Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica

Implementación en Verilog de Unidad de Generacion de Rayos para GPU Theia.

Por:

Josué David Vargas Amador

Ciudad Universitaria "Rodrigo Facio", Costa Rica

Diciembre de 2015

Implementación en Verilog de Unidad de Generacion de Rayos para GPU Theia.

Por:

Josué David Vargas Amador

IE-0499 Proyecto eléctrico

Aprobado por el Tribunal:

MSc. Diego Valverde Garro
Profesor guía

MSc. Carlos Duarte Martínez
Profesor lector

MSc. Rodolfo Brenes Fernández
Profesor lector

Índice general

Índice de figuras

Índice de tablas

1 Introducción

1.1 Justificación

Los sistemas computacionales actuales poseen, dentro de su arquitectura módulos de hardware especializados llamados Unidades de Procesamiento Gráfico (GPU, por sus siglas en inglés) encargados de acelerar el proceso de representación de objetos tridimensionales en la pantalla del computador.

Las unidades de procesamiento gráfico permiten la visualización de objetos mediante el cálculo de las primitivas que conforman el modelo abstracto de las imágenes. Las GPU implementan distintos algoritmos de representación gráfica, entre estos, uno es el algoritmo de Ray Casting.

El algoritmo de Ray Casting genera vectores (rayos) normalizados desde la perspectiva del usuario y calcula la intersección de los rayos con los objetos del escenario, además colabora con la formación de los colores, con la finalidad de crear las imágenes mostradas en pantalla.

Entonces los cálculos para la representación de objetos visuales en una GPU de tipo raycasting requieren de una arquitectura interna capaz de la generación de vectores (rayos) normalizados, para luego usar estas estructuras de datos en los módulos dedicados a la intersección de rayos. La generación de rayos requiere de instrucciones capaces de realizar cálculos aritméticos como multiplicaciones, sumas y restas, así como operaciones especializadas que permitan aproximar los valores de raíces cuadradas, por lo que el diseño lógico de una unidad dedicada facilitaría el proceso de creación rayos y permitiría añadir flexibilidad y modularidad al diseño de todo el GPU.

Dentro de las referencias encontradas se hallan proyectos de hardware relacionados al diseño de arquitecturas de trazado de rayos que implementan unidades de generación de rayos propias como SaarCor de la Universidad de Saarland (Schmittler et al. (2004)) y RayCore de la Universidad de Sejong (Nah et al. (2014)).

En el caso de la GPU de tipo raycasting Theia, las especificaciones arquitectónicas indican la necesidad de una Unidad de Generación de Rayos (RGU, por sus siglas en inglés). La RGU debe poseer un conjunto de instrucciones necesarias para el cálculo de la normalización de vectores tridimensionales empleados en las siguientes etapas de funcionamiento del GPU.

1.2 Alcances del proyecto

La GPU Theia se encuentra en su tercera iteración, y en esta etapa tiene dos módulos principales dentro de su descripción de RTL en el lenguaje Verilog: la unidad de generación de rayos normalizados (RGU) y el módulo de intersección de rayos de tipo AABB (siglas en inglés de Axis Aligned Bounding Boxes).

El propósito del presente proyecto es la implementación conductual de una RGU cuya descripción posea las instrucciones necesarias para el funcionamiento apropiado de la generación de rayos normalizados. Estas instrucciones deben ser capaces de proveer la información necesaria para programar el módulo de la RGU de manera que permita el cálculo aproximado del inverso de la raíz cuadrada empleando el método iterativo para aproximación de raíces.

A partir del diseño inicial se deben buscar las condiciones apropiadas que permitan obtener una Unidad de Generación de Rayos que produzca resultados con porcentajes de error promedio inferior a 1 porciento respecto al valor real del inverso de las raíces cuadradas de los vectores. Posterior a esto se debe plantear un ambiente de verificación funcional que permita afirmar que el módulo RGU está cumpliendo con su papel dentro de la arquitectura y que puede generar la información requerida por los módulos de intersección de rayos.

1.3 Objetivos

Objetivo general

Desarrollar el modelo por comportamiento en Verilog de una Unidad de Generación de Rayos de un GPU tipo ray casting.

Objetivos específicos

Desarrollar el modelo por comportamiento en Verilog de una Unidad de Generación de Rayos de un GPU tipo ray casting.

- Investigar bibliografía sobre el mecanismo generación de rayos.
- Definir el mecanismo de generación de rayos normalizados en el GPU.
- Verificar el comportamiento funcional de la Unidad de Generación de Rayos en el GPU.

1.4 Metodología

1. Se procederá a investigar los conceptos fundamentales de la arquitectura de la GPU, el algoritmo de ray casting y sobre los posibles mecanismos de la generación de rayos normalizados.

2. Se buscará la implementación final de la arquitectura interna de la RGU de modo que contenga las instrucciones necesarias para la normalización.

3. Se simulará la ejecución del código en la RGU para generar los rayos normalizados necesitados por los módulos de intersección de rayos de tipo AABB.

4. Se verificará el comportamiento funcional del módulo RGU con la finalidad de establecer un marco de referencia para la futura validación del resto de la versión actual del GPU Theia.

1.5 Desarrollo

Este proyecto se estructura por medio de capítulos, cada uno tiene como tarea aclarar los siguientes tópicos:

1. Capítulo I: Introducción. Muestra la justificación del proyecto, los alcances y limitaciones, los objetivos y la metodología que permite cumplir los mismos.

2. Capítulo II: Antecedentes y Marco Teórico. Introduce al lector conceptos claves de arquitectura de unidades de procesamiento gráfico, algoritmo de raycasting, y plantea los casos de proyectos donde se han implementado chips.

3. Capítulo III: Implementación final de la unidad de generación de rayos. Aquí se describe la arquitectura final de la unidad, así como el método empleado usando las instrucciones de ésta para implementar la unidad.

4. Capítulo IV: Prueba de verificación funcional. Se comprueba la funcionalidad del módulo conductual de lenguaje Verilog por medio de un ambiente de verificación apropiado.

5. Capítulo V: Conclusiones y recomendaciones. Se muestran posibles resultados del proyecto y reflexiones sobre el futuro del proyecto.

2 Marco Teórico

2.1 GPU

Definición

Las unidades de procesamiento gráfico se encargan de rápidamente renderizar (representar) objetos 3D en forma de píxeles en la pantalla de la computadora, típicamente, por medio de arquitecturas de hardware basadas en la técnica de rasterización. A continuación se mencionan las etapas principales del pipeline de gráficos convencional [Akenine-Moller et al. (2008)]:

1. El programa de usuario proporciona los datos al GPU en la forma de primitivas como puntos, líneas y polígonos que describen la geometría 3D.

2. Etapa geométrica: las primitivas geométricas son procesadas en base a los vértices y son transformados de coordenadas 3D a triángulos 2D en la pantalla..

3. Etapa de rasterización: en esta etapa se dibuja una imagen mediante el uso de los datos anteriormente generados así como de los cálculos computacionales por píxel. La salida es un conjunto de píxeles donde cada píxel posee sus propios atributos (color, sombras, etc).

Los conjuntos de datos muy grandes que deben ser visualizados en tres dimensiones normalmente son creados usando representaciones de superficies mediante el dibujo de primitivas geométricas que crean mallas poligonales (en la mayoría de casos son mallas triangulares), pero las técnicas convencionales al usarse en el renderizarizado de datos volumétricos producen pérdidas en la visualización. Las técnicas de renderización de volumen tienen más información que los métodos de renderización por superficie pero poseen una mayor complejidad y mayores tiempos de renderización [Akenine-Moller et al. (2008)].

2.2 Raycasting

A continuación, se presentan detalles acerca del algoritmo de raycasting en el cual se basa el GPU Theia para su funcionamiento.

Definición

El algoritmo de raycasting funciona haciendo cálculos a un píxel a la vez, y para cada píxel la tarea básica es encontrar el objeto que es observado en la posición correspondiente a ese píxel en la imagen, o sea parte del observador hacia los objetos a visualizar contrario al método por rasterización. Se puede decir que cada píxel ve en una dirección distinta y cualquier objeto que es observado por un píxel debe intersectar el rayo proveniente desde el punto de vista de la cámara. El objeto esperado es aquel que es intersectado primero por el rayo más cercano a la cámara. Una vez que el objeto es encontrado, se emplea el punto de intersección, la superficie normal, y alguna otra información del objeto para definir el color de cada píxel [Shirley y Marschner (2009)].

Entonces se puede decir que un algoritmo de raycasting tiene tres partes básicas:

1. Generación de rayo: donde se calcula el origen y la dirección de cada rayo (vector) del píxel correspoendiente en la vista de la cámara.

2. Intersección de rayo: donde se determina el objeto más cercano en la intersección del rayo proveniente de la cámara.

3. Shading: donde se calcula el color del píxel basado en los resultados de la intersección de rayos.

Con el objetivo de generar rayos, primero se necesita una representación matemática de un rayo. Un rayo en realidad es solo un punto de origen y una dirección propagación. El rayo consiste en una línea paramétrica en 3D que va desde el ojo llamado punto e, hasta un segundo punto s ubicado en el plano de la imagen. La ecuación del rayo viene dada por::

$$p(t) = e + t(s - e) \tag{2.1}$$

Esta fórmula implica que se empieza en el punto e y se avanza a través del vector s-e hasta llegar al punto p. Valores negativos de t implican que se encuentra el rayo detrás del ojo.

Un seudocódigo sobre el algoritmo de raycasting se muestra se muestra abajo en el seudocódigo 2.

En términos generales se puede afirmar que la técnica de raycasting evalúa el color de cada pixel en la imagen al disparar un rayo a través de la escena desde la posición del observador. Si el rayo intersecta el volumen, el color del pixel es calculado muestreando los datos a lo largo del rayo en un número finito de posiciones en el volumen y combinando cada resultado en uno solo. Este método tiene una limitación al ejecutarse en los CPUs: para vólumenes de datos grandes el tiempo de renderización para una sola imagen es muy

Algorithm 1 Algoritmo de Raycasting

```
1: procedure RAY-CAST
2:     for cada pixel do
3:         Construya un rayo desde el ojo
4:         for cada objeto en la escena do
5:             Encuentre la intersección con el rayo
6:             Guarde esta intersección si es la más cercana
```

alto para visualización en tiempo real [Marques et al. (2009)]. En general el algoritmo de ray-casting, se enfoca en las técnicas de mayas para representar superficies para representar objetos.

2.3 Arquitecturas con unidades de generación de rayos

Existen en la literatura pocas referencias a unidades de raycasting o de ray tracing (variante del algoritmo de ray casting que genera rayos secundarios por medio de la recursión en el punto de incidencia del rayo original) que mencionen explícitamente dentro de su arquitectura la implementación de unidades de generación de rayos, las principales referencias son dos proyectos de hardware provenientes de la Universidad de Saarland (SaarCOR y DRPU) y RayCore de la Universidad de Sejong (Nah et al. (2014)). En los artículos no se mencionan detalles de cómo se implementaron unidades de generación de rayos, solo se hablan de ellas de forma muy general.

En el caso de la GPU de tipo raycasting Theia, las especificaciones arquitectónicas indican la necesidad de una Unidad de Generación de Rayos (RGU, por sus siglas en inglés). La RGU debe poseer un conjunto de instrucciones necesarias para el cálculo de la normalización de vectores tridimensionales empleados en las siguientes etapas de funcionamiento del GPU.

SaarCOR

SaarCOR es el nombre de la unidad de ray tracing que fue diseñado para un chip de uso específico que está conectado por medio de un sistema de bus ”host”(huésped) a otros chips que están en la misma placa de computadora. SaarCOR está dividido en tres unidades principales: la unidad de generación de rayos y de ”shading”(RGS), el núcleo de ray tracing (RTC) y una unidad de manejo de acceso a memoria (RTC-MI) [Schmittler et al. (2004)].

DRPU

DRPU es el diseño y la implementación de un ASIC para procesamiento de ray tracing que posee capacidades de programabilidad similares a un GPU convencional en la universidad de Saarland [Woop et al. (2006)]. La arquitectura consiste en dos partes principales: Unidades de Ray Casting (encargadas de las estructuras espaciales) y un Procesador de "Shading"(encargado de hacer labores de sombreado y de generación de rayos).

RayCore

RayCore es el diseño y la implementación de una unidad de ray tracing para dispositivos móviles y de bajo consumo por parte de miembros de la Universidad de Sejong en Corea del Sur. Dentro de RayCore existen dos unidades principales: una Unidad de Creación de Árboles (TBU) y una Unidad de Ray-Tracing (RTU). Dentro de RTU hay una unidad de generación de rayos tanto primarios como secundarios definidos por la unidad Set-Up y por la unidad de "shading", respectivamente [Nah et al. (2014)].

2.4 Arquitectura de TheiaV3

Introducción a TheiaV3

TheiaV3 es la tercera iteración del GPU multinúcleo de tipo raycasting Theia. El proyecto Theia es un proyecto de la modalidad Open Source (Código Libre) para experimentar con el hardware gráfico 3D (Valverde (2015)].

El principal objetivo del proyecto Theia es proveer un ambiente Open Source incluyendo un código RTL funcional, un ambiente de pruebas y un lenguaje libre de alto nivel y un compilador para programar a Theia. En el diseño de Theia se plantea, por asuntos de simplicidad de diseño, que utilice el formato de sus datos en punto fijo en vez de emplear punto flotante.

El hardware de Theia es descrito usando código RTL escrito en Verilog 2001. Para realizar una simulación completa del código RTL, se necesita tanto un conjunto de archivos que representan los parámetros de entrada, así como el código de usuario en representación binaria.

Descripción general del sistema

Theia es una unidad de procesamiento gráfico (GPU) multinúcleo, que tiene distintos bloques que interactúan entre ellos con la finalidad de renderizar cuadros de imágenes.

En la figura se muestran los principales bloques funcionales de Theia así como la memoria principal que se encuentra en el exterior del GPU. La memo-

ria principal es una memoria de tipo RAM que es empleada para almacenar las variables geométricas, el código, entre otros detalles.

Unidad de Generación de Rayos

La Unidad de Generación de Rayos (RGU por sus siglas en inglés) es un módulo de hardware encargado de la generación de las estructuras de datos que representan los rayos que son enviados a los otros módulos encargados de la intersección de los mismos. La RGU tiene un conjunto limitado de operaciones aritméticas así un conjunto de instrucciones propias orientadas a la generación de los rayos por lo que carece de instrucciones de control de flujo. En la figura 2.1 se puede ver los bloques principales del RGU donde los azules representan memorias y los verdes bloques de lógica, además a la par de cada flecha se encuentra el nombre que correspondería a ese flujo funcional.

2.5 Métodos de normalización

La Unidad de Generación de Rayos de TheiaV3 debe realizar la operación del inverso de la raíz cuadrada con el objetivo de normalizar el rayo (vector) visto desde el observador, para lo cual se debe implementar dentro de la unidad un mecanismo que permita calcular una aproximación por medio de instrucciones aritméticas simples y de una forma rápida. A continuación se describen los métodos posibles.

Existen varios tipos de posibles algoritmos para el cálculo de raíces cuadradas pero en realidad para la implementación en microprocesadores hay pocos, y estos caben en dos categorías: multiplicativos y sustractivos. Los métodos multiplicativos normalmente se implementan en hardware junto con el multiplicador de la unidad de punto flotante y permiten realizar operaciones rapidamente, y por otro los métodos sustractivos emplean hardware dedicado a estas operaciones, lo cual incrementa la latencia y los vuelve técnicas más lentas (Soderquist y Leeser (1997)].

Métodos multiplicativos

Los algoritmos multiplicativos se suelen emplear para realizar cálculos estimados de raíces cuadradas usando iteraciones a partir de un estimado inicial. Emplear este tipo de técnicas reducen la ejecución de una operación de raíz cuadrada en una serie de multiplicaciones, sustracciones, y corrimientos de bits. Además vale la pena resaltar que estos métodos numéricos convergen cuadráticamente, lo cual implica que un estimado inicial apropiado preciso proporcionará resultados más precisos por cada iteración. Las técnicas empleadas frecuentemente en microprocesadores son los métodos de Newton-Raphson

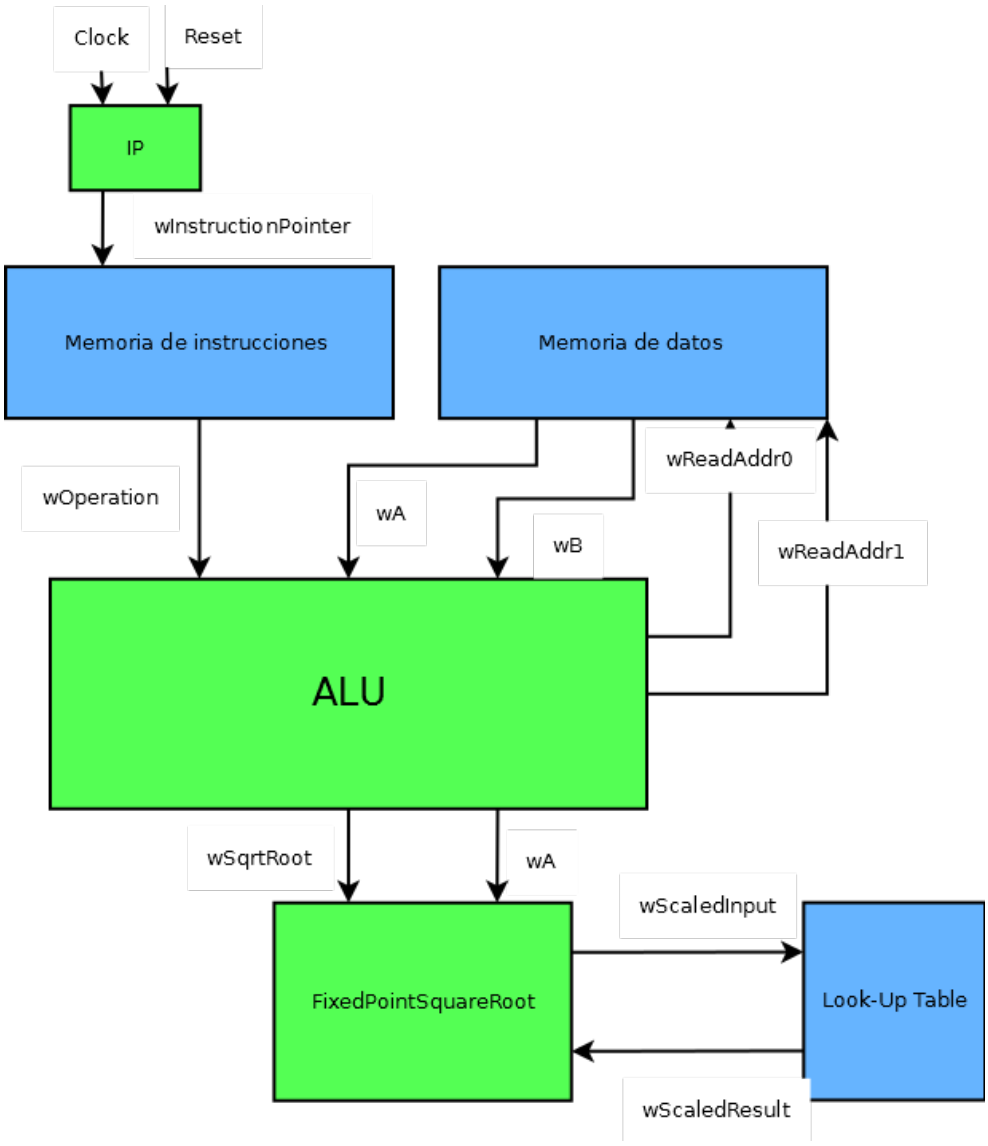

Figura 2.1: Diagrama de los bloques funcionales del RGU

y de Goldschmidt, donde ambos métodos comparte muchos detalles en común pero se diferencian en el orden en que realizan las operaciones (Soderquist y Leeser (1997)].

Newton-Raphson

El método de Newton-Raphson es un método iterativo de convergencia cuadrática que emplea multiplicaciones sucesivas para aprovechar las capacidades de multiplicación rápida de los procesadores contemporáneos [Schulte y Wires (1999)].

Al ser Newton-Raphson un método plenamente iterativo, con el objetivo de reducir las iteraciones que se realizan en los cálculos se suele emplear tablas de hardware las cuales se emplean como punto de partida en el proceso de iteraciones.

El método de Newton-Raphson se fundamenta encontrar la mejor aproximación posible al valor de las raíces o ceros de una función F(x) donde $F(x) = 0$.

Entonces mediante el método de Newton-Raphson si se inicia iterando a partir de un valor X_0 aproximado al valor deseado, entonces se tiene la expresión 2.2.

$$\begin{aligned} f'(x_0) &= \frac{f(x_0)}{x_0 - x_1} \\ x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} \end{aligned} \tag{2.2}$$

Aplicando la definición del método de Newton-Raphson a la ecuación $x^{-2} - S = 0$ donde la raíz es $\frac{1}{\sqrt{S}}$ se obtiene la expresión (2.3):

$$R_{i+1} = R_i(3 - SR_i^2)/2 \tag{2.3}$$

donde S es el valor de la base de la raíz y R_i el valor del cálculo de la raíz cuadrada en la iteración i. El valor de la iteración 0 es el valor que debe salir de las tablas de aproximación.

Goldschmidt

El método de Goldschmidt para división y aplicada también para raíces cuadradas, surgió de la tesis de graduación de maestría de Ingeniería Eléctrica del MIT por parte de Robert Goldschmidt en 1964 [Goldschmidt (1964)]. Esta técnica se basa en la aproximación de la raíz cuadrada por medio de productos sucesivos donde si b_0 es el valor de la base de la raíz cuadrada se busca que se cumpla que $b_n = b_0 Y_0 Y_1 ... Y_n = 1$.

El método de Goldschmidt es ideal para aplicaciones que implementan de forma separada la multiplicación de la suma y en general se emplea en multiplicadores en pipeline. Un detalle de este método es que

Las ecuaciones (2.4) y (2.5) son necesarias para el cumplimiento de este método numérico. El método de Goldschmidt, contrario al de Newton, no es autocorregible lo cual implica que existan errores acumulados [Markstein (2004)].

$$b_i = b_{i-1} Y_{i-1}^2 \tag{2.4}$$

$$Y_i = (3 - b_i)/2 \tag{2.5}$$

2.6 Punto fijo sin signo

Una palabra de N-bits cuando es representada en la forma de un número racional en punto fijo, puede tomar los valores dados por el subconjunto P pertenecientes a los racionales no negativos como se observa en la ecuación (2.6).

$$P = \{p/2^b \quad | \quad 0 \leq p \leq 2^N - 1, \quad p \in \mathbb{Z}\} \tag{2.6}$$

En la ecuación anterior se tiene que P contiene 2^N elementos. Además la nomenclatura P(a,b) representa el subconjunto P, suponiendo que $a = N - b$.

Por otra parte, el valor de un número binario de N-bits X, perteneciente al subconjunto P está dado por la ecuación (2.7).

$$X = (1/2^b) \sum_{n=0}^{N-1} 2^n x_n \tag{2.7}$$

donde X_n representa el bit n del número X. El rango de números que puede tomar como representación el número X es de 0 hasta $(2^N - 1)/2^b = 2^a - 2^{-b}$.

La representación binaria de un número X en punto fijo de 6-bits donde la coma está ubicada a 2 bits a la derecha, o sea $b = 2$, tiene la forma de $x_3 x_2 x_1 x_0 x_{-1} x_{-2}$. Por ejemplo en el diseño de Theia se emplea el punto fijo, el cual emplea la coma decimal en el bit 17 para la representanción de los números.

2.7 Verificación funcional

Generalidades

Un ambiente de verificación funcional modela el universopara el diseño y debe soportar toda las acciones que pueden ocurrir sobre él. Este ambiente se conoce como banco de pruebas o "test bench" [Wile et al. (2005)].

Un ambiente básico de verificación consiste en las siguientes partes:

- Diseño bajo verificación (DUV).
- Componente de estímulo.
- Componente de monitoreo.
- Componente de chequeo.
- Scoreboard.

Componente de estímulo

Esta parte del banco de pruebas manipula las entradas del DUV ("driver") e imita el comportamiento de entidades vecinas. La generación de estímulos no deben restringirse a lo que el DUV es capaz de recibir; de esta forma se estresa el DUV y se pueden encontrar condiciones límite que solo se pueden ver luego de millones de ciclos una prueba a nivel de sistema. Manipula las entradas del DUV ("driver") [Wile et al. (2005)].

Componente de chequeo

La componente de chequeo (çhecker") es un tipo especial de monitor que solo colecta salidas del DUV encargado de validar que una funcionalidad del diseño se comporta de forma correcta [Wile et al. (2005)]. El checker necesita entender el estimulo para predecir el resultado funcional y comparar el resultado actual. El checker tiene que revisar que se cumplan los siguientes puntos:

- Todas las respuestas sean recibidas.
- Todas las salidas corresponden al valor esperado.
- Que no haya actividad sin estimulo.

Scoreboard

Un scoreboard es una locación temporal que contiene información que el checker puede requerir.

Existen dos formas en que se puede establecer la relación entre el checker y el scoreboard:

- El checker contiene el modelo de referencia y el scoreboard se encarga de examinar las entradas para almacenar la información necesaria cuando una transacción ocurre. El checker utiliza el modelo de referencia para transformar la información y comparar el resultados esperado con el resultado actual.
- En el segundo método, el scoreboard tiene el modelo de referencia y calcula los valores esperados con base en los estímulos. Cuando el checker observa ciertos eventos en el DUV, llama al scoreboard para recibir el valor esperado y comparar.

Tipos de checkers

El trabajo de los "checkers" es asegurar que el DUV se comporta correctamente con base en el estimulo. Existen 4 fuentes principales de checkers:

1. Las entradas y salidas del diseño.
2. El contexto del diseño.
3. Las reglas de micro arquitectura del diseño.
4. La arquitectura del diseño.

Las entradas y salidas del diseño

Cualquier bug en el diseño en algún momento se debe manifestar a las salidas del diseño. El código de chequeo utiliza las entradas para predecir las salidas. Una inconsistencia ocurre cuando los datos provenientes del Diseño bajo Verificación no coinciden con los datos de la componente chequeo.

El contexto del diseño

Cuando se verifica el Lenguaje de Descripción de Hardware (HDL) a bajo nivel es importante para el ingeniero verificador comprender el diseño entender la funcionalidad a alto nivel (por contexto). El ingeniero encargado de verificación debe conocer el proyecto a gran escala aún cuando solo deba encargarse de una específica sección del diseño.

Las reglas de micro arquitectura del diseño

Los verificadores crean muchos de los checkers a partir de propiedades basadas en la microarquitectura o las estructuras internas del diseño, por lo que los ingenieros en verificación deben entender el interior del diseño.

Se utiliza la especificación de la microarquitectura para definir propiedades como:

- Estados inválidos.
- Transiciones invalidas.
- Datos inválidos.
- Temporización incorrecta de señales de control.
- Tamaño de buffers y colas.

La arquitectura del diseño

Mientras la microarquitectura define las estructuras que componen el diseño, la arquitectura dicta como debe actuar el diseño desde una especificación de alto nivel encargada protocols, unidades de procesamiento programable y estruturas del sistema en general.

3 Desarrollo de la aplicación

3.1 Generalidades

En esta sección se presentan generalidades acerca de la implementación del algoritmo de generación de rayos y de las operaciones necesarias para generar los vectores en el dominio del espacio.

Vectores normalizados

Los vectores normalizados en el espacio están dados por un punto Y que se dirige a un punto X, por lo cual se sabe que X-Y es la magnitud sin normalizar del vector, por lo que se necesita posteriormente dividir entre la raíz cuadrada del valor absoluto X-Y, para así lograr obtener el vector normalizado que requiere la Unidad de Generación de Rayos (RGU) a su salida.

$$F = \frac{X - Y}{\sqrt{|X - Y|}} \tag{3.1}$$

Por la ecuación (3.1) se puede apreciar que se necesita obtener el inverso de la raíz cuadrada de la resta de los dos puntos en el espacio para lo cual se necesitaría de la capacidad de calcular la raíz cuadrada de un vector, dado lo anterior lo que sigue es definir las operaciones mínimas para obtener el inverso de la raíz cuadrada dentro de un intervalo de valores.

Elección del Newton Raphson

Dentro de los métodos multiplicativos para realizar el cálculo del inverso de la raíz cuadrada se tenían dos métodos principales: el método de Newton-Raphson y el método de Goldschmidt.

De estos dos métodos empleados en la aproximación de divisiones se puede observar que el método de Goldschmidt a la hora de adaptarse para el cálculo de la raíz cuadrada adquiere casi la misma forma que el método de Newton-Raphson adaptado a las raíces cuadradas, además que el método de Goldschmidt está diseño para calcular dos iteraciones (el numerador y el denominador) por lo que está más orientado a estructuras que usen de manera intensiva pipeline como la Unidades de Punto Flotante.

Por otra parte otros métodos llamados sustractivos como el método SRT implican el uso de muchas estructuras de hardware especializadas encargadas

de realizar múltiples corrimientos para la correcta aproximación de las raíces por lo que además resulta ser un método lento en comparación con los métodos multiplicativos.

Por todo lo anterior el método de Newton-Raphson fue elegido entre las opciones existentes.

3.2 Operaciones con el método de Newton Raphson

El método de Newton-Raphson para la aproximación del inverso de la raíz cuadrada entonces se puede desglosar en dos partes:

1. Obtención de una aproximación de la raíz cuadrada
2. Iteración sobre la aproximación de la raíz cuadrada

Se necesita un aproximación muy precisa que facilite la convergencia rápida al valor del inverso de la raíz cuadrada por lo cual se guardan algunos valores en una memoria (LookUp Table) que permitan un valor inicial lo suficientemente robusto como evitar múltiples iteraciones innecesarias que implicarían varios ciclos de reloj.

Por medio de iteraciones sobre la aproximación obtenida de una tabla se puede mejorar tal valor del inverso de la raíz, empleando básicamente solo tres operaciones:

1. Resta
2. Multiplicación
3. Un corrimiento hacia la izquierda

Dado que la implementación del RGU se realiza en un FPGA que posee al interior módulos de multiplicación incrustados es innecesario diseñar un sistema de multiplicación convencional en la descripción en Verilog.

Con respecto al corrimiento hacia la izquierda esta operación corresponde al dos en el denominador que se halla en la ecuación (2.3).

3.3 Consideraciones del Punto Fijo

La multiplicación de números enteros no presenta mayor complicación, pero el incoveniente proviene cuando se intentan realizar operaciones en formato de punto fijo.

En el formato de punto fijo se define una cantidad de bits que representan la parte entera, y la restante cantidad de bits representan la parte fraccionaria del número (esta parte se corresponde a la escala).

Cada vez que se multiplica resulta que el número resultante se aumenta respecto a la cantidad de bits que tenga la parte fraccionaria, lo cual implica que se debe tener un registro para resultados lo suficientemente grande para no perder los bits que se puedan perder por tal efecto.

Después de haber realizado la multiplicación se puede desplazar los bits hacia la izquierda una cantidad de veces igual al número de la escala en cuestión con el fin de obtener un resultado del mismo tamaño de bits que los operandos.

Al final esto deriva en una operación de multiplicación especial con respecto a la que se suele usar.

La resta de números en punto fijo permanece intacta ya que se mantiene igual el número de bits del resultado de tal operación.

3.4 Estructura del RGU

La Unidad de Generación de Rayos (RGU) usa dos instancias de memorias: una para las intrucciones y otra para el almacenamiento de datos. La tarjeta Papilio que ha implementado el modelo del GPU Theia posee una memoria DRAM lo cual facilitaría enormemente el acceso a memoria y permitiría al dispositivo tener una mayor programabilidad.

Por lo anterior se puede apreciar que el RGU tiene la capacidad de programar las instrucciones básicas que tiene así como su orden, con el objetivo de permitir la experimentación del dispositivo así como explotar las capacidades de debugging que posee las instrucciones del GPU Theia por medio de comunicación serial.

La RGU también tiene una instrucción especial llamada PUSH que habilita a la cola que hay a la salida del RGU almacenar el valor que se calculó por medio de la tabla (LUT) y las iteraciones.

Actualmente el RGU tiene pipeline de instrucciones que le permite evitar los peligros de datos (*data hazards*) pues inicialmente se necesitaban colocar instrucciones NOP intermedias ("búrbujas") para lograr que los cálculos realizados en la instrucciones se pudieran guardar apropiadamente, ahora se hace *forwarding* de los datos de la intrucción actual a la siguiente instrucción en caso de dependencia de datos en los registros.

3.5 Instrucciones de iteración

A continuación se plantean las instrucciones necesarias (en seudocódigo) para la ejecución del método de Newton-Raphson para el cálculo de la raíz cuadrada.

Algorithm 2 Método de Newton-Raphson

```
1: procedure NEWTON-RAPHSON PARA RN
2:     Busque X en LUT para hallar R0
3:     for i hasta N do
4:         Multiplique Ri por Ri y guarde en S0
5:         Multiplique X por S0 y guarde en S1
6:         Reste X menos S1 y guarde en S2
7:         Multiplique S2 por Ri
8:         Desplace un bit a la derecha a S2 y guarde en Ri
9:     Enviar dato válido
```

3.6 Cálculo de raíces con valores de entrada grandes

Inicialmente se eligió trabajar con una tabla de 128 (2^7) valores pero se debe abarcar el espacio de (2^15) posibles valores enteros, debido a tal necesidad de abarcar el resto de números se creó un módulo llamado FixedPointSquareRoot encargado de tomar el número de entrada y dividirlo entre 256 (2^8) desplazando el número 8 bits a la derecha, buscar el número más cercano en la tabla y después multiplicarlo por 16 (2^4) que es la raíz cuadrada de 256: algo semejante a calcular $\sqrt{256 * X/256} = 16 * \sqrt{X/256}$

Con esta técnica se logra aumentar el rango de posibles valores de entrada y salida que iba a experimentar la RGU, y con ello mejorar el dominio de cobertura.

3.7 Modelo del banco de pruebas

El test bench implementando en esta ocasión solo consiste en una componente que genera las señales de entrada seudoaleatorias para un número de iteraciones específico en el método de Newton-Raphson y una componente de chequeo basado en salidas que se encarga de verificar que el comportamiento es el adecuado. La componente de chequeo consiste en un archivo de Verilog que imprime las señales en un archivo y un script en python que luego lee el archivo y calcula el error existente entre los números en punto fijo que salen

del RGU y el valor real del inverso de la raíz cuadrada para luego calcular un error promedio para cada corrida relacionada a un número de iteraciones específico.

Luego se recopila la información de correr cada script y se genera una gráfica que permite tomar decisiones de diseño pertinentes tanto a la precisión del RGU como a la cantidad de instrucciones almacenadas en el espacio de memoria RAM respectiva.

4 Resultados

4.1 Prueba inicial revisada por señales de simulador

En primera instancia es importante seleccionar señales que permitan verificar que el funcionamiento del dispositivo es el correcto, y que ejerciten las operaciones descritas anteriormente en la implementación del método de Newton-Raphson.

Se corrieron 32 pruebas con distintas variaciones en los bancos de prueba, donde se dividían 4 pruebas, para cada conjunto de iteraciones de 1 hasta 4 en el método de Newton-Raphson, relacionadas a un subconjunto de entrada que va de 8 bits hasta los 15 bits. Los bancos de prueba en la componente de chequeo guardaban los datos que luego se procesaron por medio de un programa en python que convirtió que calcula el error real que se había producido por medio de la Unidad de Generación de Rayos.

En esta sección del trabajo se muestran dos capturas de pantalla que corresponden a ciertas vistas del simulador que emplea el ISE de Xilinx, mostrando las etapas de una iteración del método de Newton-Raphson en el cálculo del inverso de la raíz cuadrada del número 2 en formato de punto fijo con escala 17.

En la imagen 1 se muestra en wA el valor de entrada correspondiente a 0x400000 que equivale a 2 en representación de punto fijo con la coma corrida 17 espacios hacia la izquierda. Se observa como en la primera instrucción el valor del registro de salida rResult toma el valor 0x16A09 que corresponde al valor de aproximación que provee la tabla de valores iniciales. De hecho el número 0x16A09 equivale en números reales aproximadamente a 0.7070999, esto se sabe al traducir 0x16A09 a decimal y después dividirlo entre 2^{17}, ya en este caso la escala es igual a 17.

En la tabla 4.1 se pueden apreciar información básica de las señales, con el objetivo de mejorar la comprensión de las señales vistas en el simulador.

En la imagen 1 se pueden seguir viendo las operaciones que corresponden a los pasos intermedios de multiplicación por los que pasa.

En la imagen 2 se observa que después de realizar las operaciones de multiplicación, resta y desplazamiento se obtiene que en la última operación el registro rResult adquiere el valor de 0x16A0A, equivalente a 0.7071075, lo cual implica que la iteración respecto al valor ini mejoró la aproximación del

Señales importantes	Importancia
wA	Valor del Registro A dentro del RGU
wB	Valor del Registro B dentro del RGU
wDestination	Número de registro que se escribe en memoria
wOperation	Código que identifica cada instrucción
rResult	Registro con el valor de salida del RGU

Tabla 4.1: Información básicas de señales en el simulador

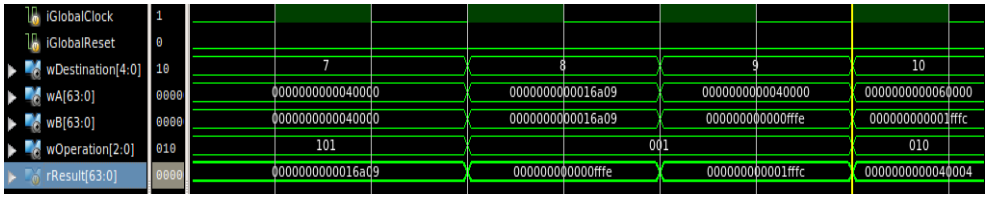

Figura 4.1: Primera parte de captura de la simulación de señales

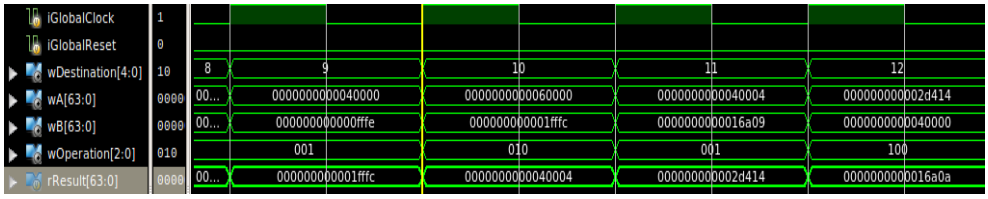

Figura 4.2: Segunda parte de captura de la simulación de señales

inverso de la raíz cuadrada pues el valor real es aproximadamente 0.7071067. Lo anterior implica la tendencia de mejorar la precisión siempre y cuando se encuentre un valor inicial cercano al valor meta.

4.2 Pruebas en los distintos rangos de bits de entrada

El RGU inicialmente emplea números de 32 bits con 15 bits de parte entera. La parte entera de estos números se emplea como valor de entrada a una tabla de memoria con 128 valores (7 bits). La tabla de memoria está encargada de generar una valor de iteración inicial, por lo que se implementó un módulo llamado FixedPointSquareRoot cuya capacidad es aumentar el rango de cobertura de la unidad, pero dicho módulo aumenta el error al hacer una estimación ya que hace un corrimiento hacia la derecha de los 8 bits menos significativos de la parte entera de los números en punto fijo para poder encontrar un valor en la

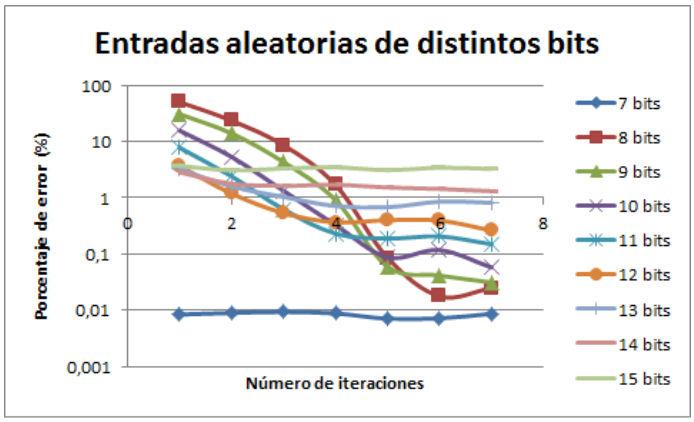

Figura 4.3: Gráfico sobre los porcentajes de error ante iteraciones con distintos rangos de bits

tabla que abarca solo números de 7 bits (128 posibilidades).

En la tablas **??** y 4.3 y en la figura 4.3 cuya escala en el eje Y es logarítmica, se pueden observar que los porcentajes de error en 8 y 9 bits al iniciar las iteraciones eran bastante altos (de hasta 50 porciento) pero conforme se iteraba 6 y 7 veces se alcanzaba un porcentajes de error promedio inferiores a 0.05 porciento.

En general la tendencia de las entradas con distintos bits es disminuir su porcentaje de error conforme se van aumentando las iteraciones. Solo en los casos de 14 y 15 bits se puede apreciar que se pierde el cáracter de disminución observado en los casos de menor cantidad de bits y más bien se genera un error casi constante, lo cual estaría dado por la incapacidad del hardware de proporcionar valores iniciales más precisos para lo cual se necesitaría tablas con mayor cantidad de valores, y de hecho los errores máximos en el caso de entradas de 15 bits llega casi el 10 porciento en promedio después de 7 iteraciones.

Sucede que en valores de entrada de solo 8 y 9 bits el error inicial es mucho ya que siempre se está haciendo un corrimiento de 8 bits a la parte entera del número en el módulo de FixedPointSquareRoot, y tal efecto se podría disminuir al reducir el corrimiento de 8 bits a un corrimiento de 4 bits, lo cual implicaría correr la coma en los números en punto fijo de modo que haya 21 bits de parte fraccionaria y 11 bits de parte entera.

Iteración	7	8	9	10	11	12
1	0,00841071	50,0658633	30,7774517	15,6596942	7,76805536	3,66909914
2	0,00894086	23,5363945	13,9400145	5,31296183	2,41015261	1,18594374
3	0,00930179	8,55821424	4,29449621	1,32698968	0,62065573	0,53281164
4	0,00876417	1,69111624	0,90878392	0,32663346	0,23015816	0,36576529
5	0,00706618	0,08517355	0,05935672	0,08746336	0,18863339	0,39944359
6	0,00716321	0,0177213	0,04125542	0,11741878	0,20503292	0,39260082
7	0,00864808	0,02467144	0,03124886	0,05765008	0,14925324	0,26242017

Tabla 4.2: Tabla de iteraciones de 7 a 12 bits de entrada con su respectivo error

Iteración	13	14	15
1	3,21100282	2,83521291	3,69484261
2	1,58770231	1,77968444	3,06908
3	1,03472789	1,63220475	3,31805053
4	0,71357453	1,70383192	3,46803755
5	0,67879743	1,52332552	3,10363742
6	0,8514918	1,43644598	3,41601461
7	0,81234608	1,30897039	3,25471677

Tabla 4.3: Tabla de iteraciones de 13 a 15 bits de entrada con su respectivo error

5 Conclusiones y recomendaciones

- Se logró desarrollar apropiadamente un modelo conductual en lenguaje Verilog que permitiera describir el funcionamiento del hardware requerido para la generación de rayos normalizados que estará dentro de la arquitectura del GPU Theia.
- Se investigó sobre posibles métodos para el cálculo de inversos de raíces cuadradas, en el cual, al final se decidió usar el método de Newton-Raphson.
- Después del trabajo de investigación se definieron las instrucciones y mecanismo internos que permitirían al módulo de RTL lograr generar vectores normalizados en formato de punto fijo, dentro de distintos grados de precisión dependiendo del número de iteraciones posibiles.
- Se realizó una verificación funcional básica de la Unidad de Generación de Rayos (RGU) donde se trató de cubrir distintas formas de calcular los inversos de las raíces cuadradas, principalmente variando el número de iteraciones por medio de las instrucciones o variando el dominio (número de bits) que podía tener el valor de la parte entera de los números en punto fijo.
- Se crearon en general 32 bancos de prueba con sus respectivas componentes de generación y chequeo para probar distintos escenarios donde la precisión del cálculo de la Unidad de Generación de Rayos estaba a prueba para observar su comportamiento general y tomar decisiones como detener el número de iteraciones en 3.
- El tamaño de las tablas que contienen valor inicial con el que se realiza la iteración inicial en el método de Newton-Raphson definen el error que va a tener una raíz al salir de la Unidad de Generación de Rayos.

Bibliografía

Akenine-Moller, T., Haines, E., y Hoffman, N. (2008). *Real-Time Rendering*. A K Peter.

Goldschmidt, E. (1964). Applications of division by convergence. Master's thesis, Electrical Engineering School M.I.T.

Markstein, P. (2004). Software division and square root using goldschmidts algorithms. *Real Number and Computers, Dagstuhl, Germany*.

Marques, R., Santos, L. P., Leskovsky, P., y Paloc, C. (2009). Gpu raycasting. *17 Portuguese Conference on Computer Graphics, Portugal, 2009*.

Nah, J., KWON, H., Kim, D., Jeong, C., Park, J., HAN, T., Manocha, D., y Park, W. (2014). Raycore: A ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics, Vol. 33*.

Schmittler, J., Woop, S., Wagner, D., Paul, W., y Slusallek, P. (2004). Realtime ray tracing of dynamic scenes on an fpga chip. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*.

Schulte, M. y Wires, K. (1999). High-speed inverse square roots. *14th IEEE Symposium on Computer Arithmetic*.

Shirley, P. y Marschner, S. (2009). *Fundamentals of Computer Graphics*. A K Peters.

Soderquist, P. y Leeser, M. (1997). Floating-point division and square root: Choosing the right implementation. *IEEE Micro*.

Valverde, D. (2015). Architecture specification version 0.1.

Wile, B., Goss, J., y Roesner, W. (2005). *Comprehensive Functional Verification The Complete Industry Cycle*. Morgan Kaufmann.

Woop, S., Brunvand, E., y Slusallek, P. (2006). Estimating performance of a ray-tracing asic design. *IEEE Symposium on Interactive Ray Tracing*.