Escola Politècnica Superior
d'Enginyeria de Vilanova i la Geltrú

UNIVERSITAT POLITÈCNICA DE CATALUNYA

PROJECTE FI DE CARRERA

TÍTOL:
Co-diseño hardware-software de una unidad en coma flotante para microprocesador de 32 bits.

AUTOR: Rubén Lumbiarres López

TITULACIÓ: Enginyeria en automàtica i electrònica industrial

DIRECTOR: Mariano López García

DEPARTAMENT: Enginyeria electrònica

TÍTOL:

Co-diseño hardware-software de una unidad en coma flotante para microprocesador de 32 bits

COGNOMS: Lumbiarres López **NOM:** Rubén

TITULACIÓ: Enginyeria en automàtica i electrònica industrial

ESPECIALITAT: **PLA:** 2003

DIRECTOR: Mariano López García

DEPARTAMENT: Enginyeria electrònica

QUALIFICACIÓ DEL PFC

TRIBUNAL

PRESIDENT	SECRETARI	VOCAL
José Antonio Soria Pérez	Albert Fabregat Sanjuan	Francisco Javier Ruiz Vegas

DATA DE LECTURA: 18-7-2008

Aquest Projecte té en compte aspectes mediambientals: □ Sí □ No

PROJECTE FI DE CARRERA

RESUM (màxim 50 línies)

El uso de números en coma flotante es muy habitual en la programación software para la resolución de todo tipo de algoritmos. Dada la particular forma de codificar valores en este formato, definida en la norma IEEE 754, realizar operaciones básicas requiere de librerías software específicas y tiempos de proceso considerables para su ejecución.

El presente proyecto aporta una solución hardware para la resolución de operaciones aritméticas de números expresados en coma flotante de simple precisión (32 bits). Las operaciones que se realizan son: suma, resta, producto, división, inversa y raíz cuadrada junto con las trigonométricas: seno, coseno y tangente. Estas unidades aritméticas son conocidas como 'Float Point Unit'(FPU).

El diseño de la arquitectura interna de la FPU se describe mediante un lenguaje de descripción hardware (VHDL), de forma modular y con la particularidad de que puede implementarse toda o, únicamente, el bloque que resuelve la operación deseada.

La FPU se implementa en una FPGA junto con un sistema basado en microprocesador Microblaze (Xilinx) y bus OPB (IBM), mediante el que se comunican ambos elementos dando lugar a un sistema embebido. De forma complementaria se diseñan la interface y el driver necesarios para la transferencia de información a través del bus de comunicaciones.

El conjunto procesa algoritmos de forma que la resolución de operaciones en coma flotante se realiza transfiriendo los datos y la operación a la FPU, que tras un procesado de cálculo retorna el resultado al microprocesador.

Una vez puesto el sistema en funcionamiento se mide el tiempo (en ciclos de reloj) que se precisa para resolver cada operación, considerando el tiempo necesario para el acceso al bus OPB. Se compara el tiempo necesario para resolver dichas operaciones por software y con la FPU que el procesador incorpora y se observa la mejora obtenida en cada caso.

También se desarrollan dos algoritmos que permiten comprobar la fiabilidad de la FPU, comparando los resultados obtenidos mediante una ejecución puramente software e incorporando la FPU diseñada.

Paraules clau (màxim 10):

Co-diseño	Coma flotante	Algoritmo	CORDIC
Implementación	Máquina de estados	VHDL	FPGA
MicroBlaze	Bus OPB		

"Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter,

mais quand il n'y a plus rien à retrancher."

“La perfección se alcanza, no cuando no queda nada por agregar,

sino cuando no hay nada más por quitar”

Antoine de Saint-Exupéry
Terre des Hommes 1939

Tabla de contenido

Índice de figuras

Índice de tablas

1 Introducción

El presente proyecto propone una solución hardware para resolver operaciones en coma flotante de 32 bits según se definen en la norma IEE754.

El objetivo perseguido es incorporar a un sistema basado en microprocesador, un coprocesador matemático que realice operaciones en coma flotante de forma más rápida a como se resuelven mediante la ejecución de un software estándar. Además el proceso se realiza de forma paralela, liberando al microprocesador para realizar tareas adicionales. La consecuencia de usar la FPU (Float Point Unit) es que se aumenta considerablemente el rendimiento del sistema.

Por otro lado la unidad diseñada puede implementarse dentro de otros coprocesadores que necesiten la manipulación de números expresados en coma flotante, e incluso puede ser compartida entre varios de ellos.

La estructura modular del sistema y la inclusión de un control lógico secuencial permite la realización de algoritmos complejos formados por las operaciones básicas de la unidad. El sistema es capaz de tomar los valores de salida como elementos de entrada para reprocesarlos sin necesidad de pasar por el control del microprocesador, minimizando de este modo los tiempos de acceso a los valores de entrada y salida.

La unidad es universal, pudiendo conectarse a cualquier bus mediante la adición de un driver de bus adecuado; en el presente proyecto se ha optado por el estándar de IBM OPB (On-chip Peripheral Bus) que se trata de un bus de 32 bits de datos y 32 bits de direcciones.

Como microprocesador se ha utilizado el procesador de 32 bits “Microblaze” y las herramientas de desarrollo “Platform Studio 8.2” ambos de la casa XILINX. Tanto el microprocesador como la unidad se implementan utilizando una FPGA XILINX Spartan3 2000 de 2.000.000 de puertas.

2 Metodologías para la implementación de sistemas digitales

Cuando abordamos la solución de un problema mediante la aplicación de algoritmos digitales podemos optar por tres alternativas diferentes para el diseño:

- Implementación “Todo Hardware”
- Implementación “Todo Software”
- Co-diseño “Software-Hardware”

A continuación se describen con detalle cada una de las soluciones así como las ventajas e inconvenientes particulares de cada una de ellas.

2.1 Implementación todo "Hardware"

Se elabora un sistema mediante el uso de puertas lógicas y circuitos integrados varios que resuelven el algoritmo especificado. La posterior evolución de este sistema es la aparición de circuitos integrados ASIC (circuito integrado de aplicación específica) y los sistemas reconfigurables basados en FPGA y CPLD en los que se implementa un algoritmo determinado que resuelve un problema en concreto. Estos sistemas tienen como principales ventajas:

- La gran velocidad de respuesta del circuito (mayor para el caso ASIC).
- El tiempo de respuesta está claramente acotado: condición fundamental para los sistemas en tiempo real.
- Los procesos son concurrentes: los procesos paralelos funcionan simultáneamente.

Como desventajas se pueden aportar las siguientes:

- Elevado tiempo de diseño y desarrollo (sobre todo en circuitos ASIC): por tanto elevada inversión en I+D.
- Herramientas de diseño y desarrollo sofisticadas que precisan ingenieros con la formación adecuada.
- Baja o nula versatilidad: los circuitos únicamente pueden resolver el algoritmo para el que han sido diseñados y modificar sus características implica rehacer el diseño.

A pesar de que la simulación informática puede reducir el tiempo de desarrollo éste continúa siendo elevado, en comparación con una implementación software.

2.2 Implementación todo "Software"

La aparición de los microprocesadores y los ordenadores personales permiten desarrollar programas que implementan algoritmos que, a su vez, resuelven problemas complejos. El uso de lenguajes de programación de alto nivel facilita la creación de dichos algoritmos. La aparición de herramientas de libre distribución reduce el costo del equipo necesario.

Por otro lado los algoritmos complejos requieren de cientos o miles de líneas de código que generan archivos ejecutables de gran tamaño y que requieren un tiempo de ejecución considerable. Las principales ventajas de estos sistemas de desarrollo son:

- Herramientas de desarrollo baratas, simples y fáciles de localizar: básicamente un ordenador personal, una tarjeta de adquisición de datos y un compilador de libre distribución.
- Tiempo de desarrollo corto: en comparación con el sistema todo "hardware".
- Gran versatilidad: si deseamos modificar las prestaciones o características de un algoritmo podemos modificar el código del programa sin necesidad de modificar circuito alguno.

- Coste de desarrollo reducido.

Las principales desventajas de estos sistemas de desarrollo son:

- El tiempo de ejecución de un algoritmo está directamente ligado a la máquina que lo ejecuta.
- Tiempo de respuesta no acotado: determinar el tiempo máximo de ejecución de un algoritmo de miles de líneas de programa es una tarea complicada, si deseamos un sistema en tiempo real debemos acudir a herramientas de trabajo más complejas y caras que nos permita controlar dichos tiempos de ejecución.
- Los procesos no son concurrentes: en un ordenador sólo puede ejecutarse un proceso por cada CPU de forma simultánea.

La limitación aparece cuando dichos algoritmos han de ejecutarse en máquinas que, por diversas razones, disponen de un procesador de velocidad limitada o una autonomía de funcionamiento pobre.

2.3 Co-diseño Software Hardware.

Podemos combinar las ventajas de los dos sistemas anteriores, aunque también se heredan los inconvenientes. (Véase figura 1)

El diseño software-hardware implica utilizar un microprocesador para aquellas tareas para las que es idóneo y un circuito específico, o coprocesador, para aquellas que lo permitan. De esta forma el sistema consiste en un microprocesador que ejecuta un programa y, en el momento en el que se necesite, el microprocesador pasa los datos al coprocesador y es éste quien los procesa devolviendo el resultado al primero.

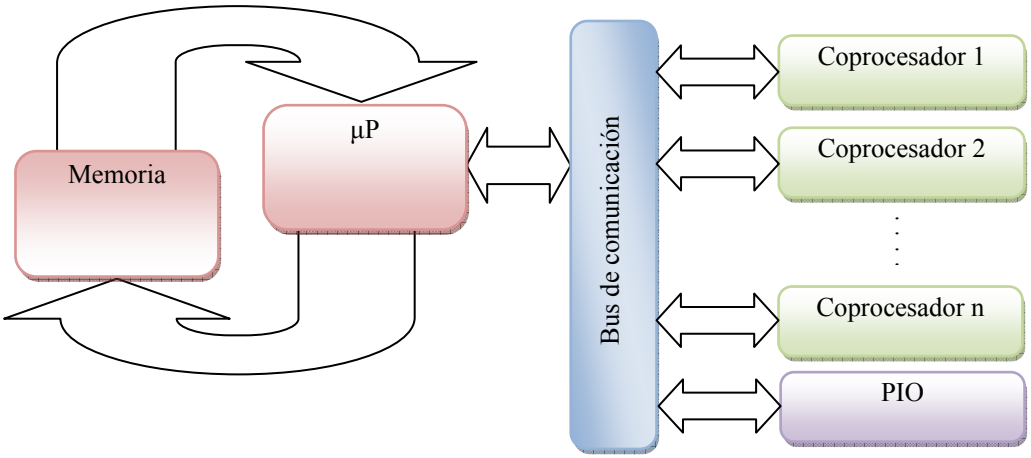


Figura 1: Sistema software-hardware

Si el sistema es complejo podemos disponer de diversos coprocesadores que realizan funciones de forma concurrente entre ellos y con el microprocesador.

Es necesario proveer de un bus de comunicaciones para establecer el intercambio de información entre las diferentes partes.

Para la implementación de este tipo de sistemas es conveniente saber que partes de nuestro algoritmo implican un mayor tiempo de cálculo, analizar si puede paralelizarse y dirigirlas a un coprocesador específico que únicamente resuelve dicha cuestión pero de forma rápida, liberando el procesador que puede realizar diferentes tareas hasta que el coprocesador concluya el cálculo.

2.4 Hardware reconfigurable.

El hardware reconfigurable incorpora bloques lógicos básicos cuya interconexión puede programarse y, por tanto, definir la funcionalidad a voluntad. El máximo exponente del hardware reconfigurable es la FPGA (Field programed gate array), existiendo en el mercado modelos que pueden programarse tantas veces como sea necesario.

Estos elementos nos permiten configurar el funcionamiento de un coprocesador y testear su funcionamiento una y otra vez sin necesidad de modificar la circuitería.

Los lenguajes de descripción hardware (VHDL, Verilog, ABEL) tienen similitudes con los lenguajes de programación de alto nivel, permitiendo un nivel de abstracción elevado cuando se pretende configurar el hardware sin la necesidad de entrar a diseñar a nivel de puerta lógica.

Un microprocesador resulta de la interconexión de miles, cientos de miles o millones de puertas dependiendo de la complejidad del mismo. Podemos, por tanto, definir el funcionamiento de un microprocesador mediante un lenguaje de descripción hardware e implementarlo en una FPGA pudiendo disponer en un mismo dispositivo del microprocesador y los coprocesadores necesarios para el funcionamiento del sistema.

Utilizando el hardware reconfigurable el tiempo de diseño y desarrollo de un sistema software-hardware se reduce considerablemente llegando a aproximarse a los costos de diseñar un sistema todo software.

El presente proyecto pretende aprovechar las ventajas del hardware reconfigurable para implementar el sistema descrito en la introducción.

2.5 Objetivos del proyecto.

Los objetivos del proyecto son:

1. Describir e implementar en VHDL[10] una FPU capaz de operar con números de 32 bits y realizar las siguientes operaciones: suma, resta, producto, cociente, inversa, raíz cuadrada, seno, coseno y tangente.

2. Realizar las pruebas oportunas para comprobar el correcto funcionamiento y verificar la mejora de rendimiento que se obtiene mediante la uso de la FPU diseñada.

Para la comprobación de los resultados será necesario cumplir los siguientes objetivos secundarios:

3. Describir e implementar en VHDL[10] una interface de comunicación con el BUS OPB.
4. Utilizar el bus OPB[5] para la comunicación entre un microprocesador "microblaze" de XIlinx y la FPU según el esquema de la figura 2.

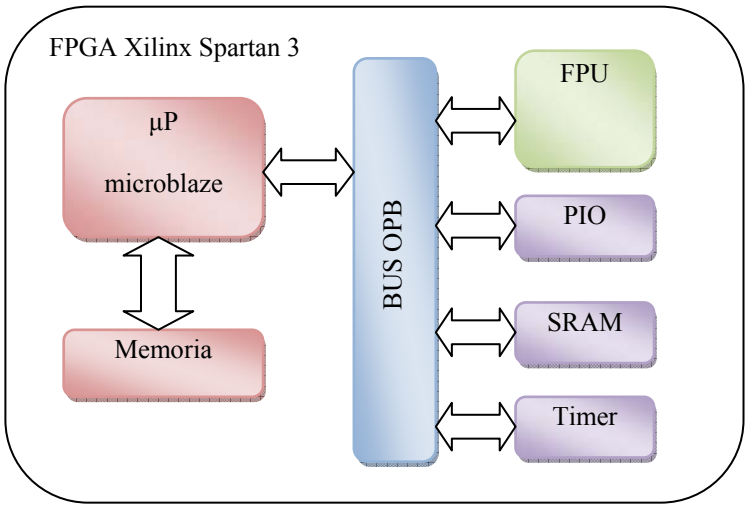


Figura 2: Elementos del sistema

3 Aritmética en coma flotante (Float Point)

3.1 Representación de números en coma flotante

La representación de números binarios en coma flotante es algo parecido a lo que conocemos como notación exponencial de números en base 10.

Los diferentes valores están definidos por tres elementos claramente diferenciados.

- El signo, positivo o negativo.
- El exponente de la base, positivo o negativo.
- La mantisa, un valor decimal comprendido entre 1 y la base de representación.

Un ejemplo de notación exponencial en base 10 es el siguiente:

$$-10^{12} \cdot 2.4367$$

Y un ejemplo de número en coma flotante es el siguiente:

$$+2^{100101} \cdot 1.100101001$$

Para el almacenamiento de números en coma flotante en memorias, su manipulación y la realización de operaciones con microprocesadores se establece la norma IEE 754 que se explica a continuación.

3.2 Norma IEE 754

Esta norma es el estándar más extendido para las computaciones en punto flotante y seguido por muchos microprocesadores, compiladores y unidades de coma flotante. Este estándar define formatos para la representación de números en punto flotante así como números no normalizados (incluyendo el cero) y excepciones como NaN (Not a number) e Infinito.

Se especifican 4 formatos para la representación de números en coma flotante:

- Precisión simple, 32 bits.
- Precisión simple extendida, más de 43 bits (prácticamente no utilizada).
- Precisión doble, 64 bits.
- Precisión doble extendida, más de 79 bits (usualmente se usan 80 bits).

Puesto que la FPU diseñada en este proyecto trabaja con números en coma flotante de precisión simple se definirá únicamente este estándar, los demás pueden consultarse en la propia norma.

3.2.1 Números normalizados en coma flotante de precisión simple

Los tres elementos fundamentales de una representación en coma flotante son: signo, exponente y mantisa;

Signo:

El bit 31 se corresponde al bit de signo y es 1 para números negativos y 0 para números positivos.

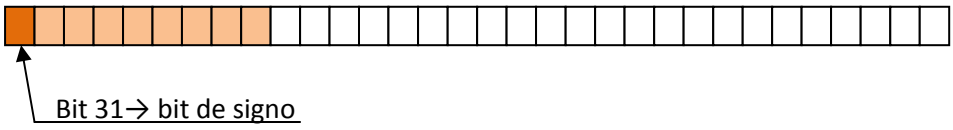


Figura 3: Bit de signo

Exponente:

Está representado por los bits 23 a 30,

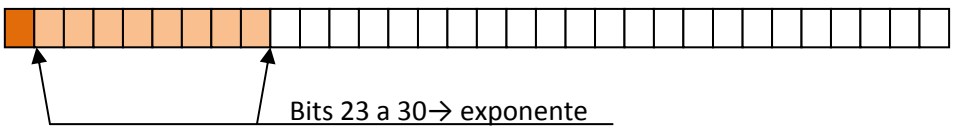


Figura 4: Exponente

Puesto que se ha de representar valores positivos y negativos se utiliza la codificación exceso 127 consistente en que al valor real del exponente se le suma 127 y se representa en 8 bits, para recuperar el valor real se resta dicha cantidad; por ejemplo:

Exponente	Codificación en exceso 127	Representación binaria
-24	-24 + 127 = 103	b 0110 0111
58	+58 + 127 = 185	b 1011 1001

Tabla 1: Codificación en exceso 127

De este modo considerando que con 8 bits podemos realizar 256 combinaciones desde "*b 0000 0000*" hasta "*b 1111 1111*" y que esta última se reserva para casos especiales, como veremos más adelante, los valores entre los que puede encontrarse el exponente serán:

Representación binaria	Decodificación en exceso 127	Exponente
b 0000 0000	0 - 127 = -127	-127
b 1111 1110	254 - 127 = +127	+127

Tabla 2: Representación binaria

Mantisa:

Representada en los bits 0 a 22.

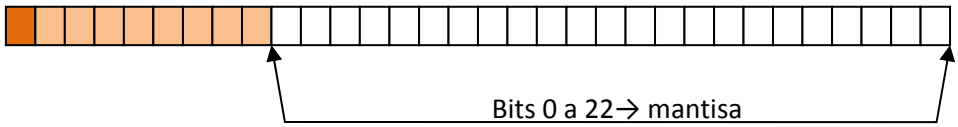


Figura 5: Mantisa

Como se indicó con anterioridad, la mantisa es un número decimal comprendido entre 1 y la base de representación, si en este caso la base es 2 el valor estará comprendido entre 1 y 2.

En cualquier caso la parte entera de la mantisa siempre valdrá "*1*" razón por la cual no es necesario representarla ni almacenarla; consiguiendo de esta forma un bit más de resolución en la mantisa. A este bit se le llama "**bit implícito**" y aunque no es necesario representarlo, sí es necesario considerarlo cuando hay que operar con números expresados con esta norma.

3.2.2 Números no-normalizados en coma flotante de precisión simple

Las definiciones del punto anterior son válidas excepto en los siguientes casos:

Cero:

Se codifica con 32 ceros

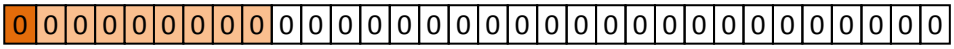
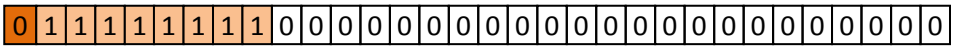


Figura 6: Representación del cero

Infinito:

Se codifica como el exponente con todos los bits en "1" y la mantisa con todos los bits en "0". En este caso hay que diferenciar entre +∞ y -∞.

+∞:



-∞:

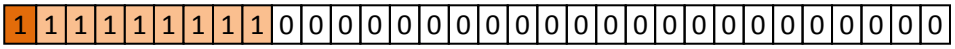


Figura 7: Representación de ± infinito

NaN:

Se codifica como el exponente con todos los bits en "1" y la mantisa con un valor distinto de "0":

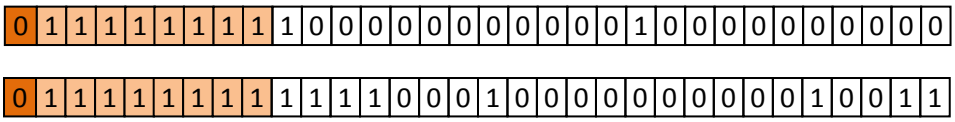


Figura 8: Representación de NaN

Se reservan dos casos:

QNaN: cuando el resultado de una operación no es un número, por ejemplo:

$$\frac{0}{0} \qquad \frac{\pm\infty}{\pm\infty} \qquad \infty - \infty \qquad \sqrt{valor < 0}$$

O bien uno de los operandos está codificado como NaN:

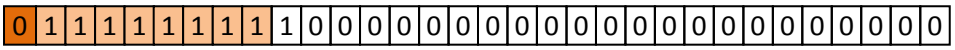


Figura 9: Representación de QNaN

SNaN: cuando una variable no está inicializada, es decir se ha declarado y no se ha asignado ningún valor.

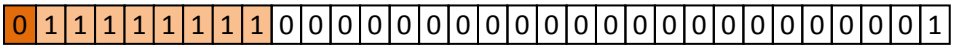


Figura 10: Representación de SNaN

3.2.3 Redondeo

Puesto que estamos tratando con mantisas que contienen números decimales se produce un truncamiento si el número de decimales sobrepasa a la cantidad de bits disponibles para almacenarla. Si se provoca un truncamiento, sin tener en consideración posibles redondeos, el error cometido se va incrementando.

Para determinar el redondeo se añaden a la mantisa tres bits a los que se denomina bits de guarda. Estos bits se consideran tras la operación y su valor determina cómo ha de realizarse el redondeo según estas tres opciones:

Redondeo "Nearest even"

Es el más habitual y utilizado genéricamente en cientos de aplicaciones técnicas y cotidianas, corresponde a redondear al siguiente valor más próximo, por ejemplo:

Valor	Redondeo
3,3234	3,323
-6,1248	6,125
12,0005	12,001

Tabla 3: Redondeo "Nearest even"

Redondeo "Round to zero"

El valor no se redondea, simplemente se trunca, por ejemplo:

Valor	Redondeo
3,3234	3,323
-6,1248	6,124
12,0005	12,000

Tabla 4: Redondeo "Round to zero"

Redondeo "Round up"

El valor se redondea hacia $+\infty$ si es positivo o hacia $-\infty$ si es negativo, por ejemplo:

Valor	Redondeo
3,3234	3,324
-6,1248	-6,125
12,0005	12,001

Tabla 5: Redondeo "Round up"

Redondeo "Round down"

El valor se redondea hacia $-\infty$ si es positivo o hacia $+\infty$ si es negativo, es el caso contraria al anterior, por ejemplo:

Valor	Redondeo
3,3234	3,323
-6,1248	-6,124
12,0005	12,000

Tabla 6: Redondeo "Round down"

En el presente proyecto se ha utilizado el redondeo "Nearest even"

3.2.4 Precisión y representatividad

Este formato de codificación de cifras decimales permite la representación de valores con 6 cifras significativas como por ejemplo 2.14681, $8.14537 \cdot 10^{-12}$, $-3.16714 \cdot 10^{20}$,…

Los límites de los valores a representar son los siguientes $\pm 5{,}87747 \cdot 10^{-39}$ y $\pm 3{,}40282 \cdot 10^{+38}$, los valores inferiores, en valor absoluto, a $5{,}87747 \cdot 10^{-39}$ se consideran 0, los valores superiores a $+3{,}40282 \cdot 10^{+38}$ se consideran $+\infty$ y los valores inferiores a $-3{,}40282 \cdot 10^{+38}$ se consideran $-\infty$.

3.3 Aritmética

Para la realización de operaciones en coma flotante es necesario procesar de forma distinta el valor de los exponentes, el valor del bit de signo y el valor de las mantisas.

En el caso de la mantisa es necesario añadir el bit implícito, el bit de desbordamiento, el de, "overflow" y los de guarda.

La mantisa se procesa de la siguiente forma:

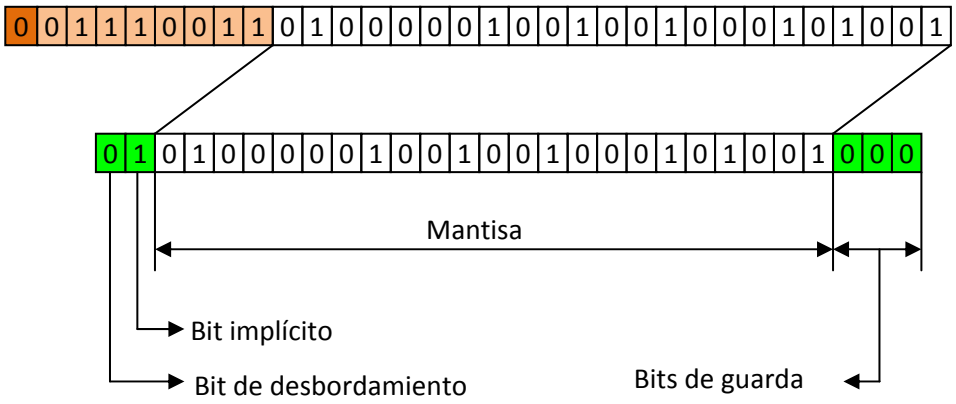


Figura 11: Bits adicionales

Convirtiéndose en un número binario de 28 bits.

Sabiendo que el valor de la mantisa se encuentra entre:

$$1{,}0000000000000000000000 \rightarrow 1{,}1111111111111111111111$$

es fácil acotar los posibles resultados que se obtendrán de las operaciones.

En cuanto al bit de signo y el exponente no precisan manipulación alguna previa a la realización de operaciones.

3.3.1 Adición y sustracción

Para la realización de sumas es necesario que ambos operandos tengan el mismo exponente, si no es el caso, ha de desplazarse uno de ellos con el objeto de conseguir este objetivo.

El desplazamiento ha de realizarse en el número de valor absoluto inferior con objeto de que la pérdida de bits significativos sea lo menor posible y, el mencionado desplazamiento, se realiza hacia la derecha en tantos bits como diferencia de valor exista entre los exponentes. Si la diferencia fuese mayor de 28 bits el operador de menor valor absoluto puede considerarse cero pues no afectará al resultado.

En aritmética decimal esta operación equivale a alinear los números por el punto decimal antes de operarlos.

Una vez “alineados”, el exponente del resultado parcial es el exponente del número de mayor valor absoluto y la mantisa del resultado es la suma o la resta de mantisas en función de los signos.

Ejemplo:

$$12.546 + (-0.465) = 12.079 \quad \textbf{(1)}$$

Operando	Signo	Exponente		Mantisa con los bits extra añadidos
12,546	0	3	10000010	011001000101111000110101000
-0,465	1	-2	01111101	011101110000101000111101100
Desplazamiento del exponente		5		
12,546	0	3	10000010	011001000101111000110101000
-0,467	1	3	10000010	000000111011100001010001111
Puesto que los signos son diferentes se resta		3	10000010	011000001010010111100011001

Tabla 7: Ejemplo de suma

Se indican en verde los bits de acarreo, implícito y de guarda, en azul los que se añaden al realizar el desplazamiento.

El resultado sólo queda pendiente de redondeo y normalización.

Si se desea realizar una resta se cambia el signo del segundo operador y se aplica el mismo algoritmo.

3.3.2 Producto

En el producto han de operarse los signos, los exponentes y las mantisas por separado realizando las siguientes operaciones.

Signo:

Se aplica la regla matemática de producto de signos y que se realiza mediante la función lógica XOR.

Regla matemática			Equivalencia binaria		
+	+	+	0	0	**0**
+	-	-	0	1	**1**
-	+	-	1	0	**1**
-	-	+	1	1	**0**

Tabla 8: Procesado de signos

Exponentes:

Se suman los exponentes teniendo en consideración que la codificación incluye un exceso 127, la operación a realizar es la siguiente:

$$Exp_A + Exp_B - 127 = Exp_{Res} \quad (2)$$

Mantisas:

Se multiplican las mantisas y, sabiendo que las mantisas están acotadas entre 1 y 1.99999 el resultado máximo se dará cuando ambas mantisas tengan el valor 1.99999. El producto como máximo resultará 3.99999.

El resultado mínimo se dará cuando ambas mantisas tengan el valor 1 el producto como mínimo resultará 1.

Dado que el producto de dos números de 28 bits resulta un número de 56 bits se trunca eliminando todos los bits de menor peso que los de redondeo.

Operación:

Operando	Signo	Exponente		Mantisa con los bits extra añadidos
12,546	0	3	10000010	01100100010111100011010100**0**
-0,465	1	-2	01111101	01110111000010100011110110**0**0
-5,83389	1	1	10000001	10111010101011110011100110**1**1

Tabla 9: Ejemplo de producto

El resultado sólo queda pendiente de redondeo y normalización.

3.3.3 Cociente e inversa

En el cociente han de operarse los signos, los exponentes y las mantisas por separado realizando las siguientes operaciones.

Signo:

Se aplica la regla matemática de producto de signos y que se realiza mediante la función lógica XOR, de igual modo que se realiza en la multiplicación.

Exponentes:

Se suman los exponentes teniendo en consideración que la codificación incluye un exceso 127, la operación a realizar es la siguiente:

$$Exp_A - Exp_B + 127 = Exp_{Res} \quad (3)$$

Mantisas:

Se dividen las mantisas, el resultado máximo se dará cuando la mantisa del numerador sea máxima (1,99999) y la del denominador mínima (1) el resultado estará acotado por 1,99999 en el límite superior.

El resultado mínimo se dará cuando el numerador tenga el valor mínimo (1) y el denominador el valor máximo (1,99999), en este caso el resultado será 0,5 como máximo.

El cociente se realiza hasta que se alcanza el número de bits que el sistema de representación precisa.

Operación:

Operando	Signo	Exponente		Mantisa con los bits extra añadidos
12,546	0	3	10000010	01100100010111100011010100 000
-0,465	1	-2	01111101	01110111000010100011110110 00
-26,9806	1	5	10000100	00110101111101100001011011 11

Tabla 10: Ejemplo de cociente

Para la realización de la inversa se fija 1 como numerador:

Operando	Signo	Exponente		Mantisa con los bits extra añadidos
1	0	0	01111111	01000000000000000000000000 0
-0,465	1	-2	01111101	01110111000010100011110110 00
-2,15054	1	2	10000001	00100010011010001001101000 10

Tabla 11: Ejemplo de inversa

Ambos resultados sólo quedan pendientes de redondeo y normalización.

3.3.4 Raíz cuadrada

En la raíz cuadrada han de operarse los signos, los exponentes y las mantisas por separado realizando las siguientes operaciones.

Signo:

Sólo pueden calcularse raíces cuadradas de argumento mayor o igual a cero, por lo tanto si el argumento es negativo no es necesario calcular ningún resultado.

Exponentes:

El exponente ha de dividirse por dos, esta operación consiste en un desplazamiento hacia la derecha de los bits que representan el exponente.

Es primordial que el exponente sea par, por tanto si no lo es ha de desplazarse la mantisa y corregir el exponente para que lo sea, de esta forma se puede realizar la división entera del exponente entre dos.

Teniendo lo anterior y que el exponente está codificado en exceso 127 la operación es la siguiente.

$$\left(\frac{\text{Exp}-127}{2}\right)+127=\text{Exp}_{\text{Res}} \qquad \textbf{(4)}$$

Mantisas:

Se realiza la raíz cuadrada de la mantisa, el resultado máximo se dará cuando la mantisas del numerador sea máxima (1,99999) y estará acotado por 1,41421 en el límite superior.

El resultado mínimo se dará en el caso contrario el numerador en el valor mínimo (1) y estará acotado en 1.

Operación:

Operando	Exponente		Mantisa con los bits extra añadidos
12,546	3	10000010	01100100010111100011010100 00
Exponente sin codificación	3	00000011	
Corrección del exponente y desplazamiento de la mantisa	4	00000100	00 1100100010111100011010100 0
3,54203	2	00000010	00 1110001010110000101011000 0
Re codificación del exponente	2	10000001	00 1110001010110000101011000 0

Tabla 12: Ejemplo de raíz cuadrada

El resultado sólo queda pendiente de redondeo y normalización.

3.3.5 Funciones trigonométricas

3.3.5.1 Algoritmo CORDIC

El algoritmo CORDIC (Coordinate Rotation Digital Computer), propuesto en 1959 por Jack E.Volder, permite el cálculo de funciones trigonométricas de una forma muy simple, rápida y precisa. Su base teórica es la siguiente:

Si disponemos de un vector de módulo 1 situado sobre el eje X sus coordenadas (X_0, Y_0) valen (1, 0):

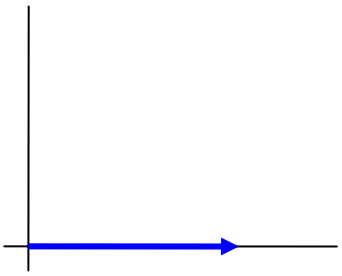


Figura 12: Vector de inicio del CORDIC

y lo giramos un determinado ángulo α_0:

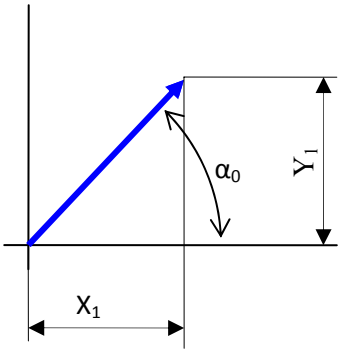


Figura 13: Rotación del vector

las nuevas coordenadas se calculan mediante la siguiente fórmula:

$$X_1 = X_0 \cdot \cos(\alpha_0) - Y_0 \cdot \operatorname{sen}(\alpha_0)$$
$$Y_1 = Y_0 \cdot \cos(\alpha_0) + X_0 \cdot \operatorname{sen}(\alpha_0) \quad \textbf{(5)}$$

si se extrae el coseno como factor común obtenemos:

$$X_1 = \cos(\alpha_0) \cdot \left(X_0 - Y_0 \cdot \frac{\operatorname{sen}(\alpha_0)}{\cos(\alpha_0)} \right) = \cos(\alpha_0) \cdot \left(X_0 - Y_0 \cdot \tan(\alpha_0) \right)$$
$$Y_1 = \cos(\alpha_0) \cdot \left(Y_0 + X_0 \cdot \frac{\operatorname{sen}(\alpha_0)}{\cos(\alpha_0)} \right) = \cos(\alpha_0) \cdot \left(Y_0 + X_0 \cdot \tan(\alpha_0) \right) \quad \textbf{(6)}$$

Si se aplica otro giro al vector puede calcularse de nuevo las coordenadas aplicando la misma fórmula:

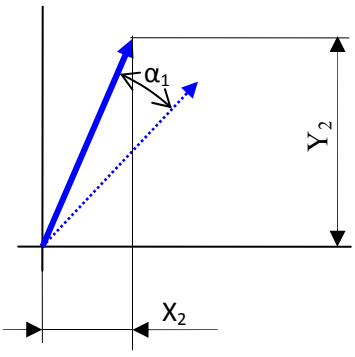


Figura 14: Nueva rotación

$$X_2 = \cos(\alpha_1) \cdot \left(X_1 - Y_1 \cdot \tan(\alpha_1) \right)$$
$$Y_2 = \cos(\alpha_1) \cdot \left(Y_1 + X_1 \cdot \tan(\alpha_1) \right) \quad \textbf{(7)}$$

podemos considerar un pseudogiro:

$$X_2 = K\cdot\left(X_1 - Y_1\cdot\tan(\alpha_1)\right)$$
$$Y_2 = K\cdot\left(Y_1 + X_1\cdot\tan(\alpha_1)\right) \quad \textbf{(8)}$$

afectado por una constante:

$$K = K_1 = \cos(\alpha_1) \quad \textbf{(9)}$$

de esta forma se puede alcanzar cualquier ángulo (con límite en el sumatorio de los valores de la serie) aplicando de forma reiterativa la misma operación con ángulos cada vez menores de la siguiente forma:

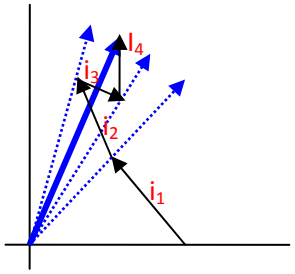


Figura 15: Rotaciones sucesivas

que de forma genérica se representa como:

$$X_{(n+1)} = K\cdot\left(X_n - Y_n\cdot S\cdot\tan(\alpha_n)\right)$$
$$Y_{(n+1)} = K\cdot\left(Y_n + X_n\cdot S\cdot\tan(\alpha_n)\right)$$
$$K = \cos(S\cdot\alpha_0)\cdot\cos(S\cdot\alpha_1)....\cdot\cos(S\cdot\alpha_n) \quad \textbf{(10)}$$

y el ángulo sigue la siguiente secuencia:

$$Z_{(n+1)} = \left(Z_n + S\cdot E_n\right) \quad \textbf{(11)}$$

donde "Z" es el ángulo acumulado, "E" es el ángulo de la última iteración y "S" es una función cuyo valor es 1 o -1 según el ángulo acumulado sea menor o mayor que el ángulo objetivo.

La serie de ángulos utilizada para la implementación del algoritmo puede ser cualquiera con la única condición que los valores de los ángulos sean cada vez menores.

Ejemplo:

Calculo del seno de 0,65 rad mediante la iteración, se toma como ángulos la siguiente serie:

$$Zn_{n=0..32} = \frac{\pi}{2^n + 2} = (n\ es\ el\ número\ de\ la\ iteración) = \frac{\pi}{3}, \frac{\pi}{4}, \frac{\pi}{6}, \frac{\pi}{10}, \frac{\pi}{18}, ... \quad \textbf{(12)}$$

Con esta serie de ángulos el valor de la constante K es:

$$K = \prod_{n\ 0}^{32} \cos\left(\frac{\pi}{2^n + 2}\right) = 0{,}285118106 \qquad \textbf{(13)}$$

La evolución del ángulo es la que se muestra en la figura 16:

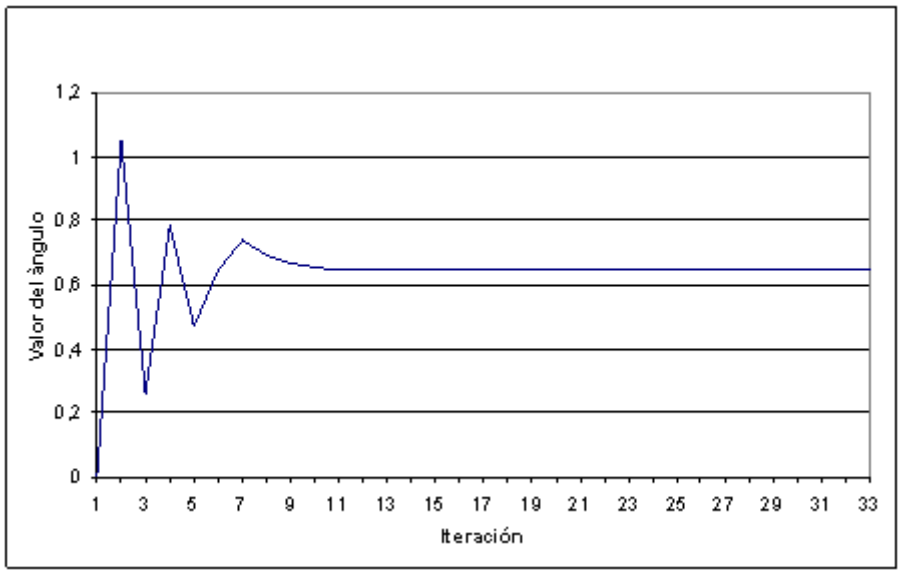


Figura 16: Evolución del ángulo en las iteraciones

Resultado previo a la aplicación de la constante K:

Coseno	Seno	
0,796083799	0,605186406	Solución real
2,792119417	2,122581461	Solución por iteraciones

Tabla 13: Test del CORDIC

Resultado tras la aplicación de l factor K:

Coseno	Seno	
0,796083799	0,605186406	Solución real
0,796083799	0,605186405	Solución por iteraciones

Tabla 14: Test del CORDIC corregido

Se aprecia como el resultado es muy preciso con 32 iteraciones.

La implementación de este algoritmo, a priori, necesita sumadores, comparadores y multiplicadores, la modificación propuesta por Jack E.Volder, que recibe el nombre de CORDIC, establece los siguientes condicionantes que mejoran considerablemente la implementación:

Los ángulos siguen la siguiente serie, que al ser constantes pueden implementarse en una memoria ROM:

$$Zn_{n=0..32} = \arctan\left(2^{-n}\right) = (n \quad es \quad el \quad número \quad de \quad la \quad iteración) = \\ = \arctan\left(2^{-0}\right), \arctan\left(2^{-1}\right), \arctan\left(2^{-2}\right), ... \qquad \textbf{(14)}$$

de esta forma los productos,

$$\left(Y_n \cdot S \cdot \tan(\alpha_n)\right) \quad y \quad \left(X_n \cdot S \cdot \tan(\alpha_n)\right) \tag{15}$$

que aparecen en la fórmula del algoritmo CORDIC, pueden resolverse con un desplazamiento a la derecha de valor "n" y no es necesario el uso de multiplicadores para realizarlo.

Por otro lado el valor de K es el siguiente para esta serie de ángulos:

$$K = \prod_{n\ 0}^{32} \cos\ \left(\arctan\left(2^{-n}\right)\right) = 0{,}607252935 \tag{16}$$

Se establece el vector de inicio en:

$$\left(K\ ,\ 0\right) = \left(0{,}607252935\ ,\ 0\right) \tag{17}$$

evitando de esta forma la corrección debida al factor K, y evitando un multiplicador adicional.

Ejemplo con los condicionantes explicados:

Calculo del seno de 0,65 rad mediante la iteración (véase la evolución del ángulo en la figuta17)

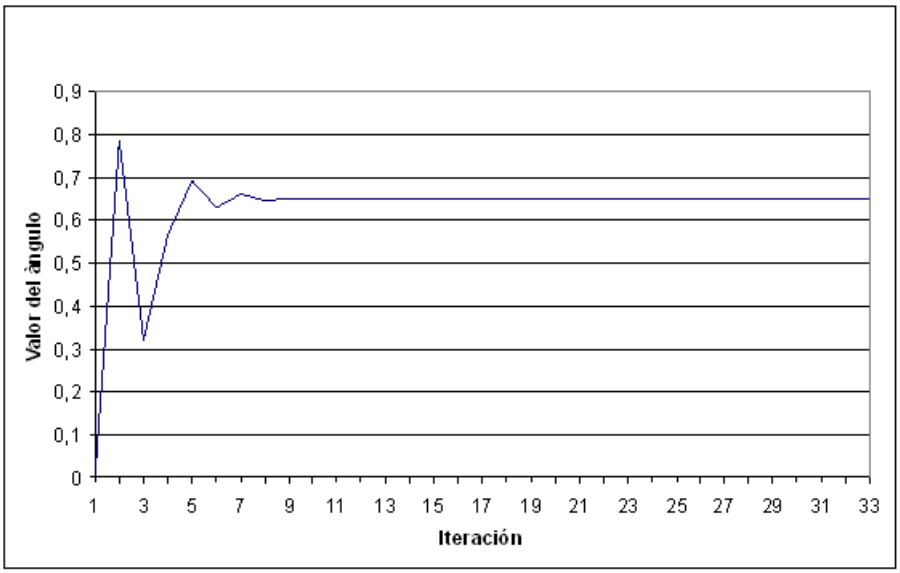


Figura 17: Evolución del ángulo en las iteraciones con la serie CORDIC

Resultado tras la aplicación del factor K:

Coseno	Seno	
0,796083799	0,605186406	Solución real
0,796083799	0, 605186406	Solución CORDIC

Tabla 15: Test del CORDIC con la serie real

3.3.5.2 Senos y cosenos

Mediante del algoritmo explicado anteriormente se puede conseguir no sólo calcular el seno y el coseno sino calcularlos de forma simultánea.

Para la implementación de este algoritmo es necesario convertir el valor del ángulo en un entero de 32 bits que facilite la comparación, el desplazamiento, las sumas y las restas.

Posteriormente se transforma de nuevo en un número expresado en coma flotante de 32 bits. Esta limitación condiciona los valores máximos y mínimos que pueden entrarse al algoritmo de CORDIC.

3.3.5.3 Tangente

La tangente se obtiene mediante la división del seno y el coseno obtenido por CORDIC.

3.3.6 Redondeo y normalización

Como se ha comentado con anterioridad, el resultado de las operaciones da valores no normalizados y no redondeados debido a que para operar es necesario añadir los bits de desbordamiento e implícito por un lado y los de guarda por otro.

Tras las operaciones es necesario normalizar el resultado para que cumplan la norma IEEE-754, el proceso es el siguiente:

Normalización:

Para realizarla es necesario desplazar la mantisa hasta que el primer bit que vale "1" comenzando por la izquierda se sitúe en la posición del bit implícito, en el ejemplo marcado en rojo. Este desplazamiento ha de compensarse en el exponente de forma que: si se desplaza a la izquierda es necesario aumentar el exponente y si se desplaza a la izquierda es necesario disminuirlo. Los bits añadidos siempre son "0".

Ejemplo de corrección a la izquierda.

Valor	Signo	Exponente		Mantisa con los bits extra añadidos
3,54203	0	2	10000001	00111000101011000010101100000
Valor	Signo	Exponente corregido		Mantisa desplazada un bit a la izquierda
3,54203	0	1	10000000	01110001010110000101011000000

Tabla 16: Normalización a la izquierda

Ejemplo de corrección a la derecha.

Valor	Signo	Exponente		Mantisa con los bits extra añadidos
-5,83389	1	1	10000001	10111010101011110011100110 11
Valor	Signo	Exponente corregido		Mantisa desplazada un bit a la derecha
-5,83389	1	2	10000010	01011101010101111001110011 01

Tabla 17: Normalización a la derecha

Redondeo:

Se realiza tras la normalización y puede ser “al alza”, cuando el valor de los bits de guarda supera estrictamente el valor 3=”011”, o a la baja, si no lo supera, según la siguiente tabla resumen:

Tipo de redondeo	Valor de los bits de guarda		
Al la baja	0	0	0
	0	0	1
	0	1	0
	0	1	1
Al alza	1	0	0
	1	0	1
	1	1	0
	1	1	1

Tabla 18: Redondeo

El redondeo a la baja elimina los bits de guarda sin afectar a la mantisa tal y como se aprecia en el siguiente ejemplo.

Valor	Signo	Exponente		Mantisa con los bits extra añadidos
3,54203	0	1	10000000	01110001010110000101011000 00
Valor	Signo	Exponente		Mantisa redondeada
3,54203	0	1	10000000	01110001010110000101011000

Tabla 19: Ejemplo de redondeo a la baja

El redondeo al alza elimina los bits de guarda y aumenta una unidad la mantisa tal y como se aprecia en el siguiente ejemplo:

Valor	Signo	Exponente		Mantisa con los bits extra añadidos
-2,15054	1	1	10000000	01000100110100010011010001 00
Valor	Signo	Exponente		Mantisa redondeada
-2,15054	1	1	10000000	01000100110100010011010001

Tabla 20:Ejemplo de redondeo al alza

Tras el redondeo y la normalización puede expresarse el número según la norma establecida únicamente eliminando el bit de desbordamiento y el implícito y concatenando los valores en binario del signo, el exponente y la mantisa.

Valor	Signo	Exponente		Mantisa con los bits extra eliminados
-2,15054	1	1	10000000	01000100110100010011010001
Valor	Codificación IEEE-754			
-2,15054	11000000000100110100010011010001			

Tabla 21: Representación tras el redondeo

4 Diseño y estructura interna de la unidad de coma flotante

4.1 Consideraciones previas

Los algoritmos empleados para la realización de operaciones básicas en coma flotante se han elegido atendiendo a los siguientes parámetros:

- **La velocidad de respuesta del algoritmo:** un algoritmo sumamente complejo, que implique un circuito combinacional muy extenso, provoca que la latencia de dicho circuito sea grande y, por tanto, la frecuencia de trabajo pequeña.

- **Segmentación en bloques pequeños:** Siempre es mejor la descomposición de los algoritmos en pequeños bloques combinacionales que se secuencian mediante una máquina de estados finitos. La complejidad de dichos bloques es menor aportando las siguiente ventajas:

    - Definición sencilla del funcionamiento, puesto que el algoritmo es sencillo.

    - Puede simularse y depurarse con facilidad, la localización de cualquier "*glitch*" es rápida. Puede garantizarse el correcto funcionamiento de estos bloques y avanzar sobre seguro en el desarrollo de un sistema más complejo.

    - La latencia es pequeña, puede trabajase a frecuencias más grandes y compensar de este modo el hecho de que son necesarios más ciclos de reloj para completar el algoritmo.

    - Por la misma razón, un bloque rápido limita en menor medida la frecuencia máxima de trabajo de todo el sistema.

    - Los diseños pueden secuenciarse mediante la tecnología "*pipe line*" mejorando drásticamente la respuesta del sistema y pueden ejecutarse de forma concurrente si fuese necesario.

- **El área ocupada:** cuando se trabaja con FPGA's la limitación estriba en la cantidad de puertas que se consumen para implementar uno u otro algoritmo. Los algoritmos que consumen menos puertas permiten un mejor aprovechamiento del dispositivo.

Ejemplo:

Un comparador de 32 bits con registros de entrada y salida, como el de la figura 18, se obtienen los siguientes datos de implementación:

- Área de FGA consumida 67 "Slices"

- La frecuencia máxima de funcionamiento está limitada por el "path" crítico y es de 130.344MHz.

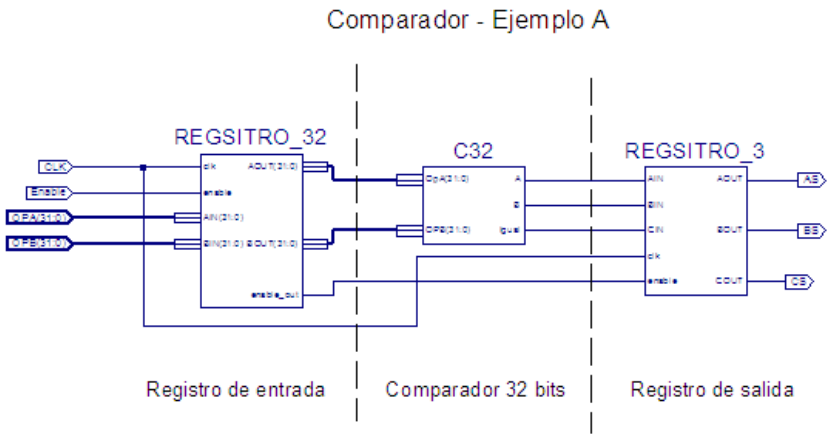


Figura 18: Comparador 32 bits

Si el sistema se descompone en dos comparadores de 16 bits más una lógica inicial que separe los valores de 32 bits en sus correspondientes números de 16 bits alto y bajo para que puedan ser comparados por separado y una lógica final que interprete el resultado de los dos comparadores, como el de la figura 19, se obtienen los siguientes resultados de implementación:

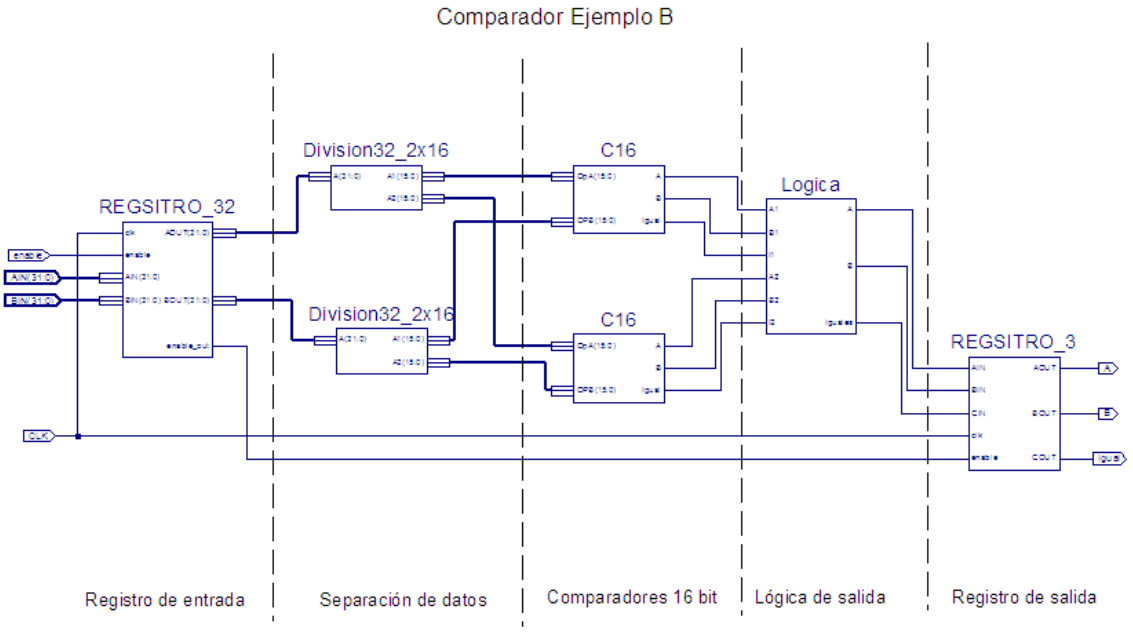


Figura 19: Comparador 2 x 16 bits

- Área de FGA consumida 70 “Slices”
- La frecuencia máxima de funcionamiento es de 428.816MHz

Puede apreciarse que el área consumida en la implementación es prácticamente la misma y además conseguimos cuadruplicar la frecuencia máxima de funcionamiento.

4.2 Layout general

El sistema se ha estructurado en bloques independientes, habitualmente llamados entidades en VHDL, que realizan una operación cada uno y unos módulos de pre-procesado y control lógico que se encargan de decodificar la operación, verificar las excepciones, secuenciar el funcionamiento en caso de ser necesaria la intervención de una o más operaciones y normalizar el valor antes de ser devuelto.

Se procederá a explicar bloque por bloque, de más alto nivel a más bajo nivel, como se ha procedido a la implementación siguiendo el camino que los datos siguen en el proceso.

4.3 Sistema principal "FPU"

4.3.1 Puertos de entrada y salida

El sistema principal recibe el nombre de FPU. Es la entidad de más alto nivel y dispone de los siguientes puertos de entrada y salida:

Entradas		
Nombre	Tipología	Descripción
operacion	Bus 8 bits	Operación a realizar
op_a	Bus 32 bits	Primer operando, valor expresado en coma flotante
op_b	Bus 32 bits	Segundo operando, valor expresado en coma flotante
inicio	Bit	Pulso, de 1 ciclo de reloj de ancho, que inicia el funcionamiento
clk	Bit	Reloj general del sistema
reset	Bit	'Reset' general de la unidad de coma flotante
Salidas		
Nombre	Tipología	Descripción
resultado	Bus 32 bits	Resultado de la operación solicitada
fin	Bit	Operación finalizada, esta señal se desactiva con el 'reset' general de la FPU.

Tabla 22: Puertos de la entidad FPU

El esquema de implementación puede consultarse en el Anexo 1.1.

Diagrama de flujo

El funcionamiento se inicia con un pulso en la entrada "*inicio*", ha de haberse escrito los operandos "*op_a*" y "*op_b*" así como la operación deseada "*operacion*" con anterioridad, si no el sistema reutiliza los valores que existiesen. Tras la operación se obtiene el resultado y se activa la señal "*fin*", estos valores permanecerán hasta que se produzca el "*reset*" de la unidad (Véase figura 20).

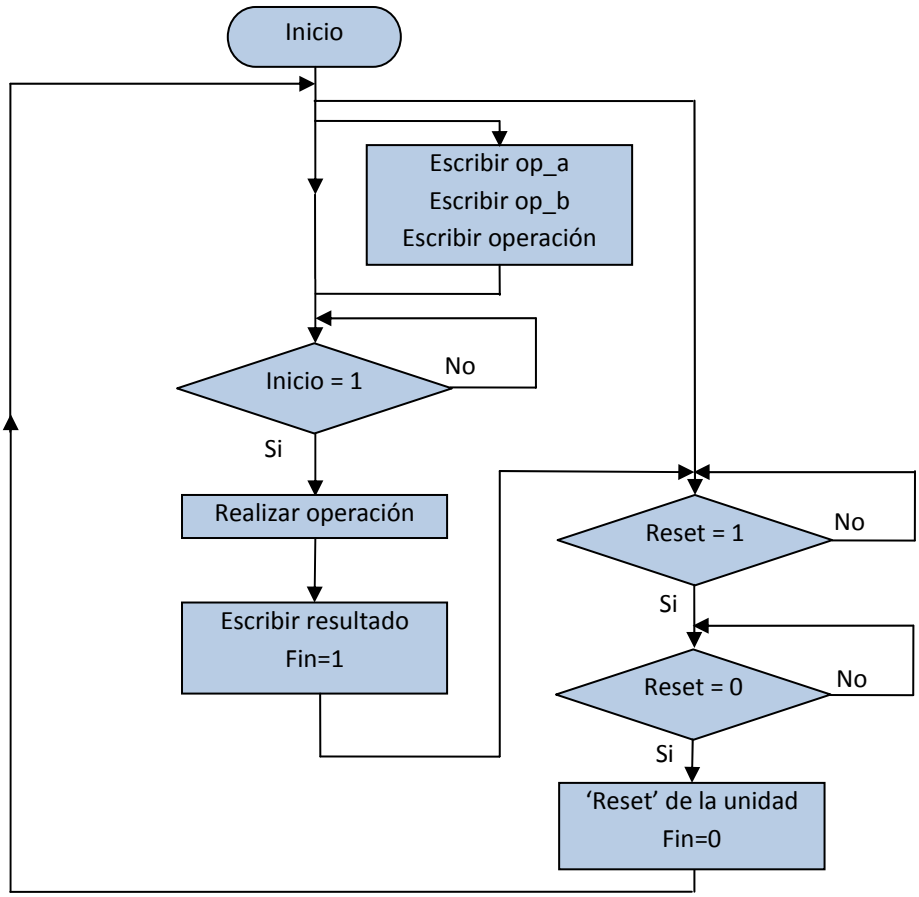


Figura 20: Diagrama de flujo de la entidad FPU

La duración del proceso depende de la operación deseada y se especifica en el apartado de resultados.

El sistema FPU puede realizar las siguientes operaciones que se codifican según la siguiente tabla:

Operación	Codificación
Suma	1
Resta	2
Producto	3
Cociente	4
Inversa	5
Raíz cuadrada	6
Seno	7
Coseno	8
Tangente	9

Tabla 23: Codificación de las operaciones

El esquema de implementación puede consultarse en el anexo 1.1.

4.4 Estructura interna de "FPU"

El sistema principal está dividido en tres bloques: "Control_lógico", "FPU_in", y "FPU_normalizar" y puertas lógicas adicionales, la interconexión entre estos tres bloques puede verse en el anexo 1.2.

4.4.1 Bloque "Control_logico"

Este bloque se encarga de gestionar el flujo de la información, direccionando los operadores de entrada, "*op_a*" y "*op_b*", hacia el bloque "FPU_in", direcciona la operación programada y secuencia el funcionamiento del sistema en operaciones iterativas como el caso del cálculo de la tangente.

Los puertos de entrada y salida son los siguientes:

Entradas		
Nombre	Tipología	Descripción
operacion	Bus 8 bits	Operación a realizar
op_a	Bus 32 bits	Primer operando, valor expresado en coma flotante
op_b	Bus 32 bits	Segundo operando, valor expresado en coma flotante
inicio	Bit	Pulso, de 1 ciclo de reloj de ancho, que inicia el funcionamiento
clk	Bit	Reloj general del sistema
reset	Bit	'Reset' general de la unidad de coma flotante
fin_cordic	Bit	Final del algoritmo CORDIC
res_1	Bus 32 bits	Resultado parcial 1, valor expresado en coma flotante
res_2	Bus 32 bits	Resultado parcial 2, valor expresado en coma flotante
cero	Bit	Resultado de la operación previa es cero
infinito	Bit	Resultado de la operación previa es infinito
NaN	Bit	Resultado de la operación previa es NaN
Salidas		
Nombre	Tipología	Descripción
habilita_fin	Bit	Permite el final de la operación
reset_FPU	Bit	Reset de la unidad de cálculo, pulso de 1 ciclo de reloj.
op_a_out	Bus 32 bits	Primer operando parcial, valor expresado en coma flotante
op_b_out	Bus 32 bits	Segundo operando parcial, valor expresado en coma flotante
inicio_fpu	Bit	Iniciar el cálculo, pulso de 1 ciclo de reloj.
operacion_out	Bus 8 bits	Operación parcial.

Tabla 24": Puertos de la entidad "Control lógico

Diagrama de flujo (véase figura 21)

Se actúa en función de la operación requerida y tras un pulso en la entrada "*inicio*". Si es inversa, se programa el valor "*1*" expresado en coma flotante como "*op_a_out*" y el valor recibido como "*op_a*" pasa a ser el "*op_b_out"* y en *"operación_out"* se programa una división. Si es la tangente se opera tal y como se indica en la siguiente página y si es cualquiera otra operación únicamente se dirigen los datos "*op_a*" , "*op_b*" y "*operacion*" a las salidas "*op_a_out*", "*op_b_out*" y *"operación_out"* respectivamente. Este proceso dura un ciclo de reloj.

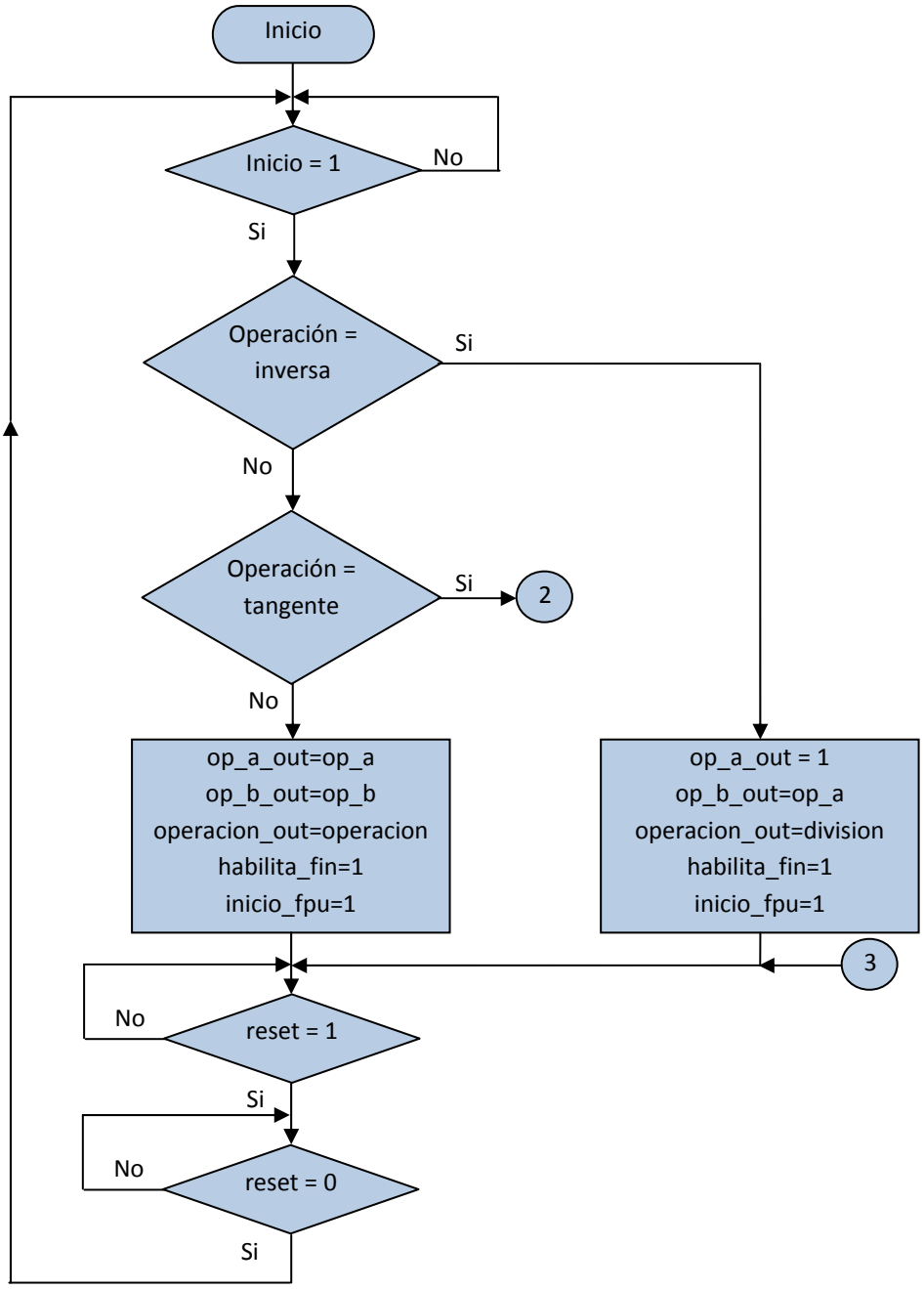


Figura 21: Diagrama de flujo de "Control lógico"

En caso de tratarse de una tangente (véase figura 22) se procede programando la operación seno y bloqueando la salida del pulso que indica el final de la operación. Una vez resuelto el seno y el coseno, que se leen de las entradas "res_1" y "res_2" respectivamente, se resetea la unidad FPU_in, se programa un cociente en "operación_out" cuyo resultado nos dará la tangente deseada.

Este proceso añade 4 ciclos de reloj a parte de los necesarios para la realización del CORDIC y del cociente.

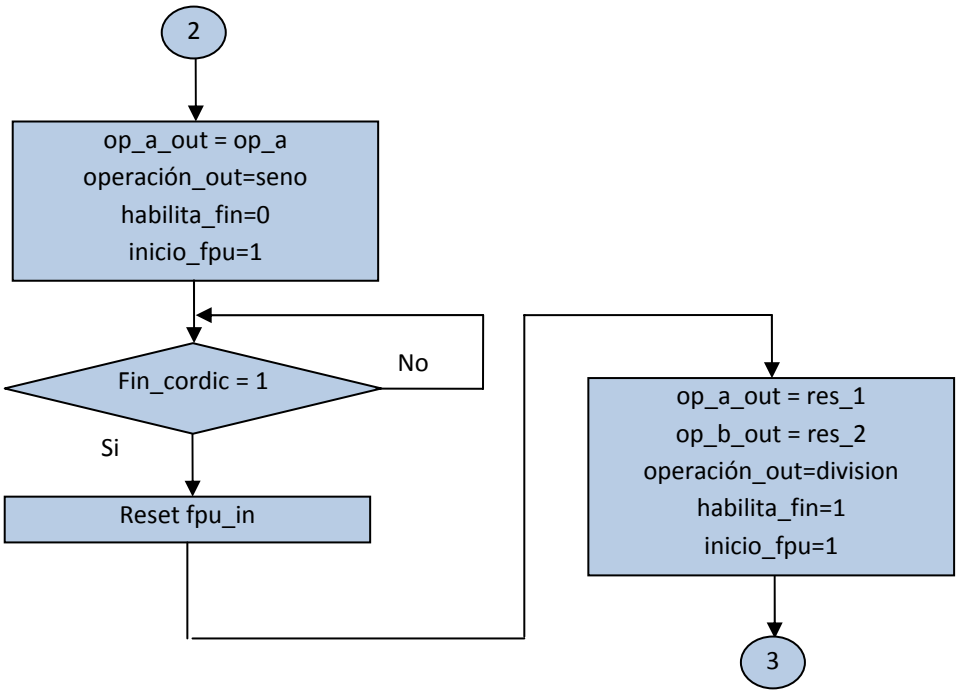


Figura 22: Diagrama de flujo de la tangente

Una vez concluidos los dos pasos anteriores se espera el "*reset*" externo de toda la unidad.

4.4.2 Bloque "FPU_normalizar"

La función de este bloque es convertir los valores obtenidos de la operación y que incluyen el bit implícito, el de desbordamiento y los de guarda. Este bloque procede también al redondeo del valor.

Si el resultado de la operación previa es un valor no normalizado, se devuelve su correspondiente codificación. Si el resultado de la operación previa es uno se devuelve la codificación en coma flotante del valor 1 que es "00111111100000000000000000000000". Si el resultado es infinito se conserva el signo del resultado distinguiendo entre $+\infty$ y $-\infty$.

Entradas		
Nombre	Tipología	Descripción
Inicio	Bit	Pulso, de 1 ciclo de reloj de ancho, que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	'Reset' general de la unidad de coma flotante
uno	Bit	Resultado de la operación previa es uno
cero	Bit	Resultado de la operación previa es cero
infinito	Bit	Resultado de la operación previa es infinito
NaN	Bit	Resultado de la operación previa es NaN
Valor	Bus 37 bits	Valor de la operación previa con bits extras y sin redondear
Salidas		
Nombre	Tipología	Descripción
resultado	Bus 32 bits	Resultado final normalizado.
fin	Bit	Indica que se ha acabado el proceso, se desactiva con el reset.

Tabla 25: Puertos de la entidad "FPU_normalizar"

Diagrama de flujo

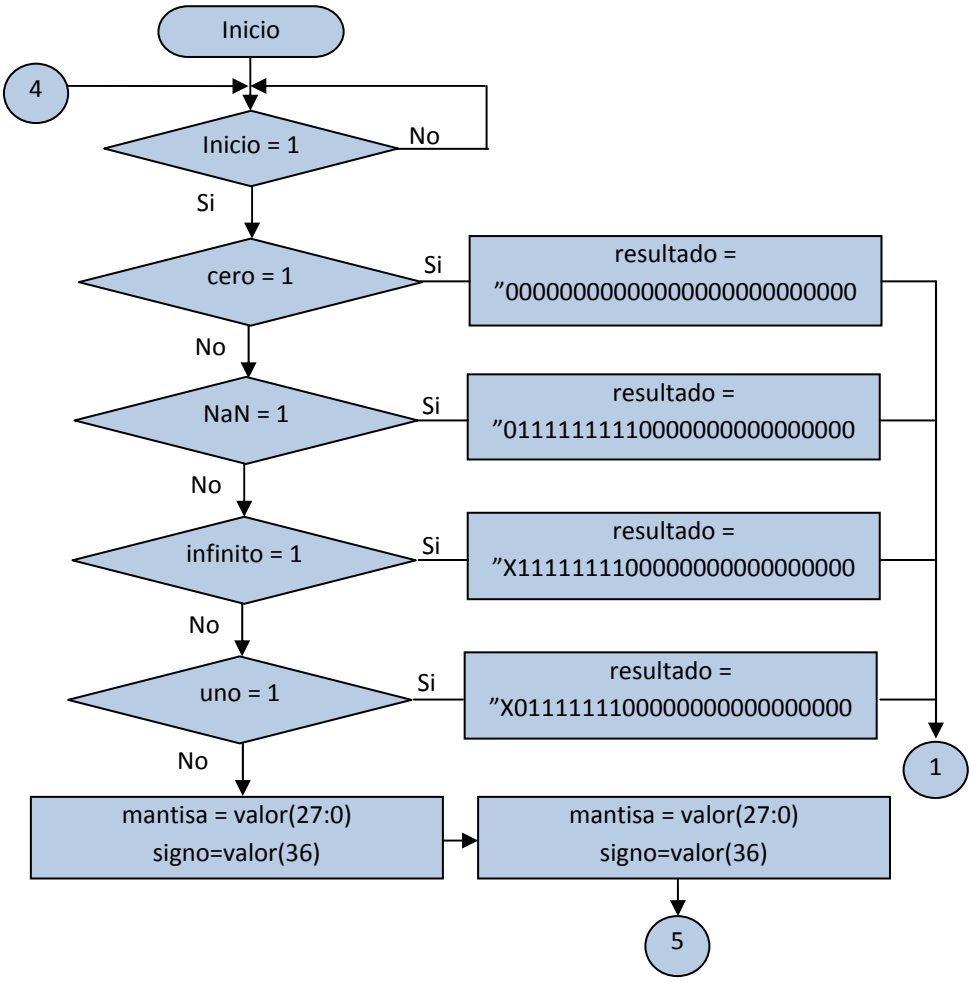


Figura 23: Diagrama de flujo de la entidad "FPU_normalizar" (Excepciones)

Tras el pulso de inicio, se comprueba si ha ocurrido alguna excepción y se escribe el resultado correspondiente, en caso de que no haya ocurrido se separa el valor en: signo, exponente y resultado para procesarlos por separado. Estas operaciones se completan en un ciclo de reloj.

El siguiente paso consiste en localizar el bit de mayor peso de la mantisa cuyo valor es '1' y que a la postre será el bit implícito.

Este proceso se realiza comprobando el bit 27 de la variable mantisa y, si es cero, se desplaza la mantisa hacia la izquierda hasta que sea uno. Es necesario saber cuántos desplazamientos de han producido para corregir el exponente en el siguiente paso.

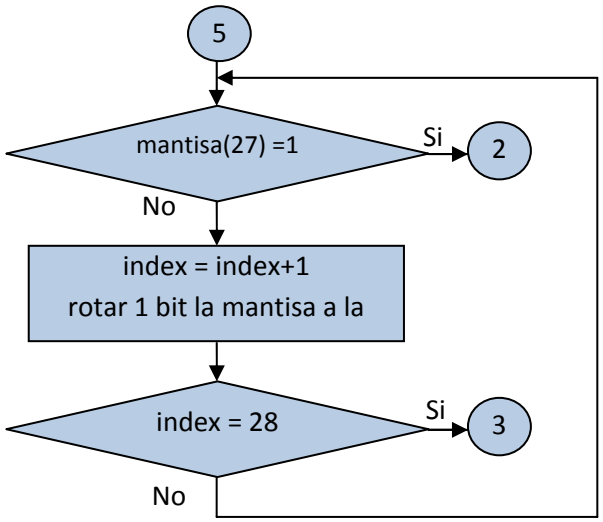


Figura 24: Diagrama de flujo de la entidad "FPU_normalizar" (desplazamiento)

Si se rota toda la mantisa y todos sus bits son cero, el valor de salida será cero, esta condición se verifica observando si el contador "index" alcanza el valor 28. Esta operación dura un ciclo de reloj por cada desplazamiento.

Por último es necesario efectuar la corrección del exponente sumando al exponente del valor de entrada el valor (1-index) obtenido en el paso anterior.

Para el redondeo (véase figura 25) procede concatenando en una variable el exponente corregido y los bits (26:4) de la mantisa, que son los más significativos excluyendo el bit implícito. Se observan los bits menos significativos de la mantisa y si superan el valor "0111" se la suma una unidad a la variable de redondeo, si no, se deja tal cual.

Por último se asigna a la salida el resultado obtenido con el signo correspondiente.

Todo este proceso se realiza en un ciclo de reloj.

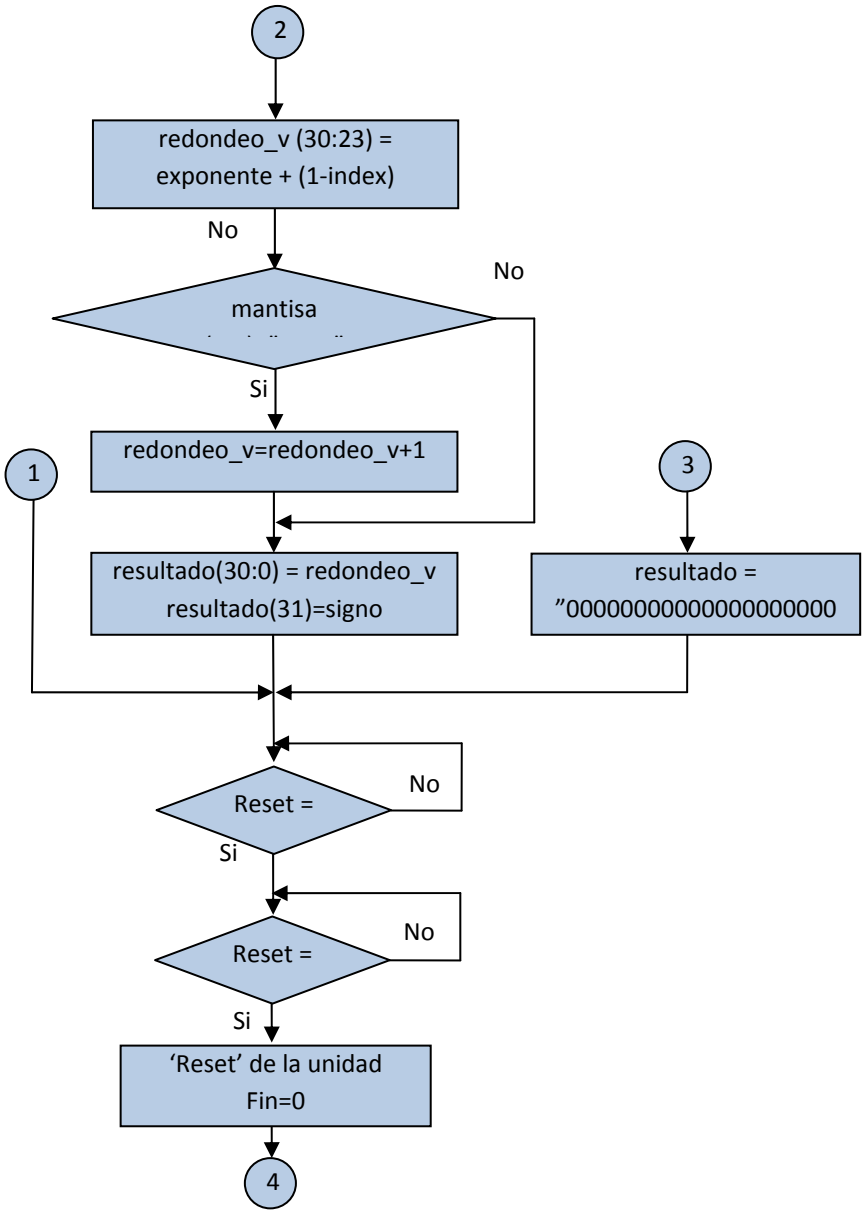


Figura 25: Diagrama de flujo de la entidad "FPU_normalizar" (Redondeo)

Todo el proceso tiene una duración de entre 3 y 30 ciclos de reloj, siendo 6 ciclos la moda en cuanto al tiempo de respuesta.

4.5 Estructura interna de “FPU_in”

Este bloque es el que realiza las operaciones en cuyo interior se implementan diversos bloques con funciones claramente definidas.

Podemos distinguir dos tipos de bloques: los de pre-procesado y los de operación. Las dos puertas lógicas realizan funciones de pre-procesado previos a la operación resta.

Los bloques de operación se encuentran conectados en paralelo de forma que comparten las mismas entradas y son las siguientes:

Entradas		
Nombre	Tipología	Descripción
Inicio	Bit	Pulso, de 1 ciclo de reloj de ancho, que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	'Reset' general de la unidad de coma flotante
Enable	Bit	Se habilita la salida
op_a	Bus 32 bits	Primer operador
op_a_status	Bus 3 bits	Estado del primer operador
op_b	Bus 32 bits	Segundo operador (Excepto "Raiz" y "FPU_CORDIC")
op_b_status	Bus 3 bits	Estado del primer operador (Excepto "Raiz" y "FPU_CORDIC")
Salidas		
Nombre	Tipología	Descripción
resultado	Bus 37 bits	Resultado final sin normalizar.
Cero	Bit	El resultado obtenido es cero
infinito	Bit	El resultado obtenido es infinito
NaN	Bit	El resultado obtenido es NaN
Fin	Bit	Fin de la operación

Tabla 26: Puertos de la entidad "FPU_in"

Los bloques "Raiz" y "FPU_CORDIC" no disponen de entrada para el segundo operador puesto que son operaciones que precisan un solo operador.

Los bloques "FPU_CORDIC" y "operacion_erronea" tienen diferente disposición de entradas y salidas y se explicarán con detenimiento en el apartado correspondiente.

Las salidas de los bloques se activan con la entrada "*enable*" correspondiente, en caso de no activarse dicha entrada permanecen en estado de alta impedancia.

El esquema de implementación de "FPU_in" puede consultarse en el anexo 1.3.

4.5.1 Bloque "operación_decoder"

Bloque combinacional de pre-procesado que básicamente es un decodificador que activa una línea diferente de la salida en función de la operación programada.

Entradas		
Nombre	Tipología	Descripción
Entrada_operacion	Bus 8 bits	Operación programada en la FPU
Salidas		
Nombre	Tipología	Descripción
salida	Bus 10 bits	Resultado final normalizado.

Tabla 27: Puertos de la entidad "Operacion_decoder"

La tabla de funcionamiento es la siguiente:

Operación	Codificación	Línea activa
Suma	1	Salida(1)
Resta	2	Salida(2)
Producto	3	Salida(3)
Cociente	4	Salida(4)
Raiz cuadrada	6	Salida(6)
Seno	7	Salida(7)
Coseno	8	Salida(8)
Tangente	9	Salida(9)
Operación errónea	Cualquier otro	Salida(0)

Tabla 28: Respuesta de la entidad "Operacion_decoder"

La utilización de 8 bits para la codificación de 9 operaciones deja la puerta abierta a posibles ampliaciones futuras sin perder recursos, pues el sistema no implementa aquellas señales que no se utilizan.

4.5.2 Bloque "operador_scan"

Bloque combinacional que tiene la misión de detectar posibles valores no normalizados (+∞,-∞,0,NaN) de los valores de entrada e informar a los bloques que realizan las operaciones para que actúen en consecuencia.

Entradas		
Nombre	Tipología	Descripción
op_a	Bus 32 bits	Primer operador
op_b	Bus 32 bits	Segundo operador
Salidas		
Nombre	Tipología	Descripción
op_a_status	Bus 3 bits	Status del primer operador
op_b_status	Bus 3 bits	Status del segundo operador

Tabla 29: Puertos de la entidad "operador_scan"

Descripción del bus de salida:

Bit	Codifica
op_a_status (0)	Se activa si el primer operador es cero
op_a_status (1)	Se activa si el primer operador es infinito
op_a_status (2)	Se activa si el primer operador es NaN
op_b_status (0)	Se activa si el segundo operador es cero

op_b_status (1)	Se activa si el segundo operador es infinito
op_b_status (2)	Se activa si el segundo operador es NaN

Tabla 30: Identificación de líneas del bus de "status"

4.5.3 Bloque "Division"

Para la realización de este bloque se tiene en cuenta el algoritmo de división en coma flotante. Su funcionamiento se ha secuenciado mediante una maquina de estados finitos con cuatro estados: ESPERA, COMPARA, FINAL y ESPERA_RESET; donde cada estado cumple funciones claramente diferenciadas.

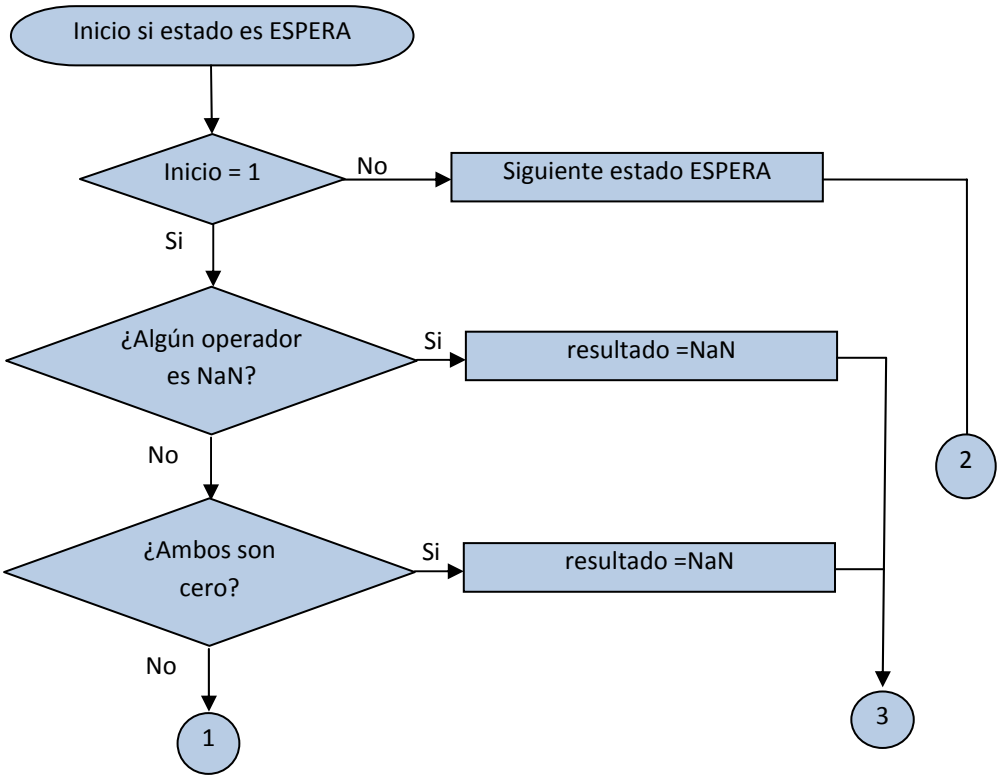
A continuación se describe la función de cada uno de los estados.

Estado ESPERA:

El bloque espera la llegada de un pulso en la entrada inicio y cuando lo hace realiza las siguientes operación es:

- Esperar un pulso en la entrada inicio.
- Comprobar excepciones.
- Si no las hay, cargar las mantisas de los operadores con los bits implícitos y desbordamiento.

Este estado se ejecuta cada ciclo de reloj y tiene una duración de un ciclo de reloj.



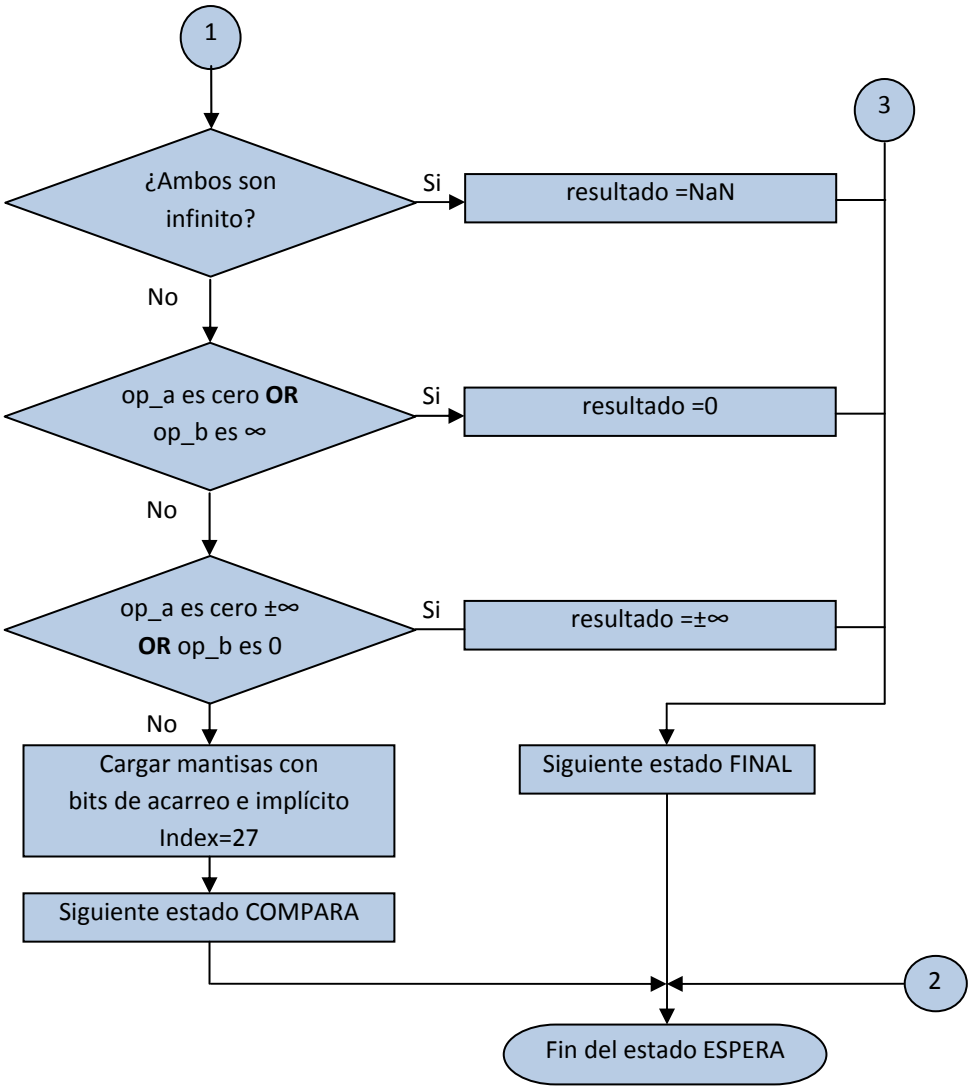


Figura 26: Diagrama de flujo de la entidad "División" (Estado Espera)

Estado COMPARA:

Se compara el dividendo con el divisor:

- Si el dividendo es mayor se le resta el divisor y el cociente es 1.
- Si el dividendo es menor se desplaza el dividendo un bit a la izquierda, lo que equivale a añadir un cero a la derecha, el cociente es cero.
- Si son iguales el cociente es 1 y la división es exacta.

Este estado se ejecuta cada ciclo de reloj y tiene una duración de 27 ciclos de reloj, uno por cada bit del cociente.

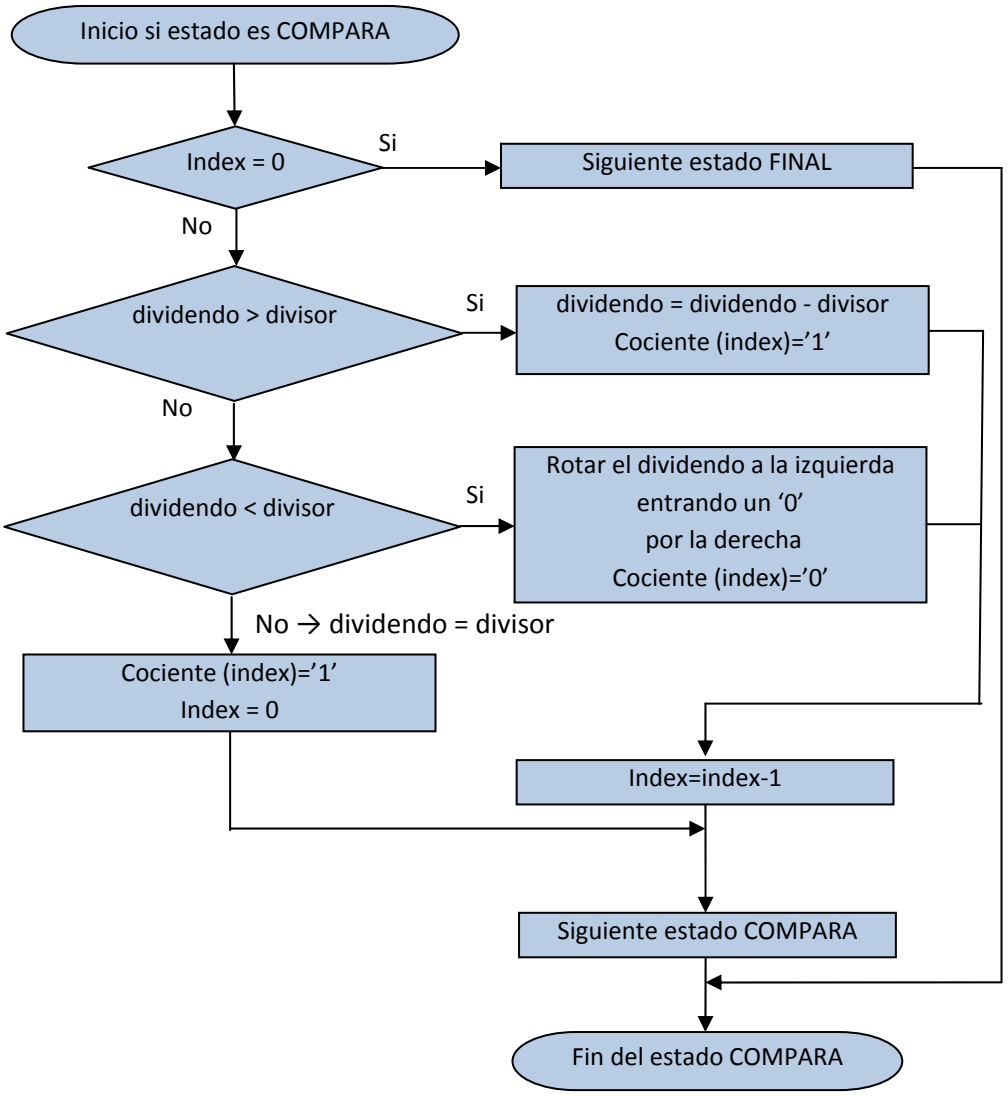


Figura 27: Diagrama de flujo de la entidad "División" (Estado Compara)

Estado FINAL:

Se efectúa la salida del resultado y se ejecuta en un ciclo de reloj.

- Se calcula el valor del exponente del resultado.
- Se escribe la mantisa del resultado con los bits de desbordamiento, implícito y de guarda.
- Se calcula el signo del resultado.
- Se activa la señal "*fin*", que indica al siguiente bloque que se ha concluido el cálculo.

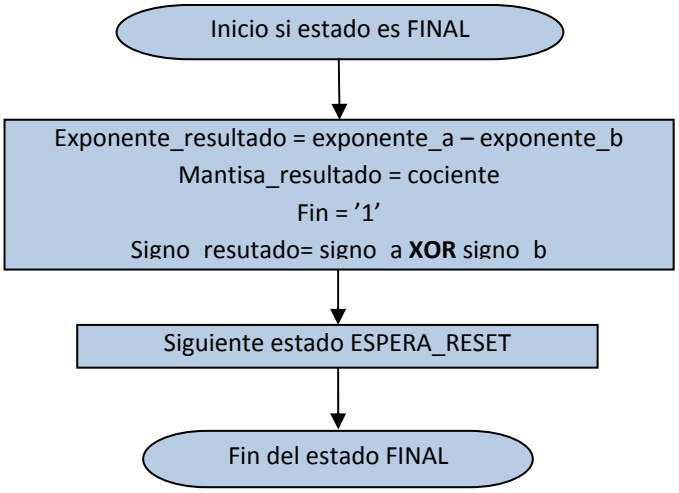


Figura 28: Diagrama de flujo de la entidad "División" (Estado Final)

Estado ESPERA_RESET:

Se desactiva la señal fin y se programa el estado ESPERA, se ejecuta en un ciclo de reloj pero no incrementa el tiempo de proceso al ser concurrente al siguiente paso.

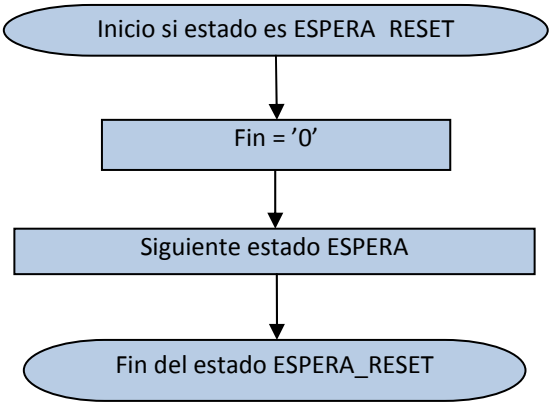


Figura 29: Diagrama de flujo de la entidad "División" (Estado Espera_reset)

El bloque dispone de un “reset” asíncrono que efectúa las siguientes operaciones:

- Fuerza ‘0’ en todas las salidas.
- Fuerza el estado “ESPERA”.

4.5.4 Bloque “Raiz”

Para la realización de este bloque se tiene en cuenta el algoritmo de raíz cuadrada en coma flotante. Su funcionamiento se ha secuenciado mediante una maquina de estados finitos con cinco estados: ESPERA, EXPONENTE, RADICAR1, FINAL y ESPERA_RESET; donde cada estado cumple funciones claramente diferenciadas.

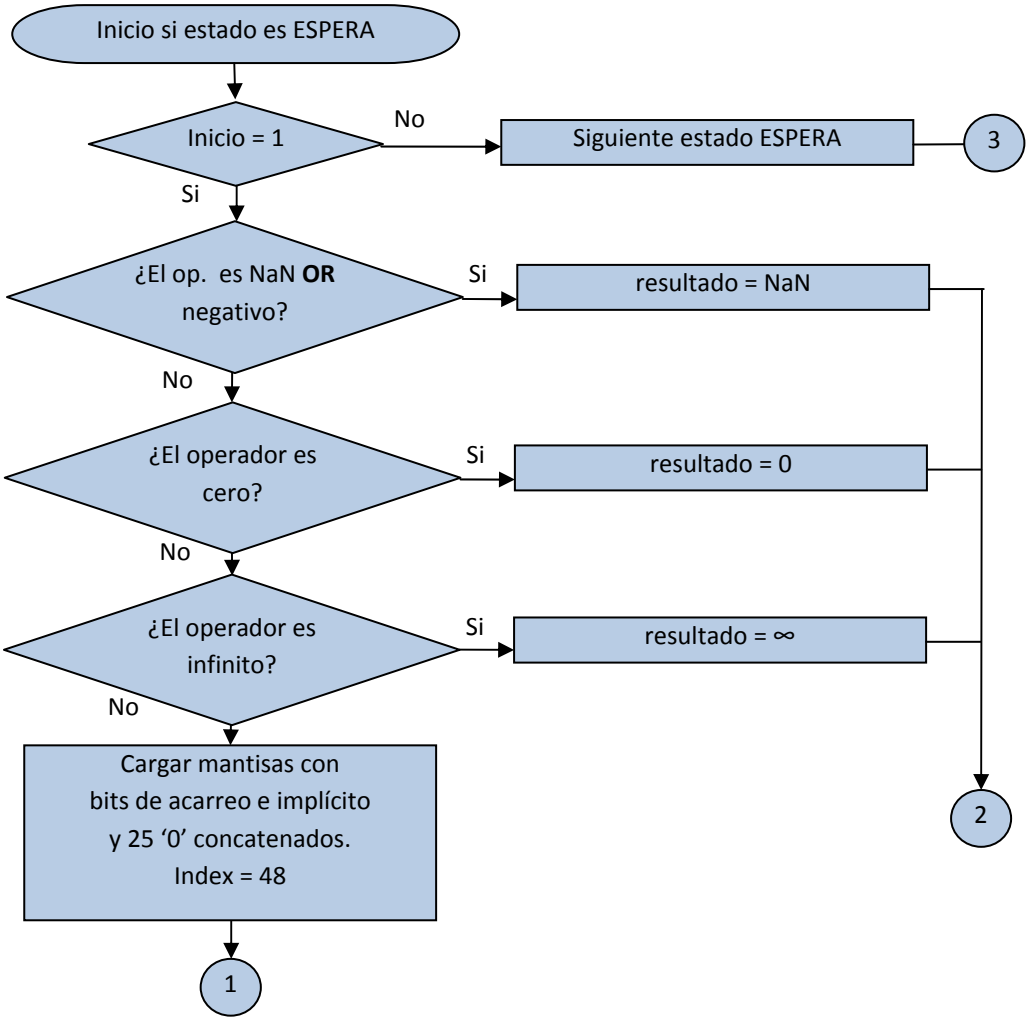
A continuación se describe la función de cada uno de los estados.

Estado ESPERA:

El bloque espera la llegada de un pulso en la entrada inicio y cuando lo hace realiza las siguientes operaciones:

- Esperar un pulso en la entrada inicio.
- Comprobar excepciones.
- Si no las hay: cargar la mantisa del operador con los bits necesarios para poder hacer la raíz cuadrada, si el resultado ha de tener 23 bits el radicando ha de ser de 46 bits. Se carga el exponente y se comprueba su signo.
- Se ponen a cero las señales y variables que intervienen en el algoritmo de radicación

Este estado se ejecuta cada ciclo de reloj y tiene una duración de un ciclo de reloj.



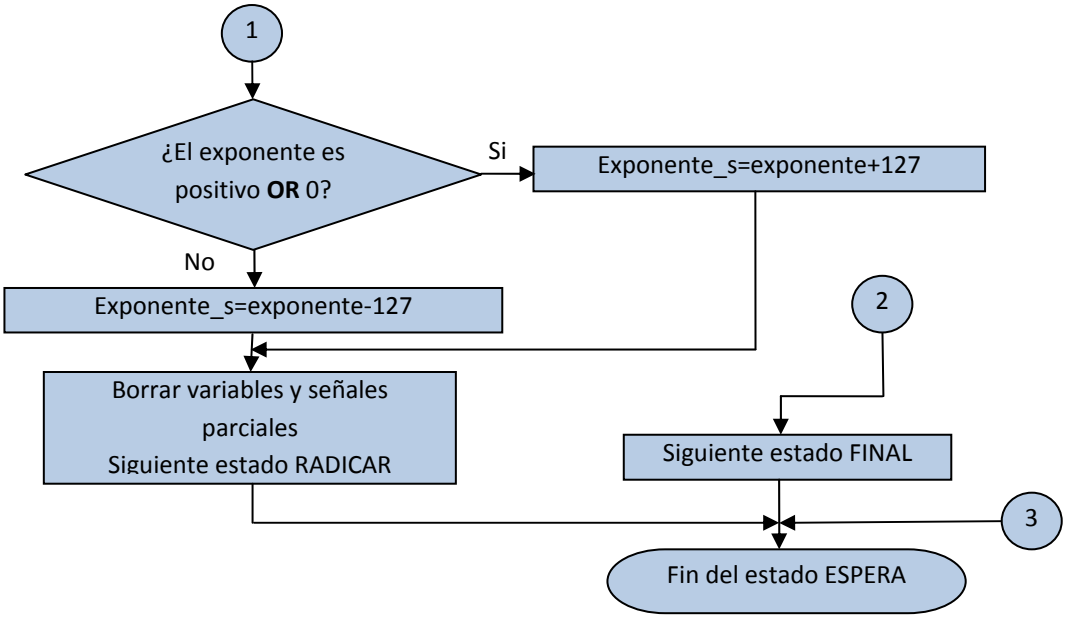


Figura 30: Diagrama de flujo de la entidad "Raiz" (Estado Espera)

Estado EXPONENTE:

Se comprueba el signo del exponente:

- Si el exponente es impar se corrige la mantisa desplazándola un bit a la izquierda y añadiendo un cero por la derecha para compensar el bit que se perderá al dividir el exponente por dos.

Este estado tiene una duración de un ciclo de reloj.

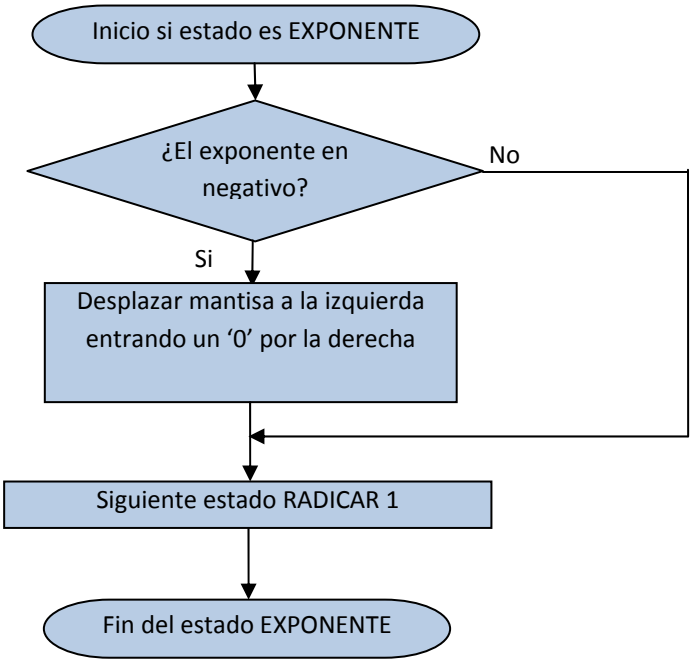


Figura 31: Diagrama de flujo de la entidad "Raiz" (Estado Exponente)

Estado RADICAR1:

Es el estado que implementa el algoritmo de radicación:

- Se toman los dos bits más significativos de la mantisa.
- Si es posible se le resta '01' y se desplaza a la izquierda el resultado entrando un '1' por la derecha
- Si no es posible restarle '01' y se desplaza a la izquierda el resultado entrando un '0' por la derecha
- Al resultado de uno de los dos pasos anteriores se le añaden los siguientes dos bits significativos y se reintenta la resta.
- Se realiza la iteración hasta completar todos los bits de la mantisa.

Este estado tiene una duración de un ciclo de reloj por cada dos bits de la mantisa.

El operador "&" representa concatenación en VHDL.

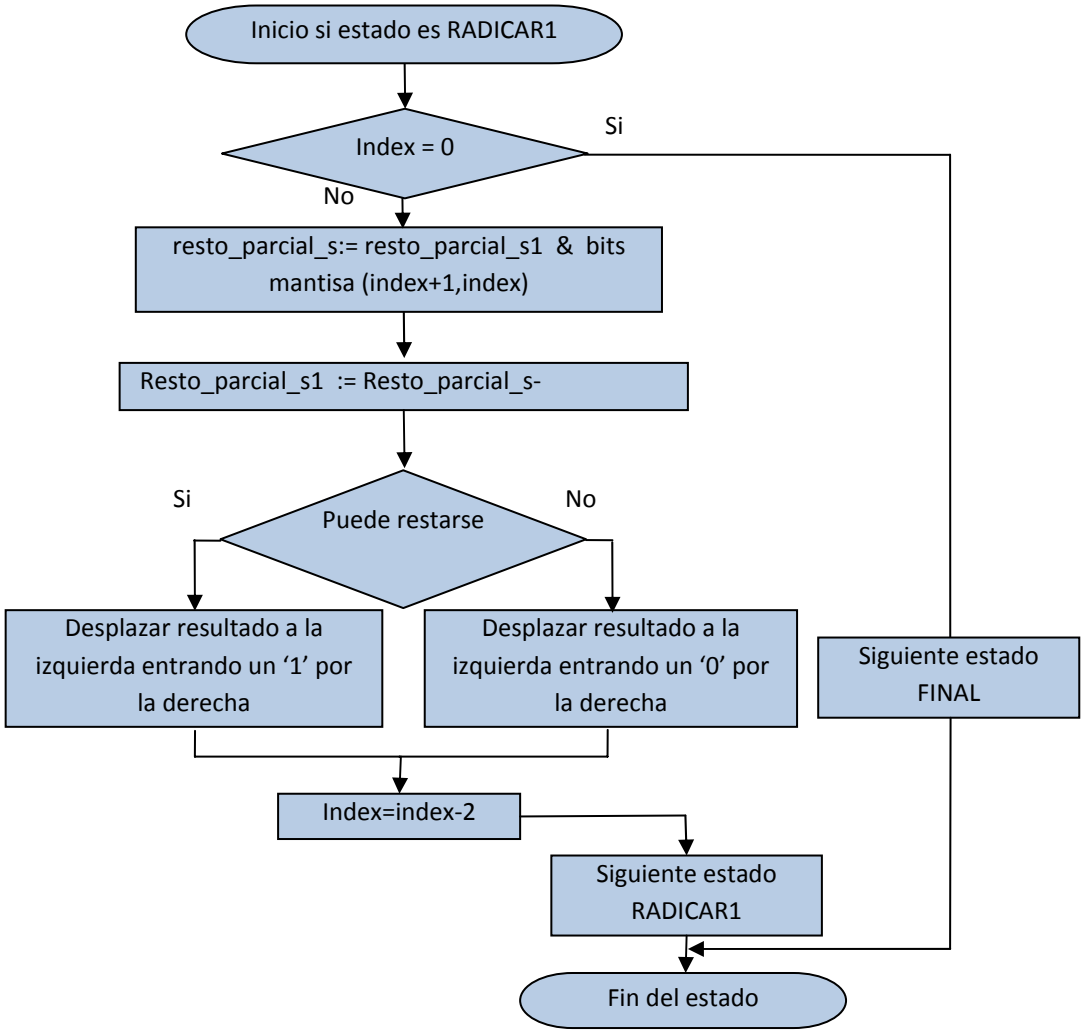


Figura 32: Diagrama de flujo de la entidad "Raiz" (Estado Radicar1)

Estado FINAL:

Se efectúa la salida del resultado y se ejecuta en un ciclo de reloj.

- Se calcula el valor del exponente del resultado.
- Se escribe la mantisa del resultado con los bits de desbordamiento, implícito y de guarda.
- Se fijael signo del resultado en '0', siempre positivo.
- Se activa la señal "*fin*", que indica al siguiente bloque que se ha concluido el cálculo.

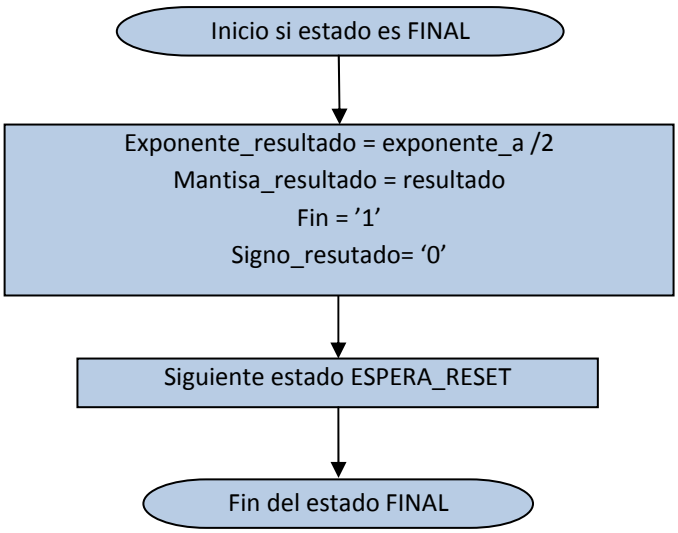


Figura 33: Diagrama de flujo de la entidad "Raiz" (Estado Final)

Estado ESPERA_RESET:

Se desactiva la señal fin y se programa el estado ESPERA, se ejecuta en un ciclo de reloj pero no incrementa el tiempo de proceso al ser concurrente al siguiente paso.

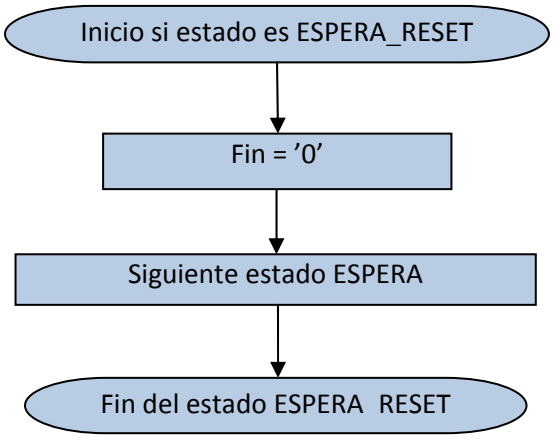


Figura 34: Diagrama de flujo de la entidad "Raiz" (Estado Espera_reset)

El bloque dispone de un “reset” asíncrono que efectúa las siguientes operaciones:

- Fuerza ‘0’ en todas las salidas.
- Fuerza el estado “ESPERA”.

4.5.5 Bloque “Operacion_erronea”

La misión de este bloque es provocar una respuesta NaN si se ha codificado una operación no implementada. Se ejecuta en un ciclo de reloj.

Entradas		
Nombre	Tipología	Descripción
Inicio	Bit	Pulso, de 1 ciclo de reloj de ancho, que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	‘Reset’ general de la unidad de coma flotante
enable	Bit	Se habilita la salida
Salidas		
Nombre	Tipología	Descripción
fin	Bit	Se ha concluido el proceso
NaN	Bit	El resultado en NaN

Tabla 31: Puertos de la entidad "Operacion_erronea"

Diagrama de flujo

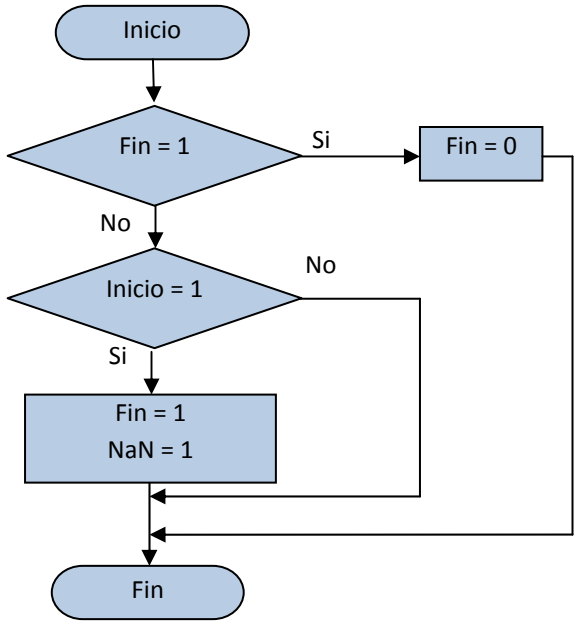


Figura 35: Diagrama de flujo de la entidad "Operacion_erronea"

4.6 Estructura interna de "FPU_suma"

Este bloque es el que se encarga de efectuar la suma y la resta, está formado por dos bloques internos:

- Suma_comparador
- Suma operación

Puede consultarse el esquema de implementación en el anexo 1.4. A continuación e especifican-las funciones de cada bloque:

4.6.1 Bloque "Suma_comparador"

La función que desempeña este bloque es:

- Gestionar las excepciones.
- Comparar los operandos y determinar el mayor y el menor.
- Determinar el signo que tendrá el resultado.

El proceso se ejecuta en un solo ciclo de reloj.

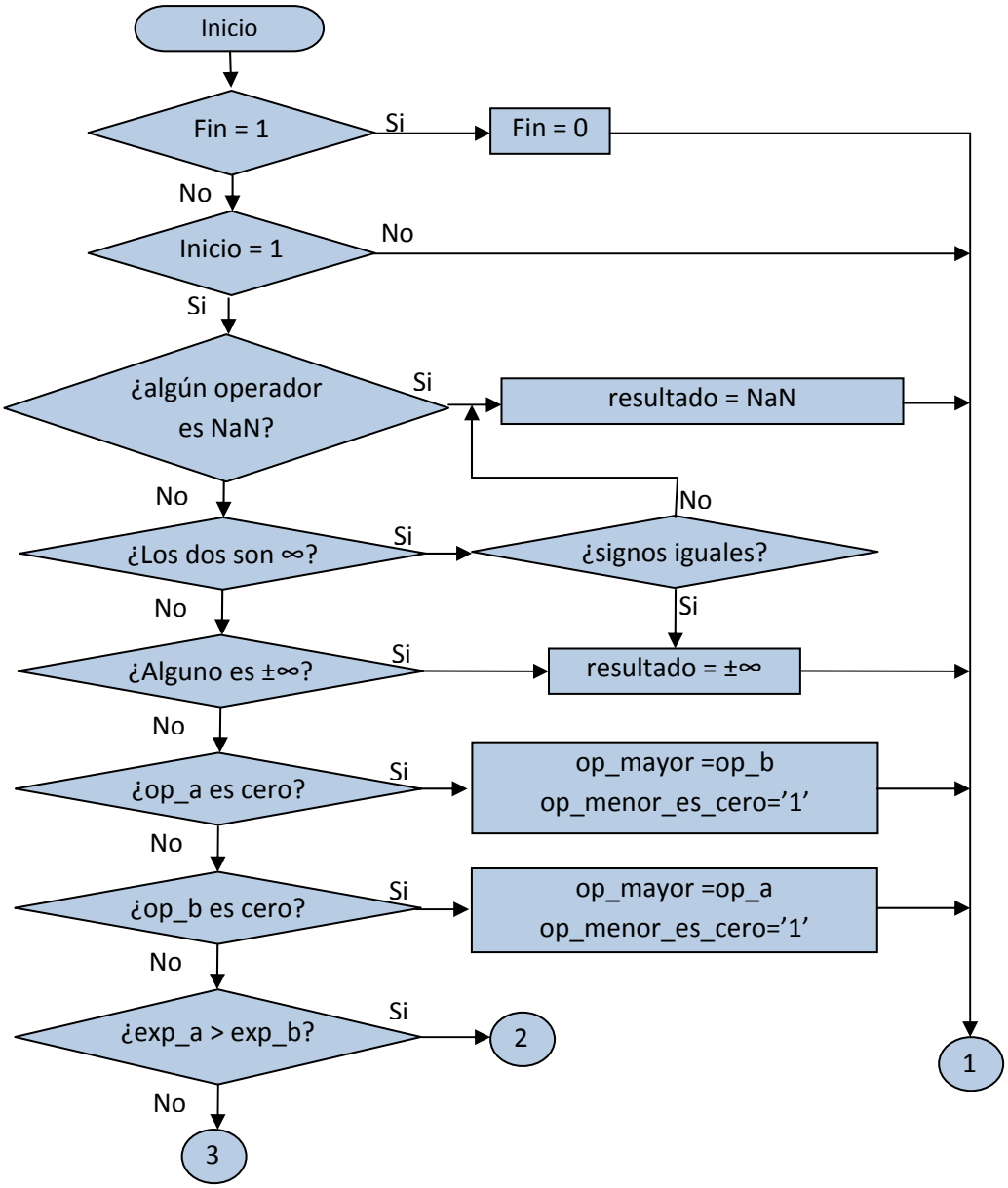
Entradas		
Nombre	Tipología	Descripción
Inicio	Bit	Pulso que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	'Reset' general de la unidad de coma flotante
enable	Bit	Se habilita la salida
op_a	Bus 32 bits	Primer operador
op_a_status	Bus 3 bits	Estado del primer operador
op_b	Bus 32 bits	Segundo operador
op_b_status	Bus 3 bits	Estado del primer operador

Salidas		
Nombre	Tipología	Descripción
op_mayor	Bus 32 bits	Operador de valor absoluto mayor
op_menor	Bus 32 bits	Operador de valor absoluto menor
Signo_resultado	Bit	Signo del resultado de la operacion
NaN	Bit	El resultado obtenido es NaN
cero	Bit	El resultado obtenido es cero

Salidas		
Nombre	Tipología	Descripción
infinito	Bit	El resultado obtenido es infinito
Op_menor_es_cero	Bit	El segundo operador es cero
fin	Bit	Fin del proceso
sumar	Bit	Los valores op_mayor y op_menor • han de sumarse ’0’ • han de restarse ‘1’

Tabla 32: Puertos de la entidad "Suma_comparador"

Diagrama de flujo



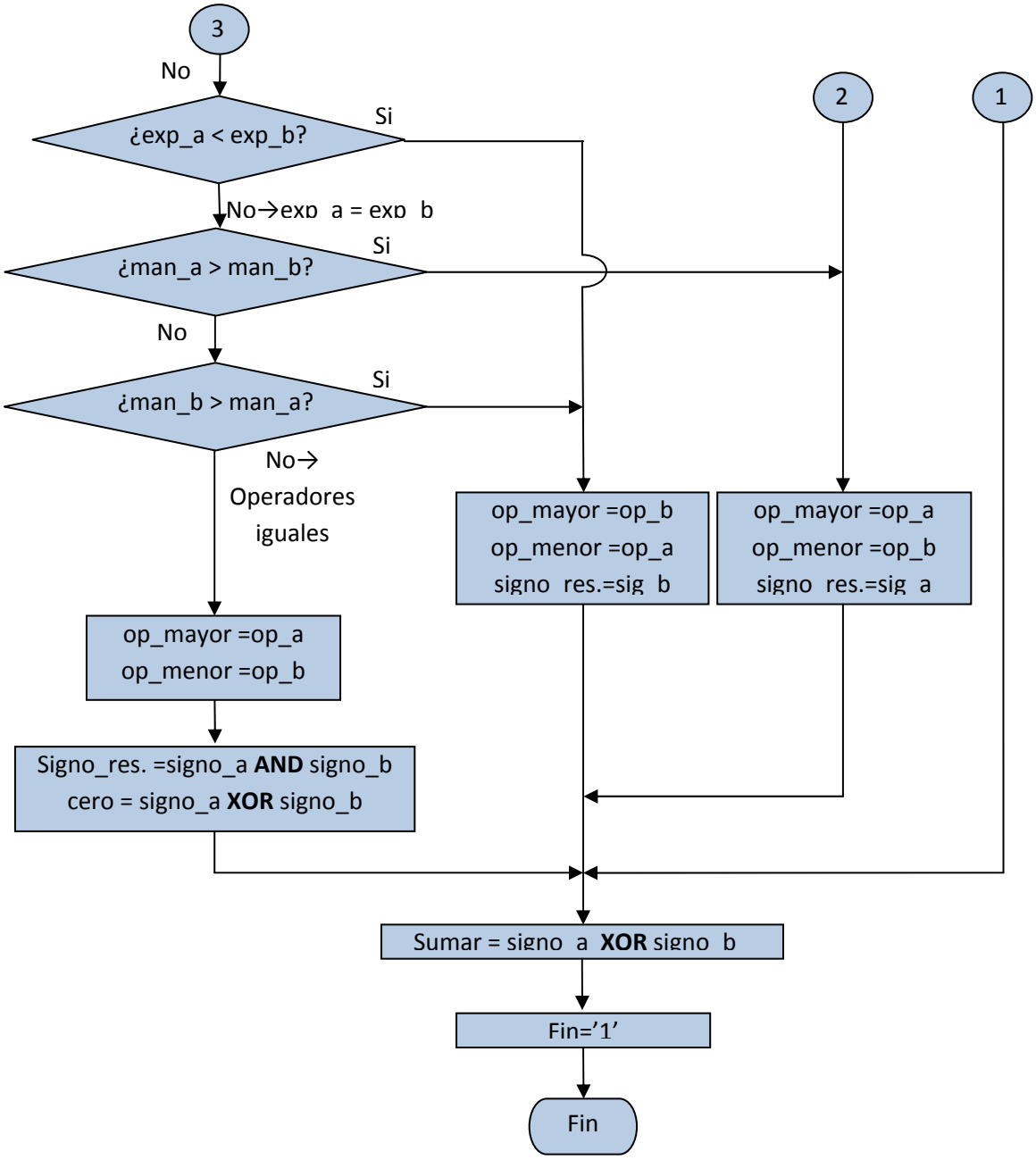


Figura 36: Diagrama de flujo de la entidad "Suma_comparador"

4.6.2 Bloque "Suma_operacion"

Este bloque realiza la operación, su funcionamiento se ha secuenciado mediante una maquina de estados finitos con tres estados: ESPERA, FINAL y ESPERA_RESET; donde cada estado cumple funciones claramente diferenciadas. Se ejecuta en dos ciclos de reloj.

La disposición de entradas y salidas se indica en la siguiente tabla:

Entradas		
Nombre	Tipología	Descripción
Inicio	Bit	Pulso que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	'Reset' general de la unidad de coma flotante
enable	Bit	Se habilita la salida
op_mayor	Bus 32 bits	Operador de valor absoluto mayor
op_menor	Bus 32 bits	Operador de valor absoluto menor
Signo	Bit	Signo del resultado de la operación
NaN_in	Bit	El resultado ha de ser NaN
Cero_in	Bit	El resultado ha de ser cero
Infinito_in	Bit	El resultado ha de ser infinito
op_menor_es_cero	Bit	El operador menor es cero
sumar	Bit	Los valores op_mayor y op_menor • han de sumarse '1' • han de restarse '0'
Salidas		
Nombre	Tipología	Descripción
resultado	Bus 37 bits	Resultado final sin normalizar.
cero	Bit	El resultado obtenido es cero
infinito	Bit	El resultado obtenido es infinito
NaN	Bit	El resultado obtenido es NaN
fin	Bit	Fin de la operación

Tabla 33: Puertos de la entidad "Suma_operacion"

A continuación se especifica el funcionamiento y el diagrama de flujo de los estados que forman el bloque

Estado ESPERA:

- Espera el inicio del proceso,
- Calcula el desplazamiento de las mantisas
- Efectúa la operación

Se ejecuta en un ciclo de reloj.

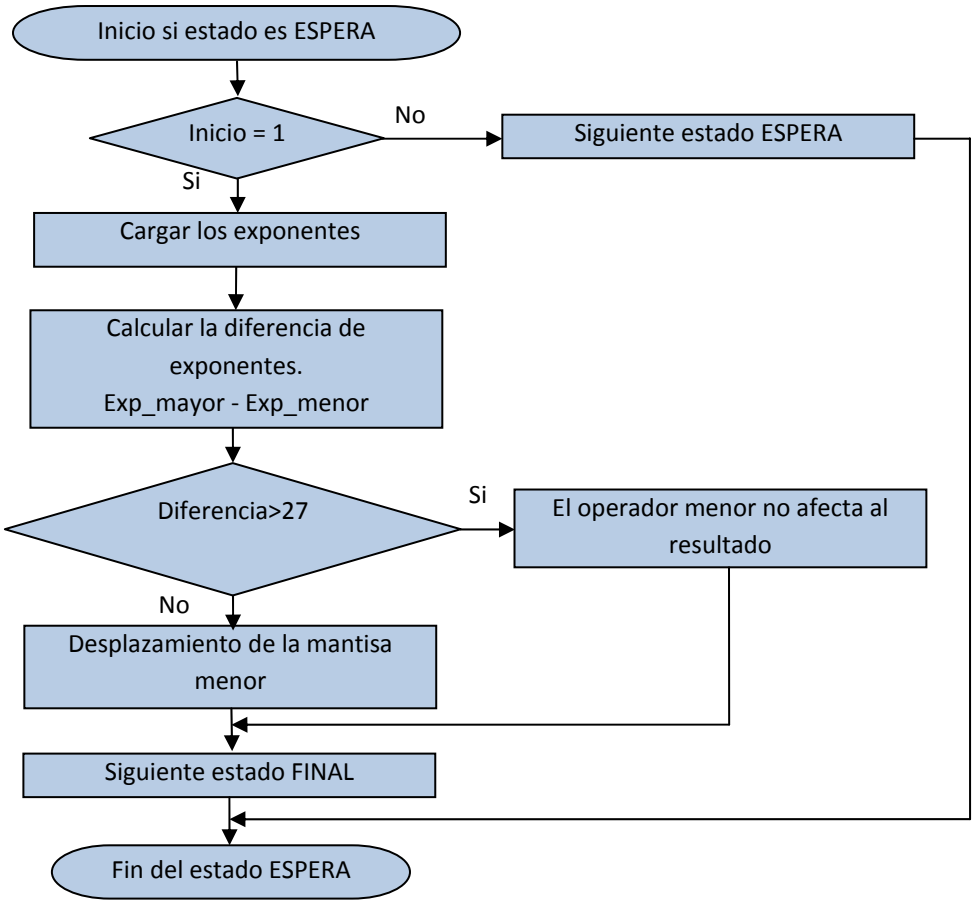
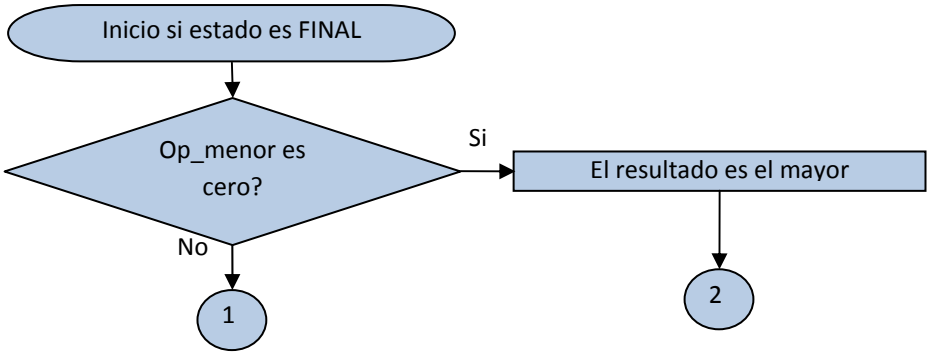


Figura 37: Diagrama de flujo de la entidad "Suma_operacion" (Estado Espera)

Estado FINAL:

- Espera el inicio del proceso, se calcula el desplazamiento de las mantisas y se efectúa la operación

Se ejecuta en un ciclo de reloj.



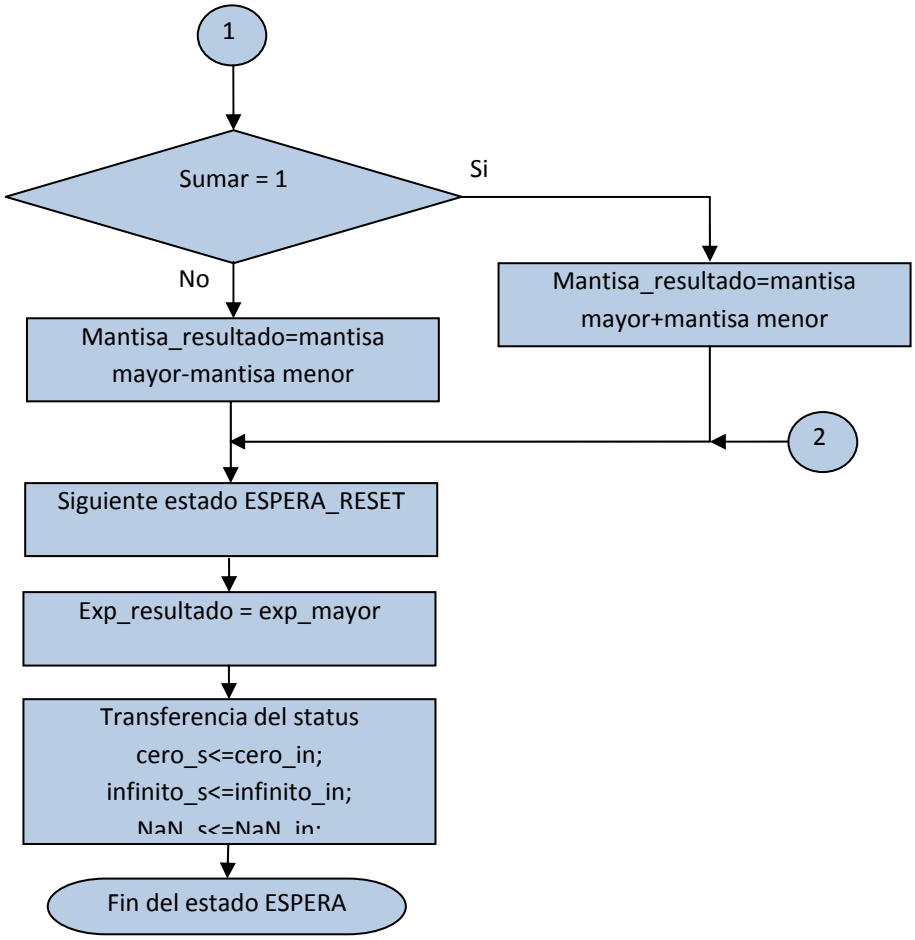


Figura 38: Diagrama de flujo de la entidad "Suma_operacion" (Estado Final)

Estado ESPERA_RESET:

Se desactiva la señal fin y se programa el estado ESPERA, se ejecuta en un ciclo de reloj pero no incrementa el tiempo de proceso al ser concurrente al siguiente paso.

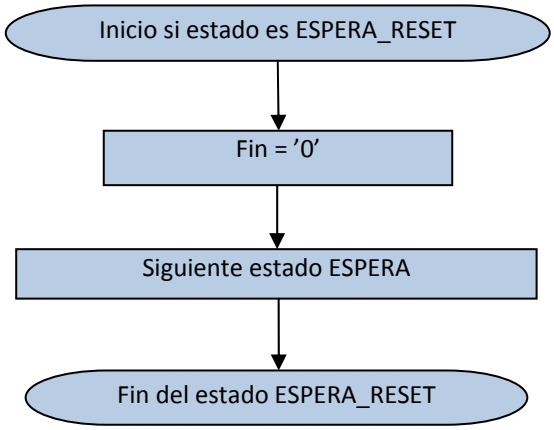


Figura 39: Diagrama de flujo de la entidad "Suma_operacion" (Estado Espera_reset)

El bloque dispone de un "reset" asíncrono que efectúa las siguientes operaciones:

- Fuerza '0' en todas las salidas.
- Fuerza el estado "ESPERA".

4.7 Estructura interna de "FPU_multiplicar"

Para efectuar del algoritmo de multiplicaciónes necesario realizar la suma de los exponentes y la multiplicación de las mantisas. Las mantisas son números binarios de 28 bits y un multiplicador de este tamaño tiene un tiempo de respuesta considerable que condiciona la velocidad de respuesta de todo el sistema. El sistema no puede ser más rápido que la más lenta de sus partes.

Para abordar este problema se ha dividido una multiplicación de 22 bits en cuatro multiplicaciones de 12 según el siguiente razonamiento particularizado para 16 bits.

supongamos los siguientes números de 16 bits expresados en hexadecimal:

$$234D \times 19A3 \quad \textbf{(18)}$$

podemos descomponer el producto de la siguiente forma:

$$(2300+4D) \times (1900+A3) \quad \textbf{(19)}$$

obteniendo los siguientes productos cruzados:

$$(2300 \times 1900)+(2300 \times A3)+(4D \times 1900)+(4D \times A3) \quad \textbf{(20)}$$

sabiendo que los ceros a la derecha sólo modifican el resultado en un desplazamiento a la izquierda y por tanto no es necesario operaros, podemos resolver el problema con los siguientes productos de 8 bits, al efectuar la suma final será necesario considerar los ceros previamente eliminados.

$$(23 \times 19)+(23 \times A3)+(4D \times 19)+(4D \times A3) \quad \textbf{(21)}$$

Para no incurrir en un desmesurado consumo de recursos de la FPGA hay que implementar el sistema de forma que todos los productos los realice el mismo multiplicador, será necesario secuenciar el proceso. Y para que el hecho de secuenciar el proceso no nos penalice el número de ciclos que tardamos en resolver la multiplicación es necesario crear una estructura "pipeline" en la que se estén realizando multiplicaciones y sumas de forma simultánea.

La estructura "pipeline" diseñada tiene 3 etapas que son las siguientes:

- **Multiplicar_seccionar:** separa los valores de entrada de 16 bits en las correspondientes parejas de valores de 8 bits de cada producto cruzado.
- **Multiplicar_producto:** realiza los productos cruzados.

- **Multiplicar_totaliza:** realiza las sumas considerando los consiguientes desplazamientos provocados por los ceros existentes a la derecha de cada producto cruzado y que en la multiplicación se han obviado.

La secuencia de trabajo de la estructura “pipeline” es la siguiente:

Primer ciclo de reloj:

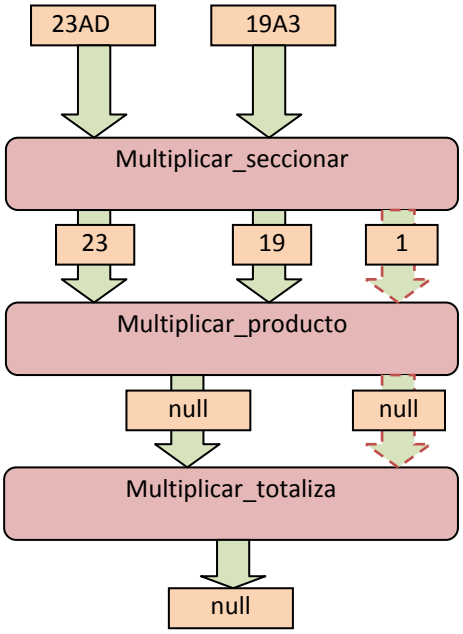


Figura 40: Primer ciclo del "pipeline" del producto

Segundo ciclo de reloj:

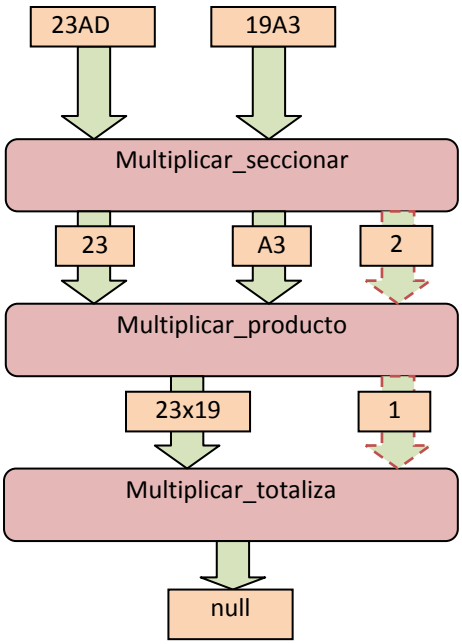


Figura 41: Segundo ciclo del "pipeline" del producto

Tercer ciclo de reloj:

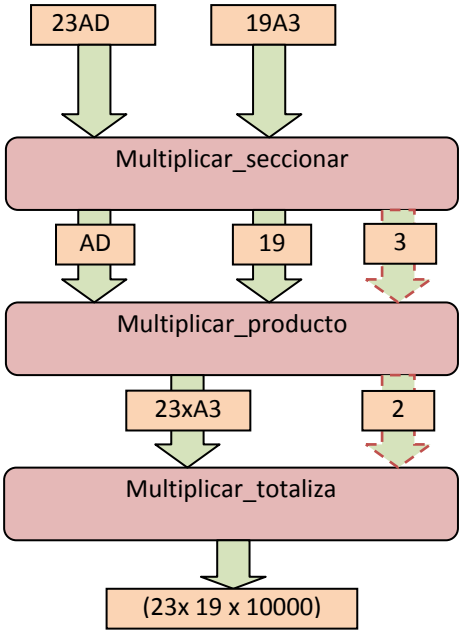


Figura 42: Tercer ciclo del "pipeline" del producto

Cuarto ciclo de reloj:

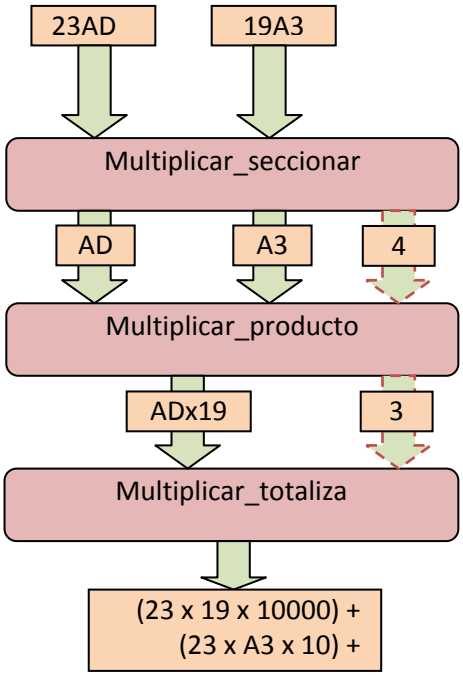


Figura 43: Cuarto ciclo del "pipeline" del producto

Quinto ciclo de reloj:

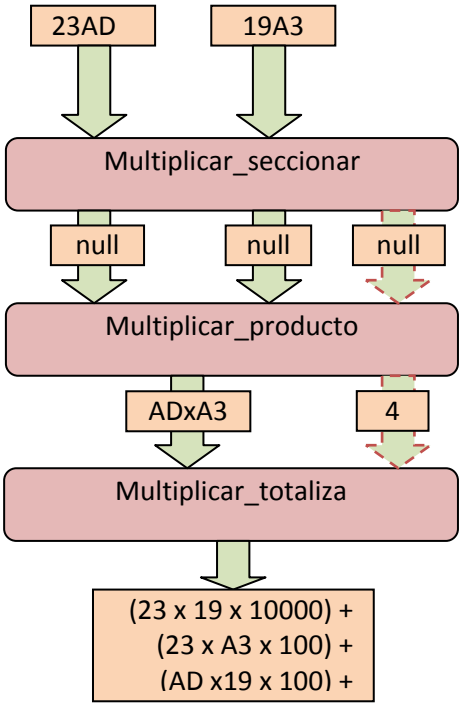


Figura 44: Quinto ciclo del "pipeline" del producto

Sexto ciclo de reloj:

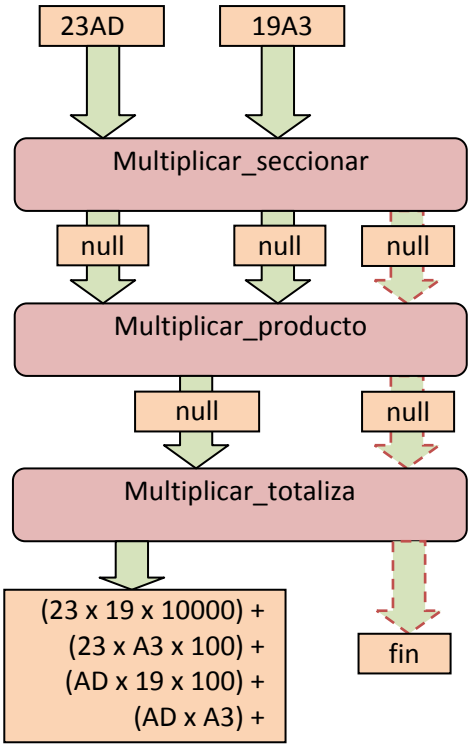


Figura 45: Sexto ciclo del "pipeline" del producto

El sistema incorpora información que se transmite de etapa a etapa y que indica en que punto de la secuencia se encuentra el proceso.

El esquema de implementación se puede consultar en el anexo 1.5.

4.7.1 Bloque "Multiplicar_seccionar"

Este bloque se encarga de seccionar los números de 28 bits de las mantisas en sus dos partes correspondientes de 14 bits y se los prepara para el siguiente bloque de la estructura pipeline.

También efectúa la separación de los signos y de los exponentes para que el bloque "Multiplicar_totalizar" opere con ellos.

Dispone de la siguiente disposición de entradas y salidas.

Entradas		
Nombre	Tipología	Descripción
Inicio	Bit	Pulso que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	'Reset' general de la unidad de coma flotante
op_a	Bus 32 bits	Operador de valor absoluto mayor
op_b	Bus 32 bits	Operador de valor absoluto menor
Salidas		
Nombre	Tipología	Descripción
calcula	Bit	Inicia el funcionamiento del "pipeline"
etapa	Bus 4 bits	Genera el número de etapa que ha de procesar la siguiente etapa, se transmite junto con los datos.
valor_a	Bus 12 bits	Valores de la mantisa del primer operador para el "pipeline"
signo_a	Bit	Signo del primer operador
exp_a	Bus 12 bits	Exponente del primer operador
valor_b	Bit	Valores de la mantisa del segundo operador para el "pipeline"
signo_b	Bit	Signo del segundo operador
exp_b	Bit	Exponente del segundo operador

Tabla 34: Puertos de la entidad "Multiplicar_seccionar"

El proceso de ejecuta cada ciclo de reloj y dura un ciclo de reloj, también dispone de un "reset" asíncrono que inicializa el proceso.

A continuación se muestra el diagrama de flujo de este bloque.

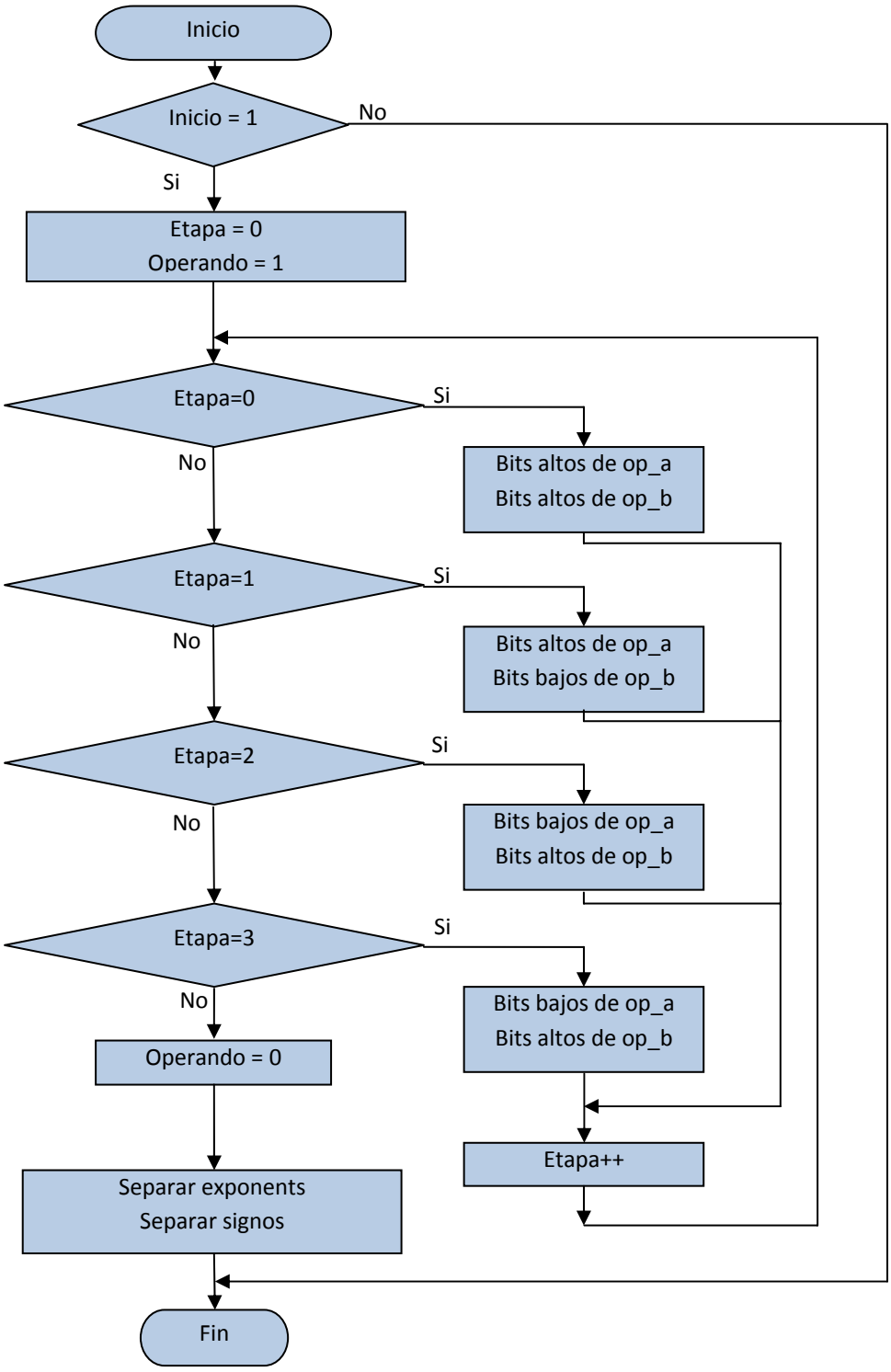


Figura 46: Diagrama de flujo de la entidad "Multiplicar_seccionar"

4.7.2 Bloque "Multiplicar_producto"

Este bloque se realiza las multiplicaciones parciales, se ejecuta cada ciclo de reloj y su duración es de un solo ciclo de reloj.

También efectúa la separación de los signos y de los exponentes para que el bloque "Multiplicar_totalizar" opere con ellos.

Dispone de la siguiente disposición de entradas y salidas.

Entradas		
Nombre	Tipología	Descripción
Operando_in	Bit	Pulso que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	'Reset' general de la unidad de coma flotante
valor_a	Bus 12 bits	Primer valor de 12 bits a multiplicar
valor_b	Bus 2 bits	Segundo valor de 12 bits a multiplicar
etapa_in	Bus 4 bits	Etapa del "pipeline"
Salidas		
Nombre	Tipología	Descripción
Operando_out	Bit	Inicia el funcionamiento de la siguiente etapa
etapa_out	Bus 4 bits	Etapa del "pipeline"
salida	Bus 24 bits	Resultado del producto

Tabla 35: Puertos de la entidad "Multiplicar_producto"

El diagrama de flujo es el de la siguiente figura:

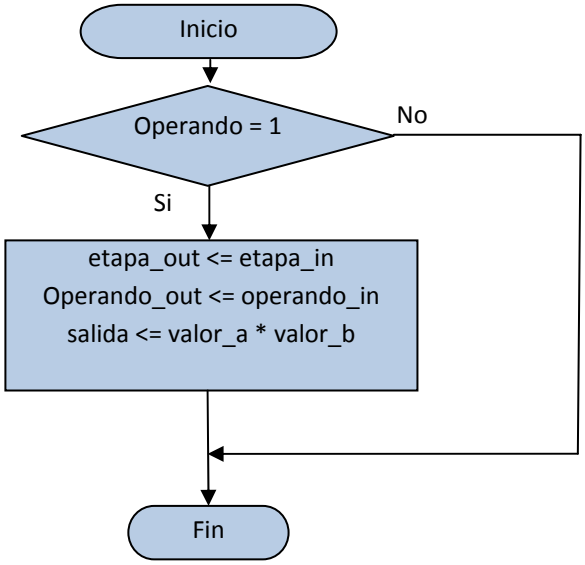


Figura 47: Diagrama de la entidad "Multiplicar_producto"

4.7.3 Bloque "Multiplicar_totaliza"

Este bloque se realiza las sumas parciales considerando los ceros que se han obviado en el proceso, se ejecuta cada ciclo de reloj y su duración es de un solo ciclo de reloj.

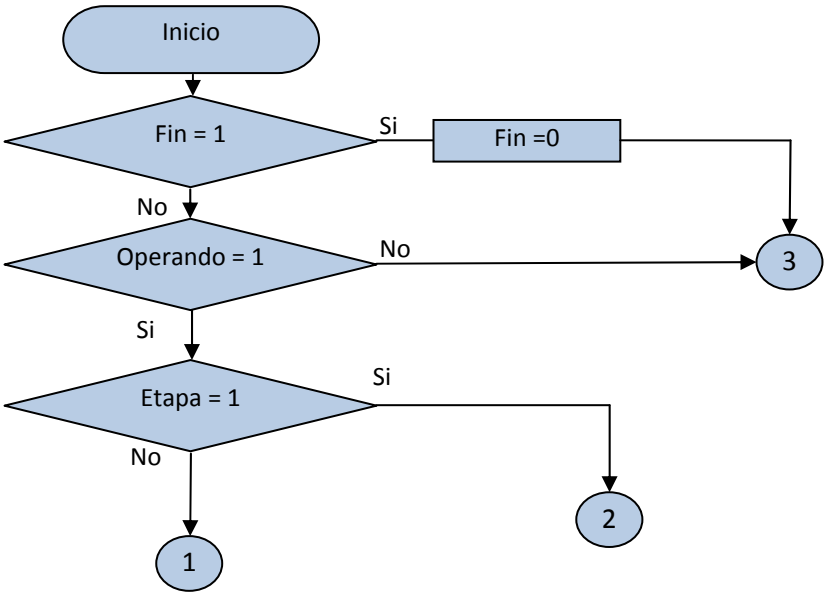
También efectúa la suma de los exponentes y asigna el signo del resultado.

Dispone de la siguiente disposición de entradas y salidas.

Entradas		
Nombre	Tipología	Descripción
Operando_in	Bit	Pulso que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	'Reset' general de la unidad de coma flotante
enable	Bit	Habilita la salida de datos
entrada_dato	Bus 24 bits	Valor a totalizar
signo_resultado	Bit	Signo del resultado
exp_a	Bus 8 bits	Exponente del operando_a
exp_b	Bus 8 bits	Exponente del operando_b
etapa_in	Bus 4 bits	Etapa del "pipeline"
Salidas		
Nombre	Tipología	Descripción
Resultado	Bus 27 bits	Resultado final de la multiplicación sin normalizar
fin	Bus 4 bits	Fin de la operación

Tabla 36: Puertos de la entidad "Multiplicar_totaliza"

A continuación se muestra el diagrama de flujo correspondiente a este bloque:



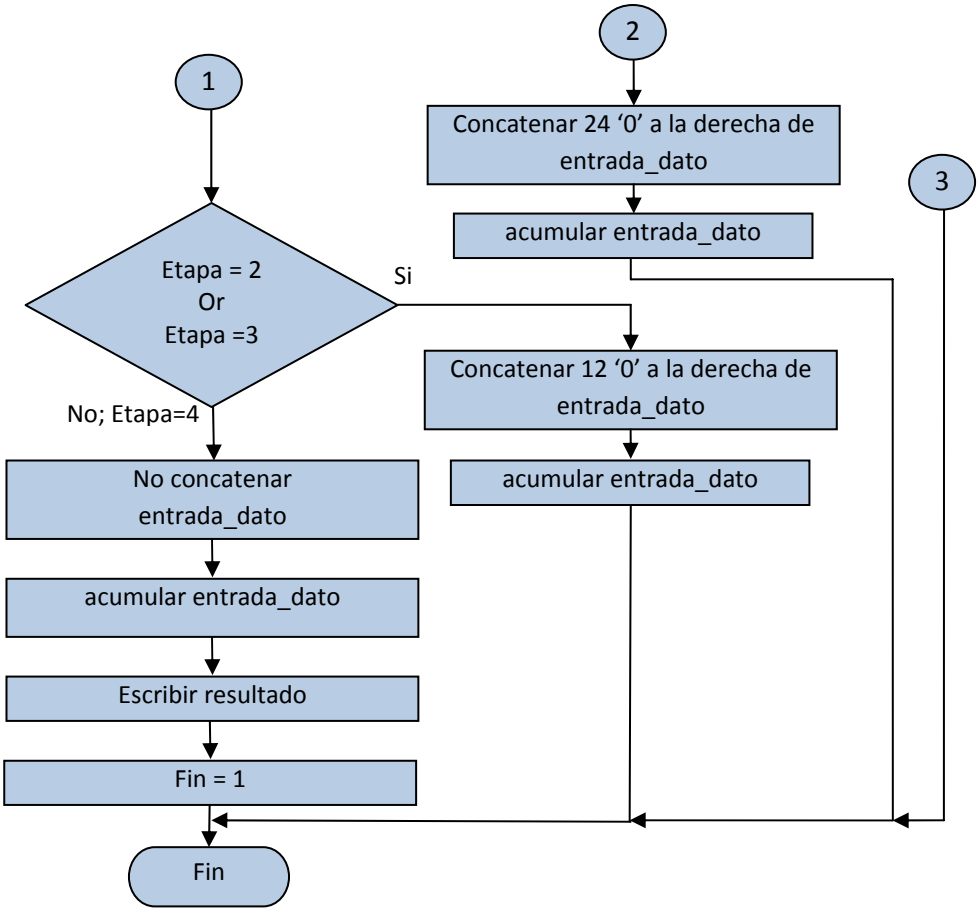


Figura 48: Diagrama de flujo de la entidad "Multiplicar_totaliza"

4.7.4 Bloque "Multiplicar_excepciones"

Este bloque comprueba si alguno de los datos incluye alguna excepción como infinito, cero, NaN. Si es el caso bloquea la salida del "pipeline" y fuerza la salida del resultado indicado. Se ejecuta en un ciclo de reloj pero no implica aumento de tiempo de proceso pues su funcionamiento es concurrente al "pipeline".

Incorpora una entrada de "reset" asíncrono que inicializa las variables a su estado predeterminado.

Dispone de la siguiente disposición de entradas y salidas.

Entradas		
Nombre	Tipología	Descripción
Inicio	Bit	Pulso que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	'Reset' general de la unidad de coma flotante
enable	Bit	Habilita la salida de datos
op_a_status	Bus 3 bits	Status del operador a
op_b_status	Bus 3 bits	Status del operador b

Salidas		
Nombre	Tipología	Descripción
Cero	Bit	El resultado ha de ser cero
Infinito	Bit	El resultado ha de ser infinito
NaN	Bit	El resultado ha de ser NaN
Fin	Bit	Fin con excepciones
Stop	Bit	Bloquear salida del “pipeline”

Tabla 37: Puertos de la entidad "Multiplicar_excepciones"

El diagrama de bloques es el siguiente:

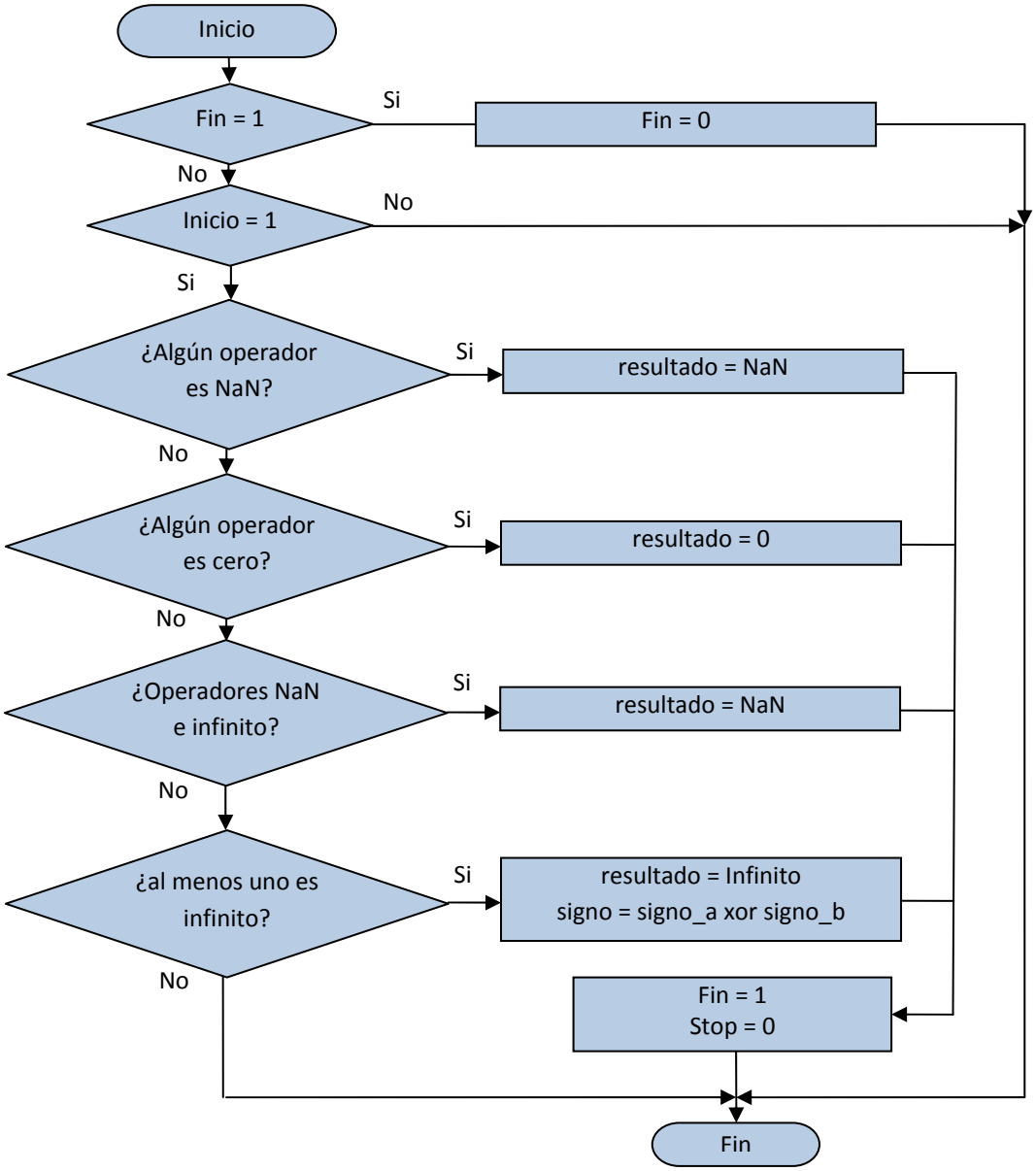


Figura 49: Diagrama de flujo de la entidad "Multiplicar_excepciones"

4.8 Estructura interna de "FPU_cordic"

Este es sin duda el bloque más complejo de todo el sistema pues precisa de un pre y post proceso del dato consistente en:

- reducir el ángulo al primer cuadrante, descontando las vueltas necesarias,
- conversión de coma flotante a coma fija de 32 bits previa a la realización del algoritmo, el ángulo se codifica de la siguiente forma:

  $[0 ,2\pi[\quad \rightarrow \quad [0 ,2[\quad \rightarrow$

  [00000000000000000000000000000000,11111111111111111111111111111111]

- reconversión a coma flotante del resultado obtenido

Para la rápida realización de este algoritmo se opta por la codificación de pi implícito del ángulo dato. De forma que una vuelta corresponde a un ángulo de 2π en radianes y de 2 en codificación pi implícito. La única operación necesaria para adaptar los valores es dividir el ángulo entre pi.

El esquema de implementación puede consultarse en el anexo 1.6.

4.8.1 Bloque "CORDIC_s_c_normaliza"

Este bloque transforma el valor del ángulo codificado en coma flotante, del operador de entrada, en un valor de 32 bits reducido al primer cuadrante y determina en qué cuadrante está, para el posterior procesado del signo.

La conversión de codificación de coma flotante en coma fija de 32 bits reduce el rango de funcionamiento del bloque, con las siguientes limitaciones:

- Si el exponente es mayor de 20 implica que el valor es múltiplo de 2 y por tanto el ángulo es cero.
- Si el exponente es menor de -14 el ángulo se redondea a cero.

También detecta si el ángulo es múltiplo de 2, lo que implica que el valor del ángulo reducido al primer cuadrante es cero.

Incorpora una entrada de Reset asíncrono que inicializa las variables a su estado predeterminado.

El funcionamiento se ha secuenciado mediante una maquina de estados finitos con siete estados: ESPERA, NORMALIZA, EXP_POSITIVO, EXP_NEGATIVO, CUADRANTE, FINAL, ESPERA_RESET; donde cada estado cumple funciones claramente diferenciadas. Se ejecuta en cinco ciclos de reloj.

Dispone de la siguiente disposición de entradas y salidas.

Entradas		
Nombre	Tipología	Descripción
Inicio	Bit	Pulso que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	'Reset' general de la unidad de coma flotante
Operador_in	Bus 32 bits	Operador en coma flotante
Salidas		
Nombre	Tipología	Descripción
fin	Bit	Fin del bloque, activa el siguiente
signo	Bit	Signo del valor de entrada
es_cero	Bit	El ángulo reducido es cero
angulo_small	Bit	El angulo es pequeño y ha de tratarse como tal
resultado	Bus 32 bits	Valor expresado en coma fija de 32 bits
cuadrante	Bus 2 bits	Cuadrante del ángulo

Tabla 38: Puertos de la entidad "CORDIC_s_c_normaliza"

Se muestra el diagrama de flujo de cada uno de los estados:

Estado ESPERA:

En este estado se carga el exponente y las mantisa del ángulo cuanod hay un pulso en la entrada inicio.

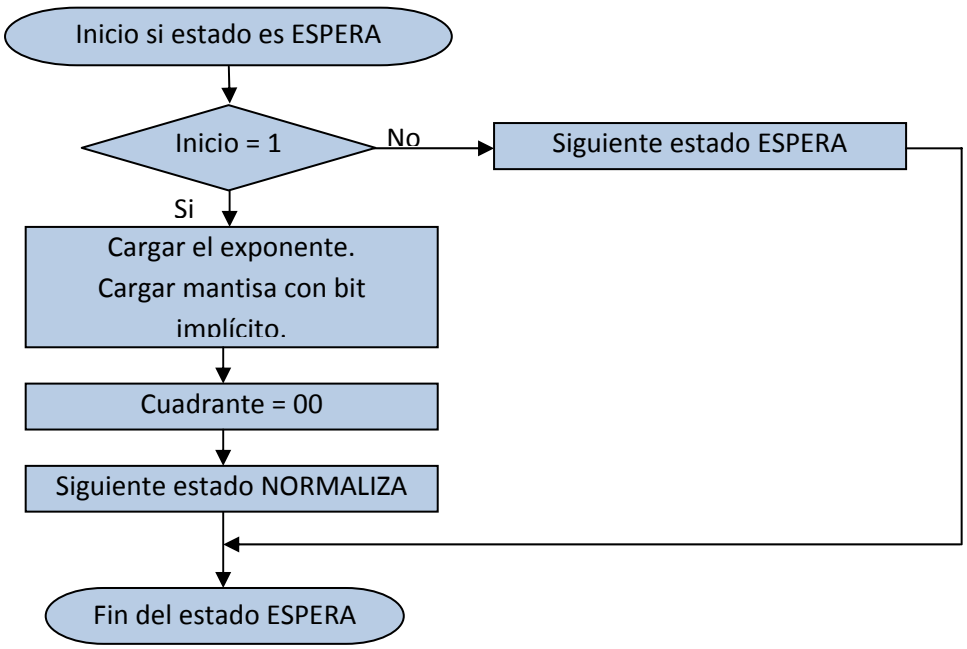


Figura 50: Diagrama de flujo de la entidad "CORDIC_s_c_normaliza" (Estado Espera)

Estado NORMALIZA:

Verifica el rango del ángulo y determina si el exponente es positivo o negativo.

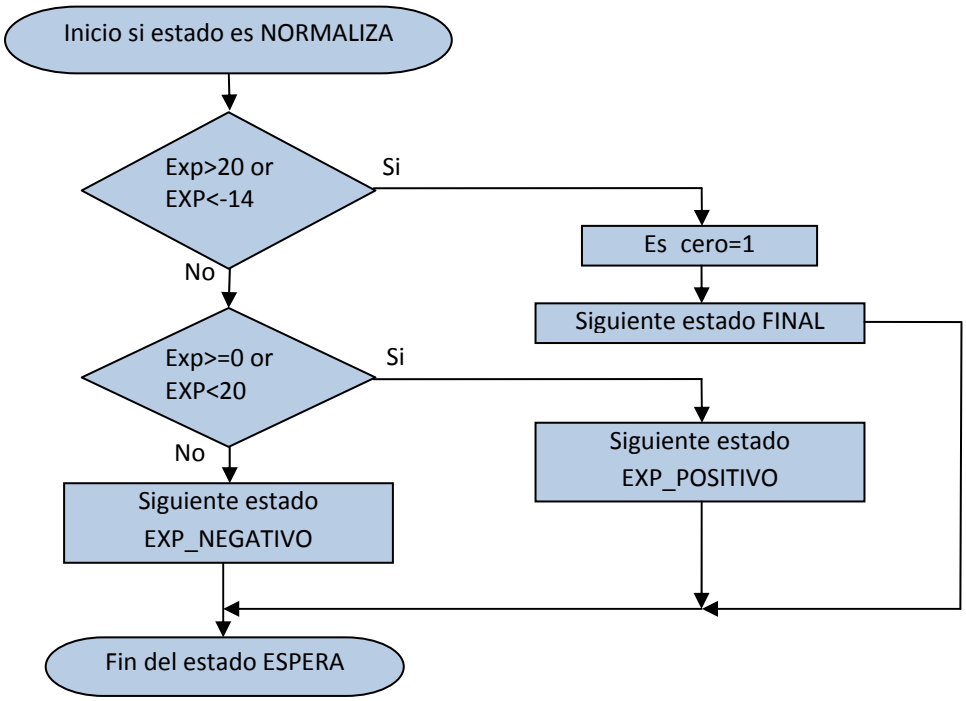


Figura 51: Diagrama de flujo de la entidad "CORDIC_s_c_normaliza" (Estado Normaliza)

Estados EXP_POSITIVO y EXP_NEGATIVO:

Desplaza el exponente los bits necesarios para adaptarlo a la notación de coma fija de 32 bits elegida

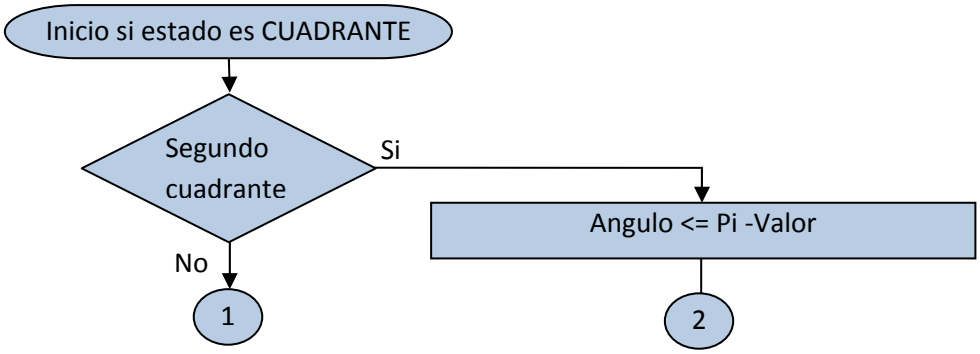
Posteriormente se accede al estado CUADRANTE

Estado CUADRANTE:

Se calcula el cuadrante y se reduce al primero para realizar el cálculo.

Es necesario conocer el cuadrante para el posterior cálculo del signo del resultado.

El cuadrante viene marcado por los dos bits más significativos del valor del ángulo expresado en coma fija.



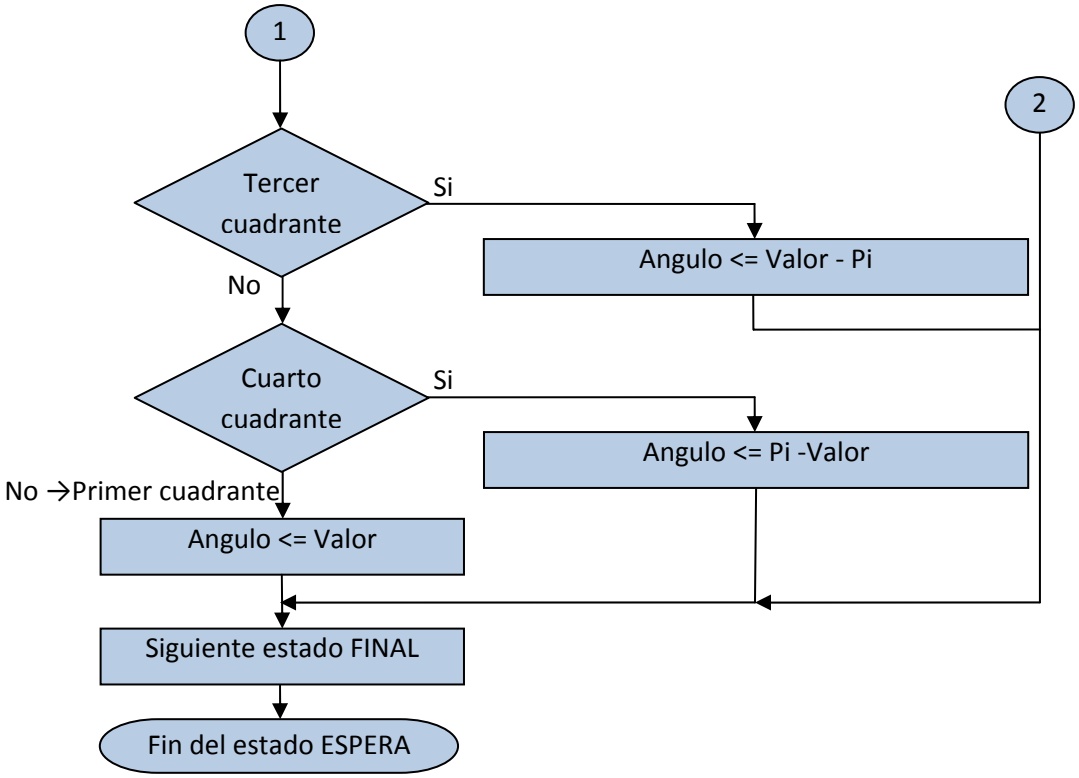


Figura 52: Diagrama de flujo de la entidad "CORDIC_s_c_normaliza" (Estado Cuadrante)

Estado FINAL:

Se escribe el valor del cuadrante.

Se escribe el valor del ángulo en coma fija.

Se escribe el signo del ángulo.

Se activa la señal “fin”, que indica al siguiente bloque que se ha concluido el cálculo.

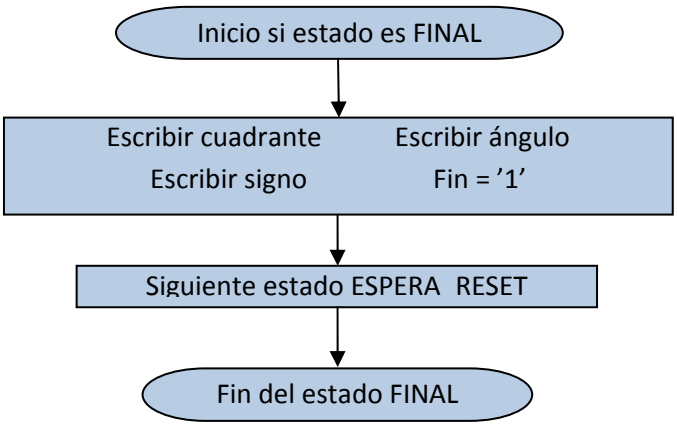


Figura 53: Diagrama de flujo de la entidad "CORDIC_s_c_normaliza" (Estado Final)

ESPERA_RESET

Se desactiva la señal fin y se programa el estado ESPERA, se ejecuta en un ciclo de reloj pero no incrementa el tiempo de proceso al ser concurrente al siguiente paso.

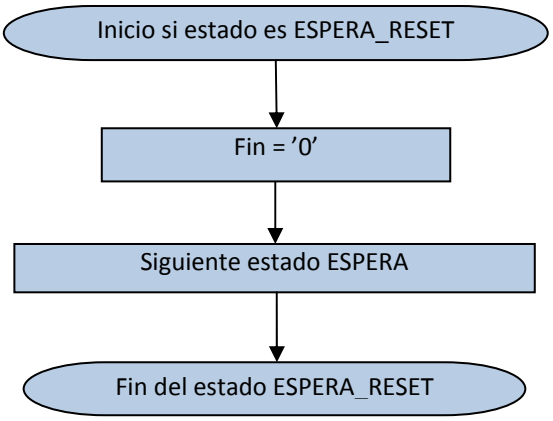


Figura 54: Diagrama de flujo de la entidad "CORDIC_s_c_normaliza" (Estado Espera_reset)

El bloque dispone de un "reset" asíncrono que efectúa las siguientes operaciones:

- Fuerza '0' en todas las salidas.
- Fuerza el estado "ESPERA".

4.8.2 Bloque "CORDIC_signo"

Este bloque calcula el signo del resultado en función del signo del ángulo, del cuadrante en el que se encuentre y de la operación.

Se trata de un bloque combinacional cuyo funcionamiento es concurrente al algoritmo.

Dispone de la siguiente disposición de entradas y salidas.

Entradas		
Nombre	Tipología	Descripción
Signo	Bit	Signo del
Cuadrante	Bus 2 bits	Cuadrantre del ángulo
Salidas		
Nombre	Tipología	Descripción
Signo_seno	Bit	Signo del resultado Seno
Signo_seno	Bit	Signo del resultado Coseno

Tabla 39: Puertos de la entidad "CORDIC_signo"

Tabla de verdad

Entradas			Salidas	
Cuadrante		Signo del ángulo	Signo_seno	Signo_coseno
0	0	0	0	0
0	0	1	1	0
0	1	0	0	1
0	1	1	1	1
1	0	0	1	1
1	0	1	0	1
1	1	0	1	0
1	1	1	0	0

Tabla 40: Tabla de verdad de la entidad "Cordic_signo"

4.8.3 Bloque "CORDIC_algoritmo"

Es el bloque en el que se implementa el algoritmo CORDIC (se puede ver el esquema de implementación en el apartado 5.7) y está formado por dos bloques el primero "CORDIC_s_c_ROM_1" es una memoria ROM de 32 x 32 que almacena los ángulos que cumplen la propiedad de CORDIC. Se trata de un bloque combinacional con la siguiente disposición de entradas salidas.

Entradas		
Nombre	Tipología	Descripción
ROM_ADDR	Bus 5 bits	BUS de direcciones
Salidas		
Nombre	Tipología	Descripción
ROM_DATA	Bus 32 bits	BUS de datos

Tabla 41: Puertos de la ROM de ángulos

El segundo bloque "CORDIC_s_s_algoritmo_al" implementa el algoritmo recursivo de CORDIC que nos proporciona el seno y el coseno del ángulo y además lo hace de forma simultánea. Este algoritmo se completa en 32 ciclos de reloj y dispone de la siguiente asignación de entradas y salidas:

Entradas		
Nombre	Tipología	Descripción
clk	Bit	Reloj general del sistema
reset	Bit	Reset general del sistema
inicio	Bit	Señal de inicio del algoritmo
angulo_small	Bit	Tratamiento especifico de ángulos pequeños
ROM_DATA	Bus 32 bits	Valor del ángulo de la iteración de CORDIC en coma fija
angulo_in	Bus 32 bits	Valor del ángulo de entrada en coma fija

Salidas		
Nombre	Tipología	Descripción
fin	Bit	Indica la conclusión del algoritmo
ROM_ADDR	Bus 5 bits	BUS de direcciones de la ROM de ángulos
resultado_seno	Bus 32 bits	Seno del ángulo en coma fija
resultado_coseno	Bus 32 bits	Coseno del ángulo en coma fija

Tabla 42: Puertos de la entidad "CORDIC_algoritmo"

El funcionamiento se secuencia con una maquia de estados de tres estados denominados: ESPERA, CORDIC, ESPERA_RESET. A continuación se describen el funcionamiento de cada uno de los estados.

El esquema de implementación puede consultarse en el anexo 1.7.

Estado ESPERA:

Carga el ángulo, inicializa los valores de X_s, Y_s y A_s que posteriormente serán le coseno, el seno y el ángulo obtenido tras las iteraciones respectivamente.

Se inicializa la dirección del bus de direcciones de ángulos y el contador de iteración.

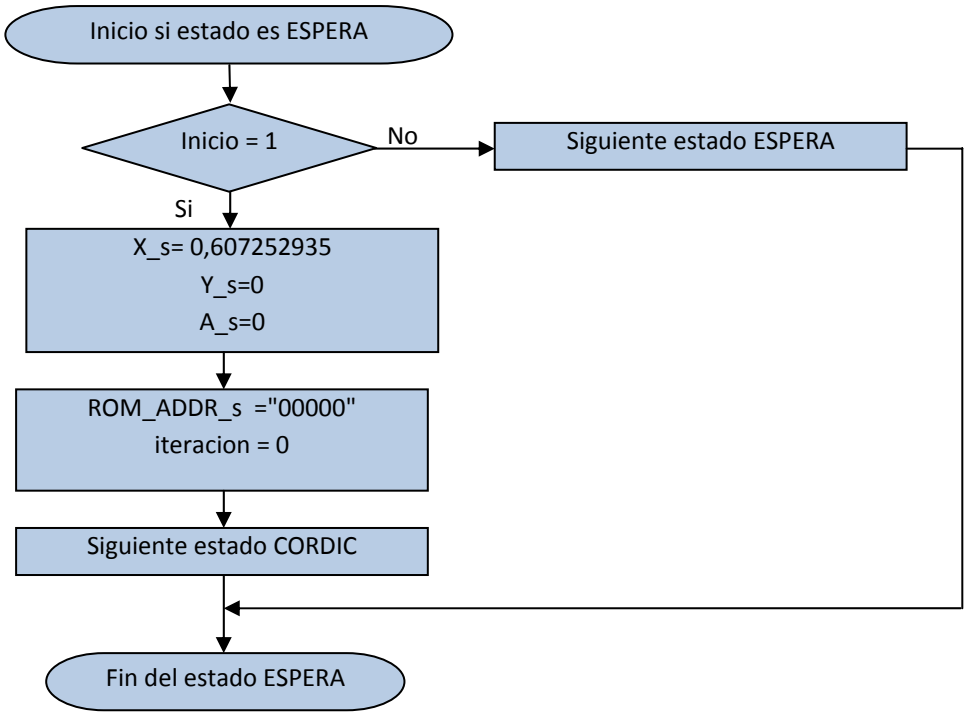


Figura 55: Diagrama de flujo de la entidad "CORDIC_algoritmo" (Estado Espera)

Estado CORDIC:

Se ejecutan las iteraciones del algoritmo de CORDIC

Una vez concluido se escriben los resultados y se activa la salida fin para indicar que se ha concluido el algoritmo.

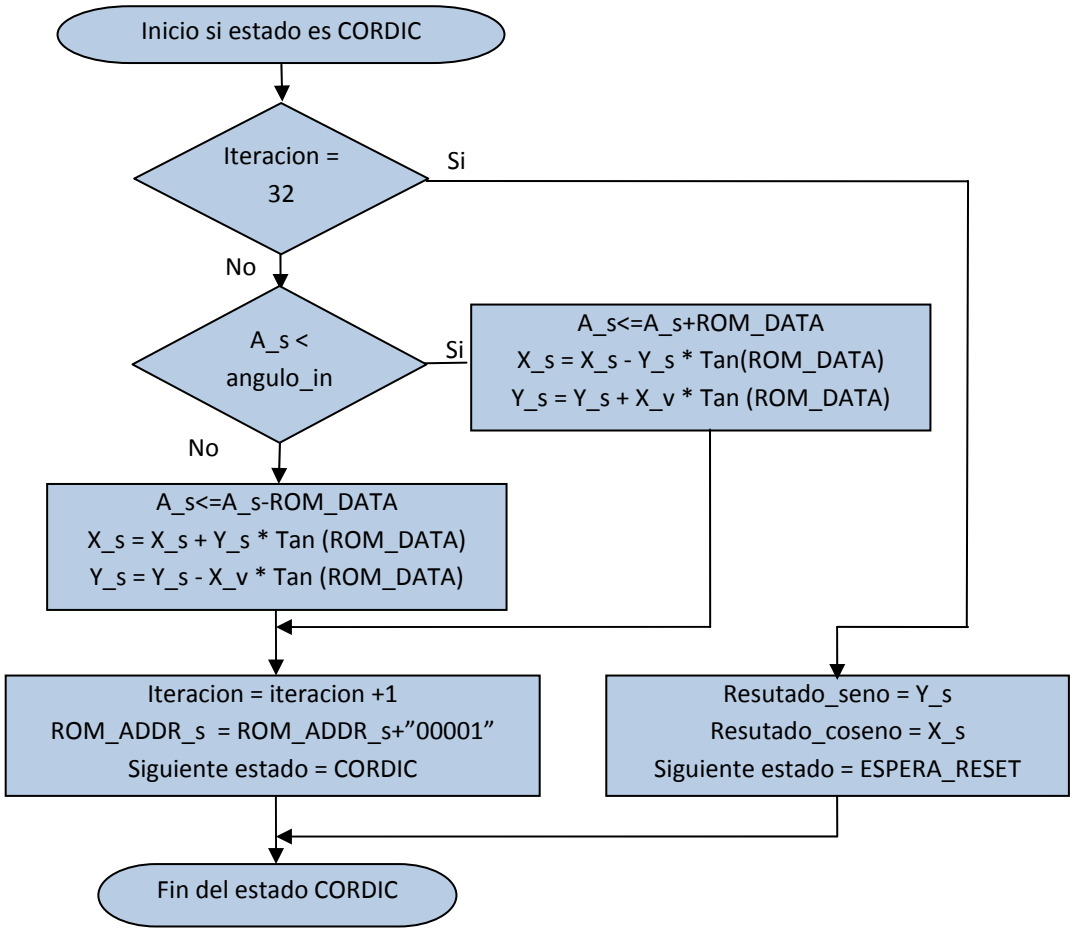


Figura 56: Diagrama de flujo de la entidad "CORDIC_algoritmo" (Estado CORDIC)

Estado ESPERA_RESET

Se desactiva la señal fin y se programa el estado ESPERA_RESET, se ejecuta en un ciclo de reloj pero no incrementa el tiempo de proceso al ser concurrente al siguiente paso.

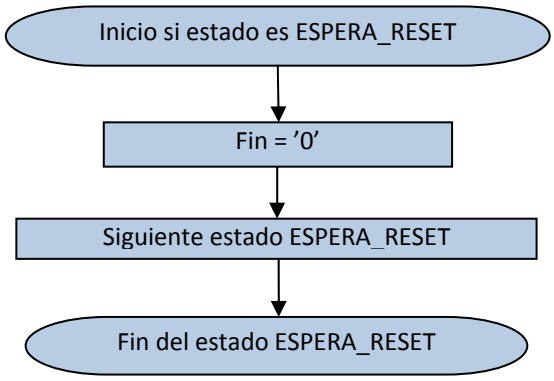


Figura 57: Diagrama de flujo de la entidad "CORDIC_algoritmo" (Estado Espera_reset)

El bloque incorpora una señal de "reset" conectada al de la unidad de coma flotante que reinicializa la máquina de estados al estado ESPERA y los valores al estado inicial.

4.8.4 Bloque "CORDIC_s_c_prenormaliza"

La misión de este bloque es re codificar el resultado del seno y el coseno obtenidos en coma fija para que el siguiente bloque pueda devolver el resultado según la norma. SU funcionamiento se secuencia mediante una máquina de estados con cuatro estados: ESPERA, CALCULASESGO, FINAL y ESPERA_RESET.

La disposición de entradas y salidas es la siguiente:

Entradas		
Nombre	Tipología	Descripción
clk	Bit	Reloj general del sistema
reset	Bit	Reset general del sistema
inicio	Bit	Señal de inicio del algoritmo
angulo_small	Bit	Tratamiento especifico de ángulos pequeños
signo	Bit	Signo del resultado
valor_in	Bus 32 bits	Valor del ángulo de entrada en coma fija
Salidas		
Nombre	Tipología	Descripción
fin	Bit	Indica el fin del proceso
Resultado_out	Bus 37 bits	Valor pre-normalizado

Tabla 43: Puertos de la entidad "CORDIC_s_c_prenormaliza"

Se muestra a continuación el diagrama de flujo de cada uno de los estados.

Estado ESPERA:

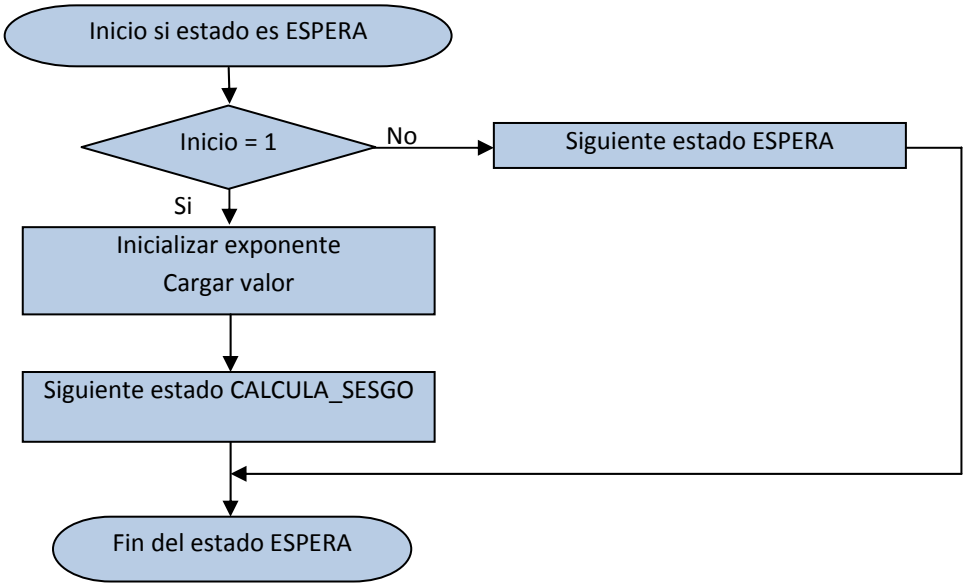


Figura 58: Diagrama de flujo de la entidad "CORDIC_s_c_prenormaliza" (Estado Espera)

Estado CALCULA_SESGO:

Se busca el bit más significativo distinto de cero del valor a procesar.

Se desplaza a la izquierda eliminando los ceros y se adapta el valor del exponente.

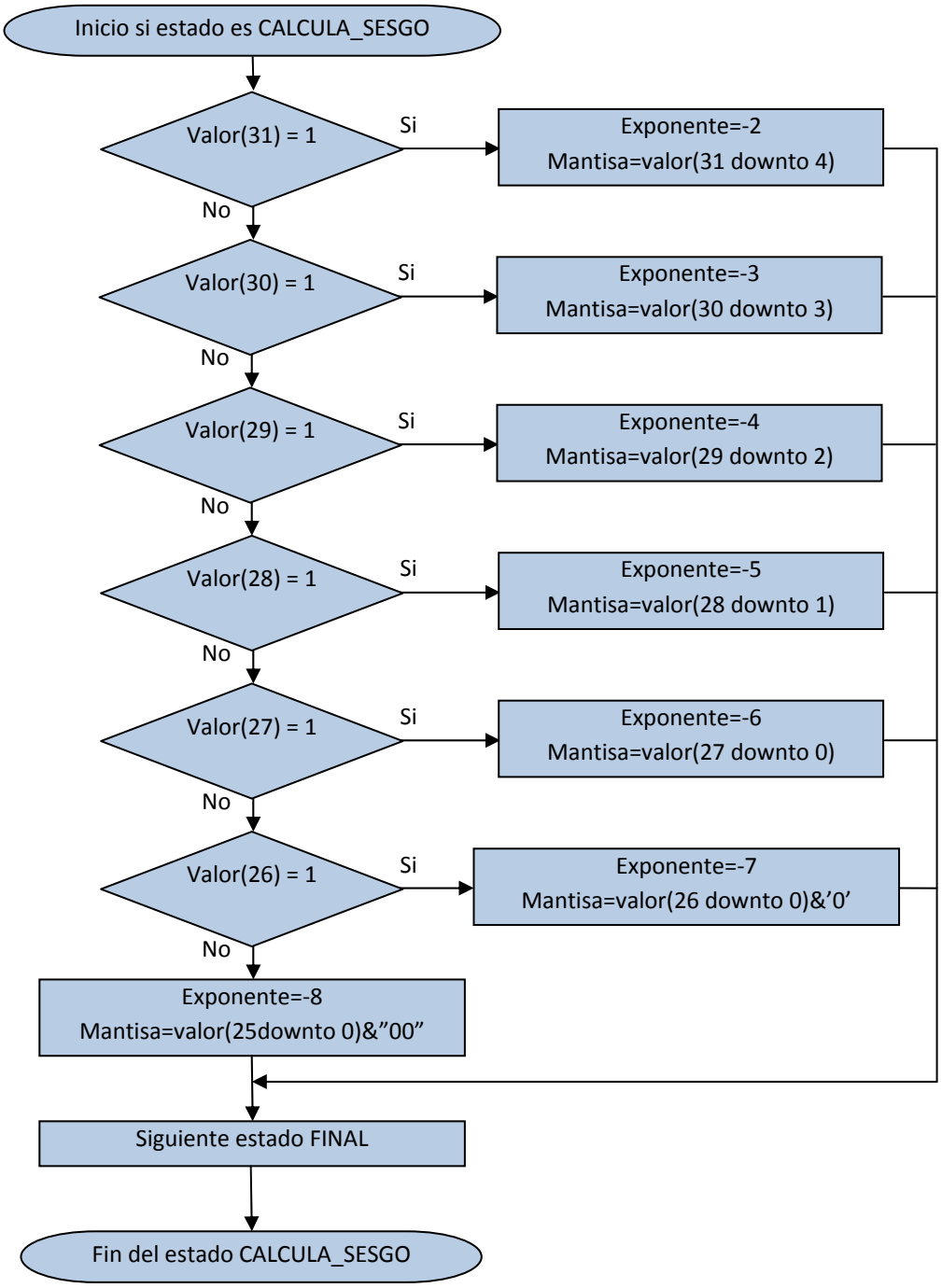


Figura 59: Diagrama de flujo de la entidad "CORDIC_s_c_prenormaliza" (Estado Calcula Sesgo)

Estado FINAL:

Se escribe el valor del cuadrante y se activa la señal de fin de operación.

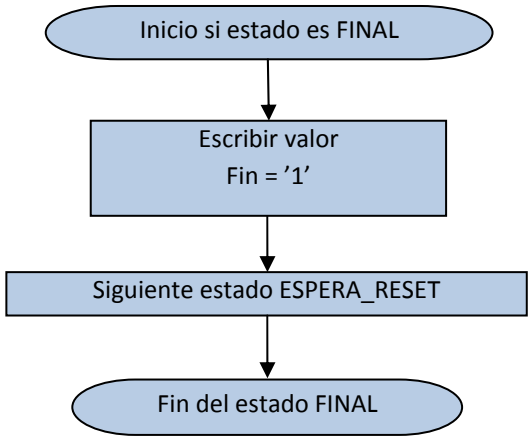


Figura 60: Diagrama de flujo de la entidad "CORDIC_s_c_prenormaliza" (Estado Final)

Eatado ESPERA_RESET:

Se desactiva la señal fin y se programa el estado ESPERA, se ejecuta en un ciclo de reloj pero no incrementa el tiempo de proceso al ser concurrente al siguiente paso.

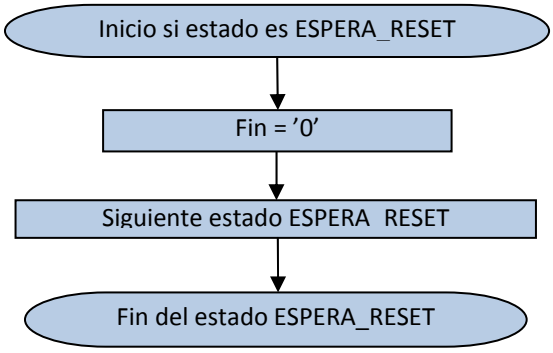


Figura 61: Diagrama de flujo de la entidad "CORDIC_s_c_prenormaliza" (Estado Espera_reset)

El bloque se resetea con el reset general de la unidad, éste programa el estado ESPERA. Además se instancia dos veces una para el proceso del SENO y otra para el proceso de COSENO.

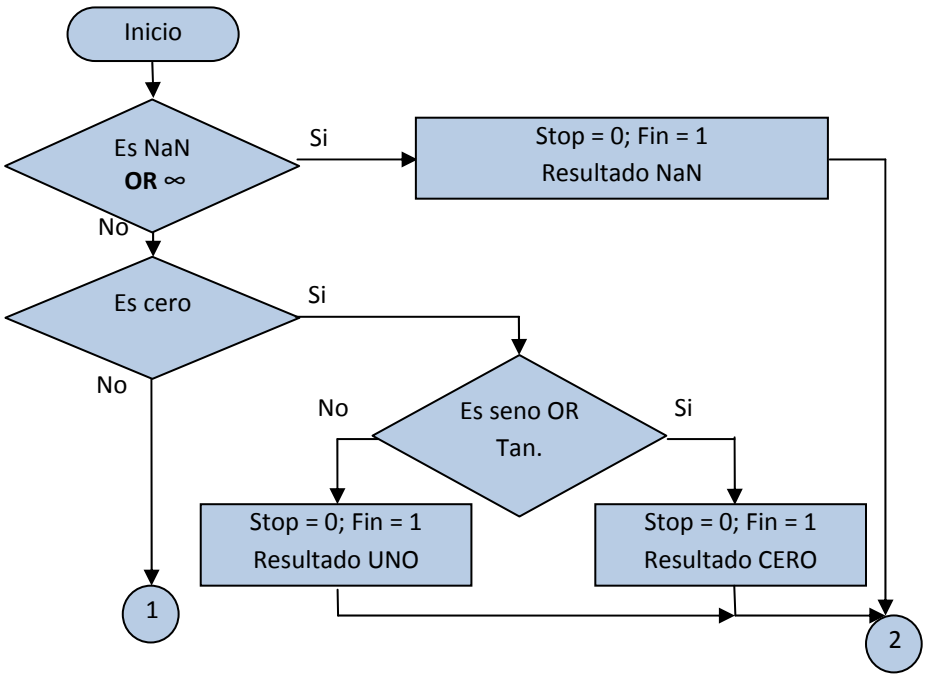
4.8.5 Bloque “CORDIC_s_c_excepciones”

Este bloque controla la aparición de excepciones, se ejecuta tras el bloque “CORDIC_s_c_normaliza” y lo hace de forma concurrente al algoritmo, si se produce alguna excepción lo indica y desestima el resultado del algoritmo.

La disposición de entradas y salidas es la siguiente:

Entradas		
Nombre	Tipología	Descripción
Inicio	Bit	Pulso que inicia el funcionamiento
Clk	Bit	Reloj general del sistema
Reset	Bit	'Reset' general de la unidad de coma flotante
enable	Bit	Habilita la salida de datos
seno	Bit	Se está calculando el Seno
coseno	Bit	Se está calculando el Coseno
tangente	Bit	Se está calculando la Tangente
op_a_status	Bus 3 bits	Status del operador a
valor	Bus 32 bits	Valor del ángulo en coma fija
Salidas		
Nombre	Tipología	Descripción
Cero	Bit	El resultado ha de ser cero
Infinito	Bit	El resultado ha de ser infinito
NaN	Bit	El resultado ha de ser NaN
Uno	Bit	El resultado ha de ser uno
Fin	Bit	Fin con excepciones
Stop	Bit	Bloquear salida del "pipeline"

Tabla 44: Puertos de la entidad "CORDIC_s_c_execciones"



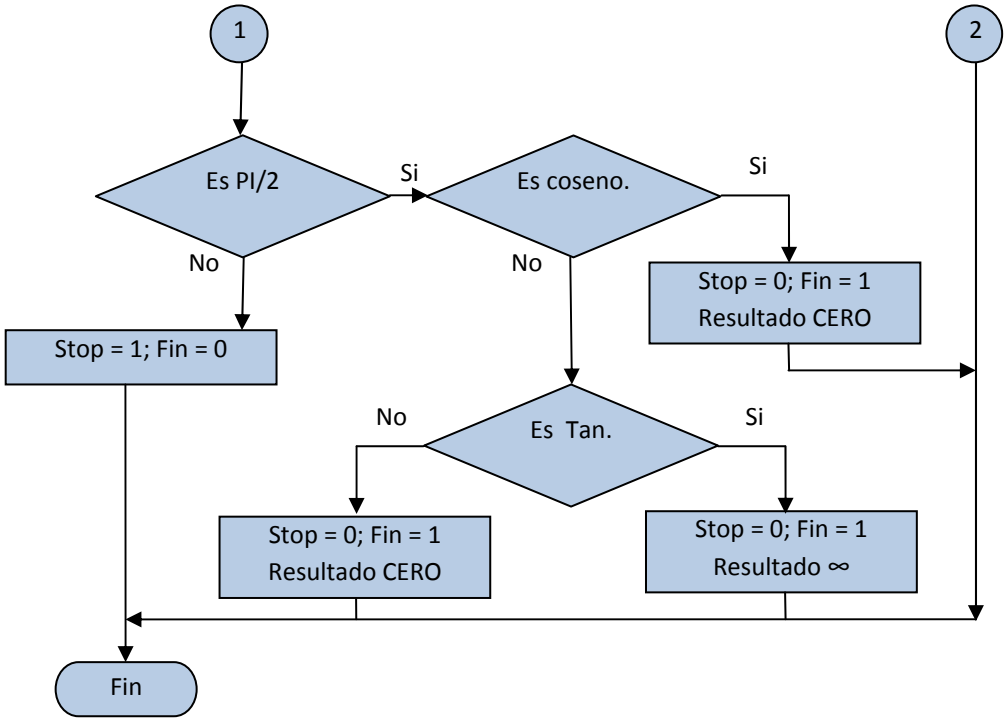


Figura 62: Diagrama de flujo de la entidad "CORDIC_s_c_preexcepciones"

5 Simulación.

Como condiciones de funcionamiento se establece que la frecuencia máxima de trabajo sea de 74 Mhz. Este dato lo proporciona la herramienta de síntesis, y es necesario realizar un reset de la unidad como mínimo un ciclo de reloj antes que la señal de inicio.

En este apartado se presentan las simulaciones del funcionamiento de cada bloque de la FPU con los ciclos de reloj que dura cada operación y con los tiempos de ejecución para una frecuencia de reloj de 50 MHz.

5.1 Reset de la unidad.

Cuando se activa la señal de reset, se establece en nivel bajo la señal de fin que indica que la operación previa había sido concluida. El reset es asíncrono y mantendrá la unidad inoperativa mientras permanezca en estado alto.

La señal de inicio, realiza la operación indicada en la entrada “operacion[7:0]” con los valores que se encuentran codificados en las entradas “operando_a[31:0]” y “operando_b[31:0]”, obviamente deben haber sido escritas con anterioridad.

Véase en la figura siguiente el cronograma del reset de la unidad.

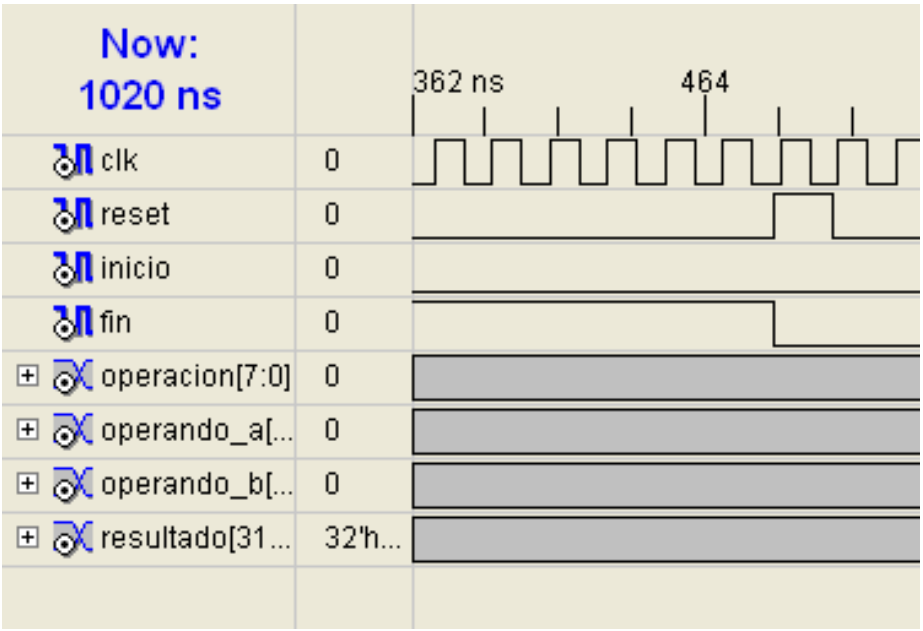


Figura 63: Cronograma de simulación del reset de la FPU

5.2 Operación errónea.

Si se programa una operación que la FPU no tiene implementada se retorna el valor NaN y resuelve en 2 ciclos de reloj. Las operaciones válidas se codifican entre 1 y 9.

Véase en la figura siguiente el cronograma de respuesta si la operación programada no existe.

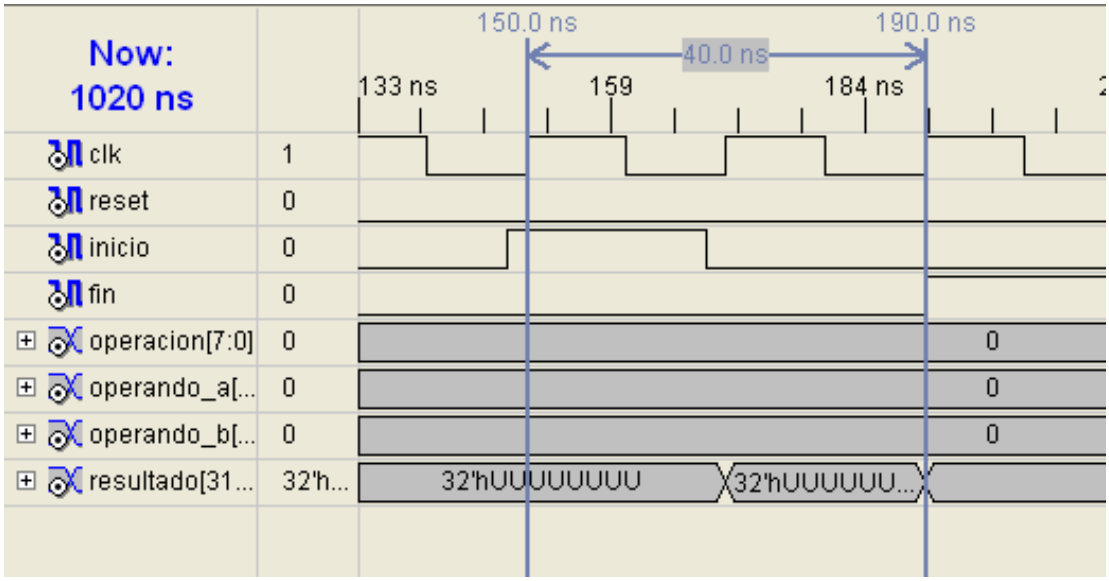


Figura 64: Cronograma de simulación de "Operación errónea"

5.3 Operación Suma

La operación se codifica como operación=1 y se prueba con los siguientes valores:

Resultados		
Nombre	Valor decimal	Codificación en coma flotante
Operador_a	3,123	32’h4047Df3D
Operador_b	71,672	32’h428F5810
Resultado	74,795	32’h4295970A
Ciclos de reloj	7	

Tabla 45: Condiciones de simulación de la suma

En la figura siguiente puede verse el cronograma de respuesta de la unidad.

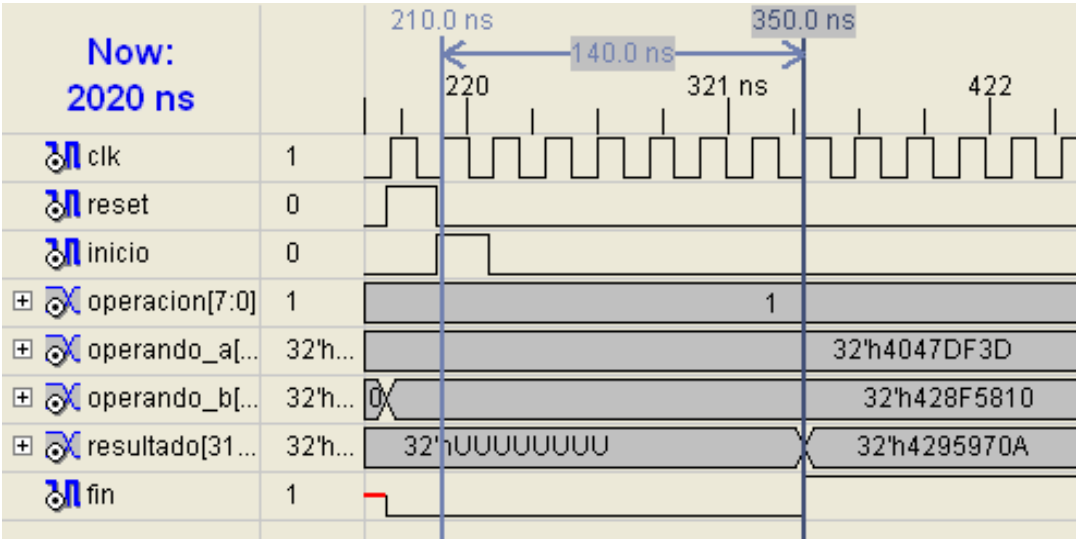


Figura 65: Cronograma de simulación de la suma

5.4 Operación Resta

La operación se codifica como operación=2 y se prueba con los siguientes valores:

Resultados		
Nombre	Valor decimal	Codificación en coma flotante
Operador_a	3,123	32’h4047Df3D
Operador_b	71,672	32’h428F5810
Resultado	-68.549	32’hC2891916
Ciclos de reloj	7	

Tabla 46: Condiciones de simulación de la resta

En la figura siguiente puede verse el cronograma de respuesta de la unidad.

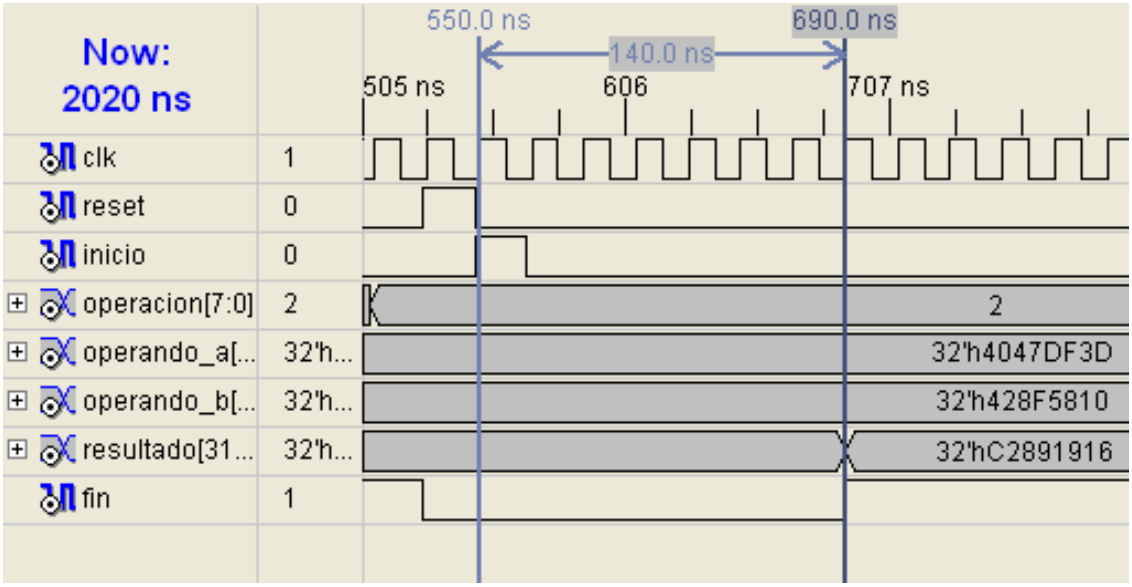


Figura 66: Cronograma de simulación de la resta

5.5 *Operación Producto*

La operación se codifica como operación=3 y se prueba con los siguientes valores:

Resultados		
Nombre	Valor decimal	Codificación en coma flotante
Operador_a	3,123	32'h4047Df3D
Operador_b	71,672	32'h428F5810
Resultado	223,832	32'h435FD4E9
Ciclos de reloj	10	

Tabla 47: Condiciones de simulación del producto

En la figura siguiente puede verse el cronograma de respuesta de la unidad.

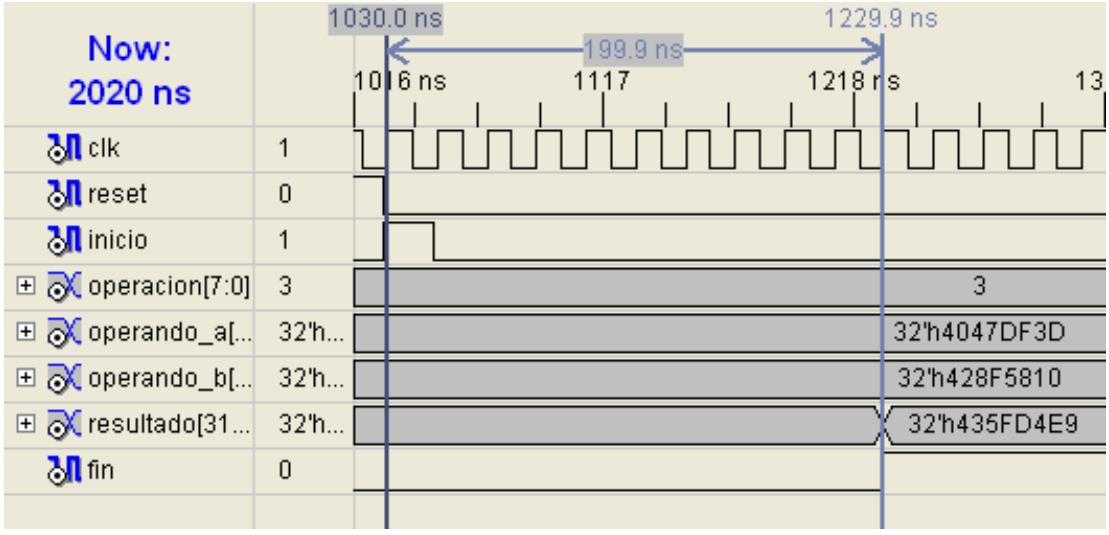


Figura 67: Cronograma de simulación del producto

5.6 *Operación Cociente*

La operación se codifica como operación=4 y se prueba con los siguientes valores:

Resultados		
Nombre	Valor decimal	Codificación en coma flotante
Operador_a	3,123	32’h4047Df3D
Operador_b	71,672	32’h428F5810
Resultado	0,0435735	32’h3D327A22
Ciclos de reloj	30	

Tabla 48: Condiciones de simulación del cociente

En la figura siguiente puede verse el cronograma de respuesta de la unidad.

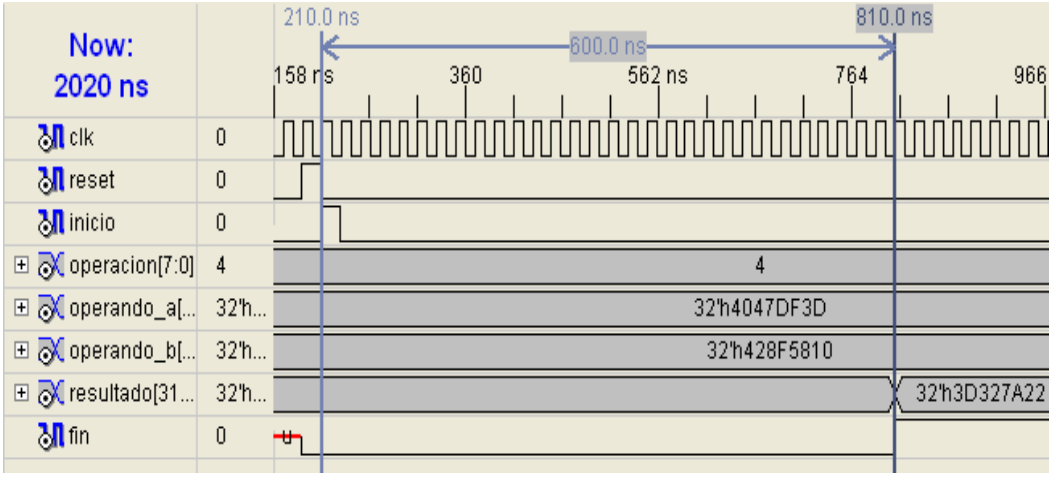


Figura 68: Cronograma de simulación del cociente

Si los operandos son múltiplos entre si el tiempo de respuesta se acorta:

Resultados		
Nombre	Valor decimal	Codificación en coma flotante
Operador_a	3,123	32’h4047DF3D
Operador_b	6,246	32’h40C7 DF3D
Resultado	0,5	32’h3F000000
Ciclos de reloj	8	

Tabla 49: Condiciones de simulación del cociente exacto

En la figura siguiente puede verse el cronograma de respuesta de la unidad.

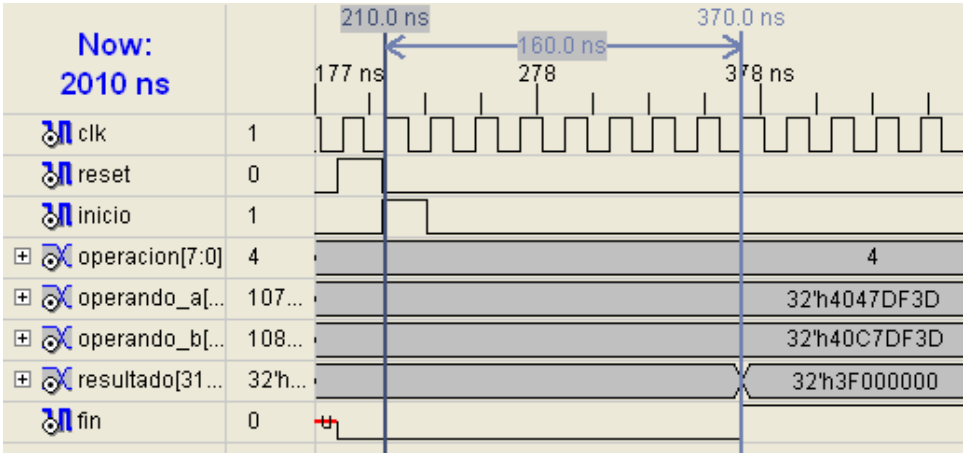


Figura 69: Cronograma de simulación del cociente exacto

5.7 Operación Inversa

Se trata del mismo bloque que realiza la operación anterior con la salvedad que el operador_b no se utiliza para nada, es por tanto innecesario escribirlo, la operación se codifica con el valor 5.

Resultados		
Nombre	Valor decimal	Codificación en coma flotante
Operador_a	3,123	32'h4047DF3D
Resultado	0,320205	32'h3EA3F1E4
Ciclos de reloj	32	

Tabla 50: Condiciones de simulación de la inversa

En la figura siguiente puede verse el cronograma de respuesta de la unidad.

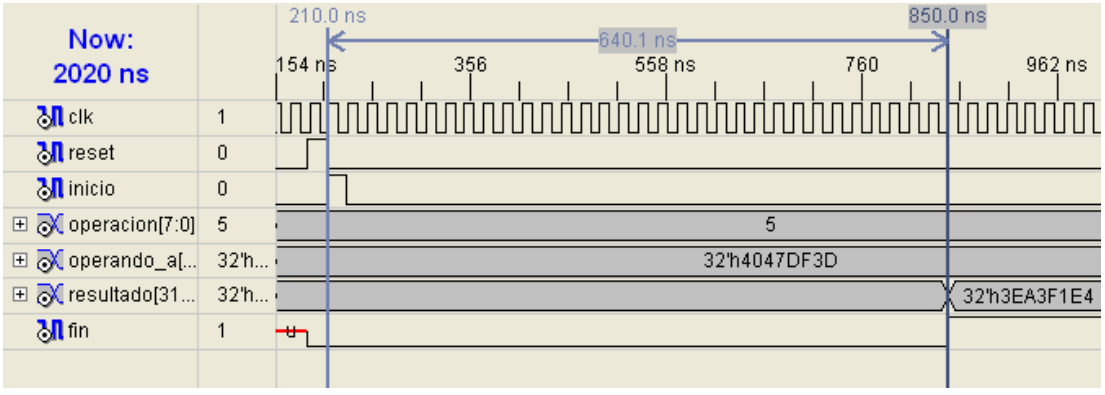


Figura 70: Cronograma de simulación de la inversa

5.8 *Operación Raíz cuadrada*

Se trata del mismo bloque que realiza la operación anterior con la salvedad que el operador_b no se utiliza para nada, es por tanto innecesario escribirlo.

Resultados		
Nombre	Valor decimal	Codificación en coma flotante
Operador_a	3,123	32'h4047DF3D
Resultado	1,7672	32'h3FE233A6
Ciclos de reloj	32	

Tabla 51: Condiciones de simulación de la raíz cuadrada

En la figura siguiente puede verse el cronograma de respuesta de la unidad.

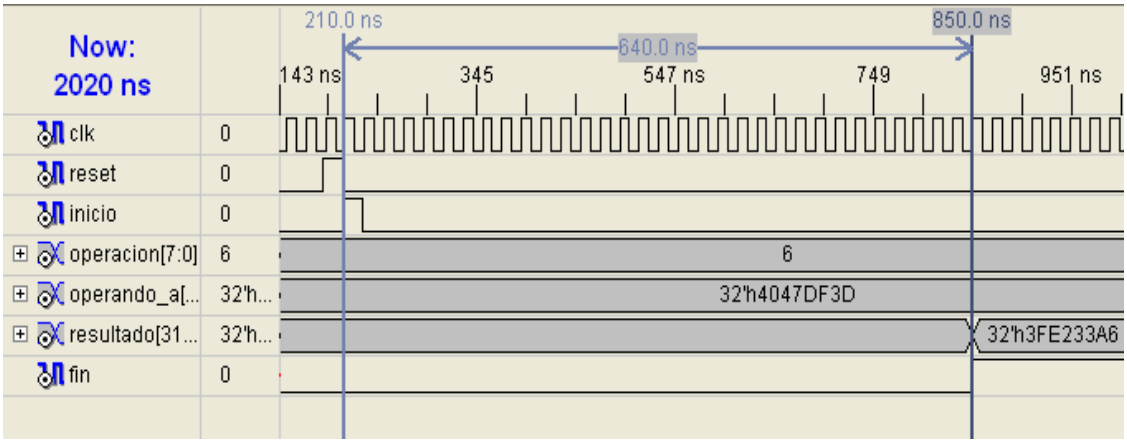


Figura 71: Cronograma de simulación de la raíz cuadrada

5.9 *Operación Seno*

Se trata del mismo bloque que realiza la operación anterior con la salvedad que el operador_b no se utiliza para nada, es por tanto innecesario escribirlo.

Resultados		
Nombre	Valor decimal	Codificación en coma flotante
Operador_a	0,32	32'h3EA3D70A
Resultado	0,844328	32'h3F5825E0
Ciclos de reloj	44	

Tabla 52: Condiciones de simulación del seno

En la figura siguiente puede verse el cronograma de respuesta de la unidad.

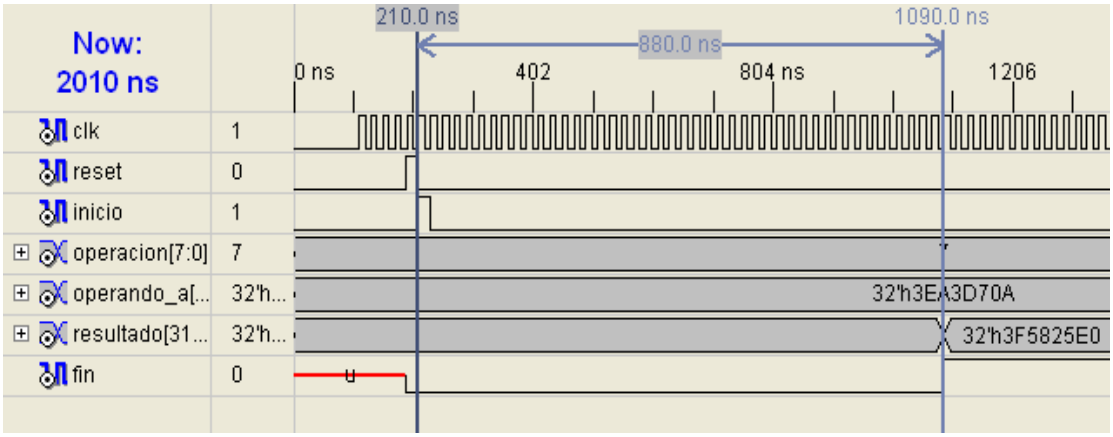


Figura 72: Cronograma de simulación del seno

5.10 Operación Coseno

Se trata del mismo bloque que realiza la operación anterior con la salvedad que el operador_b no se utiliza para nada, es por tanto innecesario escribirlo.

Resultados		
Nombre	Valor decimal	Codificación en coma flotante
Operador_a	0,32	32'h3EA3D70A
Resultado	0,535827	32'h3F092BF2
Ciclos de reloj	44	

Tabla 53: Condiciones de simulación del coseno

En la figura siguiente puede verse el cronograma de respuesta de la unidad.

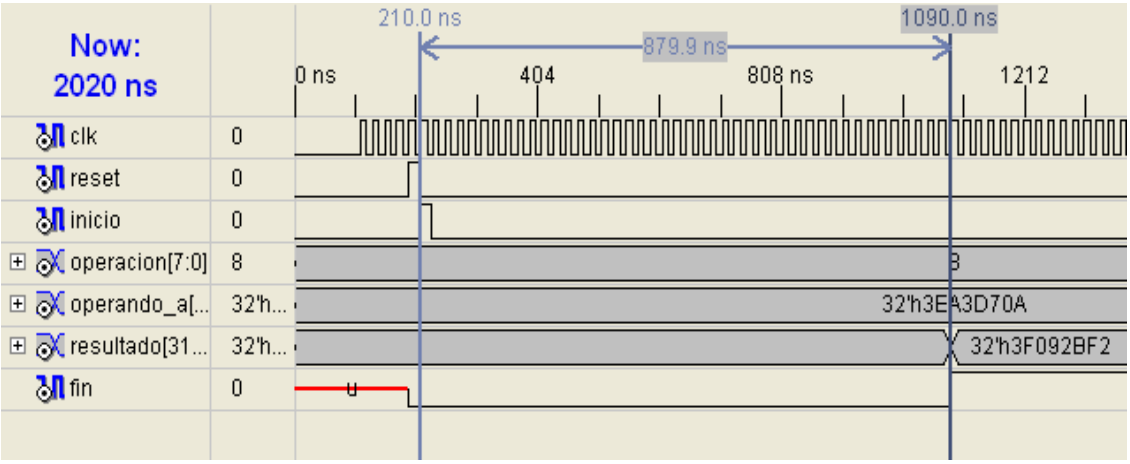


Figura 73: Cronograma de simulación del coseno

5.11 Operación Tangente

La tangente combina el algoritmo CORDIC para la obtención de seno y el coseno y el algoritmo de la división para la obtención de la tangente por la división de los dos anteriores.

Resultados		
Nombre	Valor decimal	Codificación en coma flotante
Operador_a	0,32	32’h3EA3D70A
Resultado	1,57575	32’h3FC9B21A
Ciclos de reloj	78	

Tabla 54: Condiciones de simulación de la tangente

En la figura siguiente puede verse el cronograma de respuesta de la unidad.

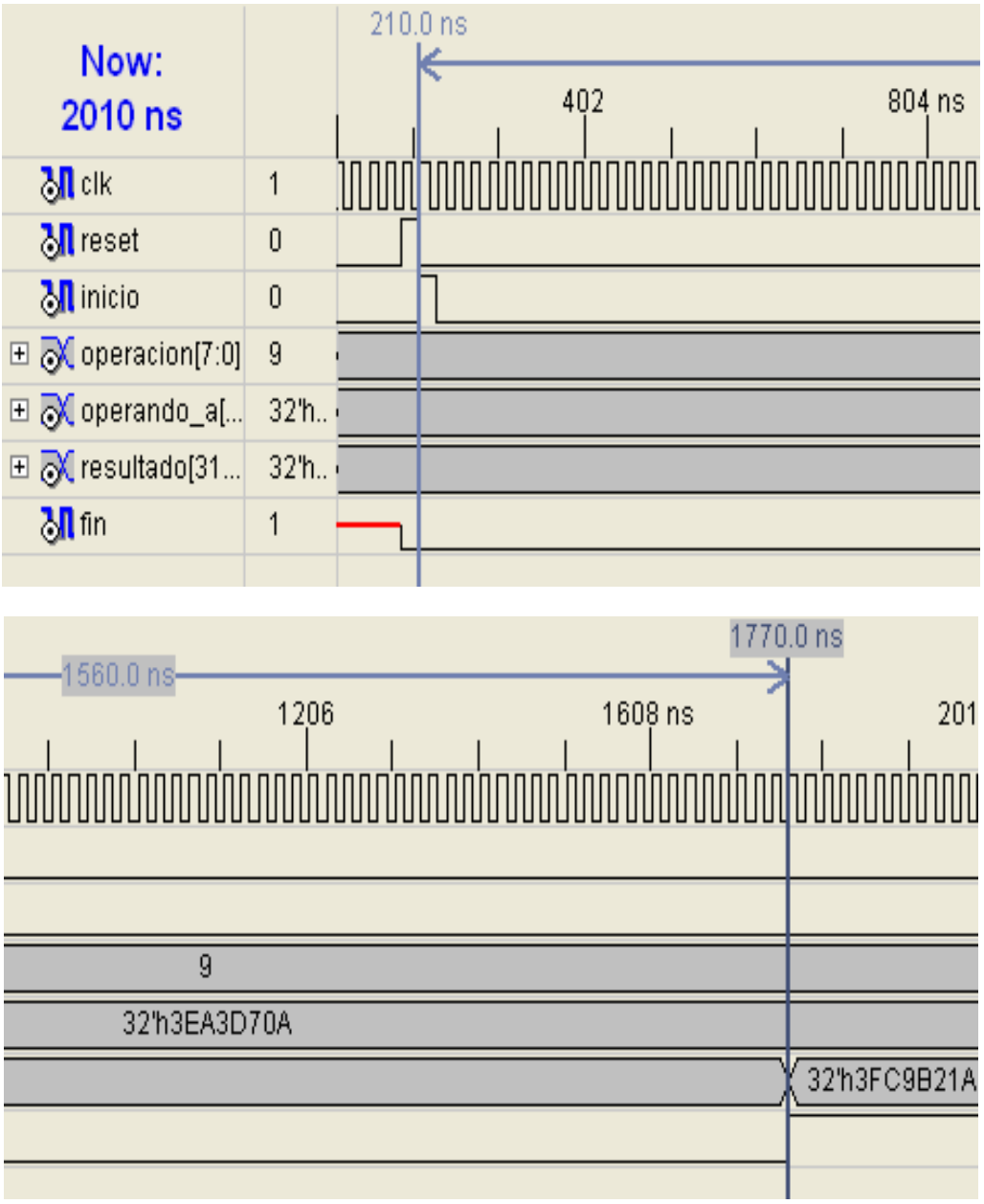


Figura 74: Cronograma de simulación de la tangente

5.12 Resumen

El siguiente cuadro incluye el resumen de los resultados obtenidos en las simulaciones, que coinciden con los resultados obtenidos en la experimentación.

Los valores expresados corresponden únicamente al tiempo que la unidad de coma flotante invierte en resolver los algoritmos implementados, pero hay que considerar que el objetivo de esta unidad es ser conectada a un microprocesador para resolver las operaciones que éste necesite y por tanto hay que tener en cuenta el tiempo de escritura de los datos y de lectura del resultado para valorar el tiempo real que se necesita para cada operación.

Resultados	
Operación	Ciclos de reloj
Suma	7
Resta	7
Producto	10
Cociente	30
Inversa	32
Raíz cuadrada	32
Seno	44
Coseno	44
Tangente	78

Tabla 55: Duración de las operaciones (ciclos de reloj)

6 Implementación física

6.1 Recursos consumidos de FPGA y frecuencia de funcionamiento

Un dato a tener en cuenta en diseños basados en FPGA es la cantidad de recursos necesarios para implementar el sistema, que en nuestro caso particular es[1]:

```
Device utilization summary:
---------------------------
Selected Device : 3s2000fg676-4

 Number of Slices:                     2103  out of  20480    10%
 Number of Slice Flip Flops:           1596  out of  40960     3%
 Number of 4 input LUTs:               3859  out of  40960     9%
 Number of IOs:                         109
 Number of bonded IOBs:                   0  out of    489     0%
 Number of MULT18X18s:                    1  out of     40     2%
=========================================================================
```

Tabla 56: Resumen de recursos de la FPGA consumidos

[1] Datos obtenidos del informe de síntesis de la herramienta de síntesis de XILINX

Y el estudio de tiempo arroja un resultado de periodo mínimo de 13.427 ns con una frecuencia máxima de funcionamiento de algo más de 74 MHz. [1]

```
Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 13.427ns (Maximum Frequency: 74.477MHz)
   Minimum input arrival time before clock: 10.748ns
   Maximum output required time after clock: 0.720ns
   Maximum combinational path delay: No path found
```

Tabla 57: Path crítico y frecuencia máxima de trabajo

Del informe anterior se obtiene que los datos han de estar escritos 10,784ns antes del inicio del proceso y estarán disponibles a la salida 0,720 ns después del ciclo de reloj pertinente, se deduce con facilidad que un ciclo de reloj de margen para las dos operaciones es suficiente.

6.2 Hardware de pruebas.

Para testear el funcionamiento de la unidad se conecta a un microprocesador MicroBlaze de Xilinx definido en VHDL, según el siguiente diagrama de bloques:

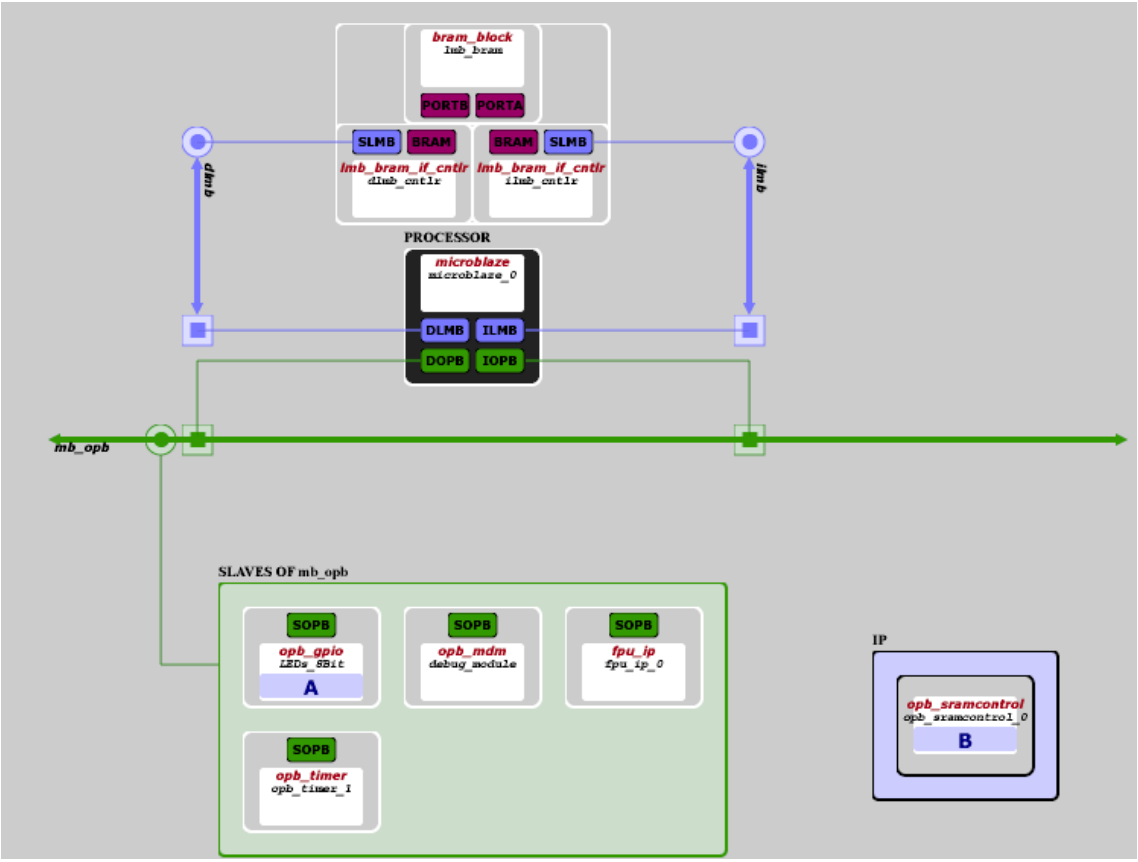


Figura 75: Esquema del hardware de pruebas

Donde, además del bloque de la unidad FPU, se añaden los siguientes bloques conectados al bus OPB:

- **LEDs_8bits:** Indicador de 8 leds para monitorizar bits.
- **Debug_module:** Necesario para debugar el código de programa en C
- **Opb_timer1:** Timer para medir tiempo de proceso de diversas secciones de los programas.
- **Opb_sramcontrol:** Driver de acceso a la memoria RAM de 2MB externa al procesador.

El procesador dispone de unidad en coma flotante que realiza sumas, restas, productos y cocientes. Se compara el funcionamiento de la unidad diseñada con en la ejecución con la FPU interno y sin ella. También se compara el funcionamiento si se usa la memoria interna del procesador si se usa la externa o si se usa una combinación de ellas.

6.3 Acceso al bus OPB

Para la conexión de la unidad diseñada al bus OPB es necesario realizar un driver de conexión que facilite la escritura de datos desde el bus y hacia el bus y da acceso al control de la unidad para realizar el "reset" y para iniciar la operación, se puede ver a continuación el diagrama de bloques interior de la entidad "**fpu_ip_0**" que contiene al driver y a la unidad FPU.

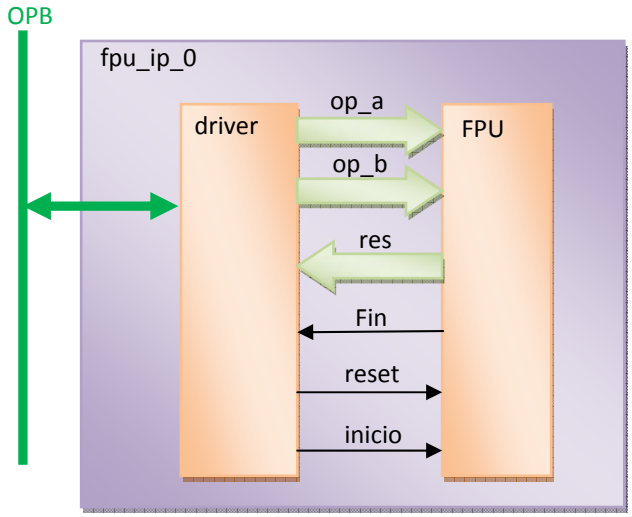


Figura 76: Estructura interna del driver OPB

El driver detecta cuando el bus OPB direcciona al módulo fpu_ip_0 según el siguiente cuadro de direcciones asignadas por el XPS2, por tanto la unidad tiene asignadas 512K con la dirección 0x7A000000 como dirección base.

[2] Xilinx Platform Studio (Herramienta de desarrollo del Xilinx)

Name	Address	Base Address	High Address	Size	Lock	Instance	Bus Connection
				U		mb_opb	
SLMB		0x00000000	0x0000ffff	64K		dlmb_cntlr	dlmb
SLMB		0x00000000	0x0000ffff	64K		ilmb_cntlr	ilmb
SOPB		0x40000000	0x4000ffff	64K		LEDs_8Bit	mb_opb
SOPB		0x41400000	0x4140ffff	64K		debug_module	mb_opb
SOPB		0x41c00000	0x41c0ffff	64K		opb_timer_1	mb_opb
SOPB		0x70a00000	0x70bfffff	2M		opb_sramcontrol_0	mb_opb
SOPB		0x7a000000	0x7a07ffff	512K		fpu_ip_0	mb_opb

Tabla 58: Direcciones del bus OPB

El driver realiza las siguientes funciones según la dirección que el bus OPB indique en cada momento.

- **BASE+00, escritura:** el dato recibido corresponde al operador_a.
- **BASE+04, escritura:** el dato recibido corresponde al operador_b.
- **BASE+08, escritura:** el dato recibido en los 8 bits menos significativos del bus de datos corresponde a la operación deseada, tras escribir la operación se inicia el cálculo.
- **BASE+12, escritura:** inicio del cálculo, el dato recibido es irrelevante.
- **BASE+16, escritura:** reset de la unidad, el dato recibido es irrelevante.
- **BASE+00, lectura:** se escribe en el bus de datos el operador_a.
- **BASE+04, lectura:** se escribe en el bus de datos el operador_b.
- **BASE+08, lectura:** se escribe en los 8 bits menos significativos del bus de datos la última operación realizada.
- **BASE+20, lectura:** se escribe en el bus de datos el resultado de la operación y posteriormente se fuerza el reset de la unidad.
- **BASE+24, lectura:** se escribe en el bus de datos el estatus de la operación según el siguiente código:
  - **0 → En espera u operando.**
  - **1 → Operación finalizada.**
- **BASE+otro, escritura/escritura:** se ignora el dato y no se realiza ninguna función.

La unidad se "resetea" por software, tal y como se indica en el siguiente apartado, o por hardware cuando se "resetea" todo el sistema (microprocesador y demás coprocesadores existentes) mediante el pulsador externo.

6.4 Librería ”C” de acceso al módulo FPU

Para acceder a la FPU desde un programa en C es suficiente con escribir en la posición de memoria correspondiente a la operación que se desea realizar según se especifica en la”Tabla 58: Direcciones del bus OPB” vista en el apartado anterior.

Se han definido los siguientes punteros para el acceso a las diversas posiciones de memoria que controlan la unidad FPU.

```
//=====================================================
#define  FPU_IP_BASEADDRESS (int)0x7a000000
//punteros de operadores y control a memoria para datos FPU
float *fpu_datos = FPU_IP_BASEADDRESS; //+0 operando 1 (escritura o lectura)
                                       //+1 operando 2 (escritura o lectura)
                                       //+3 inicio operación
                                       //(escritura - valor indiferente)
                                       //+4 reset fpu
                                       //(escritura - valor indiferente)
                                       //+5 resultado (lectura)


int *fpu_operacion = FPU_IP_BASEADDRESS+8;     //operación (8 bits bajos)
                                               //escritura o lectura

int *fpu_status = FPU_IP_BASEADDRESS+24;       //status de la operación
                                               //lectura - bit bajo
                                               // 0-> operando
                                               // 1-> fin de la operación
```

Se han definido las siguientes funciones para el uso de la unidad FPU.

RESET de la unidad

Se procede a realizar un reset (por sofrware) de la unidad cuando se escribe un dato cualquiera en la posición de memoria indicada en el puntero “*(fpu_datos+4)”, la siguiente función “C” realiza la escritura.

```
//Reset de la unidad
void reset_fpu (void)
      {
            *(fpu_datos+4) = 0x0;            //reset fpu
            return;
      }
```

Acceso a la unidad para operación de dos operandos (suma, resta, producto, cociente)

Se procede de la siguiente manera: se escribe el primer operador en la posición de memoria indicada por el puntero ”*fpu_datos”, el segundo en la que indica el puntero ”*(fpu_datos+1)” y la operación en la que indica el puntero “*fpu_operación”. Es esta última escritura la que inicia el proceso de cálculo.

La función espera a que se concluya el cálculo mediante la observación de la posición de memoria indicada por el puntero “fpu_status” y finalmente se lee el resultado de la posición indicada por el puntero”*(fpu_datos+5)” para devolver el valor.

```
//operación de dos operandos
float operar_fpu_2 (float A, float B, int operacion)
	{
		//Acceso a mi FPU
		*fpu_datos = A;			//operador_a
		*(fpu_datos+1)= B;		//operador_b
		*fpu_operacion = operacion;	//operacion e inicio
		while (*fpu_status==0){;}	//esperar final
		return *(fpu_datos+5);		//leer resultado

	}
```

Acceso a la unidad para operación de un operando (inversa, raíz, seno, coseno, tangente)

Se procede de la siguiente forma: se escribe el primer y único operador en la posición de memoria indicada por el puntero"*fpu_datos" y la operación en la que indica el puntero "*fpu_operación". Es esta última escritura la que inicia el proceso de cálculo.

La función espera a que se concluya el cálculo mediante la observación de la posición de memoria indicada por el puntero "fpu_status" y finalmente se lee el resultado de la posición indicada por el puntero"*(fpu_datos+5)" para devolver el valor.

```
//operación de un solo operando.
float operar_fpu_1 (float A, int operacion)
	{
		//Acceso a mi FPU
		*fpu_datos = A;			//operador_a
		*fpu_operacion = operacion;	//operación e inicio
		while (*fpu_status==0){;}	//esperar final
		return *(fpu_datos+5);		//leer resultado

	}
```

6.4.1 Ejemplos de uso

A continuación se muestran ejemplos de uso de las funciones explicadas:

- Acceso a la FPU para división en coma flotante entre los valores "media" y "num_datos", el resultado se almacena en la variable "valor", se utiliza la función que escribe dos operadores.

  ```
  valor = operar_fpu_2(media , num_datos , 4); // dividir
  ```

- Acceso a la FPU para la raíz cuadrada en coma flotante del valor "parcial", el resultado se almacena en la variable "parcial", se utiliza la función que escribe un operador.

  ```
  parcial = operar_fpu_1 (parcial, 6) ; //raíz cuadrada
  ```

- Reset por software del la unidad.

  ```
  reset_fpu();
  ```

6.5 Software de pruebas

Se han desarrollado dos tipos de programas de pruebas, uno para medir el tiempo necesario para realizar cada operación, considerando el tiempo de acceso a memoria y el tiempo de proceso, y un segundo tipo en el que se resuelve un algoritmo determinado y se compara resultados.

En ambos casos se han compilado y ejecutado en los siguientes supuestos para comparar resultados:

- Compilado todo en BRAM, todo en SRAM y compartido.
- Ejecutado sin FPU, con la FPU del micro activa y con el diseño objeto de este proyecto.

6.5.1 Software de medición

Con la interacción del "timer OPB" y la realización de la misma operación múltiples ocasiones sobre valores distintos medimos el tiempo medio de acceso a la unidad en ciclos. La frecuencia de funcionamiento del sistema es de 40 MHz.

El diagrama de bloques del programa es el siguiente:

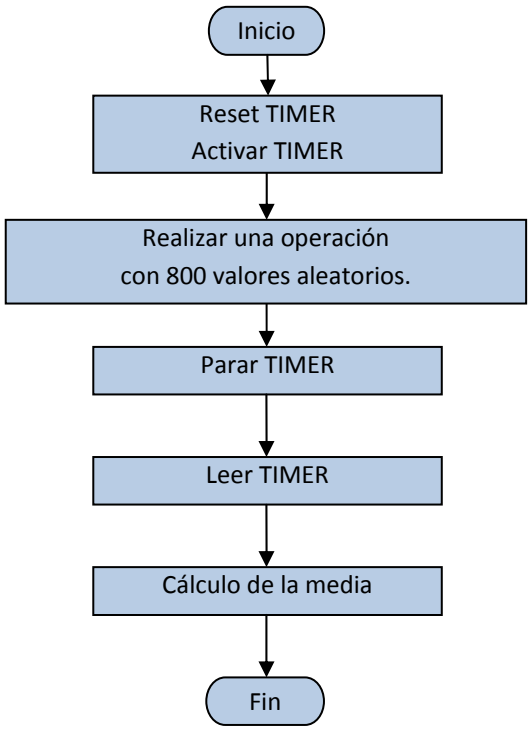


Figura 77: Diagrama de flujo del software de pruebas

Los valores obtenidos se monitorizan mediante el "debugger" que facilita la herramienta de desarrollo de XILINX.

6.5.2 Resultados obtenidos con el software de medición

Las siguientes tablas muestran el tiempo, en ciclos de reloj, que el procesador MicroBlaze ha invertido en resolver cada una de las operaciones.

La columna etiquetada "SIN FPU" recoge los ciclos de reloj que se han invertido en la ejecución de la operación realizada por software con la librería "Standard C Math", la columna etiquetada "CON FPU" muestra los invertidos en la ejecución de la operación con el uso de la unidad FPU que incorpora MicroBlaze [6] y la columna "FPU DISEÑADA" muestra los invertidos en la ejecución de la operación con la unidad diseñada en este proyecto.

La tabla con titulo "MEJORA" indica el factor de tiempo que se ha utilizado en resolver la operación en relación con el tiempo que ha invertido la FPU diseñada en este proyecto.

Puesto que el procesador utilizado puede utilizar memoria interna (BRAM) y memoria externa (SRAM) se han realizado pruebas en tres supuestos:

- Se compila todo el programa en BRAM, el tiempo de ejecución es pequeño pues el acceso a esta memoria es rápido. La memoria BRAM está limitada a 64 KB, razón por la cual es preciso disponer de memoria externa.

- Se compila todo el programa en SRAM, el tiempo de ejecución es grande pues el acceso a esta memoria se realiza mediante el bus OPB. La memoria externa en este caso es e 2 MB.

- Se compila en SRAM y se almacenan las variables en BRAM, el tiempo de ejecución se encuentra entre los dos casos anteriores.

6.5.2.1 Compilación en memoria interna

La compilación íntegra en memoria interna (BRAM) arroja los siguientes resultados: hay que considerar que el procesador tiene acceso directo a la memoria interna tanto para datos como para código de programa y como ésta es muy rápida la ejecución también lo es.

	Todo BRAM		
	SIN FPU	CON FPU	FPU DISEÑADA
Seno	16124	16124	129
Coseno	19840	19480	129
Tangente	34460	34460	159
Suma	448	40	107
Resta	484	40	118
Producto	607	40	122
División	672	40	137
Inversa	665	33	114
Raíz	3798	3799	114

Tabla 59: Ciclos invertidos en resolver las operaciones en compilación BRAM

La mejora experimentada indica que se resuelve entre 124 y 200 veces más rápido las funciones trigonométricas y 33 veces la raíz cuadrada tanto si se usa o no la FPU propia del procesado, para el resto de operaciones la mejora está entre 4 y 6 veces si no se usa la FPU y si se utiliza el proceso es más lento.

	Mejora experimentada	
	SIN FPU	CON FPU
Seno	124,99	124,99
Coseno	153,80	151,01
Tangente	216,73	216,73
Suma	4,19	0,37
Resta	4,10	0,34
Producto	4,98	0,33
División	4,91	0,29
Inversa	5,83	0,29
Raíz	33,32	33,32

Tabla 60: Mejora observada en compilación BRAM

Observando los resultados puede llegarse a las siguientes conclusiones:

- el microprocesador no usa la FPU interna para la resolución de las operaciones trigonométricas ni para la raíz cuadrada, dado que se invierten los mismos esté ésta habilitada o no.
- La FPU del micro es más rápida que la diseñada en este proyecto puesto que no necesita realizar acceso al bus OPB para operar.

6.5.2.1 Compilación en memoria externa

En este caso el procesador ha de obtener los datos y el código de programa de la memoria externa, pasando por tanto por el bus OPB, los tiempos de acceso a memoria son más grandes y por tanto la ejecución de programa más lenta.

	Todo SRAM		
	SIN FPU	CON FPU	FPU DISEÑADA
Seno	215348	96169	351
Coseno	208510	118433	351
Tangente	362412	205842	390
Suma	2974	146	393
Resta	2574	146	393
Producto	3380	146	393
División	3736	146	432
Inversa	3736	118	351
Raíz	22346	22596	351

Tabla 61: Ciclos invertidos en resolver las operaciones en compilación SRAM

En este caso la mejora es espectacular llegando a casi 1000 veces más rápido en cálculo de la tangente si no se usa la FPU y 500 si sí se usa. En cambio la FPU del micro sigue siendo más rápida que la diseñada.

	Mejora experimentada	
	SIN FPU	CON FPU
Seno	613,53	273,99
Coseno	594,05	337,42
Tangente	929,26	527,80
Suma	7,57	0,37
Resta	6,55	0,37
Producto	8,60	0,37
División	8,65	0,34
Inversa	10,64	0,34
Raíz	63,66	64,38

Tabla 62: Mejora observada en compilación SRAM

6.5.2.2 Compilación en memoria combinada

El procesador guarda parte del código en memoria interna y parte en memoria externa, es el compilador quien decide que datos van a cada sitio, los resultados se encuentran a medio camino entre los dos anteriores.

	Combinado BRAM-SRAM		
	SIN FPU	CON FPU	FPU DISEÑADA
Seno	46951	21381	199
Coseno	45239	26539	199
Tangente	78767	45755	239
Suma	631	81	205
Resta	603	81	205
Producto	767	81	205
División	796	81	225
Inversa	796	64	179
Raíz	4224	4297	179

Tabla 63: Ciclos invertidos en resolver las operaciones en compilación BRAM/SRAM

La mejora también se encuentra entre los datos obtenidos en las compilaciones anteriores, la FPU del procesador sigue siendo más rápida que la diseñada en este proyecto.

	Mejora experimentada	
	SIN FPU	CON FPU
Seno	235,93	107,44
Coseno	227,33	133,36
Tangente	329,57	191,44
Suma	3,08	0,40
Resta	2,94	0,40
Producto	3,74	0,40
División	3,54	0,36
Inversa	4,45	0,36
Raíz	23,60	24,01

Tabla 64: Mejora observada en compilación BRAM/SRAM

6.5.3 Algoritmos de prueba

Se han desarrollado dos programas que resuelven dos algoritmos que son los siguientes:

- Calculo de la media y la desviación típica de 800 valores aleatorios.

$$\bar{x} = \frac{\sum_{i=1}^{800} x_i}{800} \quad (22) \qquad \sigma = \sqrt{\frac{\sum_{i=1}^{800} \left(x_i - \bar{x}\right)^2}{799}} \quad (23)$$

- Resolución del siguiente algoritmo trigonométrico

$$N = \sum_{i=1}^{800} (\sin(\alpha_i)^2 + \cos(\alpha_i)^2) \quad (24)$$

Tal que el valor de N ha de resultar igual a 800 tras la ejecución.

Los objetivos de estos programas son los siguientes:

- Resolver un algoritmo trigonométrico y uno que no lo sea usando diseño software-hardware mediante la utilización de la FPU diseñada en este proyecto.
- Ver cómo se comporta la unidad diseñada cuando se combinan operaciones diversas.
- Comprobar la fidelidad de las operaciones usando como patrón el resultado obtenido mediante el uso de la librería "Standard C Math" propia del C.
- Comprobar la memoria necesaria para almacenar los programas en los dos casos.

Los resultados se presentan del mismo modo que en el apartado anterior y son los siguientes:

Todo BRAM					
	SIN FPU		CON FPU		FPU DISEÑADA
	Ciclos	Mejora	Ciclos	Mejora	Ciclos
Algoritmo estadístico	3883171	10,16	2529056	6,615	382320
Algoritmo trigonom.	30455395	66,83	28792480	63,18	455699

Todo SRAM					
	SIN FPU		CON FPU		FPU DISEÑADA
	Ciclos	Mejora	Ciclos	Mejora	Ciclos
Algoritmo estadístico	22981812	18,61	14957846	12,11	1234819
Algoritmo trigonom.	181530444	130,3	171713026	123,2	1393679

BRAM - SRAM					
	SIN FPU		CON FPU		FPU DISEÑADA
	Ciclos	Mejora	Ciclos	Mejora	Ciclos
Algoritmo estadístico	4675289	7,153	3096970	4,738	653638
Algoritmo trigonom.	40510002	52,85	38378335	50,07	766441

Tabla 65: Resultados observados en la resolución de algoritmos

En todos los casos se obtiene una mejora del resultado sobretodo con la resolución del algoritmo trigonométrico.

El valor numérico del resultado ha sido el mismo en todos los casos en los que se ha resuelto el algoritmo dando por verificada la fiabilidad de los algoritmos implementados.

La memoria necesaria para el almacenamiento de los programas que resuelven los algoritmos también varía en función de si se utiliza o no la FPU diseñada. En la tabla 66 se puede observar la reducción que se produce cuando el algoritmo se resuelve con la FPU diseñada.

Memoria utilizada					
	SIN FPU		CON FPU		FPU DISEÑADA
	Bytes	Mejora	Bytes	Mejora	Bytes
Algoritmo estadístico	15484	5,83	12824	4,83	2656
Algoritmo trigonom.	23992	16,56	21840	15,08	1448

Tabla 66: Memoria utilizada para la resolución de los algoritmos

7 Conclusiones y líneas futuras de trabajo

De los objetivos principales especificados en el apartado 2.5 se han asumido los siguientes:

- Se ha diseñado una unidad aritmética en coma flotante para operar con números de simple precisión según se describen en la norma IEEE 754.
- La FPU diseñada realiza las operaciones: suma, resta, producto, cociente, inversa, raíz cuadrada, seno, coseno y tangente.
- Se ha comprobado su correcto funcionamiento y se ha utilizado para resolver algoritmos de cálculo en los que se ha demostrado la mejora que su uso representa respecto de la resolución de los mismos algoritmos mediante software.

En cuanto a los objetivos secundarios se han asumido los siguientes.

- Se ha desarrollado una "interface" de conexión con el bus OPB y un "driver" en C que opera facilitan el intercambio de datos entre un microprocesador y la FPU diseñada.
- Se ha implementado en una FPGA el sistema, formado por microprocesador, bus y co-procesadores, y se ha comprobado su correcto funcionamiento.

Vistos los objetivos asumidos se puede asegurar que el diseño presentado es óptimo para las siguientes aplicaciones:

- Resolución de operaciones en coma flotante en procesadores que no dispongan de unidad propia.
- Resolución de aquellas operaciones que la unidad FPU del procesador no implementa.
- Utilización de la unidad diseñada en cualquier sistema que implemente un bus OPB o cualquier otro tipo de bus, mediante el desarrollo del driver adecuado.

- Implementación de la unidad o parte de ella dentro de sistemas que precisen la realización de operaciones en coma flotante de forma sistemática.

- Utilización de la unidad por co-procesadores conectados al bus OPB, sin que éstos deban depender del microprocesador para la resolución de operaciones en coma flotante. De esta forma se pueden resolver algoritmos de forma concurrente.

- Aun considerando que la unidad FPU del propio microprocesador es más rápida se puede repartir el cálculo entre la unidad interna y externa para optimizar el tiempo de proceso de un algoritmo determinado.

Hay que comentar que el presente proyecto es el punto de partida para el desarrollo de algoritmos complejos formados por la combinación de los bloques que forman la unidad. Es la necesidad de cada sistema o de cada algoritmo quien determina como resolver cada caso.

El uso de hardware reconfigurable (FPGA) permite implementar los bloques necesarios tantas veces como sea preciso, asegurando el funcionamiento concurrente de los mismos y optimizando los tiempos de ejecución, aunque una vez más, el dilema se encuentra en el equilibrio entre el área de FPGA ocupada y el tiempo de proceso deseado.

Se recomiendan las siguientes líneas de desarrollo para el presente proyecto:

- Optimización de los bloques para reducir el área de FPGA ocupada. Dejar área sin ocupar permite el aprovechamiento óptimo del dispositivo mediante la implementación de otros elementos o co-procesadores.

- Optimización de los bloques para aumentar la frecuencia máxima de trabajo. En este caso, la frecuencia máxima de trabajo de la unidad diseñada, sobre la FPGA en la que se ha implementado, es de 74MHz, este valor representa la máxima frecuencia a la que los elementos del sistema pueden trabajar.

- Desarrollo de bloques funcionales que resuelvan operaciones como logaritmos, exponenciales, operaciones con números complejos,… con el fin de aumentar las prestaciones del sistema.

- Mejora de la precisión del algoritmo CORDIC cuando se trabaja con ángulos próximos 0º y 90º.

- Implementar otras operaciones que puedan resolverse mediante el uso del algoritmo CORDIC.

- Optimización del driver de comunicación para reducir el tiempo de acceso al bus OPB y aumentar, de este modo, la velocidad de respuesta del conjunto.

Bibliografía

[1]-Algoritmos CORDICOS
Tesis de Esther Leyva Suárez
Instituto Politécnico Nacional, Escuela Superior de Física y Matemáticas
MEXICO D.C.

[2]-FPGA Implementation of Sine and Cosine Generators Using the CORDIC Algorithm
Tanya Vladimirova and Hans Tiggeler
Surrey Space Centre
University of Surrey, Guildford, Surrey, GU2 5XH

[3]-Máquinas algorítmicas como opción didáctica de sistemas digitales complejos
T. POLLÁN, B. MARTÍN y J. PONCE DE LEÓN
Escuela Universitaria de Ingeniería Técnica Industrial de Zaragoza.
Departamento de Ingeniería Electrónica y Comunicaciones. Universidad de Zaragoza. España

[4]-Special software & hardware development. Punto flotante y el estándar IEEE-754
Gabriel Dubatti
http://www.ingdubatti.com.ar/index.htm?artmath5.htm

[5]-On-Chip Peripheral Bus Architecture Specifications Version 2.1
IBM Corporation

[6] Xilinx® Microblaze processor
http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm

[7] Xilinx® Intellectual Property
http://www.xilinx.com/ipcenter/

[8]-Design and implementation of a Modular and Portable IEE 754 compliant Float Point Unit
Kingshuk Karuri, Rainer Leupers,
Gerd Ascheid, Heinrich Meyr
Institute for Integrated Signal Processing Systems, RWTH Aachen University, Germany

[9]-Diseño Digital Utilizando Lógicas Programables
Guillermo Güichal
Universidad Tecnológica Nacional
Facultad Regional Bahía Blanca (Argentina)

[10]-VHDL Lenguaje para síntesis y modelado de circuitos 2ª Edicion
Fernando Pardo, José A. Boluda
Editorial RA-MA 2004