Agradecimientos

Este trabajo va dedicado a mis padres, que posibilitaron la culminación de mi carrera, debido al apoyo brindado y su gran insistencia.

Quisiera agradecer a mis directores el M.Sc. Prof. Oscar N. Bria y el Ing. Prof. Horacio A. Villagarcía por su apoyo y excelentes consejos, a los profesores de la Facultad de Informática, a mi hermano Gerrit y a mi novia Andrea por apoyarme y soportarme siempre. Y por último un agradecimiento muy especial para Constanza, mi sobrina, que me acompañó durante las pausas que me tomé en la realización del trabajo.

Indice

Agradecimientos i

Indice ii

Introducción vi
El algoritmo de cómputo numérico CORDIC vi
Problema a resolver vii
Objetivos a cumplir vii
Motivaciones y expectativas vii
Organización del informe viii

Capítulo 1 – El algoritmo CORDIC 1
1.1 Fundamento teórico 1
1.2 Cálculo del seno y del coseno 6
1.3 Transformación de coordenadas polares a cartesianas y viceversa 7
1.4 Arcotangente y módulo de un vector 8
1.5 CORDIC generalizado 8
1.5.1 Caso lineal 8
1.5.2 Caso hiperbólico 9
1.6 Arquitecturas para implementar el algoritmo CORDIC 10
1.6.1 La arquitectura Bit-Paralela Iterativa 10
1.6.2 La arquitectura Bit-Paralela Desplegada 11
1.6.3 La arquitectura Bit-Serie Iterativa 12

Capítulo 2 – Metodologías de diseño de hardware 14
2.1 Herramientas CAD 14
2.2 Diseño Bottom-Up 16
2.3 Diseño Top-Down 17
2.4 Ventajas del diseño Top-Down 18
2.5 Descripción de un diseño 19

Capítulo 3 – El lenguaje de descripción de hardware VHDL 20
3.1 VHDL describe comportamiento 20
3.2 VHDL describe estructura 21
3.3 Un ejemplo de descripción en VHDL 21
3.4 Modelo de tiempo basado en eventos discretos 24
3.5 VHDL como lenguaje de programación 25
3.6 Elementos de sintaxis 25
3.6.1 Comentarios 25
3.6.2 Identificadores 25
3.6.3 Números 25
3.6.4 Caracteres 26
3.6.5 Cadenas de caracteres (Strings) 26
3.6.6 Cadenas de bits 26
3.6.7 Tipos de datos del lenguaje 26
3.6.7.1 Tipo entero 26
3.6.7.2 Tipos físicos 27

3.6.7.3 Tipos de punto flotante 27
3.6.7.4 Tipo enumerativo 27
3.6.7.5 Tipo arreglo 28
3.6.7.6 Tipo registro 29
3.6.7.7 Tipo puntero 29
3.6.7.8 Tipo archivo 30
3.6.7.9 Subtipos 30
3.6.8 Declaración de objetos de datos 30
3.6.9 Atributos 31
3.6.10 Expresiones y operadores 32
3.6.10.1 Operadores lógicos 32
3.6.10.2 Operadores de desplazamiento 32
3.6.10.3 Operadores relacionales 32
3.6.10.4 Operador de concatenación 33
3.6.10.5 Operadores aritméticos 33
3.6.11 Construcciones secuenciales 33
3.6.11.1 Asignación a variables 33
3.6.11.2 Sentencia IF 33
3.6.11.3 Sentencia CASE 34
3.6.11.4 Sentencia nula 34
3.6.11.5 Aserciones (Assertions) 34
3.6.11.6 Sentencia de bucle 34
3.6.12 Subprogramas y paquetes 35
3.6.12.1 Funciones y procedimientos 35
3.6.12.2 Sobrecarga de operadores (Overloading) 37
3.6.12.3 Paquetes (Packages) 38
3.6.12.4 Alcance, visibilidad y utilización de los paquetes 38
3.6.13 Declaración de entidad 38
3.6.14 Declaración de arquitectura 39
3.6.14.1 Bloques 40
3.6.14.2 Declaración de componentes 40
3.6.14.3 Instanciación de componentes 41
3.6.15 Asignación a las señales 41
3.6.16 Ejecución secuencial: Procesos y la sentencia WAIT 42
3.6.17 Asignación concurrente a señales 44
3.6.18 Unidades y bibliotecas 45
3.6.19 La sentencia GENERATE 45
3.6.20 La unidad de configuración 46
3.6.20.1 Especificación de configuración 46
3.6.20.2 Declaración de configuración 47
3.6.21 Buses y resolución de señales 48
3.6.22 Aserciones concurrentes 49
3.6.23 Invocación a procedimiento concurrente 49
3.6.24 Transacciones nulas 50
3.7 Descripción de bancos de prueba 50
3.7.1 Método tabular 50
3.7.2 Utilización de archivos 51
3.7.3 Utilización de un algoritmo 51

Capítulo 4 – Descripción del algoritmo CORDIC en VHDL 52

4.1 Herramienta de desarrollo 53
4.2 La descripción funcional algorítmica 54
4.2.1 La descripción del package CORDIC 54
4.2.2 El banco de pruebas para la descripción funcional algorítmica 56
4.3 Descripción de las arquitecturas particulares 57
4.3.1 El formato numérico 57
4.3.2 Componentes comunes a ambas arquitecturas 59
4.3.2.1 El sumador completo (Full-Adder) 60
4.3.2.2 La unidad de suma 61
4.3.2.3 El multiplexor de dos bits 63
4.3.3 La descripción bit-paralela desplegada y sus componentes 63
4.3.3.1 La unidad de desplazamiento aritmético cableada 64
4.3.3.2 La entidad y arquitectura correspondientes a una iteración 65
4.3.3.3 Descripción final de la arquitectura bit-paralela desplegada 67
4.3.4 La arquitectura bit-paralela iterativa y sus componentes 69
4.3.4.1 La unidad de desplazamiento aritmético para la descripción iterativa 69
4.3.4.2 Las compuertas lógicas de múltiples entradas 71
4.3.4.3 La tabla de búsqueda 73
4.3.4.4 El multiplexor múltiple de dos bits 75
4.3.4.5 Circuitos secuenciales 76
4.3.4.6 Los elementos de almacenamiento: Flip-flops 77
4.3.4.7 El flip-flop D 77
4.3.4.8 El registro 80
4.3.4.9 El contador de iteraciones 81
4.3.4.10 La unidad de control 83
4.3.4.11 Diseño y descripción de la unidad de control 84
4.3.4.12 Descripción final de la arquitectura bit-paralela iterativa 91
4.3.5 El banco de pruebas para las descripciones particulares 94

Capítulo 5 – Simulación de las descripciones 95

5.1 Herramientas de simulación 95
5.2 Valores para la simulación 97
5.3 Simulación de la descripción funcional algorítmica 98
5.4 Simulación de las arquitecturas particulares 102
5.4.1 Cálculo de valores 102
5.4.2 Visualización de ondas 108

Conclusiones 115

Acerca del uso de VHDL 115
Acerca de las descripciones del algoritmo CORDIC 116
Acerca de la exactitud de los resultados numéricos 116
Perspectivas sobre trabajos futuros 117

Anexos **118**

119
119
122

123
123
124
124
125
126
127

128

131

136

141

143

Introducción

El algoritmo de cómputo numérico CORDIC

Existen gran cantidad de algoritmos eficientes [7] [8] que pueden emplearse para el cálculo de diversas funciones matemáticas, sin embargo sólo algunos de éstos algoritmos pueden implementarse adecuadamente en hardware. Entre estos algoritmos se destaca una clase de los mismos basada únicamente en sumas y desplazamientos, colectivamente denominados algoritmos CORDIC. Esta clase de algoritmos pueden utilizarse para calcular funciones trigonométricas circulares, hiperbólicas y funciones lineales.

El cálculo de las funciones trigonométricas está basado en rotaciones de vectores. La denominación CORDIC, es un acrónimo de COordinate Rotation DIgital Computer o en castellano, Computadora Digital para Rotación de Coordenadas.

El algoritmo CORDIC fue desarrollado originalmente como una solución digital para los problemas de navegación en tiempo real. El trabajo original es acreditado a Jack Volder [1] quien investigó el algoritmo CORDIC para el caso de rotaciones circulares. Ciertas extensiones a la teoría de CORDIC están basadas en trabajos de John Walther [5] entre otros, y proveen soluciones para una clase más amplia de funciones.

Como lo propuso Jack Volder el algoritmo CORDIC realiza únicamente operaciones de suma y desplazamiento, que lo hace idóneo para ser implementado en hardware. No obstante al implementar dicho algoritmo se puede optar por diversas arquitecturas de diseño y se debe balancear la complejidad del circuito respecto del desempeño. Las arquitecturas utilizadas para implementar el algoritmo CORDIC son, bit-paralela desplegada (Bit-Parallel Unrolled), bit-paralela iterativa (Bit-Parallel Iterative) y bit-serie iterativa (Bit-Serial Iterative) [2].

Problema a resolver

En el presente trabajo se estudia el algoritmo de cómputo numérico CORDIC y se describen algunas de sus variantes arquitecturales. Para comenzar se realizará una descripción a nivel funcional algorítmica con un alto nivel de abstracción del algoritmo en VHDL, utilizando aritmética en punto flotante proporcionada por el lenguaje. El propósito es validar el funcionamiento de algoritmo mediante el cálculo del seno, del coseno y del arcotangente. Seguidamente se realizará la descripción de dos de las arquitecturas mas comunes (CORDIC bit-paralelo desplegado y CORDIC bit-paralelo iterativo) utilizando aritmética de punto fijo como forma de representación numérica.

La descripción de las arquitecturas particulares, se llevará a cabo a partir del funcionamiento correcto de la descripción funcional algorítmica. Otro aspecto interesante es la modificación de los parámetros principales que afectan a las arquitecturas, ancho de palabra y número de iteraciones. Para verificar el funcionamiento de las descripciones, se utilizarán como referencia a las funciones seno, coseno y arcotangente. La validación se llevará a cabo utilizando MATLAB™ como herramienta de comparación.

Objetivos a cumplir

Objetivo primario

Estudiar el algoritmo de cómputo numérico CORDIC y utilizar el lenguaje de descripción de hardware VHDL para describir algunas de sus variantes arquitecturales, utilizando aritmética de punto fijo.

Objetivo secundario

Realizar simulaciones de las arquitecturas descriptas, modificando los parámetros que afectan a las mismas (ancho de palabra y número de iteraciones) para validar su funcionamiento y determinar la exactitud que se obtiene en los resultados basándose en el cálculo de las funciones seno, coseno y arcotangente.

Motivaciones y expectativas

La descripción de este algoritmo trae aparejado una serie de aspectos altamente positivos en el área del diseño de hardware considerado desde una perspectiva informática.

Es necesario realizar un estudio profundo de las técnicas de diseño utilizadas tanto para circuitos combinatorios como secuenciales, logrando la interiorización y comprensión de las metodologías, técnicas y herramientas de software utilizadas comúnmente en ingeniería, pero esta vez dentro del ámbito de la informática.

Este trabajo no se limita sólo a un estudio teórico, sino que el resultado se vea reflejado en una descripción terminada que en un futuro pueda ser implementada en una plataforma específica con cambios adecuados.

Por otra parte éste desarrollo obliga al estudio de las tres arquitecturas más comunes del algoritmo, culminando con la descripción y simulación de dos de las mismas.

Por otra parte el algoritmo CORDIC presenta una serie características importantes:

- El cálculo de las funciones seno y coseno se lleva a cabo en forma simultánea.
- Puede ampliarse fácilmente en un futuro para realizar el cálculo, no sólo de funciones circulares, sino también de funciones hiperbólicas y lineales.
- Está basado únicamente en sumas y desplazamientos resultando especialmente idóneo para ser descripto en hardware.

Organización del Informe

El presente informe se dividió en cinco capítulos, las conclusiones y cinco anexos.

El *capítulo 1* trata en profundidad los aspectos matemáticos del algoritmo CORDIC, así como sus casos, empezando por el caso circular para concluir con una breve explicación de los casos lineal e hiperbólico. El capítulo concluye con la explicación de las variantes arquitecturales del algoritmo, abordando cada una de las arquitecturas más comunes utilizadas para su implementación.

Los *capítulos 2 y 3* tratan sobre metodologías de diseño de hardware tradicionales y asistidas por computadora explicando con gran detalle el lenguaje de diseño VHDL. Se culmina el capítulo 3 con la exposición de diversas metodologías para generar bancos de prueba en VHDL, módulos vitales para llevar a cabo la simulación de un diseño.

El *capítulo 4* realiza un análisis previo explicando cuales son las arquitecturas descriptas y las herramientas utilizadas, para continuar con la descripción detallada cada una de las arquitecturas y sus componentes. Se comienza por la descripción funcional algorítmica, pasando finalmente a la descripción de las arquitecturas particulares. En este capítulo también se introducen técnicas estudiadas sobre diseño de sistemas digitales.

Los detalles de simulación de las arquitecturas descriptas se explican en el *capítulo 5*. Como resultado de la simulación se obtienen resultados que permiten validar el funcionamiento correcto de las descripciones, así como observar la exactitud que se obtiene en los resultados al alterar los parámetros que afectan a las descripciones (ancho de palabra y número de iteraciones). No se realiza un estudio detallado de errores, se analizan los resultados obtenidos durante las simulaciones. Las simulaciones se basaron en el cálculo del seno, del coseno y del arcotangente.

En la sección *conclusiones*, se exponen las conclusiones obtenidas acerca del uso de VHDL, de la descripción del algoritmo CORDIC y de la exactitud numérica alcanzada durante la simulación. Por último se presentan algunas perspectivas acerca de trabajos futuros.

Los anexos, denominados *A*, *B*, *C* y *D*, abordan temas con consideraciones sobre convergencia del algoritmo CORDIC, lógica booleana, uno de los scripts de MATLAB™ utilizado para analizar los archivos producto de la simulación y algunos bancos de prueba desarrollados en VHDL para validar las descripciones. El anexo *E* contiene resultados adicionales sobre las simulaciones.

Por último se incluye una breve descripción sobre el contenido del CD-ROM adjunto.

Capítulo 1

El algoritmo CORDIC

CORDIC es un acrónimo de COordinate Rotation DIgital Computer que en castellano significa Computadora Digital para Rotación de Coordenadas. El algoritmo original fue propuesto por Jack Volder en el año 1959 [1], con el propósito de calcular funciones trigonométricas mediante la rotación de vectores. La rotación de vectores puede utilizarse a su vez para conversión de sistemas de coordenadas (cartesiano a polar y viceversa), magnitud de vectores y como parte de funciones matemáticas mas complejas como ser la Transformada Rápida de Fourier (Fast Fourier Transform, FFT) y la Transformada Coseno Discreta (Discrete Cosine Transform, DCT).

1.1 Fundamento teórico

El algoritmo CORDIC, proporciona un método iterativo para rotar vectores ciertos ángulos determinados y se basa exclusivamente en sumas y desplazamientos. El algoritmo original tal como fue propuesto inicialmente, describe la rotación de un vector bidimensional en el plano cartesiano. Su funcionamiento se deduce de la fórmula general para rotación de vectores como se muestra a continuación

$$\begin{aligned} x' &= x\cos\theta - y\,\mathrm{sen}\,\theta \\ y' &= y\cos\theta + x\,\mathrm{sen}\,\theta \end{aligned} \qquad (1.1)$$

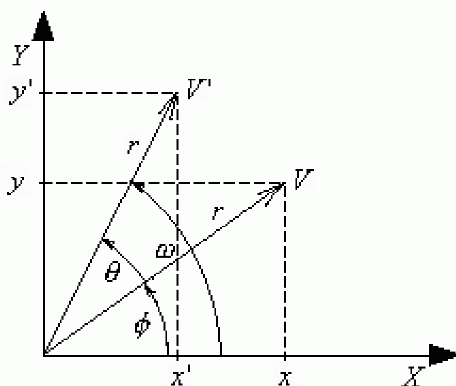


Figura 1.1

La fórmula (1.1) se deduce a partir de las identidades trigonométricas conocidas, en donde

$$V:\begin{cases} x = r\cos\phi \\ y = r\ \mathrm{sen}\,\phi \end{cases} \qquad (1.2.a) \qquad\qquad V':\begin{cases} x' = r\cos\omega \\ y' = r\,\mathrm{sen}\,\omega \end{cases} \qquad (1.2.b)$$

Si $r = 1$ y $\theta = \omega - \phi$ entonces $\omega = \theta + \phi$. Utilizando las identidades para el seno y el coseno de la suma de ángulos se tiene

$$\begin{aligned} \mathrm{sen}(\alpha+\beta) &= \mathrm{sen}\,\alpha\cos\beta + \cos\alpha\ \mathrm{sen}\,\beta \\ \cos(\alpha+\beta) &= \cos\alpha\cos\beta - \mathrm{sen}\,\alpha\ \mathrm{sen}\,\beta \end{aligned} \qquad (1.3)$$

Si se sustituye $\omega = \theta + \phi$ en (1.2.b), aplicando (1.3) y (1.2.a), se obtiene

$$\begin{aligned} x' &= \cos(\theta+\phi) = \cos\theta\cos\phi - \mathrm{sen}\,\theta\,\mathrm{sen}\,\phi = x\cos\theta - y\,\mathrm{sen}\,\theta \\ y' &= \mathrm{sen}(\theta+\phi) = \mathrm{sen}\,\theta\cos\phi + \cos\theta\,\mathrm{sen}\,\phi = x\,\mathrm{sen}\,\theta + y\cos\theta \end{aligned}$$

Con lo que se obtiene la fórmula general (1.1), la cual se puede reagrupar considerando que $\cos\theta \neq 0$ [4] de forma tal que

$$\begin{aligned} x' &= \cos\theta\left(x - y\,\mathrm{tg}\,\theta\right) \\ y' &= \cos\theta\left(y + x\,\mathrm{tg}\,\theta\right) \end{aligned} \qquad (1.4)$$

Si además

$$\cos\theta = \sqrt{\cos^2\theta} = \sqrt{\frac{1}{\frac{1}{\cos^2\theta}}} = \frac{1}{\sqrt{\frac{1}{\cos^2\theta}}} = \frac{1}{\sqrt{\frac{\mathrm{sen}^2\,\theta + \cos^2\theta}{\cos^2\theta}}} = \frac{1}{\sqrt{\frac{\cos^2\theta}{\cos^2\theta} + \frac{\mathrm{sen}^2\,\theta}{\cos^2\theta}}} =$$

$$= \frac{1}{\sqrt{1 + \mathrm{tg}^2\,\theta}}$$

entonces la ecuación (1.4) puede expresarse como

$$\begin{aligned} x' &= \frac{x - y\,\mathrm{tg}\,\theta}{\sqrt{1 + \mathrm{tg}^2\,\theta}} \\ y' &= \frac{y + x\,\mathrm{tg}\,\theta}{\sqrt{1 + \mathrm{tg}^2\,\theta}} \end{aligned} \qquad (1.5)$$

En el algoritmo CORDIC, las rotaciones de vectores son reemplazadas por pseudorotaciones [7] como se muestra en la figura 1.2.

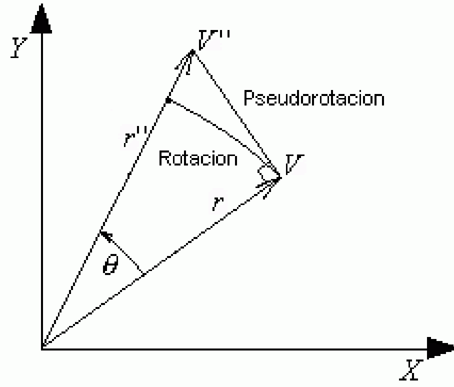


Figura 1.2

Mientras que una rotación real no cambia la magnitud del vector V una vez rotado, una pseudorotación incrementa su magnitud en

$$V'' = V\sqrt{1+\operatorname{tg}^2\theta} \tag{1.6}$$

El vector pseudorotado V'' incrementa su magnitud en $\sqrt{1+\operatorname{tg}^2\theta}$ respecto del vector original V. La pseudorotación por el ángulo θ está caracterizada por las ecuaciones

$$\begin{aligned} x'' &= x - y\operatorname{tg}\theta \\ y'' &= y + x\operatorname{tg}\theta \end{aligned} \tag{1.7}$$

La rotación de un ángulo puede descomponerse en una suma de rotaciones mas pequeñas. Si se asume inicialmente que $x = x_0$, $y = y_0$ y $z = z_0$, luego de n iteraciones para una rotación real se obtiene,

$$\begin{aligned} x_n' &= x\cos\left(\sum_{i=0}^{n-1}\theta_i\right) - y\operatorname{sen}\left(\sum_{i=0}^{n-1}\theta_i\right) \\ y_n' &= y\cos\left(\sum_{i=0}^{n-1}\theta_i\right) + x\operatorname{sen}\left(\sum_{i=0}^{n-1}\theta_i\right) \\ z_n' &= z - \left(\sum_{i=0}^{n-1}\theta_i\right) \end{aligned} \tag{1.8}$$

mientras que para una pseudorotación,

$$\begin{aligned} x_n'' &= \left(x\cos\left(\sum_{i=0}^{n-1}\theta_i\right) - y\operatorname{sen}\left(\sum_{i=0}^{n-1}\theta_i\right)\right)\prod_{i=0}^{n-1}\sqrt{1+\operatorname{tg}^2\theta_i} \\ y_n'' &= \left(y\cos\left(\sum_{i=0}^{n-1}\theta_i\right) + x\operatorname{sen}\left(\sum_{i=0}^{n-1}\theta_i\right)\right)\prod_{i=0}^{n-1}\sqrt{1+\operatorname{tg}^2\theta_i} \\ z_n'' &= z - \left(\sum_{i=0}^{n-1}\theta_i\right) \end{aligned} \tag{1.9}$$

, en donde $\sum_{i=0}^{n-1} \theta_i = \theta$

A las ecuaciones 1.8 y 1.9 se agrega una tercera ecuación para z_n que se denomina *acumulador angular* [7] porque incluye las rotaciones efectuadas hasta el momento.

En la figura 1.3 se muestra un vector *V* rotado sobre otro vector *V'* mediante rotaciones menores.

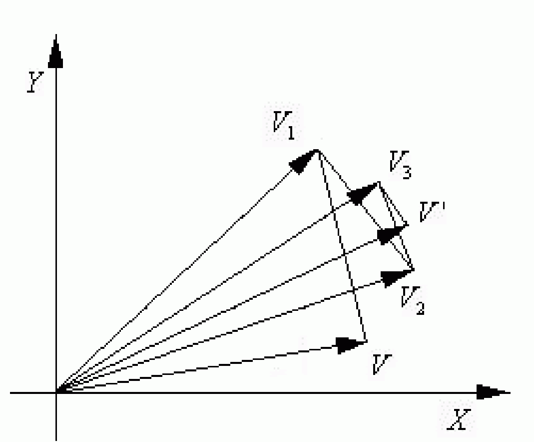


Figura 1.3

Si los ángulos de rotación se restringen de forma tal que $\operatorname{tg}\theta = \pm 2^{-i}, i \in \mathrm{N}$, la multiplicación por la tangente, se reduce a una operación de desplazamiento. Los diversos ángulos pueden obtenerse realizando una serie de rotaciones elementales cada vez mas pequeñas [Anexo A]. En cada iteración, en lugar de determinar si se debe rotar o no, se elige el sentido de rotación. Partiendo de las ecuaciones dadas en (1.4), y considerando que $\cos(\theta_i) = \cos(-\theta_i)$, una iteración puede expresarse como

$$\begin{aligned} x_{i+1} &= K_i\left(x_i - y_i d_i 2^{-i}\right) \\ y_{i+1} &= K_i\left(y_i + x_i d_i 2^{-i}\right) \end{aligned} \tag{1.10}$$

en donde, $K_i = \cos\theta_i = \cos\left(\operatorname{arctg} 2^{-i}\right) = \frac{1}{\sqrt{1 + \operatorname{tg}^2\theta_i}} = \frac{1}{\sqrt{1 + 2^{-2i}}}$ y $d_i = \pm 1$ dependiendo del sentido de rotación.

El factor de expansión K_i debe ser multiplicado por las ecuaciones correspondientes a las pseudorotaciones para obtener la rotación real.

El ángulo final de una rotación está determinado por la suma algebraica, según el sentido de rotación, de los ángulos de las rotaciones elementales. Si se elimina K_i de las ecuaciones iterativas, se obtiene un algoritmo basado en sumas y desplazamientos. El factor K_i puede aplicarse al final del proceso como una constante K_n, definida como sigue

$$K_n = \lim_{n\to\infty} \prod_{i=0}^{n} K_i \cong 0{,}6073$$

El valor exacto de K_n depende del número de iteraciones. Se define además

$$A_n = \frac{1}{K_n} = \prod_{i=0}^{n-1} \frac{1}{K_i} = \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}} \cong 1{,}6467$$

Como se explicó anteriormente, en cada paso de iteración, en lugar de decidir si rotar o no, se decide el signo o sentido de la rotación a efectuar. Por lo tanto cada ángulo final se puede representar mediante un vector de signos, en donde cada componente corresponde a un ángulo de la secuencia de ángulos elementales. Dichos ángulos elementales se almacenan en una tabla de búsqueda (Lookup Table, LUT). Con esto último, se modifica el acumulador angular para dar lugar a la ecuación,

$$z_{i+1} = z_i - d_i \operatorname{arctg}\left(2^{-i}\right) \qquad (1.11)$$

Dependiendo del sistema angular con el que se trabaje, se almacenan en la tabla las arcotangentes correspondientes en ese sistema.

El algoritmo CORDIC se opera normalmente en dos modos. El primero, se denomina *rotación* (rotation) que rota el vector de entrada un ángulo específico que se introduce como parámetro. El segundo modo, denominado *vectorización* (vectoring), rota el vector de entrada hacia el eje *X*, acumulando el ángulo necesario para efectuar dicha rotación.

En el caso de una *rotación*, el acumulador angular se inicializa con el ángulo a rotar. La decisión sobre el sentido de rotación en cada paso de iteración, se efectúa para minimizar la magnitud del ángulo acumulado. Por ello, el signo que determina el sentido de rotación, se obtiene del valor de dicho ángulo en cada paso. Para el modo rotación, las ecuaciones son

$$\begin{aligned} x_{i+1} &= x_i - y_i d_i 2^{-i} \\ y_{i+1} &= y_i + x_i d_i 2^{-i} \\ z_{i+1} &= z_i - d_i \operatorname{arctg}\left(2^{-i}\right) \end{aligned} \qquad (1.12)$$

en donde, $d_i = \begin{cases} -1 & ,si\ z_i < 0 \\ 1 & ,si\ z_i \geq 0 \end{cases}$

partiendo de las ecuaciones (1.9), luego de *n* etapas, cuando $z_n \to 0$[1] entonces $\sum_{i=0}^{n-1} \theta_i = z$ para $x = x_0$, $y = y_0$, $z = z_0$ [7] se obtiene

[1] De aquí en más se utiliza w_n para una variable de la forma w_n'', a los efectos de simplificar las expresiones.

$$\begin{aligned} x_n &= A_n\left(x_0 \cos z_0 - y_0 \operatorname{sen} z_0\right) \\ y_n &= A_n\left(y_0 \cos z_0 + x_0 \operatorname{sen} z_0\right) \\ z_n &= 0 \\ A_n &= \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}} \end{aligned} \tag{1.13}$$

Para una *vectorización,* el ángulo ingresado se rota para alinearlo con el eje *X*. Para obtener este resultado, en lugar de minimizar la magnitud del acumulador angular, se minimiza la magnitud del componente *y*, ya que si $y = 0$ entonces el vector se encuentra sobre el eje *X*. Asimismo se utiliza el signo del componente *y* para determinar la dirección de rotación. Si el acumulador angular se inicializa con cero, al final del proceso contendrá el ángulo de rotación adecuado. Por lo tanto se pueden deducir las siguientes ecuaciones

$$\begin{aligned} x_{i+1} &= x_i - y_i d_i 2^{-i} \\ y_{i+1} &= y_i + x_i d_i 2^{-i} \\ z_{i+1} &= z_i - d_i \operatorname{arctg}\left(2^{-i}\right) \end{aligned} \tag{1.14}$$

en donde, $d_i = \begin{cases} -1 & ,si\ y_i \geq 0 \\ 1 & ,si\ y_i < 0 \end{cases}$

partiendo de las ecuaciones (1.9), luego de *n* etapas cuando $y_n \to 0$ entonces $\operatorname{tg}\left(\sum_{i=0}^{n-1}\theta_i\right) = -\frac{y}{x}$ para $x = x_0$, $y = y_0$, $z = z_0$ [7] se obtiene

$$\begin{aligned} x_n &= A_n\sqrt{x_0^2 + y_0^2} \\ y_n &= 0 \\ z_n &= z_0 + \operatorname{arctg}\left(\frac{y_0}{x_0}\right) \\ A_n &= \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}} \end{aligned} \tag{1.15}$$

El algoritmo CORDIC está restringido a ángulos θ tales que $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$ debido a que los ángulos elementales convergen sólo dentro de esos límites [Anexo A] [4]. Para tratar ángulos que caen fuera de ese rango, se puede reducir al primer o cuarto cuadrante.

1.2 Cálculo del seno y del coseno

El modo rotación del algoritmo CORDIC puede utilizarse para calcular el seno y el coseno de un ángulo en forma simultánea. Si se inicializa $y_0 = 0$, de la fórmula (1.13) se obtiene

$$\begin{aligned}
x_n &= A_n x_0 \cos z_0 \\
y_n &= A_n x_0 \operatorname{sen} z_0 \\
z_n &= 0 \\
A_n &= \prod_{i=0}^{n-1} \sqrt{1+2^{-2i}}
\end{aligned} \tag{1.16}$$

Si además $x_0 = \frac{1}{A_n}$, de (1.16) se tiene que

$$\begin{aligned}
x_n &= \cos z_0 \\
y_n &= \operatorname{sen} z_0 \\
z_n &= 0
\end{aligned} \tag{1.17}$$

que calcula el seno y el coseno del ángulo inicial z_0.

1.3 Transformación de coordenadas polares a cartesianas y viceversa

La transformación *T*, que hace corresponder coordenadas polares a cartesianas, se define como

$$T : \begin{cases} x = r \cos\theta \\ y = r \operatorname{sen}\theta \end{cases} \tag{1.18}$$

Si se reemplaza en (1.13), $x_0 = \frac{1}{A_n} r$ (módulo)

$y_0 = 0$

$z_0 = \theta$ (fase)

se obtiene la transformación *T*.

La transformación de coordenadas cartesianas a polares se puede llevar a cabo utilizando la transformación inversa *T'*

$$T' : \begin{cases} r = \sqrt{x^2 + y^2} \\ \theta = \operatorname{arctg}\left(\frac{y}{x}\right) \end{cases} \tag{1.19}$$

Utilizando el modo vectorización de CORDIC, se puede sustituir en (1.15)

$$\begin{aligned}
x_0 &= x \\
y_0 &= y \\
z_0 &= 0
\end{aligned}$$

Sin embargo el algoritmo también computa la constante A_n que puede ser eliminada multiplicando por su inverso, lo cual agrega cierta complejidad al algoritmo inicial ya que se precisa de un multiplicador con el mismo ancho en bits que el resultado.

1.4 Arcotangente y módulo de un vector

El arcotangente se puede calcular utilizando el modo vectorización de CORDIC.
Si se inicializa $z_0 = 0$ y se sustituye en (1.15), se obtiene el término z_n

$$
\begin{aligned}
x_n &= A_n \sqrt{x_0^2 + y_0^2} \\
y_n &= 0 \\
z_n &= z_0 + \operatorname{arctg}\left(\frac{y_0}{x_0}\right) \\
A_n &= \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}}
\end{aligned}
\tag{1.20}
$$

El término x_n representa el módulo de un vector en el espacio bidimensional en donde x_0 e y_0 son sus componentes. Nuevamente para eliminar a la constante A_n se debe agregar al algoritmo un multiplicador.

1.5 CORDIC generalizado

El algoritmo CORDIC básico puede ser generalizado para proveer una herramienta aún mas potente para el cálculo de funciones. Esta generalización fue propuesta por Walther [5]. Se define

$$
\begin{aligned}
x_{i+1} &= x_i - \mu\, y_i d_i 2^{-i} \\
y_{i+1} &= y_i + x_i d_i 2^{-i} \\
z_{i+1} &= z_i - d_i f\left(2^{-i}\right)
\end{aligned}
\tag{1.21}
$$

La única diferencia que presenta este algoritmo con el básico, es la introducción del parámetro μ en la ecuación correspondiente al componente x y la función *f(.)* que adquiere significado según el sistema que se utilice. El parámetro μ y la función *f(.)* pueden asumir uno de los tres valores que se muestran a continuación

$\mu = 1$ y $f(x) = \operatorname{arctg}(x)$	Rotación Circular (CORDIC básico)
$\mu = 0$ y $f(x) = x$	Rotación Lineal
$\mu = -1$ y $f(x) = \operatorname{tgh}^{-1}(x)$	Rotación Hiperbólica

1.5.1 Caso lineal

Si se parte de (1.21) con $\mu = 0$ y $f(x) = x$ se obtiene

$$\begin{aligned} x_{i+1} &= x_i \\ y_{i+1} &= y_i + x_i d_i 2^{-i} \\ z_{i+1} &= z_i - d_i 2^{-i} \end{aligned} \qquad (1.22)$$

Para el modo rotación, donde $d_i = \begin{cases} -1 & ,si\ z_i < 0 \\ 1 & ,si\ z_i \geq 0 \end{cases}$

la rotación lineal produce

$$\begin{aligned} x_n &= x_0 \\ y_n &= y_0 + x_0 z_0 \\ z_n &= 0 \end{aligned} \qquad (1.23)$$

Por lo tanto a partir de estas ecuaciones puede obtenerse el producto de dos valores *a.b*, sustituyendo

$$\begin{aligned} x_0 &= a \\ y_0 &= 0 \\ z_0 &= b \end{aligned}$$

En modo vectorización, donde $d_i = \begin{cases} -1 & ,si\ y_i \geq 0 \\ 1 & ,si\ y_i < 0 \end{cases}$

la rotación lineal produce

$$\begin{aligned} x_n &= x_0 \\ y_n &= z_0 - \frac{y_0}{x_0} \\ z_n &= 0 \end{aligned} \qquad (1.24)$$

Por lo tanto a partir de estas ecuaciones puede obtenerse el cociente de dos magnitudes *a/b*, sustituyendo

$$\begin{aligned} x_0 &= b \\ y_0 &= -a \\ z_0 &= 0 \end{aligned}$$

1.5.2 Caso hiperbólico

Si se parte de (1.21) con $\mu = -1$ y $f(x) = \text{tgh}^{-1}(x)$ se obtiene

$$\begin{aligned} x_{i+1} &= x_i + y_i d_i 2^{-i} \\ y_{i+1} &= y_i + x_i d_i 2^{-i} \\ z_{i+1} &= z_i - d_i \,\text{tgh}^{-1}\left(2^{-i}\right) \end{aligned} \qquad (1.25)$$

Para el modo rotación, donde $d_i = \begin{cases} -1 & , si\ z_i < 0 \\ 1 & , si\ z_i \geq 0 \end{cases}$

la rotación hiperbólica produce

$$\begin{aligned} x_n &= A_n\left(x_0 \cosh z_0 + y_0 \operatorname{senh} z_0\right) \\ y_n &= A_n\left(y_0 \cosh z_0 + x_0 \operatorname{senh} z_0\right) \\ z_n &= 0 \\ A_n &= \prod_{i=0}^{n-1} \sqrt{1-2^{-2i}} \cong 0{,}80 \end{aligned} \qquad (1.26)$$

Para el modo vectorización, donde $d_i = \begin{cases} -1 & , si\ y_i \geq 0 \\ 1 & , si\ y_i < 0 \end{cases}$

la rotación hiperbólica produce

$$\begin{aligned} x_n &= A_n \sqrt{x_0^2 - y_0^2} \\ y_n &= 0 \\ z_n &= z_0 + \operatorname{tgh}^{-1}\left(\frac{y_0}{x_0}\right) \\ A_n &= \prod_{i=0}^{n-1} \sqrt{1-2^{-2i}} \end{aligned} \qquad (1.27)$$

Las rotaciones elementales en el sistema hiperbólico no convergen. Sin embargo Walther [5] demostró que se pueden conseguir los resultados esperados si se repiten algunas de las iteraciones efectuadas [6].

Las funciones trigonométricas hiperbólicas equivalentes a las funciones trigonométricas circulares pueden obtenerse de manera similar a partir de la ecuación 1.26.

1.6 Arquitecturas para implementar el algoritmo CORDIC

1.6.1 La arquitectura Bit-Paralela Iterativa

Cuando el algoritmo CORDIC se implementa en hardware, una de las arquitectura correspondientes es la arquitectura bit-paralela iterativa [3]. La denominación de paralela se debe a la forma en que se opera con las componentes *X*, *Y* y *Z*. En esta arquitectura cada etapa del algoritmo consiste de un registro para almacenar la salida, una unidad de desplazamiento y un sumador algebraico. Al comenzar el cálculo, los valores iniciales para x_0, y_0 y z_0 ingresan en forma paralela a los registros a través del multiplexor. El bit más significativo del componente *Z* ó *Y* en cada paso de iteración determina la operación a efectuar por el sumador algebraico en modo rotación o vectorización respectivamente. Las señales correspondientes a los componentes *X* e *Y* son desplazadas y luego restadas o sumadas a las señales sin desplazar, correspondientes al componente opuesto. El componente *Z* combina los valores almacenados en el registro con valores que obtiene de una tabla de búsqueda (Lookup Table, LUT) de arcotangentes precalculadas con una cantidad de

entradas proporcional a la cantidad de iteraciones. Esta tabla de búsqueda puede ser implementada como una memoria ROM. La dirección de memoria cambia acorde al número de iteración. Para *n* iteraciones el valor a calcular puede ser obtenido en la salida. Se requiere de un controlador que puede ser implementado como una máquina de estados para controlar los multiplexores, la cantidad de desplazamientos y el direccionamiento de las constantes precalculadas. Esta representación tiene como ventaja el uso eficiente de hardware, debido a que los recursos (registros, multiplexores, unidades de desplazamiento y sumadores) son reutilizados en cada iteración. En la figura 1.4 se muestra el esquema correspondiente a la arquitectura descripta. La señal Modo especifica el modo de operación (Rotación o Vectorización). Se considera además en el esquema que las componentes tienen un ancho de *m* bits. Sin embargo se puede ampliar el ancho en bits de los buses internos para obtener una mejor exactitud [6].

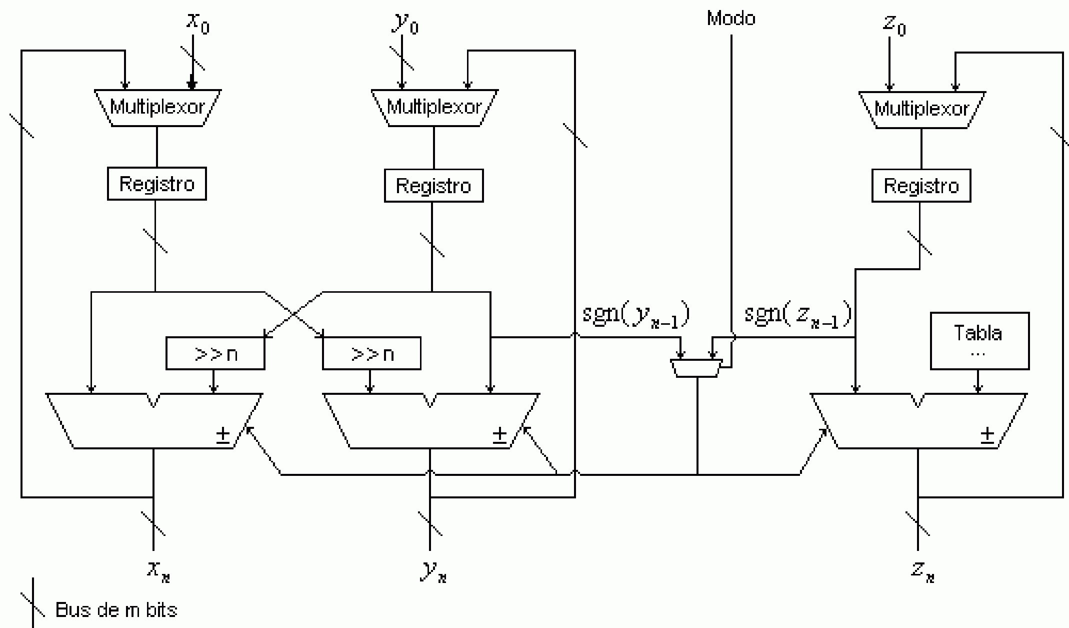


Figura 1.4

1.6.2 La arquitectura Bit-Paralela Desplegada

En lugar de almacenar el resultado de cada paso de iteración en registros y volver a utilizar los mismos recursos, el diseño puede desplegarse [3] como muestra la Figura 1.5. El diseño se separa en etapas correspondientes a cada iteración. Cada etapa está compuesta por los mismos componentes, dos unidades de desplazamiento y dos sumadores algebraicos. Por consiguiente la salida de una etapa corresponde a la entrada de la siguiente etapa. Los valores iniciales para x_0, y_0 y z_0 se ingresan en paralelo a la primera etapa.

Este diseño introduce dos ventajas importantes. Las unidades de desplazamiento así como las constantes correspondientes a cada iteración pueden ser cableadas. La segunda ventaja radica en que el circuito es puramente combinatorio, y por lo tanto no se necesita de una unidad de control, simplificando enormemente el diseño. La desventaja que presenta esta arquitectura es la enorme cantidad de espacio que requiere su implementación.

En el esquema se supone un ancho de palabra de *m* bits. La señal Modo indica el modo de operación al igual que en el caso iterativo.

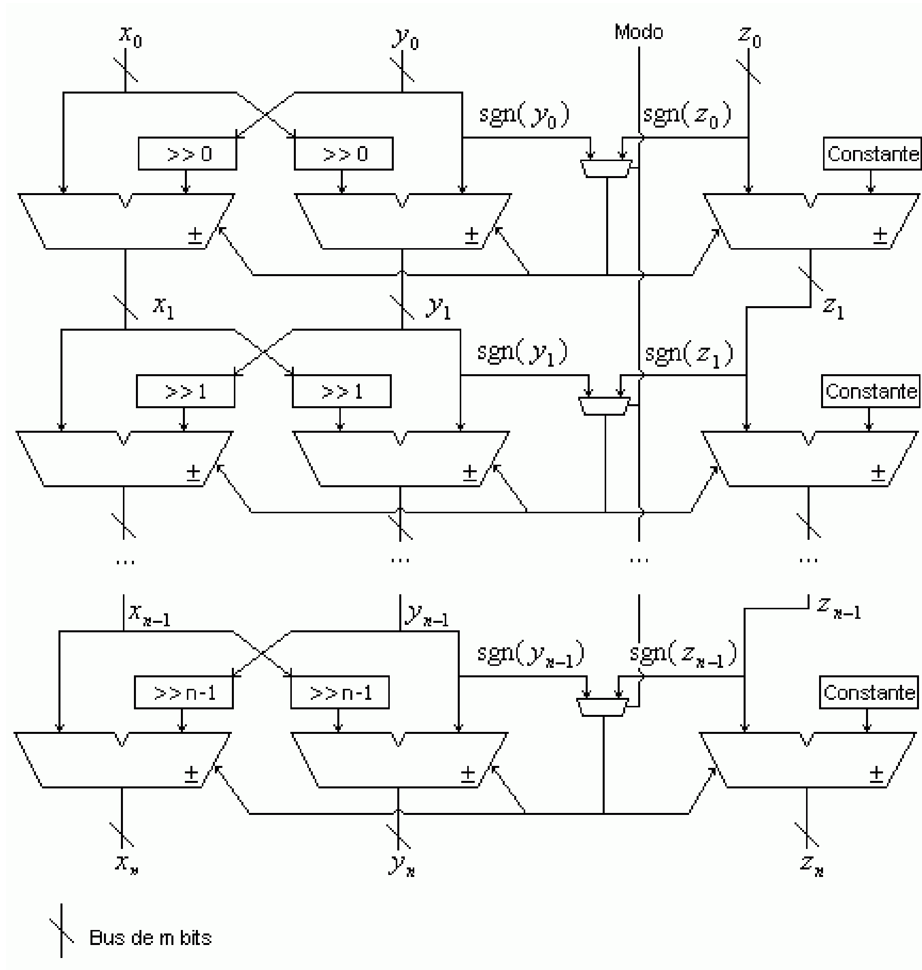


Figura 1.5

1.6.3 La arquitectura Bit-Serie Iterativa

En un diseño en serie, se procesa un bit por vez, con lo cual las interconexiones se reducen al ancho de un bit [1] [3]. El desempeño δ de esta arquitectura se puede calcular como

$$\delta = \frac{velocidad\ de\ reloj}{número\ de\ iteraciones \cdot ancho\ de\ palabra}$$

La arquitectura resultante se muestra en la Figura 1.6. Cada etapa del algoritmo consiste en un multiplexor, un registro de desplazamiento y un sumador algebraico bit-serie. El sumador algebraico se implementa como un sumador completo en el que una resta se lleva a cabo sumando el complemento a dos del valor a restar. La operación a efectuar por el sumador algebraico se obtiene del bit de signo del componente *z* para el modo rotación. Para el modo vectorización, del bit de signo se obtiene del componente *y*.

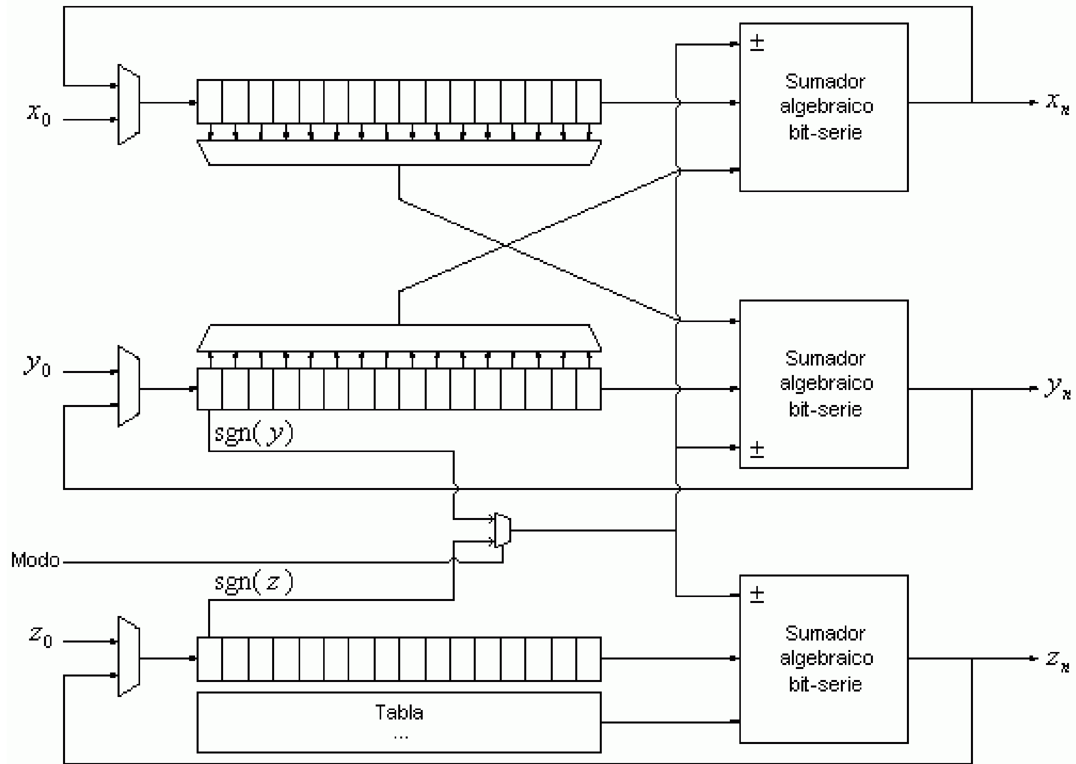


Figura 1.6

La operación de desplazamiento (multiplicación por 2^{-i}) se lleva a cabo leyendo el bit *i – 1* considerado a partir del extremo derecho del registro de desplazamiento. Se puede usar un multiplexor para cambiar la posición de acuerdo a la iteración actual. Los valores iniciales para x_0, y_0 y z_0 ingresan al registro de desplazamiento por el extremo izquierdo en el esquema. Cuando el primer bit que fue ingresado al registro es procesado por el sumador algebraico, el multiplexor permite nuevamente el ingreso de los bits sumados al registro de desplazamiento. Por último, cuando se han completado todas las iteraciones los multiplexores permiten el ingreso de un nuevo valor al registro de desplazamiento y el valor calculado se obtiene en la salida. Alternativamente la carga y lectura del registro de desplazamiento puede efectuarse en paralelo para simplificar el diseño. La desventaja que presenta esta arquitectura con respecto a las que procesan los vectores de entrada en forma paralela es que se introduce un retardo proporcional al ancho de palabra en cada etapa, ocasionado por los desplazamientos. Por otra parte se requiere de hardware de control mas complejo. La ventaja que presenta esta arquitectura es que los buses de interconexión entre los componentes son del ancho de un bit y el registro de desplazamiento está integrado con el registro intermedio, minimizando espacio al momento de su implementación.

Capítulo 2

Metodologías de diseño de hardware

Las metodologías de diseño de hardware denominadas Top-Down, basadas en la utilización de lenguajes de descripción de hardware, han posibilitado la reducción de los costos en la fabricación de circuitos integrados. Esta reducción se debe a la posibilidad de describir y verificar el funcionamiento de un circuito mediante la simulación del mismo, sin necesidad de implementar un prototipo físicamente.

2.1 Herramientas CAD

La metodología de diseño asistida por compuatdora (Computer Aided Design, CAD), emplea técnicas gráficas para soportar el proceso de diseño. La introducción de dichas técnicas en el proceso de diseño de circuitos electrónicos es fundamental, ya que mas allá de proveer interfaces gráficas para asistir el proceso, brinda la posibilidad de simular y verificar la descripción antes de llevar a cabo su implementación, minimizando el costo de elaborar circuitos potencialmente defectuosos y acelerando el diseño global [20].

El diseño de hardware tiene un problema fundamental, que no existe en el diseño de software. Este problema es el alto costo del ciclo de diseño-prototipación-verificación (figura 2.1), ya que el costo del prototipo por lo general es bastante elevado.

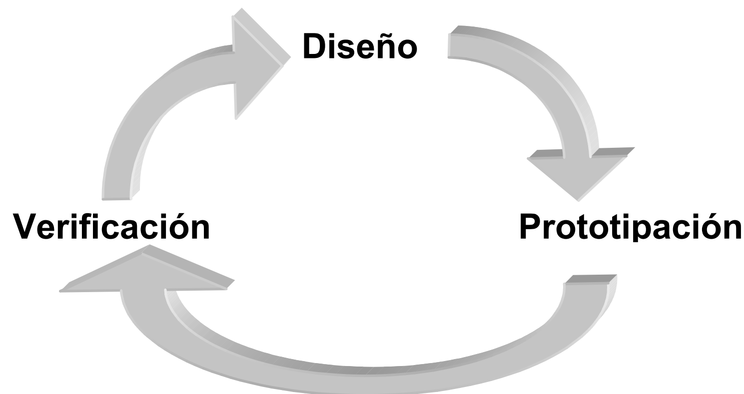


Figura 2.1

Por lo tanto se busca minimizar el costo del ciclo anterior, incluyendo la fase de prototipación únicamente al final del mismo. Esto se consigue mediante la inclusión de una fase de simulación y verificación que elimina la necesidad de elaborar físicamente un prototipo.

En el ciclo de diseño hardware las herramientas CAD están presentes en todos los pasos. En primer lugar en la fase de descripción de la idea, que será un sistema eléctrico, un diagrama en bloques, etc. Luego en la fase de simulación y verificación en donde las diversas herramientas permiten realizar simulación por eventos, funcional, digital o eléctrica considerando el nivel de simulación requerido. La última etapa es comprendida por herramientas especializadas en la fabricación del circuito propiamente dicho y se orientan a la fabricación de circuitos impresos o Circuitos Integrados de Aplicación Específica (Application Specific Integrated Circuits, ASIC). Estas herramientas permiten realizar microcircuitos así como la programación de dispositivos que así lo requieran.

A continuación se enumeran y explican brevemente algunas de la posibles herramientas que pueden utilizarse durante el diseño e implementación del hardware:

Descripción mediante esquemas: Consiste en describir el circuito mediante un esquema que representa la estructura del sistema. Mas allá de un simple diagrama de líneas puede incluir información sobre tiempos, referencias, cables, conectores, etc.

Grafos y diagramas de flujo: La descripción se realiza por medio de grafos, autómatas o redes de Petri. La diferencia con la captura de esquemas es que este tipo de descripción es funcional o de comportamiento y no de estructura como sucede en el caso anterior.

Lenguajes de descripción: Son lenguajes de computadora especializados que permiten describir un circuito digital. Esta descripción usualmente se puede llevar a cabo a diferentes niveles. Puede ser estructural, en donde se muestra la arquitectura del diseño, o bien de comportamiento, en donde se describe el comportamiento o funcionamiento del circuito global y no de los componentes por los cuales está compuesto.

Simulación de sistemas: Estas herramientas se utilizan para la simulación global del sistema. Los componentes que se simulan son de alto nivel, es decir del producto una vez terminado.

Simulación funcional: Este tipo de simulación se utiliza para validar el funcionamiento de un sistema digital a bajo nivel (nivel de compuertas), sin embargo no se toman en consideración factores físicos de los componentes a simular como ser retrasos, problemas eléctricos, etc. Unicamente de registra el comportamiento del circuito frente a ciertos estímulos dados.

Simulación digital: Esta simulación es muy parecida a la simulación funcional, pero considerando los retrasos y factores que no se consideran en la anterior. De esta forma se garantiza el funcionamiento correcto del circuito digital a ser implementado.

Simulación eléctrica: Es la simulación de mas bajo nivel ya que se realiza a nivel de componentes básicos (transistores, resistencias, etc). El resultado de dicha simulación es prácticamente el mismo que en la realidad. Se utiliza tanto para circuitos analógicos como digitales.

Impementación de circuitos impresos: Con estas herramientas se realiza el trazado de líneas e implementación posterior de los circuitos impresos en donde irán montados los componentes.

Implementación de circuitos integrados: Son las herramientas que se emplean al final del ciclo de implementación. Permiten la realización de diferentes máscaras que intervienen en la implementación del circuito final.

Programación de dispositivos: Alternativamente a la implementación de los circuitos mediante máscaras, se puede emplear lógica programable. Los dispositivos de lógica programable permiten la implementación el circuito mediante la programación de los mismos. Posteriormente pueden ser reutilizados en caso de querer modificar el diseño o el circuito por completo. Ejemplos de dichos dispositivos son: PAL (Programmable And Logic), FPGA (Field Programmable Gate Arrays) y PLD (Programmable Logic Devices).

2.2 Diseño Bottom-Up

Esta metodología de diseño comprende la descripción del circuito mediante componentes que pueden agruparse en diferentes módulos, y éstos últimos a su vez en otros módulos hasta llegar a representar el sistema completo que se desea implementar, como muestra la figura 2.2.

La metodología Bottom-Up no implica una estructuración jerárquica de los elementos del sistema. Simplemente reúne componentes de bajo nivel para formar el diseño global.

En un diseño Bottom-Up se comienza realizando una descripción con esquemas de los componentes del circuito. Estos componentes se construyen normalmente a partir de otros que pertenecen a una biblioteca que contiene componentes básicos, que representan unidades funcionales con significado propio dentro del diseño. Estas unidades son denominadas primitivas, ya que no es necesario disponer de elementos de mas bajo nivel para el diseño que se desea realizar [20].

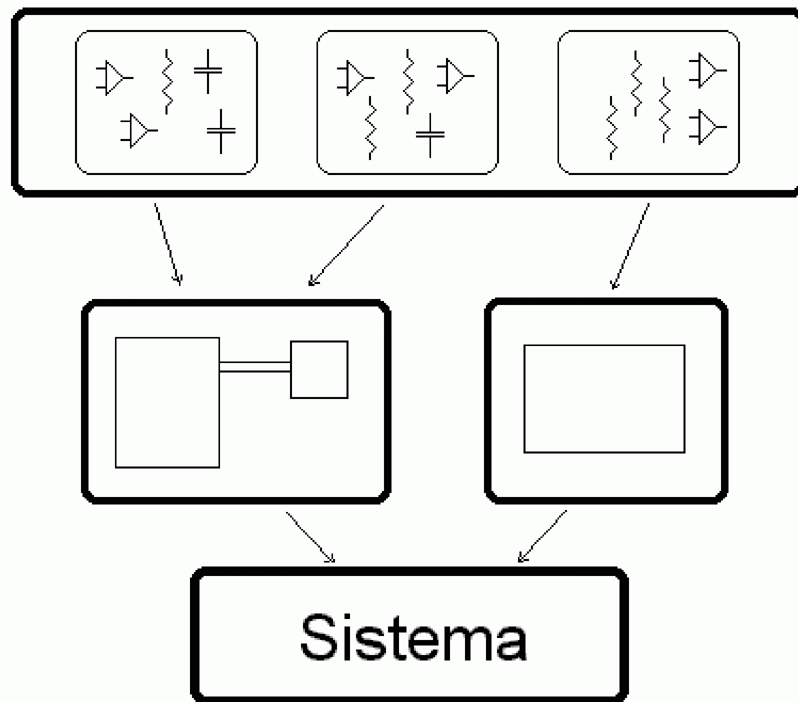


Figura 2.2

Esta metodología de diseño es útil para diseños medianamente pequeños. Para diseños mayores ésta metodología resulta impráctica, debido a que el funcionamiento adecuado del producto final no está garantizado luego de unir decenas de componentes. El hecho de unir un número elevado de componentes entre sí, sin una estructura jerárquica que permita organizarlos (en bloques, por ejemplo) dificulta el análisis del circuito, aumentando la posibilidad de cometer errores.

La metodología Bottom-Up es la que se utiliza desde los primeros tiempos de diseño, ya que el proceso de integración de elementos básicos se había logrado automatizar completamente. Las primeras herramientas de diseño permitían llevar a cabo una descripción sencilla a bajo nivel y posteriormente se procedía a la implementación. La implementación se realizaba empleando otras herramientas que se integraban al proceso. De esta manera se obtenía un Circuito Integrado de Aplicación Específica o un Circuito Impreso (Printed Circuit Board, PCB).

2.3 Diseño Top-Down

El diseño Top-Down consiste en capturar una idea con un alto nivel de abstracción, implementarla partiendo de la misma, e incrementar el nivel de detalle según sea necesario. El sistema inicial se va subdividiendo en módulos, estableciendo una jerarquía. Cada módulo se subdivide cuantas veces sea necesario hasta llegar a los componentes primarios del diseño como muestra el esquema de la figura 2.3.

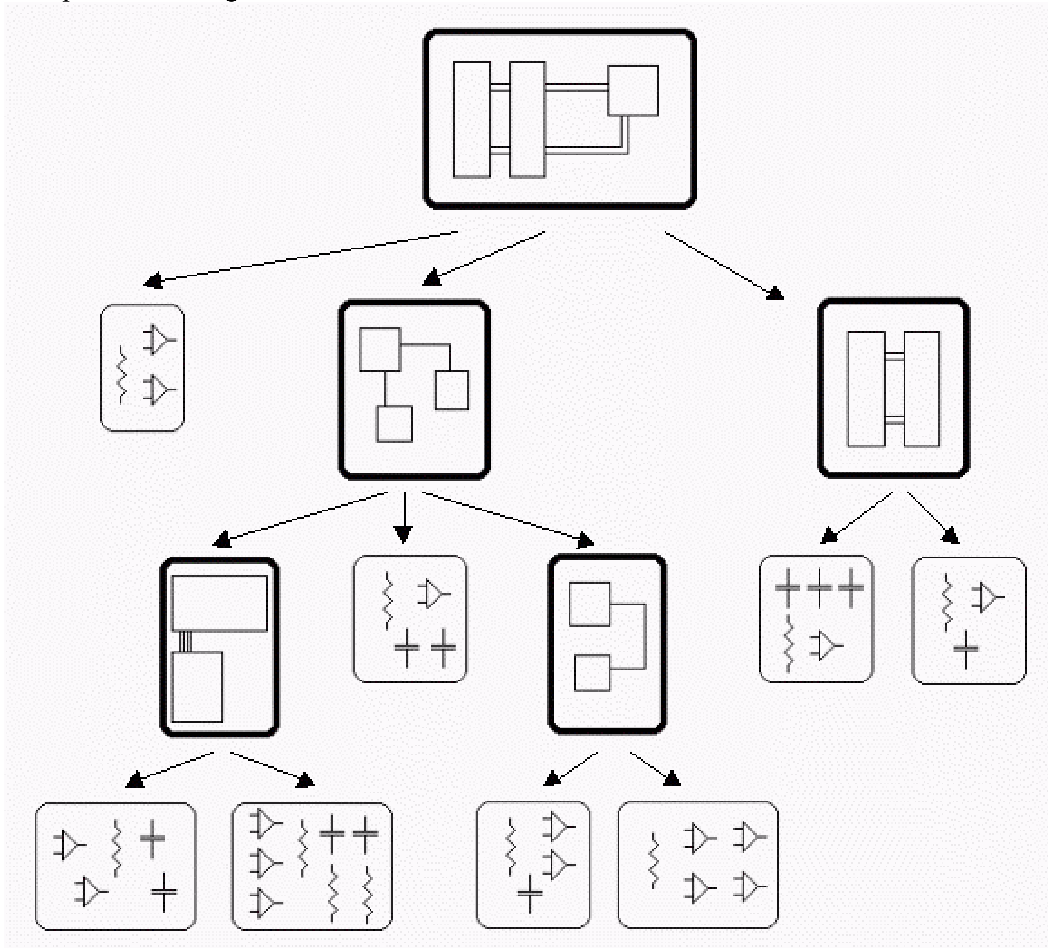


Figura 2.3

Sin embargo actualmente, es necesario realizar diseños más complejos en menos tiempo. De ésta manera se deja de lado la metodología de diseño Bottom-Up.

La metodología Top-Down evita los problemas que surgen con el empleo de la metodología Bottom-Up ya que el diseño inicial es subdividido en subdiseños que a su vez se pueden seguir subdividiendo hasta llegar a diseños mucho menores y más sencillos de tratar. En el caso del diseño de hardware, esto se traduciría en subdividir el diseño inicial en módulos hasta llegar a los componentes primarios o primitivas.

Las herramientas actuales permiten utilizar en forma automática la metodología Top-Down, lo que permite a las herramientas de síntesis sofisticadas llevar a cabo la implementación de un circuito final, partiendo de una idea abstracta y sin necesidad de que el diseñador deba descomponer su idea inicial en componentes concretos [19] [20].

2.4 Ventajas del diseño Top-Down

Una de las principales ventajas del diseño Top-Down es que el diseñador puede especificar el diseño en un alto nivel de abstracción sin necesidad de considerar el mismo inicialmente a nivel de compuertas. Las herramientas incluidas en el paquete de VHDL, podrían generar el esquema de compuertas lógicas correspondientes a una descripción funcional dada. Sin embargo las herramientas de síntesis actuales aún son incapaces de traducir ciertas descripciones de comportamiento en una descripción a nivel de compuertas, por ello la especialización y el pasaje entre los distintos niveles se puede realizar manualmente, refinando el diseño. La capacidad de sintetizar descripciones funcionales puras se irá introduciendo en el lenguaje con el correr del tiempo.

En el proceso de diseño se utilizan tecnologías genéricas, lo que posibilita que la tecnología de implementación no se fije hasta los últimos pasos del proceso. De ésta manera se pueden reutilizar los datos del diseño únicamente cambiando la tecnología de implementación.

La descripción del circuito a distintos niveles de detalle, así como la verificación y simulación del mismo, permiten reducir la posibilidad de incluir errores.

Diseño modular: El diseño Top-Down ofrece como ventaja que la información se estructura en forma modular. Como el diseño se realiza a partir del sistema completo y se subdivide en módulos, permite que las subdivisiones se realicen de forma que los mismos sean funcionalmente independientes. El diseño Bottom-Up no contempla la división en partes funcionalmente independientes. Esta es la desventaja fundamental del diseño Bottom-Up. El resultado final puede aparecer confuso al no estar dividido en módulos independientes.

Diseño jerárquico: En un diseño electrónico entran en juego una cantidad considerable de componentes. Estos diseños deben organizarse de tal forma que resulte fácil su comprensión. Una forma de organizar el diseño es la creación de un diseño modular jerárquico. Un diseño jerárquico está constituido por niveles en donde cada uno es una especialización del nivel superior. La organización jerárquica es una consecuencia directa de aplicar la metodología Top-Down.

2.5 Descripción de un diseño

Luego de concebir la idea del circuito que se pretende diseñar, se debe realizar la descripción del mismo.

En un principio las herramientas CAD, brindaban únicamente la posibilidad de trazar los dibujos referentes al diseño. El diseñador realizaba la descripción sobre un papel utilizando componentes básicos y trasladaba el diseño a la computadora para obtener una representación más ordenada. Con la llegada de computadoras con mayor capacidad de cálculo y herramientas mas sofisticadas, no sólo se realiza el dibujo del circuito, sino su descripción completa y la simulación del mismo, para prever el comportamiento aparente que tendrá una vez implementado. Las herramientas de diseño modernas permiten describir un circuito a distintos niveles de abstracción y es la computadora la que lleva a cabo la idea en forma concreta [17].

Básicamente, las herramientas actuales permiten dos tipos de descripciones:

Descripción comportamental: Se describe el comportamiento del circuito, sin poner énfasis en su arquitectura. Dicha descripción se realiza mediante un lenguaje de hardware específico. No se especifican señales ni elementos de bajo nivel.

Descripción estructural: Consiste en enumerar los componentes de un circuito y sus interconexiones. Se puede llevar a cabo mediante esquemas, en cuyo caso se realiza una descripción gráfica de los componentes del circuito, o bien mediante un lenguaje, en cuyo caso se enumeran los componentes del circuito y sus interconexiones.

Capítulo 3

El lenguaje de descripción de hardware VHDL

VHDL es un lenguaje diseñado para describir sistemas electrónicos digitales. Surgió del programa VHSIC (Very High Speed Integrated Circuits) impulsado por Departamento de Defensa del gobierno de los Estados Unidos de América. Durante el transcurso del programa se hizo notoria la falta de un lenguaje para describir circuitos electrónicos. De esta forma se desarrolló el lenguaje VHDL (VHSIC Hardware Design Language) que fue estandarizado inicialmente en el año 1987 por el Instituto de Ingenieros Eléctricos y Electrónicos (Institute of Electrical and Electronics Engeneers, IEEE) en los Estados Unidos de América. Dicho estándar se conoce como IEEE Std 1076-1987. Posteriormente en el año 1993 se revisó este estándar originando el estándar conocido como IEEE Std 1076-1993 con algunos cambios respecto del anterior. El lenguaje VHDL está diseñado para cubrir varias necesidades que surgen durante el proceso de diseño. Permite realizar una descripción funcional o de comportamiento del circuito, utilizando técnicas procedurales y familiares de programación. Permite describir la estructura del diseño y declarar las entidades y subentidades que lo forman especificando una jerarquía entre las mismas y como son sus interconexiones. Por último permite simular el diseño y sintetizarlo con herramientas de síntesis especiales, permitiendo su manufacturado y encapsulado como un microcircuito. Un aspecto que resulta importante destacar es que gracias a este tipo de lenguajes, se elimina la fase de prototipación de componentes lo cual reduce los costos de diseño y producción.

3.1 VHDL describe comportamiento

Cuando comienza la fase de diseño de un sistema electrónico digital, resulta útil realizar una descripción funcional o de comportamiento del mismo. Una descripción VHDL usualmente está compuesta por un conjunto de entidades y subentidades, algunas de las cuales se pueden adquirir manufacturadas. De esta forma se elimina la necesidad de diseñarlas por completo. Sin embargo su comportamiento resulta necesario durante la fase de simulación del sistema. En este punto es conveniente realizar una descripción funcional del componente, que podrá ser utilizada como versión previa para simular el sistema final. La descripción funcional no hace referencia a la estructura interna del componente, que es visto como una caja negra, sino sólo se refiere a su funcionamiento.

Muchas veces la descripción funcional se divide a su vez en dos, dependiendo del nivel de abstracción y del modo en que se ejecutan las instrucciones. Estas dos formas se denominan *algorítmica* y de *flujo de datos*.

3.2 VHDL describe estructura

Un sistema electrónico digital puede ser descripto como un módulo con puertos de entrada y de salida como muestra la figura 3.1. Los valores de los puertos de salida son una función de los valores de los puertos de entrada y del estado del sistema.

Figura 3.1

Una manera de describir un componente o sistema digital es a través de los componentes que lo forman. Cada componente es la instancia de alguna entidad. Sus respectivos puertos están interconectados por señales (figura 3.2). Este tipo de descripción se denomina *estructural* o *de estructura*.

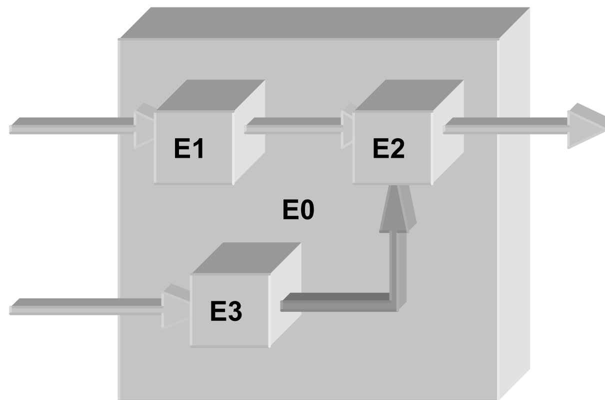


Figura 3.2

3.3 Un ejemplo de descripción en VHDL

El lenguaje VHDL presenta tres estilos de descripción que dependen del nivel de abstracción. El menos abstracto es el nivel estructural mientras que el mas abstracto y lejano a una posible implementación física es el algorítmico. A modo de ejemplo, para ilustrar cada uno de los estilos, se describirá un multiplexor de dos bits, cuya descripción se reutilizará en el trabajo final. Un posible esquema del multiplexor se muestra en la figura 3.3.

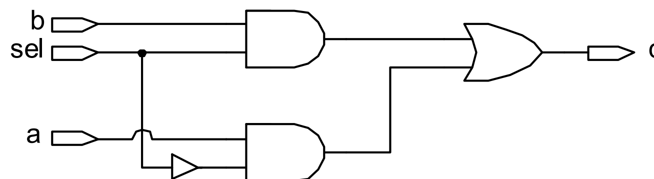


Figura 3.3

La descripción de un circuito en VHDL comienza con la definición de la entidad. Lo primero que se definen son sus entradas y salidas. Se denomina entidad porque en VHDL la palabra clave que se utiliza para su definición es `ENTITY`.

```
ENTITY Mux IS

      PORT( a:    IN bit;
            b:    IN bit;
            sel:  IN bit;

            c:    OUT bit
      );

END Mux;
```

La entidad definida con el nombre `Mux` tiene tres entradas y una salida. Las entradas y la salida son de tipo `bit`. El tipo `bit` es un tipo predefinido del lenguaje que contiene los valores '0' y '1'.

Para este ejemplo la entidad en VHDL es única, ya que se describe el mismo esquema del multiplexor, con la finalidad de ilustrar los distintos estilos, sin embargo esto no es necesario.

La entidad anteriormente declarada tiene asociados tres cuerpos descriptos con diferentes estilos. Estos cuerpos se denominan arquitecturas y se definen en VHDL mediante la palabra clave `ARCHITECTURE`. A continuación se muestran éstas arquitecturas. Cada arquitectura está descripta con un estilo distinto, comenzando por la descripción algorítmica, pasando por la descripción de flujo de datos para culminar con la descripción estructural.

En la descripción algorítmica no se hace referencia a los operadores lógicos presentes en el esquema del multiplexor, sino sólo al funcionamiento del mismo. Con una secuencia sencilla de instrucciones se puede describir el circuito. La forma en que se ejecutan las descripciones algorítmicas es secuencial, como se verá mas adelante en este capítulo, por eso se le da la denominación de algorítmica para diferenciarla de las descripciones concurrentes. La ejecución secuencial se hace posible mediante la inclusión de la palabra clave `PROCESS`. Sin embargo ésta no es la única construcción secuencial que posee VHDL, otras construcciones de naturaleza secuencial son las funciones y los procedimientos, estructuras de alto nivel que también pueden emplearse para realizar descripciones funcionales.

```
ARCHITECTURE Algoritmica OF Mux IS
BEGIN
      PROCESS(a, b, sel)
      BEGIN
            IF (sel='1') THEN
                  c<=b;
            ELSE
                  c<=a;
            END IF;
      END PROCESS;

END Algoritmica;
```

Una descripción algorítmica de mas alto nivel puede realizarse utilizando un procedimiento. Este tipo de descripción es la más lejana a una implementación física. Una descripción procedural del multiplexor puede ser de la siguiente forma

```
PROCEDURE Mux(a,b,sel:IN bit;c:OUT bit) IS
BEGIN
      IF (sel='1') THEN
            c:=b;
      ELSE
            c:=a;
      END IF;

END Mux;
```

A veces resulta interesante describir el circuito de forma que esté mas cercano a una posible implementación física. El lenguaje VHDL posee una forma de describir circuitos que permite la paralelización de instrucciones, y a su vez es más parecida a una descripción estructural pero que se clasifica como de comportamiento. Todas las sentencias que se encuentran entre el `BEGIN ... END` del cuerpo de la arquitectura se ejecutan en forma paralela y mientras dure la simulación. Esta es una característica importante frente a los lenguajes de programación convencionales, en los que las sentencias a iterar deben encerrarse dentro de un bucle. VHDL soporta la ejecución concurrente por omisión en el cuerpo de una arquitectura. El estilo que se presenta a continuación se denomina de flujo de datos.

```
ARCHITECTURE FlujoDeDatos OF Mux IS
BEGIN
      c<=(a and (not sel)) or (b and sel);

END FlujoDeDatos;
```

Una descripción estructural es la más cercana a la implementación física de un circuito y hace referencia a los componentes que intervienen y a sus interconexiones. Los componentes que se utilizan son declarados en la parte declarativa de la arquitectura e instanciados en el cuerpo de la misma las veces que sea necesario.

```
ARCHITECTURE Estructural OF Mux IS

      COMPONENT not IS

            PORT( a:    IN bit;
                  b:    OUT bit
                  );

      END COMPONENT;

      COMPONENT and IS

            PORT( a:    IN bit;
                  b:    IN bit;
                  c:    OUT bit
                  );

      END COMPONENT;
```

```
		COMPONENT or IS

			PORT( a:	IN bit;
			      b:	IN bit;
			      c:	OUT bit
			      );

		END COMPONENT;

	SIGNAL x,y,z:bit;

	BEGIN
		Not1: not PORT MAP(sel,x);
		And1: and PORT MAP(b,sel,y);
		And2: and PORT MAP(a,x,z);
		Or1: or PORT MAP(y,z,c);

	END Estructural;
```

Una descripción en VHDL puede ser mixta, es decir puede comprender varios estilos. Por ejemplo, en una descripción estructural, los elementos mas bajos de la jerarquía de diseño pueden definirse en forma de flujo de datos, debido a que el lenguaje incluye un conjunto de operadores predefinidos, como es el caso de los operadores lógicos, que pueden emplearse en la descripción.

3.4 Modelo de tiempo basado en eventos discretos

Una vez que se ha descripto un sistema, se debe realizar una simulación del mismo. Esto se lleva a cabo simulando el pasaje del tiempo en etapas discretas. En algún momento del tiempo de simulación, un módulo es estimulado cambiando los valores de sus señales de entrada. El módulo responde modificando sus señales de salida como función de su entrada a través de su comportamiento, que está caracterizado por su estado interno. Los nuevos valores para las señales de salida, serán visibles a su entorno en un paso discreto de simulación posterior. Si el nuevo valor de la señal, es distinto al que poseía previamente, se dispara un evento que estimula a otro módulo y así sucesivamente.

La simulación comienza con una fase de inicialización en la cual a las señales se les da un valor inicial, se ejecutan las descripciones de los módulos que intervienen en el diseño y el tiempo de simulación se pone a cero. Todas las señales que deben ser activadas en ese instante de tiempo son ejecutadas y simuladas, lo que ocasiona que se disparen eventos en otras señales como consecuencia de la simulación. Luego de la fase inicial, todos los módulos que se han estimulado en la fase inicial ejecutan el cuerpo de su descripción ocasionando cambios en sus señales de salida. Una vez que se ha completado un paso discreto de tiempo, la simulación se repite en el próximo paso.

El objeto de la simulación, es verificar y reunir información acerca del funcionamiento completo del sistema. Usualmente se lleva a cabo en un ambiente controlado con ciertas facilidades para inspeccionar el valor de las señales a lo largo de la simulación y en instantes de tiempo determinados.

3.5 VHDL como lenguaje de programación

El comportamiento de un módulo puede ser descripto haciendo uso de técnicas convencionales de programación que brinda el lenguaje VHDL. La sintaxis del lenguaje se ha basado en el lenguaje de programación ADA que también es un estándar definido por el Departamento de Defensa de los Estados Unidos de América. La elección de utilizar el lenguaje ADA como modelo para VHDL proviene del hecho de que ADA tiene una orientación hacia sistemas de tiempo real y hardware sin limitación de la arquitectura.

3.6 Elementos de sintaxis

3.6.1 Comentarios

Los comentarios en VHDL son cadenas de caracteres alfabéticos y números que comienzan con dos guiones '--' y se extienden hasta el final de la línea actual. Son ignorados por el lenguaje ya que su propósito es precisamente el de introducir aclaraciones.

```
-- Esto es un comentario
```

3.6.2 Identificadores

Los identificadores en VHDL están constituidos por las palabras reservadas del lenguaje y los que define el programador. Los identificadores pueden estar formados por letras, números y el carácter especial '_' (guión bajo). Deben empezar siempre con una letra y el lenguaje no es sensible a mayúsculas y minúsculas.

3.6.3 Números

Los números se pueden representar en base diez o en cualquier base que sea potencia de dos entre dos y dieciséis, sin embargo esto debe indicarse explícitamente por medio del carácter '#' (numeral). Si el número incluye un punto '.' Representa un número real, de lo contrario representa un número entero. Los dígitos de un número entero pueden separarse por medio del carácter '_'. Los números reales pueden contener un exponente al final, precedido por la letra 'E'.

Ejemplos:

```
0     6     12_345_890  47.987_9         -- Números en base diez

2.89E-8                                  -- Número real en donde el
                                         -- punto se desplaza 8
                                         -- lugares hacia la izquierda

1.77E+6                                  -- Número real en donde el
                                         -- punto se desplaza 6
                                         -- lugares hacia la derecha

2#11000010#       2#1.000_111#           -- Números en binario
4#2312#           4#23.122#              -- Números en base cuatro
16#4BAC#          16#4F.8_9#             -- Números en hexadecimal
```

3.6.4 Caracteres

Los caracteres están constituidos por un carácter ASCII encerrado entre comillas simples ' '.

Ejemplos:

```
'a'   'A'   '0'   '%'
```

3.6.5 Cadenas de caracteres (Strings)

Los Strings se construyen encerrando una secuencia de caracteres ASCII entre comillas dobles. Para incluir una comilla dentro del String, se deben incluir dos comillas seguidas.

Ejemplos:

```
"Esto es un String"
"Este String contiene una comilla antes del final """
"890*"
```

3.6.6 Cadenas de bits

Las cadenas de bits se utilizan para representar arreglos compuestos por los caracteres '0' y '1'. Las cadenas de bits se encierran entre comillas dobles. Además es posible definir cadenas de bits en base dieciséis, ocho o dos precediendo el valor a codificar con las letras `X`, `O` o `B` respectivamente.

Ejemplos:

```
X"4BFF"    O"2320"    B"10011110"
```

3.6.7 Tipos de datos del lenguaje

VHDL proporciona tipos de datos básicos y formas de combinarlos para construir tipos compuestos. Los tipos básicos incluyen números, magnitudes físicas y enumerativos. Existen tipos predefinidos estándar. Los tipos compuestos que provee el lenguaje son arreglos, registros, punteros y archivos. Los tipos se declaran con la palabra clave `TYPE`.

3.6.7.1 Tipo entero

Una variable de tipo entero toma valores de tipo entero en un rango especificado. Un identificador declarado como entero puede asumir un valor dentro el rango –214783648 a 214783647. Los rangos pueden ser ascendentes o descendentes.

Ejemplos:

```
TYPE Byte IS RANGE 0 TO 255;        -- Declara un tipo BYTE con
                                    -- rango ascendente de 0 a
                                    -- 255
TYPE Index IS RANGE 31 DOWNTO 0;    -- Declara un tipo Index con
                                    -- rango descendente de 31 a 0
```

Existe además un tipo predefinido denominado `Integer`. El rango de este tipo viene definido por la implementación que se utilice puede variar de un compilador a otro. Por lo tanto se podría definir al tipo entero como sigue:

```
TYPE Integer IS RANGE Implementation_defined;
```

en donde el rango `Implementation_defined` va del valor entero mínimo al máximo soportado por la implementación del lenguaje.

3.6.7.2 Tipos físicos

Un tipo de datos físico es un tipo numérico que se puede utilizar para representar alguna magnitud física como por ejemplo tiempo, o tensión eléctrica. La declaración de un tipo físico incluye una magnitud denominada base y un conjunto de subunidades que son múltiplos de la unidad base.

Ejemplo:

```
TYPE Clockunits IS
      UNITS
            seg;
            min = 60 seg;
            hs = 60 min;
      END UNITS;
```

que define el tipo de la unidad de tiempo de un reloj convencional. Para asignar a un identificador declarado de este tipo, se puede especificar una unidad de tiempo completa, como por ejemplo `2 hs 30 min 0 seg`.

3.6.7.3 Tipos de punto flotante

Un tipo de punto flotante es una aproximación discreta al conjunto de los números reales en un rango determinado. Un tipo declarado flotante debe llevar siempre un punto, de lo contrario sería considerado por el lenguaje como entero. El rango máximo de números en punto flotante debe incluir al menos 6 decimales y está determinado por la implementación.

Ejemplos:

```
TYPE Real IS RANGE –1.0E38 TO 1.0E38;          -- Tipo Real predefinido
TYPE Wave IS RANGE -32768.0 TO 32767.0;
TYPE Probability IS RANGE 0.0 TO 1.0;
```

3.6.7.4 Tipo enumerativo

Un tipo enumerativo o enumerado es un conjunto ordenado de identificadores. Cada miembro de ese conjunto ordenado debe ser distinto dentro de una misma declaración.

Ejemplos:

El tipo de datos carácter `Character` es un enumerativo que incluye a todos los caracteres ASCII.

```
TYPE Bit IS ('0','1');
TYPE Boolean IS (True,False);
TYPE Hexdigit
      IS ('0','1','2','3','4','5','6','7','8','9',A,B,C,D,E,F);
```

3.6.7.5 Tipo arreglo

Un arreglo es una colección homogénea de datos indexados. Los arreglos pueden ser de una sola dimensión o varias. Además pueden ser restringidos, lo que significa que los límites se fijan en el momento de la declaración o irrestrictos en cuyo caso los límites se determinan mas adelante en tiempo de ejecución o declaración de identificadores. Un arreglo está formado por un rango discreto y un tipo de datos que indica el tipo de los elementos por los cuales está formado el arreglo.

Ejemplos:

```
TYPE Vector IS ARRAY(1 to 100) OF Integer;
TYPE Matriz_booleana
      IS ARRAY(Integer RANGE <>, Integer RANGE <>) OF Boolean;
```

En el primer caso, se define un arreglo de 100 enteros, en el segundo una matriz booleana de rango irrestricto de enteros. Por ejemplo un objeto puede ser declarado como `Matriz_Booleana` de la siguiente forma:

```
Matriz_adyacencia:Matriz_booleana(1 TO 10, 1 TO 10);
```

lo que declara una matriz de adyacencia de 10 filas por 10 columnas.

El lenguaje presenta dos tipos arreglo predefinidos irrestrictos, el tipo `String` que se declara como

```
TYPE String IS ARRAY(Natural RANGE <>) OF Character;
```

y el tipo vector de bits

```
TYPE Bit_vector IS ARRAY(Natural RANGE <>) OF Bit;
```

Un elemento de un arreglo puede ubicarse indexando el nombre del arreglo. Si se supone que `A` y `B` son arreglos de una y dos dimensiones respectivamente entonces los objetos indexados `A(3)` y `B(1,2)` se refieren a elementos indexados de los arreglos. De hecho se puede acceder a un conjunto de elementos de un arreglo utilizando la técnica que se conoce como slicing. Si `A` es un arreglo de 10 elementos, se pueden obtener los elementos de las posiciones 5 a 8 mediante `A(5 to 8)`. De la misma manera se pueden asignar valores a las posiciones 5 a 8 utilizando un aggregate. Si se supone que `A` es un arreglo declarado como

```
TYPE A IS ARRAY(1 TO 10) OF Integer;
```

y se desean asignar valores a los elementos del mismo se puede utilizar un aggregate posicional, de la forma

```
(1,4,0,0,3,3,0,0,0,0)
```

en la que los elementos están listados en el orden en que serán asignados al arreglo de izquierda a derecha. Alternativamente se puede utilizar un aggregate con posicionamiento nombrado

```
(1,4, 5 => 3,6 => 3,OTHERS => 0)
```

que tiene el mismo efecto que el caso anterior. En este caso el índice de cada elemento se da explícitamente.

3.6.7.6 Tipo registro

Un registro es una colección de datos heterogéneos. Los datos son nombrados, lo que significa que a diferencia de los arreglos se deben acceder por nombre de campo. Un campo es un identificador que se asocia con un tipo de datos determinado dentro del registro. Los registros se declaran utilizando la palabra clave `RECORD`. Para acceder a los campos de un registro se utiliza el nombre del objeto de tipo registro separado por un punto del nombre del campo.

Ejemplo:

```
TYPE Persona IS RECORD
      Nombre:String(1 TO 30);
      Edad:Natural;
END RECORD;
```

Si se desea acceder al nombre de la persona `P` se puede hacer `P.Nombre`

Para asignar valores a los registros, se pueden utilizar aggregates como ocurre con arreglos

```
("Juan Perez", 28)
```

da valores a los campos de un objeto registro `Persona`.

3.6.7.7 Tipo puntero

VHDL soporta la declaración de punteros mediante la palabra clave `ACCESS`. Estas variables pueden crearse dinámicamente, reservando memoria en tiempo de ejecución. Los punteros son una estructura de alto nivel de VHDL, por lo que se utilizan solo para realizar la descripción algorítmica del sistema y pueden ser utilizados sólo con variables.

Ejemplos:

```
TYPE PEntero IS ACCESS Integer;          -- Declara un puntero a entero
```

La creación de un puntero se hace mediante la palabra clave `NEW` y el espacio se libera mediante la sentencia `DEALLOCATE`.

Ejemplos:

```
p:PEntero;         -- Declara un puntero a Integer

p:=NEW Integer;    -- Crea el puntero dinámicamente
DEALLOCATE(p);     -- Libera el espacio
```

3.6.7.8 Tipo archivo

El tipo de datos archivo es una estructura de alto nivel de VHDL, la cual es apropiada para almacenar datos durante la simulación o realizar bancos de prueba (Test-Bench). Los archivos se almacenan en disco.

Un archivo puede almacenar cualquier tipo de datos de VHDL. La forma de declarar un archivo difiere en ambos estándares de VHDL (‘87 y ‘93), como se muestra a continuación

```
TYPE Nombre_tipo IS FILE OF Integer;           -- Común a ambos

FILE Nombre_lógico:Nombre_tipo IS [ modo ] "archivo.ext"
FILE Nombre_lógico:Nombre_tipo [ OPEN modo ] IS "archivo.ext"
```

La segunda declaración corresponde al estándar ’87, mientras que la tercera al estándar ’93.

El `Nombre_lógico` de un archivo es el nombre por el cual se lo referencia en el código. En VHDL ’87 el modo puede ser `IN` u `OUT`, indicando que se va a leer o escribir el archivo. En VHDL ’93, el modo puede ser `WRITE_MODE`, `READ_MODE` o `APPEND_MODE` y `READ_MODE` se toma por omisión.

El tipo de datos archivo como tipo de datos abstracto, posee las siguientes operaciones asociadas

```
PROCEDURE Read(VARIABLE Nombre:INOUT Nombre_tipo; Valor:OUT Integer);
PROCEDURE Write(Nombre:INOUT Nombre_tipo;Valor:IN Integer);
FUNCTION Endfile(VARIABLE Nombre:IN Nombre_tipo) RETURN Boolean;
```

3.6.7.9 Subtipos

Los subtipos son restricciones o subconjuntos de tipos existentes. Existen dos clases de subtipos. La primera clase se utiliza para restringir tipos de un tipo escalar a un rango específico. La otra clase de subtipos se utiliza para restringir el rango de un tipo arreglo irrestricto especificando valores para sus índices. Se declaran mediante la palabra clave `SUBTYPE`.

Ejemplos:

```
SUBTYPE Natural IS Integer RANGE 0 TO Highest_integer;
SUBTYPE Digits IS Character RANGE ‘0’ TO ‘9’;
SUBTYPE Mayusculas IS Character RANGE ‘A’ TO ‘Z’;

SUBTYPE Nombre IS String(1 TO 30);
SUBTYPE Bus16 IS Bit_vector(15 DOWNTO 0);
```

3.6.8 Declaración de objetos de datos

Un objeto en VHDL es un elemento nombrado que tiene asociado un valor de un tipo de datos específico. Los objetos en VHDL se clasifican en tres grupos: constantes, variables y señales.

Las constantes se declaran mediante la palabra clave `CONSTANT`, las variables mediante `VARIABLE` y las señales mediante `SIGNAL`.

Las variables son elementos secuenciales de alto nivel, por lo que pueden aparecer únicamente dentro de construcciones secuenciales o procedurales. Las variables se utilizan en descripciones algorítmicas y no se pueden mapear en hardware.

Una señal especifica una conexión entre dos entidades, módulos o componentes y representan las interconexiones entre los dispositivos físicos de un sistema. Pueden aparecer en construcciones secuenciales o concurrentes. Una señal se declara de la siguiente forma:

```
SIGNAL Nombre_señal:Tipo_datos Tipo_señal [:= Valor_omision];
```

En donde `Tipo_datos` es el conjunto de valores que transporta la señal.

Ejemplos:

```
CONSTANT PI:Real := 3.1416;
CONSTANT e:Real := 2.7182;

VARIABLE a:Integer;
VARIABLE b:Real:=0;                -- A las variables se les puede
                                   -- dar un valor inicial

SIGNAL bus:Bit_vector(31 DOWNTO 0);
```

3.6.9 Atributos

Los tipos y objetos en VHDL pueden tener información asociada denominada atributos. Existen un número predefinido de atributos asociados a un tipo determinado.

Si t es un tipo enumerativo, entero, flotante o físico se definen los siguientes atributos:

`t'left`	Límite izquierdo del tipo t
`t'right`	Límite derecho del tipo t
`t'low`	Límite inferior del tipo t
`t'high`	Límite superior del tipo t

Si `x` es un miembro de un tipo `t` como el anterior y `n` un entero, se definen los siguientes atributos:

`t'pos(x)`	Posición de `x` dentro del tipo `t`
`t'val(n)`	Elemento `n` dentro del tipo `t`
`t'leftof(x)`	Elemento que está a la izquierda de `x` en `t`
`t'rightof(x)`	Elemento que está a la derecha de `x` en `t`
`t'pred(x)`	Elemento que está delante de `x` en `t`
`t'succ(x)`	Elemento que está detrás de `x` en `t`

Si `a` es un vector y `n` es un entero que está en el rango de dimensiones de `a`, se pueden utilizar los atributos:

`a'left(n)`	Límite izquierdo del rango de dimensión `n` de `a`
`a'right(n)`	Límite derecho del rango de dimensión `n` de `a`
`a'low(n)`	Límite inferior del rango de dimensión `n` de `a`

`a'high(n)`	Límite superior del rango de dimensión `n` de `a`
`a'range(n)`	Rango del índice de dimensión `n` de `a`
`a'length(n)`	Longitud del índice de dimensión `n` de `a`

Suponiendo que s es una señal, los atributos mas usuales que se definen son:

`s'event`	Retorna `True` si se ha producido un evento en la señal `s`
`s'stable(time)`	Retorna `True` si s estuvo estable durante el período de tiempo `time`

3.6.10 Expresiones y operadores

Los operadores se utilizan para combinar expresiones formando expresiones mas complejas.

3.6.10.1 Operadores lógicos

Los operadores lógicos que incluye el lenguaje son `AND`, `OR`, `NAND`, `NOR`, `XOR` y `NOT`, y operan sobre datos de tipo `Bit`, `Bit_Vector` o `Boolean`, o arreglos de dichos tipos elemento a elemento. Para datos de tipo `Bit` o `Boolean`, los operadores binarios se comportan como *short-circuit* lo que significa que únicamente se evalúa la expresión de la derecha si la de la izquierda no alcanza para determinar el resultado.

3.6.10.2 Operadores de desplazamiento

Operan sobre datos de tipo `Bit_Vector`. Se utilizan en forma infija, es decir que el objeto a desplazar se ubica a la izquierda del operador y la cantidad a su derecha.

`SLL, SRL`	Desplazamiento lógico a izquierda y a derecha. Desplazan un vector un número de bits a izquierda o derecha, ingresando un cero por el lado opuesto al sentido de desplazamiento.
`SLA, SRA`	Desplazamiento aritmético a izquierda y a derecha. Efectúan el mismo desplazamiento que el caso anterior pero conservando el signo.
`ROL, ROR`	Rotación a izquierda y a derecha. Realiza la rotación del operando a izquierda o derecha. Los bits que salen por un lado, ingresan nuevamente al operando por el lado opuesto.

Ejemplos:

```
Dato SLL 2;        -- Desplaza Dato dos lugares a la izquierda

Dato SRA 1;        -- Desplaza Dato un lugar hacia la derecha
                   -- conservando el signo

Dato ROR 4;        -- Rota Dato cuatro veces a la derecha
```

3.6.10.3 Operadores relacionales

Esta clase de operadores requieren datos del mismo tipo y retornan siempre un valor booleano.

`=, /=` Operador de igualdad. Compara dos datos del mismo tipo y retorna `True` si son iguales, `False` en caso contrario.

`<,<=,>,>=` Operadores de comparación. Comparan por mayor, menor, mayor o igual y menor o igual, y tienen el significado habitual.

3.6.10.4 Operador de concatenación

El operador de concatenación (`&`) opera sobre vectores. Dados dos vectores `A` y `B`, se define el vector concatenación `C := A & B` como el vector que posee como primeros elementos a los elementos del vector `A` y como últimos a los del vector `B`.

3.6.10.5 Operadores aritméticos

Los operadores aritméticos son los mismos que se presentan usualmente en todos los lenguajes de programación. Operan sobre datos de tipo numérico. Los mas comunes son: (`**`) potencia, `ABS()` valor absoluto, (`*`) multiplicación, (`/`) división, `MOD` resto módulo, `REM` resto, (`+`) suma y (`-`) resta.

3.6.11 Construcciones secuenciales

Como ocurre con los lenguajes procedurales, VHDL ofrece la posibilidad de alterar el estado de los objetos y controlar el flujo de ejecución.

3.6.11.1 Asignación a variables

El valor de una variable se puede modificar con la operación de asignación. La variable adquiere el nuevo valor inmediatamente, en el momento de la asignación. Esta es la diferencia fundamental con las señales, que adquieren su valor en el próximo instante de simulación. Una asignación tiene la siguiente sintaxis:

```
variable := expresión;
```

La variable y la expresión deben tener el mismo tipo base. La variable puede ser un aggregate, en cuyo caso las variables listadas deben ser nombres de objetos. Como primer paso se evalúan los nombres en el aggregate, luego la expresión y por último se procede a la asignación de cada componente a cada objeto.

3.6.11.2 Sentencia IF

La sentencia `IF` permite la selección de las sentencias a ejecutar, basándose en una o mas condiciones. Tiene la siguiente sintaxis:

```
IF Condición THEN
      Sentencias
[ ELSIF Condición THEN
      Sentencias ]
[ ELSE
      Sentencias ]
END IF;
```

Las condiciones se evalúan en el orden en que están listadas y cuando se verifica una, se ejecuta el bloque de sentencias correspondiente.

3.6.11.3 Sentencia CASE

La sentencia CASE permite la selección de las sentencias a ejecutar, basándose en una condición de selección. Su sintaxis es:

```
CASE Expresión IS
      WHEN Selección1 =>
            Sentencias
      ...

      WHEN Seleccióni =>
            Sentencias

      [ WHEN OTHERS =>
            Sentencias ]
END CASE;
```

La expresión debe ser cualquier tipo discreto o subtipo de un tipo discreto. Las selecciones deben cubrir todas las alternativas posibles de valores que puede adquirir la expresión. No puede haber selecciones duplicadas.

3.6.11.4 Sentencia nula

La sentencia nula NULL no tiene efecto alguno. Su función es hacer explícito el hecho de que hay veces en las cuales no se requiere realizar ninguna acción. Se suele usar en sentencias CASE, ya que esta última exige considerar todas las alterativas y muchas veces no se debe definir alternativa para las mismas.

3.6.11.5 Aserciones (Assertions)

Una aserción se utiliza para verificar una condición determinada y reportar cuando se la viola. La sintaxis para una aserción es:

```
ASSERT Condición
      [ REPORT Expresión ]
      [ SEVERITY Expresión ];
```

Si la cláusula REPORT está presente, el resultado de la expresión debe ser un String. Este es un mensaje que se reporta cuando la condición es falsa. La cláusula SEVERITY debe ser de tipo Severity_level e indica el nivel de gravedad de la violación. Los posibles niveles son: NOTE, WARNING, ERROR y FAILURE. El nivel por omisión es ERROR.

3.6.11.6 Sentencias de bucle

VHDL presenta una estructura de bucle básico que puede ser extendido para concebir los bucles FOR y WHILE presentes en otros lenguajes. La sintaxis es:

```
Etiqueta_loop:
      [ WHILE Condición | FOR Identificador IN Rango_discreto ] LOOP
            Sentencias
      END LOOP;
```

Si se omiten las sentencia WHILE o FOR el bucle resultante se ejecuta indefinidamente.

La construcción WHILE ejecuta el bucle siempre y cuando la condición evalúe a True.

La construcción FOR itera el bucle la cantidad de veces como elementos hay en el rango. El identificador adquiere los valores en el rango especificado y no puede ser modificado, ya que es una constante que se genera como índice del FOR. Sólo puede ser leído su valor.

Existe además dos sentencias adicionales para modificar el comportamiento de un bucle. La primera de ambas sentencias es NEXT que termina la iteración actual y pasa a la siguiente y la segunda, EXIT termina el bucle actual. La sintaxis de ambas sentencias se define como

```
NEXT [ Etiqueta_loop ] [ WHEN Condición ];
EXIT [ Etiqueta_loop ] [ WHEN Condición ];
```

Si se omite la etiqueta, la sentencia se aplica al bucle mas interno, de lo contrario al nombrado en la misma.

3.6.12 Subprogramas y paquetes

VHDL provee mecanismos de encapsulamiento de código a través de funciones y procedimientos, y de código y datos por medio de paquetes.

3.6.12.1 Funciones y procedimientos

Las funciones y procedimientos en VHDL, se declaran con la siguiente sintaxis:

```
FUNCTION Nombre [ (Lista_parametros_formales) ] RETURN Tipo

PROCEDURE Nombre [ (Lista_parametros_formales) ]
```

Lista_parametros_formales representa una secuencia de parámetros formales separados por punto y coma. Esta forma de declaración se denomina diferida, ya que solo se especifica el encabezado y parámetros de los subprogramas y no su cuerpo. En el caso de la función, también se especifica el valor de retorno. Esta forma de declaración se utiliza en declaración de paquetes, ya que en VHDL los identificadores deben ser declarados siempre antes de ser usados.

La lista de parámetros formales puede estar compuesta por variables, señales o constantes y se les puede asignar un valor por omisión. Los parámetros en un subprograma se clasifican en tres tipos, según el sentido de transferencia de valores:

IN	El parámetro puede ser leído únicamente (Entrada)
OUT	El parámetro puede ser escrito únicamente (Salida)
INOUT	El parámetro puede ser leído y escrito (Entrada/Salida)

Ejemplos:

```
PROCEDURE Operacion(    VARIABLE a:IN Integer;VARIABLE b:IN Integer;
                        VARIABLE res:OUT Integer);
```

Declara el encabezado de un procedimiento que recibe dos variables enteras `a` y `b`, y retorna un resultado en la variable `res`.

```
FUNCTION Convert(       VARIABLE num:IN Real;
                        CONSTANT precision:IN Integer := 16)

                  RETURN Bit_vector;
```

Declara el encabezado de una posible función que recibe un parámetro flotante y la precisión de la conversión a realizar y retorna un vector de bits con el número convertido. Si no se ingresa parámetro actual para la precisión, se asume precisión de 16 bit.

La invocación a los subprogramas se realiza por medio del nombre del mismo con la lista de parámetros actuales adecuados. Un procedimiento se invoca como una sentencia, mientras que una función, se invoca como parte de una expresión o una expresión por si sola. La lista de asociación para los parámetros puede ser posicional, por nombre o una combinación de ambos métodos.

Si `x, y, z` son variables enteras. Para invocar al procedimiento `Operacion` puede utilizarse cualquiera de las tres formas:

```
Operacion(x,y,z);
Operacion(b=>y,res=>z,a=>x);
Operacion(x,y,res=>z);
```

Para especificar el cuerpo de un subprograma se utiliza la sintaxis:

```
FUNCTION Nombre [ (Lista_parametros_formales) ] RETURN Tipo IS
      Declaracion_objetos
BEGIN
      Sentencias
END Nombre;

PROCEDURE Nombre [ (Lista_parametros_formales) ] IS
      Declaracion_de_objetos
BEGIN
      Sentencias
END Nombre;
```

En la parte de declaración de objetos, se pueden declarar variables, constantes tipos, subtipos e incluso otros subprogramas cuya visibilidad se extiende localmente.

Cuando se invoca un subprograma, sus sentencias se ejecutan hasta que se encuentra la sentencia `END` o hasta que se encuentre una sentencia `RETURN`. La sentencia `RETURN` tiene la siguiente sintaxis:

```
RETURN [ Expresion_de_retorno ];
```

en donde `Expresion_de_retorno` es una expresión que se retorna al subprograma invocador una vez que ha sido evaluada. Si una sentencia `RETURN` se utiliza en el cuerpo de un procedimiento, no debe contener expresión de retorno.

Ejemplo:

```
FUNCTION Convert(bits:IN Bit_vector) RETURN Integer IS
      VARIABLE res:Integer:=0
BEGIN

      FOR i IN bits'range LOOP
            res:=res*2+Bit'pos(bits(i));
      END LOOP;

      RETURN res;

END Convert;
```

3.6.12.2 Sobrecarga de operadores (Overloading)

La sobrecarga de subprogramas permite a dos o mas subprogramas tener el mismo nombre, siempre y cuando el número y tipo de parámetros sean distintos en cada uno. Cuando se invoca un subprograma que está sobrecargado, se verifica que los parámetros formales, su orden y tipo concuerden con los reales provistos en la invocación. De esta manera se escoge el subprograma adecuado.

Ejemplo:

```
PROCEDURE Get(c:OUT Character);
PROCEDURE Get(i:OUT Integer);
```

La sobrecarga no solo se limita a nombres de subprogramas, sino también a operadores. Los operadores son tipos especiales de funciones que pueden ser utilizadas en forma infija. Cuando se sobrecarga un operador, el nombre del mismo se encierra entre comillas.

Ejemplo:

```
FUNCTION "+"(a,b:IN String) RETURN String IS
BEGIN
      -- Cuerpo de la función
END "+";
```

La "suma de Strings" puede definirse por ejemplo como concatenación de los mismos.

Un operador puede ser invocado de dos maneras distintas. Para el ejemplo anterior si `a, b, c` son Strings, entonces

```
c := a + b;
```

o bien

```
c := "+"(a,b);
```

3.6.12.3 Paquetes (Packages)

Un paquete en VHDL es una colección de constantes, tipos y operaciones. Normalmente se utilizan para encapsular la implementación y para declarar tipos abstractos. Un paquete se divide en dos partes: una declaración de paquete, que define su interfaz, y un cuerpo que implementa las operaciones que exporta la interfaz. El cuerpo también puede contener declaración de tipos y constantes, y puede ser obviado, en caso de que el paquete conste únicamente de constantes y tipos. El esqueleto genérico de un paquete se define como sigue:

```
PACKAGE Nombre_paquete IS

      Declaración de tipos y subtipos
      Declaración de constantes
      Declaración de encabezados de subprogramas

END Nombre_paquete;

PACKAGE BODY Nombre_paquete IS

      Declaración de tipos y subtipos
      Declaración de constantes
      Implementación de subprogramas

END Nombre_paquete;
```

Las declaraciones de tipos y constantes que aparecen en el encabezado, no necesitan aparecer nuevamente en el cuerpo ya que su visibilidad se extiende en él.

3.6.12.3 Alcance, visibilidad y utilización de los paquetes

Una vez que se ha implementado un paquete, puede ser utilizado mediante la cláusula

```
USE Nombre_paquete.Elemento;
```

El alcance y visibilidad de un paquete se extiende al módulo que contiene la cláusula `USE`. Los objetos exportados por el paquete pueden ser utilizados anteponiendo el nombre del paquete ante los mismos separado por un punto.

La cláusula `Elemento` indica que objeto del paquete nombrado se desea hacer visible en el módulo. Si se desean hacer visibles todos sus componentes, se puede usar la palabra clave `ALL`. De lo contrario se especifica el nombre del objeto.

3.6.13 Declaración de entidad

Un sistema digital se diseña como un conjunto de componentes interconectados. Cada componente tiene un conjunto de puertos que constituyen su interfaz con el entorno. En VHDL la declaración de un componente se denomina entidad y se especifica como sigue:

```
ENTITY Nombre_entidad IS

      GENERIC(Lista_genericos);
      PORT(Lista_señales);

[ BEGIN
      Sentencias ]
END Nombre_entidad;
```

La `Lista_genericos` está formada por parámetros que son pasados a la entidad en el momento de su creación. Deben ser siempre constantes y pueden incluirse varias separados por punto y coma.

La `Lista_señales` la constituyen los puertos que ofician de interfaz con el exterior. Debe incluir únicamente señales las cuales pueden ser de cualquiera de los tres modos `IN`, `OUT` o `INOUT`.

La forma general de declaración de los parámetros es:

```
Nombre_parametro : Modo Tipo;
```

El bloque opcional de sentencias, se ejecuta en el momento que se instancia la entidad y puede utilizarse para verificar por ejemplo si los parámetros genéricos son correctos.

Ejemplo:

```
ENTITY ROM IS

      GENERIC(width,size:Natural:=16);
      PORT( ena:IN Bit;
            addr:IN Bit_Vector(size-1 DOWNTO 0);
            data:OUT Bit_Vector(width-1 DOWNTO 0)
            );

END ROM;
```

Declara la entidad ROM especificando el ancho de palabra y la cantidad de direcciones como parámetros genéricos.

3.6.14 Declaración de arquitectura

Con la declaración de entidad, se especifica únicamente la interfaz del módulo. El comportamiento del módulo puede ser descripto de varias formas en los cuerpos de la arquitectura. Cada arquitectura representa una vista distinta de la entidad. Se puede realizar la descripción de una arquitectura de manera puramente funcional, utilizando técnicas de programación convencionales. De otro modo se puede implementar una arquitectura como una colección de componentes interconectados de manera estructural.

Un cuerpo de arquitectura se especifica como:

```
ARCHITECTURE Nombre_arquitectura OF Nombre_entidad IS

      Declaración de tipos y subtipos
      Declaración de subprogramas
      Declaración de componentes
      Declaración de variables, señales, constantes

BEGIN

      Sentencias concurrentes
      Bloques concurrentes
      Procesos

END Nombre_arquitectura;
```

3.6.14.1 Bloques

Las unidades dentro de una arquitectura pueden ser descriptas como bloques, que están conectados a otros bloques por medio de señales.

La forma de declarar un bloque es la siguiente:

```
Etiqueta_bloque:
            BLOCK
                  GENERIC(Lista_genericos);
                  GENERIC MAP(Lista_map_genericos);
                  PORT(Lista_señales);
                  PORT MAP(Lista_map_señales);
            BEGIN
                  Sentencias concurrentes
                  Bloques concurrentes
                  Procesos
            END BLOCK [Etiqueta_bloque];
```

`Lista_genericos` y `Lista_señales`, son las constantes genéricas y señales que exporta el bloque.

`Lista_map_genericos` y `Lista_map_señales` son las asociaciones (genéricos y señales) con los que se conecta el bloque al exterior.

3.6.14.2 Declaración de componentes

Una arquitectura puede hacer uso de entidades descriptas en otras librerías. Para ello debe declarar un componente que luego será instanciado en el momento que se emplee. La sintaxis a la que responde la declaración de componente es:

```
COMPONENT Nombre_componente IS

      GENERIC(Lista_genericos_locales);
      PORT(Lista_señales_locales);

END COMPONENT;
```

La declaración de componente debe corresponderse con la declaración de una entidad en la librería.

Por ejemplo para el caso de la memoria ROM ejemplificada anteriormente, la declaración de componente para esa entidad, sería

```
COMPONENT ROM IS

      GENERIC(width,size:Natural:=16);
      PORT( ena:IN Bit;
            addr:IN Bit_Vector(size-1 DOWNTO 0);
            data:OUT Bit_Vector(width-1 DOWNTO 0)
            );

END COMPONENT;
```

3.6.14.3 Instanciación de componentes

Un componente declarado se debe instanciar para su uso. La forma de instanciar un componente se realiza mediante la palabra clave `MAP`.

```
Etiqueta_componente:Nombre_componente
                  [ GENERIC MAP(Lista_genericos_actuales) ]
                  [ PORT MAP(Lista_señales_actuales) ];
```

en donde `Lista_genericos_actuales` es una lista de constantes (parámetros actuales) pasadas al componente y `Lista_señales_actuales` es la lista de señales por las cuales el componente se conecta en la arquitectura donde está siendo instanciado. La asociación de parámetros actuales puede realizarse de manera posicional o nombrada como ocurría en el caso de los subprogramas.

A modo de ejemplo, si se desean instanciar dos memorias ROM como las declaradas anteriormente se procede de la siguiente manera:

```
Mem1: ROM GENERIC MAP(8,1024) PORT MAP(ena,addr1,data1);
Mem2: ROM GENERIC MAP(16,1024) PORT MAP(ena,addr2,data2);
```

En este caso se instancia a la ROM dos veces. En el primer caso se trata de una ROM de 1204 bytes y en el segundo caso de una ROM de 1024 words.

3.6.15 Asignación a las señales

La asignación a una señal determina una o mas transacciones sobre la señal. Se realiza de la siguiente manera:

```
Señal_destino <= [ TRANSPORT ] Valor [ AFTER Medida_de_tiempo Unidad ];
```

La forma mas simple de asignación, es si se omiten la palabra clave `TRANSPORT` y `AFTER`, con lo que la señal adquiere su valor en el próximo paso de simulación. Si se desea que una señal adquiera el valor asignado luego de un período posterior a la asignación se utiliza la palabra clave `AFTER`. A modo de ejemplo sea `sen` una señal de tipo `Bit_vector` que debe adquirir los valores ‘0’ luego de 5 nanosegundos,

```
sen <= '0' AFTER 5 ns;        -- Adquiere el valor '0' luego de 5 ns
```

Se pueden emplear diversas unidades de tiempo. La unidad mas pequeña es el femtosegundo (fs). Si la asignación anterior es ejecutada a los 15 ns de tiempo de simulación, la señal `sen` tomará su nuevo valor a los 20 ns. Si ahora se realiza otra asignación con la siguiente forma a la señal `sen`,

```
sen <= '1' AFTER 10 ns, '0' AFTER 15 ns;
```

se agregan las transacciones sobre la señal, provocando que la misma cambie cada 5 ns en un período de 15 ns. Si se ejecuta una asignación a una señal y existen transacciones posteriores en el tiempo que aún no han surtido efecto, son eliminadas y las nuevas colocadas en su lugar. Esta última característica en la asignación puede ser modificada incluyendo la palabra clave `TRANSPORT`. Si se omite la palabra clave `TRANSPORT`, se asume que el modo es `INERTIAL`. Por lo tanto,

TRANSPORT: si se incluye esta palabra clave, se dice que la asignación tiene *transport delay*, en cuyo caso todas las transacciones anteriores que deban ser llevadas a cabo luego de la transacción nueva son eliminadas antes de establecer la nueva.

INERTIAL: si se incluye la palabra clave `INERTIAL` o no se incluye palabra clave alguna, se dice que la asignación tiene *inertial delay*. En este caso, todas las transacciones que deban ser llevadas a cabo luego de la transacción nueva son eliminadas antes de establecer la nueva, como ocurre en el caso de transport. Seguidamente, se examinan todas las transacciones anteriores cuyo efecto deba producirse luego del efecto de las nuevas. Si existe alguna con un valor diferente a la nueva transacción, se eliminan todas las anteriores a la que posee el valor distinto. Las restantes, permanecen.

3.6.16 Ejecución secuencial: Procesos y la sentencia WAIT

Como se mencionó anteriormente, para expresar el comportamiento de un sistema digital se puede utilizar la forma de descripción funcional o de comportamiento. La unidad principal para describir a un sistema funcionalmente es el proceso (process). El proceso es un módulo secuencial de código que puede ser activado conforme ocurran eventos sobre señales determinadas. Cuando mas de un proceso se activa al mismo tiempo, éstos ejecutan en forma concurrente. La forma de especificar un proceso es la siguiente:

```
[ Nombre_proceso: ]
      PROCESS [ (Lista_sensibilidad) ]
            Declaración de tipos y subtipos
            Declaración de subprogramas
            Declaración de componentes
            Declaración de variables, señales, constantes
      BEGIN
            Sentencias secuenciales
            [ Sentencia WAIT ]
      END PROCESS;
```

Un proceso es el único lugar junto con las funciones y los procedimientos dentro de un programa en VHDL donde se pueden declarar y utilizar variables. Generalmente las variables se utilizan para almacenar estados en un modelo. Dentro de un proceso puede haber cualquier cantidad de asignación a señales. Las asignaciones a las señales determinar un *driver* para esa señal. Una señal puede poseer a lo sumo un *driver*, de lo contrario provocaría un problema de resolución.

Al iniciarse la simulación, se ejecuta el cuerpo del proceso. Una vez que se ha ejecutado el cuerpo, el proceso se detiene para comenzar nuevamente. Un proceso puede ser suspendido por medio de una sentencia `WAIT`, que tiene la forma:

```
WAIT  [ ON Nombre_de_señal ]
      [ UNTIL Condición_de_espera ]
      [ FOR Tiempo_de_espera ]
```

La lista de sensibilidad que acompaña a la sentencia `WAIT`, especifica una lista de señales `Nombre_de_señal` a las cuales el proceso es sensible mientras está suspendido, en efecto la ejecución continuará si se produce un evento sobre alguna de las mismas y la condición de espera es verdadera. La expresión de `Tiempo_de_espera` indica el tiempo que el proceso permanecerá suspendido, si es obviada el proceso esperará indefinidamente sobre la lista de señales. Se puede agregar una condición que determina que el proceso espere mientras ésta sea verdadera.

Alternativamente se puede expresar la lista de sensibilidad al comienzo del proceso, que tiene el mismo efecto que colocar una sentencia `WAIT` al final del mismo. Si se encuentra presenta la lista de sensibilidad al comienzo del proceso, el mismo no puede incluir sentencias `WAIT` adicionales.

Ejemplo:

```
PROCESS (reset, clock)
      VARIABLE state:Boolean:=False;
BEGIN
      IF reset THEN
            state:=False;
      ELSIF clock THEN
            state:=NOT state;
      END IF;

      q<=state AFTER 1 ns;

END PROCESS;
```

Al comenzar la simulación, el cuerpo del proceso es ejecutado, luego de lo cual se suspende aguardando algún cambio en alguna señal de su lista de sensibilidad. Cuando se produce algún cambio, se vuelve a ejecutar el cuerpo del proceso. Si cambió la señal de `reset` de `False` a `True`, el estado `False` se asigna a la señal de salida `q` cada 1 ns; si `reset` es `False` y `clock` es `True`, cambia el estado entre `True` y `False`, y se asigna a la señal de salida `q` cada 1 ns.

El siguiente ejemplo muestra una señal q que adquiere valores ‘1’ y ‘0’ alternadamente cada 2 ns

```
PROCESS
BEGIN

      WAIT FOR 2 ns;
      q<='1';

      WAIT FOR 2 ns;
      q<='0';

END PROCESS;
```

3.6.17 Asignación concurrente a señales

VHDL provee una notación denominada asignación concurrente a señales, que puede ser utilizada para expresar procesos que describen un driver para una señal. La asignación puede ser condicional o selectiva.

La asignación condicional es una abreviación para procesos que contienen asignaciones a señales dentro de sentencias `IF`. Por lo tanto el proceso:

```
PROCESS (clk)
BEGIN
      IF clk='0' THEN
            q<=False;
      ELSE
            q<=True;
      END IF;
END PROCESS;
```

puede expresarse con una asignación condicional de la siguiente manera:

```
q<=False WHEN clk='0' ELSE True;
```

Formalmente, la sentencia de asignación condicional tiene la sintaxis:

```
Nombre_señal<=valor1 WHEN Condición ELSE valor2;
```

La asignación selectiva se utiliza como abreviación de un proceso que tiene asignación a una señal dentro de un bloque `CASE`. Por lo tanto un proceso descripto como:

```
PROCESS
BEGIN
      CASE Expresión IS
            WHEN Caso_1 =>
                  q<=Valor_1;
            WHEN Caso_2 =>
                  q<=Valor_2;

            ...
            WHEN Caso_n =>
                  q<=Valor_n;
      END CASE;

      WAIT [Cláusula_de_sensiblidad]

END PROCESS;
```

posee una notación abreviada equivalente:

```
WITH Expresión SELECT
      q<=valor_1 WHEN caso_1,
            Valor_2 WHEN Caso_2,
            ...
            Valor_n WHEN Caso_n;
```

3.6.18 Unidades y bibliotecas

Las descripciones en VHDL se almacenan en archivos de diseño, que son archivos de texto que contienen las sentencias que conforman los distintos módulos que integran el sistema. Usualmente se almacenan con la extensión .VHD

El compilador es quien se encarga de analizarlas, compilarlas y incluirlas en un archivo de biblioteca, también llamado librería. Cada módulo se incluye por separado y se denomina unidad de biblioteca. Las unidades de biblioteca primarias son las declaraciones de entidad, declaraciones de paquete y declaraciones de configuración. Las unidades de biblioteca secundarias están integradas por las arquitecturas y cuerpos de paquete. Se denominan secundarias, ya que su especificación depende de la declaración de las unidades primarias.

Las unidades de biblioteca se agrupan en bibliotecas que se almacenan como archivos separados con un nombre. Para utilizar las unidades de una biblioteca determinada, se debe especificar su inclusión de la siguiente forma:

```
LIBRARY Nombre_de_biblioteca;
USE Nombre_de_biblioteca.[ ALL ];
```

en donde la cláusula `LIBRARY` indica al compilador VHDL que incluya el archivo de biblioteca con `Nombre_de_biblioteca` en el diseño. La cláusula `USE` especifica que componentes de `Nombre_de_biblioteca` desean utilizarse. Si se indica la palabra clave `ALL`, se asume la inclusión de todos los componentes, en caso contrario se debe utilizar `Nombre_de_biblioteca` seguido del nombre del componente a usar. Por ejemplo,

```
LIBRARY Memory;
USE Memory.ROM;
```

incluye la biblioteca `Memory` en el diseño y hace visible para su utilización únicamente a la unidad `ROM`. La visibilidad de un componente se extiende a la unidad de diseño en donde se lo nombra.

Existen dos bibliotecas que son visibles a todos los diseños y no deben ser incluidas explícitamente. La primera es la biblioteca `WORK`, en donde el compilador incluye el diseño actual. La segunda biblioteca es `STD`, que contiene la funciones y tipos predefinidos por el lenguaje.

3.6.19 La sentencia GENERATE

La sentencia `GENERATE` provee la capacidad de describir estructuras regulares como vectores de bloques, procesos o instancias de componentes dentro del cuerpo de una arquitectura. Su sintaxis es:

```
Etiqueta_generate:
      FOR Rango_generate GENERATE
            [ IF Condición GENERATE
                  Instancia_de_componente
                  Bloque
                  Proceso
            END GENERATE ]

            Instancia_de_componente
            Bloque
            Proceso
      END GENERATE;
```

El `Rango_generate` describe la cantidad de veces que el esquema se va a generar y puede depender de una variable que puede ser utilizada en los esquemas. La sentencia `IF` se utiliza para casos excepcionales en los patrones. El siguiente ejemplo muestra el cuerpo de un sumador binario, utilizando la sentencia `GENERATE`:

```
FOR i IN sum'left DOWNTO 0 GENERATE
LowBit:
      IF i=0 GENERATE
            FA: FullAdder PORT MAP(a2(0),b2(0),cIn,sum(0),c(0));
      END GENERATE;

OtherBits:
      IF i/=0 GENERATE
            FA: FullAdder PORT MAP(a2(i),b2(i),c(i-1),sum(i),c(i));
      END GENERATE;
END GENERATE;
```

Ambos esquemas son similares salvo por el carry de entrada que es una señal proveniente del exterior y debe sumarse únicamente al bit menos significativo. En los casos restantes, cada `FullAdder` se conecta con el siguiente.

3.6.20 La Unidad de configuración

3.6.20.1 Especificación de configuración

Un componente no es mas que una interfaz sin funcionalidad asociada. Por ello se lo debe enlazar con una entidad y arquitectura. Esto se debe a que un mismo componente puede tener mas de una arquitectura relacionada.

La forma de declarar este enlace entre un componente y una entidad se consigue mediante la sentencia `FOR`. La sintaxis para el caso de un componente es la siguiente

```
FOR Lista_de_referencias : Nombre_de_componente
      [ USE ENTITY Nombre_de_entidad (Nombre_de_arquitectura) ]
      [ USE CONFIGURATION Nombre_de_configuración ]
      [ GENERIC MAP (Parámetros_genericos) ]
      [ PORT MAP (Puertos) ];
```

La `Lista_de_referencias` es una lista separada por comas de las referencias realizadas del componente `Nombre_de_componente`, es decir de las entidades y arquitecturas que se deseen utilizar como descripción del componente. En esta lista se pueden utilizar también las palabras clave `ALL` y `OTHERS` para seleccionar todas las referencias hechas del componente, o todas las que

queden por enlazar con una entidad. Normalmente todas las referencias de un componente están enlazadas con la misma entidad y arquitectura, pero de todos modos hay ocasiones en las que resulta útil utilizar otra entidad o arquitectura diferente para algún componente crítico.

Luego de la sentencia `FOR` se utiliza la cláusula `USE`, que permite enlazar con una entidad seguida con la arquitectura correspondiente entre paréntesis. Opcionalmente se puede enlazar con un bloque de configuración el cual posee información sobre la entidad a enlazar.

La sintaxis de enlazado entre componente y entidad también depende de dónde se coloque la sentencia. La sentencia anteriormente descripta se ubica en la parte declarativa de la arquitectura y se usa para enlazar componentes con sus respectivas referencias.

3.6.20.2 Declaración de configuración

La declaración de configuración se realiza en un bloque especial de VHDL denominado *Unidad de Configuración*. En este bloque se pueden especificar los mismos enlaces de componentes con entidades como ocurre en la especificación de configuración. La forma general de una declaración de configuración es la siguiente

```
CONFIGURATION Nombre_de_configuración OF Nombre_de_entidad IS
      [ Sentencia_use ]
      [ Definición_de_atributo ]
      [ Definición_de_grupos ]
      Bloque_de_configuración
END Nombre_de_configuración;
```

La unidad de configuración está siempre asociada a una entidad. Todo el contenido de una configuración se refiere a la entidad indicada en `Nombre_de_entidad`.

La definición de grupos se utiliza para definir una colección de elementos mediante la palabra clave `GROUP`. Se utiliza normalmente para pasar información acerca de una entidad a la herramienta de síntesis del circuito si ésta lo requiere.

El `Bloque_de_configuración` es la única parte obligatoria dentro del cuerpo de una unidad de configuración. Se refiere a las posibles estructuras presentes en una arquitectura. Dichas estructuras se organizan en tres niveles; los elementos que se encuentran en la propia arquitectura, los elementos que se encuentran en bloques y los elementos dentro de sentencias `GENERATE`. Como éstos dos últimos se estructuran en niveles, la configuración, también. La forma general del bloque de configuración es como sigue

```
FOR Especificación_de_bloque
      [ Sentencia_use ]
      [ Bloque_de_configuración ]
      [ Configuración_de_componente ]
END FOR;
```

Hay tres elementos que pueden constituir la `Especificación_de_bloque`. Usualmente se agrega el nombre de una arquitectura, pero también puede ser el identificador de un bloque `BLOCK` o el de un bloque `GENERATE`. Como ocurre en el caso de la configuración, también se pueden incluir cláusulas `USE`. Por último se puede volver a definir un `Bloque_de_configuración` debido a la naturaleza jerárquica de esta unidad. Un posible ejemplo para una unidad de configuración podría ser el siguiente

```
CONFIGURATION OperConf OF Oper IS
      FOR FuncionalOper
        FOR Bloque1
          FOR c1: y2 USE ENTITY Gates.And2(Functional);
          END FOR;
          FOR  genear
            FOR c2: Suma USE CONFIGURATION Operadores.SumaConf;
            END FOR;
          END FOR;
        END FOR;
      END FOR;
END OperConf;
```

Para escoger la arquitectura funcional para la entidad `Suma`

```
CONFIGURATION SumaConf OF Suma IS
      FOR Funcional;
      END FOR;
END SumaFuncional;
```

Si el diseño es complejo, conviene realizar todos los enlaces entre componentes y entidades en un bloque de configuración. Las ventajas que presente este enfoque frente a la especificación de la configuración, es que se puede expresar jerarquía y además brinda la posibilidad de incluir la configuración en un archivo para tal fin, evitando editar las distintas arquitecturas cada vez que se desee hacer algún cambio.

3.6.21 Buses y resolución de señales

Como se dijo anteriormente cada señal debe tener una sola fuente, sin embargo en muchos casos es necesario conectar diversas fuentes a la misma señal. Esto puede traer aparejado una serie de problemas ya que si mas de una fuente modifica la señal al mismo tiempo debe existir un modo de arbitrar cual de todos los valores debe adquirir la señal. Esta clase de problemas se denomina resolución de señales. Para manejar este tipo de situaciones VHDL introduce un nuevo tipo de datos para las señales denominado tipo resuelto.

Un tipo resuelto incluye en su definición una función de resolución que toma todas las fuentes para la misma señal para determinar el valor final de la misma. Un tipo resuelto se declara como un subtipo, respetando la sintaxis:

```
SUBTYPE Nombre_tipo_resuelto IS Nombre_función Nombre_tipo_fuentes;
```

El `Nombre_de_funcion` es el nombre de una función de resolución declarada previamente que debe tomar como argumento un vector irrestricto del tipo de las señales a resolver y debe retornar una señal de ese tipo. A modo de ejemplo se introduce lo siguiente:

```
TYPE Logic_tristate IS ('0','1',Z);
TYPE Logic_tristate_array IS ARRAY (Integer RANGE <>)
      OF Logic_tristate;

FUNCTION Resolved(sen:IN Logic_tristate_array)
      RETURN Logic_tristate;

SUBTYPE Res_logic_tristate IS Resolved logic_tristate;
```

```
FUNCTION Resolved(sen:IN Logic_tristate_array)
      RETURN Logic_tristate IS
BEGIN
      FOR i IN sen'range LOOP
            IF sen(i)='0' THEN
                  RETURN '0';
            END IF;
      END LOOP;

      RETURN '1';
END Resolved;
```

El tipo triestado representa tres posibles valores, ‘0’, ‘1’ y Z (alta impedancia). El tipo `Res_logic_tristate` corresponde al tipo resuelto. La función utilizada para resolver retorna ‘0’ si alguna señal es ‘0’, en caso contrario retorna ‘1’, ya que todas las señales son ‘1’ o Z. Este tipo de resolución se denomina AND-cableado que será ‘0’ si al menos uno lo es.

3.6.22 Aserciones concurrentes

Una aserción concurrente es análogo a un proceso que contiene una aserción seguida de una sentencia `WAIT`. Por ejemplo:

```
ASRT: ASSERT Condición REPORT Mensaje SEVERITY Nivel_severidad;
```

es equivalente a:

```
ASRT: PROCESS
      BEGIN
            ASSERT Condición REPORT Mensaje SEVERITY Nivel_severidad;
            WAIT ON Lista_sensibilidad;
      END PROCESS;
```

En este caso la lista de sensibilidad debe contener a todas las señales que intervienen en la condición. Si no se incluyen señales en la lista, el proceso se activa durante la fase de inicialización y se suspende para siempre.

3.6.23 Invocación a procedimiento concurrente

Una invocación a procedimiento concurrente es equivalente a un proceso que contiene únicamente un llamado a procedimiento seguido de una sentencia `WAIT`. Un procedimiento llamado de esta manera no puede tener parámetros formales de tipo variable, ya que las variables tienen sentido únicamente en entornos secuenciales. La lista de sensibilidad del proceso debe incluir las señales que son parámetros de modo `IN` o `INOUT` del procedimiento.

Por ejemplo el llamado:

```
PRO: Nombre_paquete.operacion(p1,p2,p3);
```

es equivalente a:

```
PRO:  PROCESS
      BEGIN
            Nombre_paquete.operacion(p1,p2,p3);
            WAIT ON p1,p2,p3;
      END PROCESS;
```

3.6.24 Transacciones nulas

Una transacción nula especifica la desconexión de una fuente de una señal. Cuando hay varias fuentes conectadas a una misma señal (en efecto, se trata de una señal resuelta) hay circunstancias en las cuales un dispositivo no debe emitir un valor. Su sintaxis es:

```
señal <= NULL AFTER Tiempo;
```

3.7 Descripción de bancos de prueba

Una de las partes mas costosas en tiempo de diseño de sistemas digitales es la verificación de su funcionamiento. Con las metodologías de diseño tradicionales, esto no se podía comprobar hasta que el dispositivo era implementado. Esto traía como resultado que los fallos se tornen insostenibles en algunos casos. El lenguaje VHDL, permite acelerar el proceso de verificación, brindando la posibilidad de llevar a cabo una simulación exhaustiva del modelo, que si es correcto, puede ser implementado directamente. La forma mas directa de llevar a cabo la simulación de un modelo sencillo es modificar las entradas y observar las salidas. Para modelos complejos esta forma de simulación podría resultar poco flexible, por eso VHDL incluye la posibilidad de definir un banco de pruebas.

Un banco de pruebas consiste en un conjunto de entradas denominadas patrones de prueba. Con el lenguaje VHDL se puede modelar el banco de pruebas independientemente de la herramienta de simulación. El banco de pruebas puede ser utilizado en cualquier fase de simulación y para cualquier nivel de abstracción de una definición, posibilitando un ahorro en tiempo y esfuerzo.

Un banco de pruebas es una entidad simple sin puertos de entrada y salida, y su arquitectura tiene como señales internas las entradas y salidas del circuito. El único componente es el correspondiente a la entidad que se va a simular.

La construcción de un banco de pruebas puede realizarse mediante tres métodos, clasificados según como se generen los patrones de prueba.

3.7.1 Método tabular

El método tabular consiste en elaborar una tabla con entradas y la respuesta prevista del circuito. Esta tabla usualmente se almacena en un vector y forma parte del código VHDL en donde está declarado el componente ligado a la entidad que se pretende simular. Se verifica que la respuesta del circuito coincida con la respuesta prevista almacenada previamente en el vector.

3.7.2 Utilización de archivos

El método tabular es independiente del modelo a ser simulado. El modelo y el banco de pruebas pueden estar en archivos distintos, aportando modularidad y portabilidad. El mismo banco de pruebas puede usarse para simular descripciones a distintos niveles de abstracción del componente.

Para simulaciones más costosas o para tener una mejor organización en el banco de pruebas, puede ser apropiado separar los vectores de prueba del mecanismo de simulación. La ventaja de la separación es que el banco de pruebas se independiza aún mas del componente a verificar, ya que si se realizaran modificaciones sobre los patrones no habrá peligro de modificar accidentalmente el componente simulado. El lenguaje VHDL soporta esta separación mediante el uso de archivos y sus operaciones de entrada/salida asociadas.

El acceso a archivos no solo se limita a la lectura de patrones de prueba, sino también al almacenamiento de los resultados de la simulación en un archivo. Esto último puede resultar útil en caso de que se deseen comparar dos archivos con resultados para validar el funcionamiento de un circuito.

La estructura del código que describe el banco de pruebas es similar al caso tabular. La entidad no tiene entradas ni salidas y referencia únicamente al componente que se desea simular.

3.7.3 Utilización de un algoritmo

Los métodos descriptos anteriormente son apropiados para patrones poco extensos. La desventaja que presenta el método que utiliza archivos, es que los accesos al disco incrementan el tiempo, retardando la simulación.

Para salvar estos problemas se puede elaborar un banco de pruebas en donde los patrones se generen en forma algorítmica, es decir, los vectores se generan por medio de un algoritmo o una fórmula matemática. Un ejemplo interesante podría ser la verificación de un circuito de suma. Las sumas pueden realizarse mediante un algoritmo descripto en VHDL en forma funcional y ser comparadas en tiempo de ejecución con el resultado calculado por el circuito.

Los vectores de prueba no están predefinidos, se generan conforme avanza la simulación dentro del código en VHDL.

Capítulo 4

Descripción del algoritmo CORDIC en VHDL

Una vez analizado el algoritmo CORDIC se optó por realizar tres descripciones. Las tres descripciones se limitaron a considerar el caso circular del algoritmo, es decir cuando $\mu = 1$ y por lo tanto $f(x) = arctg(x)$. Partiendo de la versión genérica de CORDIC

$$x_{i+1} = x_i - \mu\, y_i d_i 2^{-i}$$
$$y_{i+1} = y_i + x_i d_i 2^{-i}$$
$$z_{i+1} = z_i - d_i f\left(2^{-i}\right)$$

se llega a

$$x_{i+1} = x_i - y_i d_i 2^{-i}$$
$$y_{i+1} = y_i + x_i d_i 2^{-i}$$
$$z_{i+1} = z_i - d_i \operatorname{arctg}\left(2^{-i}\right)$$

en donde, $d_i = \begin{cases} -1 & , si\ z_i < 0 \\ 1 & , si\ z_i \geq 0 \end{cases}$,ó bien $d_i = \begin{cases} -1 & , si\ y_i \geq 0 \\ 1 & , si\ y_i < 0 \end{cases}$

, para el modo *rotación* o *vectorización* respectivamente.

El modo rotación se puede utilizar para calcular las funciones seno y coseno, mientras que en modo vectorización se puede obtener el arcotangente.

La primera descripción del algoritmo se realizó a nivel funcional algorítmico, sin considerar los aspectos propios de una arquitectura en particular como las introducidas en la sección 1.6. Dicha descripción se utilizó para comprender y validar el funcionamiento del algoritmo.

Se han descripto además dos arquitecturas particulares de las tres vistas en la sección 1.6. Las arquitecturas por las que se ha optado son: bit-paralela desplegada y bit-paralela iterativa. Se ha optado por describir las arquitecturas bit-paralelas, ya que poseen componentes en común que pueden ser reutilizados en ambos casos. Por otra parte ambas arquitecturas están desarrolladas sobre el mismo concepto acerca del paralelismo de operandos con la diferencia que la arquitectura desplegada es combinatoria y la iterativa es secuencial, motivando la descripción de circuitos combinatorios y secuenciales en VHDL. El formato numérico con el que operan las arquitecturas es en complemento a dos y se utiliza artimética de punto fijo.

La descripción de las arquitecturas particulares se ha realizado combinando los estilos de descripción de flujo de datos y estructural, y a nivel de compuertas lógicas. Se considera que las mismas puedan ser adaptadas a una herramienta de síntesis en un futuro con cambios adecuados.

4.1 Herramienta de desarrollo

Para desarrollar y simular las descripciones, se utilizó una versión limitada de la herramienta Veribest VBVHDL 99.0. La herramienta provee un ambiente con un compilador y simulador de VHDL, ambos integrados en la interfaz. El ambiente brinda la posibilidad de crear proyectos y diversos módulos que forman parte del diseño. Además posee diversas bibliotecas con funciones predefinidas para manejar diversos tipos de datos.

El compilador permite compilar código generado para cualquiera de los estándares de VHDL introducidos en el capítulo 3, el estándar del año 1987 y el estándar de 1993. Se pueden compilar archivos por separado o todo el proyecto de una vez. En la figura 4.1 se muestra una imagen del ambiente para trabajar en VHDL.

El simulador posee un visualizador de ondas (waveform viewer), que permite seleccionar las señales que se desean monitorear conforme avanza la simulación. Los resultados de la simulación pueden ser salvados en disco o impresos. Por otra parte incluye un debugger para depurar las porciones secuenciales del código (bloques process, funciones y procedimientos). El debugger permite ver el contenido de las variables, establecer breakpoints y forzar la ejecución paso a paso.

El ambiente es bastante completo, sin embargo se han encontrado algunos errores (bugs) atribuidos al hecho de tratarse de una versión limitada.

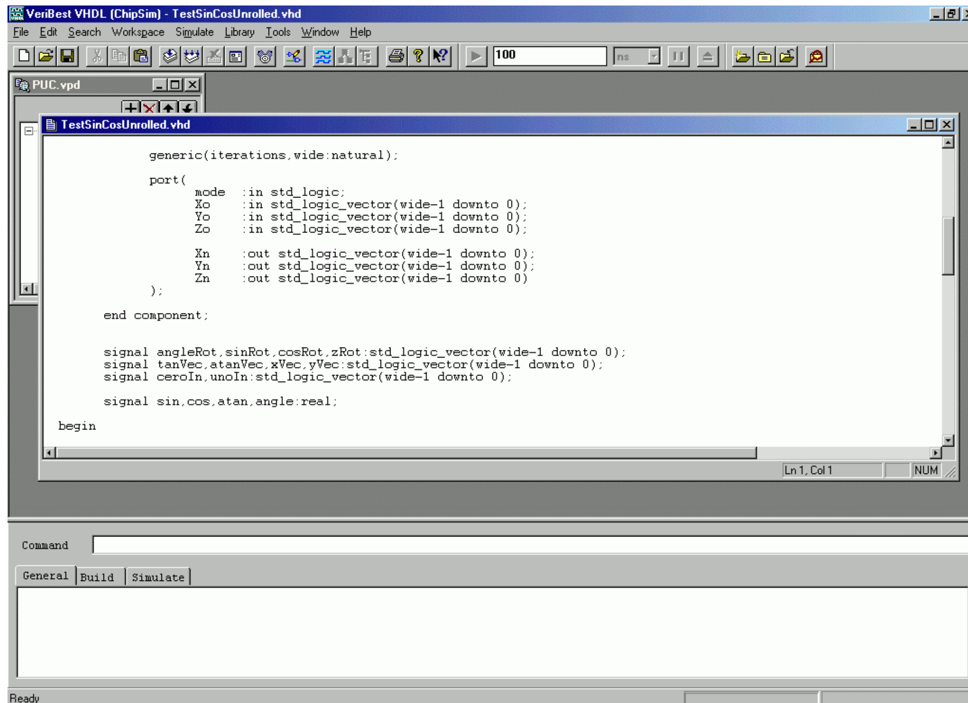


Figura 4.1

4.2 La descripción funcional algorítmica

4.2.1 La descripción del package CORDIC

La primera descripción que se desarrolló es funcional algorítmica. Está basada en el comportamiento del algoritmo y no en una arquitectura en particular. El propósito de la descripción funcional, es el de comprender y validar el funcionamiento del algoritmo comparando los resultados obtenidos con la herramienta MATLAB™. El formato numérico sobre el cual trabaja esta descripción es el tipo de datos `REAL` provisto por el lenguaje.

Para un entendimiento claro y conciso de los pasos seguidos en el desarrollo de la descripción, se introducirá el pseudocódigo del algoritmo CORDIC como se puede deducir de la sección 4.1, para concluir finalmente que se puede pasar de manera casi inmediata a una descripción funcional del mismo.

```
x : real = x0
y : real = y0
z : real = z0
xAnterior, yAnterior : real

Comienzo CORDIC

      Si Rotación entonces

            Desde i = 0 hasta Numero_De_Iteraciones-1

                  xAnterior = x
                  yAnterior = y

                  x = x + yAnterior * signo(z) * 2^-i
                  y = y - xAnterior * signo(z) * 2^-i
                  z = z - signo(z) * arctan(2^-i)

            Fin Desde

      Sino, Vectorización entonces

            Desde i = 0 hasta Numero_De_Iteraciones-1

                  xAnterior = x
                  yAnterior = y

                  x = x + yAnterior * signo(yAnterior) * 2^-i
                  y = y - xAnterior * signo(yAnterior) * 2^-i
                  z = z + signo(yAnterior) * arctan(2^-i)

            Fin Desde

      Fin Si

Fin CORDIC
```

Las variables `x0`, `y0` y `z0`, contienen los valores de entrada para el algoritmo. Estos valores se asignan a las variables `x`, `y` y `z`. Las variables `xAnterior` e `yAnterior` se utilizan para recuperar el valor que poseían las variables `x` e `y` en la iteración anterior. El pseudocódigo anterior representa

al algoritmo CORDIC para el caso circular y considera ambos modos de operación, rotación y vectorización.

La traducción a VHDL del pseudocódigo es prácticamente directa y se utilizó un procedimiento para describir el algoritmo. Este procedimiento se incluyó en un package junto con sus tipos y funciones auxiliares. El encabezado de dicho package se muestra a continuación:

```
Package CORDIC is

      constant Iteraciones:integer:=32;
      type Mode is (Rotating,Vectoring);

      procedure cordic(x0,y0,z0:in real;xo,yo,zo:out real;Modo:in Mode);

end CORDIC;
```

El procedimiento que implementa el algoritmo CORDIC recibe los tres valores de entrada en las variables `x0`, `y0` y `z0`, asi como el modo de operación que puede ser `Rotating` para el modo rotación o `Vectoring` para el modo vectorización. Los valores de salida se retornan en los parámetros `xo`, `yo` y `zo`. A continuación se transcribe el cuerpo de package

```
Package body CORDIC is

      function sgn(x:in real) return real is
      begin
            if x>=0.0 then
                  return 1.0;
            else
                  return -1.0;
            end if;
      end;

      procedure cordic(x0,y0,z0:in real;xo,yo,zo:out real;Modo:in Mode) is
      variable x:real:=x0;
      variable y:real:=y0;
      variable z:real:=z0;
      variable xAnt:real;
      begin

            if Modo=Rotating then
                  for i in 0 to Iteraciones-1 loop
                        xAnt:=x;
                        x:=x-y*sgn(z)*2.0**(-i);
                        y:=y+xAnt*sgn(z)*2.0**(-i);
                        z:=z-sgn(z)*arctan(2.0**(-i));
                  end loop;
            else
                  for i in 0 to Iteraciones-1 loop
                        xAnt:=x;
                        x:=x+y*sgn(y)*2.0**(-i);
                        z:=z+sgn(z)*arctan(2.0**(-i));
                        y:=y-xAnt*sgn(y)*2.0**(-i);
                  end loop;
            end if;
```

```
        xo:=x;
        yo:=y;
        zo:=z;

    end cordic;

end CORDIC;
```

En la especificación del package se declara una constante para almacenar el número de iteraciones. El número de iteraciones se ha establecido en 32. La elección del número 32 es arbitraria, ya que el propósito de la descripción funcional es únicamente el de validar el funcionamiento del algoritmo y posteriormente puede ser modificada fácilmente.

Se realizó la descripción de una función auxiliar para calcular el signo de un número real. La descripción se hizo debido a que la función que provee la biblioteca de funciones matemáticas de VHDL retorna cero en caso de que su argumento sea cero por lo que no se ajustaba a las necesidades.

Finalmente se presenta la descripción de un procedimiento que encapsula al algoritmo CORDIC en VHDL. Como se puede apreciar, la traducción del pseudocódigo a la versión algorítmica del algoritmo es prácticamente directa.

4.2.2 El banco de pruebas para la descripción funcional algorítmica

La descripción funcional del algoritmo se encapsuló en un package de VHDL junto con sus funciones y tipos de datos auxiliares. Mediante esta organización, el procedimiento que resuelve el algoritmo puede ser incluido en cualquier proyecto como un componente mas de la biblioteca de trabajo `WORK`.

Para verificar el funcionamiento de la descripción se desarrollaron dos bancos de pruebas utilizando archivos y la metodología algorítmica. Los patrones de verificación se generaron mediante un algoritmo y los resultados se almacenaron en archivos para realizar comparaciones.

Los bancos de pruebas se describieron mediante una entidad sin comunicación con el exterior, como se explicó en la sección 3.7. En las arquitecturas correspondientes a dicha entidad, se incluye el package CORDIC para llevar a cabo la simulación. La arquitectura del primer banco de pruebas está implementada como un proceso que itera sobre los ángulos θ entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$, ya que éste es el intervalo de convergencia del algoritmo. El segundo banco de pruebas verifica los resultados del algoritmo para los ángulos $-\frac{\pi}{2},-\frac{\pi}{3},-\frac{\pi}{4},-\frac{\pi}{6},0,\frac{\pi}{6},\frac{\pi}{4},\frac{\pi}{3},\frac{\pi}{2}$.

Conforme se generan los diversos ángulos, éstos son suministrados a la función `cordic`. Por lo tanto para calcular el seno y el coseno de un ángulo θ (denominado `tita`), se debe invocar al procedimiento con los siguientes parámetros

```
        cordic( K, 0.0, tita, cos, sen, zo, Rotating );
```

en donde `K` es la constante definida en el capítulo 1 y representa el factor de corrección del algoritmo, `tita` es el ángulo para el cual se calculan el seno (`sen`) y el coseno (`cos`). El algoritmo

debe operar en modo rotación. Los valores de los distintos parámetros para calcular el seno y el coseno, se deducen de la expresión dada en el apartado 1.2.

El arcotangente puede calcularse operando el algoritmo en modo vectorización. Se debe suministrar al procedimiento la tangente del ángulo cuyo arcotangente se quiere calcular. La invocación al procedimiento es de la siguiente forma

```
cordic( 1.0, tan(tita), 0.0, xo, yo, atan, Vectoring );
```

La función `tan()` es proporcionada por la biblioteca matemática de VHDL. Los parámetros se pasan de acuerdo a lo explicado en el apartado 1.4.

El código fuente correspondiente a un banco de prueba ejemplifica en el anexo D.

4.3 Descripción de las arquitecturas particulares

Las descripciones que se explican a continuación, a diferencia de la descripción funcional algorítmica, están basadas en dos variantes arquitecturales del algoritmo CORDIC, denominadas bit-paralela desplegada y bit-paralela iterativa. Las arquitecturas de los componentes auxiliares descriptas a lo largo de este capítulo son variantes de diversas arquitecturas, es decir que cada descripción no es la única posible. Los datos de entrada y salida están representados en complemento a dos y se utiliza aritmética de punto fijo.

4.3.1 El formato numérico

Como se mencionó en la sección 4.1, las simulaciones de ambas arquitecturas se realizaron para el caso circular del algoritmo CORDIC calculandose las funciones seno y coseno en modo rotación y el arcotangente en modo vectorización.

El lenguaje VHDL soporta una implementación de los niveles lógicos estándar de la IEEE mediante las unidades de biblioteca `ieee.std_logic_1164`.

Los niveles lógicos definidos son nueve en total y se agrupan bajo el tipo `std_logic`. Los mas utilizados son el '0' o cero lógico, el '1' o uno lógico, el 'U' o desconocido (*Uninitialized)* y el 'Z' o *Alta Impedancia*. A su vez el paquete `std_logic_1164` define un tipo vector de elementos `std_logic` denominado `std_logic_vector`. Este tipo de datos agrupa en un vector los niveles lógicos definidos anteriormente. El nivel denominado 'U' resulta especialmente útil en simulación para determinar cuando a una señal aún no se le asignó un valor, facilitando la detección de errores en el diseño.

El formato numérico con el que operan las arquitecturas reúne las siguientes características:

- Los números están representados en complemento a dos.
- Se emplea aritmética de punto fijo.
- El ancho de palabra para las componentes X, Y y Z se estableció en 16 y 32 bit.
- Se utilizan los tipos de datos definidos en la unidad de bilioteca `ieee.std_logic_1164`.

Para representar los valores correspondientes al seno y al coseno, se emplearon dos bits para la parte entera de la representación numérica como muestra la tabla 4.1. Se estableció dicho formato

debido a que las funciones seno y coseno tienen como imagen valores reales x tales que $-1 \le x \le 1$ y el algoritmo converge para θ en el rango $-\frac{\pi}{2} \le \theta \le \frac{\pi}{2}$.

b_1	b_0	·	b_{-1}	b_{-2}	b_{-3}	b_{-4}	b_{-5}	...

Tabla 4.1

Si se supone un ancho de palabra de *m* bits, el valor decimal correspondiente a un número representado en este formato, se calcula de la siguiente manera [9]

$$valor\ decimal = b_1 \cdot 2^1 + b_0 \cdot 2^0 + \sum_{i=1}^{m-2} b_{-i} \cdot 2^{-i}$$

Sin embargo las arquitecturas trabajan internamente con un ancho de palabra mayor para obtener mejor exactitud en el resultado [6] y para facilitar la configuración durante la simulación. Por lo tanto los 16 y 32 bits del ancho de palabra internamente se reservan para la parte fraccionaria, con lo que los dos bits correspondientes a la parte entera se agregan como bits de extensión. Si la parte fraccionaria del número está formada por los 16 bits menos significativos de la representación se tendrá en realidad un ancho de palabra de 18 bits en total. Esta forma de representación presenta las siguientes características:

- Se aumenta la exactitud del resultado debido a que se disponen de dos bits mas en la representación.
- La parte entera puede extenderse fácilmente para calcular el arcotangente como se explicó en la sección 1.4.
- No se pierde precisión en la codificación de la tabla de arcotangentes como se verá en el capítulo 5.

A modo de ejemplo, se expresan los valores de algunos números correspondientes al formato recién explicado, ordenados en forma ascendente, en la tabla 4.2

Número Real	Relación	Bus externo (16 bits)	Bus interno (18 bits)
$-\frac{\pi}{2}$	≅	1001101101111001	100110110111100001
-1.5	=	1010000000000000	101000000000000000
-1	=	1100000000000000	110000000000000000
-0.707	≅	1101001011000001	110100101100000011
0	=	0000000000000000	000000000000000000
0.707	≅	0010110100111111	001011010011111101
1	=	0100000000000000	010000000000000000
1.5	=	0110000000000000	011000000000000000
$\frac{\pi}{2}$	≅	0110010010000111	011001001000011111

Tabla 4.2

Para calcular el arcotangente fue necesario aumentar la cantidad de bits correspondientes a la parte entera, debido a que para $-\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}$ se tiene que $-\infty < \mathrm{tg}(\theta) < \infty$, escapando a la representación en punto fijo. El cálculo del arcotangente se limitó entre los valores para $\mathrm{tg}(\theta)$ entre $\mathrm{tg}\left(-\frac{4\pi}{9}\right)$ y $\mathrm{tg}\left(\frac{4\pi}{9}\right)$ que en grados equivale a $\mathrm{tg}(-80°)$ y $\mathrm{tg}(80°)$.

Por lo tanto, el número de bits correspondientes a la parte entrea se debe aumentar para poder representar la $\mathrm{tg}\left(\pm\frac{4\pi}{9}\right) \cong \pm 5{,}671281$. Volviendo a la sección 1.4, puede observarse que el argumento para el arcotangente se expresó en forma de cociente $\frac{y_0}{x_0}$. Esto permite representar la tangente de ±∞ sustituyendo $y_0 = \pm 1$ y $x_0 = 0$. Para representar un valor v cercano a ±∞ se puede sustituir $y_0 = \pm 1$ y $x_0 = \frac{1}{v}$. De ésta forma sería posible limitar la parte entera de la representación numérica a 2 o 3 bits. Sin embargo la descripción en hardware de la recíproca $f(x) = \frac{1}{x}$ [26] introduce complejidad adicional al diseño.

Se describió además un package denominado *TypeConversion*, cuyo propósito es el de agrupar las funciones que efectúan la conversión entre los diversos formatos numéricos. Las funciones brindan la posibilidad de convertir entre los formatos decimal y el formato en punto fijo explicado anteriormente. A continuación se transcribe la especificación del package con sus dos funciones principales:

```
package TypeConversion is

      function conv_real(x:in std_logic_vector;intSize:in natural) return real;
      function conv_real(x:in real;size:in natural;intSize:in natural)
                        return std_logic_vector;
end TypeConversion;
```

La utilidad de este package radica en que las pruebas pueden ser generadas con valores reales proporcionados por el lenguaje VHDL y ser adaptados al formato utilizado en el diseño.

La primera función efectúa la conversión del sistema de punto fijo al tipo de datos `REAL` provisto por el lenguaje VHDL. La segunda función realiza la conversión inversa, es decir, recibe un número de tipo `REAL`, el ancho de palabra al cual convertir y la cantidad de dígitos correspondientes a la parte entera. Retorna un vector binario con el formato anteriormente descripto.

4.3.2 Componentes comunes a ambas arquitecturas

Como se explicó anteriormente, el algoritmo CORDIC opera efectuando un conjunto de iteraciones. En cada iteración se realizan sumas y desplazamientos, por lo tanto ambas arquitecturas descriptas hacen uso de un sumador, una unidad de desplazamiento y un multiplexor de dos bits para determinar el modo de operación. Tanto para la arquitectura desplegada como para la iterativa pueden emplearse la misma arquitectura del sumador y del multiplexor. Sin embargo la unidad de

desplazamiento difiere en ambos casos. Esto se debe a que en la arquitectura desplegada la unidad de desplazamiento es distinta para cada iteración, como se explico en la sección 1.6.2, por lo que puede ser cableada. En la arquitectura iterativa, la unidad de desplazamiento se reutiliza incrementándose la cantidad de desplazamientos efectuados, proporcionalmente al número de iteración.

4.3.2.1 El sumador completo (Full-Adder)

En la figura 4.2 se muestra un posible esquema de un sumador completo de 1 bit. Posee tres señales de entrada y dos de salida. Las entradas corresponden a los valores que se suman y al acarreo (carry). Las salidas corresponden al resultado de la suma y al acarreo generado. Las señales de acarreo hacen posible que al instanciar varias celdas se puedan construir sumadores con un ancho de bits determinado. Cabe aclarar que ésta no es la única arquitectura existente de un sumador completo, sin embargo fue la que se utilizó en la descripción con VHDL. Otras arquitecturas pueden confeccionarse por ejemplo utilizando sumadores parciales (Half-Adder) [7].

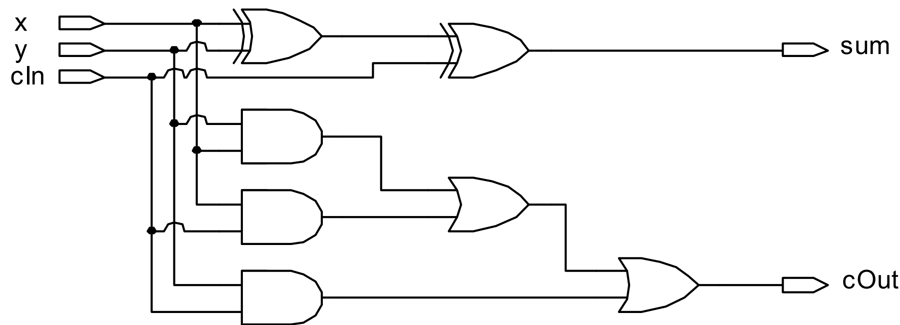


Figura 4.2

También se muestra el código VHDL correspondiente al sumador a nivel de compuertas lógicas. El estilo de descripción utilizado es de flujo de datos. No se utilizó el estilo estructural, debido a que VHDL incorpora los operadores lógicos predefinidos, con lo que no es necesario volver a implementarlos como entidades.

```
entity FullAdder is

      port(
            x:    in std_logic;
            y:    in std_logic;
            cIn:  in std_logic;
            sum:  out std_logic;
            cOut: out std_logic
      );

end FullAdder;

architecture FullAdderDataflow of FullAdder is
begin

      cOut<=(x and y) or (x and cIn) or (y and cIn);
      sum<=x xor y xor cIn;

end FullAdderDataflow;
```

4.3.2.2 La unidad de suma

Para sumar dos números binarios de un ancho de palabra determinado, se pueden combinar en secuencia los sumadores completos (Full-Adder) originando diversas arquitecturas como ser: *Ripple Carry Adder, Ripple Carry Bypass Adder, Carry Look-ahead Adder*, etc. [7] [8]. Se ha optado por describir la arquitectura denominada *Ripple Carry Adder* porque es la mas sencilla en cuanto a descripción, pero la que ofrece el desempeño mas pobre. Sin embargo no se realizarán pruebas de desempeño considerando el tiempo, con lo que esta arquitectura satisface la descripción. En esta arquitectura el carry de salida generado en una etapa pasa a la etapa siguiente como carry de entrada, tal como se muestra en la figura 4.3.

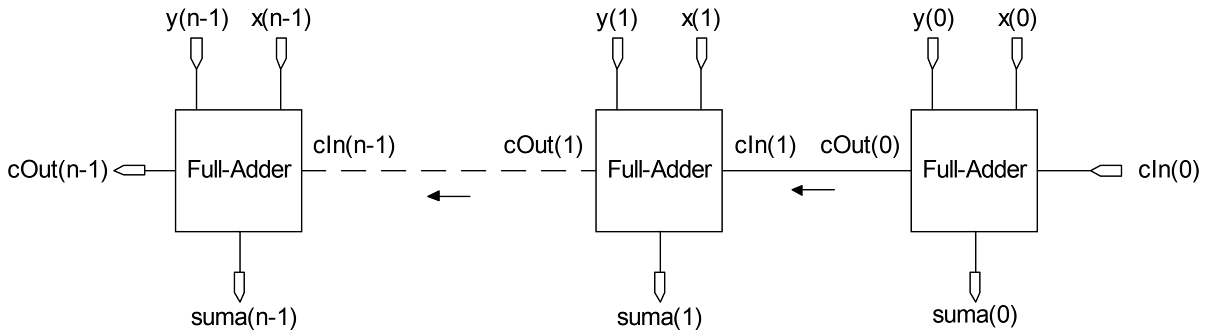


Figura 4.3

Esta arquitectura puede extenderse fácilmente a un sumador algebraico [8] [12] agregando una etapa de inversión a cada componente del vector binario `y`, y estableciendo la señal del carry de entrada (`cIn`) al valor binario 1. Este carry se suma al valor invertido, efectuándose de esta forma la resta de números en complemento a dos. El esquema final se muestra en la figura 4.4.

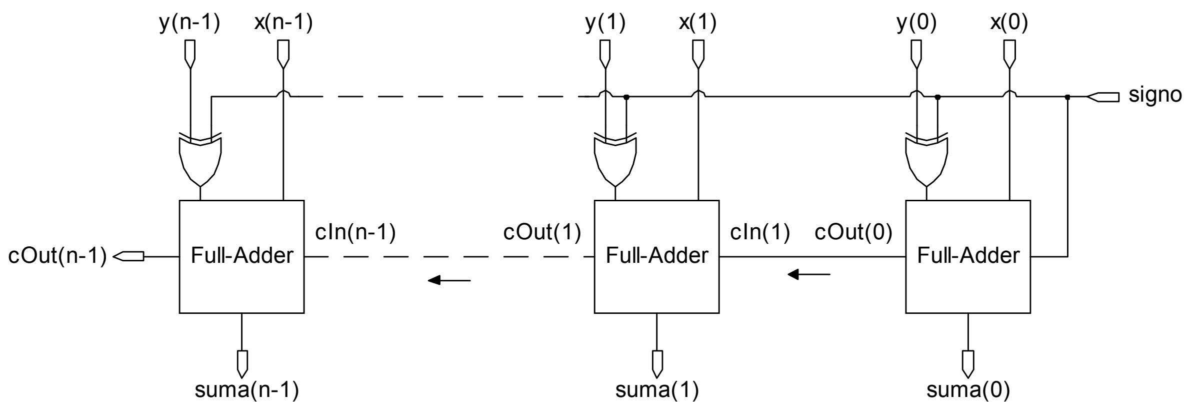


Figura 4.4

El código en VHDL correspondiente al sumador algebraico se expone a continuación. Los estilos empleados para su descripción son estructural y de flujo de datos. El estilo de flujo de datos puede observarse en la codificación de la etapa de inversión.

```
entity Adder is

      port(
            a:    in std_logic_vector;
            b:    in std_logic_vector;
            cIn:  in std_logic;
            op:   in std_logic;
            sum:  out std_logic_vector;
            cOut: out std_logic
      );

end;

architecture AdderStructure of Adder is

      component FullAdder
            port(
                  x:    in std_logic;
                  y:    in std_logic;
                  cIn:  in std_logic;
                  sum:  out std_logic;
                  cOut: out std_logic
            );
      end component;


      signal c:std_logic_vector(sum'left downto 0);
      signal a2,b2:std_logic_vector(b'left downto 0);

begin

      a2<=a;

      Stages:
      for i in sum'left downto 0 generate

            LowBit:
            if i=0 generate
                  b2(0)<=b(0) xor op;
                  FA: FullAdder port map(a2(0),b2(0),cIn,sum(0),c(0));
            end generate;

            OtherBits:
            if i/=0 generate
                  b2(i)<=b(i) xor op;
                  FA: FullAdder port map(a2(i),b2(i),c(i-1),sum(i),c(i));
            end generate;

      end generate;

      cOut<=c(sum'left);

end AdderStructure;
```

Para replicar estructuras VHDL incluye una construcción denominada GENERATE (sección 3.6.19). La descripción estructural del sumador se lleva a cabo replicando mediante la construcción FOR...GENERATE el sumador completo explicado en la sección 4.4.2.1.

4.3.2.3 El multiplexor de dos bits

Un esquema del multiplexor de dos bits fue introducido en la sección 3.3 con el objeto de ilustrar los diversos estilos de descripción en VHDL. El multiplexor tiene tres entradas y una salida. Se utiliza para tomar el valor de una de las dos señales de datos de entrada de acuerdo al valor de la señal de selección. Las entradas se denominan `a`, `b` y `sel`. Cuando `sel` vale '0' el valor de la entrada a pasa a la salida `c`, en caso contrario lo hace el valor de `b`. En la figura 4.5 se expone el esquema del multiplexor.

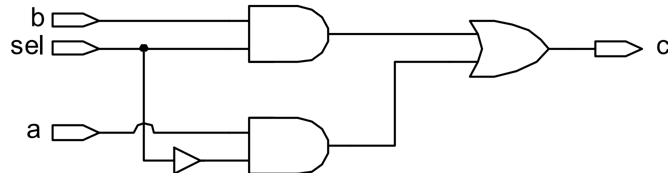


Figura 4.5

La descripción que se ha escogido es la que utiliza el estilo de flujo de datos y se muestra a continuación:

```
entity Mux is

      port(
            a:    in std_logic;
            b:    in std_logic;
            sel:  in std_logic;
            c:    out std_logic
      );

end Mux;

architecture MuxDataflow of Mux is
begin
      c<=(a and (not sel)) or (b and sel);
end MuxDataflow;
```

El diagrama que se utilizará para esquematizar a un multiplexor de dos bits es el correspondiente a la figura 4.6.

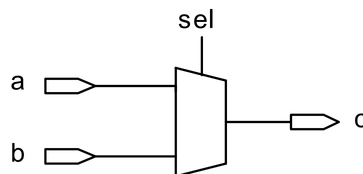


Figura 4.6

4.3.3 La descripción bit-paralela desplegada y sus componentes

La primera descripción de una arquitectura particular del algoritmo CORDIC se basó en la arquitectura bit-paralela desplegada. El diseño está dividido en etapas acorde al número de iteraciones. Como resultado se obtiene un circuito combinatorio. A continuación se explica la

descripción de la unidad de desplazamiento cableada y por último la versión desplegada del algoritmo.

4.3.3.1 La unidad de desplazamiento aritmético cableada

La división de un número binario por una potencia de dos, puede llevarse a cabo mediante el desplazamiento a derecha del mismo una determinada cantidad de veces. Cuando se explicó la arquitectura bit-paralela desplegada en la sección 1.6.3, se mencionó que la unidad de desplazamiento es distinta para cada etapa. Esto hace posible la descripción de una unidad de desplazamiento cableada [3]. El mecanismo de extensión de signo se lleva a cabo asignando el bit más significativo a los bits más significativos del resultado de acuerdo a la cantidad de desplazamientos. A continuación se muestra un esquema de la unidad de desplazamiento. A modo de ejemplo se ilustra para 1 desplazamiento (figura 4.7 (a)) y 3 desplazamientos (figura 4.7 (b)) sobre un operando de 8 bits.

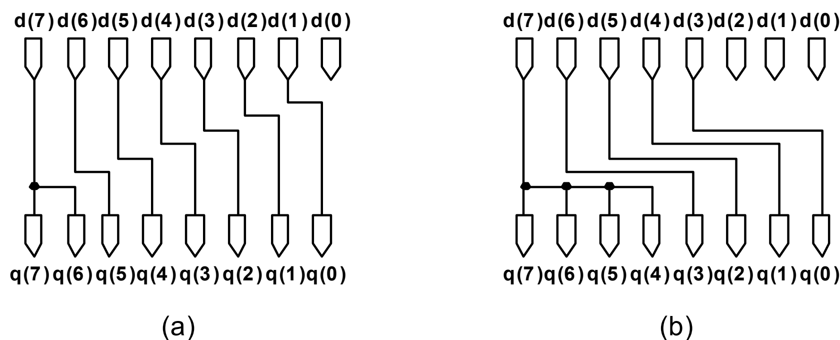


Figura 4.7

El código VHDL se muestra a continuación. Se emplearon los estilos de diseño estructural y de flujo de datos. Como se puede observar, se introduce por primera vez una constante genérica, denominada `shifts`, mediante la palabra clave `GENERIC` en la declaración de entidad. Esta constante contiene la cantidad de desplazamientos a efectuar de acuerdo al número de iteración. Por ejemplo si `shifts` vale 1 y 3 se obtienen arquitecturas correspondientes a los esquemas anteriores.

```
entity WiredRightShifter is

      generic(shifts:natural);
      port(
            d     :in std_logic_vector;
            q     :out std_logic_vector
      );

end WiredRightShifter;

architecture WiredRightShifterStructure of WiredRightShifter is
begin

      SignExt:
      for i in d'high downto d'high-shifts+1 generate
            q(i)<=d(d'high);
      end generate;
```

```
	WShift:
	for i in d'high-shifts downto 0 generate
		q(i)<=d(i+shifts);
	end generate;

end WiredRightShifterStructure;
```

4.3.3.2 La entidad y arquitectura correspondientes a una iteración

El esquema que se muestra en la figura 4.8 representa una iteración o etapa dentro de la arquitectura bit-paralela desplegada del algoritmo CORDIC. Utiliza como componentes un sumador, una unidad de desplazamiento y un multiplexor. El sumador es instanciado tres veces. Cada instancia del sumador corresponde a una de las componentes *X, Y* y *Z*. La unidad de desplazamiento se instancia dos veces, una para *X* y otra para *Y*, y el multiplexor sólo una vez y se utiliza para determinar el signo de la componente *Y* ó *Z* dependiendo del modo de operación.

La figura muestra una iteración o etapa del diseño. El identificador *i* dentro de las unidades de desplazamiento y de la constante para el acumulador angular refleja la iteración actual y por consiguiente el número de desplazamientos a derecha y la constante correspondiente a esta etapa.

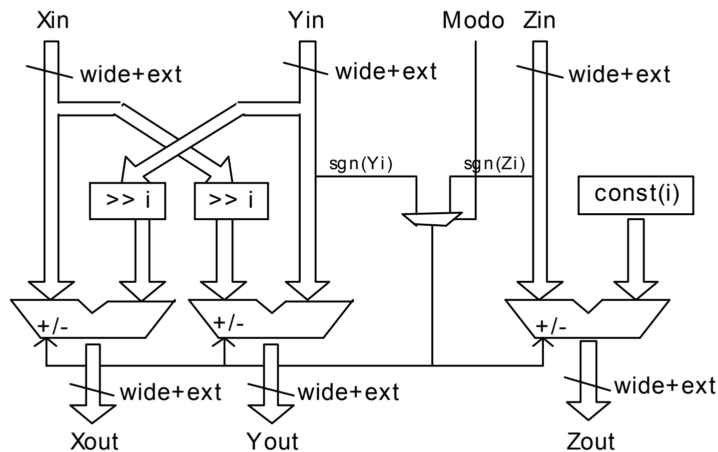


Figura 4.8

La entidad recibe cuatro señales de entrada. Tres señales representan a las componentes *X* (`Xin`), *Y* (`Yin`) y *Z* (`zIn`) que proveen a la etapa actual los valores de la etapa anterior. La cuarta señal determina el modo de operación. Los valores correspondientes a los componentes *X* e *Y* se desplazan a derecha el número indicado de veces y se suman o restan al valor del componente opuesto. El valor correspondiente a *Z* se suma o resta con la constante correspondiente a la iteración actual. Las señales de salida son tres y toman el resultado de la etapa actual. La etiqueta `wide+ext` indica el ancho del bus interno.

La descripción en VHDL se realizó con el estilo estructural principalmente, y se muestra a continuación. Los parámetros genéricos representan la iteración actual (`iteration`), el ancho de palabra (`wide`), la extensión interna de bits (`ext`) como se explicó en la sección 4.4.1 y la constante que se suma al acumulador angular (`const`). La iteración actual se utiliza para que cada etapa conozca el número que le corresponde en la instanciación final y pueda configurar adecuadamente las unidades de desplazamiento.

```
entity Iteration is

      generic(iteration,wide,ext:in natural;const:std_logic_vector);
      port(
            mode  :in std_logic;
            Xin   :in std_logic_vector(wide+ext-1 downto 0);
            Yin   :in std_logic_vector(wide+ext-1 downto 0);
            Zin   :in std_logic_vector(wide+ext-1 downto 0);

            Xout  :out std_logic_vector(wide+ext-1 downto 0);
            Yout  :out std_logic_vector(wide+ext-1 downto 0);
            Zout  :out std_logic_vector(wide+ext-1 downto 0)
      );

end Iteration;

architecture IterationStructure of Iteration is

      component Mux is
            port(
                  a     :in std_logic;
                  b     :in std_logic;
                  sel   :in std_logic;
                  c     :out std_logic
            );
      end component;

      component Adder is
            port(
                  a     :in std_logic_vector;
                  b     :in std_logic_vector;
                  cIn   :in std_logic;
                  op    :in std_logic;
                  sum   :out std_logic_vector;
                  cOut  :out std_logic
            );
      end component;

      component WiredRightShifter is
            generic(shifts:natural);
            port(
                  d     :in std_logic_vector;
                  q     :out std_logic_vector
            );
      end component;

      signal S0,S1,Si,Sj:std_logic;
      signal Xshifted,Yshifted:std_logic_vector(wide+ext-1 downto 0);
      signal Xtemp,Ytemp,Ztemp,wconst:std_logic_vector(wide+ext-1 downto 0);
      signal cOutX,cOutY,cOutZ:std_logic;

begin

      It0:
      if iteration=0 generate
            Xshifted<=Xin;
            Yshifted<=Yin;
      end generate;

      ItN0:
```

```
        if iteration>0 generate
              SHX: WiredRightShifter generic map(iteration)
                                     port map(Xin,Xshifted);
              SHY: WiredRightShifter generic map(iteration)
                                     port map(Yin,Yshifted);
        end generate;

        ADDX: Adder port map(Xin,Yshifted,S0,S0,Xtemp,cOutX);
        ADDY: Adder port map(Yin,Xshifted,S1,S1,Ytemp,cOutY);
        ADDZ: Adder port map(Zin,wconst,S0,S0,Ztemp,cOutZ);

        wconst(wide-1 downto 0)<=const;
        wconst(wide+ext-1 downto wide)<=(others=>'0');

        Si<=Zin(wide+ext-1);
        Sj<=not Yin(wide+ext-1);
        S0<=not S1;
        SMUX: Mux port map(Sj,Si,mode,S1);

        Xout<=Xtemp;
        Yout<=Ytemp;
        Zout<=Ztemp;

end IterationStructure;
```

En la porción declarativa de la arquitectura se declaran los tres componentes; el sumador, la unidad de desplazamiento aritmética y el multiplexor. El sumador es instanciado tres veces en el cuerpo de la arquitectura y el multiplexor una vez. La unidad de desplazamiento sólo se instancia si el número de iteración es mayor que cero, debido a que en la primera iteración se desplaza cero veces, lo que equivale a no realizar desplazamiento alguno. Esta instanciación selectiva de la unidad de desplazamiento se hace mediante la sentencia `IF...GENERATE` que incorpora el lenguaje VHDL. De acuerdo a la condición que se cumpla, se genera una u otra parte de la arquitectura.

4.3.3.3 Descripción final de la arquitectura bit-paralela desplegada

Una vez diseñada la etapa genérica de la versión desplegada, ésta debe ser instanciada tantas veces como iteraciones efectúe el algoritmo. La entidad que se describe a continuación corresponde al diseño desplegado del algoritmo CORDIC e instancia el componente Iteration, correspondiente a la entidad explicada en la sección anterior. El esquema final corresponde al que se explicó en la sección 1.6.3.

El código VHDL se muestra a continuación. El estilo utilizado nuevamente es estructural, salvo en la parte del código que realiza la asignación de las señales de entrada `Xo`, `Yo` y `Zo` y las señales de salida `Xn`, `Yn` y `Zn`, que es de flujo de datos.

La entidad tiene cuatro señales de entrada de las cuales tres corresponden a los valores iniciales de los componentes `Xo`, `Yo` y `Zo` y el modo de operación, y tres señales de salida `Xn`, `Yn` y `Zn` que constituyen los valores de salida del algoritmo. Además se proveen cuatro constantes genéricas correspondientes al número de iteraciones, al ancho de palabra y a los valores de las constantes que se suman al acumulador angular.

La arquitectura correspondiente a la entidad `ParalellUnrolledCORDIC`, simplemente replica mediante un esquema `FOR...GENERATE` tantas etapas como iteraciones se especifiquen. Dichas etapas se conectan unas con otras mediante las señales `wireX`, `wireY` y `wireZ` que

constituyen vectores de conexión. La etapa inicial a su vez recibe los valores de las señales de entrada. Los valores, producto de la etapa final, se asignan a las salidas. Las constantes que se suman al acumulador angular se pasan a la entidad `Iteration` una a una como un parámetro genérico.

```
entity ParalellUnrolledCORDIC is

      generic(iterations,wide,ext:natural;arctanLUT:STDLVectorW);
      port(
            mode  :in std_logic;
            Xo    :in std_logic_vector(wide-1 downto 0);
            Yo    :in std_logic_vector(wide-1 downto 0);
            Zo    :in std_logic_vector(wide-1 downto 0);

            Xn    :out std_logic_vector(wide-1 downto 0);
            Yn    :out std_logic_vector(wide-1 downto 0);
            Zn    :out std_logic_vector(wide-1 downto 0)
      );

end ParalellUnrolledCORDIC;

architecture ParalellUnrolledCORDICStructure of ParalellUnrolledCORDIC is

      type ConnectVector is array(iterations downto 0)
                                    of std_logic_vector(wide+ext-1 downto 0);

      component Iteration is
            generic(iteration,wide,ext:in natural;const:std_logic_vector);
            port(
                  mode  :in std_logic;
                  Xin   :in std_logic_vector(wide+ext-1 downto 0);
                  Yin   :in std_logic_vector(wide+ext-1 downto 0);
                  Zin   :in std_logic_vector(wide+ext-1 downto 0);

                  Xout  :out std_logic_vector(wide+ext-1 downto 0);
                  Yout  :out std_logic_vector(wide+ext-1 downto 0);
                  Zout  :out std_logic_vector(wide+ext-1 downto 0)
            );
      end component;

      signal wireX,wireY,wireZ:ConnectVector;

begin

      UC:
      for i in 0 to iterations-1 generate
            Iter: Iteration generic map(i,wide,ext,arctanLUT(i))
                              port map (mode,wireX(i),wireY(i),wireZ(i),
                                          wireX(i+1),wireY(i+1),wireZ(i+1));
      end generate;

      wireX(0)(wide+ext-1 downto ext)<=Xo;
      wireX(0)(ext-1 downto 0)<=(others=>'0');

      wireY(0)(wide+ext-1 downto ext)<=Yo;
      wireY(0)(ext-1 downto 0)<=(others=>'0');

      wireZ(0)(wide+ext-1 downto ext)<=Zo;
      wireZ(0)(ext-1 downto 0)<=(others=>'0');
```

```
	Xn<=wireX(iterations)(wide+ext-1 downto ext);
	Yn<=wireY(iterations)(wide+ext-1 downto ext);
	Zn<=wireZ(iterations)(wide+ext-1 downto ext);

end ParalellUnrolledCORDICStructure;
```

4.3.4 La descripción bit-paralela iterativa y sus componentes

La variante arquitectural que se describe a continuación difiere de la versión desplegada en que se reutilizan los sumadores y unidades de desplazamiento en cada iteración. A tal fin, se agrega memoria y una señal de reloj al diseño. Como resultado se obtiene una estructura mas compacta que la anterior pero a la vez mas compleja. El circuito que se obtiene deja de ser combinatorio para convertirse en secuencial. Sin embargo mantiene una parte combinatoria compuesta por los sumadores y la unidad de desplazamiento.

4.3.4.1 La unidad de desplazamiento aritmético para la descripción iterativa

Cuando se estudió la arquitectura bit-paralela desplegada, se introdujo una unidad de desplazamiento cableada. Se puede utilizar una unidad cableada debido a que los desplazamientos a derecha son fijos para cada etapa. Por el contrario, en la arquitectura bit-paralela iterativa la unidad de desplazamiento, así como los sumadores se reutilizan en cada etapa y la cantidad de desplazamientos se incrementa en cada iteración.

La arquitectura para la unidad de desplazamiento que se presenta aquí se denomina *Barrel Shifter Aritmético* porque proporciona el mecanismo de extensión de signo. Un Barrel Shifter [11] está compuesto por una serie de etapas sucesivas. Cada etapa i efectúa un desplazamiento a distancia 2^{-i} sobre la etapa anterior. Por lo tanto una unidad de desplazamiento como el Barrel Shifter requiere de $\log_2 n$ etapas para desplazar por completo una palabra de n bits. La ventaja que presenta este tipo de arquitectura es que se trata de un circuito combinatorio. Se eligió esta arquitectura para la unidad de desplazamiento en favor de otras existentes [7] [8], debido a que se simplifica el diseño de la unidad de control (sección 4.4.4.11), ya que no debe controlar la cantidad de desplazamientos efectuados.

En la figura 4.9 se ejemplifica un Barrel Shifter Aritmético. Para simplificar el diagrama, el ancho de palabra se limitó a 8 bit. Cada celda representa un multiplexor de dos bits como el introducido en la sección 4.4.2.3.

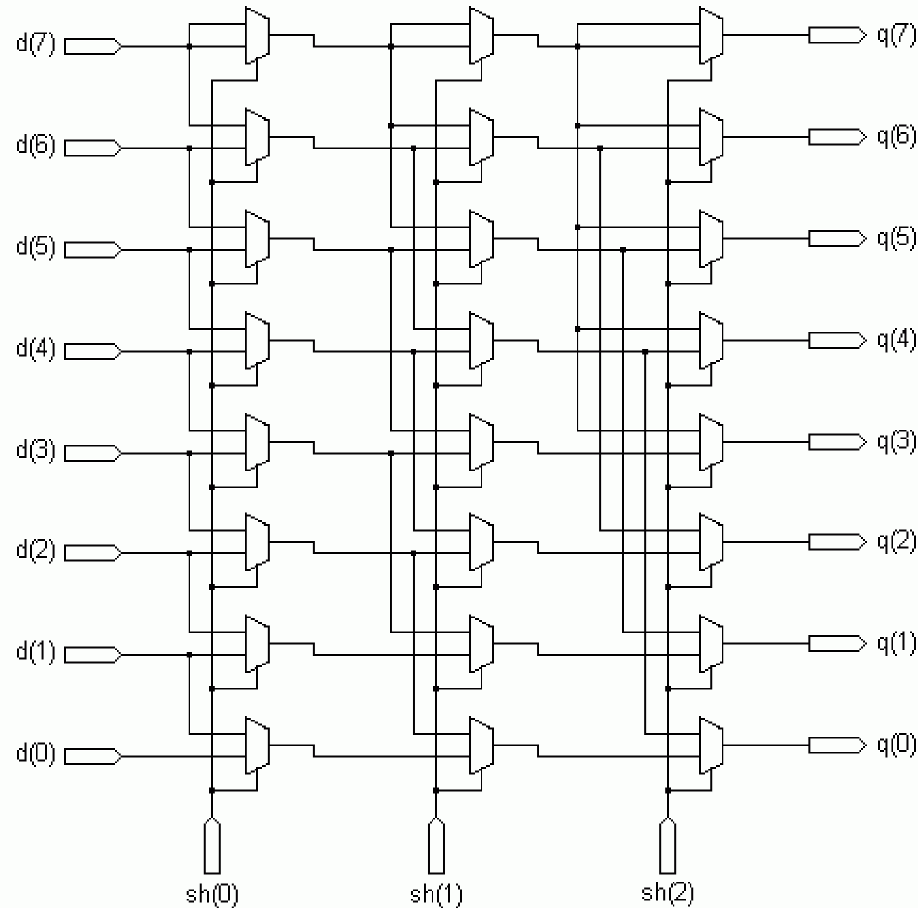


Figura 4.9

Las señales proporcionadas por el vector `sh` determinan la cantidad de desplazamientos a efectuar sobre la palabra en el vector binario `d`. Cada bit del vector `sh` codifica una cantidad de desplazamientos que es potencia de dos, así si `sh(0)` está activado, se debe realizar un desplazamiento, si `sh(1)` está activado, dos desplazamientos y para `sh(2)`, cuatro desplazamientos. Para realizar tres desplazamientos se pueden poner a 1 los bits correspondientes a `sh(0)` y `sh(1)`. Si los tres bits del vector `sh` están puestos a cero, no se efectúa desplazamiento y si están activados, se efectúa la máxima cantidad de desplazamientos. Para un dígito binario del vector binario `d` cada etapa comprende un multiplexor que determina si se debe realizar un desplazamiento. El resultado del desplazamiento se obtiene en el vector binario de salida `q`.

El código VHDL correspondiente al Barrel Shifter Aritmético se transcribe a continuación. Se utilizó el estilo de descripción estructural. Para describir el Barrel Shifter Aritmético se utilizaron dos bloques `GENERATE` anidados. VHDL permite anidar tantos bloques `GENERATE` como se desee, para describir estructuras complejas. El bloque `GENERATE` interno genera tantos multiplexores como el ancho de palabra. El bloque externo genera cada una de las etapas, de acuerdo al máximo que se desee desplazar, que corresponde al $\log_2 n$, donde n es el ancho de palabra. Esta descripción resulta interesante porque ilustra la potencia del lenguaje VHDL.

Los parámetros genéricos que recibe la entidad (`stages` y `wide`), corresponden al número de etapas y al ancho de palabra.

```
entity BarrelRightShifter is

	generic(stages,wide:natural);
	port(
		d:	in std_logic_vector;
		sh:	in std_logic_vector;
		q:	out std_logic_vector
	);

end BarrelRightShifter;

architecture BarrelRightShifterStructure of BarrelRightShifter is

	component Mux is
		port(
			a:	in std_logic;
			b:	in std_logic;
			sel:	in std_logic;
			c:	out std_logic
		);
	end component;

	type qIntStages is array(0 to stages) of
				std_logic_vector(wide-1 downto 0);

	signal qInt:qIntStages;

begin

	BSAStages:
	for i in 0 to stages-1 generate

		WideR:
		for j in d'range generate

			SignExt:
			if j>=((wide-1)-(2**i)) generate
				MUXS: Mux port
				map(qInt(i)(j),qInt(i)(wide-1),sh(i),qInt(i+1)(j));
			end generate;

			BShift:
			if j<((wide-1)-(2**i)) generate
				MUXS: Mux port
				map(qInt(i)(j),qInt(i)(j+(2**i)),sh(i),qInt(i+1)(j));
			end generate;

		end generate;

	end generate;

	qInt(0)<=d;
	q<=qInt(stages);

end BarrelRightShifterStructure;
```

4.3.4.2 Las compuertas lógicas de múltiples entradas

VHDL proporciona las compuertas lógicas en forma de operadores predefinidos. Sin embargo dichos operadores toman únicamente dos operandos y muchas veces es necesario utilizar

compuertas de más de dos entradas. El lenguaje no proporciona compuertas genéricas de múltiples entradas, por ello se describieron una compuerta AND y una compuerta OR. Esta estructura resulta útil cuando se deben configurar compuertas de diversas entradas. Cada compuerta recibe un vector de entrada y retorna el resultado de la operación lógica sobre los elementos del vector. En la figura 4.10 se ejemplifica una compuerta AND de *n* entradas de un vector denominado `d`. La estructura de la compuerta OR es similar.

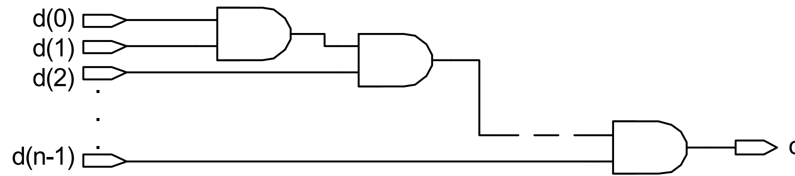


Figura 4.10

La descripción en VHDL se realizó de forma estructural en combinación con flujo de datos. La porción de código descripta utilizando flujo de datos corresponde a la operación AND entre dos entradas. Nuevamente para generar la estructura de la compuerta se utilizó la sentencia `GENERATE`. Dentro del `FOR...GENERATE` se utilizan dos sentencias `IF...GENERATE` que discriminan el caso para la primera compuerta que toma sus entradas de los elementos 0 y 1 del vector de las restantes entradas que se combinan con la salida de la compuerta anterior. La cantidad de compuertas generadas depende del tamaño del vector `d`.

```
entity MultiAND is

      port(
            d:in std_logic_vector;
            c:out std_logic
      );

end MultiAND;

architecture MultiANDStructure of MultiAND is

      signal cTemp:std_logic_vector(d'left-1 downto 0);

begin

      AndG:
      for i in 0 to d'left-1 generate

            And0:
            if i=0 generate
                  cTemp(i)<=d(0) and d(1);
            end generate;

            AndN:
            if i/=0 generate
                  cTemp(i)<=d(i+1) and cTemp(i-1);
            end generate;

      end generate;

      c<=cTemp(d'left-1);

end MultiANDStructure;
```

4.3.4.3 La tabla de búsqueda

La tabla de búsqueda que almacena las arcotangentes para el caso iterativo del algoritmo CORDIC puede almacenarse en una memoria ROM, como se explicó en la sección 1.6.1.

Una memoria ROM posee una señal de entrada correspondiente a la dirección a la que se debe acceder y una señal de salida correspondiente al dato que se obtiene de la dirección.

El esquema básico de una memoria ROM se puede describir como sigue: La dirección proporcionada es decodificada por un demultiplexor para activar la línea de dirección correspondiente en la memoria. Al activarse la línea en la memoria ROM, los valores que contiene se hacen visibles en la señal de salida.

El esquema se muestra en la figura 4.11. Se ejemplifica para una memoria ROM con un bus de direcciones de 2 bits y un bus de datos de 4 bits.

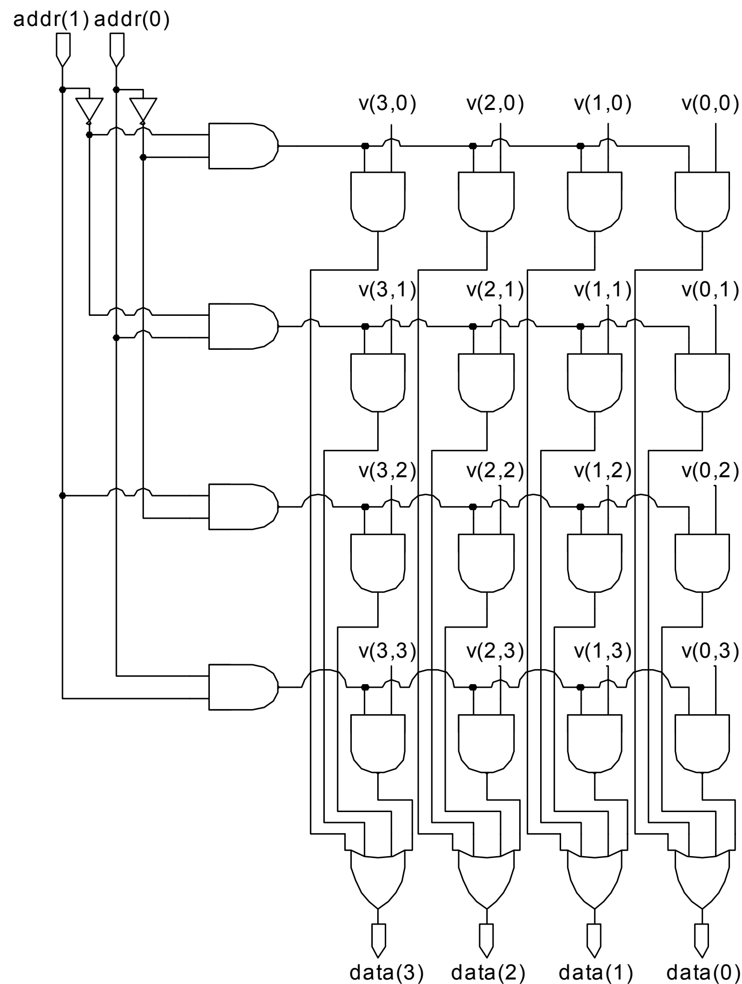


Figura 4.11

Usualmente, una memoria ROM utiliza fusibles en la intersección de las líneas provenientes del demultiplexor y las líneas de datos, y los valores binarios se codifican como fusibles sanos o quemados [12] [17]. Para realizar la descripción, los fusibles se cambiaron por compuertas AND en

donde una entrada está conectada a la señal proveniente del multiplexor y el valor de la otra entrada proviene de un vector de valores binarios denominado `v`. Los elementos de la forma `v(x,y)` representan los posibles valores de cada posición en la memoria. Se optó por esta descripción porque ilustra la descripción de una estructura compleja en VHDL y porque representa la posible estructura interna de una memoria ROM. Por otra parte la entidad en VHDL recibe un vector genérico (`arctanLUT`), con lo que la memoria puede configurarse del tamaño adecuado al problema.

La descripción VHDL hace uso de los componentes auxiliares descriptos en la sección 4.4.4.2, una compuerta AND y una compuerta OR con múltiples entradas.

```
entity ROM is

      generic(arctanLUT:STDLVectorW);
      port(
            addr:in std_logic_vector;
            data:out std_logic_vector
      );

end ROM;

architecture ROMStructure of ROM is

      component MultiOR is
            port(
                  d:in std_logic_vector;
                  c:out std_logic
            );
      end component;

      component MultiAND is
            port(
                  d:in std_logic_vector;
                  c:out std_logic
            );
      end component;


      type connectDec is array(0 to (2**(addr'left+1))-1) of
                        std_logic_vector(addr'range);
      type connectDat is array(data'range) of
                        std_logic_vector((2**(addr'left+1))-1 downto 0);

      signal connectAnd:connectDec;
      signal lineSel:std_logic_vector(0 to (2**(addr'left+1))-1);
      signal toOR:connectDat;
      signal dataIn:std_logic_vector(data'range);

begin
      ROMDecoder_i:
      for i in 0 to (2**(addr'left+1))-1 generate

            ROMDecoder_j:
            for j in 0 to addr'left generate
                  Dec_a:
                  if ((i/(2**j)) mod 2)=1 generate
                        connectAnd(i)(j)<=addr(j);
                  end generate;
```

```
            Dec_not_a:
            if ((i/(2**j)) mod 2)=0 generate
                  connectAnd(i)(j)<=not addr(j);
            end generate;
      end generate;

      MAND: MultiAND port map(connectAnd(i),lineSel(i));

end generate;


ROMData_i:
for i in data'range generate
      ROMData:
      for j in 0 to (2**(addr'left+1))-1 generate
            toOR(i)(j)<=lineSel(j) and arctanLUT(j)(i);
      end generate;

      MOR: MultiOR port map(toOR(i),dataIn(i));
end generate;

data<=dataIn;

end ROMStructure;
```

4.3.4.4 El multiplexor múltiple de dos bits

La arquitectura iterativa requiere de tres multiplexores múltiples de ancho *n*, donde *n* es el ancho de palabra. Estos multiplexores se utilizan para posibilitar el ingreso de las componentes *X*, *Y* y *Z* al sistema al comienzo del cómputo. Un multiplexor de ancho *n*, de dos bits se puede construir instanciando *n* multiplexores de dos bits que comparten la señal de selección (`sel`). Cada bit de entrada del multiplexor corresponde a dos componentes homólogos de dos vectores de entrada. El esquema recién explicado se muestra en la figura 4.12.

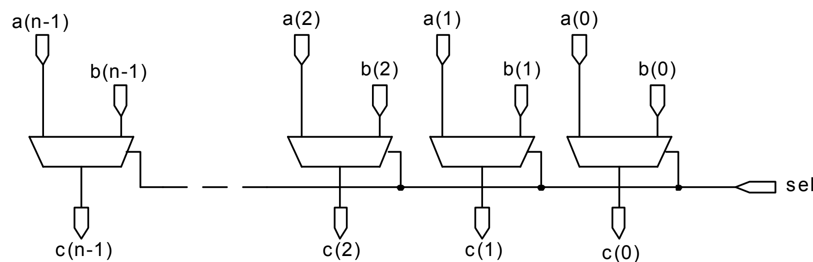


Figura 4.12

El código VHDL resultante genera *n* multiplexores de dos bits y se muestra a continuación. Se utilizó el estilo de descripción estructural. De acuerdo al valor de la señal `sel`, los contenidos vectores `a` ó `b` pasan al vector de salida `c`.

```
entity WideMux is

      port(
            a     :in std_logic_vector;
            b     :in std_logic_vector;
            sel   :in std_logic;
            c     :out std_logic_vector
      );
```

```
end WideMux;

architecture WideMuxStructure of WideMux is

      component Mux is
            port(a,b,sel:in std_logic;c:out std_logic);
      end component;

begin

      WideM:
      for i in a'range generate
            CMUX: Mux port map(a(i),b(i),sel,c(i));
      end generate;

end WideMuxStructure;
```

4.3.4.5 Circuitos secuenciales

La arquitectura bit-paralela desplegada es combinatoria. La lógica combinatoria es capaz de implementar operaciones como suma, resta y desplazamiento. Sin embargo la realización de secuencias útiles de operación con el uso exclusivo de lógica combinatoria exige disponer en cascada varias estructuras a la vez. El hardware resultante es muy costoso y rígido. Para realizar secuencias útiles y flexibles de operaciones, se deben diseñar circuitos capaces de almacenar información entre las operaciones realizadas por los circuitos de combinación. Tales circuitos se denominan *secuenciales* [12] [16].

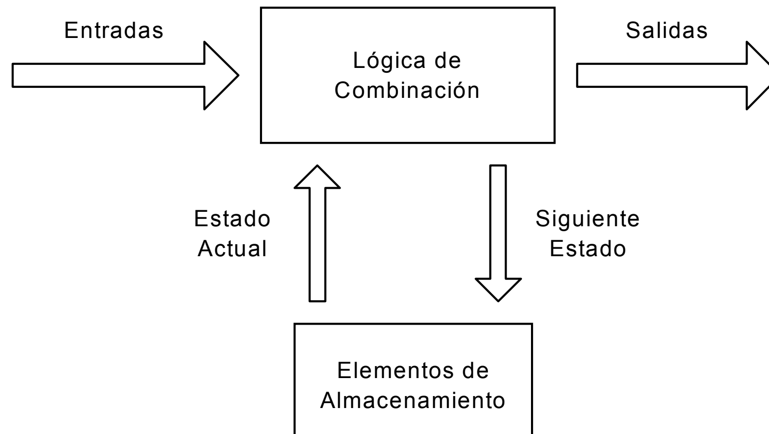


Figura 4.13

En la figura 4.13 se muestra un diagrama en bloques de un circuito secuencial compuesto por lógica combinatoria y elementos de almacenamiento. Estos elementos se interconectan para formar el circuito. Los elementos de almacenamiento son circuitos capaces de retener información binaria. La información almacenada en estos elementos define el estado del circuito secuencial. El circuito secuencial recibe información binaria de su ambiente a través de las entradas. Estas en combinación con el estado actual de los elementos de almacenamiento, determinan el valor binario de las salidas. También determinan los valores para especificar el siguiente estado de los elementos de almacenamiento. El diagrama de bloques muestra que las salidas de un circuito secuencial no

sólo son función de las entradas, sino también del estado actual de los elementos de almacenamiento. El siguiente estado de estos elementos, también es función de las entradas y el estado actual. Por lo tanto un circuito secuencial se especifica por una secuencia de tiempos de entradas, estados internos y salidas.

Existen dos tipos de circuitos secuenciales, clasificados según el tiempo en que se observan sus entradas y cambios en los estados internos. Un *circuito secuencial sincrónico* se define por el conocimiento de sus señales en estados discretos de tiempo. Un *circuito secuencial asincrónico* depende de las entradas en cualquier instante y el orden en el tiempo del cambio de las mismas [12].

Por lo general se prefieren los circuitos sincrónicos. Esto se debe a que los que responden al modelo asincrónico definen su comportamiento de acuerdo a los retardos de propagación de las compuertas y a la sincronización de los cambios en las entradas complicando su diseño.

Un circuito secuencial sincrónico emplea señales que afectan a los elementos de almacenamiento en instantes discretos de tiempo. La sincronización se consigue mediante una señal de reloj. De esta manera los elementos del circuito resultan afectados en relación con cada pulso. En la práctica estos pulsos de reloj se aplican con otras señales que especifican el cambio requerido en los dispositivos de almacenamiento. Las salidas de los elementos de almacenamiento sólo cambian de valor ante un pulso de reloj.

4.3.4.6 Los elementos de almacenamiento: Flip-flops

Los elementos de almacenamiento que se utilizan en los circuitos secuenciales sincrónicos reciben el nombre de *flip-flops*. Un flip-flop es un dispositivo binario capaz de almacenar un bit de información. Los flip-flops reciben sus entradas del circuito lógico combinatorio y también de una señal de reloj con pulsos a intervalos fijos. Estos dispositivos sólo cambian de estado en respuesta al cambio en una señal de reloj y no únicamente en respuesta a su estado. Si el flip-flop cambia cuando la señal de reloj pasa de cero a uno, se habla de flanco ascendente, de lo contrario se lo denomina flanco descendente. Hasta que no se dispare un pulso de reloj, las salidas del flip-flop no pueden cambiar, aunque las salidas del circuito de combinación que maneja sus entradas sí lo hagan. Por consiguiente la transición de un estado al siguiente sucede sólo a intervalos fijos, lo que produce una operación sincrónica [12] [13] [15].

Un flip-flop puede tener una o dos salidas, una para el valor normal del bit almacenado y otra para su complemento.

4.3.4.7 El flip-flop D

El flip-flop D puede ser visto como una celda de memoria básica. Posee cuatro señales de entrada, la entrada de datos denominada `d`, las señales de `set` y `clr` (clear) que se utilizan para establecer el valor 1 o 0 respectivamente, y una señal de reloj denominada `clk`. Además posee dos señales de salida, `q` y `qBar`, en donde se refleja el valor de la salida y su complemento. El dato que ingresa al flip-flop por la señal `d`, pasa a la salida `q` cuando la señal `clk` cambia de 0 a 1 (flanco ascendente) en el ejemplo siguiente. A partir de ese momento cualquier modificación en la entrada no es visible en la salida hasta que se produzca nuevamente un flanco ascendente en la señal de reloj. Las señales `set` y `clr` son independientes de la señal de reloj y se activan a nivel bajo (cuando son puestas a 0). En la figura 4.14 se muestra el esquema de una posible implementación del flip-flop D [13] [21].

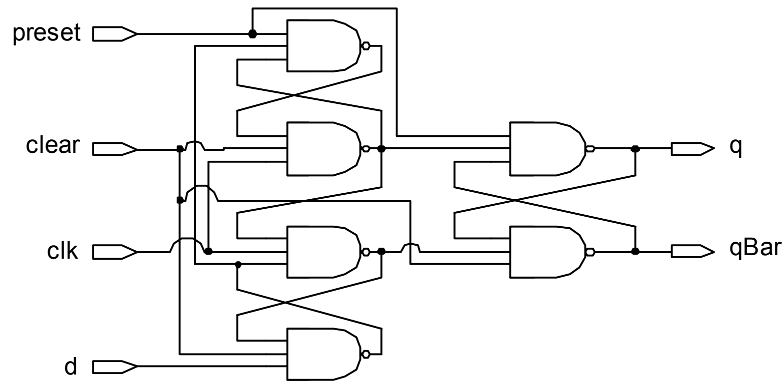


Figura 4.14

La tabla de verdad del flip-flop D, se muestra en la tabla 4.3 , el símbolo ↑ indica una transición de 0 a 1.

set	clear	clk	d	q	qBar
0	1	x	x	1	0
1	0	x	x	0	1
0	0	x	x	1	1
1	1	↑	0	0	1
1	1	↑	1	1	0
1	1	0	x	q	qBar

Tabla 4.3

La descripción en VHDL del flip-flop D se puede realizar a partir de la tabla de verdad, es decir observando su comportamiento. De esta forma obtenemos la descripción de la primera arquitectura denominada `DFFBehavior`. En esta arquitectura se emplea el estilo de descripción algorítmico. Esta primera descripción se llevó a cabo para comprender el funcionamiento del flip-flop. El atributo `event` que se utiliza en la última rama del `IF` se utiliza para detectar la ocurrencia de un evento (cambio) en una señal y en este caso para detectar el flanco ascendente de la señal `clk`.

Para la misma entidad en VHDL se describió una segunda arquitectura a nivel de compuertas lógicas denominada `DFFDataflow` ajustándose al esquema que se presentó. El estilo de descripción es de flujo de datos, y hace uso de las compuertas lógicas predefinidas del lenguaje.

En esta descripción se introduce por primera vez una especificación de configuración y se le da el nombre `DFFConf`. El propósito es hacer corresponder la entidad `DFF` con una de ambas arquitecturas. En este caso se enlaza la arquitectura de flujo de datos.

```
entity DFF is

      port(
            set   :in std_logic;
            clr   :in std_logic;
            clk   :in std_logic;
            d     :in std_logic;

            q     :out std_logic;
            qBar  :out std_logic
      );

end DFF;

architecture DFFBehavior of DFF is
begin

DFFProc:
      process(set,clr,clk)
      begin
            if set='0' then
                  q<='1';
                  qBar<='0';
            elsif clr='0' then
                  q<='0';
                  qBar<='1';
            elsif clk'event and clk='1' then
                  q<=d;
                  qBar<=not d;
            end if;

      end process;

end DFFBehavior;

architecture DFFDataflow of DFF is

      signal a,b,c,e:std_logic;
      signal qInt,qBarInt:std_logic;

begin

      a<=not(set and e and b);
      b<=not(a and clr and clk);
      c<=not(b and clk and e);
      e<=not(c and clr and d);

      qInt<=not(set and b and qBarInt);
      qBarInt<=not(qInt and clr and c);

      q<=qInt;
      qBar<=qBarInt;

end DFFDataflow;

configuration DFFConf of DFF is

      for DFFDataflow
      end for;

end configuration;
```

El diagrama que se utilizará para esquematizar un flip-flop D se muestra en la figura 4.15

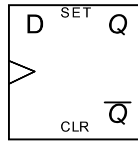


Figura 4.15

4.3.4.8 El registro

Como se explicó en la sección anterior, el flip-flop D constituye una celda de memoria básica capaz de almacenar un bit. Un registro puede considerarse como un conjunto de celdas de memoria capaz de almacenar la información binaria correspondiente a una palabra. La descripción de un registro es inmediata a partir de la descripción del flip-flop D. La entidad que describe el registro posee las mismas entradas y salidas que el flip-flop D con la única diferencia que está diseñado para almacenar una palabra de información en lugar de un solo bit. El esquema de un registro construido a partir de un conjunto flip-flops D [8] se presenta en la figura 4.16

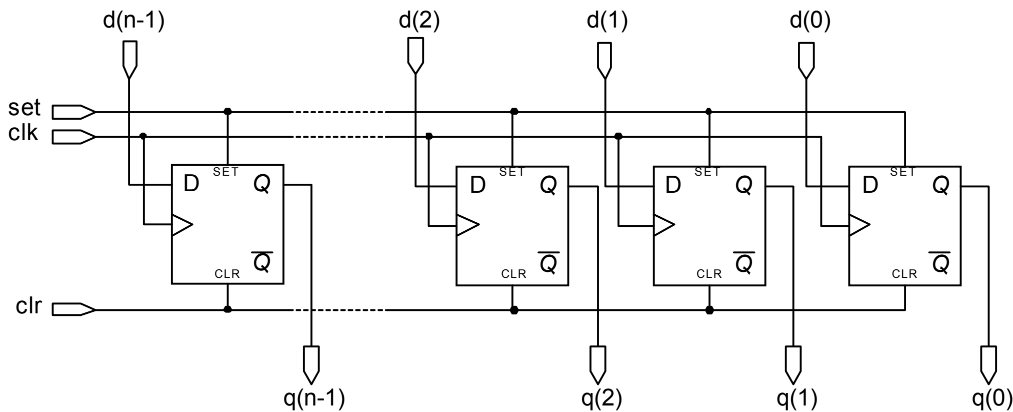


Figura 4.16

En el código VHDL se puede ver como se genera el arreglo de flip-flops para dar lugar al registro. El ancho del registro se determina con el ancho de palabra con que se instancia el componente. El estilo de descripción utilizado es estructural.

```
entity Reg is

      port(
            clk   :in std_logic;
            set   :in std_logic;
            clr   :in std_logic;
            d     :in std_logic_vector;

            q     :out std_logic_vector
      );

end Reg;
```

```
architecture RegStructure of Reg is

      component DFF is
            port(
                  set:in std_logic;
                  clr   :in std_logic;
                  clk   :in std_logic;
                  d     :in std_logic;

                  q     :out std_logic;
                  qBar  :out std_logic
            );
      end component;

      signal qBar:std_logic_vector(d'range);

begin
      DFF_REG:
      for i in d'range generate

            F: DFF port map(set,clr,clk,d(i),q(i),qBar(i));

      end generate;

end RegStructure;
```

4.3.4.9 El contador de iteraciones

Se puede construir un contador binario conectando entre sí tantos flip-flops D como el ancho en bits que se desee que tenga el contador [10]. La forma de organizarlos, es conectar la salida `qBar` de un flip-flop con la entrada de reloj del siguiente y su entrada `d`. Este tipo de contadores se denominan *Contadores Asincrónicos* [15]. En un contador asincrónico el reloj externo está conectado únicamente a la entrada de reloj del primer flip-flop. Por lo tanto el primer flip-flop cambia su estado en el flanco descendente de cada pulso de reloj. Sin embargo el siguiente flip-flop cambia su estado cuando lo hace el primero. Debido al retardo de propagación a través de un flip-flop, la transición del pulso de reloj y de la salida de los flip-flops nunca ocurre en el mismo instante. Por ello los flip-flops no cambian su estado al mismo tiempo, causando una operación asincrónica. El esquema de un contador como el recientemente descripto se ve en la figura 4.17

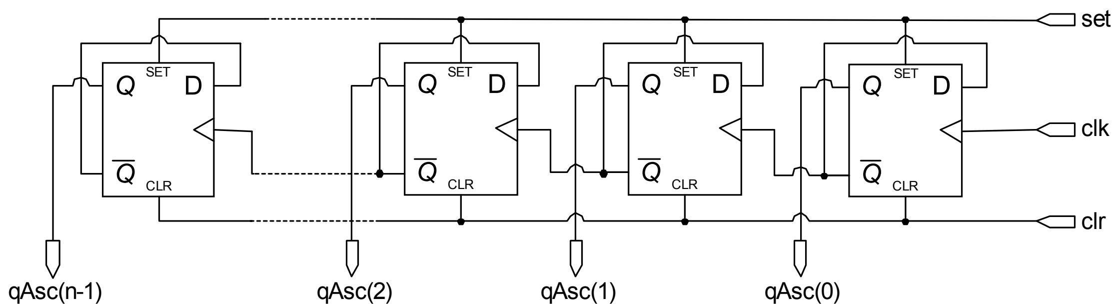


Figura 4.17

El propósito del contador dentro de la arquitectura es el de generar, en cada etapa, el número binario que determina cuanto se deben desplazar las componentes X e Y y proporcionar la dirección de memoria a la tabla de arcotangentes para actualizar el acumulador angular.

El código en VHDL de este contador se expone a continuación. El estilo de descripción utilizado en la arquitectura es estructural. Las señales de `set` y `clr` permiten que el contador pueda inicializarse poniendo todos los flip-flops en cero o uno. La cuenta se incrementa con un flanco ascendente en la señal de reloj, `clk`. El valor actual del contador puede ser leído en el vector de salida `qAsc`.

```
entity Counter is

      port(
            set   :in std_logic;
            clr   :in std_logic;
            clk   :in std_logic;
            qAsc  :out std_logic_vector
      );

end Counter;

architecture CounterStructure of Counter is

      component DFF is
            port(
                  set   :in std_logic;
                  clr   :in std_logic;
                  clk   :in std_logic;
                  d     :in std_logic;

                  q     :out std_logic;
                  qBar  :out std_logic
            );
      end component;

      signal qBarInt:std_logic_vector(qAsc'range);

begin
      DFF_COUNTER:
      for i in qAsc'range generate
            D0:
            if i=0 generate
                  D: DFF port map(set,clr,clk,qBarInt(i),qAsc(i),qBarInt(i));
            end generate;

            DN:
            if i/=0 generate
                  D: DFF port map(set,clr,
                              qBarInt(i-1),qBarInt(i),qAsc(i),qBarInt(i));
            end generate;
      end generate;

end CounterStructure;
```

4.3.4.10 La unidad de control

Un circuito secuencial está compuesto por elementos de lógica combinatoria y elementos de control. El lugar que ocupan los elementos de control en un circuito secuencial se ve claramente en la figura 4.18

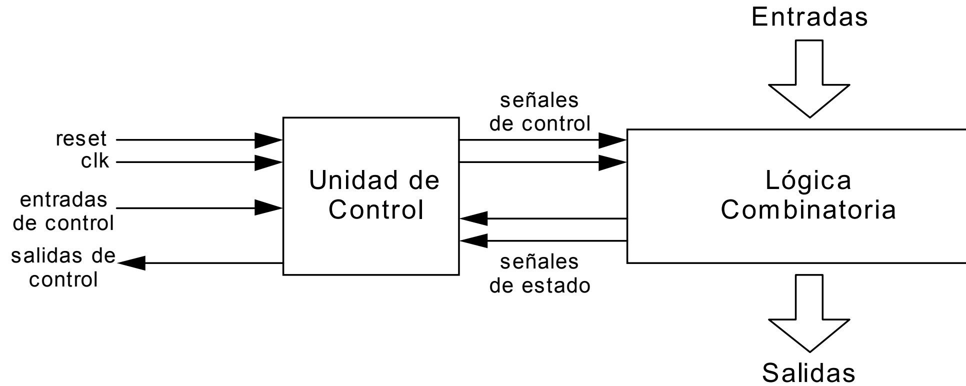


Figura 4.18

Los elementos combinatorios requieren de un control para que el movimiento de datos ocurra en el sentido y tiempo adecuados.

Los elementos de control pueden describirse como una máquina de estados finitos. La máquina de estados tiene una entrada de reloj para realizar el secuenciamiento y una señal de reset para su inicialización. Típicamente consiste de una interfaz con señales de control de entrada de los restantes elementos del sistema y genera señales de control como respuesta. Es decir, provee las señales de control necesarias y monitorea el estado de ejecución del sistema por posibles modificaciones en la secuencia de control.

Una máquina de estados se construye a partir de un algoritmo hardware que consiste en una cantidad finita de pasos que determinan cual será la próxima operación a realizar sobre los datos [16].

Un diagrama de flujo es una manera adecuada de especificar la secuencia de pasos y rutas de decisión de un algoritmo. El diagrama de flujo de un algoritmo de hardware debe tener características especiales que lo vinculen al hardware que implementa el algoritmo. Por lo tanto se emplea un diagrama de flujo especial, denominado diagrama de Máquina de Estado Algorítmico (Algorithmic State Machine, ASM) [12] [16] para definir algoritmos de hardware digital.

La gráfica ASM se parece a un diagrama de flujo convencional, pero con la diferencia de que nombra explícitamente los distintos estados y las señales de control afectadas.

Básicamente contiene tres elementos: *la caja de estados, la caja de decisiones* y *la caja de salida condicional*, como se muestra en la figura 4.19.

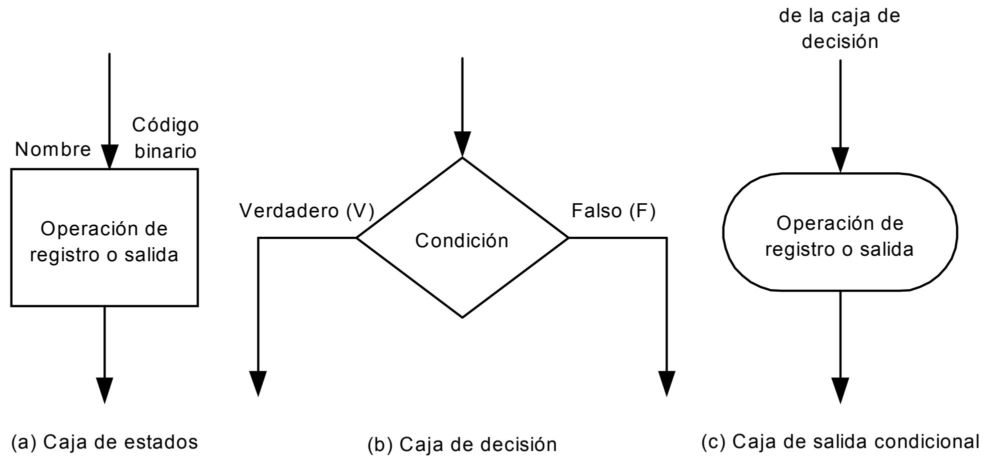


Figura 4.19

La *caja de estados* (figura 4.19 (a)) define un estado en la secuencia de control. En la esquina superior izquierda se coloca el nombre del estado y en la derecha el código binario correspondiente a ese estado. Las señales de control definidas en su interior son incondicionales y se mantienen activas mientras la máquina se encuentra en ese estado, es decir en un ciclo de reloj.

La *caja de decisión* (figura 4.19 (b)) está asociada siempre con una caja de estados y define que señal de entrada debe ser testeada durante el estado actual. De acuerdo al resultado de la condición, que puede ser verdadero o falso se toma un camino de ejecución u otro. Si surgiese la necesidad de testear mas de una señal, las cajas de decisión se pueden disponer en cascada.

La *caja de salida condicional* (figura 4.19 (c)) define una o mas señales de control que se deben activar en el estado actual, solamente cuando una señal de entrada asociada posee un valor determinado. Es por eso que este tipo de caja está siempre asociada a una caja de decisión. La señal de control se activará asociada a la señal de entrada y no estará activa necesariamente en toda la duración del estado actual.

4.3.4.11 Diseño y descripción de la unidad de control

Como primera etapa en el diseño de la unidad de control para la versión iterativa del algoritmo CORDIC, se llevó a cabo la especificación global del sistema. Dicha especificación se basó en el funcionamiento del algoritmo y en los elementos que constituyen el mismo. Una vez comprendido el funcionamiento, se realizo un esquema con las señales de control que conectan a la unidad de control con el resto del sistema.

La figura 4.20 representa un diagrama en bloques del sistema, mostrando la unidad de control (UC) y las señales de control y estado. Como se ve en el diagrama, el algoritmo recibe las tres componentes `Xo`, `Yo` y `Zo` junto con el modo de operación. Las otras dos señales de entrada las constituyen la señal de `reset` y el reloj (`clk`). La señal de `reset` se utiliza para poner a la máquina de estados en su estado inicial, y la señal de reloj proporciona el sincronismo. Las señales de salida denominadas `Xn`, `Yn` y `Zn`, contendrán los resultados del algoritmo. La señal `done` determina el fin del cálculo y la validez del resultado obtenido en las señales de salida.

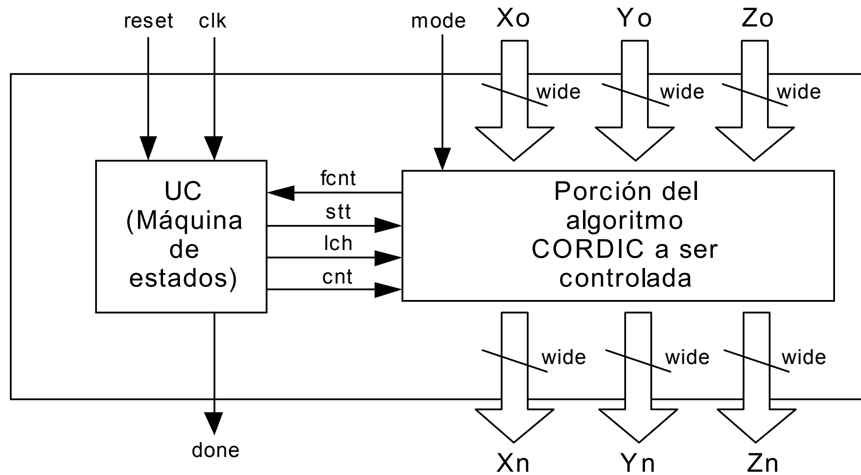


Figura 4.20

El identificador `wide` que aparece junto a las componentes es una constante arbitaria que expresa el ancho en bits de cada una.

Las cuatro señales de control se detallan en la tabla 4.4

Denominación	Nombre Real	Sentido	Propósito
fcnt	Finished count	Entrada	Se activa cuando el número de iteraciones llegó al máximo establecido, indicando el fin del cálculo.
stt	Start	Salida	Se activa cuando la máquina de estados comienza a operar luego de un reset. Abre el paso de los multiplexores para el ingreso de nuevos valores.
lch	Latch	Salida	Se activa cuando el valor calculado en una iteración debe ser almacenado en los registros intermedios para su uso en la etapa siguiente.
cnt	Count	Salida	Se activa cuando el contador que determina el número de iteración actual debe ser incrementado en uno.

Tabla 4.4

Como segunda etapa en el diseño se construyó el diagrama ASM junto con la asignación de los diversos estados y su codificación en binario, como se muestra en la figura 4.21. De esta manera

se logró describir el funcionamiento de la unidad de control y sincronizar la activación de las diversas señales.

Los diversos estados por los cuales atraviesa la ejecución de la unidad de control, se dedujeron a partir del diagrama y se introducen en la tabla 4.5. La codificación binaria de estados se realizó siguiendo el patrón de códigos Gray como muestra la tabla 4.6. La codificación Gray de estados determina que dos estados adyacentes sean etiquetados con códigos binarios que difieran únicamente en un bit.

Estado actual	Estado siguiente
InitS	LatchS
LatchS	ShiftAddS
ShiftAddS	LatchS
ShiftAddS	EndS
EndS	EndS

Tabla 4.5

Estado	Código binario
InitS	01
LatchS	00
ShiftAddS	10
EndS	11

Tabla 4.6

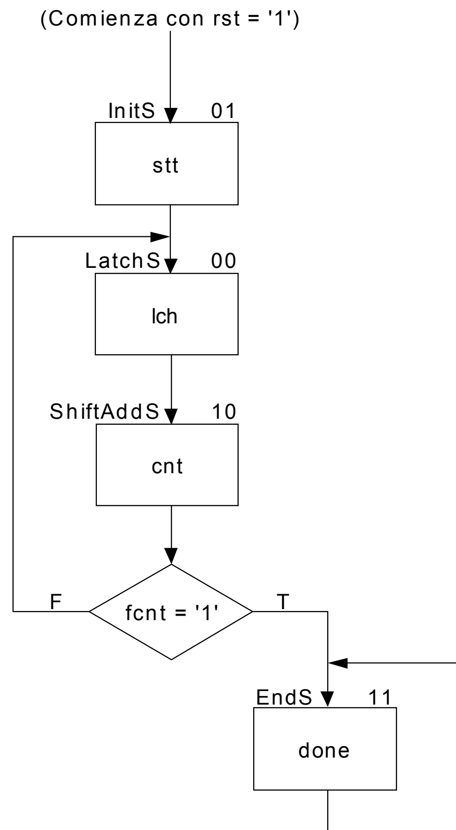


Figura 4.21

A partir del diagrama ASM, se puede llevar a cabo una descripción funcional de la unidad de control en VHDL. En la descripción funcional, el conjunto de estados se declara como un tipo de datos enumerativo y no se considera por ahora la codificación en binario de los mismos.

```
entity FSM is

      port(
            clk:in std_logic;
            rst:in std_logic;
            fcnt:in std_logic;

            stt:out std_logic;
            lch:out std_logic;
            done:out std_logic;
            cnt:out std_logic
      );

end FSM;

architecture BehaviorFSM of FSM is

      type States is (InitS,LatchS,ShiftAddS,EndS);

      signal state:States:=EndS;
```

```
begin

	stt<='1' when state=InitS else '0';
	lch<='1' when state=LatchS else '0';
	cnt<='1' when state=ShiftAddS else '0';
	done<='1' when state=EndS else '0';

StateMachine:
	process(clk)
	begin
		if rst='1' then

			state<=InitS;

		elsif clk'event and clk='0' then

			case state is

				when InitS=>

					state<=LatchS;

				when LatchS=>

					state<=ShiftAddS;

				when ShiftAddS=>

					if fcnt='1' then

						state<=EndS;

					elsif fcnt='0' then

						state<=LatchS;

					end if;

				when EndS=>

					state<=EndS;

			end case;

		end if;

	end process;

end BehaviorFSM;
```

En la declaración de entidad se especifican las señales de entrada y salida como se las introdujo al principio, durante la fase de especificación.

Las señales de salida se activan en los estados adecuados y el funcionamiento de la máquina de estados comienza con un reset del sistema. La operación de reset ubica a la máquina en el estado inicial denominado `InitS`. En cada flanco descendente del reloj se pasa de un estado al siguiente.

La descripción anterior es funcional y aborda ambos estilos de descripción, el algorítmico y el de flujo de datos. También se puede hacer una descripción a nivel de compuertas. En una

descripción a nivel de compuertas lógicas los elementos de almacenamiento están caracterizados por flip-flops, que almacenan los diversos estados. De ahí la importancia de asignar una codificación binaria a cada estado.

Para realizar una descripción a nivel de compuertas, se necesitan ecuaciones booleanas [Anexo B] para los estados siguientes y para cada una de las señales de control. Como la unidad de control cuenta con cuatro estados, se utilizaron dos flip-flops para almacenar cada dígito binario de un estado determinado.

Se llamó a cada flip-flop, A y B respectivamente, y a cada próximo estado DA y DB. Con la información de funcionamiento de la unidad de control y las señales de control se confeccionó una tabla que se muestra en la tabla 4.7, denominada *tabla de transiciones* [14], ya que de la misma se determinan las transiciones entre los estados y el valor de las señales.

Señales de entrada	Estado actual		Siguiente estado		Señales de salida			
fcnt	A	B	DA	DB	stt	lch	cnt	done
x	0	1	0	0	1	0	0	0
x	0	0	1	0	0	1	0	0
x	1	0	0	0	0	0	1	0
1	1	0	1	1	0	0	0	0
x	1	1	1	1	0	0	0	1

Tabla 4.7

Una vez armada la tabla, se procedió a obtener las funciones booleanas para las señales de próximo estado y las señales de salida a partir de las señales de entrada y estado actual.

$$DA = \overline{A} \cdot \overline{B} + fcnt \cdot A \cdot \overline{B} + A \cdot B = \overline{A} \cdot \overline{B} + A \cdot (fcnt \cdot \overline{B} + B)$$

$$DB = fcnt \cdot A \cdot \overline{B} + A \cdot B = A \cdot (fcnt \cdot \overline{B} + B)$$

$$stt = \overline{A} \cdot B$$

$$lch = \overline{A} \cdot \overline{B}$$

$$cnt = A \cdot \overline{B}$$

$$done = A \cdot B$$

Dadas las funciones anteriores, el esquema resultante correspondiente a la unidad de control a nivel de compuertas se muestra en la figura 4.22.

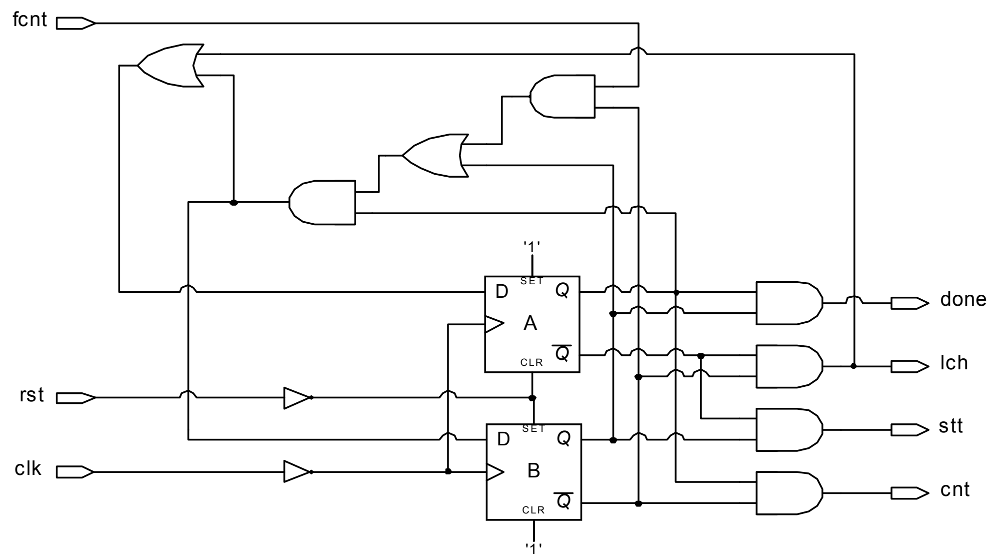


Figura 4.22

Una descripción en VHDL correspondiente a la figura anterior es la que se muestra a continuación

```
architecture FSMStructure of FSM is

      component DFF is

            port(

                  set         :in std_logic;
                  clr         :in std_logic;
                  clk         :in std_logic;
                  d           :in std_logic;

                  q           :out std_logic;
                  qBar        :out std_logic
            );

      end component;

      signal a,da,na,b,db,nb:std_logic;
      signal nclk,nrst:std_logic;
      signal nab,dba:std_logic;

begin

      stt<=na and b;
      lch<=na and nb;
      cnt<=a and nb;
      done<=a and b;

      da<=dba or nab;
      db<=dba;

      nclk<=not clk;
```

```
        nrst<=not rst;

        DFFA: DFF port map('1',nrst,nclk,da,a,na);
        DFFB: DFF port map(nrst,'1',nclk,db,b,nb);

        nab<=na and nb;
        dba<=a and (b or (fcnt and nb));

end FSMStructure;
```

La entidad correspondiente a la arquitectura a nivel de compuertas es la misma que para la arquitectura funcional.

4.3.4.12 Descripción final de la arquitectura bit-paralela iterativa

Habiendo terminado con la descripción de los componentes propios de la arquitectura iterativa, se procedió a conectarlos para dar lugar al diseño final.

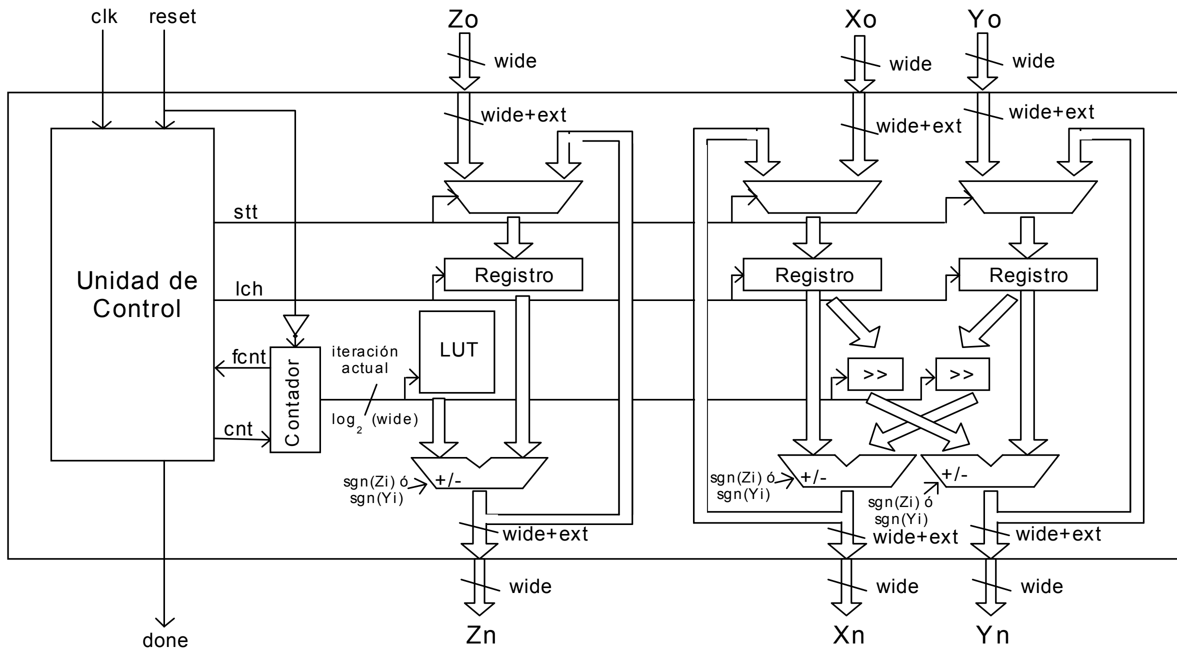


Figura 4.23

La figura 4.23 muestra el diseño final completo. Luego de la activación de la señal de `reset`, la unidad de control habilita la señal `stt` que activa el ingreso de los datos de los componentes *X*, *Y* y *Z* al sistema. Durante el siguiente estado, la unidad de control activa la señal `lch`, que habilita los registros para almacenar los valores de las componentes. La precencia de registros es necesaria debido a la naturaleza combinatoria de las unidades de desplazamiento y los sumadores algebraicos. De lo contrario la realimentación sin sincronismo produciría un resultado inadecuado. Una vez que los valores son almacenados en los registros la unidad de control activa la señal `cnt`, que produce un incremento en el contador de iteración. El número binario correspondiente a la iteración se utiliza por la tabla de arcotangentes (memoria ROM) para codificar

la dirección de la constante a sumar al acumulador angular y para indicar a las unidades de desplazamiento la cantidad de lugares que deben desplazar los operandos. Seguidamente la unidad de control habilita la señal lch y se repite el ciclo. El cálculo se detiene cuando la unidad de control recibe la activación de la señal `fcnt`, que se habilita cuando la cuenta alcanza el número de iteraciones preestablecido. En ese momento la unidad de control activa la señal `done` y finaliza el cálculo.

La señal `mode` no se incluyó para simplificar el diagrama. Dependiendo de su valor, el procesador CORDIC opera en modo rotación o vectorización. Las etiquetas `wide` y `wide+ext` indican el ancho de los buses externo e interno respectivamente.

Seguidamente se transcribe el código en VHDL de la arquitectura CORDIC bit-paralela iterativa. El estilo de descripción utilizado es estructural. Se ha obviado la declaración de componentes en la parte declarativa de la arquitectura.

La entidad `ParalellIterativeCORDIC` tiene seis señales de entrada. Tres de las señales corresponden a los componentes *X*, *Y* y *Z*. Las señales de entrada restantes corresponden a la señal de reloj denominada `clk`, a la señal de reset, denominada `rst` y al modo de operación (`mode`). A diferencia de la arquitectura desplegada, ésta arquitectura cuenta con una señal de reloj, debido a que se trata de un circuito secuencial.

Las señales de salida, corresponden a las salidas de los componentes *X*, *Y* y *Z* y a la señal `done`, que se activa una vez que terminó el cálculo.

```
entity ParalellIterativeCORDIC is

      generic(iterations,wide,ext:natural;arctanLUT:STDLVectorW);

      port(
            clk   :in std_logic;
            rst   :in std_logic;
            mode  :in std_logic;
            Xo    :in std_logic_vector(wide-1 downto 0);
            Yo    :in std_logic_vector(wide-1 downto 0);
            Zo    :in std_logic_vector(wide-1 downto 0);

            Xn    :out std_logic_vector(wide-1 downto 0);
            Yn    :out std_logic_vector(wide-1 downto 0);
            Zn    :out std_logic_vector(wide-1 downto 0);
            done  :out std_logic
      );

end ParalellIterativeCORDIC;

architecture ParalellIterativeCORDICStructure of ParalellIterativeCORDIC is

      use work.typeconversion.all;
      use std.textio.all;

      constant stages:natural:=integer(ceil(log2(real(wide))));

      signal Xin,Yin,Zin:std_logic_vector(wide+ext-1 downto 0);
      signal Xtemp,Xtemp0,Xtemp1:std_logic_vector(wide+ext-1 downto 0);
      signal Ytemp,Ytemp0,Ytemp1:std_logic_vector(wide+ext-1 downto 0);
      signal Ztemp,Ztemp0,Ztemp1:std_logic_vector(wide+ext-1 downto 0);
      signal Xshifted,Yshifted,wconst:std_logic_vector(wide+ext-1 downto 0);
```

```
	signal const:std_logic_vector(wide-1 downto 0);
	signal stt,lch,fcnt,cnt,nrst,cOutX,cOutY,cOutZ,doneInt:std_logic;
	signal S0,S1,Si,Sj:std_logic;
	signal currIter,endIter:std_logic_vector(stages-1 downto 0);
	signal cantIter:std_logic_vector(stages-1 downto 0)
	                                          :=convert(iterations-1,stages);
begin

	StageController: FSM port map(clk,rst,fcnt,stt,lch,doneInt,cnt);

	cantIter<=cantIter;
	endIter<=not (currIter xor cantIter);
	MAND: MultiAND port map(endIter,fcnt);

	nrst<=not rst;

	SCNT: Counter port map(nrst,'1',cnt,currIter);

	Xin(wide+ext-1 downto ext)<=Xo;
	Xin(ext-1 downto 0)<=(others=>'0');

	Yin(wide+ext-1 downto ext)<=Yo;
	Yin(ext-1 downto 0)<=(others=>'0');

	Zin(wide+ext-1 downto ext)<=Zo;
	Zin(ext-1 downto 0)<=(others=>'0');

	MUXX: WideMux port map(Xtemp,Xin,stt,Xtemp0);
	MUXY: WideMux port map(Ytemp,Yin,stt,Ytemp0);
	MUXZ: WideMux port map(Ztemp,Zin,stt,Ztemp0);

	REGX: Reg port map(lch,'1','1',Xtemp0,Xtemp1);
	REGY: Reg port map(lch,'1','1',Ytemp0,Ytemp1);
	REGZ: Reg port map(lch,'1','1',Ztemp0,Ztemp1);

	SHX: BarrelRightShifter generic map(stages,wide+ext)
	                              port map(Xtemp1,currIter,Xshifted);
	SHY: BarrelRightShifter generic map(stages,wide+ext)
	                              port map(Ytemp1,currIter,Yshifted);

	ADDX: Adder port map(Xtemp1,Yshifted,S0,S0,Xtemp,cOutX);
	ADDY: Adder port map(Ytemp1,Xshifted,S1,S1,Ytemp,cOutY);
	ADDZ: Adder port map(Ztemp1,wconst,S0,S0,Ztemp,cOutZ);

	wconst(wide-1 downto 0)<=const;
	wconst(wide+ext-1 downto wide)<=(others=>'0');

	LUT: ROM generic map (arctanLUT) port map(currIter,const);

	Si<=Ztemp1(wide+ext-1);
	Sj<=not Ytemp1(wide+ext-1);
	S0<=not S1;
	SMUX: Mux port map(Sj,Si,mode,S1);

	Xn<=Xtemp(wide+ext-1 downto ext);
	Yn<=Ytemp(wide+ext-1 downto ext);
	Zn<=Ztemp(wide+ext-1 downto ext);

	done<=doneInt;

end ParalellIterativeCORDICStructure;
```

4.3.5 El banco de pruebas para las descripciones particulares

Para validar el funcionamiento de las descripciones particulares, se describió un banco de pruebas utilizando la metodología algorítmica para generar los vectores de prueba.

En esencia, los bancos de prueba para las versiones desplegada e iterativa del algoritmo son similares. La diferencia fundamental entre ambos es la inclusión de una señal de reloj en el caso iterativo. Esto fue necesario debido a la naturaleza secuencial del diseño. Para el diseño desplegado, la existencia de un reloj no fue necesaria debido a la naturaleza combinatoria del mismo.

El reloj se describió como una señal llamada `clk` de tipo `std_logic` que alterna su valor binario entre 0 y 1 cada unidad de tiempo. Se tomó como unidad de tiempo básico el nanosegundo.

```
clk <= not clk after 1 ns;
```

El banco de pruebas consta de un `PROCESS` para generar las pruebas y de las instancias de los componentes correspondientes al algoritmo CORDIC.

La simulación de la arquitectura desplegada de todos modos necesitó la inclusión de tiempo. Para avanzar en la simulación se incluyó dentro del `PROCESS` una sentencia `WAIT` que provoca que el simulador pase al siguiente instante de simulación.

En el anexo D se presenta a modo de ejemplo un banco de prueba para las arquitecturas iterativa y desplegada.

Capítulo 5

Simulación y verificación de las descripciones

Luego de describir un circuito digital, éste debe ser simulado para verificar y validar su comportamiento. Se realizaron simulaciones para verificar el funcionamiento de la descripción funcional algorítmica y de las arquitecturas particulares. Las simulaciones realizadas son de índole funcional, lo que permitió verificar el funcionamiento correcto de cada descripción. Se modificaron dos parámetros de interés: el ancho de palabra y el número de iteraciones. En las simulaciones no se tomaron en cuenta retrasos ni tiempos de los componentes, ya que los retrasos adquieren sentido cuando se tiene en consideración una plataforma específica.

5.1 Herramientas de simulación

Para realizar la simulación de los diseños, se empleó el simulador VeriBest VBVHDL 99.0 que forma parte de la herramienta de desarrollo junto con el compilador de VHDL.

La simulación de un diseño, se lleva a cabo estableciendo la entidad que se encuentra en el tope de la jerarquía del diseño que se desea simular. Esta entidad, denominada Entidad Raíz (Root Entity), usualmente será el banco de pruebas que se utiliza para llevar a cabo las verificaciones.

El código VHDL compilado, puede ser ejecutado y depurado con el uso del debugger. Para realizar una simulación se debe ejecutar el simulador y seguidamente se debe establecer el tiempo de duración de la simulación.

Asimismo el ambiente de trabajo incluye un Visor de Ondas (Waveform Viewer) cuyo aspecto se muestra en la figura 5.1. Este visor permite monitorear los valores de las señales luego de la simulación. Se pueden abrir tantas ventanas como se desee para monitorear diversas partes de un diseño. Las señales se pueden incluir de a una o por componentes (figura 5.2).

Los valores de las señales se pueden analizar una vez que la simulación ha terminado, al haberse alcanzado el tiempo fijado previamente. El simulador almacena los valores de la simulación completa en archivos temporales en el disco rígido. Este último aspecto puede resultar problemático para simulaciones de larga duración y diseños que poseen una gran cantidad de señales.

El visor de ondas únicamente tiene sentido para simulaciones en donde el diseño emplea señales. Los diseños algorítmicos hacen uso de variables que son elementos del lenguaje de alto nivel y no pueden ser visualizadas. Sin embargo para analizar diseños funcionales secuenciales el simulador cuenta con un debugger que es idéntico al de cualquier lenguaje de programación

convencional. Este debugger permite ubicar puntos de parada (breakpoints) y monitorear el contenido de variables.

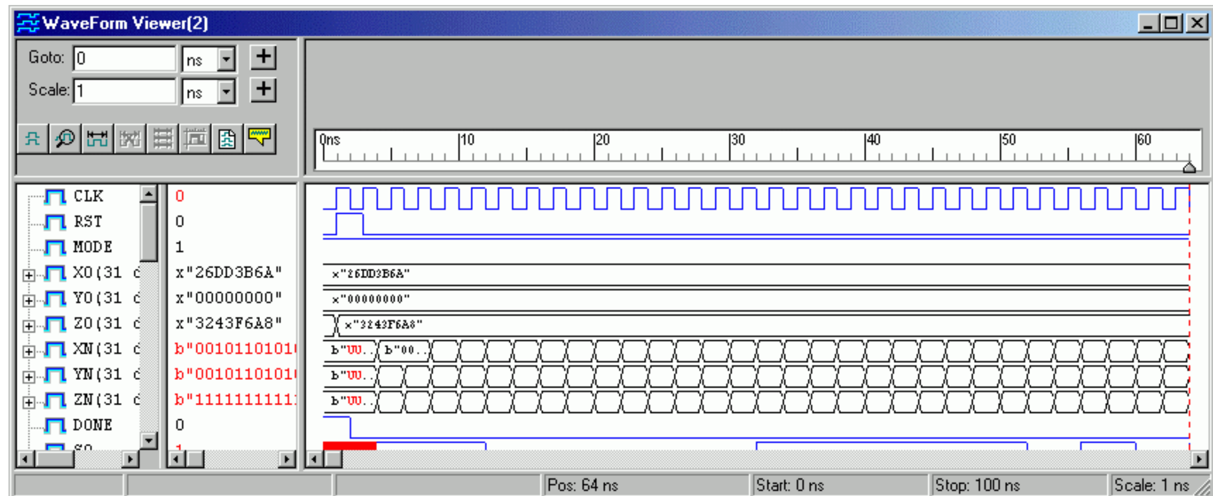


Figura 5.1

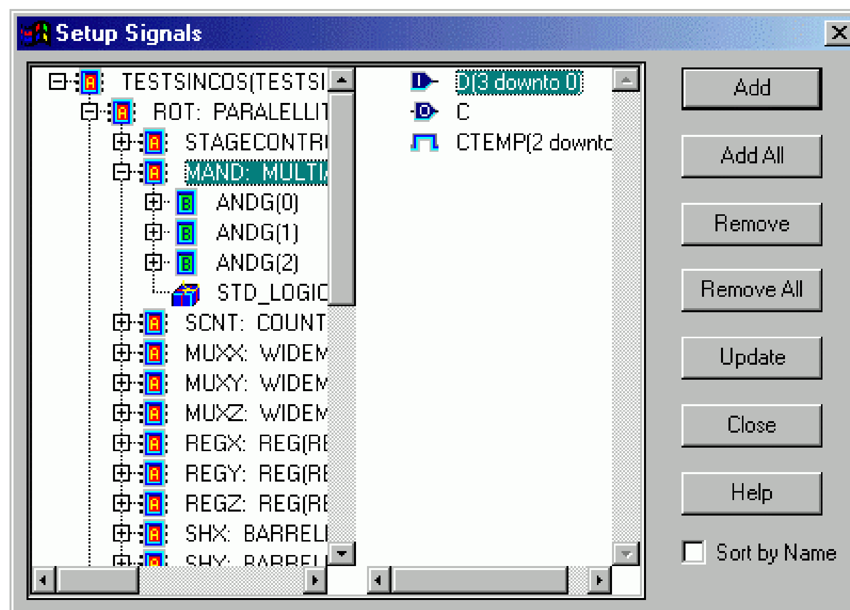


Figura 5.2

Las simulaciones de la descripción funcional algorítmica realizaron sobre una computadora basada en un procesador Intel Pentium de 200Mhz con 32Mb de memoria RAM y 2Gb de espacio libre en disco rígido. Para simular las descripciones de las arquitecturas particulares, se recurrió a una computadora basada en un procesador Intel Pentium III de 800Mhz con 128Mb de memoria RAM y 35Gb de espacio libre en disco rígido. Se optó por cambiar de sistema debido a la capacidad de memoria y procesamiento que requería el simulador para funcionar adecuadamente.

5.2 Valores para la simulación

La simulación del seno y del coseno se realizó para los ángulos θ entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$, rango de convergencia del algoritmo CORDIC, y para los ángulos $-\frac{\pi}{2},-\frac{\pi}{3},-\frac{\pi}{4},-\frac{\pi}{6},0,\frac{\pi}{6},\frac{\pi}{4},\frac{\pi}{3},\frac{\pi}{2}$.

Para el arcotangente, el rango para las descripciones que utilizan aritmética de punto fijo, se limitó por un lado a los valores de la tangente entre $\mathrm{tg}\left(-\frac{4\pi}{9}\right)$ y $\mathrm{tg}\left(\frac{4\pi}{9}\right)$. Por otro lado se realizó la simulación en todo el rango de convergencia del algoritmo, como se explicó en la sección 4.3.1. La función recíproca $f(x)=\frac{1}{x}$ no se describió en VHDL, los valores adecuados fueron suministrados por el banco de pruebas y la biblioteca matemática del lenguaje.

El incremento de iteración que se escogió para el rango es de 1' (un minuto angular) que expresado en radianes es $\frac{\pi}{10800}$.

La simulaciones de la descripción algorítmica no llevaron gran cantidad de tiempo, sin embargo la simulación de cada arquitectura particular demandó en promedio entre 3 y 4 horas. El tamaño de los archivos temporales para las arquitecturas, almacenados durante la simulación, fue de aproximadamente 20Gb.

No se describió un banco de pruebas para suministrar todas las posibles entradas de acuerdo al ancho de palabra, debido al tiempo que demandaría el cálculo. Para las entradas con ancho de palabra de 16 bits y 16 iteraciones, por ejemplo, el cálculo de 10800 valores demandó aproximadamente 3 horas.

Los valores para las tres descripciones se compararon con los calculados por MATLAB™ [Anexo C]. La comparación se realizó calculando el error absoluto *E* que se obtuvo entre ambos métodos. Para calcular el error se utilizó la siguiente fórmula:

$$E(x)=\left|V_{CORDIC}(x)-V_{MATLAB}(x)\right|$$

El error absoluto puede expresarse en bits y en la práctica puede utilizarse para decidir cuantos bits se consideran exactos para la representación de los resultados. Se emplea la siguiente fórmula:

$$D_2(x)=\lceil -\log_2 E(x)\rceil -1$$

En ambos casos, *V(x)* representa el valor calculado por el algoritmo.

Las simulaciones se llevaron a cabo modificando el número de iteraciones entre 4 y un máximo, que para las arquitecturas particulares está determinado por el ancho de palabra y para la descripción funcional algorítmica se estableció en 32. Para cada número de iteraciones se realizaron las simulaciones correspondientes dentro del rango de convergencia. El número de iteraciones se incrementó de a dos.

5.3 Simulación de la descripción funcional algorítmica

La simulación de la descripción funcional algorítmica se llevó a cabo para validar el funcionamiento del algoritmo CORDIC. Los resultados se compararon con los valores que se obtuvieron al evaluar dichas funciones con MATLAB™. La descripción funcional opera con el tipo `REAL` de VHDL que es una representación en punto flotante basada en el estándar IEEE 754 de doble precisión.

A modo de ejemplo, se siguen los pasos que efectúa el algoritmo para calcular el seno y el coseno de $\frac{\pi}{4} \cong 0{,}78539$ y se exponen los valores que adquieren los distintos parámetros del mismo. El número de dígitos fraccionarios se restringió a cinco ya que el objetivo del ejemplo es únicamente el de ilustrar el funcionamiento del algoritmo. Al final del cálculo, los valores del coseno y del seno, se obtienen de las variables `X` e `Y` respectivamente.

i	X	Y	Z	2^{-i}	$\mathrm{arctg}(2^{-i})$	signo(Z)
0	0.60725	0.00000	0.78539	1.00000	0.78539	+
1	0.60725	0.60725	-0.00000	0.50000	0.46364	-
2	0.91087	0.30362	0.46364	0.25000	0.24497	+
3	0.83497	0.53134	0.21866	0.12500	0.12435	+
4	0.76855	0.63571	0.09431	0.06250	0.06241	+
5	0.72882	0.68375	0.03189	0.03125	0.03123	+
6	0.70745	0.70652	0.00065	0.01562	0.01562	+
7	0.69641	0.71758	-0.01496	0.00781	0.00781	-
8	0.70202	0.71214	-0.00715	0.00390	0.00390	-
9	0.70480	0.70939	-0.00324	0.00195	0.00195	-
10	0.70618	0.70802	-0.00129	0.00097	0.00097	-
11	0.70688	0.70733	-0.00032	0.00048	0.00048	-
12	0.70722	0.70698	0.00016	0.00024	0.00024	+
13	0.70705	0.70716	-0.00007	0.00012	0.00012	-
14	0.70713	0.70707	0.00004	0.00006	0.00006	+
15	0.70709	0.70711	-0.00001	0.00003	0.00003	-
16	**0.70711**	**0.70709**	0.00001	0.00001	0.00001	+

A continuación se muestran los resultados de la simulación para los valores $-\frac{\pi}{2}, -\frac{\pi}{3}, -\frac{\pi}{4}, -\frac{\pi}{6}, 0, \frac{\pi}{6}, \frac{\pi}{4}, \frac{\pi}{3}, \frac{\pi}{2}$ con 32 iteraciones en las tablas 5.1, 5.2 y 5.3 respectivamente. En las tablas también se muestra el valor obtenido por MATLAB™ a modo de referencia.

Angulo	Seno MATLAB	Seno CORDIC
$-\frac{\pi}{2}$	-1	-1
$-\frac{\pi}{3}$	-8.660254037844386e-001	-8.660254038110794e-001
$-\frac{\pi}{4}$	-7.071067811865475e-001	-7.071067811182992e-001
$-\frac{\pi}{6}$	-4.999999999999999e-001	-4.999999999538575e-001
0	0	1.295764953995623e-010
$\frac{\pi}{6}$	4.999999999999999e-001	4.999999999538575e-001
$\frac{\pi}{4}$	7.071067811865475e-001	7.071067812547968e-001
$\frac{\pi}{3}$	8.660254037844386e-001	8.660254038110794e-001
$\frac{\pi}{2}$	1	1

Tabla 5.1

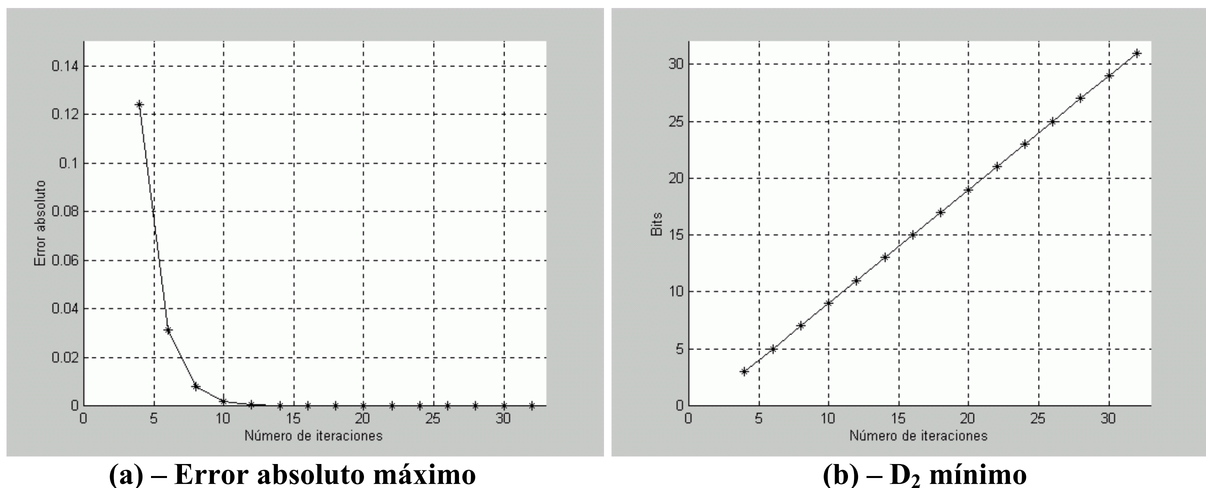
Angulo	Coseno MATLAB	Coseno CORDIC
$-\frac{\pi}{2}$	0	1.295764953995623e-010
$-\frac{\pi}{3}$	5.000000000000001e-001	4.999999999538575e-001
$-\frac{\pi}{4}$	7.071067811865476e-001	7.071067812547968e-001
$-\frac{\pi}{6}$	8.660254037844387e-001	8.660254038110794e-001
0	1	1
$\frac{\pi}{6}$	8.660254037844387e-001	8.660254038110794e-001
$\frac{\pi}{4}$	7.071067811865476e-001	7.071067811182992e-001
$\frac{\pi}{3}$	5.000000000000001e-001	4.999999999538575e-001
$\frac{\pi}{2}$	0	1.295764953995623e-010

Tabla 5.2

Tangente	Arcotangente MATLAB	Arcotangente CORDIC
$\mathrm{tg}\left(-\frac{\pi}{2}\right)$	-1.570796326794896	-1.570796326665320
$\mathrm{tg}\left(-\frac{\pi}{3}\right)$	-1.047197551196598	-1.047197551249879
$\mathrm{tg}\left(-\frac{\pi}{4}\right)$	-7.853981633974483e-001	-7.853981633009298e-001
$\mathrm{tg}\left(-\frac{\pi}{6}\right)$	-5.235987755982988e-001	-5.235987755450179e-001
$\mathrm{tg}(0)$	0	1.295765189457968e-010
$\mathrm{tg}\left(\frac{\pi}{6}\right)$	5.235987755982988e-001	5.235987755450179e-001
$\mathrm{tg}\left(\frac{\pi}{4}\right)$	7.853981633974483e-001	7.853981633009298e-001
$\mathrm{tg}\left(\frac{\pi}{3}\right)$	1.047197551196598	1.047197551249879
$\mathrm{tg}\left(\frac{\pi}{2}\right)$	1.570796326794896	1.570796326665320

Tabla 5.3

Se calcularon los valores del seno, del coseno y del arcotangente en el rango de convergencia del algoritmo, modificando el número de iteraciones entre 4 y 32. A partir de los valores obtenidos mediante las simulaciones y MATLAB™, se calculó el error absoluto máximo. Con dicho error se obtuvo la mínima cantidad de bits considerados exactos, según la fórmula D_2. Los valores obtenidos se muestran en las tablas 5.4 y 5.5, y se grafican en la figura 5.3. La figura 5.3 a) corresponde a la gráfica del error absoluto máximo y la figura 5.3 b) a la cantidad mínima de bits válidos para cada iteración. Los gráficos del error para las tres funciones son similares por lo que se expone solo uno, sin embargo existen diferencias que pueden observarse en la tabla 5.4. En el caso de la cantidad mínima de dígitos binarios considerados exactos los valores son idénticos para las tres funciones simuladas.



(a) – Error absoluto máximo **(b) – D_2 mínimo**

Figura 5.3

Iteraciones	Error absoluto máximo		
	Seno	Coseno	Arcotangente
4	1.239941434116256e-001	1.239941434117357e-001	1.243323987289236e-001
6	3.123814548303316e-002	3.123814548316328e-002	3.123942535351132e-002
8	7.802869457692359e-003	7.802869457822481e-003	7.803355672933332e-003
10	1.943815911317360e-003	1.943815911447482e-003	1.950192443141596e-003
12	4.801867943289434e-004	4.801867944251720e-004	4.865316636393224e-004
14	1.217712861756731e-004	1.217712860590511e-004	1.220351082866778e-004
16	3.049289234331211e-005	3.049289221112618e-005	3.049398127026766e-005
18	7.605526813447838e-006	7.605526681261909e-006	7.627125881315422e-006
20	1.897498003292086e-006	1.897498132005099e-006	1.906345085678218e-006
22	4.737062145668269e-007	4.737060940937510e-007	4.767281169826276e-007
24	1.185509823607711e-007	1.185508512833306e-007	1.191542309264548e-007
26	2.968034067130088e-008	2.968021136154352e-008	2.977507684853720e-008
28	7.444936066941521e-009	7.445063551769770e-009	7.448033967349943e-009
30	1.858476015442756e-009	1.858603500271006e-009	1.862529797058698e-009
32	4.624036500922024e-010	4.622801516585006e-010	4.656410812486910e-010

Tabla 5.4

Iteraciones	D_2 mínimo
	Bits
4	3
6	5
8	7
10	9
12	11
14	13
16	15
18	17
20	19
22	21
24	23
26	25
28	27
30	29
32	31

Tabla 5.5

Se puede observar en la figura 5.3 a) y en la tabla 5.4 que el error absoluto disminuye a medida que aumenta el número de iteraciones. En la figura 5.3 b) y en la tabla 5.5 se puede ver como se incrementa el mínimo en bits exactos.

5.4 Simulación de las arquitecturas particulares

5.4.1 Cálculo de valores

La simulación de las descripciones que utilizan aritmética de punto fijo se realizó con un ancho de palabra de 16 y 32 bits. Los valores obtenidos para el seno y el coseno tanto en la arquitectura bit-paralela desplegada como en la iterativa fueron iguales, ya que si bien se trata de dos arquitecturas distintas, la funcionalidad es la misma para ambas.

La exactitud de los resultados proporcionados por el algoritmo CORDIC no depende únicamente del ancho de palabra de las componentes X, Y y Z, sino también del número de iteraciones que ejecuta el algoritmo. Se observó que el número de iteraciones está limitado por el ancho de palabra, es decir que con un ancho de n bits se obtienen n iteraciones útiles. Este hecho se debe a la multiplicación del factor 2^{-i} por las componentes X e Y, en donde i representa el número de iteración. Esta multiplicación equivale a desplazar las componentes hacia la derecha i veces. Entonces, cuando $i \geq n$, el resultado del desplazamiento para un número positivo valdrá cero. Para la componente Z o acumulador angular ocurre algo similar con el valor de $arctg(2^{-i})$. A modo de ejemplo se muestra la tabla de arcotangentes para un ancho de palabra de 32 bits en VHDL. Cuando $i \geq 32$, $arctg(2^{-i})$ vale cero.

```
constant arctanLUT:STDLVectorW(0 to 31):=(
                        "11001001000011111101101010100010",
                        "01110110101100011001110000010101",
                        "00111110101101101110101111110010",
                        "00011111110101011011101010011011",
                        "00001111111110101010110111011100",
                        "00000111111111110101010101101111",
                        "00000011111111111110101010101011",
                        "00000001111111111111110101010101",
                        "00000000111111111111111110101011",
                        "00000000011111111111111111110101",
                        "00000000001111111111111111111111",
                        "00000000000111111111111111111111",
                        "00000000000011111111111111111111",
                        "00000000000001111111111111111111",
                        "00000000000000111111111111111111",
                        "00000000000000011111111111111111",
                        "00000000000000001111111111111111",
                        "00000000000000000111111111111111",
                        "00000000000000000011111111111111",
                        "00000000000000000001111111111111",
                        "00000000000000000000111111111111",
                        "00000000000000000000011111111111",
                        "00000000000000000000001111111111",
                        "00000000000000000000000111111111",
                        "00000000000000000000000011111111",
                        "00000000000000000000000001111111",
                        "00000000000000000000000000111111",
                        "00000000000000000000000000011111",
                        "00000000000000000000000000001111",
                        "00000000000000000000000000000111",
                        "00000000000000000000000000000011",
                        "00000000000000000000000000000001"
                        );
```

Con el formato numérico explicado en la sección 4.3.1 se utiliza todo el ancho de palabra de la tabla de arcotangentes. Si se hubiesen reservado lugares para la parte entera dentro de la tabla de

búsqueda, se perderían dígitos menos significativos, limitando aún mas el número de iteraciones útiles.

Para las descripciones particulares también se calcularon los valores de los ángulos $-\frac{\pi}{2},-\frac{\pi}{3},-\frac{\pi}{4},-\frac{\pi}{6},0,\frac{\pi}{6},\frac{\pi}{4},\frac{\pi}{3},\frac{\pi}{2}$. Los valores se muestran en las tablas 5.6, 5.7 y 5.8 para 16 y 32 bits efectuando 16 y 32 iteraciones respectivamente. Se los convirtió a punto flotante con MATLAB™ a fin de obtener un resultado legible. Las arcotangentes para $\mathrm{tg}\left(-\frac{\pi}{2}\right)$ y $\mathrm{tg}\left(\frac{\pi}{2}\right)$ se calcularon sustituyendo $y_0 = \pm 1$, $x_0 = 0$ y $z_0 = 0$ en la ecuación $z_n = z_0 + \mathrm{arctg}\left(\frac{y_0}{x_0}\right)$, como se explicó en la sección 4.3.1.

Angulo	Seno MATLAB	Seno CORDIC (16 bits)	Seno CORDIC (32 bits)
$-\frac{\pi}{2}$	-1	-1	-1
$-\frac{\pi}{3}$	-8.660254037844386e-001	-8.660888671875000e-001	-8.660254040732980e-001
$-\frac{\pi}{4}$	-7.071067811865475e-001	-7.070312500000000e-001	-7.071067783981562e-001
$-\frac{\pi}{6}$	-4.999999999999999e-001	-5.000000000000000e-001	-4.999999981373549e-001
0	0	0	0
$\frac{\pi}{6}$	4.999999999999999e-001	4.999389648437500e-001	4.999999981373549e-001
$\frac{\pi}{4}$	7.071067811865475e-001	7.070312500000000e-001	7.071067774668336e-001
$\frac{\pi}{3}$	8.660254037844386e-001	8.660888671875000e-001	8.660254031419754e-001
$\frac{\pi}{2}$	1	9.999389648437500e-001	9.999999990686774e-001

Tabla 5.6

Angulo	Coseno MATLAB	Coseno CORDIC (16 bits)	Coseno CORDIC (32 bits)
$-\frac{\pi}{2}$	0	-6.103515625000000e-005	0
$-\frac{\pi}{3}$	5.000000000000001e-001	5.000000000000000e-001	4.999999962747097e-001
$-\frac{\pi}{4}$	7.071067811865476e-001	7.071533203125000e-001	7.071067811921239e-001
$-\frac{\pi}{6}$	8.660254037844387e-001	8.660278320312500e-001	8.660254050046206e-001
0	1	9.999389648437500e-001	9.999999990686774e-001
$\frac{\pi}{6}$	8.660254037844387e-001	8.660888671875000e-001	8.660254031419754e-001
$\frac{\pi}{4}$	7.071067811865476e-001	7.070922851562500e-001	7.071067811921239e-001
$\frac{\pi}{3}$	5.000000000000001e-001	4.999389648437500e-001	4.999999981373549e-001
$\frac{\pi}{2}$	0	0	9.313225746154784e-010

Tabla 5.7

Tangente	Arcotangente MATLAB	Arcotangente CORDIC (16 bits)	Arcotangente CORDIC (32 bits)
$\mathrm{tg}\left(-\frac{\pi}{2}\right)$	-1.570796326794896	-1.570800781250000	-1.570796329528093
$\mathrm{tg}\left(-\frac{\pi}{3}\right)$	-1.047197551196598	-1.047241210937500	-1.047197550535202
$\mathrm{tg}\left(-\frac{\pi}{4}\right)$	-7.853981633974483e-001	-7.854003906250000e-001	-7.853981647640467e-001
$\mathrm{tg}\left(-\frac{\pi}{6}\right)$	-5.235987755982988e-001	-5.235595703125000e-001	-5.235987771302462e-001
$\mathrm{tg}(0)$	0	0	-1.862645149230957e-009
$\mathrm{tg}\left(\frac{\pi}{6}\right)$	5.235987755982988e-001	5.234375000000000e-001	5.235987752676010e-001
$\mathrm{tg}\left(\frac{\pi}{4}\right)$	7.853981633974483e-001	7.852783203125000e-001	7.853981629014015e-001
$\mathrm{tg}\left(\frac{\pi}{3}\right)$	1.047197551196598	1.047119140625000	1.047197548672557
$\mathrm{tg}\left(\frac{\pi}{2}\right)$	1.570796326794896	1.570678710937500	1.570796327665448

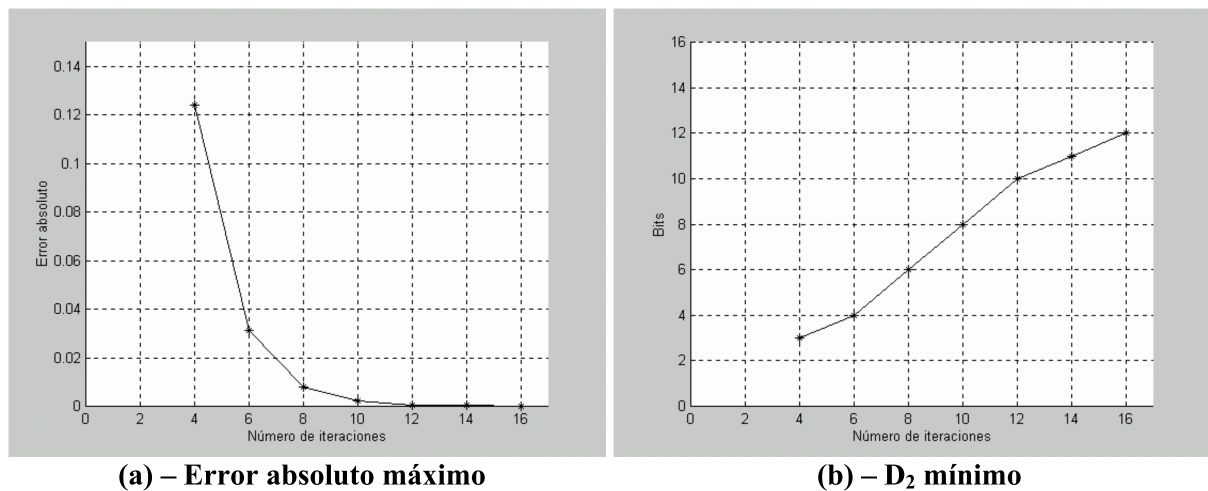
Tabla 5.8

Para las descripciones particulares se calculó el seno, el coseno y el arcotangente dentro de los rangos establecidos en la sección 5.2. El arcotangente, se calculó además para la tangente entre $\mathrm{tg}\left(-\frac{\pi}{2}\right)$ y $\mathrm{tg}\left(\frac{\pi}{2}\right)$ utilizando la función recíproca provista en el banco de pruebas. De ésta manera se pudo reducir a 3 la cantidad de bits correspondientes a la parte entera y se pudo cubrir todo el rango de convergencia del algoritmo CORDIC.

En las figuras 5.4 y 5.5 se exponen el error absoluto (figuras 5.4 y 5.5 a)) y los bits de acuerdo a la fórmula D_2 (figuras 5.4 y 5.5 b)) para el seno y el coseno, con un ancho de palabra de 16 y 32 bits respectivamente.

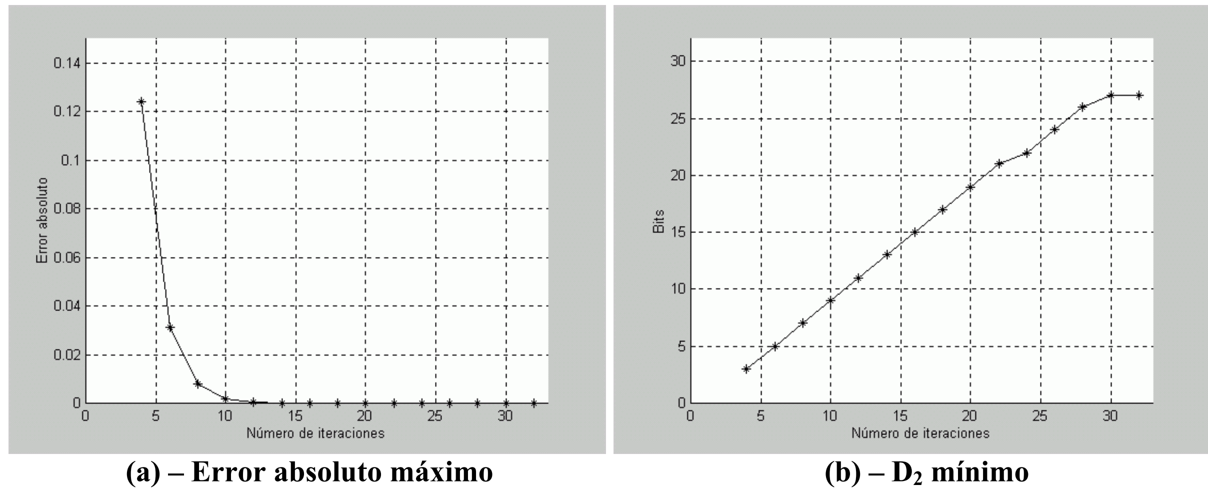
En las tablas 5.9, 5.10, 5.11 y 5.12 se muestran a modo de resumen los extremos de los resultados obtenidos a partir de las descripciones particulares, junto con los de la descripción funcional algorítmica. Las tablas de valores detallados y los gráficos restantes se han reservado para el anexo E.

Se puede observar que el error absoluto máximo disminuye y la cantidad mínima de bits considerados exactos crece cuando aumenta el número de iteraciones que efectúa el algoritmo de forma parecida a lo que ocurría con la descripción funcional algorítmica. De acuerdo a los resultados de las simulaciones, se puede inferir que cada iteración adicional incrementa la exactitud del resultado en 1 bit aproximadamente.



(a) – Error absoluto máximo **(b) – D_2 mínimo**

Figura 5.4 – Seno / Coseno (16 bits)



(a) – Error absoluto máximo **(b) – D_2 mínimo**

Figura 5.5 – Seno / Coseno (32 bits)

Seno para ángulos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$			
Ancho	Iteraciones	E (máximo)	D_2 (mínimo)
Tipo `REAL`	4	1.239941434116256e-001	3
	32	4.624036500922024e-010	31
16 bits	4	1.242654148457032e-001	3
	16	1.958996423664927e-004	12
32 bits	4	1.239941431395144e-001	3
	32	5.156055915556124e-009	27

Tabla 5.9

Coseno para ángulos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$			
Ancho	Iteraciones	E (máximo)	D_2 (mínimo)
Tipo `REAL`	4	1.239941434117357e-001	3
	32	4.622801516585006e-010	31
16 bits	4	1.240364158710175e-001	3
	16	2.309664794707678e-004	12
32 bits	4	1.239941440706783e-001	3
	32	6.584441936130503e-009	27

Tabla 5.10

Arcotangente para la tangente entre $\text{tg}\left(-\frac{\pi}{2}\right)$ y $\text{tg}\left(\frac{\pi}{2}\right)$			
Ancho	Iteraciones	E (máximo)	D_2 (mínimo)
Tipo `REAL`	4	1.243323987289236e-001	3
	32	4.656410812486910e-010	31
16 bits	4	1.244063634178296e-001	3
	16	3.053631790548294e-004	11
32 bits	4	1.243324004358084e-001	3
	32	5.184624507492685e-009	27

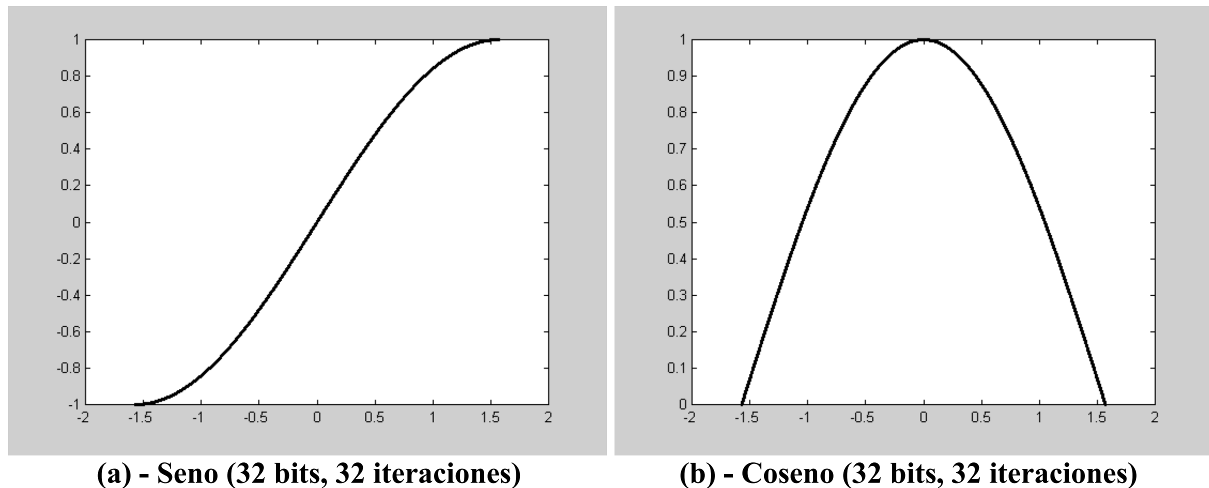
Tabla 5.11

Arcotangente para la tangente entre $\text{tg}\left(-\frac{4\pi}{9}\right)$ y $\text{tg}\left(\frac{4\pi}{9}\right)$			
Ancho	Iteraciones	E (máximo)	D_2 (mínimo)
16 bits	4	1.250465874394296e-001	2
	16	9.284920377063966e-004	10
32 bits	4	1.243324013065256e-001	3
	32	1.486753375967708e-008	26

Tabla 5.12

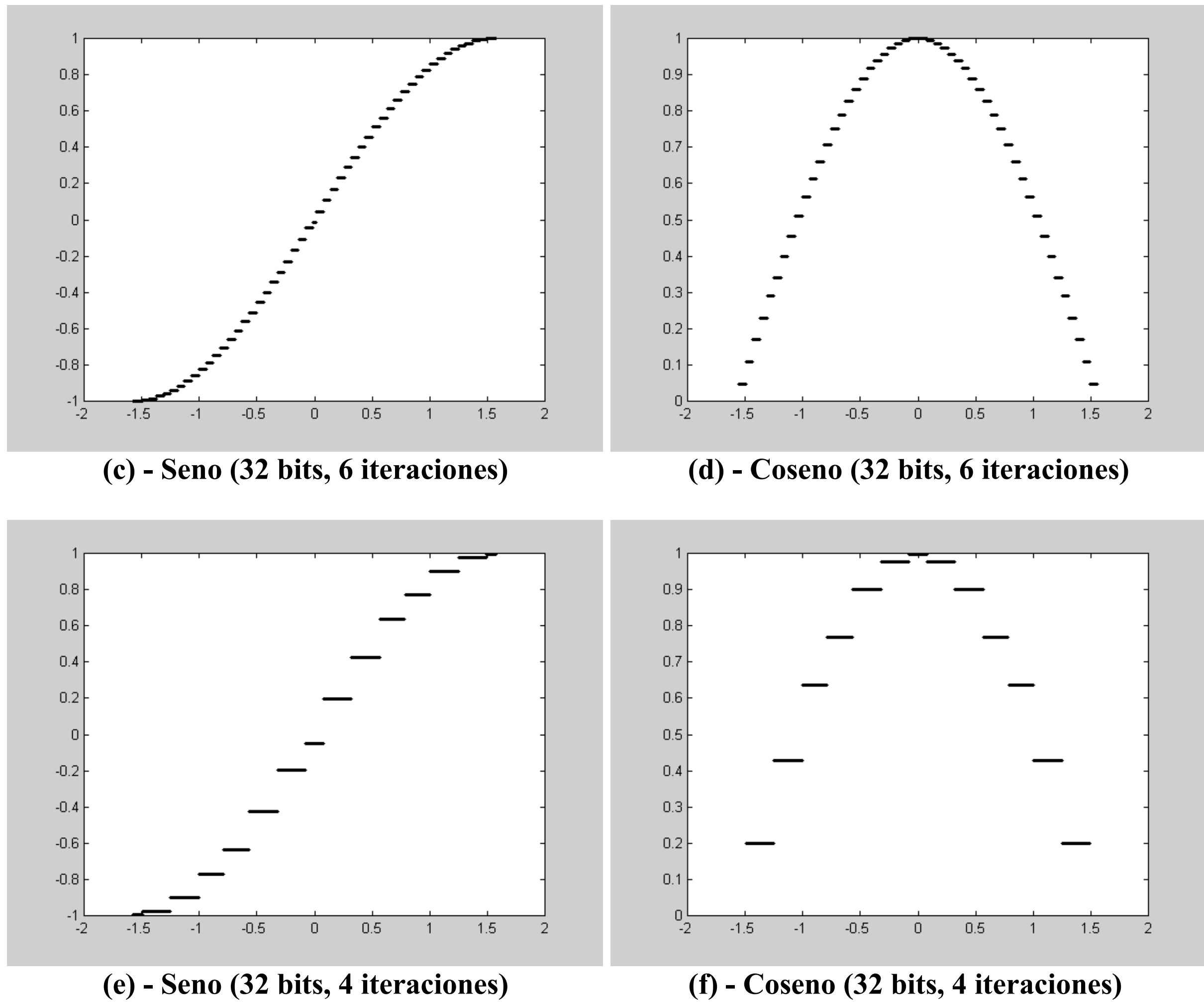
Como se observa en las tablas anteriores el error absoluto máximo disminuye a medida que aumentan las iteraciones. Los valores para el error son aproximados, un mayor detalle de los valores se expone en el anexo E.

Por último, en la figura 5.6 se muestra a modo de ejemplo la gráfica del seno y del coseno dentro del rango de convergencia con un ancho de palabra de 32 bits, para 32, 6 y 4 iteraciones. Como puede observarse, con un número bajo de iteraciones, aumenta la repetición de valores.



(a) - Seno (32 bits, 32 iteraciones) **(b) - Coseno (32 bits, 32 iteraciones)**

Figura 5.6



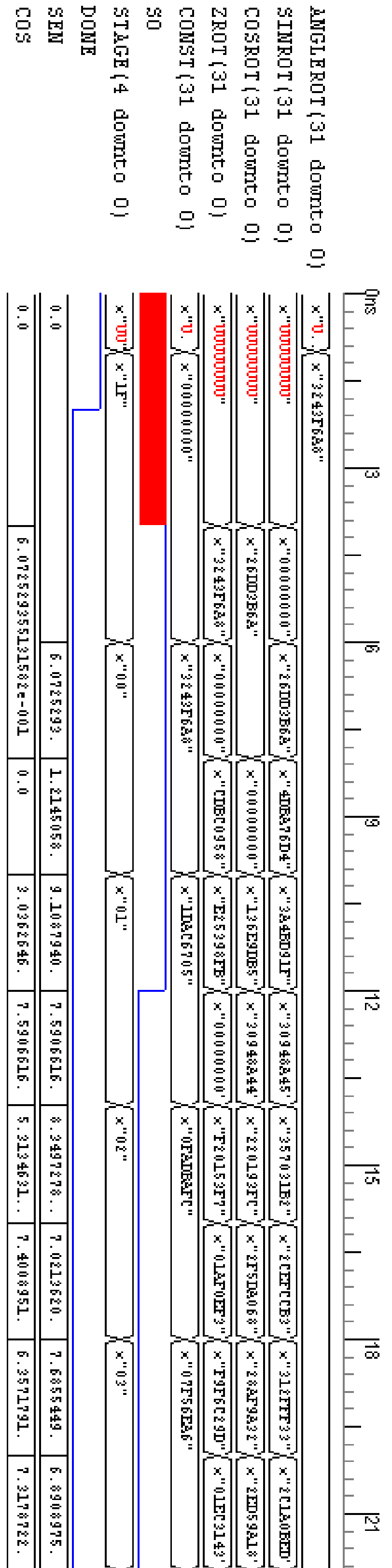
(c) - Seno (32 bits, 6 iteraciones) **(d) - Coseno (32 bits, 6 iteraciones)**

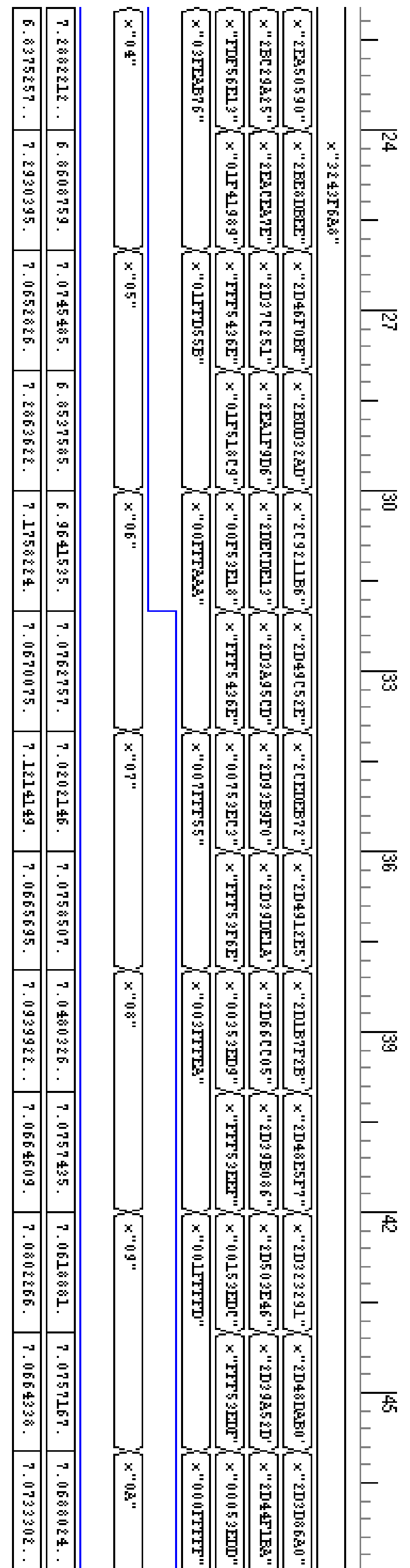
(e) - Seno (32 bits, 4 iteraciones) **(f) - Coseno (32 bits, 4 iteraciones)**

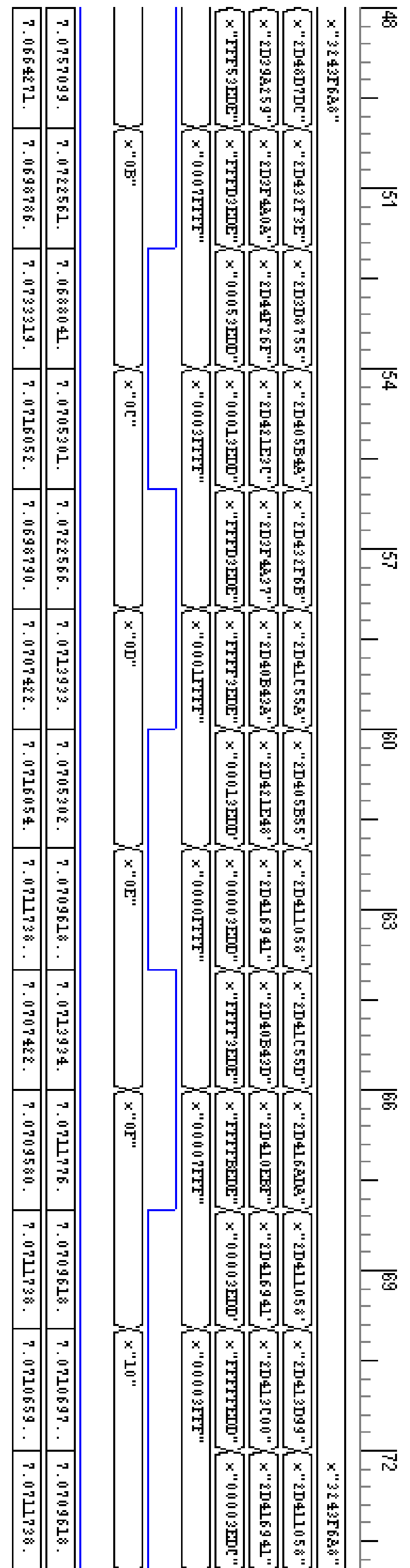
Figura 5.6

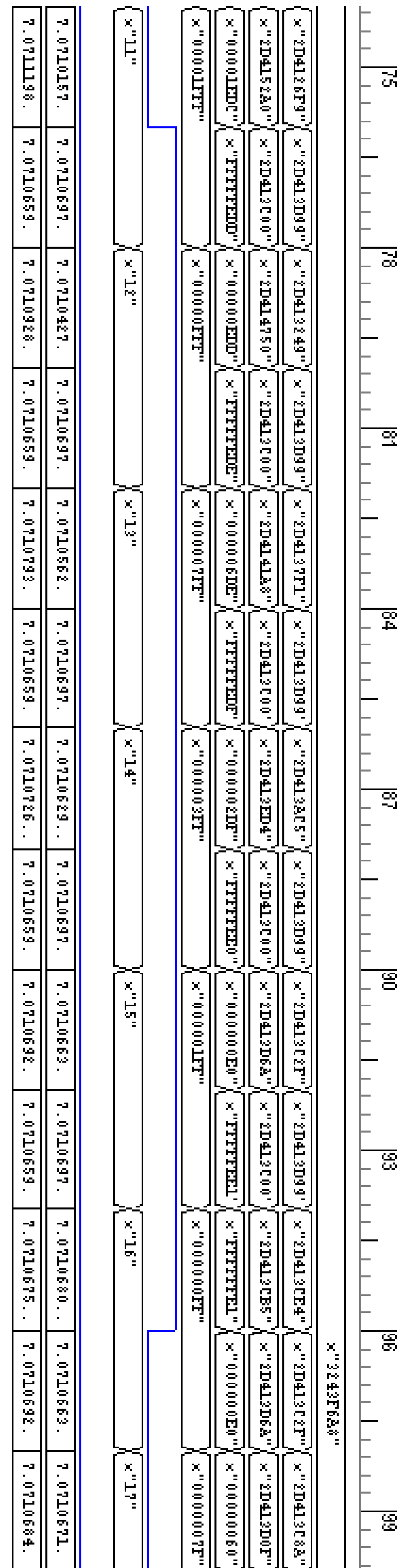
5.4.2 Visualización de ondas

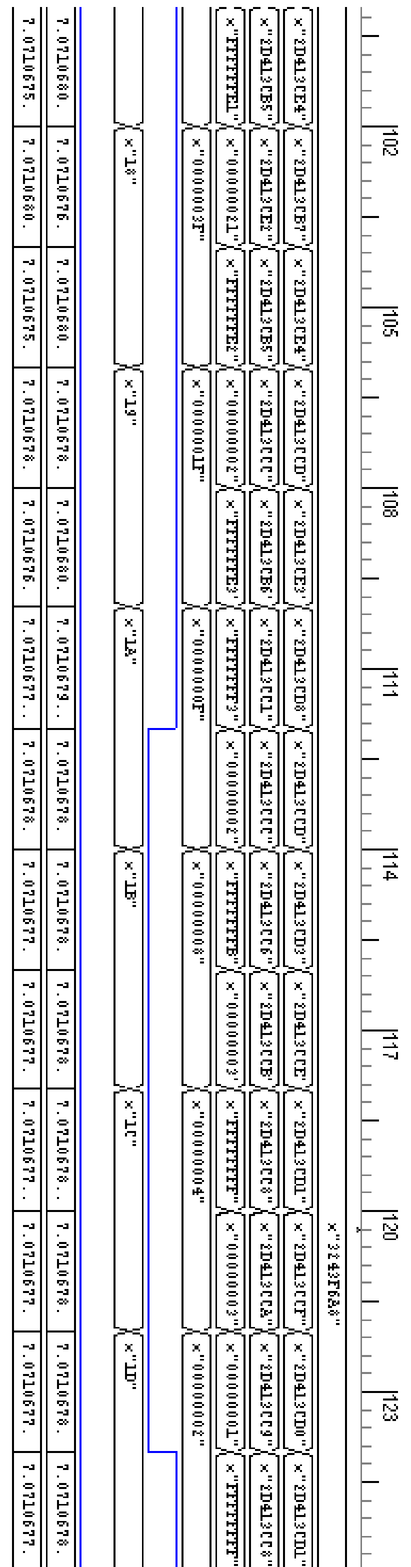
El visor de ondas es una herramienta del simulador de VHDL que resulta especialmente útil durante el diseño. Su función es permitir la visualización y monitorización de señales. La visualización que se expone a continuación es una simulación típica de la descripción de la arquitectura bit-paralela iterativa. Muestra los cambios que se producen en las señales conforme avanza el tiempo de simulación. Se ejemplifica con el cálculo de las funciones seno y coseno para el ángulo $\frac{\pi}{4}$ con un ancho de palabra de 32 bits y para 32 iteraciones. El tiempo de simulación se establece en la descripción del banco de pruebas. En el caso de la simulación funcional no es importante la unidad de tiempo que se utilice. El tiempo de simulación adquiere importancia en las simulaciones digitales, porque es ahí en donde se toman en consideración los retrasos y tiempos de los componentes. Sin embargo se debe establecer un tiempo arbitrario para poder obtener los resultados adecuadamente y provocar el avance del tiempo en el simulador. Para la simulación que se muestra a continuación el tiempo está determinado por el reloj, y se estableció en 1 ns.

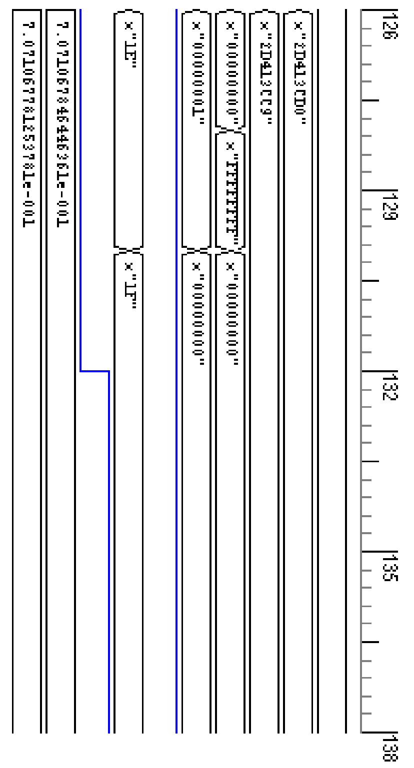

ANGLEROT(31 downto 0)
SINROT(31 downto 0)
COSROT(31 downto 0)
ZROT(31 downto 0)
CONST(31 downto 0)
S0
STAGE(4 downto 0)
DONE
SEN
COS










126
129
132
135
138
x"2D413CD0"
x"2D413CC9"
x"00000000"
x"FFFFFFFF"
x"00000000"
x"00000001"
x"00000000"
x"1F"
x"1F"
7.0710678464463961e-001
7.0710677812537841e-001

Conclusiones

El presente trabajo planteó como objetivo primario el de estudiar el algoritmo CORDIC y utilizar el lenguaje de descripción de hardware VHDL para describir algunas de sus arquitecturas. Como objetivo secundario se propuso simular las descripciones realizadas modificando los parámetros de interés: ancho de palabra y número de iteraciones, para determinar la exactitud que se obtiene en los resultados. La simulación se llevó a cabo con el cálculo de las funciones seno, coseno y arcotangente.

Acerca del uso de VHDL

- VHDL permite diseñar, modelar y verificar un sistema desde un alto nivel de abstracción, refinando el diseño con la posibilidad de culminar con la descripción del mismo hasta el nivel de compuertas. La posibilidad de describir un sistema digital con un alto nivel de abstracción es importante para comprender inicialmente el funcionamiento del mismo, sin necesidad de codificarlo previamente en otro lenguaje.

- Al estar basado en un estándar (IEEE Std 1076-1987 y IEEE Std 1076-1993) puede utilizarse para minimizar problemas de portabilidad.

- El lenguaje permite, de ser necesario, verificar cada componente del sistema por separado.

- Las simulaciones funcionales en VHDL se pueden llevar a cabo fácilmente mediante la descripción algorítmica de los bancos de prueba. Si bien las simulaciones exhaustivas llevan mucho tiempo, se pueden realizar de manera sencilla, ya que las diferentes metodologías existentes para describir bancos de prueba permiten generar gran cantidad de patrones de verificación.

- Los diseños realizados con VHDL no sólo pueden ser simulados, sino también sintetizados con herramientas adecuadas.

- Los componentes descriptos con VHDL para un diseño pueden reutilizarse posteriormente en otros diseños.

- Con el lenguaje VHDL el diseño de sistemas digitales no sólo está limitado a la ingeniería, sino que abre inmensas posibilidades para programadores en general.

- El uso de VHDL no sólo es importante en la industria, sino también para la enseñanza ya que provee una forma legible y estándar de especificar y describir sistemas digitales.

Acerca de las descripciones del algoritmo CORDIC

- La descripción funcional algorítmica con un alto nivel de abstracción, posibilitó la comprensión del funcionamiento del algoritmo CORDIC. El correcto funcionamiento de ésta, facilitó la descripción de las dos arquitecturas particulares.

- Los resultados obtenidos al calcular las funciones matemáticas para ambas arquitecturas particulares fueron los mismos, como era de esperar ya que ambos diseños poseen idéntica funcionalidad.

- Los bancos de prueba permitieron llevar a cabo la simulación de las descripciones en forma sencilla. Sin embargo, el tiempo de ejecución de la simulación resultó prolongado en la mayoría de las pruebas y se necesitó gran potencia de cómputo.

- Para realizar la descripción de las dos arquitecturas particulares del algoritmo se tuvieron que estudiar aspectos relacionados con el diseño lógico de circuitos combinatorios y secuenciales. La descripción modular y simulación en VHDL del algoritmo CORDIC permitieron comprender el funcionamiento de muchos de los componentes básicos que lo constituyen, como por ejemplo el flip-flop D, y su interacción con otros componentes del sistema.

- La potencia y expresividad del lenguaje VHDL permitieron describir en forma compacta componentes con cierto nivel de complejidad como por ejemplo el Barrel Shifter y la tabla de búsqueda para la arquitectura iterativa del algoritmo.

Acerca de la exactitud de los resultados numéricos

El estudio analítico detallado de los errores del algoritmo CORDIC no era objetivo del presente trabajo. Sin embargo al analizar los resultados de las simulaciones, pudo inferirse lo siguiente respecto de la exactitud de los mismos.

- La exactitud del algoritmo es dependiente del ancho de palabra utilizado para las tres componentes *X, Y* y *Z*, así como del número de iteraciones efectuadas. Las operaciones fundamentales que efectúa un procesador CORDIC son de suma y desplazamiento en las cuales la longitud finita en el ancho de palabra introduce errores, que resultan acrecentados por la representación de punto fijo.

  Si se considera por ejemplo a las componentes x_i e y_i con un ancho de palabra m, en la etapa i, si $i \leq m$ entonces x_{i+1} e y_{i+1} se actualizan con el valor truncado de $y_i \cdot 2^{-i}$ y $x_i \cdot 2^{-i}$ respectivamente. Si $i > m$ entonces $x_{i+1} = x_i$ e $y_{i+1} = y_i$, y la actualización valdrá cero. Por lo tanto, el ancho de palabra limita el número máximo de iteraciones útiles, y en principio la mayor exactitud se consigue luego de m iteraciones.

- La exactitud de la rotación de un ángulo α está determinada por la exactitud de la aproximación que se obtiene con la suma de los ángulos α_i. Los errores en aquellas sumas, que se producen por operar con valores truncados, pueden cancelar el aporte de un mayor número de iteraciones.

- Además existe otro tipo de error que tiene lugar cuando alguna de las componentes *X* o *Y* es negativa. En el caso de las iteraciones finales, la actualización de las componentes debería acercarse siempre a cero debido a la gran cantidad de desplazamientos a derecha que se efectúan. Esto ocurre únicamente cuando las componentes *X* o *Y* son positivas. Sin embargo cuando son negativas el resultado se acerca a *–1*.

  A modo de ejemplo se puede considerar el número *1Eh* o decimal positivo *30* representado con un ancho de palabra de 8 bits. El valor desplazado a derecha varias veces se acerca a cero. Si ahora se considera el negativo de dicho número con la representación en complemento a dos, su valor sería *E2h*. El valor desplazado a derecha varias veces ya no se acerca a cero, sino a *FFh* que en complemento a dos es igual a *–1*.

Perspectivas sobre trabajos futuros

- Si bien se hicieron simulaciones funcionales, en un futuro sería interesante realizar también simulaciones digitales teniendo en cuenta retrasos propios de los componentes del sistema para una plataforma específica.

- Plantear la implementación del algoritmo sobre una plataforma de lógica programable.

- Extender el algoritmo considerando los casos hiperbólico y lineal.

- Analizar una representación en punto flotante para las componentes *X*, *Y* y *Z*.

- Analizar arquitecturas alternativas y modificaciones a las arquitecturas descriptas. En particular, la arquitectura bit-paralela desplegada puede modificarse agregando un registro entre cada iteración o luego de varias iteraciones originando una arquitectura desplegada tipo “pipeline”. De esta manera se observa un retardo inicial hasta llenar el pipeline pero para el cálculo de secuencias resulta útil ya que se obtiene un resultado por cada ciclo de reloj. Esta modificación resulta sencilla de realizar con la arquitectura descripta. Simplemente se debe intercalar el registro descripto para la versión iterativa entre cada par de etapas. La arquitectura resultante deja de ser puramente combinatoria debido a la inclusión de elementos de memoria.

Anexo A

El teorema de convergencia

Esta versión del teorema de convergencia es una traducción directa del trabajo de [4].

A.1 Teorema de Convergencia

Sea $\sigma_0 \geq \sigma_1 \geq ... \geq \sigma_n \geq 0$ *una secuencia decreciente de números positivos que satisface*

$$\sigma_k \leq \sigma_n + \sum_{j=k+1}^{n} \sigma_j, \quad 0 \leq k \leq n$$

Sea r un número que satisface

$$|r| \leq \sigma_n + \sum_{j=0}^{n} \sigma_j$$

Se define la secuencia $s_0 = 0$ *y* $s_{k+1} = s_k + \rho_k \sigma_k$, $k = 0, 1, ..., n$ *en donde*

$$\rho_k = \mathrm{sgn}(r - s_k) = \begin{cases} 1, & r \geq s_k \\ -1, & r < s_k \end{cases}$$

Entonces

$$|r - s_k| \leq \sigma_n + \sum_{j=k}^{n} \sigma_j, \quad 0 \leq k \leq n$$

En particular, $|r - s_{n+1}| \leq \sigma_n$

Demostración. La demostración se realizara por inducción sobre k. Para k = 0, se tiene

$$|r - s_0| = |r| \leq \sigma_n + \sum_{j=0}^{n} \sigma_j$$

Si se asume que el teorema es válido para k, se obtiene la siguiente hipótesis inductiva (HI)

$$|r - s_k| \leq \sigma_n + \sum_{j=k}^{n} \sigma_j$$

Considerando el caso para k + 1, se debe demostrar que vale la siguiente tesis inductiva (TI)

$$|r - s_{k+1}| \leq \sigma_n + \sum_{j=k+1}^{n} \sigma_j$$

Si se opera partiendo del lado izquierdo de la TI, se tiene que

$$|r - s_{k+1}| = |r - s_k - \rho_k \sigma_k|$$

Si $r - s_k \geq 0$, entonces $\rho_k = 1$ y $|r - s_k - \rho_k \sigma_k| = \left||r - s_k| - \sigma_k\right|$

Si $r - s_k < 0$, entonces $\rho_k = -1$ y $|r - s_k - \rho_k \sigma_k| = |r - s_k + \sigma_k| = \left||r - s_k| - \sigma_k\right|$

Por lo tanto en ambos casos $|r - s_{k+1}| = \left||r - s_k| - \sigma_k\right|$

De la primera desigualdad

$$-\left(\sigma_n + \sum_{j=k+1}^{n} \sigma_j\right) \leq -\sigma_k \leq |r - s_k| - \sigma_k$$

Por HI

$$|r - s_k| - \sigma_k \leq \left(\sigma_n + \sum_{j=k}^{n} \sigma_j\right) - \sigma_k = \left(\sigma_n + \sum_{j=k+1}^{n} \sigma_j\right)$$

Si se combinan ambas desigualdades

$$|r - s_{k+1}| = \left||r - s_k| - \sigma_k\right| \leq \sigma_n + \sum_{j=k+1}^{n} \sigma_j$$

Lo que demuestra que el teorema vale para k + 1.

Por último $-\sigma_n \leq |r - s_n| - \sigma_n \leq 2\sigma_n - \sigma_n = \sigma_n$ y entonces $|r - s_{n+1}| = \left||r - s_n| - \sigma_n\right| \leq \sigma_n$ lo que completa la demostración.

■

Teorema. *Para* $n > 3$*, la secuencia* $\sigma_k = \text{arctg}\left(2^{-k}\right)$ $;k = 0,1,..,n$ *satisface las hipótesis del teorema de convergencia citado en el punto anterior, para todo* $|r| \leq \frac{\pi}{2}$.

Demostración. La secuencia

$$\text{arctg}\left(2^{0}\right), \text{arctg}\left(2^{-1}\right), \text{arctg}\left(2^{-2}\right), ..., \text{arctg}\left(2^{-n}\right)$$
$$= \text{arctg}(1), \text{arctg}\left(\frac{1}{2}\right), \text{arctg}\left(\frac{1}{4}\right), ..., \text{arctg}\left(\frac{1}{2^{n}}\right)$$

es claramente una secuencia decreciente de números positivos.

El Teorema del Valor Medio (TVM) afirma que existe un número c entre a y b tal que,

$$\frac{\text{arctg}\, b - \text{arctg}\, a}{b - a} = \frac{1}{1 + c^{2}} \quad ,a < c < b$$

Sea $a = 2^{-(j+1)}$ y $b = 2^{-j}$ en la ecuación del TVM. Entonces $b - a = 2^{-(j+1)}$ y además se verifica que

$$\frac{1}{1 + c^{2}} < \frac{1}{1 + a^{2}} = \frac{1}{1 + 2^{-2(j+1)}} = \frac{2^{2(j+1)}}{1 + 2^{2(j+1)}}$$

Luego

$$\sigma_j - \sigma_{j+1} = (b - a)\frac{1}{1 + c^{2}} \leq \frac{1}{2^{j+1}}\frac{2^{2(j+1)}}{1 + 2^{2(j+1)}} = \frac{2^{j+1}}{1 + 2^{2(j+1)}}$$

Sean $a = 0$ y $b = 2^{-j}$ en la ecuación del TVM. Entonces se verifica que

$$\frac{1}{1 + c^{2}} > \frac{1}{1 + b^{2}} = \frac{1}{1 + 2^{-2j}} = \frac{2^{2j}}{1 + 2^{2j}} \qquad \text{y} \qquad \sigma_j = b\frac{1}{1 + c^{2}} \geq \frac{1}{2^{j}}\frac{2^{2j}}{1 + 2^{2j}} = \frac{2^{j}}{1 + 2^{2j}}$$

se combinan las desigualdades que involucran a las σ_j utilizando series.

$$\sigma_k - \sigma_n = \left(\sigma_k - \sigma_{k+1}\right) + \left(\sigma_{k+1} - \sigma_{k+2}\right) + ... + \left(\sigma_{n-1} - \sigma_n\right)$$

$$= \sum_{j=k}^{n-1}\left(\sigma_j - \sigma_{j+1}\right) \leq \sum_{j=k}^{n-1}\frac{2^{j+1}}{1 + 2^{2(j+1)}} = \sum_{j=k+1}^{n}\frac{2^{j}}{1 + 2^{2j}} \leq \sum_{j=k+1}^{n}\sigma_j$$

con lo que se concluye que

$$\sigma_k \leq \sigma_n + \sum_{j=k+1}^{n}\sigma_j \quad ,0 \leq k < n$$

Como la secuencia $\text{arctg}(1)+\text{arctg}\left(\frac{1}{2}\right)+\text{arctg}\left(\frac{1}{4}\right)+...+\text{arctg}\left(\frac{1}{2^n}\right)>\frac{\pi}{2} \quad ,n\geq 3$ se puede concluir que

$$|r|\leq\frac{\pi}{2}<\sum_{j=0}^{3}\text{arctg}\left(2^{-j}\right)<\sigma_n+\sum_{j=0}^{n}\sigma_j$$

con lo que queda demostrado el teorema.

■

A.2 Convergencia del algoritmo CORDIC

Si se define la secuencia $s_k=\phi-z_k=\sum_{j=0}^{k-1}d_j\sigma_j$ puede apreciarse que

$$\begin{aligned} &s_0=\phi-z_0=0\\ &...\\ &s_{k+1}=\sum_{j=0}^{k}d_j\sigma_j=s_k+d_k\sigma_k \end{aligned}$$

Con $r=\phi$ se tiene que $\rho_k=\text{sgn}(r-s_k)=\text{sgn}(\phi-s_k)=\text{sgn}(z_k)=d_k$

Por lo tanto la secuencia $|\phi-s_{n+1}|\leq\sigma_n=\text{arctg}\left(2^{-n}\right)\leq\frac{1}{2^n}$ satisface el teorema de convergencia y se puede concluir que la secuencia utilizada por el algoritmo CORDIC, converge de acuerdo a lo establecido en el teorema anterior.

■

Anexo B

Funciones booleanas

El álgebra de Boole provee las operaciones y las reglas para trabajar con el conjunto $\{0, 1\}$. Los dispositivos electrónicos pueden estudiarse utilizando este conjunto y las reglas asociadas al álgebra de Boole. Las tres operaciones utilizadas mas comúnmente son complemento, suma booleana (OR) y producto (AND).

B.1 Funciones y expresiones booleanas

Sea $B = \{0, 1\}$. La variable x se denomina Variable booleana si asume únicamente valores del conjunto B. Una función de B^n, el conjunto $\{(x_1, x_2, \dots, x_n) \mid x_i \in B, 1 \leq i \leq n\}$ en B se denomina función booleana de grado n.

Las funciones booleanas pueden representarse usando expresiones construidas a partir de variables y operaciones booleanas. Las expresiones booleanas en las variables $x_1, x_2, \dots, x_n$ se definen en forma recursiva como sigue

$0, 1, x_1, x_2, \dots, x_n$ son expresiones booleanas.
Si E_1 y E_2 son expresiones booleanas, entonces E_1, $(E_1 . E_2)$ y $(E_1 + E_2)$ son expresiones booleanas.

Cada expresión booleana representa una función. Los valores de esta función se obtienen sustituyendo 0 y 1 en las variables presentes en la expresión.

Las funciones booleanas F y G de n variables se dicen equivalentes si y solo si $F(b_1, b_2, \dots, b_n) = G(b_1, b_2, \dots, b_n)$, cuando $b_1, b_2, \dots, b_n \in B$.

Una función booleana de grado 2 es una función de un conjunto con cuatro elementos, pares de elementos del conjunto $\{0, 1\}$ en B, un conjunto con dos elementos. De manera tal que existen 16 funciones booleanas diferentes de grado 2.

B.2 Identidades del álgebra booleana

Las identidades del álgebra booleana son particularmente útiles para simplificar el diseño de circuitos. Son proposiciones equivalentes y se pueden demostrar utilizando tablas de verdad. Las identidades se muestran en la tabla B.1

Identidad	Nombre
$\overline{\overline{x}} = x$	Doble complemento
$x + x = x$ $x \,.\, x = x$	Idempotencia
$x + 0 = x$ $x \,.\, 1 = x$	Identidad
$x + 1 = 1$ $x \,.\, 0 = 0$	Dominancia
$x + y = y + x$ $x \,.\, y = y \,.\, x$	Conmutatividad
$x + (y + z) = (x + y) + z$ $x \,.\, (y \,.\, z) = (x \,.\, y) \,.\, z$	Asociatividad
$x + y \,.\, z = (x + y) \,.\, (x + z)$ $x \,.\, (y + z) = x \,.\, y + x \,.\, z$	Distributividad
$\overline{(x.y)} = \overline{x} + \overline{y}$ $\overline{(x + y)} = \overline{x}.\overline{y}$	DeMorgan

Tabla B.1

B.3 Representación de funciones booleanas

Expansiones de suma-producto

Un *minitérmino* de las variables booleanas $x_1, x_2, \ldots, x_n$ es un producto booleano $y_1 \,.\, y_2 \ldots y_n$ en donde $y_i = x_i$ o bien $y_i = \overline{x_i}$. Un *literal* es una variable booleana o su complemento. Por lo tanto un minitérmino es un producto de n literales con un literal para cada variable.

Un minitérmino tiene un valor de *1* si y solo si cada variable y_i tiene un valor de *1*.

Tomando sumas booleanas de distintos minitérminos se puede construir una expresión booleana con un conjunto específico de valores. En particular una suma booleana de minitérminos tiene un valor de 1 cuando exactamente uno de los minitérminos en la suma tiene valor *1* y adquiere el valor 0 para cualquier otra combinación de valores de las variables.

Una expansión de suma-producto es una suma de minitérminos. Los minitérminos en la suma booleana corresponden a aquellas combinaciones de valores en los cuales la función adquiere el valor *1*.

A modo de ejemplo se puede encontrar la función booleana correspondiente a la tabla B.2

x	*y*	*z*	*F(x,y,z)*
0	*0*	*0*	*0*
1	*0*	*0*	*0*
0	*1*	*0*	*0*
1	*1*	*0*	*0*
0	*0*	*1*	*0*
1	*0*	*1*	*0*
0	*1*	*1*	*1*
1	*1*	*1*	*1*

Tabla B.2

Para representar *F*, se necesita una expresión que valga *1* en caso de que $x = 0$ e $y = z = 1$ o bien $x = y = z = 1$. Dicha expresión se puede construir por medio de una suma booleana de dos productos diferentes. Por lo tanto la función *F* quedaría

$$F(x, y, z) = \overline{x}.y.z + x.y.z$$

B.4 Completitud funcional

Toda función booleana puede representarse por una suma de minitérminos. Cada minitérmino es el producto booleano de variables booleanas o sus complementos. Esto demuestra que cada función booleana puede expresarse con los operadores +, . y ‾ [23]. Como cada función booleana se puede representar, se dice que el conjunto {+, . , ‾ } es *funcionalmente completo*. Un conjunto menor funcionalmente completo puede construirse si se consigue expresar alguno de los operadores en términos de los otros dos. Por lo tanto si se utilizan las leyes de DeMorgan, se pueden eliminar las sumas booleanas utilizando la identidad

$$x + y = \overline{\overline{x}.\overline{y}}$$

Esto significa que el conjunto { ‾ , .} es funcionalmente completo. De manera similar se puede deducir que el conjunto { ‾ , +} también es funcionalmente completo aplicando la segunda ley de DeMorgan

$$x.y = \overline{\overline{x} + \overline{y}}$$

Sin embargo se pueden obtener conjuntos funcionalmente completos, aún mas pequeños. Se define la operación | o (NAND), que dados dos variables booleanas retorna el complemento del producto booleano y \ (NOR) que dadas dos variables booleanas retorna el complemento de la suma booleana. Si se construye un conjunto con cada uno de los operadores, { | } y { \ }, se puede demostrar que ambos conjuntos son funcionalmente completos.

$$\overline{x} = x \mid x$$

$$x.y = (x \mid y) \mid (x \mid y)$$

$$\overline{x} = x \setminus 0$$
$$x.y = (x \setminus 0) \setminus (y \setminus 0)$$

En virtud de la completitud funcional de { . , ¯ }, queda demostrada la completitud de { | } y { \ }.

B.5 Compuertas lógicas

El álgebra booleana se utiliza para modelar los circuitos electrónicos. Un dispositivo electrónico está constituido por un número de circuitos. Cada circuito puede diseñarse aplicando las reglas del álgebra de Boole. Los elementos básicos de los circuitos se denominan *compuertas*. Cada tipo de compuerta representa una operación booleana. En la figura B.1 se muestran los diversos tipos de compuertas. Cada una corresponde a una operación determinada.

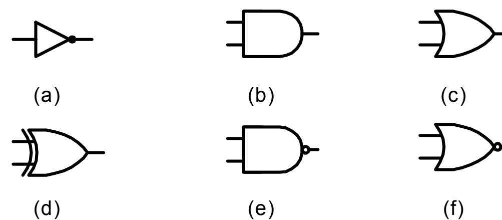


Figura B.1

La compuerta que se observa en la figura B.1 (a) se denomina inversor y representa la operación booleana de negación o NOT, y produce el complemento del valor dado como entrada. En la figura B.1 (b) se presenta la compuerta que representa el producto o AND y en la figura B.1 (c), la compuerta que representa la suma booleana u OR. Las tres últimas compuertas representan las operaciones XOR u OR exclusivo, NAND y NOR. La operación XOR a diferencia del OR, retorna 1 únicamente cuando los valores de entrada son distintos. El funcionamiento de las operaciones NAND (figura B.1 (e)) y NOR (figura B.1 (f)) se explicó en la sección anterior.

Las compuertas anteriores, se pueden utilizar para desarrollar circuitos lógicos combinatorios.

Circuito combinatorio: Se denomina circuito combinatorio a un circuito lógico cuya salida depende únicamente de la entrada y no del estado actual del circuito. En otras palabras, estos son circuitos que se construyen únicamente combinando las diversas compuertas lógicas y por lo tanto carecen de memoria.

B.6 Minimización de circuitos lógicos

La eficiencia de un circuito combinatorio depende del número y organización de la compuertas lógicas que lo comprenden. El diseño de un circuito lógico combinatorio comienza con su especificación mediante una tabla de verdad. A partir de la tabla se pueden utilizar las expansiones de suma-producto para diseñar un conjunto de compuertas lógicas que implementen el circuito. Sin embargo la expansión de suma-producto puede contener mas términos de los realmente necesarios. Los términos que difieren en una sola variable, de tal manera que en un término ocurre la variable y en otro término ocurre su complemento, se pueden combinar. A modo de ejemplo se considera una expansión de suma-producto con las características mencionadas anteriormente, junto con la forma de combinar los términos:

$$\bar{x}.y.z + x.y.z = (\bar{x} + x).(y.z) = 1.(y.z) = y.z$$

La expansión inicial utiliza tres compuertas lógicas y un inversor, mientras que la expansión final utiliza sólo una compuerta.

Para reducir el número de términos en una expresión booleana, se pueden utilizar las identidades definidas en la sección B.2 para encontrar los términos que se puedan combinar. Sin embargo esta tarea puede complicarse a medida que aumenta el número de variables.

Algunos de los métodos que se utilizan para simplificar expresiones booleanas los constituyen el *Mapa de Karnaugh* que es un método gráfico para encontrar los términos que se pueden combinar en una expresión y el método de *Quine-McCluskey* utilizado en expresiones con un gran número de variables [23] [24].

Anexo C

El script de MATLAB™

MATLAB™ es una marca registrada de **The MathWorks, Inc.**

Para realizar el análisis de los resultados de la simulación, se emplearon los resultados calculados con MATLAB™ como patrón de comparación. Se implementó un script para leer de los diversos archivos los valores generados durante las pruebas. Estos valores se compararon con los generados por MATLAB™.

El código que se transcribe a continuación corresponde a un script como el utilizado para realizar el análisis y los gráficos del seno de las descripciones particulares. Los scripts para el coseno y el arcotangente son similares.

```
% Comparar Matlab con CORDIC
clear all;

% Numero de bits de los operandos y parte entera
parteEntera=2;
precision=32;

j=1;
iter=4;
while iter<=precision

   if iter<10
      d=strcat('0',num2str(iter));
   else
      d=num2str(iter);
   end

   % Abro el archivo para leer
   s=strcat('sen',num2str(precision),d,'_min.txt');
   b=fopen(s);
   line2=fgetl(b);

   x(1)=-pi/2;%-4*pi/9;
   num=bin2dec(line2);

   % Es negativo?
   if line2(1)=='1'
      num=pow2(precision)-num;
      num=num/pow2(precision-parteEntera)*(-1);
   else
      num=num/pow2(precision-parteEntera);
```

```
    end

    temp=abs(num-sin(x(1)));
    y(1)=temp;

    if temp==0
       w(1)=precision-parteEntera;
    else
       w(1)=ceil(-log2(temp))-1;
       if w(1)>precision-parteEntera
          w(1)=precision-parteEntera;
       end
    end

    i=2;
    while 1
       line2=fgetl(b);

       if ~isstr(line2)
          break;
       end

       x(i)=x(i-1)+(pi/10800);
       num=bin2dec(line2);

       % Es negativo?
       if line2(1)=='1'
          num=pow2(precision)-num;
          num=num/pow2(precision-parteEntera)*(-1);
       else
          num=num/pow2(precision-parteEntera);
       end

       temp=abs(num-sin(x(i)));
       y(i)=temp;

       if temp==0
          w(i)=precision-parteEntera;
       else
          w(i)=ceil(-log2(temp))-1;
          if w(i)>precision-parteEntera
             w(i)=precision-parteEntera;
          end
       end

       i=i+1;
    end

    % Cierro los archivos
    fclose(b);

    maxErr(j)=max(y);

    minL2(j)=min(w);

    it(j)=iter;

    j=j+1;
    iter=iter+2;
end
```

```
plot(it,maxErr,'k*-');
xlabel('Número de iteraciones');
ylabel('Error absoluto');
axis([0 precision+1 0 0.15]);
grid;

plot(it,minL2,'k*-');
xlabel('Número de iteraciones');
ylabel('Bits');
axis([0 precision+1 0 precision]);
grid;
```

Anexo D

Los bancos de prueba

Para suministrar los patrones de prueba a cada descripción del algoritmo CORDIC, se describieron bancos de prueba. A modo de ejemplo se transcribe un banco de prueba para la descripción funcional algorítmica y uno para cada descripción particular. Se describieron varios bancos de prueba que se utilizaron para efectuar las pruebas con el arcotangente y los ángulos particulares. Dichos bancos de prueba son similares a los que se presentan aquí y se incluyen junto con las descripciones en el CD-ROM adjunto.

El banco de pruebas que se muestra a continuación corresponde a la descripción funcional algorítmica del algoritmo CORDIC. El bucle While suministra al procedimiento `cordic` los ángulos comprendidos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$ en modo rotación y la tangente de los ángulos en modo vectorización. Los resultados se almacenan en formato `REAL` provisto por el lenguaje VHDL que a su vez utiliza internamente la representación estándar IEEE 759 de 64 bits. Dichos valores son utilizados por MATLAB™ para realizar las comparaciones.

```
entity TestCordic is
end TestCordic;

architecture TestCordicArch of TestCordic is

	use work.Cordic.all;
	use std.textio.all;

begin

Prueba:
	process
	variable tita,K:real;
	variable sen,cos,atan:real;
	variable xo,yo,zo:real;
	variable linea:line;

	type fi_real is file of real;

	file salidaSen:fi_real open write_mode is "senFunc32_min.r";
	file salidaCos:fi_real open write_mode is "cosFunc32_min.r";
	file salidaATan:fi_real open write_mode is "atanFunc32_min.r";

	begin
```

```
            K:=1.0;
            for i in 0 to Iteraciones-1 loop
                  K:=K*(1.0/sqrt(1.0+2.0**(-2*i)));
            end loop;

            tita:=-math_pi/2.0;
            while tita<=math_pi/2.0 loop

                  wait for 1 ns;

                  cordic(K,0.0,tita,cos,sen,zo,Rotating);
                  cordic(1.0,tan(tita),0.0,xo,yo,atan,Vectoring);

                  write(salidaSen,sen);
                  write(salidaCos,cos);
                  write(salidaAtan,atan);

                  tita:=tita+(math_pi/10800.0);

            end loop;

            wait;

      end process;

end TestCordicArch;
```

Seguidamente se transcriben los bancos de pruebas utilizados para calcular las funciones seno y coseno, correspondiente a las descripciones bit-paralela iterativa y bit-paralela desplegada. Puede observarse la declaración del componente `ParalellIterativeCORDIC` y `ParalellUnrolledCORDIC` cuya descripción fue explicada en el capítulo 4. La descripción iterativa es secuencial, por lo tanto utiliza una señal de reloj como forma de sincronismo. Para la descripción desplegada la señal de reloj no fue necesaria. Sin embargo se incluyó retardos mediante el uso de la construcción `wait for 1 ns`. Dichos retardos deben incluirse luego de las asignaciones secuenciales a las señales, ya que éstas adquieren el valor en el próximo instante de simulación, como se explicó en las secciones 3.6.11.1 y 3.6.15.

Los patrones de prueba son generados utilizando el tipo de datos `REAL` provisto por VHDL y convertidos a la representación en punto fijo. Los resultados de la simulación se almacenan en archivos de texto. El formato con que se almacenan los valores es como cadena de caracteres compuestas por ‘1’ y ‘0’.

```
entity TestSinCosIterativeRange is
end TestSinCosIterativeRange;

architecture TestSinCosArch of TestSinCosIterativeRange is

      use work.typeconversion.all;
      use std.textio.all;

      constant ext:natural:=2;

      component ParalellIterativeCordic is

            generic(iterations,wide,ext:natural;arctanLUT:STDLVectorW);
            port(
                  clk   :in std_logic;
                  rst   :in std_logic;
```

```
                mode  :in std_logic;
                Xo    :in std_logic_vector(wide-1 downto 0);
                Yo    :in std_logic_vector(wide-1 downto 0);
                Zo    :in std_logic_vector(wide-1 downto 0);

                Xn    :out std_logic_vector(wide-1 downto 0);
                Yn    :out std_logic_vector(wide-1 downto 0);
                Zn    :out std_logic_vector(wide-1 downto 0);
                done  :out std_logic
            );

    end component;


    signal angleRot,sinRot,cosRot,zRot:std_logic_vector(wide-1 downto 0);
    signal ceroIn,K:std_logic_vector(wide-1 downto 0);

    signal clk:std_logic:='0';
    signal rstRot,doneRot:std_logic:='0';

begin

    clk<=not clk after 1 ns;

    ceroIn<=conv_real(0.0,wide,ext);

    ROT: ParalellIterativeCORDIC generic map (iterations,wide,ext,arctanLUT)
            port map(clk,rstRot,'1',K,ceroIn,angleRot
                        ,cosRot,sinRot,zRot,doneRot);
Prueba:
    process
    variable tita,Kr:real;
    variable linea:line;
    variable s:string(1 to wide);

    file salidaSen:text open write_mode is "sen_min.txt";
    file salidaCos:text open write_mode is "cos_min.txt";

    begin

            Kr:=1.0;
            for i in 0 to iterations-1 loop
                Kr:=Kr*(1.0/sqrt(1.0+2.0**(-2*i)));
            end loop;

            K<=conv_real(Kr,wide,ext);

            tita:=-math_pi/2.0;
            while tita<=math_pi/2.0 loop

                wait for 1 ns;
                angleRot<=conv_real(tita,wide,ext);
                rstRot<='1','0' after 2 ns;
                wait until doneRot='1';

                s:=convert(sinRot);
                write(linea,s);
                writeline(salidaSen,linea);
                s:=convert(cosRot);
                write(linea,s);
                writeline(salidaCos,linea);
```

```
                tita:=tita+(math_pi/10800.0);

            end loop;

            wait;

      end process;

end TestSinCosArch;




entity TestSinCosUnrolledRange is
end TestSinCosUnrolledRange;

architecture TestSinCosArch of TestSinCosUnrolledRange is

      use work.TypeConversion.all;
      use work.ConfTypes.all;
      use std.textio.all;

      constant ext:natural:=2;

      component ParalellUnrolledCORDIC is

            generic(iterations,wide,ext:natural;arctanLUT:STDLVectorW);

            port(
                  mode  :in std_logic;
                  Xo    :in std_logic_vector(wide-1 downto 0);
                  Yo    :in std_logic_vector(wide-1 downto 0);
                  Zo    :in std_logic_vector(wide-1 downto 0);

                  Xn    :out std_logic_vector(wide-1 downto 0);
                  Yn    :out std_logic_vector(wide-1 downto 0);
                  Zn    :out std_logic_vector(wide-1 downto 0)
            );

      end component;


      signal angleRot,sinRot,cosRot,zRot:std_logic_vector(wide-1 downto 0);
      signal ceroIn,K:std_logic_vector(wide-1 downto 0);

begin

      ceroIn<=conv_real(0.0,wide,ext);


      ROT: ParalellUnrolledCORDIC generic map (iterations,wide,ext,arctanLUT)
                        port map('1',K,ceroIn,angleRot,cosRot,sinRot,zRot);
Prueba:
      process
      variable tita,Kr:real;
      variable linea:line;
      variable s:string(1 to wide);

      file salidaSen:text open write_mode is "sen_min.txt";
      file salidaCos:text open write_mode is "cos_min.txt";
```

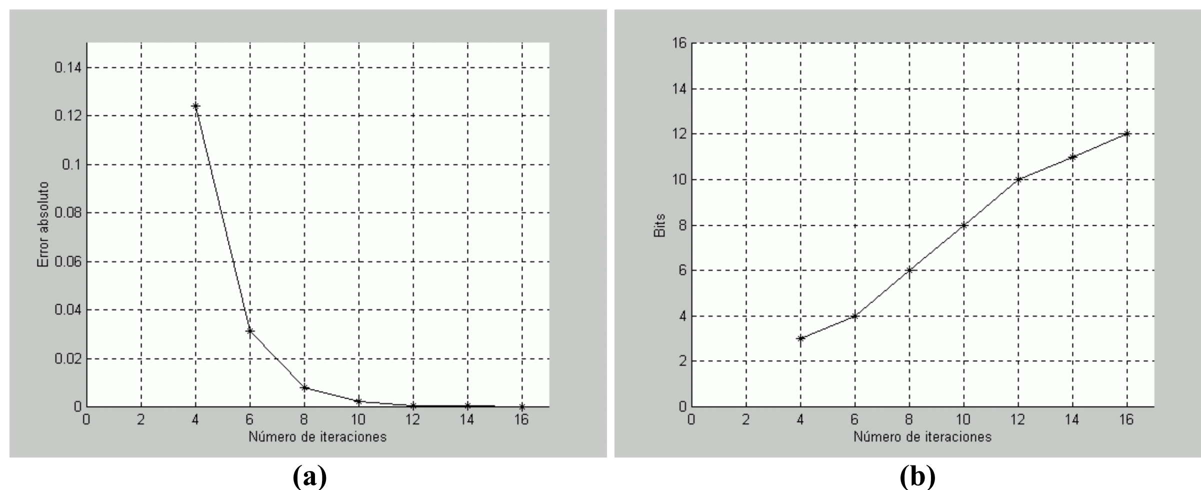
```
	begin

		Kr:=1.0;
		for i in 0 to iterations-1 loop
			Kr:=Kr*(1.0/sqrt(1.0+2.0**(-2*i)));
		end loop;

		K<=conv_real(Kr,wide,ext);

		tita:=-math_pi/2.0;
		while tita<=math_pi/2.0 loop
			wait for 1 ns;

			angleRot<=conv_real(tita,wide,ext);

			wait for 1 ns;

			s:=convert(sinRot);
			write(linea,s);
			writeline(salidaSen,linea);
			s:=convert(cosRot);
			write(linea,s);
			writeline(salidaCos,linea);

			tita:=tita+(math_pi/10800.0);

		end loop;

		wait;

	end process;

end TestSinCosArch;
```

Anexo E

Resultados adicionales de las simulaciones

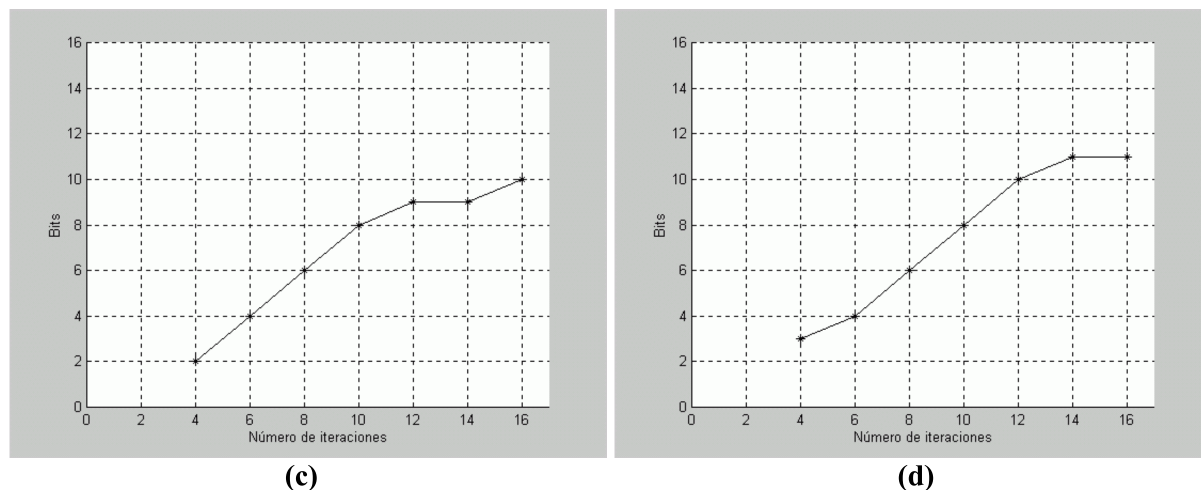
Este anexo presenta en detalle los valores obtenidos como aplicación de las fórmulas introducidas en el capítulo 5 a los valores de simulación. Las tablas E.1, E.2, E.3 y E.4 exponen los valores para el error y la cantidad bits considerados exactos para el seno, el coseno y el arcotangente. El ancho de palabra utilizado es de 16 y 32 bits respectivamente. El número de iteraciones se estableció entre 4 y el máximo permitido por cada representación y se toman en cuenta únicamente iteraciones pares. Para cada iteración se obtuvo el máximo error absoluto y el mínimo número de bits considerados exactos.

En la figura E.1 se graficaron los valores obtenidos a partir de las tablas E.1 y E.2 con un ancho de palabra de 16 bits, entre 4 y 16 iteraciones.



(a) **(b)**

Figura E.1



(c) (d)

Figura E.1

Figura E.1 (a) Error absoluto máximo para el seno, el coseno y el arcotangente.

Figura E.1 (b) D_2 mínimo para el seno y el coseno para ángulos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$.

Figura E.1 (c) D_2 mínimo para el arcotangente para la tangente entre $\operatorname{tg}\left(-\frac{4\pi}{9}\right)$ y $\operatorname{tg}\left(\frac{4\pi}{9}\right)$.

Figura E.1 (d) Arcotangente para la tangente entre $\operatorname{tg}\left(-\frac{\pi}{2}\right)$ y $\operatorname{tg}\left(\frac{\pi}{2}\right)$.

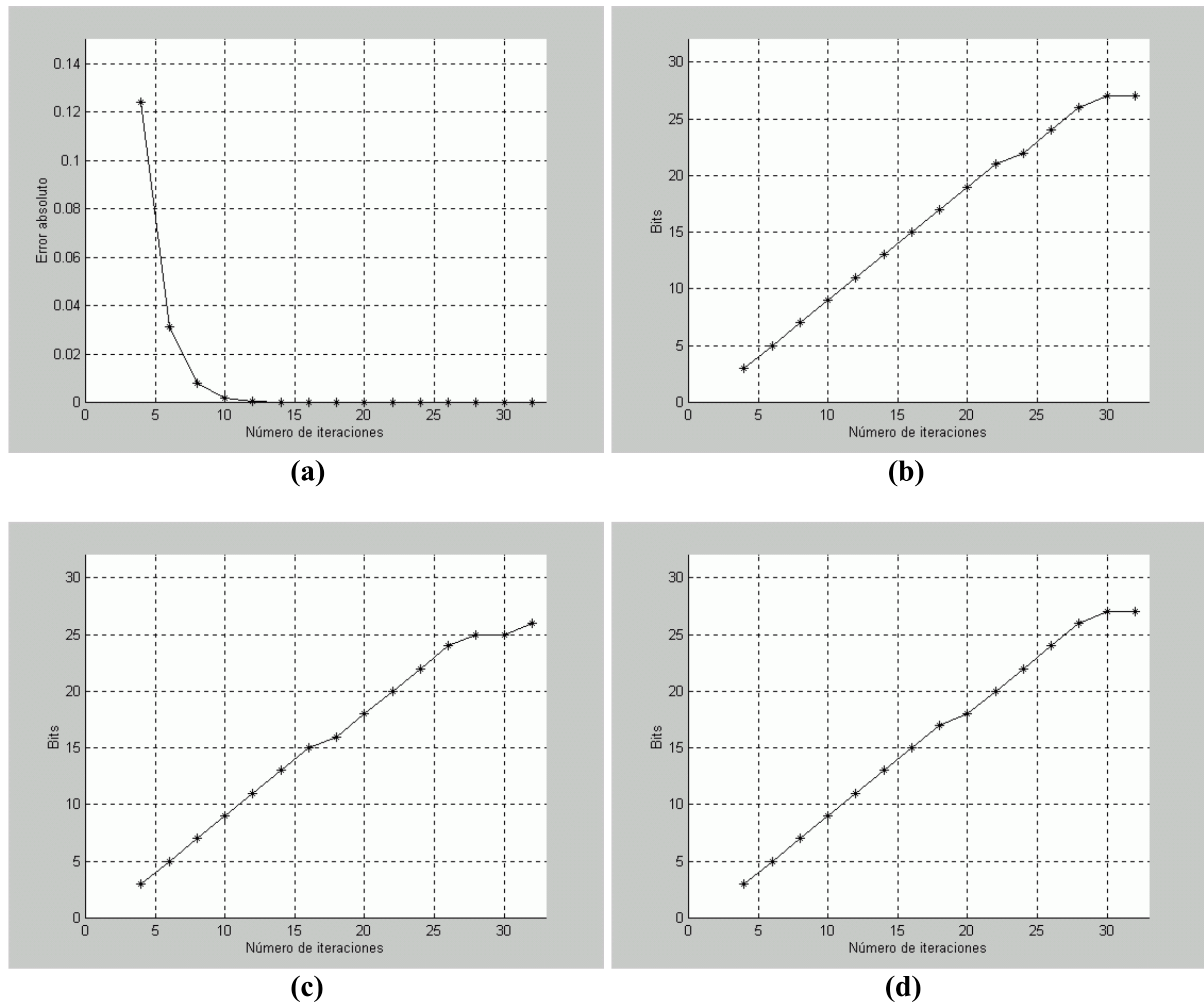
Iteraciones	Error absoluto máximo			
	Seno para ángulos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$	Coseno para ángulos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$	Arcotangente para la tangente entre $\operatorname{tg}\left(-\frac{4\pi}{9}\right)$ y $\operatorname{tg}\left(\frac{4\pi}{9}\right)$	Arcotangente para la tangente entre $\operatorname{tg}\left(-\frac{\pi}{2}\right)$ y $\operatorname{tg}\left(\frac{\pi}{2}\right)$
4	1.242654148457032e-001	1.240364158710175e-001	1.250465874394296e-001	1.244063634178296e-001
6	3.125321150525362e-002	3.125321150565330e-002	3.177330324836658e-002	3.136034472714555e-002
8	7.815711505523178e-003	7.815711505653300e-003	8.441897398442699e-003	7.987395340179893e-003
10	2.017371661503614e-003	2.057450760274235e-003	2.706708914947286e-003	2.132474795413409e-003
12	5.743289877431085e-004	5.966167681695500e-004	1.263187349130668e-003	7.587697925139913e-004
14	2.569347986164927e-004	3.038418683520838e-004	1.011584498366580e-003	3.925588550139914e-004
16	1.958996423664927e-004	2.309664794707678e-004	9.284920377063966e-004	3.053631790548294e-004

Tabla E.1

Iteraciones	D_2 mínimo		
	Seno/Coseno para ángulos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$	Arcotangente para la tangente entre $\mathrm{tg}\left(-\frac{4\pi}{9}\right)$ y $\mathrm{tg}\left(\frac{4\pi}{9}\right)$	Arcotangente para la tangente entre $\mathrm{tg}\left(-\frac{\pi}{2}\right)$ y $\mathrm{tg}\left(\frac{\pi}{2}\right)$
4	3	2	3
6	4	4	4
8	6	6	6
10	8	8	8
12	10	9	10
14	11	9	11
16	12	10	11

Tabla E.2

Los gráficos siguientes (figura E.2) muestran los valores obtenidos a partir de las tablas E.3 y E.4 con un ancho de palabra de 32 bits, entre 4 y 32 iteraciones.



(a) (b)

(c) (d)

Figura E.2

Figura E.2 (a) Error absoluto máximo para el seno, el coseno y el arcotangente.

Figura E.2 (b) D_2 mínimo para el seno y el coseno para ángulos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$.

Figura E.2 (c) D_2 mínimo para el arcotangente para la tangente entre $\text{tg}\left(-\frac{4\pi}{9}\right)$ y $\text{tg}\left(\frac{4\pi}{9}\right)$.

Figura E.2 (d) Arcotangente para la tangente entre $\text{tg}\left(-\frac{\pi}{2}\right)$ y $\text{tg}\left(\frac{\pi}{2}\right)$.

Iteraciones	Error absoluto máximo			
	Seno para ángulos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$	Coseno para ángulos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$	Arcotangente para la tangente entre $\text{tg}\left(-\frac{4\pi}{9}\right)$ y $\text{tg}\left(\frac{4\pi}{9}\right)$	Arcotangente para la tangente entre $\text{tg}\left(-\frac{\pi}{2}\right)$ y $\text{tg}\left(\frac{\pi}{2}\right)$
4	1.239941431395144e-001	1.239941440706783e-001	1.243324013065256e-001	1.243324004358084e-001
6	3.123814549996406e-002	3.123814550036374e-002	3.123942820663440e-002	3.123942634393396e-002
8	7.802870429594816e-003	7.802870429994502e-003	7.803358718109011e-003	7.803356855408573e-003
10	1.943816735883058e-003	1.943816736282744e-003	1.950195635858632e-003	1.950193773158193e-003
12	4.801866786794673e-004	4.801876100982982e-004	4.865390763477695e-004	4.865334883570327e-004
14	1.217706381252542e-004	1.217706380086322e-004	1.220401462596055e-004	1.220354288240522e-004
16	3.049261076426904e-005	3.049261090166608e-005	3.050059367224779e-005	3.049500568151099e-005
18	7.606289815668271e-006	7.605358630490694e-006	7.639537851766853e-006	7.629303226536210e-006
20	1.898674902647834e-006	1.897866252119351e-006	1.917491953329353e-006	1.908147221341139e-006
22	4.737516319380797e-007	4.738057796799255e-007	4.876745887116840e-007	4.780406609450338e-007
24	1.198464223417517e-007	1.198468229622607e-007	1.300467200593402e-007	1.213032859714858e-007
26	3.080390488519046e-008	3.250020143980014e-008	4.272921561021548e-008	3.312430174595704e-008
28	9.346526469444782e-009	1.033577046172240e-008	2.037747381944399e-008	1.120520232156252e-008
30	5.156055915556124e-009	7.057820383682411e-009	1.565326057662020e-008	6.825139675648017e-009
32	5.156055915556124e-009	6.584441936130503e-009	1.486753375967708e-008	5.184624507492685e-009

Tabla E.3

Iteraciones	D_2 mínimo		
	Seno/Coseno para ángulos entre $-\frac{\pi}{2}$ y $\frac{\pi}{2}$	Arcotangente para la tangente entre $\operatorname{tg}\left(-\frac{4\pi}{9}\right)$ y $\operatorname{tg}\left(\frac{4\pi}{9}\right)$	Arcotangente para la tangente entre $\operatorname{tg}\left(-\frac{\pi}{2}\right)$ y $\operatorname{tg}\left(\frac{\pi}{2}\right)$
4	3	3	3
6	5	5	5
8	7	7	7
10	9	9	9
12	11	11	11
14	13	13	13
16	15	15	15
18	17	16	17
20	19	18	18
22	21	20	20
24	22	22	22
26	24	24	24
28	26	25	26
30	27	25	27
32	27	26	27

Tabla E.4

Bibliografía

[1] Volder, J., “The CORDIC Trigonometric Computing Technique,” IRE Trans. Electronic Computing, Vol EC-8, pp. 330-334 September 1959.

[2] Giacomantone J.O., Villagarcía H. y Bria O. “A fast CORDIC Co-Processor Architecture for Digital Signal Processing Applications,” VI Congreso Argentino de Ciencias de la Computación (CACIC 2000). Octubre 2000 - Ushuaia - Argentina. Publicado en Anales, pág. 587 a 596.

[3] Andraka, R., “A survey of CORDIC algorithms for FPGA based computers,” Andraka Consulting Group, Inc., 1998.

[4] Edwards, B.H. and Underwood, J.M., “How Do Calculators Calculate Trigonometric Functions?,” Proceedings of the Ninth Annual International Conference on Technology in Collegiate Mathematics, Reno, Nevada, November, 1998.

[5] Walther, J.S., “A unified algorithm for elementary functions,” Spring Joint Computer Conf. 1971, Proc., pp. 379-385.

[6] Dawid, H. and Meyr, H., “CORDIC Algorithms and Architectures,” Synopsys Inc., Aachen University of Technology, 1999.

[7] Parhami, B., “Computer Arithmetic - Algorithms and Hardware Designs,” Oxford University Press, 2000.

[8] Zimmermann, R., “Computer Arithmetic: Principles, Architectures and VLSI Design,” Integrated Systems Laboratory, ETH Zürich, 1999.

[9] Yates R., “Fixed-Point Arithmetic: An Introduction,” Digital Sound Labs, 2001.

[10] Deschamps, J.P., “Diseño de Circuitos Integrados de Aplicación Específica,” Editorial Paraninfo, 1993.

[11] “FPGA Application Note: Barrel Shifter,” Atmel Corp., http://www.atmel.com, 1999.

[12] Morris Mano, M. and Kime C.R., “Fundamentos de Diseño Lógico y Computadoras,” Prentice-Hall Hispanoamericana, 1998.

[13] Nave, C.R., “HyperPhysics: Digital Electronic Circuits,” Georgia State University, http://hyperphysics.phy-astr.gsu.edu/hbase/hframe.html, 2001.

[14] Gillard, P., “CS 3724, Sequential Logic Devices and State Machines,” University of Newfoundland, Department of Computer Science, http://web.cs.mun.ca/~paul/cs3724/material/web/notes/node11.html, 1997.

[15] Li, A., “Multimedia Teaching of Introductory Digital Systems,” http://www.eelab.usyd.edu.au/digital_tutorial/part2/hpage.html, 1997.

[16] Ormerod J. "Digital Integrated Circuit Design," Bolton Institute, http://www.ami.bolton.ac.uk/courseware/dicdes/, 1998.

[17] Turner, J., "CS/EE 260, Chapter 6 - Digital Computers I," Washington University, 2000.

[18] Bhasker J., "A VHDL Primer," Prentice-Hall PTR, 1995.

[19] Villar, E., Terés, L., Olcoz, S., Torroja, Y., "VHDL Lenguaje estándar de diseño electrónico," McGraw-Hill, 1998.

[20] Pardo, F. y Boluda J., "VHDL Lenguaje para síntesis y modelado de circuitos," Alfaomega Grupo Editor, 2000.

[21] "The VHDL Scout Handbook, Third Edition," Compass Design Automation Inc., 1994.

[22] "IEEE Standard VHDL Language Reference Manual, Std. 1076-1993," IEEE 1993.

[23] Rosen, K. H., "Discrete Mathematics and its Applications," McGraw-Hill, 1988.

[24] Belton, D., "Karnaugh Maps," http://www.ee.surrey.ac.uk/Projects/Labview/minimisation/karnaugh.html, 1998.

[25] Jaquenod G.A., Villagarcía H.A. y De Giusti M.R., "Towards a Field Configurable non-homogeneous Multiprocessors Architecture," A publicar.

[26] Bria, O.N., Giacomantone J.O., "Descripciones VHDL para calcular la función recíproca en precisión simple," CeTAD, Facultad de Ingeniería, Universidad Nacional de La Plata.