Título:	Implementación *RTL/Verilog* de un procesador de *shader* para una GPU
Alumno:	Iván Pizarro Calvo
Director/Ponente:	Roger Espasa Sans
Departamento:	Arquitectura de computadores
Fecha:	20/06/2012

DATOS DEL PROJECTO

Título del proyecto: Implementación RTL/Verilog de un procesador de shader para una GPU

Nombre del estudiante: Iván Pizarro Calvo

Titulación: Ingeniería Informática

Créditos: 37.5

Director/Ponente: Roger Espasa Sans

Departamento: Arquitectura de computadores

MIEMBROS DEL TRIBUNAL

Presidente: Agustín Fernández Jiménez

Vocal: Mónica Sanchez Soler

Secretario: Roger Espasa Sans

CUALIFICACIÓN

Cualificación numérica:

Cualificación descriptiva:

Fecha:

Índice

1. Introducción

En las últimas décadas se ha registrado un incremento en la investigación y desarrollo de las arquitecturas hardware dedicadas específicamente a la representación de gráficos en tiempo real, impulsado en gran parte por la industria del videojuego.

Estos dispositivos llamados procesadores gráficos (GPU – *Graphical Processing Unit*), han evolucionado desde un *pipeline* completamente fijo, hasta uno programable, como sucede hoy día en las tarjetas gráficas que se encuentran en el mercado. Además de proporcionar una arquitectura programable, han demostrado tener una capacidad de cálculo que en algunos casos supera el de los procesadores de propósito general.

Conociendo la gran eficiencia realizando cálculos para gráficos, se empezó a aprovechar para la resolución de otro tipo de problemas que no guardaban relación con el renderizado. La combinación de los procesadores gráficos de poder realizar cálculos matemáticos de forma rápida con una gran capacidad de paralelización era utilizado en tareas tales como procesado de vídeo, resolución de ecuaciones matemáticas o cualquier otro tipo de problema que pudiera explotar las características de una GPU.

Este interés por utilizar una GPU para fines para los que en un principio no había sido desarrollada, acabó por acuñar el término GPGPU (*General-purpose GPU*), y ha propiciado el desarrollo de un lenguaje llamado CUDA (*Compute Unified Device Architecture*) por parte de nVIDIA, que permite beneficiarse de las características de una GPU sin necesidad de conocer con detalle la programación del *pipeline* gráfico.

Aun teniendo en cuenta que han marcado un punto de inflexión en el desarrollo de ciertas aplicaciones, la utilización de una GPU no siempre es la mejor opción, como sucede con los problemas que no se puedan paralelizar o que tengan mucho control de flujo.

1.1. Objetivos

El objetivo del proyecto es diseñar e implementar un procesador de *shader* para una GPU utilizando *RTL/Verilog*. En concreto se implementará para la GPU ATTILA, de la que se dispone una implementación en C++.

ATTILA solo implementa una GPU por software, por lo que la idea de este proyecto es que la implementación del procesador de *shaders* sirva para que en proyectos posteriores se puedan ir añadiendo otras etapas del pipeline gráfico hasta conseguir una implementación de la GPU completamente hardware.

El diseño e implementación en *Verilog* de este procesador, aun siendo dedicado para gráficos, comparte muchas características de los procesadores vistos durante la carrera, y sirve para ampliar los conocimientos adquiridos en las asignaturas de arquitectura de computadores.

Alcanzar el objetivo del proyecto, ha requerido de conocimientos relacionados con el proyecto ATTILA y el diseño de hardware utilizando un lenguaje de descripción de hardware:

- Conocer la arquitectura de la GPU ATTILA.
- Estudiar las características del procesador de *shaders* implementado por ATTILA.
- Modificar el procesador de *shaders* de ATTILA con el fin de poder utilizarlo para realizar pruebas.
- Estudiar el diseño de hardware en *Verilog*.
- Aprender el funcionamiento de las herramientas de simulación.
- Diseñar una serie de pruebas para validar el modelo.

1.2. Organización de la memoria

A continuación se explica de forma breve el contenido de este documento y la organización de los apartados.

En los apartados 2, 3 y 4 se explican los conceptos básicos para entender el entorno del proyecto, en concreto se explicará el pipeline gráfico, el simulador de GPU ATTILA, nociones básicas de lenguajes de descripción de hardware y Verilog, y por último el entorno donde se ha desarrollado.

En el apartado 5 se hace una presentación de los procesadores de *shaders*, y qué función tienen dentro del proceso de renderización.

En el apartado 6 se describe brevemente el entorno de trabajo sobre el que se ha desarrollado el proyecto.

A partir del apartado 7 se describe el trabajo realizado hasta conseguir los objetivos que se habían propuesto. Se incluyen en estos apartados la descripción del pipeline del procesador diseñado, explicación de sus etapas, dependencias de datos, cortocircuitos y la definición de las instrucciones que soporta.

En el apartado 11 se presentan los estudios realizados para la validación del modelo y los resultados obtenidos.

En los apartados 12 y 13 se presenta la planificación del proyecto a partir de un análisis del tiempo empleado en cada etapa del proyecto y una valoración económica del mismo.

Por último, en el apartado 14 se encuentran las conclusiones del proyecto que se ha desarrollado y las líneas de trabajo futuro.

Los apartados 15 y 16 incluyen la bibliografía y anexos con información adicional que complementan el proyecto.

2. Pipeline gráfico

Antes de explicar el diseño del procesador, se introducirán los conceptos básicos para entender la situación del proyecto dentro de la arquitectura de una GPU.

El objetivo de la GPU es, a partir de un conjunto de vértices en coordenadas 3D, obtener una imagen bidimensional. Este proceso se conoce como renderización y se puede dividir en dos grandes etapas:

- Geometría.
- Rasterización.

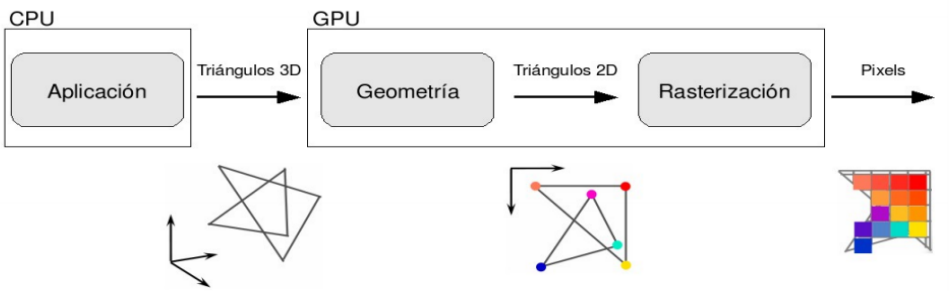

Figura 2.1 *Pipeline* gráfico simplificado

Como se observa en la figura, la CPU es la encargada de alimentar la GPU con triángulos 3D, para que sean transformados a triángulos 2D y por último se obtengan los píxeles que formarán la escena.

2.1. Etapa de geometría

En esta etapa se realizan todas las operaciones sobre los vértices de los polígonos, y se subdivide en varias etapas.

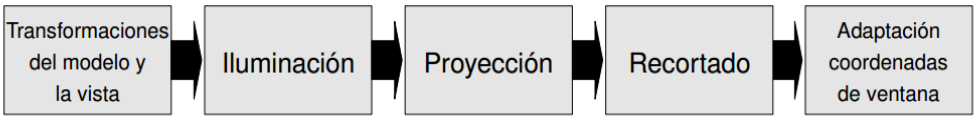

Figura 2.2 Etapa de geometría

2.1.1. Transformaciones del modelo y de la cámara

En un principio, el modelo se encuentra definido en su propio espacio de coordenadas, llamado 'espacio del modelo', con lo que aun no ha sido transformado. Cada uno puede tener unas transformaciones asociadas que permiten situarlo y orientarlo de forma diferente al resto, esto se conoce como instancia del modelo y es muy útil ya que permite tener varias copias sin replicar la información referente a la geometría.

Una vez se han aplicado las transformaciones al modelo, éste se encuentra en 'coordenadas de mundo' o 'espacio de mundo', único para toda la escena.

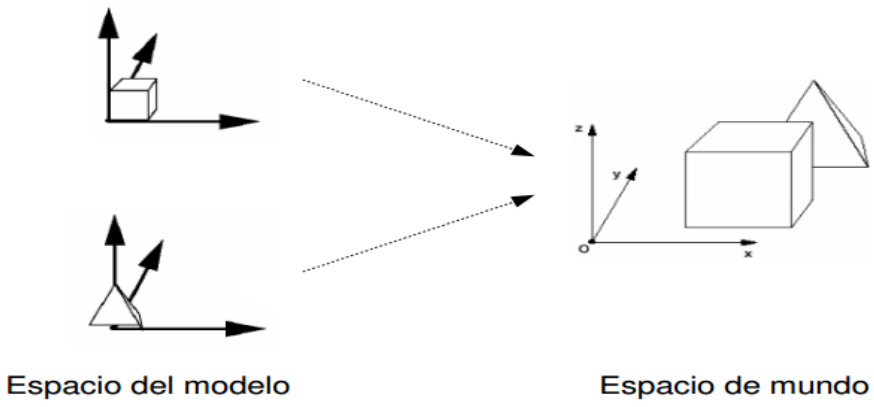

Figura 2.3 Transformaciones del modelo

De igual forma, la cámara de la escena tiene una posición, orientación y ángulo de apertura que definirá la pirámide de visión. El nuevo espacio de coordenadas se conoce como espacio de cámara o de observador.

Todas las transformaciones se implementan utilizando matrices 4x4, de forma que varias transformaciones se pueden concatenar calculando el producto de todas las matrices.

2.1.2. Iluminación

Uno de los objetivos principales en el mundo de los gráficos, es hacer que los modelos tengan una apariencia lo más realista posible, para ello se añade como información a cada vértice el color y textura, además de incorporar iluminación a la escena. La aportación de la luz al color de de cada vértice se calcula con ecuaciones que intentan aproximar la naturaleza de la luz en el mundo real.

Considerando que la mayoría de modelos se representan con triángulos, el color de cada vértice se calcula en función de los parámetros anteriores, y el color interior se interpola a partir de los vértices, utilizando por ejemplo la interpolación de Gouraud.

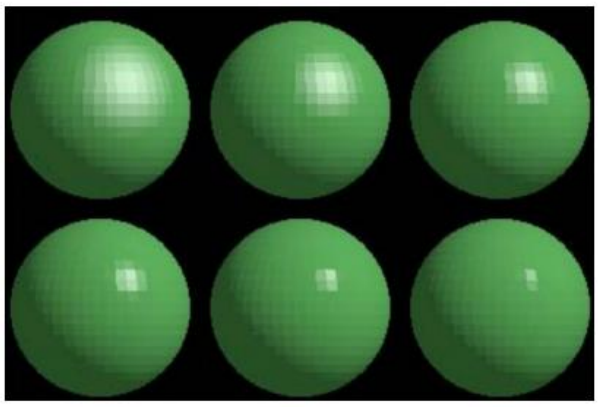

Figura 2.4 Ejemplos de iluminación

2.1.3. Proyección

Una vez se ha calculado la iluminación, se calcula la transformación de proyección. El objetivo es pasar el volumen de visión a un cubo unitario o volumen de visión canónico.

Hay dos tipos de proyección, ortogonal y perspectiva.

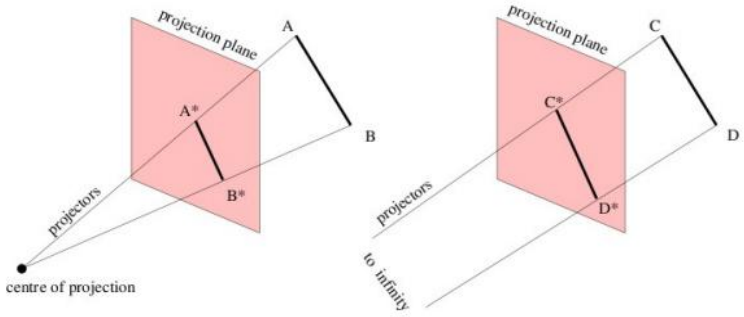

Figura 2.5 Proyección perspectiva y ortogonal

La diferencia es que en la proyección perspectiva los objetos que se encuentran más lejos se ven más pequeños, y las líneas paralelas convergen en el horizonte. Es este tipo de proyección el que simula cómo el ser humano percibe la realidad.

El volumen de visión se conoce geométricamente como *frustrum* y es una pirámide con base rectangular. Será este *frustrum* lo que se transformará a un cubo unitario después de la proyección. Una vez acabado este paso los modelos se encuentran definidos en coordenadas normalizadas de dispositivo.

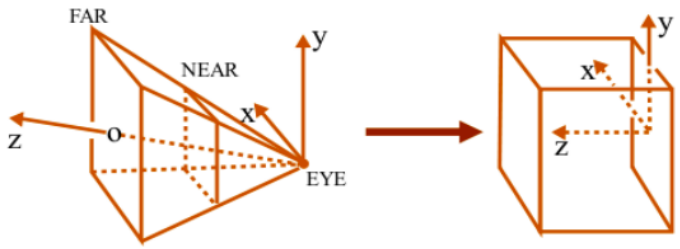

Figura 2.6 Normalización de coordenadas

2.1.4. Clipping

Esta etapa se encarga de eliminar aquellas primitivas que se encuentran fuera del volumen de visión. Aunque esto no es imprescindible para obtener una renderización correcta, sí que es importante porque el hecho de eliminar primitivas, reduce la cantidad de información que se tiene que procesar en etapas posteriores.

Hay tres casos posibles, que la primitiva se encuentre íntegra dentro del volumen de visión, que esté completamente fuera, o una parte dentro y otra fuera. En el primer y segundo caso son aceptadas y rechazas respectivamente, en el último se recorta la parte que está fuera, construyendo una nueva primitiva.

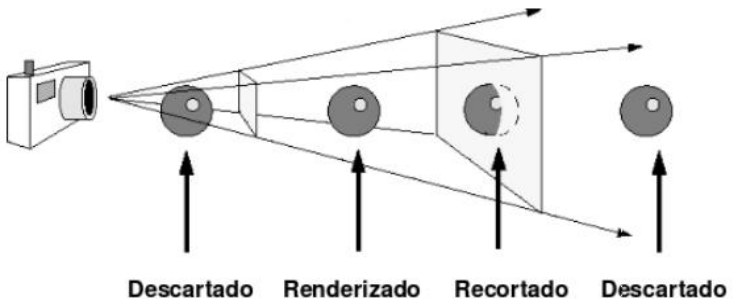

Figura 2.7 *Clipping*

2.1.5. Adaptación a coordenadas de ventana

En este punto ya solo se procesarán aquellas primitivas que entran dentro del volumen de visión, aun así, sus coordenadas todavía son tridimensionales. Las coordenadas x e y se transforman a coordenadas de pantalla, y se conserva la z (-1 <= z <= 1), formando las coordenadas de ventana. Las tres componentes se pasan a la etapa de rasterización.

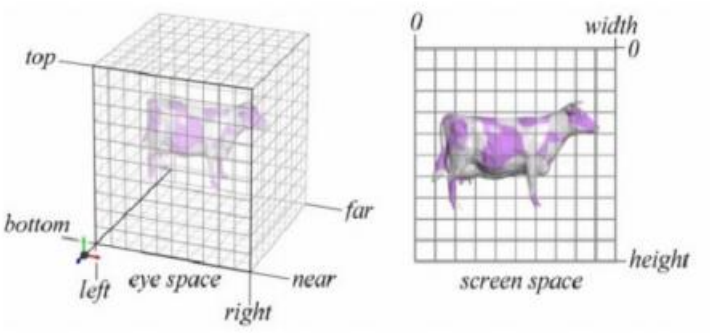

Figura 2.8 Adaptación de coordenadas de ventana

2.2. Etapa de rasterización

Después de todos los pasos anteriores, se construirán los fragmentos que formarán parte de cada primitiva, asignando a cada fragmento el color, coordenadas de textura,... Esta etapa no opera con polígonos, sino con fragmentos. Intuitivamente, un fragmento es un candidato a ocupar el lugar de un pixel en la imagen final.

Para guardar el resultado final se utiliza una matriz bidimensional, sin embargo, al llegar a esta etapa lo único que se tiene son las coordenadas de cada vértice de cada primitiva, por lo tanto hay que calcular qué fragmentos forman la primitiva.

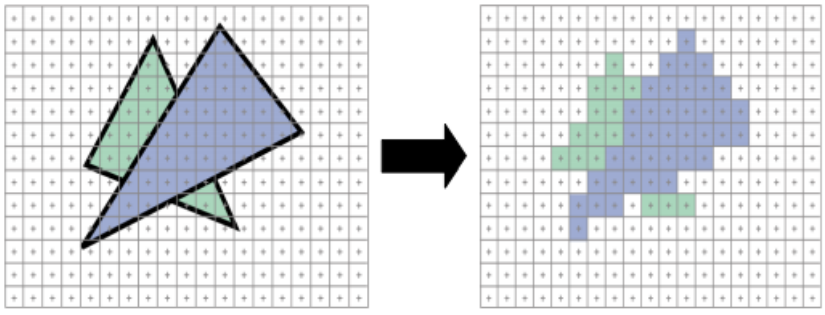

Figura 2.9 Ejemplo de rasterización de un triángulo

Por último se pasan una serie de *tests* que determinan la visibilidad de las primitivas en la imagen final. El hardware gráfico dispone de un buffer de profundidad, del mismo tamaño que el buffer de color, donde para cada fragmento se guarda la componente z, que representa la profundidad respecto a la cámara. Para hacer esto se utiliza el algoritmo del *Z-buffer*, éste permite que las primitivas se rendericen en cualquier orden, quedando al final las que están más cerca de la cámara. El único caso en que el orden es importante es si algunas primitivas son transparentes.

Por otra parte la texturización también se realiza en esta etapa, y consiste en asignar puntos de una textura, comúnmente bidimensional, a un modelo.

Figura 2.10 Ejemplo de texturización de una esfera

El último paso es actualizar el buffer de color con los valores de los fragmentos que sean visibles desde la cámara con los valores calculados en todas las etapas anteriores.

3. Proyecto ATTILA

3.1 Introducción

ATTILA es un simulador de una GPU desarrollado por un grupo de investigación del Departamento de Arquitectura de Computadores de la UPC.

La arquitectura que se simula presenta las características principales de una GPU actual. El simulador sigue el paradigma de simulación basado en eventos discretos a nivel de ciclos. Está implementado en C++ y permite incorporar nuevas funcionalidades de forma modular. Además es altamente configurable, lo que permite evaluar parámetros de la arquitectura para después analizarlos.

ATTILA implementa el modelo unificado de pipeline gráfica. En este modelo los procesadores de *shader* se pueden utilizar tanto para procesamiento de vértices (*vertex shader*) como de fragmentos (*fragment shaders*). Además de permitir una configuración que simula el modelo tradicional de *pipeline* fijo.

3.2 Pipeline de ATTILA

La GPU está compuesta por las siguientes unidades funcionales, que en su conjunto implementan el *pipeline* gráfico descrito en el punto anterior:

Streamer

Es el encargado de obtener los datos de geometría del controlador de memoria, convertirlos al formato interno y pasarlos a los procesadores de *shader* para el procesamiento de vértices (*vertex shading*).

Primitive assembly

Recoge los vértices procesados y forma las primitivas a partir del conjunto de datos que recibe de la etapa anterior.

Clipping

Determina qué triángulos se encuentran dentro del *frustrum* de visión y elimina los que están fuera.

Triangle setup

Se calculan los planos que se forman con los bordes del triángulo y los parámetros necesarios para interpolar la profundidad de los fragmentos que lo forman.

Fragment generator

Recorre el área del triangulo y genera agrupaciones de fragmentos. ATTILA soporta dos algoritmos de generación de fragmentos: uno basado en agrupaciones y otro recursivo.

Hierarchical Z

Detecta qué fragmentos ya han sido cubiertos por otros ya renderizados antes de que lleguen al *test* de profundidad y descartarlos de antemano. También se descartan los fragmentos que se marcaron como fuera de la ventana de renderizado.

Z & Stencil test

Recibe los fragmentos en grupos de 2x2 llamados *quads* (unidad de transmisión utilizada a partir de esta etapa) y determina si pasan los *tests* de profundidad y *stencil*. Utiliza una cache con el fin de explotar la localidad en los accesos al buffer de profundidad y *stencil*.

Interpolator

Interpola los atributos de los fragmentos a partir de los valores de los vértices del triángulo. Utiliza interpolación lineal con corrección de la perspectiva. Después transfiere los *quads* a los *shaders* para que sean procesados (*fragment shading*).

Blend

Se encarga de actualizar el buffer de color a partir de los fragmentos procesados. Al igual que la unidad Z y *Stencil test* implementa una cache y soporta el borrado rápido.

Memory controller

Es la unidad encargada de acceder a la memoria de la GPU. Todas las unidades de la GPU que necesitan acceder a memoria lo hacen a través de este controlador. La unidad mínima de acceso a memoria tiene un tamaño de 64 bytes e implementa la especificación GDDR3.

Shader

Son procesadores con registros de cuatro elementos. Se encargan de procesar los vértices y los fragmentos dependiendo del programa que tenga asignado. Su implementación en *RTL/Verilog* es el objetivo de este proyecto.

Texture Unit

Hay una por cada *shader* y se encarga de acceder y filtrar la información de las texturas. Implementa una cache con el fin de disminuir los accesos a memoria a través del *memory controller*.

En la imagen se muestra un esquema de la arquitectura de ATTILA. Como se puede ver tiene en común las características principales del *pipeline* gráfico.

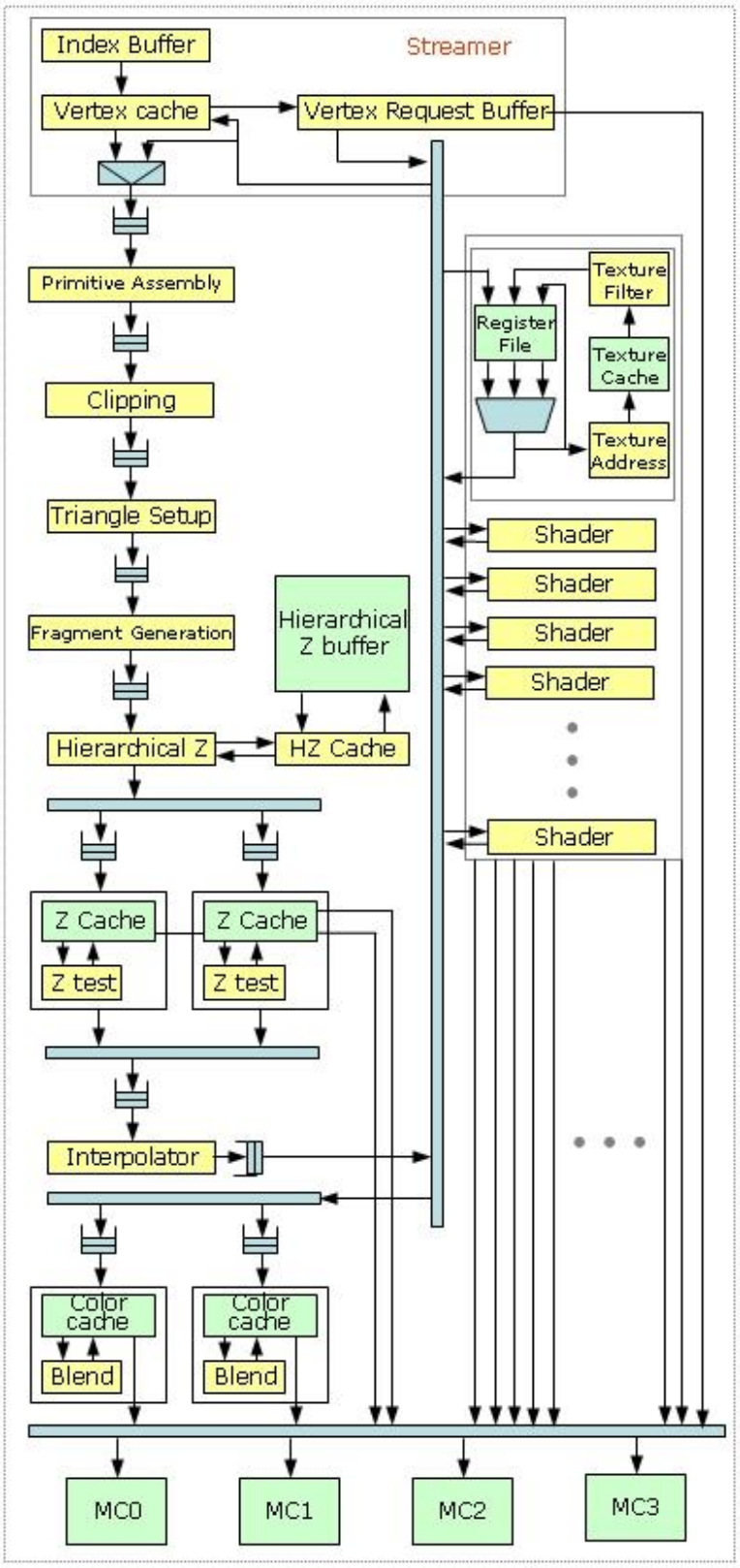

Figura 3.1 Arquitectura de ATTILA

4. Hardware Description Languages

En electrónica, un lenguaje de descripción de hardware (HDL) es cualquier lenguaje para la descripción y diseño formal de circuitos electrónicos, normalmente lógica digital. Puede describir el funcionamiento de un circuito, su diseño y organización, y verificarlo utilizando herramientas de simulación.

HDL define el comportamiento temporal y espacial del sistema electrónico. Igual que en los lenguajes de programación concurrente, la sintaxis y semántica de HDL incluye notaciones para expresar concurrencia. Sin embargo, en contraposición a la mayoría de lenguajes de programación, se define explícitamente la noción del tiempo.

HDL se utiliza para escribir especificaciones para el hardware, que luego se ejecutarán. El diseño utilizando HDL permite implementar y simular el hardware sin la necesidad de crearlo físicamente. El hecho de que se pueda ejecutar da la sensación de estar tratando con un lenguaje de programación convencional, cuando en realidad se trata de un lenguaje de especificación o modelado. Los simuladores soportan tanto modelos digitales como analógicos.

Es posible representar la semántica del hardware usando lenguajes tradicionales como C++, añadiendo librerías. Además, estos lenguajes no incluyen ninguna capacidad de expresar el tiempo, y es por esto que no podrían funcionar de la misma forma que los lenguajes de descripción de hardware.

A partir del lenguaje se pueden inferir operaciones lógicas y producir una serie de primitivas hardware que implementen el comportamiento especificado. Para hacer esto se utilizan unas herramientas llamadas sintetizadores. Los sintetizadores de lógica digital, toman normalmente cambios de flanco en la señal de reloj como medida de tiempo del circuito.

Con el desarrollo de los sintetizadores, los lenguajes de descripción de hardware ganaron protagonismo, pasando a un primer plano en cuanto a diseño de sistemas

digitales. Los ficheros compilados escritos en HDL se pasan a nivel de puertas lógicas y transistores. Escribir código sintetizable requiere práctica por parte del diseñador.

A través de los años se ha hecho un esfuerzo por mejorar los lenguajes de descripción de hardware. Uno de los más importantes es *Verilog*, que es el que se ha utilizado para esto proyecto, explicado en el siguiente punto.

4.1. Verilog

Verilog es un lenguaje de descripción de hardware que se utiliza para modelar sistemas electrónicos. La diferencia fundamental con un lenguaje de programación de *software* es que permite describir la naturaleza intrínseca del hardware.

Como cualquier lenguaje de descripción de hardware, *Verilog* incluye formas de describir la propagación del tiempo y la dependencia de señales. Como estos conceptos forman parte de la semántica del lenguaje, los diseñadores pueden escribir descripciones de grandes circuitos de forma compacta y concisa.

En el momento de la introducción de *Verilog* (1984), se presenció un incremento en la productividad de diseño de circuitos hasta ahora nunca vista, en comparación a las herramientas que se utilizaban hasta el momento.

Los diseñadores de *Verilog* pretendían que la sintaxis del lenguaje fuese parecida a la de C, que por aquel entonces estaba ampliamente utilizado en la industria de desarrollo de software. Igual que C, *Verilog* es *case-sensitive* y tiene un preprocesador básico, menos sofisticado que ANSI C/C++.

Su control de flujo (*if/else, for, while, case*,...) es equivalente, al igual que la prioridad de las operaciones. Las diferencias se encuentran en la declaración de variables y en la notación para indicar principio y final de bloques, además algunas otras diferencias de menor importancia.

Un diseño en *Verilog* consiste en una jerarquía de módulos. Los módulos encapsulan el diseño de la jerarquía, y se comunican con otros módulos a través de sus entradas y salidas, definidas como puertos de entrada, salida o entrada/salida. Internamente, un módulo contiene una serie de declaraciones, definición de bloques secuenciales o concurrentes, así como también pueden incluir definiciones de sub-módulos.

El concepto de *'cable'* denota tanto el señal en cuanto a bit (4 estados: 1, 0, *floating, undefined*), como 'débil' y 'fuerte, permitiendo que sea transparente a la hora de diseñar. Cuando un cable tiene múltiples fuentes de información, se decide su valor en función de cada una y los valores débil o fuerte.

A partir del diseño descrito en *Verilog* se puede alcanzar una implementación física, para esto se utiliza comúnmente una FPGA (*Field Programmable Gate Array*). El flujo de diseño desde que define la arquitectura hasta llegar a obtener una implementación física sigue los pasos que se muestran en el esquema siguiente:

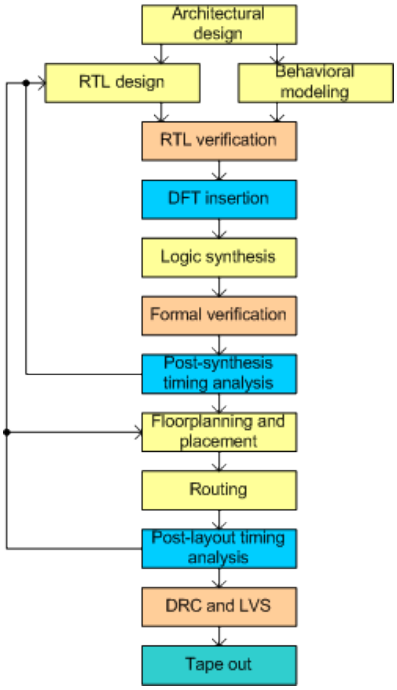

Figura 4.1 Flujo de diseño. De RTL a implementación física

Un conjunto de sentencias escritas en *Verilog* son sintetizables. Los módulos que se ajustan a un estilo de código sintetizable, conocido como RTL (*register-transfer level*), se pueden realizar físicamente con un software de sintetización. Este software sigue un algoritmo que transforma el código fuente de *Verilog* en una *netlist*, un equivalente lógico a la descripción del circuito utilizando primitivas lógicas (*AND, OR, NOT, flip-flops*, etc.) que están disponibles en una FPGA específica.

Una FPGA es un dispositivo que contiene bloques de lógica ya implementados, de los que solo falta realizar sus conexiones, esto se configura utilizando un lenguaje HDL.

La lógica programable puede reproducir las funciones desde una puerta lógica o un sistema combinacional, hasta sistemas complejos. La utilización de FPGAs en el diseño de hardware proporciona una gran flexibilidad a la hora de trabajar diseñando sistemas electrónicos.

5. Shaders

5.1. Introducción

Hasta 2001, momento en que *nVIDIA* comercializó la GeForce 3, la arquitectura de una tarjeta gráfica no permitía que los desarrolladores programaran sus propios *shaders*. Hasta entonces el pipeline tenía un aspecto como el de la figura.

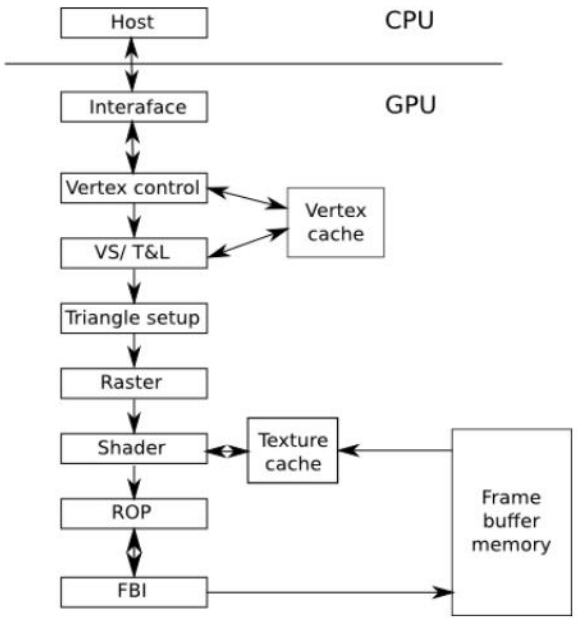

Figura 5.1 *Pipeline* fijo

Los *shaders* son pequeños programas que substituyen las etapas VS/T & L i *shading* de la figura anterior, dando lugar a un pipeline programable, que en las tarjetas más modernas permite incluso modificar la geometría de los modelos 3D.

El uso de *shaders* permite realizar cálculos sobre los vértices o los píxeles, de forma que por cada uno se ejecute un código. Esto permite implementar modelos de iluminación alternativos y todo tipo de efectos especiales. Los procesadores de *shader* están diseñados específicamente para realizar con eficiencia operaciones matemáticas en coma flotante, tales como divisiones, raíces cuadradas, etc.

El *pipeline* visto en la figura 5.1 evolucionaba a uno completamente programable,

donde los procesadores de *shader* toman protagonismo al permitir crear pequeños programas para manipular los vértices y los pixeles sin depender de un algoritmo definido de antemano.

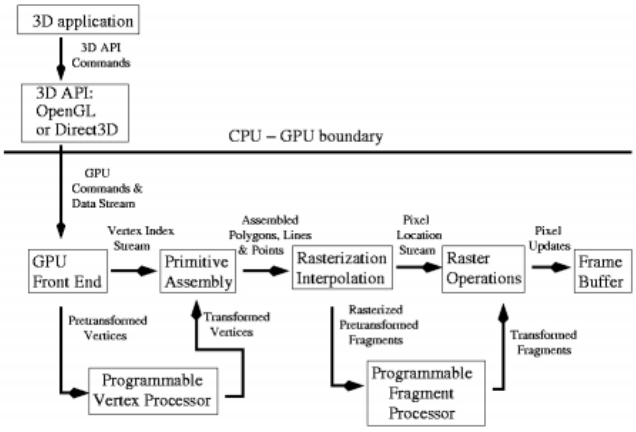

Figura 5.2 *Pipeline* programable

A medida que al hardware gráfico se le añadían más procesadores, se acercaba cada vez más a las CPU de alto rendimiento paralelo, hasta el punto que se podían resolver problemas científicos de un alto coste computacional utilizando una GPU.

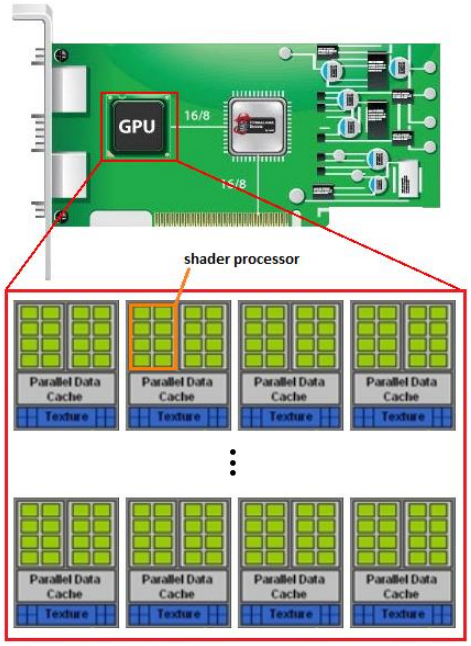

Figura 5.3 Arquitectura de una GPU

Cada uno de estos *cores* es mucho más sencillo que el que se puede encontrar en una CPU, sin embargo al tener un número tan elevado, dota a la GPU de un paralelismo mucho mayor.

Su simplicidad se debe a que los *shaders* que ejecuta se utilizan para hacer cálculos de iluminación, modificar fragmentos y vértices, para lo que no se requiere un gran número de operaciones ni mucho control de flujo.

5.2. Procesadores gráficos unificados

La arquitectura de las GPU de generaciones anteriores disponía de unidades de ejecución separadas para el procesamiento de vértices y de fragmentos. Aunque en principio se tenían ciertas ventajas, también tenían efectos negativos en la eficiencia. Por ejemplo, si una escena con mucho coste en el *fragment shader* y poco en el *vertex*, la carga de una estará al máximo, mientras la otra estará en prácticamente en reposo. La única forma de solucionar esto es unificándolos, de manera que se pueda asignar la carga de forma dinámica.

Figura 5.4 Comparación de *pipeline* unificado y separado

ATTILA implementa el *pipeline* unificado, así que como el procesador que se ha implementado ejecuta las instrucciones de ATTILA, será capaz de ejecutar tanto *vertex shaders*, como *fragment/pixel shader* una vez se complete el juego de instrucciones y se implementen el resto de módulos necesarios.

6. Entorno de trabajo

El proyecto se ha desarrollado en un sistema Linux, y todas las herramientas utilizadas se pueden conseguir de forma gratuita:

- Compilador/Simulador *Verilog*.
- Compilador C++.
- Editor de texto.

El compilador y simulador de *Verilog* se puede obtener de la página oficial de *Altera*, la versión concreta utilizada para el proyecto ha sido *ModelSim-Altera Starter Edition 6.6d for Quartus II v11.0 SP1*. Esta herramienta es fundamental para el proyecto, ya que permite hacer simular el procesador mostrando las señales en un cronograma como el de la figura 7.1.

El compilador para C++ está incluido por defecto en la distribución de Linux, será necesario para compilar el código de *ATTILA* y otros programas que se utilizan en la validación del modelo.

Para escribir el código en *Verilog*, es suficiente con un editor de texto, cualquiera de los que se encuentran disponibles por defecto sirven.

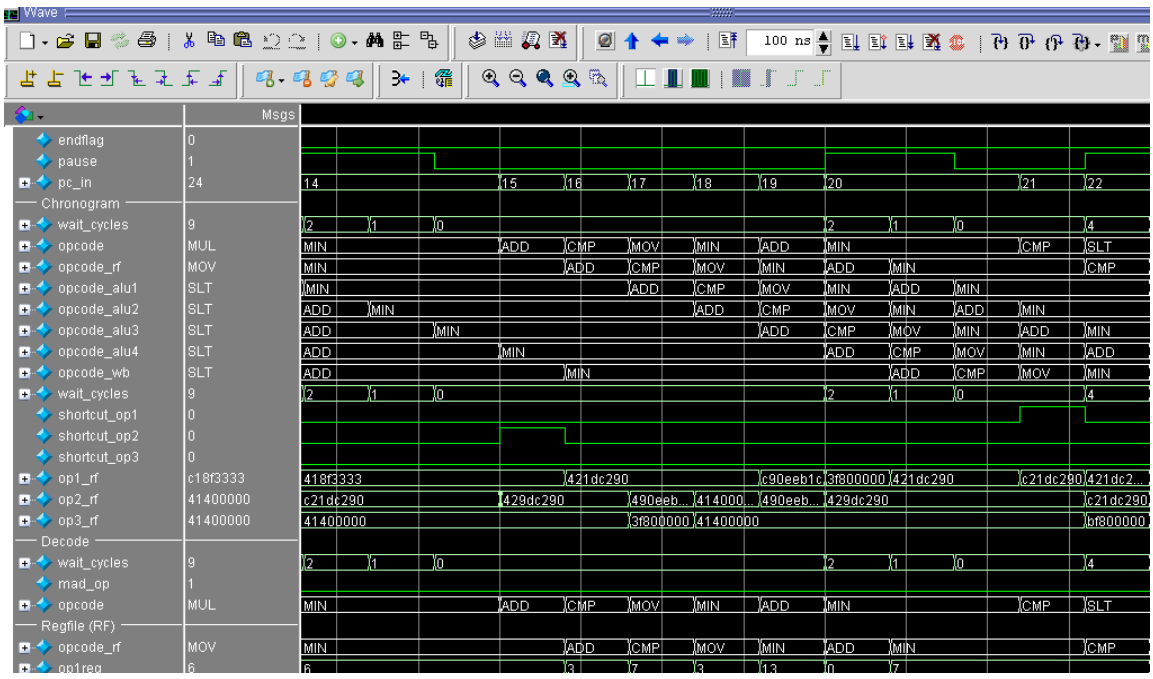

Figura 6.1 Ventana de simulación de *modelsim*

7. Diseño del *pipeline*

7.1. Introducción

Antes de empezar con la implementación del procesador, es imprescindible tener claro el diseño, esto evitará que más tarde se tengan que hacer correcciones innecesarias, ahorrando tiempo y trabajo, ya que modificar una implementación ya hecha es bastante costoso.

El diseño más sencillo para un procesador es no segmentado, sin embargo el rendimiento que se obtiene es muy limitado, por lo tanto lo más apropiado es diseñar un *pipeline* segmentado, de forma que a cada ciclo se pueda lanzar una instrucción, y por lo tanto también acabar una a cada ciclo. Aun así, como máximo se puede obtener un IPC = 1 (Instrucciones Por Ciclo), este no será siempre el caso porque algunas instrucciones requieren varios cálculos para completarse, o pueden producirse dependencias de datos, obligando al procesador a bloquearse.

Las etapas que se han definido para la implementación del procesador son las siguientes:

- Fetch
- Decode
- Register file
- ALU
- Write back

De esta forma, se pueden estar ejecutando simultáneamente, tantas instrucciones como etapas tiene el *pipeline*. Como se ha explicado en la introducción, el procesador de *shaders* de este proyecto es una implementación en *Verilog* del que ya dispone ATTILA, con lo que las instrucciones que se ejecutarán ya están definidas.

Una de las características más importantes es que opera con datos en coma flotante, concretamente se codifican siguiendo el estándar IEEE 754 (Anexo III). Es importante

tener esto en cuenta, ya que *Verilog* no dispone de mecanismos ya implementados para tratar con ellos directamente, por lo que será necesaria una unidad de cálculo adicional para este caso.

Otro elemento que se ha incorporado, y que se explicará en el apartado siguiente, son los cortocircuitos. Éstos permiten reducir la latencia de una instrucción porque se permite alimentar una etapa con un dato ya calculado por una instrucción anterior, pero que aun no se ha escrito en el banco de registros.

En la figura 8.1 se muestra un esquema simplificado del procesador que se ha implementado para el proyecto.

Cada etapa del *pipeline* se ha implementado en *Verilog* en un módulo distinto, además de algunos módulos adicionales. La implementación final está compuesta por los siguientes módulos:

- Fetch: implementa la etapa *fetch*.
- Decode: implementa la etapa *decode*.
- Regfile: implementa la etapa *register file*. Además utiliza un sub-módulo para modificar los operandos.
    - Neg_abs: aplica a cada operando el flag *neg*, *abs*, o ambos en función de la instrucción.
- ALU: implementa la etapa ALU, y a su vez utiliza dos sub-módulos.
    - FPU: se encarga de hacer los cálculos en coma flotante.
    - Comparator: se encarga de resolver las comparaciones en coma flotante.
- Write back: implementa la etapa *write back*.
- Shader: es el módulo de más alto nivel, y es el encargado de conectar las

entradas y salidas de los módulos de cada etapa.

A parte de los módulos que implementan el procesador, se ha añadido uno que servirá para validar el modelo, es el módulo *validation*, que se explicará en el apartado 11.

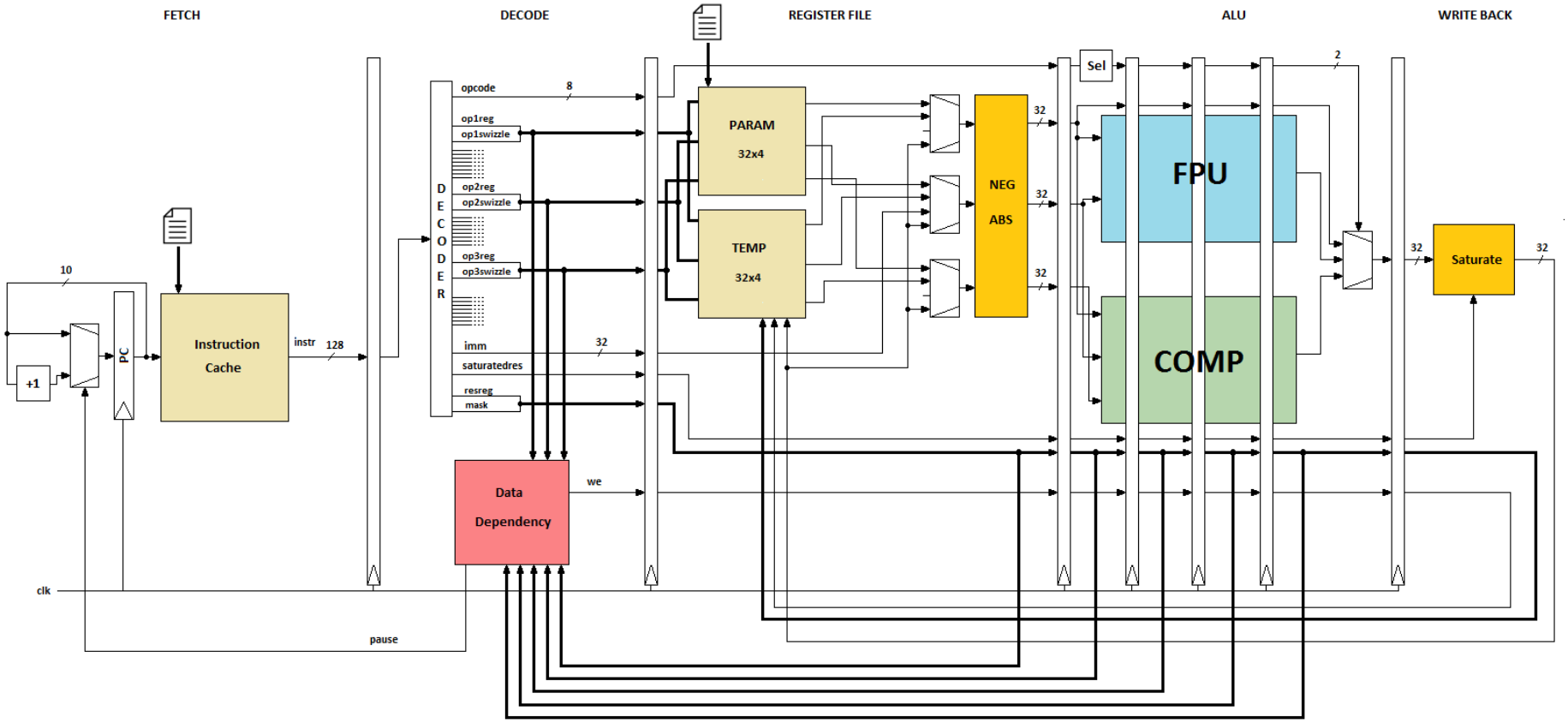

Figura 7.1 Esquema simplificado del *pipeline* implementado

7.2. Fetch

7.2.1. Descripción

Es la primera etapa del procesador, y su función es simplemente proporcionar a la etapa de *decode* la instrucción a ejecutar.

Los elementos principales son una memoria donde al inicio de la simulación se cargan las instrucciones que forman el *shader*, es decir, el programa a ejecutar, y el PC (*Program Counter*), que como su nombre indica es un contador que guarda el valor de la siguiente instrucción. Como norma general, a cada ciclo se incrementa el PC en una unidad.

Cada instrucción tiene un tamaño de 128 bits, será necesario por tanto que el tamaño de la memoria sea 128xN, siendo N el número máximo de instrucciones que podrá tener un programa. Para este proyecto se ha considerado que 1024 era suficiente.

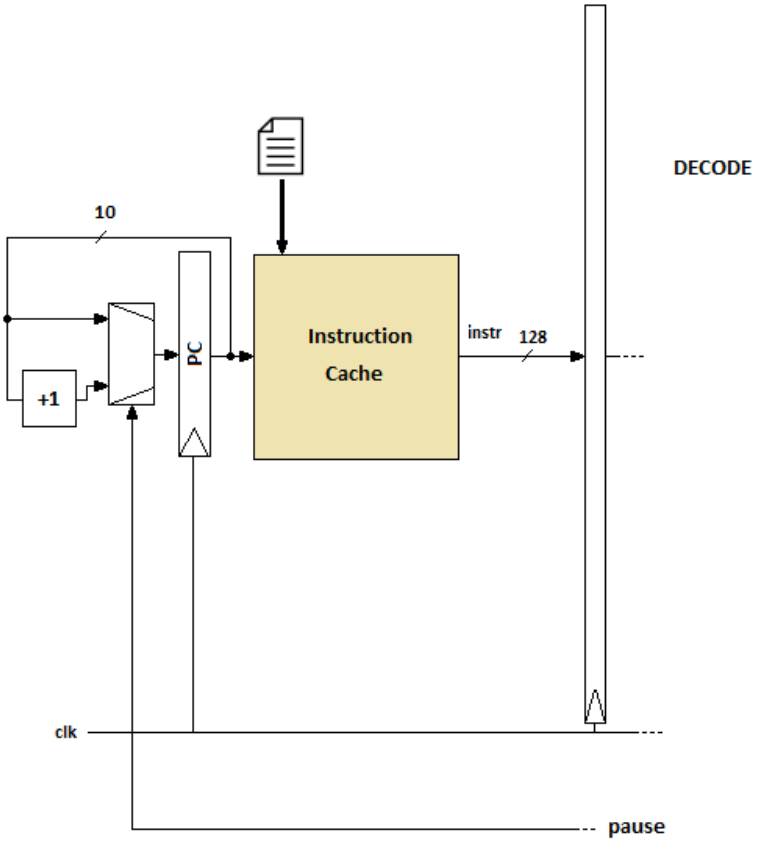

Figura 7.2 Esquema de la etapa *fetch*

7.2.2. Implementación

A cada ciclo se lee la instrucción de memoria a donde apunta el PC, y se pasa a la etapa de decodificación. Si no hay bloqueo del procesador el PC se incrementa.

La memoria puede almacenar hasta 1024 instrucciones, cada una de 128 bits, por lo tamaño del PC es de 10 bits, $2^{10} = 1024$. Su implementación es igual que la de un banco de registros y se inicializa al arrancar el procesador con las instrucciones que se han generado a partir de un fichero que contiene el código fuente.

La única información que proporciona a la siguiente etapa son los 128 bits de la instrucción leída de memoria.

En caso de dependencia de datos, la señal *pause* que proviene de la etapa de decodificación indica que hay que bloquear el procesador, provocando que el PC no se actualice.

7.3. Decode

7.3.1. Descripción

Es la segunda etapa del *pipeline* y como su nombre indica se encarga de decodificar la instrucción que proviene de la etapa de *fetch* a partir de los 128 bits que recibe.

El conjunto de funciones de esta etapa son:

- Decodificación de la instrucción.
- Detectar dependencias de datos.
- Indicar el bloqueo del procesador.
- Indicar a las etapas posteriores si la instrucción es válida y si tiene permiso de escritura.
- Activar los cortocircuitos.

La decodificación solo requiere los bits que se reciben de la etapa *fetch*. Por otra parte, según los registros de la instrucción decodificada, se detectarán las dependencias de datos, comprobando que ninguno de ellos coincide con el registro de destino de una instrucción anterior que aun no ha acabado, si es el caso indicará que es necesario bloquear el procesador. Por último indica si la instrucción que se está decodificando es válida para ser ejecutada en el resto del *pipeline*, y si tiene permiso de escritura. La activación de los cortocircuitos permitirá reducir en un ciclo la latencia de una instrucción cuando tenga dependencia de datos.

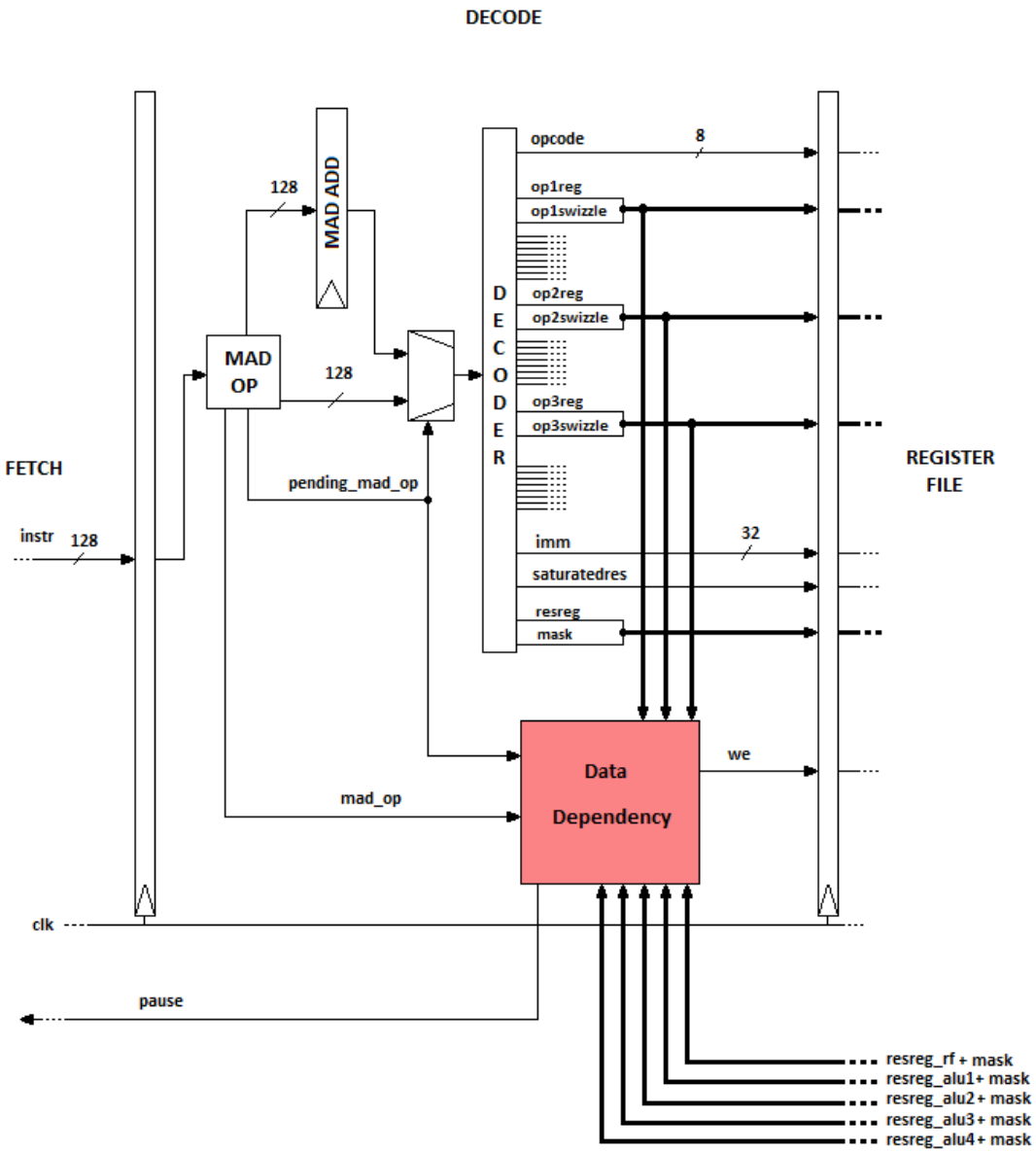

Figura 7.3 Esquema de la etapa *decode*

7.3.2. Implementación

La decodificación para todas las instrucciones se hace siempre de la misma forma, ya que el tamaño de cada una es fijo (128 bits), por lo tanto solo hay que dividir los 128 bits en los campos que forman la instrucción (Anexo III).

Un caso especial es la decodificación de la instrucción MAD, que al haberse dividido en dos (MUL + ADD), se genera un MUL específico para esta operación y se bloquea el procesador hasta que el cálculo acabe. Una vez acabado, se manda a ejecutar la parte del ADD, también generada de forma especial para este caso, y que se ha guardado en un registro a la hora de decodificar el MAD original, indicado como “MAD ADD” en la figura 8.3.

Para determinar si hay dependencia de datos con una instrucción que se encuentra en una etapa más avanzada, utiliza señales que provienen de cada una de las etapas posteriores indicando el banco y el registro de destino. Necesita conocer el registro de destino de cada las instrucciones que se están ejecutando en las etapas *Register File*, *ALU1*, *ALU2*, *ALU3* y *ALU4*, además de si la instrucción que se encuentra en cada una de estas etapas es válida. Es necesaria toda la información del registro de destino, es decir, número de registro, banco y máscara. La detección de dependencia de datos se explica ampliamente en el apartado 8 de este documento.

Cuando la dependencia de datos se ha resuelto, se considera que la instrucción es válida, y se activa una señal que se propagará por el resto del *pipeline*. Esta señal es necesaria para detectar dependencias de datos posteriores correctamente.

Al igual que la señal de validez, la señal de permiso de escritura se activa una vez no hay dependencia de datos. El comportamiento de ambas señales es idéntico excepto en dos casos:

- Ejecución de la multiplicación de la instrucción MAD.
- Ejecución de la instrucción NOP.

En esta etapa también se activan las señales correspondientes a la utilización de

cortocircuitos que se utilizarán si es necesario en la etapa *register file*. La implementación en este caso, utiliza la señal que proviene de la etapa *ALU4*. Si la dependencia solo se tiene con la instrucción que se encuentra en esta etapa, se utilizará un cortocircuito para reducir un ciclo el bloqueo. Los cortocircuitos se explican detalladamente en el apartado 9.

7.4. Register file

7.4.1. Descripción

En esta tercera etapa, la instrucción ya está decodificada, y es momento de leer los valores de los registros para hacer el cálculo.

Siguiendo las especificaciones del procesador de *shaders* de *ATTILA*, hay dos bancos de registros, uno que guarda valores constantes y que se carga al inicio de la simulación (banco PARAM), y otro que guardará valores temporales (banco TEMP).

Como el procesador está diseñado para soportar el modo SOA, los registros trabajarán sobre una sola componente (.x, .y, .z, .w), así pues, cada banco de registros estará formado por 32 registros de 128 bits, o lo que es lo mismo, cada registro de 128 bits formará 4 de 32 bits, uno por cada componente.

El procesador de *ATTILA* soporta también instrucciones *SIMD4* (*Simple Instruction Multiple Data*), esto quiere decir que con solo una instrucción se puede realizar 4 cálculos. En este proyecto no se ha implementado porque las nuevas generaciones de GPU tienden a seguir el modelo SOA. La única variación en esta etapa sería la lectura de registros, no la distribución de los bancos.

A cada ciclo el banco de registros proporciona a la ALU tres operandos para que realice el cálculo, no importa si los operandos que se piden son de una instrucción válida o no, en caso de no serlo la ALU hará algún cálculo de todas formas, pues no hay un mecanismo para evitarlo, pero no importa porque ese resultado nunca se escribirá en el banco de registro, con lo que en ningún caso se modificaría el estado del procesador. Aun así hay que tener en cuenta que si se alimenta la ALU con datos aleatorios, podría darse el caso que se hiciera una división por cero, u otro tipo de excepción. La solución es tan simple como activar la señal de escritura en el banco de registros solamente cuando se trate de una instrucción válida.

La lectura de los valores se hace a partir de los datos proporcionados por la etapa *decode*. En general la lectura se hace directamente de los bancos de registros, sin embargo, en caso de haber dependencia de datos, se utilizará un cortocircuito para

reducir la espera en un ciclo, de forma que se descarta el valor leído del banco de registros, y se utiliza el resultado proporcionado desde la etapa *write back*. También es posible que el segundo operando sea un inmediato codificado en la propia instrucción ('imm' en la figura 8.6).

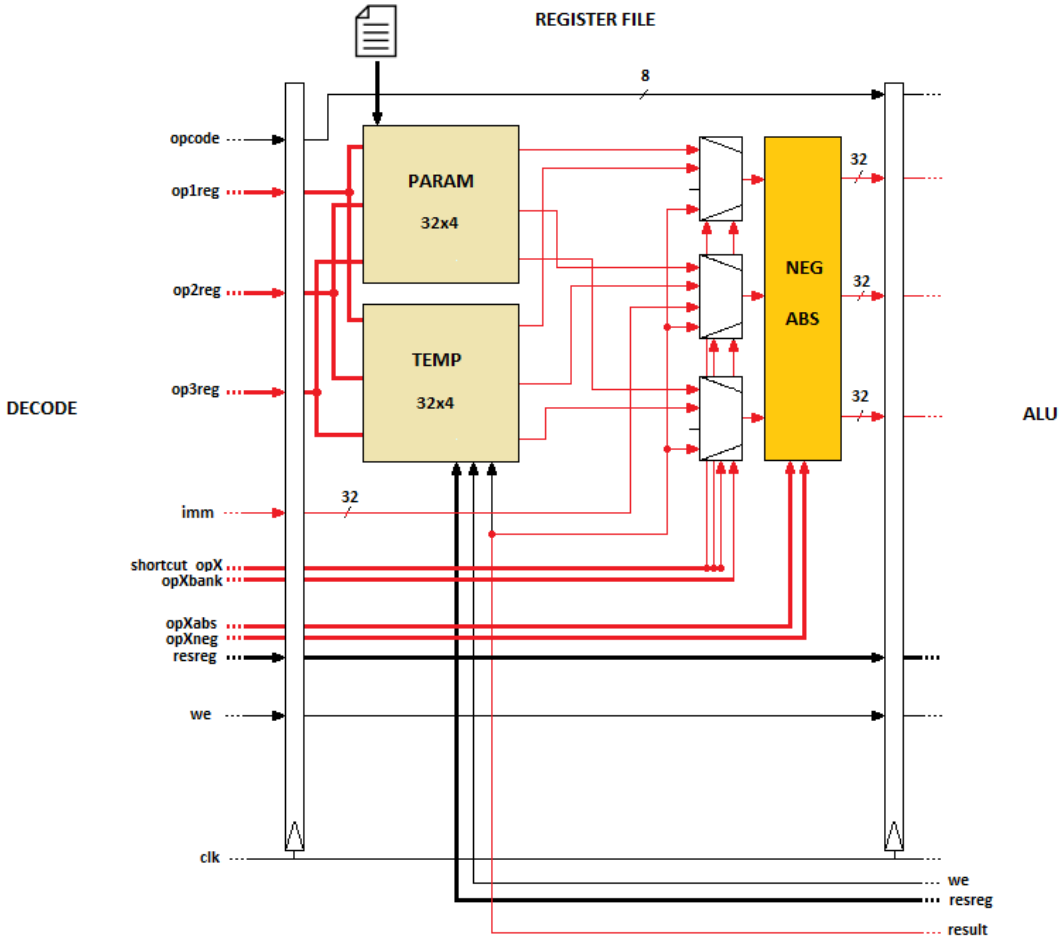

Figura 7.4 Esquema de la etapa *register file*

7.4.2. Implementación

Una vez se conoce la finalidad de esta etapa, se detalla la implementación que se ha seguido para conseguir el funcionamiento descrito. Se explicará la arquitectura de los bancos de registro, y cómo se hace la lectura de los registros a partir del número de registro, banco y *swizzle*, o en caso de utilizar un cortocircuito, de qué forma se hace la selección.

7.4.2.1. Arquitectura de los bancos de registro

Como se ha explicado antes, se dispone de dos bancos, uno que guarda valores constantes (PARAM), que se inicializa al arrancar el procesador, y otro para datos temporales (TEMP). Cada uno está formado por 32 registros, cada uno formado por cuatro componentes (.x, .y, .z, .w), esto es, una matriz de 32x4 posiciones.

La implementación en *Verilog* se ha hecho de la misma forma que el lenguaje de programación C guarda las matrices en memoria, es decir, como si fuera un *array* de 32x4 posiciones, donde el registro que se quiere leer se calcula de la siguiente manera:

```
opXaddr = (opXreg << 2) + swizzle
```

Esta implementación es más simple que definir los bancos como una matriz, aunque *Verilog* lo soporte, y haya que hacer un cálculo previo para acceder a los registros.

En la figura 8.7 se muestra el esquema de la arquitectura de un banco de registros, y el cálculo para acceder a cada registro.

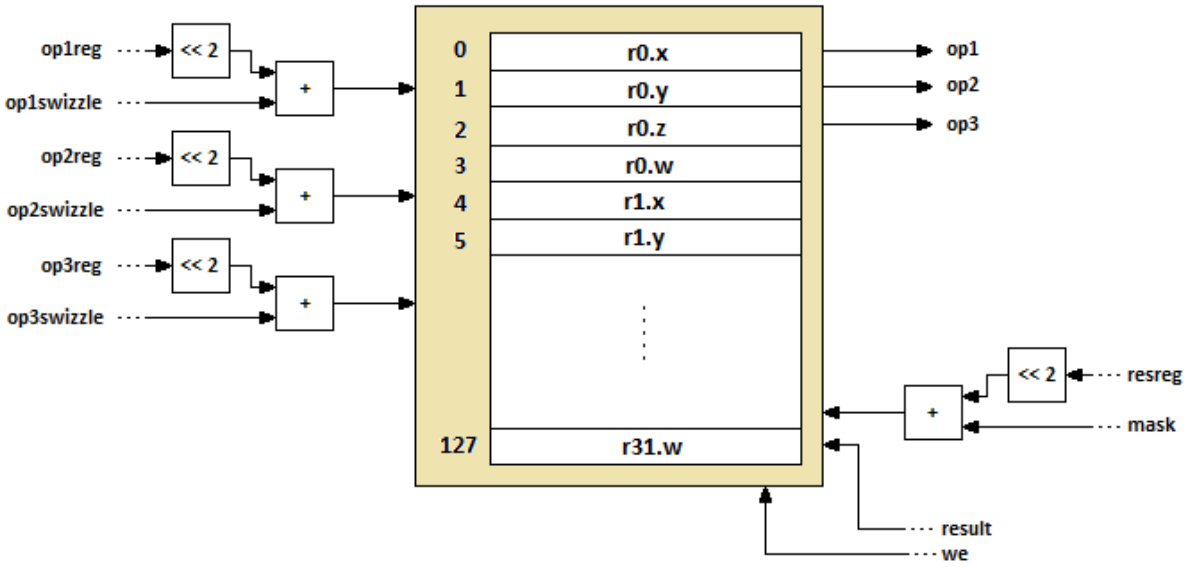

Figura 7.5 Arquitectura del banco de registros

Lectura de los operandos

En cada ciclo se leen tres operandos de los bancos de registros, aunque la instrucción no los requiera, se leerán tres y será la ALU la encargada de utilizarlos. No siempre se necesitan los datos que se encuentran en un registro, por ejemplo si hay dependencia, se tendrá que utilizar un cortocircuito. El último caso que queda por tratar, es tomar como operando el campo de inmediato que se ha decodificado en la instrucción, esto solo puede suceder en las instrucciones que tienen dos operando, donde el inmediato substituye siempre al segundo operando.

La selección del dato que se utilizará se observa en la figura 8.8, ambos bancos proporcionan tres operandos cada uno, en función de los parámetros de la instrucción, por otra parte se tienen cables que proporcionan directamente el valor del resultado que se encuentra en la etapa *write back* (cortocircuito), y el valor del inmediato de la etapa *decode*, solo para el segundo operando.

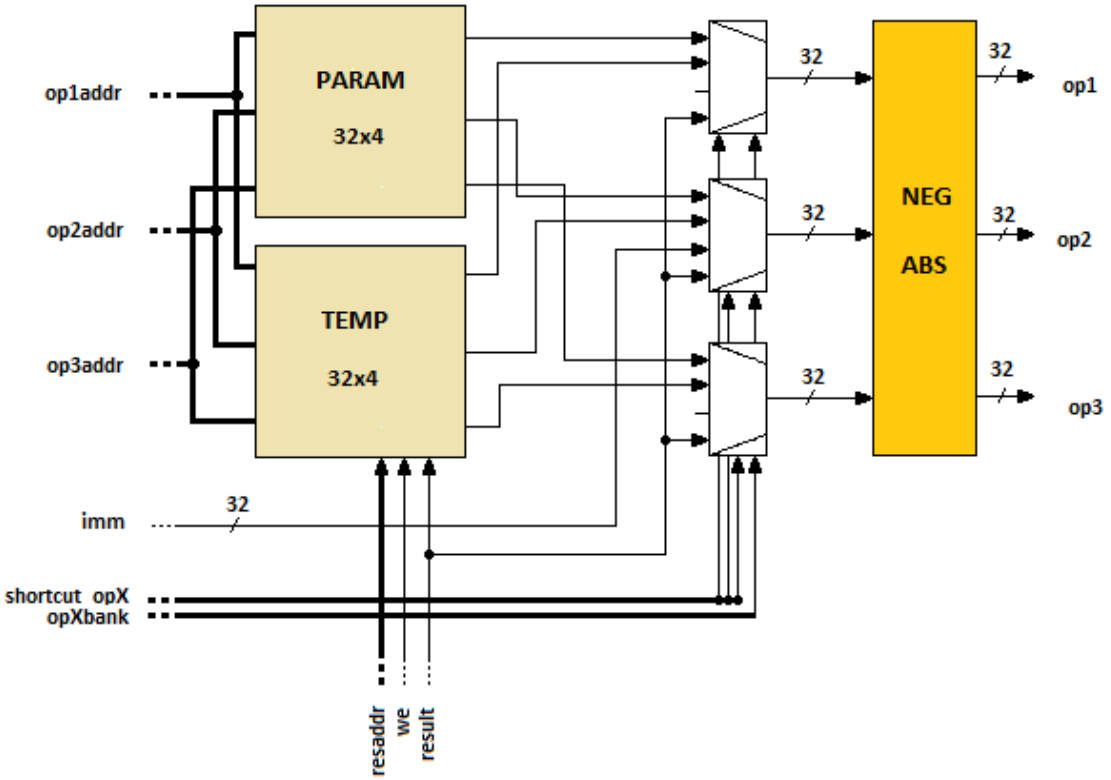

Figura 7.6 Selección de los operandos

En caso de ser necesario un cortocircuito se selecciona directamente esa entrada en los multiplexores, igual pasa con el inmediato, solo que en este caso hay que tener en cuenta si la instrucción utiliza un inmediato, o en realidad está definiendo otros parámetros utilizando esos bits, esto se sabe porque el segundo operando tendrá como valor de *op2bank* 0x6. Por último, si no se selecciona ninguna de las dos opciones anteriores, el valor que se utilizará proviene de uno de los dos bancos, en función de los bits que definen cada *opXbank*.

7.5. ALU

7.5.1. Descripción

Llegados a esta etapa, el procesador ya está en disposición de hacer el cálculo de la operación. La ALU (*Arithmetic Logic Unit*) permite cuatro operaciones: sumar, restar, multiplicar y dividir números en coma flotante siguiendo el estándar IEEE 754, además de incluir un módulo para la comparación de valores.

Implementar una ALU, con una unidad para hacer cálculos en coma flotante (FPU – *Floating Point Unit*) desde cero es un trabajo complicado, por lo que se ha optado por buscar una ya implementada. En un principio el objetivo era que pudiera hacer cálculos complejos como 1/sqrt(x), sin(x) o cos(x), pero ante la imposibilidad de encontrar una con estas características, se tomó la decisión de solo ejecutar las instrucciones que fueran posibles con las cuatro operaciones básicas.

Era necesario para poder utilizar una FPU externa, que fuera segmentada, en nuestro caso cada operación requiere 4 ciclos. Esto no es del todo real, ya que la multiplicación y división son procesos iterativos que dependen de la longitud de los operandos, sin embargo como el procesador no llegará a la fase de síntesis, es suficiente para este proyecto.

Las señales básicas para su funcionamiento son cuatro, dos para cada uno de los operandos de 32 bits, la operación a realizar y otra que indica el tipo de redondeo a aplicar al resultado de la operación. Como salida se tiene el resultado de 32 bits, y las señales que indican una excepción. Siguiendo el comportamiento de ATTILA, el procesador no parará su ejecución si se da una de estas excepciones, pues el estándar IEEE 754 contempla valores especiales para estos casos.

A cada ciclo la ALU produce un nuevo resultado, no importa porque el resultado solo se escribirá en el banco de registros cuando esté activado el permiso de escritura.

Una parte que era necesaria para la ejecución de las instrucciones era la comparación de números en coma flotante. La FPU que se encontró no disponía de esta unidad, por lo tanto ha sido necesario implementarla desde cero. Se ha aprovechado este hecho

para hacerla específica para el procesador del proyecto, de forma que pueda calcular directamente el resultado para todas las instrucciones de comparación que se pueden ejecutar.

La ALU se divide pues, en dos partes, por un lado la FPU y por otro el comparador. En la siguiente figura se observa el esquema simplificado de la ALU.

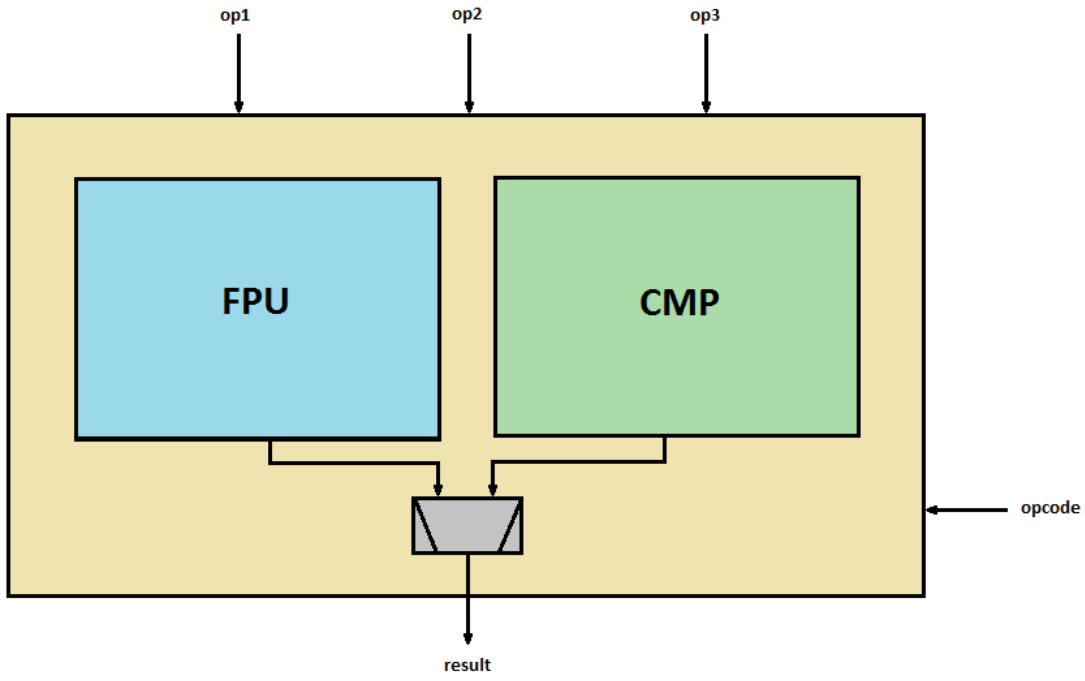

Figura 7.8 Esquema simplificado de la ALU

El resultado de ALU se escoge entre el que da la FPU y el comparador en función de la instrucción que ejecuta. Tanto en un caso como en el otro se tarda siempre 4 ciclos, como se ha especificado para todas las instrucciones del procesador.

En la figura 8.10 se muestra un esquema de la etapa ALU con más detalle.

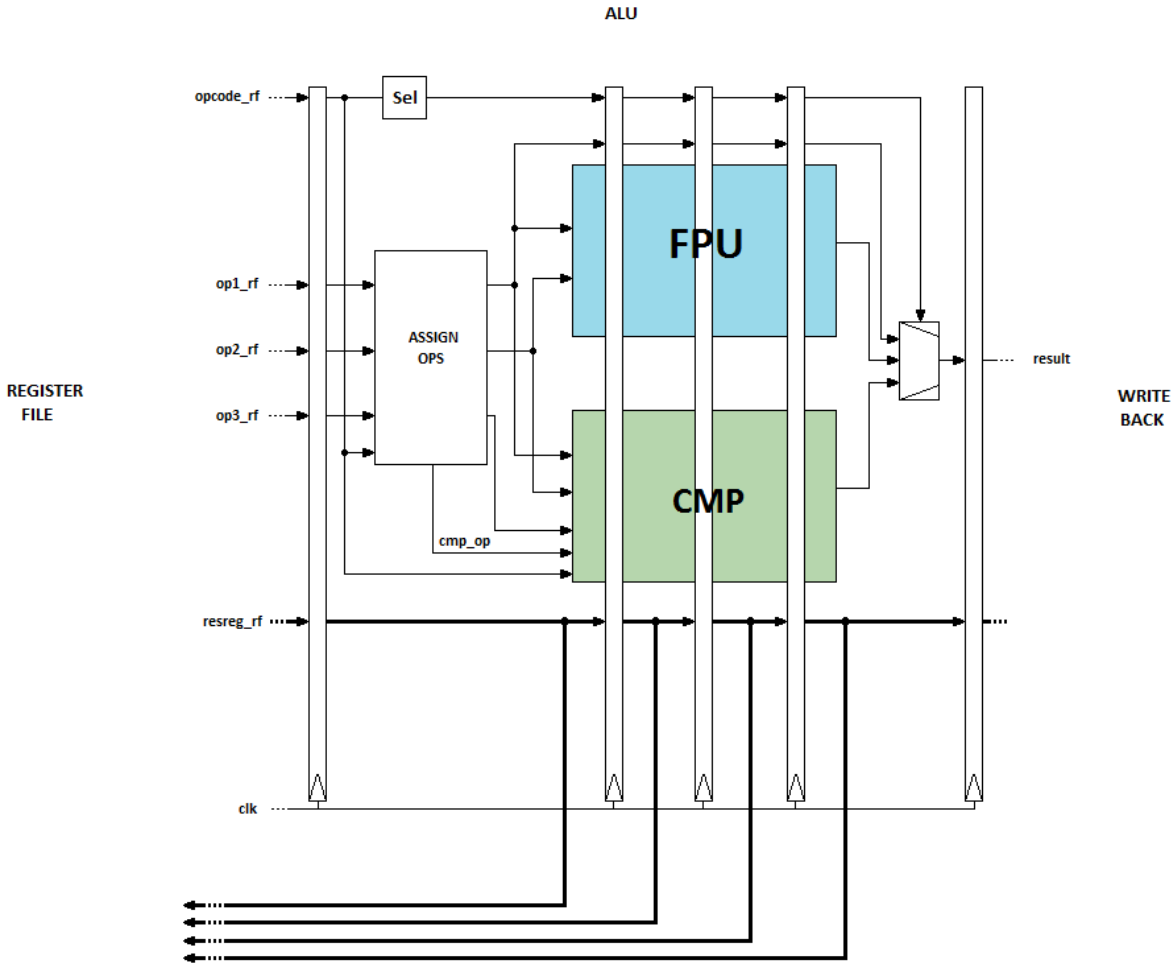

Figura 7.9 Esquema de la etapa ALU

Los operandos para hacer el cálculo que se reciben de la etapa *register file*, se tienen que modificar para algunas instrucciones en concreto, es lo que en la figura anterior aparece indicado como 'Assign ops'.

7.5.2. Floating Point Unit

Como se ha comentado en el punto anterior, la ALU de este proyecto utiliza una FPU ya implementada que puede calcular sumas, restas, multiplicaciones y divisiones. En cualquiera de estos casos se invierten 4 ciclos hasta obtener el resultado final, hecho que hace que el comportamiento no sea real, porque como se describirá a continuación, estas operaciones utilizando el formato especificado por IEEE 754 necesitan una serie de pasos para completarse, que hace imposible el cálculo de la multiplicación y la división en cuatro ciclos.

El formato que se utiliza en este proyecto es la precisión simple (32 bits), esto es:

- 1 bit de signo.
- 8 bits de exponente.
- 23 bits de mantisa.

En el anexo II se puede encontrar una descripción más detallada del estándar IEEE 754.

La FPU que se ha escogido puede calcular una operación en coma flotante cada ciclo, tomando como entrada la operación, modo de redondeo y los operandos, dando el resultado cuatro ciclos después.

Las operaciones que soporta se indican en la siguiente tabla, para este proyecto solo son necesarias las cuatro primeras:

Fpu_op	Operación
0	Suma
1	Resta
2	Multiplicación
3	División
4	Conversión de Int a Float
5	Conversión de Float a Int

Los modos de redondeo también se especifican en el estándar y los soporta todos, hay cuatro:

0	Redondeo al más cercano
1	Redondeo a 0
2	Redondeo a +INF
3	Redondeo a –INF

La arquitectura de la FPU incluye dos unidades de pre-normalización que ajustan las mantisas y los exponentes, una para sumas y restas, y la otra para multiplicaciones y divisiones. Los bloques dedicados para cada operación hacen el cálculo de cada una. Después se pasa a un bloque de normalización común, donde además se redondea el resultado.

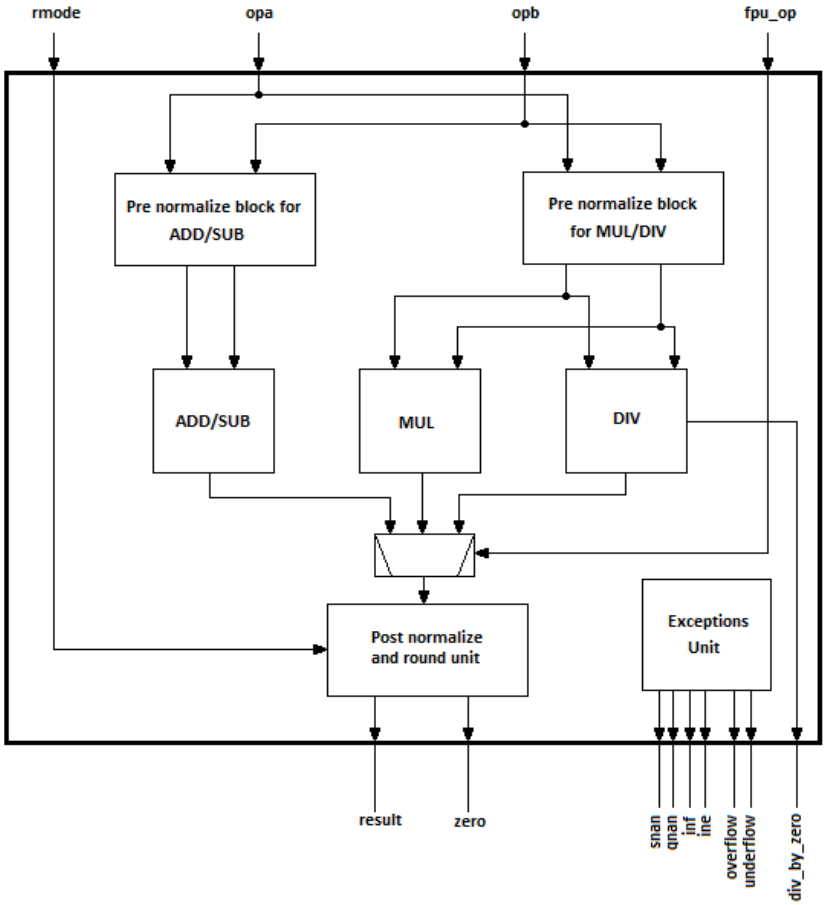

Figura 7.11 Esquema de la FPU

Por último, el estándar contempla cinco excepciones, y define cuándo ocurren y qué sucede. Los valores especiales tienen una representación específica para cada caso. Las excepciones que se pueden dar son:

- Operación inválida.
    - INF +/- INF
    - 0 x INF
    - 0 / 0
    - INF / INF
    - x mod 0
    - Sqrt(x) si x < 0
- Inexacto, cuando el resultado redondeado no coincide con el real.
- Overflow.
- Underlfow.
- División por cero.

Una vez se conoce el funcionamiento general de la unidad siguiendo el estándar, es interesante conocer a grandes rasgos los pasos a seguir para el cálculo de cada operación.

Suma/resta

El procedimiento es el mismo para los dos casos:

- Extraer signos, exponentes y magnitudes.
- Tratar los operandos especiales.
- Desplazar la mantisa del número con exponente más pequeño a la derecha |e1 – e2| bits.
- Fijar el exponente del resultado al máximo de los exponentes.
- Si la operación es una suma y los signos iguales, o si es una resta y los signos

son diferentes, se suman las mantisas, si no, se restan.

- Detectar *overflow* de la mantisa.
- Normalizar la mantisa, desplazándola a la derecha o a la izquierda hasta que el dígito más significativo esté delante de la coma decimal.
- Redondear el resultado y re-normalizar la mantisa si es necesario.
- Corregir el exponente en función de los desplazamientos realizados sobre la mantisa.
- Detectar *overflow* o *underflow* del exponente.

Multiplicación

Los pasos que sigue la multiplicación son:

- El signo del resultado se calcula con la operación *xor*.
- La mantisa del resultado es igual al producto de las mantisas.
    - Como los dos operandos se encuentran en el intervalo [1, 2], el resultado estará en [1, 4). Esto puede requerir una normalización, desplazando a la derecha y ajustando el exponente del resultado.
- Si la mantisa resultado es del mismo tamaño que la de los operandos habrá que redondear, lo que puede obligar a una normalización posterior.
- El exponente del resultado es igual a la suma de los exponentes de los operandos. Al sumar los exponentes se está sumando dos veces el sesgo (desplazamiento), por lo que habrá que restarlo: exp_r = |exp1 + exp2 – sesgo|.

División

El proceso es similar a la multiplicación, sin embargo hay que tener en cuenta algunos puntos:

- Al hacer la resta de los exponentes hay que considerar que los sesgos se

anularán, así que hay que volver a sumarlo.

- Al operar con números normalizados, la mantisa del resultado será:
    - 0,5 < r < 2. Esto implica que la única normalización posible es mediante un desplazamiento a la izquierda, para esto se necesitará un bit adicional, *g*.
- Junto este bit, será necesario al hacer la división, añadir un bit más de redondeo, *r*, a la derecha de *g*, y con el resto del resultado se hará una *or* lógica en el bit de signo.

Ahora que se conocen los pasos que se siguen en cada caso, se puede entender por qué una multiplicación o una división no pueden hacerse en cuatro ciclos. El hecho de necesitar el producto o la división de las mantisas, entendiéndolas como números enteros, obligaría a seguir un algoritmo iterativo, que dada la longitud de la mantisa (23 bits), no podría acabarse en cuatro ciclos, salvo en casos concretos.

7.5.3. Comparador

El comparador de la ALU se ha implementado específicamente para el proyecto. El cálculo se divide en dos partes: determinar si el primer operando es mayor (GT – *Greater Than*), igual (EQ – *Equal*), menor (LT – *Less Than*) o diferente (NEQ – *Not* Equal) que el segundo. Después se calcula el resultado final en función de la operación que se esté ejecutando.

Para hacer una comparación son suficientes dos ciclos, pero como el resto de instrucciones, se ha decidido que se inviertan cuatro ciclos.

Las primera fase del comparador solamente se encarga de decidir, a partir de dos operandos, si el primero es mayor, igual o menor, para después utilizar esta información en el siguiente ciclo. Para esta comparación se han tenido en cuenta todos los casos posibles, incluyendo aquellos que dependen de las definiciones concretas para números especiales definidas en el estándar IEEE 754.

Los casos especiales que se pueden encontrar es si alguno de los dos operandos es +INF, -INF, +0.0, -0.0 o NaN. En estos casos la comparación sigue los criterios siguientes:

- +0.0 y -0.0 son comparados como iguales (EQ).
- NaN se considera diferente a cualquier otro, incluso a sí mismo.
- +INF es considerado como mayor, a no ser que se compare con NaN.
- -INF es considerado como menor, a no ser que se compare con NaN.

Para el resto de casos se puede hacer una comparación en función del bit de signo, el exponente y la mantisa para determinar el orden de los dos operandos.

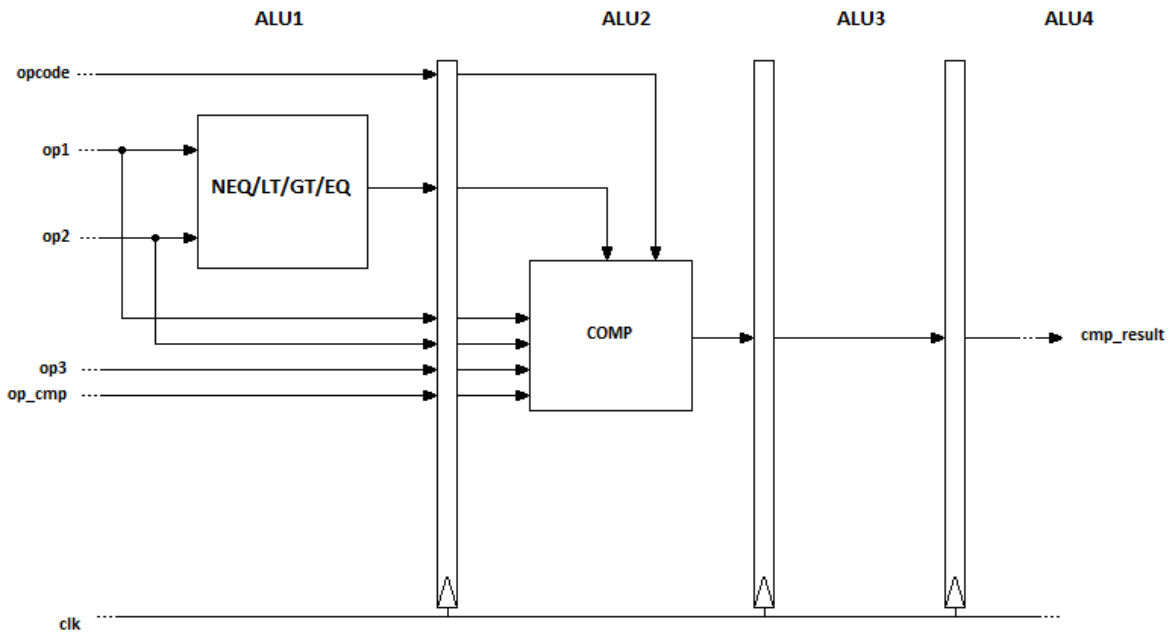

Figura 7.12 Esquema del comparador

En función del resultado obtenido en la primera fase y el *opcode* de la instrucción, se calcula el resultado final, en la etapa ALU2, que hay que propagar 2 ciclos más.

7.6. Implementación

La implementación de la ALU se compone por una parte, el módulo de la FPU, con lo cual solo es necesario conectarle las entradas y salidas de forma adecuada, y por otro lado el comparador que también está implementado en un módulo separado.

Los operandos varían en función de la instrucción, en algunos casos incluso es necesario asignarles un valor constante. Las instrucciones utilizan los siguientes valores como operandos:

Instrucción	OP1	OP2	OP3	OP especial CMP
NOP				
ADD	OP1	OP2		
MAX	OP1	OP2		
MIN	OP1	OP2		
MOV	OP1			
MUL	OP1	OP2		
RCP	1.0f	OP1		
SGE	OP1	OP2		
SLT	OP1	OP2		
CMP	OP1	0.0f	OP2	OP3

Tabla 7.1 Asignación de operandos

La instrucción MAD no se contempla porque en la etapa *decode* se ha dividido en un ADD y un MUL. Por último, es necesario añadir un cable adicional para la instrucción CMP, que selecciona entre op2 y op3, en función de si op1 es menor que 0.

Ambos módulos utilizan como entrada los valores leídos en la etapa *register file*, sin embargo, en función de la instrucción, se deberán utilizar de una forma u otra. Por ejemplo si la operación es de comparación, el resultado generado por la FPU no será válido, así que se escogerá el resultado del comparador después de cuatro ciclos, y viceversa si es una operación aritmética.

La instrucción MOV es un caso especial, donde el valor de op1 se propaga durante 4 ciclos y se asigna como resultado sin necesidad de efectuar ningún cálculo.

7.7. Write back

7.7.1. Descripción

Esta última etapa es la encargada de escribir el resultado en el banco de registros. Hay dos opciones en cuanto al resultado final en función de si la instrucción tiene marcado el bit de *saturate* o no. En el caso de saturar el resultado, hay un circuito encargado de ello, que genera el resultado final dentro del rango [0, 1], sino, simplemente se escribe el resultado de la ALU.

Es muy común que se pueda acceder para escribir y leer del banco de registros en un mismo ciclo, ocupando la primera mitad del ciclo en la escritura, y la segunda en la lectura. En este caso se permite la escritura y lectura simultánea siempre y cuando no sean el mismo registro, ya que en caso contrario el valor leído no corresponderá al último valor actualizado porque se asignará al final del ciclo.

La escritura en el banco de registros, siempre será en TEMP, y solamente se escribirá si la señal de permiso de escritura está activada.

En la siguiente imagen se observa el esquema de esta etapa, aparecen marcados en rojo aquellos cables necesarios.

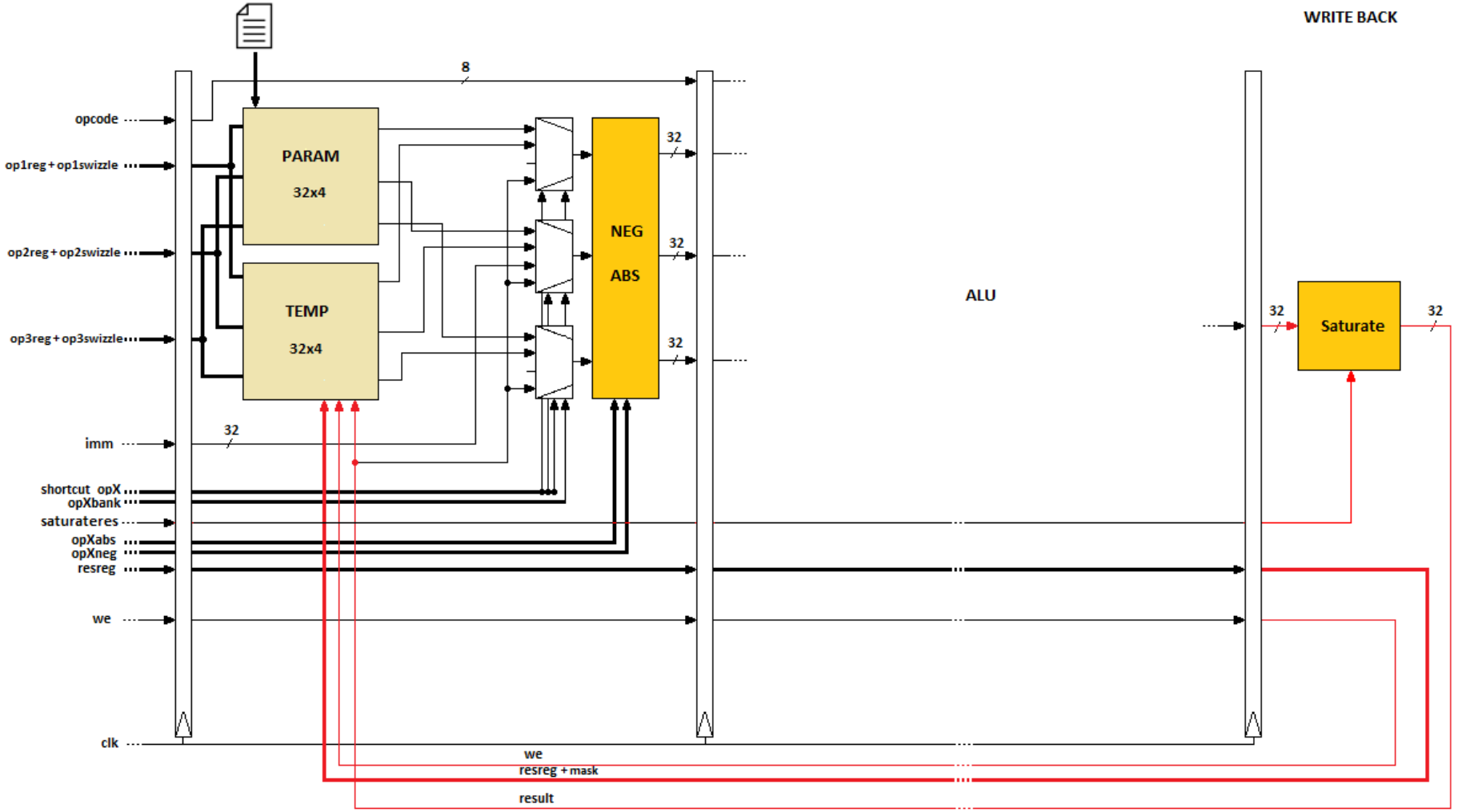

Figura 7.13 Esquema de la etapa *write back*

7.7.2. Implementación

El registro de destino se calcula de la misma forma que para la lectura:

```
resaddr = (resreg << 2) + mask
```

Para escribir un resultado en el banco de registros es necesario el registro de destino, la máscara y el permiso de escritura, que se han ido propagando por todo el *pipeline* porque son necesarios en esta etapa, además de ser imprescindible en la detección de dependencias.

El último paso antes de escribir el resultado en el banco de registros es la función de *saturate*, en función de si la instrucción así lo especificaba. El bit de saturación se ha propagado por todo el *pipeline*, ya que de otra manera sería imposible saber si se tiene que saturar o no.

Saturate

En caso de activarse el bit *saturatedres*, el resultado generado por la ALU se restringe al intervalo [0, 1]. Pueden darse tres casos:

- El resultado es menor que 0.
- El resultado está comprendido en el intervalo [0, 1].
- El resultado es mayor que 1.

En el primer caso solo es necesario comprobar si el bit de signo es igual a 1, en cuyo caso se tratará de un valor negativo, con lo que el resultado final será 0.

El segundo caso es el único en el que no se requiere modificar el resultado, porque el valor ya está dentro del intervalo [0, 1].

En el último caso se comprueba que el bit de signo sea 0, y que el exponente sea mayor o igual que 0x7F, lo que indica que se trata de un valor mayor o igual a 1.0, y el resultado final que se escribirá en el banco de registros será 1.0.

8. Dependencias de datos

Una dependencia de dato es una situación en la que una instrucción del programa depende del resultado de alguna anterior que aún no ha finalizado.

Hay tres tipos de dependencias:

- RAW: *Read After Write*
- WAR: *Write After Read*
- WAW: *Write After Write*

8.1. Dependencias RAW

Se da cuando se necesita leer un dato que aún no se ha calculado. Por ejemplo cuando tenemos dos instrucciones y el resultado de la primera es uno de los operandos de la segunda:

```
I1: add r1 <- r2, r3

I2: add r5 <- r1, r4
```

En este caso hasta que no acabe la primera instrucción, no se deberían leer los datos de la segunda, o utilizará un valor de r1 que no corresponde. Este tipo de dependencia es típico en los procesadores segmentados, y por lo tanto uno de los problemas del procesador de este proyecto.

8.2. Dependencias WAR

En este caso el problema se encuentra en que una instrucción podría escribir un valor en un registro, antes de que otra haya tenido tiempo de leerlo:

```
I1: div r1 <- r2, r3

I2: add r2 <- r4, r5
```

Esta situación puede darse si por ejemplo dos instrucciones pueden ejecutarse a la vez, utilizando "ramas" del procesador diferentes. Es muy típico tener una rama para cálculos aritméticos que necesiten muchos ciclos, y otra para operaciones más simples como comparar.

8.3. Dependencias WAW

Se da cuando dos instrucciones tienen como destino el mismo registro, y como es de esperar, el resultado que deberá tener el registro al acabar la ejecución es el de la última.

```
I1: div r1 <- r2, r3

I2: add r1 <- r4, r5
```

Igual que en el caso de *WAR*, tener múltiples ramas puede acarrear este problema, en el ejemplo si la división tarda 20 ciclos y la suma 4, la segunda instrucción escribirá el resultado en r1 (el resultado correcto), sin embargo el valor final de r1 será el calculado por la división.

8.4. Dependencias de datos del procesador implementado

Una vez vistos los tres tipos de dependencias, y conociendo como está diseñado el *pipeline* del procesador, es fácil ver que tan solo hay que vigilar que no se produzcan dependencias de tipo *RAW*, ya que el *pipeline* es fijo y solo tiene una rama de ejecución para todas las instrucciones.

La eliminación de las dependencias se puede solucionar a la hora de compilar el programa, ya sea incluyendo NOPs entre instrucciones que tengan dependencias, lo que hace bajar el rendimiento, o reordenarlas en la medida de lo posible. Sin embargo esto no siempre es posible y en concreto el programa para ensamblar de ATTILA no lo hace, así que hay que solucionar este problema con hardware. A partir de ahora solo se tratarán dependencias *RAW*.

El primer paso es detectar cuándo hay una dependencia de datos, esto se hace en la etapa de decodificación, mientras se decodifica es posible saber qué registros se utilizarán como operandos. Con esta información y conociendo qué registro tiene como destino cada una de las instrucciones que se está ejecutando en las siguientes etapas del *pipeline*: *register file,* ALU1, ALU2, ALU3, ALU4, excepto con *write back* porque para cuando tenga que leer, el dato ya será el correcto en el banco de registros. Además tenemos que saber si cada en cada una de estas etapas hay una instrucción "válida", es decir, que no hay un dato aleatorio producto de un bloqueo anterior o en caso que el procesador aun no esté completamente inicializado.

Existirá una dependencia de datos cuando uno de los operandos de la instrucción que se está decodificando coincida con el registro de destino de alguna de las etapas que se han listado antes y sea de una instrucción válida.

Cuando se detecta una dependencia se bloqueará el procesador a la espera de que se resuelva. El peor caso será tener una dependencia con la instrucción que se encuentra en la etapa de lectura de registros (*register file*), en cuyo caso habrá que bloquear el procesador durante 5 ciclos, es decir el número de ciclos necesarios hasta que dicha instrucción llegue a la etapa de *write back*, será entonces cuando la instrucción que ha detectado la dependencia pueda continuar. Si la dependencia se tiene con la

instrucción en la etapa ALU1, habrá que bloquear durante 4 ciclos, si es con ALU2, 3 ciclos, y así sucesivamente, esto se ha implementado mediante un contador que cada vez que se detecta una dependencia se carga con el valor correspondiente y se va decrementando en cada ciclo. Los cortocircuitos se activan solo cuando existe dependencia con la etapa ALU4.

El caso descrito es el caso general, sin embargo hay que tener en cuenta que no todas las instrucciones tienen el mismo número de operandos, así pues hay que diferenciar aquellas que tengan solo uno, de las que tienen dos o tres. Saber cuántos operandos tiene se puede hacer de dos formas:

- A partir del *opcode* y una tabla rellenada a priori con el número de operandos de cada una.

- Mirando el campo *opXbank* de 3 bits, que tendrá el valor 0x7 en hexadecimal en caso que el operando en cuestión no exista para esa instrucción.

En la implementación final del procesador se utiliza la 2ª opción para reducir el número de comparaciones.

Un caso especial la instrucción MAD, que tiene 3 operandos, sin embargo no es necesario comprobar que el 3º tenga dependencia porque ese valor depende del resultado la primera operación del MAD (ADD) y por lo tanto siempre tendrá una dependencia directa con la instrucción anterior (MUL).

Por último, hay que tener en cuenta que las dependencias de datos solo pueden darse cuando los operandos son del banco de registros TEMP. Esto es tan simple como comprobar por cada operador que su banco es ese, si no lo es, se tratará de un error o de un dato del banco PARAM, con el que nunca puede haber dependencia puesto que nunca se escribe nada.

El rendimiento del procesador se reduce drásticamente debido a las dependencias de datos. En la siguiente gráfica se observa cómo el IPC (Instrucciones por ciclo) está lejos de ser el ideal, con una media de 0,52.

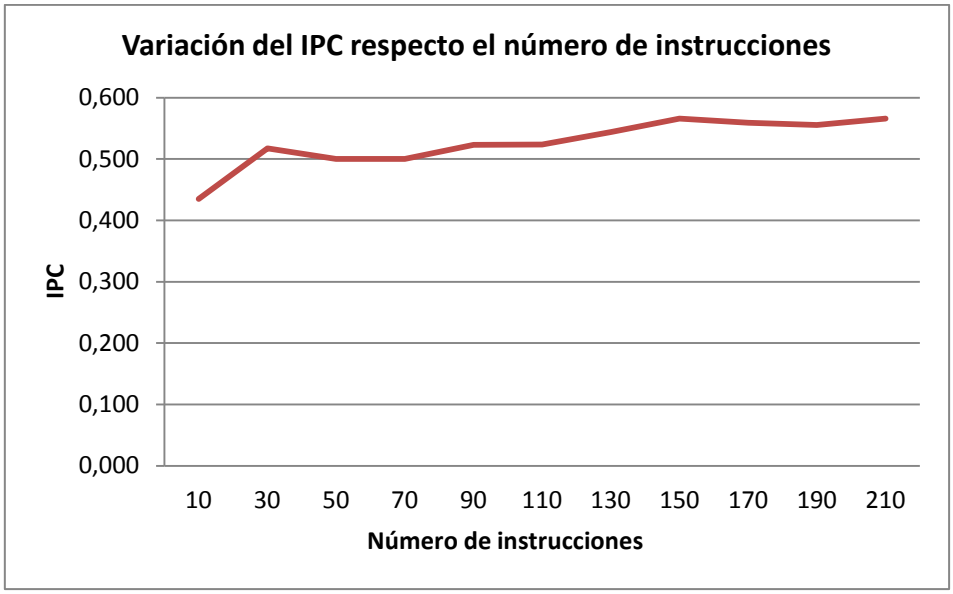

Figura 8.1 Rendimiento del procesador (IPC) en función del número de instrucciones

Este rendimiento es debido exclusivamente a las dependencias de datos. La única mejora posible sin modificar el *pipeline* sería reordenar las instrucciones para reducir al máximo el número de dependencias.

9. Cortocircuitos

Los cortocircuitos son un mecanismo para obtener un dato que ya se encuentra calculado correctamente en algún punto del *pipeline*, pero que aun no se ha escrito en el banco de registros, de manera que se añade *hardware* para proporcionar un camino para que ese dato se pueda utilizar en otra etapa, lo que obliga además a colocar un multiplexor adicional para seleccionar el camino adecuado y una lógica que lo controle.

Como en este procesador no se permite la escritura y lectura de un registro en un mismo ciclo, ocupando la primera mitad del ciclo en escribir, y la segunda en leer, se ha añadido un cortocircuito para obtener el resultado directamente de la etapa *write back*, lo que reduce en 1 ciclo la espera en caso de dependencia de datos.

Supongamos el siguiente programa:

```
mov r1.y, r1.x
slt r9.z, r4.y, r7.z
sge r9.z, r0.x, r1.x
add r6.x, r7.y, r11.z
mul r9.z, r0.x, r1.y
```

La instrucción MUL tiene una dependencia de datos con la instrucción MOV, si no se tuviera un cortocircuito, la ejecución sería la siguiente:

Instrucción	Ciclo													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
mov r1.y, r1.x	F	D	RF	A1	A2	A3	A4	WB						
slt r9.z, r4.y, r7.z		F	D	RF	A1	A2	A3	A4	WB					
sge r9.z, r0.x, r1.x			F	D	RF	A1	A2	A3	A4	WB				
add r6.x, r7.y, r11.z				F	D	RF	A1	A2	A3	A4	WB			
mul r9.z, r0.x, r1.y					F	D	D	D	RF	A1	A2	A3	A4	WB

Figura 9.1 Cronograma de ejecución sin cortocircuitos

Los ciclos 6 y 7 se pierden por culpa del bloqueo provocado por la dependencia de datos. La lectura de los registros se hace en el ciclo 9, porque hasta el 8 no se ha

escrito el valor correcto.

Incorporando un cortocircuito, no es necesario bloquear 2 ciclos, porque en la etapa de lectura se seleccionará el camino para leer el valor que se encuentra en la etapa *write back*. El cronograma que se obtiene se muestra a continuación:

Instrucción	Ciclo													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
mov r1.y, r1.x	F	D	RF	A1	A2	A3	A4	WB						
slt r9.z, r4.y, r7.z		F	D	RF	A1	A2	A3	A4	WB					
sge r9.z, r0.x, r1.x			F	D	RF	A1	A2	A3	A4	WB				
add r6.x, r7.y, r11.z				F	D	RF	A1	A2	A3	A4	WB			
mul r9.z, r0.x, r1.y					F	D	D	RF	A1	A2	A3	A4	WB	

Figura 9.2 Cronograma de ejecución con cortocircuitos

La diferencia es tan solo de un ciclo, sin embargo a medida que el programa aumenta el número de instrucciones, esta reducción puede ser una mejora significativa.

Se han realizado pruebas activando y desactivando los cortocircuitos. Se puede observar que el rendimiento del procesador utilizando cortocircuitos mejora a medida que el programa es más grande.

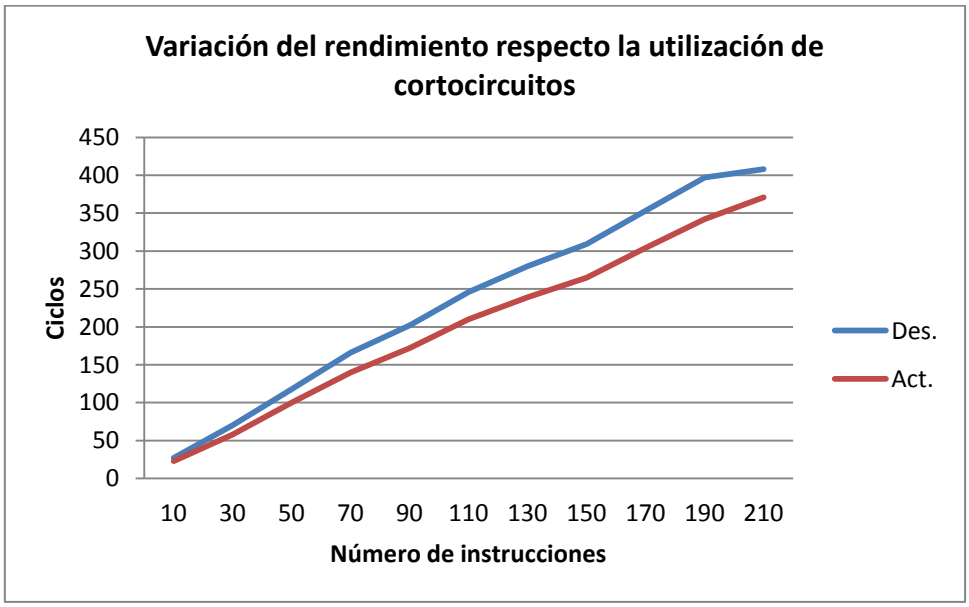

Figura 9.3 Comparativa de rendimiento en función de la utilización de cortocircuitos

10. Instrucciones

La selección del conjunto de instrucciones es una tarea que se ha hecho a la vez que se diseñaba el *pipeline*. *ATTILA* permite ejecutar dos tipos de instrucciones, SIMD4 (*Simple Instruction Multiple Data*) y SOA, la diferencia es básicamente que SOA actúa solo sobre un componente (.x, .y, .z, .w). La versión que se ha implementado para este proyecto es SOA porque es el modelo hacia donde se dirigen las siguientes generaciones de GPU.

Una vez decidido que se implementará únicamente el modo SOA, hay que escoger qué instrucciones se podrán ejecutar con la ALU seleccionada. Las que se han implementado en la versión final se detallan en los siguientes apartados.

Todas las instrucciones están codificadas con 128 bits, en el anexo III se puede encontrar una tabla con los campos que forman cada instrucción. Los valores en rojo no se utilizan.

Antes de pasar a la descripción detallada y la implementación de cada una de las instrucciones que soporta el procesador, en la siguiente lista se describen brevemente cada una de ellas.

A todos los operandos de las instrucciones se les pueden aplicar dos *flags*, uno que niega el valor del operando, y otro que devuelve su valor absoluto. Se pueden aplicar a la vez, calculando primero el valor absoluto y después negando el resultado obtenido.

opcode	format	type	description
00	NOP		No operation
01	ADD dest, source1, source2	arithmetic, float point	Float point 32-bit addition
13	MAD result, source1, source2, source3	arithmetic, float point	Multiply first operand with second operand and add third operand
14	MAX result, source1, source2	arithmetic, float point	Maximum between first operand and second operand
15	Min result, source1, source2	arithmetic, float point	Minimum between first and second operand
16	MOV result, source	movement	Move first operand to result
17	MUL result, source1, source2	arithmetic, float point	Multiplication for 32-bit float point
19	RCP result, source	arithmetic, float point	Reciprocate
1E	SGE result, source1, source2	comparison, float point	Set to 0.0 (false) or 1.0 (true) based on a greater-or-equal-than comparison
22	SLT result, source1, source2	comparison, float point	Set to 0.0f (false) or 1.0 (true) based on a little-than comparison
2D	CMP result, source1, source2, source3	arithmetic, float point	Selects between source2 and source2 based on the value of source1 (<0?)

Tabla 10.1 Descripción de las instrucciones implementadas

10.1. NOP

Esta instrucción no efectúa ningún cambio sobre el estado del procesador (No-Operation). Es suficiente con poner a 0 el permiso de escritura cuando se ejecuta, sin embargo por definición su codificación será todo ceros, excepto en el caso que sea la última instrucción del programa, y entonces tenga el bit *endflag* a 1.

10.2. ADD

Como su nombre indica es la instrucción que ejecuta la suma. Toma como operandos los valores de la etapa *register* file y calcula su suma en coma flotante.

Sintaxis

```
add[_sat] regres[.mask], [-][|]regop1[.swizzle][|], [-][|]regop2[.swizzle][|]

add[_sat] regres[.mask], [-][|]regop1[.swizzle][|], [-][|]imm32[|]
```

Pseudocódigo

```
src1 = readFloatPointScalar(source1)
src2 = readFloatPointScalar(source2)
res = src1 + src2
writeFloatPointResult(res)
```

El esquema del camino de datos se puede observar en la figura 10.1.

10.3. MAD

MAD es la abreviación de *Multiply and Add*. Tiene tres operandos, calcula el producto de los dos primeros y al resultado añade el tercero. Al tratarse de una instrucción con tres operandos no permite que el segundo sea un inmediato codificado en la instrucción. Puede obtener los operandos de cualquiera de los dos bancos o de un cortocircuito.

El hecho de tratarse de una instrucción compuesta obliga a tratarla de forma diferente a la hora de decodificar. Como se ha explicado en el apartado 7.3, dedicado a la decodificación, esta instrucción se decodifica de forma diferente al resto.

Al hacer la separación entre MUL y ADD en la etapa de decodificación es posible ignorar que existe esta instrucción MAD en el resto del *pipeline*, y así poder utilizar directamente la suma y la multiplicación que ya estaban implementadas.

En un primer momento, no se hacía esta separación, si no que se ejecutaba la multiplicación al detectar un MAD en la etapa de *register file*. Mientras se ejecutaba, se enviaban NOPs hasta que acabase, para al final cambiar a una suma utilizando el tercer operando, utilizando una señal auxiliar. Este mecanismo funcionaba pero es demasiado complejo y propenso a errores, la implementación final es mucho más simple y simplifica el control, ya que no hay la necesidad de añadir señales auxiliares.

Sintaxis

```
mad[_sat] resreg[.mask], [-][|]regop1[.swizzle][|],
                         [-][|]regop2[.swizzle][|],
                         [-][|]regop3[.swizzle][|]
```

Pseudocódigo

```
src1 = readFloatPointScalar(source1)
src2 = readFloatPointScalar(source2)
src3 = readFloatPointScalar(source3)
res = src1 * src2 + src3
writeFloatPointResult(res)
```

10.4. MAX

El funcionamiento de esta instrucción, tal como indica su nombre, es escoger como resultado el máximo de sus operandos. Para hacer esto se utiliza el comparador de la ALU, ya que al tratarse de números en coma flotante no se pueden comparar directamente como se haría con enteros.

La ALU dispone de un módulo dedicado exclusivamente a la comparación de números en coma flotante, diseñado específicamente para este proyecto, explicado en el apartado 7.5.3, por lo tanto es suficiente con indicarle los operandos y el *opcode* para que calcule el máximo.

Sintaxis

```
max[_sat] resreg[.mask], [-][|]regop1[.swizzle][|], [-][|]regop2[.swizzle][|]

max[_sat] resreg[.mask], [-][|]regop1[.swizzle][|], [-][|]imm32[|]
```

Pseudocódigo

```
src1 = readFloatPointScalar(source1)
src2 = readFloatPointScalar(source2)
res = (src1 > src2) ? src1 : src2
writeFloatPointResult(res)
```

El esquema del camino de datos se puede observar en la figura 10.4.

10.5. MIN

El comportamiento de esta instrucción es el opuesto a la instrucción MAX. El resultado es el menor valor de los dos operandos de entrada. Utiliza el comparador de la ALU para hacer el cálculo.

Sintaxis

```
min[_sat] resreg[.mask], [-][|]regop1[.swizzle][|], [-][|]regop2[.swizzle][|]

min[_sat] resreg[.mask], [-][|]regop1[.swizzle][|], [-][|]imm32[|]
```

Pseudocódigo

```
src1 = readFloatPointScalar(source1)

src2 = readFloatPointScalar(source2)

res = (src1 < src2) ? src1 : src2

writeFloatPointResult(res)
```

El esquema del camino de datos se puede observar en la figura 10.4.

10.6. MOV

Se utiliza para mover el valor de un registro a otro. Se han considerado dos opciones a la hora de implementarla, la primera es utilizar la ALU con la operación suma y 0.0 como segundo operador, aprovechando que la ALU hace un cálculo cada ciclo, esto evitaría añadir *hardware* adicional pero tiene el problema de tratar con una excepción en caso de darse. La segunda opción, que es la que se ha implementado, añade cuatro registros para propagar el dato que se está moviendo.

La segunda opción tiene la ventaja de que es mucho más simple de implementar, y en versiones futuras del procesador sería más sencillo reducir la latencia de esta operación. El único inconveniente es el *hardware* adicional necesario.

Sintaxis

```
mov[_sat] resreg[.mask], [-][|]regop1[.swizzle][|]
```

Pseudocódigo

```
src1 = readFloatPointScalar(source1)
res = src1
writeFloatPointResult(res)
```

El esquema del camino de datos se puede observar en la figura 10.1.

10.7. MUL

Es la instrucción que calcula la multiplicación, con un funcionamiento igual que el de la suma. Tarda 4 ciclos en ejecutarse, esto es porque para hacer este cálculo en coma flotante, hay que multiplicar las mantisas, y *Verilog* permite hacer una multiplicación entera utilizando el operador '*', con lo cual se puede hacer en un ciclo, sin embargo esto solo sirve en el caso que solo se quiera simular el circuito, ya que la

implementación real es iterativa y por lo tanto depende de la longitud de los operandos.

Sintaxis

```
mul[_sat] resreg[.mask], [-][|]regop1[.swizzle][|], [-][|]regop2[.swizzle][|]

mul[_sat] resreg[.mask], [-][|]regop1[.swizzle][|], [-][|]imm32[|]
```

Pseudocódigo

```
src1 = readFloatPointScalar(source1)
src2 = readFloatPointScalar(source1)
res = src1 * src2
writeFloatPointResult(res)
```

El esquema del camino de datos se puede observar en la figura 10.1.

10.8. RCP

Esta instrucción se conoce como *reciprocal* y calcula la división entre 1.0 y el operando leído en la etapa *register file*.

Igual que en el caso de la suma y la multiplicación, es suficiente con utilizar la ALU para dividir. Como el funcionamiento de la división está pensado para el caso general, hay que pasarle como primer operando un 1.0 en el formato IEEE 754, y como segundo, el valor leído del banco de registros.

Como en la multiplicación, 4 ciclos para el cálculo de una división en coma flotante no es real, y habría que implementar una división iterativa.

Sintaxis

```
rcp[_sat] resreg[.mask], [-][|]regop1[.swizzle][|]
```

Pseudocódigo

```
src1 = readFloatPointScalar(source1)
res = 1.0f / src1
writeFloatPointResult(res)
```

El esquema del camino de datos se puede observar en la figura 10.3.

10.9. SGE

Es una instrucción de comparación compara dos valores y da como resultado 0.0 o 1.0 según si se satisface la condición 'mayor o igual' (>).

Para hacer esto, igual que en el caso de *max* y *min*, se utiliza el comparador de la ALU, que ya genera directamente el resultado correcto.

Sintaxis

```
sge[_sat] resreg[.mask], [-][|]regop1[.swizzle][|], [-][|]regop2[.swizzle][|]

sge[_sat] resreg[.mask], [-][|]regop1[.swizzle][|], [-][|]imm32[|]
```

Pseudocódigo

```
src1 = readFloatPointScalar(source1)
src2 = readFloatPointScalar(source2)
res = (src1 >= src2) ? 0.0f : 1.0f
writeFloatPointResult(res)
```

El esquema del camino de datos se puede observar en la figura 10.4.

10.10. SLT

Es el caso análogo a la instrucción *SGE* pero con la comparación contraria. Al igual que todas las instrucciones de comparación, utiliza el comparador de la ALU. Después de cuatro ciclos se da como resultado 0.0 o 1.0 según si se satisface o no la condición 'menor que' (<).

Sintaxis

```
slt[_sat] resreg[.mask], [-][|]regop1[.swizzle][|], [-][|]regop2[.swizzle][|]

slt[_sat] resreg[.mask], [-][|]regop1[.swizzle][|], [-][|]imm32[|]
```

Pseudocódigo

```
src1 = readFloatPointScalar(source1)
src2 = readFloatPointScalar(source2)
res = (src1 < src2) ? 1.0f : 0.0f
writeFloatPointResult(res)
```

El esquema del camino de datos se puede observar en la figura 10.4.

10.11. CMP

El cálculo que realiza esta instrucción es diferente al que se podría pensar intuitivamente. Compara el primer operando con 0.0, y en función de si se satisface la condición 'menor que' (<), se asigna como resultado el valor del operando 2 o 3.

Para hacer la comparación se sigue el mismo procedimiento que en las instrucciones *max*, *min*, *sge* y *slt*. Como el resultado final es directamente src2 o src3, hay que

propagarlos mientras se hace la comparación, para al final poder escribirlo en el banco de registros.

Sintaxis

```
Cmp[_sat] resreg[.mask], [-][|]regop1[.swizzle][|],
                       , [-][|]regop2[.swizzle][|],
                       , [-][|]regop3[.swizzle][|]
```

Pseudocódigo

```
src1 = readFloatPointScalar(source1)

src2 = readFloatPointScalar(source2)

src3 = readFloatPointScalar(source3)

res = (src1 < 0.0f) ? src2 : src3;

writeFloatPointResult(res)
```

El esquema del camino de datos se puede observar en la figura 10.5.

10.12 Caminos de datos

10.12.1 Camino de datos: ADD/MUL

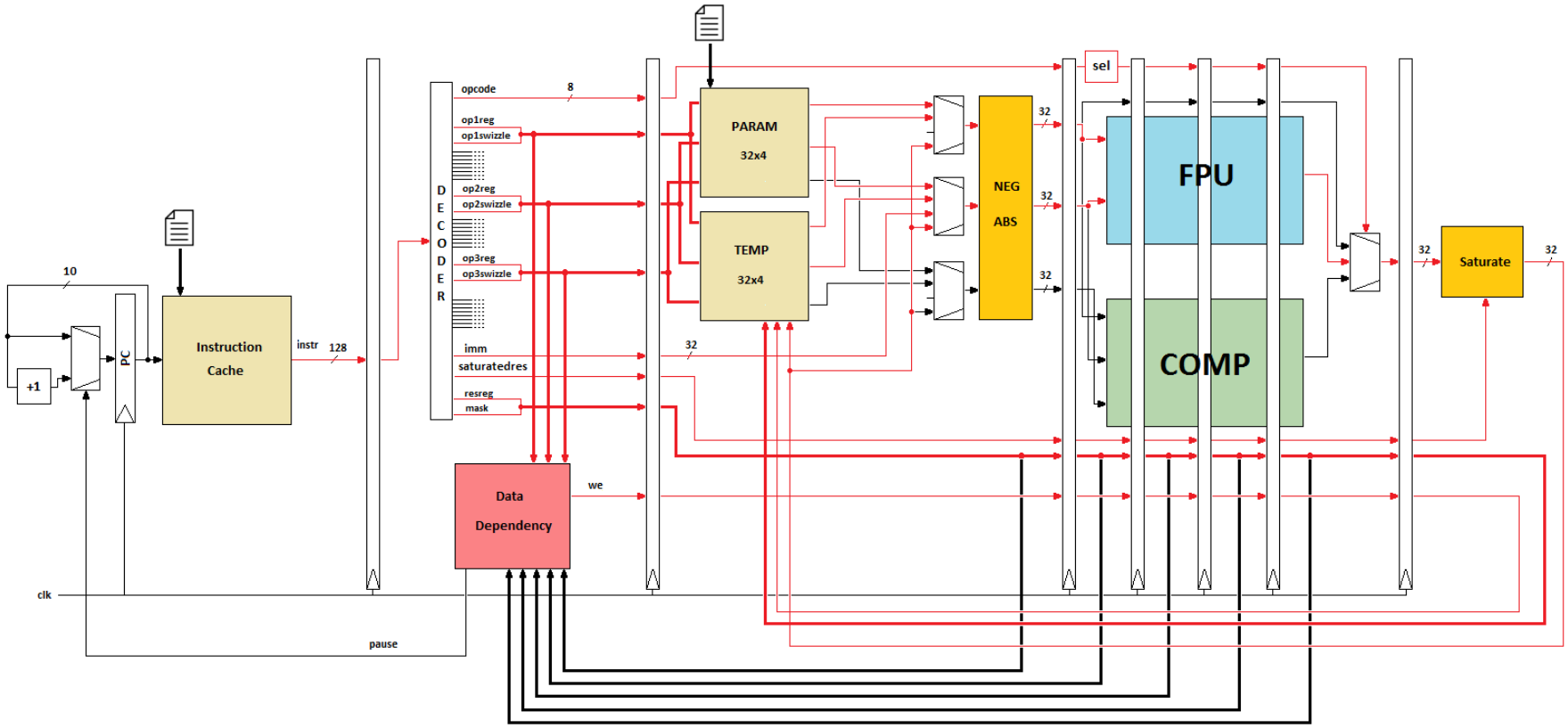

Figura 10.1 Camino de datos para las instrucciones ADD y MUL

10.12.2 Camino de datos MOV

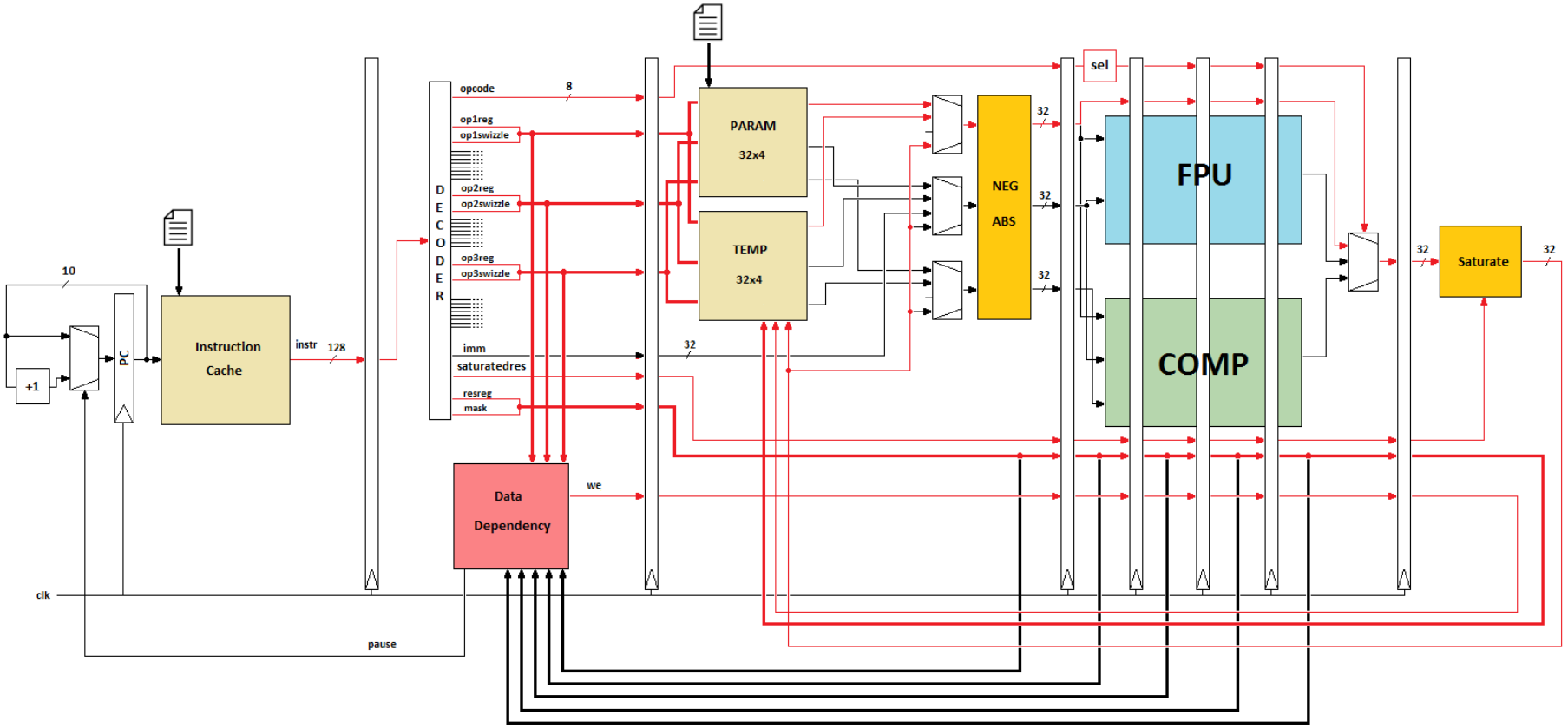

Figura 10.2 Camino de datos para la instrucción MOV

10.12.3 Camino de datos RCP

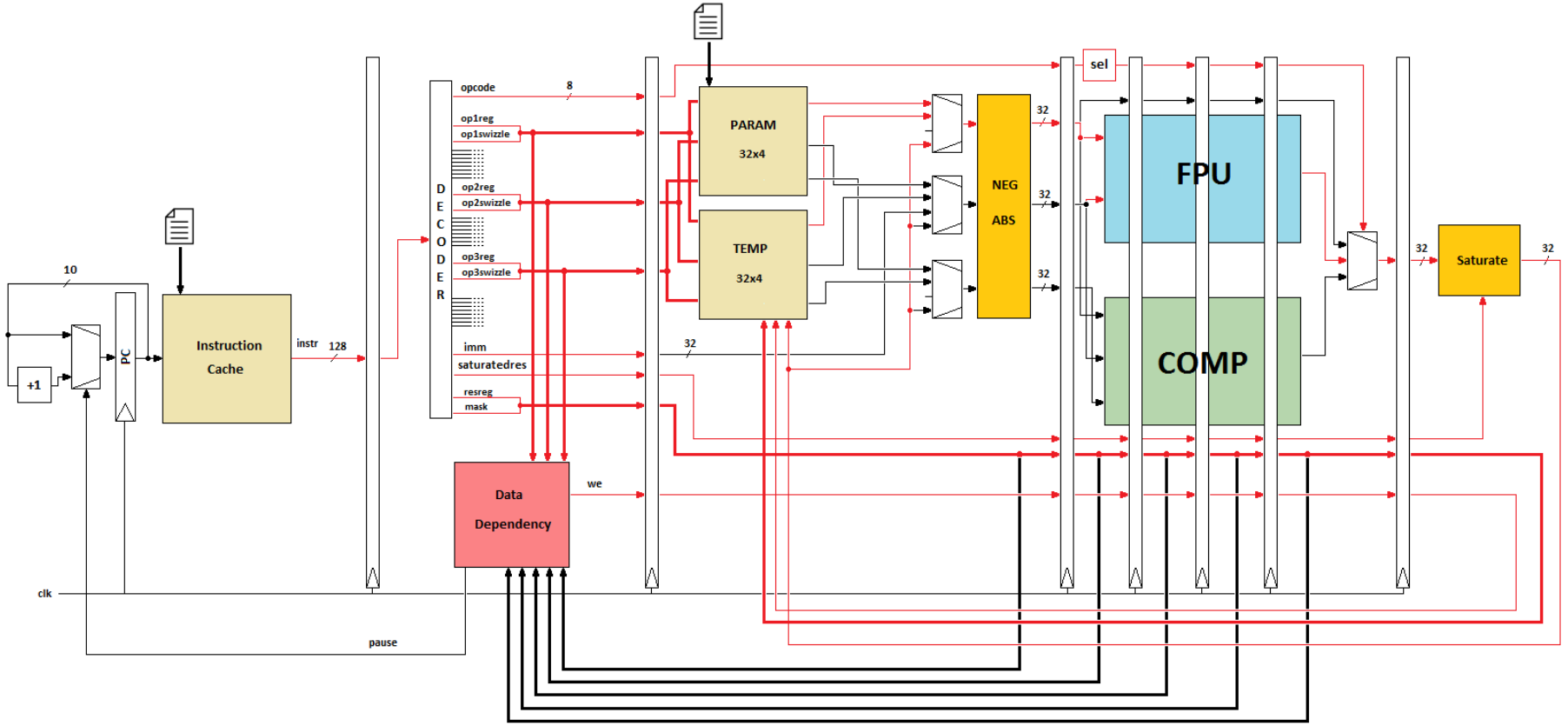

Figura 10.3 Camino de datos para la instrucción RCP

10.12.4 Camino de datos MAX/MIN/SGE/SLT

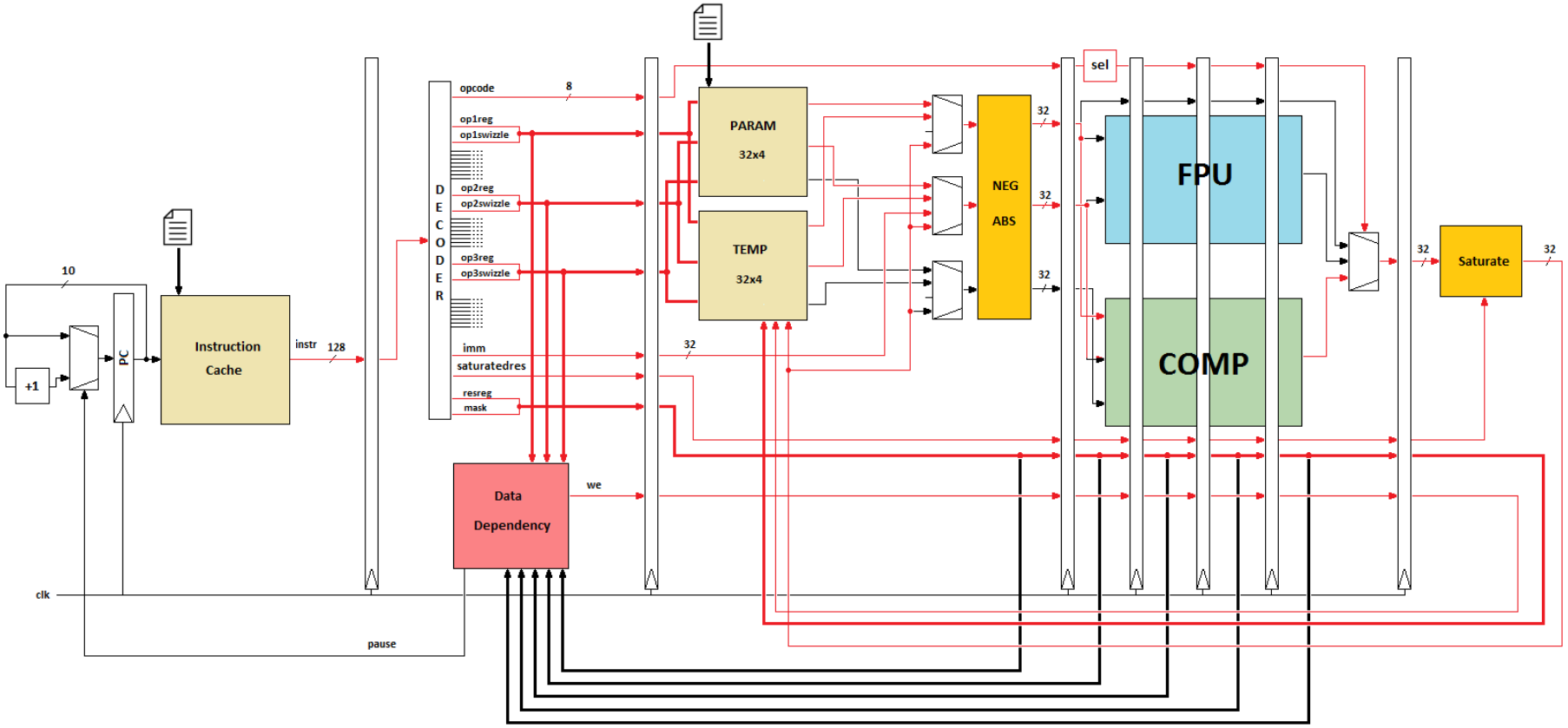

Figura 10.4 Camino de datos para las instrucciones MAX, MIN, SGE y SLT

10.12.5 Camino de datos CMP

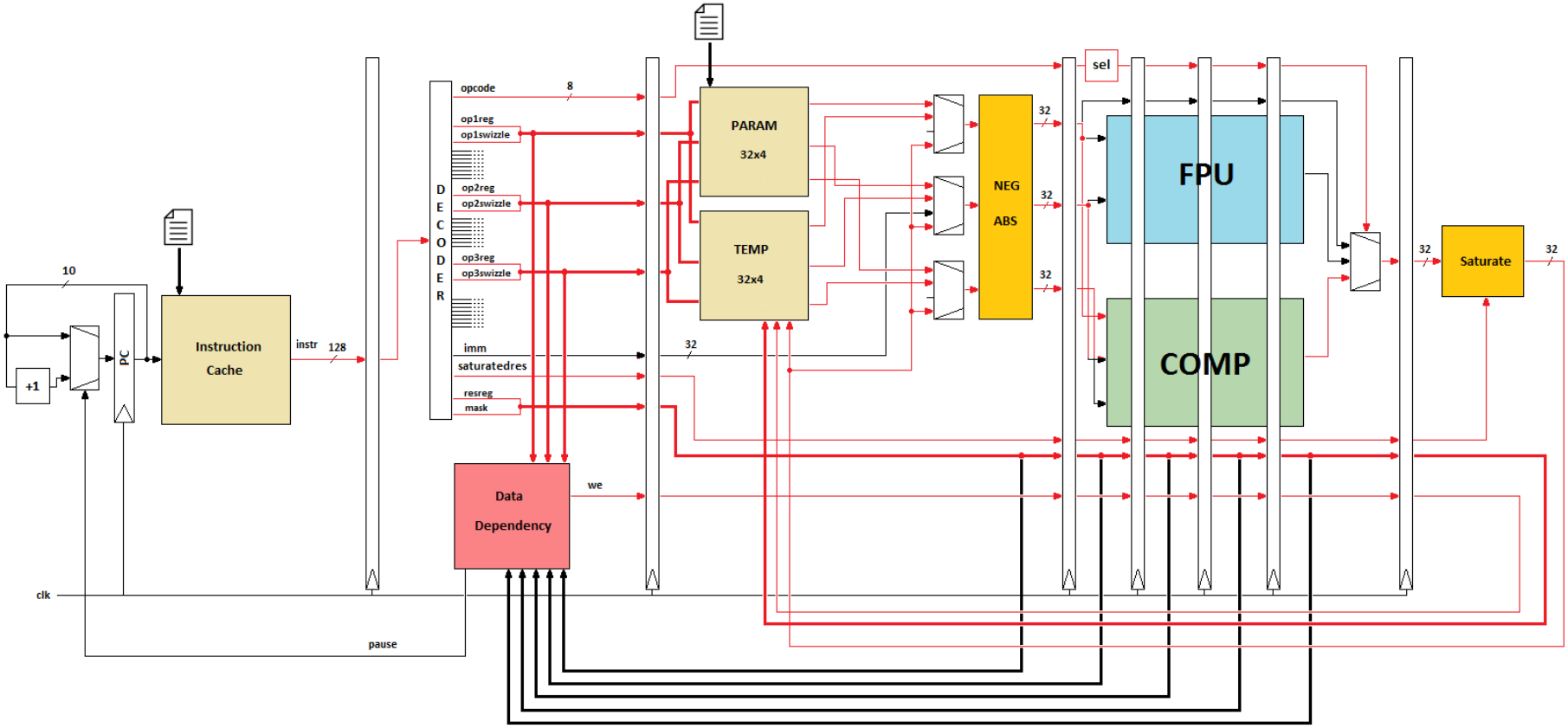

Figura 10.5 Camino de datos para la instrucción CMP

11 Validación del modelo

11.1 Descripción de las pruebas

Una vez el procesador está implementado, hay que comprobar que el funcionamiento es correcto. Este trabajo sería imposible hacerlo a mano, aunque se haya ido utilizando el simulador a medida que se iban implementando funcionalidades, son necesarios otros mecanismos con los que saber con más certeza que el modelo es válido, además de agilizar las comprobaciones.

Como se dispone de la versión del *shader* ya implementada en ATTILA, se puede utilizar para comprobar los resultados, sin embargo habrá que modificar el código para adaptarlo a las necesidades de este proyecto.

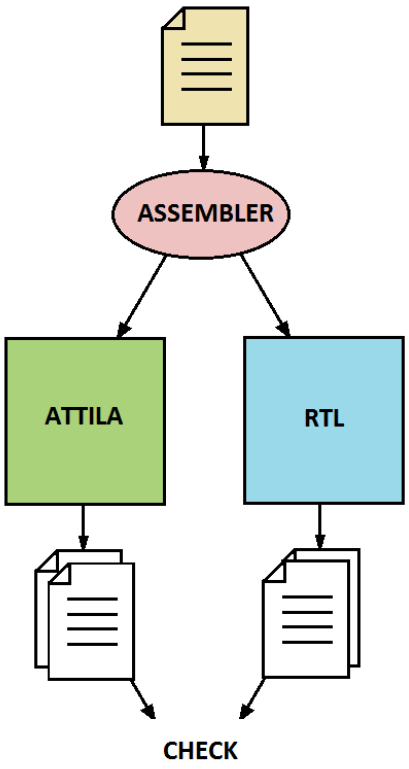

Figura 11.1 Esquema de ejecución de las pruebas

El procedimiento que se sigue para comprobar que un programa concreto se ha ejecutado correctamente se basa en la comparación de los resultados de cada instrucción. Tanto ATTILA como el procesador escriben en un fichero el resultado de cada instrucción después de ejecutarla, a medida que se ejecuta el programa. El fichero que resulta en cada caso es una lista de valores, uno por cada instrucción del programa, con el resultado de cada instrucción, con lo que se puede saber con certeza que el programa se ha ejecutado correctamente, o en caso contrario detectar diferencias, y en qué instrucción concreta se ha producido un fallo.

Esta técnica es muy útil, además de comprobar si un programa se ejecuta correctamente para saber qué instrucción no ha generado el resultado esperado, ayudando a localizar el error durante el desarrollo del proyecto.

Adicionalmente se compara el resultado final del banco de registros, lo que confirma que no solo los resultados son correctos, si no que se han escrito en los registros que se indicaba.

En un primer momento se utilizaron *shaders* reales de juegos para comprobar el funcionamiento del procesador, extraídos utilizando ATTILA. Aunque todos han resultado correctos, debido a la poca variedad de instrucciones, se ha desarrollado otro sistema para realizar pruebas que consiste en la generación de *shaders* aleatorios.

Otro de los objetivos de las pruebas, es garantizar que se han probado todas las combinaciones de operandos en todas las instrucciones, de forma que se tenga la seguridad de que se han cubierto todos los casos.

La idea es incluir en el código en V*erilog* una serie de contadores que para cada *shader* ejecutado cuenten, para cada instrucción:

- El valor de cada operando con el que se ha ejecutado: positivo, negativo, 0, -0, +INF, -INF, NaN.
- El origen de cada operando: banco TEMP, banco PARAM, *immediate*, cortocircuito.

Para cada ejecución, se genera un fichero con estos datos. Una vez se han ejecutado suficientes programas, se acumula el resultado de todas las pruebas y se calcula cuantas veces se ha ejecutado cada instrucción con una combinación de operandos de los tres tipos anteriores.

Las pruebas se han realizado utilizando un *script* que ejecuta programas aleatorios. Cada programa tiene un tamaño de 100 instrucciones, longitud que se ha considerado adecuada para realizar un número tan elevado de pruebas. Los resultados obtenidos para cada uno de los tres tipos de pruebas que se han considerado se detallan en los apartados siguientes. Se han probado alrededor de 30.000 *shaders* que han servido para validar el funcionamiento.

En el siguiente apartado se describen los resultados de las pruebas realizadas.

11.2 Resultados de las pruebas

11.2.1 Pruebas según el valor de los operandos

Este conjunto de pruebas ha consistido en contar, para cada instrucción de las implementadas, con qué operandos se ha ejecutado. Los casos considerados son los que soporta el estándar IEEE 754:

- Positivos
- Negativos
- +/-0
- +/-INF
- NaN

Para cada combinación de op1, u op1-op2, según la instrucción, se representa en las gráficas siguientes el número de veces que se ha dado. Esto demuestra que se han ejecutado todas las combinaciones posibles de forma correcta.

La escala del eje Y es logarítmica (base 10), para apreciar con más claridad el número de combinaciones ejecutadas en cada caso.

Resultados para las instrucciones MOV y RCP

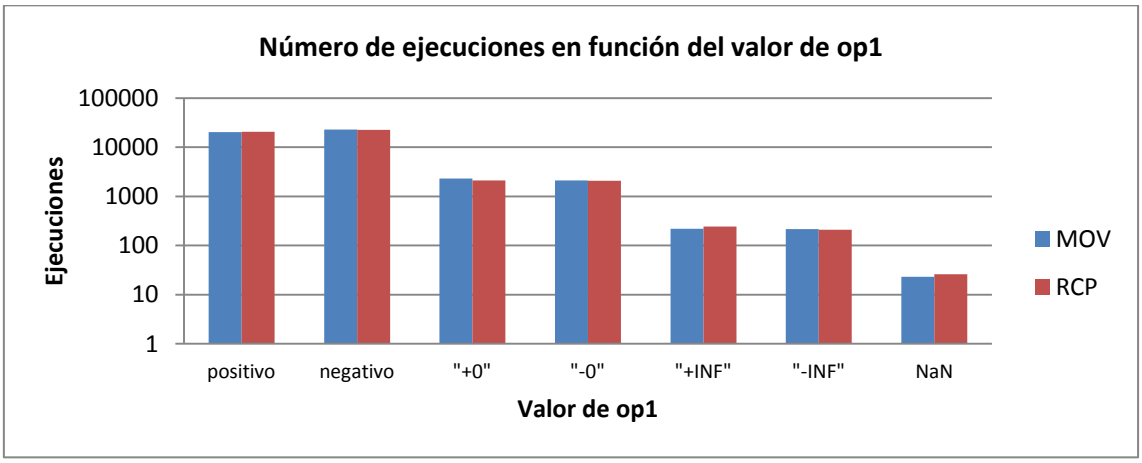

Figura 11.2 Ejecuciones de las instrucciones MOV y RCP con cada tipo de operando

Resultados para las instrucciones ADD, MAX, MIN, SGE y SLT

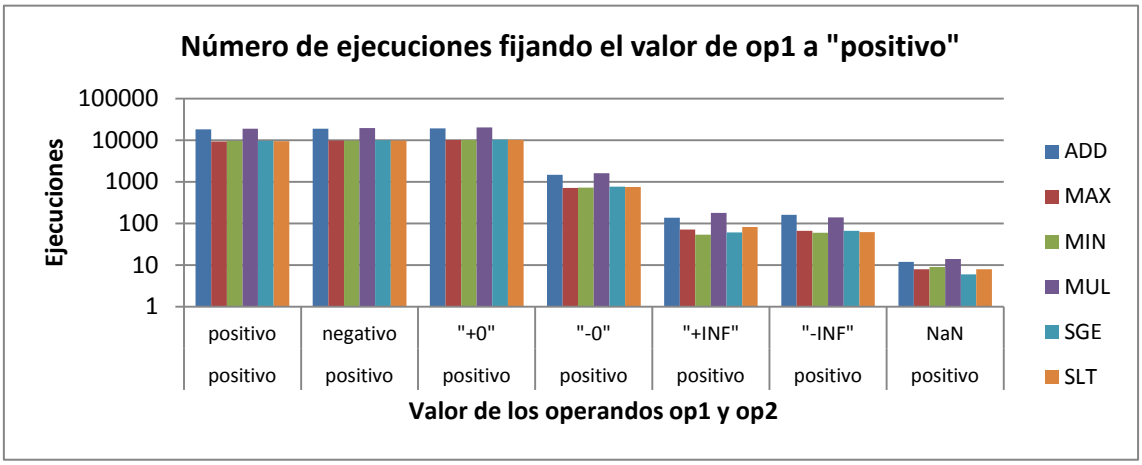

Figura 11.3 Ejecuciones para las combinaciones con op1 positivo

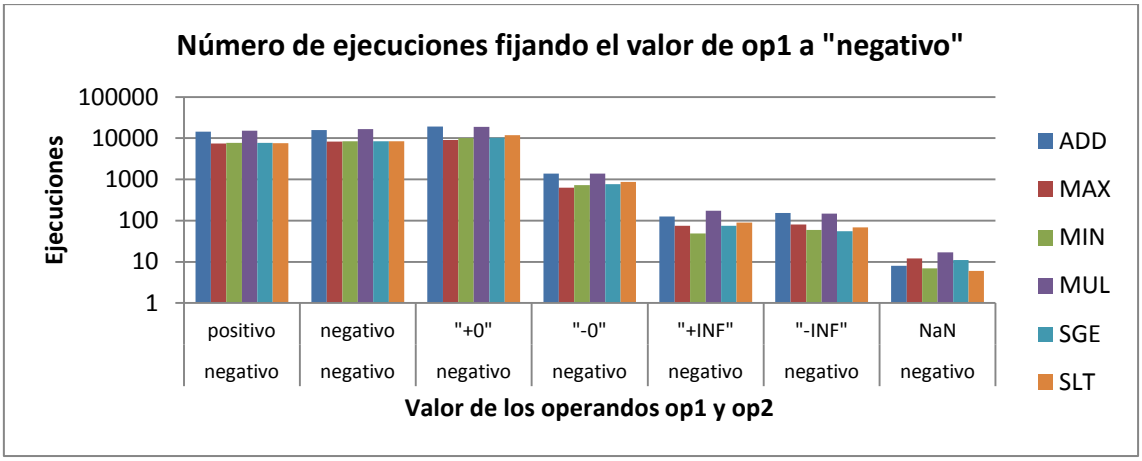

Figura 11.4 Ejecuciones para las combinaciones con op1 +0

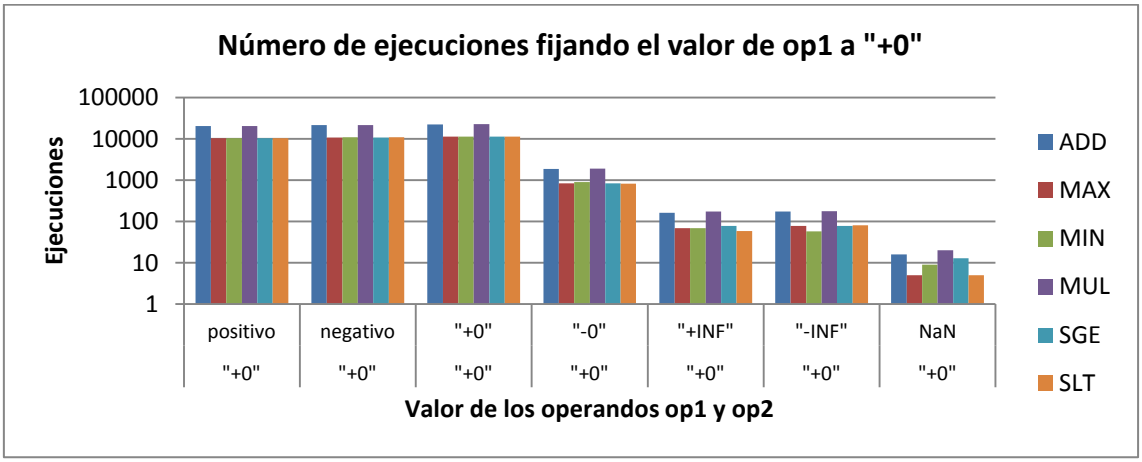

Figura 11.5 Ejecuciones para las combinaciones con op1 negativo

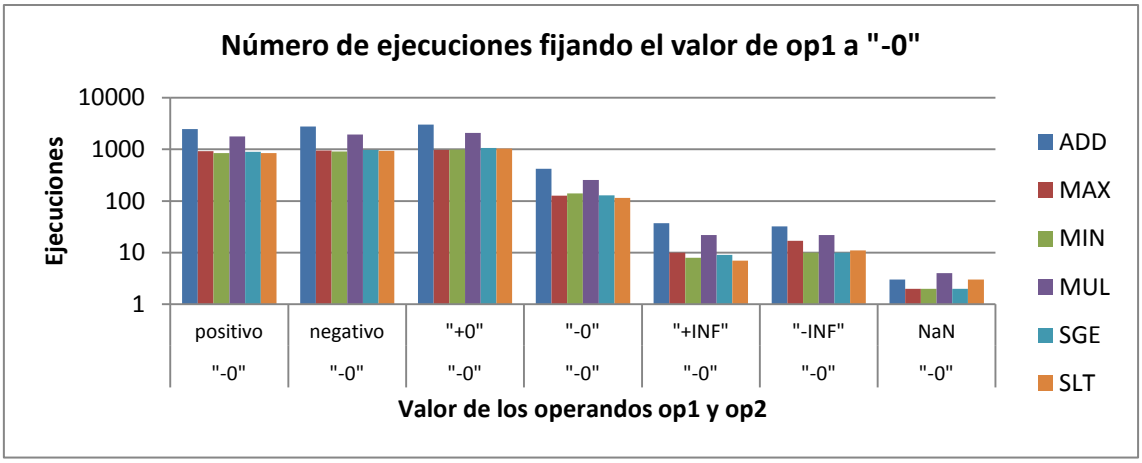

Figura 11.6 Ejecuciones para las combinaciones con op1 -0

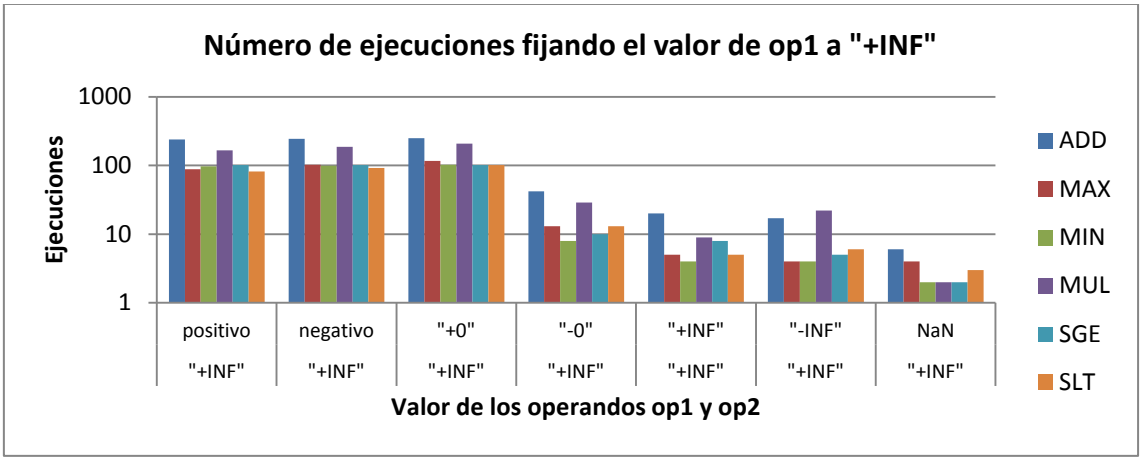

Figura 11.7 Ejecuciones para las combinaciones con op1 +INF

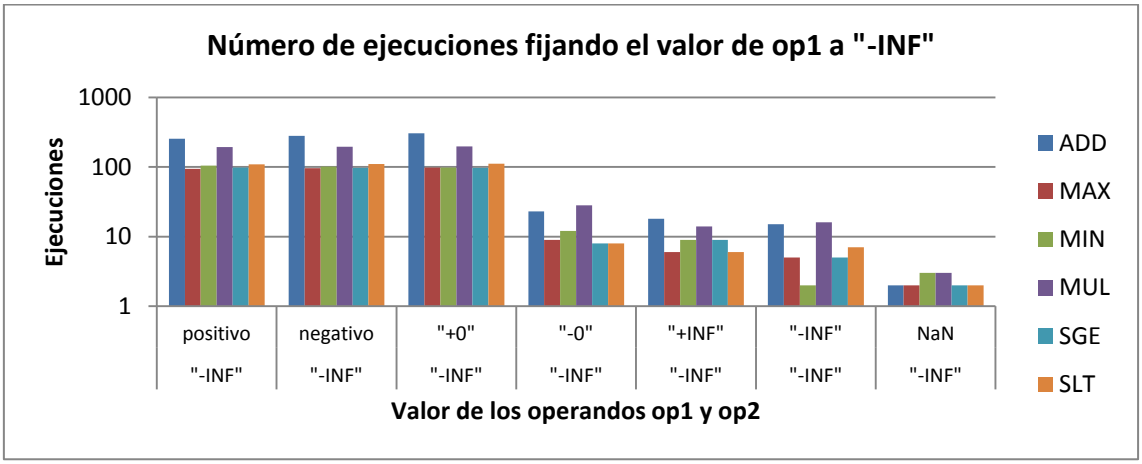

Figura 11.8 Ejecuciones para las combinaciones con op1 -INF

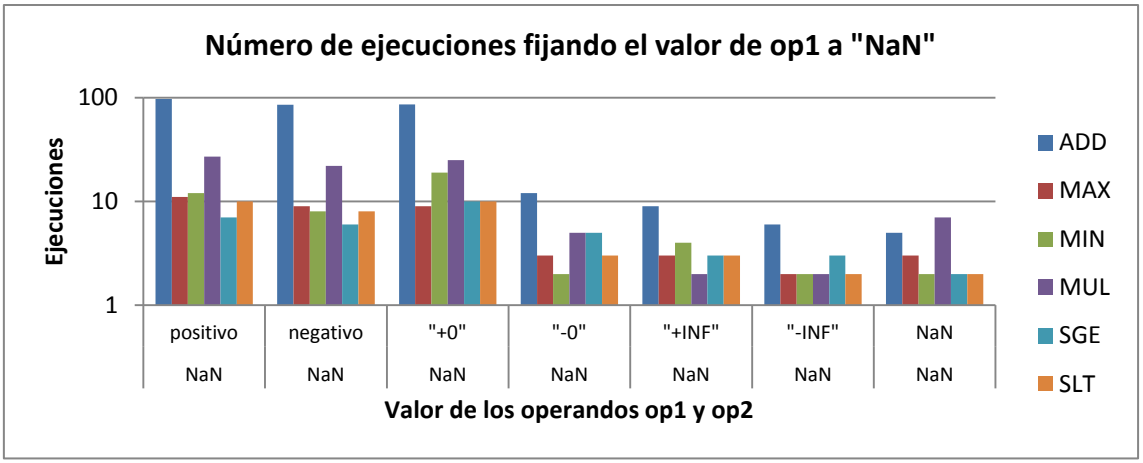

Figura 11.9 Ejecuciones para las combinaciones con op1 NaN

La instrucción CMP se incluye dentro de los casos de la instrucción SLT, dado que la comparación es la misma para el caso en que el op2 sea “+0”.

Como se puede observar los casos que más se dan son aquellos en los que intervienen operandos positivos o negativos, aun así es posible que con ciertas combinaciones de instrucciones se acaben produciendo valores especiales como -0, +/-INF o NaN para los cuales todas las instrucciones han generado los resultados correctos.

11.2.2 Pruebas según el origen de los operandos

Este segundo conjunto de pruebas ha consistido en comprobar que cada instrucción ha obtenido sus operandos de todos los caminos de datos posibles:

- Banco de registros TEMP.
- Banco de registros PARAM.
- Inmediato codificado en la instrucción.
- Cortocircuito.

Esta prueba sirve además para confirmar que solo las instrucciones que tienen dos operandos utilizan el camino del inmediato.

El eje Y muestra el porcentaje de instrucciones que se han ejecutado con cada combinación de caminos de datos.

Resultados para las instrucciones MOV y RCP

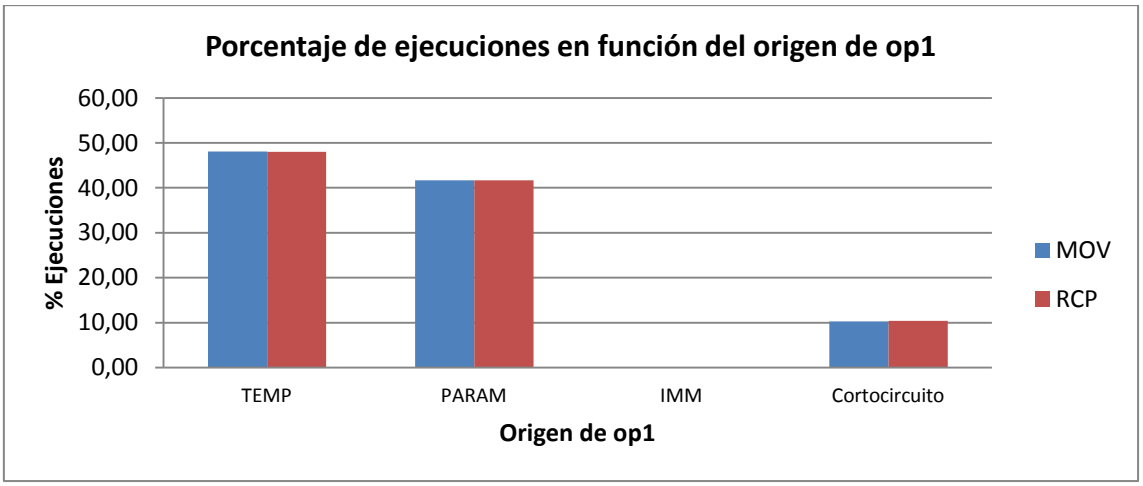

Figura 11.10 Ejecuciones para las instrucciones MOV y RCP

Al tratarse de instrucciones con solo un operando, nunca se utiliza el inmediato codificado en la instrucción.

Resultados para las instrucciones ADD, MAX, MIN, MUL, SGE y SLT

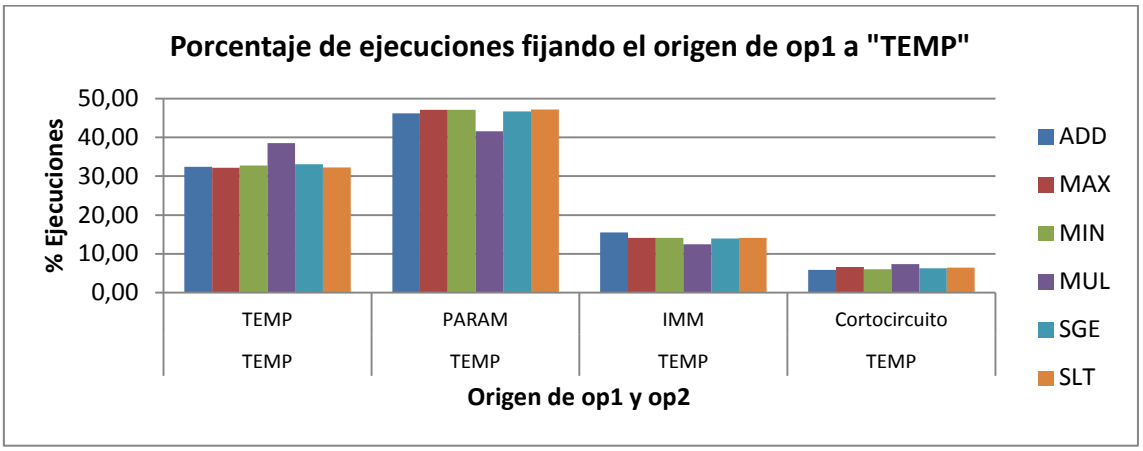

Figura 11.11 Ejecuciones para las combinaciones donde op1 proviene de TEMP

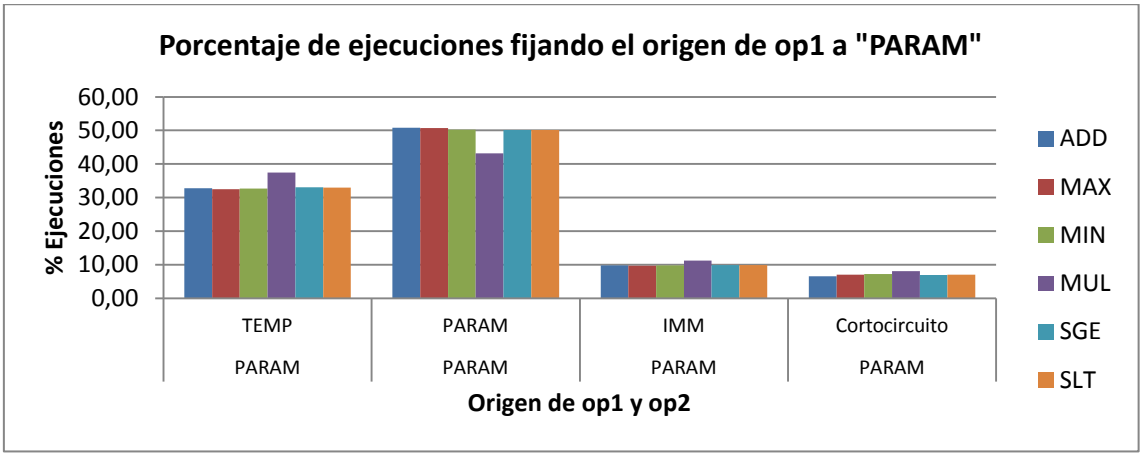

Figura 11.12 Ejecuciones para las combinaciones donde op1 proviene de PARAM

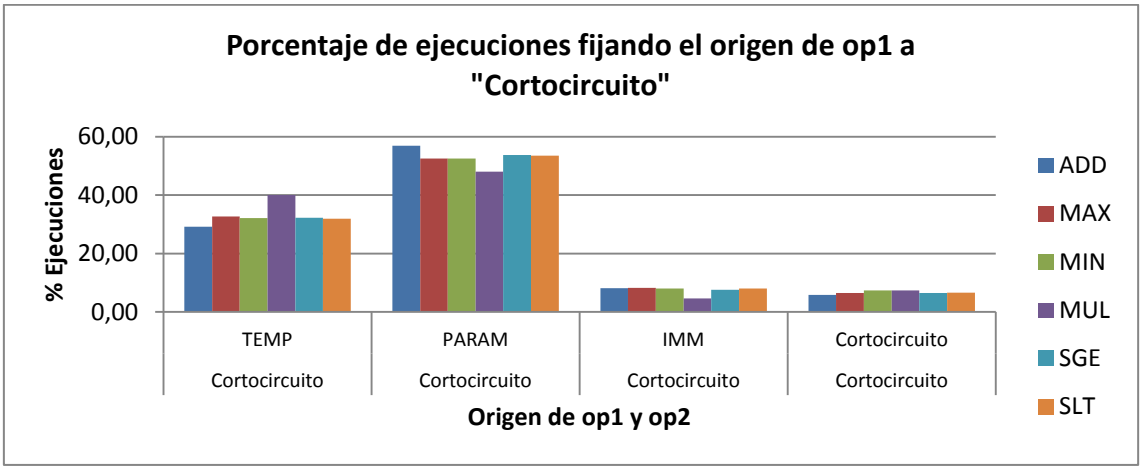

Figura 11.13 Ejecuciones para las combinaciones donde op1 proviene del cortocircuito

Resultados para la instrucción CMP

Para esta instrucción no se ha dado ningún caso de utilización del inmediato codificado en la instrucción. Se muestran solo el resto de combinaciones.

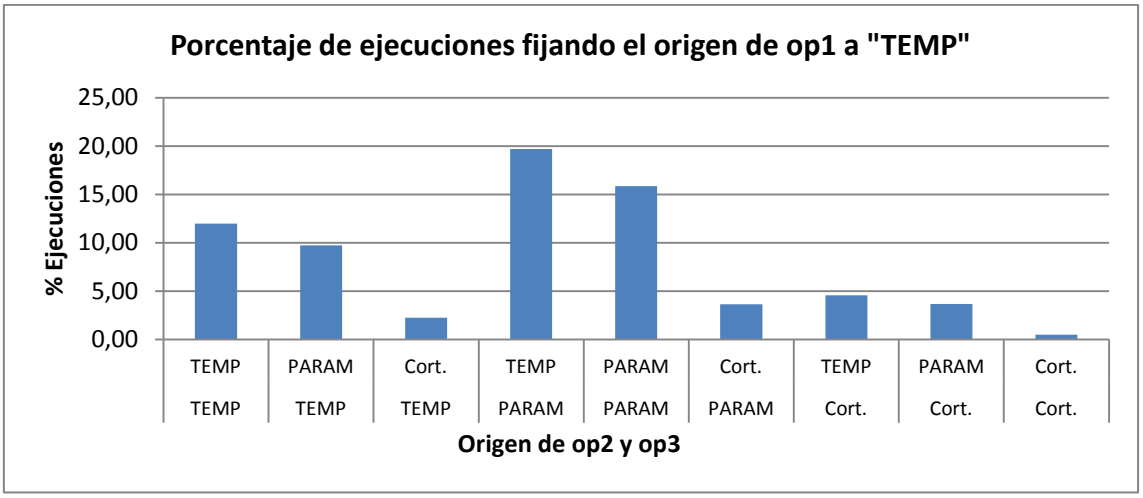

Figura 11.14 Ejecuciones para las combinaciones donde op1 proviene de TEMP

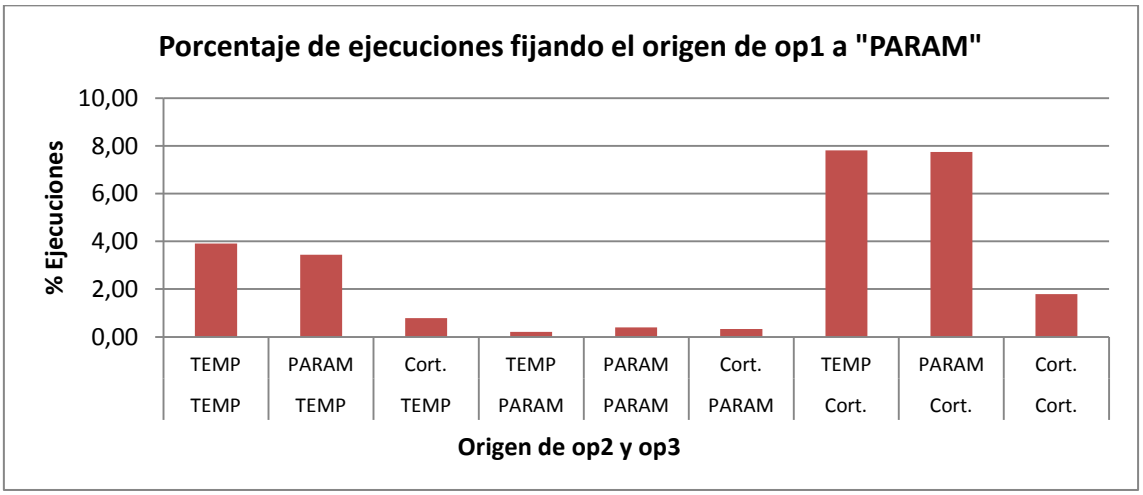

Figura 11.15 Ejecuciones para las combinaciones donde op1 proviene de PARAM

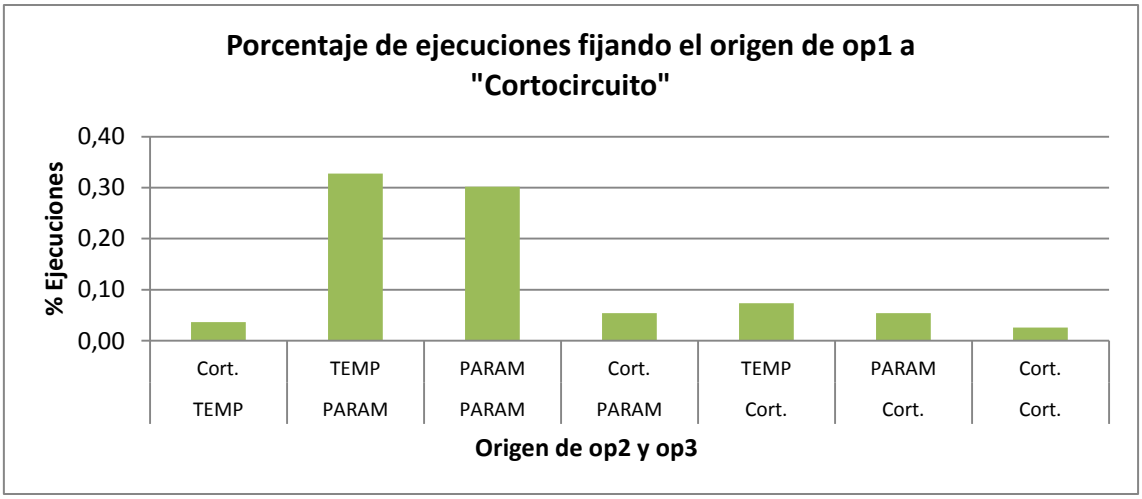

Figura 11.16 Ejecuciones para las combinaciones donde op1 proviene del cortocircuito

La similitud entre todas las gráficas es debida a que las pruebas se han realizado con programas generados de forma aleatoria, y para todas las instrucciones, la probabilidad de utilizar los mismos datos es la misma.

Como conclusión de las pruebas realizadas, se ha confirmado que el procesador diseñado utiliza todos los tipos de valores para realizar cálculos, además de obtenerlos de todos los caminos posibles, para cada operación. Esto, añadido a que los *tests* resultan correctos tras la comparación con el resultado de ATTILA, demuestra que el procesador tiene un funcionamiento correcto en todos los casos.

12 Planificación del proyecto

Para el desarrollo del proyecto se han seguido los siguientes pasos recogidos en el diagrama:

- Definición del proyecto.
- Estudio del *pipeline* gráfico.
- Estudio de la arquitectura de ATTILA.
- Estudio de los lenguajes de descripción de hardware.
- Estudio de las herramientas de simulación.
- Diseño del *pipeline*.
- Búsqueda de una unidad de cálculo en coma flotante (FPU).
- Definición de las instrucciones.
- Implementación del *pipeline*.
- Modificación del software de ATTILA (*ShaderTest*).
- Diseño de las pruebas para la validación del modelo.
- Validación del modelo.
- Evaluación de los resultados obtenidos.
- Documentación.

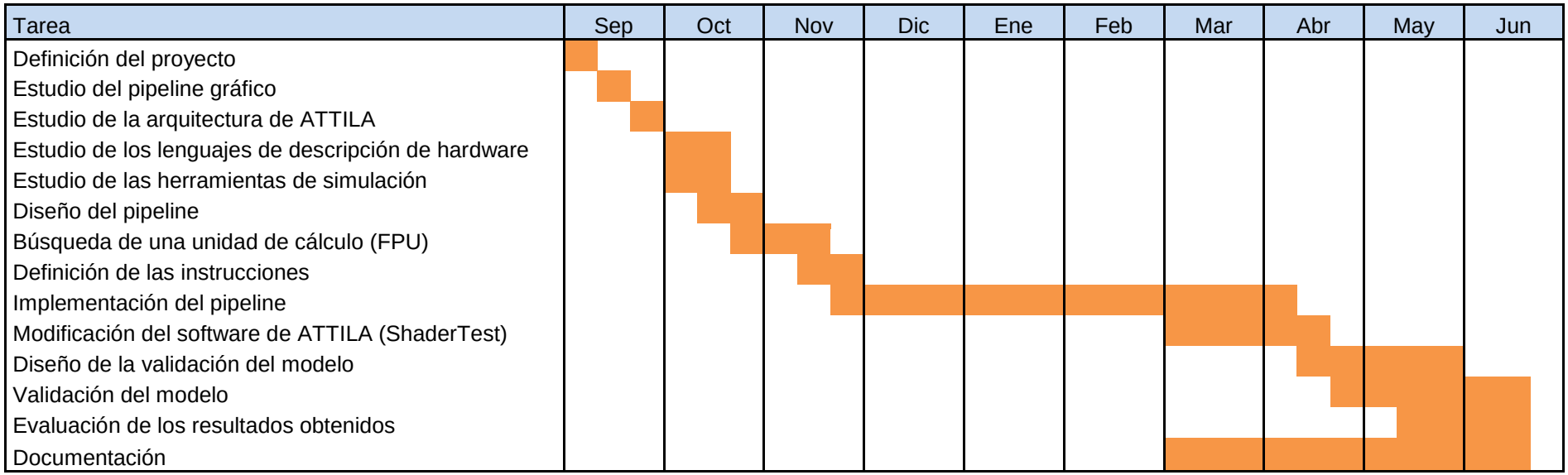

Figura 12.1 Diagrama de Gantt

13 Análisis de viabilidad económica

En este punto se realiza un cálculo aproximado del coste económico del proyecto, teniendo en cuenta las horas dedicadas por trabajador. Se estima que las horas dedicadas por cada trabajador son:

- Arquitecto: 400h
- Diseño del hardware: 350h
- Grupo de pruebas: 180h

Se han tenido en cuenta los siguientes costes por hora para arquitecto, diseño del hardware y grupo de pruebas:

- Arquitecto: 40 € / hora
- Diseño del hardware: 30 € / hora
- Grupo de pruebas: 25 € / hora

El coste por cada trabajador se calcula en función de las horas dedicadas por cada uno al proyecto:

- Coste arquitecto = 40 euros / hora * 400 horas = 16.000 €
- Coste diseño del hardware = 30 euros /hora * 350 horas = 10.500 €
- Coste grupo de pruebas = 25 euros / hora * 180 horas = 4.500 €

El coste final se calcula como la suma de cada uno de los costes, dando un total:

Coste total = coste analista + coste programador + coste grupo de pruebas = **31.000 €**

14 Conclusiones

Como capítulo final se comentarán las conclusiones extraídas del desarrollo del proyecto, comparando los objetivos alcanzados con los que se plantearon cuando se definió el proyecto, las posibles líneas de trabajo futuro que se proponen y el análisis económico.

El principal objetivo de este proyecto era diseñar e implementar un procesador de *shaders* en *RTL/Verilog* para la GPU ATTILA. El resultado final del proyecto se ha cumplido y se ha conseguido un procesador completamente diseñado en hardware que puede ejecutar un subconjunto de instrucciones de ATTILA.

Aunque en un principio no se dio tanta importancia a las pruebas, en la fase final del proyecto, además de la implementación del procesador se han diseñado una serie de pruebas para validar el funcionamiento del mismo. Estas pruebas demuestran que la implementación realizada permite ejecutar las instrucciones descritas correctamente.

Dentro de los objetivos también estaba aprender el desarrollo de hardware utilizando *Verilog.* Los objetivos del proyecto se han cubierto, sin embargo hay muchas áreas entorno al diseño del procesador que se pueden ampliar que quedan como trabajo futuro.

14.1 Trabajo futuro

En este proyecto se ha implementado un procesador de *shaders* capaz de ejecutar un subconjunto de las instrucciones de ATTILA, a partir de aquí se pueden seguir muchas ramas de desarrollo, ya sea en cuanto a mejorar el rendimiento del pipeline, añadir instrucciones, o implementar el procesador físicamente utilizando una FPGA.

En cuanto a mejoras en el pipeline diseñado, se proponen seguir algunos de estos puntos:

- Implementación de una ALU que siga el algoritmo iterativo de multiplicación y división.
- Añadir nuevas instrucciones hasta completar el juego de instrucciones de ATTILA.
- Disminuir la latencia de aquellas instrucciones que sea posible.

Otras líneas de trabajo

- Implementar el procesador utilizando una FPGA.
- Implementar nuevos bloques en *Verilog* que interactúen con el procesador de *shaders*, de manera que se vaya construyendo una GPU totalmente hardware.

El hecho de que el proyecto ATTILA estuviera desarrollado completamente en software, ofrece una infinidad de posibilidades en cuanto a la continuación de este proyecto.

15 Bibliografía

[1] *Computer organization and design: the hardware/software interface.* David A. Patterson and John L. Hennessy. McGraw-Hill, 1994.

[2] *Computer architecture: a quantitative approach*. John L. Hennessy and David A. Patterson. Elsevier, Morgan, Kayfmann, 2007.

[3] *Verilog HDL: An Easy Approach for the Beginners*. Md. Liakot Ali, Dr. Ishak Aris and Dr. Roslina sidek. Jun 2010.

[4] *The Verilog Hardware Description Language*. Donald Thomas and Philip Moorby. Oct 2008.

[5] *ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures*. Víctor Moya, Carlos González, Jordi Roca, Agustín Fernández and Roger Espasa.

[6] *A SIMD-efficient 14 Instruction Shader Program for High-Throughput Microtriangle Rasterization*. Jordi Roca, Victor Moya, Carlos Gonzalez, Vicente Escandell, Albert Murciego, Agustin Fernandez and Roger Espasa.

[7] *Shader Performance Analysis on a Modern GPU Architecture*. Víctor Moya, Carlos González, Jordi Roca, Agustín Fernández and Roger Espasa.

[8] *A Single (Unified) Shader GPU Microarchitecture for Embedded Systems*.
Víctor Moya, Carlos González, Jordi Roca, Agustín Fernández and Roger Espasa.

[9] *Caracterización e implementación de algoritmos de compresión en la GPU ATTILA*. Christian Perez. Master Thesis for the Graduate Studies. Jan 2008.

[10] *Shader generation and compilation for a programmable GPU*. Jordi Roca. Master Thesis for the Graduate Studies. Jul 2005.

16 Anexos

16.1 Anexo I: Estructura y simulación del proyecto

En este apartado se explicará cómo está estructurado el código del proyecto, y cómo escribir y compilar un *shader* para luego simularlo utilizando el procesador.

Estructura del código

La carpeta raíz es 'shader', de la que cuelgan tres carpetas con el siguiente contenido:

Shader	Descripción
rtl	
fetch.v	Implementación de la etapa fetch
decoder.v	Implementación de la etapa decode
regfile.v	Implementación de la etapa register file
alu.v	Implementación de la etapa ALU
comparator.v	Implementación del comparador de la ALU
shader.v	Implementación del top module
validation.v	Implementación de los contadores de validación
defines.v	Definiciones de constantes usadas en el proyecto
fpu_32	Subcarpeta que contiene el código de la FPU
*.v	Ficheros que implementan la FPU
sim	
Makefile	Makefile para compilar el código en Verilog que se encuentra en ~/shader/rtl
t.do	Fichero para simular el procesador utilizando modelsim
test.v	Implementación del módulo para la simulación
test	
random_test.cpp	Código fuente del generador de tests
ieee_754_generator.cpp	Código fuente del generador de números en coma flotante
rom.sh	Script que ensambla el shader y crea un fichero para la lectura al incio de la simulación
test.sh	Script que comprueba si un shader dado se ejecuta correctamente
coverage.sh	Script que acumula los contadores para la validación del modelo en un fichero
autotest.sh	Script genera y prueba N tests aleatorios
ShaderProgramTest	Subcarpeta con el código de ATTILA para ejecutar los shaders y comparar el resultado con el del RTL

Tabla 16.1 Organización del proyecto

Simulación del proyecto

Para simular el proyecto tan solo hay que descargar el simulador *modelsim* de la web de *Altera*. Una vez descargado se utiliza la herramienta *vsim*.

Vsim tiene una línea de comandos en la cual hay que situarse en el directorio ~/shader/sim, de la misma forma que se haría en Linux, y ejecutar 'do t.do'. Aparecerá una ventana con el *waveform* de la simulación.

Para la ejecución de los *tests* es suficiente con utilizar el *script* 'autotest.sh' con los parámetros adecuados:

- Semilla inicial para generar programas aleatorios.
- Semilla final.

Se irán ejecutando *tests* generados aleatoriamente e irá indicando si el resultado es correcto o no. En caso de no serlo indicará la primera instrucción que ha fallado junto al valor de sus operandos y el del resultado.

16.2 Anexo II: IEEE 754

IEEE 754 es el estándar utilizado para aritmética en coma flotante. Define formatos para la representación de números en coma flotante, incluyendo el cero y otros valores especiales como *infinito* o *NaN* (*Not A Number*), y un conjunto de operaciones en coma flotante que trabaja sobre estos valores, así como cuatro modos de redondeo y cinco excepciones.

Especifica cuatro formatos para la representación:

- Precisión simple (32 bits).
- Precisión doble (64 bits).
- Precisión simple extendida (>= 43 bits), no se suele utilizar.
- Precisión doble extendida (>= 80 bits), se suele implementar con 80 bits.

Como este proyecto ha consistido en la implementación de un procesador de 32 bits, solo se explicará el formato de precisión simple.

Los 32 bits se distribuyen de la siguiente manera:

- 1 bit de signo (bit 31).
- 8 bits de exponente (bits 30-23).
- 23 bits de mantisa (bits 22-0).

El estándar define especifica el formato de cada uno de los valores posibles, incluyendo +/-INF, +/-0 y NaN.

Clase	Exp	Fracción
Ceros	0	0
Números desnormalizados	0	Distinto de 0
Números normalizados	1-254	cualquiera
Infinitos	255	0
NaN (Not a Number)	255	Distinto de 0

Tabla 16.2 Clases de valores definidos en el estándar IEEE 754

Como consecuencia de estos valores especiales, se han establecido normas que resuelven cuál es el resultado de las operaciones en cada caso:

- División por 0 produce +/-INF, excepto 0/0 que resulta como NaN.
- Log(+/-0) produce –INF. Log de un número negativo resulta como NaN.
- *Reciprocal square root* (RSQ) o *square root* (SQRT) de un número negative produce NaN. Las únicas excepciones son: SQRT(-0) = -0 y RSQ(-0) = -INF.
- INF – INF = NaN.
- (+/-)INF / (+/-) INF = NaN.
- (+/-) INF * 0 = NaN.
- 'NaN' operación 'valor' = NaN.
- La comparación de dos valores, al menos uno de ellos NaN, siempre da como resultado NEQ (*Not Equal*).
- +0 es comparado como EQ (*Equal*) a -0.
- +INF es considerado mayor que cualquier número, excepto NaN.
- -INF es considerado menor que cualquier número, excepto NaN.

Los algoritmos para la suma, resta, multiplicación y división de valores utilizando este estándar se ha explicado en el apartado dedicado a la FPU, apartado 7.5.2.

16.3 Anexo III: Campos de las instrucciones

qword	bits	size	name	description
0	0 - 7	8	opcode	Defines the instruction opcode (operation)
	8	1	endflag	Defines if the instruction finishes the current thread/program
	9	1	waitpoint	Defines if the instruction is a explicit wait point for data pending from units outside the shader processor (texture unit)
	10	1	predicated	Defines if the instruction is predicated
	11	1	invertpred	Defines if the register predicating the instruction must be negated
	16 - 12	5	predreg	Defines the predicate register used by the instruction
	19 - 17	3	op1bank	Defines the first source operand register bank
	20	1	op1negate	For integer and float point operands this flag defines if the first operand value is negated For predicate register operands this flag defines if the first operand value is inverted (NOT)
	21	1	op1absolute	For integer and float point operands this flag defines if the instruction uses the first operand absolute value For predicate operands this flag specifies that an immediate TRUE or FALSE value (based on the value of the op1negate flag) is used as the first operand.

	24 - 22	3	op2bank	Defines the second source operand register bank
	25	1	op2negate	For integer and float point operands this flag defines if the second operand value is negated For predicate register operands this flag defines if the second operand value is inverted (NOT)
	26	1	op2absolute	For integer and float point operands this flag defines if the instruction uses the second operand absolute value For predicate operands this flag specifies that an immediate TRUE or FALSE value (based on the value of the op1negate flag) is used as the second operand.
	29 - 27	3	op3bank	Defines the third source operand register bank
	30	1	op3negate	Defines if the third operand must be negated
	31	1	op3absolute	Defines if the third operand uses the absolute value
	34 - 32	3	resbank	Defines the result operand register bank
	35	1	saturatedres	For integer and float point instructions this flag defines if the instruction result is saturated (clamped to the [0, 1] range) For predicate instructions this flag defines if the instruction result is inverted (NOT)
	39 - 36	4	mask	Defines the write mask for the result register
	40	1	relmode	Defines if relative mode addressing is used for the constant operand
	42 - 41	2	reladdr	Defines the address register used for relative addressing mode into the constant bank

	44 - 43	2	reladdcomp	Defines the component of the address register used as a index for relative addressing mode into the constant bank
	53 - 45	9	reloffset	Defines the offset into the constant bank for relative addressing mode.
	54 - 63	10	reserved	Reserved for future use, should be 0.
For instructions with register only operands				
1	0 - 7	8	op1reg	Defines the register for the instruction first source operand
	15 - 9	8	op1swizzle	Defines the swizzling to be applied to the instruction first source operand
	23 - 16	8	resreg	Defines the register for the instruction result operand
	31 - 24	8	op2reg	Defines the register for the instruction second source operand
	39 - 37	8	op2swizzle	Defines the swizzling to be applied to the instruction second source operand
	47 - 40	8	op3reg	Defines the register for the instruction third source operand
	55 - 48	8	op3swizzle	Defines the swizzling to be applied to the instruction third source operand
	63 - 56	8	reserved	Reserved for future use (should be zero)
For with immediate second input operand				
1	0 - 7	8	op1reg	Defines the register for the instruction first source operand
	15 - 9	8	op1swizzle	Defines the swizzling to be applied to the instruction first source operand
	23 - 16	8	resreg	Defines the register for the instruction result operand
	31 - 24	8	reserved	Reserved for future use (should be zero)
	63 - 32	32	immediate	Defines the immediate value used as the instruction second source operand (broadcast to all components for SIMD instructions)