UNIVERSIDAD DE GUANAJUATO

CAMPUS IRAPUATO - SALAMANCA

DIVISIÓN DE INGENIERÍAS

"Diseño e implementación de un módulo FPGA del álgebra del cuaternión aplicado a robótica"

TESIS

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN INGENIERÍA ELÉCTRICA

(Opción: Instrumentación y Sistemas Digitales)

PRESENTA:

Ing. Elizabeth Martínez Romero

DIRECTOR:

Dr. Mario Alberto Ibarra Manzano

CO-DIRECTOR:

Dr. David Camarena Martínez

Salamanca, Guanajuato Abril 2018

Salamanca, Gto., a 10 de Abril del 2018.

M. en I. HERIBERTO GUTIÉRREZ MARTIN
JEFE DE LA UNIDAD DE ADMINISTRACIÓN ESCOLAR
P R E S E N T E-

Por medio de la presente, se otorga autorización para proceder a los trámites de impresión, empastado de tesis y titulación al alumno(a) Martínez Romero Elizabeth del ***Programa de Maestría en*** Ingeniería Eléctrica y cuyo número de ***NUA*** es: 143491 del cual soy director. El título de la tesis es: Diseño e implementación de un módulo FPGA del álgebra del cuaternión aplicado a robótica

Hago constar que he revisado dicho trabajo y he tenido comunicación con los sinodales asignados para la revisión de la tesis, por lo que no hay impedimento alguno para fijar la fecha de examen de titulación.

A T E N T A M E N T E

Mario Alberto Ibarra Manzano
NOMBRE Y FIRMA
DIRECTOR DE TESIS
SECRETARIO

David Camarena Martínez
NOMBRE Y FIRMA
DIRECTOR DE TESIS

Juan Gabriel Aviña Cervantes
NOMBRE Y FIRMA
PRESIDENTE

Horacio Rostro G.
Dr. Horacio Rostro González
NOMBRE Y FIRMA
VOCAL

Agradecimientos Institucionales

Expreso mi más sincera gratitud hacia la Universidad de Guanajuato, especialmente a la División de Ingenierías del Campus Irapuato-Salamanca, por la formación que recibí. A todos los doctores que me instruyeron durante estos años que me permitieron obtener el título de maestro en ingeniería.

Este trabajo fue realizado gracias al apoyo recibido a través del Consejo Nacional de Ciencia y Tecnología de México, CONACYT, bajo la beca otorgada en la convocatoria titulada "BECAS NACIONALES", con el número de becario 302120.

Índice general

Índice de figuras

Índice de tablas

Resumen

En esta tesis se presenta el diseño e implementación del sistema para el álgebra del cuaternión, aplicados en el procesamiento de imágenes y robótica usando FPGA (por sus sigas en inglés Field Programmable Gate Array). El sistema esta integrado por 8 módulos que desarrollan las operaciones entre dos cuaterniones, suma, resta, multiplicación, división, complemento, norma, normalización e inverso; donde se pueden seleccionar uno de los doce resultados(ya que cuatro operaciones se repiten debido a que solo utilizan un cuaternión de entrada) de las operaciones mencionadas teniendo como entrada dos cuaterniones. Se presenta también un sistema de rotación el cual esta integrado por varios módulos del álgebra del cuaternión, este sistema se le aplica a imágenes del tipo RGB de tamaño 512×512 píxeles y se les puede rotar en los ejes X, Y y Z de manera particular o combinada. Donde la principal aportación del trabajo es la rotación de imágenes por medio de los cuaterniones. Además se presentan resultados y perspectivas del proyecto.

Capítulo 1

Introducción

Los cuaterniones son considerados una extensión de los números complejos, donde se puede conocer al cuaternión como un número hipercomplejo, el cual consta de cuatro partes, una parte real y tres imaginarias [1], los cuales fueron desarrollados por la necesidad de describir puntos en el espacio tridimensional. Una característica importante de los cuaterniones es su estrecha relación con las rotaciones en R^3, así como su principal ventaja de no singularidad y simplicidad comparada con los ángulos de Euler [2]. Estos números fueron introducidos por el físico matemático William Rowan Hamilton en 1843 [1] y desde entonces han sido aplicados en diferentes áreas de la investigación como se explicará a lo largo de este trabajo.

Por otra parte, desde que salio la primer FPGA a la venta en 1985, ha sido una herramienta muy importante y utilizada en diferentes ámbitos como el reconocimiento de voz, sistemas aeroespaciales, emulación de hadware, sistemas de visión y procesamiento digital de señales, por mencionar algunas, debido a que pueden programarse un sin número de veces, además son de bajo costo y se han caracterizado por su rapidez al procesar y entregar los resultados de forma paralela [3].

1.1. Objetivo

Diseñar e implementar una arquitectura en FPGA del álgebra del cuaternión (forma paralela) y la rotación de imágenes provenientes de una cámara embebida en un robot por medio de cuaterniones en FPGA.

1.2. Justificación

El principal problema para la detección de obstáculos, es construir el mapa en el cual está posicionado el robot, y esta tesis da una herramienta que ayudará a la ubicación del mismo. De esta manera, da una aportación para la formación del mapa. La ventaja de este proyecto es el uso de la FPGA, dado que es un dispositivo reconocido por su rapidez al procesar información, por lo cual se pudo trabajar con los cuaterniones.

Actualmente no se cuenta con investigaciones donde se realice el álgebra del cuaternión ni a los cuaterniones para hacer rotaciones de imágenes usando un FPGA, es por esta razón que se planeó este proyecto. Además se busca desarrollar un proyecto mayor para un sistema de detección de obstáculos en una FPGA; el principal problema del mismo es construir el mapa en el cual está posicionado analizando la rotación de la imagen.

1.3. Hipótesis

Es posible realizar la implementación en tiempo real del álgebra del cuaternión por medio de una FPGA, el cual consta de las operaciones básicas del mismo, que son suma, resta, multiplicación, división, normalización, complemento, norma e inverso del cuaternión, mismas que se utilizan para la rotación de las imágenes en 3 dimensiones X, Y y Z obtenidas por el robot.

1.4. Planteamiento

Al utilizar los ángulos de Euler en rotaciones de imágenes existe la probabilidad de que se indetermine la rotación, por lo cual se pretende utilizar los cuaterniones para evitar este problema y poder rotar imágenes sin inconvenientes. Es por esta razón que serán desarrollados dos bloques fundamentales, el primero de ellos corresponde a la aritmética del cuaternión y será la base para desarrollar; el segundo bloque, el cual realizará la rotación de un flujo de datos de una imagen en los tres ejes.

1.5. Antecedentes

Diversas investigaciones se han enfocado en el desarrollo de mapas de ocupación, mapas topológicos así como en los mapas métricos para representar el ambiente que percibe un robot. A su vez se han implementado algoritmos de navegación reactiva donde el robot responde de manera inmediata a los estímulos del entorno utilizando sensores ultrasónicos para la detección y evasión de obstáculos, además de usar la odometría para construir un mapa de ocupaciones de dos dimensiones en tiempo real, empleando la plataforma de robótica móvil DaNI 2.0 [4].

Por otra parte los cuaterniones se han utilizado en la representación de un cuerpo rígido para la manipulación de la rotación y translación, usando el método TRIAD y el Gauss-Newton [5], en el trabajo se utiliza una mesa suspendida en aire del grupo de desarrollo de sistemas aeroespaciales (GDSA), en el Instituto de Ingeniería de la UNAM. El proceso consta de la adquisición de datos por medio de una unidad de mediciones inerciales, procesamiento y visualización virtual.

El uso de una FPGA en la mayoría de las investigaciones es relevante debido a su rapidez de procesamiento, independientemente del área donde se este realizando, en una variedad de temas en diferentes campos como su uso

en la agencia de seguridad nacional de Estados Unidos [6], videojuegos [7], medicina [8] por mencionar algunos. Como se mostrará en el Capítulo 2, los cuaterniones son usados para filtros [9],imágenes [10], física [11], aeronáutica [5], gráficos de computadora entre muchas otras aplicaciones, sin embargo, ninguna se relaciona directamente con el trabajo realizado en esta tesis.

Diversas investigaciones tratan de eliminar los ángulos de Euler en rotaciones, por lo que proponen hacer el cambio a cuaterniones; otra aplicación de este cambio se puede ver en la investigación hecha por Joao Luis Marin [9] y su equipo de colaboradores, donde aplican los cuaterniones a un filtro Kalman para la estimación de la orientación de un cuerpo rígido, donde los cuaterniones proporcionan los parámetros del filtro de Kalman y quedan ecuaciones lineales, lo que produce una mayor agilidad en el procesamiento para poder obtener una aproximación en tiempo real.

Hay diferentes enfoques para la visualización de cuaterniones y sus relaciones con gráficos en computadoras. Andrew J. Hanson [12] utiliza rotaciones en tres dimensiones, y los convierte a cuaterniones para poderlos visualizar en cuatro dimensiones por medio de los programas OpenGl y QuaRot.

Otra aplicación de rotaciones de los cuaterniones para cuestiones de gráficos por computadora, es la realizada por Hart de la universidad de Washington [13], en la investigación realizada por Hart realiza un simulador del cuaternión a través de en un DGI Indigo Elan.

1.6. Organización de la tesis

El contenido de la tesis se dividió en cinco capítulos, los cuales se describen a continuación:

En el Capítulo 1 se da una introducción de este trabajo, así como el obje-

tivo principal, justificación, hipótesis y un panorama general de los trabajos realizados anteriormente relacionados con los temas principales de la tesis.

En el Capítulo 2 se presenta el estado del arte de diferentes investigaciones y aplicaciones en los cuales se han utilizado los temas claves de la tesis, en diferentes aplicaciones, debido a que no existe un trabajo, el cual relacione exactamente los mismos temas que en este trabajo de tesis fueron utilizados.

El Capítulo 3 se presentan los diseños de las arquitecturas realizadas para cada tipo de operación básica, así como el módulo de rotación y las operaciones que fueron diseñadas para poder implementarlo correctamente.

El Capítulo 4 muestra los resultados de los diseños e implementaciones de cada una de las operaciones y módulos realizados durante este trabajo, así como los recursos que cada una utilizó en la FPGA, además de una comparativa de los errores de cada uno de los resultados obtenidos en el trabajo, comparándolos con los esperados por medio de MATLAB®.

Finalmente, la conclusión y perspectivas de este trabajo se presentan en el Capítulo 5.

Capítulo 2

Estado del arte

En este capítulo se revisará la historia del cuaternión, las operaciones matemáticas básicas y complementarias, así como los trabajos que se han realizado usando cuaterniones, rotaciones de imágenes y FPGA .

2.1. Cuaternión

El cuaternión fue inventado en 1843 por Hamilton y a partir de su descubrimiento ha sido muy utilizado en diferentes áreas de las ciencias. Es importante conocer que un cuaternión esta constituido por 4 partes, una real y tres imaginarias, se puede describir como un número hipercomplejo. En la literatura se encuentra el uso del cuaternión aplicado en diferentes áreas las cuales se explicaran a continuación.

Ken Shoemake, de la universidad de Pensilvania, publicó un artículo llamado Quaternions [14], en el cual lo usa como un pequeño tutorial con material de los cuaterniones, así mismo aborda las rotaciones aplicado a gráficos en computadora y da un pequeño código para hacer una conversión entre un cuaternión a la matriz de rotación usando el programa C++ como interfaz. Habla también de los métodos utilizados en las curvas del cuaternión.

El trabajo de Vivianne Mahecha [10] detecta bordes en imágenes de color

usando **MATLAB®** y la transformada de Fourier del lado izquierdo, en la cual como utiliza los cuaterniones se puede dividir en dos transformadas de Fourier de dos dimensiones, las cuales se pueden sumar. así como su transformada inversa de Fourier del lado izquierdo para trabajar a la imagen como una sola entidad en el dominio de la frecuencia. El proceso que realiza es pasar una imagen a cuaternión, de ahí calcula la transformada de Fourier al cuaternión, una vez teniendo la transformada pasa por dos filtros, el primero detecta los bordes horizontales, mientras que el segundo filtra los bordes verticales; teniendo estos filtros combina los dos y al resultado se le aplica la transformada inversa de Fourier.

Los cuaterniones también han sido utilizados para la clasificación de espectros de la región espectral del infrarrojo cercano (near infrared, NIR) [11], las cuales se aplican a residuos sólidos de plásticos; crean un cuaternión con los coeficientes obtenidos calculando la distancia y se utilizan como parámetros para su clasificación.

Otra aplicación de los cuaterniones ha sido en la visión por computadora, Chenlei Guo lo expone en una conferencia del IEEE [15]. Él y su equipo realizaron un método el cual llaman Phase Spectrum Quaternion Fourier Transform (PQFT), representando el valor de cada píxel como un cuaternión compuesto con intensidad, color y movimiento ya que es menos trabajo computacional comparado con los siguientes cuatro métodos, Phase spectrum of Fourier Transform, Spectral Residual (RS), SaliencyToolBox (STB), Neuromorphic Vision Toolkit (NVT). Con base en estos cuatro métodos, el método propuesto detecta más objetos correctos en imágenes naturales y vídeos que los demás métodos, aparte de ser más rápido.

Hamilton necesitaba extender los números complejos para representar a dimensiones superiores del plano, por lo que llegó a la conclusión de que podía representarse en cuatro dimensiones, todo cuaternión puede representarse como la suma de un número real y tres números imaginarios, $q = a_r + a_i\hat{i} + a_j\hat{j} + a_k\hat{k}$. El cuaternión es una clase más amplia que los números

complejos, además son miembros de un cuerpo no conmutativo, por lo cual se debe de tener en cuenta que $\hat{i}^2 = \hat{j}^2 = \hat{k}^2 = \hat{i}\hat{j}\hat{k} = -1$.

Definiendo a los elementos del cuaternión uno con la letra a y dependiendo el subindice será la parte real a_r y sus partes imaginarias a_i, a_j y a_k, mientras al segundo cuaternión lo representaremos con la letra b de la misma manera que el primero.

2.1.1. Suma y resta de cuaterniones

La suma y la resta de cuaterniones es semejante a la de los números complejos, se añaden o decrementan las partes reales y las imaginarias por separado para obtener como resultado un cuaternión. La suma de cuaterniones tiene las siguientes propiedades:

Es conmutativa:	$q_1 + q_2 = q_2 + q_1$
Es asociativa:	$(q_1 + q_2) + q_3 = q_1 + (q_2 + q_3)$

Por lo que la suma y resta de cuaterniones de forma matemática se realiza con las Ecuaciones 3.2.1 y 3.2.1.

$$q_1 + q_2 = (a_r + b_r) + (a_i + b_i)\hat{i} + (a_j + b_j)\hat{j} + (a_k + b_k)\hat{k} \tag{2.1}$$

$$q_1 - q_2 = (a_r - b_r) + (a_i - b_i)\hat{i} + (a_j - b_j)\hat{j} + (a_k - b_k)\hat{k} \tag{2.2}$$

2.1.2. Multiplicación de cuaterniones

La multiplicación es un poco más larga de desarrollar, ya que no se tiene relación directa como en los números reales debido a que la multiplicación no es conmutativa. El resultado de esta operación es un cuaternión, en el cual se debe de tener especial cuidado en las multiplicaciones de las partes imaginarias de acuerdo a la Tabla de Cayley (Tabla 2.1). Las propiedades de la multiplicación son las siguientes:

No es conmutativa: $q_1 * q_2 \neq q_2 * q_1$

Es asociativa: $(q_1 * q_2) * q_3 = q_1 * (q_2 * q_3)$

*	**1**	**i**	**j**	**k**
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

Tabla 2.1: Tabla de Cayley para la multiplicación de cuaterniones.

También es posible desglosar la multiplicación de cuaterniones y definirla con la Ecuación 3.2.2 de una manera agrupada utilizando la misma, ya que a Hamilton le costó mucho trabajo deducir las reglas para multiplicar sus cuaterniones.

Para calcular la multiplicación de cuaterniones de una manera ordenada y a la cual se llega con la ecuación Ecuación 3.2.2 se utiliza la Tabla 2.2, sea $q_1 = a_r + a_i\hat{i} + a_j\hat{j} + a_k\hat{k}$ y $q_2 = b_r + b_i\hat{i} + b_j\hat{j} + b_k\hat{k}$, y recordando que la multiplicación no es conmutativo la tabla aplica para la multiplicación de $q_1 * q_2$ por lo que el primer cuaternion debe de escribirse en la columna, mientras que los elementos del segundo cuaternión deben de ir en la primer fila. Una vez que se ponen los datos de esa manera, se procede a multiplicar los elementos de renglón por columna,

q_1* q_2	b_r	$b_i\hat{i}$	$b_j\hat{j}$	$b_k\hat{k}$
a_r	$a_r b_r$	$a_r b_i\hat{i}$	$a_r b_j\hat{j}$	$a_r b_k\hat{k}$
$a_i\hat{i}$	$a_i b_r\hat{i}$	$a_i b_i\hat{i} * \hat{i}$	$a_i b_j\hat{i} * \hat{j}$	$a_i b_k\hat{i} * \hat{k}$
$a_j\hat{j}$	$a_j b_r\hat{j}$	$a_j b_i\hat{j} * \hat{i}$	$a_j b_j\hat{j} * \hat{j}$	$a_j b_k\hat{j} * \hat{k}$
$a_k\hat{k}$	$a_k b_r\hat{k}$	$a_k b_i\hat{k} * \hat{i}$	$a_k b_j\hat{k} * \hat{j}$	$a_k b_k\hat{k} * \hat{k}$

Tabla 2.2: Tabla de ayuda para la multiplicación de cuaterniones.

$$q_1 * q_2 = (a_r b_r - a_i b_i - a_j b_j - a_k b_k) + (a_i b_r + a_r b_i + a_j b_k - a_k b_j)\hat{i} +$$

$$(a_j b_r + a_k b_i + a_r b_j - a_i b_k)\hat{j} + (a_k b_r + a_j b_i + a_i b_j - a_r b_k)\hat{k} \quad (2.3)$$

2.1.3. División de cuaterniones

Esta operación se necesitan dos cuaterniones de entrada y es una combinación de tres operaciones de cuaterniones (multiplicación, complemento y módulo) como lo describe la Ecuación 3.2.3. Otra manera de visualizarlo, será con la explicación de la operación inverso, la cual se explica más adelante.

$$\frac{q_1}{q_2} = q_1 {q_2}^{-1} = \frac{q_1 \bar{q_2}}{|q_2|^2} \quad (2.4)$$

2.2. Operaciones complementarias

2.2.1. Complemento de un cuaternión

El complemento o conjugado de un cuaternión se reduce a negar la parte imaginaria del cuaternión, por lo cual, las partes imaginarias cambiarán de signo, es decir, si es positivo en el complemento será negativo y si es negativo en el complemento tendrá signo positivo. Si recordamos que habíamos definido $q_1 = a_r + a_i\hat{i} + a_j\hat{j} + a_k\hat{k}$, esto se puede escribir con la Ecuación 3.3.1.

$$\bar{q_1} = a_r - a_i\hat{i} - a_j\hat{j} - a_k\hat{k} \quad (2.5)$$

2.2.2. Norma de un cuaternión

La norma de un cuaternión siempre será positivo y un número real. Para obtenerla se sigue la Ecuación 3.3.2. Esta operación es usada en la división tal como se mostró en la Ecuación 3.2.3.

$$|q| = \sqrt{a_r^2 + a_i^2 + a_j^2 + a_k^2} \quad (2.6)$$

2.2.3. Normalización de un cuaternión

Si la norma o módulo de un cuaternión es igual a 1, se dice que es un cuaternión unitario, sin embargo, en la mayoría de los casos se presentan cuaterniones cuya norma o módulo no es igual a 1, debido a ello se recurre al uso de la normalizacion, para garantizar que la norma sea la unidad. La Ecuación 3.3.3 muestra la manera en la que se normaliza un cuaternión.

$$q_n = \frac{q}{|q|} \tag{2.7}$$

2.2.4. Inverso de un cuaternión

El inverso de un cuaternión se calcula para poder realizar la división de un par de cuaterniones. Con la Ecuación 3.3.4 se puede observar como se obtiene el inverso de un cuaternión usando también las operaciones anteriormente explicadas.

$$q^{-1} = \frac{\bar{q}}{|q|^2} \tag{2.8}$$

2.3. Conversiones

Como se sabe las conversiones son sumamente utilizadas en bastantes ámbitos, y han servido para expresar una misma unidad a distintas y poder operarlas de una mejor manera. En este caso al igual que las unidades existen conversiones las cuales ayudan a transformar en otra representación. Un cuaternión se puede transformar a diferentes representaciones y viceversa. En este trabajo se desarrollaron solo las dos primeras conversiones, pero se dejarán las ecuaciones de las demás posibilidades, las cuales se presentan a continuación.

Ángulos de Euler $\Longleftrightarrow$ cuaternión
Matriz de Rotación $\Longleftrightarrow$ cuaternión
Vector y ángulo $\Longleftrightarrow$ cuaternión

Entonces teniendo un cuaternión se puede convertir en una matriz de rotación, en un vector con dirección y en los ángulos de Euler, así como con esas tres representaciones se puede llegar a construir un cuaternión.

2.3.1. Conversión de ángulos de Euler a cuaternión

El conjunto de coordenadas angulares que especifican la orientación de un sistema de ejes ortogonales en movimiento, se llaman ángulos de Euler. Los algunos de Euler expresan la posición de un sistema de rotación con punto fijo mediante tres rotaciones sucesivas: Precesión α (Giro alrededor de un eje fijo), Nutación β (giro al rededor del eje perpendicular al fijo y a otro solidario) y Rotación propia γ (giro alrededor del eje solidario al cuerpo). En esta conversión necesita los tres ángulos de Euler mostrados en la Figura 2.1, cada uno genera un cuaternión, por si se quiere solamente trabajar con un ángulo. En el caso de que todos los ángulos tienen un valor, se genera otro cuaternión con la multiplicación de los tres cuaterniones intermedios, como lo muestra la Ecuación 2.9.

$$
\begin{aligned}
(\alpha, \beta, \gamma) \\
Q_x = [cos(\frac{\alpha}{2}), sen(\frac{\alpha}{2}), 0, 0] \\
Q_y = [cos(\frac{\beta}{2}), 0, sen(\frac{\beta}{2}), 0] \\
Q_z = [cos(\frac{\gamma}{2}), 0, 0, sen(\frac{\gamma}{2})] \\
Q = Q_x * Q_y * Q_z
\end{aligned} \tag{2.9}
$$

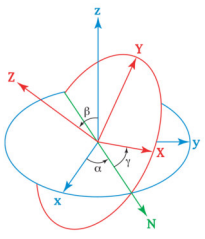

Figura 2.1: Representación gráfica de los ángulos de Euler.

2.3.2. Conversión de cuaternión a ángulos de Euler

Así como se puede realizar la conversión de ángulos de Euler a cuaternión [16], también se puede realizar en forma contraria, sea $q = a_r + a_i\hat{i} + a_j\hat{j} + a_k\hat{k}$ un cuaternión, se podrán encontrar los ángulos de Euler como lo describe la Ecuación 2.10.

$$\begin{bmatrix} \phi \\ \theta \\ \varphi \end{bmatrix} = \begin{bmatrix} (arctan[2(a_r a_i + a_j ak)]/(1 - 2(a_i^2 + a_j^2)) \\ arcsin[2(a_r a_j - a_k a_i)] \\ (arctan[2(a_r a_k + a_i aj)]/(1 - 2(a_j^2 + a_k^2)) \end{bmatrix} \tag{2.10}$$

2.3.3. Conversión cuaternión a matriz de rotación

Para el cálculo de la rotación se puede realizar como forma alterna por medio de la matriz de rotación [16], por lo tanto es importante poder tener dicha matriz, por está razón se define si se tiene el cuaternión $q = (a_r, a_i, a_j, a_k)$ donde se ponen los elementos de cuaternión en representación de coordenadas y se muestra la conversión de la matriz en la Ecuación 2.11.

$$R = \begin{bmatrix} a_r^2 + a_i^2 - a_j^2 - a_k^2 & 2a_i a_j - 2a_r a_k & 2a_i a_k + 2a_r a_j \\ 2a_i a_j + 2a_r a_k & a_r^2 - a_i^2 + a_j^2 - a_k^2 & 2a_j a_k - 2a_r a_i \\ 2a_i a_k - 2a_r a_j & 2a_j a_k + 2a_r a_i & a_r^2 - a_i^2 - a_j^2 + a_k^2 \end{bmatrix} \tag{2.11}$$

2.3.4. Conversión de vector y ángulo a cuaternión

Existe otra conversión, en la cual a partir de un vector con coordenadas en el eje X, Y y Z y su dirección [17], se puede formar un cuaternión de la manera que ilustra la Ecuación 2.12. Si $\theta, V = (a_x, a_y, a_y)$ donde a_x, a_y y a_z son las coordenadas del vector en su respectivo eje y recordando que el cuaternión tiene una parte real y tres imaginarias i,j y k quedaría formado

como lo describe la Ecuación 2.12.

$$q = cos(\frac{\theta}{2}) + (a_x)sen(\frac{\theta}{2})\hat{i} + (a_y)sen(\frac{\theta}{2})\hat{j} + (a_z)sen(\frac{\theta}{2})\hat{k} \tag{2.12}$$

2.3.5. Conversión de cuaternión a vector y ángulo

Teniendo un cuaternión $q = (a_r, a_i, a_j, a_k)$, y si se desea pasar a un vector y su dirección, se necesita calcular la magnitud de la parte imaginaria del cuaternión, y el vector resultante se forma normalizando las partes imaginarias del cuaternión como se muestra en las ecuaciones 2.13.

$$\begin{aligned} s &= \sqrt{a_i^2 + a_j^2 + a_k^2} \\ \theta &= \frac{2cos(a_r)}{s} \\ V &= \frac{(a_i, a_j, a_k)}{s} \end{aligned} \tag{2.13}$$

2.4. Rotación

G. F. Torres del Castillo [18] desarrolló la rotación y la reflexión usando los cuaterniones, el cual establece que siempre que se hagan rotaciones en tercera dimensión al rededor del origen, es la composición de dos reflexiones sobre el origen; en esté trabajo se muestra de forma matemática la relación de las rotaciones con en el espacio tridimensional, mediante el uso de las operaciones básicas del álgebra vectorial, sin embargo, esté artículo se manejan cuaterniones puros (sin parte real).

Ken Shoemake publicó en 1985 [14] un artículo en el cual explica matemáticamente la rotación por medio de cuaterniones y el beneficio de usar estos comparado con los ángulos de Euler, ademas de hacer estas comparativas debido a que utilizan curvas de cuaterniones, las cuales son usadas por medio de interpolaciones, llegando a la conclusión de que los interpolantes

de cuaternión spline son fáciles de usar e implementar, debido a que son robustos, eficientes, consistentes y flexibles comparado con las técnicas de transliteración geometrica (geometric tranliteración), ecuaciones diferenciales y mezclas de arco(arc blends).

Como se ha mencionado, se puede rotar un punto por medio de cuaterniones, la operación se realiza usando operaciones como las multiplicación, suma, complemento, además de el producto punto y cruz; con este conjunto de operaciones se realiza la rotación como se muestra en la Ecuación 2.14.

$$R(P) = qP\bar{q}R(P) = (q_r^2 - \bar{q} \cdot \bar{q})P + 2q_r(q \times P)q + 2\bar{q}(\bar{q} \cdot P) \tag{2.14}$$

Donde $R(P)$ será el punto rotado, P serán las coordenadas del vector a rotar, q como hasta ahora, será el cuaternión, el cual se dividirá en dos, el q_r es la parte real del cuaternión, mientras $\bar{q}$ es la parte imaginaria del cuaternión, el "$\cdot$representa la operación del producto punto y el "$\times$representa la operación producto cruz de dos vectores.

2.5. FPGA - Field Programmable Gate Array

Como se mencionó en el Capítulo 1 los FPGA (Field Programmable Gate Array) fueron introducidos al mercado en 1984 por la compañía Xilinx. Dicha platamorma utiliza bloques de lógica reprogramables los cuales se pueden implementar sin necesidad de armar circuitos de manera física o llegar a realizar algún circuito impreso. Entre las muchas cualidades que tiene un FPGA se pueden destacar las siguientes: rendimiento, rápida implementación, precio, fiabilidad y un fácil mantenimiento [19].

Las implementaciones en FPGA se usan en diferentes campos de investigación, por ejemplo Franchini y sus colaboradores realizaron una implementación del álgebra geométrica de Clifford en cuatro dimensiones, esta implementación tiene aplicaciones en robótica, visión y en gráficos, con elementos de longitud variable en una arquitectura CliffArchy la cual acepta

nuevos operandos para este tipo de álgebra [20].

Robert D. Tuney y Chris H Dick realizaron un trabajo en el cual usan la FPGA para la rotación y cambio de tamaño de imágenes en tiempo real [21], analizando los métodos de interpolador bicubica y el bilineal tradicional. Utilizan imágenes de 512×512 píxeles en donde las amplifican hasta 1024 y las reducen hasta 128.

Una implementación en FPGA con cuaterniones la presentan Ying He y Zhong-ke Shi en una investigación para determinar la posición, usando un algoritmo Peano-Baker, al realizarlo en FPGA aceleran el procesamiento al reutilizar módulos y usar la técnica del pipelining para optimizarlo; una de sus principales características es que se puede comparar su precisión con equipos comerciales que son costosos [22].

La multiplicación de cuaterniones ha sido la operación más investigada de la aritmética, Marek Parfeniuk [23] y sus equipo de investigación, diseñaron en MATLAB esta operación con coeficientes constantes, y reconocen que sus arquitecturas tienen sus ventajas y desventajas, su trabajo es equilibrado entre el rendimiento y los recursos utilizados, ellos proponen una arquitectura *Computadora Digital para Rotación de Coordenadas (CORDIC Inside Lifting)* la cual no tiene perdidas de los datos de entrada, pero al igual que en este trabajo de tesis, los autores reconocen que no es una arquitectura que utilice pocos recursos y es tardada en la implementación de la FPGA.

Existe también una implemetación para un banco de filtros digitales para unitarios basados en multiplicaciones de cuaterniones [24]. Verenik los sintetizó en una FPGA de la marca Xilinx basándose en una aritmética distribuida de punto fijo con un buen rendimiento en cuestiones del área en el chip.

Capítulo 3

Metodología

En este capítulo se presenta el diseño de un módulo de la aritmética del cuaternión; se describe cada una de las operaciones que lo comprende. Así mismo, se presenta el desarrollo del módulo de rotación, como sus variantes en las operaciones. Dichos módulos fueron diseñados en VHDL estándar, simulados en ModelSim ALTERA 10.1 y sintetizado en Quartus II 13.0

3.1. Descripción general

Se diseñó, desarrolló e implementó un módulo de la aritmética del cuaternión. Esté módulo consta de ocho operaciones, cuatro de ellas utilizan dos cuaterniones de entrada como la suma, resta, multiplicación y división; mientras que las otras aplican la operación complemento, inverso, norma y normalización de un cuaternión. Estas operaciones ya fueron explicadas en el capítulo anterior.

En la Figura 3.1 se muestra el bloque aritmético del cuaternión, a la izquierda se observan dos cuaterniones como entrada, el cuaternión A(morado) y el B(magenta), las cuales se dirigen a cada uno de los bloques. En la parte inferior se puede observar otra entrada representada por las letras OPC, la cual es la opción para desplegar el resultado, dependiendo de la operación

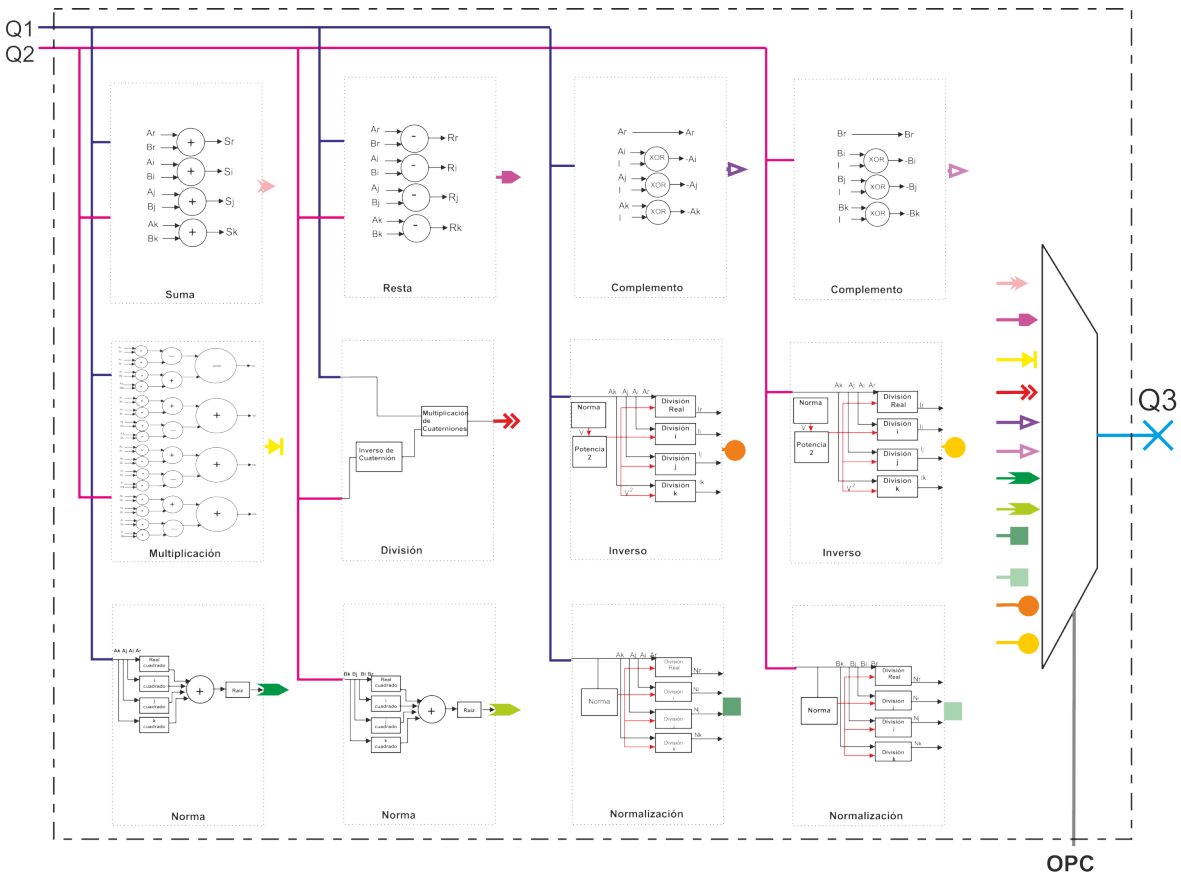

Figura 3.1: Diagrama del módulo de la aritmética del cuaternión.

seleccionada. Las opciones de desplegado del resultado se muestran en la Tabla 3.1, ademas cuenta con las entradas de reset y reloj.

En las siguientes secciones se explicarán cada una de las operaciones que realiza este módulo. Se explica el diseño de las ocho operaciones que conforman el bloque aritmético. Se dividirá en dos partes las operaciones, las que usan un solo cuaternión y las que usan dos cuaterniones para realizarla. Antes de diseñar cada una de las operaciones, se investigó la similitud a los números reales y sus diferencias, para identificar las similitudes y diferencias que tienen con las operaciones que normalmente se utilizan en la vida cotidiana, obteniéndolas se prosiguió a diseñarlas.

OPC	OPERACIÓN	Sim
0001	Suma	$Q_1 + Q_2$
0010	Resta	$Q_1 - Q_2$
0011	Multiplicación	$Q_1 \times Q_2$
0100	División	Q_1/Q_2
0101	Complemento Q_1	$\bar{Q_1}$
0110	Complemento Q_2	$\bar{Q_2}$
0111	Norma Q_1	$\|Q_1\|$
1000	Norma Q_2	$\|Q_2\|$
1001	Normalización Q_1	Q_{n1}
1010	Normalización Q_2	Q_{n2}
1011	Inverso Q_1	Q_1^{-1}
1100	Inverso Q_2	Q_2^{-1}

Tabla 3.1: Tabla de opciones para el despliegue de resultado de las operaciones realizadas por el módulo.

3.2. Operaciones básicas

3.2.1. Suma y resta de cuaterniones

Analizando las propiedades de estas dos operaciones se diseñaron estos dos módulos como se muestra en la Figura 3.2 y en la Figura 3.3. Las dos figuras muestran a mano izquierda dos cuaterniones de entrada, el cuaternión A y el cuaternión B, los cuales se suman o restan, esto es reales con reales e imaginarios con imaginarios, teniendo como resultado un cuaternión formado por las operaciones individuales. En la Figura 3.2 el cuaternión resultante es S mientras que en la Figura 3.3 es R.

$$q_1 + q_2 = (a_r + b_r) + (a_i + b_i)\hat{i} + (a_j + b_j)\hat{j} + (a_k + b_k)\hat{k}$$

$$q_1 - q_2 = (a_r - b_r) + (a_i - b_i)\hat{i} + (a_j - b_j)\hat{j} + (a_k - b_k)\hat{k}$$

Tanto la suma como la resta tienen dos cuaterniones de entradasy se obtiene un cuaternión de salida, estos tres cuaterniones estan formados por 4 partes como ya se ha explicado en el capítulo anterior en las Ecuaciones 3.2.1 y 3.2.1; cabe mencionar que cada parte del cuaternión es de 32 bits en for-

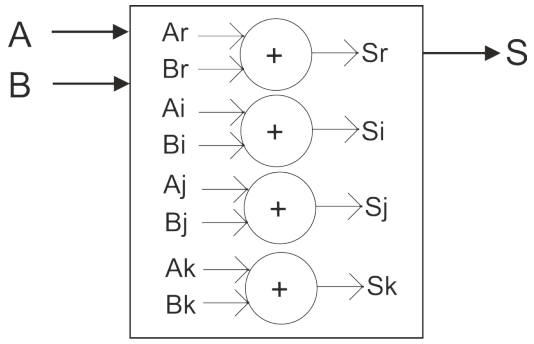

Figura 3.2: Diagrama del módulo de suma de cuaterniones.

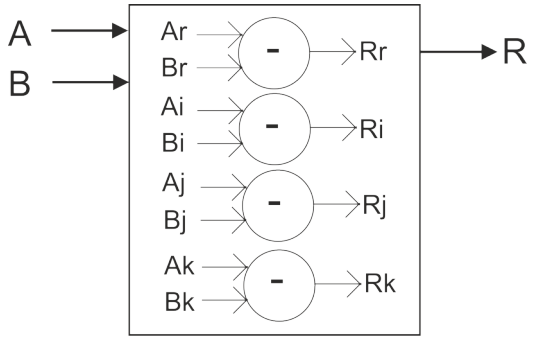

Figura 3.3: Diagrama del módulo de la resta de cuaterniones.

mato 5.27 usando complemento a 2.

3.2.2. Multiplicación de cuaterniones

Teniendo la multiplicación desglosada como lo describe la Ecuación 3.2.2, se planteó dividir la multiplicación en bloques, para que cada parte tenga sus sumas y multiplicaciones correspondientes como se muestra en la Figura 3.4.

$$q_1 * q_2 = (a_r b_r - a_i b_i - a_j b_j - a_k b_k) + (a_i b_r + a_r b_i + a_j b_k - a_k b_j)\hat{i} +$$

$$(a_j b_r + a_k b_i + a_r b_j - a_i b_k)\hat{j} + (a_k b_r + a_j b_i + a_i b_j - a_r b_k)\hat{k}$$

Esta operación necesita dos cuaterniones de entrada el cuaternión A y el cuaternión B, estos cuaterniones cuentan con 32 bits en cada parte que lo

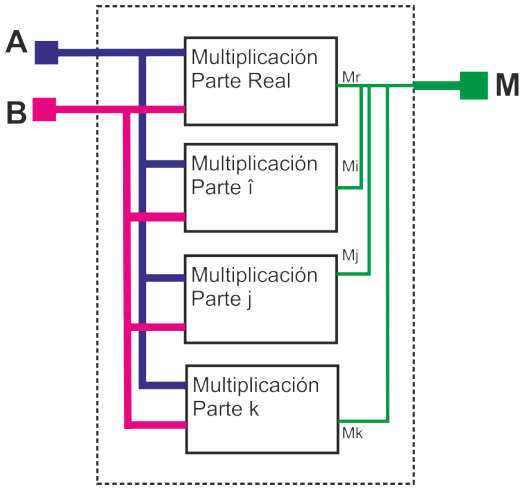

Figura 3.4: Diagrama del bloque de la multiplicación de cuaterniones.

conforman, con un formato de 5.27 (5 cifras enteras y 27 fraccionarias) en complemento a 2; mientras que la salida M se necesita desplegar en 64 bits cada parte que lo conforma con un formato de 10.54 (10 cifras enteras y 54 fraccionarias) en complemento a 2. Ahora se mostrará cómo está formada cada parte en la Figura 3.5, las cuales corresponden al diagrama interno de cada parte de la multiplicación, en el cual se puede apreciar que se realizaron 16 multiplicaciones y 12 sumas como lo describe la Ecuación 3.2.2.

3.2.3. División de cuaterniones

Esta operación necesita también dos cuaterniones de entrada A y B, los cuales generan un tercer cuaternión D, como se mostró con la Ecuación 3.2.3, la cual una vez explicado la operación inverso, se puede simplificar a que la operación división, utiliza la multiplicación y el inverso del cuaternión. Esta operación, necesita primero calcular el inverso del denominador, para posteriormente multiplicarlo con el numerador. Estos cuaterniones se despliegan en un formato 5.27 en complemento a 2 cada una de sus cuatro partes de 32 bits. Dicha operación no se puede realizar directamente como se está acostumbrado en los números reales, por lo cual realiza primero el complemento

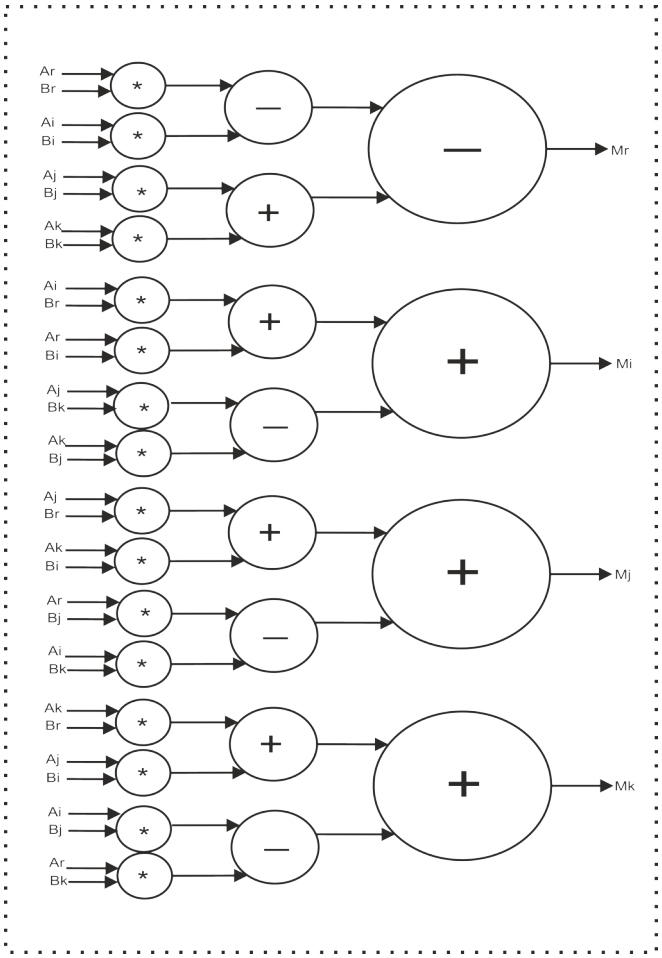

Figura 3.5: Diagrama de la parte interna de la multiplicación de cuaterniones.

del denominador y la norma del mismo, para calcular el inverso como se muestra en la Figura 3.6. Esta operación también utiliza el método de iteraciones sucesivas para poder llegar a desplegar el resultado. Para obtener la potencia cuadrada del cuaternión se utiliza la operación de multiplicación de cuaterniones, usando el mismo cuaternión en las dos entradas.

$$\frac{q_1}{q_2} = q_1 {q_2}^{-1} = \frac{q_1 \bar{q_2}}{|q_2|^2}$$

El método de iteraciones sucesivas utiliza la máquina de estados mostrada en la Figura 3.9, la cual controla el número de iteraciones que se deben

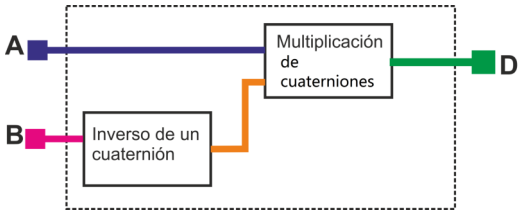

Figura 3.6: Diagrama de los bloques que conforman la división.

de realizar para obtener el resultado de la división. Si el resultado es igual al dividendo deja de iterar, si no es igual vuelve a iterar las veces necesarias, razón por la cual esta operación contiene retardo, ya que depende del número de bits utilizados y del número de ciclos que tarda en realizarlo.

3.3. Operaciones complementarias

3.3.1. Complemento de un cuaternión

El complemento o conjugado de un cuaternión se reduce a negar la parte imaginaria del cuaternión como se explicó en la sección 2.2.1 (Ecuación 3.3.1), por lo cual se crea un vector de unos haciendo la operación XOR para negar las partes imaginarias como se muestra en la Figura 3.7.

$$\bar{q_1} = a_r - a_i\hat{i} - a_j\hat{j} - a_k\hat{k}$$

Se debe recordar que la operación XOR se utiliza como inversor condicional [25], por esta razón se ha utilizado en este bloque, ya que solo se necesita invertir las partes imaginarias del cuaternión. Esta operación esta formada por un cuaternión de entrada, obteniendo un cuaternión de salida, estos dos cuaterniones están formados por cuatro partes, cada una de 32 bits en formato 5.27 y complemento a 2.

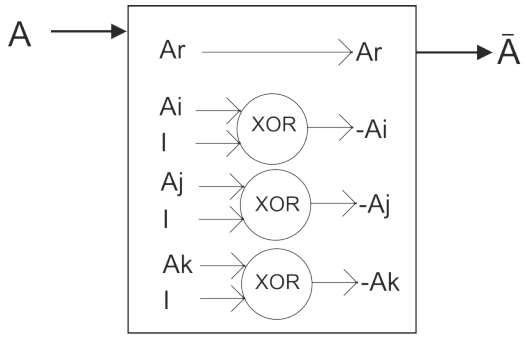

Figura 3.7: Diagrama a bloques del complemento de un cuaternión.

3.3.2. Norma de un cuaternión

Antes de explicar esta operación, es necesario explicar la operación de la raíz cuadrada, debido a que es utilizado para la realización de la operación de la norma o módulo de un cuaternión. El cálculo de la raíz esta basado en el método de aproximaciones sucesivas, el diagrama a bloques se muestra en la Figura 3.8.

$$|q| = \sqrt{a_r^2 + a_i^2 + a_j^2 + a_k^2}$$

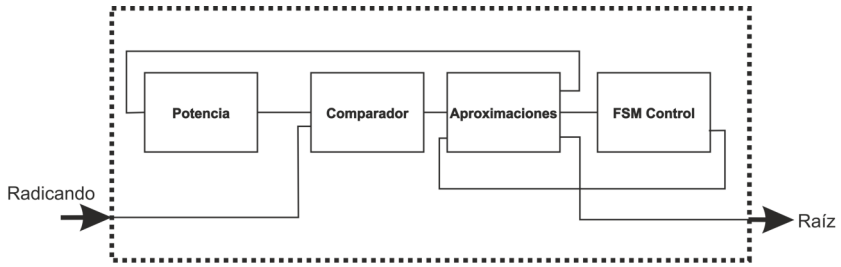

Figura 3.8: Diagrama a bloques de la raíz cuadrada.

Este bloque utiliza las aproximaciones sucesivas, comparador y la máquina de estados controla si vuelve a iterar o si la comparación es la correcta para obtener el resultado. En la Figura 3.9 se muestra el grafo de la máquina de estados utilizada. Esta máquina de estados controla el número de iteraciones que se deben de realizar en cada una de las operaciones. Si el resultado

es igual al radicando deja de iterar, si no es igual vuelve a iterar las veces necesarias, razón por la cual esta operación contiene retardo, ya que depende del número de bits utilizados el número de ciclos que tarda en llegar al resultado. La operación norma o módulo del cuaternión, como se explicó en el Capítulo 2 y lo rige la Ecuación 3.3.2, utiliza un cuaternión de entrada y genera uno de salida, los dos con sus cuatro partes de 32 bits con formato 5.27 en complemento a 2.

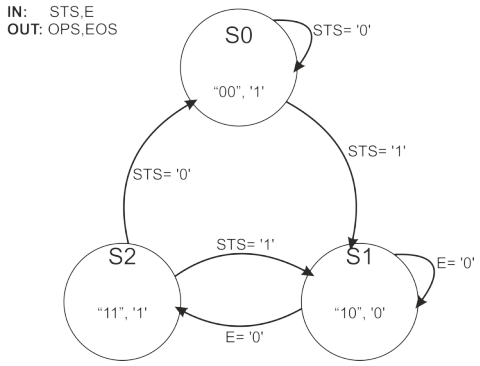

Figura 3.9: Máquina de estados del control de la raíz cuadrada.

3.3.3. Normalización de un cuaternión

Se utiliza la normalización cuando se necesita que la norma del cuaternión sea la unidad; dicha operación fue explicada en el capítulo anterior y lo describe la Ecuación 3.3.3 la cual ayuda para formar el diagrama a bloques de la normalización como se muestra en la Figura 3.10.

$$q_n = \frac{q}{|q|}$$

Como se observa en la Figura 3.10, esta operación calcula el módulo del cuaternión y se realizan 4 divisiones de números reales. Esta operación genera un retraso debido a la división y a la raíz cuadrada calculadas, ya que utilizan el método de aproximaciones sucesivas. La entrada es un cuaternión

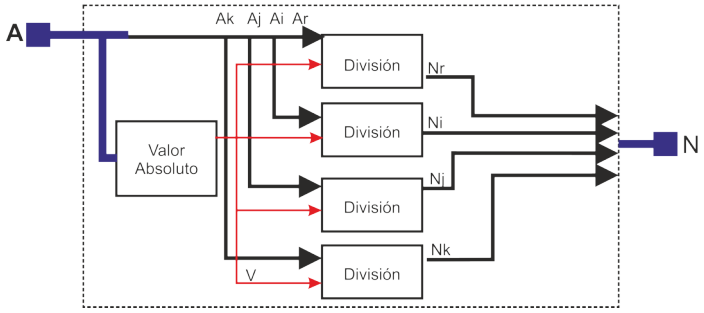

Figura 3.10: Diagrama a bloques de normalización de un cuaternión.

A y de salida N, cada cuaternión consta de cuatro partes de 32 bits en complemento a 2 y con formato 5.27.

3.3.4. Inverso de un cuaternión

En este bloque se utiliza nuevamente el cálculo de la norma y el módulo de potencia, en el cual se utiliza el mismo cuaternión en las dos entradas de la multiplicación, así como el conjugado del dividendo como se mostró con la Ecuación 3.3.4 en la sección 2.2.4. En la Figura 3.11 se muestra el diagrama a bloques del inverso de un cuaternión.

$$q^{-1} = \frac{\bar{q}}{|q|^2}$$

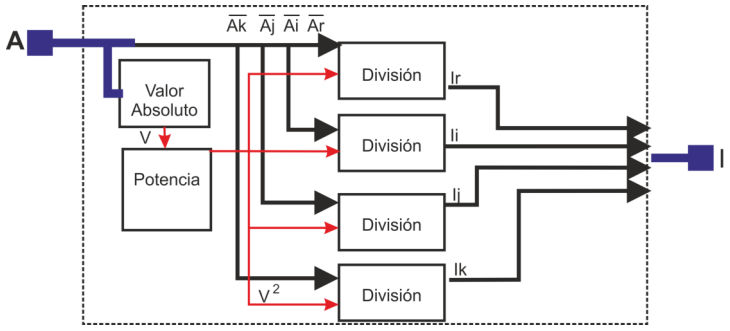

Figura 3.11: Diagrama a bloques de inverso de un cuaternión.

Esta operación necesita de entrada un cuaternión A, cada parte de este

cuaternión es de 32 bits con formato 5.27 en complemento a 2, esta operación da como resultado un cuaternión I el cual esta formado por sus cuatro partes con formato 0.32 en complemento a 2.

3.4. Rotación con cuaterniones

La rotación con cuaterniones es una mejor manera de utilizar la rotación con respecto de los ángulos de Euler, debido a que con los cuaterniones no se presentan indeterminaciones. Esta subsección explicará que módulos se desarrollaron para aplicar la rotación de cuaterniones y poderlas utilizar en rotación de imágenes.

3.4.1. Bloque de rotación

El bloque general de Rotación esta conformado por los bloques que se muestran en la Figura 3.12. El archivo que se realizó en VHDL contiene un bloque llamado Adquisición de Datos, el cual se utiliza tres veces debido a que las imágenes que se inyectarán son imágenes RGB, para adquirir los datos que van entrando en cada eje. Este módulo de adquisición de datos convierte los datos del archivo a números binarios en el formato que se utiliza en el programa, una vez teniéndolos en binario y con el formato correcto, pasa al siguiente bloque de Rotación, el cual procesa el archivo. Teniendo el punto rotado pasa a otro bloque llamado Escritura de Datos, el cual va convirtiendo los datos binarios a decimales y los va escribiendo en un archivo de texto, por lo cual, en el bloque jerárquico se tienen los 3 buses de entrada así como los tres buses de salida que contienen la los datos de la imagen RGB ya rotada.

El bloque interno al jerárquico es el de la Figura 3.13, este módulo necesita tres ángulos en el rango de 0 a 90 grados, los cuales, por cuestiones de visualización y mejor entendimiento del bloque, se introducen como entrada en formato de ángulos de Euler, en los tres ejes, X, Y y Z, sin embargo,

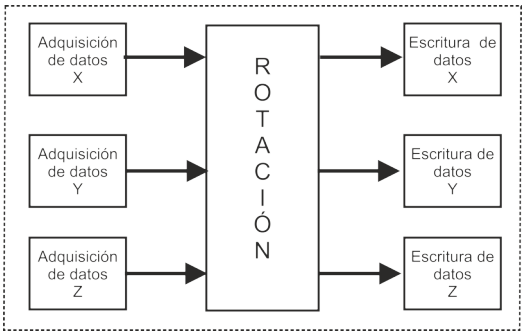

Figura 3.12: Diagrama a bloques del módulo de Rotación Jerárquico.

estos no se utilizan como tal debido a las desventajas arriba mencionadas, por lo cual se hace una conversión para utilizarlos como cuaterniones, usando la Ecuación 2.9. Cada ángulo es de 9 bits, y puede ir incrementando en intervalos de un grado, a este mismo bloque le llegan los datos convertidos en binario y entran al siguiente módulo interno, en el cual se le realiza la rotación.

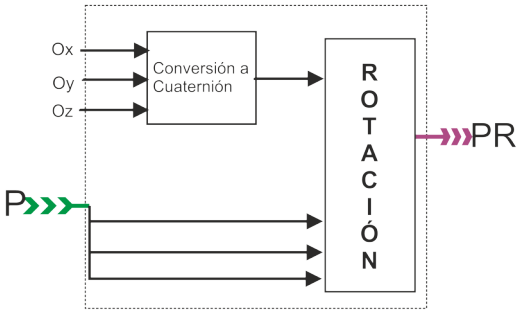

Figura 3.13: Diagrama a bloques del módulo de Rotación.

El bloque rotación mostrado en la Figura 3.15 realiza la operación de rotación de los datos ya convertidos junto con el cuaternión, el cual incluye los ángulos a los cuales se rotará cada eje, y el punto al cual se le aplicará la rotación por medio de la Ecuación 2.14 vista en el capítulo anterior. Teniendo la conversión, el bloque queda diseñado como lo muestra la Figura 3.14, como se muestra, ahora este diagrama a bloques tiene un cuaternión de entrada, que surgió de la conversión de los ángulos, el cual cuenta con sus cuatro

partes de 32 bits y un formato de 5.27 bits en complemento a 2, así como el punto a rotar y como salida obtenemos el punto ya rotado.

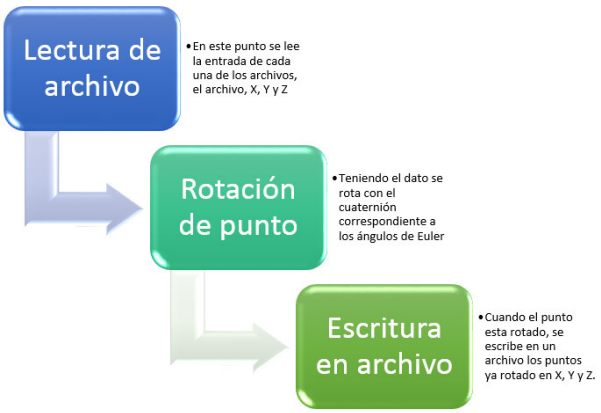

Figura 3.14: Diagrama de Rotación.

Como se puede observar en la Ecuación 2.14, el módulo de rotación necesita el calculo del producto punto y del producto cruz, los cuales fueron diseñados y compactados en el módulo rotación. En el cual tiene como entradas el punto a rotar de la imagen y el cuaternión que indica la rotación se realizará en los tres ejes y se obtiene el punto ya rotado como salida. Por lo cual, el proceso de la rotación se puede resumir con la Figura 3.15

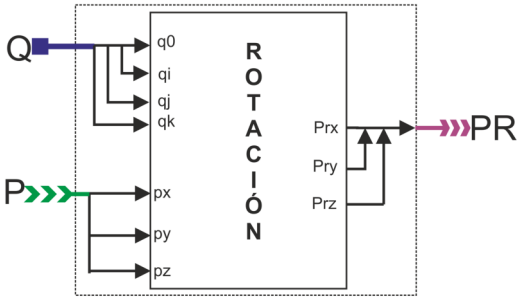

Figura 3.15: Diagrama a bloques del módulo de Rotación con cuaterniones.

Teniendo los archivos de entrada y de salida en decimal, se prosiguió a procesarlos en el programa MATLAB®, únicamente para corroborar que la

rotación se estaban realizando de forma correcta, por lo cual se realizó la rotación por medio de cuaterniones haciendo uso de MATLAB® y viendo sus resultados, los cuales se consideran como los resultados esperados. Por otra parte, los archivos que se escribieron en la ejecución del algoritmo se utilizan ahora en MATLAB®, para agregar una imagen y ver los resultados obtenidos por este método y por VHDL.

Se necesitó manipular el archivo generado por VHDL, haciendo una interpolación usando MATLAB® para que quedarán los índices de la imagen enteros y poder inyectar las imágenes RGB.

Una vez que se tuvo toda la metodología y los programas desarrollados se prosigue a realizar pruebas para dar resultados concretos del trabajo, los cuales se verán en el siguiente capítulo.

Capítulo 4

Pruebas y resultados

En este capítulo se mostrará todo lo realizado en este trabajo, desde las simulaciones realizadas en VHDL, pruebas con las imágenes y los resultados analizados con el error cuadrático medio, así mismo, el análisis del error, retardo y los recursos utilizados en cada uno de los bloques implementados.

4.1. Simulaciones

4.1.1. Bloque aritmético

Las siguientes figuras muestran los resultados de las simulaciones del bloque aritmético, para homogeneizar los formatos y evitar algún desborde, se consideró tener el cuaternión de salida de un tamaño de 64 bits, así cada operación desplegaran el resultado en formato 10.54 en complemento a 2. Además se optó por seleccionar un par de cuaterniones y aplicarles todas las operaciones, usando el orden de la misma comanda mostrada en la Tabla 3.1 e ir desplegando los resultados de un par de operaciones por simulación, cada operación simuló por $1us$ para que se apreciara. Los cuaterniones que se eligieron para hacer las simulaciones fueron los siguientes:

$$q_A = 15 + 2\hat{i} + 8\hat{j} + 3\hat{k}$$

$$q_B = 2 + 7\hat{i} + 10\hat{j} + 14\hat{k}$$

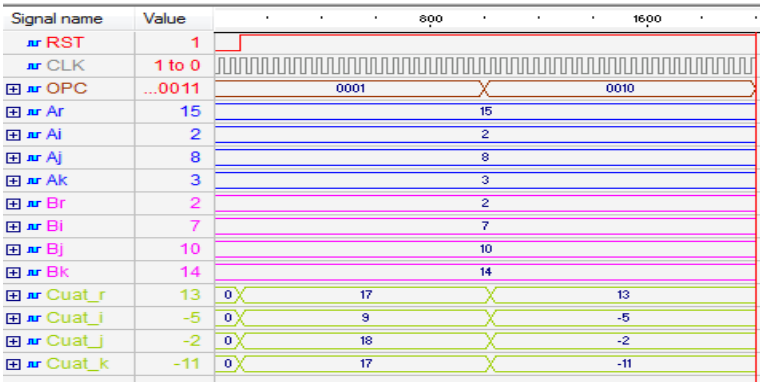

Figura 4.1: Simulación de la suma y resta de cuaterniones.

En cada simulación se mostraran 15 señales de salida las cuales se diferenciaran con colores y con unas abreviaturas, las cuales se mostraran en el mismo orden a lo largo de la sección. La señal Reset se abreviará RST (rojo), la señal de Reloj del FPGA,con la abreviación de sus siglas en inglés CLK (gris), la señal del desplegado de operaciones se abreviará OPC (café), el primer cuaternion A (azules), cuaternion B (magenta) y el cuaternion resultante (verde).

En la Figura 4.1 se observa la salida de 15 señales(usando diferentes colores para visualizar correctamente, la primer señal RST (rojo), la cual por convención se realiza para inicializar las salidas en ceros, esta señal se activa en bajo al inicio del bloque y se optó por activarla 100 ns. La segunda salida es la simulación del reloj del FPGA y se maneja con la abreviación de sus siglas en inglés CLK (gris), esta señal esta en bajo 20 ns y en alto 20 ns, para simular los 50 MHz. La tercer señal es la opción OPC (café), dependiendo el valor que se encuentre en ella es el resultado que se va a desplegar, en este caso se muestra dos cambios, la primera es la combinación para desplegar la suma y a mitad de la simulación se cambia para obtener el resultado de la resta. Posteriormente siguen cuatro señales(azules) estás señales corresponden a los valores del cuaternión A, dependiendo de la parte del cuaternión que puso la letra respectiva: “r”para la parte real y “i, j, y k”para las partes imaginarias. Las siguientes cuatro señales(magenta) corresponden a los

valores del segundo cuaternión B usando la misma denotación del primer cuaternión. Para finalizar se muestran las ultimas cuatro señales(verdes) que corresponden al cuaternión de salida, en esta parte se pueden observar tres cambios, el primero corresponde a la salida cuando esta activo el reset, por eso se observan todas las salidas del cuaternión en ceros, una vez que se desactiva el reset (100 ns) el resultado cambia y da la salida de la operación suma, por lo cuaternión resultante de la suma es $q = 17 + 9\hat{i} + 18\hat{j} + 17\hat{k}$, y por ultimo en esa sección de señales se despliega la salida de la resta que es $q = 13 - 5\hat{i} - 2\hat{j} + 11\hat{k}$.

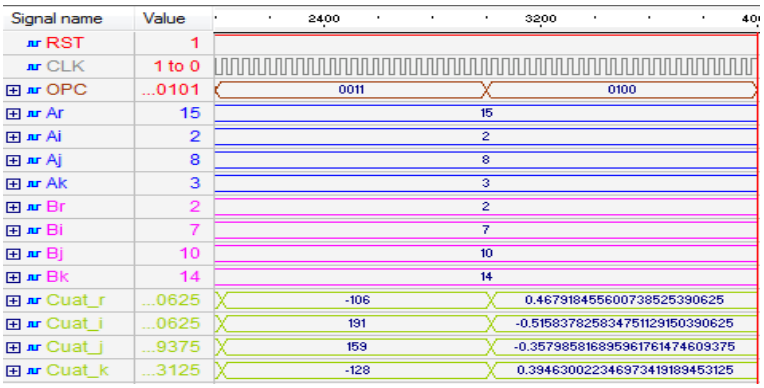

Figura 4.2: Simulación de la multiplicación y división de cuaterniones.

Se debe recordar que la señal de reset, la señal de reloj y los valores de cada parte de los dos cuaterniones no van a cambiar en ninguna simulación. En la Figura 4.2 las primeras dos señales siguen igual que la Figura 4.1 con una pequeña variación debido a que el reset no está activo en ningún tiempo, la tercer señal tiene dos cambios, se muestra primero con la combinación de multiplicación y a la mitad de la simulación se hizo el cambio para la división, las 8 señales que le prosiguen siguen siendo los cuaterniones originales, sin embargo,las ultimas cuatro señales, siempre tendrán un cambio, en este caso los primeros valores de manera vertical corresponden al cuaternión de salida referente a la multiplicación es $q = -106 + 191\hat{i} + 159\hat{j} - 128\hat{k}$, mientras que la segunda columna de estas cuatro señales, corresponden a los resultados de la división el cual forma el

siguiente cuaternión $q = 0.46791 - 0.51583\hat{i} - 0.35798\hat{j} + 0.39463\hat{k}$. Como se puede observar, estos resultados a diferencia de las tres operaciones anteriores ya nos dan números decimales, por lo cual en el texto se hará un truncamiento del cuaternión resultante a 5 cifras, las demás cifras se pueden apreciar en la imagen. También a diferencia de las otras operaciones, es que si se realiza la operación manual de la división, el resultado esperado es $q = 0.47564 - 0.52435\hat{i} - 0.36389\hat{j} + 0.40114\hat{k}$ el cual comparado con el obtenido se tiene un error, sin embargo, eso se analiza y comenta en otra sección.

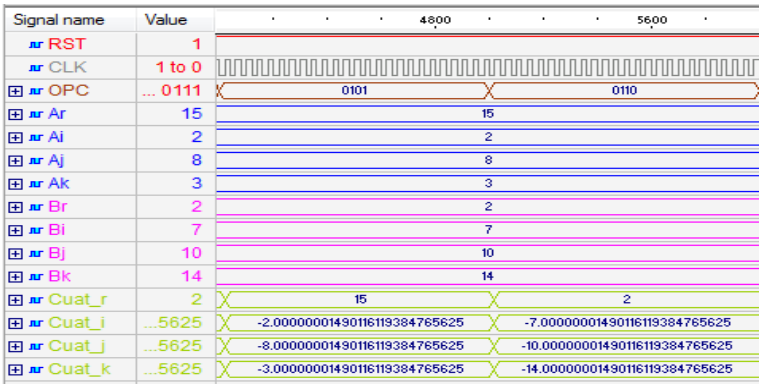

Figura 4.3: Simulación de la operación complemento.

La siguiente simulación será la de la operación complemento Figura 4.3, al igual que en la simulación anterior las dos primeras señales permanecen iguales, sin embargo, la señal de opción primero esta en la combinación para calcular el complemento del primer cuaternión y hace un cambio para calcular el complemento del segundo cuaternión, las señales azules y magenta siguen representando los cuaterniones de prueba, mientras que los primeros resultados del cuaternión resultante muestra el complemento del primer cuaternión $q = 15 - 2\hat{i} - 8\hat{j} - 3\hat{k}$ seguidos del resultado del complemento de el segundo cuaternión $q = 2 - 7\hat{i} - 10\hat{j} - 14\hat{k}$. Se puede observar que varia muy poco el resultado esperado al obtenido, ya que esta operación lo que hace es cambiar de signo solamente a las partes imaginarias, y la parte real sigue con el mismo valor.

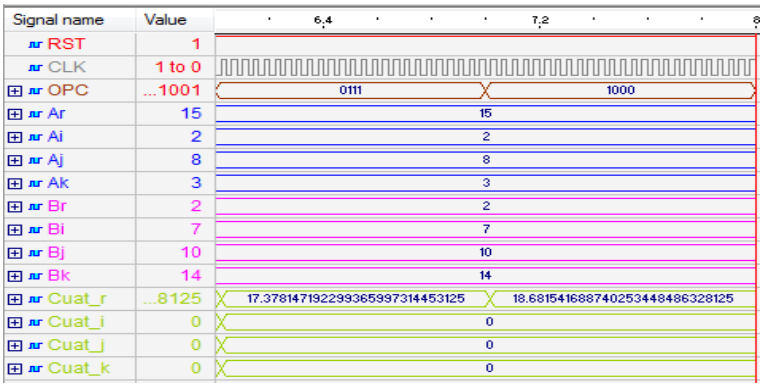

Figura 4.4: Simulación de la operación norma.

En la Figura 4.4 se tiene las señal de reset sin activarse, el reloj en la segunda señal, mientras que la tercera esta en la opción de norma del cuaternión uno y a mitad de la simulación se pasa a calcular la norma del segundo cuaternión, las siguientes señales se tienen los dos cuaterniones de entrada y por ultimo las cuatro señales de salida, primero muestran la norma, cabe recordar que la norma de un cuaternión es un numero escalar y no un cuaternión, es por eso que solo aparece el valor en la parte real del resultado y las partes imaginarias aparecen en ceros. Al hacer manual el calculo de estas dos operaciones obtenemos que la norma del primer cuaternión es 17.37814 mientras que la norma del segundo cuaternión es 18.68154, comparando con los resultados obtenidos son los mismos.

En las simulaciones se pueden apreciar los resultados del bloque aritmético, en este caso solo se presenta un par de cuaterniones por cuestiones de espacio, sin embargo, se hicieron más pruebas para determinar los errores y retardos del bloque.

4.1.2. Bloque rotación

Para mostrar los resultados del bloque de rotación en VHDL, se realizó también la rotación de imágenes por medio de cuaterniones en MATLAB®. Con los indices obtenidos por ambos casos, se procedió a inyectar las imágenes para poder analizarlos. Se realizaron 90 pruebas de rotación a cada imagen,

por lo que en total se realizaron 450.

En la Figura 4.5 se muestran las imágenes que se utilizaron para realizar las pruebas de este trabajo, las cinco imágenes son imágenes RGB de 600×480 píxeles, todas tomadas con la misma cámara.


Imagen 1


Imagen 2


Imagen 3

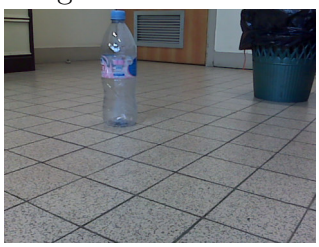
Imagen 4

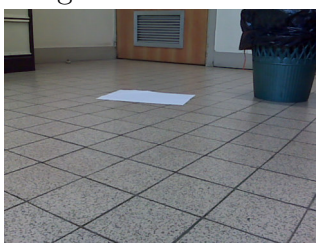
Imagen 5

Figura 4.5: Imágenes utilizadas para efectos de pruebas.

A las imágenes mostradas en la Figura 4.5 se les hicieron varias pruebas rotando cada una de las imágenes de 0 a 90 grados, en cada uno de los ejes. También se realizaron las rotaciones de las imágenes combinando rotaciones en los 3 ejes con diferentes ángulos cada uno.

A continuación se muestran algunos resultados obtenidos en este trabajo de tesis con diferentes rotaciones, se ilustrarán los resultados tanto en MATLAB ® como en VHDL, por lo que las dos imágenes que se mostrarán en cada una de las pruebas, la de la derecha será la obtenida por el algoritmo desarrollado y la de la izquierda será la imagen con los resultados esperados realizados en MATLAB ®.

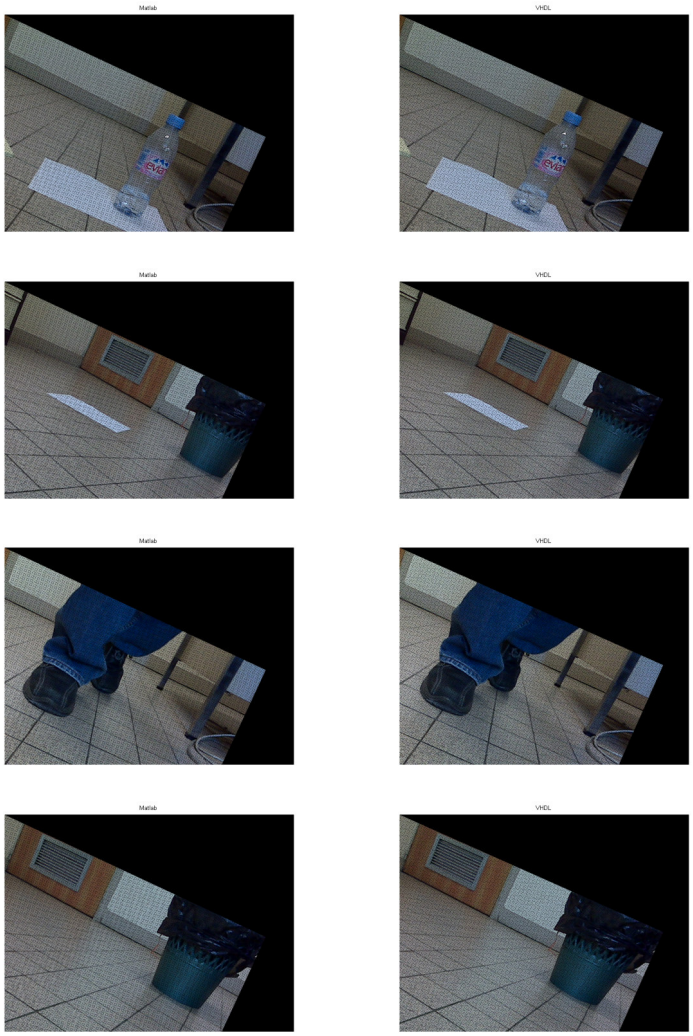

Figura 4.6: Perspectiva de la imagen rotada los ejes Y y Z.

En la Figura 4.6 se muestra las cinco imágenes con una rotación de $\theta_x = 0°, \theta_y = 5°$ y $\theta_z = 25°$, usando de entrada la imagen en cada una de nuestra base de datos Figura 4.5. Estas cinco pruebas muestran en la primer columna los resultados esperados por MATLAB ®, mientras que la segunda columna muestra los resultados obtenidos con el algoritmo propuesto.

Figura 4.7: Perspectiva de la imagen rotada los tres ejes.

En la Figura 4.7 se muestra una de las pruebas realizadas a la segunda imagen de base para las pruebas, a esta imagen se le aplicó una rotación de $\theta_x = 45°, \theta_y = 5°$ y $\theta_z = 5°$. La imagen mostrada en la izquierda es la esperada, mientras que la derecha es la obtenida por el algoritmo propuesto.

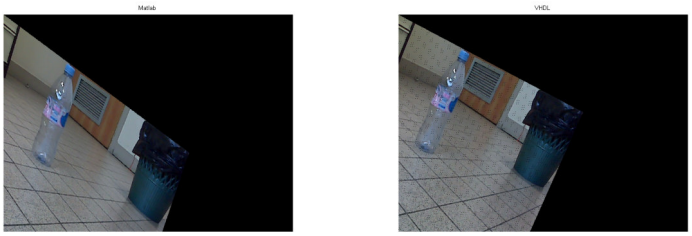

Figura 4.8: Perspectiva de la imagen cuatro con rotación en los ejes tres ejes.

En la Figura 4.8 se muestra una de las pruebas realizadas a la cuarta imagen de la base de datos. A esta imagen se le aplicó una rotación de $\theta_x = 5°, \theta_y = 45°$ y $\theta_z = 25°$. La imagen mostrada en la izquierda es la obtenida por medio de MATLAB ®, mientras que la derecha es la obtenida por VHDL.

En la Figura 4.9 se muestra una de las pruebas realizadas a la primer imagen de las elegidas para realizar las pruebas, a esta imagen se le aplicó una rotación de $\theta_x = 30°$ poniendo en los demás ejes cero grados. La imagen mostrada en la izquierda es la esperada y obtenida por MATLAB ®, mientras

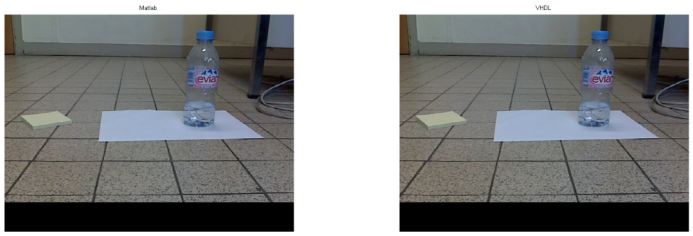

Figura 4.9: Perspectiva de la imagen uno rotada en el eje X.

que la derecha es la obtenida por el algoritmo propuesto.

Figura 4.10: Perspectiva de la imagen dos rotada en el eje Y.

En la Figura 4.10 se muestra una pruebas realizadas a la segunda imagen de las elegidas para realizar las pruebas. A esta imagen se le aplica solo la rotación en el eje Y con $\theta_y = 45°$ poniendo en los demás ejes cero grados. Esta imagen, al igual que las demás pruebas se muestra la imagen esperada y la obtenida.

En la Figura 4.11 se le aplica una rotación a la imagen dos exclusivamente en el eje Z, por lo que $\theta_x = 0°, \theta_y = 0°$ y $\theta_z = 35°$ donde se pueden apreciar los resultados obtenidos.

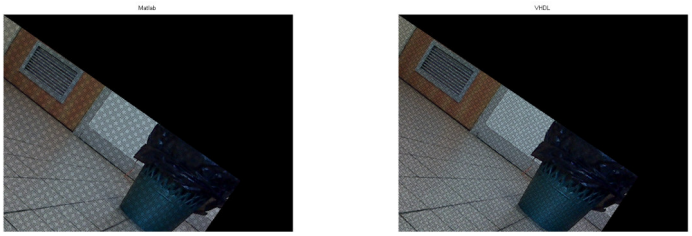

Figura 4.11: Perspectiva de la imagen tres rotada en el eje Z.

4.2. Error y retardo

4.2.1. Bloque aritmético

Con base en las pruebas, la Tabla 4.1 muestra los resultados del error y retardos obtenidos en este trabajo, en el cual cabe mencionar que para la norma, normalización e inverso se utilizó el reloj de la FPGA (50MHz). Se puede observar que la suma, resta, multiplicación y conjugado tienen un retardo de un ciclo de reloj, ya que su procesamiento es casi inmediato, en el caso de la norma, el retardo aumenta, por el uso de las iteraciones sucesivas. La normalización e inverso son un poco más tardados de arrogar el resultado ya que se utilizan dos veces las iteraciones sucesivas, una para la norma o módulo y la otra para realizar la división.

Operación	**Retardo**		**Error**	
	Ciclos de reloj	Tiempo	min	max
Suma	1	20 ns	0	1.4e-8
Resta	1	20 ns	-7.4e-9	7.4e-9
Conjugado	1	20 ns	0	1.4e-8
Multiplicación	1	20 ns	0	1.1e-16
Norma	65	1.3 ms	0	1.4e-8
Normalización	133	2.66 ms	0	1.29e-8
Inverso	133	2.66 ms	0	2.32e-10
División	139	2.78 ms	0	7.4e-9

Tabla 4.1: Tabla de error y retardos en el bloque aritmético.

Tomando en cuenta que el menor número que se puede representar es el 2^{-31}, el error máximo que se puede obtener en la suma es el doble del error mínimo de cada elemento del cuaternión. Para el caso de la resta el error mínimo es que se reste a 0 el error y el máximo error menos un cuaternión 0. En la multiplicación el error se va disminuyendo por el número de multiplicaciones entre ellos, por lo que cada vez se va haciendo más pequeño. Para la norma o módulo se tiene el error máximo al igual que en la suma por el diseño utilizado. La normalización y el inverso es muy parecido el diseño que se implementó, pero en el inverso hay una potencia al cuadrado lo que hace que el error baje aún más que el error de la normalización. Cabe recalcar que el formato de la entrada y salida es 5.27 y 10.54 en complemento a 2 respectivamente.

4.2.2. Bloque rotación

Con base en las pruebas realizadas en esta tesis se obtuvieron los siguientes resultados en cuanto al error que se tiene en este modulo, los valores se calcularon por medio de MATLAB ® con cada una de las imágenes, a las que se les hizo la comparativa con las imágenes que deberían de formarse con el mismo algoritmo pero en dicha plataforma.

Cada imagen y a cada prueba se analizó el error cuadrático medio, así como el número de falsos positivos y negativos, por lo cual se llegaron a los siguientes resultados.

La Figura 4.12 muestra el error cuadrático medio de cada imagen por eje. Las primeras tres cruces mostradas en la gráfica representan el error cuadrático medio de la primer imagen, la primera corresponde al eje X, la segunda al eje Y y la tercera al eje Z; le presiden los tres puntos verdes, que corresponden al error cuadrático medio de la segunda imagen, las tachas color magenta corresponden a la tercer imagen, los triángulos invertidos corresponden a la cuarta imagen, mientras que los cuadrados azules corresponden a la quinta

imagen, todas ellas se muestran en el mismo orden de los ejes que la primer imagen.

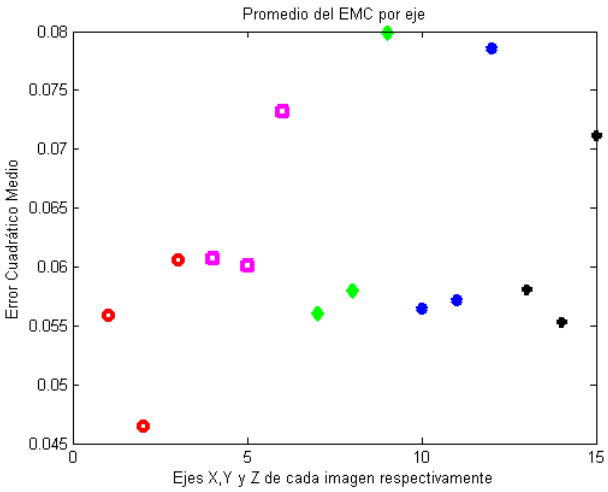

Figura 4.12: Error cuadrático medio.

En la Figura 4.13 se muestra el error cuadrático medio exclusivo de la primer imagen analizada, en esta imagen se muestra los tres ejes, el eje color magenta corresponde al error que existe en el eje X, mientras que el color rojo es el error del eje Y y por último el azul es el error en el eje Z. Se muestra el error cuadrático medio desde 5 hasta 90 grados en incrementos de 5. Se puede observar que los ángulos pares no tienen error, debido a que la conversión de ángulos a cuaterniones se necesita calcular la mitad del ángulo, y la LUT que se utiliza va de grado en grado. Sin embargo cuando es múltiplo de 5, al calcular la mitad es número decimal, por lo cual toma un valor que no es el exacto.

En la Figura 4.14 se muestra el promedio del error cuadrático medio por eje promedio de las cinco imágenes. El orden en el que aparecen es el siguiente, en la figura se simboliza con un círculo rojo, el promedio del error cuadrático medio de las 5 imágenes del eje X, mientras que el promedio del eje Y se representa por un cuadrado color magenta y por último el eje Z con un rombo color azul.

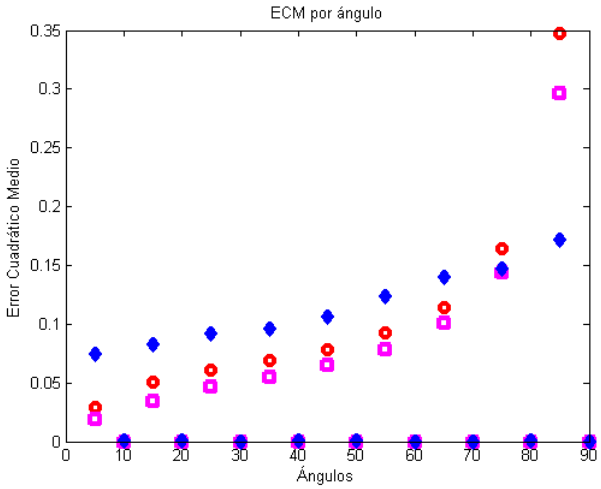

Figura 4.13: Error cuadrático medio por ángulos de la imagen 1.

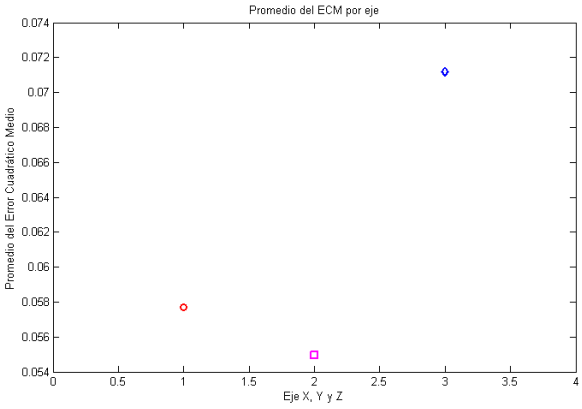

Figura 4.14: Error cuadrático medio promedio por ejes de las cinco imágenes.

En la Figura 4.15 se muestra el promedio del error cuadrático medio de cada uno de los ejes de las cinco imágenes utilizadas en las pruebas, en cada eje se promedió el ángulo de 0 a 90 grados en intervalos de 5 grados. Se representa el eje X, por medio de círculos rojos, el eje Y se representa por cuadrados de color magenta, mientras que el eje Z con rombos azules.

Los falsos negativos que presentan los resultados son bajos, dependiendo

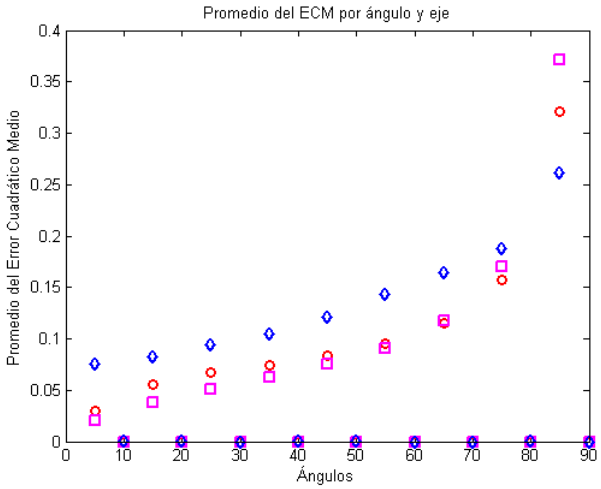

Figura 4.15: Errores cuadráticos medios globales.

del ángulo de rotación del eje en el que se le esta aplicando. Los ejes X y Y con un ángulo menor o igual a 50 grados tiene un porcentaje de falsos negativos de 0 % a 1.77 %, sin embargo, si se aumenta el grado mayor a 50 y menor a 90 grados va de un 0 a 19.32 %, debido al número de píxeles que se pierden en la rotación, porque salen del tamaño que se están analizando. Mientras que el eje Z con un ángulo menor o igual a 50 grados tiene un porcentaje de falsos negativos de 0 % a 24.48 %, sin embargo, si se aumenta el ángulo mayor a 50 y menor a 90 grados va de un 0 a 22.72 % y esto va relacionado con la interpolación. Estos resultados se obtuvieron al aplicar solo un ángulo de rotación de 0 a 90 grados en incrementos de 5. Los porcentajes de cero obtenidos de los falsos negativos fueron en los ángulos múltiplos de 10.

Por otra parte los resultados de falsos positivos de las pruebas de los ejes X y Y se podrían despreciar, debido a que el porcentaje que se maneja va en el rango de 0 % a 0.00039 % independientemente el ángulo de la rotación, sin embargo, para el eje Z el porcentaje ronda de 0 % a 22.72 %.

4.3. Superficie

En este apartado se desplegarán los recursos utilizados en este proyecto por cada bloque realizado, es importante recalcar que se desarrollaron todos los módulos y fueron sintetizados por medio del programa ACTIVE-HDL 9.1 implementándose en la FPGA 4CE115 incluida en la Altera Cyclone IV con el modelo DE2-115 en el programa QUARTUS II 13.0.

4.3.1. Bloque aritmético

Este módulo tiene como entradas dos cuaterniones de 32 bits en formato 5.27 en complemento a 2, un vector de control de 3 bits que indica la opción de la operación y como salida está un cuaternión de 64 bits con formato 10.54 en complemento a 2, el cual proviene de cualquiera de las opciones de las operaciones a desplegar, el bloque desarrollado puede desplegar el resultado de la suma, resta, multiplicación, división, complemento o conjugado, norma o módulo, normalización e inverso de cada cuaternión dependiendo la opción que se deseé calcular, como se muestra en la Tabla 3.1.

Con base en la implementación, las tablas muestran los elementos lógicos, funciones combinacionales, registros y multiplicadores utilizados por cada una de las operaciones que consta el bloque Aritmético. Teniendo en cuenta los recursos usados por cada una de las operaciones, la Tabla 4.2 muestra los recursos utilizados por el bloque aritmético en el cual integra las 12 operaciones que se realizan en el módulo, así como los porcentajes utilizados en la implementación. En la Tabla 4.2 se puede apreciar que el número de elementos lógicos utilizados en el bloque aritmético es solamente del 10.267 %, el cual se considera bajo comparado al total que se tienen disponibles, debido a las operaciones básicas realizadas; los registros utilizados fueron cerca el 2.749 % de los disponibles, dichos registros son utilizados por las máquinas de estados, las sincronizaciones y los elementos de memoria que contiene el bloque, sin embargo, el número de multiplicadores es muy elevado, debido

a la resolución que se planeó manejar de 32 bits, ya que son cuatro partes del cuaternión, que al ser multiplicadas se debe de extenderlos al doble del tamaño para evitar un desbordamiento en el resultado. Cabe recalcar que cada multiplicación de cuaterniones realiza 16 multiplicaciones, además, la división, normalización y la raíz cuadrada llevan consigo multiplicaciones, por estas razones este recurso es el más utilizado.

Recursos	**Usados**	**Totales**	%
Total combinational functions	11445	114480	9.997
Registros	3138	114480	2.741
Elementos lógicos	11754	114480	10.267
Multiplicadores embebidos 9-bits	480	532	90.226

Tabla 4.2: Tabla de recursos totales utilizados por el bloque aritmético.

También se mostrarán los recursos utilizados por cada una de las operaciones en la Tabla 4.3, en la que se optimizan los recursos al hacer el bloque aritmético. Además muestra los resultados de los recursos individuales de cada operación antes de ser compactados en el bloque aritmético, por lo que se puede obtener más información de los elementos lógicos, registros y multiplicadores que cada operación necesitó. Con esta tabla se comprueba que las operaciones suma, resta y complemento no utilizan ningún multiplicador ni registros, estas operaciones solamente utilizan recursos lógicos, mientras que la división es la operación que utiliza más recursos de multiplicadores, le sigue la multiplicación de cuaterniones, inverso, normalización y norma respectivamente.

4.3.2. Bloque rotación

Por otra parte, en la Tabla 4.4 se muestran los recursos utilizados con la implementación realizada para el bloque de rotación, se resumen los elementos lógicos utilizados, funciones combinatorias, registros y multiplicadores por el bloque de rotación, el cual comparado con los recursos utilizados por

Suma	Funciones combinatorias	132
	Registros	0
	Elementos lógicos	132
	Multiplicadores embebidos 9-bits	0
Resta	Funciones combinatorias	132
	Registros	0
	Elementos lógicos	132
	Multiplicadores embebidos 9-bits	0
Multiplicación	Funciones combinatorias	2240
	Registros	0
	Elementos lógicos	2240
	Multiplicadores embebidos 9-bits	128
Complemento	Funciones combinatorias	93
	Registros	0
	Elementos lógicos	93
	Multiplicadores embebidos 9-bits	0
Norma	Funciones combinatorias	821
	Registros	97
	Elementos lógicos	822
	Multiplicadores embebidos 9-bits	40
Normalización	Funciones combinatorias	1755
	Registros	386
	Elementos lógicos	1760
	Multiplicadores embebidos 9-bits	72
Inverso	Funciones combinatorias	1811
	Registros	321
	Elementos lógicos	1811
	Multiplicadores embebidos 9-bits	80
División	Funciones combinatorias	4050
	Registros	1985
	Elementos lógicos	4317
	Multiplicadores embebidos 9-bits	208

Tabla 4.3: Tabla de recursos utilizados por cada operación.

el bloque aritmético se puede notar que es muy bajo el consumo de superficie que la rotación ofrece, debido a que es menor la cantidad de información que procesa.

Resurcursos	**Usados**	**Totales**	%
Funciones combinatorias	8398	114480	7.335
Registros	272	114480	0.237
Elementos lógicos	8434	114480	7.367
Multiplicadores embebidos 9-bits	288	532	54.135

Tabla 4.4: Tabla de recursos totales utilizados por el bloque aritmético.

Con esta implementación de la rotación se utiliza un poco más de la mitad de los multiplicadores totales que nos ofrece la FPGA, cabe mencionar, como en el apartado anterior, que son los recursos más utilizados debido al tamaño de cada parte del cuaternión 32 bits y para poder realizar la rotación, como se vio en el Capítulo 2, se necesitan sumas y restas así como los productos punto, cruz y normales de los mismos cuaterniones a rotar. Se utilizan menos del 1 % de los registros y el 7.367 % de los elementos lógicos totales de esta FPGA.

4.4. Comparativa

Lo que se realizó en este trabajo no se puede comparar directamente con algún otro trabajo, debido a que no se encuentra información en el cual se haga lo mismo o parecido a este, ya que la mayoría se enfoca solamente a la matemática sin implementarlo. Los trabajos que llegan a implementar el cuaternión lo utilizan para otras aplicaciones sin analizar el tiempos ni superficies en que lo usan.

Capítulo 5

Conclusiones y Perspectivas

Este trabajo presenta una solución para el álgebra del cuaternión para ser aplicado en futuros trabajos en el área de robótica, así como la rotación de imágenes RGB por medio de una FPGA. Estos dos módulos fundamentales se llevaron a cabo usando una Altera Cyclone IV.

El modulo de la aritmética del cuaternión cuenta con las 8 operaciones básicas utilizando como entrada dos cuaterniones, dependiendo la operación, se realizan en forma paralela las 8 operaciones y por medio de un mutiplexor el usuario decide que resultado es el que se despliega. La rapidez que ofrece la FPGA es de gran ayuda para realizar todas las operaciones y el usuario pueda tener los resultados automáticamente para la vista del humano, sin embargo, como se vio en el capítulo anterior, cuenta con retardos, sobre todo las operaciones donde se utilizan métodos iterativos para obtener el resultado.

Los resultados muestran que la implementación funciona correctamente y tiene un margen de error relativamente bajo debido a la precisión de entradas y salidas, las cuales son de 32 bits punto fijo con formato 5.27, excepto los bloques del inverso con formato 0.32 y el de la multiplicación con 10.54. Ciertos bloques se reutilizan para el cálculo de otras operaciones ganando tiempo en diseño pero causando un mayor retardo, sin embargo, como se mencionó anteriormente, el cálculo se puede considerar que es rápido.

Por otra parte, el bloque de rotación da buenos resultados, como se puede ver cuando se analizó el error cuadrático medio en los ejes X y Y, sin embargo en el eje Z se presenta mayor error, esto debido a la interpolación que se necesita realizar para que los índices de la matriz queden en números enteros y no en decimales. Esta es la razón por la cual las imágenes mostradas tienen más píxeles negros.

Cabe mencionar que entre mayor es la inclinación se pierden más píxeles y en la mayoría de las ocasiones incrementa el error, debido a que baja el número de píxeles a comparar.

Como trabajos a futuro se podría bajar la resolución de los cuaterniones para mejorar los recursos utilizados en la aritmética del cuaternión, sin embargo, incrementaría el error eso respecto a la arquitectura del cuaternión.

Otra mejora que se le podría hacer a la aritmética del cuaternión, seria agregar las operaciones de producto cruz y el producto punto, las conversiones de cuaternión a ángulos de Euler y de cuaternión a la matriz de rotación debido a que estos módulos ya fueron diseñados e implementados para la rotación.

Como trabajo futuro de la rotación de imágenes, se podría aumentar la resolución de los ángulos y que no fuera en incrementos de 1 sino en incrementos de 0.5, o en su defecto de 0.1, así se espera que se tengan mejores resultados, también podría experimentarse con otro formato de imágenes u otros tamaños; así mismo este módulo, debido a los pocos recursos que utiliza, se podría implementar en un FPGA más pequeña.

APÉNDICE A

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Bloque_Aritmetica is
port(
        RST     :       in      std_logic;
        CLK     :       in      std_logic;
        OPC             :       in      std_logic_vector(3 downto 0);
        --      0001 Suma                       a+b
        --      0010 Resta                      a-b
        --      0011 Multiplicacion     a*b
        --      0100 Division           a/b
        --      0101 Conjugado          a
        --      0110 Conjugado          b
        --      0111 Valor Absoluto a
        --      1000 Valor Absoluto b
        --      1001 Normalizacion      a
        --      1010 Normalizacion      b
        --      1011 Inverso            a
        --      1100 Inverso            b

        --Entrada Cuaternion A
        Ar: in std_logic_vector(31 downto 0);
        Ai: in std_logic_vector(31 downto 0);
        Aj: in std_logic_vector(31 downto 0);
        Ak: in std_logic_vector(31 downto 0);
        --Entrada Cuaternion B
        Br: in std_logic_vector(31 downto 0);
        Bi: in std_logic_vector(31 downto 0);
        Bj: in std_logic_vector(31 downto 0);
        Bk: in std_logic_vector(31 downto 0);
        --Cuaternion resultante
        Cuat_r: out std_logic_vector(63 downto 0);
        Cuat_i: out std_logic_vector(63 downto 0);
        Cuat_j: out std_logic_vector(63 downto 0);
        Cuat_k: out std_logic_vector(63 downto 0)

);
end Bloque_Aritmetica;

architecture Jerarquico_Total of Bloque_Aritmetica is

component Suma_Cuaternion
        port(
        --Entrada Cuaternion A
        Ar: in std_logic_vector(31 downto 0);
        Ai: in std_logic_vector(31 downto 0);
        Aj: in std_logic_vector(31 downto 0);
        Ak: in std_logic_vector(31 downto 0);
        --Entrada Cuaternion B
```

```
        Br: in std_logic_vector(31 downto 0);
        Bi: in std_logic_vector(31 downto 0);
        Bj: in std_logic_vector(31 downto 0);
        Bk: in std_logic_vector(31 downto 0);
        --Cuaternion resultante
        Sr: out std_logic_vector(37 downto 0);
        Si: out std_logic_vector(37 downto 0);
        Sj: out std_logic_vector(37 downto 0);
        Sk: out std_logic_vector(37 downto 0)
        );
end component Suma_Cuaternion;
component  Resta_Cuaterniones
        port(
        --Entrada Cuaternion A
        Ar: in std_logic_vector(31 downto 0);
        Ai: in std_logic_vector(31 downto 0);
        Aj: in std_logic_vector(31 downto 0);
        Ak: in std_logic_vector(31 downto 0);
        --Entrada Cuaternion B
        Br: in std_logic_vector(31 downto 0);
        Bi: in std_logic_vector(31 downto 0);
        Bj: in std_logic_vector(31 downto 0);
        Bk: in std_logic_vector(31 downto 0);
        --Cuaternion resultante
        Rr: out std_logic_vector(37 downto 0);
        Ri: out std_logic_vector(37 downto 0);
        Rj: out std_logic_vector(37 downto 0);
        Rk: out std_logic_vector(37 downto 0)
        );
end component Resta_Cuaterniones;
component Multiplicacion_Cuaterniones
        port(
        --Entrada Cuaternion A
        Ar: in std_logic_vector(31 downto 0);
        Ai: in std_logic_vector(31 downto 0);
        Aj: in std_logic_vector(31 downto 0);
        Ak: in std_logic_vector(31 downto 0);
        --Entrada Cuaternion B
        Br: in std_logic_vector(31 downto 0);
        Bi: in std_logic_vector(31 downto 0);
        Bj: in std_logic_vector(31 downto 0);
        Bk: in std_logic_vector(31 downto 0);
        --Cuaternion resultante
        Mr: out std_logic_vector(63 downto 0);
        Mi: out std_logic_vector(63 downto 0);
        Mj: out std_logic_vector(63 downto 0);
        Mk: out std_logic_vector(63 downto 0)
        );
end component Multiplicacion_Cuaterniones;

component Cuaternion_Conjugado is
        port(
        --Entrada Cuaternion A
        Ar: in std_logic_vector(31 downto 0);
        Ai: in std_logic_vector(31 downto 0);
        Aj: in std_logic_vector(31 downto 0);
        Ak: in std_logic_vector(31 downto 0);

        --Cuaternion Conjugado
        Cr: out std_logic_vector(31 downto 0);
        Ci: out std_logic_vector(31 downto 0);
        Cj: out std_logic_vector(31 downto 0);
        Ck: out std_logic_vector(31 downto 0)
        );
end component Cuaternion_Conjugado;
component Valor_Absoluto_Cuaternion
        port(
```

```
        RST     :       in  std_logic;
        CLK     :       in  std_logic;
        --STS_SAR :     in  std_logic;
        Ar              :       in      std_logic_vector(31 downto 0);
        Ai              :       in      std_logic_vector(31 downto 0);
        Aj              :       in      std_logic_vector(31 downto 0);
        Ak              :       in      std_logic_vector(31 downto 0);
        EOS_SAR :       out std_logic;
        q_abs   :       out     std_logic_vector(31 downto 0)
        );
end component Valor_Absoluto_Cuaternion;

component Calculo_Inverso
port(
        RST     :       in      std_logic;
        CLK     :       in      std_logic;
        Evabs   :       in      std_logic;
        q_abs   :       in      std_logic_vector(31 downto 0);
        Cr              :       in      std_logic_vector(31 downto 0);
        Ci              :       in      std_logic_vector(31 downto 0);
        Cj              :       in      std_logic_vector(31 downto 0);
        Ck              :       in      std_logic_vector(31 downto 0);
        EOS_Ir  :       out     std_logic;
        EOS_Ii  :       out     std_logic;
        EOS_Ij  :       out     std_logic;
        EOS_Ik  :       out     std_logic;
        --Cuaternion resultante
        I_r: out std_logic_vector(31 downto 0);
        I_i: out std_logic_vector(31 downto 0);
        I_j: out std_logic_vector(31 downto 0);
        I_k: out std_logic_vector(31 downto 0)
        );
end component Calculo_Inverso;
component Calculo_Division is
        port(
        RST     : in  std_logic;
        CLK     : in  std_logic;
        --Entrada Cuaternion A
        Ar: in std_logic_vector(31 downto 0);
        Ai: in std_logic_vector(31 downto 0);
        Aj: in std_logic_vector(31 downto 0);
        Ak: in std_logic_vector(31 downto 0);
        --Entrada Cuaternion B
        Br: in std_logic_vector(31 downto 0);
        Bi: in std_logic_vector(31 downto 0);
        Bj: in std_logic_vector(31 downto 0);
        Bk: in std_logic_vector(31 downto 0);
        --Cuaternion resultante
        Dr: out std_logic_vector(31 downto 0);
        Di: out std_logic_vector(31 downto 0);
        Dj: out std_logic_vector(31 downto 0);
        Dk: out std_logic_vector(31 downto 0)
        );
end component Calculo_Division;
component Calculo_Normalizacion
        port(
        RST     : in  std_logic;
        CLK     : in  std_logic;
        STS_SAR : in  std_logic;
        Vabs    : in  std_logic_vector(31 downto 0);
        Ar              : in  std_logic_vector(31 downto 0);
        Ai              : in  std_logic_vector(31 downto 0);
        Aj              : in  std_logic_vector(31 downto 0);
        Ak              : in  std_logic_vector(31 downto 0);
        EOS_SAR_r : out std_logic;
        EOS_SAR_i : out std_logic;
        EOS_SAR_j : out std_logic;
```

```
        EOS_SAR_k : out std_logic;
        N_r     :       out     std_logic_vector(31 downto 0);
        N_i     :       out     std_logic_vector(31 downto 0);
        N_j     :       out     std_logic_vector(31 downto 0);
        N_k     :       out     std_logic_vector(31 downto 0)
        );
end component Calculo_Normalizacion;
component Resultados is
        port(
        RST     : in std_logic;
        CLK     : in std_logic;
        OPC             : in std_logic_vector(3 downto 0);
        Sr, Si, Sj, Sk  : in std_logic_vector(37 downto 0);
        Rr, Ri, Rj, Rk  : in std_logic_vector(37 downto 0);
        Mr, Mi, Mj, Mk  : in std_logic_vector(63 downto 0);
        Cr1,Ci1,Cj1,Ck1 : in std_logic_vector(31 downto 0);
        Cr2,Ci2,Cj2,Ck2 : in std_logic_vector(31 downto 0);
        Vr1                             : in std_logic_vector(31 downto 0);
        Vr2                             : in std_logic_vector(31 downto 0);
        Nr1,Ni1,Nj1,Nk1 : in std_logic_vector(31 downto 0);
        Nr2,Ni2,Nj2,Nk2 : in std_logic_vector(31 downto 0);
        Ir1,Ii1,Ij1,Ik1 : in std_logic_vector(31 downto 0);
        Ir2,Ii2,Ij2,Ik2 : in std_logic_vector(31 downto 0);
        Dr, Di, Dj, Dk  : in std_logic_vector(31 downto 0);
        Cuat_r,Cuat_i: out std_logic_vector(63 downto 0)--Resultados
        Cuat_j,Cuat_k: out std_logic_vector(63 downto 0)--Resultados
        );
end component Resultados;
signal Sr, Si, Sj, Sk: std_logic_vector(37 downto 0);
signal Rr, Ri, Rj, Rk: std_logic_vector(37 downto 0);
signal Mr, Mi, Mj, Mk: std_logic_vector(63 downto 0);
signal Cr1, Ci1, Cj1, Ck1: std_logic_vector(31 downto 0);
signal Cr2, Ci2, Cj2, Ck2: std_logic_vector(31 downto 0);
Signal Vr1      : std_logic_vector(31 downto 0);
Signal Vr2      : std_logic_vector(31 downto 0);
signal Nr1, Ni1, Nj1, Nk1: std_logic_vector(31 downto 0);
signal Nr2, Ni2, Nj2, Nk2: std_logic_vector(31 downto 0);
signal Ir1, Ii1, Ij1, Ik1: std_logic_vector(31 downto 0);
signal Ir2, Ii2, Ij2, Ik2: std_logic_vector(31 downto 0);
signal Dr, Di, Dj, Dk: std_logic_vector(31 downto 0);
signal EIr1,EIi1,EIj1,EIk1: std_logic;
signal EIr2,EIi2,EIj2,EIk2: std_logic;
signal Evabs1,Evabs2,Enr1,Eni1,Enj1: std_logic;
signal Enk1,Enr2,Eni2,Enj2,Enk2: std_logic;

begin

        B01: Suma_Cuaternion port map(Ar,Ai,Aj,Ak,
        Br,Bi,Bj,Bk,Sr,Si,Sj,Sk);
        B02: Resta_Cuaterniones port map(Ar,Ai,Aj,Ak,
        Br,Bi,Bj,Bk,Rr,Ri,Rj,Rk);
        B03: Multiplicacion_Cuaterniones port map(Ar,Ai,Aj,Ak,
        Br,Bi,Bj,Bk,Mr,Mi,Mj,Mk);
        B04: Cuaternion_Conjugado port map(Ar,Ai,Aj,Ak,
        Cr1,Ci1,Cj1,Ck1);
        B05: Cuaternion_Conjugado port map(Br,Bi,Bj,Bk,
        Cr2,Ci2,Cj2,Ck2);
        B06: Valor_Absoluto_Cuaternion  port map(RST,CLK,Ar,Ai,
        Aj,Ak,Evabs1,Vr1);
        B07: Valor_Absoluto_Cuaternion  port map(RST,CLK,Br,Bi,
        Bj,Bk,Evabs2,Vr2);
        B08: Calculo_Inverso port map(RST,CLK,Evabs1,
        Vr1,Ar,Ai,Aj,Ak,EIr1,EIi1,EIj1,EIk1,Ir1,Ii1,Ij1,Ik1);
        B09: Calculo_Inverso port map(RST,CLK,Evabs2,
        Vr2,Br,Bi,Bj,Bk,EIr2,EIi2,EIj2,EIk2,Ir2,Ii2,Ij2,Ik2);
        B10: Calculo_normalizacion port map(RST,CLK,Evabs1,
        Vr1,Ar,Ai,Aj,Ak,Enr1,Eni1,Enj1,Enk1,Nr1,Ni1,Nj1,Nk1);
```

```
B11: Calculo_normalizacion port map(RST,CLK,Evabs2,
Vr2,Br,Bi,Bj,Bk,Enr2,Eni2,Enj2,Enk2,Nr2,Ni2,Nj2,Nk2);
B12: Calculo_Division port map(RST,CLK,Ar,Ai,Aj,
Ak,Ir2,Ii2,Ij2,Ik2,Dr,Di,Dj,Dk);
B13: Resultados port map(RST,CLK,OPC,Sr,Si,
Sj,Sk,Rr,Ri,Rj,Rk,Mr,Mi,Mj,Mk,Cr1,Ci1,Cj1,Ck1,
Cr2,Ci2,Cj2,Ck2,Vr1,Vr2,Nr1,Ni1,Nj1,Nk1,Nr2,Ni2,
Nj2,Nk2,Ir1,Ii1,Ij1,Ik1,Ir2,Ii2,Ij2,Ik2,Dr,Di,Dj,Dk,
Cuat_r, Cuat_i,Cuat_j,Cuat_k);
end jerarquico_total;
```

APÉNDICE B

```
library IEEE;
library work;
use std.textio.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_textio.all;
use IEEE.std_logic_unsigned.all;
use work.libreriaRotacionAngulos.all; --Libreria de programas


entity Rotacion_M_libreria is
        generic(
    aentX    : string  := "imagen_in_X.txt";--Datos a leer de coordenadax X
    atxX  : string  := "imagenCL_out_X.txt";--Datos escritos coordenadas X
        aentY    : string  := "imagen_in_Y.txt";--Datos a leer de coordenadax Y
    atxY  : string  := "imagenCL_out_Y.txt";--Datos escritos coordenadas Y
        aentZ    : string  := "imagen_in_Z.txt";--Datos a leer de coordenadax Z
    atxZ  : string  := "imagenCL_out_Z.txt";--Datos escritos coordenadas Z
    nbp : integer := 32;
    nbb : integer := 32;
        m       :       integer :=32;
        n       :       integer :=15;
        k       :       integer :=30720
        );
        port(
        RST     :       in      std_logic;
        CLK     :       in      std_logic;
        --Angulos de Euler
        angX    :       in      std_logic_vector(8 downto 0);-- Valor del angulo de euler en X
        angY    :       in      std_logic_vector(8 downto 0);-- Valor del angulo de euler en Y
        angZ    :       in      std_logic_vector(8 downto 0)-- Valor del angulo de euler en Z
        );

end Rotacion_M_libreria;

architecture Jerarquico of Rotacion_M_libreria is
component SimCamaraAST
  generic(
    arch: string  := "imagen_in.txt";
    nbp : integer := 32;
    nbb : integer := 32
  );
  port(
    clk   : in  std_logic;
    rst   : in  std_logic;
    ready : in  std_logic;
    valid : out std_logic;
    data  : out std_logic_vector(nbb-1 downto 0);
    sop   : out std_logic;
```

```
    eop   : out std_logic
  );
end component SimCamaraAST;

component Rotacion_Completa
        port(
        angX:   in      std_logic_vector(8 downto 0);
        angY:   in      std_logic_vector(8 downto 0);-- Angulos
        angZ:   in      std_logic_vector(8 downto 0);
        Px      :       in      std_logic_vector(31 downto 0);--Coordenadas de puntos
        Py      :       in      std_logic_vector(31 downto 0);
        Pz      :       in      std_logic_vector(31 downto 0);
        PRx     :       out     std_logic_vector(63 downto 0);-- Punto rotado
        PRy     :       out     std_logic_vector(63 downto 0);
        PRz     :       out     std_logic_vector(63 downto 0)
        );
end component Rotacion_Completa;

component SimDespAST
  generic(
    arch  : string  := "imagen_out.txt";
    nbp   : integer := 64;
    nbb   : integer := 64
  );
  port(
    clk   : in  std_logic;
    rst   : in  std_logic;
    ready : out std_logic;
    valid : in  std_logic;
    data  : in  std_logic_vector(nbb-1 downto 0);
    sop   : in  std_logic;
    eop   : in  std_logic
  );
end component SimDespAST;
signal ready_cam_X, valid_cam_X: std_logic;
signal sop_cam_X, eop_cam_X: std_logic;
signal ready_cam_Y, valid_cam_Y: std_logic;
signal sop_cam_Y, eop_cam_Y: std_logic;
signal ready_cam_Z, valid_cam_Z: std_logic;
signal sop_cam_Z, eop_cam_Z: std_logic;
signal DEpx,DEpy,DEpz: std_logic_vector(31 downto 0);
signal data_entrada_X,data_entrada_Y: std_logic_vector(31 downto 0);
signal data_entrada_Z: std_logic_vector(31 downto 0);
signal data_salida_X2,data_salida_Y2: std_logic_vector(63 downto 0);
signal data_salida_Z2: std_logic_vector(63 downto 0);
signal data_salida_X,data_salida_Y: std_logic_vector(31 downto 0);
signal,data_salida_Z: std_logic_vector(31 downto 0);

begin
        B0: SimCamaraAST generic map(aentX, 32, 32) port map(CLK, RST,
        ready_cam_X, valid_cam_X, data_entrada_X, sop_cam_X, eop_cam_X);
        B1: SimCamaraAST generic map(aentY, 32, 32) port map(CLK, RST,
        ready_cam_Y, valid_cam_Y, data_entrada_Y, sop_cam_Y, eop_cam_Y);
        B2: SimCamaraAST generic map(aentZ, 32, 32) port map(CLK, RST,
        ready_cam_Z, valid_cam_Z, data_entrada_Z, sop_cam_Z, eop_cam_Z);
        B3: Rotacion_Completa port map(AngX,AngY,AngZ,data_entrada_X,
        data_entrada_Y, data_entrada_Z,data_salida_X2,  data_salida_Y2,
        data_salida_Z2);

        data_salida_X<=data_salida_X2(58 downto 27);
        data_salida_Y<=data_salida_Y2(58 downto 27);
        data_salida_Z<=data_salida_Z2(58 downto 27);
        B4: SimDespAsT  generic map(atxX, 32, 32) port map(CLK, RST,
        ready_cam_X, valid_cam_X, data_salida_X, sop_cam_X, eop_cam_X);
        B5: SimDespAsT  generic map(atxY, 32, 32) port map(CLK, RST,
        ready_cam_Y, valid_cam_Y, data_salida_Y, sop_cam_Y, eop_cam_Y);
        B6: SimDespAsT  generic map(atxZ, 32, 32) port map(CLK, RST,
```

```
        ready_cam_Z, valid_cam_Z, data_salida_Z, sop_cam_Z, eop_cam_Z);

end Jerarquico;
```

APÉNDICE C

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

package libreriaRotacionAngulos is

        component Sum_Res
        port(
        H: in std_logic; --Opcion de suma o resta
        --Entradas
        A: in std_logic_vector(63 downto 0);--Entradas de la operacion
        B: in std_logic_vector(63 downto 0);
        --Salida
        S: out std_logic_vector(63 downto 0)  --Resultado de la operacion
        );
        end component Sum_Res;

        component  Multi
        port(
        --Entradas
        A: in std_logic_vector(31 downto 0);--Entradas de la multiplicacion
        B: in std_logic_vector(31 downto 0);
        --Salida
        M: out std_logic_vector(63 downto 0)--Resultado multiplicacion
        );
        end component Multi;

        component LUT_Coseno
        port(
        X : in std_logic_vector(8 downto 0); --Valor del angulo
        Y : out std_logic_vector(31 downto 0)-- Coseno de ese angulo
        );
        end component LUT_Coseno;

        component LUT_Seno
        port(
        X : in std_logic_vector(8 downto 0);--Valor del angulo
        Y : out std_logic_vector(31 downto 0)-- Seno de ese angulo
         );
        end component LUT_Seno;

        component FIFO
        generic(
        m       :       integer :=32;
        n       :       integer :=9;
        k       :       integer :=300
        );
        port(
        RST     :       in      std_logic;
        CLK     :       in      std_logic;
```

```
OPR     :       in      std_logic_vector(1 downto 0);
DE      :       in      std_logic_vector(m-1 downto 0);
DS      :       out     std_logic_vector(m-1 downto 0);
E       :       out     std_logic;
F       :       out     std_logic
);
end component FIFO;

component ProductoCruz
port(
V1x: in std_logic_vector(31 downto 0);
V1y: in std_logic_vector(31 downto 0);--Entradas primer vector
V1z: in std_logic_vector(31 downto 0);

V2x: in std_logic_vector(31 downto 0);
V2y: in std_logic_vector(31 downto 0);--Entradas segundo vector
V2z: in std_logic_vector(31 downto 0);

i: out std_logic_vector(63 downto 0);
j: out std_logic_vector(63 downto 0);-- Vector resultante del producto cruz
k: out std_logic_vector(63 downto 0)
);
end component ProductoCruz;

component Multiplicacion_Cuaterniones
port(
--Entrada Cuaternion A
Ar: in std_logic_vector(31 downto 0);
Ai: in std_logic_vector(31 downto 0);--Primer cuaternion de entrada
Aj: in std_logic_vector(31 downto 0);
Ak: in std_logic_vector(31 downto 0);
--Entrada Cuaternion B
Br: in std_logic_vector(31 downto 0);
Bi: in std_logic_vector(31 downto 0);--Segundo cuaternion de entrada
Bj: in std_logic_vector(31 downto 0);
Bk: in std_logic_vector(31 downto 0);
--Cuaternion resultante
Mr: out std_logic_vector(63 downto 0);
Mi: out std_logic_vector(63 downto 0);--Cuaternion resultante
Mj: out std_logic_vector(63 downto 0);
Mk: out std_logic_vector(63 downto 0)
);
end component Multiplicacion_Cuaterniones;

component ProductoPunto
port(
V1x: in std_logic_vector(31 downto 0);
V1y: in std_logic_vector(31 downto 0);--Primer vector de entrada
V1z: in std_logic_vector(31 downto 0);

V2x: in std_logic_vector(31 downto 0);
V2y: in std_logic_vector(31 downto 0);--Segundo vector de enttrada
V2z: in std_logic_vector(31 downto 0);

VP: out std_logic_vector(63 downto 0)--Escalar resultante
);
end component ProductoPunto;

component Angulos_a_Cuaternion
port(
angX    :       in      std_logic_vector(8 downto 0);
angY    :       in      std_logic_vector(8 downto 0);--Angulos de Euler
angZ    :       in      std_logic_vector(8 downto 0);
Q_w     :       out     std_logic_vector(63 downto 0);
Q_i     :       out     std_logic_vector(63 downto 0);
Q_j     :       out     std_logic_vector(63 downto 0);--Cuaternion resultante
Q_k     :       out     std_logic_vector(63 downto 0)
```

```
);
end component Angulos_a_Cuaternion;

component RotacionQ
port(
--Cuaternion Q
q0      :       in      std_logic_vector(31 downto 0);
qx      :       in      std_logic_vector(31 downto 0);
qy      :       in      std_logic_vector(31 downto 0);--Cuaternion del los angulos de Euler
qz      :       in      std_logic_vector(31 downto 0);
--Vector P
p0      :       in      std_logic_vector(31 downto 0);
px      :       in      std_logic_vector(31 downto 0);
py      :       in      std_logic_vector(31 downto 0);--Coordenadas a rotar a rotar
pz      :       in      std_logic_vector(31 downto 0);
--Rotacion
PRx     :       out     std_logic_vector(63 downto 0);
PRy     :       out     std_logic_vector(63 downto 0);--Rotacion de las coordenadas
PRz     :       out     std_logic_vector(63 downto 0)
);
end component RotacionQ;

component  Modulo_Rotacion
port(
--Angulos de Euler
angX    :       in      std_logic_vector(8 downto 0);
angY    :       in      std_logic_vector(8 downto 0);-- Angulo de Euler
angZ    :       in      std_logic_vector(8 downto 0);
--Vector P
p0      :       in      std_logic_vector(31 downto 0);
px      :       in      std_logic_vector(31 downto 0);
py      :       in      std_logic_vector(31 downto 0);
pz      :       in      std_logic_vector(31 downto 0);
--Rotacion
PRx     :       out     std_logic_vector(63 downto 0);
PRy     :       out     std_logic_vector(63 downto 0);
PRz     :       out     std_logic_vector(63 downto 0)
);
end component Modulo_Rotacion;

component RotacionFIFO_Angulos
generic(
m       :       integer :=32;
n       :       integer :=9;
k       :       integer :=300
);
port(
RST     :       in      std_logic;
CLK     :       in      std_logic;
OPR     :       in      std_logic_vector(1 downto 0);
--Angulos de Euler
angX    :       in      std_logic_vector(8 downto 0);
angY    :       in      std_logic_vector(8 downto 0);
angZ    :       in      std_logic_vector(8 downto 0);
--Vector P
--DEp0  :       in      std_logic_vector(31 downto 0);
DEpx    :       in      std_logic_vector(31 downto 0);
DEpy    :       in      std_logic_vector(31 downto 0);
DEpz    :       in      std_logic_vector(31 downto 0);
--Rotacion
PRx     :       out     std_logic_vector(63 downto 0);
PRy     :       out     std_logic_vector(63 downto 0);
PRz     :       out     std_logic_vector(63 downto 0)
);
end component RotacionFIFO_Angulos;

component RotacionMatrizVector
```

```
port(
R11     :       in      std_logic_vector(31 downto 0);
R12     :       in      std_logic_vector(31 downto 0);
R13     :       in      std_logic_vector(31 downto 0);
R21     :       in      std_logic_vector(31 downto 0);
R22     :       in      std_logic_vector(31 downto 0);
R23     :       in      std_logic_vector(31 downto 0);
R31     :       in      std_logic_vector(31 downto 0);
R32     :       in      std_logic_vector(31 downto 0);
R33     :       in      std_logic_vector(31 downto 0);

Px      :       in      std_logic_vector(31 downto 0);
Py      :       in      std_logic_vector(31 downto 0);
Pz      :       in      std_logic_vector(31 downto 0);

PRx     :       out     std_logic_vector(63 downto 0);
PRy     :       out     std_logic_vector(63 downto 0);
PRz     :       out     std_logic_vector(63 downto 0)

);
end component RotacionMatrizVector;

end libreriaRotacionAngulos;
```

Bibliografía

[1] Rowan Hamilton William. *Elements of Quaternions*. Chelsea Pub Co, 1969.

[2] Nikolai A. Petrovsky Marek Parfieniuk and Alexander A. Petrovsky. *Rapid Prototyping Technology - Principles and Functional Requirements*. InTech, Chapters, 2011.

[3] Alexey Kuznetsov. History of the programmable logic. url www.fpgacentral.com/docs/fpga-tutorial/history-programmable-logic, 2009.

[4] Gomar Vera Yazmin. *Detección y Evasión de Obstáculos mediante plataforma robótica móvil DaNI 2.0*. Universidad de Guanajuato, 2012.

[5] Castilla Gallardo Manuel Alejandro. *Seguimiento virtual en tiempo real de maniobras de estabilización de un simulador de vuelo satelital*. Universidad Nacional Autónoma de México, 2013.

[6] M. McLean and J. Moore. FPGA- based single chip crytographic solution. url www.mil-embedded.com/pdfs/NSA.Mar07.pdf, 2007.

[7] Yanlin Zhu Kai Liu, Yuliang Yang. Tetris game design based on the FPGA. *Consumer Electronics, Communications and Networks (CECNet)*, 2012.

[8] J Leinen S. Suslov R.Patzak H. Winkler K.Schwan P.Dillinger, J.F. Vogelbruch. FPGA bases real-time image segmentation for medical systems and data processing. *Real Time Conference*, 2005.

[9] E.R. Bachmann R.B. McGhee M.J. Zyda J.L. Marins, X. Yun. An extended kalman filter for quaternion-based orientation estimation usig marg sensor. *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 4(3):2003–2011, 2001.

[10] Vivianne A. Mahecha. Aplicación de los números hipercomplejos o cuaterniones en imágenes de color. *Ingeniería y Desarrollo*, 23:77–83, 2008.

[11] J.L; Alberdi Primicia J.; Navarrete Marin J.J.; Oller González J.C. Barcala Riveira, J.M.; Fernández Marrón. *Aplicacion de las Wavelets y los Cuaterniones a la Clasificación de Espectros NIR*, volume 1027. Ciemat, 2003.

[12] Andrew J.Hanson. Visualizing quaternions. urlwww.pdfs.semanticscholar.org, 2005.

[13] L.H. Kauffman J.C. Hart, G.K. Francis. Visualizing quaternion rotation. *Transactions on Graphics*, 3(13):256–276, 1994.

[14] Ken Shoemake. Quaternions. url www.cs.ucr.edu/ vbz/resources/quatut.pdf, 1993.

[15] Qi Ma y Liming Zhang Chenlei Guo. Spatio-temporal saliency detection using phase spectrum of quaternion fourier transform. *Computer Vision and Pattern Recognition*, 1:23–28, 2008.

[16] NASA Mission Planning and Analysis Division. *Euler Angles, Quaternions and Transformation Matrices*. 1977.

[17] Martin John Baker. Maths - axis angle to quaternion. url www.euclideanspace.com/maths/geometry/rotations/conversions/angleToQuaternion/, 1998-2015.

[18] G. F. Torres del Castillo. La representación de rotaciones mediante cuaterninoes. *Miscelánea Matemática*, 1(23):43–50, 1999.

[19] National Instruments. Top 5 benefits. url www.ni.com/white-paper/6984/en/, 2012.

[20] F.Sorbello G. Vassallo S.Vitabile S. Franchini, A.Gentile. An FPGA implementation of a quadruple-bades multipler for 4D clifford algebra. *Digital System Desig Architectures, Methods and Tools*, (11):743–751, 2008.

[21] Robert D. Turney y Chris H. Dick. Real time image rotation and resizing, algorithms and implementations. urlwww.xilinx.com/products/logicore/dsp, 1999.

[22] Z. Shi Y. He. FPGA implementation of a floating-point quaternion-based attitude determination solution using peano-baker algorithm. *Procedia Engineerin*, 29:2279 – 2284, 2012.

[23] Nicolai A. Petrovsky Marek Parfieniuk and Alexabder A. Petrovsky. *Rapid Prototyping of Quaternion Multiplier: From Matrix Notation to FPGA-Based Circuits*. Dr. M. Hoque, 2011.

[24] A. Petrovsky A. Verenik, M. Parfieniuk. An FPGA implementation of the distributed arithmetic based quaternionic multipliers for paraunitary filter banks. *Mixed Desing of Integrayed Circuits and Systems*, (14):605–610, 2007.

[25] Fletcher William. *An engineering approach to digital design*. Prentice-Hall, 1980.