Escuela
Politécnica
Superior

Desarrollo de una arquitectura hardware para procesamiento especializado

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Vladyslav Korkoshko Tereshchenko
Tutor/es:
Marcelo Saval Calvo
Jose Luís Sánchez Romero

Julio 2020

Universitat d'Alacant
Universidad de Alicante

Desarrollo de una arquitectura hardware para procesamiento especializado

Basado en CORDIC

Autor
Vladyslav Korkoshko Tereshchenko

Tutor/es
Marcelo Saval Calvo
Tecnología Informática y Computación
Jose Luís Sánchez Romero
Tecnología Informática y Computación

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2020

Abstract

Resumen

If something is hard for you to achieve
do not suppose that it is beyond human capacity rather
if something is possible and suitable for human beings
consider that it is within your reach too.

Marcus Aurelius.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

1 Introducción

1.1 Motivación y contexto

CORDIC es un método desarrollado en los años 60 que ha sido utilizado en múltiples ocasiones en proyectos de diferente gama, pero con la adición de Floating-point Unit (FPU) y sus posteriores mejoras en el procesamiento de las operaciones en punto flotante ademas de una reducción de coste de este, hizo bajar la popularidad de CORDIC ya que el procesamiento de CORDIC posee una gran latencia. Además, en software es mas lento y por lo tanto, no tan atractivo a la hora de elegir el método.

Aun así, CORDIC puede ocupar un espacio el cual una FPU no es lo mas óptimo. Múltiples métodos y algoritmos se han quedado atrás en el mundo académico y no han sido usados para solucionar problemas reales, pero CORDIC sigue siendo estudiado y después de mas de 50 años se siguen publicando artículos relacionados con el método, por lo que hay una razón por la que sería interesante estudiar sus posibles aplicaciones.

El método engloba una gran cantidad de diferentes algoritmos que se construyen de la misma manera y emplean las mismas operaciones estipuladas por Jack. E Volder. Esta flexibilidad permite el uso del algoritmo en diferentes áreas de la informática, como el procesamiento de imágenes, comunicación o robótica, entre otros.

El ecosistema de FPGAs está ahora mismo en alza gracias a la reducción de coste de los componentes y la mejora de las especificaciones. Además, se ha visto un gran número de herramientas de código abierto que anteriormente no existía, y se basaba mayoritariamente en herramientas de desarrollo software multiplataforma y dispositivos hardware propietarios. Un claro ejemplo son las *crowdfunded* FPGAs que hay en el mercado, que ofrecen unas características muy interesantes a un precio relativamente bajo.

Referencias a algunas de estas FPGAs.

Esto permite poner vista a nuevos proyectos con FPGAs que anteriormente eran muy costosos ya que se ve una clara bajada de coste de cómputo en un futuro muy cercano.

Para finalizar, CORDIC permite un aprendizaje de los lenguajes de descripción de hardware y la implementación del código en hardware especializado, como por ejemplo una FPGA, ya que es un método fácil de entender e implementar en hardware en su definición mas básica (cálculo de funciones sin() y cos()).

1.2 Método CORDIC

En 1959, Volder publica el primer artículo del algoritmo que denominó CORDIC. Este algoritmo fue originalmente como un algoritmo de propósito específico para dispositivos digitales que necesitan un funcionamiento en tiempo real. En concreto, este algoritmo fue diseñado para el cómputo del sistema de navegación del avión Convair B-56 Hustler para reemplazar el sistema analógico a uno digital para ser mas preciso y de tiempo real.

El algoritmo publicado por Volder tenía como objetivo resolver funciones trigonométricas y la conversión de coordenadas rectangulares a polares, aunque desde el primer momento se estipula que este método puede ser usado para para el cómputo de otras operaciones matemáticas.

El aspecto más importante de CORDIC es la forma de resolver el problema. Solo requiere las operaciones de sumar, restar, movimiento de bits (*bitshift*) y una *lookup table* (LUT). Esto es importante, ya que trae muchas características que en un futuro dependen de esta simplicidad.

1.2.1 Revisiones y mejoras de CORDIC

Walther (1971), de la división de *Hewlet-Packard Laboratories* (HP) publicó en 1971 un artículo con un algoritmo CORDIC que unificaba el cálculo de funciones elementales como la multiplicación, división, exponenciales, raíces cuadradas, entre otros, con las misma operaciones básicas especificadas anteriormente.

Además, en este artículo ya se hace mención de una unidad de hardware para el proceso de valores en punto flotante. El dispositivo diseñado en HP aceptaba valores de 48 bits y 32 bits en punto flotante.

El algoritmo de CORDIC fue usado por HP, Texas Instruments y otros para crear calculadoras digitales de un tamaño reducido. Un ejemplo de esto es HP-35, la cual realizaba las funciones trigonométricas mediante este método. Además, CORDIC fue usado en algunos procesadores, ya sea como un *coprocesador* o integrado dentro del mismo, como el Intel 8087 y algunos de sus posteriores generaciones, generalmente para reducir el número de puertas lógicas y complejidad de la FPU.

1.2.2 Presente y futuro de CORDIC

Poner mas cosas aquí. Citas

Aunque el algoritmo no sea la mejor método para realizar las operaciones estipuladas anteriormente, sigue siendo muy atractivo por su simplicidad a la hora de implementarlo en hardware, ya que se puede usar el mismo algoritmo iterativo para todas las operaciones usando el diseño *shift-add* básico. Muchos sistemas embebidos y FPGAs no tienen una unidad dedicada al procesamiento de punto flotante, por lo que lo hace un candidato ideal.

2016 Study of CORDIC backing up development and reduced cost of FPGA and improved performance

Los nuevos desarrollos de CORDIC se han centrado en mejorar el *throughput*, la reducción de la complejidad del hardware necesitado para la implementación y la latencia propia del método.

La mayoría que ahora mismo son solo de 50 Years of CORDIC. Poner las actualizadas tmb.

Algunas de las aplicaciones(usos?) de CORDIC son:

- *Direct Kinematics Solution* (DKS) o solución de cinemática directa para manipuladores de robots seriales. Las rotaciones y translaciones de los eslabones(links?) que generan nuevas coordenadas son calculadas por CORDIC. Un método similar es usado para la cinemática inversa.

- CORDIC tambien se ha usado como una unidad funcional de un procesador de robot para temas de redundancia y cálculo de detección de colisión.

- Dentro de los gráficos 3D nos encontramos con procesamiento intensivo geométrico de rotación de vectores 3D, luminosidad y interpolación, candidatos perfectos para uso del método.
- Descomposición QR. Hay variantes de CORDIC que ofrecen esta funcionalidad.
- Dentro del procesamiento de señal podemos encontrar un gran número de implementaciones para el cálculo de *Discrete Fourier transform* (DFT).

  Hay mas cosas que poner aquí de DFT

- En el ámbito de la comunicación, CORDIC puede ser usado para la modulación digital y análoga. Dependiendo de los parámetros del método, es posible hacer cálculo digital de una multitud de modulaciones.

1.3 Objetivos

El objetivo principal es el estudio y desarrollo de alternativas al procesamiento de funciones hiperbólicas y trigonométricas de punto flotante según el estándar del IEEE 754. En concreto se centra el algoritmo CORDIC (COordinate Rotation DIgital Computer).

Se pretende encontrar ventajas e inconvenientes del algoritmo, soluciones propuestas y realizar una simulación e implementación de un algoritmo CORDIC básico en hardware utilizando numeración en punto flotante.

Como se verá en los siguientes apartados, la complejidad de algunas soluciones a los problemas inherentes del funcionamiento de CORDIC hace que el alcance del desarrollo del algoritmo sea simplemente una demostración teórica y práctica de como implementar un algoritmo de este tipo, dentro del marco académico.

Por último, hay que destacar que esta memoria(?) se centra principalmente en el aspecto del tratamiento de los datos

1.4 Estructura del documento

2 CORDIC

2.1 Descripción de CORDIC

Revisar de nuevo si las formulas estan correctas. Comentar Linear/Circular/Hyperbolic

En esta sección se describirá el funcionamiento principal del algoritmo CORDIC según descrito por Jack E. Volder.

La rotación de un vector de dos dimensiones $\boldsymbol{p_0} = [x_0, y_0]$ con un ángulo θ para obtener un vector $\boldsymbol{p_n} = [x_n, y_n]$ se puede realizar con un producto matriz $\boldsymbol{p_n} = \boldsymbol{R_{p0}}$, donde $\mathbf{R}$ es la matriz de rotación.

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Si sacamos el coseno de la matriz de rotación, se puede obtener un valor K, el cual se convierte en una constante.

$$\mathbf{R} = [(1 + \tan^2\theta)^{-1/2}] \begin{bmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{bmatrix}$$

Nombraremos como $K = [(1 + \tan^2\theta)^{-1/2}]$ el factor escala. Eliminando K obtenemos una matriz de pseudo-rotación $\boldsymbol{R_c}$ tal que

$$\boldsymbol{R_c} = \begin{bmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{bmatrix}$$

La operación pseudo-rotación realizada sobre el vector p_0 por un ángulo θ y cambia su magnitud por un factor $K = \cos\theta$.

Para obtener la simplicidad dentro del hardware necesitamos:

- Descomponer las rotaciones en una secuencia de rotaciones elementales con ángulos predefinidos que se pueda implementar con un coste hardware mínimo.
- Eliminar el factor escala K, ya sea finalizando la operación con una simple multiplicación o ignorarlo directamente.

En primer lugar, CORDIC realiza una rotación iterativa con una lista de ángulos predefinidos $\alpha_i = \arctan(2^{-i})$ de manera que $\tan(\alpha_i) = 2^{-i}$ se puede implementar en hardware como un desplazamiento de i posiciones.

Ya que estamos limitando la operación $\tan\theta$ a una lista de rangos predefinidos anteriormente podemos asumir que

$$K_i = \frac{1}{sqrt(1 + 2^{-2_i})}$$

El factor escala K_i altera la magnitud del vector a rotar independientemente del valor del ángulo.

Ya que el factor escala no depende del ángulo de las micro-rotaciones, no necesitamos incluir el factor dentro de las operaciones, lo que nos da $p_n = R_c p_0$.

El factor final K tiene un valor de 1.6467605, por lo cual simplemente se puede escalar el valor final por K.

Los valores finales que nos interesan son los siguientes:

$$x_{i+1} = x_i - \delta_i \times 2^{-i} \times y_i$$
$$y_{i+1} = y_i - \delta_i \times 2^{-i} \times x_i$$
$$\omega_{i+1} = \omega_i - \delta_i \times \alpha_i$$

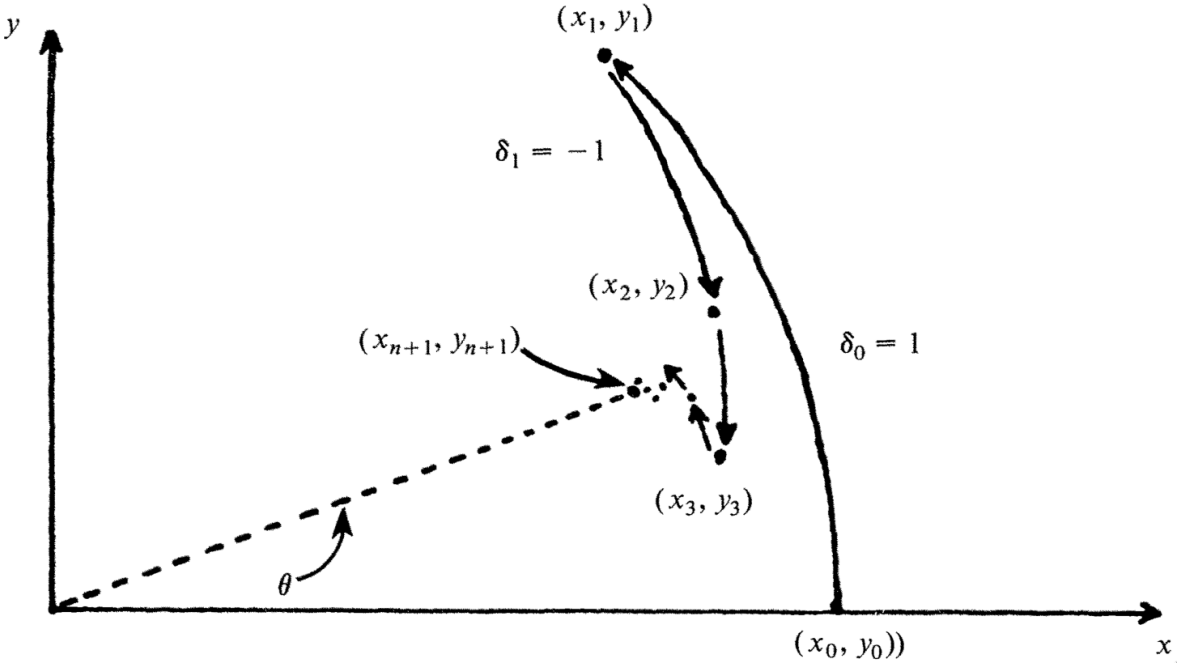

Figura 2.1: Rotation Mode. Figura de Schelin (1983)

CORDIC puede operar de dos formas: *Rotation Mode* (RM) (2.1) y *Vectoring Mode* (VM) (2.2). La principal diferencia es en como se eligen las micro-rotaciones. En RM, la dirección de cada micro-rotación depende del signo de ω_i. Si ω_i es positivo $\delta_i = 1$, si no $\delta_i = -1$. En VM, el vector se rota hacia el eje de x, por lo que la componente y tiende a 0.

Posteriormente, J.S. Walther propuso un CORDIC generalizado unificado para realizar multitud de operaciones matemáticas, pero esto se encuentra fuera del alcance de esta memoria (Vea 2.1).

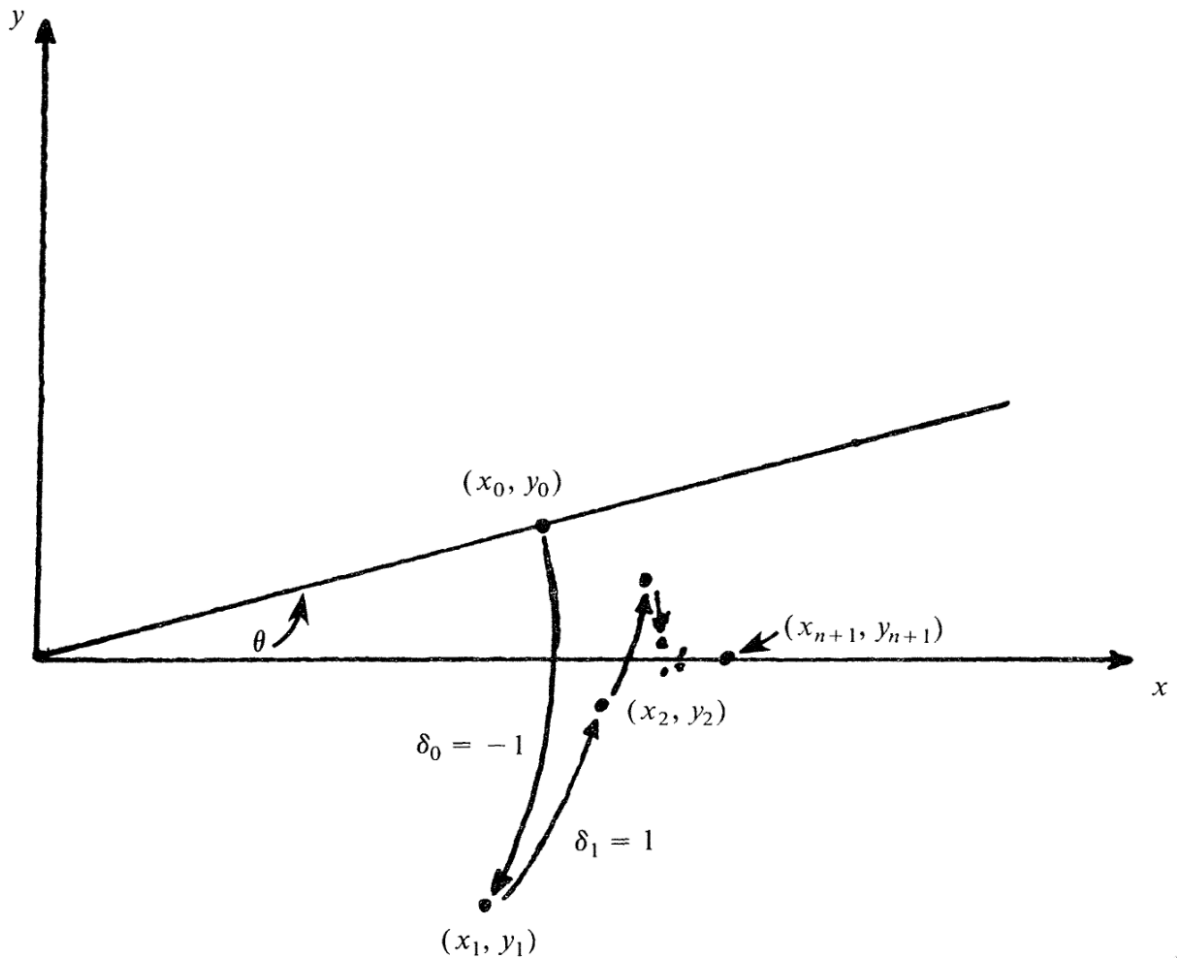

Figura 2.2: Vectoring Mode. Figura de Schelin (1983)

2.2 Mejoras a CORDIC

En el apartado de introducción se explicaron algunas aplicaciones de CORDIC, sin detallar el tipo o las modificaciones realizadas. En este apartado se muestran algunas de estas mejoras importantes que han permitido a CORDIC mantenerse competitivo. Muchas de estas soluciones están descritas en Meher y cols. (2009).

Como ya se comentó anteriormente, el cómputo de CORDIC es originalmente un proceso totalmente secuencial por dos razones principales: Las micro-rotaciones depende de los valores la rotación computados anteriormente, lo que sería el valor intermedio y las iteración $(i + 1)$ solo puede comenzar después de que se complete la iteración i.

Otro de los problemas es que la precisión del valor final es lineal, requiere $n + 1$ iteraciones para tener una precisión de n bits. Esto significa que la latencia depende del número de bits que se pasan a CORDIC.

Operación	Configuración	Inicialización	Salida	Anotaciones
$\cos\theta, \cos\theta, \tan\theta$	CC-RM	$x_0 = 1$ $y_0 = 0$ y $\omega_0 = \theta$	$x_= \cos\theta$ $y_n = \sin\theta$	$\tan\theta = (\sin\theta/\cos\theta)$
$\cosh\theta, \sinh\theta, \tanh\theta, \exp(\theta)$	HC-RM	$x_0 = 1$ $y_0 = 0$ y $\omega_0 = \theta$	$x_n = \cos\theta$ $y_n = \sinh\theta$	$x_n = \sqrt{a}$ $\omega_n = 1/2\ln(a)$
$\ln(a), \sqrt{a}$	HC-VM	$x_0 = a+1$ $y_0 = a-1$ y $\omega_0 = 0$	$x_n = \sqrt{a}$ $\omega_n = 1/2\ln a$	$\ln(a) = 2\omega_n$
$\arctan(a)$	CC-VM	$x_0 = a$ $y_0 = 1$ y $\omega_0 = 0$	$\omega_n = \arctan(a)$	
división(b/a)	LC-VM	$x_0 = a$ $y_0 = b$ y $\omega_0 = 0$	$\omega_n = b/a$	
Polar a rectangular	CC-RM	$x_0 = R$ $y_0 = 0$ y $\omega_0 = \theta$	$x_n = R\cos\theta$ $y_n = R\sin\theta$	
Rectangular a polar $\tan^- 1(b/a)$ y $\sqrt{a^2+b^2}$	CC-VM	$x_0 = a$ $y_0 = b$ y $\omega_0 = 0$	$x_n = \sqrt{a^2+b^2}$ $\omega_n = \arctan(b/a)$	

Tabla 2.1: Diferentes configuraciones de CORDIC, segun Meher y cols. (2009)

2.2.1 4-Radix

Una solución a la reducción del número de iteraciones es calcular CORDIC con una base mayor, como por ejemplo 4. La fórmula final seria tal que:

$$x_{i+1} = x_i - \delta_i \times 4^{-i} \times y_i$$
$$y_{i+1} = y_i - \delta_i \times 4^{-i} \times x_i$$
$$\omega_{i+1} = \omega_i - \delta_i \times \alpha_i$$

El valor K tendría una fórmula:

$$K_i = \frac{1}{sqrt(1 + \delta_i^2 * 4^{-2_i})}$$

De esta manera, para tener una salida de n bits de precisión solo necesitamos $n/2$ micro-rotaciones a cambio de mas complejidad dentro del hardware. Un punto a comentar es que el factor de escala K ahora depende de δ, el cual puede tener 5 valores diferentes.

Finalmente se quiere comentar que también existen soluciones de base mas alto, como 8-Radix.

2.2.2 Angle Recording

Los métodos de *Angle Recording* (AR) quieren reducir el número de iteraciones mediante una combinación lineal de ángulos de las micro-rotaciones. Este método es ideal para procesamiento de señal, imagen, donde el ángulo de rotación se conoce *a priori*.

2.2.3 Hybrid CORDIC

CORDIC híbrido se basa en la idea de que los valores menos significativos del cálculo de ángulo pueden ser reemplazados por 2^-j ya que $\tan(2^-j) \approx 2^-j$ si el valor j es lo suficientemente grande. De esta manera podemos calcular los valores mas pequeños con un simple movimiento de bits y no necesitamos conocer el signo de rotación ya que en todo momento es positivo.

2.2.4 Redundant-Number-Based CORDIC

Este tipo de CORDIC pretende mejorar el rendimiento de las sumas y restas mediante un uso de numeración redundante para no tener que realizar el cálculo de los posibles restos que puede obtener una operación aritmética. La desventaja de este método es que necesitas mas espacio en hardware para implementar.

2.2.5 Pipelined CORDIC

El uso de *pipelined* CORDIC es bastante extenso ya que cada iteración del método es idéntica y los valores anteriores después de su cálculo ya no nos interesan, es posible obtener por cada ciclo de ejecución un nuevo resultado. Para esto necesitamos tener un registro por cada etapa de CORDIC (vea 2.3).

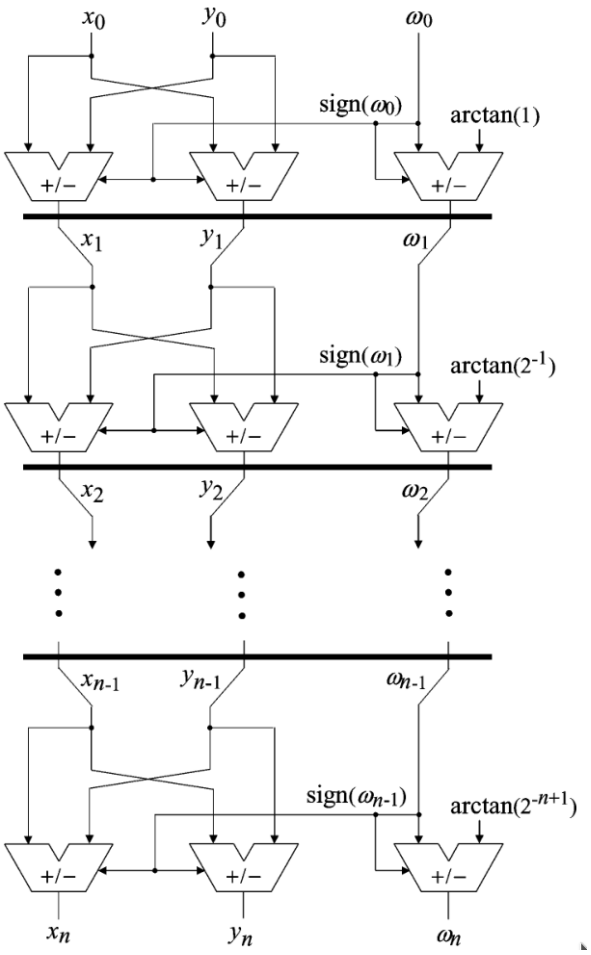

Figura 2.3: CORDIC convencional con *pipelining*

2.3 CORDIC y punto flotante

Parker (2011) Muestra claramente algunos de los problemas que conlleva implementar el estándar del IEEE 754 en hardware. En concreto estos problemas son relaciones con algunos aspectos de las FPGAs, pero puede ser aplicable a cualquier problema donde el hardware es limitado. Algunos de estos problemas son:

- La representación de la mantisa incluye un 1 implícito. El valor real va desde [1:1.999..] en vez de [0:0.000..].

Precisión	Rango	Throughput	Latencia	Transistores
16-bits	$\pm 2^{16}$	100ns	2.2 μs	70000
Entradas	Salidas	Consumo	Espacio	Procesador
72	67	1.3W	1.2 cm^2	CMOS(2μ)

Tabla 2.2: Características de FP CORDIC. Datos de de Lange y cols. (1988)

- En vez de usar complemento a dos, el estándar incluye un bit de signo.
- Cada operación aritmética conlleva una normalización de la mantisa para alinear el punto decimal hacia la izquierda, además de tener que ajustar el exponente de forma acorde.
- VHDL y Verilog no tienen una implementación de operaciones con punto flotante. Un lenguaje de descripción de hardware mas nuevo, como SystemVerilog si tiene estas operaciones, pero la complejidad de ellas conlleva a perder mucho espacio, el cual puede estar bastante limitado en las FPGAs.

Dentro de la evolución del uso de punto flotante en el método hay una variedad importante de arquitecturas y algoritmos de CORDIC.

Una de las primeras implementaciones de un tipo de punto flotante, explicado por Leibson (2005) fue sobre el año 1954 con la calculadora de sobremesa HP 9100A. La calculadora tenía 16 registros, numerados de forma hexadecimal y podía guardar valores de punto flotante con 10 dígitos de mantisa y 2 dígitos de exponente en formato BCD. La mantisa y el exponente podían guardar valores positivos y negativos.

Según Walther (1971), la primera implementación de punto flotante tenía grandes limitaciones en el tema de hardware a la hora de convertir los valores entre BCD a base 2 y, además, no había un estándar de punto flotante en aquel entonces, por lo que cada diseñador creaba su propio formato.

Un diseño de CORDIC que mas se asemeja a las nuevas propuestas es el de de Lange y cols. (1988), donde proponen un procesador de CORDIC de punto flotante y con *pipelining* . Este chip podía realizar hasta 10^7 rotaciones por segundo gracias al sistema de *pipelining*. La entrada de valores era de 21 bits en punto flotante, 16 bits de mantisa y 5 bits para el exponente en complemento a dos. La salida era también en punto flotante (vea 2.4). Cabe destacar que el tipo usado en esta arquitectura no es estándar de IEEE.

Para realizar las micro-rotaciones estipuladas anteriormente, la base del propio algoritmo, el procesador transforma el valor de punto flotante en punto fijo para el trabajo interno y posteriormente devuelve los valores en punto flotante. El valor K es calculado en el momento de la conversión final del valor para devolver.

Como se puede observar, operar con punto fijo para realizar movimiento de bits es mucho mas fácil que tener que hacerlo directamente con punto flotante, por lo que los diseñadores generalmente, en muchos de los ejemplos mostrados posteriormente, realizan una conversión de punto fijo a punto flotante, o algunas veces directamente reciben valores en punto fijo para simplemente convertir una sola vez a punto flotante al final del algoritmo.

Hekstra y Deprettere (1993) presentaron un algoritmo de CORDIC de 32 bit de precisión con el estándar de IEEE 754. Este algoritmo puede realizar la rotación de un vector punto

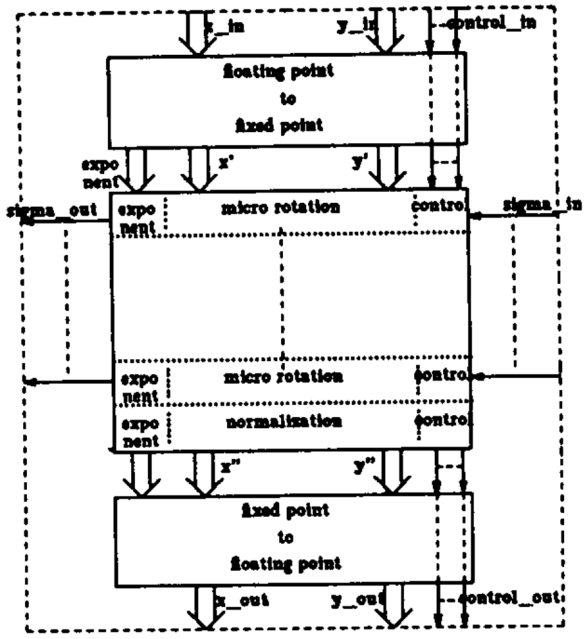

Figura 2.4: Arquitectura de FP CORDIC. Figura de de Lange y cols. (1988)

flotante (x, y) con un ángulo de punto flotante α. Este algoritmo fue diseñado con la intención de implementarlo en una unidad funcional para aplicaciones de procesamiento digital (DSP).

Para realizar las micro-rotaciones se da ciertos cambios, como el uso del método *Block Floating Point* (BFP), el cual permite un uso de aritmética con punto fijo aunque el valor sea punto flotante. La ventaja de este método es la reducción de hardware y el coste de tiempo que puede traer las mismas funciones aritméticas comparadas a punto flotante pero sin perder el rango numérico que le da la ventaja a este.

Zhou y cols. (2008) diseñaron un co-procesador FPGA basado en CORDIC con doble precisión (64 bits estándar IEEE 754) y *pipelining*. Este diseño se centra explícitamente en las FPGAs, ya que como se puede ver anteriormente, y según lo descrito en este artículo, la mayoría se centraba dentro del área del ASIC, por lo que no había tantos ejemplos de implementaciones en hardware para FPGAs. Además, es de los pocos artículos con una implementación de 64 bits.

El diseño se basaba en tres fases: Transformación de los valores de punto flotante a punto fijo, método CORDIC y una fase de normalización, donde se devuelve el valor en punto flotante del IEEE 754.

Los resultados finales en este artículo muestran un *speedup* considerable comparando a una CPU de la época, en concreto la AMD Athlon 64 Processor 3200+ (vea 2.3). Otros experimentos mostraban la reducción de espacio en el hardware comparados a otras soluciones y un error de resultados razonable.

Nguyen y cols. (2015) propone un CORDIC con punto flotante de baja latencia. El algoritmo propuesto reduce el número de iteraciones eligiendo un grupo particular de constantes para conseguir un resultado cercano al real.

La reducción del número de ángulos a escoger se basa en elegir los ángulos hasta llegar a

Programa	AMD Athlon (μs)	Mult-CORDICs					
		Num_C	Num_M	Num_D	Num_AS	Tiempo(μs)	Speedup
Problema 1	342.86	3	4	1	2	6.95	49.3
Problema 2	129.39	2	8	0	7	7.04	18.4
Problema 3	125.43	1	2	1	0	6.52	19.2
Media	199.23	2	4.7	0.67	3	6.84	28.7

Tabla 2.3: Resultados de 64FP CORDIC y una CPU AMD Athlon 64 Processor 3200+. Num_C es el número de co-procesadores de CORDIC usados en el experimento.

							Diseño de este artículo
Dispositivo	Xilinx Vertex 5	Xilinx Vertex 5	Xilinx Vertex 7	Altera Stratix II	Xilinx Vertex 6	Altera Stratix IV	Altera Stratix IV
Latencia (ciclos)	93	-	130	-	-	36	12/20/26
Frecuencia	86.1	133.8	280	195.1	253.5	258.3	175.7
LUTs	3152	26811	6514	6469	13744	5612	1139
Registros	-	16274	4725	5372	-	4231	498
Memoria	2832	-	4894	-	-	3575	11
DSP	0	2	9	-	96	32	8

Tabla 2.4: Tabla de comparaciones del CORDIC propuesto con otros trabajos parecidos. Tabla de Nguyen y cols. (2015)

un umbral particular para tener un error de cálculo muy cercano a como si se hiciera el de todas las constantes. Un punto a tener en cuenta es que el valor K tendrá un valor diferente en cada operación del algoritmo, por lo que se tiene que recalcular en cada operación.

En cuanto a las entradas y salidas del algoritmo, la entrada es un valor del ángulo de punto fijo de 24 bits y las salidas son el sin/cos con un valor de 32 bits IEEE 754 cada una.

2.3.1 Artículos mas recientes

Ahora se va a mostrar los artículos mas actualizados de CORDIC. Estos artículos muestran un interés general por el método y da confianza sobre el futuro de CORDIC.

Hou y cols. (2019) diseñan un algoritmo CORDIC usando el estándar IEEE 754 de doble precisión (64 bits).

La metodología del tratamiento de los datos es muy parecida a la literatura descrita anteriormente, la entrada es un valor de punto flotante que en un módulo de pre-procesamiento realiza una conversión a punto fijo y además se procesa las excepciones que podrían aparecer en este momento, como por ejemplo un NaN.

La unidad de procesamiento de CORDIC utiliza el llamado *Point 4-step Iterative Processing Unit*, el cual ocupa mas espacio que un CORDIC tradicional, pero logra calcular por cada ciclo 16 micro-rotaciones de CORDIC, reduciendo efectivamente el tiempo por 4 (vea 2.5). El número de iteraciones es menor a otros métodos de CORDIC, como por ejemplo Para-CORDIC (vea Juang y cols. (2004)).

Como se comentó anteriormente, CORDIC puede realizar una multitud de operaciones

Tipo de CORDIC	Tradicional	Para-CORDIC	Hybrid	SF CORDIC	Artículo mencionado
Número de iteraciones	64	17	24	24	15

Tabla 2.5: Número de iteraciones comparando el multiplicador CORDIC con punto flotante de Yeshwanth y cols. (2018) a otras soluciones.

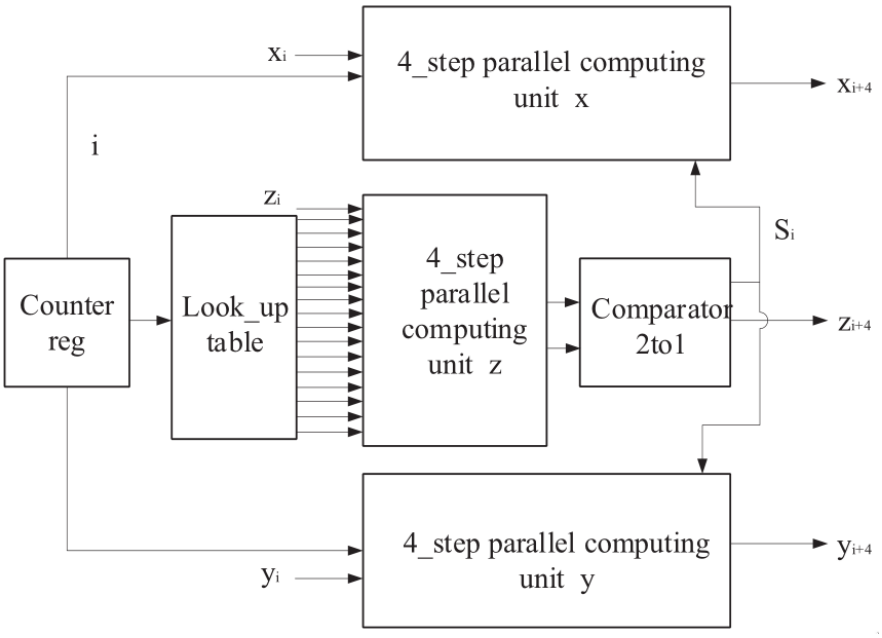

Figura 2.5: Unidad de procesamiento de CORDIC de doble precisión. Figura extraída de Yeshwanth y cols. (2018)

Multiplicador punto flotante	**Parámetros digitales**		
	Latencia (ns)	Área (**LUTs**)	**Consumo (mW)**
Multiplicador CORDIC	5.259	679	96
Multiplicador Vedic	26.634	489	95

Tabla 2.6: Comparativa entre un multiplicador Vedic y CORDIC de Yeshwanth y cols. (2018).

matemáticas según la necesidad del problema. Yeshwanth y cols. (2018) aprovechan el algoritmo de CORDIC con *pipelining* para optimizar el rendimiento del multiplicador de las mantisas para el IEEE 754 de 32 bits. Los resultados obtenidos han sido comparados al método de Vedic. Los tiempos de CORDIC son considerablemente mejores, pero ocupa un área en hardware mayor que Vedic.

En el área de la robótica, Evangelista y cols. (2018)

Dentro del trabajo de los radares de apertura sintética (*Synthetic Aperture Radar*) (SAR), Fang y cols. (2019) proponen una mejora de los tiempos de cálculo de operaciones trigonométricas, raíces, multiplicación.

El artículo sigue las pautas de muchos otros en diseñar un módulo de pre-procesamiento y post-procesamiento para la conversión de punto flotante a punto fijo y viceversa, con ciertos ajustes para no perder mucha precisión. El diseño fue implementado en una FPGA para mostrar la eficiencia de uso de espacio de hardware y precisión de los resultados.

Wang y cols. (2020) presentan una implementación hardware basado en CORDIC generalizado hiperbólico (GH CORDIC) para el cálculo de raíces cuadradas de base n en punto

flotante de precisión simple. Se utilizan múltiples CORDIC en diferentes configuraciones.

El cálculo de la raíz en base n se realiza totalmente en punto flotante, sin ningún tipo de conversión a punto fijo. Las operaciones realizadas en el diseño son mas complejas a raíz de operar con el estándar del IEEE. Además, se usa *pipelining* para mejorar la latencia. El cálculo final son 58 ciclos para una palabra de 32 bits.

Los resultados finales del artículo muestran una mejora de *throughput*, eficiencia energética y mejor precisión de los resultados.

3 Implementación de CORDIC

3.0.1 Herramientas

3.0.1.1 Linux

Todo el trabajo de implementación ha sido realizado dentro de una máquina Linux. Generalmente las herramientas para diseño de nivel hardware son propietarias y desarrolladas para Windows.

Ya que se tiene un buen conocimiento de Linux, se ha optado por buscar herramientas alternativas de código abierto que puedan ofrecer un desarrollo similar a las propietarias.

3.0.1.2 Verilog

El lenguaje de descripción de hardware (HDL) escogido ha sido Verilog/SystemVerilog. Verilog es un subconjunto del lenguaje mas nuevo SystemVerilog. Aunque no se hayan utilizado ninguna nueva parte del diseño, se eligió desde un principio para tener posibilidad de ampliación o necesidad de alguna nueva parte en un futuro.

Ya que en la carrera se realizó prácticas con VHDL, se decidió por usar un lenguaje diferente para obtener un amplio conocimiento de los lenguajes principales. Además, Verilog tiene una similitud con el lenguaje C, por lo que puede ser mas atractivo si se tiene algún conocimiento de este lenguaje.

3.0.1.3 Verilator

La herramienta elegida para la compilación y comprobación del funcionamiento de CORDIC ha sido Verilator. Esta herramienta es gratuita y de código abierto. Además esta desarrollada para Ubuntu, pero funciona en casi todas las distribuciones de Linux.

Su principales características son la compilación del código de Verilog a C++ o SystemC. Este código es mucho mas optimizado y permite un mejor rendimiento a la hora de simular. Además, permite diseñar *testbench* en C++, por lo que puede facilitar mucho la forma de desarrollar si se tiene un conocimiento de C++.

Aunque no se vaya a usar en gran parte las ventajas que ofrece Verilator, podemos sacar de provecho para dar una entrada a futuros proyectos.

3.0.2 Configuración de Verilator

Para compilar nuestro código Verilog/SystemVerilog necesitaremos usar el comando *verilator*. Un ejemplo básico sería el siguiente:

```
verilator -Wall --cc cordic.sv
```

Después de ejecutar el comando, Verilator crea una carpeta llamada *obj_dir*, donde podemos encontrar un fichero Makefile, que en este caso su nombre sería *Vcordic.mk*. Este makefile crea el ejecutable final.

Para poder implementar los testbench en C++, se debe de inicializar Verilator con las variables de *argc* y *argv*. Para poder acceder a las entradas y salidas de Verilog necesitamos crear una instancia de nuestro módulo a probar. En este caso seríá un tipo *Vcordic*, el cual esta incluido en "obj_dir/Vcordic.h"

Código 3.1: Definición de una TestBench de Verilator

```
class TestBench{
    unsigned long m_tickcount;
    Vcordic *m_core;
    VerilatedVcdC *m_trace;

    TestBench(void){
        m_core = new Vcordic;
        m_tickcount = 0l;
        Verilated::traceEverOn(true);
    }

   ...
};
```

Dentro de la definición del *TestBench* tenemos un tipo *Vcordic**, el cual permitirá acceder a las variables del módulo. El tipo *m_tickcount* permite tener un tiempo de referencia interno en nuestra *testbench*.

La última línea del constructor es una llamada a Verilator para poder llamar a cualquier función para trazar las señales del módulo para luego ser usadas en un programa como *GTKWave*.

Para poder trazar las señales necesitamos definir dos funciones: *opentrace()* y *close()*.

Código 3.2: Definición de funciones para trazar la simulación.

```
virtual void opentrace(const char *name) {
    m_trace = new VerilatedVcdC;
    m_core->trace(m_trace, 99);
    m_trace->open(name);

}

virtual void close(void) {
    m_trace->close();
    delete m_trace;
    m_trace = NULL;
}
```

Estas dos funciones se usarán dentro de la función *main()* antes y después de pasar por el bucle principal.

Ahora necesitamos crear las funciones de *tick*() y *reset*().

Código 3.3: Definición de funciones *reset()* y *tick()*

```
virtual void reset(void){

   m_core->iReset = 1;
   this->tick();
   m_core->iReset = 0;
}

virtual void tick(void){
   m_tickcount++;

   m_core->iClock = 0;
   m_core->eval();

   if(m_trace)
      m_trace->dump(10*m_tickcount-2);

   m_core->iClock = 1;
   m_core->eval();

   if(m_trace)
      m_trace->dump(10*m_tickcount);


   m_core->iClock = 0;
   m_core->eval();

   if (m_trace) {
      m_trace->dump(10*m_tickcount+5);
      m_trace->flush();
   }

}
```

La función *tick*() incrementa el tiempo de referencia interno, evalúa cualquier lógica combinatoria con una función interna de Verilator, *eval*(), ya que es posible de que el valor haya cambiado antes de haber sido llamado *tick*(). Posteriormente cambia el reloj *iClock* a 1 y evalúa el estado y vuelve a cambiar el reloj a 0.

Además, la función *tick()* tiene unas llamadas al trazado de las señales si el usuario lo ha inicializado con las llamadas apropiadas anteriormente.

La función reset() cambia la entrada de *iReset* a 1, evalúa el módulo con un *tick*() y vuelve a cambiar la entrada a 0.

A partir de ahora, solo necesitamos tener un bucle en nuestra función main para probar el módulo.

3.0.3 Implementación básica

Ya que CORDIC tiene la ventaja de ser muy fácil de implementar en hardware, ya sea en términos de latencia o espacio, una implementación básica de CORDIC no es compleja de diseñar.

Las entradas de nuestro módulo $iX, iY, iAngle, iClock$ y $iReset$. Las salidas son oX y oY. Inicialmente todos las entradas y salidas son enteros de 32 bits.

Código 3.4: Módulo CORDIC

```
module cordic(iClock, iReset, iAngle, iX, iY, oX, oY);
```

Antes de iniciar CORDIC, necesitamos conocer en que cuadrante se encuentra el ángulo de entrada. Cada artículo citado ha tomado una decisión propia a la hora de optimizar el algoritmo. Una manera fácil de saber el cuadrante en el que se encuentra es hacer el módulo de $2 * pi$ radianes y empaquetando el valor del ángulo en un registro de 32 bits. En este caso, 0 grados se traduce a un valor de todos los bits a 0 y un valor de 359.999... sería todo los bits a 1, aunque en nuestro caso solo hay valores enteros. Esta transformación tiene que ser realizada por el usuario antes de pasarla por la entrada $iAngle$. Aunque esto es posible implementarlo dentro del módulo o uno externo, por simplificar se ha decidido no hacerlo para centrarse en CORDIC.

Ahora solo tenemos que leer los 2 bits mas significativos para saber en que cuadrante se encuentra el ángulo. Este paso solo se ejecuta al principio del algoritmo.

Código 3.5: Pre-rotación de iAngle pasado por el usuario.

```
begin
    case(quadrant)
        2'b00,
        2'b11:
        begin
            X <= iX;
            Y <= iY;
            Z <= iAngle;
        end

        2'b01:
        begin
            X <= -iY;
            Y <= iX;
            Z <= {2'b00,iAngle[29:0]};
        end

        2'b10:
        begin
            X <= iY;
            Y <= -iX;
            Z <= {2'b11,iAngle[29:0]};
        end
    endcase
end
```

La LUT de ángulos predefinidos de $\arctan(2^{-i})$ tiene el mismo formato que $iAngle$.

Código 3.6: Tabla de arctan.

```
wire signed [31:0] atan_angle [0:30]

```

```
assign atan_angle[00] = 32'h20000000; // arctan(2^0)
assign atan_angle[01] = 32'h12E4051D;
assign atan_angle[02] = 32'h9FB385B;
...
assign atan_angle[28] = 32'h2;
assign atan_angle[29] = 32'h1;
assign atan_angle[30] = 32'h0;
```

Se ha definido un total de 31 valores, aunque como se verá mas tarde no van a ser necesarios todos los valores de la tabla para conseguir un valor cercano al real.

Ahora definimos 3 registros y 3 *wire*.

Código 3.7: Registros y *wire* para operar con CORDIC

```
reg signed [IOWidth-1:0] X;
reg signed [IOWidth-1:0] Y;
reg signed [31:0] Z;

wire signed [31:0] X_shr = X >>> i-1;
wire signed [31:0] Y_shr = Y >>> i-1;
wire Z_sign = Z[31];
```

Estos registros nos servirán para guardar los resultados de cada micro-rotación de CORDIC y los *wire* guardan la operación de movimiento de bits de X e Y y el signo de Z.

Código 3.8: Bucle principal de CORDIC

```
begin
    X <= Z_sign ? X + Y_shr : X - Y_shr;
    Y <= Z_sign ? Y - X_shr : Y + X_shr;
    Z <= Z_sign ? Z + atan_angle[i-1] : Z - atan_angle[i-1];
end
```

Este sería el código representando las fórmulas finales que se han obtenido a la hora de despejar la función **R**.

3.0.3.1 Testbench de CORDIC

Para probar el funcionamiento, utilizaremos la *testbench* definida anteriormente. Dentro de nuestro fichero "main.cpp" necesitamos inicializar Verilator con la siguiente línea:

```
Verilated::commandArgs(argc,argv);
```

Despues de incluir el fichero "TestBench.h" en el "main.cpp", debemos de crear un tipo *TestBench* y cargar nuestras con los valores desdeados.

Código 3.9: Creación del tipo TestBench y entrada de valores desde C++ al módulo de CORDIC en Verilog

```
TestBench *tb = new TestBench();

const float K = 1.646760;
int32_t angle = atoi(argv[3]);
int32_t X = atoi(argv[1]);
int32_t Y = atoi(argv[2]);
```

```
7
8  tb->m_core->iX = X / K;
9  tb->m_core->iY = Y / K;
10 tb->m_core->iAngle = (pow(2,32)*angle)/360;
```

Después de crear el objeto *TestBench*, introducimos los valores de X e Y dividiéndolo antes por el factor K. El ángulo es transformado a un valor módulo de 360 y elevado a 2^{32} para que tenga el mismo formato que la tabla de arctan que tenemos predefinida en nuestro código Verilog.

El factor K es extraído de los valores a pasar antes de la ejecución. Dependiendo de las especificaciones esto se podría dejar con el factor K si no fuera necesario.

Finalmente, iniciamos la traza y el bucle para ir ciclo por ciclo en nuestra ejecución.

Código 3.10: Bucle principal de main.cpp

```
1 tb->opentrace("test.vcd");
2 for(int x=0;x<NUM_TICKS;x++)
3 {
4     tb->tick();
5     ...
6 }
7 tb->close();
```

Este *testbench* es prácticamente idéntico en todas las implementaciones que se van a realizar, por lo que solo se va a comentar en la sección de CORDIC básico.

3.0.4 Implementación *pipeline*

El *pipelining* es un método clásico de conectar elementos de procesamiento en series para mejorar el rendimiento de los sistemas. Muchos artículos de CORDIC proponen un sistema con algún tipo de *pipelining*. En este punto se mostrará un ejemplo de *pipelining* básico que se puede implementar con el código implementado en el punto anterior.

La operación de micro-rotación que realiza CORDIC puede ser diseñada de forma que los valores se muevan en serie por todas las etapas del método. Para esto necesitamos el mismo número de registros que etapas para guardar los valores intermedios.

Código 3.11: Registros de CORDIC ampliados para *pipelining*

```
1 localparam ETAPAS = 16;
2
3 reg signed [IOWidth:0] X [0:ETAPAS-1];
4 reg signed [IOWidth:0] Y [0:ETAPAS-1];
5 reg signed [31:0] Z [0:ETAPAS-1];
```

Verilog tiene un constructor llamado *generate*. Sus funciones principales son la instanciación de los *items* del módulo, cambio de la estructura del diseño a partir de los parámetros pasados por Verilog y verificación funcional de los módulos. En nuestro caso lo usaremos para generar todas las etapas de ejecución del método CORDIC.

Los bloques de *generate* son generados antes de la ejecución del circuito, ya que no es posible añadir o eliminar hardware en tiempo de ejecución.

Para generar los bloques necesitamos una variable *genvar*. Además, todos los *wire* que asignan el tipo de signo del ángulo y los movimientos de bits son también generados dentro de *generate*.

Código 3.12: Ejecución principal de CORDIC con *pipelining*

```
genvar i;
generate
for (i = 0;i < (ETAPAS-1);i=i+1)
begin
    wire Z_sign;
    wire signed [IOWidth:0] X_shr, Y_shr;

    assign X_shr = X[i] >>> i;
    assign Y_shr = Y[i] >>> i;

    assign Z_sign = Z[i][31];

    always @(posedge iClock)
    begin
        // add/subtract shifted data
        X[i+1] <= Z_sign ? X[i] + Y_shr : X[i] - Y_shr;
        Y[i+1] <= Z_sign ? Y[i] - X_shr : Y[i] + X_shr;
        Z[i+1] <= Z_sign ? Z[i] + atan_angle[i] : Z[i] - atan_angle[i];
    end
end
endgenerate
```

Como se puede observar, el espacio que ocupa un CORDIC con *pipelining* es mayor que el básico, pero tenemos la ventaja de una mejora del *throughput*. A partir del ciclo 16 estamos recibiendo cada ciclo un nuevo resultado.

Ya que esta implementaciones no han sido probadas en hardware real y solamente simuladas, hay algunos aspectos que no se tienen en cuenta que si pueden ocurrir en un sistema real, como por ejemplo en una FPGA.

- No hay una forma de comprobar que los resultados obtenidos son válidos. Esto viene del hecho de que el módulo va a estar ejecutándose continuamente cada ciclo sin tener en cuenta si su entrada es válida o no. Una forma de arreglar esto es mediante una entrada *Clock Enable* (CE), la cual esta conectada a todos las etapas y es la única que permite mover los datos de un etapa a otra.
- Es posible que el dispositivo al que estamos transmitiendo no esta preparado para recibir los datos. La solución a este problema es tener conectada una entrada de *BUSY*, que dependiendo del tipo de transmisor que sea tendrá una especificación u otra. Para que se pueda transmitir datos de forma correcta, *BUSY* tendría un valor 0 y *Clock Enable* un valor 1.

3.0.5 Implementación con punto flotante

Para la implementación de punto flotante necesitamos realizar algunas modificaciones al tipo de datos que estamos tratando. Las entradas iX y iY eran enteros con signo de 32 bits.

En esta implementación necesitamos usar un nuevo formato de punto fijo, en concreto el formato Q.

El formato Q(m,n) nos permite operar con el mismo hardware que un entero, por lo que reduce la complejidad de operar directamente con el estándar IEEE 754. En concreto, el formato es Q(8,24), con 8 bits de entero y 24 bits fraccionarios.

En el código de "main.cpp", tenemos que modificar la línea de entrada de valores iX y iY para que estén acorde al formato Q(8.24). Para ello necesitamos elevar el valor que pasamos por argumento a 2^{24}.

Código 3.13: Cambios de entrada de valores de C++ a Verilog en la implementación de Punto Flotante

```
...
tb->m_core->iX = (X * pow(2,24)) / K;
tb->m_core->iY = (Y * pow(2,24)) / K;
...
```

Dentro del código de Verilog, el bucle principal de micro-rotaciones y el movimiento de bits se queda igual que en las otras implementaciones. La división en formato fijo se puede realizar si el divisor es 2^i, por lo que podemos hacer el movimiento de bits necesario por CORDIC sin problemas.

A la hora de obtener el resultado necesitamos transformar el valor en el estándar del IEEE 754. Para ello se ha diseñado una máquina de estado finita (FSM).

Una forma muy general de diseñar una Finite State Machine (FSM) es mediante dos registros: *current_state* y *next_state*. Después, un parámetro es declarado por cada estado que se encuentre en el diagrama de estados. Los parámetros necesitan un valor en concreto para poder usar su nombre declarado en las asignaciones siguientes.

Código 3.14: Registros para controlar los estádos y parámetros locales de estado para controlar la máquina de estado finita.

```
reg [2:0] current_state = w_normal_op, next_state;

localparam w_normal_op = 3'b000,
             w_check_sign = 3'b001,
             w_bit_shift_count = 3'b010,
             w_to_fp = 3'b011,
             w_reset = 3'b100;
```

Aquí podemos ver una declaración de 2 registros mencionados anteriormente con 3 bits cada uno, para poder codificar un máximo de 2^3 estados diferentes. Al crear los parámetros tenemos que asignarles un valor de 3 bits.

Primero se va a comentar una idea general de cada estado y luego se entrará en los detalles del funcionamiento. El modo de operación de la FSM es la siguiente:

- *w_normal_op* es el modo de operación normal de CORDIC con *pipelining*. La FSM se encuentra en este modo hasta encontrar con un nuevo valor en el último registro de nuestra *pipeline*, la cual se refiere al valor final calculado por glscordic.
- *w_check_sign* es un estado intermedio que transforma los valores negativos de complemento a 2 a un valor binario normal. Antes de convertirlo se guarda el signo, que viene representado por el último bit (31).

- *w_bit_shift_count* calcula el movimiento de bits que se debe de realizar para alinear los bits con la mantisa.
- *w_to_fp* realiza la transformación final a punto flotante y reinicia todos los valores intermedios que se han usado para operar correctamente.
- *w_reset* reinicia el módulo poniendo todos los registros, entradas y salidas a 0.

Una FSM tiene los llamados *state memory*, *next state logic* y *output logic*.
El primer estado de memoria (*state memory*) es el que se dedica simplemente a actualizar el estado cada vez que hay una subida de reloj.

Código 3.15: *state memory* de CORDIC

```
always@(posedge iClock or negedge iReset)
 begin: STATE_MEMORY
    if(iReset == 1'b1)
        current_state <= w_reset;
    else
        current_state <= next_state;
 end
```

El siguiente bloque es el *next state logic*. Este bloque solo es activado por el registro *current_state*, lo que convierte esta FSM una máquina de Moore

```
always@(current_state)
 begin: NEXT_STATE_LOGIC
    case(current_state)
        default:
        begin
        end

        w_normal_op:
            begin
                if(last_x_value != X[ETAPAS-1] || last_y_value != Y[ETAPAS-1] ↩
                    ↪ begin
                    next_state = w_check_sign;
                end
            end

        w_check_sign:
            if(sign_checked) begin
                next_state = w_bit_shift_count;
            end
        w_bit_shift_count:
            if(tempXOut[bitmov_x] == 1 && tempYOut[bitmov_y] == 1 ) begin
                next_state = w_to_fp;
            end
        w_to_fp:
            if(fp_out) begin
                next_state = w_normal_op;
            end
```

```
    endcase
end
```

```
always @(current_state)
begin: OUTPUT_LOGIC
 case(current_state)
    w_normal_op,

    w_check_sign:
    begin
        fp_out = 0;
        fpX_out = 0;
        fpY_out = 0;

        tempXOut = X[ETAPAS-1];
        sign_x = tempXOut[31];

        tempYOut = Y[ETAPAS-1];
        sign_y = tempYOut[31];


        if(tempXOut[31] == 1)
        begin
            tempXOut = ~(tempXOut-1);
        end


        if(tempYOut[31] == 1)
        begin
            tempYOut = ~(tempYOut-1);
        end

        sign_checked = 1;
    end

    w_bit_shift_count:
    begin
        sign_checked = 0;
        if(tempXOut[bitmov_x] != 1 )
        begin
            bitcount_x = bitcount_x + 1;
            bitmov_x = bitmov_x - 1;
        end
        if(tempYOut[bitmov_y] != 1)
        begin
            bitcount_y = bitcount_y + 1;
            bitmov_y = bitmov_y - 1;
        end
    end

```

```
48      w_to_fp:
49      begin
50          if(!fp_out)
51          begin
52
53              $display("before %b and %d",tempXOut,tempXOut);
54              if(tempXOut[30:23] != 0) begin
55                  $display("oki value %d",(8-bitcount_x));
56                  tempXOut = tempXOut >> (8-bitcount_x);
57              end
58              else begin
59                  $display("outting %d and %b",(8-bitcount_x),~((8-bitcount_x)-1)↩
                        ↪ );
60                  tempXOut = tempXOut << ~((8-bitcount_x)-1);
61              end
62
63              $display("after %b and %d",tempXOut,tempXOut);
64
65              tempXOut[30:23] = (127 + (8 - (bitcount_x+1)));
66              tempXOut[31] = sign_x;
67              fpX_out = tempXOut;
68              bitmov_x = 31;
69              bitcount_x = 0;
70
71              if(tempYOut[30:23] != 0) begin
72                  tempYOut = tempYOut >> (8 - bitcount_y);
73              end
74              else begin
75                  tempYOut = tempYOut << ~((8 - bitcount_y)-1);
76              end
77
78              tempYOut[30:23] = (127+(8 - (bitcount_y+1)));
79              tempYOut[31] = sign_y;
80              fpY_out = tempYOut;
81              bitmov_y = 31;
82              bitcount_y = 0;
83              fp_out = 1;
84          end
85      end
86      default:
87          begin
88          end
89
90  endcase
91
92    end
```

4 Conclusiones

Bibliografía

de Lange, A., van der Hoeven, A., Deprettere, E., y Bu, J. (1988, junio). An optimal floating-point pipeline CMOS CORDIC processor. En *1988., IEEE International Symposium on Circuits and Systems* (pp. 2043–2047 vol.3). doi: 10.1109/ISCAS.1988.15343

Dhume, N., y Srinivasakannan, R. (2012). Parameterizable CORDIC-Based Floating-Point Library Operations. , 18.

Evangelista, G., Olaya, C., y Rodríguez, E. (2018, agosto). Fully-pipelined CORDIC-based FPGA Realization for a 3-DOF Hexapod-Leg Inverse Kinematics Calculation. En *2018 WRC Symposium on Advanced Robotics and Automation (WRC SARA)* (pp. 237–242). doi: 10.1109/WRC-SARA.2018.8584238

Fang, L., Xie, Y., Li, B., y Chen, H. (2019). Generation scheme of chirp scaling phase functions based on floating-point CORDIC processor. *The Journal of Engineering*, *2019*(21), 7436–7439. (Conference Name: The Journal of Engineering) doi: 10.1049/joe.2019.0623

Hekstra, G., y Deprettere, E. (1993, junio). Floating point Cordic. En *Proceedings of IEEE 11th Symposium on Computer Arithmetic* (pp. 130–137). doi: 10.1109/ARITH.1993.378100

Hou, N., Wang, M., Zou, X., y Liu, M. (2019, julio). A Low Latency Floating Point CORDIC Algorithm for Sin/Cosine Function. En *2019 IEEE 4th International Conference on Signal and Image Processing (ICSIP)* (pp. 751–755). (ISSN: null) doi: 10.1109/SIPROCESS.2019.8868623

Intro - Verilator - Veripool. (s.f.). Descargado 2020-07-16, de `https://www.veripool.org/wiki/verilator`

Juang, T.-B., Hsiao, S.-F., y Tsai, M.-Y. (2004, agosto). Para-CORDIC: parallel CORDIC rotation algorithm. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *51*(8), 1515–1524. (Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers) doi: 10.1109/TCSI.2004.832734

LaMeres, B. J. (2019a). Computer System Design. En B. J. LaMeres (Ed.), *Quick Start Guide to Verilog* (pp. 143–186). Cham: Springer International Publishing. Descargado 2020-07-27, de `https://doi.org/10.1007/978-3-030-10552-5_11` doi: 10.1007/978-3-030-10552-5_11

LaMeres, B. J. (2019b). Modeling Finite State Machines. En B. J. LaMeres (Ed.), *Quick Start Guide to Verilog* (pp. 113–127). Cham: Springer International Publishing. Descargado 2020-07-27, de `https://doi.org/10.1007/978-3-030-10552-5_8` doi: 10.1007/978-3-030-10552-5_8

LaMeres, B. J. (2019c). Verilog Constructs. En B. J. LaMeres (Ed.), *Quick Start Guide to Verilog* (pp. 13–22). Cham: Springer International Publishing. Descargado 2020-07-27, de https://doi.org/10.1007/978-3-030-10552-5_2 doi: 10.1007/978-3-030-10552-5_2

Leibson, S. (2005). *The 9100 Part 2.* Descargado 2020-07-02, de http://www.hp9825.com/html/the_9100_part_2.html

Manual-verilator - Verilator - Veripool. (s.f.). Descargado 2020-07-23, de https://www.veripool.org/projects/verilator/wiki/Manual-verilator

Meher, P. K., Valls, J., Juang, T.-B., Sridharan, K., y Maharatna, K. (2009, septiembre). 50 Years of CORDIC: Algorithms, Architectures, and Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *56*(9), 1893–1907. (Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers) doi: 10.1109/TCSI.2009.2025803

Nguyen, H.-T., Nguyen, X.-T., Hoang, T.-T., Le, D.-H., y Pham, C.-K. (2015). Low-resource low-latency hybrid adaptive CORDIC with floating-point precision. *IEICE Electronics Express*, *12*(9), 20150258–20150258. Descargado 2020-07-01, de https://www.jstage.jst.go.jp/article/elex/12/9/12_12.20150258/_article doi: 10.1587/elex.12.20150258

Parker, M. (2011). Abstract – Floating Point. *DesignCon 2011*, 15. Descargado de https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/designcon2011-floating-point-design-flow.pdf

Schelin, C. W. (1983, mayo). Calculator Function Approximation. *The American Mathematical Monthly*, *90*(5), 317–325. Descargado 2020-07-02, de https://doi.org/10.1080/00029890.1983.11971220 (Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/00029890.1983.11971220) doi: 10.1080/00029890.1983.11971220

Volder, J. (1959, marzo). The CORDIC computing technique. En *Papers presented at the the March 3-5, 1959, western joint computer conference* (pp. 257–261). San Francisco, California: Association for Computing Machinery. Descargado 2020-02-25, de https://doi.org/10.1145/1457838.1457886 doi: 10.1145/1457838.1457886

Walther, J. S. (1971). A unified algorithm for elementary functions. En *Proceedings of the May 18-20, 1971, spring joint computer conference on - AFIPS '71 (Spring)* (p. 379). Atlantic City, New Jersey: ACM Press. Descargado 2020-02-25, de http://portal.acm.org/citation.cfm?doid=1478786.1478840 doi: 10.1145/1478786.1478840

Wang, Y., Luo, Y., Wang, Z., Shen, Q., y Pan, H. (2020, abril). GH CORDIC-Based Architecture for Computing N th Root of Single-Precision Floating-Point Number. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *28*(4), 864–875. (Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems) doi: 10.1109/TVLSI.2019.2959847

Yeshwanth, B., Venkatesh, V., y Akhil, R. (2018, diciembre). High-Speed Single Precision Floating Point Multiplier using CORDIC Algorithm. En *2018 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)* (pp. 135–141). doi: 10.1109/ICEECCOT43722.2018.9001506

Zhou, J., Dou, Y., Lei, Y., Xu, J., y Dong, Y. (2008, septiembre). Double Precision Hybrid-Mode Floating-Point FPGA CORDIC Co-processor. En *2008 10th IEEE International Conference on High Performance Computing and Communications* (pp. 182–189). doi: 10.1109/HPCC.2008.14

Lista de Acrónimos y Abreviaturas

CORDIC	COordinate Rotation DIgital Computer.
FPGA	Field-programmable gate array.
FPU	Floating-point Unit.
FSM	Finite State Machine.
HDL	Hardware Description Language.
IEEE	Institute of Electrical and Electronics Engineers.
LUT	Lookup Table.
TFG	Trabajo Final de Grado.
VHDL	Very High Speed Integrated Circuit Hardware Description Language.