Contenido

[PARTE INTRODUCTORIA 2](#_Toc337478470)

[PRIMERA PARTE: JART Y RTENGINE 4](#_Toc337478471)

[SEGUNDA PARTE: RAYTRAC 5](#_Toc337478472)

[TERCERA PARTE: CIERRE 6](#_Toc337478473)

[PARTE INTRODUCTORIA 7](#_Toc337478474)

[0. INTRODUCCION 7](#_Toc337478475)

[1. CONTEXTO Y ALCANCE DE LA IMPLEMENTACION DE LA SOLUCION 9](#_Toc337478476)

[2. CONTEXTO DEL PROBLEMA 9](#_Toc337478477)

[3. APLICACIÓN DE RENDERING : JART 10](#_Toc337478478)

[3.1. Descripción General 10](#_Toc337478479)

[3.2. Implementación 10](#_Toc337478480)

[4. RtENGINE 10](#_Toc337478481)

[4.1. Capa de Software 10](#_Toc337478482)

[4.2. Capa de Hardware 10](#_Toc337478483)

[5. RAYTRAC 11](#_Toc337478484)

[5.1. Adaptadores de formato 11](#_Toc337478485)

[5.2. Pipeline Aritmético 11](#_Toc337478486)

[5.3. Interfaz de Programación 11](#_Toc337478487)

[5.4. Control de Flujo 11](#_Toc337478488)

[5.5. Simulación y Máxima Frecuencia 11](#_Toc337478489)

[6. RESULTADOS 12](#_Toc337478490)

[6.1. Pruebas Realizadas 12](#_Toc337478491)

[6.1.1. Pruebas de verificación. 12](#_Toc337478492)

[6.1.2. Pruebas de Validación. 13](#_Toc337478493)

[6.2. Resultados y Conclusiones 14](#_Toc337478494)

[6.2.1. Verificación: Resultados. 14](#_Toc337478495)

[6.2.2. Validación: Resultados. 17](#_Toc337478496)

[6.2.3. Conclusiones. 17](#_Toc337478497)

[6.3. Trabajo a Futuro 18](#_Toc337478498)

# PARTE INTRODUCTORIA

***Capitulo 1. Contexto/Alcance de la implementación de la solución S.*** en este capítulo se explica en que consiste la implementación de la solución *S*: La manera en que se va a implementar, con qué herramientas, que interfaces se piensan usar (de programación, de usuario etc), para generar una visión integral de la solución desde la perspectiva de lo implementado.

Del libro que existe en este momento puede tomarse elementos de:

***+Herramientas y Plataformas Tecnológicas,*** en esta sección se realiza un recuento de los recursos tecnológicos a usarse, recursos clasificados en recursos de hardware y software.

***Capitulo 2. Contexto del Problema P.*** En este capítulo se describe de manera breve el algoritmo de trazado de rayos. En el lugar de la lectura que corresponda, se contrastará de manera concreta con el problema *P* descrito en el capítulo 0.

De esta manera se genera en el lector, una visión matemática del problema *P.*

# PRIMERA PARTE: JART Y RTENGINE

*Se pretende dar una organización al documento que vaya de lo general a lo específico*

***Capítulo 3***, Se describe la implementación de la aplicación de rendering JART.

*Seccion 3.1. Descripción general del JART. Qué es, cómo se utiliza etc.*

*Sección 3.2. Implementación del JART. Implementación de la resolución geométrica y del color a partir de una estructura de kernels.*

***Capítulo 4,*** *Se describe la implementación del RtEngine, el componente híbrido de hw/sw del sistema.*

*Sección 4.1. Qué es el RtEngine, cuales son sus bloques funcionales.*

*Sección 4.2. Se describe como está integrada la capa de software, la cual está conformada por los drivers, sistema operativo y capas de abstracción de hardware de todo el sistema, dando forma a un conjunto de servicios y al modelo de programación del sistema.*

*Sección 4.3. Se describe como está integrada la capa de hardware, se hace referencia inicialmente a los componentes de terceros (Altera) y por último se hace referencia al RayTrac que es el componente desarrollado en este trabajo.*

# SEGUNDA PARTE: RAYTRAC

***Capítulo Quinto,*** Introducción al *RayTrac,* qué es, qué hace, qué circuitos lo componen, cuál es la filosofía de diseño y qué tipo arquitectura será usada para su implementación.

*Sección 6.1.* Adaptadores de formato. El *RayTrac,* procesa vectores que en conjunto pueden tener hasta 192 bits de ancho y en el *RtEngine* circulan datos de 32 bits de ancho. Para cargar datos en el *RayTrac* y para que este entregue resultados, se necesitarán *adaptadores de formato*, que conviertan los *datos en vectores* y los *vectores en datos.* Este capítulo detalla el diseño, arquitectura e implementación de estos adaptadores.

*Sección 6.2.* Se detalla la arquitectura del *pipeline* *aritmético*, el circuito que ejecuta las operaciones aritméticas que desarrolla el *RayTrac* como procesador de vectores. Los detalles del diseño desarrollo e implementación por tipo de operación, se encuentran descritos en esta sección.

*Sección 6.3.* Describe la interfaz de programación implementada para realizar aplicaciones con el *RtEngine, pero al nivel del RayTrac*. La interfaz de programación tiene dos aspectos principales: la configuración de una función especializada o *kernel* y la programación de las fuentes de flujo de entrada, la cantidad de datos que vienen en este flujo y la configuración del destino de los resultados que van en los flujos de salida*.*

*Sección 6.4.* se explica en detalle el control de flujo de datos, el cual es controlado por una *máquina de estados* *para el control de flujo.* Se describe los mecanismos que usa el *RayTrac* para realizar transacciones de lectura de datos, cargar vectores y comenzar transacciones de escritura de datos, como resultados.

*Sección 6.5.* Simulación. Para realizar la simulación del RayTrac y su pipeline aritmético, se desarrolló un par de scripts que permiten la observación de ciertas señales, que se encuentran en los diseños RTL de jerarquía más bajo y automatizan el proceso de despliegue de resultados en archivos tipo *.csv* , o cómo lo indican sus siglas en inglés, archivos separado por comas. El objetivo de estas herramientas es simular una secuencia de instrucciones enviadas al RayTrac, emulando ***funcionalmente*** el comportamiento del bus externo (Avalon) y verificar los resultados.

Adicionalmente se realiza la simulación de tiempos, la cual es importante para saber si la segmentación de los circuitos combinatorios que componen el *RayTrac* es suficiente y calcular la máxima frecuencia soportada por estos.

# TERCERA PARTE: CIERRE

***Capítulo 7. Resultados.***

*Sección 7.1. Se detallan las pruebas realizadas y los resultados a modo de verificación por cada uno de los 3 escenarios de ejecución: Sw, Hw con un RayTrac y Hw con dos RayTrac.*

*Sección 7.2.* Se validan los resultados generales y se presentan las conclusiones del trabajo en general. A partir de los resultados se debe enunciar la validez, total, parcial o nula de la tesis, propuesta como solución al problema.

*Sección 7.3.* Se enuncian posibles trabajos, por hacer a partir de este, entre los que se encuentran mejoras a la solución de la aproximación Se describen otras aplicaciones que se pueden ver beneficiadas con la solución.

# PARTE INTRODUCTORIA

# INTRODUCCION

*Rendering* es el proceso computacional en el que los elementos e interacciones de un modelo representado en 3 dimensiones espaciales (polígonos, fuentes de luz, etc.), son proyectados contra un plano arbitrario que también pertenece al modelo, para formar en él una imagen sintetizada.

Para calcular y realizar esta proyección, un sistema de cómputo ejecuta un algoritmo de *rendering.* Existen varias clases de algoritmos, siendo los principales el algoritmo de *raster*  y el algoritmo de *ray tracing* o en español algoritmo de trazado de rayos.

El algoritmo del trazado de rayos *se considera* cualitativamente, en términos de calidad de las imágenes y otros criterios, muy superior, ya que básicamente, simula el comportamiento óptico de la luz, ejecutando el cálculo de los fenómenos físicos de absorción, refracción y reflexión.

El problema del algoritmo de trazado de rayos es la carga computacional que asume el procesador al ejecutarlo. El desempeño de dicha ejecución es siempre menor que la de un algoritmo de *raster* para la misma escena.

Por ejemplo, un desarrollador de videojuegos, optará por ejecutar un algoritmo de *raster* cuyo desempeño le brinda una *tasa de refresco* mínimo, así no tenga las prestaciones de calidad visual que el algoritmo de *trazado de rayos* ofrece*.*

Para que el algoritmo de trazado de rayos no sea desechado, su desempeño debe ser mayor, en comparación con los algoritmos cuyo desempeño los convierte en la opción para elegir.

Dada la falta de desempeño del algoritmo de trazado de rayos, comparada con el desempeño de otros algoritmos de uso frecuente, *se propone* una solución que aborda el problema desde el *hardware en el que corre el algoritmo.* La solución consiste en el diseño e implementación de un procesador vectorial, que acelere los cálculos críticos y explote ciertas características especificas en la ejecución de un algoritmo de trazado de rayos, obteniendo un trazador de rayos en hardware y acelerando su desempeño, aumentando la posibilidad de elegir al algoritmo de trazado de rayos, por encima de otros algoritmos convencionales de *rendering.*

*Objetivo General.*

Diseñar e implementar una arquitectura de trazado de rayos para síntesis de imágenes.

*Objetivos Específicos.*

* A. Diseñar, simular e implementar los circuitos aritméticos que soporten la resolución geométrica de un sistema de trazado de rayos.
* B. Implementar una interfaz operativa, humano – máquina, mediante codiseño y herramientas propietarias.
* C. Probar la implementación de la arquitectura de trazado de rayos, a partir de software.

“Para el cumplimiento de estos objetivos se implementó un procesador vectorial llamado *RayTrac (A),* una interfaz entre el operativallamada *RtEngine (B)* y una aplicación de *rendering* a la que llamamos *JART (Just Another Ray Tracer) (C) .*

El *JART* es una aplicación de *rendering,* que presta servicio de síntesis de imágenes y medición del desempeño del algoritmos de trazado de rayos subyacente. Como tal es un componente de software, construido encima de los servicios que presta el componente *RtEngine.*

El *RtEngine* es la plataforma en la que se está implementando la solución, es un componente híbrido de software y hardware. Su servicio principal es servir de interfaz entre el *JART* y el *RayTrac.* El *RtEngine* está construido encima de los servicios que proveen diversas *API’s (Application Programming Interface), IP’s* de hardware y el componente *RayTrac.*

El *RayTrac* es un procesador vectorial orientado a *SIMD (Single Instruction Multiple Data),* diseñado para procesar operaciones vectoriales en formato punto flotante IEEE 754. La función de este componente es acelerar las operaciones críticas del algoritmo de trazado de rayos y explotar las características especiales del algoritmo.*”*

Al implementar la solución planteada, se espera hacer mediciones de desempeño a partir de 2 formas de ejecución características:

* Ejecución por software, Nios II.
* Ejecución Acelerada por Hardware, RayTrac.

La primera ejecución simplemente espera obtener una medición de referencia del desempeño del algoritmo.

En la segunda ejecución se debe obtener una medición con un desempeño más alto que el obtenido en la primera ejecución.

El primer modo de ejecución, representa el escenario en el que se ejecuta el algoritmo de trazado de rayos sobre una máquina con recursos de cómputo convencionales.

El segundo modo de ejecución, representa el escenario en el que se ejecuta el algoritmo de trazado de rayos, usando un procesador vectorial periférico.

# CONTEXTO Y ALCANCE DE LA IMPLEMENTACION DE LA SOLUCION

# CONTEXTO DEL PROBLEMA

PRIMERA PARTE : JART Y RTENGINE

# APLICACIÓN DE RENDERING : JART

# Descripción General

# Implementación

# RtENGINE

Descripción Gral del RtEngine. Diagrama de Bloques. Ejemplos de posibles aplicaciones. Descripción breve de la aplicación a implementar.

El *RtEngine* es una plataforma de Hardware y Software, para desarrollar aplicaciones que basadas en matemática vectorial principalmente. El diseño e implementación de la plataforma del *RtEngine* es dominio de este trabajo. Adicionalmente el diseño de una aplicación que utiliza la plataforma es también parte del alcance de este trabajo, en particular, la aplicación propuesta dentro de los objetivos es la implementación de un sistema de síntesis de imágenes a partir de una técnica de *render* denominada *Ray Tracing* o en español Trazado de Rayos.

# Arquitectura

El *RtEngine* se puede describir como una arquitectura en capas, que van desde la capa de aplicación hasta la capa física:

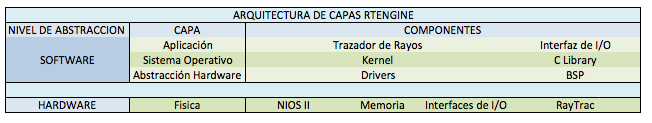


Tabla . Arquitectura por capas RtEngine.

Hay dos niveles de abstracción, el nivel *software* y el *hardware.* El nivel de *software* está conformado por 3 capas:

* *Aplicación:* La capa de aplicación es la capa donde residen los programas, las aplicaciones que hacen *algo,* con el *RtEngine,* explotándola. Los componentes que residen en esta capa son, el *trazador de rayos* y la *interfaz de I/O.*
* *Sistema Operativo:* En esta capa residen los servicios a los que puede acceder la capa de *aplicación.* Como tal la capa de sistema operativo expone una interfaz visible a las aplicaciones, dichos servicios son administrados y ejecutados por los componentes de la capa, *kernel* y *drivers.* El *kernel* es el componente de sistema operativo que abstrae el uso de los recursos de hardware relacionados principalmente con el procesador, la memoria y los temporizadores.

Entre los componentes del sistema operativo tenemos la librería C. Aunque no hace parte integral del sistema operativo, si ofrece y expone servicios a ese nivel. Los servicios que expone la librería C, facilitan la portabilidad de aplicaciones de plataforma a plataforma.

* *Abstracción Hardware :* Esta capa de software es la que aísla el sistema operativo del hardware. El código en esta capa es la implementación en software de los servicios de hardware que posee el sistema.

Los *drivers* exponen servicios para el manejo de los periféricos de hardware, entre ellos el driver para manejar el *RayTrac.* Detalles de cómo funciona el driver y los servicios que expone en el capítulo IX. *Aplicación de Trazado de Rayos JART.*

La *BSP, Board Support Package,* es una capa de software que expone servicios específicos al hardware. Los utiliza la capa de sistema operativo, no es usual que la utilice la capa de aplicación y no perder la portabilidad. Esta capa es mas utilizada para implementar los servicios *estándar* que expone la capa de sistema operativo.

* *Hardware:* Los componentes de hardware que conforman el *RtEngine: Nios II, Bus Avalon, Interfaz de E/S, Memoria, Periféricos y RayTrac.*

# Plataforma RtEngine

La plataforma del RtEngine tiene dos componentes principales, un componente hardware y componente software.

# Especificaciones Hardware.

El *RtEngine* está implementado por una serie de componentes electrónicos, que en conjunto son programables y que para ello cuentan con interfaces de entrada y salida.

|  |  |  |
| --- | --- | --- |
| Tipo de Componente | Componente | Descripción |
| Memory | 32 Mbytes DDR SDRAM | Memoria para el uso general de las aplicaciones y para implementación de *DMA, Direct Memory Access,* para el Display y para la interfaz de comunicación TSE. |
| 1 Mbyte Synchronous SSRAM | Memoria de uso general, para el espacio de datos de las aplicaciones etc. |
| 16 Mbytes External Flash | Memoria de uso general, especialmente para el almacenamiento de instrucciones, sistema operativo, bootloaders. Etc. |
| Clocking and PLLs | 50 Mhz on board clock | Clock que viene con la tarjeta de desarrollo. |
| Altera Avalon PLL generated clock 100 Mhz | Clocks generados mediante PLL dentro de la FPGA 3C25F324C6 |
| Altera Avalon PLL generated clock 60 Mhz |
| Altera Avalon PLL generated clock 40 Mhz |
| DDR SDRAM generated clock 66.5 Mhz, 133.0 Mhz | Reloj generado por la memoria DDR SDRAM |
| Switches and Indicators | 4 User Controlled Buttons | Botones de uso general |
| 4 User Controlled Leds | Leds de uso general |
| Display | 800x480 LCD Touch Display | Display de hasta 24 bits por pixel e interfaz Touch Screen. |
| Interfases de Comunicación | JTAG USB Cable | Cable para la programación del RtEngine y depuración. |
| RS232 Serial Interfaz | Interfaz Serial. Hasta 115200 bps (baudios por segundo). |
| Elementos de Procesamiento | NIOS II/S | 100 MHz + 1Kbyte Cache + Lvl 1Debugging Support |
| RayTrac Configurable Kernel | 100 MHz, Vectorial Kernel Configurable Processor |

Tabla . Componentes de Hardware Disponibles en el RtEngine.

Como se puede observar en la Tabla 16, la mayoría de componentes son los que trae el *NEEK.* Ver, *Introducción, Herramientas y Plataforma Tecnológica.*

# Aplicaciones Software.

El *RtEngine* utiliza 2 aplicaciones como componentes de Software.

El primer componente, es un script que sirve para ingresar una escena generada en *Blender 3D,* una suite de modelado, animación y *render*, al *RtEngine,* para que este genere la imagen. Este script, corre embebido dentro del programa. Está escrito en Python y se anexa a este documento. Este programa hace parte del *RtEngine,* pero el análisis del mismo, su diseño e implementación están por fuera del alcance de los objetivos de este trabajo.

El segundo componente consiste en la aplicación que corren en el *RtEngine:* el trazador de rayos. Se utilizó la aplicación *JART (Just Another RayTracer[[1]](#footnote-1))*

El trazador de rayos esta originalmente escrito para Linux y en el marco de este trabajo se porteó al sistema *RtEngine.*

El diseño, arquitectura y algunos detalles del porte al sistema *RtEngine* son detallados en el capítulo IX. *Aplicación de Trazado de Rayos JART.*

# Sistema Operativo.

El *RtEngine* usa el sistema Operativo Micro C/OS – II, sistema operativo para implementación de aplicaciones en tiempo real. Este sistema operativo viene dentro de la suite de Altera y por eso se usa para implementar el *RtEngine.*

# RayTrac Driver.

El driver del *RayTrac* expone a la aplicación *JART* un conjunto de servicios para operar el dispositivo. Los servicios expuestos utilizan la abstracción de procesamiento de *flujos* y *kernels.*

El driver provee al programador con una API para operar al *RayTrac,* la referencia de esta *API* se anexa a este documento.

Para la implementación del driver se sigue los lineamiento sugeridos por ALTERA.[[2]](#footnote-2)

SEGUNDA PARTE: RAYTRAC

# RAYTRAC

# Adaptadores de formato

La unidad aritmética vectorial en punto flotante, *RayTrac* tiene una interfaz de entrada y salida: los bloques de adaptación de formato. Los flujos de datos de entrada son poseen datos que en su forma, distintos a los que opera el *kernel* configurado en el *RayTrac* y también es distinto el formato de los datos que conforman los flujos de salida que el de los resultados que entrega el *kernel* configurado en el *RayTrac.*

La adaptación es necesaria porque la entrada de datos al *RayTrac* es a través de un único bus de datos de 32 bits de ancho. Pero las operaciones que realiza el *RayTrac* son con vectores los cuales poseen mínimo 3 componentes de cada uno y en general se utilizan dos vectores por operación. Para que una operación pueda “arrancar” o ser “valida” deben encontrarse todos los datos de ambos vectores presentes.

La necesidad de la adaptación también corre por cuenta del flujo de datos de salida. Ya que el resultado entregado por el *RayTrac* es en general un vector de 3 o 4 componentes de 32 bits cada uno y el bus de datos de salida es de 32 bits de ancho. Lo que hace necesario un control de flujo para serializar cada uno de los componentes del vector.

Son 2 los elementos de adaptación de flujos que posee el *RayTrac*:

* Adaptador 32:192 (Flujo de Entrada).
* Adaptador 128:32 (Flujo de Salida).

En este capítulo se detalla la implementación realizada. La función de estos bloques es servir de interfaz de entrada y salida al *RayTrac*.

Los mecanismos de interfaz de entrada y salida del *RayTrac* son transparentes al usuario que desarrolla software para el *RayTrac,* por lo tanto la descripción de la implementación hecha es de una interfaz de *Hardware* y no corresponde a una interfaz de *Software.*

# Adaptador de formato de flujos de Salida (AFFs).

El *adaptador de formato de flujos de salida* tiene la función de tomar los flujos que vienen como resultados de 128 bits, desde el *PA* transformarlos en flujos de 32 bits, mediante la creación de una secuencia de 4 flujos de 32 bits cada uno. El problema es que la salida del adaptador es el bus de datos de salida externo al *RayTrac* y al ser este un bus arbitrado, no es accesible todo el tiempo. El mecanismo de arbitramento no es determinístico, o sea, no se puede predecir en que momento el acceso al bus será otorgado. Por lo tanto los datos se deben “encolar” para ello se utiliza una cola de 32 bits de ancho, en donde se van almacenando secuencialmente los flujos 32 bits secuenciales. Una vez el mecanismo de arbitramento otorga el acceso al bus se activa la señal de lectura de la cola.

El *adaptador de formato de flujos de salida* se compone de una cola de resultados vectoriales, que emite flujos de 128 bits de ancho, una máquina de estados de transición y un cola de salida de datos que permite flujos de 32 bits de ancho.

## Cola de Resultados .

La cola de resultados, es un *fifo* de 128 bits de ancho. Almacena vectores de 4 componentes y su entrada esta conectada a la salida del *pipeline aritmético,* desde las señales de resultado de los bloques aritméticos que correspondan según el *kernel*  que se encuentre configurado al momento. La escritura de los datos en la cola o  *push in* está a cargo de los mecanismos de sincronización el *pipeline aritmético.* La lectura o

*push out* está a cargo del sistema de adaptación.

La cola de resultados provee la señal *empty* para indicar al resto del *RayTrac* que se encuentra vacía. Esto lo utiliza el adaptador de formato de flujos de datos de salida, pero también la *máquina de estados* del *RayTrac,* para la detección de ciertos eventos, como el final del procesamiento de un flujo de datos a través del *kernel* configurado.

Para implementar la descripción de la cola se utilizó la librería *altera*\_*mf*, la megafunción *scififo.* A continuación el siguiente bloque define las características de la implementación:

|  |  |
| --- | --- |
| **Ítem** | **Result Q** |
| Width | 128 |
| Num Words | 256 |
| Sync Signals | Wr, Ack, Empty. |
| M9k Blocks / LCells | M9K |

Tabla . Configuración de la cola de resultados del Adaptador de Formato de Flujos de Salida.

Máquina de estado adaptación de flujos.

Las solución al problema de la adaptación de los flujos de datos de salida, es escoger secuencialmente los componentes del vector en la salida de la cola de resultados.

La máquina de adaptación de flujos de salida, es una máquina de estados de 4 estados. Estos estados son VX,VY,VZ,SC. El objetivo de cada estado es decodificar que dato debe transitar hacia la interfaz externa. La máquina se activará cada vez que haya un resultado en la cola de resultados.

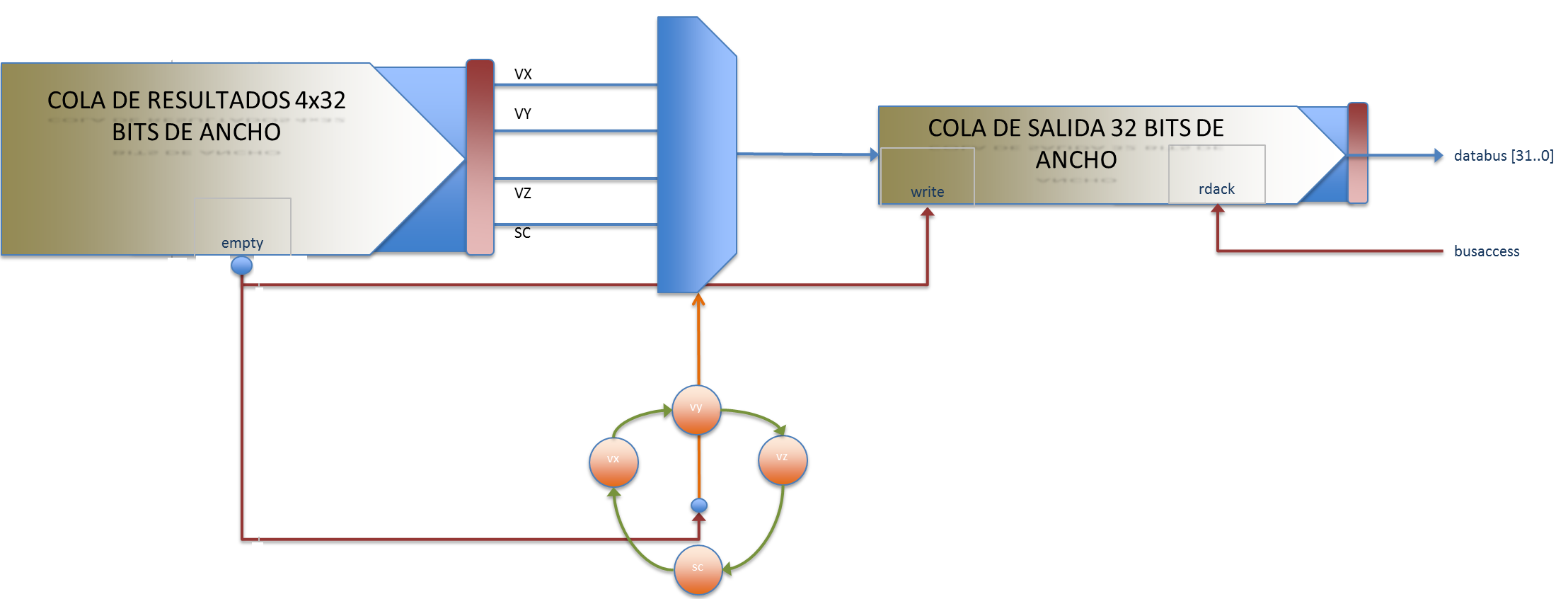


Ilustración . Adaptador de Formato de Flujos de Salida. La máquina de estados, según el estado en que se encuentre insertara los datos Vx, Vy, Vz y Sc, provenientes de la cola de resultados en la cola de salidas, transformando el flujo de 128 bits en un flujo de 32.

Se mencionó la interfaz externa en el párrafo anterior. Esa interfaz no es propiamente el bus de interconexión externo al *RayTrac.* Naturalmente se puede pensar en primera instancia que se le entregue el dato al bus externo, pero esto no es real, porque para esto deben estar sincronizados dos eventos: el evento que señaliza que la cola de resultados no está vacía y el evento que señaliza que el bus de datos a la salida se encuentra disponible. Por lo tanto y para sincronizar estos dos eventos se utiliza una cola de salida extra.

## Cola de Salida.

La cola de salida en cumple dos funciones: soporte para el control de flujo y tránsito de datos a lo largo del *pipeline aritmético* y sincronizar los eventos *empty* de la cola de resultados y la disponibilidad del bus externo de datos de salida, hacia donde deben transitar el flujo.

Los datos a los que se hace *push in,* provienen de la cola de resultados y el orden con que estos datos se cargan en la cola de salida está controlado por la máquina de estados de adaptación de flujos de salida.

Para leer los datos de la cola de salida, se debe esperar a que la *MEFC,* inicie una *transacción*  de carga en el bus de datos de salida. El bus externo de datos de salida notifica el acceso a la cola de salida, para que esta entregue el dato.

|  |  |
| --- | --- |
| **Ítem** | **Result Q** |
| Width | 32 |
| Num Words | 256 |
| Sync Signals | Wr, Ack, Empty, UsedW |
| M9k Blocks / LCells | M9K |

Tabla . Configuración de la cola de salida del Adaptador de Formato de Flujos de Salida.

# Bloque Adaptador de Flujos de Entrada.

Los flujos de datos que provienen del bus de dato exterior, son de 32 bits. El *pipeline aritmético* procesa vectores de 3 componentes de 32 bits cada uno. Por lo tanto ántes de que se carguen efectivamente los datos dentro del *pipeline aritmético,* es necesario primero precargar por los menos 6 de estos datos para operaciones de dos vectores y 3 de estos datos para operaciones unarias. Este bloque controla la secuencia de entrada de datos y carga de los vectores una vez se encuentran todos enganchados dentro del *RayTrac.*

## Máquina de Estados de Sincronización de Carga.

*La máquina de Estados de Sincronización de Carga sirve para distribuir los datos que entran al* RayTrac *en los registros que forman los vectores que operará el* kernel *conforme van llegando.*

Si el kernel configurado es una operación unaria (p.ej. Magnitud de un Vector), entonces la máquina transitará por los estados AXBX, AYBY,AZBZ, cargando tres valores por vector, si el kernel configurado es una operación de dos vectores (p.ej Suma de Vectores), entonces la máquina irá por los estados AX,AY,AZ,BX,BY,BZ, cargando seis valores por cada para de vectores.

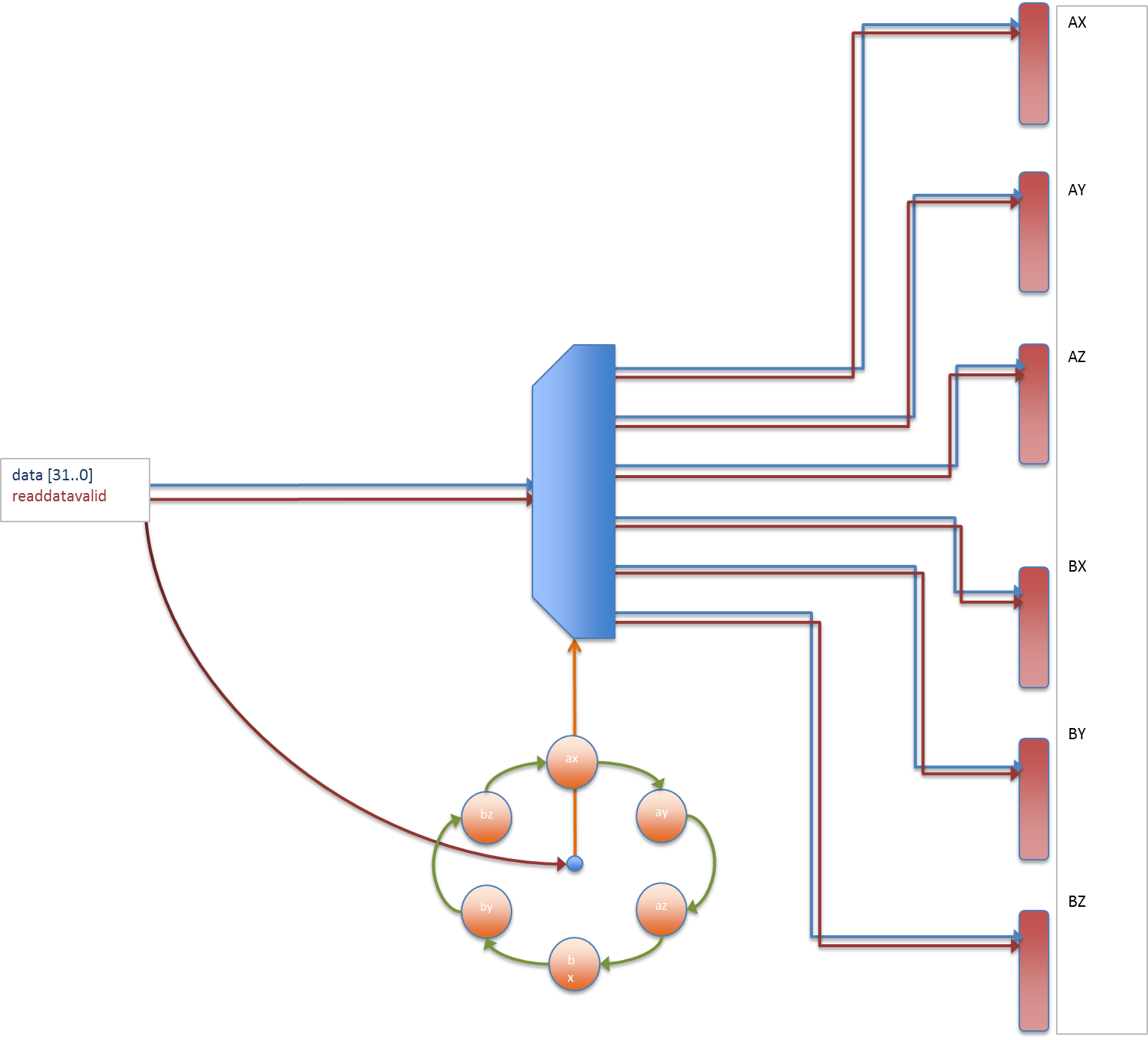
******

Ilustración . Adaptador de Formato de Flujos de Entrada. La máquina de estados determina en que registro se escribe los datos validos que vienen de un bus externo.

La idea es cargar secuencialmente los datos en seis registros con la información de los siguientes vectores a operar dentro del *kernel* configurado. Una vez se encuentren cargados los seis registros con datos *“validos”*, se procede a procesarlos en la siguiente etapa del *RayTrac,*  la cual corresponde al *pipeline aritmético*. El último registro de la cadena de sincronización de carga servirá de señal para dar como valido los seis datos que se encuentran cargados en los registros de datos.

La máquina de estados se activa cada vez que hay un dato valido en la interfaz externa de entrada, la cual es un bus de datos que permite flujos con un ancho de 32 bits. Entre los 6 registros de 32 bits cada uno hay 192 bits, una vez están cargados los datos en los registros se tiene entonces el flujo rearmado pero el ancho ahora es de 192 bits. La relación del adaptador es de 32 a 192. El diseño implementado es un demultiplexor con su controlador gobernado por la máquina de estados.

El *AFFe* se constituye como el circuito de entrada de flujos en un contexto de procesamiento de flujos o como la entrada desde el bus de datos externo de entrada en el contexto de la arquitectura del *RayTrac*.

# Pipeline y Arquitectura de Operaciones Aritméticas

El pipeline aritmético es un conjunto de circuitos combinatorios conectados de manera secuencial mediante registros. Cada circuito tiene las entradas y las salidas registradas. Dado que los circuitos están conectados de manera secuencial, las salidas registradas de un circuito, son las entradas registradas del siguiente. A cada circuito combinatorio con entradas y salidas registradas, se le denomina *etapa.*

La misión principal *de las etapas* en el pipeline es ejecutar operaciones de naturaleza aritmética, tales operaciones son:

* Suma.
* Resta.
* Multiplicación.
* Inversión.
* Raíz Cuadrada.

# Arquitectura pipeline aritmético.

El pipeline aritmético cuenta con once bloques que realizan diversas operaciones aritméticas en punto flotante:

* 3 Sumadores.
* 6 Multiplicadores.
* 1 Raíz Cuadrada
* 1 Inversor.

Cada circuito tiene una latencia específica que se detalla más adelante en esta sección, la latencia está determinada en ciclos de reloj. La salidas y las entradas de cada circuito están conectadas a la *IDC.*

El pipeline cuenta con 6 registros de entradas de operandos. Una vez entran los operandos al pipeline estos se van transformado etapa tras etapa, dentro de cada circuito aritmético. Cada vez que salen de la última etapa de cualquier circuito aritmético entran a la *interconexión decodificada de circuitos aritméticos* y de ahí continúan al siguiente circuito aritmético que corresponda o al bloque de *adaptación de formato de flujos de salida*, si el resultado ya está listo.

Como la *interconexión decodificada de circuitos aritméticos* puede configurar distintas funciones especializadas o *kernels,* la longitud o extensión del camino de datos resultante puede cambiar y a lo largo de este puede variar el número de circuitos aritméticos que procesarían los vectores de entrada.

La interconexión resultante de las diferentes operaciones resulta en *kernels*  u operaciones compuestas, que no son más que operaciones vectoriales y estas son:

* Producto Punto
* Producto Cruz
* Suma de Vectores
* Resta de Vectores
* Multiplicación de un Vector por un número Escalar.
* Magnitud de Vectores
* Normalización de Vectores

Estas operaciones son las que realmente se especifican en la interfaz de programación del *RayTrac*, en la sección 5.3.

Cada *kernel* configurado supone pues un *pipeline “distinto”* . El detalle de los *kernels* configurados para cada operación se encuentra detallado en la sección 5.3.3,*, Interconexión Decodificada de Circuitos Aritméticos.*

# Representación de Datos en Punto Flotante.

Los flujos de entrada y de salida en el *RayTrac*, están representados en el sistema aritmético base 2 y sus valores aritméticos están representados en punto flotante sencillo, usando el estándar IEEE 754[[3]](#footnote-3).

Este estándar de representación viene dado por un trama de 32 bits. Esta trama tiene 3 campos que define el estándar:

* Signo: Es un campo de un bit, que en general, define si el valor del número a representar es menor que cero. El estándar permite representar el pseudo - valor “-0”.
* Exponente: Es un campo de 8 bits y su propósito es representar un exponente con offset.
* Mantisa: Es un campo de 23 bits. Representa un factor por el que se debe multiplicar un valor potencia de 2, para obtener el valor representado. Los valores de la mantisa son fijos y pertenecen al rango [1.0,2.0). La representación de la mantisa en punto fijo que se denota A(1,23). La notación indica que la mantisa se representa con 24 bits, uno más que el campo de la trama correspondiente, la razón es que el bit de la parte entera siempre debe ser 1.

Los bloques que componen las operaciones soportan el estándar IEEE 754 de representación. Por lo tanto si se requiere operar 2 valores enteros, estos deben ser convertidos en el código a estándar IEEE 754, mediante software y esto causa un overhead en el performance.

## Punto Fijo vs. Punto Flotante.

Cuando se implementa en hardware los bloques aritméticos, usando punto flotante, se garantiza que el tamaño de palabra del resultado y los parámetros son idénticos. Una arquitectura de punto fijo debe truncar el valor de los bits menos significativos para poder garantizar esto. Si la arquitectura en hardware no los truncará y simplemente asumiera el valor de palabra distinto en la salidas, sería imposible reutilizar los mismos bloques para hacer distintas operaciones, por ejemplo sería imposible usar los mismos bloques de suma, para implementar producto punto y suma de vectores.

b0

b1

b2

b3

b4

b5

b6

b7

b0

b1

b2

b3

b4

b5

b6

b7

b8

b9

b10

b11

b12

b13

b14

b15

b0

b1

b2

b3

b4

b5

b6

b7

Ilustración . Suma en punto fijo. Los 2 sumandos poseen 8 bits cada uno y el resultado de la suma deriva en un resultado de 16.

El problema de mayor repercusión en la arquitectura de punto fijo es que el tamaño de los operandos y de los resultados, los cuales difieren, provocando que no se puedan realizar sumas acumulativas, puesto que en cada iteración de dicha suma, se requeriría de un circuito que soporte parámetros con el doble de ancho que la iteración anterior.

Finalmente si la arquitectura optará por el truncamiento para mantener fijo el tamaño de los operandos, es inminente la perdida de precisión y por lo tanto perdida de información. Esto no es aceptable.

Sin embargo existen razones para preferir la implementación de una unidad de punto fijo en lugar de punto flotante y la más importante de ellas es la cantidad de celdas lógicas a usar.

Sin embargo el uso de circuitos de punto flotante, resultaría favorable, a pesar que dicha implementación utilizaría más celdas lógicas que el circuito de punto fijo, es claro que resulta mucho más funcional y tiene grandes ventajas en la precisión.

En la siguiente ilustración se observa la suma de 2 números en punto flotante, los cuales poseen un bit para representar el signo, 8 para el exponente y 23 para la mantisa. Cada componente, -signo, exponente o mantisa-, debe ser operado con su contraparte en el otro parámetro, aunque esto conlleve un mayor número de celdas lógicas al final se obtiene un resultado del mismo ancho que los parámetros, lo cual resulta una ventaja en términos de diseño bastante importante.

b8

b9

b10

b11

b12

b13

b14

b15

b0

b1

b2

b3

b4

b5

b6

b7

b17

b18

b19

b20

b21

b22

b16

b29

b23

b24

b25

b26

b27

b28

b30

b31

b8

b9

b10

b11

b12

b13

b14

b15

b0

b1

b2

b3

b4

b5

b6

b7

b17

b18

b19

b20

b21

b22

b16

b29

b23

b24

b25

b26

b27

b28

b30

b31

b8

b9

b10

b11

b12

b13

b14

b15

b0

b1

b2

b3

b4

b5

b6

b7

b17

b18

b19

b20

b21

b22

b16

b29

b23

b24

b25

b26

b27

b28

b30

b31

Ilustración . Suma en punto fijo. Los 2 sumandos poseen 8 bits cada uno y el resultado de la suma deriva en un resultado de 16.

# Suma IEEE 754, 32 Bits, Punto Flotante de Precisión Simple.

# Abstracción Algebraica de la Operación.

Se tiene los valores F0 y F1 cuya representación es en punto flotante:

F0 = s0 2e0 f0

F1 = s12e1 f1

Donde si toma el valor de -1 o 1 dependiendo del signo, ei es un valor entero y fi es una fracción en el rango [1.0,2.0).

F0 + F1 = s0 2e0 f0 + s12e1 f1

Por razones prácticas en la ilustración de este ejemplo, se supone que e0 > e1. Se tiene entonces:

Se puede reemplazar el valor de la diferencia de los exponentes, multiplicando y dividiendo al mismo tiempo por una potencia de 2 elevada a :

Y reemplazamos e0 en la expresión:

Finalmente multiplicar por una potencia de 2 elevada a un numero negativo, en representación binaria, significa hacer un corrimiento de bits a la derecha:

Sin embargo la fracción resultado *fres’*puede que no haya quedado normalizada, dentro del rango [1.0,2.0). Así que se debe normalizar, realizando corrimientos a la izquierda o a la derecha, los bits de *fres’* para obtener la fracción final y restando o sumando respectivamente, a e0, el número de corrimientos hechos, obteniendo finalmente *2res*.

# Diagrama de Flujo.

|  |  |
| --- | --- |
|  | Etapa 0   * Cálculo del signo del segundo operando (s1). * Cálculo de la diferencia de los exponentes (delta). * Si el exponente e0 es menor e1, se intercambian todos los operandos. |
| Etapa 1   * Decodificación “One Hot” de la diferencia de exponentes (deltaOneHot). * Correr los bits de la mantisa de menor exponente. * Signar la mantisa de mayor exponente. |
| Etapa 2   * Signar la mantisa corrida. |
| Etapa 3   * Sumar las mantisas. |
| Etapa 4   * Sacar el signo de la mantisa resultante. |
| Etapa 5   * Calcular el corrimiento que requerirá la mantissa (delta). |

Tabla . Diagrama de flujo de la suma flotante. La columna de la izquierda ilustra el diagrama, la columna en la derecha, detalla los bloques correspondientes en la columna izquierda. Cada etapa se encuentra precedida o sucedida por flipflops.

## Diagrama de Bloques

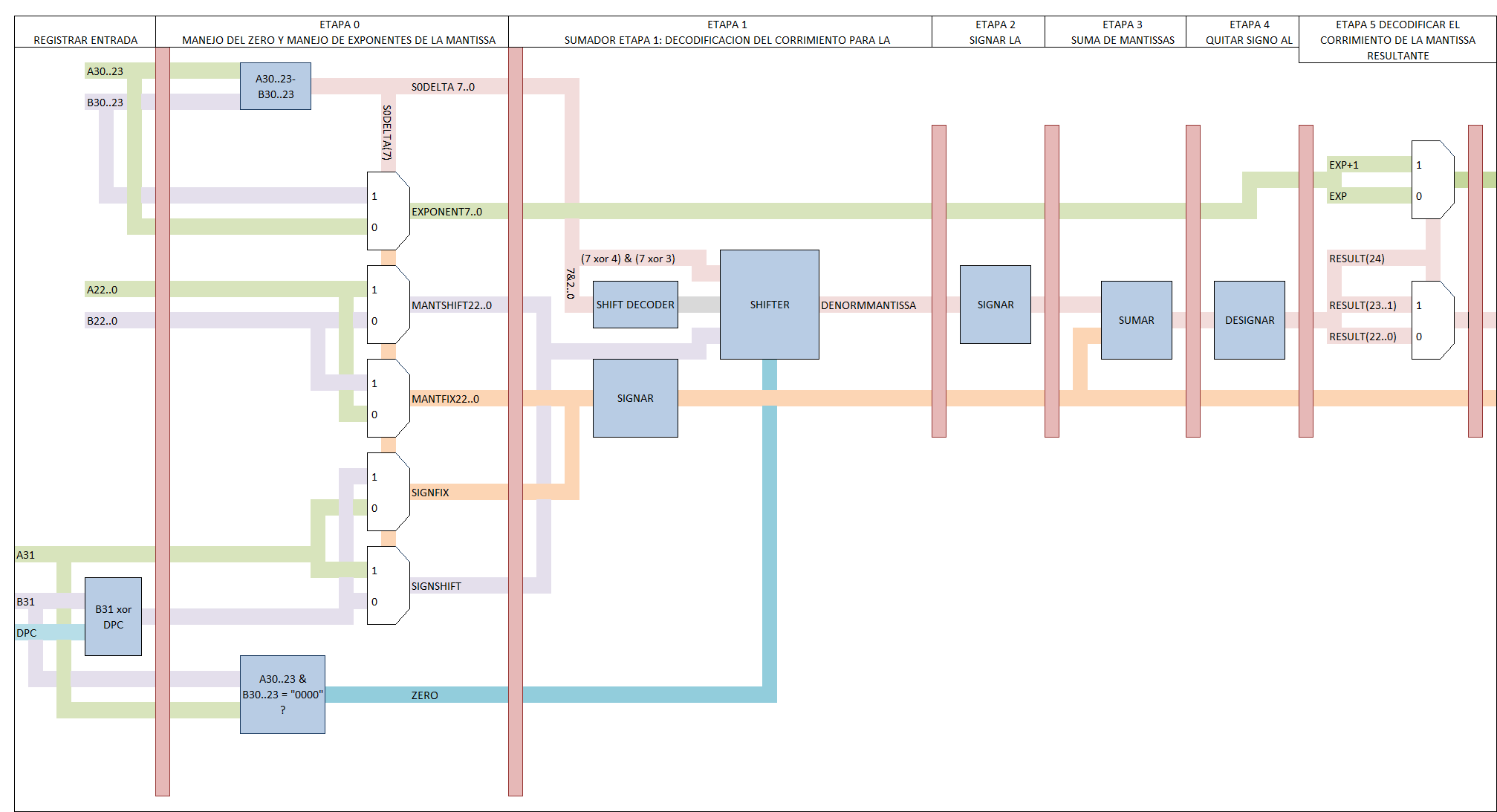


Ilustración . Diagrama de Bloques de la suma en punto flotante de 32 bits. Archivo RTL: fadd32.vhd

# Recursos Utilizados, Sumador Punto Flotante.

Los recursos utilizados por la implementación se encuentran detallados en la sección,

# Multiplicación 32 Bits, Punto Flotante IEEE 754 Precisión Simple.

# Abstracción Algebraica de la operación.

Se tiene los valores F0 y F1 cuya representación es en punto flotante:

*F0 = s0 2e0 f0*

*F1 = s12e1 f1*

Donde si toma el valor de -1 o 1 dependiendo del signo, ei es un valor entero y fi es una fracción en el rango [1.0,2.0).

El producto de los dos valores será el siguiente:

Si el valor de la fracción calculada *fres* es mayor o igual a 2.0, entonces se debe dividir por 2 y el valor del exponente se le debe sumar 1.

# Diagrama de Flujo.

|  |  |
| --- | --- |
|  | Etapa 0   * Suma de Exponentes * Multiplicación de Signos * Multiplicación de Mantisas |
| Etapa 1   * Normalización de la mantisa resultante. |
| Etapa 2   * Normalización del exponente resultante. |

Tabla . Diagrama de flujo que describe las etapas combinatorias del multiplicador en punto flotante.

# Diagrama de Bloques.

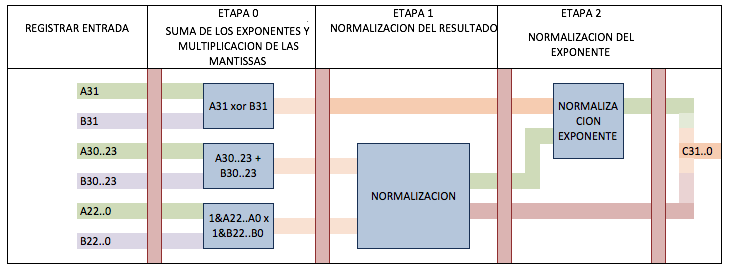


Ilustración . Diagrama de bloques del circuito de multiplicación en punto flotante IEEE754.

# Recursos Utilizados.

Los recursos utilizados por la implementación se encuentran detallados en

# Inversor, 32 Bits en Punto Flotante IEEE 754, Precisión Sencilla.

# Abstracción Algebraica de la Operación.

Se tiene el valor Fi cuya representación es en punto flotante:

*Fi = si 2ei fi*

Donde si toma el valor de -1 o 1 dependiendo del signo, ei es un valor entero y fi es una fracción en el rango [1.0,2.0).

El valor elevado a la -1 potencia sería:

Dado que si es 1 o -1, el valor de si-1 es si. El valor de el inverso de Fi es entonces:

Si el valor de la fracción calculada *fi-1* es menor a 1, entonces se debe multiplicar por 2 y el valor del exponente se le debe restar 1.

El circuito está calculado usando lógica combinatoria, excepto las expresiones y , las cuales utilizan una tabla de búsqueda, implementada en una memoria ROM de 1024 direcciones y 18 bits de ancho de palabra. El direccionamiento de esa tabla es con una palabra de 10 bits de ancho, que corresponden a los 10 bits más altos de la mantisa *fi*. Se ignoran los 13 bits restantes de la mantisa y por lo tanto habrá una pérdida de precisión. Usar mas bits habría requerido memorias más grandes y es posible aumentar la precisión aumentar la precisión mediante este método.

# Diagrama de Flujo.

er = -er

fr = memsqrimpar(fr[22..13])

Ilustración . Diagrama de Flujo Inversor 32 bits punto flotante IEEE754. Se cambia de signo al exponente y direccionando con los primeros 10 bits de la mantissa se calcula el valor de la fracción invertida.

# Diagrama de Bloques.

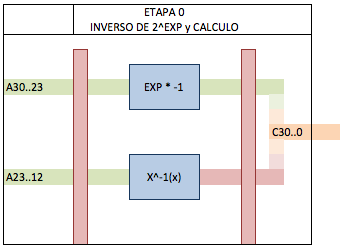


Ilustración . Diagrama de bloques del multiplicador flotante en 32 bits IEEE754. RTL: invr32.vhd

# Recursos Utilizados.

Los recursos utilizados por la implementación se encuentran detallados en el

# Raíz Cuadrada IEEE 754, 32 Bits Floating Point, Single Precision.

# Abstracción Algebraica.

Se tiene el valor F0 cuya representación es en punto flotante:

*Fi = si 2ei fi*

Al calcular la raíz cuadrada se asume que si vale 1, puesto que el dominio de la función raíz cuadrada es [0,∞) ei es un valor entero y fi es una fracción en el rango [1.0,2.0).

El valor elevado a la -1 potencia sería:

El valor del exponente resultante *ei*/2 debe ser un valor entero. Por lo tanto el valor de *ei* debe ser un número par. Si *ei* no es un número par, se requerirá de una normalización del exponente y la fracción. Finalmente el valor de la raíz cuadrada se calculará así:

El circuito está calculado usando lógica combinatoria, excepto las expresiones o y , las cuales utilizan una tabla de búsqueda, implementada en una memoria ROM de 1024 direcciones y 18 bits de ancho de palabra. El direccionamiento de esa tabla es con una palabra de 10 bits de ancho, que corresponden a el bit menos significativo del exponente y a los 9 bits más altos de la mantisa *fi*. Se ignoran los 14 bits restantes de la mantisa y por lo tanto habrá una pérdida de precisión. Usar mas bits habría requerido memorias más grandes y es posible aumentar la precisión mediante este método.

# Diagrama de Flujo

Expsq = Expr >> 1

fsq = memsqrimpar(fr[22..14])

Exp0 impar?

fsq = memsqrpar(fr[22..14])

Ilustración . Diagrama de Flujo Operación Raíz Cuadrada en formato IEEE754. El exponente se divide entre dos mediante el corrimiento de un bit a la derecha. Usado una de dos memorias, en función de la paridad del exponente de entrada, se cálcula la mantissa.

# Diagrama de Bloques

## 

Ilustración . Diagrama de bloques del multiplicador en 32 bits de punto flotante IEEE754.RTL: sqrt32.vhd

# Recursos Utilizados

Los recursos utilizados por la implementación se encuentran detallados en el

# Interfaz de Programación

Como se ha mencionado se considera al *RayTrac* un *Procesador de Flujos.* La función que cumple un procesador de este tipo es la de una función especializada o *Kernel.* A este *kernel* entran y salen flujos de datos, con datos por procesar en los flujos de entrada y con datos procesador en los flujos de salida.

Dado lo anterior son dos los aspectos que se pueden programar en el *RayTrac:* el *kernel* o función especializada a ejecutar y la fuente y destino de los flujos de entrada y salida respectivamente. La *acto* de programar se lleva a cabo mediante la escritura en ciertos registros ubicados en el *bloque de registros de programación y control de flujo*.

# Bloque de Registros de Programación y Control de Flujo.

El bloque de registros consiste en un conjunto de *16 egistros de 32 bits de ancho*, que sirven para programar y controlar el *RayTrac.*

A continuación se enumeran y describen dichos registros.

|  |  |  |
| --- | --- | --- |
| *Nombre/s* | *Descripción* | *Dirección* |
| *Reg\_ctrl* | *Registro de control: En este registro se llevan a cabo las configuraciones que definen la operación o kernel a ejecutar, la operación de el adaptador de formato de flujos de salida, la operación del adaptador de flujos de datos de entrada y se almacenan algunas banderas para soportar el control de flujo.*  *Más adelante se detalla este registro.* | *BASE+0* |
| *Reg\_Vx/Reg\_Vy/Reg\_Vz/Reg\_Sc* | *Estos registros almacenan el último vector registrado en el adaptador de formato de flujos de salida.* | *BASE+4 / BASE+8 / BASE+12 / BASE+16* |
| *Reg\_Scratch00* | *Registro Scratch, para realizar pruebas almacenar datos etc. En general se utiliza para depuración del hardware.* | *BASE + 20* |
| *Reg\_OutputCounter* | *Registro que cuenta el número de datos que han salido hacia el bus de datos externo. El programador puede escribir cualquier valor en este registro, perdiendo la cuenta que el sistema lleve.* | *BASE + 24* |
| *Reg InputCounter* | *Registro que al igual que OutputCounter, cuenta datos, pero de entrada. Es escribible por el programador.* | *BASE + 28* |
| *Reg FetchStart* | *Registro en el que se programa la dirección externa donde se almacena el bloque de flujo de datos de entrada.* | *BASE + 32* |
| *Reg SinkStart* | *Registro en el que se programa la ubicación externa a donde se direccionará el flujo de salida de datos* | *BASE + 36* |
| *Reg\_Ax/Reg\_Ay/Reg\_Az / Reg\_Bx/Reg\_By/Reg\_Bz* | *Registros en los que se almacena el último flujo de entrada “formateado” por el adapatador de formatos de flujo de entrada, es decir en estos registros se cargan el último par de vectores que entraron en el sistema.* | *BASE + 40/ BASE + 44/*  *BASE + 48/*  *BASE + 52/*  *BASE + 56/*  *BASE + 60* |

Tabla . Nombre, descripción y direcciones del los registros del bloque de registros de programación y control de flujo.

Los registros que se utilizan para programar el *kernel* usando la *interconexión decodificada de circuitos aritméticos* son el registro de control, reg\_ctrl, el registro de dirección de fuente de flujo de datos de entrada, reg\_fetchstart y el registro de dirección de escritura de resultados del flujo de datos de salida, reg\_sinkstart.

# Programación del *kernel* o función especializada y direcciones de almacenamiento de flujos de entrada y salida.

El *RayTrac* utiliza un grupo de circuitos aritméticos para ejecutar las operaciones que el programador configura. Para ordenar el camino que los datos recorren desde la entrada, hasta la salida y por el cual se van transformando, el sistema utiliza la *interconexión decodificada de circuitos aritméticos.* Este circuito decodifica función especializada o *kernel* configurado por el programador y está decodificación determina entonces el camino de datos dentro del *pipeline aritmético.*

Haciendo uso del reg\_ctrl, el programador configura el *kernel* o función especializada a ejecutar. A continuación se detallan los campos de este registro.

|  |  |  |
| --- | --- | --- |
| *Bit No.* | *Nombre (rw)* | *Descripción* |
| *0* | *cmb (rw)* | *Este bit describe la manera en que el adaptador de flujos de entrada carga o formatea los flujos de entrada. Este campo aplica para operaciones que requieren dos vectores A y B. Se ignora para operaciones unarias como la normalización de vectores o el cálculo de magnitud de vectores.*  *0: Todos los flujos de entrada se cargan secuencialmente en seis componentes, que conforman los 2 vectores A y B, de la siguiente manera: Ax, Ay, Az, Bx, By, Bz, Ax, Ay ….. y así sucesivamente. Una vez exista otro dato disponible entrante, empieza de nuevo la secuencia.*  *1: Todos los flujos de entrada se cargan secuencialmente en los 3 componentes que conforman el vector B, salvo los primeros 3 datos del flujo de entrada que se cargarán en el vector A, de la siguiente manera: Ax, Ay, Az, Bx, By, Bz, Bx, By, Bz, Bx, By, Bz, Bx…… y así sucesivamente.* |
| *[3:1]* | *dcs (rw)* | *Kernel o función especializada a ejecutar.*  *El valor de estos 3 bits codifica la operación a realizar y es función de la interconexión decodificada de circuitos aritméticos decodificarlos, para interconectar las operaciones aritméticas que definen un kernel.*  *000: Suma de Vectores. A+B.*  *001: Resta de Vectores. A-B.*  *011: Producto Cruz. AxB.*  *100: Producto Punto. A.B.*  *110: Magnitud y Normalización de Vectores. |A| y A\*1/|A|.*  *111: Multiplicación Simple Componente a Componente.* |
| *[5:4]* | *vtsc (rw)* | *Este campo define la tasa de adaptación de flujos de salida.*  *00 ó 10: Solo leer el componente vectorial de la cola de resultados 96:32. Se debe usar este valor cuando se quiere configurar un kernel cuyos resultados solo sean vectores, ejemplo: Producto Cruz.*  *01: Solo leer el componente escalar de la cola de resultados 32:32. Se debe usar este valor cuando se quiere configurar un kernel cuyos resultados son exclusivamente valores escalares, por ejemplo el producto punto o la magnitud vectorial.*  *11: Leer el componente vectorial y escalar de la cola de resultados 128:32. Cuando se selecciona la operación Magnitud y Normalización Vectorial (dcs=110), se debe tener en cuenta que en la cola de resultados se almacenaran cuatro valores por resultados: Los 3 componentes del vector normalizado y la magnitud del vector original. Es deseable leer los cuatro datos y en ese caso se utiliza el valor vtsc=11.* |
| *6* | *dma (rw)* | *0: El sistema funciona como un stream processor, al que se le configura un kernel o función especializada para ejecutar y una fuente de flujos de entrada y un destino para los flujos de salida. A este modo de le llama modo RayTrac.*  *1: El sistema funciona como un direct memory access, escribiendo bulks de datos utilizando los registro REG\_FETCHSART como dirección externa de lectura de datos y REG\_SINKSTART como dirección externa de escritura de datos. El tamaño del bulk se escribe en el campo de bits nfetch de este mismo registro.* |
| *[13:7]* | *Banderas: (r)* | *[ap : dp : pl : pp : wp : dc : fc]*  *Consultar el capítulo V. Control de flujo de datos y sincronización externa. Para información más detallada sobre las banderas a continuación.*  *ap: address pending flag.*  *1: Está pendiente por ser enganchada la dirección y tamaño del bulk programado del flujo de datos de entrada.*  *0: No hay pendientes en el momento enganches de dirección de flujos de entrada.*  *dp: data pending flag.*  *1: En el momento hay una transmisión de flujos de datos de entrada y no ha terminado.*  *0: En el momento está pendiente la entrada de flujos de entrada al sistema.*  *pl: parameter loading flag.*  *0: La máquina de estados del adaptador de formato de flujos de entrada se encuentra en el estado Ax.*  *1: La máquina de estados del adaptador de formato de flujos de entrada se encuentra en la mitad de la carga de algún vector, por lo tanto puede estar en cualquier estado menos en Ax.*  *pp: pipeline pending flag.*  *0: No se encuentra en el momento dato alguno transitando entre los circuitos aritméticos ni la interconexión decodificada de circuitos aritméticos en el momento.*  *1: Existen datos transitando en las etapas de los circuitos aritméticos o en la interconexión decodificada de circuitos aritméticos.*  *wp: write on memory pending flag.*  *0: No existen flujos de salida para ser direccionados hacia una dirección en el exterior del RayTrac.*  *1: Hay flujos de salida ya formateados pendientes por ser escritos en una dirección externa al RayTrac.*  *fc: fetch condition flag.*  *0: No existen direcciones de flujos de entrada pendientes por ser enganchadas o la tasa de tránsito de datos por el RayTrac es muy alta y no se deben traer dentro más flujos de datos de entrada.*  *1: Existen direcciones de flujos de datos de entrada pendientes por ser enganchadas y la tasa de tránsito de datos por el RayTrac es baja.*  *dc: drain condition flag.*  *0: No es prioritario escribir flujos de salida, afuera del RayTrac.*  *1: En el momento las condiciones del RayTrac en general indican que es prioritario escribir flujos de salida, afuera del RayTarc.* |
| *14* | *rlsc (rw)* | *0: Una vez se escribe el ultimo flujo de datos de salida entonces no se reinicia el adaptador de formato de flujos de entrada. Esto quiere decir que sin importar en que estado se encuentre la máquina de estados del adaptador de formato de flujos de entrada, conservará su estado.*  *1: Una vez se escribe el último flujo de datos de salida entonces se reiniciaría el adaptador de formato de flujos de entrada. Esto quiere decir que sin importar en que estado se encuentre la máquina de estados del adaptador de formato de flujos de entrada, volverá al estado AX.* |
| *15* | *rom(r)* | *0: Todos los registro se encuentran en modo lectura y escritura donde aplique.*  *1: Algunos registros se encuentran en modo escritura. Esto ocurre cuando el kernel está ejecutando operaciones. Los registros en dicha condición son:*   * *Reg\_OutputCounter* * *Reg\_InputCounter* * *Reg\_FetchStart* * *Reg\_SinkStart* * *Reg\_Ctrl* |
| *[30:16]* | *Nfetch (rw)* | *Número de datos que posee el siguiente flujo de entrada. Este valor se encuentra en cero todo el tiempo. Una vez se le asigne un valor distinto a cero, el bit rom se colocará en cero y el sistema comenzará a ingresar los flujos de entrada procesarlos y sus resultados escribirlos. Max Bulk Size=215-1.* |
| *31* | *Irq(rw)* | *Una vez se escriba el último dato del último flujo de resultados de salida, este bit se colocará en uno para señalar el evento del fin del procesamiento del flujo.*  *La escritura de este bit SOLO es posible una vez se haya activado la interrupción y SOLO se puede escribir en él el valor cero. Esto sirve como mecanismo para acusar recibo desde el proceso al que se le notifica la interrupción.* |

Tabla . Registro de Control y Programación REG\_CTRL.

# Interconexión decodificada de circuitos aritméticos.

Una vez se programa el registro de control con el *kernel* que se quiere ejecutar en el *pipeline aritmético*, se configuran las conexiones de las salidas y las entradas de los circuitos del *pipeline aritmético.* A continuación se detalla la configuración de las interconexiones.

|  |  |
| --- | --- |
| DCS | Circuito Interconectado |
| 011: AxB |  |
| 000 y 001: A+B y A-B |  |
| 110:  |A| y A/|A| |  |
| 100:  A.B |  |
| 111:  A\*B |  |

Tabla . Detalle de los circuitos con la interconexión decodificada, a partir de los bits DCS del registro de control, que contienen el kernel a ejecutar codificado.

En la , se detalla los cinco tipos de interconexiones decodificadas a partir del campo de bits en el registro de control *dcs.*

## Producto Cruz (011).

Los componentes de los vectores A y B, son distribuidos a través de la *interconexión decodificada de circuitos aritméticos* hacia los seis multiplicadores del sistema. Transcurren 3 ciclos mientras los datos transitan por los multiplicadores. Los 6 productos salen a la interconexión y entran a los 3 sumadores configurados en modo restador. Los 3 resultados de las restas componen el vector resultante, que transitan a la interconexión decodificada y por ahí a la salida del *pipeline aritmético.*

## Suma (000) Resta (001)

Los 6 componentes de los vectores entran a la interconexión decodificada y de ahí son dirigidos como sumandos a los 3 sumadores del sistema. Los datos transitan a través de las 7 etapas de los sumadores. Los resultados de los 3 sumadores conforman el vector resultante y salen del *pipeline aritmético* vía la interconexión decodificada de datos.

## Normalización y o Magnitud (110)

El vector a operar entra en la *interconexión decodificada de caminos* y de ahí se bifurca en dos caminos. En el primer camino entra a la cola de sincronización *q1* y de ahí transitará hasta encontrarse con el resultado del cálculo de la magnitud de si mismo 21 ciclos de reloj después.

En el segundo camino los 3 componentes del vector, *x, y* y *z,* se dirigen a los 2 puertos de los multiplicadores *p0, p1*  y *p2* respectivamente.

Los datos transitan por los multiplicadores, mientras que el vector original transita por la cola de sincronización. De los 3 productos, 2 transitan hacia el sumador *as0* y el otro se dirige hacia la cola de sincronización *q0.* Mientras las suma *as0* toma lugar, el tercer producto transita por la cola de sincronización *q0.* Cuando el resultado de la suma *as0* se encuentra listo, este y el tercer producto que es “halado” de la cola, se encuentran en la *interconexión decodificada* y transitan hacia el sumador *as1,* calculando de esta manera la magnitud cuadrada del vector.

La magnitud cuadrada del vector entra al bloque de raíz cuadrada, el resultado de la raíz, que corresponde a la magnitud del vector inicial, transita por la conexión decodificada y se bifurca en dos caminos: un camino entrega el la magnitud como resultado y el otro utiliza la magnitud para calcular el vector normal.

## Camino 1 : Magnitud.

Una vez ha sido calculada la magnitud del vector este entra en la cola de sincronización *q2.* El objetivo de esta cola de sincronización es que el valor de la magnitud del vector se encuentre a la salida al mismo tiempo que ahí se encuentre el vector normalizado.

En la , se observa que después del bloque 21, que corresponde al circuito combinatorio que calcula la raíz cuadrada del cuadrado de la magnitud del vector, el resultado transita hacia la cola desde la *interconexión decodifica* desde el bloque 22 y por esta transita durante 5 ciclos de reloj, para finalmente entrar de nuevo a la *interconexión decodificada de circuitos aritméticos*  en el bloque 27.

## Camino 2: Vector Normal.

Una vez ha sido calculada la magnitud del vector este entra en el bloque 23 en la *interconexión decodificada* *de circuitos aritméticos,* al mismo tiempo que lo hace el vector inicial que estuvo transitando por la cola de sincronización *q1.*

De ahí cada el valor de la magnitud del vector entra en uno de los puertos de cada uno de los multiplicadores *p3, p4* y *p5.* De la misma manera los componentes del vector de entrada, *x, y* y *z,* entran a uno de los puertos de los multiplicadores *p3, p4* y *p5.* Tres ciclos de reloj después los 3 productos de cada uno de los multiplicadores *p3, p4* y *p5,* corresponden a los componentes *x, y* y *z*, del vector normalizado.

Habrán transcurrido 27 ciclos de reloj desde que el vector ingresó al *pipeline aritmético* hasta el momento en que en la *interconexión decodificada de circuitos aritméticos* aparecen su magnitud y el vector normalizado.

## Producto Punto (100):

Los vectores A y B ingresan a la *interconexión decodificada de circuitos aritmético* y cada pareja (*Ax,Bx), (Ay,By)* y *(Az, Bz),* se dirige a los 2 puertos de los multiplicadores *p0, p1* y *p2* respectivamente.

Los datos transitan por los multiplicadores, mientras que el vector original transita por la cola de sincronización. De los 3 productos, 2 transitan hacia el sumador *as0* y el otro se dirige hacia la cola de sincronización *q0.* Mientras las suma *as0* toma lugar, el tercer producto transita por la cola de sincronización *q0.* Cuando el resultado de la suma *as0* se encuentra listo, este y el tercer producto que es “halado” de la cola, se encuentran en la *interconexión decodificada* y transitan hacia el sumador *as1,* calculando de esta manera el producto punto entre los vectores A y B.

Una vez ha sido calculada el producto punto entre los vectores A y B este entra en la cola de sincronización *q2.* No tiene ninguna utilidad entrar en la cola *q2.* Esta cola de sincronización se encuentra allí para efectos de sincronización de la operación de normalización/magnitud, sobre la operación de producto punto no tiene efecto alguno más que aumentar la latencia desde que ingresan los vectores al *pipeline aritmético* hasta que se obtiene un resultado, sin que esto tenga efecto positivo o negativo en el desempeño[[4]](#footnote-4).

## Producto Simple, componente a componente (111):

Los vectores A y B ingresan a la *interconexión decodificada de circuitos aritmético* y cada pareja (*Ax,Bx), (Ay,By)* y *(Az, Bz),* se dirige a los 2 puertos de los multiplicadores *p0, p1* y *p2* respectivamente. Los valores de los productos transitan hacia el exterior del *pipeline aritmético* vía la *interconexión decodificada de circuitos aritméticos.*

# Secuencia de programación, y configuración de un kernel.

El programador debe seguir la siguiente secuencia de pasos.

1. Escribir en el registro reg\_fecthstart, la dirección de memoria donde se encuentra el flujo de datos de entrada, o sea los parámetros para operar.
2. Escribir la dirección donde se van a guardar los resultados en el registro reg\_sinkstart. El programador es responsable por el segmento de memoria donde se vaya a escribir esté disponible, tenga capacidad suficiente y esté asegurado para evitar posibles *race conditions*.
3. Escribir en el registro de control reg\_ctrl: qué función o *kernel* configurar en el *pipeline aritmético* campo de bits *dsc*, si la función va a retornar resultados vectoriales, escalares o ambos, en el campo de bits *vtsc,* el modo de carga de operandos si es combinatorio o no en el bit *dsc.*
4. Escribir el número de datos a operar o tamaño del bulk, en el registro reg\_ctrl. Si son 2 vectores los que se van a ingresar el número es 6, si son 100 vectores los que se van a operar el número es 300. Tener en cuenta que el formato de los valores almacenados debe ser punto flotante sencillo IEEE754, lo que implica que cada uno ocupa 4 bytes de memoria. Para especificar este valor se debe utilizar el campo de bits nfetch. *El kernel comenzará a procesar datos tan pronto se especifique el bulk size.* Por eso debe ser este valor el último programado en la secuencia.

# Ejemplo

El siguiente código refleja la secuencia explicada. El objetivo es ejecutar 300 operaciones de producto punto.

*#define reg\_fetchstart 32*

*#define reg\_sinkstart 36*

*#define reg\_ctrl 0*

*#define dotprod 0x6*

*#define vectorial 0x10*

*#define scalar 0x08*

*#define vect\_n\_scalar 0x18*

*#define bulk\_sizer(x) (x<<16)*

*Uint \* ptr\_to\_raytrac = (uint\*) raytrac\_0\_base;*

*Float input\_stream[600];*

*Float output stream[200];*

*Uint index;*

*/\* Crear los vectores a operar \*/*

*for (i=0:i<600;i++)*

*input\_stream[i]=rand(); //Numero al azar entre 0 y 1.*

*/\*Paso Numero 1: Escribir la dirección de flujo de entrada en el registro fetch start \*/*

*ptr\_to\_raytrac[reg\_fetchstart] = input\_stream;*

*/\*Paso Numero 2: Escribir la dirección del flujo de salida en el registro sink start \*/*

*ptr\_to\_raytrac[reg\_sinkstart] = output\_stream;*

*/\*Paso Numero 3: Configurar el kernel y los adaptadores de formato : producto punto, devolviendo valores escalares únicamente \*/*

*ptr\_to\_raytrac[reg\_ctrl] = (dot\_prod+vectorial)<<1;*

*/\*Paso Numero 4: Configurar el bulk size\*/*

*ptr\_to\_raytrac[reg\_ctrl] += bulksizer(300);*

*/\*Paso 5 esperar a que se termine de ejecutar los 300 productos punto.\*/*

*while!!(ptr\_to\_raytrac[reg\_ctrl] & 0x80000000));*

El ejemplo anterior es una forma muy básica de programar el *RayTrac.* Uno de los productos de este proyecto es una API que haga realmente transparente al usuario la secuencia de programación y se enfoque más en la aplicación que está realizando.

La última línea hace referencia a un *polling* del bit de interrupción, esto no es una buena práctica de programación. Lo que se debe hacer es utilizar la interfaz de programación del NIOS II, procesador que soporta Interrupciones internas o controlador externo de interrupciones y escribir una *ISR o rutina de servicio de interrupción.*

En el capitulo diez que describe la implementación del trazador de rayos hay ejemplos de código, sin embargo junto con el software driver del RayTrac se añaden ejemplos. Driver y ejemplos son anexos a este documento.

# Control de Flujo

El *RayTrac* posee recursos de procesamiento con una capacidad definida. No se puede cargar con datos para operar indefinidamente, sin entregar resultados porque habría perdida de datos.

En termino de control de flujo y sincronización, el *RayTrac* se puede modelar como un conjunto de *colas* en secuencia una detrás de la otra en general.

Para el análisis de control de flujo y sincronización se hacen dos análisis: uno de flujo de datos a través de las *colas* y un análisis de carga / descarga de las mismas*.*

Existen varios procesos que actúan sobre estas *colas:* leyendo en ellas, halando datos ó escribiendo en ellas, insertando datos. A ellas

Existen también abstracciones las cuales se definen en este documento como *colas abstractas.*

En general, el uso de funciones combinatorias cuyas entradas son las señales de sincronización de las colas abstractas y normales, permiten la detección de eventos, con los cuales los procesos administrativos *toman decisiones,* sobre varios aspectos, como el siguiente estado de dichos procesos, la activación de proceso de escritura o de lectura o la generación de interrupciones para señalizar eventos particulares a procesos externos al *RayTrac.*

A continuación se realiza una descripción detallada de estos modelos, el modelo de flujo de datos y el modelo de sincronización de datos.

# Modelamiento por *colas.*

El sistema puede ser visto como un conjunto de colas que actúan, en general, de manera secuencial: la salida de una cola está relacionada con la entrada de la otra y así sucesivamente.

Se modela el control de flujo de datos y sincronización en el *RayTrac* a partir de colas normales y colas abstractas.

# Colas Abstractas.

Aunque es evidente que no todos los circuitos del *RayTrac* son colas, realizar la abstracción del control de flujo en términos de las mismas, simplifica el diseño y la implementación, puesto que las colas o *fifo,* han sido utilizadas como elementos de sincronización de recurso de computo o almacenamiento.[[5]](#footnote-5)

Una cola en su conjunto de recursos posee memoria, para el almacenamiento de datos y señales de sincronización. Se *define* como *cola abstracta,* aquella que en su conjunto de recursos solo posee señales de sincronización. El objetivo de utilizar una cola abstracta es utilizar sus señales de sincronización para llevar a cabo tareas de esta naturaleza. Es irrelevante con aquello que se *supone se encuentra almacenado.*

El siguiente ejemplo utiliza el escenario de uso de una *cola abstracta.* Suponga que se desea ejecutar una transferencia *DMA* (*Acceso Directo a Memoria*) para transferir un bloque de 1 Kbyte de memoria desde la dirección de memoria *af* hasta la dirección de memoria final *ad* .

Lo que ocurre es que en algún momento el Hardware controlador de memoria va a recibir la dirección fuente *add\_fuente (flujo de entrada)* el tamaño de ese bloque y la dirección de destino *add\_destino (flujo de salida).* En ese instante, antes de comenzar la transferencia se constituye una *cola abstracta* *de direcciones pendientes por ser leídas* con las siguientes características.

* La cola de manera abstracta almacena direcciones para ser leídas. *Dicho almacenamiento no existe como tal, es la abstracción de la* ***existencia tareas*** *pendientes por realizar, en este caso las direcciones a ser leídas.*
* La cola posee señal *q,* la cual es abstraída con la primera dirección del flujo de entrada *af.*
* Si el controlador de hardware puede hacer transferencia de bloques máximo de un 1Kbyte podemos abstraer la señal *full* = 1, afirmando que la *cola abstracta de direcciones pendientes por ser leídas está llena.*
* Dada la anterior condición podemos afirmar que la cola no se encuentra vacía, o sea que *empty = 0.*
* De igual manera podemos configurar el contador de espacio usado en la cola, *used = 1024.*
* Cuando el bus de direcciones enganche la primera dirección, *add\_fuente*, esta se descartará de la cola, puesto que ya no se necesita y se debe dar paso a la siguiente dirección, *add\_fuente+1*. Para ello la señal de lectura por descarte *rdack,* se coloca en 1. Dado que la cola es *abstracta* entonces, se actualiza la *abstracción:* se suma 1, a la siguiente dirección, para obtener el siguiente dato de la *cola abstracta, add\_fuente+1,* la señal *full* pasa a 0, la señal *empty* continua en 0 y el contador de espacio usado, *used,* resta 1, dejando su valor en 1023.

Es fácil para cualquier sistema de arbitramento, o control de flujo, que necesite hacer lectura de datos en bloque o *bulks*, utilizar la *cola de abstracción de direcciones pendientes por ser leídas,* para saber si comienza un enganche de datos en la memoria o no.

Aunque no exista la cola como tal, las señales de sincronización de la *cola abstracta* indican al proceso controlador del flujo de datos, que el controlador de memoria no podría enganchar más direcciones para leer, puesto que la señal de la *cola abstract*, *full,* se encuentra en 1, otorgando el estatus de *ocupado* al controlador de memoria, hasta que el sistema de direccionamiento, capture la dirección que hay en la salida de la *cola abstracta*

# Colas de sincronización y control de flujo.

El *RayTrac* como se observó en los capítulos 3 y 4, utiliza en su operación de ejecución de *kernel,* varias colas haciendo uso de su capacidad de almacenamiento y de las señales de sincronización.

La siguiente tabla destalla las colas normales y abstractas que son usadas en el control de flujo:

|  |  |  |
| --- | --- | --- |
| COLA | DESCRIPCION | ABSTRACTA |
| *Pipeline aritmético : q1* | La función de estas colas es identificar si existen datos transitando en el pipeline aritmético. | NO |
| *Pipeline aritmético: q2* | NO |
| *Adaptador de Formato de Flujos de Salida: qresultados* | La función de estas dos colas es identificar si existen resultados pendientes por ser escritos en un flujo de salida. | NO |
| *Adaptador de Formato de Flujos de Salida: qoutb* | NO |
| *Cola de direcciones pendientes por enganchar.  qadd* | *Cola abstracta* formada por la escritura en el campo de bits del registro REG\_CTRL en el campo NFETCH. Se descartará las direcciones conforme se vaya enganchando en el bus de direccionamiento. | SI |
| *Cola de datos pendientes por descargar.  qfetch* | *Cola abstracta* formada por la escritura en el campo de bits del registro REG\_CTRL en el campo NFETCH. Se descartará los datos cuando el bus de datos de entrada notifique que el dato del ciclo de reloj presente es valido. | SI |

Tabla . Colas usadas en la sincronización y control de flujo del RayTrac.

Las colas de la , son las colas que utilizan los procesos administrativos del *RayTrac* para realiza el control de flujo y la sincronización del sistema.

# Eventos detectados por el sistema de sincronización y control de flujo.

Los siguientes son los eventos que se pueden generar:

|  |  |  |
| --- | --- | --- |
| EVENTO | EVENTO(S) | SEÑALES INVOLUCRADAS |
| Zero Transit | No hay datos transitando en el RayTrac, no hay direcciones pendientes por cargar en el bus de direcciones y no hay datos pendientes por bajar de la memoria. | *PA: q1\_empty, q2\_empty.*  *AFFs: qresultados\_empty, qoutb\_empty* |
| Write Pending | Hay datos en la cola de salida. | *AFFs: qoutb\_empty* |
| Pipeline Pending | Hay datos transitando en el *pipeline aritmético* o en la cola de resultados. | *AP: q1\_empty, q2\_empty*  *AFFs: qresultados\_empty* |
| Fetch Data Pending | Hay datos pendientes por ser leídos desde una dirección externa. | *qfetch\_empty* |
| Load Address Pending | Hay direcciones pendientes por enganchar en el bus de direcciones. | *qadd\_empty* |
| Flood Condition | Evento que indica que el sistema se encuentra apto para comenzar una transacción de descarga de datos desde una dirección externa al *RayTrac,* esto significa que hay pocos datos transitando en el *RayTrac* y hay direcciones pendientes por cargar. | *qfetch\_empty*  *AFFs: qoutb\_almost\_full* |

Tabla . Tabla de eventos para control de flujo y sincronización.

|  |  |  |
| --- | --- | --- |
| EVENTO | EVENTO(S) | SEÑALES INVOLUCRADAS |
| Drain Condition | Evento que indica que el sistema se encuentra listo para comenzar una transacción de escritura de resultados en una memoria externa, esto significa que hay una cantidad significativa de datos transitando por el *RayTrac* que hay que evacuar o que, si hay pocos datos transitando en él, la única cola con datos sea la cola *qoutb.* | *qadd\_empty*  *qfetch\_empty,*  *AFFs: qoutb\_empty, qoutb\_almost\_empty*  *AP: q0\_empty, q1empty* |

Tabla . Tabla de eventos para control de flujo y sincronización. (Cont.)

# Procesos Administrativos de las Colas.

Son los procesos administrativos de las colas, los encargados de ejecutar tareas pequeñas del control de flujo y sincronización.

Estos utilizan las señales de sincronización de las colas normales y las colas *abstractas,* para la *toma de decisiones*, sobre si se descarga o carga una cola de sincronización.

Los administradores de estos recursos son 5 procesos encargados de administrar el contenido de estas colas. A cada cola le corresponden 2 procesos, uno que la carga con datos y el otro que la descargar, descartando datos de la misma. Los procesos utilizan la detección de eventos para activar estas cargas o descargas de datos desde las colas.

Los procesos son: *Estado SOURCE*, *Estado SINK*, *Cadena UpLoad*, *Cadena de Sincronización* y *Avalon Memory Mapped Master Interface.*

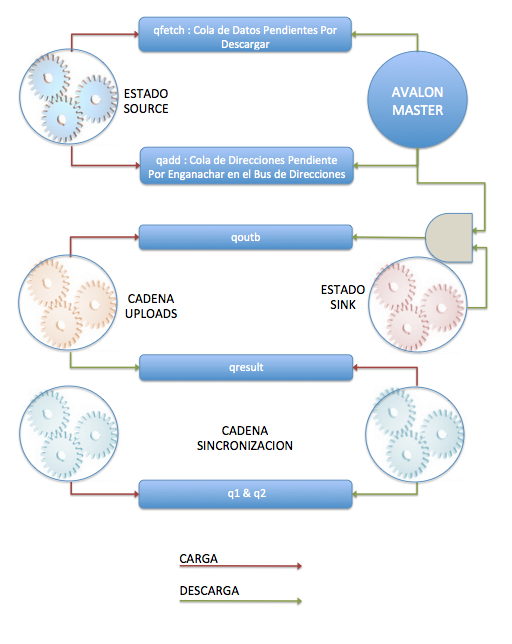
**

Ilustración . Procesos Administrativos y Colas, este diagrama no representa el flujo de datos, representa como están distribuidos los procesos administrativos, para cargar o descargar las colas.

En la se detalla qué colas son descargadas por qué procesos, mediante las flechas de color verde y qué colas son cargadas por qué procesos. Nótese que una cola en particular, debe tener un proceso que la descargue y otro que la cargue, para poder balancear su carga, es decir garantizar que no quede vacía perdiendo desempeño el sistema entero y que tampoco quede llena, con datos necesitando entrar en ella y queden descartados.

En las siguiente secciones se detalla cada uno de los procesos y como interactúa con las colas que le corresponde.

# Cadena de Sincronización.

El *pipeline aritmético* viene antecedido por el *adaptador de formatos de flujos de entrada.* Resulta que el adaptador no pueden entregar ni recibir datos a una tasa de flujo de datos por ciclo de reloj. El ritmo con el que entregan y reciben datos dependen de condiciones externas al *RayTrac.* Esto quiere decir que no *siempre* van a existir datos *validos* en la entrada del *pipeline aritmético:* a veces habrá vectores cargados en la entrada y a veces no.

Sin embargo *físicamente* habrá un valor presente en la entrada del *pipeline aritmético* que corresponde a un valor *inválido.* Dado que los registros no cuentan con una señal de *enable* conectado a un proceso de control de flujo que los gobierne, dicho valor entrará al *pipeline aritmético* en el siguiente ciclo de reloj.

Hay razones para no implementar una señal de *enable* propagada por todo el *pipeline aritmético.* El efecto neto de utilizar una *única* señal de *enable* para detener o permitir el flujo de los datos es un alto *fan out* y un posible exceso en el uso de recursos de interconexión así como aumento en el tiempo de compilación, análisisy *fitting.* Por lo anterior se ha descartado el uso de una señal de esa naturaleza.

Sin señales de *enable,* se infiere que el *pipeline aritmético* se ha diseñado para que *siempre* permita el paso de los datos al siguiente circuito combinatorio o a la siguiente fase de la i*nterconexión decodificada*.

No se encuentra implementado un mecanismo que niegue el flujo de datos inválidos, lo que implica que los datos que no son *validos* serán enganchados, cada ciclo de reloj, en el *pipeline aritmético.* Esto genera dos problemas, el primero es que se puede llenar la cola de salida del *adaptador de formato de flujo de datos de salida* con basuray el segundo problema consiste en que las colas de sincronización *q0 q1 y q2* se llenarían con basura. La consecuencia de estos dos problemas es la perdida de datos validos a la entrada de las colas que se encontrarán llenas de basura y no tengan espacio para alojarlos.

Para evitar el agotamiento de las colas de sincronización del *pipeline* y la cola de salida del *adaptador de formato de flujo,* se implementa un mecanismo de sincronización que señale que datos son validos o inválidos.

Los operandos inválidos entrarán al *pipeline aritmético,* serán operados, obtendrán resultados y llegarán al final del *pipeline aritmético.*

Dada la naturaleza del *pipeline aritmético,* en la que los circuitos aritméticos reciben operaciones distintas cada ciclo de reloj, podríamos considerarlo un recurso agotable en términos de desempeño. No de almacenamiento. Los datos inválidos que transiten por estos circuitos no estarán usando recursos que en el momento estén necesitando otros datos que si sean validos.

Sin embargo, las colas de sincronización de resultados intermedios *q0, q1* y *q2* si son agotables en espacio y no pueden ser utilizados por elementos inválidos, porque aunque una de sus funciones es servir de mecanismo de almacenamiento para resultados intermedios, su otra función es señalar eventos para soportar el control de flujo y la sincronización del *RayTrac* entero, en particular señalan si existen o no datos validos transitando por el *pipeline aritmético,* ver Por lo tanto en las colas de sincronización *q0,q1 y q2,* no pueden entrar datos inválidos. Un mecanismo de escritura y lectura, a la entrada y salida de estas colas, debe discriminar entre datos validos e inválidos.

Para tal fin se define una *cadena de sincronización*. A continuación se ilustra de qué manera funciona esta cadena, tomando como ejemplo el *kernel* para calcular producto punto:

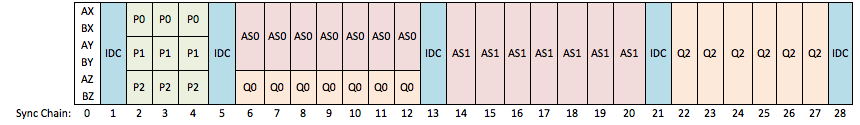


Ilustración . Detalle del pipeline aritmético configurado para ejecutar un kernel de producto punto. La frontera entre el adaptador de formato de flujos de entrada y el pipeline aritmético se encuentra entre los registros 0 y 1.

En la parte inferior hay una cadena de números llamada *Sync Chain.* Es la *cadena de sincronización*. La cadena posee 29 posiciones pero la primera posición, la posición 0, no pertenece al *pipeline aritmético,* esta posición se encuentra en el *adaptador de formato de flujos de entrada.*

Cada posición de la cadena tiene un valor *“v”* o *“nv”*. Si la posición *n* tiene el valor *v*, quiere decir que los circuitos combinatorios que en la ilustración se encuentren *“encima”* de la posición *n,* serán considerados validos.

La *cadena de sincronización* es el proceso administrativo que tiene la función de cargar las colas *q1* y *q2* de sincronización del *pipeline aritmético* con datos.

La cadena de sincronización también está encargada de escribir datos en la cola *q0.* Sin embargo la cola *q0* no participa en el proceso de control de flujo y sincronización general. Sirve para almacenar el valor intermedio del tercer producto del producto punto.

En el caso del *kernel* para ejecutar producto punto, debe haber un flujo con 6 valores, correspondiente a los 3 componentes de los 2 vectores a operar, para que la posición cero sea considerada valida. Un ciclo de reloj después se tendrá el siguiente escenario:

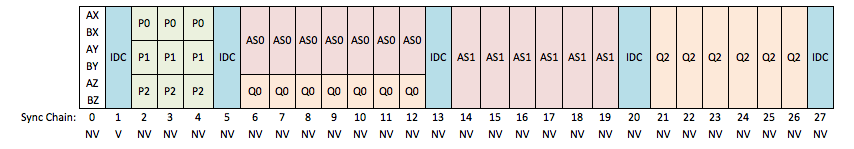
**

Ilustración . Kernel producto punto y cadena de sincronización, un ciclo después de que ingresan 2 vectores validos al pipeline aritmético.

En la , se observa cómo se consideran validos los datos que se encuentran en la primera etapa de la *interconexión decodificada.* Cuatro ciclos después se tendrá el siguiente panorama:

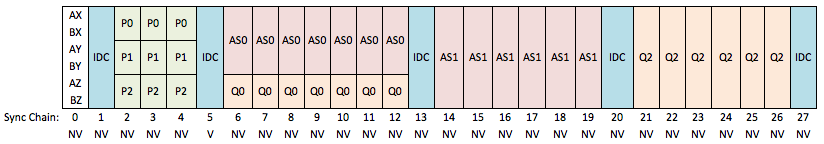


Ilustración . Cuatro ciclos de reloj después de estado en la .

La ilustración muestra que la posición 5 es la valida, a diferencia de la situación en la , no solo el valor *v* cumple la función de indicar que los valores en la interconexión son validos, adicionalmente el valor *v* habilitará la escritura de datos dentro de la cola *q0.* Usando este mecanismo se previene la entrada de datos “basura” en la cola de sincronización y solo de datos validos.

8 Ciclos de reloj después encontramos el siguiente estado:

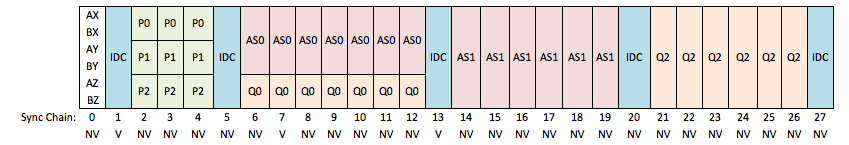


Ilustración . Ocho ciclos de reloj después de estado en la .

La posición 13, la posición 7 y la posición 1 están señaladas como posiciones validas. Lo primero que se destaca es la entrada cada 6 ciclos de reloj de un nuevo par de vectores validos al *pipeline aritmético.*

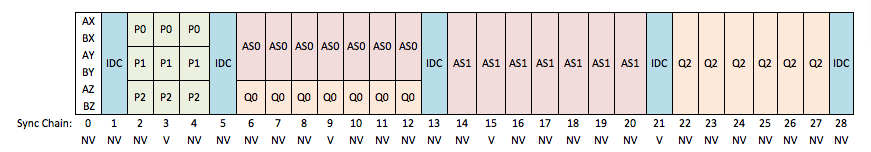
Esta condición de 6 ciclos de reloj por cada par de vectores depende de la velocidad con la que el *AFFe* carga los parámetros en la entrada del *pipeline aritmético.* Dado que el *ancho de banda* es apenas de un valor (32 bits) por ciclo de reloj, cargar 6 datos tomará 6 ciclos de reloj.

En este caso la validez de la posición 13 del *kernel,* cumple una función especial, descartar de la cola *q0* el valor que había entrado 8 ciclos de reloj previos. De esta manera se libera espacio para nuevos datos validos que requieran entrar en la cola.

El dato que se descarta de la cola *q0*, no se pierde porque desde que entró*,* se puede asumir que dos ciclos de reloj después se encontrará en la posición 13 puesto que la cola está configurada en modo *Show Ahead.*

*Modo Show Ahead* significa que el primer dato que entró en la cola será *visible* a la salida de la cola, sin necesidad de una señal de lectura que habilite su salida. La señal de lectura, *rd,* se reemplaza por una señal de descarte, *ack,* cuya función es descartar el valor visible de la cola una vez no sea requerido en ella[[6]](#footnote-6).

8 ciclos y 15 ciclos después encontramos los siguientes estados.



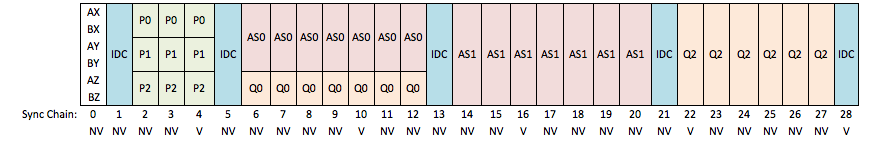


Ilustración . De Arriba a abajo, 8 y 15 ciclos después de estado de la ilustración 16.

8 ciclos después, en la posición 21 tenemos el resultado de la operación de producto punto, el valor valido habilita la escritura de la cola *q2*, para el ingreso del producto punto en ella.

15 ciclos después en la posición 28 tendremos el valor del producto punto listo para salir del *pipeline aritmético* vía la i*nterconexión decodificada* hacia la cola de resultados en el *adaptador de formato de flujos de salida.* La señal *v* de la cadena de sincronización habilitará dicha escritura.

El mecanismo de lectura y descarte ilustrado con la cola *q0* y *q2* aplica para la cola *q1* y para la escritura de la cola de resultados que se encuentra en el *adaptador de formato de flujos de salida.* La configuración en modo *Show Ahead,* es la misma para *todas* las colas que se implementan en el *RayTrac.*

## Implementación de la cadena de sincronización.

La *cadena de sincronización* se puede implementar con un registro de corrimiento de 28 bits y la entrada al primer registro la controla el circuito *adaptador de formato de flujos de entrada* una vez termina de cargar los parámetros de entrada completamente.

## Diagrama de Bloques.

El siguiente diagrama de bloques explica lo implementado:

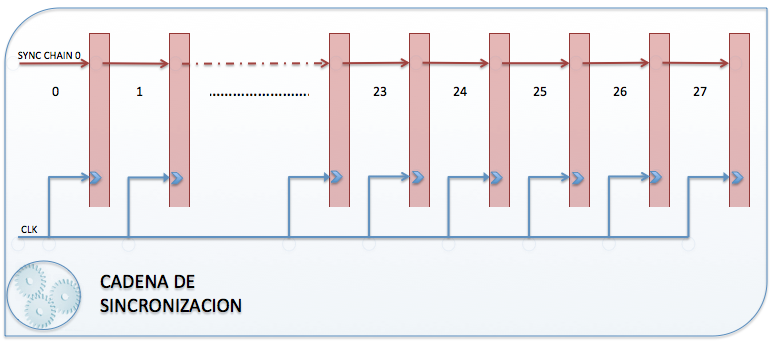


Ilustración . Cadena de sincronización del pipeline aritmético.

La implementación es un registro de corrimiento, la señal *Sync Chain 0*, es controlada por el *adaptador de formato de flujos de entrada* y una vez estén cargados todos los valores requeridos para hacer una operación, se colocará en ‘1’ esta señal. De esa manera se validan los datos que en conjunto comienzan a transitar con la señal de la cadena a lo largo del *pipeline aritmético.*

## Recursos Utilizados.

Los recursos utilizados para implementar la *cadena de sincronización*, están detallados en el *Capítulo VIII. Implementación e Integración del Sistema.*

# Cadena de *upload* o carga.

El *pipeline aritmético* entrega resultados a la cola de resultados del *adaptador de formato de flujo de salida,* en tres posibles formas.

* Resultado Vectorial, 3 componentes.
* Resultado Escalar, 1 valor.
* Resultado Vectorial y Escalar, 3 componentes 1 un valor.

Esto se debe a la naturaleza de los *kernel* que se pueden configurar usando la *interconexión decodificada.* Hay operaciones que entregan solo resultados vectoriales, por ejemplo el producto cruz y otras entregan solo resultados escalares, por ejemplo producto punto.

Sin embargo cuando se ejecuta la operación para calcular una normal, de un vector sucede que la *interconexión decodificada* permite entregar el resultado del cálculo de la magnitud del vector operando. Por lo tanto se puede escoger cuando se ejecuta la normalización del vector, si se quiere recibir exclusivamente el vector, exclusivamente la magnitud, o si se quiere recibir ambos.

Para configurar que tipo de resultado se quiere se configura el registro de control del *RayTrac, REG\_CTRL,* específicamente los el campo de bits *vtsc,* ver

Surge un problema. El ancho de banda de salida del *RayTrac,* cuando el mecanismo externo le otorga el acceso, para escribir resultados en alguna dirección es de 32 bits solamente.

Es por esto que se implementó el *adaptador de formato de flujo de salida,* para hacer la carga en el bus de externo, de datos 4:1 o 3:1 o 1:1. El adaptador cuenta con la *cola de resultados* con los datos de los resultados calculados por el *pipeline aritmético* y con un buffer a la salida la *cola de resultados.*

Cada vez que ocurre el evento de disponibilidad de un resultado se deben escribir los 4 ó 3 ó único datos en la cola de resultados de manera secuencial y en el siguiente orden:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1º | 2º | 3º | 4º | *vtsc* | *Observación* |
| X | Y | Z | Escalar | 11 | Resultados Vectoriales & Escalares |
| X | Y | Z |  | 00/10 | Resultados Vectoriales |
| Escalar |  |  |  | 01 | Resultados Escalares |

Tabla . Operación de la Cadena UpLaod.

La operación ilustrada en la tabla ilustra la operación de la *Cadena UpLoad*. La *Cadena UpLoad* es el proceso administrativo que carga y descarga las colas de resultados y de salida de datos, respectivamente, ver .

## Implementación de la *cadena de upload.*

Se utiliza una máquina de cuatro estados *VX, VY, VZ y SC*. Cuando el evento de presencia de datos en la *cola de resultados* ocurre (*qresultados\_empty=0),* se activa la escritura en la cola de salida, *qoutb,* según el estado que se encuentre activo se selecciona que porción del registro de la cola de salida se escribe y según el estado y el valor del campo de bits *vtsc,* se selecciona el siguiente estado.

## Diagrama de Bloques.

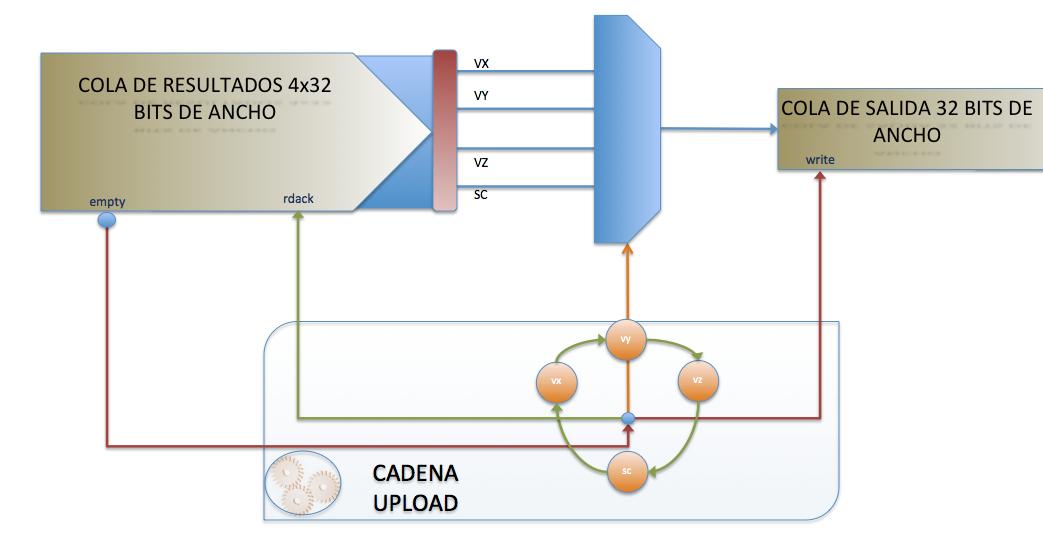
**

Ilustración . Diagrama de bloques Cadena de Upload.

La ilustración muestra la selección de escritura decodificada según el estado de la máquina de estados, la cola de resultados y la cola de salida.

La cola de salida y la cola de resultados, son cargadas y descargadas respectivamente por el proceso administrativo *Cadena Upload*, implementado mediante la máquina de estados.

La descarga de la cola de resultados y carga de la cola de salida ocurre cuando se han cargado todos los componentes y/o valores escalares del resultado. Dado que las colas del *RayTrac* están todas configuradas en modo *Show Ahead*, la descarga de la cola de resultados consiste en descartar la cola mediante el uso de la señal *rdack.*

## Recursos Utilizados.

Los recursos utilizados para implementar la *cadena de upload*, están detallados en el *Capítulo VIII. Implementación e Integración del Sistema.*

# Estados SINK y SOURCE.

Los estados Sink y Source pertenecen a la máquina de estados encargada del control de flujo general y de descarga/carga de datos externos.

El estado SINK es usado para identificar en que momento se deben sacar datos por fuera del *RayTrac* hacia el bus de datos de salida exterior y el SOURCE es usado para identificar si se pueden cargar datos dentro del *RayTrac.*

## Estado SOURCE

El *Estado SOURCE* es un estado que pertenece a la máquina de control de flujo. Para más detalles sobre la máquina de estados de control de flujo consultar en la sección 5.3. de sincronización externa.

Cuando el programador ha escrito las direcciones de memoria externas al *RayTrac,* de los flujos de entrada y los de salida, en los registro *REG\_FECTHSTART y REG\_SINKSTART,* respectivamente y el número de transferencias que piensa realizar: el *Estado SOURCE,* carga 2 colas: *qfetch y qadd,* ver .

Como se vio en la Tabla 10, existe una condición o evento llamado *Flood Condition*, está condición requiere que el programador haya escrito en el campo *nfetch* del registro de control y programación *REG\_CTRL,* el tamaño del flujo de entrada y que haya suficiente espacio en la cola de salida del *RayTrac,* para que el *Estado SOURCE,* comience una transacción de carga de direcciones y descarga de flujo de datos.

Por esto el *Estado SOURCE* solo comenzará transacciones de enganche de direcciones solamente si ocurre el evento *Flood Condition*. Ver

## Estado SINK

El *Estado SINK* es un estado que pertenece a la máquina de control de flujo. Para más detalles sobre la máquina de estados de control de flujo consultar en la sección 5.3. de sincronización externa.

El *Estado SINK* es el estado en el que se ejecuta la transmisión de flujos de datos de salida del *RayTrac.* El flujo es dirigido a una memoria externa al *RayTrac,* de manera secuencial a un a dirección especificada por el programador en el registro REG\_SINKSTART.

El *Estado SINK* como proceso administrativo afecta a la cola *qoutb,* descargándola cuando el bus de datos de salida otorga acceso al *RayTrac.*

Para que el *Estado SINK* pueda comenzar una transacción de carga de flujo de datos de salida en una memoria externa al *RayTrac,* se necesita que ocurra el evento o se cumpla condición *Drain Condition*, esta condición es verdadera si hay muchos datos en la cola de salida esperando por ser evacuados, también se cumple si hay pocos datos en la cola de salida esperando a ser evacuados al mismo tiempo que no haya datos validos transitando por el *RayTrac.*

# Avalon Memory Mapped Master.

El *Avalon Memoy Mapped Master,* es la interfaz de transmisión y recepción de datos que posee el *RayTrac.* Soporta transacciones de escritura y lectura de datos en conjunto con el bus externo *Avalon* de Altera ®.

# Control de Flujo General del RayTrac.

El control de flujo tiene las siguientes funciones:

* Verificar las condiciones para comenzar transacciones de carga o descarga de flujos de datos.
* Iniciar las transacciones de descarga o carga de flujos de datos de entrada o salida.
* Generar una señal de interrupción para notificar a un proceso exterior que el procesamiento de un flujo de entrada en particular terminó.

El control del flujo del *RayTrac,* no puede detener o reiniciar el flujo de los datos que a este ya han entrado, porque no se ha implementado señales de *enable*, como se explicó en la sección *5.2.1 Cadena de sincronización.*

La estrategia de control de flujo es privilegiar la evacuación de datos sobre el ingreso. Para ello se verifica si hay condiciones para comenzar una transacción de carga de datos en la memoria externa *(drain condition)*, de lo contrario se verifica si hay condiciones para comenzar una carga de direcciones en el bus de direcciones *(flood condition)*, si tales condiciones no se dan, el sistema queda dando *vueltas* hasta que una de las dos condiciones se de. Si las dos condiciones se dan, se privilegia la transacción de evacuación de datos como se explica al principio de este párrafo.

# Colas de Sincronización y tasa de entrada / salida.

Para modelar el control de flujo se calcula la tasa de datos ingresados y datos descartados por ciclo de reloj en cada una de las colas, en el peor de los casos y por cada *kernel.*

El diagrama de bloques de control de flujo ilustra las colas y los procesos administrativos que toman parte en el control de flujo:

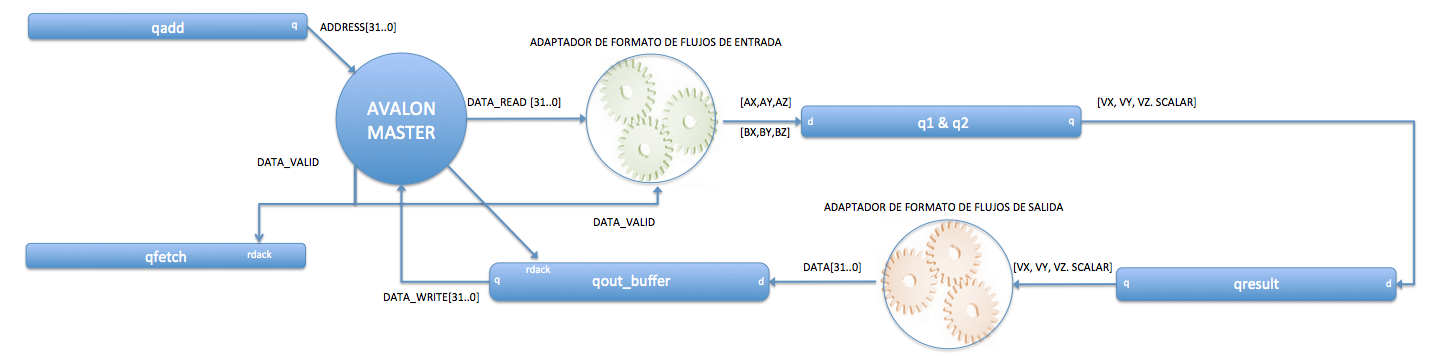


Ilustración . Colas de control de flujo y procesos administrativos de control.

Existen 2 colas abstractas, *qadd* y *qfetch* las cuales no nos interesa medir la velocidad con que se llenan o se desocupan, puesto que al ser abstractas realmente no están ocupando espacio. Las colas cuya velocidad de flujo nos interesa medir son las colas *q1&q2*, *qresult* y *qout\_buffer.*

A continuación se analiza la tasa con la que ingresan y retiran datos en las colas.

Hay que tener en cuenta que el *RayTrac* opera con vectores de 3 componentes, cada uno de 32 bits y que el bus externo de entrada de datos es de 32 bits, por lo tanto puede entrar un solo dato por ciclo de reloj. Dado lo anterior para que un vector entre al *RayTrac,* se necesita por lo menos 3 ciclos de reloj, sin embargo las operaciones de dos operandos requerirán al menos 6 ciclos de reloj, para tener datos que operar.

El *RayTrac* soporta un modo de operación combinatoria, que significa que el primer vector que viene en el flujo de datos se carga en el operando *Vector A* y el resto se cargarán siempre en el operando *Vector B.*

Esto significa que la máxima tasa de carga de datos en el *RayTrac* es de 1 vector cada 3 ciclos de reloj.

## q1&q2

Las colas de resultados intermedios *q1 y q2,* sirven para identificar la presencia o ausencia de datos validos transitando por el *pipeline aritmético.*

La velocidad máxima con la que pueden recibir datos es:

Donde *vcmax\_q1q2\_input* es velocidad de carga y se mide en flujos de datos por unidad de tiempo y la unidad de tiempo es un periodo de reloj.

El *pipeline aritmético* entrega resultados a una tasa igual a la que entran vectores en él para ser operados. Como las colas *q1 y q2* son las encargadas de monitorear el transito de datos por el *pipeline aritmético,* se deduce que la velocidad de entrega de resultados es la misma que la velocidad de descarga de las colas:

En la salida de las colas se analiza la velocidad *mínima* de descarga, para relacionarla con la velocidad *máxima* de carga y así analizar que posibilidades hay de que se llene la cola.

## qresult

Las colas de resultados entrega datos a la siguiente cola que es la cola de salida *qout\_buffer,* tan pronto cuente con datos dentro de ella. Sin embargo el ancho de la salida de la cola *qresult* difiere del ancho de la entrada de la cola *qout\_buffer.*

En el peor de los casos el *kernel* esta configurado para ejecutar la función especial de normalización de vectores y los bits *vt*  del registro de control y programación, ver *capítulo* *IV. Interfaz de Programación de Bajo Nivel del RayTrac,* están configurados para entregar 4 valores, los 3 componentes vectoriales con el vector normal y el componente escalar, con la magnitud del vector.

Por lo tanto la velocidad de carga *máxima* de esta cola es 1/3, la velocidad *máxima* de descarga es 1 (cuando solo se debe entregar como resultado apenas un escalar) y la velocidad *mínima* de descarga es 1/4 (cuando se debe entregar como resultado un vector y un escalar).

El resultado de las restas ilustrado en las ecuaciones, muestra las velocidades máximas y mínimas, a las que se llena la cola de resultados. El resultado -2/3 indica que la cola descarga datos mas rápido (1) que cuando los carga (1/3), por lo tanto la cola nunca se llenará bajo estas condiciones. El resultado 1/12 indica que la cola se llenará a un ritmo de un dato cada 12 ciclos de reloj. En este caso, si se va a llenar la cola y debe ser tenido en cuenta para el diseño de la máquina de control de flujo.

## qout\_buffer

La cola de salida tiene 2 posibles velocidades de entrada de datos: 1 y 1/4. La velocidad de salida de *qout\_buffer,* no se puede determinar porque es el mecanismo externo de la interfaz *Avalon*, la que realiza el descarte y la que realiza esta operación no está determinada. Por lo tanto asumimos el peor de los casos en que la velocidad *máxima* de descarga es 0.

La entrada de cada cola tiene una velocidad máxima de carga, la salida una velocidad mínima y máxima de descarga. La resta entre la velocidad de entrada y la velocidad de salida, da como resultado la velocidad con la que se llena la cola.

Se compila la tabla de condiciones de flujo entonces:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Q1&Q2* | | | *QRESULT* | | | *QOUTPUT\_BUFFER* | | |
| *1/3* | *0* | *1/3* | *1/3* | *-2/3* | *1* | *1* | *1* | *0* |
| *1/12* | *1/4* | *1/4* | *1/4* |

Tabla . Velocidades de carga y descarga de las colas de sincronización.

En la tabla observamos en las columnas de la izquierda debajo de cada cola, las velocidad de carga *máxima* y *mínima* donde aplique. La columna de la derecha contiene las velocidades de descarga de la cola *máxima* y *mínima* donde aplique. La columna del centro es la diferencia entre las velocidades de carga y las de descarga, valor que representa la velocidad neta con la que se va llenando la cola.

De la tabla observamos los siguientes resultados:

La velocidad de entrada y salida de las colas *q1 & q2,* es la misma, esto tiene sentido dado que el *pipeline aritmético,* no detiene el flujo de los datos que por él transitan, la velocidad de llenado del *pipeline aritmético* es 0, lo cual también tiene sentido, porque este no retiene datos.

La velocidad *máxima*  de carga de datos en la cola de resultados *qresult,* es la misma velocidad *máxima* de salida de las colas *q1&q2, 1/3.* Cuando solo se está entregando un resultado escalar (*kernels,* producto punto o magnitud vectorial), la tasa de descarga de datos *máxima* es de 1/1, haciendo que la cola se llene a una velocidad de -2/3, lo cual no tiene sentido, esto significa que la de resultados permanecerá vacía por momentos.

Cuando se está entregando cuatro datos como resultado correspondientes a 3 componentes vectoriales y uno escalar, la velocidad máxima de descarga es 1/4, esto significa que la velocidad de llenado de la cola es de 1/12. Esto significa que cada 12 clocks, si en la cola de resultados hay un flujo de entrada continuo de un resultado por cada 3 ciclos de reloj, la cola se empezará a llenar paulatinamente.

Asumiendo que el mecanismo externo de bus de datos, no le otorgue nunca el bus al *RayTrac*, para escribir datos, la velocidad máxima de llenado del buffer de salida sería 1 dato por cada ciclo de reloj, cuando solo se está entregando un valor escalar y/o de 1/4 cuando se está entregando valores vectoriales y escalares.

Después de este análisis se concluye que el efecto neto de las colas trabajando en conjunto, hace que la cola en llenarse más rápido sea la cola de salida *qout\_buffer* y sus señales de sincronización, *almost\_full y empty,* son las que determinan la estrategia de control de flujo general. La manera como se utilizan las señales de sincronización de la cola de salida de datos *qout\_buffer,* se detalla en la siguiente sección.

# Máquina de Estados de Control de Flujo.

El *RayTrac* opera en 3 modos principales: *IDLE, SOURCE y SINK. IDLE* es el estado en que el *RayTrac* se encuentra *ocioso,* sin tareas por realizar.

*SOURCE* es el estado en el que el *RayTrac* verifica las condiciones para comenzar una transacción de carga de direcciones y descarga de datos *(flood condition).*

El estado *SINK* es el estado en el que el *RayTrac* comienza una transacción de carga de datos con resultados de operaciones en una memoria externa, siempre que se den las condiciones para ello *(drain condition).*

El programador comienza una operación de escritura en el campo de bits *nfetch* en el registro de control *REG\_CTRL.* La máquina pasará al estado *SOURCE,* porque la condición *Zero Transit,* deja de cumplirse. *Ver Tabla 10. Condiciones para el control de eventos y sincronización.*

El siguiente diagrama de flujo ilustra la operación del Estado *Idle.*

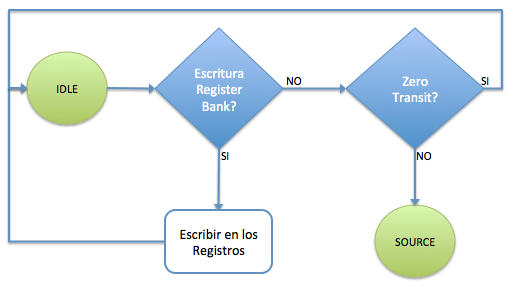


Ilustración . Estado IDLE.

El estado *IDLE* prioriza la escritura en el registro de configuración, sobre el resto de la operación de control flujo. La operación de escritura de los registros de configuración se lleva a cabo usando la interfaz *Avalon Memory Mapped Slave.* Para más información sobre la implementación de las interfaces *Avalon* en el *RayTrac,* consultar la siguiente sección, *Implementación de Interfaces Avalon.*

Si no se está llevando a cabo una operación de escritura en los registros de configuración, se verifica el tránsito de datos en las colas de control de flujo y sincronización del *RayTrac.* Si hay datos la condición *Zero Transit* es falsa y provocando que el siguiente estado sea *SOURCE,* en el siguiente ciclo de reloj, si la condición es verdadera, entonces la máquina continua ociosa y en el siguiente ciclo el estado continuará siendo *IDLE.*

Cuando el programador ha configurado los registros de fuente y destino de flujo de datos, REG\_FETCHSTART y REG\_SINKSTART, el tamaño del flujo de entrada, en el campo de bits *nfetch* del registro *REG\_CTRL,* automáticamente las colas abstractas *qadd* y *qfetch,* quedarán cargadas.

El evento o condición *Flood Condition,* se vuelve verdadero. Esta condición es usada por el estado *SOURCE* para comenzar transacciones de descarga.

El siguiente diagrama de flujo ilustra la operación de el estado *SOURCE*:

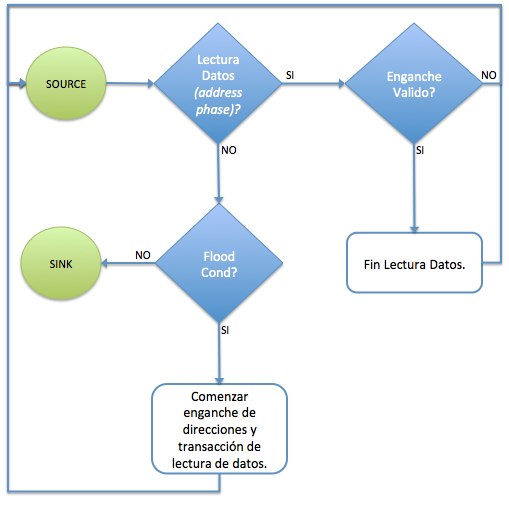


Ilustración . Estado SOURCE.

El *Estado SOURCE* prioriza si se está ejecutando un enganche de direcciones durante el ciclo de reloj. Si se está realizando la operación de enganche se verifica si ya se terminó, de haber terminado se da por finalizada la fase de enganche de dirección en la transacción de lectura.

En caso de que no se este ejecutando un enganche de dirección, se verifica si la condición *Flood Condition,* la cual verifica que se presenten las siguientes dos condiciones:

* La cola de salida *qout\_buffer* no se encuentre casi llena, *(qout\_almost\_full=0).*
* La cola de direcciones pendientes por enganchar tenga elementos, *(qadd\_empty=0).*

Si la condición *Flood Condition,* se cumple, entonces se inicia una transacción de lectura de datos, usando la interfaz *Avalon Memory Mapped Master.* Para más información sobre la implementación de las interfaces *Avalon* en el *RayTrac,* consultar la siguiente sección, *Implementación de Interfaces Avalon.*

Si *Flood Condition* no se cumple, el estado en el siguiente ciclo de reloj será el *Estado SINK.*

Cuando las direcciones se han enganchado se habrá terminado la fase de direccionamiento de la transacción de lectura. La fase de captura de datos, no necesariamente habrá comenzado. Sin embargo la fase de captura de datos trabaja de manera desacoplada de la máquina de estados de control de flujo. Esto facilita el diseño y reduce la lógica de control de flujo.

Cada vez que se engancha una dirección se reduce en uno el valor de ocupación de la cola de direcciones pendientes por enganchar. Sin embargo esto no ocurre con la cola de datos pendientes por leer, esta se descarga cuando se detecta un dato valido en el bus de entrada de datos, para más información sobre la implementación de las interfaces *Avalon* en el *RayTrac,* en particular las transacciones de lectura y escritura de datos consultar la siguiente sección, *Implementación de Interfaces Avalon.*

El *Estado SINK* tiene dos funciones: transmitir los resultados del *pipeline aritmético* a la dirección de memoria externa y notificar mediante una interrupción, si la ejecución de la operación ha terminado.

El siguiente diagrama de flujo ilustra el funcionamiento del *Estado SINK:*

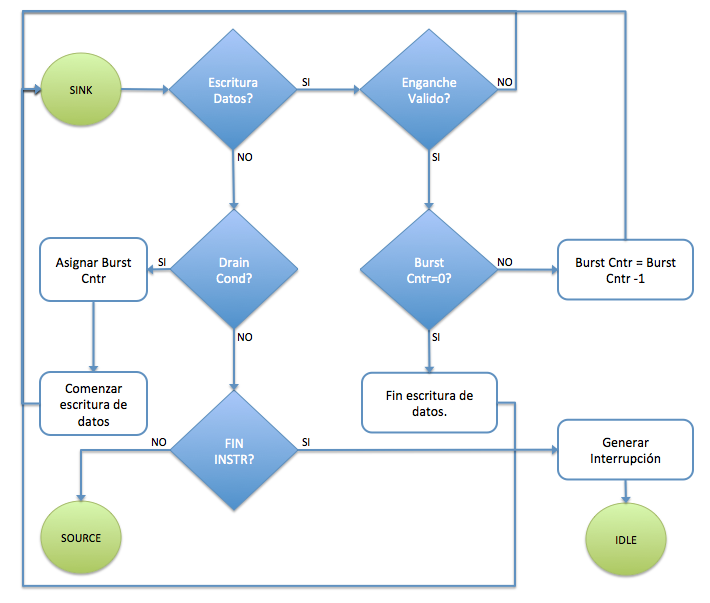
**

Ilustración . Estado SINK.

El *Estado SINK* le da prioridad a cualquier transacción de escritura de datos que se este ejecutando en el ciclo de reloj, de ser así, verifica si el dato a escribir se engancho en el bus de datos. Si el dato no se ha enganchado deja terminar el ciclo de reloj sin novedad.

Cuando se engancha el dato, se verifica si ese era el último por transmitir, si es así se da por terminada la transacción de escritura*.* Si quedan datos pendientes por transmitir, se reduce en 1 el registro de conteo de *bursts* que lleva la cuenta.

El otro caso es que no hayan transacciones de escritura llevándose a cabo. En ese caso se verifica si se dan las condiciones que exigen la evacuación o transmisión de datos hacia fuera. Dicho conjunto de condiciones ocurre cuando el evento o la condición *Drain Condition* se cumple, para tal efecto se deben dar una de las siguientes condiciones:

* La cola de salida de datos *qout\_buffer,* debe tener una cantidad de datos mínima, *qout\_buffer\_almost\_empty=0.*
* Si la hay pocos datos en la cola, *qout\_buffer\_almost\_empty=0,* y no hay datos en el resto de las colas de sincronización de datos.

Cuando las condiciones se dan, se comienza una transacción de escritura de datos, a través de la interfaz *Avalon Memory Mapped Master.*

Cuando las condiciones no se dan, la máquina de control de flujo pasará al *Estado SOURCE* si hay datos transitando en el *RayTrac,* sino hay datos transitando en el *RayTrac* entonces se genera una señal de interrupción para indicar que la operación terminó y el siguiente estado sería *IDLE.*

# Implementación de interfaces Avalon en el RayTrac.

El RayTrac implementa 3 interfaces *Avalon,* para sincronizarse y transferir datos desde y hacia el exterior. Las interfaces implementada son:

* Interfaz *Avalon Memory Mapped Master.*
* Interfaz *Avalon Memory Mapped Slave.*
* Interfaz Interrupt Sender.

# Interfaz Avalon Memory Mapped Master.

Esta interfaz es usada por el *RayTrac* para ejecutar transacciones de escritura y lectura de flujos de datos de salida y de entrada respectivamente.

Los parámetros de las operaciones que se ejecutan en el *pipeline aritmético* y los resultados de las operaciones aritméticas ejecutadas en este se leen y escriben exclusivamente por esta interfaz, cualquier valor de configuración y/o programación del *RayTrac* será escrito o leído a través de la interfaz *Avalon Memory Mapped Slave.*

# Operación de Lectura de Datos.

La lectura de datos utiliza los datos configurados en el registro *REG\_FETCHSTART* y el campo de bits *nfetch* de *REG\_CTRL.*

El registro *REG\_FETCHSTART* contiene la dirección de inicio de la transacción de lectura. Esta dirección es de 32 bits y el espacio de direccionamiento es externo al *RayTrac*. El campo de bits *nfetch*, contiene el numero de datos que se desea transferir hacia el *RayTrac.*

La transacción de lectura de datos funciona en modo *burst* o ráfaga. Esto significa que al bus Avalon se le informará el tamaño de la transacción y en que dirección se encuentran los datos a leer.

Las señales utilizadas para efectuar la transacción son:

* *Master\_waitrequest (in)*
* *Master\_read (out)*
* *Master\_burstcount (out)*
* *Master\_address (out)*
* *Master\_readdata (in)*
* *Master\_readdatavalid (in)*

La siguiente ilustración detalla un ejemplo de transacción de lectura en ráfaga.

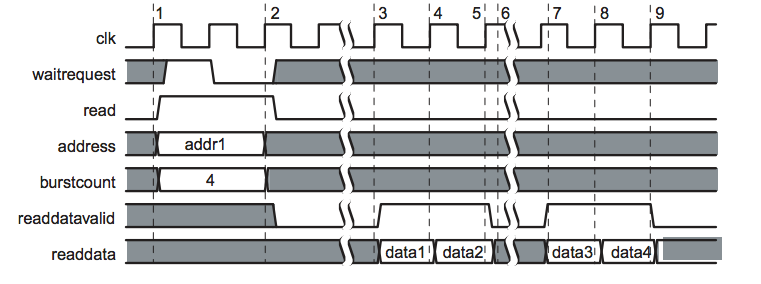


Ilustración . Transacción de lectura en ráfaga. Fuente: Altera[[7]](#footnote-7).

La interfaz inicia la transacción de lectura colocando en la señal *read,* en 1 y escribiendo la dirección que se encuentra almacenada en *REG\_FETCHSTART* en la salida *address* y escribiendo el tamaño de la ráfaga en *burstcount.* El proceso administrativo encargado de hacer el *drive* de estas señales es *Estado SOURCE.*

Inmediatamente la interconexión *Avalon* coloca la señal *waitrequest* en 1, para indicar que el bus de datos se encuentra ocupado.

Mientras que la señal *waitrequest* se encuentre en 1, las señales *read, address y burstcount* deben permanecer fijas y sin cambio.

Cuando la interconexión coloca la señal *waitrequest* en 0, estará indicando que la fase de direccionamiento de la transacción habrá terminado. Al siguiente ciclo de reloj se debe colocar en 0 la señal *read*.

Una vez quedan enganchadas las direcciones en la interconexión comienza la fase de datos de la transacción de lectura. Cada vez que la interconexión coloca en 1 la señal *readdatavalid,* se debe enganchar dentro del *RayTrac* el dato *readdata.*

El proceso que debe enganchar el dato es el *Adaptador de Formato de Flujos de Entrada.*

# Operación de Escritura de Datos.

La lectura de datos utiliza los datos configurados en el registro *REG\_SINKSTART.*

El registro *REG\_SINKTART* contiene la dirección de inicio de la transacción de escritura, a donde serán escritos los datos. Esta dirección es de 32 bits y el espacio de direccionamiento es externo al *RayTrac*.

La transacción de escritura de datos funciona en modo *burst* o ráfaga. Esto significa que al bus *Avalon* se le informará el tamaño y en qué dirección se escribirán estos datos.

* *Master\_waitrequest (in)*
* *Master\_write (out)*
* *Master\_burstcount (out)*
* *Master\_address (out)*
* *Master\_writedata (out)*

La siguiente ilustración detalla un ejemplo de transacción de una escritura en ráfaga.

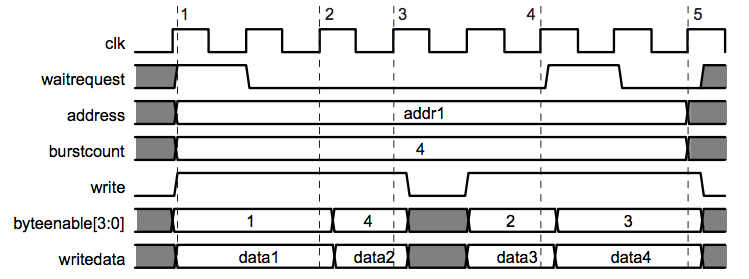


Ilustración . Transacción de escritura en ráfaga. Fuente : ALTERA[[8]](#footnote-8)

La interfaz inicia la transacción de lectura colocando en la señal *write,* en 1 y escribiendo la dirección que se encuentra almacenada en *REG\_SINKSTART* en la salida *address,* escribiendo el tamaño de la ráfaga en *burstcount* y el primer dato a escribir en *writedata.* El proceso administrativo encargado de hacer el *drive* de estas señales es *Estado SINK.*

Inmediatamente la interconexión *Avalon* coloca la señal *waitrequest* en 1, para indicar que el bus de datos se encuentra ocupado.

Mientras que la señal *waitrequest* se encuentre en 1, las señales *write, address, writedata y burstcount* deben permanecer fijas y sin cambio.

Cuando la interconexión coloca la señal *waitrequest* en 0, estará indicando que el dato a escribir fue enganchado por la interconexión al siguiente ciclo de reloj se debe colocar el siguiente dato a escribir en *writedata.*

Las señales *write, address, burstcount* deben mantenerse constantes durante toda la transacción. Pero cuando *waitrequest* sea cero y no queden mas datos por escribir la señal *write* debe quedar en 0, terminando la transacción de escritura.

# Interfaz Avalon Memory Mapped Slave.

La *Interfaz de programación de bajo nivel del RayTrac,* provee al programador con un *register bank*  de 16 registros para la configuración, programación y depuración del mismo.

La interfaz *Avalon Memory Mapped Slave* sirve para la escritura y lectura del *register bank.*

Las señales utilizadas por esta interfaz son:

* *Slave\_read (in)*
* *Slave\_readdata(out)*
* *Slave\_write(in)*
* *Slave\_writedata(in)*
* *Slave\_address(in).*

La dirección de entrada corresponde a un espacio de direccionamiento nativo, esto significa que la interconexión traduce las direcciones planas externas al espacio de direcciones nativo del *RayTrac,* haciendo que el direccionamiento sea transparente e independiente del exterior para interfaz *slave.*

# Operación de Escritura y Lectura de Datos.

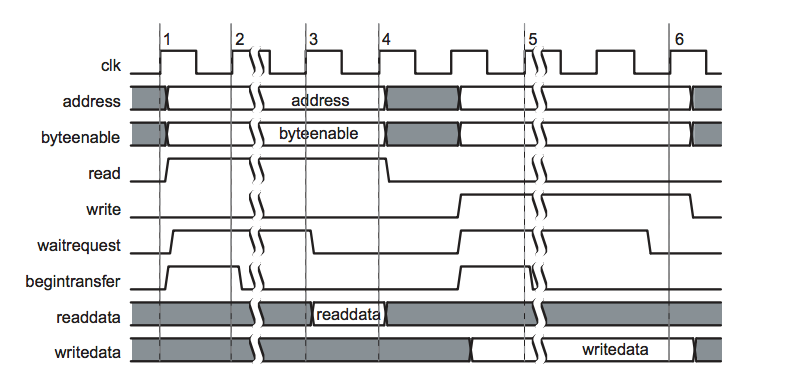


Ilustración . Transacciones de escritura y lectura en la interfaz Avalon Slave. Fuente: ALTERA.[[9]](#footnote-9)

La implementación de la interfaz *Avalon Memory Mapped Slave,* no implementa el *drive* o lectura de todas las señales ilustradas. La razón es porque la interconexión externamente las maneja y hace que el manejo de las mismas sea transparente para la interfaz implementada. Dichas señales son:

* *Slave\_byteenable :* Esta señal enmascara el conjunto de bits que desean ser leídos de los 32 bits posibles. Sin embargo el diseño de la interfaz de programación exige que sean leídos y escritos todos los bits cuando se configuran o leen registros.
* *Slave\_begintransfer :* Esta entrada señaliza el comienzo de una transacción de escritura ó lectura. Se implementa por defecto en la interconexión y es transparente para la implementación de la interfaz *slave.*
* *Slave\_waitrequest :* Esta señal le indica a la interconexión que el *RayTrac* no está disponible para ser accedido a través de la interfaz *slave.* Sin embargo este mecanismo no será implementado, porque la sincronización entre la interfaz de programación del *RayTrac* y el proceso que lo controla se hará mediante la señal de interrupción.

El siguiente diagrama de bloques ilustra la implementación de la interfaz *slave:*

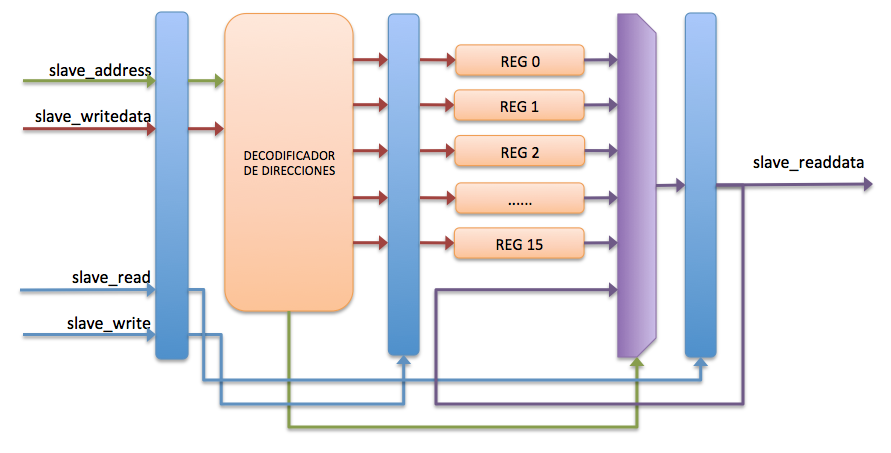
**

Ilustración . Implementación de la Intefaz Avalaon Memory Mapped Slave en el RayTrac.

La lectura de registros tiene una latencia de 2 ciclos de reloj, distribuidos en una etapa de decodificación de dirección y otra de multiplexación. La interconexión notifica la solicitud de lectura colocando la señal *read* en uno, dicha señal se engancha junto con la dirección del registro que se quiere leer, en el primer ciclo de reloj. La interconexión coloca la señal *read* en 0 y espera al siguiente ciclo por el dato. La interconexión está configurada *en hardware*, con la latencia de la transacción para saber cuantos ciclos debe esperar después de que comienza la transacción de lectura.

La escritura de registros tiene una latencia de 2 ciclos de reloj, distribuido en una etapa de enganche de decodificación de dirección y otra etapa de escritura del registro seleccionado. La transacción de escritura comienza cuando la interconexión coloca la señal *write* en 1. Una vez se engancha el dato y la dirección se da por concluida la transacción para la interconexión y esta coloca la señal *write* en 0.

La interfaz *Avalon Memory Mapped Slaved* solo sirve para configurar el *kernel* y las direcciones fuente y destino, de los *flujos de entrada* y *salida,* respectivamente.

Estos registros están disponibles para lectura en todo momento. Sin embargo solo en el *Estado IDLE,* estarán disponibles para escritura.

# Interfaz Avalon Interrupt Request (IRQ) Sender.

La interfaz de interrupción tiene la función de sincronizar el *RayTrac* con los procesos que corren al exterior. La interrupción se genera cuando la operación ha terminado. Desde el momento en que comienza la operación, hasta que se genera la interrupción los registros de configuración solo se pueden leer.

Una vez la interrupción es generada se le debe indicar al *RayTrac* que la interrupción fue atendida, para ello se debe escribir un cero en el bit mas alto del registro de configuración y programación *REG\_CTRL.*

# Simulación y Máxima Frecuencia.

# Simulación Funcional.

Se simuló la generación del conjunto de rayos iniciales que se *emiten* desde el observador hacia la imagen de un escenario 3D. En el escenario se encuentra una primitiva geométrica, un plano de proyección de 16x16 pixeles de resolución y una cámara que representa la dirección y posición del observador que *observa* la escena.

La simulación a realizar es:

1. Normalización de 16x16 rayos que salen desde un observador al plano de proyección.

# Metodología de Simulación.

Las simulaciones se llevan a cabo con la entrada y salida de archivos, hacia y desde un *testbench*, el cual es ejecutado por el programa *MODELSIM Altera Starter Edition*.

Adicionalmente se crea una herramienta utilizando el lenguaje de programación PYTHON, cuyo objetivo es analizar los códigos fuente de los RTL que describen el RAYTRAC y compilar el código VHDL con el *testbench*.

Para generar la entrada de la simulación, la cual es una escena en 3D, se utiliza el programa Blender3D, en el que se puede modelar esta escena con esferas y mediante un script se genera un archivo de entradas para la simulación.

Para visualizar los resultados de la simulación se utilizan 2 herramientas:

1. Testbench Compiler: script para hacer observables en el *testbench* aquellas señales que no son puertos de salida en el *DUT.*
2. Posterior a la ejecución del *testbench*, en un archivo CSV (*Comma Separated Values*) de valores separados por comas, se escriben los resultados de la simulación. Las filas poseerán señales o puertos de observación y las columnas avanzan hacia la derecha conforme la simulación avanza.

Para la generación de esta simulación funcional se desarrollaron 2 herramientas concretas:

1. Script de generación de *testbench*: Cuando se realiza una simulación funcional haciendo uso de un *testbench* es usual encontrar dos entidades en el *testbench*, la primera es la DUT (*Device Under Test*) o Dispositivo Bajo Test y la segunda es la entidad que controla y genera los estímulos o entradas para la simulación.

El problema que surge es que la entidad que controla los estímulos solo puede acceder a los puertos de entrada y salida de la DUT, para escribirlos y leerlos respectivamente. Dado que la DUT es por lo general la entidad top en el circuito entero, las señales y puertos que se encuentran más abajo dejan de ser visibles para la simulación, lo que supone un problema para entender los posibles errores de diseño que surjan en aquellas entidades *bottom*.

El script de generación *testbench* analiza todas las entidades instanciadas en el top *entity* y mediante reglas de comentarios (al estilo *doxygen*[[10]](#footnote-10)), detecta señales que el diseñador desea observar en la simulación y modifica los *RTL* de tal manera que esa señal sea visible en los puertos del top *entity* y de esta manera hacerlos observables para la entidad que controla y genera las entradas para la simulación.

1. El *testbench* compilado adicionalmente genera instrucciones para que en la ejecución de la simulación se genere un archivo con la imagen del *render*.

# Resultados Simulaciones.

Los resultados de las simulaciones se publican en un documento anexo al presente.

# Análisis de Tiempos.

Se realizó el análisis de tiempos para el RayTrac. El sistema posee un solo dominio de reloj de una frecuencia de 60 MHz, lo que simplifica el cuidado que se debe tener en el momento en que se realiza el diseño de los circuitos y la definición de restricciones o *constraints,* al momento de realizar el análisis de tiempos.

Sin embargo los circuitos externos al RayTrac, funcionan en un dominio de reloj de 100 MHz, por lo tanto en el capítulo de integración con los sistemas externos se realizará el análisis de tiempos, con restricciones y dominios de reloj adicionales.

Los reportes de tiempos fueron generados una vez realizada la compilación y el *fitting* del circuito en *Quartus* II.

El reporte de frecuencia máxima da por resultado 137.61 MHz. Este resultado indica que el periodo mínimo que puede tener el reloj del sistema es de 7.2 ns, tiempo satisfactorio para un sistema que se diseñó pensando para un reloj de 50 MHz cuyo periodo es de 20 ns.

El reporte de restricciones y *fmax* arrojado por la herramienta *TimeQuest* es el siguiente:

|  |  |  |
| --- | --- | --- |
| Fmax | Restricted Fmax | Clock Name |
| 137.61 MHz | 137.61 MHz | clko |

Tabla . Frecuencia máxima calculada para el RayTrac.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SDC Command | Name | Period | Waveform | Targets |
| create\_clock | clko | 20 | { 0.000 10.000 } | [get\_ports {clk}] |

Tabla . Restricciones creadas para el análisis de tiempo posterior a la operación de fitting que realiza Quartus II, que sirve como entrada para el TimeQuest Analyzer.

El reporte de tiempos se anexa al documento.

# INTEGRACION E IMPLEMENTACIÓN DEL SISTEMA RTENGINE

# Generación de Sistema en Chip Programable.

Para integrar componentes IP de Altera: Procesadores, Controladores de Memoria, Comunicación, Interfaces de Salida, etc, y el componente *custom,* la suite de desarrollo de Altera provee una herramienta para definir y generar un sistema en chip completo, *SOPC (System on a Programmable Chip) Builder.*

El sistema que se quiere definir es el *RtEngine,* el componente *custom* es el *RayTrac.*

El siguiente flujo de trabajo, representa el proceso de integración e implementación del *RtEngine* usando *SOPC Builder.*

1. Definir la especificación de entrada y salida de los componentes *custom.* En el caso del *RtEngine,* encontramos al *RayTrac,* como el componente *custom* a integrar.

La entidad definida en el RTL *raytrac.vhd[[11]](#footnote-11)* ilustra la especificación de entrada y salida:

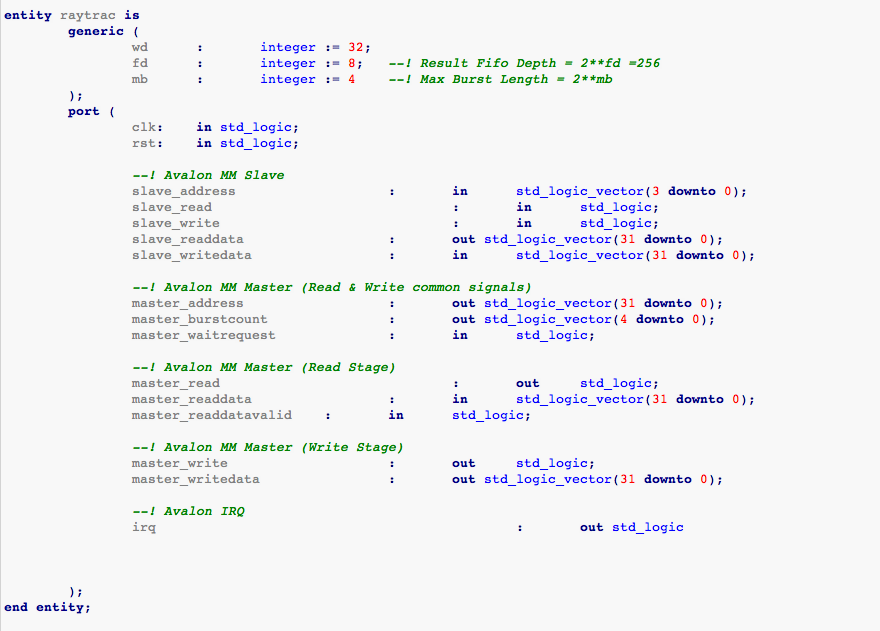


Ilustración . Definición de Entrada y Salida del RayTrac. Todas las señales pertenecen a alguna interfaz Avalon.

La especificación define las siguientes interfaces implementadas en el RayTrac:

* + Interfaz Avalon Memory Mapped Master.
  + Interfaz Avalon Memory Mapped Slave.
  + Interfaz Interrupt Request Sender (IRQ).
  + Interfaz de Reloj a 60 MHz

1. Importar el diseño a componente SOPC.

SOPC Builder cuenta con un editor de componentes, con el que se crea el componente *RayTrac* en el SOPC Builder.

La siguiente ilustración muestra la pestaña de dialogo para generar las interfaces *Avalon* a partir de las entradas / salidas del sistema.

Con el componente creado se genera un bloque, cuyo símbolo ilustra las interfaces que tiene el componente:

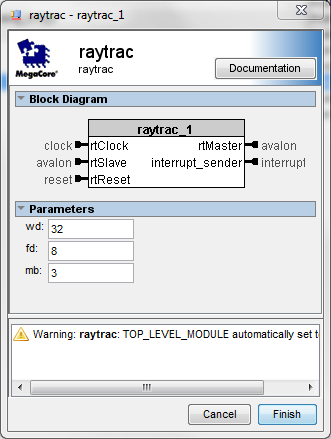


Ilustración . Componente RayTrac generado. En el símbolo se observa las interfaces, con las que se especificó al RayTrac.

1. Instanciar el componente SOPC:

La Ilustración 30, muestra un componente *RayTrac* instanciado. Las interfaces son instanciadas con los parámetros que se observan en la ilustración. En este caso los parámetros son:

* *Wd:* Ancho de bus de datos en las interfaces Master y Slave. Ancho de bus de direcciones en la interfaz Master.
* *Fd:* Parámetro para determinar la capacidad de las colas de resultado y salida en el *RayTrac.* La capacidad se calcula así:

*Numwords = 2fd=28=256.*

* *Mb:* Parámetro para determinar la máxima longitud de ráfaga, en las transacciones de recepción y transmisión de datos. Este valor es transparente a la interfaz de programación de bajo nivel del *RayTrac* y no lo afecta. Para calcular la longitud máxima de ráfaga se aplica la siguiente formula:

*Maxburst = 2mb=23=8.*

La ilustración muestra la totalidad de los componentes instanciados en el sistema, resaltados aparecen la CPU, un procesador *Nios II Processor* y la unidad aritmética vectorial *RayTrac.*

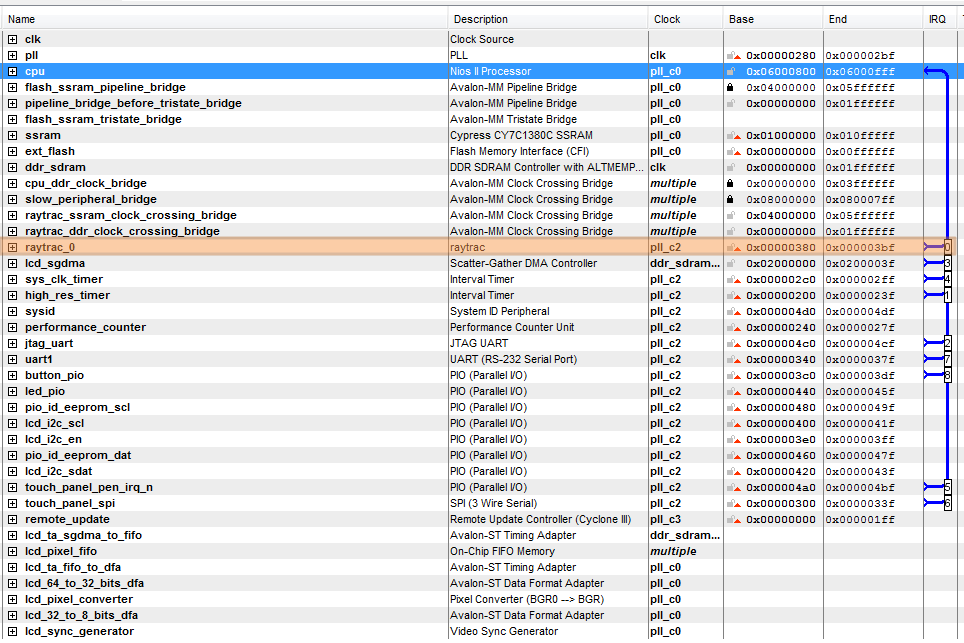


Ilustración . Sistema RtEngine, componentes instanciados en la FPGA.

La siguiente ilustración muestra como están conectadas las interfaces *Avalon* en el *RayTrac* a las interfaces de otros componentes:

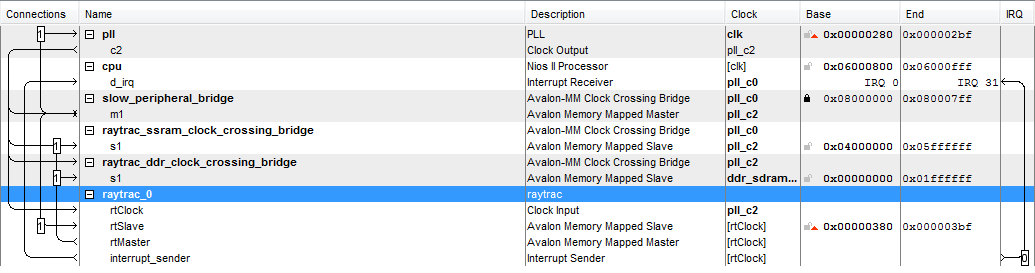


Ilustración . Componentes e interfaces Avalon conectadas a las interfaces del RayTrac.

La interfaz *rtClock,* corresponde a la interfaz de reloj de entrada del *RayTrac.* La fuente de esta entrada es la salida de un PLL de 60 MHz. Este será el dominio de frecuencia en el que trabaje el sistema.

La interfaz *rtSlave,* está conectada y comandada por la interfaz *master,* del componente *slow\_peripheral\_bridge.* Este puente lo que hace es conciliar el dominio de reloj de la *CPU (100 MHz)* y el dominio de reloj del *RayTrac (60 MHz).*

La interfaz *rtMaster,* está conectada a los puentes *raytrac\_ssram\_clock\_crossing\_bridge* y *raytrac\_ddr\_clock\_crossing\_bridge* que concilian los dominios de reloj de las memorias *ssram (100 MHz)* y *sdram* *(66.5 MHz)* respectivamente*.* A través de ellos se hacen las transacciones de escritura y de lectura en dichas memorias.

# Especificaciones.

El sistema genera un archivo *.tcl* el cual es un script del componente generado. Adicionalmente se genera un archivo *.sopc,* que describe el sistema y del cual se genera la capa de abstracción de hardware. A continuación las características del sistema generado:

* Nios II Standard, 100 MHz, RISC 32 bit, 1Kbyte Instruction Cache, Branch Prediction, Hardware Multiply, Hardware Divide. Level 1 JTAG Debug Module.
* Controladores para 32 MBytes, Synchronous Dynamic RAM, 1 Mbyte Synchronous Static RAM, 16 MBytes Flash RAM.
* JTAG Uart & RS232 UART.
* 800x480x24b VGA LCD Touchscreen.
* Unidad Aritmética Vectorial SIMD, *RayTrac,* programable.

# Recursos de Hardware Utilizados por el RtEngine.

El sistema generado se compilo en Quartus II y produjo el siguiente reporte:

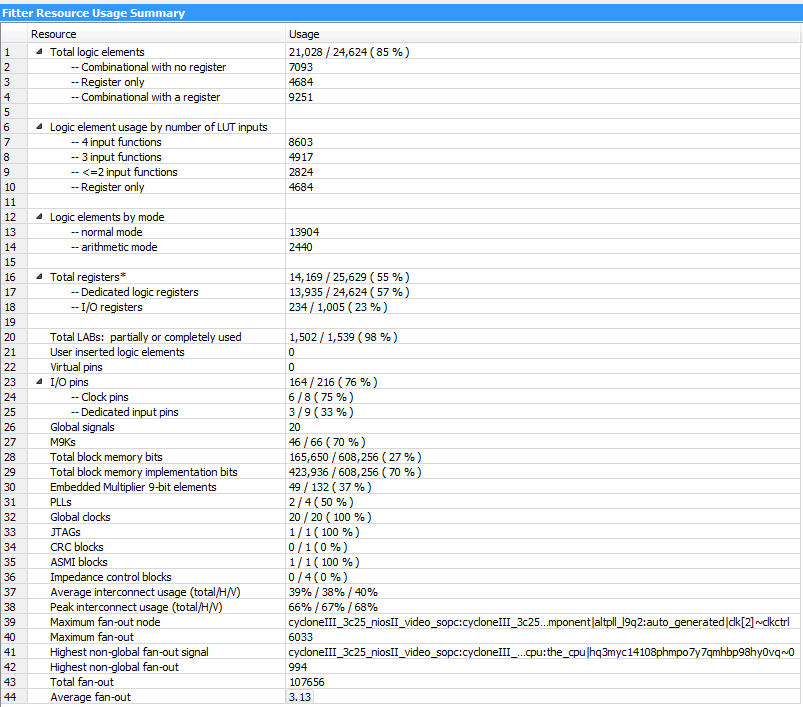


Ilustración . Recursos usados por el RtEngine. Fuente: ALTERA QUARTUS II, Fitter Report.

El sistema generado permite a la FPGA quedar holgada en recursos y se puede pensar en aumentar las prestaciones del RayTrac incluso en instanciar otro más.

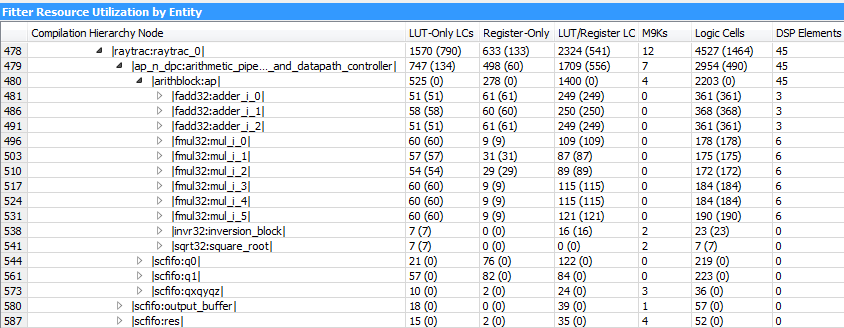


Ilustración . Recursos usados por el RayTrac. Fuente: ALTERA QUARTUS II, Fitter: Resources Used By Entity Report.

Resumiendo la información en la tabla se tiene:

* *Interconexión decodificada de circuitos aritméticos:* Utiliza 490 celdas lógicas.
* *Adaptador de formato de flujos de salida:* Utiliza en la lógica de las colas de resultados y de salida, 109 celdas lógicas y 4 bloques de memoria *m9k.*
* Sumadores *pipeline aritmético.* En promedio, cada uno utiliza 363.33 celdas lógicas, 1 multiplicador de 9x9 bits y 1 multiplicador de 18x18 bits, para realizar corrimientos.
* Multiplicadores *pipeline aritmético.* En promedio, cada multiplicador utiliza 173.5 celdas lógicas y 3 multiplicadores de 18x18 bits.
* Raíz Cuadrada *pipeline aritmético*: En promedio utiliza 7 celdas lógicas y 2 bloques de memoria *m9k.*
* Inversor *pipeline aritmético:* En promedio utiliza 23 celdas lógicas y 2 bloques de memoria *m9k.*
* Colas de sincronización *pipeline aritmético q0 y q2:* en promedio cada cola utiliza 221 celdas lógicas. *(un costo alto, se sugiere utilizar bloque m9k).*
* Cola de sincronización *pipeline aritmético q1:* utiliza 36 celdas lógicas, mucho menos que q0 y q2, pero requiere de 3 *m9ks,* para ser implementada.

Un circuito *RayTrac* requiere 4527 celdas lógicas, 12 bloques de memoria *m9k* y 45 multiplicadores 9x9 bits.

TERCERA PARTE : RESULTADOS

# RESULTADOS

# Pruebas Realizadas

Las pruebas realizadas se realizaron en dos niveles, verificación y validación. Los resultados de estas prueban confirman el logro de los objetivos, en mayor o menor media, propuestos en el Capítulo 0.

Las pruebas de verificación consisten en la medición del desempeño del RayTrac, al ejecutar las diferentes operaciones que este puede ejecutar. Estas pruebas buscan confirmar el funcionamiento y medir el desempeño de los circuitos aritméticos que ejecutan operaciones vectoriales.

Las pruebas de validación confirman el funcionamiento del sistema como arquitectura de trazado de rayos y síntesis de imágenes.

Para la realización de las pruebas se utiliza la interfaz operativa cuya implementación se describe en la primera parte de este trabajo. La realización de las pruebas en si mismo es una prueba del funcionamiento de esta interfaz operativa.

# Pruebas de verificación.

Se realizó la verificación de resultados y medición de desempeño para un conjunto de pruebas específico. El conjunto de pruebas se describe a continuación.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Tipo de Ejecución** | **Hardware: RayTrac** | **Software: Nios II** |
| **Operaciones** |  |
| * Producto Punto * Producto Cruz * Producto Simple * Suma * Normalización | | Etapa 1: 5 x 10 Ejecuciones | Etapa 1: 5 x 10 Ejecuciones |
| Etapa 2: 5 x 100 Ejecuciones | Etapa 2: 5 x 100 Ejecuciones |
| Etapa 3: 5 x 1000 Ejecuciones | Etapa 3: 5 x 1000 Ejecuciones |

Por cada operación se realizó 2 tipos de prueba: 3 pruebas en hardware y 3 en software. Se midió dos indicadores, desempeño y validez de resultados. Se realizaron pruebas en software para poder tener una referencia y criterio de evaluación para las pruebas de hardware.

Los 2 tipos de prueba están divididos en 3 etapas: una primera etapa de 5 repeticiones de 10 ejecuciones, una segunda etapa de 5 repeticiones de 100 ejecuciones y una tercera etapa final de 5 repeticiones de 1000 ejecuciones.

De cada repetición se mide el desempeño y con los 5 valores medidos se calcula el promedio y en el caso de las pruebas en hardware, se verifica la validez de los resultados, a partir de los resultados obtenidos en la pruebas de software.

# Pruebas de Validación.

Se realizó el *rendering* de varias escenas, de las cuales se midió el tiempo que tomo realizar dichos *render* en hardware y en software. También se contabilizó el número de intersecciones realizadas, el número de rayos calculados y el número de polígonos y fuentes de luz que contenía cada escena.

# Resultados y Conclusiones

# Verificación: Resultados.

Se realizó la medición del tiempo de ejecución en Hardware y Software de las siguientes operaciones:

* Producto Punto
* Producto Cruz
* Producto Simple
* Suma Vectorial
* Normalización de Vectores.

Las pruebas fueron hechas por cada operación. Se realizaron las mismas pruebas en hardware y en software, usando el RayTrac y el Nios II respectivamente.

A continuación se muestran los resultados de las pruebas de desempeño por cada operación:

* Producto Punto

|  |  |  |  |
| --- | --- | --- | --- |
| **Operación** | **Número de Iteraciones** | **Hardware (RayTrac)** | **Software (Nios II)** |
| **Tiempo de Ejecución Promedio (µsecs)** | |
| Producto Punto | 10 | 103,415 | 1153,1908 |
| 100 | 647,8 | 10760,0678 |
| 1000 | 6443 | 107059,097 |

Tabla 1. Resultado de desempeño para la operación producto punto.

En la tabla se observan los tiempos promedios de ejecución de 10, 100 y 1000 ejecuciones de la operación producto punto.

Ilustración 1. Comparativo de los tiempos por tipo de ejecución. En Violeta los tiempos de ejecución en software y en azul los tiempos de ejecución en hardware.

El eje Y está dividió logarítmicamente en base 10 y representa el tiempo en microsegundos que en promedio tomó cada etapa de 5 repeticiones. El eje X representa el número de ejecuciones en cada etapa. Finalmente se representa mediante esferas los resultados, donde el radio de cada esfera representa el tiempo promedio de ejecución de cada etapa (lo mismo que su ubicación en el eje Y). El color de la esfera representa el tipo de ejecución: azul para hardware y violeta para software.

* Producto Cruz

|  |  |  |  |
| --- | --- | --- | --- |
| **Operación** | **Número de Iteraciones** | **Hardware (RayTrac)** | **Software (Nios II)** |
| **Tiempo de Ejecución Promedio (µsecs)** | |
| Producto Cruz | 10 | 99,1008 | 2896,4072 |
| 100 | 660,794 | 29266,7532 |
| 1000 | 6468 | 255011,1938 |

Tabla 2. Resultado de desempeño para la operación producto cruz.

Ilustración 2. Comparativo de los tiempos por tipo de ejecución. En Violeta los tiempos de ejecución en software y en azul los tiempos de ejecución en hardware.

* Suma Vectorial.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operación** | **Número de Iteraciones** | **Hardware (RayTrac)** | **Software (Nios II)** |
| **Tiempo de Ejecución Promedio (µsecs)** | |
| Suma Vectorial | 10 | 103,6482 | 1137,5804 |
| 100 | 650,6 | 10778,499 |
| 1000 | 6466,2 | 107008,3562 |

Tabla 3. Resultado de desempeño para la operación suma vectorial.

Ilustración 3. Comparativo de los tiempos por tipo de ejecución. En Violeta los tiempos de ejecución en software y en azul los tiempos de ejecución en hardware.

* Producto Simple

|  |  |  |  |
| --- | --- | --- | --- |
| **Operación** | **Número de Iteraciones** | **Hardware (RayTrac)** | **Software (Nios II)** |
| **Tiempo de Ejecución Promedio (µsecs)** | |
| Producto Simple | 10 | 102,23 | 2939,9638 |
| 100 | 656,8 | 29150,4946 |
| 1000 | 6465 | 291959,025 |

Tabla 4. Resultado de desempeño para la operación producto simple.

Ilustración 4. Comparativo de los tiempos por tipo de ejecución. En Violeta los tiempos de ejecución en software y en azul los tiempos de ejecución en hardware.

* Normalización de Vectores.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operación** | **Número de Iteraciones** | **Hardware (RayTrac)** | **Software (Nios II)** |
| **Tiempo de Ejecución Promedio (µsecs)** | |
| Normalización | 10 | 99,8436 | 183,7848 |
| 100 | 653,4 | 1431,2 |
| 1000 | 6485,9376 | 14120,5224 |

Tabla 5. Resultado de desempeño para la operación normalización vectorial.

Ilustración 5. Comparativo de los tiempos por tipo de ejecución. En Violeta los tiempos de ejecución en software y en azul los tiempos de ejecución en hardware.

# Validación: Resultados.

# Conclusiones.

*Conclusiones a partir de los resultados de verificación.*

* Los resultados de verificación muestran la ventaja de implementar la ejecución de operaciones vectoriales complejas en hardware. En la mayoría de casos, se encontró que la ejecución de hardware es más rápida que la ejecución de software. Al comparar las operaciones se encuentra que la ejecución en hardware es al menos un orden de magnitud más rápida que la ejecución de software (10x).
* Los tiempos de ejecución escalan linealmente conforme aumenta el número de ejecuciones a realizar.
* Los resultados de medición del *overhead* asociado a cada operación dan evidencia del cuello de botella que supone el acceso a la memoria del sistema. A pesar de que el RayTrac realiza transferencias tipo *DMA,* dicha transferencia está sometida al arbitramento del *Bus Avalon*, por lo tanto los tiempos de acceso son variables y no determinables.
* El cuello de botella generado para acceder a los recursos de memoria desde el RayTrac o el Nios II, es un problema tan importante a resolver, como el problema de falta de desempeño cuando se ejecutan las operaciones vectoriales en software exclusivamente.

# Trabajo a Futuro

1. Julián Andrés Guarín, https://sourceforge.net/projects/jagtracer/ [↑](#footnote-ref-1)
2. ALTERA, Nios II Software Developer HandBook, Chapter 7: Developing Device Drivers for the Hardware Abstraction Layer. [↑](#footnote-ref-2)
3. # IEEE Standard 754 Floating Point Numbers [Steve Hollasch](mailto:steve@hollasch.net?Subject=IEEE%20Floating%20Point%20Page)

   [↑](#footnote-ref-3)
4. Aumentar la segmentación si puede colaborar a que la *fmax* aumente.  [↑](#footnote-ref-4)
5. TANEMBAUM, Planificadores de prioridad. [↑](#footnote-ref-5)
6. Para más información sobre la configuración de las FIFO IP de ALTERA, consultar [www.altera.com/lieratute/ug/ug\_fifo.pdf](http://www.altera.com/lieratute/ug/ug_fifo.pdf), “ALTERA SCFIFO and DCFIFO Megafunctions, User Guide”. [↑](#footnote-ref-6)
7. Altera, Avalon Interface Specifications. [↑](#footnote-ref-7)
8. Altera, Avalon Interface Specifications. [↑](#footnote-ref-8)
9. Altera, Avalon Interface Specifications. [↑](#footnote-ref-9)
10. www.doxygen.org [↑](#footnote-ref-10)
11. http://opencores.org/websvn,filedetails?repname=raytrac&path=%2Fraytrac%2Fbranches%2Ffp\_sgdma%2Fraytrac.vhd [↑](#footnote-ref-11)