UNIVERSIDAD DE QUINTANA ROO
DIVISIÓN DE CIENCIAS E INGENIERÍA

DISEÑO DE BLOQUES EN HARDWARE: RECÍPROCO DE UN NÚMERO, RAÍZ INVERSA Y SU RECÍPROCO

TESIS
PARA OBTENER EL GRADO DE
INGENIERO EN REDES

PRESENTA
LUIS ISAAC DOMÍNGUEZ GÁMEZ

DIRECTOR DE TESIS
DR. JAVIER VÁZQUEZ CASTILLO

ASESORES
DR. JAIME SILVERIO ORTEGÓN AGUILAR
DR. FREDDY IGNACIO CHAN PUC
MTI. VLADIMIR VENIAMIN CABAÑAS VICTORIA
MTI. MELISSA BLANQUETO ESTRADA

CHETUMAL QUINTANA ROO, MÉXICO, SEPTIEMBRE DE 2015

UNIVERSIDAD DE QUINTANA ROO
DIVISIÓN DE CIENCIAS E INGENIERÍA

TRABAJO DE TESIS ELABORADO BAJO SUPERVISIÓN DEL COMITÉ DE ASESORÍA Y APROBADO COMO REQUISITO PARCIAL PARA OBTENER EL GRADO DE:

INGENIERO EN REDES

COMITÉ DE TRABAJO DE TESIS

DIRECTOR: _____
DR. JAVIER VAZQUEZ CASTILLO

ASESOR: _____
DR. JAIME SILVERIO ORTEGÓN AGUILAR

ASESOR: _____
DR. FREDDY IGNACIO CHAN PUC

UNIVERSIDAD DE QUINTANA ROO
DCI DIVISIÓN DE CIENCIAS E INGENIERÍA

UNIVERSIDAD DE QUINTANA ROO
AREA DE TITULACION

CHETUMAL QUINTANA ROO, MÉXICO, SEPTIEMBRE DE 2015

Resumen

Los microcontroladores son circuitos programables capaces de realizar diversas operaciones matemáticas las cuales pueden ser implementadas a base de sumas (multiplicación, resta de números). Sin embargo el número de operaciones basadas en sumas es limitado y por lo tanto ciertas funciones no pueden ser realizadas o implementadas a través de microcontroladores. Operaciones matemáticas tales como evaluación de senos, cosenos, tangentes y sus recíprocos, exponentes, logaritmos y raíces son funciones limitadas que un microcontrolador no puede realizar, ya que la memoria de un microcontrolador puede ser limitada y con ello introducir errores al aproximar el resultado de la operación deseada. Además, un problema de la latencia puede aparecer si se deciden utilizar métodos secuenciales basados en bloques CORDIC (COordinate Rotation Digital Computer). CORDIC es un algoritmo iterativo que permite calcular funciones trigonométricas, el cual se puede implementar en hardware usando sumadores, registros de desplazamientos y LUT's. Por otra parte, métodos basados en almacenamiento de funciones en tablas requieren grandes cantidades de memoria para mantener resoluciones significativas que eviten el impacto por perdida de resolución en la evaluación de las funciones (manejo de punto fijo y cuantificación en el procesador digital de señales).

Ante esta problemática que representa el tener una memoria reducida en un microcontrolador, surge este proyecto que busca proporcionar un bloque basado en hardware que realice operaciones matemáticas avanzadas. Para los casos de este proyecto se recurre a enfocarse únicamente en las operaciones de la raíz de un número, el reciproco de la raíz de un número y el inverso de un número.

Agradecimientos

Quiero agradecer este proyecto de tesis a mi asesor y director, el Dr. Javier Vázquez Castillo por haberme permitido realizar este proyecto de tesis y brindarme las herramientas con las que este proyecto pudo llevarse a cabo.

De igual forma quiero agradecer a todos los profesores que fueron parte importante en mi formación escolar y profesional, a esos profesores que dedicaron su esfuerzo en transmitir sus conocimientos y lograron dejar en mí una huella para poder ser un profesionista completo.

Este trabajo fue financiado en la convocatoria 2015 “Apoyo a la Titulación” de la división de Ciencias e Ingeniería con recursos PROFOCIE 2014.

Así mismo agradezco a la beca otorgada al ser integrado en el proyecto “Modelado de canales dispersivos para sistemas de comunicación móvil y sus implementaciones en Hardware” en el Apoyo a la Incorporación de Nuevos PTC del Programa para el Desarrollo Profesional Docente (PRODEP) y en el proyecto “Conceptualización, modelado e implementación en hardware de canales selectivos en tiempo, frecuencia y espacio para sistemas de comunicación móvil” de CONACyT-Ciencia básica con # 241272, en los cuales tuve la oportunidad de aportar con el desarrollo de mi trabajo de tesis.

Dedicatoria

Este proyecto de tesis va para mis padres. A mi madre, que siempre me enseñó a ser paciente ante todo y siempre dar lo mejor de uno no importando quien o que esté tratando de detenerme. A mi padre, por brindarme esa curiosidad y esos conocimientos propios de un ingeniero y que hicieron en mí una persona curiosa y muy interesada en diversos aspectos de la vida.

A mis hermanos Abril, Sofía, Andrea y Humberto, quienes siempre estaban ahí para sacarme una sonrisa y hacerme sentir tranquilo y a gusto en momentos difíciles y estresantes, por su ingenio para contar cosas elocuentes y así poder alegrarle el día a las personas a su alrededor.

A mis tíos y mis abuelos, los cuales siempre estuvieron apoyándome de manera indirecta, dándome fuerzas y fortaleciendo mis aptitudes tanto profesionales como personales.

A mis amigos, desde la primaria hasta los de la universidad, porque siempre en cada etapa de mi vida estuvieron ahí, siendo una fortaleza en mis momentos de clase y ayudarme a relacionarme más y más con la gente.

Y por último a aquellas personas, que aunque no estuvieron directamente conmigo, sirven de inspiración para poder realizar mis metas y sueños y no dejarme llevar por los comentarios de los demás, siempre pensar en lo mejor para todos y sobre todo para brindar un mejor servicio profesional y personal a todos, sin olvidar claro de dónde venimos y hacia dónde vamos.

"En mi tarjeta de presentación soy un presidente corporativo, en mi mente soy un diseñador de videojuegos, pero en mi corazón soy un gamer"

Satoru Iwata (1959-2015)

Índice

Índice de figuras

Índice de tablas

Índice de ecuaciones

1. Introducción a los circuitos programables.

Cuando se aborda el diseño de un sistema electrónico y surge la necesidad de implementar una parte con hardware dedicado son varias posibilidades que hay que tomar. En la Figura 1 se han presentado las principales aproximaciones ordenándolas en función de los parámetros de costo flexibilidad, prestaciones y complejidad.

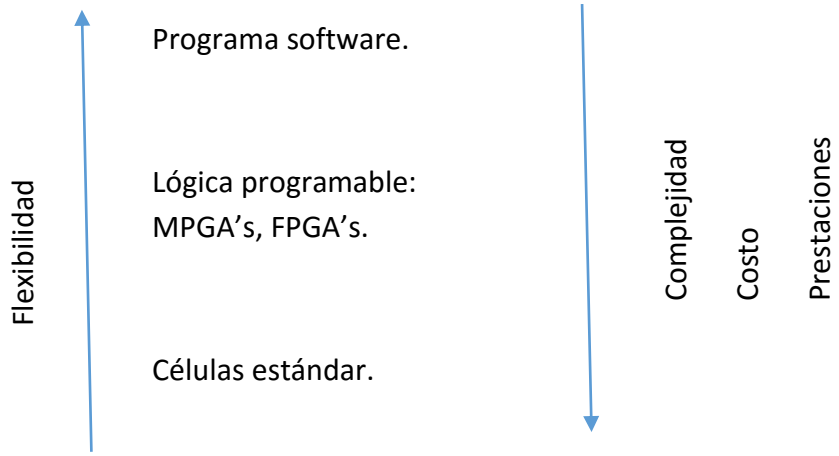

Figura 1. Diversas soluciones para el diseño de circuitos digitales

Como se puede observar en la figura, cuando se desea tener un diseño completo, esto implica en una disminución de la flexibilidad y un aumento muy considerable en costos y complejidad del diseño. Por otro lado, si se desea tener una flexibilidad mayor, reducimos el costo del diseño pero su implementación será menor y con una menor prestación.

En casos así es mejor considerar un diseño o semi-personalizado con células estándar o con un circuito previamente construido que pueda ser programado como son los FPGA's.

i. Evolución de los dispositivos programables

Se entiende como dispositivo programable aquel circuito de propósito general que posee una estructura interna que puede ser modificada por el usuario final (o a petición suya, por el fabricante) para implementar una gama de aplicaciones. El primer dispositivo que

cumplió con estas características era la memoria PROM (Programable Read Only Memory), que puede realizar un comportamiento de circuito utilizando líneas de direcciones como entradas y las de datos como salidas.

Los FPGA (Field Programmable Gate Array), introducidos por Xilinx en 1985, son el dispositivo programable por el usuario de uso general. También son llamadas LCA (Logic Cell Array). Consisten en una matriz bidimensional de bloques configurables que se pueden conectar mediante recursos generales de interconexión. Estos recursos incluyen segmentos de pista de diferentes longitudes más unos conmutadores programables para enlazar bloques a pistas o pistas entre sí.

Un FPGA es considerado la evolución más reciente de los dispositivos programables.

i. Estructura general de los FPGA's

La estructura básica de un FPGA se compone de tres elementos muy primordiales cuando estos son diseñados por Xilinx como se puede apreciar en la Figura 2:

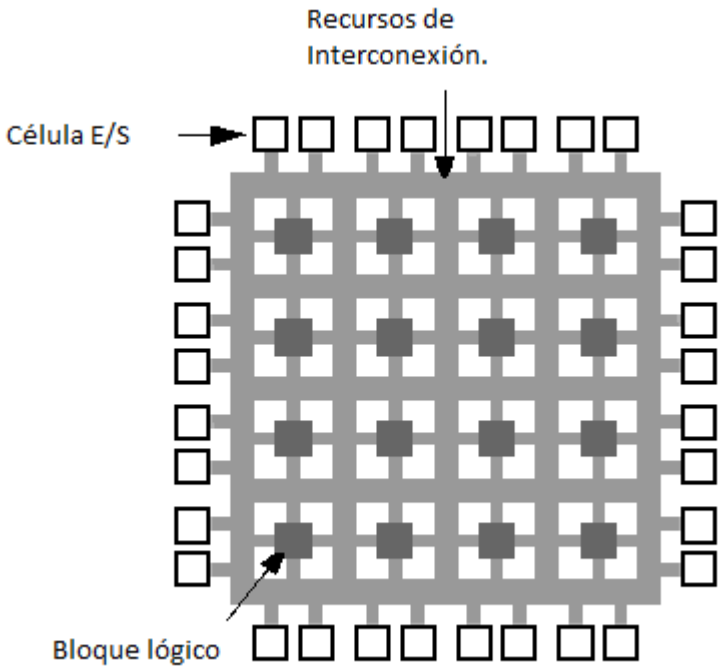

Figura 2. Estructura básica de un FPGA.

1.- **Bloques lógicos**, cuya estructura y contenido se denomina arquitectura. Hay varios tipos de arquitecturas, que varían principalmente en complejidad. Suelen incluir biestables para facilitar la implementación de circuitos secuenciales. Otros módulos de importancia son los bloques de Entrada/Salida.

2.- **Recursos de interconexión**, cuya estructura y contenido se denomina arquitectura de rutado.

3.- **Memoria RAM**, que se carga durante el RESET para configurar bloques y conectarlos.

Entre las ventajas que posee un FPGA podemos ver el bajo costo en presupuestos y su corto tiempo de producción, sin embargo como desventajas tenemos su baja velocidad de operación y baja densidad lógica (poca lógica implementable en un chip).

2. Evaluación de funciones

La evaluación de funciones comprende en sustituir el valor de una incógnita con un valor, sea entero, decimal o fraccionario y así encontrar el valor de otra incógnita o bien para comprobar una función y saber si ésta cumple con determinadas condiciones.

Evaluar una función es bastante sencillo, sin embargo en sistemas computacionales esto no es así debido a que las computadoras no poseen una capacidad para razonar funciones matemáticas complejas. Debido a esta carencia se utilizan los métodos numéricos que se encargan de aproximar las funciones a los resultados deseados.

Existen diversos métodos numéricos como el método de Newton-Raphson que es uno de los más comunes y sencillos de comprender ya que únicamente consiste en evaluar una función en un valor próximo a una raíz así como a su derivada.

De igual forma se cuenta con el método de interpolación, el método de la secante, método del punto fijo, bisección, entre otras.

El uso de métodos numéricos permite que muchos equipos computacionales puedan efectuar operaciones matemáticas complejas de una forma que la maquina pueda procesar la información en un microcontrolador.

i. Importancia de implementar funciones en un microcontrolador

Los microcontroladores son circuitos integrados o chips capaces de realizar contener una CPU, memoria y Unidades de Entrada y Salida, es decir puede efectuar operaciones como una computadora pero a un menor nivel.

Una de las ventajas de usar microcontroladores es su facilidad en su programación y lo económicos que algunos pueden resultar (esto varía de acuerdo a las características antes

mencionadas) además de que éstos pueden ser programados para realizar tareas específicas.

Sin embargo los microcontroladores poseen desventajas, una de ellas y la que más se denota, es la falta de un procesador lo suficientemente poderoso para soportar acciones más complejas, debido a su reducido tamaño, los microcontroladores poseen un microprocesador el cual está limitado a realizar operaciones básicas y sencillas; todas ellas basadas en sumas.

Debido a que un microcontrolador se encuentra con esta limitante, tiene que recurrir a otros equipos (o bloques) que ayuden a efectuar estas tareas, uno de estos equipos son los FPGA's los cuales ayudarán a que un microcontrolador pueda efectuar estas operaciones.

Y es que hoy en día hay muchas actividades que los microcontroladores, pongamos de ejemplo las consolas de videojuegos, específicamente las portátiles, son equipos en donde ahora se requiere de operaciones matemáticas para cargar elementos en 3D, dichas operaciones requieren de operaciones como exponentes o calcular raíces. Esto puede representar un problema para equipos de menor tamaño que requieren de un microcontrolador para efectuar las actividades del equipo, pero si tenemos una limitante en su microprocesador, es donde pueden entrar los sistemas programables a facilitar el trabajo a dicho microcontrolador.

Otro caso muy particular, es en el uso de filtros procesadores de señales, en ellos es necesario efectuar operaciones logarítmicas, exponenciales, radicaciones o inversos de números, lo cual puede representar un serio problema si no tenemos un sistema como un FPGA capaz de facilitar el trabajo al microcontrolador a la hora de efectuar estas operaciones.

ii. Módulos a implementar

Para el caso de este proyecto de tesis, se implementarán tres operaciones distintas las cuales se realizarán en un FPGA utilizando métodos numéricos para su manejo correcto en un circuito electrónico.

Las operaciones a implementar serán la raíz de un número, el inverso de un número y el inverso de la raíz de un número.

Sin embargo para implementar los módulos será necesario utilizar una de las operaciones como base debido a que al sustituir las tres operaciones en el método que vayamos a usar podría resultar un tanto complejo y más si éstas resultan tener un mayor grado de procesamiento para la tarjeta, en ese caso utilizaremos la operación inverso de una raíz como base ya que de ésta se pueden desprender las demás operaciones y más adelante podremos comprobar la razón de porque esta operación podría ayudarnos a optimizar mejor las demás operaciones.

El método numérico que se empleará será el método de Newton-Raphson, el cual es uno de los métodos más sencillos de usar y que facilitaría mucho su implementación en los FPGA's, más adelante veremos sus características y como es que funciona el método.

3. El método de Newton-Raphson

El método fue descrito por Isaac Newton en su ensayo "*Sobre el análisis mediante ecuaciones con un número finito de términos*", escrito en 1669 y publicado en 1711. En su publicación, Newton difería con la concepción actual del método ya que este solo era aplicado a polinomios y no era considerado para realizar las aproximaciones sucesivas x_n, sino que calculaba una serie de polinomios para aproximar la raíz de x. La descripción de Newton era más algebraica y no tenía relación alguna con el cálculo.

El método recibe el nombre de Newton-Raphson debido a que un matemático ingles de nombre Joseph Raphson (contemporáneo de Newton), quien ingreso al Royal Society en 1691 por publicar su libro "*Aequationum Universalis*" para poder aproximar raíces en 1690, en este libro describía un método de aproximar raíces similar al que realizo Newton en su libro "Método de fluxiones", en 1671, sin embargo el trabajo de Newton no se publicó hasta 1736, por lo que Raphson llego al mismo resultado de Newton y lo publicó 46 años antes de su publicación. Aunque en un inicio el resultado de Raphson no fue reconocido, debido a la popularidad de Newton, se le reconoció más tarde colocando su nombre en el mismo método.

i. Metodología del algoritmo de Newton-Raphson

El método de Newton-Raphson se caracteriza por aproximar el valor de una función $f(x)$ a cero, mediante el uso de la de la primera derivada de dicha función.

La fórmula para el método de Newton-Raphson es la siguiente:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad \textit{Ec. 1}$$

En la cual tenemos que x_n es el valor inicial de la función el cual se conoce como semilla. x_{n+1} Representa el resultado de la operación. $f(x)$ Es la función a hallar su raíz y $f'(x)$ es la primera derivada de dicha función.

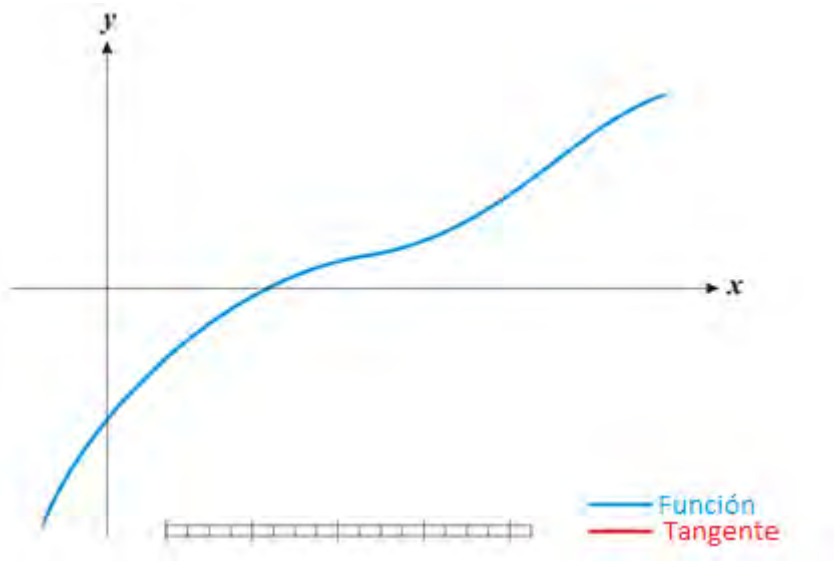

Figura 3. Función de ejemplo para Newton-Raphson.

El cálculo de la raíz de una función es presentado mediante el siguiente ejemplo gráfico: la Figura 3 muestra a una función a la cual se le desea encontrar su raíz (en línea azul); en este caso queremos aproximarla a su valor más cercano al cero mediante el uso de la derivada de la función evaluada en cierto punto.

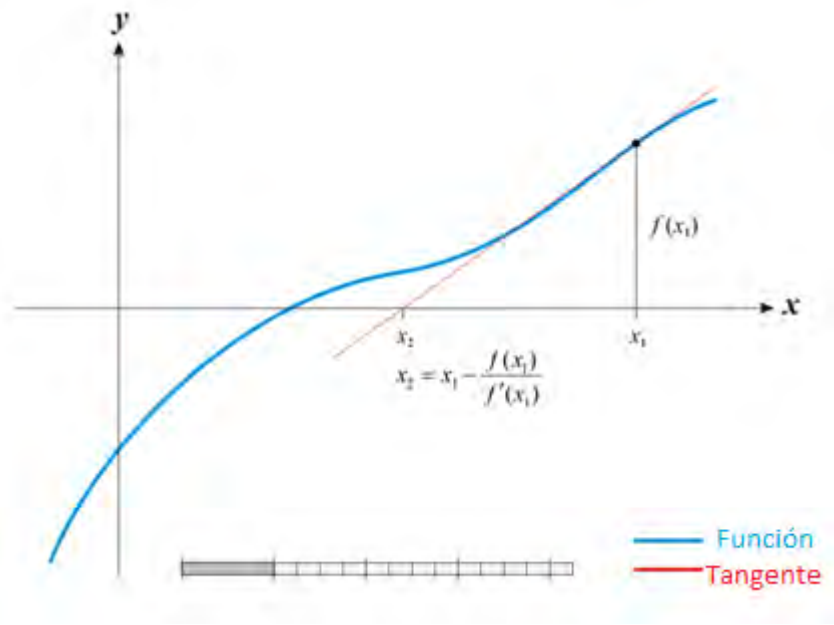

Figura 4. Primera iteración del algoritmo de Newton-Raphson.

En la Figura 4 podemos apreciar el primer paso que se realiza en el método de Newton-Raphson. Observamos cómo a partir de un valor inicial de x_1 (comúnmente llamado semilla), éste se evalúa en dicha función. Posteriormente, divide dicho valor con el correspondiente de la derivada de la misma función para obtener su tangente. Este valor es utilizado como valor inicial en la siguiente iteración.

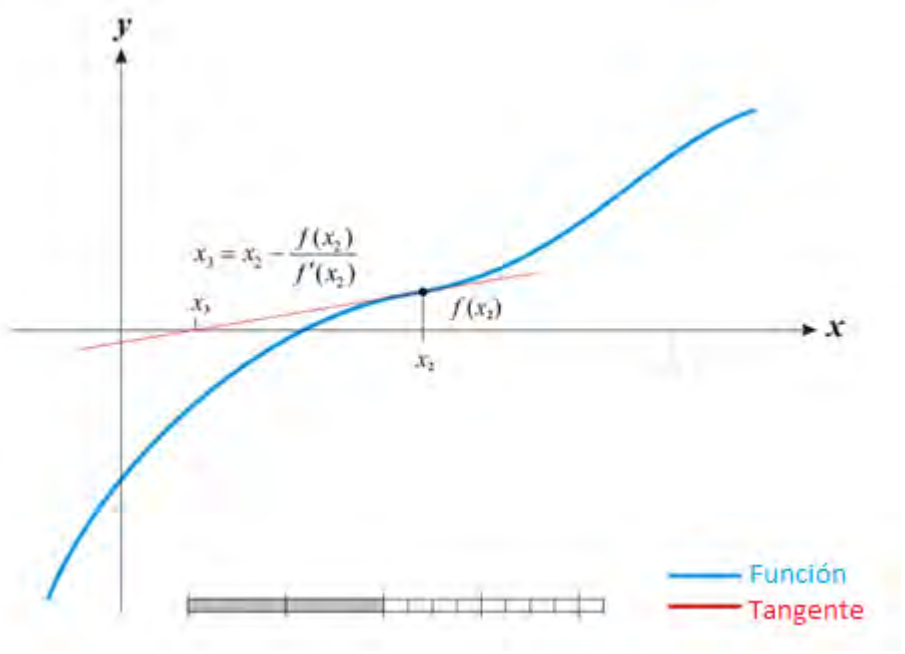

Figura 5. Segunda iteración del algoritmo de Newton-Raphson.

Una vez encontrado el valor de x_2 se realiza una segunda evaluación de Newton-Raphson. Podemos observar en la Figura 5, el resultado de la operación realizada (o segunda iteración) se aleja ligeramente del cero, esto puede ser debido a que la semilla elegida no era necesariamente la mejor, lo que ocasiona que para la tercera iteración, la raíz (valor la función) se aleje del resultado esperado.

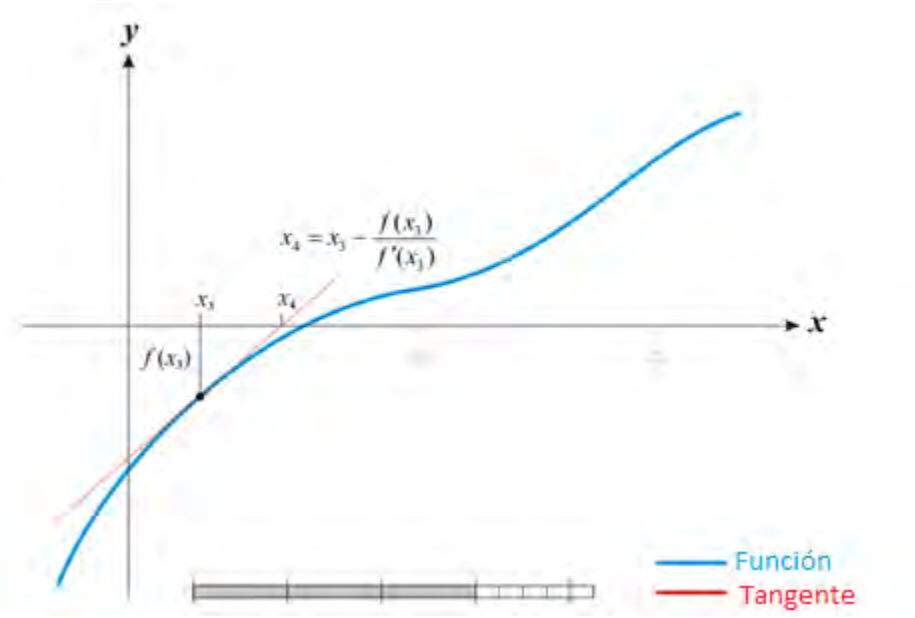

Figura 6. Tercera iteración del algoritmo de Newton-Raphson.

En la Figura 6 se evalúa Newton-Rapshon usando x_3 el cual, a pesar de lo alejado que esta del valor de cero en el sistema, podemos ver como el resultado de esta iteración es más cercana a cero, siendo x_4 el valor más próximo al resultado de momento.

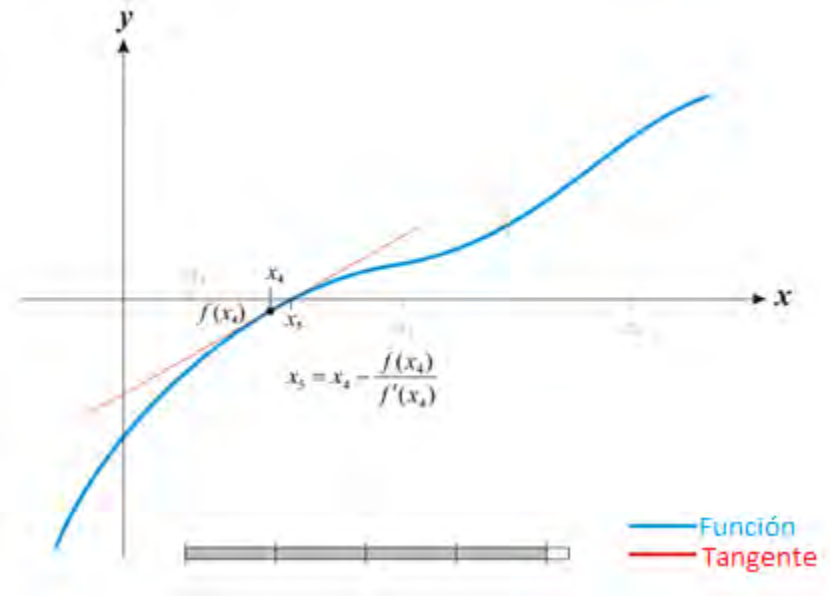

Figura 7. Cuarta iteración del algoritmo de Newton-Raphson.

La Figura 7 nos muestra el resultado de la cuarta iteración del algoritmo de Newton-Raphson, como se puede apreciar x_5 esta bastante más cercano al valor de cero, por lo que este es ahora el valor más cercano del método. Lo anterior puede deberse a que x_4 es un valor muy cercano a cero.

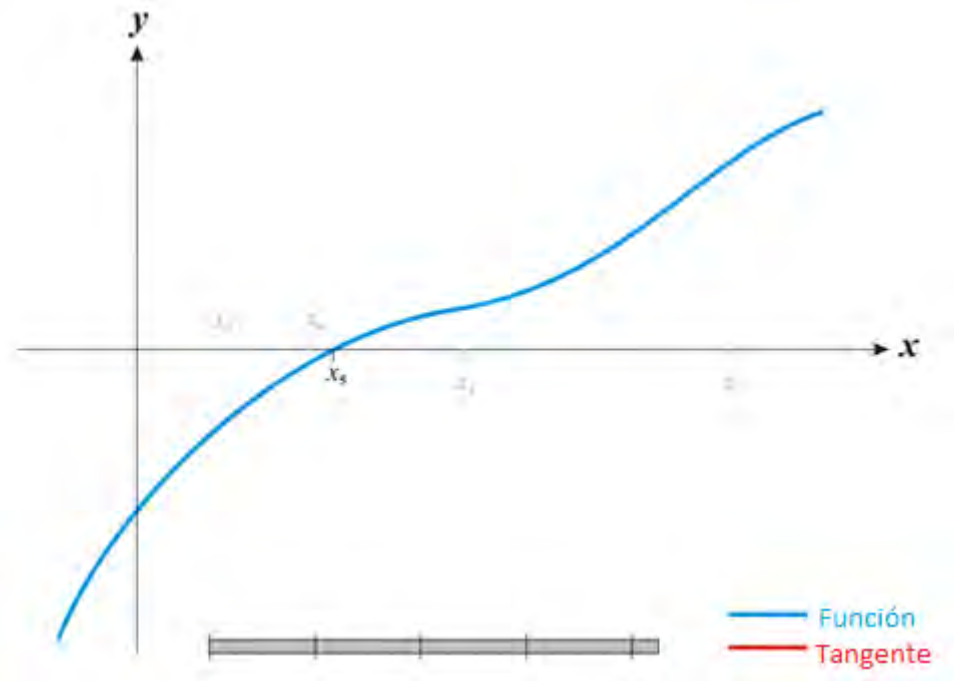

Figura 8. Resultado de Newton-Raphson.

Por último se tiene como resultado final a x_5 como la resolución definitiva a Newton-Raphson de la función planteada en la Figura 3. Como podemos apreciar en la Figura 8, el valor de x_5 es tan próximo a cero que este se puede ya tomar como un valor final del propio algoritmo.

Cabe mencionar que en muchos casos, serán necesarios más pasos en Newton-Raphson y esto se puede deber a diversos factores, como la semilla inicial (que juega un papel muy importante), la función a trabajar (usualmente las trigonométricas son un poco más complejas) y las capacidades de hardware del sistema que realiza el sistema.

4. Problemática

Los microcontroladores son circuitos programables capaces de realizar diversas operaciones matemáticas las cuales pueden ser implementadas a base de sumas (multiplicación, resta de números). Sin embargo el número de operaciones basadas en sumas es limitado y por lo tanto ciertas funciones no pueden ser realizadas o implementadas a través de microcontroladores. Operaciones matemáticas tales como evaluación de senos, cosenos, tangentes y sus recíprocos, exponentes, logaritmos y raíces son funciones especiales que un microcontrolador no pueden realizar, ya que la memoria de un microcontrolador puede ser limitada y con ello introducir errores al aproximar el resultado de la operación deseada. Además, un problema de la latencia puede aparecer si se deciden utilizar métodos secuenciales basados en bloques CORDIC (COordinate Rotation Digital Computer). CORDIC es un algoritmo iterativo que permite calcular funciones trigonométricas, el cual se puede implementar en hardware usando sumadores, registros de desplazamientos y LUT's. Métodos basados en almacenamiento de funciones en tablas requieren grandes cantidades de memoria para mantener resoluciones significativas que eviten el impacto por perdida de resolución en la evaluación de las funciones (manejo de punto fijo y cuantificación en el procesador digital de señales) (Amaya-Fernandez & Velasco-Medina).

5. Justificación

Ante esta problemática que representa el tener una memoria reducida en un microcontrolador, surge este proyecto que busca proporcionar un bloque basado en hardware que realice operaciones matemáticas avanzadas. Para los casos de este proyecto se recurre a enfocarse únicamente en las operaciones de la raíz de un número, el reciproco de la raíz de un número y el inverso de un número. De igual forma, con el bloque se desea tener un ahorro en la memoria en el microcontrolador.

6. Objetivo general

Diseñar un bloque evaluador de la raíz cuadrada, reciproco de la raíz cuadrada y la inversa de un número utilizando un hardware dedicado, el cual contendrá un algoritmo matemático y se programará utilizando un leguaje de descripción de hardware.

i. Objetivos específicos

- Evaluar funciones por el método de Newton-Raphson.
- Proponer una metodología de evaluación de funciones.
- Simular el algoritmo de evaluación (análisis en punto fijo).
- Aprender el lenguaje de descripción de hardware Verilog.
- Transformar el modelo de oro de Matlab a Verilog.
- Verificar la arquitectura diseñada.

7. Metodología

Para poder realizar este proyecto se contará con el algoritmo matemático de Newton-Raphson el cual será programado con un lenguaje de descripción de hardware. El método de Newton-Raphson es un método iterativo que nos permite aproximar la solución de un tipo de ecuación del tipo $f(x) = 0$ (López Nicolás, 2013). El método parte de un valor inicial que se introduce en una expresión relacionada con la ecuación, obteniendo así un resultado. Ese resultado se introduce en la misma expresión, obteniendo un nuevo resultado, y así sucesivamente. Si la elección del valor inicial es buena, cada vez que introducimos uno de los resultados obtenidos en esa expresión el método nos acerca más a la solución real mejor que la que tuvimos anteriormente (Palacios, s.f.). La importancia de este método es que proporciona una convergencia cuadrática al resultado en punto fijo.

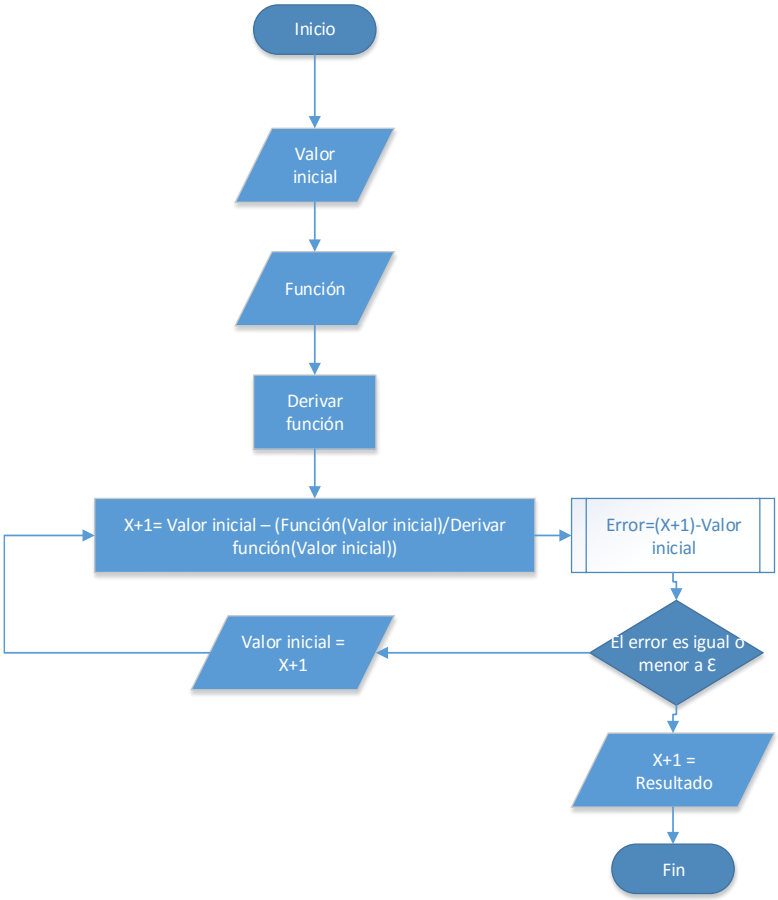

Figura 9. Algoritmo de Newton-Raphson.

Donde Ɛ representa un valor de error dado por el usuario o por las limitaciones del sistema o por restricciones de punto fijo.

Explicado el método de Newton-Raphson, procedemos a realizar las operaciones en la fórmula del método.

i. Hallar la raíz cuadrada de un número por el método de Newton-Raphson

Para hallar la raíz de un número se tiene que considerar lo siguiente:

$$x = \sqrt{R} \qquad \textit{Ec. 2}$$

Donde x es la raíz del número y R es el número entero positivo al cual se le desea hallar su raíz.

Considerando esto se realiza un despeje de la ecuación para expresarlo como una función.

$$x = \sqrt{R} \qquad \textit{Ec. 3}$$

$$x^2 = R \qquad \textit{Ec. 4}$$

$$x^2 - R = 0 \qquad \textit{Ec. 5}$$

Una vez planteada la función hay que derivarla acorde al algoritmo de Newton-Raphson.

$$f'(x) = 2x \qquad \textit{Ec. 6}$$

Una vez realizado lo anterior, procedemos a sustituir valores en la fórmula de Newton-Raphson para llegar al algoritmo iterativo.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad \textit{Ec. 7}$$

$$x_{n+1} = x_n - \frac{x_n^2 - R}{2x_n} = \frac{2x_n^2 - x_n^2 + R}{2x_n} = \frac{2x_n^2 - x_n^2}{2x_n} + \frac{R}{2x_n} = \frac{2x_n^2 - x_n^2}{2x_n} + \frac{R}{2x_n} = \frac{x_n\,(2x_n - x_n)}{2x_n} + \frac{R}{2x_n} = \frac{2x_n - x_n}{2} + \frac{R}{2x_n} = \frac{1}{2}\left(2x_n - x_n + \frac{R}{x_n}\right) \qquad \textit{Ec. 8}$$

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{R}{x_n}\right) \qquad \textit{Ec. 9}$$

Teniendo esta fórmula se busca la raíz de un número, donde:

- x_{n+1} es el valor siguiente que se tomará como raíz de la función.

- x_n es el valor actual que se toma como valor inicial para realizar el cálculo de la raíz de la función.
- R es el número entero positivo a quién se desea encontrar la raíz.

ii. Hallar el reciproco de la raíz cuadrada de un número por el método de Newton-Raphson

Para hallar el reciproco de la raíz de un número hay que considerar lo siguiente:

$$x = \frac{1}{\sqrt{R}} \qquad \textit{Ec. 10}$$

Donde x es el resultado del reciproco de la raíz de un número y R es un número entero positivo al que se le desea hallar su raíz.

Considerando esto se tiene que expresar la ecuación como función.

$$x = \frac{1}{\sqrt{R}} \qquad \textit{Ec. 11}$$

$$\sqrt{R} = \frac{1}{x} \qquad \textit{Ec. 12}$$

$$R = \frac{1}{x^2} \qquad \textit{Ec. 13}$$

$$\frac{1}{x^2} - R = 0 \qquad \textit{Ec. 14}$$

Se deriva la función.

$$f'(x) = -\frac{2}{x^3} \qquad \textit{Ec. 15}$$

Ahora se sustituye en la fórmula de Newton-Raphson de la siguiente forma:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad \textit{Ec. 16}$$

$$x_{n+1} = x_n - \frac{\frac{1}{x_n^2} - R}{-\frac{2}{x_n^3}} = x_n - \frac{x_n^3\left(\frac{1}{x_n^2} - R\right)}{-2} = x_n - \frac{x_n - x_n^3 R}{-2} = \frac{-2x_n - x_n + x_n^3 R}{-2} = \frac{2x_n + x_n - x_n^3 R}{2} =$$

$$\frac{1}{2}(2x_n + x_n - x_n^3 R) \qquad \textit{Ec. 17}$$

$$x_{n+1} = \frac{1}{2}(3x_n - x_n^3 R) \quad \textit{Ec. 18}$$

iii. Hallar el inverso de un número por el método de Newton-Raphson

Para hallar el inverso de un número es necesario considerar lo siguiente:

$$x = \frac{1}{R} \quad \textit{Ec. 19}$$

Donde x es el resultado del inverso de un número y R es el número entero positivo el cual se le quiere encontrar el inverso.

Considerando esto se despeja la ecuación para que quede como una función:

$$R = \frac{1}{x} \quad \textit{Ec. 20}$$

$$\frac{1}{x} - R = 0 \quad \textit{Ec. 21}$$

Se deriva la función.

$$f'(x) = -\frac{1}{x^2} \quad \textit{Ec. 22}$$

Ahora se sustituye en la fórmula de Newton-Raphson.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \textit{Ec. 23}$$

$$x_{n+1} = x_n - \frac{\frac{1}{x_n} - R}{-\frac{1}{x_n^2}} = x_n + x_n^2\left(\frac{1}{x_n} - R\right) = x_n + x_n - x_n^2 R \quad \textit{Ec. 24}$$

$$x_{n+1} = 2x_n - x_n^2 R \quad \textit{Ec. 25}$$

Una vez hechas las operaciones tenemos que cada operación que vamos a realizar tiene una fórmula con Newton-Raphson.

$$\sqrt{R} = \frac{1}{2}\left(x_n + \frac{R}{x_n}\right) \quad \textit{Ec. 26}$$

$$\frac{1}{\sqrt{R}} = \frac{1}{2}(3x_n - x_n^3 R) \quad \textit{Ec. 27}$$

$$\frac{1}{R} = 2x_n - x_n^2 R \qquad \textit{Ec. 28}$$

Como podemos ver con estas fórmulas se busca dar solución a las operaciones a efectuar en el FPGA, sin embargo nos topamos con diversos inconvenientes:

El **primer inconveniente** que nos topamos es con la fórmula $\sqrt{\mathrm{R}} = \frac{1}{2}\left(x_n + \frac{R}{x_n}\right)$ debido a que por limitantes del sistema una operación como $\frac{R}{x_n}$ podría implicar un mayor procesamiento en el sistema.

El **segundo inconveniente** es que se intenta buscar la solución de las tres operaciones mediante una fórmula sencilla y permita reducir el número de procesamiento en el sistema. Para esto tenemos las dos fórmulas restantes $x_{n+1} = \frac{1}{2}(3x_n - x_n^3 R)$ Ec. 18 y $\frac{1}{R} = 2x_n - x_n^2 R$ Ec. 28, como podemos observar la fórmula de $\frac{1}{R}$ es la más simple para realizar, el detalle es que esta fórmula no permite resolver las otras dos a partir de ésta (si se desea implementar las tres funciones en un mismo hardware). Por lo que la fórmula de $\frac{1}{\sqrt{R}}$ nos permitirá tener la solución a las otras dos fórmulas como sigue:

$$\frac{1}{\sqrt{R}} * R = R * R^{-\frac{1}{2}} = R^{\frac{1}{2}} = \sqrt{R} \qquad \textit{Ec. 29}$$

Así:

$$\sqrt{R} = \frac{R}{2}(3x_n R - x_n^3 R^2) \qquad \textit{Ec. 30}$$

y

$$\frac{1}{\sqrt{R}} * \frac{1}{\sqrt{R}} = \frac{1}{R} \qquad \textit{Ec. 31}$$

Por lo tanto:

$$\frac{1}{R} = \frac{1}{4}(3x_n - x_n^3 R)^2 \qquad \textit{Ec. 32}$$

Teniendo los resultados anteriores de las fórmulas ahora podremos llevar acabo las tres operaciones en el sistema y con el mismo hardware, haciendo uso de *un multiplicador adicional*.

De igual forma se presentan arquitecturas propuestas del sistema, esta arquitecturas permitirán darnos una idea de cómo poder llevar acabo las operaciones en el FPGA y dar un seguimiento de cómo se tendrán que programar las operaciones.

iv. Bloque para evaluar el inverso de un número

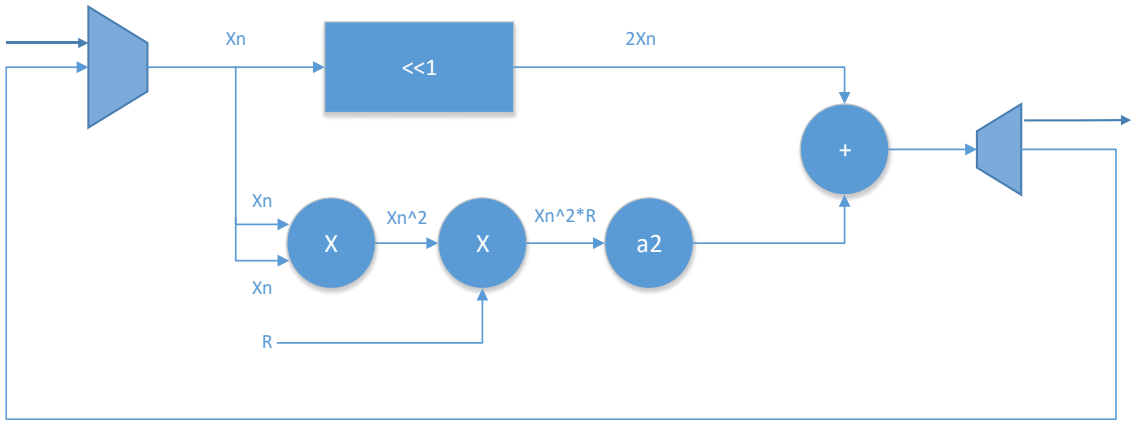

Figura 10. Diagrama de bloques para el inverso de un número.

En la Figura 10 se puede observar el diagrama de bloques del inverso de un número, el cual se describirá a continuación:

Primeramente la variable x_n pasa por un corrimiento de bits a la derecha, esto es por medio de un Shift Register, lo cual hará que se multiplique por dos el valor que se indique en x_n de ahí se pasa a un sumador.

Del otro lado del sumador se colocará nuevamente la variable x_n y se multiplicará así misma teniendo el valor de x_n al cuadrado, luego el resultado se multiplica con la variable R para obtener el otro lado del sumador, no sin antes realizar un complemento $a2$ para colocarlos en el sumador y así realizar ya la operación final del diagrama.

El complemento $a2$ es una herramienta que sirve para volver negativo el valor de un número en binario, consiste en negar el valor en bits del valor que se tiene y sumarle un 1 al final y con eso se obtiene el complemento $a2$ de un número, puede representar un valor distinto al esperado, pero si se realizan las operaciones indicadas se puede obtener el resultado como si se efectuará una resta normal.

v. Bloque para evaluar el inverso de la raíz de un número

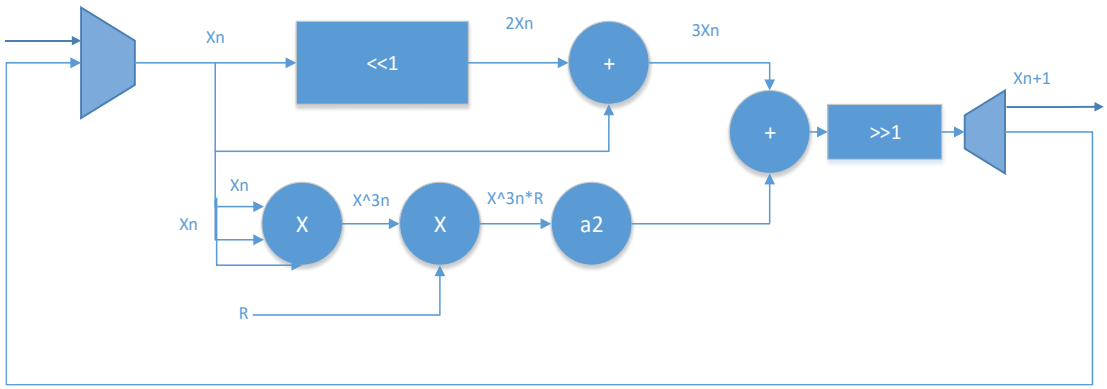

Figura 11. Diagrama de bloques para el inverso de la raíz de un número.

En la Figura 11 podemos ver el diagrama de bloques del inverso de la raíz de un número el cual se detallará a continuación:

La variable x_n entra a un Shift Register el cual recorrerá los bits a la derecha para multiplicar por 2 el valor que está en x_n luego de ello se pasa a un sumador, donde se suma $2x_n$ con x_n para tener el triple de x_n y así colocarlo en el sumador principal.

En la otra parte del sumador principal tenemos a la variable x_n la cual se introduce a un multiplicador y se multiplica 3 veces para obtener el triple de x_n, luego se coloca el triple de x_n a otro multiplicador donde se coloca con la variable R y se multiplican entre sí, luego se realiza el complemento $a2$ del resultado del multiplicador y se coloca al sumador principal.

Finalmente, el resultado del sumador principal es enviado a un Shift Register que se va a encargar realizar un corrimiento de los bits a la derecha para poder dividir entre dos el resultado que se obtenga del sumador, así cumpliendo con la $x_{n+1} = \frac{1}{2}(3x_n - x_n^3 R)$ Ec. 18.

vi. Bloque para evaluar la raíz de un número

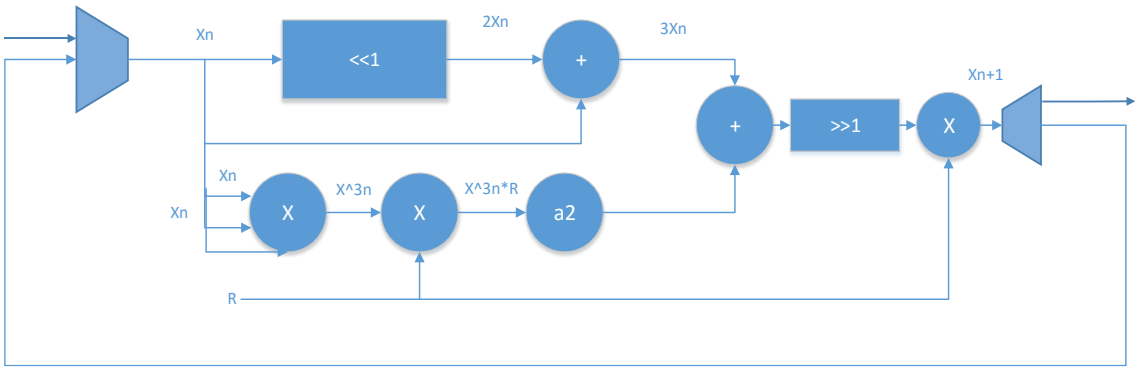

Figura 12. Diagrama de bloques para la raíz de un número.

En la Figura 12 podemos observar el diagrama de bloques de la raíz de un número.

La variable x_n entra a un Shift Register el cual recorrerá los bits a la derecha para multiplicar por 2 el valor que está en x_n luego de ello se pasa a un sumador, donde se suma $2x_n$ con x_n para tener el triple de x_n y así colocarlo en el sumador principal.

En la otra parte del sumador principal tenemos a la variable x_n la cual se introduce a un multiplicador y se multiplica 3 veces para obtener el triple de x_n, luego se coloca el triple de x_n a otro multiplicador donde se coloca con la variable R y se multiplican entre sí, luego se realiza el complemento $a2$ del resultado del multiplicador y se coloca al sumador principal.

Básicamente este diagrama se hizo basándose en la $x_{n+1} = \frac{1}{2}(3x_n - x_n^3 R)$ Ec. 18 debido a que no se utilizará la $x_{n+1} = \frac{1}{2}\left(x_n + \frac{R}{x_n}\right)$ Ec. 9 en este diagrama, debido a las limitantes que se exponen anteriormente. El cambio principal es que al final se multiplica el resultado del multiplicador principal con el valor de la variable R para obtener el resultado final.

Cabe mencionar que solo al bloque de raíz de un número se utilizó la $x_{n+1} = \frac{1}{2}(3x_n - x_n^3 R)$ Ec. 18 debido a las limitantes del sistema para realizar la operación de la $x_{n+1} = \frac{1}{2}\left(x_n + \frac{R}{x_n}\right)$ Ec. 9, sin embargo en la operación de inverso de un número se utilizó la ecuación tomada del resultado de Newton-Raphson. La razón de

porque se usaron estas arquitecturas es debido a que es probable que para futuras referencias pueda servir como base para otros proyectos a fines y para realizar comprobaciones del sistema.

8. Implementación

Para la implementación haremos uso de la herramienta conocida como Matlab para hacer un primer plano de lo que serán las funciones vistas y también para tener una forma de comprobar los resultados con el FPGA.

En Matlab se utilizarán las arquitecturas propuestas vistas anteriormente para así comprobar su funcionamiento y que tanto nos aproximan a los resultados esperados.

Para ello se hizo uso de la función de Matlab fi la cual corresponde al ToolBox de punto-fijo de Matlab. La función **fi** tiene la siguiente sintaxis:

B_q=fi(B,s,p,f).

Donde:

B_q: Será la variable que almacenará el valor en punto fijo devuelto de la función “fi” de Matlab.

fi: Es la función que devuelve un valor en punto fijo.

B: Es el valor que la función “fi” tomará para realizar el valor en punto fijo.

s: Es el bit que representará el signo del número en su valor en punto fijo.

p: Es la longitud de la palabra o el total de bits que comprenden el número en punto fijo.

f: Es la longitud en bits de la parte fraccionaria, es decir, el número de bits que utilizará Matlab para expresar los decimales en el valor en punto fijo.

Por otra parte para calcular la efectividad del valor de Newton-Raphson en punto fijo, se hará uso de una fórmula llamada la SQNR (Signal Quality to Noise Ratio) que se puede llamar como la relación señal/ruido.

La relación señal/ruido se define como la potencia de una señal que se transmite con respecto al ruido que la corrompe (Wikipedia, 2015).

Para objetos de esta tesis la relación señal/ruido nos permitirá definir qué tan cercano es el resultado del algoritmo de Newton-Raphson, con respecto al resultado real de la operación realizada en Matlab.

La fórmula a utilizar en este módulo y servirá como medio para comprobar la resolución del algoritmo será la siguiente.

$$SQNR = 10\,log_{10}\frac{S/In}{S/Out} \qquad \textit{Ec. 33}$$

Donde tenemos que:

S/In: Es la señal de entrada a la fórmula.

S/Out: Es la señal de salida o la señal de ruido a la fórmula.

A su vez **S/In** y **S/Out** contienen elementos internos los cuales son:

$$S/In = \sum f(x)^2 \qquad \textit{Ec. 34}$$

Donde tenemos que:

f(x): Es la función real del sistema a evaluar.

$$S/Out = \sum (f(x) - NR)^2 \qquad \textit{Ec. 35}$$

Donde tenemos que:

f(x): Es la función real del sistema a evaluar.

NR: Es el resultado del algoritmo de Newton-Raphson.

Teniendo la $S/In = \sum f(x)^2$ Ec. 34 y $S/Out = \sum (f(x) - NR)^2$

Ec. 35 podemos unir todo en la $SQNR = 10\, log_{10} \frac{S/In}{S/Out}$ Ec.

33 y tendremos lo siguiente:

$$SQNR = \frac{\sum f(x)^2}{\sum (f(x) - NR)^2} \qquad \textit{Ec. 36}$$

$f(x)2(fx-NR)2$ Ec. 36 será la que utilizaremos en el código en Matlab para comprobar que tan efectivo es el algoritmo de Newton-Raphson con respecto al valor real de la función.

Cabe destacar que si el valor de SQNR es igual o mayor a 60, quiere decir que la resolución es óptima y por lo tanto el algoritmo de Newton-Raphson es efectivo con respecto al resultado final obtenido.

i. Código en Matlab de raíz de un número.

En este primer código se efectuará el cálculo de la raíz de un número utilizando el punto fijo en el sistema y la resolución del algoritmo con el cálculo del SQNR.

Además se hará uso del ciclo "**for**" el cual se repetirá dos veces para así simular diferentes iteraciones del algoritmo de Newton-Raphson.

La semilla de este sistema será la propuesta en el módulo inverso de la raíz de un número.

```
% Calcula sqrt(R)

R=2; % R es el valor al que le vamos a hallar su raíz.
x=0.7; % La semilla debe elegirse en base a 1/sqrt(R)

for i=1:2  %Ciclo for que nos permitirá hallar la raíz de R en 2
iteraciones.

      R_q=fi(R,1,16,13,'RoundMode','fix'); %Volvemos "R" en un valor en
      punto fijo.
      x_q=fi(x,1,16,13,'RoundMode','fix'); %Volvemos "x" un valor en
      punto fijo.
```

```
        a= 3*x_q; % a es una sub operación dentro de la fórmula de Newton-
        Raphson.
        a_q=fi(a,1,16,12,'RoundMode','fix'); %Volvemos "a" un valor de
        punto fijo.

        b= x_q*x_q*x_q; % b es una sub operación dentro de la fórmula de
        Newton-Raphson.
        b_q=fi(b,1,32,26,'RoundMode','fix'); %Volvemos "b" un valor de
        punto fijo.

        c= b_q*R_q; % c es una suboperación de Newton-Raphson en la cual se
        usan los valores en punto fijo de "R" y "b".
        c_q=fi(c,1,32,26,'RoundMode','fix'); %Volvemos "c" un valor de
        punto fijo.

        d= a_q-c_q; % d es la suboperación de Newton-Raphson que usa los
        valores en punto fijo de "a" y "c".
        d_q=fi(d,1,32,20,'roundmode','fix'); %Volvemos "d" un valor de
        punto fijo.

        e= d_q/2; % e es la suboperación final en donde usamos el valor de
        punto fijo de "d".
        e_q=fi(e,1,21,19,'roundmode','fix'); %Volvemos "e" un valor de
        punto fijo.


        x=e_q; %Igualamos el valor de "x" con el resultado final de la
        operación de Newton-Raphson.

end

%Terminada la operación de Newton-Raphson procedemos a realizar una
%operación más.

f= e_q*R_q; %f es la operación en la cual multiplicamos el valor de
Newton-Raphson el valor de R para hallar la raíz.
f_q=fi(f,1,22,19,'roundmode','fix'); %Volvemos a "f" en un valor de punto
fijo.

f_q.data %Imprimimos el resultado de la operación en formato de punto
fijo.
f_q.bin %Imprimimos el resultado de la operación en formato binario.

SQNR = 10*log10(sum((R/sqrt(R)).^2)/sum( ((R/sqrt(R)) -f_q.data).^2));
%Realizamos el cálculo de la resolución el cual compara el resultado de
"f" con el resultado verdadero.
sprintf('ITERA = %d',i) %Imprimimos el número de iteraciones que tomo
encontrar el resultado.
sprintf('SQNR = %f',SQNR) %Imprimimos el valor de resolución.
```

ii. Código en Matlab para inverso de un número

Este código en Matlab permite calcular el inverso de un número con el método de Newton-Raphson usando un ciclo "**for**" para repetir el algoritmo dos veces. Al finalizar se utilizará la fórmula de SQNR para comprobar la resolución del algoritmo con respecto al resultado real del módulo.

```
% Calcula 1/R

R=2; %R es el número al que le vamos a hallar su reciproco.
x=0.55; %x es el valor inicial o semilla que usaremos para aproximar el
resultado.

for i=1:2 %Ciclo for que nos permitirá hallar el reciproco de R en 2
iteraciones.

      R_q=fi(R,0,16,13,'roundmode','fix'); %Volvemos "R" en un valor en
      punto fijo.
      x_q=fi(x,0,16,13,'roundmode','fix'); %Volvemos "x" un valor en
      punto fijo.

      a= 2*x_q; % a es una sub operación dentro de la fórmula de Newton-
      Raphson.
      a_q=fi(a,0,16,12,'roundmode','fix'); %Volvemos "a" un valor de
      punto fijo.

      b=x_q*x_q; % b es una sub operación dentro de la fórmula de Newton-
      Rapshon.
      b_q=fi(b,0,32,26,'roundmode','fix'); %Volvemos "b" en un valor en
      punto fijo.

      bb=b_q*R_q; % bb es la suboperación en donde utilizamos los valores
      en punto fijo de "R" y "b".
      bb_q=fi(bb,0,21,19,'roundmode','fix'); %Volvemos "bb" en un valor
      de punto fijo.

      c=a_q-bb_q; % c es la suboperación final en donde usamos el valor
      de punto fijo de "a" y de "bb".
      c_q=fi(c,0,22,19,'roundmode','fix'); %Volvemos "c" en un valor en
      punto fijo.

      x=c_q; %Igualamos el valor de "x" con el resultado final de la
      operación de Newton-Raphson.

      c_q.data %Imprimimos el resultado de la operación en formato de
      punto fijo.
      c_q.bin %Imprimimos el resultado de la operación en formato
      binario.

      %pause;
```

```
        SQNR = 10*log10(sum((1/R).^2)/sum((1/R-c_q.data).^2)); %Realizamos
        el cálculo de la resolución el cual compara el resultado de Newton-
        Raphson con el resultado verdadero.
        sprintf('ITERA = %d',i) %Imprimimos el número de iteraciones que
        tomo encontrar el resultado.
        sprintf('SQNR = %f',SQNR) %Imprimimos el valor de resolución de
        Newton-Rapshon frente al resultado real.

end
```

iii. Código en Matlab para el inverso de la raíz de un número

En el código siguiente se muestra el algoritmo de Newton-Raphson para calcular el inverso la raíz de un número usando un ciclo "**for**" de dos iteraciones para llevar acabo el algoritmo. Seguidamente se utilizará la SQNR para determinar la resolución del resultado de Newton-Raphson con el resultado real del módulo.

```
% Calcula 1/sqrt(R)

R=4; %R es el número al que le vamos a hallar el reciproco de su raíz.
x=0.4; %x es el valor inicial o semilla que usaremos para aproximar el
resultado.

for i=1:3 %Ciclo for que nos permitirá hallar el reciproco de la raíz de
R en 2 iteraciones.

        R_q=fi(R,1,16,13,'RoundMode','fix'); %Volvemos "R" en un valor en
        punto fijo.
        x_q=fi(x,1,16,13,'RoundMode','fix'); %Volvemos "x" un valor en
        punto fijo.

        a= 3*x_q; % a es una sub operación dentro de la fórmula de Newton-
        Raphson.
        a_q=fi(a,1,16,12,'RoundMode','fix'); %Volvemos "a" un valor de
        punto fijo.

        b= x_q*x_q*x_q; % b es una sub operación dentro de la fórmula de
        Newton-Rapshon.
        b_q=fi(b,1,32,26,'RoundMode','fix'); %Volvemos "b" en un valor en
        punto fijo.
```

```
        c= b*R_q; % c es una suboperación de Newton-Raphson en la cual se
        usan los valores en punto fijo de "R" y "b".
        c_q=fi(c,1,32,26,'RoundMode','fix'); %Volvemos "c" un valor de
        punto fijo.

        d= a_q-c_q; % d es la suboperación de Newton-Raphson que usa los
        valores en punto fijo de "a" y "c".
        d_q=fi(d,1,32,20,'roundmode','fix'); %Volvemos "d" un valor de
        punto fijo.

        e= d_q/2; % e es la suboperación final en donde usamos el valor de
        punto fijo de "d".
        e_q=fi(e,1,21,19,'roundmode','fix'); %Volvemos "e" un valor de
        punto fijo.

        x=e_q; %Igualamos el valor de "x" con el resultado final de la
        operación de Newton-Raphson.

        e_q.data %Imprimimos el resultado de la operación en formato de
        punto fijo.
        e_q.bin %Imprimimos el resultado de la operación en formato
        binario.

        SQNR = 10*log10(sum((1/sqrt(R)).^2)/sum( ((1/sqrt(R)) -
        e_q.data).^2)); %Realizamos el cálculo de la resolución el cual
        compara el resultado de Newton-Raphson con el resultado verdadero.
        sprintf('ITERA = %d',i) %Imprimimos el número de iteraciones que
        tomo encontrar el resultado.
        sprintf('SQNR = %f',SQNR) %Imprimimos el valor de resolución.

end
```

Como se mencionó anteriormente, la codificación en Matlab sirve para realizar comparaciones de resultados con los que se obtendrán con Verilog y el resultado esperado y comprobar su margen de error.

Luego en Verilog se procede a realizar la codificación del sistema base con el cual se podrán obtener los tres módulos a partir de uno solo.

Iniciamos con el modulo que permite calcular el inverso de la raíz de un número ya que, como se dijo anteriormente, será el que nos permita calcular los otros dos a base de operaciones mucho más sencillas.

Primeramente se realizará un multiplicador donde se realizará la primera operación del sistema, la cual es elevar al cubo el valor de la semilla x_n con el valor del número a buscarle su inverso de raíz R.

iv. Módulo de raíz de un número

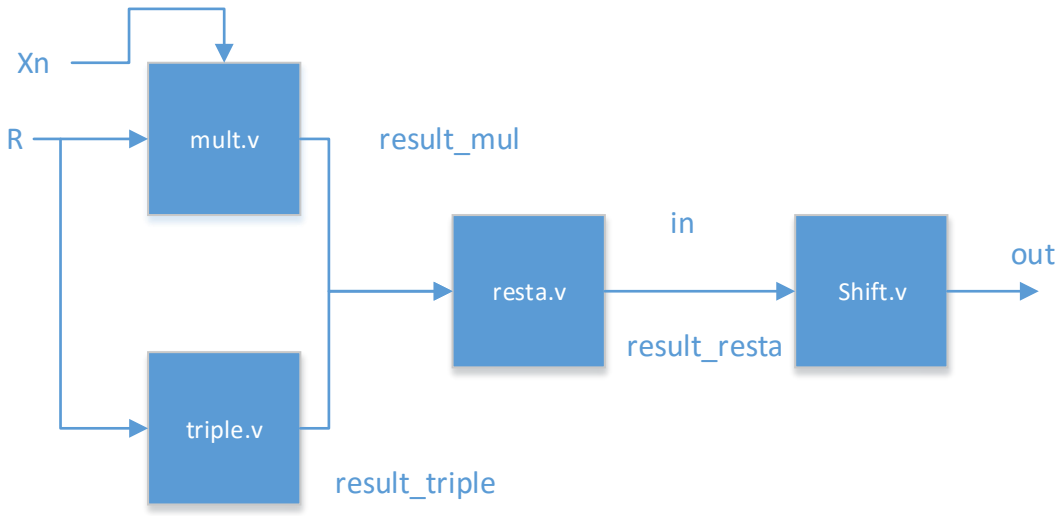

Figura 13. Diagrama de composición de los archivos y variables que comprenden el módulo de inverso de la raíz de un número.

En la Figura 6 podemos observar cómo se van a relacionar cada uno de los archivos para llevar acabo las diversas operaciones, a continuación se explicará el funcionamiento de cada archivo.

Mul.v: El código de este archivo se va a encargar de multiplicar las variables x_n y R, pero primeramente realizará la operación de elevar al cubo la variable x_n y luego procederá a multiplicar ese resultado con la variable R. Este archivo devuelve la variable result_mul.

Triple.v: El código de este archivo se encarga de multiplicar por tres la variable R. Este archivo devuelve la variable result_triple.

Resta.v: El código de este archivo toma los resultados de las variables result_mul y result_triple y las deposita en sus variables locales para poder realizar la resta entre ambas variables. Este archivo devuelve la variable result_resta.

Shift.v: El código de este archivo se va a encargar de realizar un corrimiento de bits a la izquierda y de llenar el espacio vacío con un cero. En **Shift.v** se toma la variable devuelta del archivo **Resta.v** result_resta y la almacena en su variable local como "in" del módulo, la cual es del mismo tipo de arreglo que el de la variable result_resta. El resultado de este archivo se almacena en la variable "out", que nos arroja el resultado final de la operación.

En el código del archivo **Shift.v** se realiza un Shift Register que, como se mencionó anteriormente, es un corrimiento de registro en los bits ya sea a la derecha o a la izquierda, según sea el caso. Un Shift Register tiene diversos usos, no solo se puede usar en bits, sino igual en datos almacenados en un arreglo según sea el caso.

El hacer uso de un Shift Register en nuestro proyecto permite realizar una operación sencilla pero fundamental, ya que al correr los bits a la izquierda realizamos una división entre dos cosa que se realiza en la fórmula original de $\frac{1}{\sqrt{R}}$.

Para la ejecución del proyecto se crearán dos archivos adicionales, uno llamado **shift_final.v** y **tb_mul_final.v**. El archivo **shift_final.v** se encargará de instanciar cada uno de los archivos anteriores, así como llamar las variables de salida de cada archivo y almacenarla en las variables locales de los archivos que reciben esa información. El archivo **tb_mul_final.v** servirá como cama de prueba, una cama de prueba permite asignar valores a las variables que se utilizarán en el proyecto, así mismo la cama de prueba se realizará al archivo

shift_final.v ya que este asignará los valores de la cama de prueba a cada archivo instanciado. De igual forma la cama de prueba se encargará de imprimir el resultado final.

Para la toma de variables y la impresión de resultados se utilizarán diversas formas de asignación: una, asignando variables en la cama de prueba, y otra, será en archivos de texto.

Las variables que se van asignar en la cama de prueba serán usadas en el archivo **Shift.v** a continuación se muestra que función hará y como serán declaradas.

Clk: Esta variable servirá como señal de reloj para el almacenamiento del registro en el archivo **Shift.v**.

SI: Será el bit adicional que se colocará más a la derecha luego de hacer el corrimiento a la izquierda en el archivo **Shift.v**.

Rst: Representa un "reset" en el almacenamiento del registro, si está en 1 quiere decir que borrará toda la información almacenada limpiando cada uno de los registros de la arquitectura.

En1: Esta variable representa un "enable" que permite guardar la variable in en el registro cuando el bit de **en1** está en 1, si está en 0 el dato que está en in no será almacenado y por lo tanto no se realizará el Shift.

Luego tenemos las variables que serán las que contienen los datos de la semilla (x_n) y el número al que se le desea hallar su aproximación (R).

numA, numB, num C: Serán las variables que almacenarán la semilla en el proyecto.

numD: Es la variable que almacena el número que se desea hallar su aproximación.

La razón de porque usar diversas variables para un mismo número (la semilla) es porque esta información se tomó de un multiplicador convencional y para futuras referencias si se desean hacer otra clase de pruebas en el futuro.

Los datos de dichas variables serán tomados de los archivos de texto **inputa, inputb, inputc** e **inputd** y el resultado de la operación final será almacenada en un archivo llamado **outpute**.

Es muy importante señalar que en este proyecto los datos de la semilla y del número a aproximar tienen el siguiente formato en punto fijo, es decir, en el encabezado de los códigos se colocará lo siguiente:

```
#(parameter DATA_WIDTH = 32,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)
```

Esta información será fundamental ya que permitirá servirnos de guía al momento de colocar y leer resultados.

Luego, cuando se declare una variable ya sea **numA, numB, numC, numD** o cualquier resultado se tiene que colocar lo siguiente.

```
[I_widthA-1 : -(DATA_WIDTH-I_widthA)]
```

Esto nos quiere decir que la información bit por bit se almacenará en un arreglo. Explicando cada declaración de dicho arreglo tenemos:

I_withA-1: Significa que se tomarán 4 bits de parte entera quitando un bit inicial como el signo, de igual forma representa que al momento de realizar el truncamiento de bits se realizará desde la posición 4 del arreglo del resultado de la multiplicación en **mult.v**.

DATA_WITDH: Este dato representa el total de bits que tendrá el arreglo, es decir 32 bits, por lo que cada valor de 1 o 0 se almacenará en 32 espacios.

DATA_WITHD-I_widthA: En este apartado se realiza la resta de ambos valores, dando resultado 27 bits restantes para llenar el arreglo, estos 27 bits nos servirán de guía para identificar los valores decimales en los números a colocar.

Por último tenemos lo siguiente que la declaración del arreglo, colocando los valores del mismo no queda de la siguiente manera:

```
[5-1 : -(32-5)]
```

Esto nos quiere decir que el arreglo será de 32 bits, los valores se empezarán a colocar en el bit 4 y el último bit en la posición -27 y de los 32 bits, 4 serán usados nada más como valores enteros, teniendo el primer bit como signo y los 27 bits restantes serán los decimales.

$$\underbrace{0}_{Bit\ que\ representa\ el\ signo}\quad\underbrace{0000}_{\substack{Bits\ que\ representan\\ la\ parte\ entera}}\quad\underbrace{000000000000000000000000000}_{\substack{Bits\ que\ representan\\ la\ parte\ decimal}}$$

En la parte de arriba podemos ver cómo se van a dividir los bits acorde a como los vamos a representar en los resultados finales.

Es importante tener muy en cuenta esta división de los bits ya que al momento de interpretar los resultados, esta división nos permitirá saber que número estamos colocando y cuál es el resultado que nos arroja el sistema, el programa por sí solo no realiza esta separación, por lo que queda a criterio del usuario interpretar los valores usando esta regla.

En esta regla se utiliza la mayor cantidad de bits para la parte decimal ya que esto nos puede aproximar mucho más al resultado final. Mientras mayor será la resolución de bits en la parte decimal, mayor será la aproximación del resultado de la operación al resultado real del valor a aproximar.

v. Determinación de la semilla a utilizar en cada módulo

La semilla es conocida como el valor inicial de nuestro sistema, es decir, aquel valor que iniciará con la operación de Newton-Raphson y que es de gran ayuda para poder aproximar el valor final del valor esperado.

La semilla tiene que cumplir con que debe ser lo más próxima posible al resultado final para que así el número de iteraciones que se planee hacer para el sistema sea menor y se obtenga un resultado más preciso.

La forma en como determinamos la semilla puede ser ya sea por valores aleatorios o bien utilizando diversas operaciones matemáticas.

Anteriormente se desarrolló algo conocido como la semilla mágica, creado por Silicon Graphics en los años noventa para la creación de uno de los videojuegos más populares de la época, Quake 3 (Beyond3D, 2006). El desarrollo de la semilla mágica surgió como necesidad de simplificar los cálculos de vectores, ya que en una de las operaciones para calcular un vector unitario se requiere del uso de la raíz inversa, cosa que, para los sistemas de esos tiempos, resultaba bastante complicada ya que en ese momento la creación de gráficos tridimensionales en entornos como videojuegos era algo aun prematuro. Sin embargo Silicon Graphics logró encontrar un una semilla capaz de calcular cualquier raíz inversa de manera rápida y precisa utilizando una sola iteración, llamada así la semilla mágica.

Uno de los aspectos importantes de dicha semilla mágica es que realiza un proceso donde encuentra el valor más próximo al resultado esperado, siendo este el factor principal para obtener el resultado final mucho más acertado y con un nivel de error mucho menor. Sin embargo en este trabajo no haremos uso de la semilla mágica como tal, solamente usaremos una semilla muy aproximada al resultado esperado.

Las semillas serán tomadas como los valores más próximos que se puedan tomar de nuestro resultado final y esta serán pasadas a binario, una vez hechas así, acotaremos a los 32 bits que requerimos para expresar el valor y será ingresado a nuestro sistema.

Las semillas serán colocadas en los archivos de texto **inputa, inputb e inputc**.

vi. Realización de los cálculos

En la Figura 13 se puede apreciar como las variables x_n y R son introducidas desde los archivos de texto a través de una cama de prueba, la cual coloca las variables en los códigos

de los archivos mostrados en la figura y cada código realizará la instrucción correspondiente para así obtener un resultado con el algoritmo de Newton-Raphson.

Un aspecto importante a considerar en los archivos donde vayamos a realizar multiplicaciones es que cada vez que realicemos una multiplicación, la longitud de los bits se sumará entre el total de números que se van a multiplicar, es decir, en el código del archivo **mult.v** se sumarán las longitudes de las variables **numA, numB, numC y numD** y ahí se almacenará el valor final, sin embargo al final se tendrá que realizar un truncamiento del resultado para poder expresar el resultado a los 32 bits que esperamos que tenga de longitud. Esto es muy importante a considerar ya que esto puede afectar en el resultado de la multiplicación y así afectar al resultado final del proyecto.

Las operaciones se expresan en bits (valores de ceros y unos) debido a su facilidad de manipulación, por ejemplo la operación en el archivo **shift.v** es una división entre dos, con esto podemos reducir aún más la cantidad de hardware utilizado y además de que no se pierde mucha resolución que si usamos una multiplicación más por 0.5 o bien intentar expresar una división convencional.

vii. Elaboración de los siguientes módulos

Como se mencionó anteriormente, los módulos de raíz y de inverso de un número se tomarán a partir del resultado del inverso de la raíz de un número, sin embargo, se podrán tomar de las formulas originales si así se desea en el futuro.

Lo primero que hay que hacer para realizar la operación de inverso de un número es tomar el proyecto anterior y agregar un archivo más:

viii. Módulo de inverso de un número

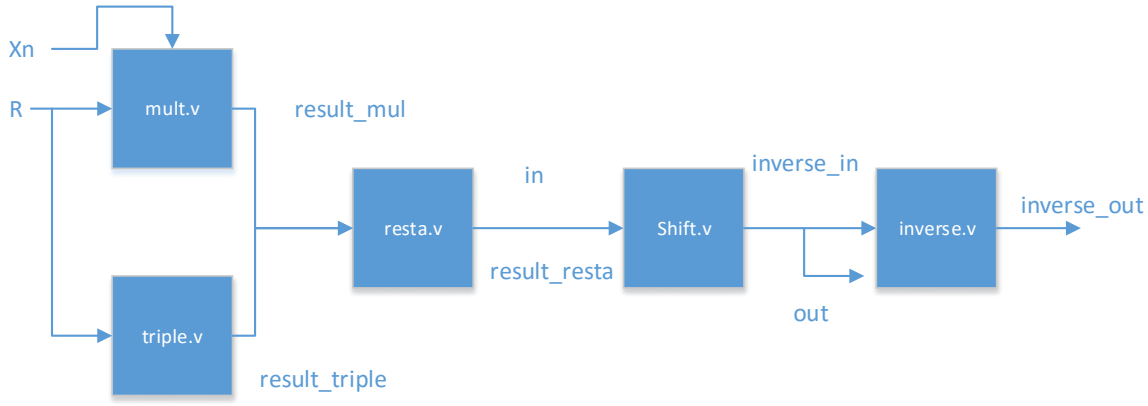

Figura 14. Diagrama de composición de los archivos y variables que comprenden el módulo de inverso de un número.

En la Figura 14 podemos observar que el proceso para realizar el inverso de un número cumple con lo establecido en la $\frac{1}{\sqrt{R}} * \frac{1}{\sqrt{R}} = \frac{1}{R}$ Ec. 31, donde se realizará una multiplicación con el resultado obtenido en **shift.v.**

El código del archivo **inverse.v** se encarga de tomar la variable de salida **out** del código del archivo **shift.v** y lo almacena en su variable local **inverse_in** la cual se elevará al cuadrado multiplicando la variable **inverse_in** con la misma variable **inverse_in**. El resultado se almacenará en una variable única llamada **inverse** la cual almacena el resultado de la multiplicación en arreglo del total de la suma de los bits de los dos **inverse_in** (32 bits), dando un resultado de 64 bits. Al final el resultado final se acota en la variable **inverse_out** para truncar el resultado a los 32 bits que debe tener la variable.

El resultado de la variable **inverse_out** se imprime en el archivo de texto **outpute.txt** donde está igual el resultado de inverso de la raíz de un número.

Como se puede observar de igual forma en la Figura 14, el proceso de inverso de un número es muy similar al del inverso de la raíz de un número solo agregando el archivo **inverse.v,** esto con el fin de poder ahorrar tiempo en a la hora de procesar el resultado, además de que simplificar el código total a utilizar en el proyecto. Sin embargo si desea hacer un

proyecto usando la $\frac{1}{R} = 2x_n - x_n^2 R$ Ec. 28 o bien la Figura 10, es completamente aceptable.

Al final se agregará un archivo más llamado **inverse_final.v** que se encargará de instanciar los demás archivos del módulo de inverso de la raíz de un número y este es llamado a su vez en la cama de prueba **tb_mul_final.v.**

ix. Módulo de raíz de un número

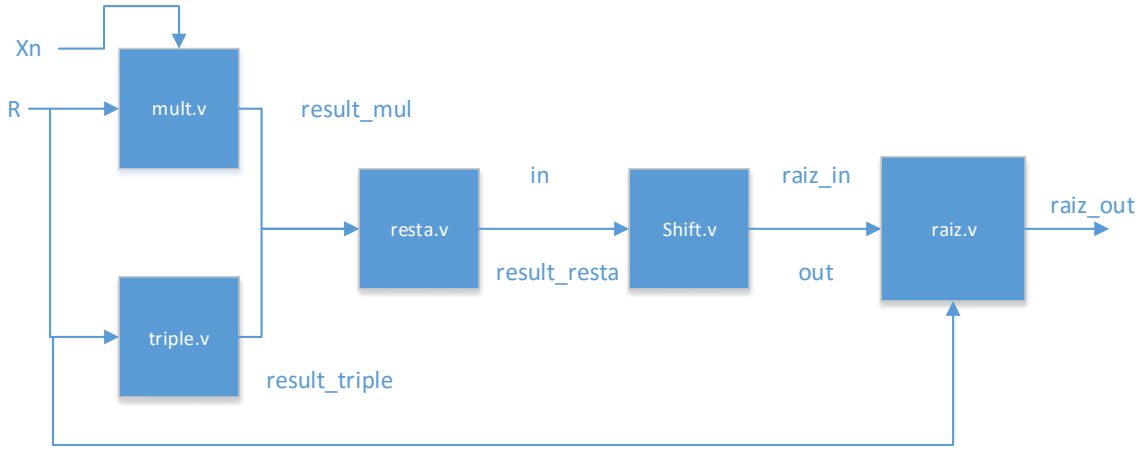

Figura 15. Diagrama de composición de los archivos y variables que comprenden el módulo de raíz de un número.

Para el módulo de raíz de un número se añade un nuevo documento llamado **raiz.v** al módulo de inverso de raíz.

En el código del archivo **raiz.v** se agregan las variables **out** del archivo **shift.v** que pasa a almacenarse la variable local **raiz_in** y la variable **R,** tal y como se muestra en la Figura 15.

Todo esto se rige con base a la $\frac{1}{\sqrt{R}} * R = R * R^{-\frac{1}{2}} = R^{\frac{1}{2}} = \sqrt{R}$ Ec. 29.

En el código del archivo **raiz.v** se multiplicaran las variables **raiz_in** con **R** para así almacenarla en la variable única **root** que servirá para almacenar el resultado de la multiplicación y tendrá una longitud de la suma de las longitudes de las variables **raiz_in** y **R,** seguidamente el valor de **root** se pasa a la variable **raiz_out** donde se trunca el resultado a los 32 bits.

Se creará el archivo **raiz_final.v** que servirá para instanciar los archivos y variables del módulo de inverso de la raíz de un número y este a su vez es llamado en la cama de prueba **tb_mult_final.v.**

9. Pruebas

En este apartado se mostrarán las diversas pruebas que se obtuvieron en los módulos implementados en Hardware mediante Verilog y las diferentes comparaciones con varias semillas propuestas.

i. Tabla de resultados del inverso de la raíz de un número

	Resultado Real	Semilla colocada a 16 bits	Resultado a 16 bits	Error a 16 bits	Semilla colocada a 32 bits	Resultado a 32 bits	Error a 32 bits
$\frac{1}{\sqrt{2}}$	$\frac{2}{\sqrt{2}}$	0.69970703125	0.70703125	0.000075531186548	0.699999995529652	0.707015626132488	0.000091155054060
$\frac{1}{\sqrt{0.5}}$	$\sqrt{2}$	1.40966796875	1.4140625	0.000151062	1.409999996423721	1.414194747805595	0.000018814567505
$\frac{1}{\sqrt{4}}$	$\frac{1}{2}$	0.48876953125	0.49951171875	0.000488281	0.489963345229626	0.49969981610775	0.000300184
$\frac{1}{\sqrt{0.25}}$	2	1.99951171875	2	0	1.945915758609772	1.999863192439079	0.000136808

Tabla 1. Tabla de resultados de inverso de la raíz de un número usando una longitud diferente en la semilla.

Como se puede observar en la Tabla 1, la longitud de bits de la semilla afecta bastante al resultado final. En la mayoría de los casos si la semilla tiene menor longitud de bits ofrece un resultado de mucha menor precisión que uno de 32 bits.

Conocer esta información es de suma importancia ya que esto afecta directamente a los números de bits a usar para la parte decimal (y en algunos casos la parte entera) ya que en diversos sistemas pueden darse casos en los que se tenga que reducir el número de bits y al hacer esto la resolución del resultado final puede variar considerablemente si se desea tener una gran precisión.

En el caso para los cálculos de la tabla se tomó lo siguiente:

Para el cálculo de $\frac{1}{\sqrt{2}}$ se toma una semilla de 16 bits la cual se coloca en los archivos de texto **inputa.txt, inputb.txt e inputc.txt** mientras que en **inputd.txt** se coloca el número 2. Seguidamente los datos almacenados en los archivos de texto, pasan la cama de prueba y arrojan un resultado, el cual es el resultado a 16 bits del sistema. Al obtener un resultado del sistema con el algoritmo de Newton-Raphson se realiza una resta para comparar el valor real con el obtenido y con eso obtenemos el error a 16 bits.

Este mismo proceso se realiza para la semilla de 32 bits y para los demás números en la Tabla 1.

ii. Tabla de resultados del inverso de un número

	Resultado Real	Semilla propuesta	Resultado del módulo	Error
$\frac{1}{2}$	0.5	0.699999995529652	0.499848999083042	0.000151000916958
$\frac{1}{0.5}$	2	1.40999998897314	1.99994678050279	0.0000053219497210
$\frac{1}{4}$	0.25	0.489963345229626	0.249699905514717	0.0000300094485283
$\frac{1}{0.25}$	4	1.94591575860977	3.99130849540233	0.008691504597670

Tabla 2. Tabla de resultados de inverso de un número.

En la Tabla 2 podemos observar las pruebas realizadas en inverso de un número, tenemos la semilla propuesta para el módulo, el resultado de la operación, así como el error que se tiene del resultado final con el resultado real.

Hay que tener en cuenta que la semilla colocada en el módulo fue la semilla usada en inverso de la raíz de un número, esto se debe porque al usar como base el módulo de inverso de la raíz de un número para realizar este módulo, si se cambia la semilla, ésta afectará al resultado de inverso de la raíz de un número y al afectar a este resultado afectará de igual forma al resultado de inverso de un número.

Para poder tener una semilla acorde al módulo, será necesario usar un módulo que no dependa del valor del resultado de inverso de la raíz de un número, en este caso sería un módulo basado en la Figura 10.

Para la Tabla 2 tenemos como ejemplo el cálculo de $\frac{1}{2}$ en el cual colocaremos una semilla de 32 bits en los archivos de texto **inputa.txt, inputb.txt e inputc.txt** mientras que en **inputd.txt** colocaremos el número 2. Seguido de esto, procedemos a iniciar el sistema, el cual tomará los archivos de texto a la cama de pruebas y se imprimirá el resultado en el archivo de texto **outpute.txt.** El resultado de dicha operación utilizando Newton-Raphson la comparamos con el resultado real y así obtenemos el error que el sistema presenta frente el resultado real de la operación.

Este proceso se repite en los demás números de la Tabla 2.

iii. Tabla de resultados de raíz un número

	Resultado Real	Semilla propuesta	Resultado del módulo	Error
$\sqrt{2}$	1.414213562373100	0.699999995529652	1.414000004529950	0.000213557843145
$\sqrt{0.5}$	$\frac{1}{\sqrt{2}}$	1.409999988973140	0.707097373902798	0.000009407283750
$\sqrt{4}$	2	0.489963345229626	1.998799264431000	0.001200735569000
$\sqrt{0.25}$	$\frac{1}{2}$	1.945915758609770	0.499456480145454	0.000543519854546

Tabla 3. Tabla de resultados de raíz de un número.

En la Tabla 3 podemos ver los resultados de las pruebas realizadas al módulo de raíz de un número. Nuevamente hacemos uso de una semilla del módulo inverso de la raíz de un número, debido a que se tiene una dependencia del resultado de dicho módulo para calcular la raíz de un número.

Si se desea tener un resultado utilizando una semilla cercana al resultado de la raíz de un número, será necesario crear un módulo con base a la Figura 11.

Como se pudo observar en Tabla 2 y Tabla 3 el factor de dependencia del módulo inverso de la raíz de un número, hace que el error sea relativamente mayor, sin embargo, este error no es algo que afecte al sistema por completo, si se desea atenuar más este error, se propone o bien ampliar el rango bits de la parte decimal en la semilla y los resultados o utilizar una semilla que se asemeje más a los resultados de los módulos mencionados.

Para la Tabla 3 tenemos como ejemplo el cálculo de $\sqrt{2}$ en el cual colocaremos una semilla de 32 bits en los archivos de texto **inputa.txt, inputb.txt e inputc.txt** mientras que en **inputd.txt** colocaremos el número 2. Seguido de esto, procedemos a iniciar el sistema, el cual tomará los archivos de texto a la cama de pruebas y se imprimirá el resultado en el archivo de texto **outpute.txt.** El resultado de dicha operación utilizando Newton-Raphson la comparamos con el resultado real y así obtenemos el error que el sistema presenta frente el resultado real de la operación.

Este proceso se repite en los demás números de la Tabla 3.

iv. Tabla de resultados con diversos números a evaluar

	Resultados reales			Semilla	Resultados finales			Errores		
	Inverso raíz	Inverso	Raíz		Inverso raíz	Inverso	Raíz	Inverso raíz	Inverso	Raíz
5	$\frac{\sqrt{5}}{5}$	0.2	2.236067977499790	0.447213590145111	0.447213597595692	0.199999995529652	2.236067987978450	-2.09573E-09	4.47035E-09	-1.04787E-08
0.75	$\frac{2\sqrt{3}}{3}$	$\frac{4}{3}$	$\frac{\sqrt{3}}{2}$	1.154700532555580	1.154700540006160	1.333333335816860	0.866025403141975	-1.62691E-09	-2.48353E-09	6.42464E-10
12	$\frac{\sqrt{3}}{6}$	0.083333333333333	$2\sqrt{3}$	0.288675129413605	0.288675136864185	0.083333328366280	3.464101642370220	-2.26937E-09	4.96705E-09	-2.72325E-08
3	$\frac{\sqrt{3}}{3}$	0.333333333333333	1.732050807568880	0.577350266277790	0.577350266277790	0.333333328366280	1.732050798833370	2.91184E-09	4.96705E-09	8.73551E-09
15	$\frac{\sqrt{15}}{15}$	0.066666666666667	$\sqrt{3}\sqrt{5}$	0.258198887109756	0.258198887109756	0.066666662693024	3.872983306646340	2.63741E-09	3.97364E-09	3.95611E-08
13	$\frac{\sqrt{13}}{13}$	0.076923076923077	3.605551275463990	0.277350097894669	0.277350097894669	0.076923072338104	3.605551272630690	2.17946E-10	4.58497E-09	2.8333E-09

7	$\frac{\sqrt{7}}{7}$	0.142857142857143	2.645751311064590	0.377964466810226	0.377964474260807	0.142857141792774	2.645751319825640	-1.25158E-09	1.06437E-09	-8.76105E-09
6	$\frac{\sqrt{6}}{6}$	0.166666666666667	$\sqrt{2}\sqrt{3}$	0.408248290419579	0.408248290419579	0.166666664183140	2.449489742517470	4.42841E-11	2.48353E-09	2.65708E-10
0.125	$2\sqrt{2}$	8	$\frac{\sqrt{2}}{4}$	2.828427121043200	2.828427121043200	7.999999977648250	0.353553384542465	3.70299E-09	2.23517E-08	6.05081E-09
11	$\frac{\sqrt{11}}{11}$	0.090909090909091	3.316624790355400	0.301511339843273	0.301511347293854	0.090909086167812	3.316624820232390	-2.71609E-09	4.74128E-09	-2.9877E-08

Tabla 4. Tabla de resultados de diez números diferentes, para probar el sistema.

En la Tabla 4 se realizó unas pruebas con 10 números diferentes y así conocer mejor la eficacia y exactitud del sistema con diversos valores. Siguiendo con lo visto en las tablas anteriores, la semilla a utilizar en estos números, es bastante similar al resultado del inverso de la raíz, siendo esta la operación principal en el sistema.

La tabla incluye los resultados reales, es decir, los resultados que las operaciones indicadas, seguido de la semilla a usar para hallar las aproximaciones, luego tenemos los resultados finales, estos son los resultados que se obtienen en el sistema luego de ingresar la semilla. Por último tenemos el índice de error en los resultados finales, estos se obtienen de restar el valor real de la operación con el valor final del sistema, esto nos permite saber que tanto fue nuestra aproximación en el sistema y que tan eficiente es el sistema al utilizar la semilla colocada.

Uno de los detalles a destacar en esta tabla es el apartado de errores, ya que en estos podemos ver que la diferencia entre los resultados reales y finales no es mucha y esto se debe a que las semillas favorecen bastante a esto, sin embargo la longitud de los bits de la parte decimal de igual forma influyen bastante en la resolución de los resultados, ya que de extenderse más la cantidad de bits de parte decimal, es muy seguro que el error llegaría a ser tan pequeño que podría semejarse más al resultado final, de igual forma, si se recortan los números de bits en la parte decimal se podría tener un error aún más grande del esperado y por lo tanto se tendrían que realizar más iteraciones para aproximar más el resultado final al real.

Ahora como ejemplo en la Tabla 4 tenemos el cálculo de $\frac{1}{\sqrt{5}}$, $\frac{1}{5}$ y $\sqrt{5}$, para ello colocaremos la semilla basada en el valor $\frac{1}{\sqrt{5}}$ ya que se tomó como base para el sistema el módulo de inverso de la raíz, colocamos la semilla en los archivos de texto y posteriormente ejecutamos el programa para que la cama de prueba realice los cálculos indicados usando la semilla. Seguidamente tomamos los resultados impresos en el archivo de texto **outpute.txt** y los comparamos con el resultado real de cada operación, así obtenemos los errores de cálculo del algoritmo con respecto a los resultados reales.

Este proceso se repite en los demás números de la Tabla 4.

Conclusiones

Se realizaron 3 bloques evaluadores de cada uno de los módulos a efectuar, uno para la raíz cuadrada de un número, otro para el módulo correspondiente al inverso de un número y finalmente un módulo para calcular el inverso de la raíz cuadrada de un número.

Se utilizó el algoritmo de Newton-Raphson para la evaluación de las funciones y así poder llevar acabo las operaciones a nivel hardware de cada una de ellas.

Se propusieron 3 fórmulas basadas en el algoritmo de Newton-Raphson, las cuales sirvieron de base para el diseño e implementación de 3 arquitecturas de hardware de cada uno de los módulos.

Se realizó en Matlab una simulación de las arquitecturas de hardware propuestas en punto flotante y punto fijo. Las arquitecturas fueron diseñadas utilizando el lenguaje de descripción de hardware Verilog y éstas fueron evaluadas introduciendo datos de manera aleatoria. Los resultados generados por la arquitectura fueron comparados con los resultados obtenidos en punto fijo y flotante de los modelos de referencia implementados en Matlab con la finalidad de conocer la exactitud de los resultados calculados.

Se comenta que se tuvo la oportunidad de aprender a programar hardware con el lenguaje Verilog, realizar diseño de hardware, y a documentarse sobre operaciones de cómputo aritmético.

Así mismo, se aprendió a implementar bloques en hardware básicos de electrónica digital, tales como el sumadores, multiplicadores, flip-flop's, registros de memoria y aplicando conocimientos teóricos para el diseño del sistema completo mediante Verilog.

Por otra parte, se aplicó una metodología bottom-top, la cual consistía en realizar partes específicas del sistema, es decir, primero se inició con la creación de multiplicadores y sumadores y poco a poco cada elemento de los módulos se fue uniendo para crear el sistema completo.

Se almacenó en una base de datos de cada una de las arquitecturas de los módulos del sistema, las cuales servirán para futuros proyectos derivados de una problemática similar a la que se plantea en este trabajo de tesis. Así también, se buscó que estas arquitecturas sirvan de base para crear nuevos módulos de cómputo aritmético utilicen el lenguaje de descripción de hardware Verilog.

Para corroborar la exactitud de los módulos implementados, se efectuaron diversas pruebas de sistema y se generaron tablas de información con los resultados obtenidos. Dichas tablas facilitaron observar diversos detalles al respecto del funcionamiento del sistema, así como su efectividad de las semillas propuestas para la resolución de cada módulo.

Finalmente, comento que este trabajo de tesis me permitió explorar técnicas de aproximación numérica para realizar cómputo aritmético a nivel de hardware.

Bibliografía

Amaya-Fernandez, F., & Velasco-Medina, J. (s.f.). *Diseño de la tangente inversa usando el algoritmo CORDIC.* Obtenido de http://www.iberchip.net/iberchip2006/ponencias/75.pdf

Beyond3D. (29 de Noviembre de 2006). *Beyond3D.* Obtenido de http://www.beyond3d.com/content/articles/8/

López Nicolás, J. M. (22 de octubre de 2013). *Gaussianos*. Obtenido de http://gaussianos.com/la-historia-del-metodo-de-newton-raphson-y-otro-caso-mas-de-mala-documentacion-en-el-cine/

Matlab. (s.f.). *MatWorks*. Obtenido de http://www.mathworks.com/help/fixedpoint/ref/fi.html

Navabi, Z. (2006). *Verilog Digital System Design Secound Edition.* McGraw-Hill.

Palacios, F. (s.f.). *Resolución aproximada de ecuaciones: Método de Newton-Raphson*. Obtenido de http://www.eupm.upc.edu/~fpq/ale-hp/modulos/aplicaciones/newton.pdf

Wikipedia. (13 de agosto de 2015). *Wikipedia*. Obtenido de https://es.wikipedia.org/wiki/Relaci%C3%B3n_se%C3%B1al/ruido

Apéndice 1. Verilog

Verilog es un lenguaje de descripción de hardware que fue diseñado en 1984 por Gateway Design Automation. Originalmente el lenguaje era usado como herramienta de evaluación y verificación. Luego de la aceptación inicial del lenguaje por parte de la industria electrónica, se desarrollaron un simulador de fallos, un analizador de tiempo, luego en 1987, una herramienta de síntesis, todo esto creado a partir de este lenguaje.

En 1987 VHDL se convirtió en un estándar de la IEEE como lenguaje de descripción de hardware. En ese momento se empezaron a realizar esfuerzos para popularizar Verilog y en 1990 OVI (Open Verilog International) fue formado y Verilog fue puesto en dominio público.

En 1993 los esfuerzos para estandarizar el lenguaje comenzaron. Éstos dieron resultado en 1995 que fue el año en el Verilog se convirtió en un estándar de la IEEE.

1. Elementos de Verilog

i. Módulos de hardware

El lenguaje de descripción de hardware Verilog (HDL) es usado para describir módulos de hardware y sistemas completos. Por lo tanto, el componente principal del lenguaje, el cual es un *module*, tiene este propósito. La descripción de un módulo de hardware consiste en la palabra **module** la cual le da nombre al módulo, la lista de puertos del módulo, la especificación de las funciones del módulo y la palabra **endmodule.**

```
module [nombre del módulo]
Se declaran las sentencias que contendrá el módulo.
Endmodule
```

ii. Asignación de sentencias

En lugar de describir compuertas primitivas, que son expresiones booleanas para describir la lógica del sistema, Verilog puede usar la sentencia **assign** para colocar resultados de expresiones a varias salidas.

```
assign inv = inverse_in * inverse_in;
```

Su formato es el siguiente: la palabra **assign** seguido de la variable a la que se le va asignar el resultado de la operación seguido del operador "=" donde se colocará la operación que se asignará a la variable.

iii. Instanciación de módulos

Otra forma para describir un componente, es describiendo cada uno de sus sub-componentes e instanciándolos para así unir componentes de un diseño de bajo nivel a un diseño de alto nivel.

```
mul #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Mult2(.numA(numA), .numB(numB), .numC(numC),.numD(numD),
.resul(result_mul));
```

Para instanciar un módulo es necesario llamar a todas y cada una de las variables que contiene el módulo que se instancio y almacenar los valores en las variables del módulo que será el principal, tal como se muestra en el ejemplo de arriba.

iv. Filp-Flop

Los Flip-flops son usados en la parte del diseño de datos por banderas y almacenamiento de datos. Un flip-flop multi bit son considerados como registros, los cuales en Verilog se codifican de manera similar a un flip-flop.

```
`timescale 1ns/100ps

module Flop (reset, din, clk, qout);
     input reset, din, clk;
     output qout;
     reg qout;
     always @ (negedge clk) begin
          if (reset) qout <= #8 1'b0;
          else qout <= #8 din;
     end
endmodule
```

El código de ejemplo mostrado arriba, muestra un flip-flop de 1 bit con *reset* sincronizado.

v. Shift Register

Otra estructura que es usada como componente de datos, es un registro con una o varias capacidades de corrimiento.

El Shift Register se encarga de almacenar un valor de entrada en bits y luego se realiza un corrimiento a la derecha o a la izquierda según sea el caso, luego el resultado se concatena con un valor que ocupa el espacio vació que se quedó en corrimiento.

```
module shift
#(parameter DATA_WIDTH = 16,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)

(    input clk, rst, en, SI,
     input  signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)] in,
     output signed[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
out
);

wire signed [I_widthA-1 : -(DATA_WIDTH-I_widthA)] in;
reg signed  [I_widthOut-1 : -(DATA_WIDTH-I_widthOut)] out;

always @(posedge clk or posedge rst)

     if (rst==1'b1)
          begin
               out <= 16'h0;
          end
```

```
     else if (en==1'b1)
          begin
               out <= { SI, in[I_widthOut-1 : -(DATA_WIDTH-
I_widthOut)+1] };
          end

endmodule
```

En el código Verilog, de arriba podemos observar un corrimiento a la derecha del valor almacenado en la variable *in* y este se concatena con la variable *SI* la cual contiene un valor de cero.

Apéndice 2. Códigos utilizados en Verilog.

i. inputa.txt

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000101100110011001100110011001
00000000000000000000000000000000
00001011010001111010111000010011
00000000000000000000000000000000
00000011111010110111000111100111
00000000000000000000000000000000
00001111100100010011110001001000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000011100100111110010010111000
00000000000000000000000000000000
00001001001111001101001110100010
00000000000000000000000000000000
00000010010011110011010011101000
00000000000000000000000000000000
00000100100111100110100111010001
00000000000000000000000000000000
00000010000100001100101010010100
00000000000000000000000000000000
00000010001110000000001101010100
00000000000000000000000000000000
00000011000001100001001000111100
00000000000000000000000000000000
00000011010001000001011110101110
00000000000000000000000000000000
00010110101000001001111001100110
00000000000000000000000000000000
00000010011010010111111011000111
00000000000000000000000000000000
```

ii. inputb.txt

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000101100110011001100110011001
00000000000000000000000000000000
00001011010001111010111000010011
00000000000000000000000000000000
00000011111010110111000111100111
00000000000000000000000000000000
00001111100100010011110001001000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000011100100111110010010111000
00000000000000000000000000000000
00001001001111001101001110100010
00000000000000000000000000000000
00000010010011110011010011101000
00000000000000000000000000000000
00000100100111100110100111010001
00000000000000000000000000000000
00000010000100001100101010010100
00000000000000000000000000000000
00000010001110000000001101010100
00000000000000000000000000000000
00000011000001100001001000111100
00000000000000000000000000000000
00000011010001000001011110101110
00000000000000000000000000000000
00010110101000001001111001100110
00000000000000000000000000000000
00000010011010010111111011000111
00000000000000000000000000000000
```

iii. inputc.txt

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000101100110011001100110011001
00000000000000000000000000000000
00001011010001111010111000010011
00000000000000000000000000000000
00000011111010110111000111100111
00000000000000000000000000000000
00001111100100010011110001001000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000011100100111110010010111000
00000000000000000000000000000000
00001001001111001101001110100010
00000000000000000000000000000000
00000010010011110011010011101000
00000000000000000000000000000000
00000100100111100110100111010001
00000000000000000000000000000000
00000010000100001100101010010100
00000000000000000000000000000000
00000010001110000000001101010100
00000000000000000000000000000000
00000011000001100001001000111100
00000000000000000000000000000000
00000011010001000001011110101110
00000000000000000000000000000000
00010110101000001001111001100110
00000000000000000000000000000000
00000010011010010111111011000111
00000000000000000000000000000000
```

iv. inputd.txt

00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00010000000000000000000000000000
00000000000000000000000000000000
00000100000000000000000000000000
00000000000000000000000000000000
00100000000000000000000000000000
00000000000000000000000000000000
00000010000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00101000000000000000000000000000
00000000000000000000000000000000
00000110000000000000000000000000
00000000000000000000000000000000
01100000000000000000000000000000
00000000000000000000000000000000
00011000000000000000000000000000
00000000000000000000000000000000
01111000000000000000000000000000
00000000000000000000000000000000
01101000000000000000000000000000
00000000000000000000000000000000
00111000000000000000000000000000
00000000000000000000000000000000
00110000000000000000000000000000
00000000000000000000000000000000
00000001000000000000000000000000
00000000000000000000000000000000
01011000000000000000000000000000
00000000000000000000000000000000

vi. mutl.v

```
module mul
#(parameter DATA_WIDTH = 16,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)

(
     input signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)] numA,
     input signed[I_widthB-1 : -(DATA_WIDTH-I_widthB)] numB,
     input signed[I_widthC-1 : -(DATA_WIDTH-I_widthC)] numC,
     input signed[I_widthD-1 : -(DATA_WIDTH-I_widthD)] numD,
     output signed[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
resul
);

wire signed[(I_widthA+I_widthB+I_widthC+I_widthD)-1:-
((4*DATA_WIDTH)-(I_widthA+I_widthB+I_widthC+I_widthD))] mul;

assign mul = numA*numB*numC*numD;

assign resul = mul[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)];

endmodule
```

v. triple.v

```
module triple
#(parameter DATA_WIDTH = 16,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)

(
     input signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)] numA,
     output signed[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
resul
);

assign resul = 3*numA;

endmodule
```

vi. resta.v

```
module resta
#(parameter DATA_WIDTH = 16,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)

(
     input signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)]
result_mul,
     input signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)]
result_triple,
     output signed[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
resta
);

assign resta = result_triple - result_mul;

endmodule
```

v. Shift.v

```
module shift
#(parameter DATA_WIDTH = 16,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)

(    input clk, rst, en, SI,
     input  signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)] in,
     output signed[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
out
);

wire signed [I_widthA-1 : -(DATA_WIDTH-I_widthA)] in;
reg signed  [I_widthOut-1 : -(DATA_WIDTH-I_widthOut)] out;

always @(posedge clk or posedge rst)

     if (rst==1'b1)
          begin
               out <= 16'h0;
          end
     else if (en==1'b1)
          begin
               out <= { SI, in[I_widthOut-1 : -(DATA_WIDTH-
I_widthOut)+1] };
          end

endmodule
```

vi. Shift_final.v

```
module shift_final

#(parameter DATA_WIDTH = 16,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)

(
     input clk, rst, en1, SI,
     input signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)] numA,
     input signed[I_widthB-1 : -(DATA_WIDTH-I_widthB)] numB,
     input signed[I_widthC-1 : -(DATA_WIDTH-I_widthC)] numC,
     input signed[I_widthD-1 : -(DATA_WIDTH-I_widthD)] numD,
     output signed[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
Shift_result
);

wire[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)] result_mul,
result_triple, result_resta;

mul #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Mult(.numA(numA), .numB(numB), .numC(numC),.numD(numD),
.resul(result_mul));

triple #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Triple(.numA(numA), .resul(result_triple));

resta #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Resta(.result_triple(result_triple),
.result_mul(result_mul), .resta(result_resta));

shift #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Shift(.in(result_resta), .clk(clk), .rst(rst), .en(en1),
.SI(SI), .out(Shift_result));

endmodule
```

vii. inverse.v

```
module inverse
#(parameter DATA_WIDTH = 16,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)

(
     input signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)]
inverse_in,
     output signed[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
inverse_out
);

wire signed[(I_widthA+I_widthB)-1:-((2*DATA_WIDTH)-
(I_widthA+I_widthB))] inv;

assign inv = inverse_in * inverse_in;

assign inverse_out = inv[I_widthOut-1 : -(DATA_WIDTH-
I_widthOut)];

endmodule
```

viii. inverse_final.v

```
module inverse_final
#(parameter DATA_WIDTH = 16,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)

(
     input clk, rst, en1, SI,
     input signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)] numA,
     input signed[I_widthB-1 : -(DATA_WIDTH-I_widthB)] numB,
     input signed[I_widthC-1 : -(DATA_WIDTH-I_widthC)] numC,
     input signed[I_widthD-1 : -(DATA_WIDTH-I_widthD)] numD,
     output signed[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
inverse_result
);

wire[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)] result_mul,
result_triple, result_resta, Shift_result;

mul #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Mult2(.numA(numA), .numB(numB), .numC(numC),.numD(numD),
.resul(result_mul));

triple #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Triple2(.numA(numA), .resul(result_triple));

resta #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Resta2(.result_triple(result_triple),
.result_mul(result_mul), .resta(result_resta));

shift #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Shift2(.in(result_resta), .clk(clk), .rst(rst),
.en(en1), .SI(SI), .out(Shift_result));
```

```
inverse #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Inverse (.inverse_in(Shift_result),
.inverse_out(inverse_result));

endmodule
```

ix. raiz.v

```
module raiz
#(parameter DATA_WIDTH = 16,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)

(
     input signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)]
raiz_in,
     input signed[I_widthD-1 : -(DATA_WIDTH-I_widthD)] numD,
     output signed[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
raiz_out
);

wire signed[(I_widthA+I_widthB)-1:-((2*DATA_WIDTH)-
(I_widthA+I_widthB))] root;

assign root = raiz_in * numD;

assign raiz_out = root[I_widthOut-1 : -(DATA_WIDTH-
I_widthOut)];


endmodule
```

x. raiz_final.v

```
module raiz_final
#(parameter DATA_WIDTH = 16,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5)

(
     input clk, rst, en1, SI,
     input signed[I_widthA-1 : -(DATA_WIDTH-I_widthA)] numA,
     input signed[I_widthB-1 : -(DATA_WIDTH-I_widthB)] numB,
     input signed[I_widthC-1 : -(DATA_WIDTH-I_widthC)] numC,
     input signed[I_widthD-1 : -(DATA_WIDTH-I_widthD)] numD,
     output signed[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
raiz_result
);

wire[I_widthOut-1 : -(DATA_WIDTH-I_widthOut)] result_mul,
result_triple, result_resta, Shift_result;

mul #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Mult3(.numA(numA), .numB(numB), .numC(numC),.numD(numD),
.resul(result_mul));

triple #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Triple3(.numA(numA), .resul(result_triple));

resta #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Resta3(.result_triple(result_triple),
.result_mul(result_mul), .resta(result_resta));

shift #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
     Shift3(.in(result_resta), .clk(clk), .rst(rst),
.en(en1), .SI(SI), .out(Shift_result));

raiz #(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
```

```
     Raiz (.raiz_in(Shift_result), .numD(numD),
.raiz_out(raiz_result));
endmodule
```

v. tb_mult_final.v

```
module tb_mul_final
#(parameter DATA_WIDTH = 32,parameter I_widthA = 5,
parameter I_widthB = 5,parameter I_widthC = 5,
parameter I_widthD = 5,parameter I_widthOut = 5,
parameter Max_time = 1000);

     reg [I_widthA-1 : -(DATA_WIDTH-I_widthA)] numA;
     reg [I_widthB-1 : -(DATA_WIDTH-I_widthB)] numB;
     reg [I_widthC-1 : -(DATA_WIDTH-I_widthC)] numC;
     reg [I_widthD-1 : -(DATA_WIDTH-I_widthD)] numD;
     reg clk;
     reg rst;
     reg en1;
     reg SI;

     wire [I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
Shift_result;
     wire [I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
inverse_result;
     wire [I_widthOut-1 : -(DATA_WIDTH-I_widthOut)]
raiz_result;


   // Otras variables
     integer file_a_in, file_b_in, file_c_in, file_d_in,
file_e_out, StatusI = 0;
     reg flag_evaluation;


shift_final
#(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
uut (
     .clk(clk),
     .rst(rst),
     .en1(en1),
     .SI(SI),
     .numA(numA),
     .numB(numB),
     .numC(numC),
     .numD(numD),
     .Shift_result(Shift_result)
);
```

```
inverse_final
#(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
uut2 (
     .clk(clk),
     .rst(rst),
     .en1(en1),
     .SI(SI),
     .numA(numA),
     .numB(numB),
     .numC(numC),
     .numD(numD),
     .inverse_result(inverse_result)
);

raiz_final
#(.DATA_WIDTH(DATA_WIDTH), .I_widthA(I_widthA),
.I_widthB(I_widthB), .I_widthC(I_widthC),.I_widthD(I_widthD),
.I_widthOut(I_widthOut))
uut3 (
     .clk(clk),
     .rst(rst),
     .en1(en1),
     .SI(SI),
     .numA(numA),
     .numB(numB),
     .numC(numC),
     .numD(numD),
     .raiz_result(raiz_result)
);


initial begin
     numA = 0;
     numB = 0;
     clk = 0;
     SI = 0;
     rst = 0;
     en1 = 0;
     #2 rst = 0;
     #2 rst = 1;
     #2 rst = 0;
     #1 en1 = 1;
end
```

```
always
     begin
     #1 clk <= ~clk;
     end

     initial
          begin
          #Max_time $finish; //Se delimita el tiempo máximo
de la simulación.
          end



// abriendo y creando los archivos de simulacion
 initial
      begin
         flag_evaluation = 1'b0;
           file_a_in  = $fopen("inputa.txt","r");
           file_b_in  = $fopen("inputb.txt","r");
           file_c_in  = $fopen("inputc.txt","r");
           file_d_in  = $fopen("inputd.txt","r");
           file_e_out = $fopen("outpute.txt","w");

      end

// poniendo los valores de entrada "a"
 initial
    begin
          while(!$feof(file_a_in))
               begin
                flag_evaluation = 1'b1;
                @(negedge clk)
                    begin
                              StatusI = $fscanf(file_a_in,"%b
\n",numA);
                    end
           end
          $fclose(file_a_in);
          $finish;
      end


// poniendo los valores de entrada "b"
 initial
    begin
```

```
          while(!$feof(file_b_in))
               begin
                @(negedge clk)
                    begin
                              StatusI = $fscanf(file_b_in,"%b
\n",numB);
                    end
           end
          $fclose(file_b_in);
          $finish;
      end

// poniendo los valores de entrada "c"
 initial
    begin
          while(!$feof(file_c_in))
               begin
                @(negedge clk)
                    begin
                              StatusI = $fscanf(file_c_in,"%b
\n",numC);
                    end
           end
          $fclose(file_c_in);
          $finish;
      end

// poniendo los valores de entrada "d"
 initial
    begin
          while(!$feof(file_d_in))
               begin
                @(negedge clk)
                    begin
                              StatusI = $fscanf(file_d_in,"%b
\n",numD);
                    end
           end
        $fclose(file_d_in);
          #10
          flag_evaluation = 1'b0;
          #10
          $fclose(file_e_out);
      end

// escribiendo datos de salida "c"
// esta etapa la controla la senal flag_evaluation
```

```
always@(negedge clk)
begin
     if(flag_evaluation == 1'b1)
          begin
               $fwrite(file_e_out,"%h %h %h %h %b %b %b
\n",numA, numB, numC, numD, Shift_result, inverse_result,
raiz_result);
          end
end

endmodule
```