UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de las instrucciones multimedia en un procesador para un sistema multicore en FPGA

Trabajo Final de Grado

Grado en Ingeniería Informática

Autor: Mario David Cariñana Albasolo

Directores:
José Flich Cardo
Xavier Molero Prieto

2 de septiembre de 2014

Resumen

El presente trabajo tiene como objetivo simular el comportamiento de un procesador MIPS vectorial en una FPGA (*Field Programmable Gate Array*) multinúcleo. Para ello se implementa un juego de instrucciones multimedia o vectoriales junto con la lógica necesaria (registros vectoriales, unidad aritmético lógica vectorial, etc.) para que dichas instrucciones se puedan ejecutar en un procesador con arquitectura MIPS. De igual modo, algunos de los elementos de la ruta de datos del procesador inicial también han tenido que ser modificados (unidad de control, etapas intermedias, memorias cache, etc.) Posteriormente se prueba y analiza la mejora del rendimiento del procesador vectorial con respecto a un procesador escalar.

Palabras clave: procesador, vectorial, juego de instrucciones.

Abstract

The present study aims to simulate the behavior of a vector MIPS processor in a multicore FPGA. For this purpose a set of multimedia or vector instructions with the necessary logic (vector registers, vector arithmetic logic unit, etc.) is implemented to ensure that these instructions can be executed on a MIPS processor architecture. Likewise some of the elements of the initial data path processor also had to be modified (control unit, intermediate stages, cache, etc.) Afterward is tested and analyzed the improved vector processing performance respect to a scalar processor.

Keywords: processor, vector, instructions.

Índice de abreviaturas

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

CU (*Control Unit*), Unidad de control

EX (*Execute*), Etapa de ejecución

EXM (*Execution Memory*), Registro intermedio de las etapas de ejecución y memoria

FPGA (*Field Programmable Gate Array*), Campo de puertas programable

HDL (*Hardware Description Lenguage*), Legunaje de descripción de hardware

HI (*High*), Alto (registro)

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

IDEX (*Instruction Decode Execute*), Registro intermedio de las etapas de decodificación de instrucciones y ejecución

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

IFID (*Instruction Fetch Instruction Decode*), Registro intermedio de las etapas de búsqueda de instrucciones y decodificación de instrucciones

LO (*Low*), Bajo (registro)

M (*Memory*), Etapa de memoria

MWB (*Memory Writeback*), Registro intermedio de las etapas de memoria y reescritura

PC (*Program Counter*), Contador de programa

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

RTL (*Register Transfer Level*), Registro de transferencia de nivel

SIMD (*Single Instruction Multiple Data*), Una sola instrucción varios datos

WB (*Writeback*), Etapa de reescritura

Índice general

Índice de figuras

Capítulo 1

Introducción, motivación y contexto

1.1. Introducción

Este trabajo pretende implementar la arquitectura SIMD (*Single Instruction Multiple Data*) sobre un procesador MIPS, pero no de la manera que el estándar de la arquitectura lo describe, [4, 5]. El estándar describe una ruta de datos puramente vectorial, registros vectoriales, ALU (*Arithmetic Logic Unit*) vectorial, etc. Sin embargo en el trabajo en la ruta de datos se incluyen los elementos vectoriales y los escalares (dos ALUs, vectorial y escalar, dos bancos de registros, vectoriales y escalares, etc.).

Cuando este trabajo se empezó a realizar, muchos, por no decir todos los detalles sobre la propuesta de SIMD que proponía MIPS se desconocían por completo. Por ello fue necesario un proceso de documentación y posteriormente otro de análisis para ver cómo se iba a proceder a la implementación del procesador.

El punto de partida fue un procesador que ya estaba implementado. El procesador en cuestión no era otro que un MIPS R2000, el que se ha ido usando como ejemplo a lo largo de toda la carrera, [6, 7, 8]. La mayor novedad que el trabajo introduce es la comentada antes, crear un nuevo procesador que aúna el procesador vectorial y el escalar.

En esta memoria se van a tratar diversos temas relacionados con el trabajo, el primero de ellos es la motivación y los objetivos del trabajo, después se tratan la aquitectura MIPS [1, 2, 3], la extensión vectorial (SIMD) de dicha arquitectura [4, 5] y las herramientas de diseño utilizadas en el trabajo. Tam-

bién se describe detalladamente la implementación y por último se describen y analizan varios casos de uso que han servido como ejemplos.

1.2. Motivación y objetivo

Hay diferentes motivaciones dentro de este trabajo, una de las principales sin lugar a duda es explorar y comprender desde un punto de vista cercano cómo funciona y cómo está estructurado un procesador, visto de una manera menos teórica que en las asignaturas de la carrera.

Otra de las motivaciones fundamentales es explorar la arquitectura SIMD [4] y descubrir una concepción completamente diferente a la tradicional que se tenía de un procesador y su comportamiento. Al trabajar con diferentes modos de operación y volúmenes de cálculo mucho mayores (vectores) se pueden aplicar diferentes planteamientos para resolver un problema de mayor eficacia.

Por otro lado, el hecho de aprender un nuevo lenguaje de programación como es en este caso Verilog [9] también es una motivación importante, porque ya es una aproximación a la vida profesional y a las condiciones de aprender un lenguaje nuevo por tu cuenta, sin estar un entorno académico.

Los objetivos que se propusieron en un primer momento eran: adaptar la ruta de datos dada para que fuera compatible con la arquitectura vectorial, desarrollar un conjunto de instrucciones bastante reducido (todas con el mismo formato de instrucción dentro de los diferentes vectoriales) y comparar los costes de implementación dentro de una FPGA del procesador escalar con respecto al nuevo procesador vectorial.

Estos objetivos fueron alcanzados con creces y se propusieron unos más ambiciosos. El primero de ellos era ampliar el juego de instrucciones añadiendo los distintos formatos de instrucciones vectoriales, con lo cual se pasó de un conjunto de 5 instrucciones a más de 18.

Dentro de dicho juego ampliado de instrucciones, se han incluido también instrucciones de memoria (*load* y *store* vectoriales) que en los primeros objetivos no habían sido propuestas debido a que para que estas instrucciones pudieran funcionar dentro de la ruta de datos los cambios que tenían que hacerse eran mucho más extensos que los que en un principio se buscaban.

Como último objetivo también se propuso la resolución de un problema vectorial que tuviera interés en un entorno real. Se escogió, siguiendo este criterio, un problema de multiplicación de matrices. Cabe destacar que se tuvo que

desarrollar una nueva instrucción que no estaba concebida en las revisiones del MIPS para poder realizar el problema de la forma que queríamos (en concreto la instrucción `addm`) la cual será tratada ampliamente con posterioridad. El código en ensamblador de dicho problema, así como la codificación de las instrucciones vectoriales, también se llevaron a cabo. Las instrucciones vectoriales fueron codificadas a mano, puesto que el simulador QTSpim (como se explicará con posterioridad) no codifica este tipo particular.

1.3. Contexto del trabajo

Hoy por hoy los procesadores multinúcleo juegan un papel fundamental en el campo de la informática, nos permiten ahorrar una gran cantidad de tiempo de cómputo realizando a la vez diferentes operaciones de un mismo problema que no tienen una dependencia de datos entre ellas. Esto es muy utilizado en diferentes contextos, algunos ejemplos son: operaciones de multiplicación de matrices (como las que vamos a ver en esta memoria), calculo de gráficos, obtención de valores estadísticos, etc.

En lo que se refiere a las instrucciones multimedia se realizan sobre múltiples datos de manera simultánea. Como se puede intuir esto supone una gran mejora temporal respecto a un juego de instrucciones escalar, puesto que una operación vectorial equivale a un bucle completo que procesaría los N elementos del registro vectorial. Lógicamente el realizar una operación vectorial tarda más tiempo que realizar una operación escalar, no obstante este coste temporal se rentabiliza en casi todos los casos. En problemas con volumenes de datos pequeños no es recomendable, puesto que retrasaría la ejecución.

Las FPGA son un dispositivo semiconductor que contiene bloques cuya interconexión y funcionalidad puede ser configurada mediante un lenguaje de descripción especializado. La lógica programable puede reproducir todo tipo de funciones, con lo cual es perfecto para simular el procesador que se ha de diseñar e implementar.

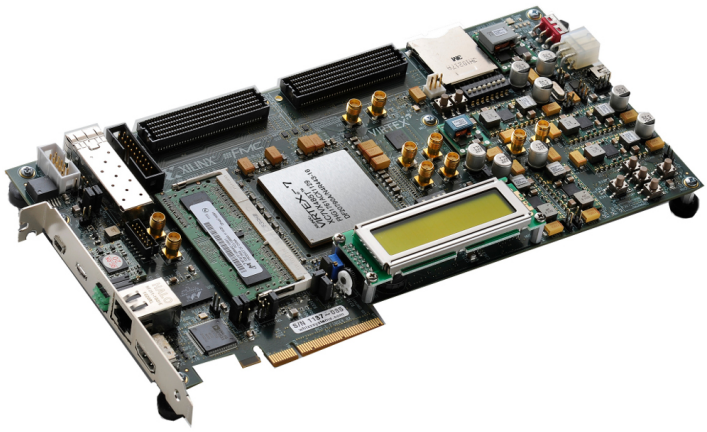

Figura 1.1: Xilinx Virtex 7.

La FPGA de la cual dispone el departamento para este tipo de proyectos se trata de una Xilinx Virtex 7 (Figura 1.1). Cuenta con dos millones de puertas lógicas que involucran a 6800 millones de transistores y con velocidades pico de hasta 28 Gb/s.

Capítulo 2

La arquitectura MIPS

2.1. Descripción de la arquitectura

Los procesadores MIPS fueron concebidos por un equipo de la Universidad de Stanford en 1981, liderado por John L. Hennessy (ver Figura 2.1). La idea básica era mejorar el rendimiento del procesador mediante la segmentación. La segmentación divide el procesador en diferentes etapas y permite el avance de instrucciones etapa a etapa.

Esto resultó innovador, puesto que los diseños tradicionales siempre se habían caracterizado por necesitar que una instrucción terminase de ejecutarse por completo en el procesador para que la siguiente comenzase su ejecución. Lógicamente el planteamiento no segmentado es mucho peor puesto que la mayor parte del procesador permanecía inactivo casi todo el tiempo.

Otro de los aspectos que MIPS cambió fue la frecuencia del reloj, puesto que esta estaba establecida como el tiempo de ciclo completo de todo el procesador, con los nuevos diseños pasó a ser la latencia de la etapa de segmentación que más tardaba.

La segmentación creó la necesidad establecer unos bloqueos de manera que las instrucciones que necesitaban varios ciclos de reloj para completarse dejasen de cargar datos desde los registros de segmentación.

Dichos bloqueos suponían un retraso importante. Para solucionar esto el diseño del MIPS se planteó como objetivo que todas las subfases de todas las instrucciones tardasen un único ciclo en su ejecución, dejando de ser necesarios los bloqueos.

Una vez ya establecido el contexto, podemos pasar a hablar del propio procesador. Internamente el procesador está compuesto por diferentes elementos

Figura 2.1: John L. Hennessy.

que serán enumerados y explicados a continuación.

La ALU es un circuito digital que calcula operaciones aritméticas y operaciones lógicas entre dos operandos, los cuales serán extraídos del banco de registros. Se diferencia la operación que se va a realizar en función del código de operación que se le pasa.

El procesador tiene las memorias cache de instrucciones y de datos separadas (tipo Harvard). Así como un banco de 32 registros de 32 bits de propósito general para operaciones con enteros ($0..$31), de los cuales el 0 siempre tiene un valor fijado a 0. También dispone de 32 registros de 32 bits que soportan la aritmética en coma flotante ($f0..$f31).

El chip del procesador incorpora el coprocesador CP0 el cual aporta funciones de manejo de excepciones, planificación del direccionamiento de memoria y control de recursos críticos. Dentro de dicho coprocesador destacan los siguientes registros:

- `$08`: registro que alberga la dirección de la excepción de direccionamiento más reciente.
- `$12`: registro de estado del procesador y control.
- `$13`: registro de causa de la interrupción mas reciente.
- `$14`: registro EPC, contador de programa de la última interrupción.

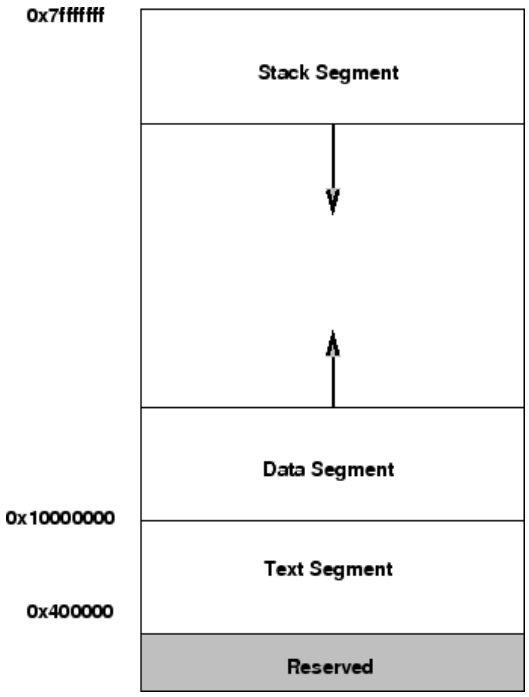

Figura 2.2: Organización de la memoria del MIPS.

También cabe destacar que dispone de 2 registros especiales ubicados en la ALU, los cuales son HI (*High*) y LO (*Low*), que sirven para almacenar los resultados de la multiplicación/división y para operaciones de transferencia de datos.

La organización de la memoria en el caso del MIPS se realiza de la siguiente manera. La memoria de instrucciones en modo usuario empieza en la dirección `0x00400000` y termina en la `0x0FFFFFFF`. La memoria de datos en modo usuario empieza en la `0x10000000` y termina en la `0x7FFFFFFF`. La memoria accesible por el usuario es de tipo RAM (*Random Access Memory*) sobre la cual se pueden realizar operaciones tanto de escritura como de lectura.

2.2. Ruta de datos de la que se parte

La ruta de datos de la que se parte está bastante bien reflejada en la Figura 2.3. Como se puede apreciar la ruta está dividida en 5 etapas por los diferentes registros de etapa los cuales son síncronos, y tienen que tener en cuenta los riesgos de datos que pueda sufrir el código. A continuación se enumerarán cada una de las etapas y los elementos que hay en ellas:

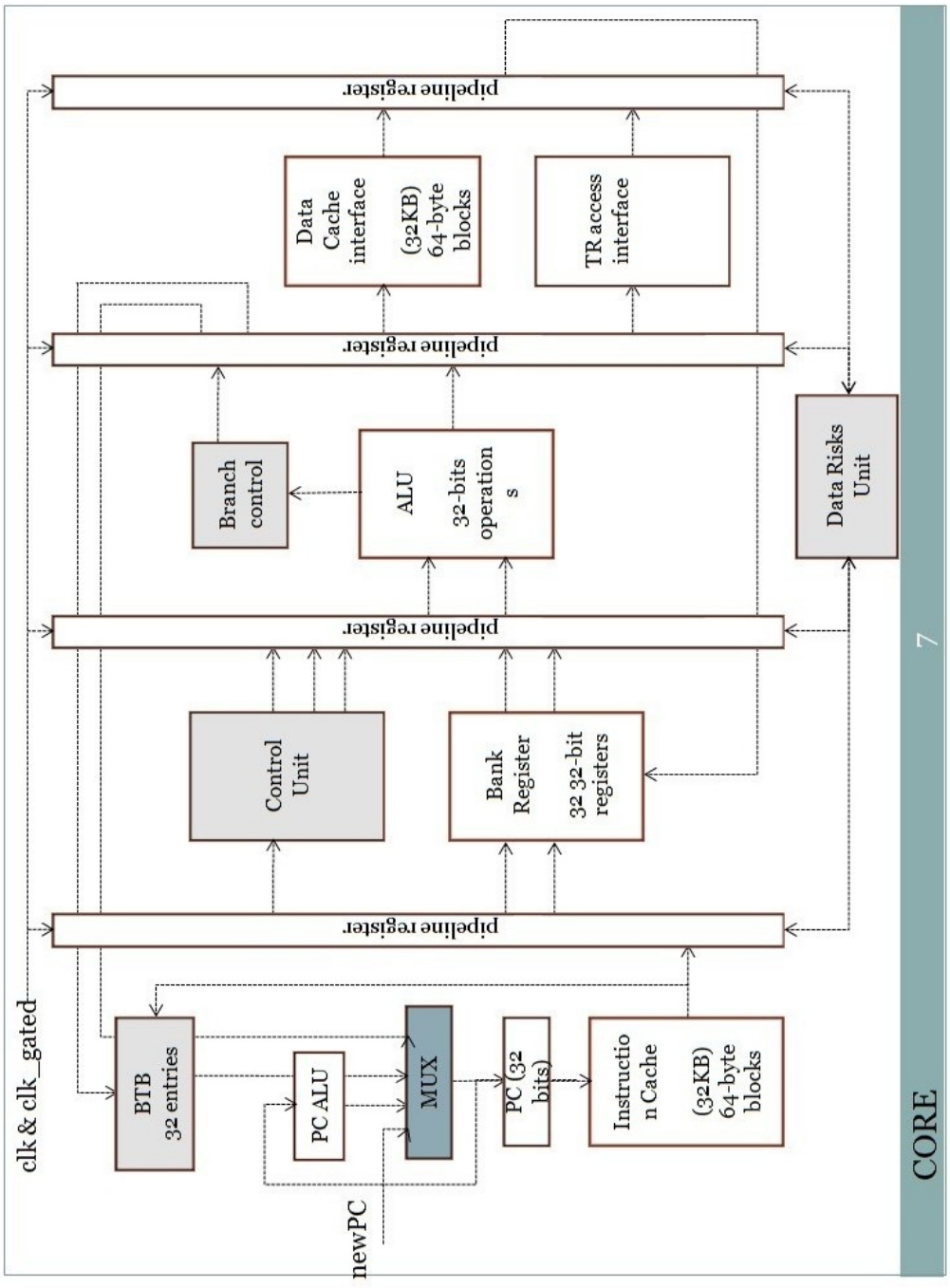

Figura 2.3: Ruta de datos inicial del trabajo.

1. La primera de ellas es IF (*Instruction Fetch*), en esta etapa se encuentra la cache de instrucciones, además de el PC (*Program Counter*), y la ALU encargada de actualizarlo en función de cómo se vaya ejecutando el código en cuestión. Además de lo ya citado se encuentran un multiplexor para el PC y un BTB de 32 entradas.

2. En segundo lugar tras pasar por el primer registro de etapa llamado IFID (*Instruction Fetch Instruction Decode*) llega la segunda etapa, ID (*Instruction Decode*). En ella encontramos el banco de registros, la extensión de signo para los datos inmediatos de las instrucciones que así lo requieran y por último la CU (*Control Unit*).

3. Tras el segundo registro, IDEX (*Instruction Decode Execute*), está la etapa EX (*Execute*), en la cual están albergados la ALU junto con sus dos multiplexores correspondientes y el *branch control*, el encargado de los saltos.

4. En el cuarto lugar se encuentra la etapa M (*Memory*) (tras pasar por el tercer registro de etapa EXM (*Execution Memory*)) en donde podemos encontrar la otra cache (la de datos), un par de multiplexores y el *tilereg_access*.

5. Por último está la etapa WB (*Writeback*) (tras pasar por MWB (*Memory Writeback*), el útimo registro de etapa), esta únicamente contiene un multiplexor para seleccionar el dato que se necesita en función de la instrucción que ya ha sido ejecutada.

Capítulo 3

Extensión multimedia

3.1. SIMD

La extensión que se ha aplicado a la ruta de datos es más compleja de lo que en un principio se podría llegar a intuir. Esto es debido a la complejidad para aunar en un solo diseño un MIPS junto con un MIPS con arquitectura SIMD. MIPS contempla la arquitectura SIMD como un ente separado de la arquitectura normal. Es decir, por un lado estaría el MIPS con una arquitectura clásica, y por otro lado el SIMD, pero no de manera unida.

La principal diferencia con la extensión vectorial es que los registros pasan de ser de 32 bits a 128 bits. Dentro de estos registros las instrucciones cuentan con diferentes modos de funcionamiento en función de la manera en que están almacenados los datos dentro de los registros de 128:

La primera opción es la doble palabra (extensión `.d` de las instrucciones), en este caso se almacenan dos datos de 64 bits dentro del registro en cuestión.

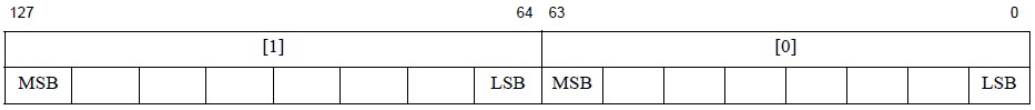

Figura 3.1: Formato para doble palabra (*Double_word*)

También se puede almacenar como palabras, es decir 4 datos de 32 bits (extensión `.w` en las instrucciones).

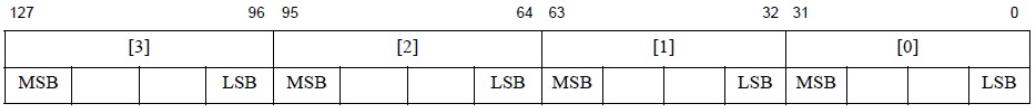

Figura 3.2: Formato para palabra (*Word*)

Otro modo es en forma de medias palabras, 8 datos de 16 bits (extensión `.h` en las instrucciones).

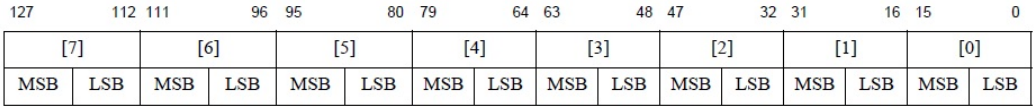

Figura 3.3: Formato para media palabra (*Half_word*)

Por último también se pueden almacenar como 16 octetos (extensión `.b` en las instrucciones).

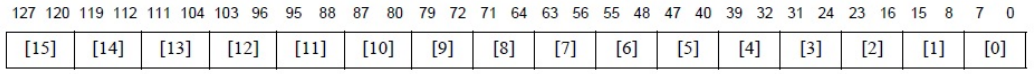

Figura 3.4: Formato para octetos (*Byte*)

Las instrucciones sin embargo no varían en lo que a la cantidad de bits utilizados se refiere. Es decir, la codificación de instrucciones sigue siendo en 32 bits. Sin embargo, los formatos dentro de esos bits sí que cambian. Es decir, en una instrucción del juego de instrucciones escalar, los bits que indican el primer registro son del 25 al 21, sin embargo en el caso de las vectoriales son del 15 al 11. En función del tipo de instrucción hay un formato u otro y se pueden clasificar de la siguiente manera:

Instrucciones aritméticas con dos registros. En el caso de este tipo de instrucciones tanto para las escalares como las vectoriales, dispone de 5 bits para indicar de que registros se leen los operandos (cada uno de los operandos) y cual es el registro de destino. La diferencia entre ellas radica en la posición en la que están ubicados dentro de los 32 bits.

Otra cosa que no varía es que tanto los 6 bits de mayor peso como los 6 de menos peso son fijos para distinguir entre instrucciones con esos bits. La mayor diferencia que existe es que en el caso de las vectoriales existe el formato de datos *data format* (este campo se puede observar en la Figura 3.5) y en las escalares no existe.

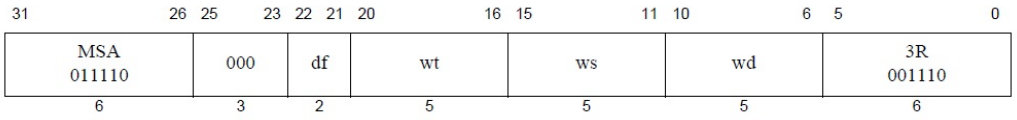

Figura 3.5: Instrucción aritmética con dos registros

Instrucciones aritméticas con un inmediato. En este caso las diferencias entre vectoriales y escalares si que son más significativas. El dato inmediato en el caso de la instrucción vetorial como se puede observar en la Figura 3.6 ocupa 5 bits del 16 al 20. Sin embargo en el caso del escalar son 16 bits, del 0 al 15. En este caso el vectorial también incluye un campo de formato de datos. La similitud mas significativa son los 6 bits de mayor peso, que conforman el código de la operación y son fijos en ambos casos.

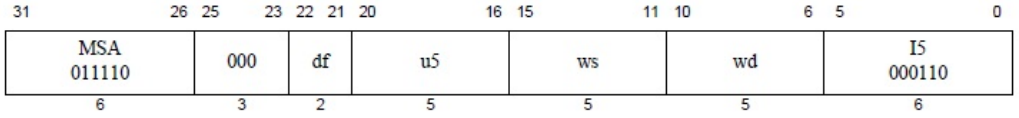

Figura 3.6: Instrucción aritmética con un dato inmediato

Instrucciones lógicas. En este caso para vectoriales y escalares las instrucciones tienen los mismos campos, y dichos campos están formados por la misma cantidad de bits. La única diferencia es que dichos campos están en diferentes posiciones. En el caso de las vectoriales como se puede apreciar en la Figura 3.7 los bits del 10 al 06 son los del registro del destino, mientras que en las escalares este campo va del bit 11 al 15.

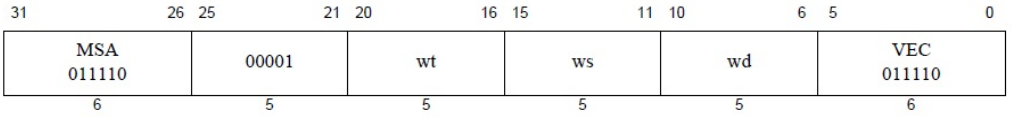

Figura 3.7: Instrucción lógica

Instrucciones de memoria. Para este tipo de instrucciones en el caso de las vectoriales también se incluye un campo de formato de datos (ver Figura 3.8), en las escalares este campo no existe. Otra diferencia es que el offset para las escalares es de 16 bits, mientras que en las vectoriales es de 10 bits. Las vectoriales también incluyen un campo de 4 bits fijo, que no incluyen las esclares. Todo lo demás es igual.

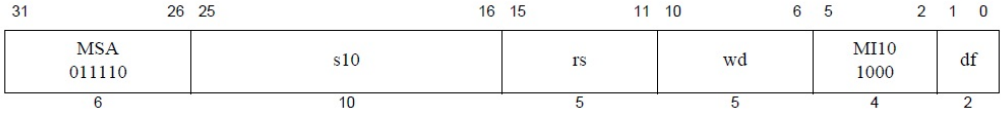

Figura 3.8: Instrucción de memoria

Otra de las cosas que es necesario destacar es que la ALU no dispone de registros HI y LO como en el caso de las ALU no vectoriales. Las instrucciones que en la arquitectura tradicional depositaban su resultado en estos registros, ahora lo depositan en el registro de destino que se le indica.

3.2. Ejemplos

A continuación se van a comparar dos códigos de ejemplo que realizan cuatro sumas, uno con instrucciones vectoriales y otro con escalares. Con instrucciones vectoriales, tan sólo se necesitan dos instrucciones `ld` para cargar los vectores, y una sola instrucción `addv.w` para realizar las sumas.

```
.data 0x10000000
A_app1: .word 1,0,1,0
B_app1: .word 0,1,0,1


.text 0x00400000

main:     la $t0, A_app1 //Se cargan las direcciones
          la $t1, B_app1
          ld $t2, 0($t0) //Se cargan los datos
          ld $t3, 0($t1)
          addv.w $t4, $t3, $t2 //Se realizan todas las multiplicaciones
```

Sin embargo con las instrucciones escalares se necesita un bucle para poder realizar las 4 sumas. El número de instrucciones aumenta notablemente, puesto que únicamente para cargar los datos se necesitan 8 instrucciones.

```
.data 0x10000000
A_app1: .word 1,0,1,0
B_app1: .word 0,1,0,1
```

```
.text 0x00400000

main:     la $t0, A_app1 //Se cargan las direcciones
          la $t1, B_app1
          li $t4, 4
salto:    lw $t2, 0($t0) //Se cargan los datos
          lw $t3, 0($t1)
          add $t5, $t2, $t3 //Se realiza una multiplicación
          addi $t0, $t0, 4 //Se actualizan las direcciones
          addi $t1, $t1, 4
          addi $t4, $t4, -1 //Se decrementa el contador
          bne $t4, $0, salto
```

Posteriormente en los ejemplos de uso se verá con más detalle una comparativa de un código escalar y uno vectorial con mediciones de tiempos de ejecución.

Capítulo 4

Herramientas de diseño

En este capítulo se describe con detalle las herramientas que han sido utilizadas a lo largo del trabajo, las más significativas y las que se van a tratar son: Xilinx ISE (*Integrated Software Enviroment*) *design suite*, Xilinx ISim y por último QTSpim.

4.1. Xilinx ISE

Es un software que permite la síntesis ("compilación") y el análisis de diseños en HDL (*Hardware Description Language*), en nuestro caso se ha desarrollado usando como lenguaje Verilog.

Gracias al ISE se pueden examinar los diagramas RTL (*Register Transfer Level*), simular el comportamiento del código que hemos desarrollado ante diferentes entradas y configurar el dispositivo sobre el que vamos a ejecutar el programa realizado.

También nos permite tener *schematics* (son una representación de la ruta de datos utilizando símbolos) para ver de qué manera organizamos y/o conectamos los diferentes módulos que hemos desarrollado con anterioridad. Gracias a ellos también podemos hacernos una idea global de cómo está nuestro diseño.

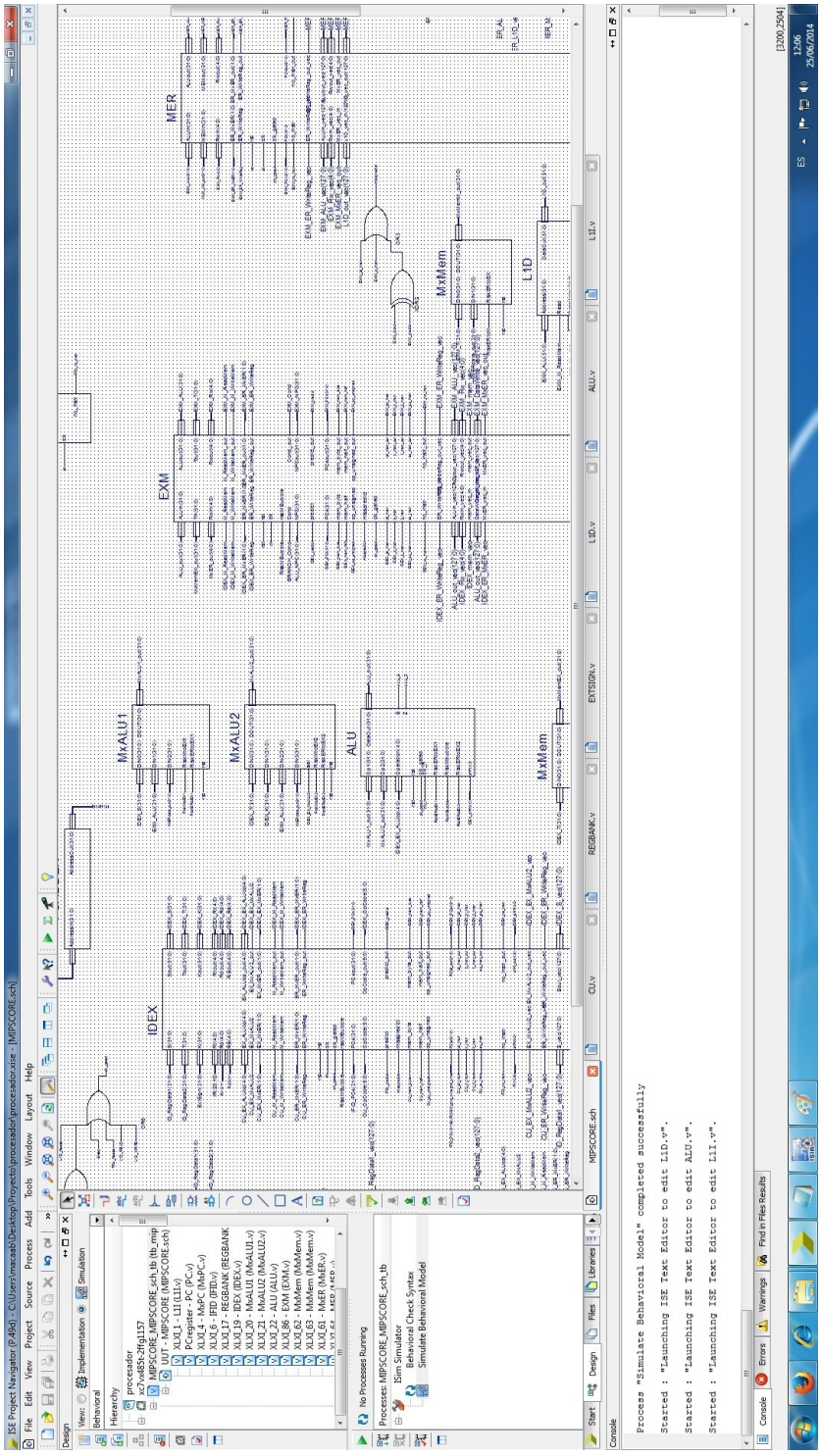

Figura 4.1: Muestra de un schematic

En la Figura 4.1 es una captura del Xilinx ISE. En ella se puede observar una ventana rectangular, que ocupa la mayor parte de la imagen, donde se pueden distinguir diferentes módulos como la ALU o el IDEX, eso es el *schematic*. La ventana inferior es la consola, por aquí se muestra información sobre el proyecto y su estado.

A la izquierda del *schematic*, se pueden apreciar dos ventanas (una encima de otra) de menor tamaño. La de arriba es la jerarquía con todos los ficheros que forman nuestro trabajo. La de abajo muestra lo que se puede hacer al fichero que tenemos seleccionado, análisis de sintaxis, etc. Los iconos que se ven entre estas dos ventanas y la del *schematic* son herramientas para modificar el fichero actual.

4.2. Xilinx Isim

El Xilinx Isim es la plataforma donde se pueden ver y manipular los resultados de la ejecución de nuestro diseño con respecto al tiempo transcurrido. Podemos ver el comportamiento global del sistema y el de cada módulo que hemos desarrollado por separado. También es posible acotarlo en un determinado periodo de tiempo y con unas determinadas circunstancias (valores de entradas determinados, etc.). Un ejemplo del resultado de una simulación es el mostrado en la Figura 4.2.

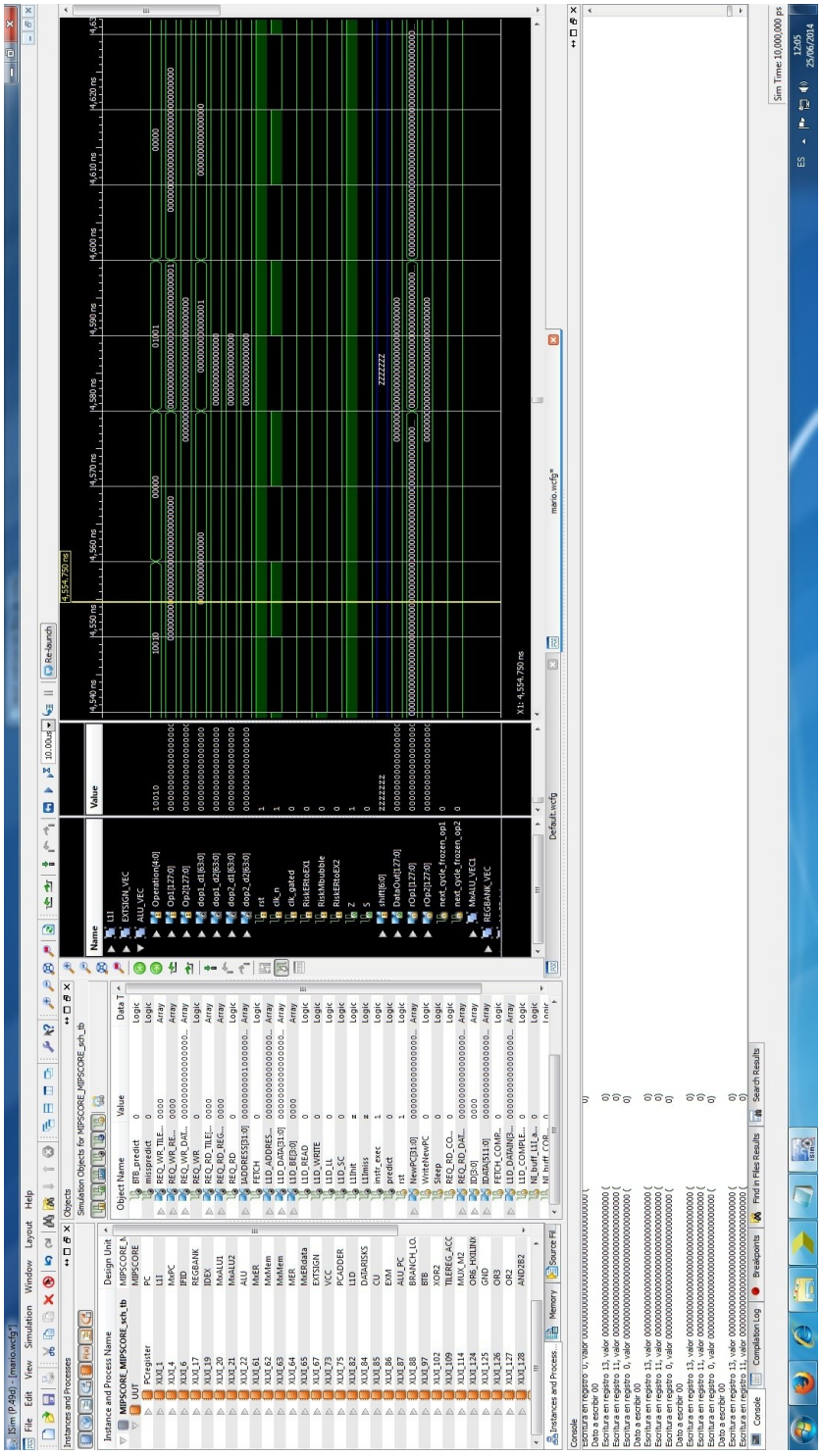

Figura 4.2: Ejemplo de Isim

En la Figura 4.2 se puede apreciar en la ventana más grande (con fondo negro) de la captura el estado en función del tiempo de algunas señales, módulos, registros, etc. que tenemos seleccionados. La ventana inferior al igual que en el Xilinx ISE se trata de la consola del programa.

En cuanto a las otras dos ventanas a la izquierda de la más grande nos sirven para seleccionar los módulos, señales y demás que queremos obtener una simulación de su comportamiento.

4.3. QTSpim

QTSpim es un simulador de un procesador MIPS escalar, en el trabajo ha servido para codificar las instrucciones. No se le ha dado demasiado uso puesto que solo codifica las instrucciones no vectoriales, con lo cual todas las codificaciones del nuevo juego de instrucciones vectorial han tenido que ser hechas a mano.

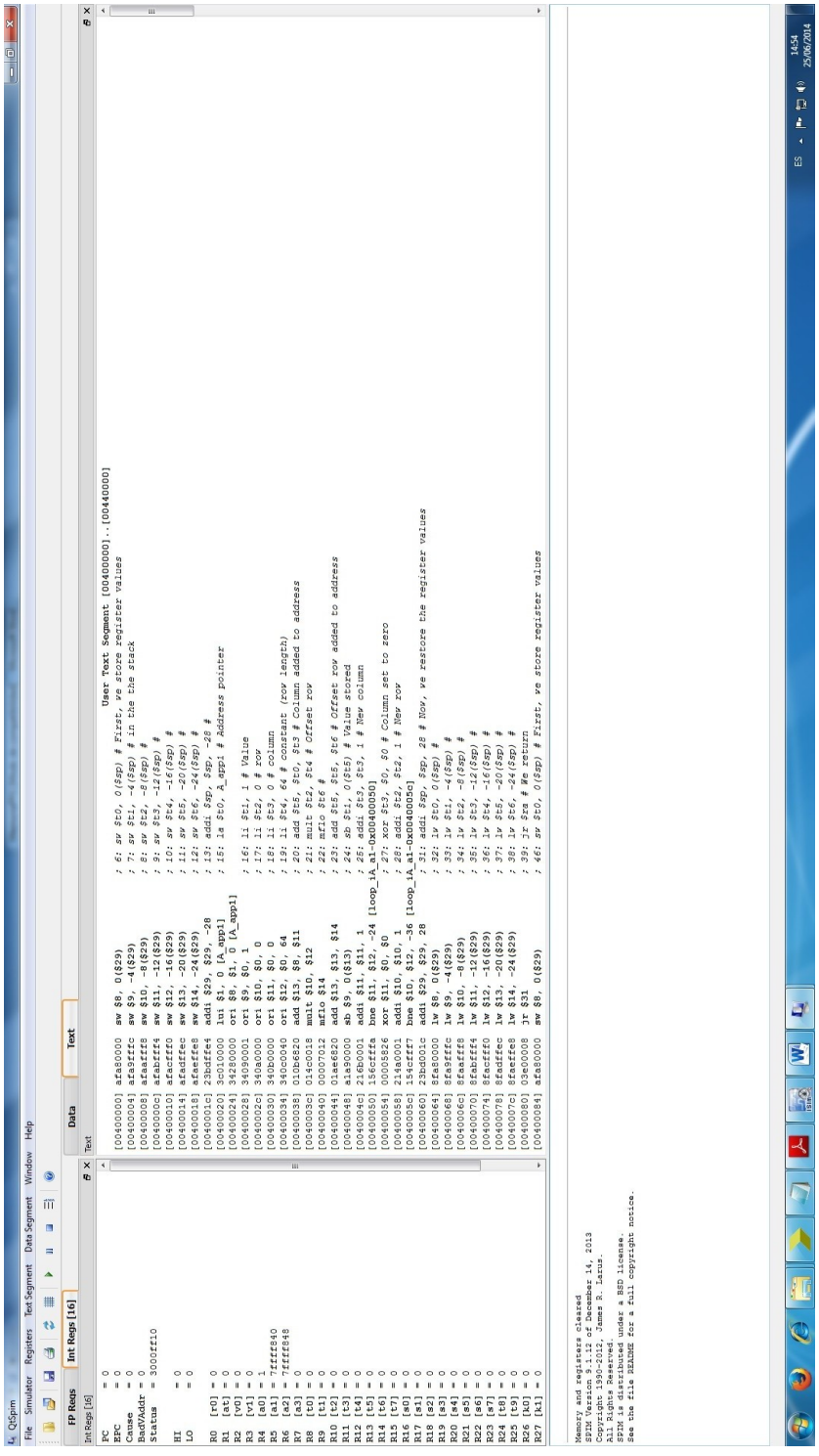

Figura 4.3: Ejecución con QTSpim

En la Figura 4.3 se puede observar una captura del QTSpim. En la ventana de mayor tamaño se pueden observar las instrucciones que le hemos introducido, su codificación y la dirección de memoria donde están ubicadas. La ventana de la izquierda nos muestra el valor de los registros. Por último la de abajo muestra información acerca de la ejecución y del propio QTSpim.

Capítulo 5

Implementación

En este capítulo se van a ir describiendo uno por uno los aspectos que se han ido modificando en la ruta de datos de la que se partía originariamente, y se explicarán también los nuevos módulos que se han introducido.

5.1. Banco de registros vectorial

La primera modificación que se introdujo en la ruta de datos fue incluir un nuevo banco de registros vectorial (ver Figura 5.1) en la misma etapa donde estaba el original. Dicho banco de registros es prácticamente igual al original salvo con la variación de que los registros son de 128 bits, que la entrada del dato a escribir (*DataWrite*) también es de 128 bits y las salidas de datos leídos (`RegOut1` y `RegOut2`) también son de 128 bits.

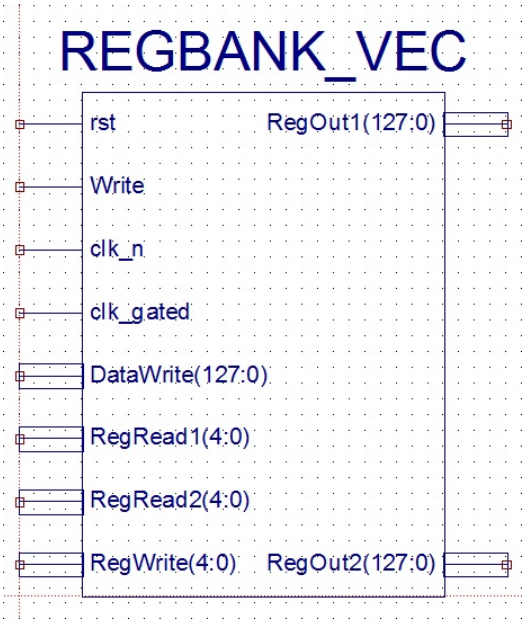

Figura 5.1: Banco de registros vectorial

Por todo lo demás es exactamente igual, las entradas `rst, clk_n y clk_gated` cumplen el mismo papel que en el banco de registros original (entrada para hacer un reinicio del módulo y entradas para controlar los pulsos de reloj). No obstante las señales que se conectan para el caso de la entrada `Write` es una señal diferente a la del banco de registros normal (aunque su funcionalidad sea la misma).

Esto es debido a que es necesario separar cuándo se ha de escribir en el banco de registros normales y cuándo en el vectorial. Las instrucciones de escritura normales activarán el `Write` normal y las vectoriales el vectorial. Realmente el código es bastante simple, cambia muy poco con respecto a la versión del banco de registros original, únicamente varía el segmento de código que aquí se incluye:

```
reg [127:0] BankRegister [31:0]; //Se crea el banco de registros
assign RegOut2 = BankRegister[RegRead2]; //Se asignan a las salidas

los datos solicitados

assign RegOut1 = BankRegister[RegRead1];
```

Tal y como está comentado en el mismo código, este fragmento crea el banco

de registros y asigna en los registros de salida los valores solicitados para la lectura.

Las entradas `RegRead1, RegRead2 y RegWrite` se utilizan para seleccionar el registro que queremos leer o escribir, en el banco de registros original tienen la misma funcionalidad.

5.2. ALU vectorial

En la ALU vectorial ha sido uno de los módulos donde más tiempo se ha invertido, puesto que se ha variado mucho a lo largo del trabajo. Desde los objetivos que se plantearon en un principio hasta el estado actual la transformación de la misma ha sido prácticamente total.

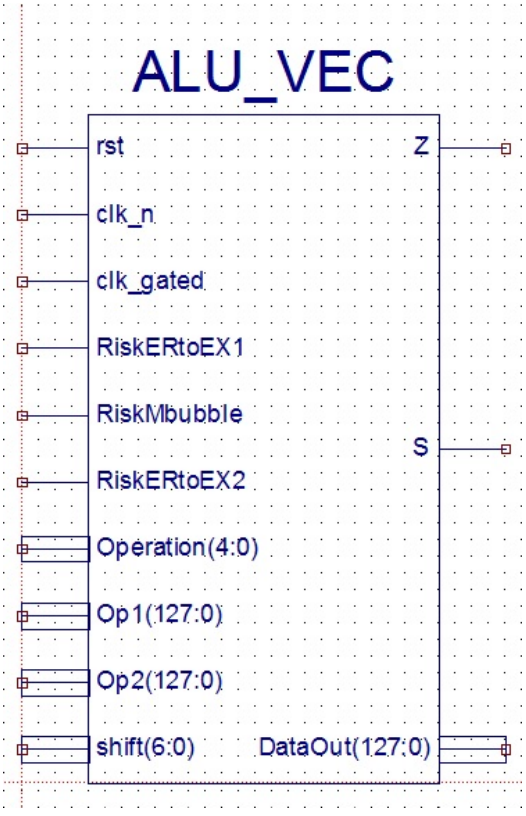

Figura 5.2: ALU vectorial

Las entradas y las salidas no varían mucho de la ALU escalar a ésta, la mayor diferencia son los operandos, la salida de datos de 128 bits, y el `shift` que como se puede apreciar en la Figura 5.2 en este caso es de 7 bits.

Tanto el `shift`, así como las entradas de riesgos (`RiskERtoEX1, RiskMbubble, RiskERtoEX2`), se han puesto para futuras ampliaciones que se quieran hacer para esta nueva ruta de datos vectorial, puesto que nosotros no utilizamos ninguna instrucción que haga uso del `shift`, y tal y como explicaremos en los siguientes apartados, tampoco utilizamos cortocircuitos para las instrucciones vectoriales (no obstante, si que se mantienen los cortocircuitos de la parte escalar).

El resto de entradas y salidas de la ALU vectorial realizan la misma función que en la ALU original. Las entradas `rst, Write, clk_n y clk_gated` se han explicado en el módulo anterior, en este realizan la misma función.

Respecto a las entradas `Op1` y `Op2` sirven para introducir los operandos en la ALU. La entrada `Operation` sirve para indicarle a la ALU la operación que tiene que realizar.

La salida `S` sirve para indicar si el valor de la salida de datos `DataOut` es menor que 0, mientras que `Z` sirve para indicar si el mismo valor de la salida de datos es igual a 0.

Para facilitar la lectura del código de la ALU vectorial se han creado diferentes conexiones, también sirven para que en función del formato de instrucción se utilicen un rango de bits u otros. Esto también se podía haber hecho acotando directamente el operador, pero como se ha dicho anteriormente se ha optado por esto debido a la mejor lectura del código. De esta manera es como se hace:

```
wire [63:0] dop1_d1 = rOp1[63:0]; //Se crea un wire de 64 bits al

cual se le asigna el valor de los primeros 64 bits del operando

rOp1

wire [63:0] dop1_d2 = rOp1[127:64];

wire [31:0] wop1_d1 = rOp1[31:0];

wire [31:0] wop1_d2 = rOp1[63:32];

wire [31:0] wop1_d3 = rOp1[95:64];

wire [31:0] wop1_d4 = rOp1[127:96];
```

```
wire [15:0] hop1_d1 = rOp1[15:0];

wire [15:0] hop1_d2 = rOp1[31:16];

wire [15:0] hop1_d3 = rOp1[47:32];

wire [15:0] hop1_d4 = rOp1[63:48];

wire [15:0] hop1_d5 = rOp1[79:64];

wire [15:0] hop1_d6 = rOp1[95:80];

wire [15:0] hop1_d7 = rOp1[111:96];

wire [15:0] hop1_d8 = rOp1[127:112];

wire [7:0] bop1_d1 = rOp1[7:0];

wire [7:0] bop1_d2 = rOp1[15:8];

wire [7:0] bop1_d3 = rOp1[23:16];

wire [7:0] bop1_d4 = rOp1[31:24];

wire [7:0] bop1_d5 = rOp1[39:32];

wire [7:0] bop1_d6 = rOp1[47:40];

wire [7:0] bop1_d7 = rOp1[55:48];

wire [7:0] bop1_d8 = rOp1[63:56];

wire [7:0] bop1_d9 = rOp1[71:64];

wire [7:0] bop1_d10 = rOp1[79:72];

wire [7:0] bop1_d11 = rOp1[87:80];

wire [7:0] bop1_d12 = rOp1[95:88];

wire [7:0] bop1_d13 = rOp1[103:96];

wire [7:0] bop1_d14 = rOp1[111:104];
```

```
wire [7:0] bop1_d15 = rOp1[119:112];

wire [7:0] bop1_d16 = rOp1[127:120];
```

El código que se ha incluido lo realiza solo para el primer operando, pero también se hace para el segundo en el código íntegro. El siguiente fragmento de código que se va a incluir es la realización de una instrucción add en sus distintos formatos (`.d`, `.w`, `.h`, `.b`) para que se vea de qué manera están realizadas las instrucciones vectoriales:

```
case (Operation)

  5'b11000: begin DataOut[63:0] <= dop1_d1 + dop2_d1; // addv.d

                  DataOut[127:64] <= dop1_d2 + dop2_d2;

            end

  5'b00011: begin DataOut[31:0] <= wop1_d1 + wop2_d1; // addv.w

                  DataOut[63:32] <= wop1_d2 + wop2_d2;

                  DataOut[95:64] <= wop1_d3 + wop2_d3;

                  DataOut[127:96] <= wop1_d4 + wop2_d4;

            end

  5'b00010: begin DataOut[15:0] <= hop1_d1 + hop2_d1; // addv.h

                  DataOut[31:16] <= hop1_d2 + hop2_d2;

                  DataOut[47:32] <= hop1_d3 + hop2_d3;

                  DataOut[63:48] <= hop1_d4 + hop2_d4;

                  DataOut[79:64] <= hop1_d5 + hop2_d5;

                  DataOut[95:80] <= hop1_d6 + hop2_d6;
```

```
                DataOut[111:96] <= hop1_d7 + hop2_d7;
                DataOut[127:112] <= hop1_d8 + hop2_d8;
            end
 5'b00001: begin DataOut[7:0] <= bop1_d1 + bop2_d1; // addv.b
                DataOut[15:8] <= bop1_d2 + bop2_d2;
                DataOut[23:16] <= bop1_d3 + bop2_d3;
                DataOut[31:24] <= bop1_d4 + bop2_d4;
                DataOut[39:32] <= bop1_d5 + bop2_d5;
                DataOut[47:40] <= bop1_d6 + bop2_d6;
                DataOut[55:48] <= bop1_d7 + bop2_d7;
                DataOut[63:56] <= bop1_d8 + bop2_d8;
                DataOut[71:64] <= bop1_d9 + bop2_d1;
                DataOut[79:72] <= bop1_d10 + bop2_d10;
                DataOut[87:80] <= bop1_d11 + bop2_d11;
                DataOut[95:88] <= bop1_d12 + bop2_d12;
                DataOut[103:96] <= bop1_d13 + bop2_d13;
                DataOut[111:104] <= bop1_d14 + bop2_d14;
                DataOut[119:112] <= bop1_d15 + bop2_d15;
                DataOut[127:120]  <= bop1_d16 + bop2_d16;
            end
```

El juego de instrucciones (con sus correspondientes modos de funcionamiento) que se ha decidido implementar ha sido el siguiente: `addv`, `subv`, `mulv`, `addvi`, `orv`, `andv`, `st`, `lv` y por último una instrucción adicional que no estaba

contemplada en el juego de instrucciones del MIPS SIMD: `addm`.

A continuación se van a describir una a una cómo funcionan de manera conceptual cada una de las instrucciones del nuevo juego de instrucciones vectorial que se ha implementado.

La primera que se va a describir es `addv`, la suma vectorial, la sintaxis para referirnos a la misma según sus operandos va a ser la siguiente: `addv wd,ws,wt`. Su funcionamiento es el mostrado aquí:

```
ADDV.B //En este modo la suma se realiza byte a byte

for i in 0 .. WRLEN/8-1 //WRLEN es la longitud del

registro (128 bits)

WR[wd]8i+7..8i <- WR[ws]8i+7..8i + WR[wt]8i+7..8i

endfor

ADDV.H //En este modo la suma se realiza halfword a halfword

for i in 0 .. WRLEN/16-1

WR[wd]16i+15..16i <- WR[ws]16i+15..16i + WR[wt]16i+15..16i

endfor

ADDV.W //En este modo la suma se realiza word a word

for i in 0 .. WRLEN/32-1

WR[wd]32i+31..32i <- WR[ws]32i+31..32i + WR[wt]32i+31..32i

endfor

ADDV.D //En este modo la suma se realiza doubleword a doubleword

for i in 0 .. WRLEN/64-1

WR[wd]64i+63..64i <- WR[ws]64i+63..64i + WR[wt]64i+63..64i

endfor
```

La siguiente no es otra que la resta vectorial o `subv`, en este caso la sintaxis es igual que en la `addv` cambiando el nombre de la propia instrucción, pero los registros son iguales, su sintaxis es esta: `subv wd,ws,wt`.

```
SUBV.B //La resta en este modo se realiza byte a byte

for i in 0 .. WRLEN/8-1 //WRLEN es la longitud del

registro (128 bits)

WR[wd]8i+7..8i <- WR[ws]8i+7..8i - WR[wt]8i+7..8i

endfor

SUBV.H //La resta en este modo se realiza halfword a halfword

for i in 0 .. WRLEN/16-1

WR[wd]16i+15..16i <- WR[ws]16i+15..16i - WR[wt]16i+15..16i

endfor

SUBV.W //La resta en este modo se realiza word a word

for i in 0 .. WRLEN/32-1

WR[wd]32i+31..32i <- WR[ws]32i+31..32i - WR[wt]32i+31..32i

endfor

SUBV.D //La resta en este modo se realiza doubleword a doubleword

for i in 0 .. WRLEN/64-1

WR[wd]64i+63..64i <- WR[ws]64i+63..64i - WR[wt]64i+63..64i

endfor
```

El caso de la `mulv` es posiblemente el más distinto con respecto a la operación no vectorial, puesto que no se disponen de registros HI y LO en la ALU vectorial, y la sintaxis de la instrucción tiene tres registros y no dos como en la escalar, `mulv wd,ws,wt`.

```
MULV.B //La multiplicacion se realiza byte a byte y se
almacena en el byte correspondiente del registro de destino
for i in 0 .. WRLEN/8-1 //WRLEN es la longitud del
registro (128 bits)
WR[wd]8i+7..8i <- WR[ws]8i+7..8i * WR[wt]8i+7..8i
endfor
MULV.H //La multiplicacion se realiza halfword a halfword y se
almacena en el halfword correspondiente del registro de destino
for i in 0 .. WRLEN/16-1
WR[wd]16i+15..16i <- WR[ws]16i+15..16i * WR[wt]16i+15..16i
endfor
MULV.W//La multiplicacion se realiza word a word y se
almacena en el word correspondiente del registro de destino
for i in 0 .. WRLEN/32-1
WR[wd]32i+31..32i <- WR[ws]32i+31..32i * WR[wt]32i+31..32i
endfor
MULV.D//La multiplicacion se realiza doubleword a doubleword y
se almacena en el byte correspondiente del registro de destino
for i in 0 .. WRLEN/64-1
WR[wd]64i+63..64i <- WR[ws]64i+63..64i * WR[wt]64i+63..64i
endfor
```

La `addvi` es muy parecida a la `addv` salvo que en vez de utilizar dos registros

para extraer los datos utiliza un único registro y un dato inmediato (que con anterioridad ha pasado por la extensión de signo vectorial). Su sintaxis es la siguiente: `addvi wd,ws,u5`.

```
ADDVI.B //El inmediato se suma a cada byte de manera independiente

for i in 0 .. WRLEN/8-1 //WRLEN es la longitud del

registro (128 bits)

WR[wd]8i+7..8i <- WR[ws]8i+7..8i + u5

endfor

ADDVI.H //El inmediato se suma a cada halfword de manera

independiente

for i in 0 .. WRLEN/16-1

WR[wd]16i+15..16i <- WR[ws]16i+15..16i + u5

endfor

ADDVI.W //El inmediato se suma a cada word de manera independiente

for i in 0 .. WRLEN/32-1

WR[wd]32i+31..32i <- WR[ws]32i+31..32i + u5

endfor

ADDVI.D //El inmediato se suma a cada doubleword de manera

independiente

for i in 0 .. WRLEN/64-1

WR[wd]64i+63..64i <- WR[ws]64i+63..64i + u5

endfor
```

El caso de la `or.v` y la `and.v` son diferentes al resto, puesto que no distinguen entre formatos vectoriales, cuando se realiza una de estas operaciones se

realiza para el conjunto del registro y se hace como una operación global. Las sintaxis de las instrucciones son estas: `or.v wd,ws,wt`, `and.v wd,ws,wt`.

```
WR[wd] <- WR[ws] or WR[wt]
```

```
WR[wd] <- WR[ws] and WR[wt]
```

Una vez tratadas todas las instrucciones que no tienen nada que ver con la lectura o escritura de memoria (salvo la `addm`, que por tratarse de un caso especial reservamos para el final) pasamos a las que sí que tienen que ver: la instrucción `st` (escritura vectorial) y la instrucción `lv` (carga vectorial).

Para el caso de la `load`, pese a que en la arquitectura SIMD también se distinguen los diferentes formatos vectoriales, solo se ha considerado necesaria la implementación parcial de ellos. Teniendo en cuenta la sintaxis: `ld wd,s10(rs)`, en el registro `wd` (de 128 bits) se carga lo que hay en la dirección que está en `rs`.

Por otro lado, en el caso de la `store` sí que se han implementado diferentes modos de funcionamiento de cara a optimizar el resultado del problema de la multiplicación de matrices. Siguiendo la sintaxis: `st wd,s10(rs)` la primera implementación que se ha llevado a cabo es con el registro entero, es decir, en `wd` (de 128 bits) se guarda en la dirección que hay en `rs`, y la segunda y última es la correspondiente al `.b`. En esta implementación se almacena el primer byte del registro `wd` en la dirección albergada en `rs`.

En último lugar está la `addm`, instrucción de la que no se ha dicho prácticamente nada hasta el momento. Esta instrucción surgió de una necesidad al realizar el problema que se expondrá después, la multiplicación de matrices.

Como bien es sabido, para multiplicar matrices es necesario multiplicar los elementos de una fila (f) de la matriz A por los elementos de una columna (c) de la matriz B, y sumar el resultado. Es decir, f1.1*c1.1+f1.2*c1.2+f1.3*c1.3. Como se puede intuir el realizar las multiplicaciones se puede hacer con la `mulv`, pero no sumar los productos de ellas.

En consenso con los directores del trabajo se decidió que la mejor opción para resolver dicho problema era diseñar e implementar una nueva instrucción.

Esta instrucción dispone de los diferentes formatos vectoriales, y su sintaxis es la siguiente: `addm wd, ws`. Su funcionamiento es el siguiente:

```
ADDM.B //Se calcula la suma de todos los bytes del registro ws
```

```
y se almacena en el registro wd
for i in 0 .. WRLEN/8-1 //WRLEN es la longitud del
registro (128 bits)
WR[wd] <- WR[wd] + WR[ws]8i+7..8i
endfor
ADDM.H //Se calcula la suma de todos los halfword del registro ws
y se almacena en el registro wd
for i in 0 .. WRLEN/16-1
WR[wd] <- WR[wd] + WR[ws]16i+15..16i
endfor
ADDM.W //Se calcula la suma de todos los words del registro ws
y se almacena en el registro wd
for i in 0 .. WRLEN/32-1
WR[wd] <- WR[wd] + WR[ws]32i+31..32i
endfor
ADDM.D //Se calcula la suma de todos los doublewords del registro ws
y se almacena en el registro wd
for i in 0 .. WRLEN/64-1
WR[wd] <- WR[wd] + WR[ws]64i+63..64i
endfor
```

5.3. Extensión de signo vectorial

La extensión de signo se tuvo que replicar también y crear una extensión vectorial, puesto que con la que había no era suficiente, ésta solo extendía los datos inmediatos hasta 32 bits, y era necesario extenderlos hasta 128. La única variación es esa, y evidentemente la salida es de 128 bits, como se puede observar en la Figura 5.3. La manera de hacerlo es exactamente igual que con la no vectorial.

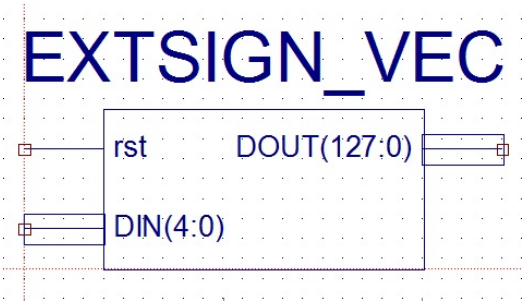

Figura 5.3: Extensión de signo vectorial

Las entradas son las mismas, y su funcionalidad indéntica. `DIN` es por donde se introduce el dato al cual se le quiere aplicar la extensión de signo. Con respecto a `rst` su funcionalidad ya ha sido explicada y es igual para todos los módulos.

5.4. Unidad de control

También fue necesario modificar la unidad de control, hasta el punto de que prácticamente se duplicó la extensión del código de la misma. Esto es debido a que se debieron crear nuevas señales para que las instrucciones funcionasen con los elementos vectoriales: nueva ALU, banco de registros vectorial, nuevos multiplexores, etc. Como se puede observar en la Figura 5.4 la cantidad de salidas de la CU es muy elevada, cosa que dificultó en gran medida el código desarrollado.

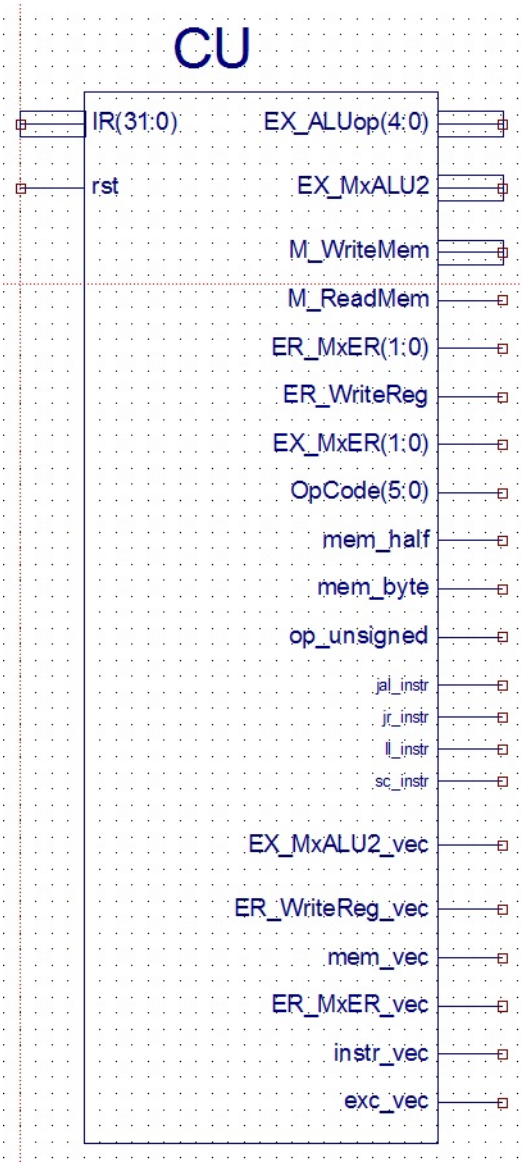

Figura 5.4: Unidad de control

Dichas señales debían ser activadas en determinadas instrucciones vectoriales y desactivadas para el resto. A la par que las señales que ya estaban creadas también fue necesario tenerlas en cuenta para las nuevas instrucciones. Evidentemente también hubo que pasar las señales nuevas a lo largo de las diferentes etapas hasta llegar al módulo donde eran necesarias.

Básicamente sirven para activar, desactivar o modificar el funcionamiento de los módulos en función de la instrucción que se ha de ejecutar, la cual se

reconoce por el dato que entra por la entrada `IR`.

Otros inconvenientes que dificultaron el código de la unidad de control fueron los formatos de las nuevas instrucciones que se han explicado anteriormente. Debido a esto, el sistema para averiguar la instrucción que se estaba ejecutando tuvo que sufrir variaciones, creándose en el camino diferentes cables para identificar los campos que distinguen si se trata de una instrucción o de otra:

```
wire [5:0] op; //Se crean los cables correspondientes a cada campo

wire [5:0] funct; // que se quiere asignar

wire [4:0] logi; //Se crean con una longitud en bits

wire [1:0] tipo;

wire [3:0] mem;

wire [1:0] tipo_store;

wire tipo_1_b;

assign op = IR[31:26]; //Aqui se asigna de donde tienen que coger

assign funct = IR[5:0]; //el valor para mostrar, en este caso de

assign logi = IR[25:21];//unos determinados bits de IR

assign arit = IR[25:23];

assign tipo = IR[22:21];

assign mem = IR[5:2];

assign tipo_store = IR[1:0];

assign tipo_1_b = IR[21];
```

Esto en un principio no tendría que suponer mayor complejidad que la ya dicha anteriormente, puesto que tanto el banco de registros como la ALU y la extensión de signo se encuentran separados para ambos casos. Pero para el caso de las instrucciones de memoria vectoriales es necesario acceder

al banco de registro no vectorial y la extensión de signo vectorial, con lo cual fue necesario modificarlos para añadir nuevas entradas y modificar su comportamiento.

Para poder realizar esto fue necesario crear una nueva señal desde la unidad de control que en función de la instrucción leída se activase o no para indicar al banco de registros y a la extensión de signo vectorial de qué tipo de instrucción se trataba.

Para conseguir un correcto funcionamiento de las instrucciones vectoriales se tuvo que desarrollar un mecanismo para desactivar el sistema de excepciones que había implementado, mientras se estuviera ejecutando cualquiera de las nuevas instrucciones (salvo `load` y `store` vectoriales, que a todos los efectos son instrucciones normales, solo que los datos fluyen en el banco de registros vectorial en lugar del normal).

Este mecanismo consiste en otra señal nueva que se activaría en la unidad de control cuando una instrucción vectorial se fuera a ejecutar para indicarle al módulo donde se manejan las excepciones y los riesgos de datos que si tenía que activar alguna excepción, al tratarse de una vectorial dicha excepción fuese falsa y no lo llevase a cabo.

Por cada nueva instrucción implementada se realizó un nuevo caso dentro de la unidad de control para determinar los valores que se daban a las salidas. Para el caso de las store vectoriales se creó una salida aposta (mem_vec) que se activaba en el caso de que se fuera a escribir el vector de manera íntegra. En el caso de `st.b` se activan tanto mem_vec como mem_byte.

5.5. Modificaciones de la ruta de datos

En este apartado se van a analizar las modificaciones que se han realizado a la propia ruta de datos que se nos proporcionaba en un principio, es decir, modificando los elementos que había en ella (no como hasta ahora que se habían analizado los nuevos elementos introducidos en la misma, a excepción de la CU a la que se le ha dedicado un apartado por su mayor extensión).

Los registros de etapa IDEX, EXM, MER han sido modificados para pasar las nuevas señales. Por cada señal en cada uno de ellos se ha tenido que crear una entrada y una salida, y modificar el funcionamiento de los mismos para incluirlas.

El módulo DATARISK también se ha tenido que modificar para deshabilitar las excepciones mientras se ejecutaban instrucciones vectoriales. Para ello

se ha añadido una entrada que se activa cuando se ejecuta una instrucción vectorial desde la CU. El nuevo comportamiento es el siguiente:

```
// risk ERtoEX: instruction in ER produces, instr in EX writes in
memory, both registers the same
assign riskERtoEX = WrRegER & WrMemEX & (RtEX == RwER) & ~instr_vec;

// risk ERtoEX1: instruction in ER produces, both registers the same
assign riskERtoEX1 =  WrRegER & (RsEX == RwER) & ~instr_vec;

// risk ERtoEX2: instruction in ER produces, MxALU2 must use T
input, both registers the same
assign riskERtoEX2 = WrRegER & (MxALU2 == 0) & (RtEX == RwER)
& ~instr_vec;

// risk ERtoM: instruction in ER produces, instruction in M writes
to memory, both registers the same (Rt in M is RwM)
assign riskERtoM = WrRegER & WrMemM & (RwM == RwER) & ~instr_vec;

// risk MtoEX1: instruction in M produces, instruction in M is not
a load, both registers the same
assign riskMtoEX1 = WrRegM & (RdMemM == 0) & (RsEX == RwM)
& ~instr_vec;

// risk MtoEX2: instruction in M produces, instruction in M is not
a load, MxALU2 must use T input, both registers the same
```

```
assign riskMtoEX2 = WrRegM & (RdMemM == 0) & (MxALU2 == 0) &

(RtEX == RwM) & ~instr_vec;


// risk MBubble: instruction in M reads from memory, both

registers are the same (Rt in M is RwM)

assign riskMBubble = (sc_instr_M | RdMemM) & ((RsEX == RwM) |

(((MxALU2 == 0) & (RtEX == RwM)))) & ~instr_vec;
```

Tal y como se ha explicado en el apartado de la CU, el banco de registros y la extensión de signo han tenido que ser modificados para que funcionasen acorde a la lógica del nuevo procesador. Para ello se han añadido entradas y modificado su código.

También se han tenido que modificar ambas caches para poder introducir las instrucciones que se querían ejecutar en las diferentes pruebas y los datos que eran necesarios de antemano para poder llevarlas a cabo.

No se ha mencionado nada al respecto con anterioridad, pero cuando se introducía un nuevo módulo a la ruta de datos, o se creaban entradas y/o salidas en uno existente, había que cablearlo con el resto de la ruta ya existente, trabajo que no ha sido para nada despreciable.

5.6. Multiplexores

En la ruta de datos también ha sido necesario introducir distintos multiplexores para conseguir la funcionalidad deseada. En concreto, se han introducido tres multiplexores, dos antes de la ALU vectorial, y uno en la última etapa.

El primer multiplexor que se va a describir es uno de los que están antes de la ALU, en concreto es el MxALU_VEC1. La finalidad de este es hacer que la ruta de datos funcione con excepciones y cortocircuitos. Como se ha dicho ya anteriormente la parte vectorial no funciona de este modo, pero se ha puesto para poder realizar la ampliación para que funcione de manera simple.

El segundo multiplexor, también localizado justo antes de la ALU, se llama MxALU_VEC2, y su función es similar, sirve para lo mismo que el expli-

cado anteriormente. No obstante, este también es necesario para realizar operaciones de lectura, escritura y cualquier operación que tenga algún dato inmediato.

Por último, el tercer multiplexor situado en la etapa ER sirve para diferenciar cuándo se tiene que llevar al banco de registros y al segundo multiplexor el dato que le está llegando desde la ALU vectorial o desde la L1D en función de si es una operación que escribe o lee de memoria o no.

5.7. Síntesis de la ruta de datos

En esta sección se va a realizar una comparativa de los costes (a nivel de síntesis) que ha tenido cada uno de los procesadores. Para la comparativa se ha eliminado el coste que suponían las caches LI y LD puesto que para ambos casos el coste de estas se supone el mismo.

Estos son los resultados de la síntesis realizada para el procesador escalar (los números de la derecha son la cantidad usada de cada elemento):

```
=========================================================================
HDL Synthesis Report

Macro Statistics
# RAMs                                                  : 5
 16x2-bit dual-port RAM                                 : 1
 16x32-bit dual-port RAM                                : 2
 32x32-bit dual-port RAM                                : 2
# Multipliers                                           : 1
 32x32-bit multiplier                                   : 1
# Adders/Subtractors                                    : 5
 32-bit adder                                           : 2
 32-bit addsub                                          : 1
 32-bit subtractor                                      : 2
# Registers                                             : 72
 1-bit register                                         : 36
 16-bit register                                        : 1
 2-bit register                                         : 4
 32-bit register                                        : 18
 4-bit register                                         : 4
 5-bit register                                         : 7
 6-bit register                                         : 1
 64-bit register                                        : 1
```

```
# Latches                                              : 121
 1-bit latch                                           : 121
# Comparators                                          : 8
 32-bit comparator equal                               : 2
 32-bit comparator greater                             : 1
 5-bit comparator equal                                : 5
# Multiplexers                                         : 155
 1-bit 16-to-1 multiplexer                             : 2
 1-bit 2-to-1 multiplexer                              : 36
 1-bit 3-to-1 multiplexer                              : 37
 2-bit 2-to-1 multiplexer                              : 13
 32-bit 2-to-1 multiplexer                             : 59
 4-bit 2-to-1 multiplexer                              : 1
 5-bit 2-to-1 multiplexer                              : 6
 6-bit 2-to-1 multiplexer                              : 1
# Logic shifters                                       : 4
 32-bit shifter logical left                           : 2
 32-bit shifter logical right                          : 2
# Xors                                                 : 1
 32-bit xor2                                           : 1
=========================================================================

=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                                 : 5
 16x2-bit dual-port distributed RAM                    : 1
 16x32-bit dual-port distributed RAM                   : 2
 32x32-bit dual-port distributed RAM                   : 2
# Multipliers                                          : 1
 32x32-bit registered multiplier                       : 1
# Adders/Subtractors                                   : 5
 32-bit adder                                          : 2
 32-bit addsub                                         : 1
 32-bit subtractor                                     : 2
# Registers                                            : 693
 Flip-Flops                                            : 693
# Comparators                                          : 8
 32-bit comparator equal                               : 2
 32-bit comparator greater                             : 1
 5-bit comparator equal                                : 5
# Multiplexers                                         : 153
```

```
 1-bit 16-to-1 multiplexer                                : 2
 1-bit 2-to-1 multiplexer                                 : 34
 1-bit 3-to-1 multiplexer                                 : 37
 2-bit 2-to-1 multiplexer                                 : 13
 32-bit 2-to-1 multiplexer                                : 59
 4-bit 2-to-1 multiplexer                                 : 1
 5-bit 2-to-1 multiplexer                                 : 6
 6-bit 2-to-1 multiplexer                                 : 1
# Logic shifters                                          : 4
 32-bit shifter logical left                              : 2
 32-bit shifter logical right                             : 2
# Xors                                                    : 1
 32-bit xor2                                              : 1
=========================================================================
```

Estos son los de la síntesis de la ruta de datos vectorial, como se puede apreciar los costes son algo mayores:

```
 =========================================================================
 HDL Synthesis Report

 Macro Statistics
 # RAMs                                                   : 7
  16x2-bit dual-port RAM                                  : 1
  16x32-bit dual-port RAM                                 : 2
  32x32-bit dual-port RAM                                 : 4
 # Multipliers                                            : 31
  16x16-bit multiplier                                    : 8
  32x32-bit multiplier                                    : 5
  64x64-bit multiplier                                    : 2
  8x8-bit multiplier                                      : 16
 # Adders/Subtractors                                     : 99
  16-bit adder                                            : 16
  16-bit subtractor                                       : 8
  32-bit adder                                            : 10
  32-bit addsub                                           : 1
  32-bit subtractor                                       : 6
  33-bit adder                                            : 1
  34-bit adder                                            : 1
  35-bit adder                                            : 1
  64-bit adder                                            : 4
  64-bit subtractor                                       : 2
  65-bit adder                                            : 1
```

```
 8-bit adder                                          : 32
 8-bit subtractor                                     : 16
# Registers                                           : 95
 1-bit register                                       : 47
 128-bit register                                     : 8
 16-bit register                                      : 1
 2-bit register                                       : 4
 32-bit register                                      : 18
 4-bit register                                       : 4
 4096-bit register                                    : 1
 5-bit register                                       : 10
 6-bit register                                       : 1
 64-bit register                                      : 1
# Latches                                             : 382
 1-bit latch                                          : 382
# Comparators                                         : 8
 32-bit comparator equal                              : 2
 32-bit comparator greater                            : 1
 5-bit comparator equal                               : 5
# Multiplexers                                        : 234
 1-bit 16-to-1 multiplexer                            : 2
 1-bit 2-to-1 multiplexer                             : 42
 1-bit 3-to-1 multiplexer                             : 37
 128-bit 2-to-1 multiplexer                           : 61
 128-bit 32-to-1 multiplexer                          : 2
 2-bit 2-to-1 multiplexer                             : 16
 32-bit 2-to-1 multiplexer                            : 64
 4-bit 2-to-1 multiplexer                             : 1
 5-bit 2-to-1 multiplexer                             : 8
 6-bit 2-to-1 multiplexer                             : 1
# Logic shifters                                      : 4
 32-bit shifter logical left                          : 2
 32-bit shifter logical right                         : 2
# Xors                                                : 1
 32-bit xor2                                          : 1
=========================================================================

=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                                : 7
 16x2-bit dual-port distributed RAM                   : 1
```

```
 16x32-bit dual-port distributed RAM                   : 2
 32x32-bit dual-port distributed RAM                   : 4
# Multipliers                                          : 31
 16x16-bit multiplier                                  : 8
 32x32-bit multiplier                                  : 4
 32x32-bit registered multiplier                       : 1
 64x64-bit multiplier                                  : 2
 8x8-bit multiplier                                    : 16
# Adders/Subtractors                                   : 96
 16-bit adder                                          : 16
 16-bit subtractor                                     : 8
 32-bit adder                                          : 10
 32-bit addsub                                         : 1
 32-bit subtractor                                     : 6
 64-bit adder                                          : 4
 64-bit subtractor                                     : 2
 65-bit adder                                          : 1
 8-bit adder                                           : 32
 8-bit subtractor                                      : 16
# Adder Trees                                          : 1
 35-bit / 4-inputs adder tree                          : 1
# Registers                                            : 5839
 Flip-Flops                                            : 5839
# Comparators                                          : 8
 32-bit comparator equal                               : 2
 32-bit comparator greater                             : 1
 5-bit comparator equal                                : 5
# Multiplexers                                         : 232
 1-bit 16-to-1 multiplexer                             : 2
 1-bit 2-to-1 multiplexer                              : 40
 1-bit 3-to-1 multiplexer                              : 37
 128-bit 2-to-1 multiplexer                            : 61
 128-bit 32-to-1 multiplexer                           : 2
 2-bit 2-to-1 multiplexer                              : 16
 32-bit 2-to-1 multiplexer                             : 64
 4-bit 2-to-1 multiplexer                              : 1
 5-bit 2-to-1 multiplexer                              : 8
 6-bit 2-to-1 multiplexer                              : 1
# Logic shifters                                       : 4
 32-bit shifter logical left                           : 2
 32-bit shifter logical right                          : 2
# Xors                                                 : 1
 32-bit xor2                                           : 1
```

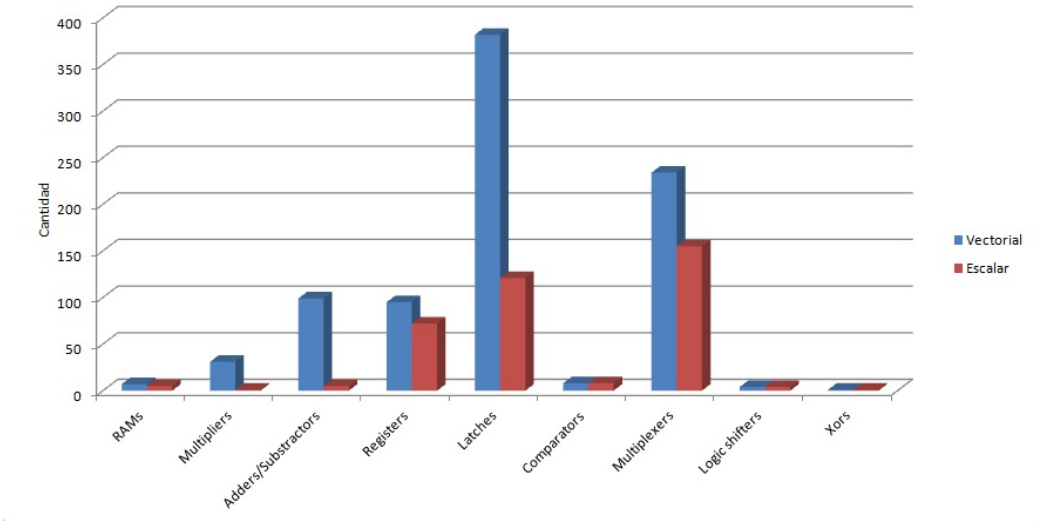

Figura 5.5: Comparativa vectorial vs. escalar

==========

Para poder observarlo con mayor claridad se ha realizado una gráfica comparativa entre ambos costes (Figura 5.5). Dicha gráfica entra menos en detalle que el resultado de la propia síntesis mostrada anteriormente, sin embargo se puede observar de una manera más simple los sobrecostes del procesador vectorial.

Como era de esperar el procesador vectorial en general tiene un coste mayor, en el mejor de los casos el mismo. Esto es debido a que para realizar la ruta de datos vectorial se han tenido que añadir y/o modificar elementos a la ruta de datos escalar, pero en ningún caso se ha podido simplificar o reducir la cantidad de módulos de la ruta original.

La diferencia es más significativa en el caso de los *latches* que triplica el número de los que tenía la ruta escalar, en el de los *adders/substractors* que de haber 5 en la ruta escalar pasa a tener 99 en la nueva ruta vectorial, y en el de los multiplexores, la nueva ruta cuenta con 80 más con respecto a la escalar.

Sin embargo el número de RAMs, comparadores, y desplazadores lógicos es prácticamente el mismo en ambas rutas de datos.

Capítulo 6

Ejemplo de uso

6.1. Comparación escalar/vectorial

A continuación se va a realizar una comparación entre escalar y vectorial. Para ello se van a ejecutar dos códigos que realizan lo mismo, pero uno con instrucciones escalares, y otro con el nuevo juego de instrucciones vectorial que acabamos de desarrollar. Ambos códigos realizan 4 multiplicaciones. Para el caso de las vectoriales el código es el siguiente:

```
.data 0x10000000
A_app1: .word 1,2,3,4
B_app1: .word 0,1,0,1


.text 0x00400000

main:     la $t0, A_app1 //Se cargan las direcciones
          la $t1, B_app1
          ld $t2, 0($t0) //Se cargan los datos
          ld $t3, 0($t1)
          mulv.w $t4, $t3, $t2 //Se realizan todas las multiplicaciones
```

Para las escalares, el código es algo más largo y complejo comparado con el vectorial. El código en cuestión es este:

```
.data 0x10000000
A_app1: .word 1,2,3,4
B_app1: .word 0,1,0,1
```

```
.text 0x00400000

main:     la $t0, A_app1 //Se cargan las direcciones
          la $t1, B_app1
          li $t4, 4
salto:    lw $t2, 0($t0) //Se cargan los datos
          lw $t3, 0($t1)
          mult $t2, $t3 //Se realiza una multiplicación
          addi $t0, $t0, 4 //Se actualizan las direcciones
          addi $t1, $t1, 4
          addi $t4, $t4, -1 //Se decrementa el contador
          bne $t4, $0, salto
```

El tiempo que ha tardado en ejecutarse el código escalar es de 780 ns, mientras que el vectorial ha tardado 300 ns. El resultado es bastante esclarecedor, viendo los resultados se justifica la creación y utilización de la parte vectorial, puesto que el código vectorial tarda menos de la mitad que el código escalar en realizar el mismo cálculo. El resultado es el esperado, puesto que el código escalar ejecuta muchas más instrucciones que el vectorial.

Como se puede observar en ninguno de los códigos se almacenan los datos calculados en nongún registro, esto es porque únicamente se quería comparar el tiempo de cálculo, pero si se hiciera una comparativa completa, el escalar todavía sería en comparación más lento, puesto que la multiplicación vectorial deja el dato multiplicado en un registro normal, mientras que la multiplicación escalar lo deja en HI y en LO, con lo cual habría que ejecutar una instrucción más (`move`).

6.2. Multiplicación de dos matrices

En este apartado se va a poner a prueba el procesador con un problema real, la multiplicación de matrices. Lo primero que se va a comprobar es que el procesador saca los resultados correctamente, y después se verá el tiempo que ha tardado en realizar dicho problema.

Se van a multiplicar dos matrices de tamaño 4x4, las matrices que se han elegido para el ejemplo son las siguientes:

$$\begin{pmatrix} 1 & 2 & 8 & 5 \\ 0 & 4 & 2 & 1 \\ 5 & 3 & 2 & 9 \\ 6 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 5 & 5 \\ 0 & 1 & 5 & 5 \\ 1 & 3 & 1 & 4 \\ 2 & 4 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 19 & 48 & 33 & 57 \\ 4 & 14 & 24 & 30 \\ 25 & 55 & 60 & 66 \\ 6 & 13 & 35 & 35 \end{pmatrix}$$

El código que se ha desarrollado en ensamblador para poder realizar la prueba es el siguiente:

```
.data 0x10000000

A_app1: .word 1,2,8,5

        .word 0,4,2,1

        .word 5,3,2,9

        .word 6,1,0,0

B_app1: .word 1,0,1,2

        .word 2,1,3,4

        .word 5,5,1,2

        .word 5,5,4,2

C_app1: .word 0,0,0,0

        .word 0,0,0,0

        .word 0,0,0,0

        .word 0,0,0,0

#Multiplicacion de ambas matrices

.text 0x00400000

main:           la $t0, A_app1       # $t0 addres A matrix

                la $t7, C_app1       # $t7 addres C matrix
```

```
                li $t6, 4           # Number of rows

loop_matrix:    ld $t4, 0($t0)      # We read A row
                la $t1, B_app1      # $t1 addres B matrix
                li $t2, 4           # 4 columns in a matrix
                nop
loop_cxy:       ld $t5, 0($t1)      # We read B column
                nop
                nop
                mulv.w $t3, $t4, $t5 # product
                nop
                nop
                addm.w $t3, $t3     # add of the results
                addi $t2, $t2, -1   # We decrement the counter
                addi $t1, $t1, 16   # Now we update pointer for
                next element in  B
                st.b $t3, 0($t7)    # We write in element C
                addi $t7, $t7, 1    # Now we update pointer for
                next element in  C
                bne $t2, $0, loop_cxy# if not zero, then loop
                addi $t0, $t0, 16   #Now we update pointer for
                next element in  A
```

```
                addi $t6, $t6, -1    # We decrement the counter

                bne $t6, $0, loop_matrix  # Next row until 0

    syscall
```

Como se puede observar en el código expuesto arriba, la matriz B se guarda transpuesta para facilitar la ejecución del problema, de este modo se simplifica mucho la manera de realizar los cálculos.

A continuación se puede apreciar en la captura de pantalla el resultado de la ejecución desde la vista del ISim del Xilinx del código anterior funcionando en el procesador diseñado en el trabajo (Figura 6.1).

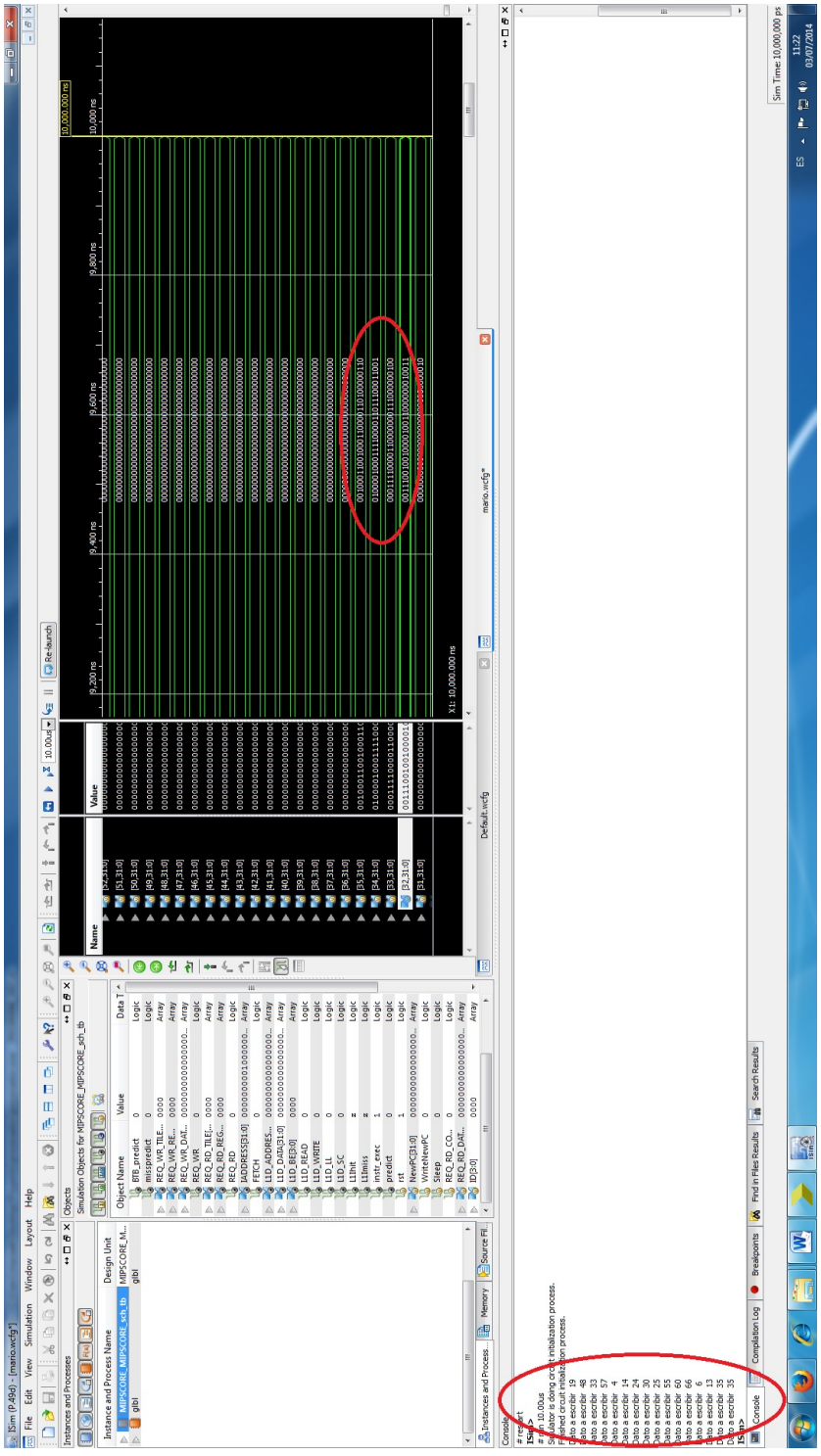

Figura 6.1: Resultado de la ejecución.

Por si no se pude apreciar con el suficiente detalle a continuación se incluyen capturas en las que se ve lo destacado con los círculos rojos en la Figura 6.1.

```
ISim>
# run 10.00us
Simulator is doing circuit initialization process.
Finished circuit initialization process.
Dato a escribir  19
Dato a escribir  48
Dato a escribir  33
Dato a escribir  57
Dato a escribir  4
Dato a escribir  14
Dato a escribir  24
Dato a escribir  30
Dato a escribir  25
Dato a escribir  55
Dato a escribir  60
Dato a escribir  66
Dato a escribir  6
Dato a escribir  13
Dato a escribir  35
Dato a escribir  35
ISim>
```

Figura 6.2: Matriz resultado obtenida de la ejecución del programa.

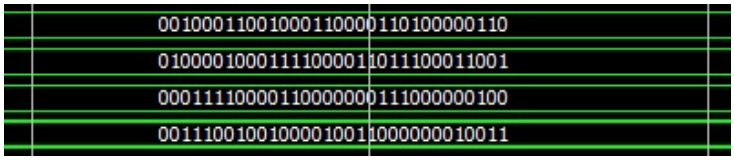

Figura 6.3: Registros con los resultados.

Como se puede ver en la Figura 6.2 el resultado es el que esperábamos (leído fila a fila). En la Figura 6.3 se puede observar como están guardados los datos en la propia memoria del procesador. Cada fila de memoria de la manera en la que están almacenados los datos representa un fila de la matriz. La ejecución ha tardado en ejecutarse 5,32 ms, como se puede apreciar el tiempo es superior a los programas que se han analizado con anterioridad, era de esperar puesto que la complejidad de este es mucho más elevada.

Capítulo 7

Conclusiones

Como se ha podido observar, el trabajo ha supuesto mucho más que la implementación en un lenguaje determinado de un procesador vectorial. También ha consistido en la búsqueda de información sobre dicho tipo de procesadores y su estructura, indagar e investigar acerca de la arquitectura de los procesadore MIPS (más de lo que se podía haber llegado a profundizar en las asignaturas de arquitectura de computadores), aprender y practicar con un nuevo lenguaje de programación HDL como es Verilog antes de llegar a realizar código de calidad con el mismo.

Además se han tenido que buscar soluciones para los problemas que se iban encontrando en el camino desde un punto de vista del desarrollo, pero también desde el punto de vista del diseño, como por ejemplo crear una nueva instrucción que solvente los problemas o las carencias del juego de instrucciones o aunar determinados aspectos de la ruta de datos que estaban diseñados de cara a uno de los dos tipos de procesadores (vectorial o escalar).

También se ha tenido que aprender a usar nuevos programas que hasta el momento se desconocían por completo, como el QTSpim (programa que simula un procesador MIPS escalar), que aunque sí que se había usado algo parecido en alguna de las asignaturas de arquitectura de computadores, nunca se había usado este programa en concreto. O el Isim, programa que simula los diseños realizados en Verilog.

Incluso la redacción de esta memoria ha supuesto un reto, puesto que es la primera vez que se ha de escribir una memoria de tal calibre en la carrera. También el hecho de que esté hecha en LaTeX fue una nueva motivación, dado que hasta el momento en la vida académica nunca lo había empleado.

Hay diversas opciones para modificar el trabajo de cara al futuro.La más evidente y que se ha dejado prácticamente lista, a falta de pocos detalles, es

la implementación de las interrupciones para la parte vectorial del procesador.

Otra de las posibles ampliaciones es completar el juego de instrucciones de la arquitectura SIMD que se detalla en el manual de la misma, aunque algunas de estas instrucciones son prácticamente, por no decir iguales, a algunas del juego de instrucciones escalar que ya está implementado.

Bibliografía

[1] MIPS Technologies, Inc. *MIPS Architecture for Programmers Volume I: Introduction to the MIPS32 Architecture*. Mountain View, CA, junio, 2008.

[2] MIPS Technologies, Inc. *MIPS Architecture for Programmers Volume II: The MIPS32 Instruction Set*. Mountain View, CA, junio, 2008.

[3] MIPS Technologies, Inc. *MIPS Architecture for Programmers Volume III: The MIPS32 Privileged Resource Architecture*. Mountain View, CA, marzo, 2001.

[4] MIPS Technologies, Inc. *MIPS Architecture for Programmers Volume IV-j: The MIPS32 SIMD Architecture Module*. Sunnyvale, CA, abril, 2014.

[5] MIPS Technologies, Inc. *MIPS SIMD Architecture*. Sunnyvale, CA, octubre 2013.

[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] Gerry Kane, J.H. *MIPS RISC Architecture*. Prentice Hall PTR, 2011.

[9] Actel Corporation. *Actel HDL Coding Style Guide*. Mountain View, CA, Julio 2009.

[10] Galindo Riaño, P.L., López Mesa, M.A., Lagares Barreiro, P., Espina Aragón, F.J., Enríquez de Salamanca, J.M., González Barroso, M.A. *Manual de LaTeX*. Servicio de publicaciones Universidad de Cádiz, 2001.

[11] Goossens, M., Mittelbach, F., Samarin, A. *The LaTeXCompanion*. Addison-Wesley publishing company, 1994.