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.

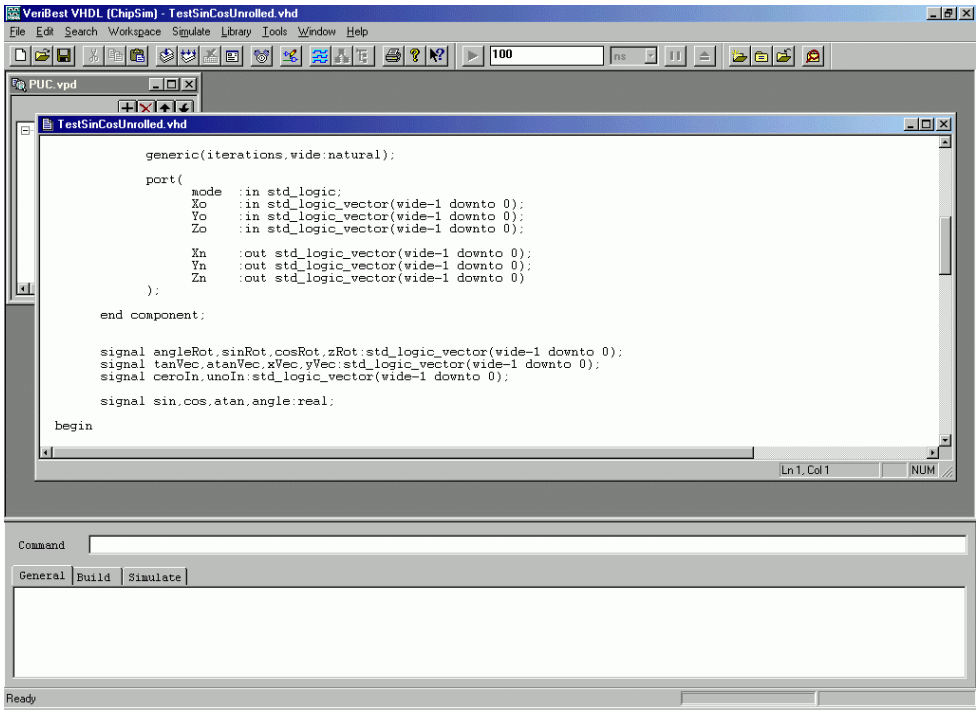

Figura 4.1

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 $-\infty < \text{tg}(\theta) < \infty$, escapando a la representación en punto fijo. El cálculo del arcotangente se limitó entre los valores para $\text{tg}(\theta)$ entre $\text{tg}\left(-\frac{4\pi}{9}\right)$ y $\text{tg}\left(\frac{4\pi}{9}\right)$ que en grados equivale a $\text{tg}(-80°)$ y $\text{tg}(80°)$.

Por lo tanto, el número de bits correspondientes a la parte entrea se debe aumentar para poder representar la $\text{tg}\left(\pm\frac{4\pi}{9}\right) \cong \pm 5{,}671281$. 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 $\pm\infty$ sustituyendo $y_0 = \pm 1$ y $x_0 = 0$. Para representar un valor *v* cercano a $\pm\infty$ 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].

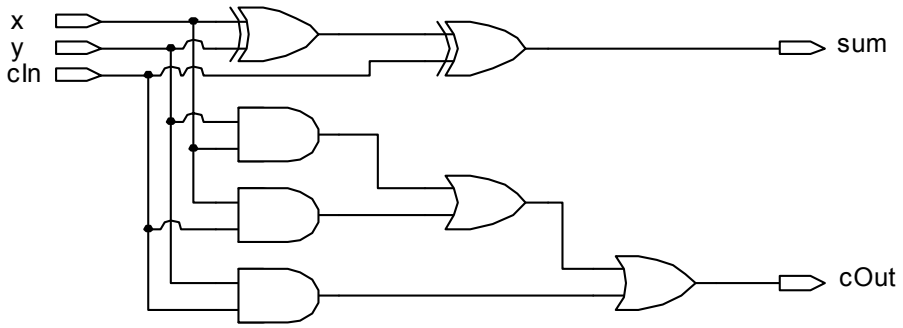

Figura 4.2

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.

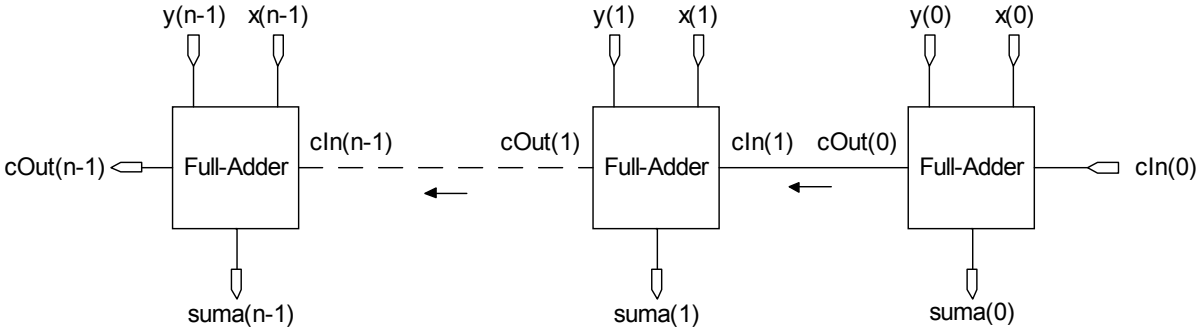

Figura 4.3

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.

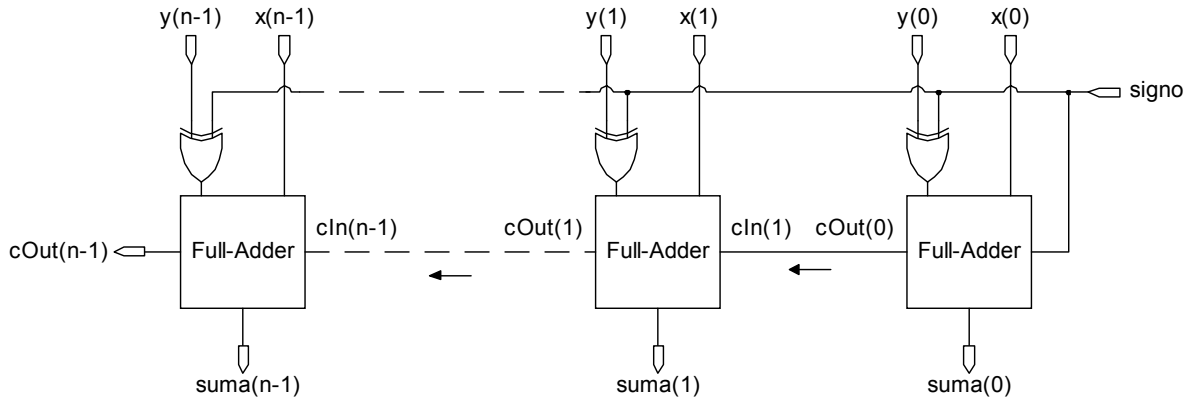

Figura 4.4

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.

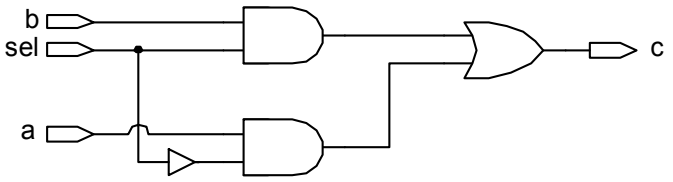

Figura 4.5

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.

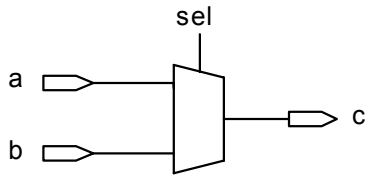

Figura 4.6

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.

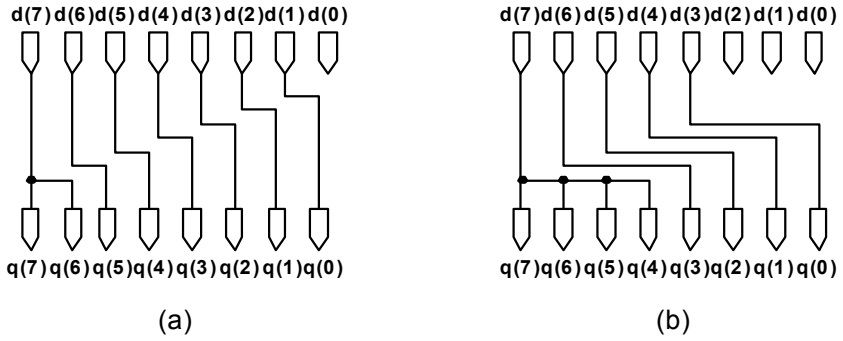

Figura 4.7

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.

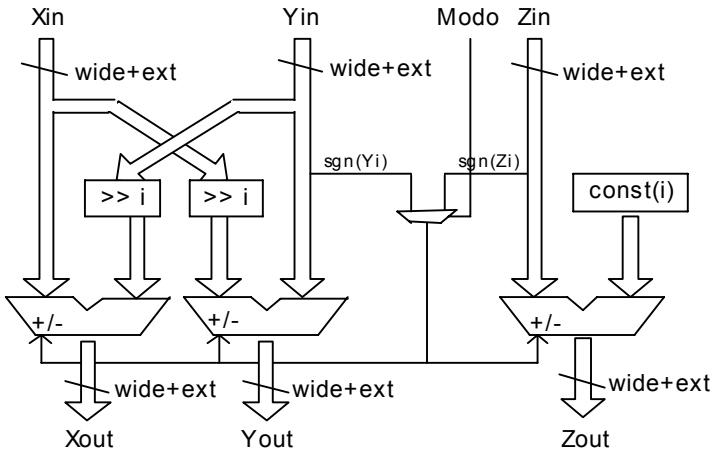

Figura 4.8

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.

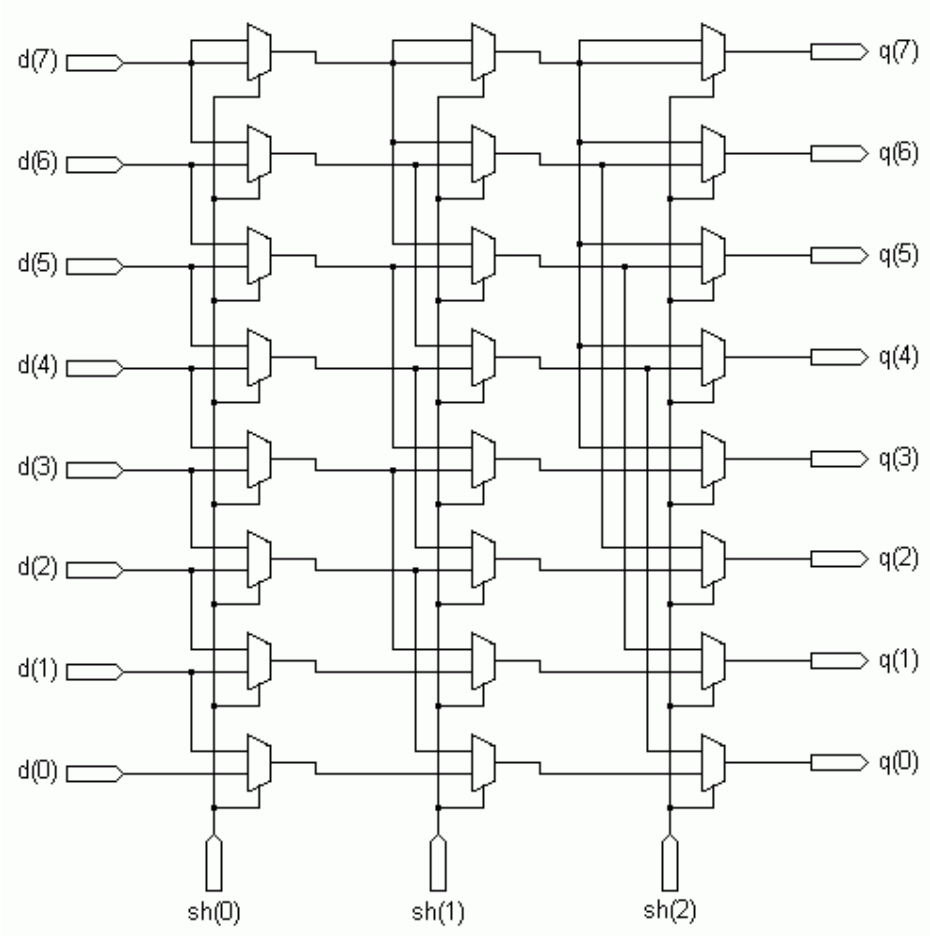

Figura 4.9

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.

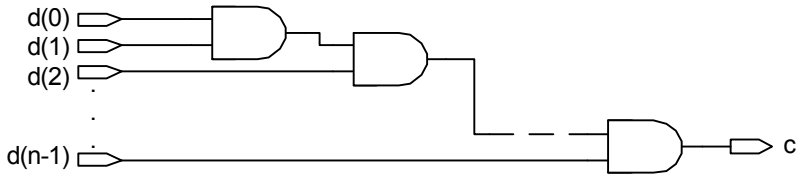

Figura 4.10

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.

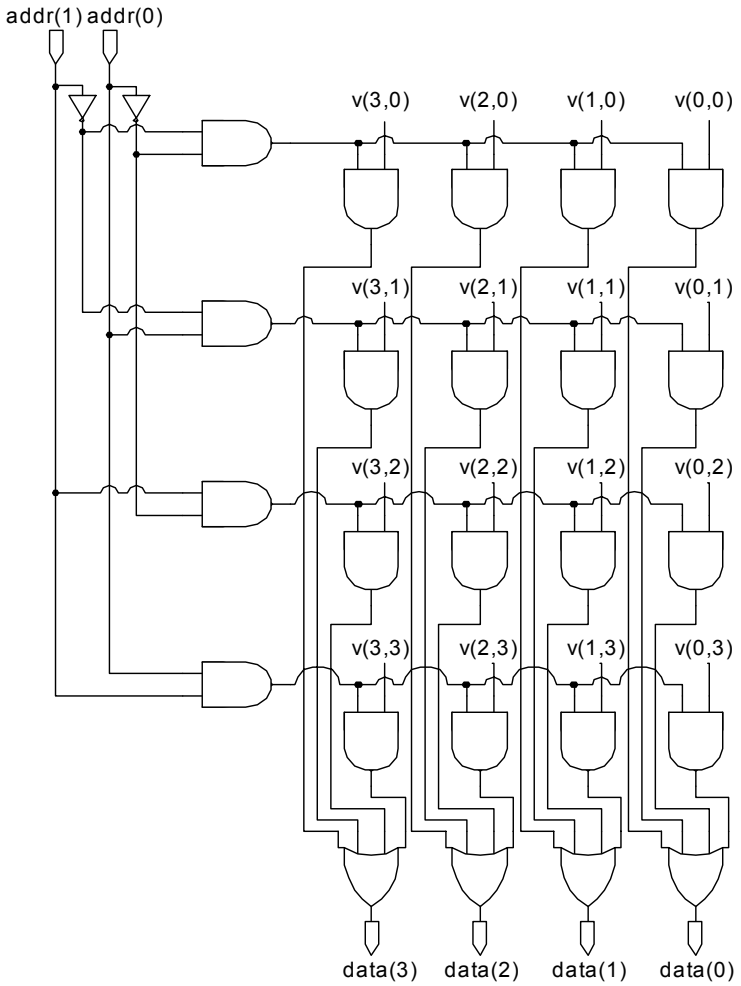

Figura 4.11

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.

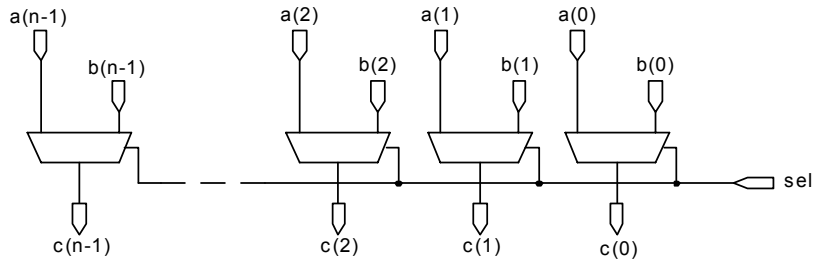

Figura 4.12

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].

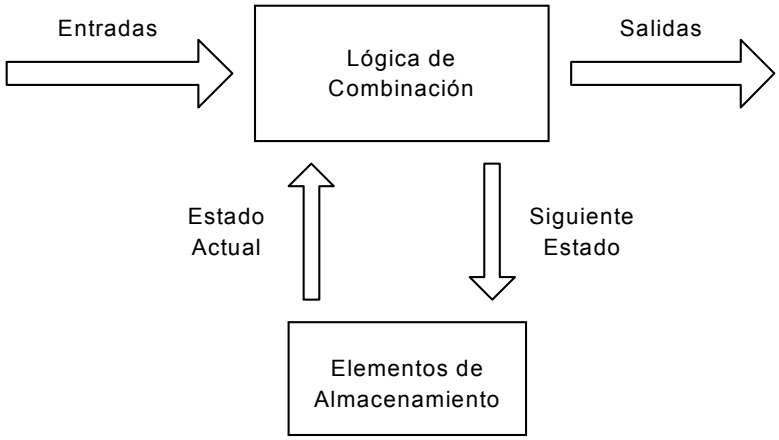

Figura 4.13

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].

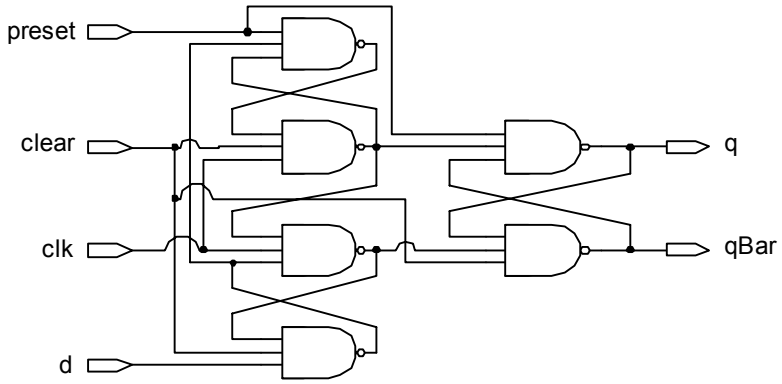

Figura 4.14

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

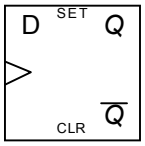

Figura 4.15

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

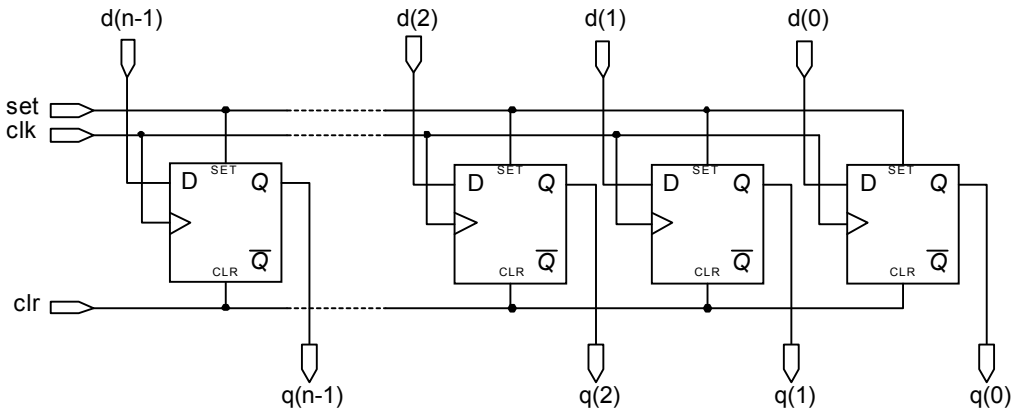

Figura 4.16

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

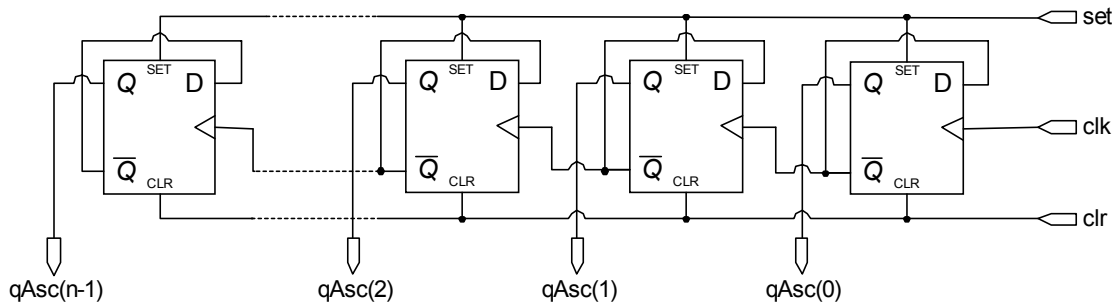

Figura 4.17

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

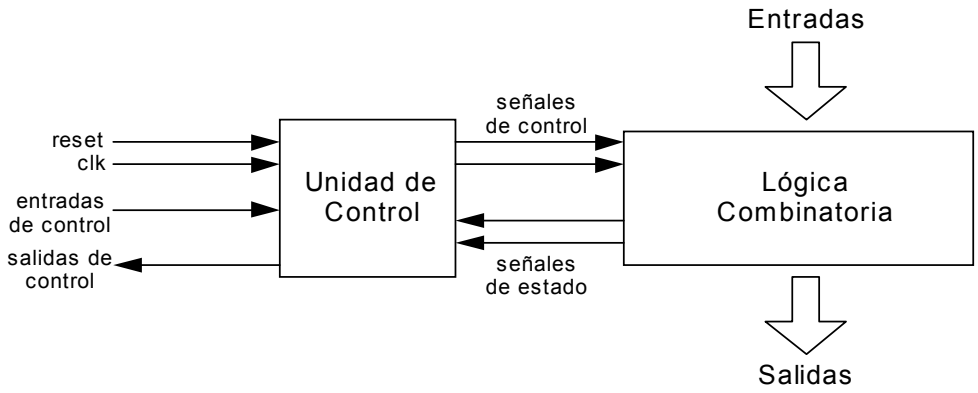

Figura 4.18

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.

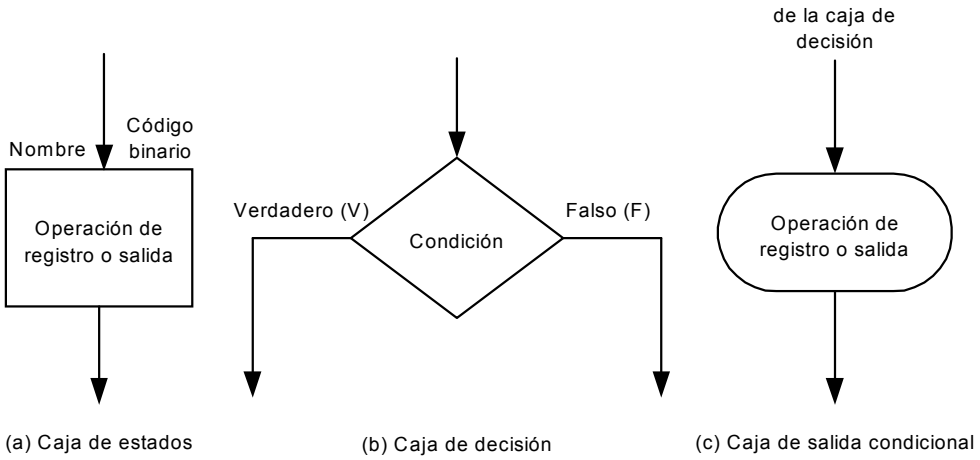

Figura 4.19

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.

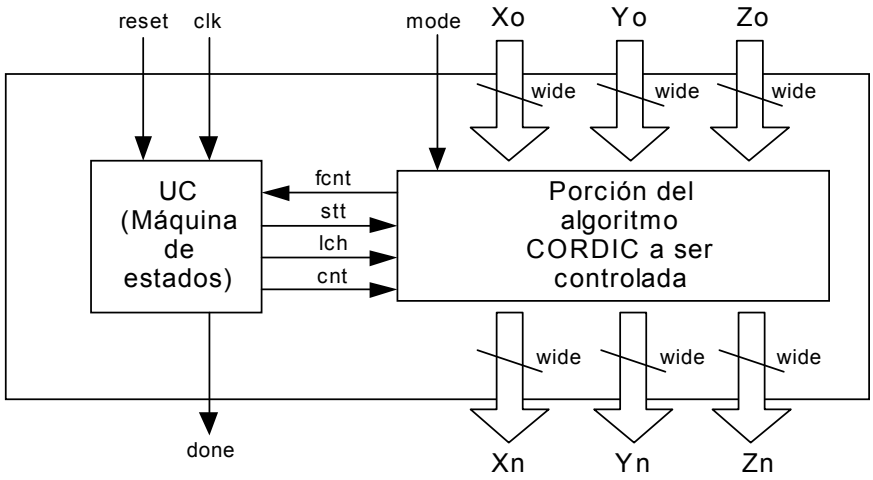

Figura 4.20

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

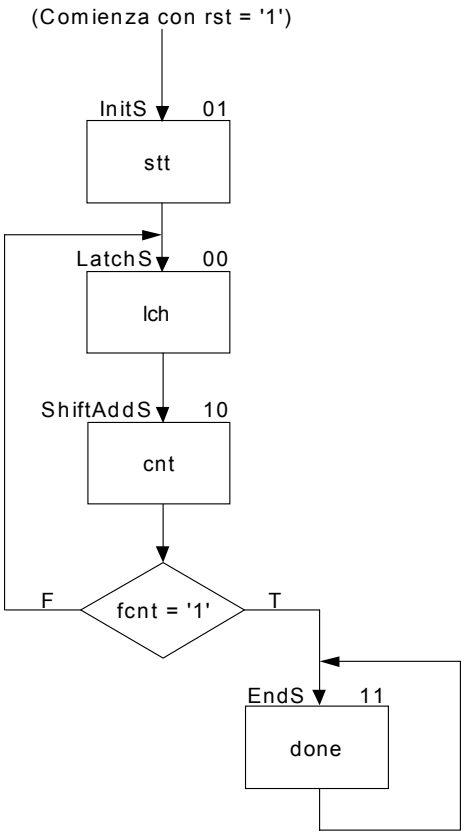

Figura 4.21

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.

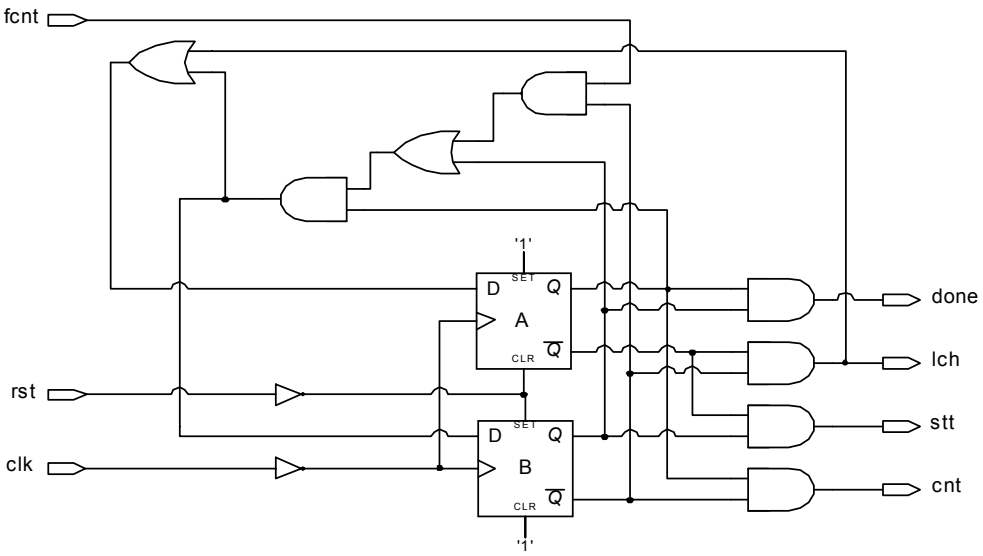

Figura 4.22

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.

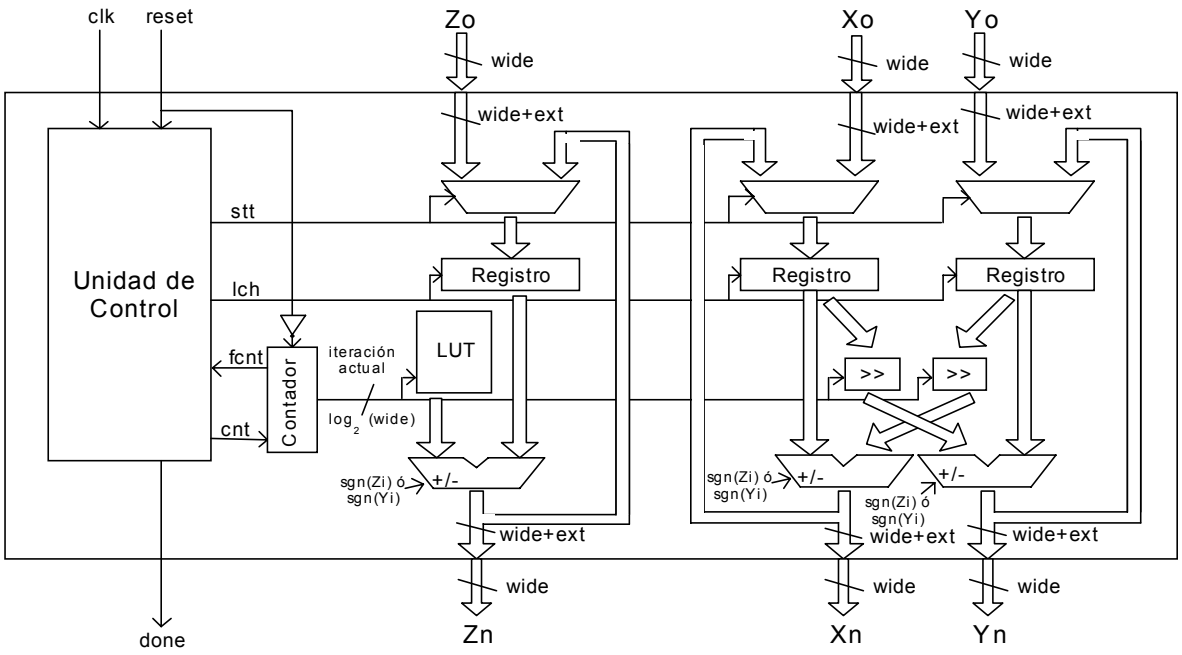

Figura 4.23

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.