UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un Procesador de Altas Prestaciones para una Arquitectura Multinúcleo en FPGA #2

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor:
Tomás Picornell Sanjuan

Tutores:
José Flich Cardo
Xavier Molero Prieto
Pedro Juan López Rodríguez

CURSO 2014 - 2015

Resumen

En este trabajo se diseña e implementa un procesador con ejecución fuera de orden siguiendo como modelo el algoritmo de Tomasulo. El procesador es reconfigurable y permite tanto la instanciación de un número variable de unidades funcionales y recursos como la obtención de diferentes configuraciones, cada una con una relación prestaciones/recursos distinta. El procesador se está integrando en la arquitectura PEAK desarrollada en el Grupo de Arquitecturas Paralelas (GAP) del Departamento de Informática de Sistemas y Computadores (DISCA) de la Universitat Politècnica de València (UPV).

El procesador incluye todos los componentes esenciales para su completa operatividad así como soporte para un conjunto amplio del juego de instrucciones de la arquitectura MIPS32. Cabe añadir que todos los componentes se han diseñado e implementado por completo en el marco del presente trabajo.

El trabajo incluye el diseño de tests de prueba y diferentes programas para verificar y validar cada componente y las diferentes configuraciones finales del procesador. Por otro lado, se ha sintetizado cada uno de los componentes con el fin de obtener los recursos que necesita para su implementación en un sistema FPGA. A lo largo del desarrollo del trabajo se han utilizado herramientas comerciales como Vivado de Xilinx, simuladores (QtSpim) y software de control de versiones (Git).

Palabras clave: procesador, PEAK, FPGA, multinúcleo, configurable.

Resum

En aquest treball es dissenya i implementa un processador amb execució fora d'ordre seguint com a model l'algorisme de Tomasulo. El processador és reconfigurable i permet tant la instanciació d'un nombre variable d'unitats funcionals i recursos com l'obtenció de diferents configuracions, cadascuna amb una relació prestacions/recursos diferent. El processador s'està integrant en l'arquitectura PEAK desenvolupada en el Grup d'Arquitectures Paral·leles (GAP) del Departament d'Informàtica de Sistemes i Computadors (DISCA) de la Universitat Politècnica de València (UPV).

El processador inclou tots els components essencials per a la seua completa operativitat així com suport per a un conjunt ampli del joc d'instruccions de l'arquitectura MIPS32. Cal afegir que tots els components s'han dissenyat i implementat per complet en el marc del present treball.

El treball inclou el diseny de tests de prova i diferents programes per a verificar i validar cada component i les diferents configuracions finals del processador. D'altra banda s'ha sintetitzat cadascun dels components amb la finalitat d'obtenir els recursos que necessita per a la seua implementació en un sistema FPGA. Al llarg del desenvolupament del treball s'han fet servir diverses ferramentes comercials com ara Vivado de Xilinx, simuladors (QtSpim) i programari de control de versions (Git).

Paraules clau: processador, PEAK, FPGA, multinucli, configurable.

Abstract

This project involves the design and implementation of a processor with out-of-order execution using the Tomasulo algorithm. The processor is configurable, allowing a variable number of resources and functional units. Different configurations can be created, each with a different performance/resource ratio. The processor is being integrated into the PEAK architecture developed by the Grupo de Arquitecturas Paralelas (GAP) del Departamento de Informática de Sistemas y Computadores (DISCA) de la Universitat Politècnica de València (UPV). PEAK is a multi-core arquitecture for multi-FPGA development environments and prototyping.

The processor includes all the essential components and is fully operational along with support for a wide array of the MIPS32 architecture instruction set. All components have been designed and implemented as part of this project.

A multitude of tests and programs have been designed to verify and validate each component along with the different configurations of the processor. The synthesis of each of the components and the processor (in it's different configurations) has also been performed with the goal of obtaining the resource usage on a FPGA system. During the development of this project different commercial tools have been used such as Xilinx Vivado, simulators (QtSpim) and version control software (Git).

Keywords: processor, PEAK, FPGA, multicore, configurable.

Índice de abreviaturas

ALU (*Arithmetic Logic Unit*), Unidad aritmético-lógica.

BRANCH (*Branch*), Unidad de salto condicional.

BTB (*Branch Target Buffer*), Predictor de saltos.

BUS (*Bus*), Canal de comunicaciones.

CAS (*Controlled Adder Subtractor*), Sumador-restador controlado.

CISC (*Complex Instruction Set Computer*), Computador con Conjunto de instrucciones complejas.

CLA (*Carry-Lookahead Adder*), Sumador con anticipación del acarreo.

COMMIT (*Commit*), Unidad de confirmación.

CPA (*Carry Propagation Adder*), Sumador con propagación serie del acarreo.

CPU (*Central Processing Unit*), Unidad central de procesamiento.

CSA (*Carry Save Adder*), Sumador con acarreo almacenado.

DEC (*Decoder*), Decodificador de instrucciones.

DISCA (*Department of Computer Engineering*), Departamento de Informática de Sistemas y Computadores.

EX (*Execute*), Etapa de ejecución.

FPGA (*Field Programmable Gate Array*), Dispositivo de puertas lógicas programables.

FPU (*Floating Point Unit*), Unidad de coma flotante.

GAP (*Parallel Architectures Group*), Grupo de Arquitecturas Paralelas.

GPR (*General Purpose Register*), Registro de propósito general.

HDL (*Hardware Description Lenguage*), Lenguaje de descripción de hardware.

ID (*Instruction Decode*), Etapa de decodificación de instrucciones.

IDE (*Integrated Development Environment*), Entorno integrado de desarrollo.

IF (*Instruction Fetch*), Etapa de búsqueda de instrucciones.

IPC (*Instruction Per Cycle*), Instrucciones por ciclo.

L1 (*First Level Cache*), Memoria cache de primer nivel.

LUT (*Lookup Table*), Tabla de consulta.

MEM (*Memory*), Unidad de memoria.

MUX (*Multiplexor*), Multiplexor.

NaN (*Not A Number*), No es un número.

PC (*Program Counter*), Contador de programa.

QNaN (*Quiet Not A Number*), NaN de tipo silencioso.

RAM (*Random Acces Memory*), Memoria de acceso aleatorio volátil.

RAW (*Read after write*), Lectura después de escritura.

RB INT (*Integer Register Bank*), Banco de registros de enteros.

RB FP (*Floating Point Register Bank*), Banco de registros de coma flotante.

RISC (*Reduced Instruction Set Computer*), Computador con Conjunto Reducido de Instrucciones.

ROB (*Re-Order Buffer*), Cola de reordenación de instrucciones.

RR (*Round Robin*), Árbitro de selección cíclico.

RS (*Reservation Station*), Estación de reserva.

SNaN (*Signalling Not A Number*), NaN con señalización.

WAR (*Write after Read*), Escritura después de lectura.

WAW (*Read after Write*), Escritura después de escritura.

WB (*Writeback*), Etapa de escritura de resultados.

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1

Introducción

En el presente trabajo se ha implementado un procesador con ejecución fuera de orden y con especulación, siguiendo la arquitectura base MIPS. Por otra parte, para poder estudiar en profundidad el comportamiento de este nuevo procesador, se ha parametrizado toda la arquitectura para así poder activar o desactivar los componentes que se quieran en cualquier momento y estudiar diferentes configuraciones. En concreto se han definido hasta siete configuraciones del procesador distintas, cada una con un número de recursos y prestaciones diferentes.

La configuración más completa está compuesta por cuatro descodificadores con una cola de reordenación de treinta y dos entradas, cuatro buses, cuatro unidades de confirmación, cuatro unidades aritméticas así como dos unidades de coma flotante con varias estaciones de reserva. También se incluyen bancos de registros para números enteros y de coma flotante.

Otro aspecto importantes del procesador es que soporta instrucciones de coma flotante, implementa predicción de salto, y por último, también cuenta con soporte de desambiguación de memoria.

Cabe destacar que este trabajo ha sido llevado a cabo por un grupo de cuatro miembros. Durante todo el proyecto se ha trabajado en equipo, especializándose cada uno de los componentes del grupo en una parte específica. La validación y ejecución final del procesador (en sus diferentes versiones), se ha realizado conjuntamente entre todo el equipo.

1.1 Contexto y Motivación

El trabajo está relacionado con el desarrollo de la arquitectura PEAK, que se está implementando en el Grupo de Arquitecturas Paralelas (GAP) del Departamento de Informática de Sistemas y Computadores (DISCA) de la Universitat Politècnica de València. PEAK es una arquitectura de procesador multinúcleo con

memoria compartida. El objetivo de PEAK es el uso de técnicas de emulación en sistemas FPGA para la investigación (así como para la docencia) en temas de diseño de nuevas arquitecturas de procesador multinúcleo. PEAK permite múltiples variantes en el diseño de la arquitectura, partiendo de la definición de los núcleos, definiendo completamente el procesador, la jerarquía de memoria y la gestión de los recursos del procesador, los protocolos de coherencia entre memorias cache y, así mismo, define el modelo de programación y el protocolo de acceso al sistema desde un computador externo.

Uno de los principales motivos para la definición y uso de PEAK es el soporte, en temas de investigación, de modelos de gestión avanzada de núcleos y memoria cache en sistemas multinúcleo. Así, el denominado concepto *capacity computing* define el uso particionado de recursos (núcleos y memorias cache) de un procesador multinúcleo, asignando cada partición a una aplicación diferente. De este modo, se permite que el procesador ejecute aplicaciones de forma concurrente sobre recursos disjuntos, permitiendo un aislamiento perfecto y así proveyendo de características de seguridad y privacidad entre aplicaciones, además de garantizar prestaciones al no tener interacciones entre aplicaciones. PEAK, para ello, define protocolos de coherencia y mecanismos de encaminamiento y reconfiguración a nivel de red dentro del procesador, que gestionados desde la capa software de control, permiten obtener tales características.

Actualmente se está trabajando en la arquitectura PEAK con el objetivo de obtener un procesador de 256 núcleos con las características anteriores. Este proyecto está enmarcado en un convenio entre la UPV y una empresa multinacional china. El objetivo del proyecto es aplicar el concepto de *capacity computing* en sistemas con un elevado número de núcleos. Adicionalmente, la arquitectura PEAK se está utilizando como punto de partida para un proyecto europeo en el contexto de HPC (*High-performance Computing*), el cual arranca en octubre del 2015. En dicho proyecto se pretende utilizar el concepto de *tile* que define la arquitectura PEAK con el objetivo de que cada *tile* tenga núcleos especializados en diferentes tareas y, sobre todo, que tengan características de prestaciones y consumo distintas. Por ejemplo, los procesadores definidos en esta memoria se pueden instanciar en diferentes *tiles* y cada uno de ellos con un número de recursos distinto (número de vías, números de unidades de FPU, etc). El objetivo del proyecto es el de implementar una asignación adecuada de aplicaciones a recursos con el fin de obtener una relación prestaciones/consumo óptimo para cada momento. Es por lo tanto, dentro de este sistema, donde se enmarca el trabajo descrito en esta memoria.

Por otro lado, desde que se comenzó a extender su uso, las FPGA (*Field Programmable Gate Array*) se están utilizado para todo tipo de propósitos. Gracias a estos dispositivos podemos prototipar y estudiar el comportamiento de los diseños digitales que creamos conveniente sin tener que implementar en silicio el diseño cada vez que hagamos una pequeña variación, ofreciendo la posibilidad de una depuración más rápida.

1.2 Objetivos

A continuación se muestran los objetivos parciales que se fijaron inicialmente para el proyecto.

- El objetivo principal del trabajo es diseñar e implementar las unidades necesarias para obtener un procesador operativo. La implementación se realizará mediante el lenguaje Verilog.
- Instrucciones de diferentes tipos: lectura/escritura en memoria y registros, aritmético-lógicas y de salto.
- Facilitar la reconfiguración del núcleo de procesamiento permitiendo otorgar más o menos recursos a la arquitectura, como pueden ser múltiples unidades aritmético-lógicas o cambiando el número de estaciones de reserva que tiene cada operador.
- Segmentar las unidades aritmético-lógicas y de memoria para soportar la ejecución de múltiples instrucciones en paralelo y soportar la desambiguación de memoria.
- Realizar algunos programas de prueba en ensamblador para ejecutarlos sobre la arquitectura y verificar el correcto funcionamiento de la implementación.
- Sintetizar el procesador y realizar un análisis sobre el espacio que ocupará cada componente de la arquitectura en un sistema FPGA. Esta parte se realizará sobre diferentes configuraciones del núcleo de procesamiento asignando más o menos recursos y observando las diferencias.
- Adaptar e incorporar el procesador en la arquitectura PEAK con el fin de utilizarlo como procesador base en un sistema FPGA.

1.3 Estructura de la Memoria

La memoria del presente trabajo se compone de seis capítulos, que procedemos a exponer a continuación:

- **Capítulo 1, Introducción:** Este capítulo expone la motivación, el contexto en el cual se ha basado el desarrollo del proyecto, así como los objetivos a abarcar.
- **Capítulo 2, Conceptos previos:** Se exponen los conocimientos necesarios para el debido entendimiento y comprensión de los temas que se abordan en esta memoria.

- **Capítulo 3, Entorno de trabajo:** En este apartado se detallan las herramientas utilizadas durante la realización del trabajo, indicando las ventajas que aportan y el porqué de su utilización.
- **Capítulo 4, Visión general del procesador:** Se muestra el conjunto de componentes para el correcto funcionamiento del procesador así como las tareas que efectúan y su interconexión.
- **Capítulo 5, Diseño de componentes específicos:** Se detallan los módulos que se han desarrollado por cada uno de los miembros específicos del equipo.
- **Capítulo 6, Verificación y resultados:** Se procede a realizar una serie de pruebas para verificar la implementación realizada, analizando a su vez la eficacia del producto final en términos de prestaciones y consumo de recursos.
- **Capítulo 7, Conclusión:** En este capítulo se detallan los objetivos cumplidos y los posibles trabajos futuros y ampliaciones.

Puesto que este trabajo se ha realizado en grupo por los alumnos Francisco Guaita, Mark Holland, Raúl Lozano y Tomás Picornell, además de la descripción del módulo asignado a cada uno realizada en el capítulo 5, cada uno de los componentes expone en la última sección de los capítulos 6 y 7 una visión particular, a partir del trabajo individual realizado.

Finalmente, se incluyen varios anexos al final de la memoria, en los cuales se detallan el interfaz de los diferentes módulos que componen el núcleo de procesamiento y los programas de prueba empleados.

1.4 Uso de la Bibliografía

Durante el trabajo nos hemos apoyado en múltiples materiales bibliográficos de los cuales aquí se destacan algunos, así como la relación que estos guardan con las diferentes partes del proyecto.

- Inicialmente, en aras a situarnos dentro del contexto histórico de los microprocesadores MIPS32, hacemos uso de la referencia bibliográfica [9] que es una entrada de la Wikipedia donde se habla de los procesadores MIPS en general.
- A su vez, como uno de los objetivos de este trabajo es construir un procesador con ejecución fuera de orden, para este propósito nos basamos en las estructuras definidas en los libros de Hennessy y Patterson sobre arquitectura de computadores [6, 7]. También hacemos uso del manual oficial de MIPS que introduce la arquitectura MIPS32 [10].

- Además, con tal de poder descodificar las instrucciones y ejecutarlas sobre nuestra arquitectura, se han seguido las codificaciones definidas en las manuales oficiales de MIPS que cubren el conjunto de instrucciones que soporta la arquitectura MIPS32 [11, 12].
- Puesto que nuestro sistema tiene soporte para instrucciones de coma flotante debemos seguir el estándar oficial de la organización IEEE que se puede consultar en [8].
- De manera similar, para algunos detalles concretos del diseño de la arquitectura, expresamente para la unidad aritmético-lógica, se ha consultado la referencia [14].
- A la hora de implementar un programa que aproximara el número π se utilizó como referencia [16].
- Finalmente, para resolver dudas respecto al uso de las herramientas utilizadas durante la composición de este proyecto consultamos el libro oficial de Git [5] donde introducen todos los conceptos con ejemplos, la documentación oficial del programa PCSPIM [13] que contiene toda la información necesaria para su uso y por último la guía de usuario de Vivado [15] proporcionado por Xilinx.

CAPÍTULO 2

Conceptos Previos

En este capítulo se presenta un breve resumen de los conceptos básicos relacionados con el trabajo desarrollado.

2.1 La Codificación IEEE 754

Este estándar surgió a causa de las diferentes formas de implementar el coma flotante en los grandes computadores en torno al año 1976. A causa de ello, se celebraron reuniones del IEEE (*Institute of Electrical and Electronics Engineers*) desde 1977 hasta 1985 que fue cuando se hizo oficial el estándar IEEE 754.

Dicho formato sirve para cálculos definidos sobre números reales (*float* y *double* en alto nivel). Además, especifica cómo deben representarse los números en coma flotante con simple precisión (32 bits) o doble precisión (64 bits), y también cómo deben realizarse las operaciones aritméticas entre ellos.

Esta notación exponencial nos permite representar rangos muy grandes de números con un número limitado de bits. Por ejemplo un número en coma flotante de simple precisión se almacena en una palabra de 32 bits tal y como se muestra en la Figura 2.1.

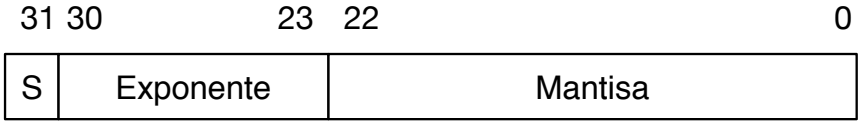

Figura 2.1: Distribución de los bits en coma flotante de simple precisión

El primer bit es el bit de signo (S), los siguientes 8 son los bits del exponente (E) y los 23 restantes son la mantisa o fracción (M). Un número cualquiera X

representado en notación exponencial se puede escribir como:

$$X = (-1)^S \times 1.M \times 2^{E-127}$$

2.2 La Arquitecura MIPS

Las siglas MIPS (*Microprocessor without Interlocked Pipelines Stages*) hacen referencia a la familia de microprocesadores desarrollados por MIPS Technologies, con arquitectura RISC (*Reduced Instruction Set Computer*).

Los procesadores MIPS fueron creados por un equipo con John L. Hennesy a la cabeza, en la Universidad de Stanford.

Tras dejar la universidad de Stanford, Hennesy funda la compañia MIPS Computer Systems con la que crea el primer modelo de CPU comercial conocido como R2000, el cual disponía de 32 registros de propósito general. Sin embargo, no contaba con soporte para instrucciones de coma flotante, con lo cual, para poder añadir este comportamiento, era necesario añadir el chip R2010.

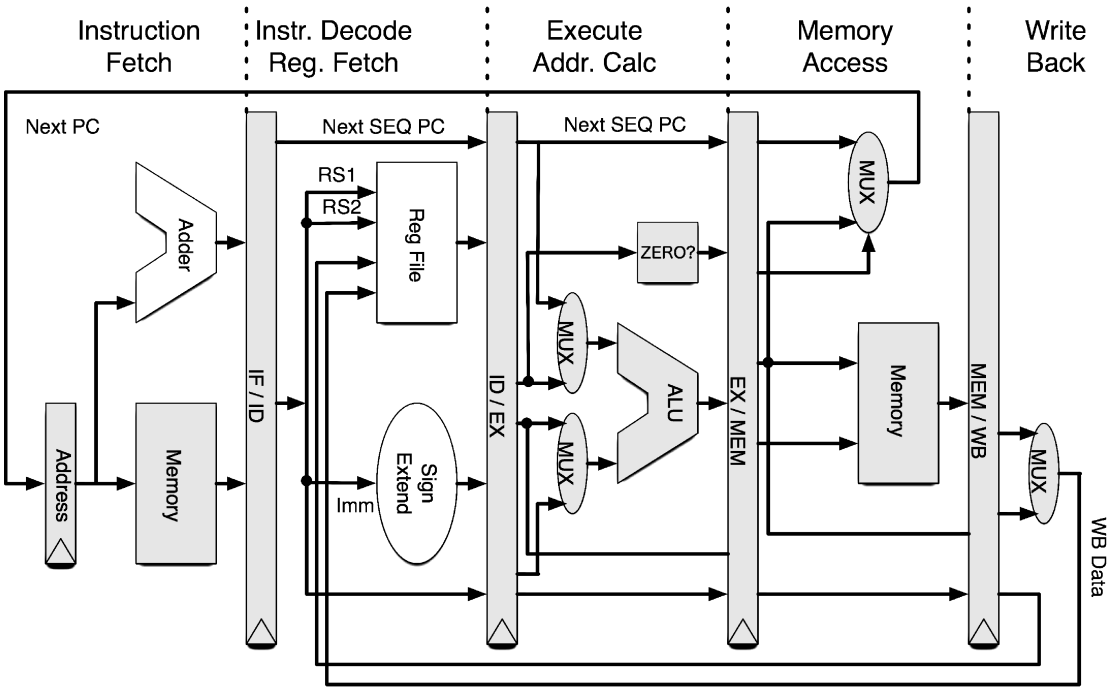

Figura 2.2: Ruta de datos de la arquitectura MIPS donde se muestran las unidades que la componen así como la relación entre estas

Poco después con el nuevo modelo R3000, que sería el sucesor del R2000, añadió una nueva memoria cache de 32 KB, además de un soporte de coherencia de cache, lo que lo hacía idóneo para el uso de varios núcleos. Un ejemplo de utilización que se le dio a este procesador fue en la consola Sony PlayStation.

El siguiente modelo de la serie, el R4000, fue presentado en el año 1991, el cual amplió el juego de instrucciones para dar soporte a una arquitectura de 64 bits. Es en esta serie donde se integra por completo el chip de operaciones de coma flotante. También se aumentó la frecuencia de reloj del modelo anterior y se perfeccionó la técnica de segmentación para lograr este aumento. Poco después del lanzamiento de dicho modelo y debido a la mala situación de la empresa, MIPS Computer Systems fue comprada por SGI, fecha a partir de la cual se comienzan a otorgar licencias de algunos de sus productos a terceros.

La consolidación del sistema de licencias MIPS se consolida a finales del siglo XX con el lanzamiento de la arquitectura MIPS32 y MIPS64, de 32 y 64 bits respectivamente, que llega hasta la actualidad, siendo los núcleos MIPS unos de los más importantes dentro del mercado de dispositivos como ordenadores portátiles, decodificadores o sintonizadores de televisión.

A estos modelos se une el R5000, el cual obtiene un rendimiento mucho mejor en coma flotante que el de sus predecesores, así como el acceso a memorias cache de mayor capacidad en dos ciclos de reloj. Dicho modelo, a diferencia de los anteriores, fue implementado por la compañía Quantum Effect Devices, fundada por antiguos ingenieros de MIPS. A su vez, dicha compañía también creó las familias R7000 y R9000 diseñadas para sistemas empotrados.

En 1994 se lanzó al mercado el modelo R8000, convirtiéndose en el primer diseño MIPS superescalar, capaz de procesar dos operaciones aritméticas sobre enteros y otras dos de memoria en cada ciclo de reloj mediante un diseño en seis chips que se descomponían en una unidad entera (con dos caches de 16 KB, una para instrucciones y otra para datos), una unidad de coma flotante, así como un controlador de memoria cache. El diseño incluía además dos unidades segmentadas de suma-multiplicación en doble precisión. A pesar de ser el primer diseño MIPS superescalar, su elevado coste fue uno de los causantes de su poco éxito fuera del mercado científico, lo que hizo que estuviera en el mercado cerca de un año.

Un año después fue lanzado el modelo R10000, cuyo diseño era en un solo chip, aumentando sensiblemente la frecuencia de reloj respecto a su predecesor. Como característica principal de este modelo cabe destacar que implementa ejecución fuera de orden, lo que permite evitar los ciclos de parada que se introducen en caso de dependencias entre las instrucciones ejecutadas, como ocurría con los demás procesadores.

Los modelos posteriores (R12000, R140000, R16000 y R16000A) se basaron en el R10000 debido a su éxito entre sus clientes, aumentando la frecuencia de reloj, así como la utilización de tecnología que permitiera reducir el área del chip al máximo posible.

En la figura 2.2, se muestra la ruta de datos completa de un procesador MIPS segmentado básico.

2.2.1. Formato de las Instrucciones

Dentro de esta arquitectura podemos distinguir tres formatos en la codificación de las instrucciones:

1. Tipo I: Son instrucciones de tipo inmediato, donde uno de los operandos es siempre un número entero codificado en 16 bits. Este formato permite dos operandos adicionales (rs y rt) codificados en registros. El código de operación (común a todos los formatos), ocupa los 6 bits de mayor peso. Este formato codifica las instrucciones aritméticas que operan con un operando inmediato, las instrucciones de carga y almacenamiento con un modo de direccionamiento desplazamiento, y las instrucciones de salto condicional con un desplazamiento de 16 bits sobre el PC.

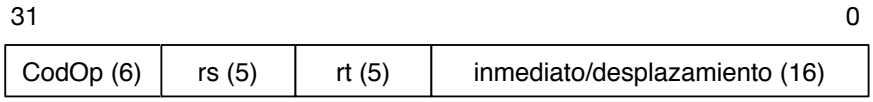

Figura 2.3: Formato de instrucciones I

2. Tipo J: Codifica instrucciones de salto incondicional. La dirección de salto se codifica como un desplazamiento de 26 bits desde el PC.

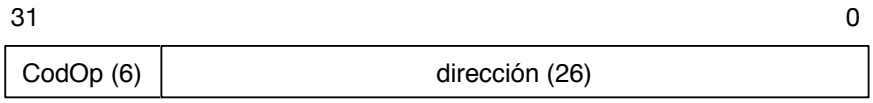

Figura 2.4: Formato de instrucciones J

3. Tipo R: Este formato es similar al tipo I, pero en este caso, las instrucciones aritméticas operan sobre datos almacenados en los registros (rd, rs y rt). Además, utiliza otros 6 bits para indicar una extensión del código de operación de la instrucción.

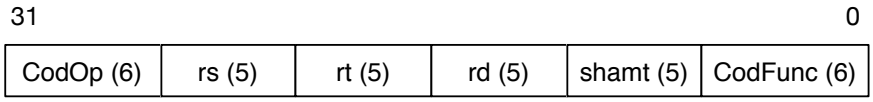

Figura 2.5: Formato de instrucciones R

2.3 Ejecución de Instrucciones

En general, un procesador a la hora de ejecutar instrucciones tiene dos formas de proceder. En un primer caso, la ejecución viene marcada por el orden en la cual están codificadas las instrucciones en el programa, mientras que en otra ese orden puede alterarse dinámicamente. En el primer modo de ejecución, el procesador lee la instrucción y la ejecuta, pero bloquea toda la ruta de datos si la instrucción en ejecución necesita algún operando, por lo que el procesador se queda a la espera de este resultado, sin dejar que otras instrucciones independientes (que no precisan de ese resultado), empiecen incluso a descodificarse.

Sin embargo, si utilizamos una implementación que cambie el orden de las instrucciones dinámicamente, podemos evitar estos bloqueos innecesarios. Además, obtenemos diversas ventajas. La primera, permite que el código compilado para una ruta de datos funcione a la correctamente en otra que contenga varias vías de ejecución, por ejemplo procesadores superescalares, eliminando así la necesidad de estar recompilando el programa para diferentes arquitecturas. La segunda ventaja radica en que permite manejar algunos casos donde las dependencias son desconocidas en tiempo de compilación. Por último, permite al procesador tolerar retardos impredecibles, como por ejemplo fallos de cache.

2.3.1. Ejecución Dinámica de Instrucciones

Como se ha mencionado anteriormente, la mayor limitación de las rutas de datos con ejecución en orden es que la ruta de datos se ve bloqueada en caso de que existan dependencias con instrucciones anteriores que aún estén en ejecución. Consideremos a modo de ejemplo, el siguiente código:

```
div.d $f0,$f2,$f4    #$f0 = $f2 / $f4
add.d $f10,$f0,$f8   #$f10 = $f0 + $f8
sub.d $f12,$f8,$f14  #$f12 = $f8 - $f4
```

Fijándonos en el código anterior, la instrucción `sub.d` no puede ejecutarse porque existe una dependencia de datos entre las dos instrucciones anteriores, a pesar de que última no tiene ningún tipo de relación con las primeras.

Para permitir la ejecución fuera de orden (*out-of-order execution*), es esencial que se divida la etapa de descodificación en dos:

1. *Issue* - Descodificación de la instrucción y comprobación de existencia de riesgos estructurales (dependencias).

2. *Read operands* - Esperar hasta que los operandos estén listos, evitando así posibles conflictos.

Por tanto, hay que distinguir entre *el comienzo de la ejecución* y *la finalización de la ejecución*, puesto que la ruta de datos permite la ejecución de varias instrucciones al mismo tiempo; de hecho, si no se permite esta ejecución en paralelo de instrucciones, se pierde la mayor ventaja de este método. Es por ello que deberán existir múltiples unidades funcionales, a fin de dotar de paralelismo al procesador.

En resumen, la ejecución dinámica de instrucciones hace que todas las instrucciones entren en orden a la etapa *Issue*. Sin embargo, estas pueden quedarse a la espera de sus operandos (*Read operands*) o pasar a ejecución, fuera de orden, alterando en la práctica la secuencia del programa, pero no su resultado.

La técnica más usada actualmente a la hora de llevar a cabo este tipo de ejecución es el denominado algoritmo de Tomaluso, creado por Robert Marco Tomasulo mientras trabajaba en IBM, añadido al diseño del IBM 360/91, modelo del año 1967. La principal diferencia respecto a lo explicado anteriormente es que este algoritmo gestione las antidependencias mediante la técnica del renombrado de registros, a la vez que permite la ejecución especulativa de instrucciones cuando dentro del código existen saltos y mecanismos de predicción del comportamiento de estos. Por ejemplo, si consideramos el siguiente código:

```
div.d $f0,$f2,$f4     #$f0 = $f2 / $f4
add.d $f6,$f0,$f8     #$f6 = $f0 + $f8
s.d   $f6,0($t1)      #MEM[0+$t1] = $f6
sub.d $f8,$f10,$f14   #$f8 = $f10 - $f14
mul.d $f6,$f10,$f8    #$f6 = $f10 * $f8
```

Se puede observar que si la instrucción `sub.d` produce el dato antes de que entre la instrucción `add.d` a ejecución (debido a la ejecución fuera de orden), el resultado de la `add.d` será erróneo. Este tipo de dependencia se conoce como antidependencia.

2.3.2. Algoritmo de Tomasulo

El esquema inventado por Robert Tomasulo permite ejecutar instrucciones solo cuando los operadores están disponibles, a la vez que se mantiene una ejecución fuera de orden, minimizando por lo tanto los conflictos *read-after-write* RAW, e introduciendo el renombrado de registros para minimizar también otros conflictos como pueden ser los *write-afther-write* WAW y *write-afther-read* WAR. Este modelo está presente en multitud de computadores modernos con las variaciones propias de la marca que produce el dispositivo, pero en rasgos generales, se sigue el algoritmo presentado.

Usando este esquema eliminamos una gran cantidad de riesgos en cuanto a dependencias. Cabe añadir que los conflictos WAR y WAW son eliminados gracias al renombrado de registros. Esta técnica, como su propio nombre indica, cam-

bia el nombre a los registros en uso, para que si estos tienen una antidependencia no afecte al resultado del programa. El método anteriormente nombrado se complementa gracias a una nueva estructura de datos llamada estación de reserva (*Reservation Stations)*, que no es más que un *buffer* donde se introducen las instrucciones con sus respectivos operandos a la espera de que estos contengan un valor válido, que se obtendrán una vez la instrucción que genera dicho valor finalice su ejecución. Para un mayor detalle del algoritmo Tomasulo, se recomienda la lectura de [6].

CAPÍTULO 3

Entorno de Trabajo

En esté capítulo se presentan las herramientas utilizadas para el desarrollo del trabajo así como una breve justificación de su uso y descripción de las funciones que ofrecen.

3.1 Xilinx Vivado

La herramienta más importante que se ha usado durante el proyecto es el entorno de desarrollo denominado Vivado, creado por la compañía Xilinx. Se escogió este producto debido a que se usarán FPGA proporcionadas por esta misma empresa. Vivado es un Entorno Integrado de Desarrollo que ofrece todas las herramientas necesarias para el diseño, implementación e integración de prototipos en FPGA. Al usar esta herramientas podremos:

1. Diseñar los módulos utilizando lenguaje Verilog o VHDL para después integrar el comportamiento descrito en el código, dentro de una FPGA.
2. Facilitar la tarea del programador. Vivado dispone de un analizador estático de la sintaxis del código.
3. Escribir pruebas de test(*testbench*) del código escrito anteriormente.
4. Ejecutar el diseño en un simulador usando programas de prueba para analizar el comportamiento del componente usando un cronograma y viendo en cada instante de tiempo los valores de las variables y así poder depurar el código más fácilmente.
5. Sintetizar el diseño, a la vez que se analiza la cantidad de componentes internos que usará el módulo sintetizado dentro de la FPGA.
6. Implementar finalmente el diseño en la FPGA.

En la figura 3.1 podemos apreciar la multitud de opciones que tenemos a nuestro alcance al utilizar este programa. La interfaz del programa está divida en cuatro partes diferenciadas:

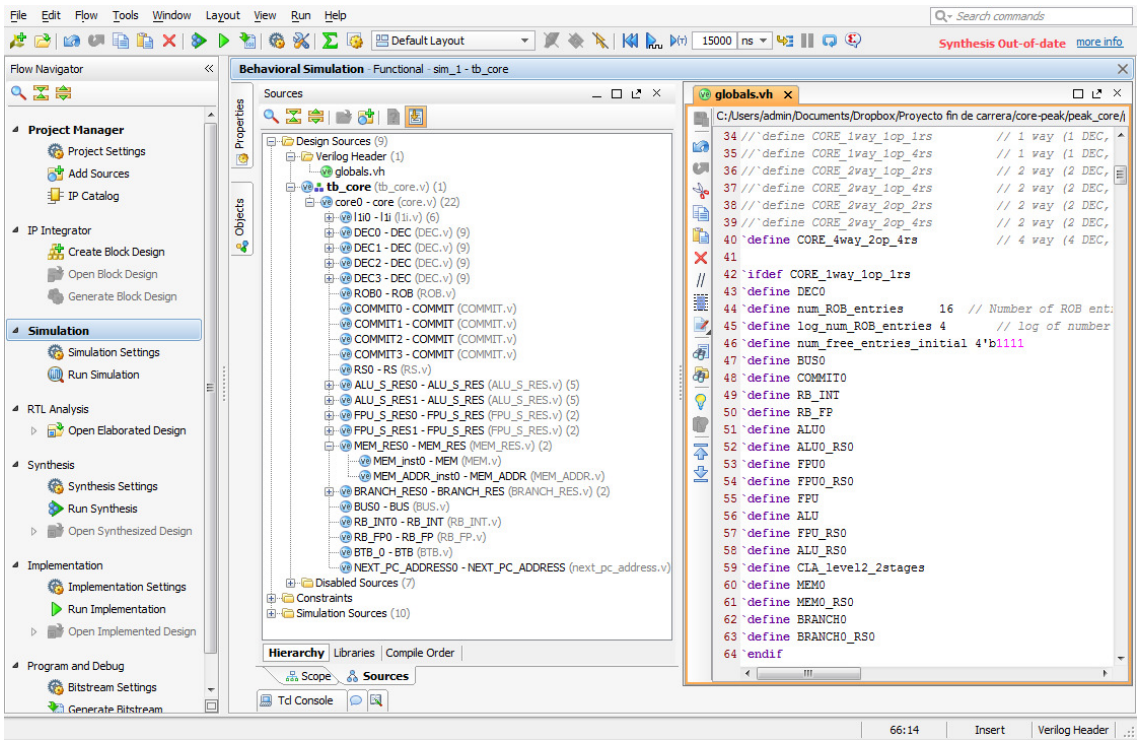

Figura 3.1: Captura de pantalla Xilinx Vivado mientras se edita un archivo

1. Barra superior, con menús y opciones, estas últimas cambian según en la etapa que estemos del desarrollo.

2. Ventana izquierda (etapa del desarrollo), según la fase en la que nos encontremos necesitamos herramientas diferentes, incluso cambiar ciertas partes de la interfaz para adecuarla a las características de la fase.

3. Ventana central, junto con la derecha, cambian según la etapa de desarrollo que esté seleccionada. En la figura 3.2 se puede observar los módulos que se han diseñado.

4. Ventana derecha, es el area principal de la interfaz, según la fase en la que se esté tiene un contenido u otro. En el caso concreto de la imagen, al estar en la etapa de simulación, esta ventana suele alternar entre contener código o un cronograma que muestra el comportamiento del diseño simulado.

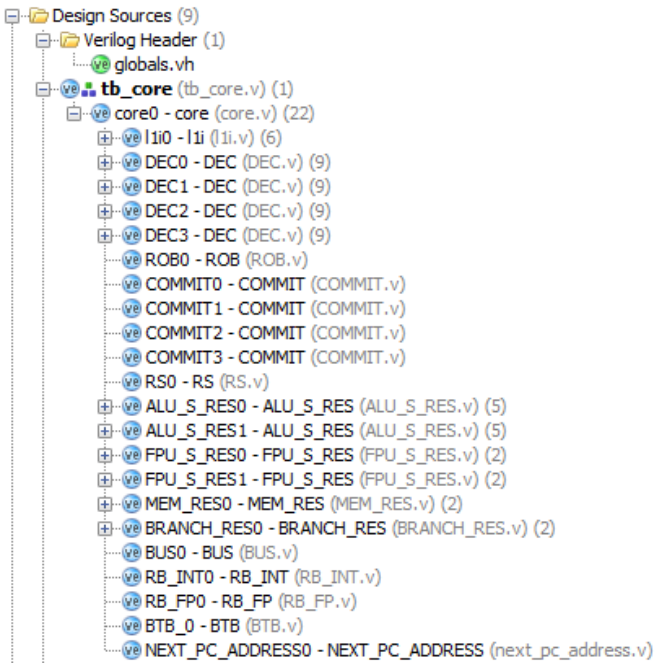

Figura 3.2: Árbol de módulos del proyecto

3.2 QtSpim

QtSpim es un simulador que ejecuta programas creados para MIPS32. Además proporciona un depurador simple, así como un conjunto mínimo de servicios del sistema operativo. A su vez, QtSpim implementa el juego de instrucciones completo de MIPS32 por lo que puede leer y ejecutar programas en lenguaje ensamblador escritos para este tipo de procesadores.

La herramienta QtSpim nos permitirá ejecutar programas MIPS32, analizando su comportamiento y comparándolo con los resultados que obtenemos en el procesador que diseñaremos en este trabajo, facilitando la tarea de depuración.

QtSpim es una versión de SPIM [13] que utiliza el *framework* Qt, facilitando así la portabilidad de su código. Como puede observarse en la figura 3.3, el contenido de los registros del procesador se ve alterado con la ejecución del programa. De esta manera, podemos comprobar si el funcionamiento del programa en el procesador diseñado es el correcto comparando la ejecución en nuestro prototipo con el obtenido en QtSpim. A su vez, también se puede ver en qué dirección de memoria están ubicadas las instrucciones, así como su codificación en código hexadecimal. Esto nos resultará útil para codificar nuestros códigos de prueba en Vivado.

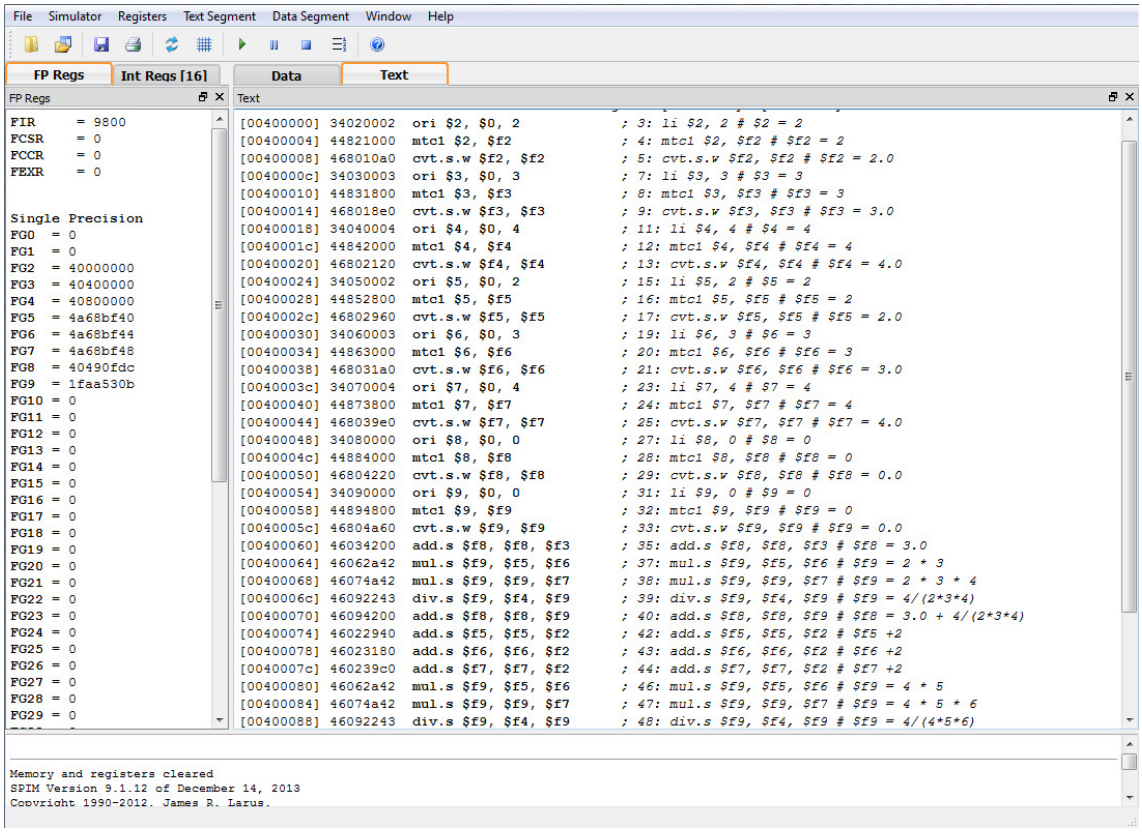

Figura 3.3: Ejemplo de ejecución en QtSpim de un programa en código ensamblador

3.3 Git

Como se ha mencionado, el presente trabajo se ha desarrollado en paralelo por un grupo de cuatro personas, por lo que se requiere de cierta sincronización y cuidado a la hora del desarrollo del código. Además, el resultado final es la suma de todos los componentes y la correctitud del procesador dependen de que todos los módulos funcionen debidamente y que sus interfaces sean trabajadas en equipo. Por esta razón se hace necesario el uso de algún tipo de herramienta de control de versiones para mantener la integridad de los archivos del mismo. Se ha elegido Git debido a la familiaridad de los miembros del equipo con dicha herramienta, a la vez que la UPV ha proporcionado un servidor para alojar un repositorio remoto, accesible desde cualquier sitio por todos los miembros del equipo.

Como se ha mencionado anteriormente, Git es un software de control de versiones. Linus Torvalds lo diseñó en un principio buscando la eficiencia y confiabilidad de versiones de aplicaciones, pensando en Git como una estructura sobre el cual otras personas pudieran escribir la interfaz de usuario o *front end*. Cabe des-

tacar que existen proyectos de gran envergadura que utilizan esta herramienta, como la programación del propio núcleo de Linux.

En la Tabla 3.1, podemos observar las ordenes básicas para la utilización de Git. El flujo de trabajo se puede observar en la figura 3.4.

Comando	**Descripción**
git add	Registrar cambios
git commit -m	Guardar cambios con un mensaje
git log	Ver los diferentes cambios de una rama
git push	Subir los cambios al repositorio remoto
git pull	Actualizar el repositorio local
git checkout *nombre de la rama*	Cambiar de rama
git checkout -b *nombre de la rama*	Crear rama
git checkout -D *nombre de la rama*	Borrar rama
git status	Estado actual de la rama
git fetch	Actualizar estado de todas las ramas

Tabla 3.1: Instrucciones básicas de Git

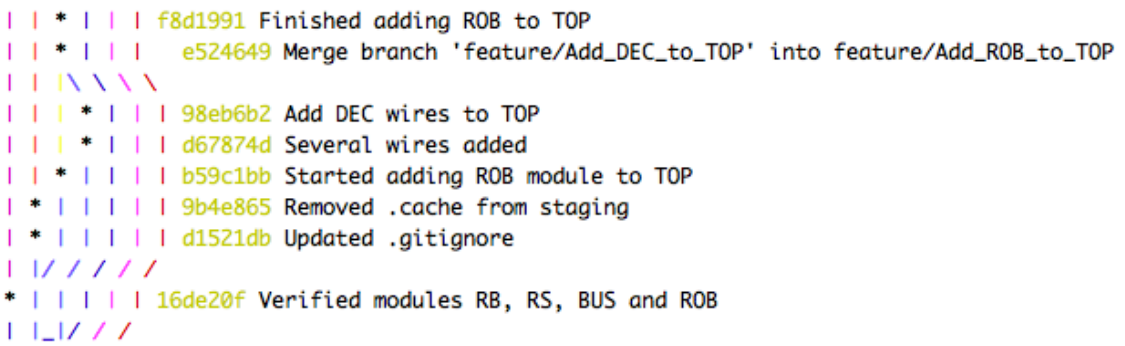

Figura 3.4: Ejemplo de flujo de trabajo usando el programa de control de versiones Git

3.4 log2chronogram

Para facilitar el análisis de la ejecución de los programas en el procesador desarrollado, se ha implementado, usando el lenguaje Python, un *script* que nos ayuda con ese proceso, creando a partir de un registro de *log* de Vivado, un cronograma más legible al ojo humano. De esta manera, podemos observar si la ejecución ha sido correcta de una manera más fácil, como se muestra en el siguiente cronograma. En el eje de abscisas se muestra el tiempo medido en ciclos de reloj mientras que el eje de ordenadas muestra las instrucciones ejecutadas. El cronograma permite conocer qué fase de cada instrucción se está realizando en un instante dado.

```
 instr/cycles          2     3     4     5     6     7     8     9
addi $8,  $0, 10     DEC   ADD   ADD   WBK   COM
addi $9,  $0, 20           DEC   ADD   ADD   WBK   COM
addi $10, $0, 30                 DEC   ADD   ADD   WBK   COM
addi $11, $0, 40                       DEC   ADD   ADD   WBK   COM
```

CAPÍTULO 4

Visión General del Procesador

Como se menciona en el primer capítulo, el proyecto se ha desarrollado de forma conjunta por cuatro alumnos, estando cada uno de ellos encargado de una parte del procesador. En este capítulo se muestra una visión general del procesador.

4.1 La Ruta de Datos

En la figura 4.1 se puede observar el esquema general del procesador desarrollado. Cabe matizar que está imagen no muestra la totalidad del diseño debido a su complejidad; por tanto solo contiene los módulos más importantes así como un resumen de las relaciones existentes entre ellos. Es decir, se han obviado conexiones para que la figura tenga un aspecto más legible. En el apéndice A se detalla el interconexionado entre módulos. Otro aspecto a resaltar en esta figura es el hecho de que no se muestran registros de segmentación.

En la figura 4.1, se aprecia cómo la ruta de datos está compuesta por varios grupos de módulos. El primero de ellos, el que está más a la izquierda, es el encargado del cálculo de la dirección de la instrucción así como de la obtención de instrucciones. Dicho grupo está compuesto por:

- El PC, registro de 32 bits que almacena la dirección de programa.
- El BTB (*Branch Target Buffer*), que permite predecir el resultado de los saltos condicionales y así reducir el efecto negativo de los riesgos de control. Está implementado mediante una unidad que contiene una tabla de registros con un número de entradas configurable. Cada entrada del módulo guarda la dirección de destino del salto, el PC de esa instrucción y unos bits para guardar la última predicción y si resultó en un acierto. Esa predicción se realiza mediante un autómata de cuatro estados con histéresis; si se predice

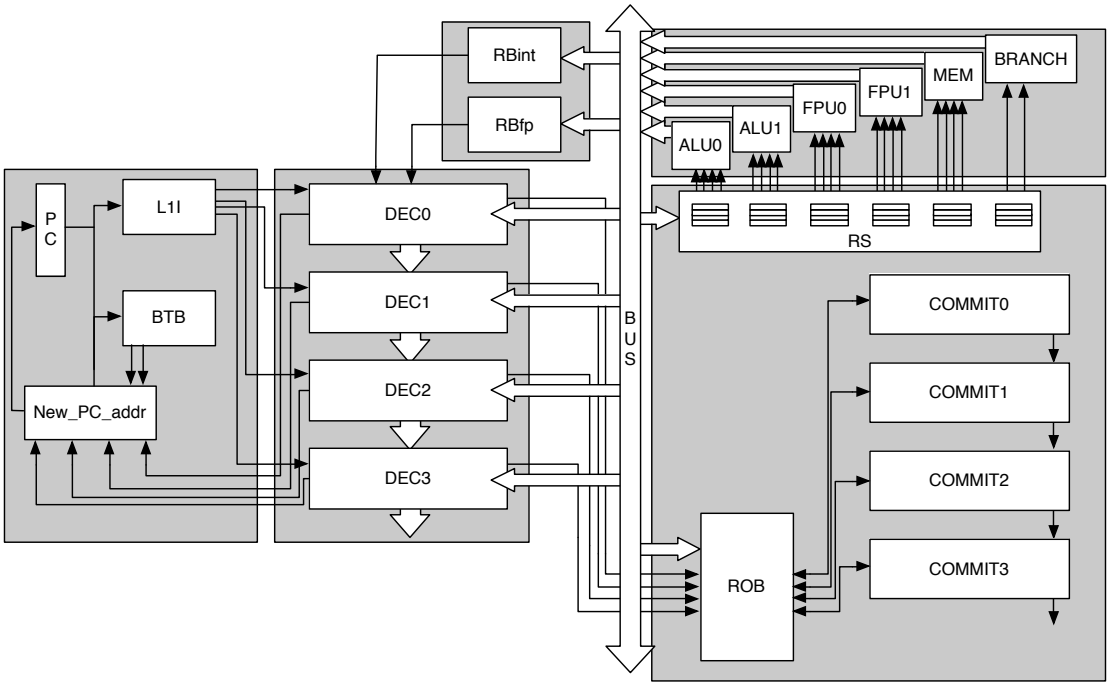

Figura 4.1: Grupos principales de la ruta de datos

que hay un salto, el módulo también realiza el cálculo de la nueva dirección de destino actualizando a su vez la entrada correspondiente de la tabla.

- El *New_PC_addr*, que calcula el nuevo contador de programa en base al resultado del BTB y de la descodificación de instrucciones.
- Por último, la cache de instrucciones, que no es más que la unidad que almacena las instrucciones a enviar a los descodificadores.

El siguiente grupo contiene únicamente las unidades de descodificación, encargadas interpretar el código binario que representan las instrucciones, y dictar el comportamiento que deben tener el resto de módulos según la instrucción entrante. Cabe añadir que esta unidad precisa de información de casi todos los componentes del procesador y envía señales de control al resto de unidades. Por tanto, este módulo es el que centraliza el control de las estructuras internas, por lo que en numerosas ocasiones es el cuello de botella del procesador. Como se ve en la figura 4.1, en nuestro diseño, hasta cuatro instrucciones pueden ser descodificadas por el procesador en paralelo.

El siguiente grupo está compuesto por los bancos de registros, que sirven para proporcionar datos de entrada y una estructura donde guardar los resultados de las instrucciones ejecutadas en la arquitectura. Se dispone de un banco de registros de enteros con 34 registros de 32 bits cada uno: el `$0` tiene un valor fijo a cero y los registros `$32` y `$33` corresponden a los registros de LO y HI que sirven para la ejecución de instrucciones que dan como resultado un valor de 64 bits (`mul.s`,

`div.s`). Adicionalmente hay un segundo banco de registros para valores de coma flotante; este banco contiene 32 registros de 32 bits. Todos los registros de cada banco están conectados directamente a las unidades descodificadoras para agilizar la interpretación de las instrucciones. Los bancos de registros también reciben información de los descodificadores (no se muestra en la figura). La interfaz de comunicación se detalla en el apéndice A.

Ahora pasamos a describir el grupo que otorga al procesador capacidad de ejecución fuera de orden, el cual estaría compuesto por el ROB, que es el responsable de almacenar las instrucciones conforme se van descodificando a la espera de que finalicen su ejecución, y las estaciones de reserva (RS) que se encargan de resolver las dependencias y facilitar las operaciones y operandos a las unidades de ejecución. El ROB guarda toda la información necesaria para ejecución de la instrucción. El comportamiento de este módulo es sencillo: añade entradas a su tabla y espera a que acaben su ejecución, que es cuando la unidad de ejecución correspondiente produce el resultado y lo envía a través del *BUS*, guardándolo en un campo de su entrada asociada en la tabla, y activando el bit de *write back* que avisa que esa instrucción ya ha finalizado, para que el módulo *COMMIT* pueda decidir que hay qué hacer con esa entrada.

Otro de los módulos indispensable para la ejecución fuera de orden es el que contiene las estaciones de reserva. Las estaciones de reserva están asociadas a cada operador, encargadas de almacenar las instrucciones con sus respectivos operandos hasta que estos sean válidos, esperando a que estos se muestren por el *BUS*, se capturen y por lo tanto el operador (ALU, FPU, *MEM* o *BRANCH*) pase a ejecutar la instrucción de la estación elegida.

Como último módulo de este grupo, encontramos el *COMMIT*, que contiene el comportamiento necesario para evaluar entradas del ROB decidir qué debe hacer con la instrucción asociada en el tabla; incluso en casos especiales, obligar al ROB a eliminar todas las entradas de la tabla. Así mismo, el *COMMIT* notifica eventos a las estaciones de reserva (confirmar escrituras) y a los bancos de registros (consolidar resultados en caso de instrucciones de almacenamiento). Nótese también que existen hasta cuatro bloques *COMMIT* conectados en serie.

A la hora de ejecutar las instrucciones disponemos de varios módulos que ejercen dicha función, entre los que se encuentra la ALU, encargada de realizar operaciones aritmético-lógicas con enteros, y la FPU, cuya función es la ejecución de instrucciones de coma flotante. Ambas unidades están encapsuladas en un envoltorio diseñado para regular el flujo de datos y señales tanto entrantes como salientes. En la figura 4.1 se contemplan dos operadores de cada clase, aunque este número puede ser cambiado en nuestro diseño. Estas unidades contienen varios suboperadores que se encuentran segmentados, lo cual hace que tengan una elevada complejidad. La unidad arítmetico-lógica incluye tres tipos distintos de sumadores-restadores pudiendo seleccionar el que deseemos a través de las opciones de configuración del procesador. También existe un operador de multiplicación, otro de división y finalmente el de operaciones lógicas o de conversión.

Cabe destacar que los registros de segmentación usados en el multiplicador son de un tamaño considerable.

En relación a la memoria (unidad *MEM*), se ha elaborado una unidad de memoria con 1024 entradas de 32 bits. Esta unidad está segmentada en dos etapas, cálculo de la dirección efectiva y acceso a memoria, con lo cual permitimos adelantar una parte de la operación que es el cálculo de la dirección en cuanto esté disponible el registro correspondiente. Esto aumenta el rendimiento dado que elimina la restricción de que estén todos los datos de la instrucción ya disponibles y libres de dependencias. También se otorga soporte a instrucciones de escritura y lectura de tres tipos que son de *byte*, de *half* y de *word*. En la etapa de cálculo de la dirección efectiva, se comprueba que la dirección obtenida es válida, lanzando una excepción en caso contrario. Esta unidad también incorpora la lógica necesaria para asegurarse de no lanzar una lectura si hay una escritura pendiente en la misma dirección de memoria (hay una dependencia entre ellas), es decir, se soporta la desambiguación de memoria, que permite la ejecución fuera de orden de lecturas y escrituras de forma segura.

Finalmente se encuentra el operador *BRANCH*, una unidad que de manera similar al resto de operadores, calcula la dirección y condición para las instrucciones de salto y al acabar envía este resultado al *BUS*.

Por último, como grupo final se encuentra el modulo *BUS* que interconecta la salida de los operadores con las estaciones de reserva y el ROB. Puesto que hay varios operadores, es necesario realizar un arbitraje mediante un algoritmo de prioridades fijas o *round-robin*. El procesador soporta hasta cuatro *BUS* en paralelo.

4.2 El flujo de datos

Para ilustrar el funcionamiento del procesador, procedemos a explicar con cierto detalle cuál sería el flujo de datos de una instrucción genérica.

La ruta de datos comienza en la cache. En ella se encuentran las instrucciones, las cuales son enviadas al descodificador en cada ciclo, utilizando los valores obtenidos desde *New_PC_addr*, es decir, usando la dirección calculada en este último módulo. Una vez se ha completado la descodificación de la instrucción, se emitirá la información necesaria para el correcto funcionamiento del resto de módulos, añadiendo la entradas en el ROB, así como actualizando los campos de los bancos de registros y añadiendo en las estaciones de reserva las instrucciones que deben pasar a ejecución en su operador correspondiente (ALU, FPU, *MEM* o *BRANCH*).

Una vez se ha finalizado la ejecución en cada uno de los operadores, estos expondrán su resultado al *BUS*, para que el resto de unidades puedan observar el resultado obtenido. Una vez las instrucciones se han completado, la fase *COM-*

MIT las extrae del ROB en el orden de programa, consolidando el resultado de su ejecución, escribiendo en el banco de registros, o cancelando todas las instrucciones en curso en caso de una especulación incorrecta o lanzamiento de una excepción. La figura 4.2 muestra un diagrama con el flujo de datos indicado.

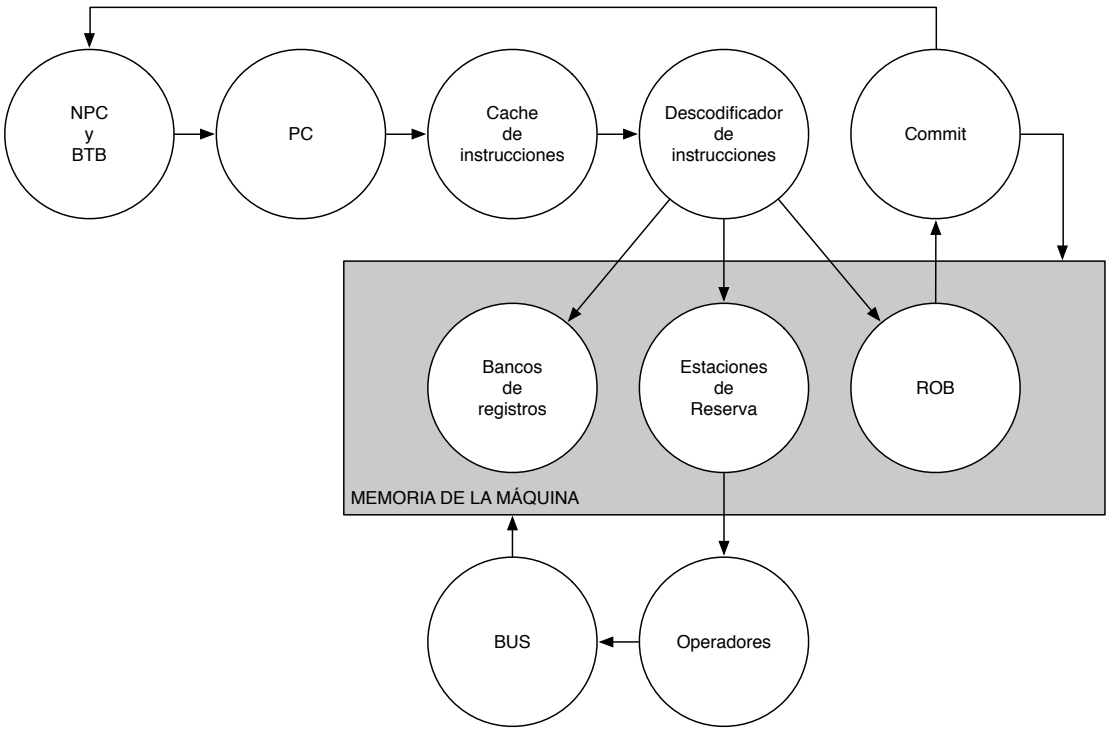

Figura 4.2: Flujo de datos general

4.3 Configuraciones del Procesador

Se ha comentado anteriormente que el procesador desarrollado es configurable, pudiendo crear instancias del mismo con diferentes especificaciones, esto se logra gracias a las sentencias `‘ifdef y ‘endif` añadidas a lo largo de todo el código. Estas se activarán o desactivarán según los `‘define` que existan en el fichero cabecera (.vh). Por ejemplo, en el siguiente código:

```
1 ‘ifdef versionA
2   $display("Procesador versión A") ;
3 ‘endif
```

Condicionamos la existencia de parte del código (línea 2), a la definición de *versionA*, en caso de que esta no esté la cabecera, el código no se activa, a efectos prácticos es como si no estuviera escrito.

Es de esta manera como se consigue esta parametrización del procesador. A continuación se muestra el código que define la primera configuración del procesador.

```
`ifdef CORE_1way_1op_1rs
`define DEC0
`define num_ROB_entries      16       // Number of ROB entries
`define log_num_ROB_entries 4         // log of number of ROB entries
`define num_free_entries_initial 4'b1111
`define BUS0
`define COMMIT0
`define RB_INT
`define RB_FP
`define ALU0
`define ALU0_RS0
`define FPU0
`define FPU0_RS0
`define FPU
`define ALU
`define FPU_RS0
`define ALU_RS0
`define CLA_level2_2stages
`define MEM0
`define MEM0_RS0
`define BRANCH0
`define BRANCH0_RS0
`endif
```

CAPÍTULO 5

Diseño de Componentes Específicos

Dado que el proyecto se ha realizado en grupo, cada miembro ha centrado sus esfuerzos en diferentes unidades. Así pues, procedemos a la explicación detallada de las mismas.

En este capítulo se muestran algunos de los principales componentes que personalmente he diseñado e implementado para después unirlos conjuntamente con el resto de los componentes diseñados por otros compañeros. En especial, se describen los operadores funcionales de enteros (ALU) y de coma flotante (FPU) que son los que van a ejecutar las operaciones del procesador para enviar el resultado al BUS.

Cabe destacar que los operadores están segmentados para poder mantener varias instrucciones en ejecución al mismo tiempo y así obtener un buen índice de instrucciones ejecutadas por ciclo (IPC). Si vemos la figura 5.1 podemos contemplar la estructura genérica de las unidades segmentadas que se describen posteriormente. La unidad se segmenta en diferentes etapas, teniendo en cada etapa un registro de segmentación con diferentes campos. Los campos a utilizar dependerán del tipo de operador. Aun así, dos campos serán comunes a todos los operadores. Este es el caso de los campos *valid* y *rob_entry*. El primero (*valid*) indica que en la etapa en cuestión hay una operación válida, mientras que el segundo (*rob_entry*) identifica la instrucción que se está ejecutando en la etapa actual. Este campo (*rob_entry*) se utilizará en la ruta de datos para identificar la operación. Las operaciones avanzarán a través del operador por medio del reloj conectado a los registros de segmentación. De hecho, el reloj (*clk*) podrá ser deshabilitado (señal *clk_enable*) en determinadas circunstancias (principalmente cuando la última etapa de segmentación tenga una operación válida y no pueda acceder al siguiente elemento de la ruta de datos, siendo este el BUS). De esta forma se evita la posible pérdida de instrucciones en la ruta.

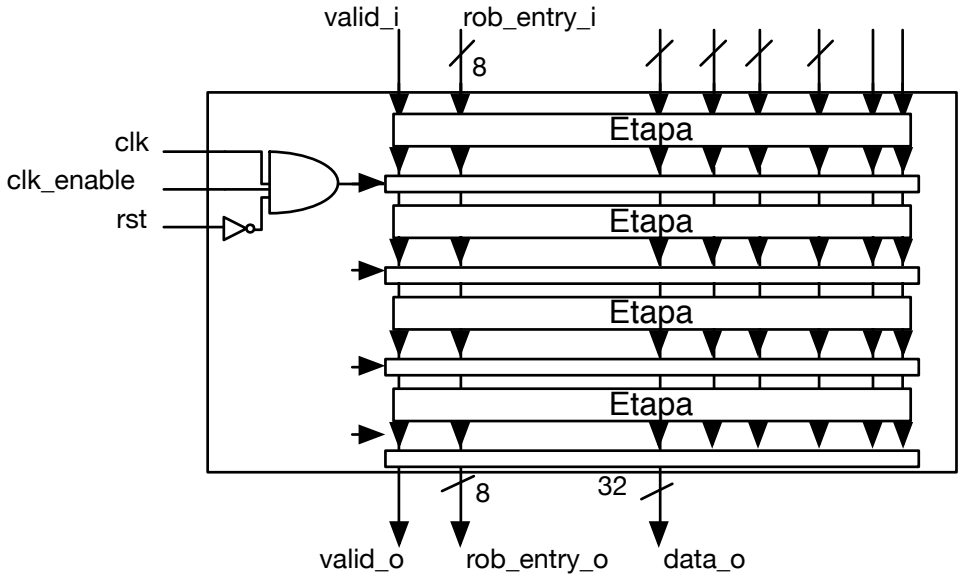

Figura 5.1: Estructura interna genérica de las unidades segmentadas

5.1 La Unidad Aritmético-Lógica (ALU)

Este elemento es uno de los componentes principales del procesador, puesto que es el que realiza las operaciones con enteros. Dicho operador, representado en la figura 5.2, incluye una lógica encargada de gestionar los siguientes aspectos:

1. Selección de la estación de reserva ($rs0_i \ldots rs3_i$) con operación válida para su ejecución, utilizando para ello un árbitro Round-Robin.

2. Multiplexor para direccionar la operación al suboperador apropiado. La unidad ALU contiene cuatro suboperadores especializados en división, multiplicación, sumas y restas, y resto de operaciones aritmético-lógicas.

3. Gestión de relojes para los suboperadores (*clk_enables*), evitando la pérdida de operaciones en caso de no poder gestionar la salida de los suboperadores.

4. Gestión de las peticiones al bus (*request_d* ... *request_others*) con los resultados de las operaciones.

5. Notificación a las estaciones de reserva para la liberación de estas (*rs0_reset* ... *rs3_reset*).

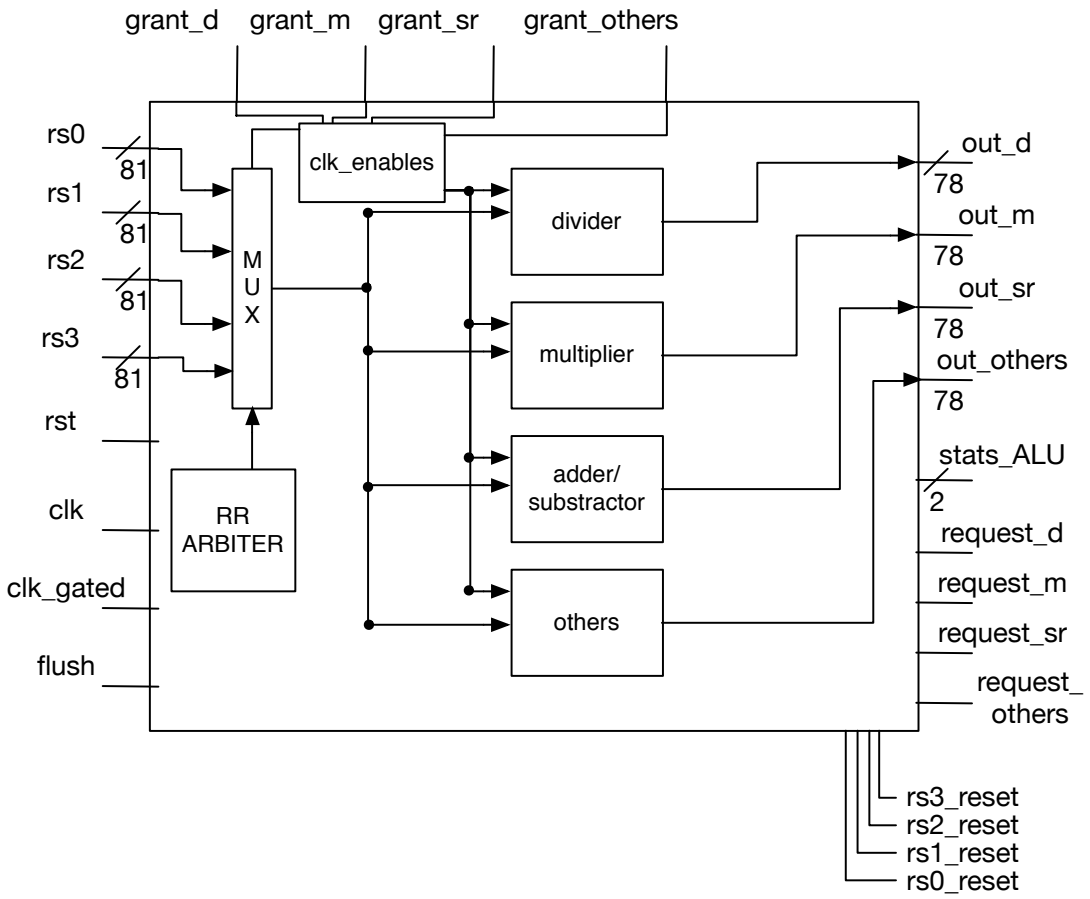

Figura 5.2: Estructura de la unidad aritmético-lógica

No nos olvidemos de que el número de estaciones de reserva del que provienen los datos depende del tipo de configuración que le hayamos definido al procesador y por lo tanto será variable pero nunca superior a cuatro.

Cada una de las estaciones de reserva nos enviará una operación a realizar con sus operandos correspondientes e indicando si dicha operación está lista para ejecutarse o aún se tienen que resolver dependencias con alguno de los operandos usados en instrucciones anteriores. Según el tipo de operación se enviarán los datos a un suboperador u otro.

Las señales *request* y *grant* sirven para solicitar el BUS (*request*) y para saber que el bus nos acepta la petición (*grant*). El bus tendrá una entrada independiente para cada suboperador. Por otra parte, la señal *flush* vacía todos los registros de esta unidad, incluidos los de los suboperadores.

Las operaciones aritméticas y lógicas con enteros resultan muy útiles ya que son las mas básicas y se utilizan en la mayoría de los programas. Esta unidad implementa una larga lista de operaciones que podemos ver separadas por suboperadores en la tabla 5.1. Todas estas operaciones se pueden consultar en [11, 12].

Nótese que los diferentes suboperadores tendrán unos tiempos de respuesta diferentes ya que se encargan de operaciones de diferentes complejidades. Así, los operadores de multiplicación y división suelen tener unos tiempos de operación de decenas de ciclos, mientas que los operadores de suma y resta tienen tiempos de respuesta menor. Así, el operador "others" tiene un tiempo de respuesta de un ciclo, debido a la sencillez de sus operaciones. Por tanto, una buena estrategia, para desacoplar operaciones complejas y así obtener buenas prestaciones, es la de implementar suboperadores especializados.

Suboperador	**Instrucción**
Divisor	`div`, `divu`
Multiplicador	`mul`
Sumador/Restador	`add`, `addu`, `addi`, `addiu`, `sub`, `subu`
Otras (lógicas y de desplazamiento)	`and`, `andi`, `or`, `ori`, `xor`, `xori`, `nor`, `slt`, `sltu`, `slti`, `sltiu`, `sll`, `sllv`, `sll16`, `srl`, `srlv`, `sra`, `srav`, `lui`

Tabla 5.1: Instrucciones que ejecuta la ALU

5.1.1. Los Sumadores-Restadores

Dentro de la ALU es donde se encuentra el suboperador de suma y resta. Lo hemos diseñado de tres formas diferentes en aras a aumentar la diversidad de posibilidades al configurar el procesador. Más adelante se detallan los resultados para comparalos debidamente y de este modo saber cuál es más rápido o cuál utiliza menos recursos.

5.1.1.1. El Sumador-Restador con CPA

La primera unidad está compuesta por bloques CPA (*Carry Propagation Adder*) o sumador con propagación serie del acarreo. Si vemos la figura 5.3, son pequeños operadores de cuatro bits, los cuales tienen en cuenta un bit de acarreo de entrada para realizar la adición y producen un bit de acarreo para el siguiente. El bloque CPA está compuesto de cuatro sumadores completos (FA) conectados en serie, propagando el acarreo generado. Este sumador completo está implementado utilizando las funciones lógicas derivadas de la suma y cálculo de acarreo a partir de tres bits a la entrada (A, B, y C).

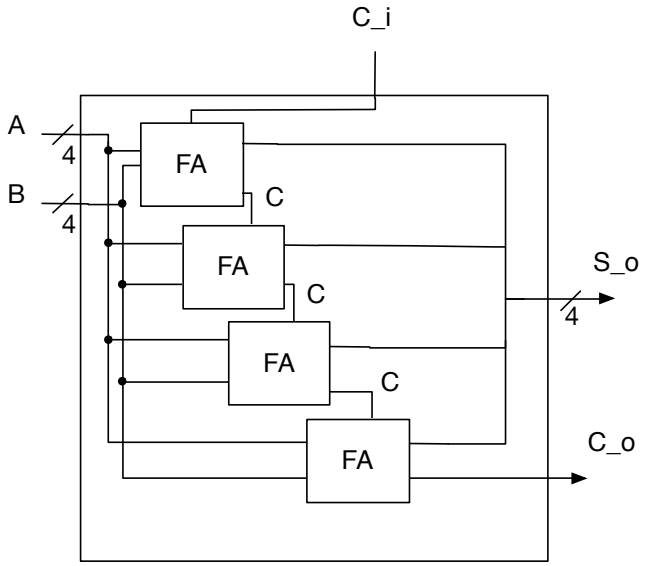

Figura 5.3: Estructura interna del bloque CPA

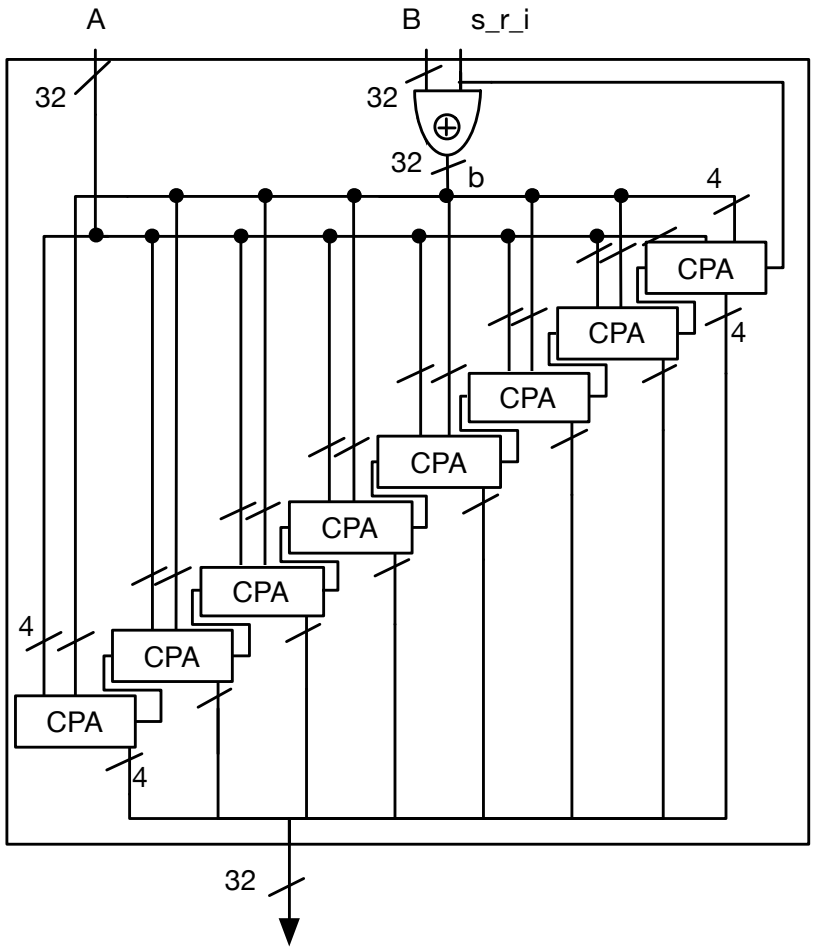

Figura 5.4: Estructura interna del sumador-restador compuesto por bloques CPA

En este caso, después de ver que los módulos CPA pueden estar enlazados propagando el acarreo de uno al otro, el primer suboperador de 32 bits que hemos implementado es el representado en la figura 5.4. Al inicio de la estructura del sumador-restador hay una puerta *or* exclusiva del operando B y el bit de operación puesto que si cambian estos bits se realiza una operación de suma o de resta ($s\_r\_i = 0$ indica suma, $s\_r\_i = 1$ indica resta, debido a que $A - B = A + Ca2(B)$).

Como vemos es un diseño simple y sencillo, pero el tiempo total de la suma es el retardo de la puerta *or* exclusiva más el tiempo de retardo de cada uno de los bloques CPA. Esto nos supone bastante tiempo para una operación de esta envergadura y se puede mejorar cambiando la estructura.

Si nos fijamos en la figura 5.5 podemos ver este mismo suboperador pero segmentado en cuatro etapas. Dicha estructura es la de uno de los distintos sumadores-restadores implementados.

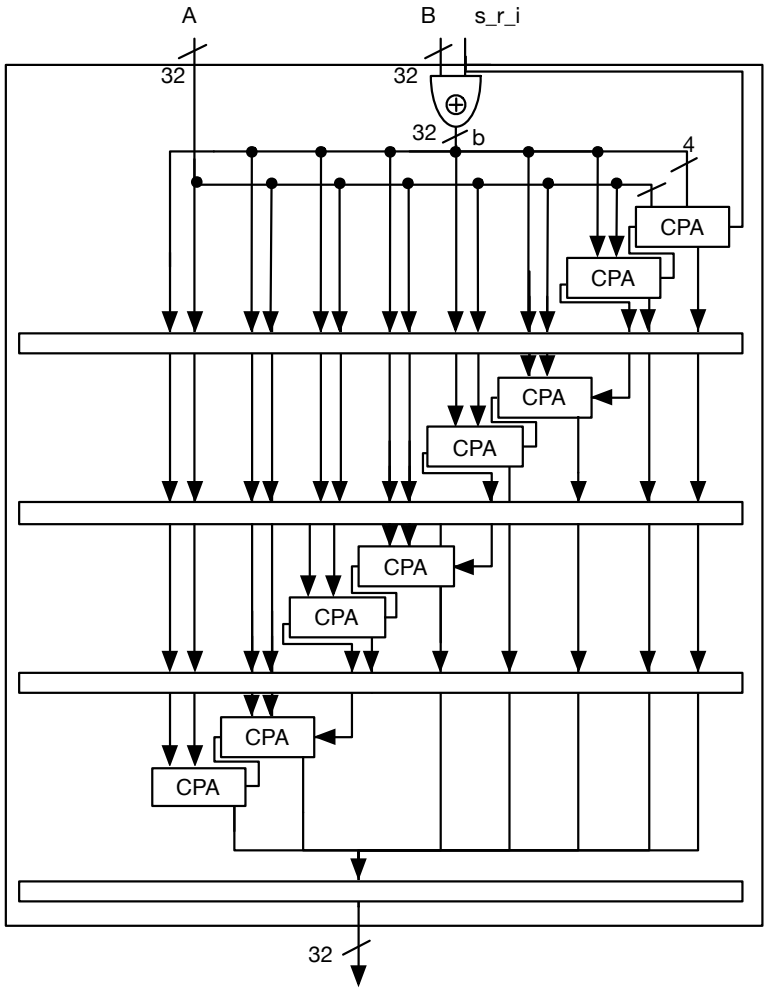

Figura 5.5: Estructura interna del sumador-restador segmentado compuesto por bloques CPA

5.1.1.2. El Sumador-Restador con CLA de un nivel

El siguiente suboperador que hemos diseñado, mejora el tiempo de retardo del anterior. Esta unidad utiliza pequeñas celdas de adición pero de otra clase, CLA (*Carry-Lookahead Adder*) o sumador con anticipación del acarreo. Estos operadores funcionan de una forma diferente ya que el acarreo se calcula de forma anticipada. Veamos la figura 5.6 para entender su funcionamiento.

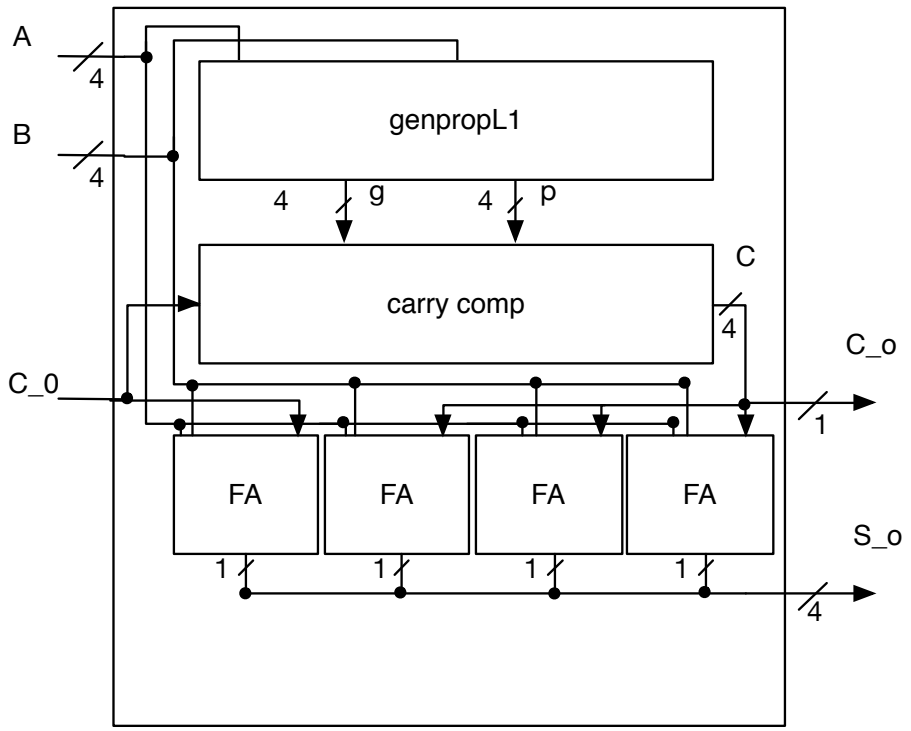

Figura 5.6: Estructura interna del bloque CLA de un nivel

En esta figura se ve un CLA de cuatro bits con sumadores completos FA (*Full Adder*). Primero usa los bloques *genpropL1* y *carry comp* que generan los acarreos para que, seguidamente, los FA funcionen en paralelo y así tener menor tiempo de retardo. El módulo *genpropL1* genera las señales g_i y p_i, denominadas de generación y propagación, respectivamente. Estas señales dependen exclusivamente de los operandos A y B. En concreto, $g_i = a_i + b_i$, mientras que $p_i = a_i \times b_i$. Por tanto, el módulo consta de cuatro puertas *AND* y cuatro puertas *OR* funcionando en paralelo.

Con estas señales, el módulo *carry comp* calcula las cuatro señales del acarreo (c_1 al c_4) utilizando la señal c_0 de entrada y desarrollando la expresión: $c_i = g_i + p_i \times c_{i-1}$. Por tanto, utiliza más puertas pero calcula los cuatro acarreos en paralelo. Una vez calculados, los cuatro sumadores completos se ejecutan en paralelo. Por lo tanto, si usamos varias unidades CLA enlazadas entre sí, tal y

como se muestra en la figura 5.7, obtenemos una unidad más eficiente que la anterior compuesta por bloques CPA.

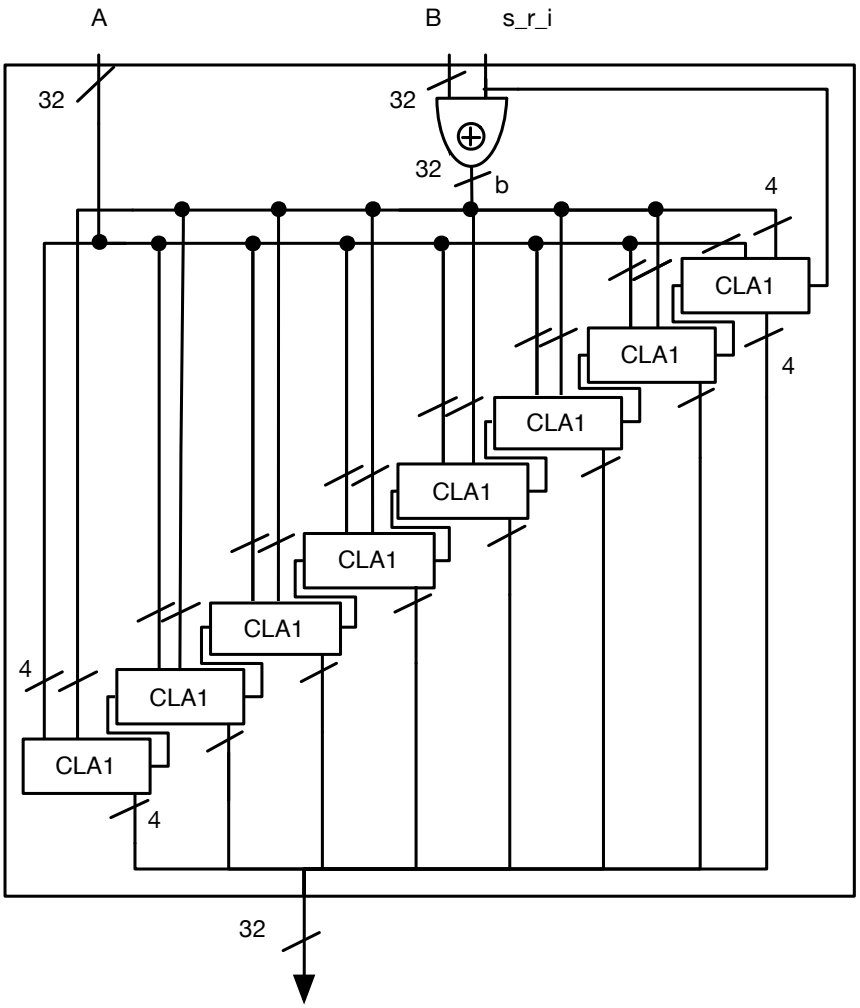

Figura 5.7: Estructura interna del sumador-restador compuesto por bloques CLA de un nivel

Por último, hemos segmentado este suboperador con varios registros de segmentación como puede verse en la figura 5.8 para que la operación se pueda realizar en cuatro ciclos de reloj. Nótese que se podría reducir el número de etapas de segmentación agrupando más operadores en una etapa.

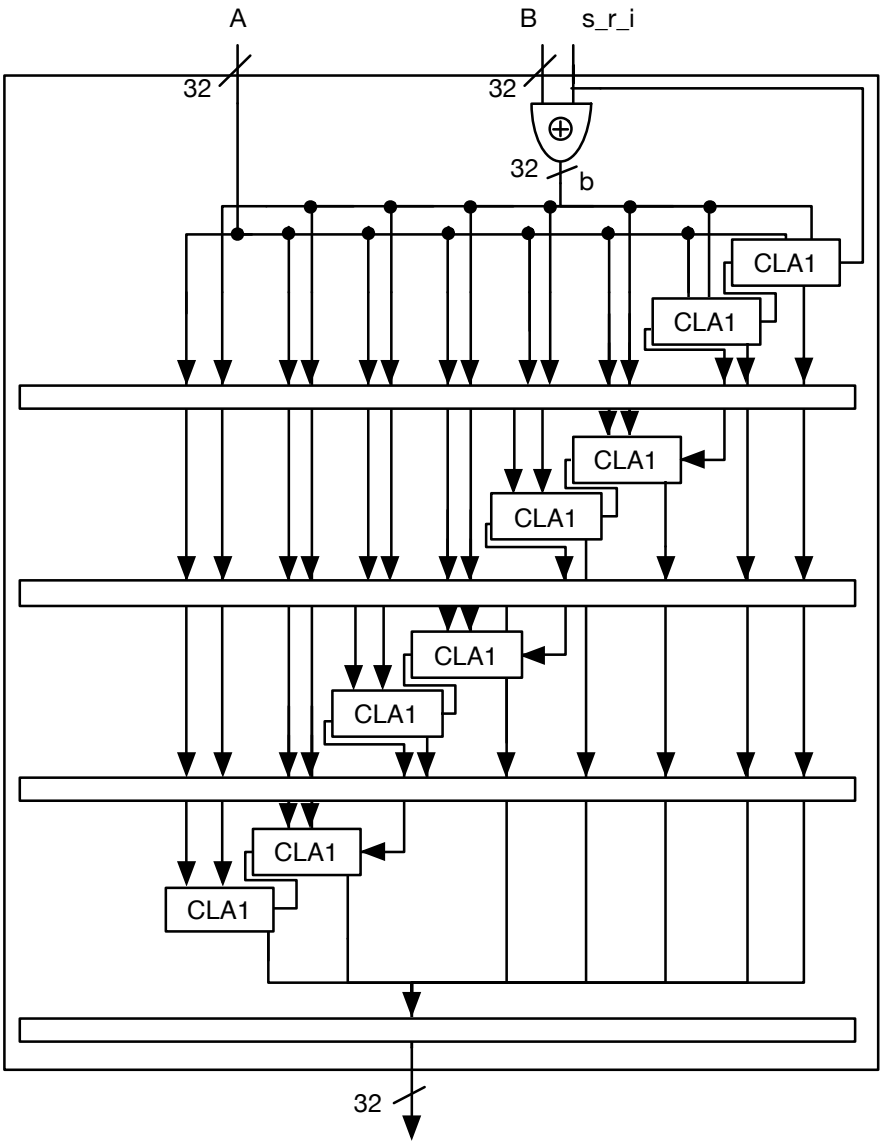

Figura 5.8: Estructura interna del sumador-restador segmentado compuesto por bloques CLA de un nivel

5.1.1.3. El Sumador-Restador con CLA de dos niveles

El próximo sumador es una optimización del anterior. Se basa en una mejora de la celda CLA representada en la figura 5.9. El bloque citado en este caso tiene dieciséis entradas por cada operando. Esta permuta se ha realizado porque en vez de tener un solo bloque que se encarga de calcular g_i o p_i disponemos de cuatro. Además, con las señales que generan estos elementos, un módulo que representa el segundo nivel las utiliza para calcular las señales G_i y P_i de segundo nivel, a partir de las señales g_i y p_i. En concreto y asumiendo: $i = k/4$

$$G_k = g_{i+3} + (p_{i+3} \times g_{i+2}) + (p_{i+3} \times p_{i+2} \times g_{i+1}) + (p_{i+3} \times p_{i+2} \times p_{i+1} \times g_i)$$

$$P_k = p_{i+3} \times p_{i+2} \times p_{i+1} \times p_i$$

Tanto los g_i y p_i como los recientemente generados G y P, son usados para calcular los acarreos pero esta vez con cuatro unidades en paralelo para ello.

Llegados a este punto disponemos de todos los acarreos calculados gracias a la estructura anterior, que solo se basa en replicar unidades utilizando más recursos y realizar las mismas operaciones pero actuando en este caso todos a la vez. Posteriormente instanciamos tantos FA como entradas por operando para calcular cada bit del resultado de la operación. Véase la figura 5.10.

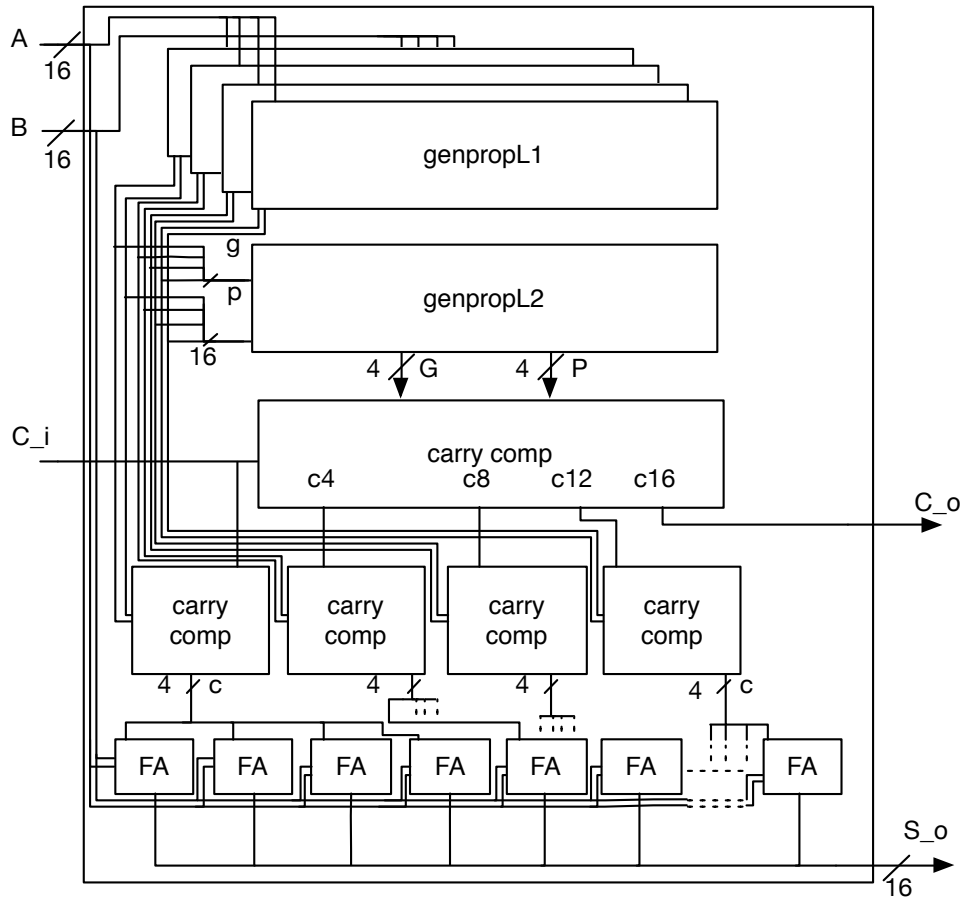

Figura 5.9: Estructura interna del bloque *CLA* de dos niveles

Por último, en la figura 5.11 se puede ver de qué forma se ha segmentado esta unidad con registros, ya que la utilización de dos bloques en vez de ocho nos permite gestionarlo con un grano más grueso.

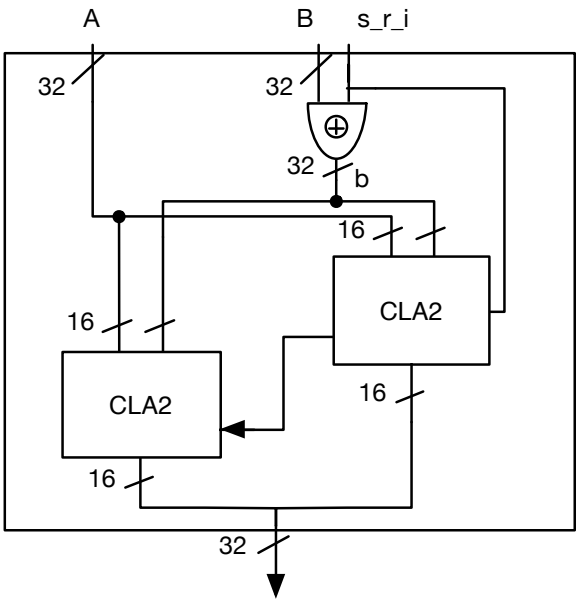

Figura 5.10: Estructura interna del sumador-restador compuesto por bloques CLA de dos niveles

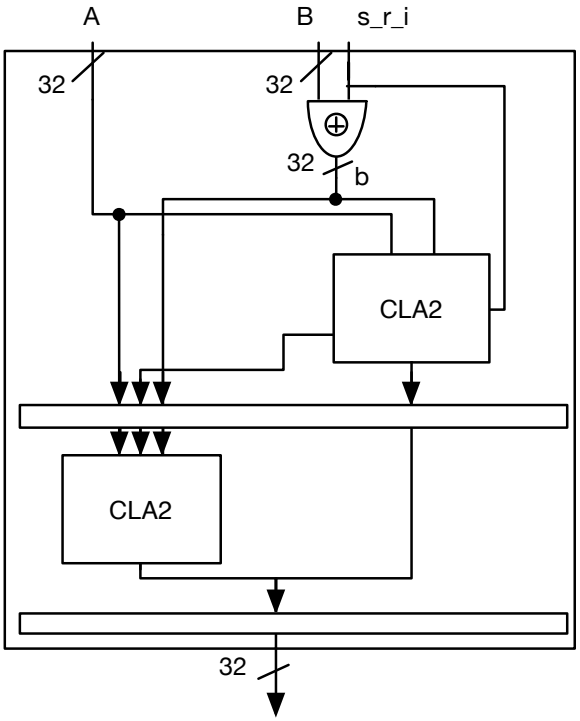

Figura 5.11: Estructura interna del sumador-restador segmentado compuesto por bloques CLA de dos niveles

5.1.2. El Multiplicador

Esta unidad es la más compleja de la ALU puesto que tiene instanciados gran cantidad de módulos en ella. En la figura 5.12 se representa la estructura interna resumida de este módulo.

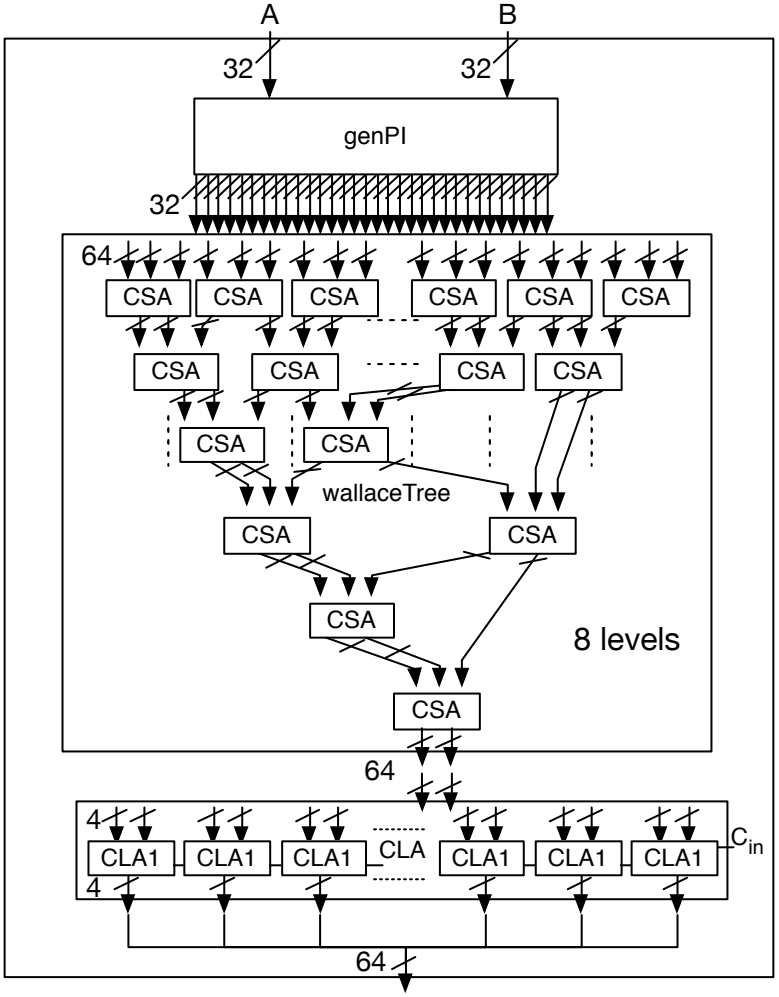

Figura 5.12: Estructura interna del multiplicador

Si pensamos en cómo hacemos una multiplicación común, primero generamos los productos intermedios para luego realizar la operación de adición y obtener el resultado de la multiplicación. Esto es justo lo que hace este operador.

El primer módulo que contiene es el que genera los productos intermedios *genPI*, que para ser exactos genera 32 productos intermedios a partir de dos operandos de 32 bits. Este produce 32 salidas de 32 bits (1 024 bits en total) que son inyectados a un árbol de Wallace (*wallaceTree*). Dicho bloque no es más que una serie de sumadores CSA (*Carry Save Adder*), que recogen tres operandos (realmente son dos y el acarreo) y devuelven el resultado en dos (resultado y acarreo).

Como dice su nombre, calculan el acarreo y lo envían al siguiente. Puesto que están enlazados en forma de árbol, de ahí el nombre del módulo que los contiene.

Llegado a esta parte de la operación, nótese que el árbol de Wallace no devuelve el resultado final, sino dos operandos que hay que sumar y para ello usamos un suboperador compuesto por varios bloques de sumadores con anticipación del acarreo CLA (*Carry-Lookahead Adder*). Este es mucho mas rápido que los otros ya que primero realiza el cálculo de los acarreos y luego la suma de todos los bits en paralelo.

El presente diseño es el de un multiplicador muy optimizado el cual se ha segmentado al igual que los otros suboperadores. Como se muestra en la figura 5.13, se han añadido tres fases de segmentación dentro del árbol de Wallace y dos dentro del sumador CLA de sesenta y cuatro bits.

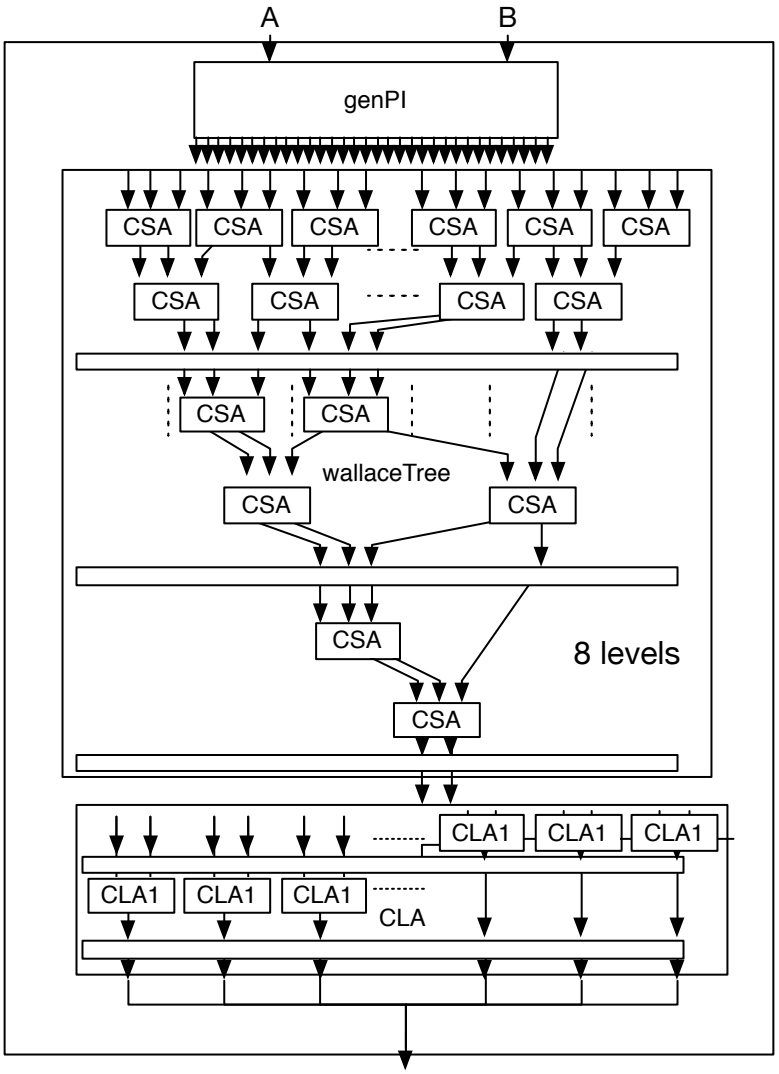

Figura 5.13: Estructura interna del multiplicador segmentado

Cabe destacar los registros de esta unidad alcanzan un tamaño considerable de hasta casi 1 000 bits que luego se van a ver reflejados en recursos de la FPGA.

5.1.3. El Divisor

El siguiente suboperador es un divisor sin restauración. La restauración surge cuando después de restar el divisor del dividendo queremos comprobar el signo del resultado y tenemos que volver a sumar el dividendo y el divisor (restauración). Con esta estructura es posible combinar la restauración y la sustracción que la sigue en una sola operación: la suma del divisor desplazado. Este diseño se puede consultar en [3].

Dentro de la ALU es donde se encuentra el divisor, entre otros suboperadores. La figura 5.14 muestra su estructura interna, y como podemos apreciar forma una especie de matriz de celdas $(n+1) \times n$, donde n es el número de bits del divisor. En estas celdas los distintos bits de los operadores se suman o restan uno a uno, para seguidamente pasar al siguiente nivel.

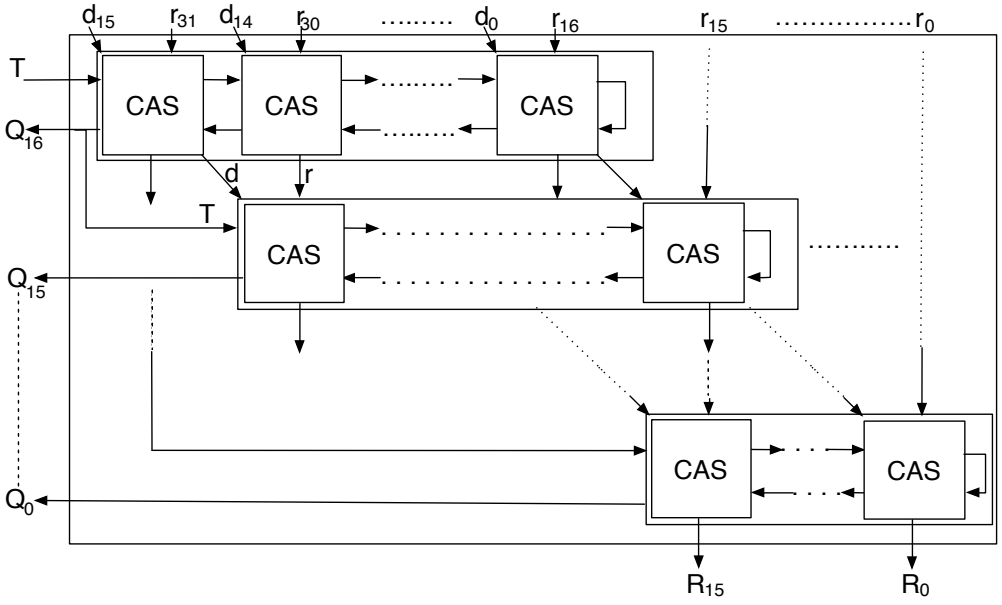

Figura 5.14: Estructura interna del divisor

Según se puede ver en la figura 5.15 las celdas de suma-resta CAS (*Controlled Adder Subtractor*) o sumador-restador controlado, tienen una señal que indica la operación a realizar con las entradas. También disponen de otro bit que es el acarreo a tener en cuenta para calcular en el posterior bloque. En cada nivel se calcula una parte del cociente y en el último obtenemos íntegramente el resto de la operación.

De este modo queda un divisor por puertas lógicas bastante eficiente y optimizado. Con tal de mejorar el tiempo de ciclo en nuestra ruta de datos, la unidad

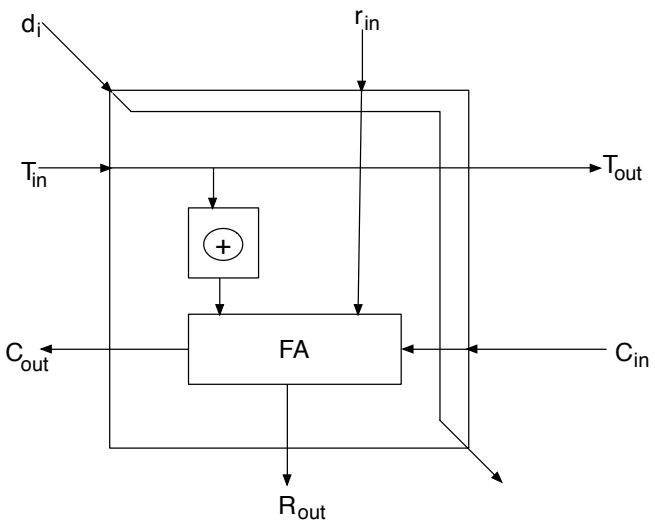

Figura 5.15: Estructura interna de las celdas de suma-resta del divisor (CAS)

se encuentra segmentada. En la figura 5.16 se muestran los tres registros de segmentación que se encuentran en dicho módulo. Esto quiere decir que tardará tres ciclos en hacer una operación de división desde el momento en que le llegan los datos al operador funcional de enteros, pero se ha bajado el tiempo de ciclo puesto que ahora un ciclo supone hacer la tercera parte de la operación y no toda.

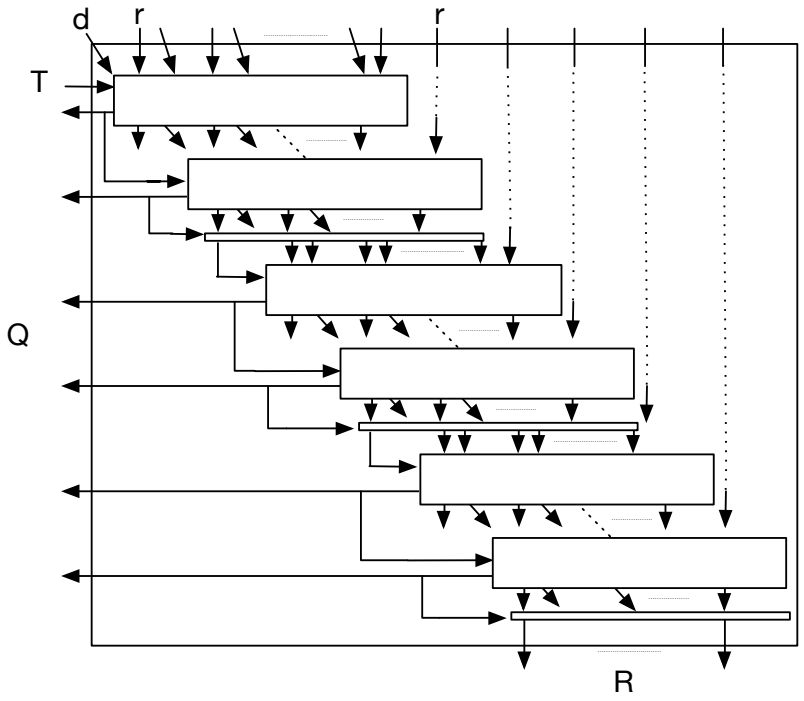

Figura 5.16: Estructura del divisor segmentado

Es importante recalcar que con el diseño inicial no se soportan operandos con signo negativo. Por tanto, para soportar divisiones con signo, hemos implementado dos módulos adicionales sencillos, los cuales implementan el preprocesamiento del signo antes de la división y el postprocesamiento del signo justo después de la división. Véase la figura 5.17.

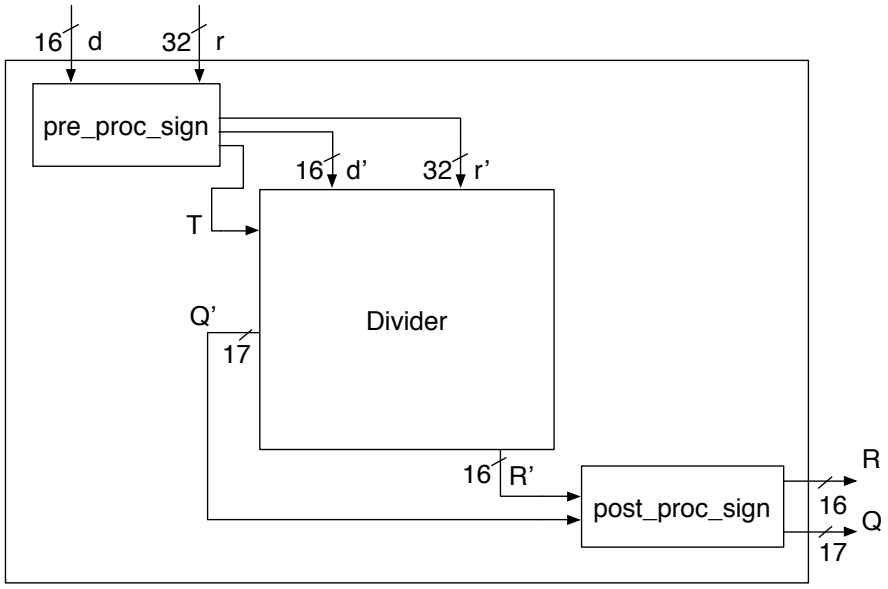

Figura 5.17: Estructura del divisor completo

5.1.4. El Suboperador Lógico y de Desplazamiento

El operador que resta explicar de entre los que se encuentran en la ALU es conocido como el suboperador *others*, el cual ofrece la posibilidad de efectuar operaciones lógicas y de desplazamiento de un solo ciclo. Estas operaciones se realizan bit a bit y son bastante ligeras a la hora de ejecutarse en cuanto a tiempo de cómputo se refiere.

En la figura 5.18 podemos contemplar la estructura interna de esta unidad, la cual consiste en un comparador que, en función de los bits de la operación que se le suministra, efectúa una operación u otra y luego guarda el resultado en un registro de segmentación situado en la parte final. Este módulo implementa una gran variedad de instrucciones y por tanto, esta lógica va a ocupar unos recursos en nuestra FPGA que después veremos detalladamente.

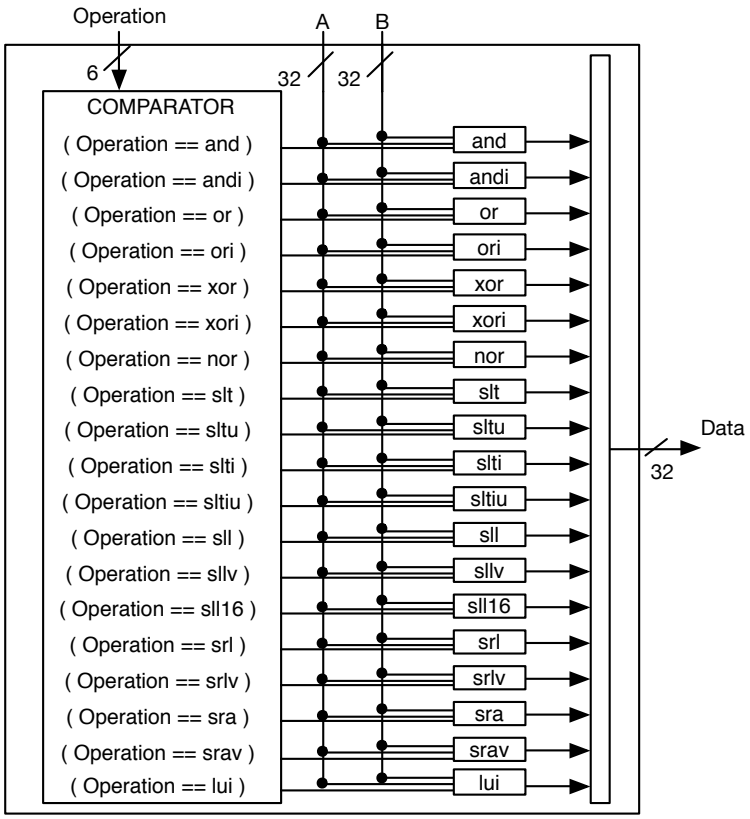

Figura 5.18: Estructura interna del suboperador lógico y de desplazamiento

5.2 La Unidad de Coma Flotante (FPU)

Por otra parte se encuentra esta unidad encargada de realizar operaciones con operandos en formato de coma flotante. Tal como se define en el apartado 2.1 el citado formato sirve para cálculos definidos sobre números reales (*float* y *double* en alto nivel) que no están soportados por la ALU. Nuestra versión tiene soporte para instrucciones de coma flotante con simple precisión. Por lo tanto, la longitud de los operandos y el resultado es de 32 bits.

Como se puede contemplar en la figura 5.19 la FPU está contenida en un envoltorio que se usa para gestionar los buses de entrada provenientes de las estaciones de reserva y la salida de resultados al bus de datos principal del procesador. Cabe destacar que se seleccionan los datos de entrada de una estación de reserva u otra según un árbitro *Round Robin*. Este mismo sistema también se ha usado en la ALU pero ahora solo tenemos una salida de la unidad funcional de coma flotante. Por otra parte, la señal *flush*, igual que en la ALU, vacía todos los registros de esta unidad.

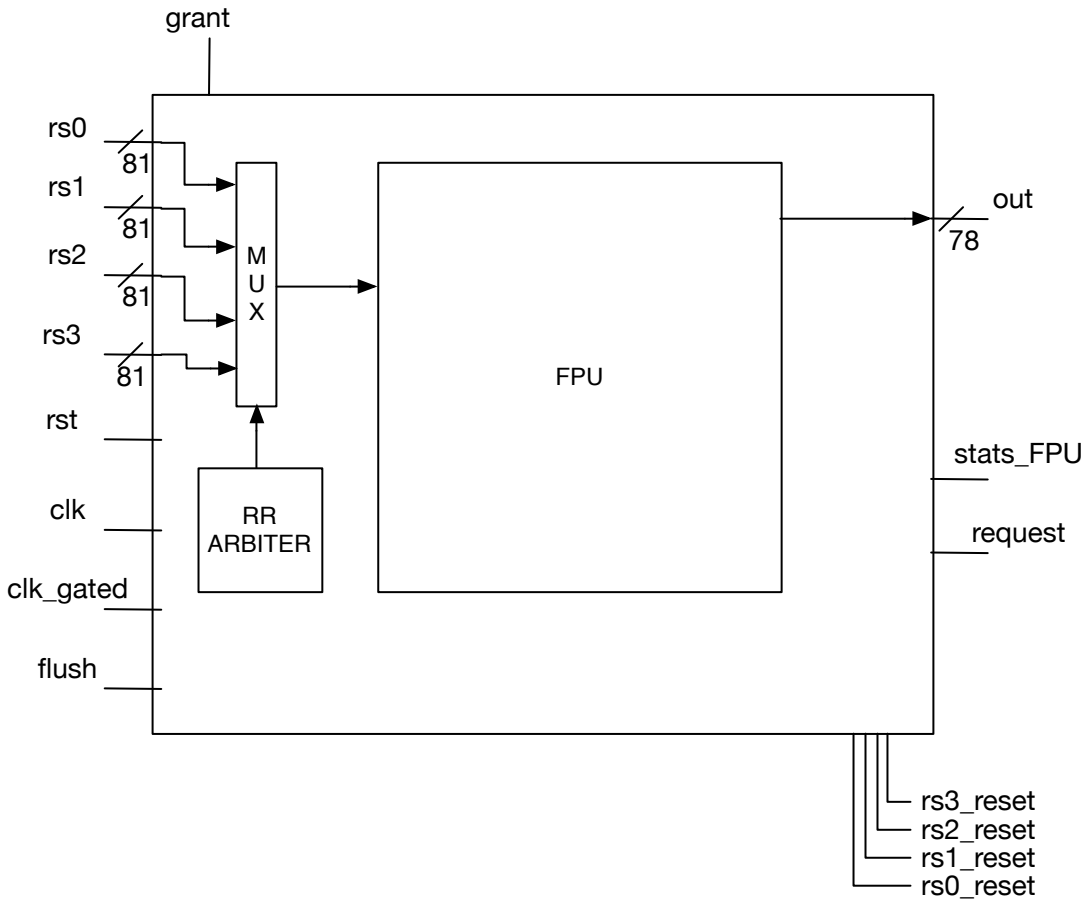

Figura 5.19: Estructura de envoltorio en la unidad de coma flotante

La estructura interna es modular, como se puede apreciar en la figura 5.20, que nos facilita la segmentación utilizando registros, de tal modo que dividiremos la ruta de datos en diferentes etapas. Las entradas que se le suministran provienen de las distintas estaciones de reserva. Están expresadas en binario con una longitud de 6 bits para las operaciones y representadas en formato IEEE 754 mediante 32 bits para los operandos.

Todos los módulos contenidos en cada fase funcionarán al mismo tiempo, pero cada uno por separado en operaciones distintas. Por ello recibirán algunas de las señales de entrada a la FPU en el caso de la primera fase o desde el registro de segmentación inmediatamente anterior en otro caso y enviarán los datos resultantes al siguiente registro.

En la tabla 5.2 encontramos las distintas instrucciones que puede ejecutar esta unidad. Para consultar la semántica de estas, podemos recurrir a [11, 12].

Dado que las operaciones en coma flotante presentan casos especiales no contemplados en aritmética entera, como infinitos, la excepción a la codificación para

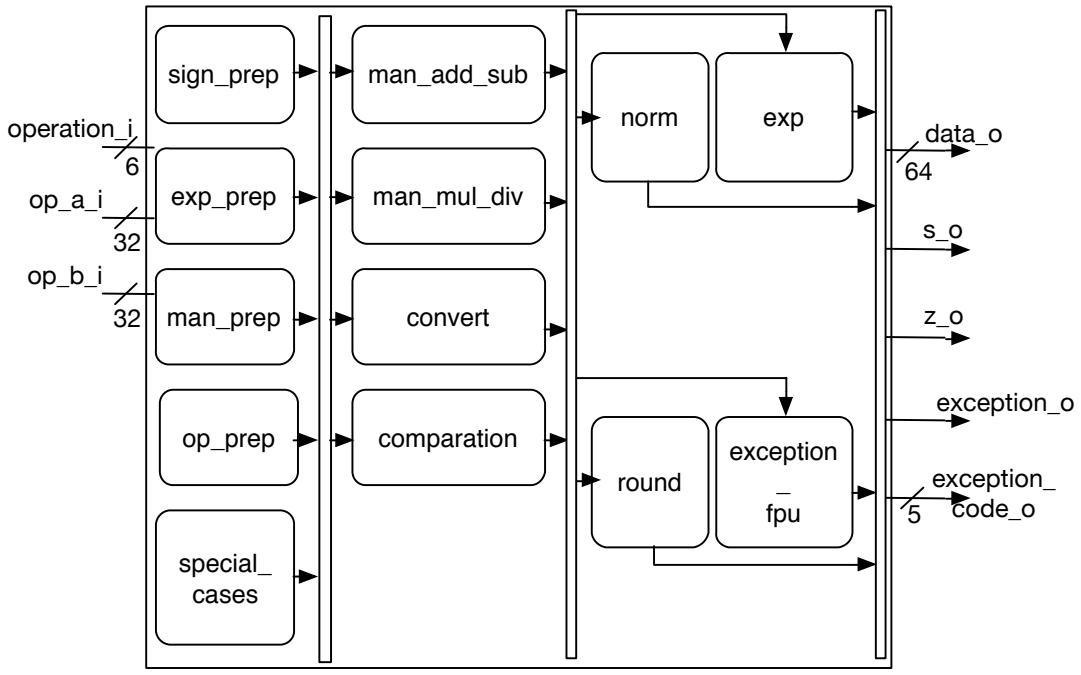

Figura 5.20: Estructura de la unidad de coma flotante

Tipo	Instrucción
Aritméticas en coma flotante	`abs.fmt`, `add.fmt`, `c.cond.fmt`, `div.fmt`, `mul.fmt`, `neg.fmt`, `sub.fmt`
Conversión en coma flotante	`cvt.s.fmt`, `cvt.w.fmt`

Tabla 5.2: Instrucciones MIPS32 que ejecuta la FPU

el cero, no es un número (*NaN*), división por cero, etc., se ha diseñado un elemento en esta unidad que analiza este tipo de casos y avisa a los otros módulos para que sepan en qué tipo de situación se encuentran. En el Apéndice B se enumeran en una tabla cada caso y su código correspondiente.

En aras a explicar la funcionalidad de esta unidad funcional, podemos ver la ruta por la que circulan los datos separada según el tipo de operación que se esté ejecutando:

- Operaciones de suma, resta, multiplicación o división.
- Operaciones de comparación.
- Operaciones de conversión, negación y valor absoluto.

5.2.1. Suma, Resta, Multiplicación o División

Inicialmente, suponiendo que es una ejecución de suma o resta, en la primera fase se calcula el signo del resultado en el módulo *sign_prep*, *exp_prep* se encarga de seleccionar el mayor exponente que será el resultante y *man_prep* de separar las mantisas del resto y realizar la extensión las mismas. La unidad *op_prep* es la encargada de evaluar la operación final en función de los casos mostrados en la tabla 5.3.

Operación	a_signo xor b_signo	a>b	Resultado	Signo del resultado
a+b	0	X	a+b	(+)
(−a)+(−b)	0	X	a+b	(−)
a+(−b)	1	Sí	a−b	(+)
a+(−b)	1	No	b−a	(−)
(−a)+b	1	Sí	a−b	(−)
(−a)+b	1	No	b−a	(+)

Operación	a_signo xor b_signo	a>b	Resultado	Signo del resultado
(−a)−b	1	X	a+b	(−)
a−(−b)	1	X	a+b	(+)
(−a)−(−b)	0	Sí	a−b	(−)
(−a)−(−b)	0	No	b−a	(+)
a−b	1	Sí	a−b	(+)
a−b	1	No	b−a	(−)

Tabla 5.3: Cálculo del signo según las operaciones de suma o resta en la FPU

Seguidamente *man_add_sub* realiza la suma o resta de mantisas y en la última etapa *norm* la normaliza. Después, *round* efectúa el redondeo al par más próximo obteniendo así la mantisa del resultado mientras *exp* comprueba si se encuentra en un caso especial y muestra el exponente final. El módulo *exception_fpu* es el que dice si la ejecución es un caso de excepción y el código asociado.

Si ejecutamos una operación de multiplicación o división, el bloque *op_prep* en este caso no tendría función alguna y en vez de realizarse una suma o resta de mantisas en *man_add_sub* la unidad *man_mul_div* es la que se encarga de multiplicarlas o dividirlas. Así pues, toda la ruta de datos restante efectúa las mismas funciones pero teniendo en cuenta la operación actual.

En la figura 5.21 podemos ver el bloque que normaliza las mantisas ya sea en una operación de suma, resta, multiplicación o división. Inicialmente se comprueba si se encuentra en formato normalizado ($1'$____), si no es así, se desplaza tantas posiciones como hagan falta. Después se comprueba según el tipo de operación si se han sobrepasado ciertos valores y se obtiene la mantisa resultante.

También hay un elemento que detectará si se produce una excepción dentro de esta unidad para avisar al módulo *exception*.

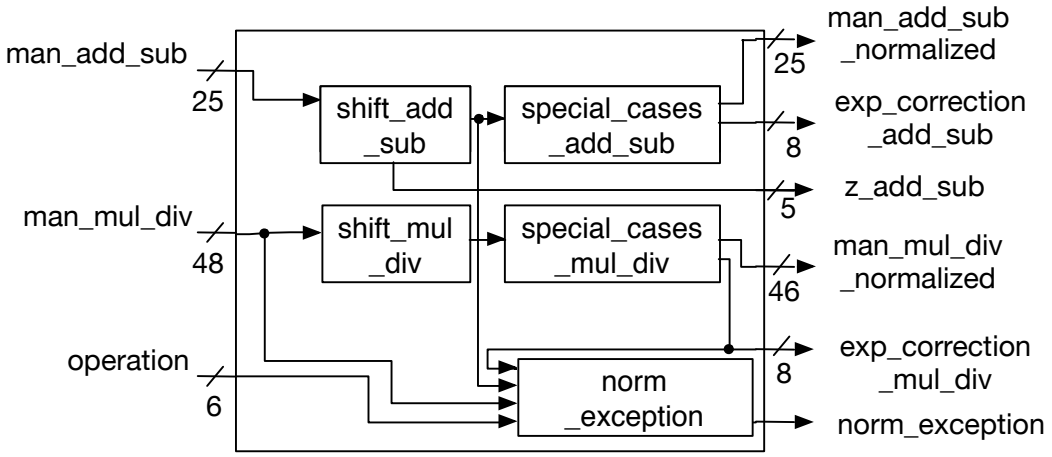

Figura 5.21: Estructura del bloque de normalización

Por otro lado, en la figura 5.22 se muestra la unidad de redondeo. Esta obtiene las mantisas de los dos operandos y las normalizadas provenientes del módulo *norm*. Cuando se está ejecutando una instrucción de multiplicación o división, inicialmente se redondea al par mas próximo. El módulo *mantissa*, como su nombre indica, calcula la mantisa del resultado teniendo en cuenta en que caso especial (si es el caso) se encuentra. Esta unidad de redondeo también contiene un elemento que indica si se ha producido una excepción durante el cálculo de la mantisa y si se da el caso se le notifica al módulo *exception_fpu*.

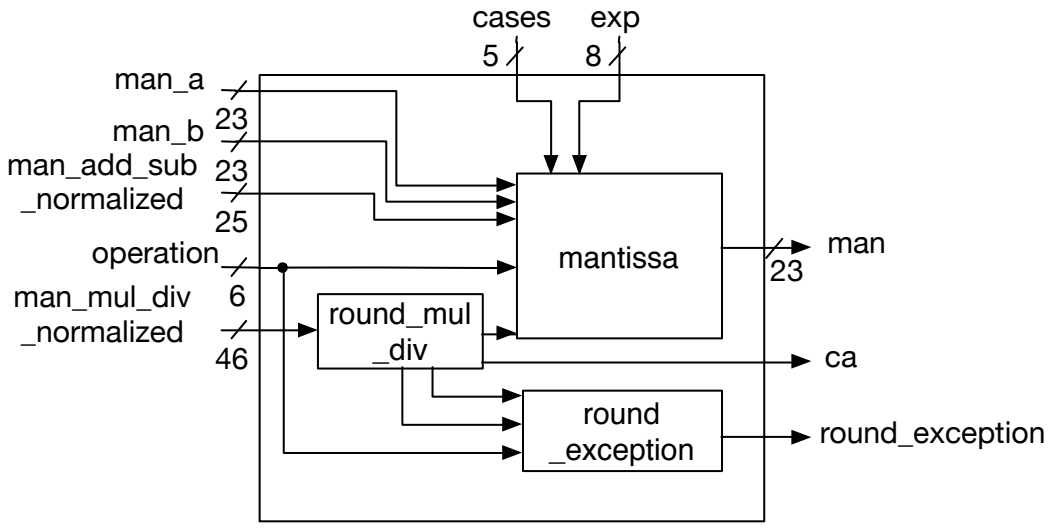

Figura 5.22: Estructura del bloque de redondeo

5.2.2. Comparación

El primer paso consiste en obtener el signo en *sign_prep*, posteriormente *comparation* realiza las comparaciones oportunas y envía la señal resultante.

Como podemos apreciar, en este tipo de operaciones participan pocas unidades en comparación a las anteriores pero no por actuar menos unidades quiere decir que van a tener un menor tiempo de cómputo ya que la FPU se encuentra segmentada en tres etapas y todas las operaciones deben pasar por ellas.

5.2.3. Conversión, Negación y Valor absoluto

Una vez lanzamos una conversión, el primer paso que se efectúa es obtener el signo en el módulo *sign_prep*. A continuación el módulo *convert*, tal y como indica su nombre, convierte el dato del formato que se le proporciona al que se le especifica en el código de operación. Después de todo, en la unidad *exception_fpu* se evalúa si el valor a convertir era un *NaN* o un infinito y, si es así, se envía una excepción de operación inválida.

Por otro lado, si la función es de negación o valor absoluto, solo actuará el bloque *sign_prep* que se encarga de invertir el bit de signo.

CAPÍTULO 6

Verificación y Resultados

En este capítulo se presentan, en forma de tablas y gráficas, los resultados obtenidos al ejecutar código ensamblador sobre el núcleo desarrollado, así como datos sobre la síntesis de los módulos con su correspondiente relación en cuanto a uso de elementos internos en la FPGA.

6.1 Resultados de Ejecución

A la hora de evaluar el sistema desarrollado, utilizamos varios programas de prueba, donde cada uno evalúa una función específica del procesador. Una vez comprobados los comportamientos básicos, pasamos a pruebas mas exhaustivas. A continuación se muestra en la tabla 6.1, a modo de visión general, algunas de las pruebas más interesantes, para dar el lector una idea de como se evaluó el correcto funcionamiento del núcleo.

Prueba	**Módulos relacionados**
Insertar más instrucciones que entradas hay en el ROB	ROB
Código con instrucciones de saltos enlazadas entre sí	BRANCH
Batería de instrucciones aritmético-lógicas	ALU
Batería de instrucciones con números en coma flotante	FPU
Batería de instrucciones de carga y almacenamiento	RB - MEM

Tabla 6.1: Visión general de las pruebas básicas realizadas

En la tabla 6.2, se puede observar el número de componentes que contiene cada configuración del procesador. Estas se diferencian por el número de vías, que es la cantidad de descodificadores, buses y bloques *COMMIT* que contiene. También se diferencian por el número de operadores ALU, FPU, *MEM* y *BRANCH* que son el número de recursos que atienden a las instrucciones de cada uno de

estos tipos. Por otro lado, variamos el número de entradas de la cola de reordenación.

Conf	Núm vias (DEC, BUS,COMMIT)	Núm ALU y FPU	Núm MEM y BRANCH	Núm RS por op	Núm entr ROB
A	1	1 1	1 1	1	16
B	1	1 1	1 1	4	32
C	2	1 1	1 1	2	16
D	2	1 1	1 1	4	32
E	2	2 2	1 1	2	16
F	2	2 2	1 1	4	32
G	4	2 2	1 1	4	32

Tabla 6.2: Número de componentes que incluye cada configuración del núcleo

6.1.1. Pruebas Exhaustivas

Una vez finalizadas las pruebas sobre el funcionamiento básico, se pasó a desarrollar programas en ensamblador que unieran varias pruebas en una sola, aumentando así la carga computacional del procesador en aras a depurar posibles fallos que no se hubieran detectado en las pruebas iniciales. Para ello se utilizan dos programas, el *axpy* y el cálculo del número pi. El código generado por estos dos programas se encuentra listado en el apéndice C.

El primer programa (*axpy*) consiste en un bucle que suma dos vectores, donde uno de ellos se multiplica por una constante. La ecuación es la siguiente:

$$z = a \cdot x + y$$

Por cada configuración de procesador se ha lanzado el programa y se ha mostrado tanto el tiempo de ejecución en ciclos, como el número de instrucciones ejecutadas. En la figura 6.1 se aprecia como la configuración con más componentes es la que mejores resultados obtiene llegando a un IPC de $1,32$. Así mismo, se observa una leve tendencia a incrementar las prestaciones al mismo tiempo que el procesador incluye mas recursos y unidades de ejecución. De hecho, en el extremo inferior, el procesador A obtiene un IPC de $0,70$. No obstante, se observa como la configuración E tiene unas prestaciones inferiores. La justificación a este comportamiento radica en que la configuración E, solamente dispone de dos estaciones de reserva por operador, mientras que las configuraciones D y F tienen cuatro. Esto hace que el procesador E sea menos eficiente a la hora de resolver dependencias. De hecho, obtiene prestaciones similares a la configuración C, la cual tiene también dos estaciones por operador. Es de observar asimismo la configuración B, que aunque tenga cuatro estaciones de reserva por operador, obtiene

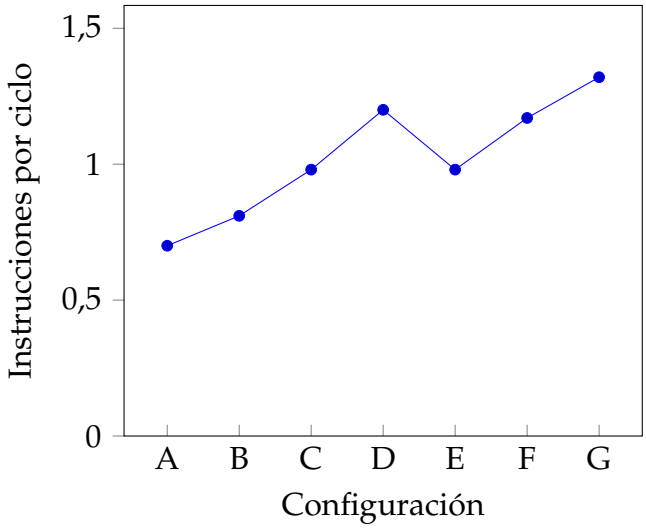

Figura 6.1: Gráficas con los resultados de ejecución en instrucciones por ciclo de las distintas configuraciones del núcleo con el bucle *axpy*.

unas prestaciones muy bajas. El motivo es la limitación en el número de vías (una solamente) lo que hace que el cuello de botella esté en la descodificación.

En cuanto al cálculo de π, se ha implementado el algoritmo de la serie de Nilakantha [16] el cual aproxima su valor mediante un bucle. El código está disponible en el apéndice C. El número de iteraciones del bucle se ha fijado en 100.

En la gráfica de la figura 6.2 se aprecia en primer lugar la misma tendencia. A más recursos mejores prestaciones. Ahora bien, el comportamiento relativo de las prestaciones es distinto. El procesador B se comporta igual que los procesadores D y E. En este caso, B dispone de 32 entradas en el ROB, D también, y E, aunque tiene solo 16, dispone de dos operadores de coma flotante. Estos procesadores funcionan mejor que la configuración C. El motivo es la falta de suficientes entradas a el ROB, solo 16. Aunque la configuración E tiene también 16 entradas, suple esta deficiencia al tener mas operadores y por tanto mas estaciones de reserva para solucionar conflictos. Nótese también que en este código que explota la aritmética en coma flotante, el número de operadores máximo es dos.

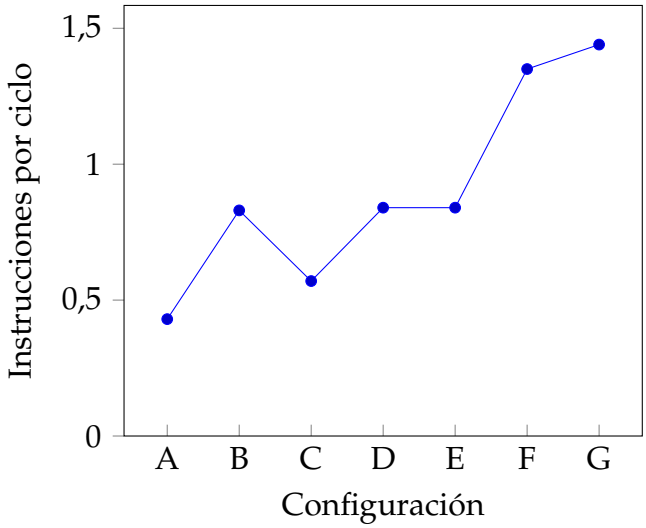

Figura 6.2: Resultados de ejecución en instrucciones por ciclo de las distintas configuraciones del núcleo con el programa π.

6.2 Síntesis de las Configuraciones del Núcleo

Después de implementar los diferentes componentes del núcleo, realizamos una estudio de los diferentes recursos en función de los recursos necesarios para albergar tanto la lógica como los registros de cada una de las unidades en nuestra FPGA.

La cantidad de recursos lógicos necesarios se encuentra expresada en tablas LUT (*Lookup Table*), o tablas de consulta. Dichas unidades son tablas de verdad que implementan pequeñas funciones lógicas, y al interconectarse entre sí, implementan la funcionalidad completa del módulo. Por otro lado, las partes de memoria necesaria para soportar los bloques alojados se encuentran representadas en registros.

La figura 6.3 muestra el porcentaje en ocupación en tablas LUT por los diferentes módulos, teniendo en cuenta una FPGA Virtex 7 xc7vx485t. Como se muestra en la figura 6.3, en la configuración mas avanzada del procesador, los módulos pertenecientes al ROB y BTB utilizan un cantidad de tablas LUT muy superior a los demás bloques, debido a que su comportamiento es más complejo que el del resto de unidades. Estos dos módulos implementan tablas en registros y cada una de sus entradas puede ser accedida tanto para lectura como para escritura. Esto hace que la lógica de control de la tabla crezca de forma significativa. Nótese que este efecto no se aprecia en los módulos de memoria cache (*L1I* y *MEM*). El motivo es que estas unidades se han implementado como recursos BRAM (recursos específicos de la FPGA) por lo que la lógica asociada se simplifica. En trabajo futuro se plantea implementar los módulos ROB y BTB con bloques BRAM.

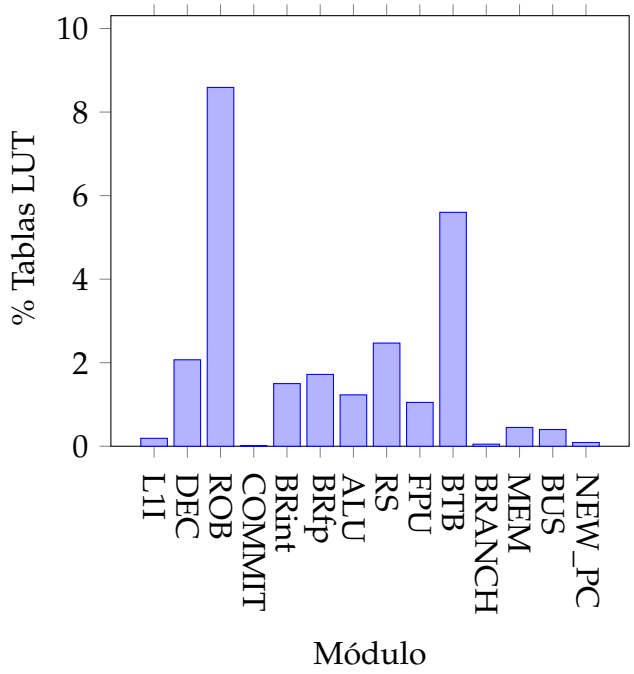

Figura 6.3: Resultados de síntesis en tablas *LUT* de la configuración G

Nótese también la simplicidad del módulo *COMMIT* y *BRANCH*. Estas dos unidades implementan funciones lógicas muy sencillas, por lo que el área de ocupación en la FPGA es prácticamente inexistente. Siguiendo con el análisis, en la figura 6.4, se muestra el porcentaje de registros que contiene cada módulo. Se observa de nuevo como los módulos ROB y BTB contienen una mayor cantidad de registros, debido a que estas unidades precisan del almacenamiento de tablas y por ello utilizan estos registros.

Lo mismo ocurre con el módulo RS, que si bien no tiene mucha lógica asociada, en términos de registros ocupa un porcentaje relevante. Esta diferencia se debe al uso exclusivo de estaciones de reserva por operadores, lo que hace que la lógica de selección de estaciones de reserva sea menor. Hay que hacer constar también que las unidades *DEC*, *COMMIT*, *BUS* y *NEW_PC* son totalmente combinacionales por lo que no ocupan en registros.

A continuación evaluamos el porcentaje de tablas LUT que utilizan las diferentes configuraciones del procesador. Tal y como se visualiza en la figura 6.5, la última configuración tiene un mayor número de componentes de mayor complejidad. El procesador G duplica la lógica del procesador F al pasar de dos a cuatro vías. En lógica se observan de hecho tres incrementos significativos. De la configuración A a B, debido al incremento de número de entradas del ROB de 16 a 32. De la configuración C a D, debido al incremento en estaciones de reserva de 2 a 4 por operador y por último de F a G, como ya se ha dicho al duplicar el número de vías del procesador.

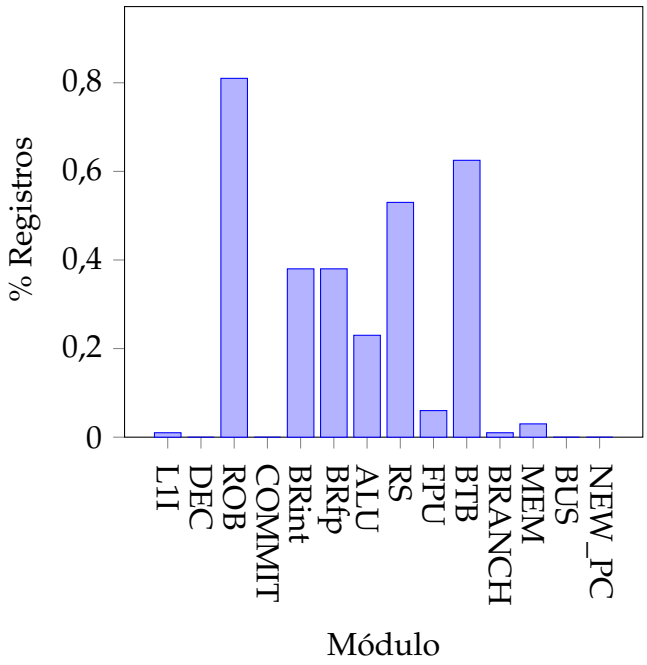

Figura 6.4: Resultados de síntesis en registros de la configuración G

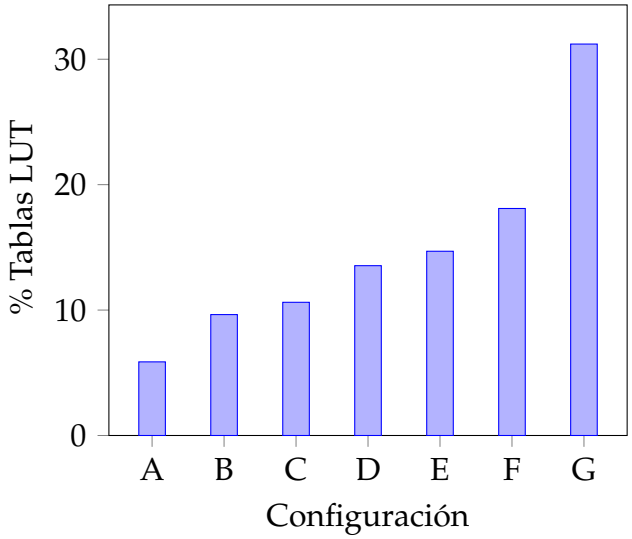

Figura 6.5: Gráfica con los resultados de síntesis en tablas *LUT* de todas las configuraciones

La tendencia anterior en lógica, sin embargo, no se manifiesta en términos de registros. En la figura 6.6, se muestra el porcentaje de registros que utiliza cada configuración del procesador. Cabe destacar que las configuraciones que menos registros utilizan son aquellas que tienen un menor número de entradas en el ROB y menos estaciones de reserva por operador, cosa que hace que descienda

el uso de estos elementos. Además también se observa un mayor porcentaje de uso de los registros cuando aumentamos el número de ALU y FPU, debido a la segmentación que está implementada mediante memorias internas.

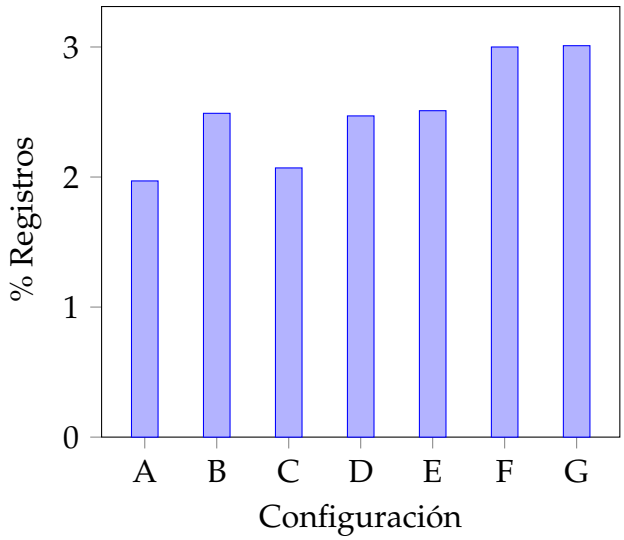

Figura 6.6: Resultados de síntesis en registros en cada configuración en todo el núcleo

6.3 Síntesis de los Componentes Específicos

Tal y como se ha detallado el funcionamiento y la estructura de nuestras unidades en el capítulo 5, en esta sección mostramos la utilización en nuestra FPGA que utilizan estos componentes. Este estudio, al igual que en el apartado anterior, mide la cantidad de recursos lógicos necesarios expresados en tablas LUT (*Lookup Table*), o tablas de consulta y las partes de memoria necesarias para soportar los bloques ubicados que se encuentran expresadas en registros.

6.3.1. Síntesis de la ALU y la FPU

Nuestros operadores principales ALU y FPU tienen un papel primordial en este procesador ya que se encargan de realizar las operaciones aritméticas enteras y de coma flotante, respectivamente. Es por ello que, si nos fijamos en la tabla 6.3, vemos que ocupan una cantidad considerable de recursos en lógica y registros.

Módulo	Núm tablas LUT	%Tablas LUT	Núm regs	% Regs
ALU	3 739	1,23 %	1 411	0,23 %
FPU	3 196	1,05 %	425	0,06 %

Tabla 6.3: Resultados de síntesis de los operadores funcionales

Estos datos se reflejan gráficamente en la figura 6.7, en la cual se comprueba que la unidad de aritmética de enteros tiene más coste que la de coma flotante tanto si hablamos de tablas LUT como de registros. Pero en la parte de memoria se diferencian mucho más y esto es debido a que la primera contiene tres suboperadores, los cuales internamente están segmentados utilizando muchos más recursos de almacenamiento que la FPU, que tiene solo un operador segmentado.

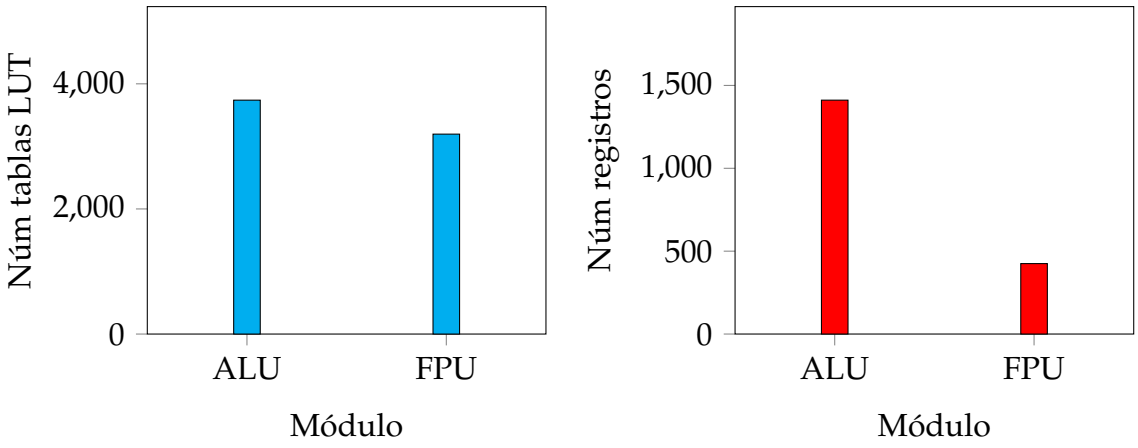

Figura 6.7: Resultados de síntesis de los operadores funcionales

6.3.2. Síntesis de los Sumadores-Restadores

Una vez se han implementado y probado los sumadores-restadores solo tenemos que ver las diferencias entre ellos en la tabla 6.4. El nombre de los módulos, nos indica el tipo de operador y el número de etapas de segmentación.

Módulo	Núm tablas LUT	%Tablas LUT	Núm regs	% Regs
SR_CPA_4stages	73	0,02 %	185	0,03 %
SR_CLA_1L_4stages	107	0,03 %	196	0,03 %
SR_CLA_2L_2stages	84	0,02 %	115	0,01 %
CPA_4bits	4	$<$ 0,01 %	0	0,00 %
CLA_1L_4bits	5	$<$ 0,01 %	0	0,00 %
CLA_2L_16bits	37	0,01 %	0	0,00 %

Tabla 6.4: Resultados de síntesis de los distintos tipos de sumadores-restadores y las celdas de suma-resta

Los dos primeros elementos de la figura 6.8 se encuentran diseñados en cuatro registros de segmentación, con lo cual deducimos que el *SR_CPA_4stages* con ocho instancias del bloque *CPA_4bits* es el que menos área ocupa tanto en lógica como en registros.

El tercer componente (*SR_CLA_2L_2stages*) debería tener ocupadas más tablas LUT porque se supone que es la versión con mayores recursos y mejores prestaciones pero observamos que no es así.

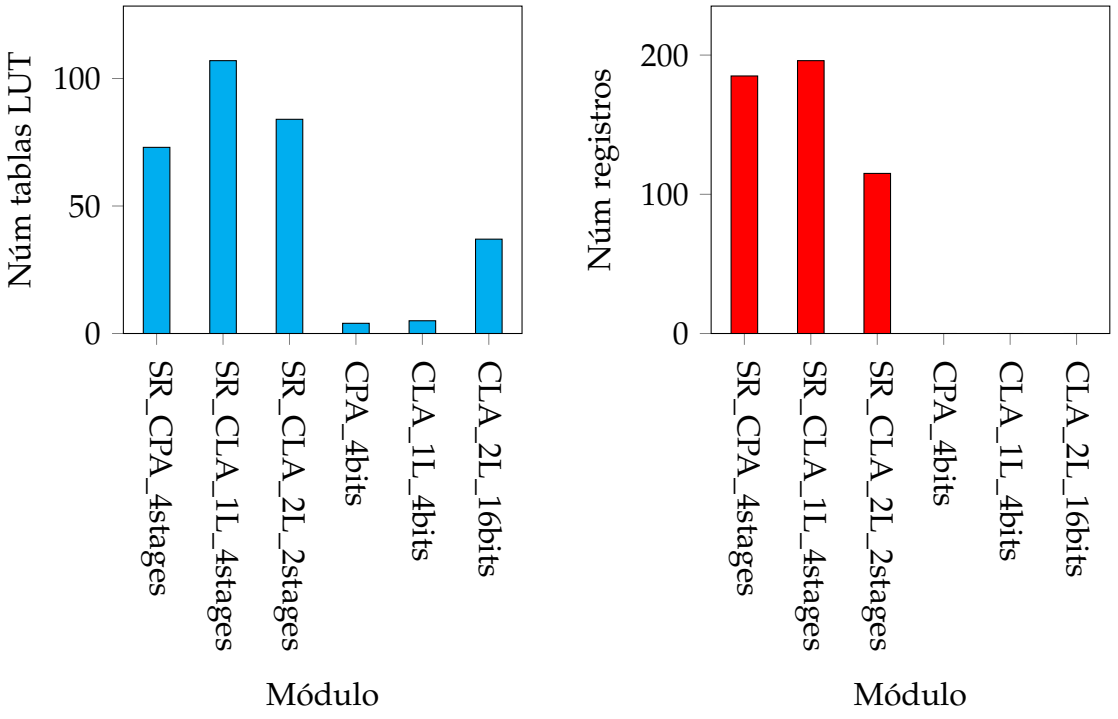

Figura 6.8: Resultados de síntesis de los distintos tipos de sumadores-restadores y las celdas de suma-resta

Esto es debido a que tenemos una restricción. El sumador-restador con bloques CLA de un nivel está segmentado en cuatro etapas y el posterior solo con dos, pero las unidades que contienen estos son distintas. Las celdas CLA de un nivel operan con 4 bits y las otras que son de dos niveles con 16, que por lo tanto, ocupan más parte lógica. Además, la unidad *SR_CLA_1L_4stages* contiene ocho instancias de los bloques de adición de un nivel frente al operador *SR_CLA_2L_2stages* que comprende solo dos unidades *CLA_2L_16bits*.

Por último, cabe destacar la ausencia de registros en las celdas de suma-resta debido a que su diseño es completamente combinacional.

6.3.3. Síntesis del Multiplicador, Divisor y Lógico-Desplazamiento

Después de analizar los sumadores-restadores que contiene la ALU, nos resta examinar los resultados del multiplicador, divisor y el suboperador lógico y de desplazamiento mostrados en la tabla 6.5.

Módulo	Núm tablas LUT	%Tablas LUT	Núm regs	%Regs
Multiplier_5stages	2 188	0,72 %	1 129	0,18 %
Divider_3stages	509	0,16 %	125	0,02 %
Others_1stage	564	0,18 %	42	$<$ 0,01 %

Tabla 6.5: Resultados de síntesis del multiplicador, divisor y el suboperador lógico-desplazamiento

Como podemos comprobar, la unidad de multiplicación necesita una gran cantidad de tablas LUT y registros de nuestra FPGA. Todo esto es debido a la gran complejidad que contiene y a causa de su estructura interna, como se ha visto en el apartado 5.1.2, compuesta por un árbol de componentes CSA de ocho niveles repartidos en tres fases de segmentación de las cuales sus registros son de un tamaño considerable. Esta diferencia en recursos tan notable, comparándola con los otros dos componentes, se puede ver en la figura 6.9.

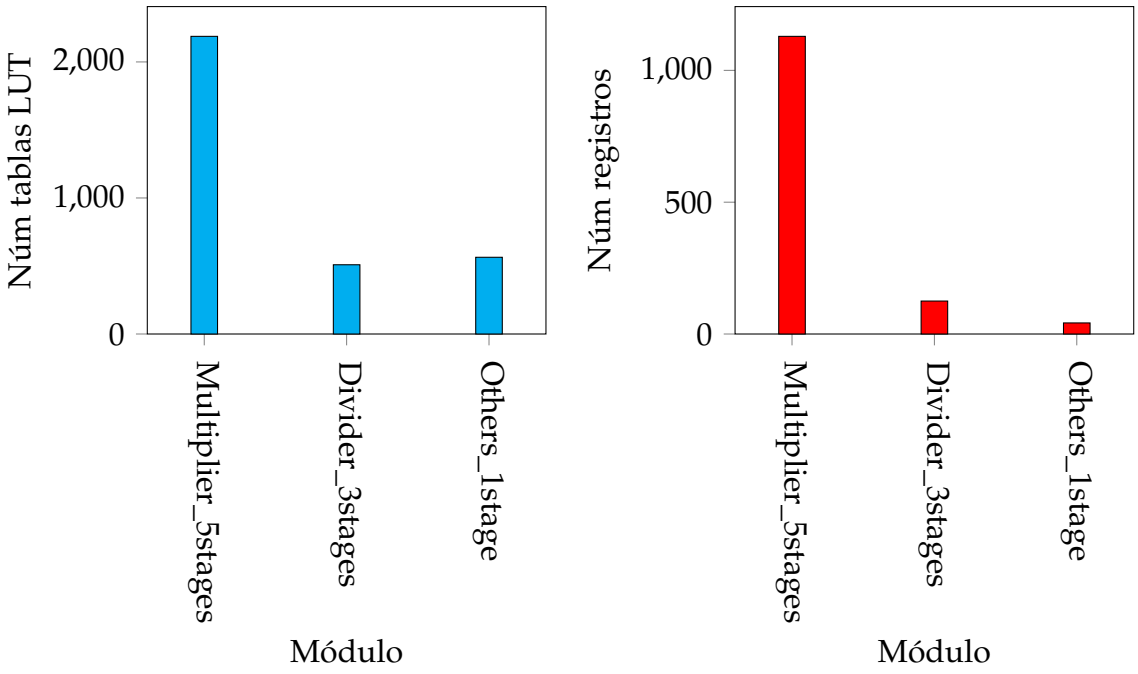

Figura 6.9: Resultados de síntesis del multiplicador, divisor y el suboperador lógico-desplazamiento

Por otro lado, el operador de división utiliza menos parte lógica que *Others* porque solo implementa la división y no múltiples operaciones como la anteriormente mencionada. El módulo *Divider_3stages*, como su nombre indica, está diseñado en tres fases de segmentación; por lo contrario, el suboperador lógico y de desplazamiento solamente contiene un registro que almacena los resultados al final de su estructura interna.

CAPÍTULO 7

Conclusiones

En este capítulo se expone a modo de resumen qué objetivos se han visto alcanzados durante el desarrollo del trabajo, así como los problemas más importantes que aparecieron a lo largo del trabajo.

7.1 Contratiempos y problemas sufridos

El principal problema fue que, a medida que avanzaba el trabajo, se añadieron funcionalidades que originalmente no se habían contemplado. Este hecho implicó un rediseño de casi todos los módulos ya implementados. Otro contratiempo muy importante fue que ninguno de los componentes del grupo conocía Verilog, lenguaje muy específico del area de Ingeniería de Computadores, por lo que al principo del trabajo se dedicó tiempo al estudio y aprendizaje de dicho lenguaje.

De cara a las pruebas se precisaba de una licencia de la empresa Xilinx, propietaria del software Vivado. Por ello se tuvo que hacer una petición expresa al GAP para que se nos otorgara una y así poder usar la herramienta completa. A la hora de realizar la síntesis de los módulos en los equipos del laboratorio, tuvimos problemas con las versiones tanto del sistema operativo como de Java; es por ello que se acabó trabajando en ordenadores personales y teniendo así un control absoluto sobre el sistema.

En un primer instante se hizo una implementación del ROB y de MEM poco óptima, en cuanto a área de la FPGA se refiere, debido al uso excesivo de registros. Por esta razón que se minimizó el uso de este tipo de elementos, a la espera de una posible traducción del comportamiento basado en registros a otro basado en bloques de memoria.

En el diseño inicial se optó por una ALU que a pesar de tener varios operadores internos (sumador-restador, multiplicador, divisor, entre otros), solo usara uno; de esta manera se escogía una instrucción de las estaciones de reserva asocia-

das. Se varió el comportamiento para que pudieran entrar en ejecución diversas instrucciones si usaban diferente recurso interno dentro de la ALU. Por ejemplo, una suma y una multiplicación ejecutándose a la vez.

Aunque inicialmente se pretendía soportar un paquete básico de instrucciones, se precisó de una ampliación para programas más complejos y así medir mejor el rendimiento del procesador. Esto requirió una gran comprensión del formato de instrucciones del MIPS, dado que ya no solo se trataba de soportar sumas y restas, sino instrucciones con una implementación más compleja, como `mtc1`, `jr`, `sw` y `cvt`, entre otras.

Una vez implementado el código se pasó a la prueba del comportamiento, usando la herramienta de simulación proporcionada por Vivado. Aun así, se necesitaba una interfaz más personalizada, y así agilizar la fase de pruebas. Para ello se creó desde cero un programa en Python que, usando como fuente un fichero *log* generado por Vivado, transformara la información en un cronograma divido por instantes de tiempo (véase 3.4).

Debido a todos los contratiempos indicados anteriormente no se pudo llegar a insertar el procesador desarrollado en el sistema FPGA, por lo que este se convertirá en un trabajo futuro.

7.2 Consideraciones finales

Cabe destacar que, como se ha mencionado en capítulos anteriores, no es individual, sino que es la suma del esfuerzo de cuatro alumnos, intentando así hacer un trabajo de más envergadura del que se podría haber realizado individualmente. A lo largo del todo el desarrollo se ha reforzado la cooperación entre los miembros, tanto en la ayuda con el código como en el uso de las herramientas, hecho que ha supuesto un perfeccionamiento de las aptitudes profesionales de los componentes del equipo.

En cuanto a la implementación del código, no solo se han conseguido los objetivos de implementar los módulos iniciales, sino que además se han diseñado, implementado y añadido otros. Por ejemplo el módulo *field_comp* del descodificador, el *New_PC_addr*, y por último una unidad que implementa el algoritmo *round robin*.

Dentro en la parte de verificación, mediante el uso de la herramienta Vivado se simuló el comportamiento que tendría el procesador dentro de la FPGA, y así se pudieron depurar errores de implementación. Cabe destacar que esta parte fue uno de los puntos fuertes del trabajo. Con el que se consiguió una gran familiarización con el entorno de simulación.

Por último, a la hora de sintetizar, se pudo observar el espacio ocupado de cada módulo y así pensar en diseños alternativos que hagan hincapié en la eficiencia del área de la FPGA.

7.3 Ampliaciones y trabajo futuro

Dado que no se pudo insertar el procesador desarrollado en el sistema FPGA, una ampliación posible sería insertar este trabajo en la FPGA junto con toda la arquitectura PEAK.

Varias unidades del núcleo desarrollado lanzan excepciones al procesador, pero nadie recoge esa información. Como trabajo futuro se debería capturar este comportamiento y obrar en consecuencia, añadiendo un módulo que centralizara esta tarea, eliminando el comportamiento de unidad de control que actualmente tiene el descodificador.

Como posible mejora se podría añadir el soporte para instrucciones con doble precisión en todo el procesador. Esto quiere decir que la longitud de los operandos y el resultado son de 64 bits, en vez de los 32 bits actuales. Con esta ampliación se obtendría una mejor precisión a la hora de realizar operaciones con enteros o de coma flotante y por lo tanto un mayor rango de representación. Cabe destacar que la unidad ROB ya incluye el soporte para valores de 64 bits en su campo *value*.

Por otro lado, gracias a la estructura modular del descodificador, una futura linea de trabajo podría ser cambiar todo el comportamiento basado en instrucciones entrantes con formato MIPS a otro, como podría ser OpenRISC, debido a que toda la interpretación del formato de instrucciones se encuentra en un único módulo (*DEC_instr*) no implicaría un gran nivel de complejidad. Siguiendo con posibles mejoras en el descodificador, se podrían añadir registros de segmentación para que esta unidad se divida en dos, con el propósito de eliminar la gran cantidad de entradas que llegan a este elemento desde el banco de registros. Así pues, en la primera etapa de segmentación del descodificador se haría una petición al banco de registros de los operandos fuente, y en la segunda se obtendrían. Como se ha podido observar, se debería por lo tanto añadir también cierta funcionalidad a los bancos de registros, para poder soportar este nuevo comportamiento del descodificador.

Otra posible optimización consistiría en cambiar la implementación del ROB para que este haga uso de los bloques de memoria configurable del sistema FPGA, reduciendo así el área que ocupa el módulo *reorder buffer* en dicho dispositivo una vez sintetizado.

En el ROB todas las instrucciones comparten una misma estructura indiferentemente del operador que utilizan. Se podría extrapolar esta idea a las estaciones de reserva. Es decir, tener estaciones de reserva genéricas y de este modo buscar un posible aumento en el uso de estas estructuras. Para identificar el tipo de las instrucciones albergadas se añadiría un campo más en esta nueva estructura.

Otro de los trabajos futuros consistiría en efectuar un reemplazo del módulo interno *man_add_sub* situado en la FPU por uno de los tres sumadores-restadores que se han implementado en la unidad aritmético-lógica. Este bloque tiene la

función de sumar o restar mantisas y se podrían intercambiar perfectamente ya que tenemos suboperadores que realizan este trabajo mucho mas rápido. Como la unidad de coma flotante se encuentra diseñada por fases, si se realiza la optimización anteriormente citada, se conseguiría un operador supersegmentado por el hecho de contener unidades internas divididas por registros de segmentación.

En aras a completar el funcionamiento de la unidad de memoria, se podría añadir el comportamiento para dos instrucciones más, estas son: `sc` (*Store Conditional Word*) y `ll` (*Load Linked Word*), donde la primera guarda una palabra completa (también de 32 bits) en memoria de forma atómica, y la segunda carga una palabra (32 bits) de memoria de forma atómica. Nótese que el propósito de ambas instrucciones es evitar condiciones de carrera gracias a la forma atómica de su ejecución.

Bibliografía

[1] Mark Gordon Arnold. *Verilog digital computer design: Algorithms into hardware*. Upper Saddle River, Prentice Hall PTR, 1999.

[2] *Mips Architecture and Assembly Language Overview.* University of Illinois at Chicago, [consulta: 22 febrero 2015, 18:00]. Disponible en: `http://logos.cs.uic.edu/366/notes/MIPS%20Quick%20Tutorial.htm`.

[3] Joseph J. F. Cavanagh. *Digital Computer Arithmetic - Design and Implementation*. McGraw-Hill, Inc., 1985.

[4] Michael D. Ciletti. *Advanced digital design with the Verilog HDL*. Upper Saddle River, Prentice Hall, 2011.

[5] *Pro Git* git-scm, 2014, [consulta: 15 enero 2015, 13:00]. Disponible en: `https://progit2.s3.amazonaws.com/en/2015-05-31-24e8b/progit-en.519.pdf`.

[6] Hennessy, J.L., Patterson, D.A. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2012.

[7] Hennessy, J.L., Patterson, D.A. *Computer organization and design: the hardware-software interface*. Morgan Kaufmann, 2012.

[8] *IEEE Standard for Binary Floating-Point Arithmetic.* IEEE Standards Board, 1985, [consulta: 9 enero 2015, 17:00]. Disponible en: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=30711`.

[9] *MIPS (procesador)* Wikipedia: The Free Encyclopedia., 26 mayo 2014 3:46 última modificación[consulta: 2 Junio 2015, 12:00]. Disponible en: `https://es.wikipedia.org/wiki/MIPS_(procesador)`.

[10] MIPS Technologies, Inc. *MIPS Architecture for Programmers Volume I: Introduction to the MIPS32 Architecture*. Mountain view, & CA, marzo, 2001.

[11] MIPS Technologies, Inc. *MIPS Architecture for Programmers Volume II: The MIPS32 Instruction Set*. Mountain view, & CA, marzo, 2001.

[12] MIPS Technologies, Inc. *MIPS Architecture for Programmers Volume II: The MIPS32 Instruction Set*. Mountain view, & CA, junio, 2003.

[13] Larus R, James. *SPIM S20 A MIPS R2000 Simulator* University of Wisconsin Madison, 1997, [consulta: 22 abril 2015, 17:00]. Disponible en: http://pages.cs.wisc.edu/~larus/SPIM/spim_documentation.pdf.

[14] Shrivastava Purnima, Tiwari Mukesh, Singh Jaikaran y Rathore Sanjay. VHDL Environment for Floating point Arithmetic Logic Unit - ALU Design and Simulation *Research Journal of Engineering Sciences*, Vol. 1(2), 1-6, Sehore, MP, INDIA, agosto, 2012.

[15] *Vivado Design Suite User Guide: Using the Vivado IDE.* Xilinx, 30 abril 2014, [consulta: 10 enero 2015, 10:00]. Disponible en: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug893-vivado-ide.pdf.

[16] *pi* MathIsFun.com [consulta: 5 julio 2015, 14:00]. Disponible en: http://www.mathsisfun.com/numbers/pi.html.

[17] *All Programmable 7 Series Product Tables and Product Selection Guide* Xilinx, 2015, [consulta: 5 julio 2015, 14:00]. Disponible en: http://www.xilinx.com/publications/prod_mktg/7-series-product-selection-guide.pdf.

APÉNDICE A

Interconexionado de Módulos

A.1 RS

Dependiendo del tipo de estación de reserva el contenido de los puertos difiere, aunque en muchos casos se asemeja y solo difiere en el puerto de salida *to_OP*, siendo la interfaz del modulo la siguiente:

- *RSx_reset* (entrada, 1 bit). Una por cada estación de reserva definida.
- *fromCOMMITx* (entrada, 9 bits). Hay tantos *fromCOMMITx* como bloques*COMMIT* se definen. Cada *fromCOMMITx* es un bus con la siguiente configuración:
  - *valid* (1 bit). Indica que el contenido es válido.
  - *flush* (1 bit). Indica que se tiene que eliminar todas las RS menos las escrituras confirmadas.
  - *confirm* (1 bit). Indica que hay que confirmar un *store*.
  - *er* (6 bits). Estación de reserva a confirmar.
- *RSx_toDEC* (salida, 147 bits). Una por cada estación de reserva definida. Cada *RSx_toDEC* contiene:
  - *used* (1 bit). Indica si la RS está siendo utilizada.
  - *LRU* (2 bits). Indica la prioridad al usar el algoritmo LRU.
  - *PC* (32 bit). Dirección de la propia instrucción
  - *operation* (6 bits). Operación a realizar.
  - *valid1* (1 bit). Indica si el campo value1 contiene un dato válido.
  - *value1* (32 bits). Contenido del operando uno.
  - *tag1* (8 bits). *Tag* asociado al operando uno.

- *valid2* (1bit[64]). Indica si el campo value2 contiene un dato válido
- *value2* (32 bits). Contenido del operando dos.
- *tag2* (8 bits). *Tag* asociado al operando dos.
- *value3* (16 bits). Contenido del operando tres(inmediato,no requiere *valid* ni *tag*)
- *rob_entry* (8 bits). *Tag* del resultado.

- *RSx_fromDEC* (entrada, 147 bits). Una por cada estación de reserva definida. Contiene el mismo formato que *toDECx*.

- *fromBUSx* (entrada, 80 bits). Tantas como buses hayan definidos. Cada *BUS* contiene la siguiente información:

  - *valid* (1 bit). Indica que el contenido del bus es válido.
  - *taken* (1 bit). Indica si la la condición del salto a sido *true*.
  - *rob_entry* (8 bits). Entrada de ROB asociada a la operación.
  - *value* (64 bits). Valor del resultado producido por un operador.
  - *exception* (1 bit). Indica si la instrucción ha generado una excepción.
  - *exception_code* (5 bits). Indica el código de excepción.

- *RSx_toOP* (salida). Una por cada estación de reserva. Dependiendo del tipo de estación de reserva el contenido de los puertos difiere:

  - Para estaciones de reserva de FPU/ALU:
    - *used* (1 bit). *RS* en uso.
    - *valid1* (1 bit). Indica si el campo *value1* contiene un dato válido.
    - *value1* (32 bits). Valor del operando uno.
    - *valid2* (1 bit). Indica si el campo value2 contiene un dato válido.
    - *value2*(32 bits). Valor del operando dos.
    - *op*(6 bits). Operación a realizar.
    - *rob_entry*(8 bits). Entrada ROB asociada.
  - Para estaciones de reserva de *MEM*:
    - *used* (1 bit). *RS* en uso.
    - *confirmed* (1 bit). Bit de escritura confirmada.
    - *LRU*(2 bits). Indica el orden temporal de las instrucciones al ser descodificadas.
    - *op*(6 bits). Operación a realizar.
    - *rob_entry* (8bits). Entrada del ROB asociada.
    - *value3*(16bits). Valor del operando uno.
    - *value2* (32 bits). Valor del operando dos.

    - *value1* (32 bits). Valor del operando uno.
    - *valid2* (1 bit). Indica si el campo *value2* contiene un dato válido.
    - *valid1* (1 bit). Indica si el campo *value1* contiene un dato válido.
  - Para estaciones de reserva de *BRANCH*:
    - *used* (1 bit). Indica *RS* en uso.
    - *PC* (32 bits). PC de la propia instrucción.
    - *op* (6 bits). Operación a realizar.
    - *rob_entry* (8 bits). Entrada de ROB asociada
    - *value3* (16 bits). Valor del operando tres.
    - *value2* (32 bits). Valor del operando tres.
    - *value1* (16 bits. Valor del operando tres.
    - *valid2* (1 bit). Indica si el campo *value2* contiene un dato válido.
    - *valid1* (1 bit). Indica si el campo *value1* contiene un dato válido.

A.2 *BUS*

Los puertos de entrada y salida de este modulo difieren en función del número de elementos definidos. A continuación se definirá cada uno de los puertos de entrada/salida:

- *DATAin_OP* (e ntrada, 78 bits). Hay tantos *DATAin_OP* como operados se hayan definido. Cada *DATAin_OP* es un *BUS* que contiene la siguiente configuración:
  - *rob_entry* (8 bits). Entrada del ROB asociada a la operación.
  - *value* (64 bits). Valor del operador.
  - *exception* (1 bit). Indica si la instrucción ha generado una excepción.
  - *exception_code* (5 bits). Indica el código de excepción.
  - En caso de tratarse de un operador de salto, la configuración aumentará una posición siendo el bit de mayor peso el siguiente:
    - *taken* (1 bit). Indica si el salto al final resulto tomado o no.
- *REQUEST_RSX* (entrada, 1 bit). Uno por cada estación de reserva definida.
- *GRANT_RSX* (salida, 1 bit). Uno por cada estación de reserva definida.
- *BUSxOut* (salida, 80 bits). Tantas como buses hayan definidos. Cada *BUS* contiene la siguiente información:
  - *valid* (1 bit). Indica que el contenido del *BUS* es válido.

- *taken* (1 bit).
- *rob_entry* (8 bits). Entrada de ROB asociada a la operación.
- *value* (64 bits). Valor del resultado producido por un operador.
- *exception* (1 bit). Indica si la instrucción ha generado una excepción.
- *exception_code* (5 bits). Indica el código de excepción.

A.3 *DEC*

El módulo *DEC* se encarga de realizar la descodificación de una instrucción. Para ello, el módulo tiene el siguiente interfaz:

- *num_ROB_entries_in* (entrada, 8 bits) Número de entradas del ROB disponibles.
- *first_ROB_entry_in* (entrada, 8 bits) Primera entrada disponible en el ROB.
- *RS_in* (entrada, 147 bits una por estación de reserva definida), cada *RS_in* contiene:
  - *used* (1 bit). Indica si la RS está siendo utilizada.
  - *LRU* (2 bits).
  - *PC* (32 bit). Dirección de la propia instrucción.
  - *operation* (6 bits). Operación a realizar.
  - *valid1* (1 bit). Indica si el campo value1 contiene un dato válido.
  - *value1* (32 bits). Contenido operando 1.
  - *tag1* (8 bits). *Tag* asociado al operando 1.
  - *valid2* (1 bit). Indica si el campo *value* 2 contiene un dato válido.
  - *value2* (32 bits). Contenido operando 2.
  - *tag2* (8 bits). *Tag* asociado al operando2.
  - *value3* (16 bits). Contenido operando 3 (inmediato, no requiere *valid* ni *tag*).
  - *rob_entry* (8 bits). *Tag* del resultado.
- *RBfp_in* (entrada, 74 bits por registro) Cada uno de los 32 registros contiene:
  - *rob_valid* (1 bit). Indica campo *rob_entry* es válido.
  - *rob_entry* (8 bits). Tag asociado al registro.
  - *value* (32 bits). Valor del registro.
  - *last_value* (32 bits). Ultimo valor pendiente de *commit*.

  - *valid_last_value* (1 bit). Indica *last_value* es válido.

- *RBint_in* (entrada, 74 bits por registro) Cada uno de los 34 registros (32 más HI y LO) contiene:
  - *rob_valid* (1 bit). Indica campo *rob_entry* es válido.
  - *rob_entry* (8 bits). *Tag* asociado al registro.
  - *value* (32 bits). Valor del registro.
  - *last_value* (32 bits). Último valor pendiente de *commit*.
  - *valid_last_value* (1 bit). Indica que *last_value* es válido.
- *instr* (entrada, 32 bits). Instrucción MIPS a descodificar.
- *enable_instr* (entrada, 1 bit). Indica si la instrucción a descodificar es válida.
- *enable_in* (entrada, 1 bit). Habilita el descodificador.
- *PC_addr* (entrada, 32 bits). Dirección de la instrucción a decodificar.
- *predict* (entrada, 1 bit). Predicción de salto de la instrucción a descodificar.
- *enable_out* (salida, 1 bit). Indica que el descodificador ha descodificado una instrucción.
- *num_ROB_entries_out* (salida, 8 bits). Número de entradas disponibles tras la decodificación.
- *first_ROB_entry_out* (salida, 4 bits). Siguiente entrada disponible del ROB.
- *RS_out* (salida, 147 bits). Contiene *RS_in* pero modificado tras la descodificación de la instrucción.
- *RBfp_out* (salida, 74 bits). Contiene *RBfp_in* pero modificado tras la descodificación de la instrucción.
- *RBint_out* (salida, 74 bits). Contiene *RBint_in* pero modificado tras la descodificación de la instrucción.
- *toROB* (salida, 85 bits). Contiene la configuración para el ROB siendo un vector de bits con los siguientes campos:
  - *valid* (1 bit). Indica que el contenido es válido
  - *disableBTB* (1 bit) Indica que el BTB debe deshabilitarse.
  - *j_jal* (2bit) Indica que la instrucción es una `j`, una `jal` o una `jr`, por lo que *taken* debe de ser igual a 1.
  - *set_wb* (1bit). Indica que el ROB debe poner WB a uno al llegar la operación desde el *DEC*, (usado en instrucciones que no pasan por ningún operador).

  - *reg_destination* (6 bits). Indica registro destino de la operación.
  - *bank_destination* (1 bit). Indica el banco de registros destino (0 enteros, 1 coma flotante).
  - *branch* (1 bit). Indica si la operación es de tipo salto.
  - *predict* (1 bit). Indica la predicción realizada.
  - *write_reg* (1 bit). Indica si la operación es de tipo aritmética (escribe en el banco de registros)
  - *store* (1 bit). Indica si la operación es un *store* a memoria.
  - *rs* (6 bits). Indica la estación de reserva utilizada (para localizar después *stores* y confirmarlas).
  - *pc_dest* (32 bit). Dirección de salto (solo `j` y `jal`).
  - *pc_dir* (32 bits). Dirección PC de la instrucción descodificada.

- *BUS_in* (entrada, 79 bits). Contiene el contenido de todos los buses del procesador.
  - *valid* (1 bit). Indica que el contenido del *BUS* es válido.
  - *Rob_entry* (8 bits). Contiene el valor de la entrada de ROB.
  - *value* (64 bits). Contiene el resultado a escribir.
  - *exception* (1 bit). Indica si la operación generó alguna excepción.
  - *exception_code* (5 bits). Contiene la causa de excepción.

A.4 ROB

El módulo ROB implementa un *reorder buffer* de 256 *slots*. El número de *slots* será parametrizable y siempre potencia de dos. Para su implementación se utilizan dos punteros, *first* y *last*, los dos implementando una cola circular. El puntero *first* apunta al *slot* que contiene la primera entrada del ROB (la instrucción que se debe retirar) mientras que el puntero *last* apunta al siguiente *slot* que se puede utilizar para ubicar una nueva entrada (la instrucción que se descodifica). Se utilizará un registro interno que llevará la cuenta de entradas disponibles.

- *fromDECx* (entrada). Contiene la configuración a escribir que proviene de un descodificador. Hay tantos *fromDECx* como descodificadores se implementen (número de vías del procesador). Mirar puerto de salida *toROB* en el descodificador para una descripción de los campos.

- *num_ROB_entries* (salida, 8 bits). Indica cuantas entradas hay disponibles en la tabla.

- *first_ROB_entry* (salida, 8 bits). Indica la primera entrada a utilizar en el ROB. Se usa en los descodificadores.
- *fromCOMMITx* (entrada, 3 bits). BUS con las órdenes del *COMMIT*. Hay tantos *fromCOMMIT* como *COMMIT* se implementen (grado del procesador). *fromCOMMITx* contiene los siguientes campos:
  - *valid* (1 bit). Indica que el contenido es válido.
  - *flush* (1 bit). Indica que se tienen que eliminar todas las entradas del ROB.
  - *remove_first_entry* (1 bit). Indica que se tiene que eliminar la primera entrada del ROB.
- *toCOMMITx* (salida, 161 bits). *BUS* con la información para el *COMMIT*. Hay tantos *toCOMMITx* como *COMMITS* se implementen (grado del procesador). *toCOMMIT* contiene los siguientes campos:
  - *valid* (1 bit). Indica que el contenido es válido
  - *rob_entry* (8 bits).
  - *disableBTB* (1 bit). Indica que esta operación no debe hacer uso del *BTB*, siempre debe fallar.
  - *instr_addr* (32 bits). Contiene el *PC* de la instrucción actual.
  - *dest_addr* (32 bits). Contiene la dirección calculada.
  - *exception* (1 bit). Indica que la instrucción a generado una excepción.
  - *exception_code* (5 bits). Contiene la causa de la excepción.
  - *register* (6 bits). Indica el registro destino.
  - *register_bank* (1 bit). Indica si el banco es de enteros (0) o coma flotante (1).
  - *value* (64 bits). Valor a escribir.
  - *branch* (1 bit). Indica si la instrucción es de salto.
  - *write_reg* (1 bit). Indica si la instrucción es de tipo aritmética (escribe en registro).
  - *predict* (1 bit). Indica la predicción realizada.
  - *taken* (1 bit). Indica si el salto fue tomado (1) o no (0).
  - *store* (1 bit). Indica si la instrucción es una *store*.
  - *rs* (5 bits). Indica la estación de reserva asociada.
  - *fromBUSx* (entrada, 80 bits). BUS entre estaciones de reserva y recursos. Hay tantos *fromBUSx* como buses implementados. Cada *BUSout* contiene los siguientes campos:

  - *valid* (1 bit). Indica que el contenido del BUS es válido.
  - *taken* (1 bit).
  - *rob_entry* (8 bits). Índice del ROB.
  - *value* (64 bits). Valor producido
  - *exception* (1 bit). Indica que se ha producido una excepción.
  - *exception_code* (5 bits). Contiene el código de excepción.

- *rst* (entrada, 1 bit). *Reset* del módulo. Cuando la señal esté a nivel bajo, y ante un flanco de subida de la señal de reloj, el módulo se debe reinicializa.

- *clk* (entrada, 1 bit). Reloj del procesador.

A.5 *BRANCH*

El módulo *BRANCH* se encarga de calcular la dirección de salto y el resultado del salto (*taken*, *not taken*). La interfaz del módulo es la siguiente:

- *RSx* (entrada, uno por estación de reserva del operador). Véase la definición del puerto *toOPx* (*BRANCH*) en el módulo RS para un mayor detalle.

- *request* (1 bit, salida). Petición al *BUS*.

- *grant* (1 bit, entrada). Reconocimiento del *BUS*.

- *rsx_reset* (1 bit, salida, uno por estación de reserva del operador). Fuerza la liberación de la estación de reserva asociada.

- *out* (salida). Datos de salida. Incluye:

  - *value* (64 bits).
  - *rob_index* (6 bits).
  - *exeption* bit (1 bit).
  - *exeption code* (5 bits).

- *reset* (entrada, 1 bit). Indica *reset* a nivel bajo del módulo.

- *clk* (entrada, 1 bit). Señal de reloj del procesador.

A.6 *COMMIT*

En cada ciclo, el bloque *COMMIT* confirma una instrucción. La interfaz del módulo es la siguiente:

- *fromROB* (entrada, 162 bits), contiene la información relevante de la entrada del ROB asociada a la operación de commit. Véase *toCOMMITx* en el módulo ROB para un detalle de sus campos.
- *enable_in* (entrada), señal que habilita el *commit*.
- *toROB* (salida, 3 bits), contiene las acciones a realizar en el ROB por parte del *commit*. Véase *fromCOMMITx* en el módulo ROB para un detalle de sus campos.
- *toRBint* (salida, 81 bits), contiene las acciones a realizar en el banco de registros de enteros por parte del *commit*. Véase *fromCOMMITx* en el módulo RB para un detalle de sus campos.
- *toRBfp* (salida, 81 bits), contiene las acciones a realizar en el banco de registros de coma flotante por parte del *commit*. Véase *fromCOMMITx* en el módulo RB para un detalle de sus campos.
- *toRS* (salida, 9 bits), contiene las acciones a realizar en las estaciones de reserva por parte del *commit*. Véase *fromCOMMITx* en el módulo RS para un detalle de sus campos.
- *toBTB* (salida, 68 bits), contiene las acciones a realizar en el módulo BTB por parte del *commit*. Véase *fromCOMMITx* en el módulo BTB para un detalle de sus campos.
- *enable_out* (salida), señal que habilita los siguientes *commit* (conectados en serie).
- *flush_toOP* (salida), señal que obliga a los operadores a parar la ejecución y eliminar las instrucciones en proceso.

A.7 ALU

La unidad ALU tiene la función de ejecutar las operaciones con enteros aritméticas y lógicas que provienen de las estaciones de reserva y enviá su resultado al bus. La interfaz del módulo es la siguiente:

- *RSx* (entrada, uno por estación de reserva del operador). Véase la definición del puerto *toOPx* (ALU) en el módulo RS para un mayor detalle.

- *request* (1 bit, salida). Petición al *BUS*.
- *grant* (1 bit, entrada). Reconocimiento del *BUS*.
- *rsx_reset* (1 bit, salida, uno por estación de reserva del operador). Fuerza la liberación de la estación de reserva asociada.
- *out* (salida). Datos de salida. Incluye:
  - *value* (64 bits).
  - *rob_index* (6 bits).
  - *exeption* bit (1 bit).
  - *exeption code* (5 bits).
- *reset* (entrada, 1 bit). Indica *reset* a nivel bajo del módulo.
- *clk* (entrada, 1 bit). Señal de reloj del procesador.

A.8 FPU

La unidad FPU tiene la función de ejecutar las operaciones en coma flotante que provienen de las estaciones de reserva y enviá su resultado al bus. La interfaz del módulo es la siguiente:

- *RSx* (entrada, uno por estación de reserva del operador). Véase la definición del puerto *toOPx* (ALU) en el módulo RS para un mayor detalle.
- *request* (1 bit, salida). Petición al *BUS*.
- *grant* (1 bit, entrada). Reconocimiento del *BUS*.
- *rsx_reset* (1 bit, salida, uno por estación de reserva del operador). Fuerza la liberación de la estación de reserva asociada.
- *out* (salida). Datos de salida. Incluye:
  - *value* (64 bits).
  - *rob_index* (6 bits).
  - *exeption* bit (1 bit).
  - *exeption code* (5 bits).
- *reset* (entrada, 1 bit). Indica *reset* a nivel bajo del módulo.
- *clk* (entrada, 1 bit). Señal de reloj del procesador.

A.9 *MEM*

El módulo *MEM* se encarga de todas las lecturas y escrituras a memoria. Tiene la siguiente interfaz.

- *RSx* (entrada, uno por estación de reserva del operador). Véase la definición del puerto *toOPx* (*MEM*) en el módulo RS para un mayor detalle.
- *request* (1 bit, salida). Petición al *BUS*.
- *grant* (1 bit, entrada). Reconocimiento del *BUS*.
- *rsx_reset* (1 bit, salida, uno por estación de reserva del operador). Fuerza la liberación de la estación de reserva asociada.
- *out* (salida). Datos de salida. Incluye:
  - *value* (64 bits).
  - *rob_index* (6 bits).
  - *exeption* bit (1 bit).
  - *exeption code* (5 bits).
- *reset* (entrada, 1 bit). Indica *reset* a nivel bajo del módulo.
- *clk* (entrada, 1 bit). Señal de reloj del procesador.

A.10 BTB

El módulo BTB se encarga de realizar las predicciones de salto, así como de calcular la dirección de la siguiente instrucción a ejecutar en caso de predicción de salto. También se encarga de actualizar la información utilizada para realizar la predicción. La interfaz del módulo es la siguiente:

- *address* (entrada, 32 bits). Dirección del PC actual.
- *addressBTB* (salida, 32 bits). Indica la dirección de salto cuando se predice.
- *enableX* (salida, 1 bit). Hay un *enable* por decodificador. El primer *enable* (*enable0*) siempre estará a uno. Los demás se calculan internamente. Cuando un *enable* está a cero impedirá que el descodificador interprete la instrucción.
- *predictX* (salida, 1 bit). Hay una señal *predict* por descodificador. Indica si la instrucción ha hecho una predicción de salto (0 no salta, 1 salta).

- *fromCOMMITx* (entrada, 67 bits). Hay un *fromCOMMIT* por cada módulo COMMIT implementado en el procesador.
- *rst* (entrada), señal de *reset* a nivel bajo. Se eliminarán todos los *tags* y los bits *valid_last_value* y se pondrán todos los valores a cero de los registros.
- *clk* (entrada). Señal de reloj.

A.11 *RBint* y *RBfp*

El banco de registro almacena y mantiene los registros del procesador. La interfaz del módulo es la siguiente:

- *fromCOMMITx* (entrada, 81 bits). Tantos como bloques *COMMIT* tenga definidos el procesador). Cada *fromCOMMIT* tendrá los siguientes campos:
  - *valid* (1 bit). Indica que el contenido del registro es válido.
  - *flush* (1 bit). De estar esta señal activa se deben eliminar todos los *tags* de los registros, así como el campo *valid_last_value*.
  - *commit* (1 bit). Indica que se debe hacer *commit* de un registro.
  - *register* (6 bits). Registro a realizar el *commit*.
  - *value* (64 bits). Valor a escribir en el registro.
- *toDEC* (salida). Véase puerto de entrada de *RBint* del módulo *DEC*.
- *fromDEC* (entrada). Contiene los mismos campos que el puerto de salida *toDEC*
- *fromBUS* (entrada, tantos como *BUS* tenga definidos el procesador). Véase *fromBUSx* del módulo RS.
- *rst* (entrada). Señal de *reset* a nivel bajo. Se eliminarán todos los *tags* y los bits *valid_last_value* y se pondrán todos los valores a cero de los registros.
- *clk* (entrada). Señal de reloj.

APÉNDICE B

Tabla de casos especiales en la FPU

Tal y como se ha mencionado en apartados anteriores, existe un elemento en la FPU que se encarga de analizar los casos especiales que presentan las operaciones en coma flotante que no están contemplados en la aritmética entera. Esta unidad avisa a las otras para que sepan en qué caso se encuentran. Se pueden encontrar operadores con valores iguales a infinito, NaN, En la tabla B.1, se muestra cada uno de estas situaciones pero cabe destacar que el orden de las mismas no tiene significado alguno.

Caso	Descripción
1	Operación de multiplicación o división y el operador A con valor NaN.
2	Operación de multiplicación o división y el operador B con valor NaN.
3	Multiplicación con el operador A con valor igual a cero y B infinito.
4	Multiplicación con el operador B con valor igual a cero y A infinito.
5	Multiplicación con ambos operadores con valores infinitos.
6	División por cero.
7	División con ambos operadores con valores infinitos.
8	Multiplicación con algún operador con valor igual a cero.
9	Conversión de entero a simple precisión y el operador A con valor igual a cero.
10	Conversión de simple precisión a entero y el operador A con valor NaN.
11	Operación de suma o resta y el operador A con valor QNaN.
12	Operación de suma o resta y el operador B con valor QNaN.
13	Operación de suma o resta con ambos operadores con valores infinitos pero signos distintos.
14	Operación de suma o resta y algún operador con valor infinito positivo.
15	Operación de suma o resta y algún operador con valor infinito negativo.
16	Operación de suma o resta y el operador A con valor igual a cero.
17	Operación de suma o resta y el operador B con valor igual a cero.
18	Operación de suma o resta con una diferencia entre los exponentes de los operadores mayor a los bits de la mantisa y el exponente A mayor al B.
19	Operación de suma o resta con una diferencia entre los exponentes de los operadores mayor a los bits de la mantisa y el exponente B mayor al A.
20	Conversión de simple precisión a entero y el operador A con valor igual a cero.
21	Suma o resta y el operador A con valor SNaN.
22	Suma o resta y el operador B con valor SNaN.
23	División y el operador A infinito.
24	División y el operador B infinito.
25	División y el operador A con valor diferente de cero y el B igual a cero.
26	División y el operador B con valor diferente de cero y el A igual a cero.
27	División y el operador A con valor diferente de cero y el A igual a uno.
28	Conversión de simple precisión a entero y el operador A infinito.
29	Operación condicional y ambos operadores con valor igual a cero.
30	Operación condicional y el operador A con valor igual a cero.
31	Operación condicional y el operador B con valor igual a cero.
0	Ninguno de los anteriores.

Tabla B.1: Casos especiales analizados en la FPU

APÉNDICE C

Códigos de prueba

En esta sección se muestran los códigos de los programas que se han utilizado para realizar las pruebas y verificar el correcto funcionamiento del procesador.

C.0.1. AXPY

El siguiente código corresponde al programa *axpy*:

```
        #; #z = a*x + y
        #; #Tamaño de los vectores: 64 palabras

.globl x
 #; #Vector x
.data 0x10000000
x:
.float 0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0
  .float 10.0,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0
  .float 20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,28.0,29.0
  .float 30.0,31.0,32.0,33.0,34.0,35.0,36.0,37.0,38.0,39.0
  .float 40.0,41.0,42.0,43.0,44.0,45.0,46.0,47.0,48.0,49.0
  .float 50.0,51.0,52.0,53.0,54.0,55.0,56.0,57.0,58.0,59.0
  .float 60.0,61.0,62.0,63.0

.globl y
#; #Vector y
y:
.float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
.float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
.float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
.float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
.float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
.float 100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0,100.0
```

```
.float 100.0,100.0,100.0,100.0
#; #Vector z
#;#   64 elementos son 256 bytes
.globl z
z:
.space 256
  #;# Escalar a
.globl a
a:
.float 1.0
  #; El código

.text 0x00400000
.globl start
start:
  la $t1, x
la $t2, y
la $t3, z
la $t0, a
lwc1 $f0, 0($t0)
addi $t4,$0,64 # cuenta de iteraciones

.globl loop
loop:
lwc1 $f2, 0($t1)
lwc1 $f4, 0($t2)
mul.s $f6, $f2, $f0
    add.s $f6, $f6, $f4
    swc1 $f6, 0($t3)
    addi $t1, $t1, 4
    addi $t2, $t2, 4
addi $t3, $t3, 4
    addi $t4, $t4, -1
    bnez $t4, loop
    .end
```

C.0.2. Cálculo de π

El siguiente código corresponde al programa π:

```
#Aproxima a pi guardando el valor en el registro $8
addi $10, 100
li $2, 2    # $2 = 2
mtc1 $2, $f2 # $f2 = 2
cvt.s.w $f2, $f2 # $f2 = 2.0

li $3, 3    # $3 = 3
```

```
mtc1 $3, $f3 # $f3 = 3
cvt.s.w $f3, $f3 # $f3 = 3.0

li $4, 4    # $4 = 4
mtc1 $4, $f4 # $f4 = 4
cvt.s.w $f4, $f4 # $f4 = 4.0

li $5, 2    # $5 = 2
mtc1 $5, $f5 # $f5 = 2
cvt.s.w $f5, $f5 # $f5 = 2.0

li $6, 3    # $6 = 3
mtc1 $6, $f6 # $f6 = 3
cvt.s.w $f6, $f6 # $f6 = 3.0

li $7, 4    # $7 = 4
mtc1 $7, $f7 # $f7 = 4
cvt.s.w $f7, $f7 # $f7 = 4.0

li $8, 0    # $8 = 0
mtc1 $8, $f8 # $f8 = 0
cvt.s.w $f8, $f8 # $f8 = 0.0

li $9, 0    # $9 = 0
mtc1 $9, $f9 # $f9 = 0
cvt.s.w $f9, $f9 # $f9 = 0.0
add.s $f8, $f8, $f3 # $f8 = 3.0

bucle:  mul.s $f9, $f5, $f6 # $f9 = 2 * 3
mul.s $f9, $f9, $f7 # $f9 = 2 * 3 * 4
div.s $f9, $f4, $f9 # $f9 = 4/(2*3*4)
add.s $f8, $f8, $f9 # $f8 = 3.0 + 4/(2*3*4)

add.s $f5, $f5, $f2 # $f5 +2
add.s $f6, $f6, $f2 # $f6 +2
add.s $f7, $f7, $f2 # $f7 +2

mul.s $f9, $f5, $f6 # $f9 = 4 * 5
mul.s $f9, $f9, $f7 # $f9 = 4 * 5 * 6
div.s $f9, $f4, $f9 # $f9 = 4/(4*5*6)
sub.s $f8, $f8, $f9 # $f8 = $f8 - 4/(4*5*6)

add.s $f5, $f5, $f2 # $f5 +2
add.s $f6, $f6, $f2 # $f6 +2
add.s $f7, $f7, $f2 # $f7 +2
addi $10, -1
bne $10, $0, bucle
```