieren del producto por el valor d, y desplazamientos a la derecha, dichos procesos se encuentran almacenados en una ALU junto con otras operaciones básicas de sumador, convertir vector de bits a entero y ajuste de mantiza. Pero es en la unidad de **registros** donde se hace la invocación del desplazamiento; para hacer esta operación únicamente se requiere restar al exponente el número que tenga el contador de iteraciones, y la multiplicación por d consiste únicamente de la operación lógica *xor* del bit de signo de la componente con el bit de signo de z, recuérdese que d se define en función del signo de z.

Entonces las salidas de este bloque son las tres componentes x, y, z exactamente con el mismo valor de entrada, y las mismas con sus respectivos desplazamientos y multiplicación por d. Otra salida de este modulo es el arco tangente tomado de la **ROM** también multiplicado por el valor de signo.

Componentes.Como su nombre lo indica, aquí es donde se calcula x, y, z: en su interior se encuentra el módulo de **Cordic**, en éste ingresan los paráametros de las ecuaciones ya formateados, únicamente para que sean sumados. El calculos de las tres componentes se hace en un sólo pulso de reloj auxiliandose de **S/R ieee754**, para mandarle los operandos organizados como el sistema de ecuaciones lo sugiere.

Como el algoritmo suma números de punto flotante, requiere de varias etapas[7], entonces el módulo **S/R ieee754** manda una señal de hecho (**done**)cuando termina. En la Figura 4.4 se muestra dicho proceso, de arriba hacia abajo:

- Ingresan los dos parámetros a sumarse
- Se restan los exponentes, y el menor se ajustará al mayor
- En función del sigo y el tamaño de los mismos, se realiza una suma ó una sustracción.

- El resultado es normalizado de tal modo que el bit más significativo puesto en uno se recorra hasta una posición después del tamaño permitido para la mantiza (23 bits para simple precisión).

- Los desplazamientos anteriores afectan al exponente, por lo tanto es necesario decrementarlo el mismo número de recorridos hechos a la derecha.

- Para realizar el redondeo, en general consiste en hacer otra suma con una constante. Por ejemplo: para redondear hacia al entero más cercano se suma 0.5.

- Probablemente la suma del paso anterior requiera una nueva normalización

- Finalmente se empaqueta el resultado en signo, exponente y mantiza.

De las etapas mencionadas arriba, aqui no se realiza el redondeo ni la segunda normalización

Cuando se hayan cumplido las cuarenta iteraciones, se selecciona el valor de la función solicitada.

En la Figura4.5 se muestra el esquema en hardware para el algoritmo de CORDIC, obsérvese que los elementos de la figura se encuentran distribuidas en el circuito entre la unidad de **memoria** y de **componetes**.

Control. El diagrama de estados que se muestra en la Figura4.6 describe la secuencia de las señales de este módulo, el circuito se encuentra en estado de espera hasta que es activado con la señal de **en**, cuando esto ocurra, le indicará al proceso de carga **ld** que tome los datos que se encuentran en los puertos de entrada, en seguida se ejecuta el proceso **format** que le da formato a los datos (operación hecha por el módulo de **registros**), en seguida se realizará la correspondiente suma **sum** y después, se comunica con **ld** para que ahora cargue los resultados calculados por **sum**, éste proceso se hará por cuarenta iteraciones. Cunado haya terminado, el control seleccionará la función que se le haya solicitado (**selfunc**),cabe recordar que el algoritmo calcula ambas funciones simultáneamente; hecho esto regresa al estdo de espera.

4.2.2. Simulación

De la simulación hecha en la Figura4.7 se puede observar los últimos pasos para obtener el coseno (señal sen_cos), del ángulo 1.30900 (señal z_ini), véase como las componentes x,y,z se calculan en varios pasos, y cuando la suma concluye lo indica con la señal *done*. Una vez que se ha llegado a la iteración final, la señal de *terminado* se pone en *1* y se detiene todo calculo. El resultado se observa en *result_fn*.

Ángulo	Seno			Coseno		
	CORDIC	Lenguaje C	error abs	CORDIC	Lenguaje C	error abs
Π/6	0,5000001490	0,5000010604	0,0000009114	0,8660240173	0,8660247916	0,0000007742
Π/4	0,7071069479	0,7071080799	0,0000011320	0,7071044445	0,7071054825	0,0000010380
Π/3	0,8660255075	0,8660266282	0,0000011207	0,4999974072	0,4999978793	0,0000004721
5Π/12	0,9693830609	0,9659266185	0,0034564424	0,2453873754	0,2588160884	0,0134287130
Π/2	0,9999588728	1,0000000000	0,0000411272	-0,0007808208	-0,0000036732	0,0007844941

Tabla 4.1: Cálculos de coseno y seno para distintos ángulos

En la tabla se muestran los cálculos para cinco ángulos hechos en hardware con el algoritmo de CORDIC y con el lenguaje C. Véase que los valores convergen entre los cuatro bits decimales de exactitud, y el valor deja de ser confiable al calcular exactamente uno de los limites mencionados en el marco teórico. (Π/2).

4.3. Conclusión

De los algoritmos pensados para su desarrollo en hardware, es posible aprovecharlos con algunas adecuaciones a preferencia del desarrollador del circuito, esto se logra aprovechando la modularidad de los procesos que involucran los métodos. Así mismo este coprocesador puede ser modificado y adaptado para que trabaje en conjunto con otros sistemas hardware o software.

Si se desea trabajar sobre la precisión y exactitud, ya es de suponerse en que parte hay que estudiar, o si se agregarán más funciones trigonométricas es necesario incorporar un algoritmo que divida en punto flotante y que aproveche lo que el coprocesador ya calcula.

De la figura se muestra que el gasto es de 5600ns para tener un resultado, lo que en software se tarda a razón de milisegundos, aún es muy pronto estimar diferencias en tiempos por que el intervalo de pulso de reloj que se le a dado (10ns) ha sido únicamente para probar resultados y funcionamiento.

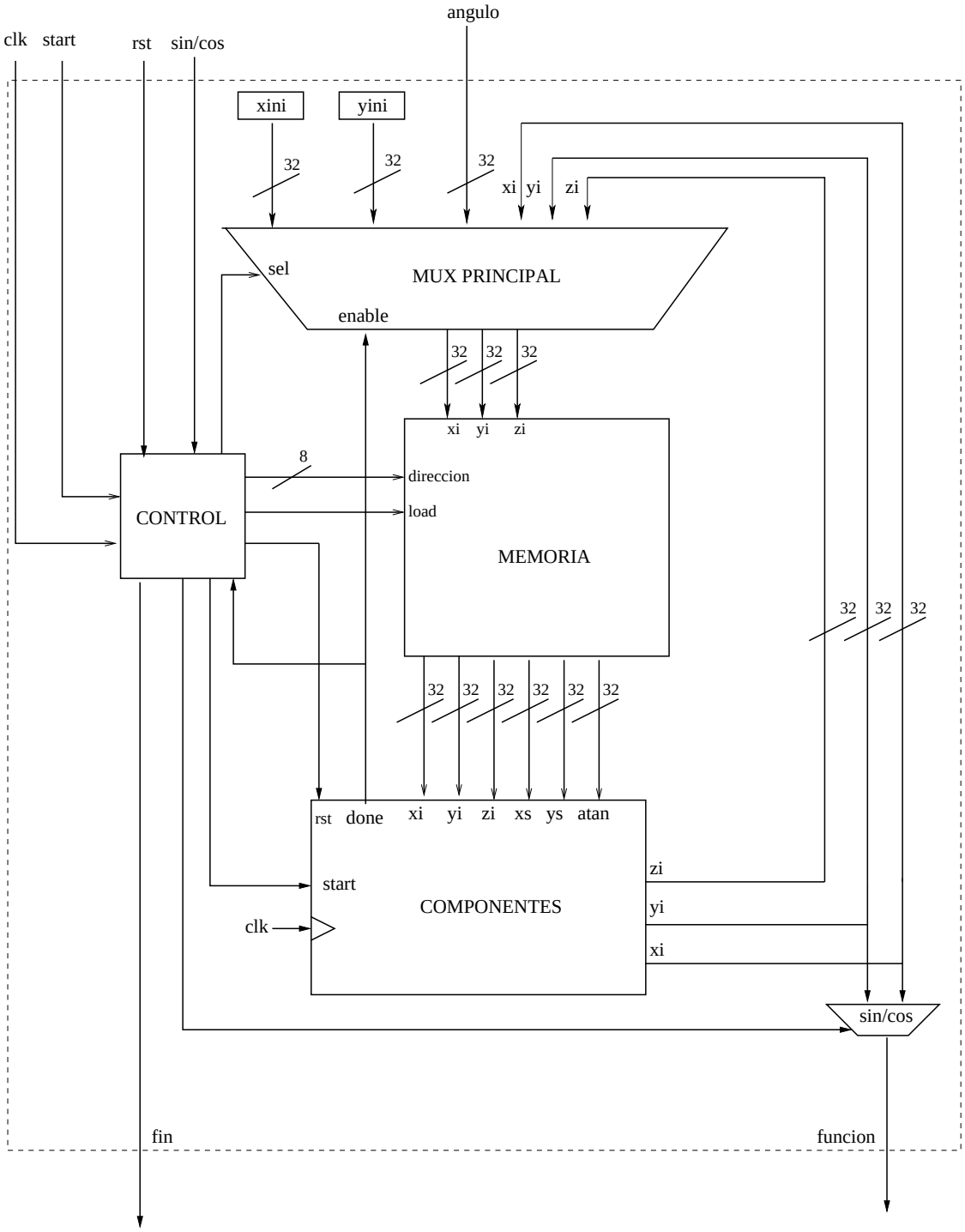

Figura 4.2: Ruta de datos del coprocesador

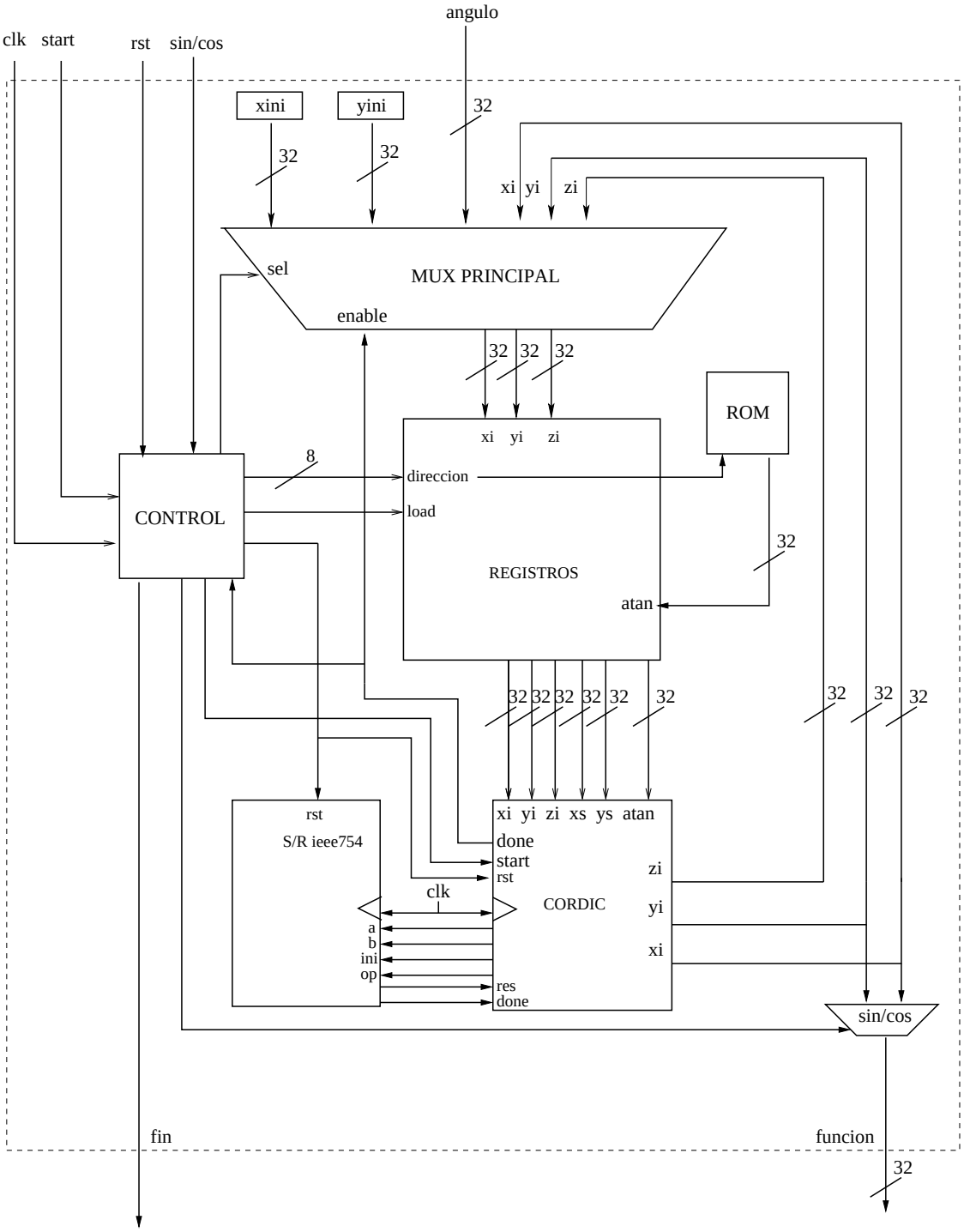

Figura 4.3: Ruta de datos detallada del coprocesador

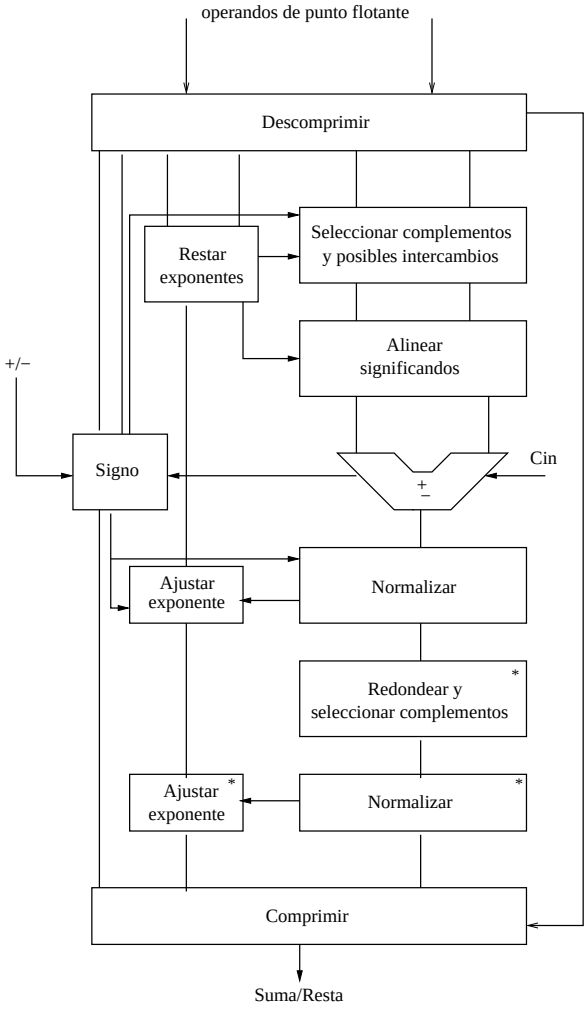

Figura 4.4: Suma de números de punto flotante

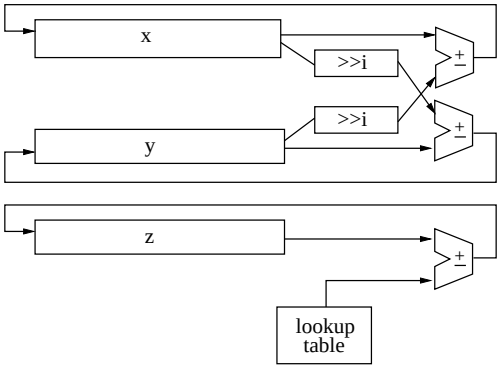

Figura 4.5: CORDIC en Hardware

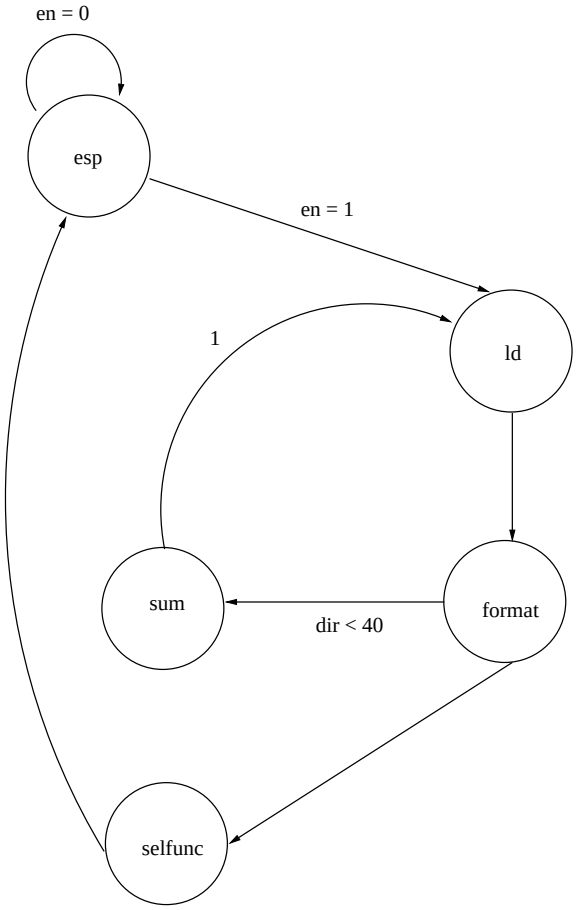

Figura 4.6: Módulo **control**. Diagrama de estados.

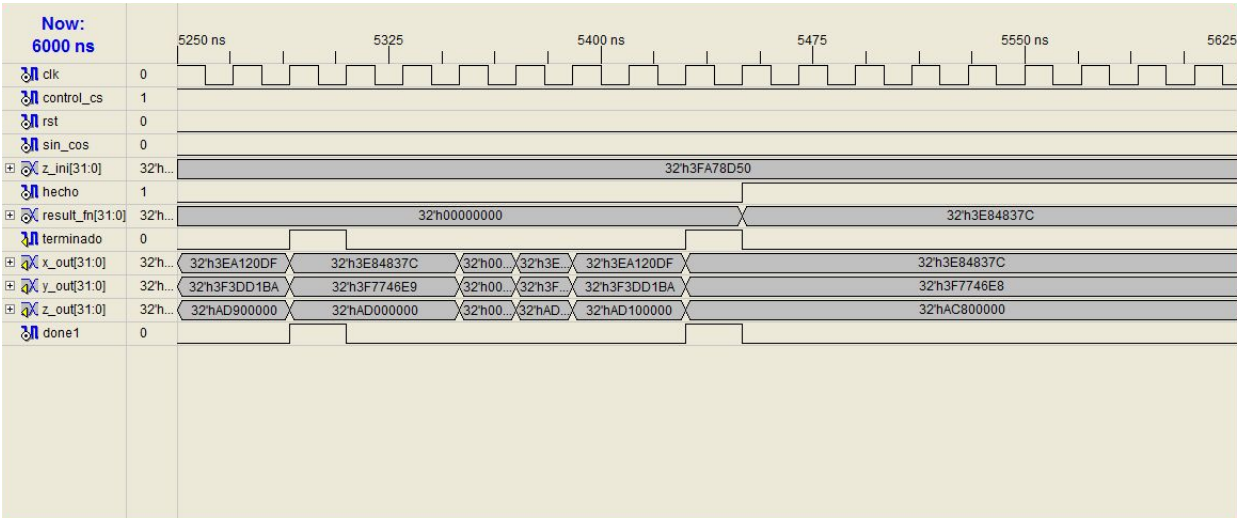

Figura 4.7: Simulación del cálculo de coseno para 1.30900rad

Apéndice A

Análisis funcional

Archivo: `originalCORDIC.c` (*descripción general del algortimo*)

```
1
2 #include <math.h>
3 #include <stdio.h>
4
5 /*************************************************************************
6 Garcia Hernandez David. Abril 2009
7
8 A continuacion se muestra la codificacion del algoritmo de CORDIC
9 para obtener senos y cosenos (en radianes).
10
11 *************************************************************************/
12         /*estructura VECTOR: X,Y angulo*/
13
14 struct vector
15 {
16         float x,y,z;
17 };
18
19 void cossinCORDIC (float angulo,int iteraciones)
20 {
21   struct vector vecRotar;
22         int i,d;
23         float x,k,Pik = 1;
24
25         /*inicializar x,y,z*/
26
27         /*colocar el vector unitario sobre el eje X*/
28         vecRotar.x = 1; vecRotar.y = 0;
29
30         /*comenzar con el angulo del cual se desea conocer el seno y el coseno*/
31         vecRotar.z = angulo;
32
33         printf("\n |     | X: %+-2.10f | Y: %+-2.10f | Z: %+-2.10f | ",
34         vecRotar.x, vecRotar.y, vecRotar.z);
35
36         for (i = 0; i < iteraciones; i++)
37         {
38                 if (vecRotar.z >= 0)
39                 d = 1;
40                 else d = -1;
```

```
41
42          k = 1/(sqrt(1+pow(2,-2*i)));
43          Pik *= k;
44
45          x = vecRotar.x;
46          vecRotar.x = vecRotar.x - vecRotar.y*d*pow(2,-i);
47          vecRotar.y = vecRotar.y + x*d*pow(2,-i);
48          vecRotar.z = vecRotar.z - d*atan(pow(2,-i));
49
50          printf("\n | %2d | X: %+-2.10f | Y: %+-2.10f | Z: %+-2.10f |
51                      k: %+-2.10f | Pik: %+-2.10f |"
52                  ,i, vecRotar.x*Pik, vecRotar.y*Pik, vecRotar.z, k, Pik);
53          }
54
55 }
56
57 int main()
58 {
59   int numIteraciones; float angulo;
60
61          angulo = 1.30900;
62          numIteraciones = 40;
63
64  cossinCORDIC (angulo,numIteraciones);
65
66 return 0;
67 }
```

Apéndice B

Análisis estructural

Archivo: `originalCORDICb.c` (*descripción basada en operaciones básicas*)

```
1#include <math.h>
2#include <stdio.h>
3
4 /*********************************************************************
5 Garcia Hernandez David. Abril 2009
6
7A continuacion se muestra la codificacion del algoritmo de CORDIC
8 para obtener senos y cosenos (en radianes).
9
10 *********************************************************************/
11
12          /*estructura VECTOR: X,Y angulo*/
13
14 struct vector
15 {
16          float x,y,z;
17 };
18
19 void cossinCORDIC (float angulo,int iteraciones)
20 {
21   struct vector vecRotar;
22          int i,d;
23          double x,k; /*x: respaldo de X n-1*/
24          double arctan[40];
25
26          /*arreglo de arcotangentes*/
27
28          arctan[ 0] = 0.7853981633974; arctan[ 1] = 0.4636476090008;
29          arctan[ 2] = 0.2449786631269; arctan[ 3] = 0.1243549945468;
30          arctan[ 4] = 0.0624188099960; arctan[ 5] = 0.0312398334303;
31          arctan[ 6] = 0.0156237286205; arctan[ 7] = 0.0078123410601;
32          arctan[ 8] = 0.0039062301320; arctan[ 9] = 0.0019531225165;
33          arctan[10] = 0.0009765621896; arctan[11] = 0.0004882812112;
34          arctan[12] = 0.0002441406201; arctan[13] = 0.0001220703119;
35          arctan[14] = 0.0000610351562; arctan[15] = 0.0000305175781;
36          arctan[16] = 0.0000152587891; arctan[17] = 0.0000076293945;
37          arctan[18] = 0.0000038146973; arctan[19] = 0.0000019073486;
38          arctan[20] = 0.0000009536743; arctan[21] = 0.0000004768372;
39          arctan[22] = 0.0000002384186; arctan[23] = 0.0000001192093;
40          arctan[24] = 0.0000000596046; arctan[25] = 0.0000000298023;
```

```
41          arctan[26] = 0.0000000149012; arctan[27] = 0.0000000074506;
42          arctan[28] = 0.0000000037253; arctan[29] = 0.0000000018626;
43          arctan[30] = 0.0000000009313; arctan[31] = 0.0000000004657;
44          arctan[32] = 0.0000000002328; arctan[33] = 0.0000000001164;
45          arctan[34] = 0.0000000000582; arctan[35] = 0.0000000000291;
46          arctan[36] = 0.0000000000146; arctan[37] = 0.0000000000073;
47          arctan[38] = 0.0000000000036; arctan[39] = 0.0000000000018;
48
49          /*inicializar x,y,z*/
50          vecRotar.x = 0.6072529554; vecRotar.y = 0;
51          /*comenzar con el angulo del cual se desea conocer el seno y el coseno*/
52          vecRotar.z = angulo;
53
54          printf("\n |     | X: %2.10f | Y: %2.10f | Z: %2.10f | ",
55          vecRotar.x, vecRotar.y, vecRotar.z);
56
57          for (i = 0; i < iteraciones; i++)
58          {
59                  if (vecRotar.z >= 0)
60                  d = 1;
61                  else d = -1;
62
63          x = vecRotar.x;
64          vecRotar.x = vecRotar.x - vecRotar.y*d*pow(2,-i);
65          vecRotar.y = vecRotar.y + x*d*pow(2,-i);
66          /*calcular z en funcion del areglo de arcotangentes*/
67          vecRotar.z = vecRotar.z - d*arctan[i];
68
69
70          printf("\n  %2d  | x: %+-2.10f | y:  %+-2.10f | z: %+-2.10f",
71                  i,vecRotar.x,vecRotar.y, vecRotar.z);
72          }
73
74 }
75
76 int main()
77 {
78 int numIteraciones; float angulo;
79
80          angulo = 1.30900;
81          numIteraciones = 40;
82
83 cossinCORDIC (angulo,numIteraciones);
84
85 return 0;
86 }
```

Apéndice C

Multiplexor Principal

Archivo: `muxPrincipla.vhdl` (*módulo en hardware del multiplexor principal*)

Nombre: Mux Principal

Descripción:

Permitir el paso del águlo inicial en radianes, únicamente para la iteración cero, así como un valor constante para X,e Y = 0. En adelante selec tomará el valor de '1' y entonces permitirá pasar los datos calculados en cada paso. Cuando el habilitador sea distinto de cero, retendrá los valores leios en el pulso de reloj anterior hasta que se le indique que sean reemplazados por los resultados actuales.

```
1 entity muxPrincipal is
2         port(
3
4                 selec:    in bit;
5                 habilitar: in bit;
6
7                 acumAngular_z: in bit_vector(31 downto 0); --ángulo inicial
8
9         -- valores para las componentes x,y,z en la i-ésima interación --
10
11                angulo_xi:     in bit_vector(31 downto 0);
12                eje_yi:        in bit_vector(31 downto 0);
13                acumAngular_zi:in bit_vector(31 downto 0);
14
15        -- componentes x,y,z seleccionados --
16
17                x_selec : out bit_vector(31 downto 0);
18                y_selec : out bit_vector(31 downto 0);
19                z_selec : out bit_vector(31 downto 0)
20                );
21 end entity;
22
23 architecture muxPrincipal of muxPrincipal is
24 begin
25        process(selec,habilitar)
26                begin
27                if selec = '0' then -- iteración cero
```

```
28
29                          -- X_0 = 0.6072529350; ( constante ) --
30                          x_selec <= "00111111000110110111010011101110";
31                          y_selec <= (others => '0'); -- y = 0;
32                          z_selec <= acumAngular_z;   -- ángulo inicial
33                  else
34                          -- iteración i-ésima --
35
36                          if habilitar'event and habilitar = '0' then
37
38                          -- nuevos valores --
39                          x_selec <= angulo_xi;
40                          y_selec <= eje_yi;
41                          z_selec <= acumAngular_zi;
42
43                          end if;
44                  end if;
45          end process;
46 end architecture muxPrincipal;
```

Apéndice D

Registros

Archivo: `registros.vhdl` (*módulo en hardware de la memoria de registros*)

Nombre: registros

Descripción:

Proporciona todos los parámetros necesarios para el sistema de ecuaciones del algoritmo. En el momento que se le indique (*carga = 1*) libera los datos: a,b,c: sin modificación, a,b desplazados *direccion* veces y multiplicados por el signo de c, y un arcotangente almacenado en una ROM en la posición que *direccion* lo indique, también multiplicado por el signo de c.

```
1 use work.alu.all;
2 entity registros is
3
4         port(
5                 a,b,c: in bit_vector(31 downto 0);
6                 direccion: in bit_vector(7 downto 0);
7                 carga: in bit;
8
9                 a_out,b_out,c_out: out bit_vector(31 downto 0);
10
11                -- a y b desplazados
12                a_out_desp,b_out_desp; : out bit_vector(31 downto 0);
13
14                --arco tangente seleccionado de la ROM
15                arctan_out: out bit_vector(31 downto 0);
16        );
17
18 end entity;
19
20 architecture registros of registros is
21
22        type tipoROM is array (0 to 39) of bit_vector(31 downto 0);
23
24        -- Memoria ROM de arcctan( 2^-i ) con 40 localidades
25        -- desde i = 0 hasta i = 39
26
```

```
27          constant atanROM:tipoROM:= (
28                                              "00111111010010010000111111011011",
29                                              "00111110111011010110001100111000",
30                                              "00111110011110101101101110110000",
31                                              "00111101111111101010110111010101",
32                                              "00111101011111111010101011011110",
33                                              "00111100111111111110101010101110",
34                                              "00111100011111111111101010101011",
35                                              "00111011111111111111111010101011",
36                                              "00111011011111111111111110101011",
37                                              "00111010111111111111111111101011",
38                                              "00111010011111111111111111111011",
39                                              "00111001111111111111111111111111",
40                                              "00111001100000000000000000000000",
41                                              "00111001000000000000000000000000",
42                                              "00111000100000000000000000000000",
43                                              "00111000000000000000000000000000",
44                                              "00110111100000000000000000000000",
45                                              "00110111000000000000000000000000",
46                                              "00110110100000000000000000000000",
47                                              "00110110000000000000000000000000",
48                                              "00110101100000000000000000000000",
49                                              "00110101000000000000000000000000",
50                                              "00110100100000000000000000000000",
51                                              "00110100000000000000000000000000",
52                                              "00110011100000000000000000000000",
53                                              "00110011000000000000000000000000",
54                                              "00110010100000000000000000000000",
55                                              "00110010000000000000000000000000",
56                                              "00110001100000000000000000000000",
57                                              "00110001000000000000000000000000",
58                                              "00110000100000000000000000000000",
59                                              "00110000000000000000000000000000",
60                                              "00101111100000000000000000000000",
61                                              "00101111000000000000000000000000",
62                                              "00101110100000000000000000000000",
63                                              "00101110000000000000000000000000",
64                                              "00101101100000000000000000000000",
65                                              "00101101000000000000000000000000",
66                                              "00101100100000000000000000000000",
67                                              "00101100000000000000000000000000"
68                          );
69
70 begin
71
72 process(carga)
73 begin
74  if carga'event and carga = '1' then
75
76          a_out <= a;
77          b_out <= b;
78          c_out <= c;
79
80          --Funciones almacenadas en un ALU--
81                          --funcion encargada de los desplazamientos y multiplicación por signo
82          a_out_desp <= despRegistro(a,direccion,c(c'left));
83          b_out_desp <= despRegistro(b,direccion,c(c'left));
84                          --el arco tangemte es tomado de la ROM de la posición que dirección
85                          --indique, para ello es necesario convertir de bits a entero.
86          arctan_out <= despRegistro(atanROM(conv_Bitvect_to_integer(direccion)),"00000000",c(c'left));
87
88  end if;
89
90 end process;
91 end architecture registros;
```

Apéndice E

CORDIC

Archivo: `cordic.vhdl` (*módulo en hardware de CORDIC*)

Nombre: cordic

Descripción:

En este módulo se encuentra almacenado el sistema de ecuaciones empleado por el algoritmo; recibe como entrada las componetes dadas por el módulo registros. Las tres componetes son calculadas al mismo tiempo creando tres instancias de un sumador.

```
1
2 entity cordic is
3         port(
4         x,y,z,arctan: in bit_vector(31 downto 0);
5         x_desp,y_desp: in bit_vector(31 downto 0);
6
7         clk: in bit;
8         ini: in bit;
9         rst: in bit;
10        terminado: out bit;
11
12        x_out: out bit_vector(31 downto 0);
13        y_out: out bit_vector(31 downto 0);
14        z_out: out bit_vector(31 downto 0)
15
16        );
17 end entity;
18
19 architecture cordic of cordic is
20
21 -- Sumador-Restador de números de punto flotante de simple precisión  --
22        component srIEEE754 is
23        port(
24                clk:in bit;
25                ini: in bit;
26                rst: in bit;
27                sumandoA:  in bit_vector (31 downto 0);
28                sumandoB:  in bit_vector (31 downto 0);
29                oper:      in bit;
30                done:     out bit;
```

```
31                 resultado: out bit_vector (31 downto 0)
32         );
33         end component srIEEE754;
34
35         signal done1,done2,done3: bit;
36         signal x_salida,y_salida,z_salida: bit_vector(31 downto 0);
37 begin
38
39         -- cálculo de componentes --
40         --  x:
41         suma1: srIEEE754 port map(clk,ini,rst,x,y_desp,'1',done1,x_salida);
42         --  y:
43         suma2: srIEEE754 port map(clk,ini,rst,y,x_desp,'0',done2,y_salida);
44         --  z:
45         suma3: srIEEE754 port map(clk,ini,rst,z,arctan,'1',done3,z_salida);
46
47                          x_out <= x_salida;
48                          y_out <= y_salida;
49                          z_out <= z_salida;
50
51                          -- señal de final de sumas --
52                          terminado <= done1 and done2 and done3;
53
54 end architecture cordic;
```

Apéndice F

Sumador Restador IEEE754

Archivo: `srIEEE754.vhdl` (*módulo en hardware de Sumador Restador IEEE754*)

Nombre: srIEEE754

Descripción:

Realiza la adición o substracción de dos números, expresados de acuerdo al estándar IEEE754.

```
1
2 use work.alu.all;
3 entity srIEEE754 is
4 port(
5         clk:in bit;
6         ini:in bit;
7         rst:in bit;
8
9         sumandoA:  in bit_vector (31 downto 0);
10        sumandoB:  in bit_vector (31 downto 0);
11        oper:      in bit;
12        done:     out bit;
13        resultado: out bit_vector (31 downto 0)
14 );
15 end srIEEE754;
16
17 architecture ArqsrIEEE754 of srIEEE754 is
18
19        --registro auxiliar, almacena exponente y mantiza--
20        signal registro: bit_vector (30 downto 0);
21
22        --bandera para intercambiar el orden de los operandos
23        signal intercambia: bit;
24
25        --registro auxiliar, indica operación a realizar
26        signal codoper: bit;
27
28        --bandera de excepción, cuando se da por terminado el proceso
29        --y se requiere "saltar" los pasos siguientes
30        signal fin: bit;
31
32        --registro auxiliar para ajuste de exponentes
```

```
33          signal ajuste: bit_vector (7 downto 0);
34
35          --contador de estados
36          signal estado: integer range 0 to 4;
37
38          --almacen del proceso de convertir bits a entero
39          signal n: integer;
40
41          --registro para descomprimir operandos--
42          signal a_signo: bit;
43          signal a_exponente: bit_vector(7 downto 0);
44          signal a_mantiza: bit_vector(41 downto 0);
45
46          signal b_signo: bit;
47          signal b_exponente: bit_vector(7 downto 0);
48          signal b_mantiza: bit_vector(41 downto 0);
49
50          -- señal de terminado
51          signal signal_done: bit;
52
53          --registros de resultado descomprimido--
54          signal res_signo: bit;
55          signal res_exponente: bit_vector(7 downto 0);
56          signal res_mantiza: bit_vector(41 downto 0);
57
58 begin
59          --empaquetado de resultado--
60          resultado <= res_signo & res_exponente & res_mantiza(41 downto 19);
61
62 process (clk,ini)
63
64          begin
65
66          if clk'event and clk = '1' then
67
68          --reset: baja las banderas de fin, y coloca en el contador de etapas en cero--
69
70          if rst = '1' then
71
72                  estado <= 0;
73                  done <= '0';
74                  signal_done <= '0';
75
76          else
77     ----------------------------
78          if ini = '1' then
79
80          case estado is
81
82-------------------------------------------------------------------------
83--------Carga de Operadores y reinicio de registros auxiliares ----------
84          when 0 =>
85                  if signal_done = '1' then
86                  done <= '0';
87                  signal_done <= '0';
88                  else
89
90                  a_signo <= sumandoA(31);
91                  a_exponente <= sumandoA(30 downto 23);
92
93                  -- agregar bit implicito y extencion de bits en la mantiza --
94                  a_mantiza <= '0' &  '1' & sumandoA(22 downto 0) & "00000000000000000";
95
96                  b_signo <= sumandoB(31);
97                  b_exponente <= sumandoB(30 downto 23);
98
```

```
99              -- agregar bit implicito y extencion de bits en la mantiza --
100             b_mantiza <= '0' &  '1' & sumandoB(22 downto 0) & "00000000000000000";
101
102             res_signo <= '0';
103             res_exponente <= (others => '0');
104             res_mantiza <= (others => '0');
105             registro <= (others => '0');
106
107             --inicialmente vale el bit de la operación solicitada--
108             -- '0' para sumar, '1' para restar --
109             codoper <= oper;
110
111             --una vez conluido el proceso, retorna al estado cero, entonces
112             --se da la salida de que ha terminado
113             done <= signal_done;
114
115             n <= 0;
116             intercambia <= '0';
117             fin <= '0';
118
119             estado <= estado + 1;
120             end if;
121
122       when 1 =>
123       ------------------------------------------------------------------------------
124       ------------------Evaluación del signo y ordenamiento de las entradas----------------
125       if oper = '0' then
126
127     --caso: suma con a igual a cero
128
129             if (a_signo & a_exponente) = "000000000" and (b_signo & b_exponente) /="000000000"
130             then
131
132                     res_signo <= b_signo; fin <= '1';
133                     registro <= b_exponente & b_mantiza(39 downto 17);
134
135             --caso: suma con b igual a cero
136
137             elsif(b_signo & b_exponente) = "000000000" and (a_signo & a_exponente) /="000000000"
138             then
139
140                     res_signo <= a_signo; fin <= '1';
141                     registro <= a_exponente & a_mantiza(39 downto 17);
142
143             else --no hay operandos igual a cero
144
145             if a_signo = b_signo then
146                     res_signo <= a_signo;
147             else                            --signos distinos entonces resta
148                     codoper <= '1';
149                     if (a_exponente & a_mantiza(39 downto 17)) =
150                     (b_exponente & b_mantiza(39 downto 17)) then
151                     -- numeros iguales con signo opuesto: resultado igual a cero
152
153                             res_signo <= '0';
154                             fin <= '1';
155
156                     elsif (a_exponente & a_mantiza(39 downto 17)) >
157                     (b_exponente & b_mantiza(39 downto 17)) then
158
159                             res_signo <= a_signo;
160                     else
161                        --caso: b mayor que a, intercambiar operandos y conservar signo de b
162                             res_signo <= b_signo;
163                             intercambia <= '1';
164                     end if;
```

```
165                 end if;
166                 end if;
167 else
168
169 --se trata de una sustracción (codper = 1)
170
171       --caso: resta con a igual a cero
172                 if (a_signo & a_exponente) = "000000000" and (b_signo & b_exponente) /="000000000"
173                 then
174                 res_signo <= '1';fin <= '1';
175                 registro <= b_exponente & b_mantiza(39 downto 17);
176
177      --caso: resta con a igual a cero
178                 elsif(b_signo & b_exponente) = "000000000" and (a_signo & a_exponente) /="000000000"
179                 then
180                 res_signo <= a_signo;fin <= '1';
181                 registro <= a_exponente & a_mantiza(39 downto 17);
182
183                 else --no hay operandos igual a cero.
184
185                 if a_signo /= b_signo then    --signos distintos, entonces sumar
186                         res_signo <= oper nand b_signo;
187                         codoper <= '0';
188                 else
189                          -- numeros iguales con signo igual, entonces resultado es cero
190                         if (a_exponente & a_mantiza(39 downto 17)) =
191                         (b_exponente & b_mantiza(39 downto 17)) then
192                                 res_signo <= '0';
193                                 fin <= '1';
194                         elsif (a_exponente & a_mantiza(39 downto 17)) >
195                         (b_exponente & b_mantiza(39 downto 17)) then
196                                 res_signo <= a_signo;
197                         else
198                         --caso: b mayor que a, intercambiar operandos y conservar signo negado de b
199                                 res_signo <= not b_signo;
200                                 intercambia <= '1';
201                         end if;
202                 end if;
203                 end if;
204 end if;
205
206         ------------------------------------------------------------------------------
207         ----------------------Evaluación del exponente y ajuste de mantiza------------
208         if a_exponente = b_exponente then
209
210            res_exponente <= a_exponente;
211            n <= 0;
212
213         elsif a_exponente > b_exponente then          -- a > b
214
215            res_exponente <= a_exponente;             --fijat tentativamente el exponente
216
217                 --obtener formato entero de la diferencia entre exponentes
218                 n  <= conv_Bitvect_to_integer(fulladder(a_exponente,b_exponente,'1'));
219
220         else                                          -- b > a
221                 res_exponente <= b_exponente;             --fijat tentativamente el exponente
222
223                 --obtener formato entero de la diferencia entre exponentes
224                 n  <= conv_Bitvect_to_integer(fulladder(b_exponente,a_exponente,'1'));
225         end if;
226
227 estado <= 2;
228
229 when 2 =>
230
```

```
231         if fin /= '1' then --si bandera esta baja, continuar con proceso correspondiente
232         ------------------------------------------------------------------------------------
233         ----------------------Suma de las mantizas------------------------------------------
234
235                 --ordenar parámetros y sumar
236                 if (intercambia='1') then
237                 res_mantiza(41 downto 0) <= fulladder(b_mantiza,a_mantiza SRL n,codoper);
238                 else
239                 res_mantiza(41 downto 0) <= fulladder(a_mantiza,b_mantiza SRL n,codoper);
240                 end if;
241         end if;
242                 estado <= 3;
243
244         when 3 =>
245  ------------------------------------------------------------------------------------------
246  ----------------------Ajuste las Mantiza Resultado----------------------------------------
247
248 if fin /= '1' then --normalizar
249         registro <= ajusteMantiza(res_mantiza,res_exponente);
250 end if;
251
252         --señal predictiva, que en la proxima etapa se termina el proceso
253         signal_done <= '1';
254
255 estado <= 4;
256
257 when 4 =>
258         -- carga de resultados en exponete y mantiza --
259         res_exponente <= registro(30 downto 23);
260         res_mantiza(41 downto 19) <= registro(22 downto 0);
261         done <= signal_done; --salida de fin
262         estado <= 0;
263 end case;
264
265 end if; -- fin de if ini
266 end if; -- fin de if rst
267 end if; -- fin de if clk
268
269 end process;
270
271 end ArqsrIEEE754;
```

Apéndice G

Unidad de Control

Archivo: `control.vhdl` (*módulo en hardware de la Unidad de Control*)

```
1 use work.alu.all;
2 entity control is
3
4         port(
5                 start_Ctrl:       in bit;  --activa el control
6                 done_Cordic:      in bit;  --señal de notificación
7                 clk:              in bit;  --pulso de reloj
8                 rst:      in bit;          --reiniciar control
9
10                start_Cordic:     out bit;  --activa al móulo del algoritmo de CORDIC
11                load_Reg:         out bit;  --indica al registrio que pude cargar
12                sel_mux:          out bit;  --indica al multiplexor la entrada que mandará a CORDIC
13                terminado: out bit;
14                dir_Reg:          out bit_vector (7 downto 0)
15        );
16
17end entity;
18
19architecture control of control is
20
21signal orden: integer range 0 to 1;
22signal inicial: integer range 0 to 1;
23signal etapa: bit_vector (7 downto 0);
24signal factor: bit_vector (7 downto 0):= "00000001";
25
26
27begin
28        dir_Reg <= etapa;
29        process (clk)
30
31        begin
32                if clk'event and clk = '1' then
33
34                if rst = '1' then
35                        --reiniciar registros
36                        etapa <= (others => '0');
37                        inicial <= 0;
38                        orden <= 0;
39                        sel_mux <= '0';
40                        load_REg <= '0';
```

```
41                          start_Cordic <= '0';
42                          terminado <= '0';
43                          else
44
45                          if start_Ctrl = '1' then
46
47                          case inicial is
48
49                          when 0 =>
50                                  --solamente en la primer iteración permite
51                                  --procesar los valores iniciales
52
53                                            load_Reg <= '1';
54                                            start_Cordic <= '1';
55                                  --incrementa contador
56                                  etapa <= fulladder(etapa,factor,'0');
57                                  sel_mux <= '1'; -- en adelante sel es uno
58                                  inicial <= 1;
59                          when 1 =>
60                                  if done_Cordic = '1' or orden = 1 then
61
62                                   if etapa = "00100111" then -- "39"
63                                                 start_Cordic <= '0'; -- pausar cordic
64                                                 terminado <= '1';
65                                   else
66                                          case orden is
67                                                 when 0 => load_Reg <= '0'; --suspende carga de registros
68
69                                                 start_Cordic <= '0';       -- pausa cordic cordic
70                                                 orden <= 1;
71
72                                                 when 1 =>
73
74                                                 load_Reg <= '1'; -- inicia carga de registros
75                                                 start_Cordic <= '1'; -- inicio cordic
76                                                 etapa <= fulladder(etapa,factor,'0');--incremeta contador
77                                                 orden <= 0;
78
79                                           end case;
80                                          end if; -- fin contador;
81                                  end if; --fin done_cordic or orden = 1
82                          end case;
83                  end if; --fin start control
84                  end if; --fin reset;
85                  end if; --fin clk
86          end process;
87 end architecture control;
```

Bibliografía

[1] George B. Thomas Jr. *Cálculo Una Variable*. Pearson Addison Weskey, 2006.

[2] Tanya Vladimirova, David Eamey, Sven Keller, and Prof Sir Martin Sweeting. Floating-Point Mathematical Co-Processor for a Single-Chip On-Board Computer. Technical report, Surrey Space CentreSchool of Electronics and Physical Sciences University of Surre, 2004.

[3] Rubén Lumbiarres López. Co-diseño hardware software de una unidad aritmética en coma flotante para microprocesador de 32 bits. Technical report, S-Escola Politécnica Superior d'Enginyeria de Vilanova i la Geltrú, 2008.

[4] J. Volder. The cordic trigonometric computing technique. Technical report, IRE Trans, 1959.

[5] Esther Leyva Suárez. *Algoritmos Córdicos*. Escuela Superior de Física y Matemáticas del Instituto Politécnico Nacional, 2004.

[6] Anton Howard. *Álgebra lineal*. Limusa wiley, 2006.

[7] Behrooz Parhami. *Computer Arithmetic ALGORITHMS AND HARDWARE DESIGNS*. Oxford, 1999.

[8] Robert Joachim Schweers. *Descripción en VHDL de arquitecturas para implementar el algoritmo CORDIC*. Universidad Nacional de La Plata, 2002.

[9] Dennis G. Zill. *Ecuaciones Diferenciales*. CENGAGE Learning, 2009.

[10] A. E. Jacquin. Fractal image coding: a review. *Proceedings of the IEEE*, 81:1451–1465, 1993.

[11] Kurt Gieck. *manual de fórmulas técnicas*. Alfaomega, 1981.

[12] Murray R. Spiegel. *Manual de fórmulas y tablas matemáticas*. McGraw-Hill, 1998.