



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

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

Resumen

El método COordinate Rotation Digital Computer (CORDIC) es un simple algoritmo que puede calcular de forma eficiente múltiples operaciones matemáticas, como funciones trigonométricas, raíces, multiplicaciones, divisiones, etc. solamente usando sumas, restas, movimiento de bits y una pequeña Lookup Table (LUT).

El auge de procesamiento con punto flotante ha hecho que se tenga que reevaluar los casos de uso del método e investigar y aportar variantes que usen este tipo de datos de forma eficiente. por lo que se ha realizado un estudio sobre las mejoras que se han ido realizando a lo largo de los 60 años y, además, se ha hecho hincapié en el posible uso de CORDIC con punto flotante.

Después de realizar un estudio extenso sobre los casos de uso e investigación actual sobre CORDIC, se ha implementado tres variantes usando el lenguaje Verilog para evaluar las mejoras expuestas en los estudios expuestos anteriormente. El primero se basa en el propio método estándar, el segundo se añade *pipelining* y un último con una conversión a punto flotante.

Tras la implementación, se ha visto una clara mejora de *throughput* frente al CORDIC básico, y se ha visto una mejora de precisión en CORDIC en punto flotante, con el inconveniente de aumento del número de ciclos para procesar. También se ha visto un aumento de complejidad en la implementación de punto flotante, un punto negativo a la hora de tener en cuenta que el método tiene una filosofía de simplicidad.

La idea de investigar los casos de uso actuales del CORDIC viene de Antonio Jimeno-Morenilla, que fue recomendado por uno de mis tutores, Marcelo Saval Calvo. Por recomendación de Antonio, Jose Luís Sánchez Romero se unió como segundo tutor para ayudar en el trabajo al tener un buen conocimiento de arquitecturas y haber trabajado con CORDIC anteriormente. Agradezco a todos los profesores que me han guiado en este trabajo y ofrecido cualquier apoyo que haya necesitado.

*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

1. Introducción	1
1.1. Motivación y contexto	1
1.2. Método CORDIC	2
1.2.1. Revisiones y mejoras de CORDIC	2
1.3. Presente y futuro de CORDIC	2
1.4. Objetivos	3
1.5. Estructura del documento	3
2. CORDIC	5
2.1. Descripción de CORDIC	5
2.2. Mejoras a CORDIC	7
2.2.1. 4-Radix	8
2.2.2. Angle Recording	8
2.2.3. Hybrid CORDIC	8
2.2.4. Redundant-Number-Based CORDIC	8
2.2.5. Pipelined CORDIC	9
2.3. CORDIC y punto flotante	9
2.3.1. Artículos mas recientes	12
3. Implementación de CORDIC	15
3.1. Herramientas	15
3.1.1. Linux	15
3.1.2. Verilog	15
3.1.3. Verilator	15
3.2. Configuración de Verilator	16
3.3. Implementación básica	18
3.3.1. Testbench de CORDIC	19
3.4. Implementación <i>pipeline</i>	22
3.4.1. Testbench de CORDIC <i>pipeline</i>	23
3.5. Implementación con punto flotante	25
3.5.1. Testbench de CORDIC punto flotante con <i>pipelining</i>	31
4. Conclusiones	35
Bibliografía	37
Lista de Acrónimos y Abreviaturas	41
A. Salidas de GTKWave de las implementaciones de CORDIC	43

Índice de figuras

2.1. Rotation Mode. Figura de Schelin (1983)	6
2.2. Vectoring Mode. Figura de Schelin (1983)	7
2.3. CORDIC convencional con <i>pipelining</i> . Extraído de Meher et al. (2009)	9
2.4. Arquitectura de FP CORDIC. Figura de de Lange et al. (1988)	11
2.5. Unidad de procesamiento de CORDIC de doble precisión. Figura extraída de Yeshwanth et al. (2018)	13
3.1. Tasa de error en CORDIC con punto flotante.	33
3.2. Número de ciclos para calcular 20 valores de forma consecutiva (cada ciclo se introduce un nuevo valor).	33
A.1. Salida de GTKWave de CORDIC básico	44
A.2. Salida de GTKWave de CORDIC con <i>pipelining</i> . Aquí se muestran todos los registros de X y solo los últimos registros de Y y Z , además de las entradas y salidas.	45
A.3. Salida de GTKWave de CORDIC con punto flotante y <i>pipelining</i> . Igual que en la figura anterior, se muestra todos los registros de X y solo los últimos registros de Y e Z . Además se muestran registros como <i>next_state</i> y <i>current_state</i> y otras, como contadores.	46

Índice de tablas

2.1. Diferentes configuraciones de CORDIC, segun Meher et al. (2009)	8
2.2. Características de FP CORDIC. Datos de de Lange et al. (1988)	11
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.	12
2.4. Tabla de comparaciones del CORDIC propuesto con otros trabajos parecidos. Tabla de Nguyen et al. (2015)	12
2.5. Número de iteraciones comparando el multiplicador CORDIC con punto flo- tante de Yeshwanth et al. (2018) a otras soluciones.	13
2.6. Comparativa entre un multiplicador Vedic y CORDIC de Yeshwanth et al. (2018).	14

Índice de Códigos

3.1. Definición de una TestBench de Verilator	16
3.2. Definición de funciones para trazar la simulación.	16
3.3. Definición de funciones <i>reset()</i> y <i>tick()</i>	17
3.4. Módulo CORDIC	18
3.5. Pre-rotación de iAngle pasado por el usuario.	18
3.6. Tabla de arctan.	19
3.7. Registros y <i>wire</i> para operar con CORDIC	19
3.8. Bucle principal de CORDIC	19
3.9. Creación del tipo TestBench y entrada de valores desde C++ al módulo de CORDIC en Verilog	20
3.10. Bucle principal de main.cpp	20
3.11. Registros de CORDIC ampliados para <i>pipelining</i>	22
3.12. Ejecución principal de CORDIC con <i>pipelining</i>	22
3.13. Cambios de entrada de valores de C++ a Verilog en la implementación de Punto Flotante	26
3.14. Registros para controlar los estados y parámetros locales de estado para controlar la máquina de estado finita.	26
3.15. <i>state memory</i> de CORDIC	27
3.16. Código de la <i>next state logic</i> de la Finite State Machine (FSM)	27
3.17. Código de <i>output logic</i> de la FSM	28
3.18. Cambio en las asignaciones de las salidas de CORDIC en punto flotante	30
3.19. Cambio en el bucle principal de CORDIC.	30

1. Introducción

1.1. Motivación y contexto

Dentro de la ingeniería informática me ha llamado mucho atención las asignaturas relacionadas de arquitecturas. Ya que en la ingeniería se da muchas áreas de informática, no es posible abarcar todo los niveles de forma muy detallada, por lo que he decidido centrarme con el Trabajo de Fin de Grado un nivel propiamente de arquitectura de computadores. Una de las recomendaciones del profesorado fue investigar el método CORDIC, el cual ha resultado ser un buen punto de comienzo para adentrarse en el nivel de circuitos digitales.

CORDIC es un método desarrollado en los años 60 que ha sido utilizado en múltiples ocasiones en proyectos de diferente tipo, pero debido a la necesidad cada vez mayor de utilizar datos decimales, y las mejoras en rendimiento de los procesadores de propósito general que incluyen una Floating-point Unit (FPU), lo que permite un cómputo mucho mas rápido de datos decimales, hizo bajar la popularidad de CORDIC ya que el procesamiento de CORDIC posee una gran latencia. Además, en software es más 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 más 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, autor de CORDIC. 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 Field-programmable gate array (FPGA) 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 (como por ejemplo *ULX3S* (2020)), que ofrecen unas características muy interesantes a un precio relativamente bajo. 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 seno y coseno).

1.2. Método CORDIC

En 1959, Volder publica el primer artículo del algoritmo que denominó COordinate ROtation DIgital Computer (CORDIC). Este algoritmo fue originalmente pensado como un método 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, 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 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 *Hewlett-Packard Laboratories* (HP) publicó 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 *co-procesador* 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.

1.3. Presente y futuro de CORDIC

Aunque el algoritmo no sea el 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.

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.

Algunas de las aplicaciones de CORDIC son:

- Direct Kinematics Solution (DKS) o solución de cinemática directa para manipuladores de robots seriales. Las rotaciones y traslaciones de los eslabones que generan
-

nuevas coordenadas son calculadas por CORDIC. Un método similar es usado para la cinemática inversa. [20]

- CORDIC también se ha usado como una unidad funcional de un procesador de robot para temas de redundancia y cálculo de detección de colisión. [30]
- 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. [19]
- Factorización QR. Hay variantes de CORDIC que ofrecen esta funcionalidad. [11]
- Dentro del procesamiento de señal podemos encontrar un gran número de implementaciones para el cálculo de Discrete Fourier transform (DFT). [2]
- En el ámbito de la comunicación, CORDIC puede ser usado para la modulación digital y analógica. Dependiendo de los parámetros del método, es posible hacer cálculo digital de una multitud de modulaciones. [28]

Como se puede ver, CORDIC es una propuesta antigua pero que ha sufrido mejoras y aún en la actualidad se utiliza. Esto genera un gran interés por su estudio e implementación.

1.4. Objetivos

El objetivo principal de este Trabajo Fin de Grado es el estudio y desarrollo de alternativas al procesamiento de funciones hiperbólicas y trigonométricas de punto flotante según el estándar del Institute of Electrical and Electronics Engineers (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.

1.5. Estructura del documento

Primero se explicará en el capítulo de CORDIC el algoritmo básico estipulado por Volder (1959). Posteriormente, en el mismo capítulo se describirá algunas de las mejoras mas populares, las cuales han sido usadas con los años, con ciertas mejoras. Finalmente, el último punto se expondrá un estado del arte centrándose en CORDIC y punto flotante.

Dentro del capítulo de Implementación de CORDIC se expondrá las herramientas usadas y se mostrará tres implementaciones de CORDIC: básica, *pipelining* y una con *pipelining* y punto flotante.

Finalmente, se hará una comparación y se desarrollará una conclusión a partir en base del estudio y trabajo realizado.

2. CORDIC

En este capítulo veremos el funcionamiento de básico, algunas posibles mejoras y el caso específico de tratamiento de datos en punto flotante.

2.1. Descripción de CORDIC

En esta sección se describirá el funcionamiento principal del algoritmo CORDIC según descrito por Volder (1959).

La rotación de un vector de dos dimensiones $\mathbf{p}_0 = [x_0, y_0]$ con un ángulo θ para obtener un vector $\mathbf{p}_n = [x_n, y_n]$ se puede realizar con un producto matriz $\mathbf{p}_n = \mathbf{R}_{p_0}$, 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 \mathbf{R}_c tal que

$$\mathbf{R}_c = \begin{bmatrix} 1 & -\tan \theta \\ \tan \theta & 1 \end{bmatrix}$$

La operación pseudo-rotación rota el vector p_0 por un ángulo θ y cambia su magnitud por un factor $K = \cos \theta$ para producir una pseudo-rotación $p'_n = R_c p_0$.

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 ignorándolo 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

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

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 $K \approx 1.6467605$, por lo cual simplemente se puede escalar el valor final por K .

Los valores finales que nos interesan son los siguientes:

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

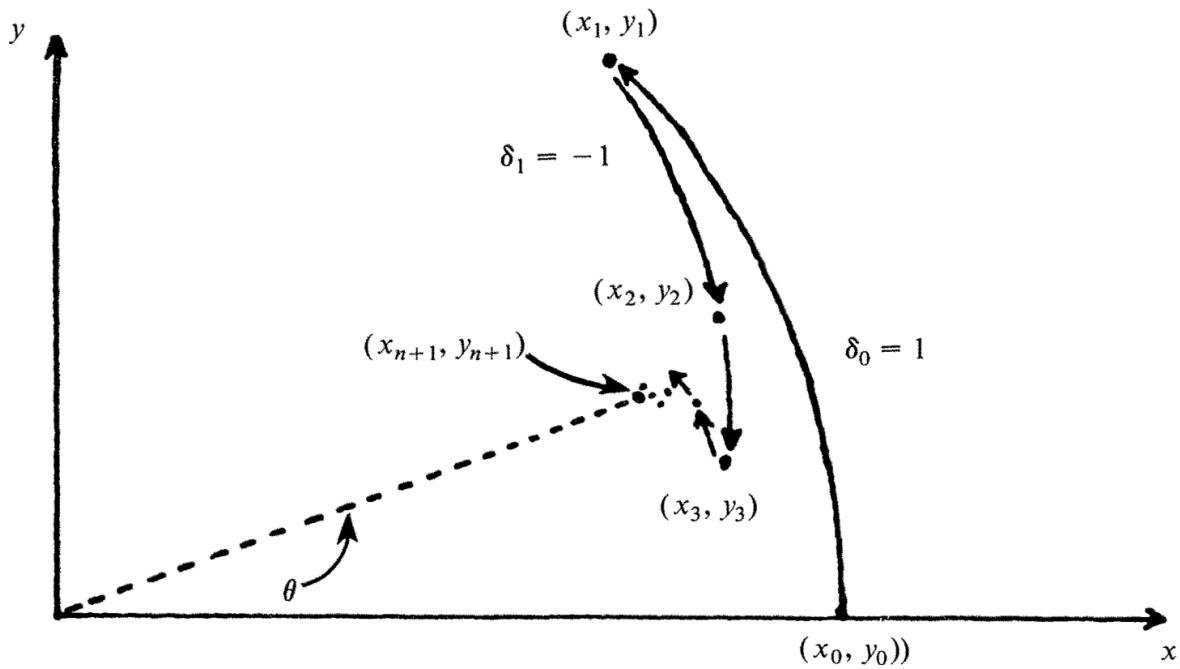


Figura 2.1: Rotation Mode. Figura de Schelin (1983)

CORDIC puede operar de dos formas: *Rotation Mode* (RM) (Figura 2.1) y *Vectoring Mode* (VM) (Figura 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 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. La tabla 2.1 muestra las diferentes configuraciones que puede tener CORDIC para obtener un tipo de operación u otro. Esto se encuentra fuera

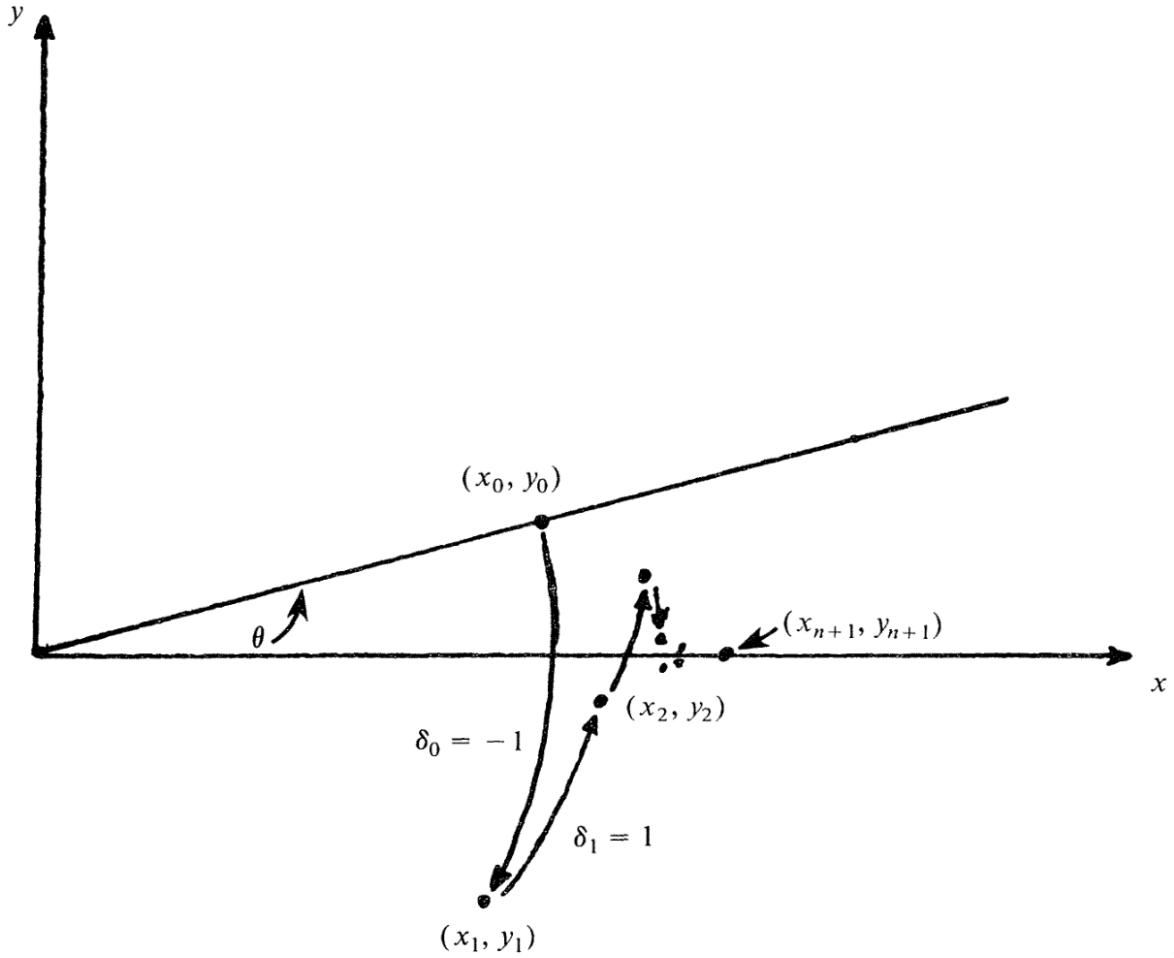


Figura 2.2: Vectoring Mode. Figura de Schelin (1983)

del alcance de esta memoria, pero se ha querido mostrar la amplitud del método para casos de uso diferentes.

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 et al. (2009), y expuestas en la tabla 2.1.

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 la 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

Operación	Configuración	Inicialización	Salida	Anotaciones
$\cos \theta, \cos \theta, \tan \theta$	CC-RM	$x_0 = 1 \ y_0 = 0 \ y \ \omega_0 = \theta$	$x_n = \cos \theta \ y_n = \sin \theta$	$\tan \theta = (\sin \theta / \cos \theta)$
$\cosh \theta, \sinh \theta, \tanh \theta, \exp(\theta)$	HC-RM	$x_0 = 1 \ y_0 = 0 \ y \ \omega_0 = \theta$	$x_n = \cosh \theta \ y_n = \sinh \theta$	$x_n = \sqrt{a} \ \omega_n = 1/2 \ln(a)$
$\ln(a), \sqrt{a}$	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 et al. (2009)

que se pasan a CORDIC.

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 (Antelo et al. (1997)) La fórmula final seria tal que:

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

El valor K tendría una fórmula:

$$K_i = \frac{1}{\text{sqr}t(1 + \delta_i^2 * 4^{-2i})}$$

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.

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 los ángulos de las micro-rotaciones. Este método es ideal para procesamiento de señal, imagen, donde el ángulo de rotación se conoce *a priori*. Algunos métodos consiguen reducir hasta un 50% el número de iteraciones, manteniendo la misma precisión. Hu y Naganathan (1993)

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. S. Wang et al. (1997)

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. Ercegovac y Lang (1990)

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, por lo que podemos aprovechar esta propiedad para calcular un nuevo valor por cada ciclo de ejecución, dando lugar a un nuevo resultado. Para esto necesitamos tener un registro por cada etapa de CORDIC (vea la figura 2.3.).

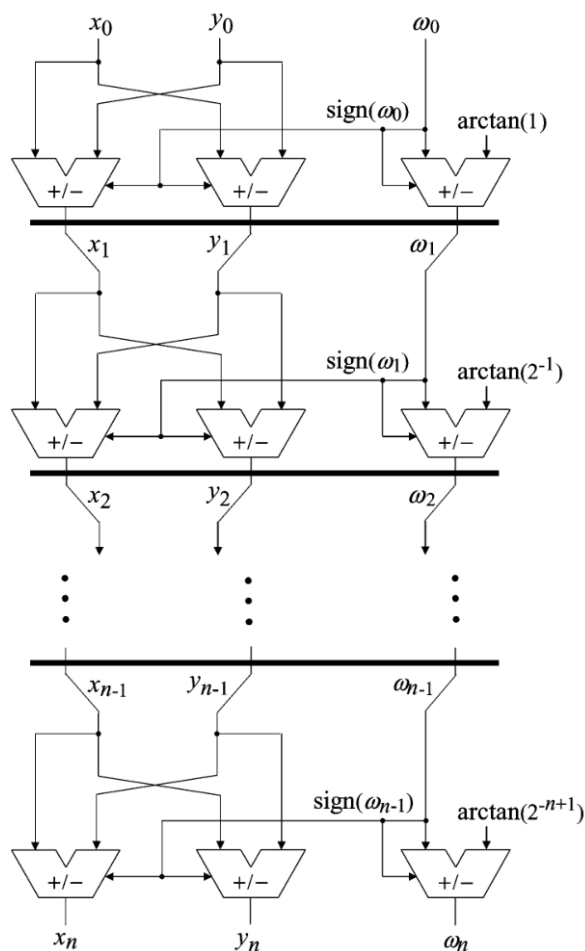


Figura 2.3: CORDIC convencional com *pipelining*. Extraído de Meher et al. (2009)

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 $[0:0.000..]$.
- 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.
- Very High Speed Integrated Circuit Hardware Description Language (VHDL) y Verilog no tienen una implementación de operaciones con punto flotante. Un lenguaje de descripción de hardware más nuevo, como SystemVerilog, sí 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 Binary-Coded Decimal (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 et al. (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 la figura 2.4). Cabe destacar que el tipo usado en esta arquitectura no es estándar de IEEE 754.

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 (vea la tabla 2.2).

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 flotante (x, y) con un ángulo de punto flotante α . Este algoritmo fue diseñado con la intención

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.

Dispositivo	Xilinx Vertex 5	Xilinx Vertex 5	Xilinx Vertex 7	Altera Stratix II	Xilinx Vertex 6	Altera Stratix IV	Diseño de Nguyen et al. (2015)
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 et al. (2015)

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

Nguyen et al. (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 un umbral particular para tener un error de cálculo muy cercano a como si se hiciera con 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 seno y coseno con un valor de 32 bits IEEE 754 cada una.

Como se puede ver en la tabla 2.3, Los tiempos de latencia son mucho mejores que los de otros trabajos, además de una reducción de LUTs, registros usados y memoria.

2.3.1. Artículos mas recientes

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

Hou et al. (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

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 et al. (2018) a otras soluciones.

ciclo 16 micro-rotaciones de CORDIC, reduciendo efectivamente el tiempo por 4 (vea la figura 2.5). El número de iteraciones es menor a otros métodos de CORDIC, como por ejemplo Para-CORDIC (vea Juang et al. (2004)).

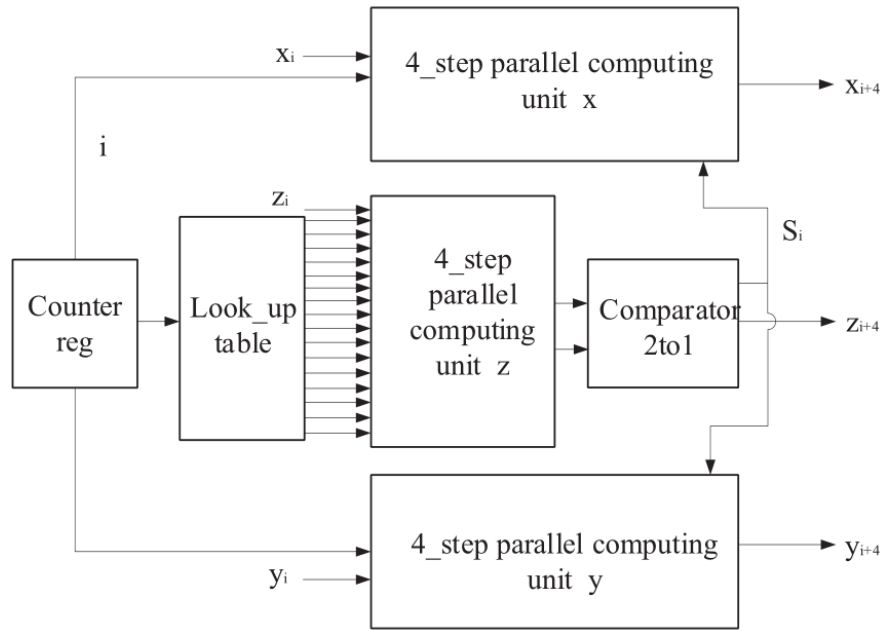


Figura 2.5: Unidad de procesamiento de CORDIC de doble precisión. Figura extraída de Yeshwanth et al. (2018)

Como se comentó anteriormente, CORDIC puede realizar una multitud de operaciones matemáticas según la necesidad del problema. Yeshwanth et al. (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 et al. (2018) diseñaron en una FPGA una arquitectura basada en CORDIC para la cinemática inversa de un robot hexápodo. El diseño usa numeración de punto flotante de 32 bits, además de *pipelining* para obtener mejoras de frecuencia y *throughput*. Los resultados son comparados a otros robots hexápodo de diferente gama, pero que generalmente tienen procesadores mucho mas potentes y con un gasto energético mucho mas alto. Los resultados además muestran un beneficio en la escalabilidad.

Dentro del trabajo de los radares de apertura sintética (Synthetic Aperture Radar (SAR)), Fang et al. (2019) proponen una mejora de los tiempos de cálculo de operaciones trigonomé-

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 et al. (2018).

tricas, 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.

Y. Wang et al. (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

En este capítulo se va a especificar y razonar las herramientas usadas para la implementación de CORDIC y se mostrará tres diferentes implementaciones, además de sus simulaciones y comparativas entre ellas.

3.1. Herramientas

3.1.1. Linux

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

3.1.2. Verilog

El lenguaje de descripción de hardware (Hardware Description Language (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.

3.1.3. 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. En concreto, todo el desarrollo ha sido realizado en dos máquinas con Arch 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.

3.2. Configuración de Verilator

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

```
1 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 sería un tipo *Vcordic*, el cual esta incluido en "obj_dir/Vcordic.h"

Código 3.1: Definición de una TestBench de Verilator

```
1 class TestBench{
2     unsigned long m_tickcount;
3     Vcordic *m_core;
4     VerilatedVcdC *m_trace;
5
6     TestBench(void){
7         m_core = new Vcordic;
8         m_tickcount = 0;
9         Verilated::traceEverOn(true);
10    }
11
12    ...
13};
```

Dentro de la definición del *TestBench* tenemos un tipo *Vcordic**, el cual permitirá acceder a las variables del módulo. La variable de tipo *unsigned long*, *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.

```
1 virtual void opentrace(const char *name) {
2     m_trace = new VerilatedVcdC;
3     m_core->trace(m_trace, 99);
4     m_trace->open(name);
5
6 }
7
8 virtual void close(void) {
9     m_trace->close();
10    delete m_trace;
```



```
11     m_trace = NULL;
12 }
```

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()*

```
1  virtual void reset(void){
2
3      m_core->iReset = 1;
4      this->tick();
5      m_core->iReset = 0;
6  }
7
8  virtual void tick(void){
9      m_tickcount++;
10
11     m_core->iClock = 0;
12     m_core->eval();
13
14     if(m_trace)
15         m_trace->dump(10*m_tickcount-2);
16
17     m_core->iClock = 1;
18     m_core->eval();
19
20     if(m_trace)
21         m_trace->dump(10*m_tickcount);
22
23
24     m_core->iClock = 0;
25     m_core->eval();
26
27     if (m_trace) {
28         m_trace->dump(10*m_tickcount+5);
29         m_trace->flush();
30     }
31
32 }
```

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.

3.3. Implementación básica

Ya que CORDIC tiene la ventaja de ser muy fácil de implementar en hardware, una implementación básica de CORDIC nos permite realizar

Las entradas de nuestro módulo *iX*, *iY*, *iAngle*, *iClock* y *iReset*. Las salidas son *oX* y *oY*. Inicialmente todas 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 abarcar este problema. Una manera fácil de saber el cuadrante en el que se encuentra es hacer el módulo de 2π 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 cambiar el ángulo en el *testbench* 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.

```
1 begin
2     case(quadrant)
3         2'b00,
4         2'b11:
5             begin
6                 X <= iX;
7                 Y <= iY;
8                 Z <= iAngle;
9             end
10
11         2'b01:
12             begin
13                 X <= -iY;
14                 Y <= iX;
15                 Z <= {2'b00, iAngle[29:0]};
16             end
17
18         2'b10:
19             begin
20                 X <= iY;
21                 Y <= -iX;
22                 Z <= {2'b11, iAngle[29:0]};
23             end
24     end
```

```

24     endcase
25 end

```

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

Código 3.6: Tabla de arctan.

```

1 wire signed [31:0] atan_angle [0:30]
2
3 assign atan_angle[00] = 32'h20000000; // arctan(2^0)
4 assign atan_angle[01] = 32'h12E4051D;
5 assign atan_angle[02] = 32'h9FB385B;
6 ...
7 assign atan_angle[28] = 32'h2;
8 assign atan_angle[29] = 32'h1;
9 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 tipos registros y 3 tipo *wire*.

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

```

1 reg signed [IOWidth-1:0] X;
2 reg signed [IOWidth-1:0] Y;
3 reg signed [31:0] Z;
4
5 wire signed [31:0] X_shr = X >>> i-1;
6 wire signed [31:0] Y_shr = Y >>> i-1;
7 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

```

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

```

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

3.3.1. 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:

```

1 Verilated::commandArgs(argc,argv);

```

Despues de incluir el fichero "TestBench.h" en el "main.cpp", debemos de crear un tipo *TestBench* y cargar las variables de la clase con los valores deseados.

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

```

1 TestBench *tb = new TestBench();
2
3 const float K = 1.646760;
4 int32_t angle = atoi(argv[3]);
5 int32_t X = atoi(argv[1]);
6 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.

Ahora se mostrará un ejemplo de una *testbench* con los siguientes parámetros.

```

1 ./cordic 1000 0 130

```

El primer valor se refiere a X , el segundo al valor Y y el último al ángulo que queremos rotar. El valor final será multiplicado por 1000 para obtener un resultado deseado, donde se pueda ver que realmente se realizan las operaciones de seno y coseno. El valor Y , según la especificación del método, tiene que tener un valor 0 para obtener el seno y coseno (véa la tabla 2.1).

```

1 tick = 0
2 cos = 0
3 sin = 607
4
5 tick = 1
6 cos = -607
7 sin = 607
8

```

```
9 tick = 2
10 cos = -304
11 sin = 911
12
13 tick = 3
14 cos = -531
15 sin = 835
16 Este es un
17
18 tick = 4
19 cos = -635
20 sin = 768
21
22 tick = 5
23 cos = -683
24 sin = 728
25
26 tick = 6
27 cos = -661
28 sin = 750
29
30 tick = 7
31 cos = -650
32 sin = 761
33
34 tick = 8
35 cos = -645
36 sin = 767
37
38 tick = 9
39 cos = -643
40 sin = 770
41
42 tick = 10
43 cos = -644
44 sin = 768
45
46 tick = 11
47 cos = -644
48 sin = 767
49
50 tick = 12
51 cos = -644
52 sin = 766
53
54 tick = 13
55 cos = -644
56 sin = 767
57
58 tick = 14
59 cos = -644
```

```

60 sin = 766
61
62 tick = 15
63 cos = -644
64 sin = 767
65
66 tick = 16
67 cos = -644
68 sin = 768
69 ...

```

La salida de seno y coseno muestran los registros X e Y . A diferencia de las siguientes implementaciones, esta te devuelve un valor distinto de 0 desde el primer ciclo, ya que se esta operando siempre sobre ese valor.

Sobre el *tick* 16, el valor se aproxima mucho al valor real de seno y coseno, por lo que no tiene mucho sentido realizar mas micro-rotaciones, ya que los valores menos significativos se encuentran ahí.

Para ver las salidas por GTKWave, vea la figura A.1.

3.4. 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 instancia-ción 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*

```

1 genvar i;

```

```

2 generate
3   for (i = 0; i < (ETAPAS-1); i=i+1)
4   begin
5     wire Z_sign;
6     wire signed [IOWidth:0] X_shr, Y_shr;
7
8     assign X_shr = X[i] >>> i;
9     assign Y_shr = Y[i] >>> i;
10
11    assign Z_sign = Z[i][31];
12
13    always @(posedge iClock)
14    begin
15      // add/subtract shifted data
16      X[i+1] <= Z_sign ? X[i] + Y_shr : X[i] - Y_shr;
17      Y[i+1] <= Z_sign ? Y[i] - X_shr : Y[i] + X_shr;
18      Z[i+1] <= Z_sign ? Z[i] + atan_angle[i] : Z[i] - atan_angle[i];
19    end
20  end
21 endgenerate

```

Como se puede observar, el espacio que ocupa un CORDIC con *pipelining* es mayor que el básico, ya que necesitamos generar el mismo número de registros que de etapas en nuestro *pipeline*, pero tenemos la ventaja de una mejora del *throughput*. A partir del ciclo 16 estamos recibiendo cada ciclo un nuevo resultado.

Ya que esta implementación no han sido probadas en hardware real y solamente simuladas, hay algunos aspectos que no se tienen en cuenta que sí 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 está conectada a todas 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 está 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.

3.4.1. Testbench de CORDIC *pipeline*

A continuación se va a mostrar la salida del *Testbench*. En esta *testbench* se ha optado por pasar cada ciclo un nuevo ángulo, comenzando por el 10, para mostrar el funcionamiento del *pipelining*. Cada ciclo después del *tick* 15 se está devolviendo un nuevo valor. Para ver las salidas por GTKWave, vea la figura A.2.

```

1 tick = 0

```

```
2
3cos = 0
4sin = 0
5tick = 1
6
7cos = 0
8sin = 0
9tick = 2
10
11cos = 0
12sin = 0
13tick = 3
14
15cos = 0
16sin = 0
17tick = 4
18
19cos = 0
20sin = 0
21tick = 5
22
23cos = 0
24sin = 0
25tick = 6
26
27cos = 0
28sin = 0
29tick = 7
30
31cos = 0
32sin = 0
33tick = 8
34
35cos = 0
36sin = 0
37tick = 9
38
39cos = 0
40sin = 0
41tick = 10
42
43cos = 0
44sin = 0
45tick = 11
46
47cos = 0
48sin = 0
49tick = 12
50
51cos = 0
52sin = 0
```

```
53 tick = 13
54
55 cos = 0
56 sin = 0
57 tick = 14
58
59 cos = 0
60 sin = 0
61 tick = 15
62
63 cos = 984
64 sin = 175
65 tick = 16
66
67 cos = 935
68 sin = 346
69 tick = 17
70
71 cos = 864
72 sin = 499
73 tick = 18
74
75 cos = 766
76 sin = 644
77 tick = 19
78
79 cos = 644
80 sin = 766
81 tick = 20
```

Como se comentó en la sección anterior, la implementación *pipeline* tiene el mismo número de registros que de etapas por las que pasa el método, por lo que en las primeras ejecuciones no vamos a recibir ningún valor, ya que el método aún no ha llegado a esa etapa.

Se puede observar una clara mejora al CORDIC básico, ya que no necesitamos esperar al resultado de una entrada antes de enviar otra, si no que cada ciclo podemos dar un nuevo valor a tratar.

3.5. 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

```

1 ...
2 tb->m_core->iX = (X * pow(2,24)) / K;
3 tb->m_core->iY = (Y * pow(2,24)) / K;
4 ...

```

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. Hay otra manera de dividir, que sería la multiplicación por un número decimal, con el cual conseguiríamos una división, pero añadir una multiplicación al bucle de CORDIC llevaría a mayor complejidad hardware y peores tiempos.

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 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 estados y parámetros locales de estado para controlar la máquina de estado finita

```

1 reg [2:0] current_state = w_normal_op, next_state;
2
3 localparam w_normal_op = 3'b000,
4             w_check_sign = 3'b001,
5             w_bit_shift_count = 3'b010,
6             w_to_fp = 3'b011,
7             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

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

El siguiente bloque es el *next state logic*. Aquí se suele utilizar el *case* para controlar a que estado pasar. Generalmente se recomienda hacer uso del *default* por si hay un error inesperado.

Este bloque solo es activado por el registro *current_state*, lo que convierte esta FSM una máquina de Moore, ya que el sistema cambia por un cambio en el estado, independiente de cualquier otra entrada. De no ser así la FSM sería una máquina de Mealy y se incluiría dentro de la lista de sensibilidad otras entradas dependientes.

Código 3.16: Código de la *next state logic* de la FSM

```
1 always@(current_state)
2   begin: NEXT_STATE_LOGIC
3     case(current_state)
4       default:
5         begin
6           end
7
8       w_normal_op:
9         begin
10          if(last_x_value != X[ETAPAS-1] || last_y_value != Y[ETAPAS-1] ↵
11            ↵ begin
12              next_state = w_check_sign;
13            end
14          end
15
16       w_check_sign:
17         if(sign_checked) begin
18           next_state = w_bit_shift_count;
19         end
20
21       w_bit_shift_count:
22         if(tempXOut[bitmov_x] == 1 && tempYOut[bitmov_y] == 1 ) begin
23           next_state = w_to_fp;
24         end
25
26       w_to_fp:
27         if(fp_out) begin
28           next_state = w_normal_op;
29         end
30     endcase
```

28 **end**

Este bloque realiza las siguientes funciones:

- *w_normal_op* Observa si el valor *x* o *y* anterior coincide con los que hay actualmente en el último registro del método CORDIC. De no ser así significa que el usuario ha dado un nuevo valor de entrada al módulo y, por tanto, necesitamos pasar por la conversión a punto flotante.
- *w_check_sign* En el momento que se recibe un *sign_checked* pasamos al contador del número de bits a mover.
- *w_bit_shift_count* Se observa si el bit en una posición determinada es 1 de las dos variables temporales que se manipulan dentro de la FSM. Si es así se pasa a la transformación de punto fijo a flotante.
- *w_to_fp* Espera hasta que el registro *fp_out* sea asignado a 1. En ese momento la transformación a punto flotante se ha finalizado.

Ahora vamos a pasar al bloque de *output logic*. Este bloque realiza todas las operaciones necesarias para convertir a punto flotante.

Código 3.17: Código de *output logic* de la FSM

```

1 always @(current_state)
2 begin: OUTPUT_LOGIC
3   case(current_state)
4     w_normal_op,
5
6     w_check_sign:
7     begin
8       fp_out = 0;
9       fpX_out = 0;
10      fpY_out = 0;
11
12      tempXOut = X[ETAPAS-1];
13      sign_x = tempXOut[31];
14
15      tempYOut = Y[ETAPAS-1];
16      sign_y = tempYOut[31];
17
18
19      if(tempXOut[31] == 1)
20      begin
21        \textit{w\_normal\_op}tempXOut = ~(tempXOut-1);
22      end
23
24
25      if(tempYOut[31] == 1)
26      begin
27        tempYOut = ~(tempYOut-1);
28      end

```

```

29
30     sign_checked = 1;
31 end
32
33 w_bit_shift_count:
34 begin
35     sign_checked = 0;
36     if(tempXOut[bitmov_x] != 1 )
37     begin
38         bitcount_x = bitcount_x + 1;
39         bitmov_x = bitmov_x - 1;
40     end
41     if(tempYOut[bitmov_y] != 1)
42     begin
43         bitcount_y = bitcount_y + 1;
44         bitmov_y = bitmov_y - 1;
45     end
46 end
47
48 w_to_fp:
49 begin
50     if(!fp_out)
51     begin
52
53         if(tempXOut[30:23] != 0) begin
54             tempXOut = tempXOut >> (8-bitcount_x);
55         end
56         else begin
57             tempXOut = tempXOut << ~((8-bitcount_x)-1);
58         end
59
60         tempXOut[30:23] = (127 + (8 - (bitcount_x+1)));
61         tempXOut[31] = sign_x;
62         fpX_out = tempXOut;
63         bitmov_x = 31;
64         bitcount_x = 0;
65
66         if(tempYOut[30:23] != 0) begin
67             tempYOut = tempYOut >> (8 - bitcount_y);
68         end
69         else begin
70             tempYOut = tempYOut << ~((8 - bitcount_y)-1);
71         end
72
73         tempYOut[30:23] = (127+(8 - (bitcount_y+1)));
74         tempYOut[31] = sign_y;
75         fpY_out = tempYOut;
76         bitmov_y = 31;
77         bitcount_y = 0;
78         fp_out = 1;
79     end

```

```

80     end
81     default:
82         begin
83             end
84
85 endcase
86
87 end

```

Las funciones realizadas por este bloque son las siguientes:

- *w_check_sign* Reinicia los valores de las ejecuciones pasadas. Posteriormente guarda en dos registros los signos de la *X* e *Y*. Por último se hace una transformación de complemento a 2 a binario normal si el valor fuese negativo y se pone el registro *sign_checked* a 1.
- *w_bit_shift_count* Hace recuento del movimiento de bits que se debe de realizar para ajustarlo con la mantisa. Los valores *bitmov* y *bitcount* inicialmente tienen un valor de 31 y 0 respectivamente. El contador de *X* e *Y* se detiene en el momento que encuentra un 1 en la posición que este actualmente.
- *w_to_fp* La variable *tempXOut* y *tempYOut* tiene que ser ajustada a la mantisa de un IEEE 754 de 32 bits, la cual son los 23 bits menos significativos. Ya que el movimiento de bits puede ser a la derecha o izquierda, dependiendo de la magnitud del valor, necesitamos tener en cuenta los dos casos.

Después de ajustar la mantisa necesitamos guardar en el exponente. Para ello tenemos que tener en cuenta el número de movimiento de bits que se ha realizado anteriormente y sumar el sesgo del exponente, que para un valor de 32 bits es 127. Esta operación se hace para *X* e *Y*.

Finalmente, reiniciamos los valores y volvemos al estado *w_normal_op*.

La salida de los valores *X* e *Y* no van directamente asignados desde los registros finales de la última etapa de CORDIC como en las anteriores implementaciones. Las salidas son asignadas en el último estado de la FSM y reiniciadas a 0 en el momento que hay una nueva operación de transformación a punto flotante.

Código 3.18: Cambio en las asignaciones de las salidas de CORDIC en punto flotante

```

1 assign oY = fpY_out;
2 assign oX = fpX_out;

```

El *pipelining* además tiene que detenerse en el momento que hay un nuevo valor a procesar, por lo que necesitamos detener el bucle principal de CORDIC hasta que termine la FSM.

Código 3.19: Cambio en el bucle principal de CORDIC.

```

1 if(next_state == w_normal_op) begin
2
3     X[i+1] <= Z_sign ? X[i] + Y_shr : X[i] - Y_shr;
4     Y[i+1] <= Z_sign ? Y[i] - X_shr : Y[i] + X_shr;

```

```

5     Z[i+1] <= Z_sign ? Z[i] + atan_angle[i] : Z[i] - atan_angle[i];
6 end
7
8 else begin
9     last_x_value <= X[ETAPAS-1];
10    last_y_value <= Y[ETAPAS-1];
11 end

```

Simplemente comprobamos si el siguiente estado es *w_normal_op* y de no ser así, detenemos toda la *pipeline* de CORDIC.

3.5.1. Testbench de CORDIC punto flotante con *pipelining*

Este *testbench* es muy parecido a la implementación con *pipelining*. Cada ciclo introducimos una nueva entrada. El primer valor es introducido como argumento de la ejecución del programa, que es:

```
1./cordic 1 0 45
```

Los demas valores empiezan por un ángulo de 10.

```

1// ticks omitidos que devuelven un valor 0.
2tick = 1
3tick = 2
4...
5tick = 18
6tick = 19
7cos = 0.707117
8sin = 0.707096
9tick = 20
10...
11tick = 23
12cos = 0.984809
13sin = 0.17364
14tick = 24
15...
16tick = 27
17cos = 0.939708
18sin = 0.341977
19tick = 28
20...
21tick = 31
22cos = 0.866003
23sin = 0.500039
24tick = 32
25...
26tick = 35
27cos = 0.766029
28sin = 0.642806
29tick = 36
30...

```

```
31 tick = 39
32 cos = 0.642806
33 sin = 0.766029
34 tick = 40
35 ...
36 tick = 43
37 cos = 0.500039
38 sin = 0.866003
39 tick = 44
40 ...
41 tick = 47
42 cos = 0.341977
43 sin = 0.939708
44 tick = 48
45 ...
46 tick = 51
47 cos = 0.17364
48 sin = 0.984809
49 tick = 52
50 ...
51 tick = 55
52 cos = -1.3113e-05
53 sin = 1
54 tick = 56
55 ...
56 tick = 59
57 cos = -0.17364
58 sin = 0.984809
59 tick = 60
60 ...
61 tick = 63
62 cos = -0.341977
63 sin = 0.939709
64 tick = 64
65 ...
66 tick = 67
67 cos = -0.500039
68 sin = 0.866003
69 ...
```

A diferencia de un *pipelining* estándar, necesitamos transformar el valor en punto flotante, por lo que tardamos 4 ciclos mas en devolver un resultado. Aun así, tenemos un *throughput* mas alto que un CORDIC tradicional y una mejor precisión en los valores devueltos.

Como tambien se puede observar, algunos valores se tienen que aproximar a su resultado real. En el tick 55 los valores representados son seno y coseno de 90. El coseno es lo suficientemente pequeño para ser considerado 0. Para ver las salidas por GTKWave, vea la figura A.3.

En este *testbench* se ha optado por no multiplicar los valores por 1000, pero se puede hacer de la misma forma que en los otros *testbench* y nos dará un resultado mas preciso.

En la figura 3.1 podemos ver el error producido en cada etapa del cálculo de un valor. A

partir de la micro-rotación 26 el error es idéntico en cada etapa. Todas las implementaciones fueron probadas con sólo 16 etapas para facilitar el desarrollo, pero pueden ampliarse perfectamente a más si es necesario una menor tasa de error.

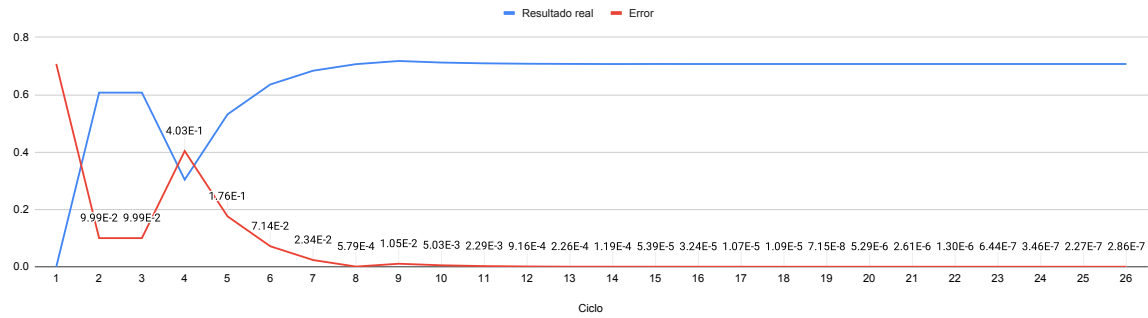


Figura 3.1: Tasa de error en CORDIC con punto flotante.

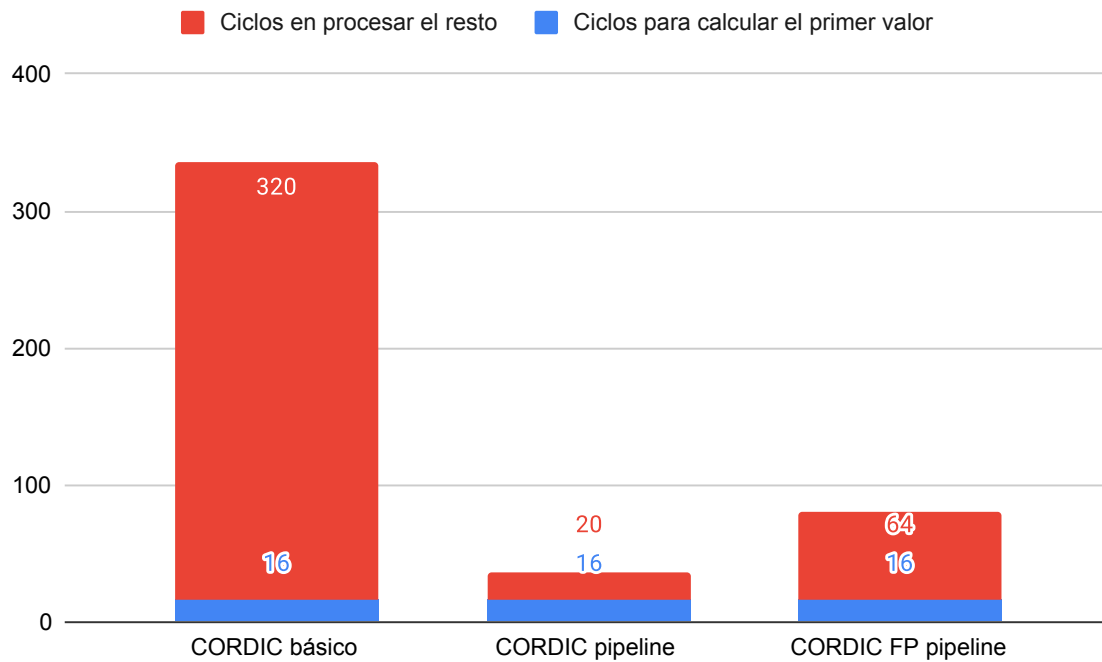


Figura 3.2: Número de ciclos para calcular 20 valores de forma consecutiva (cada ciclo se introduce un nuevo valor).

En la figura 3.2 se muestra las tres implementaciones desarrolladas anteriormente. Se hace una comparativa con un ejemplo de cálculo de 20 valores consecutivos. Cada ciclo se introduce un nuevo valor en las implementaciones con *pipelining* y en la básica se espera hasta recibir el valor final. Como se puede ver, el número de ciclos para calcular múltiples valores de forma

consecutiva baja drásticamente. Aún con el *overhead* que conlleva calcular el valor en punto flotante, tenemos una reducción muy significativa del tiempo respecto al método básico.

4. Conclusiones

En este Trabajo de Fin de Grado se ha realizado un estudio de los casos de uso del algoritmo CORDIC. Como inicio del trabajo se ha realizado un estudio de la base del CORDIC y las mejoras que se han ido introduciendo con los años para mantener el método competitivo. Este estudio se ha centrado sobretodo en la manipulación de datos con punto flotante junto con el método. Se ha observado que la manipulación del IEEE 754 en el hardware es bastante complejo, bits implícitos, valores especiales, normalización de los números, etc. y se llega a entender que, a no ser que sea necesario en la especificación del problema a solucionar no atrae el uso de este.

Tras el estudio, se ha implementado tres distintos CORDIC en Verilog para demostrar algunas mejoras que han sido mencionadas en los estudios. En concreto, un método básico, para mostrar los problemas de latencia, otro con pipelining y uno final con punto flotante. La complejidad de punto flotante, aun sin tener implementado el estándar entero, muestra un claro aumento de espacio ocupado y tiempo de diseño.

Para finalizar, este trabajo tenía intención de dar nuevos conocimientos en el área de circuitos digitales, centrándose en la sintetización lógica, mediante el lenguaje Verilog. Un punto que, debido a la situación extraordinaria de este curso, no se ha llegado a la parte de implementar el diseño en una FPGA, el cual era uno de los objetivos a realizar ya que forma parte del ciclo de desarrollo y nos mostraría un rendimiento real de las implementaciones.

Bibliografía

- Antelo, E., Villalba, J., Bruguera, J., y Zapata, E. (1997, agosto). High performance rotation architectures based on the radix-4 CORDIC algorithm. *IEEE Transactions on Computers*, 46(8), 855–870. (Conference Name: IEEE Transactions on Computers) doi: 10.1109/12.609275
- Das, B., y Banerjee, S. (2002, agosto). Unified CORDIC-based chip to realise DFT/DH-T/DCT/DST. *Computers and Digital Techniques, IEE Proceedings -*, 149, 121–127. doi: 10.1049/ip-cdt:20020429
- 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.
- Ercegovac, M., y Lang, T. (1990, junio). Redundant and on-line CORDIC: application to matrix triangularization and SVD. *IEEE Transactions on Computers*, 39(6), 725–740. (Conference Name: IEEE Transactions on Computers) doi: 10.1109/12.53594
- 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
- FFT/IFFT Block Floating Point Scaling. (2005). , 7.
- 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
- Hu, Y. (1992, julio). CORDIC-based VLSI architectures for digital signal processing. *IEEE Signal Processing Magazine*, 9(3), 16–35. (Conference Name: IEEE Signal Processing Magazine) doi: 10.1109/79.143467

- Hu, Y., y Naganathan, S. (1993, enero). An angle recoding method for CORDIC algorithm implementation. *IEEE Transactions on Computers*, 42(1), 99–102. (Conference Name: IEEE Transactions on Computers) doi: 10.1109/12.192217
- Intro - Verilator - Veripool. (s.f.). Descargado 2020-07-16, de <https://www.veripool.org/wiki/verilator>
- Juang, T.-B. (s.f.). *Area/Delay Efficient Recoding Methods for Parallel CORDIC Rotations*. Descargado 2020-07-28, de <https://www.infona.pl/resource/bwmeta1.element.ieee-art-000004145698> (Publication Title: APCCAS 2006 - 2006 IEEE Asia Pacific Conference on Circuits and Systems) doi: 10.1109/APCCAS.2006.342536
- 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
- Lang, T., y Antelo, E. (2005, abril). High-throughput CORDIC-based geometry operations for 3D computer graphics. *Computers, IEEE Transactions on*, 54, 347–361. doi: 10.1109/TC.2005.53
- Lee, C. S., y Chang, P. (1987). A maximum pipelined CORDIC architecture for inverse kinematic position computation. *IEEE Journal on Robotics and Automation*, 5(3), 445–458. Descargado 2020-07-28, de <https://www.infona.pl/resource/bwmeta1.element.ieee-art-000001087126> doi: 10.1109/JRA.1987.1087126
- 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
- ULX3S. (2020). Descargado 2020-07-30, de <https://www.crowdsupply.com/radiona/ulx3s> (Library Catalog: www.crowdsupply.com)
- Valls, J., Sansaloni, T., Perez-Pascual, A., Torres, V., y Almenar, V. (2006, septiembre). The use of CORDIC in software defined radios: a tutorial. *IEEE Communications Magazine*, 44(9), 46–50. (Conference Name: IEEE Communications Magazine) doi: 10.1109/MCOM.2006.1705978
- 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
- Walker, I., y Cavallaro, J. (1993, junio). Parallel VLSI architectures for real-time kinematics of redundant robots. En (pp. 870–877 vol.1). doi: 10.1109/ROBOT.1993.292086
- 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, S., Piuri, V., y Wartzlander, E. (1997, noviembre). Hybrid CORDIC algorithms. *IEEE Transactions on Computers*, 46(11), 1202–1207. (Conference Name: IEEE Transactions on Computers) doi: 10.1109/12.644295
- 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 (ICEEC-COT)* (pp. 135–141). doi: 10.1109/ICEECOT43722.2018.9001506
-

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

ASIC	Application-Specific Integrated Circuit.
BCD	Binary-Coded Decimal.
CORDIC	COordinate Rotation DIgital Computer.
DFT	Discrete Fourier transform.
FPGA	Field-programmable gate array.
FPU	Floating-point Unit.
FSM	Finite State Machine.
HDL	Hardware Description Language.
IEEE	Institute of Electrical and Electronics Engineers.
LSB	Least Significant Bit.
LUT	Lookup Table.
TFG	Trabajo Final de Grado.
VHDL	Very High Speed Integrated Circuit Hardware Description Language.

A. Salidas de GTKWave de las implementaciones de CORDIC

A continuación se muestran las salidas del programa GTKWave de las distintas implementaciones: básica, *pipelining* y punto flotante. Este programa es similar a otros productos comerciales, pero con la ventaja de ser de código libre. Se ha usado el programa extensivamente para encontrar errores durante el desarrollo del trabajo y ha sido de gran utilidad para entender el funcionamiento interno de cada implementación, sobre todo en la de punto flotante, donde se verá que hay una mayor cantidad de registros a tener en cuenta y manipular.

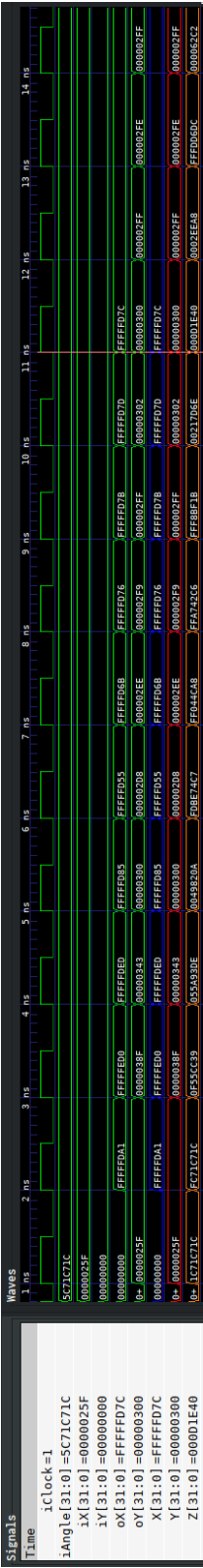


Figura A.1: Salida de GTKWave de CORDIC básico



Figura A.2: Salida de GTKWave de CORDIC con *pipelining*. Aquí se muestran todos los registros de X y solo los últimos registros de Y y Z , además de las entradas y salidas.

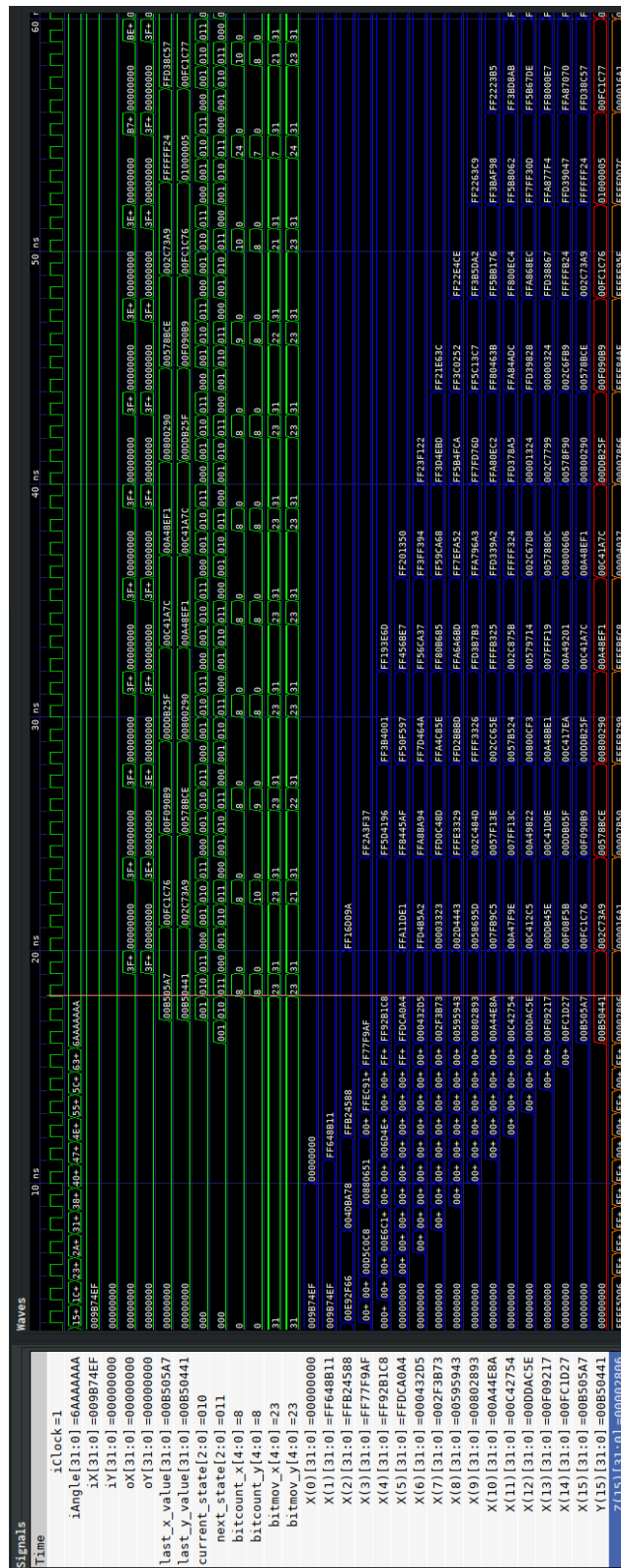


Figura A.3: Salida de GTKWave de CORDIC con punto flotante y *pipelining*. Igual que en la figura anterior, se muestra todos los registros de X' y solo los últimos registros de Y e Z . Además se muestran registros como *next_state* y *current_state* y otras, como contadores.