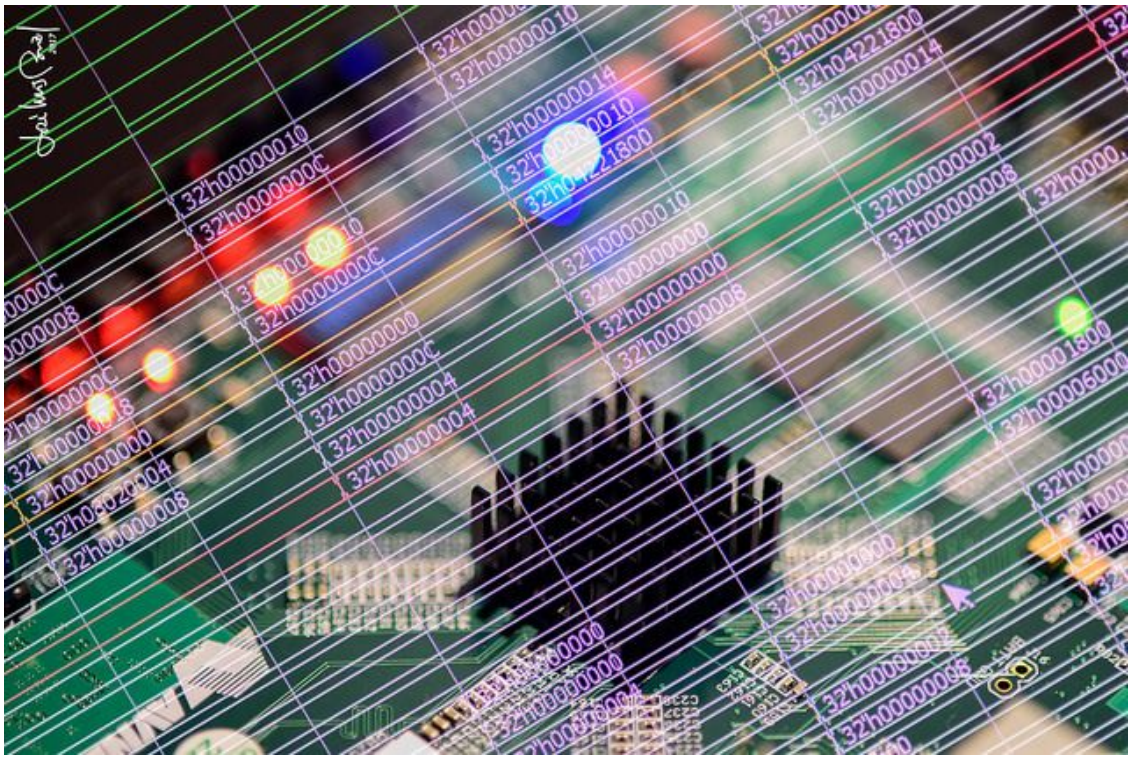
Primer Proyecto AOC II

MIPS Segmentado y Gestión de Riesgos



Autores:
Rubén Rodríguez 737215
Andrew Mackay 737069

Índice de contenidos

1. Introducción

La finalidad de este proyecto es trabajar con un lenguaje descriptivo de Hardware y un simulador de bancos de prueba y cronogramas. El punto de partida del proyecto es un procesador MIPS segmentado de 32 bits para el que no está diseñada la anticipación de operandos, resolviendo los saltos en la etapa ID y escribiendo en los bancos de registros en el flanco de bajada.

2. Trabajo Inicial

El trabajo a realizar consiste en incluir una nueva instrucción al repertorio de instrucciones con el que trabaja el MIPS en un principio. Además también se deben solucionar todos los problemas que pueden surgir en cuanto a la existencia de riesgos de datos y riesgos de control. Para llevar a cabo todas estas tareas de forma efectiva, previamente se dibujó en papel el procesador del MIPS del punto de partida, al que paso a paso, se le fueron añadiendo los diferentes componentes hardware y las conexiones necesarias entre las señales para poder obtener los objetivos deseados.

Posteriormente, se efectuó una toma de contacto con el programa de simulación con el que se iba a trabajar, logrando así la familiarización con el funcionamiento del simulador, la comprensión de la sintaxis, los cronogramas con la evolución de las señales y el significado de éstas últimas.

Así, el proceso de trabajo se ha realizado en los puntos que se muestran a continuación.

3. Resolución del riesgo estructural

Los riesgos estructurales surgen cuando se desean añadir nuevas instrucciones y el hardware del que se dispone actualmente no tiene la capacidad para poder realizarlas, y es por ello, por lo que deben realizarse cambios en el hardware ya existente y si es necesario añadir nuevos componentes.

El MIPS con el que se está trabajando dispone en su hardware de una unidad de suma en coma flotante que permite poder añadir una nueva instrucción,

denominada *ADD_FP* al actual repertorio de instrucciones del procesador. Para poder incorporar esta nueva instrucción, se ha procedido de la siguiente forma:

En primer lugar, se han tenido que realizar ligeras modificaciones en los flags de Unidad de Control del procesador para adaptarla y así poder permitir la ejecución de la instrucción ADD_FP. La Unidad de Control del procesador una vez modificada queda tal y como se muestra en la siguiente tabla:

Instrucción	Branch	RegDst	ALUSrc	MemW	MemR	MemToReg	RegW	Update_RS	FP_add
MOV	0	0	0	0	0	0	0	0	0
ADD	0	1	0	0	0	0	1	0	0
LOAD	0	0	1	0	1	1	1	0	0
STORE	0	0	1	1	0	0	0	0	0
BEQ	1	0	0	0	0	0	0	0	0
ADD_FP	0	1	0	0	0	0	1	0	1

Todas las instrucciones anteriores permanecen exactamente igual a como estaban al principio. Debe observarse que la nueva instrucción añadida, ADD_FP, es prácticamente idéntica a la instrucción ADD salvo en los tipos de datos que suma. Así las cosas, el flag *FP_add* cuando se ejecuta una instrucción ADD_FP está activado a 1 y cuando se ejecuta una instrucción ADD está a 0, permitiendo diferenciar si se están sumando enteros o números en coma flotante.

Posteriormente se ha cambiado el bit de control del multiplexor *mux_fp* a *FP_add_EX*, donde las entradas de datos se conectan a la salida del ALU de enteros y a la salida del sumador en coma flotante, por lo que permite seleccionar entre el resultado de la ALU de enteros y el de la unidad de suma en coma flotante, guardando el resultado obtenido en la salida *ALU_out_EX*. Elegirá la salida de coma flotante cuando *FP_add_EX* esté a 1.

Con todas las modificaciones de la Unidad de Control, y el nuevo hardware añadido, el procesador MIPS soporta éste tipo de operaciones, y por tanto, los riesgos estructurales desaparecen.

4. Gestión de riesgos de datos

Los riesgos de gestión de datos son aquellos que surgen cuando la instrucción que se está ejecutando depende de instrucciones previas que aún están en proceso de tratamiento. Teniendo en cuenta los tres tipos de riesgos de datos que hay (RAW, WAW WAR), se ha incorporado hardware que permite evitar los problemas de riesgo de datos. Para poder evitar este problema se ha diseñado un nuevo hardware, concretamente una unidad de anticipación (UA). Mediante el método de anticipación, se pueden usar los datos para ejecutar futuras instrucciones antes de ser almacenados en el banco de registros.

La unidad de anticipación (UA) ha sido diseñada de acuerdo a la siguiente implementación:

Se han usado dos multiplexores, que ya estaban implementados pero no se usaban, en la etapa EX cuyas entradas de control son *ctrl_muxA* y *ctrl_muxB* conectados a la salida de la unidad de anticipación. Las entradas a los multiplexores son las salidas del banco de registros, busA y busB. Si la instrucción que está en la etapa MEM quiere escribir en el mismo registro rs o rt que quiere utilizar la instrucción en ejecución, pondrá *ctrl_muxA* o *ctrl_muxB*, respectivamente, a 1. Si pasa lo mismo pero en ved de una instrucción en la etapa MEM es una que está en la etapa WB, los pondrán a 2. Si el multiplexor está a 1 coge el dato de la etapa MEM y si es 2 lo coge de WB.

Además, hemos conectado a la salida del *Mux_B* el multiplexor *muxALU_src* para que se pueda elegir entre un dato o un inmediato en casos de leer de memoria con un desplazamiento, por ejemplo. Este multiplexor y el *Mux_A* se conectan a las entradas de la ALU y el sumador en coma flotante.

También se ha modificado la Unidad de Detención (UD), si entra la operación *ADD_FP* espera hasta que la señal *DONE_FP* se activa deteniendo la fase EX, y por lo tanto, las anteriores, mediante la salida *Parar_EX*. Esta salida no deja cargar datos en los registros *pc, IF/ID, MEM/EX* y resetea *MEM/EX*, para no alterar los contenidos de la memoria mientras realiza la suma.

Por otro lado, la UD también detecta si está una instrucción de lectura de memoria en la etapa EX donde el registro en el que quiere escribir es uno de los que necesita la instrucción que viene justo después. En este caso, se activa la señal *Parar_ID* . Esta señal detiene el load del registro *pc* e *IF/ID,* y resetea el banco *ID/EX* para que no se altere la lectura de memoria.

5. Gestión de riesgos de control

Para llevar a cabo la gestión de los riesgos de control es primordial saber porque ocurren. Es necesario hacer énfasis en el hecho de que las instrucciones tienen las siguientes etapas: IF, ID, EX, MEM y WB. Los riesgos de control surgen en aquellos casos en los se procede a ejecutar una instrucción de salto, es decir, una instrucción BEQ y no se conoce la siguiente instrucción a ejecutarse en caso de efectuar el salto hasta la etapa MEM.

Ante esta situación, es necesario llevar a cabo la evaluación y el cálculo del destino en caso de efectuarse el salto lo antes posible. Para ello, hay diferentes alternativas. En primer lugar se puede detener el pipeline hasta que se conozca la instrucción, sin embargo, es una propuesta muy lenta. Como se busca la máxima efectividad y rendimiento se ha optado por asumir que no se va a tomar el salto y buscar la siguiente instrucción ejecutar.

Si la predicción de salto es acertada, no efectúa el salto y continúa ejecutando las instrucciones siguientes. En caso contrario, se desechan todas las instrucciones que estén en ejecución de forma errónea insertando ceros en los valores de control. Así, la penalización en tiempo será de un ciclo en caso de que se tome el salto.

Para conseguirlo, se usa la señal *Z* que se pone a 1 cuando las salidas *busA* y *busB* del banco de registros son iguales. También se usa la señal *branch*, una salida de la *UC* que se activa cuando la instrucción es de salto. Cuando ambas están activadas, se pone a 1 *PCSrc*, el bit de control del multiplexor *muxpc*. Este multiplexor controla la entrada al registro *pc,* el que almacena la siguiente instrucción a ejecutar.

También se han tenido que realizar modificaciones en la Unidad de Detención (UD). Dichas modificaciones consisten en programar el comportamiento de una señal, llamada *Kill_IF*. Se debe destacar el hecho de que mientras se está evaluando la instrucción de salto, se está llevando a cabo el tratamiento de las instrucciones siguientes. Por tanto, al determinar si se produce el salto o no en la etapa MEM, las instrucciones próximas ya están en proceso de ejecución. La señal *Kill_IF*, definida anteriormente, se encarga de llevar a cabo el reset del registro *IF/ID* y no deja

cargar en *pc* cuando su valor es 1. En caso contrario, cuando la señal vale 0, no se realiza el reset, continuando así, con la ejecución normal de las instrucciones.

6. Impacto de rendimiento

El rendimiento del procesador MIPS a lo largo de las diferentes etapas del proyecto ha ido cambiando notablemente. Seguidamente, se va a discutir cuáles han sido los diferentes cambios que ha experimentado.

Los riesgos de datos tienen lugar cuando existen dependencias (RAW, WAR, WAR) entre instrucciones que se ejecutan de forma dependiente. Para solucionar dicho problema de riesgo inicialmente se introducían instrucciones NOP para evitar la pérdida o alteración de los datos ya que se retrasa la ejecución un ciclo. No obstante mediante esta técnica, la cantidad de instrucciones que se deben ejecutar es relativamente grande.

Ante esta ineficiencia por el coste de las instrucciones, se han usado cortocircuitos para evitar los riesgos de datos pudiendo ejecutarse todas las instrucciones sin ningún problema a excepción del LOAD. Si no hay ninguna instrucción intermedia de por medio que use algún recurso del LOAD y que retarde la ejecución de las instrucciones siguientes, se debe esperar un ciclo adicional gracias a la incorporación de la Unidad de Anticipación.

Gracias a este hardware añadido se solucionan todas aquellas dependencias existentes entre instrucciones que son distintas del LOAD, por lo que se ahorran tantos ciclos como instrucciones NOP, salvo el caso de la instrucción LOAD, que como se ha explicado anteriormente, se debe parar un ciclo. Por lo que el rendimiento del procesador aumenta.

En cuanto a los riesgos de control, se debe enfatizar que aparecen por el hecho de que los datos que se usan para evaluar la condición de salto se calculan durante la ejecución. Así, si no se realizan cambios, habría que esperar tres ciclos para poder determinar qué instrucción se debe ejecutar, por lo que el rendimiento aumenta. Ante esta situación, se desplazó el cálculo de dirección a la etapa de ID realizando una predicción de salto.

De esta forma, si el salto se toma pues se han ganado de 3 ciclos, mientras que si no se toma se pierde un ciclo. Es obvio que en ambos casos, el número de ciclos se reduce y por consiguiente aumenta el rendimiento del procesador.

7. Pruebas de comprobación

A continuación se muestra una descripción brevemente detallada de todas las pruebas que se han hecho para constatar que el funcionamiento del procesador MIPS diseñado es correcto. Posteriormente se procederá a analizar el código de los programas, las correspondientes codificaciones de las instrucciones, cronogramas de las señales y capturas de los resultados que devuelve.

7.1 Primera prueba

En esta primera prueba representa un sencillo programa de cuatro instrucciones, dos *LOADS* y dos *BEQS* respectivamente. La finalidad de esta prueba es verificar la correcta gestión de los riesgos de datos. A continuación se muestran las instrucciones del programa:

LW r0, (r0)
LW r1,4(r0)
BEQ r0, r1, #inm
BEQ r1,r1, #inm

Como puede observarse, las instrucciones LW cargan en los registros r0 y r1 del banco de registros el contenido de la dirección de memoria apuntada por el registro r0 en la memoria.

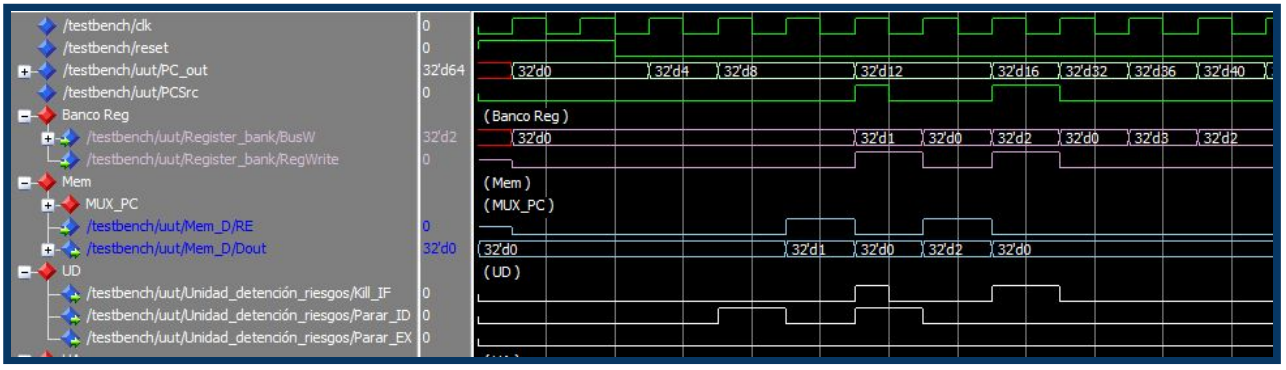
En la siguiente instrucción LW, carga en r1, el valor almacenado en la dirección de memoria apuntada por r0 sumando un desplazamiento de cuatro.

Posteriormente, compara el contenido de ambos registros r0 y r1. Si son iguales, se efectúa el salto a la dirección marcada por el valor inmediato K. En caso contrario, sigue con el secuenciamiento directo de instrucciones y procede a ejecutar el BEQ siguiente. Nótese que ahora el salto será tomado si o si ya que se está comparando un registro consigo mismo, es decir, se está forzando un salto incondicional.

En el programa anterior, se puede observar una dependencias RAW (Read-After-Write) en el segundo LW ya que se están cargando en el registro r1 el valor de la dirección apuntada por r0 sumada cuatro cuando el valor de r0 todavía no ha sido cargado porque todavía se está efectuando la lectura de memoria.

A continuación, se muestra como queda la codificación de las instrucciones anteriores en binario y hexadecimal.

Instrucción	Op	Rs	Rt	K	Hexadecimal
LW r0, (r0)	000010	00000	00000	0000000000000000	08000000
LW r1, 4(r0)	000010	00000	00001	0000000000000100	08010004
BEQ r1, r0, #inm	000100	00000	00001	1111111111111110	1001FFFE
BEQ r0, r0, #inm	000100	00000	00000	0000000000000100	10000004



En la salida de la prueba observamos como hace dos lecturas en la memoria separadas de un ciclo, aunque en teoría va un load detrás de otro. Esto sucede porque la segunda instrucción load necesita el registro r0, como aún no puede conocer el valor ya que el primer load va a escribir en ese mismo registro, la unidad de detención para la etapa ID durante un ciclo y pasa a la etapa EX una instrucción nop para no alterar nada.

Por otro lado, podemos comprobar que lee y guarda en los registros el valor correcto, un 1 que está en la posición 0 de la memoria y un 2 que está en la posición 0+4. Por último, observando *PCSrc* y *PC_out* se verifica que solo realiza el segundo salto. Empieza a hacer el primer salto pero nunca lo llega a terminar, debido a que al principio aún no se habían cargado del todo los registros.

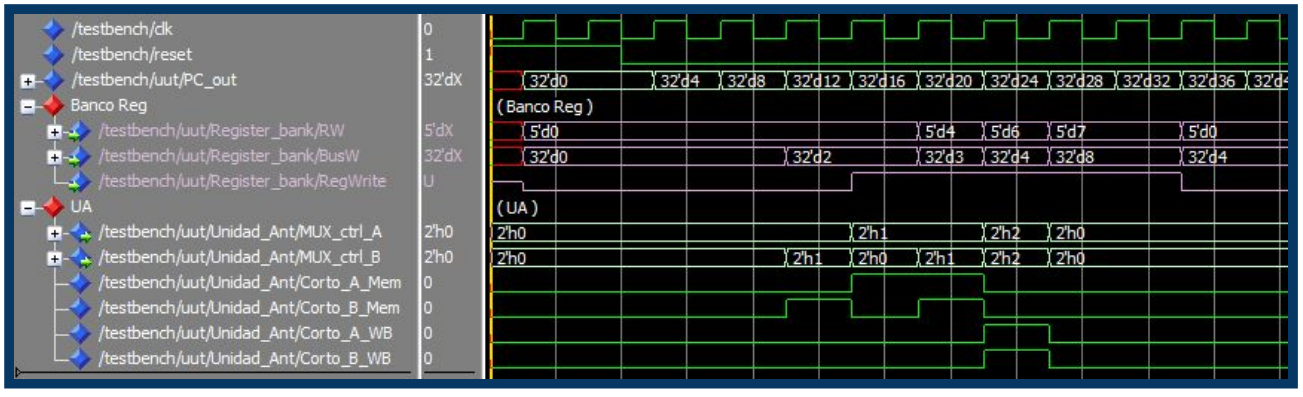
7.2 Segunda prueba

Este segundo programa consta de cinco instrucciones ADD en las que se combinan diferentes anticipaciones de datos. El objetivo es verificar que tanto la Unidad Aritmético Lógica (ALU) como Unidad de Anticipación (UA). A continuación se muestra el código del programa:

Instrucción	Descripción
ADD r0, r1, r2	Suma en r0 los datos almacenados en los registros r1 y r2
ADD r4 ,r3 , r0	Suma en r4 los datos almacenados en los registros r3 y r0
ADD r6 ,r4 , r5	Suma en r6 los datos almacenados en los registros r4 y r5
ADD r7 ,r6 , r6	Suma en r7 los datos almacenados en los registros r6 y r6
ADD r7, r6 , r6	Suma en r7 los datos almacenados en los registros r6 y r6

Codificación de las instrucciones:

Instrucción	Op	Rs	Rt	rd	shamt	funct	Hexadecimal
ADD r0, r1, r2	000001	00001	00010	00000	00000	000000	04220000
ADD r4 ,r3 , r0	000001	00011	00000	00100	00000	000000	04602000
ADD r6 ,r4 , r5	000001	00100	00101	00110	00000	000000	04853000
ADD r7 ,r6 , r6	000001	00110	00110	00111	00000	000000	04C63800
ADD r7, r6 , r6	000001	00110	00110	00111	00000	000000	04C63800



En esta prueba los registros han sido inicializados a 1, para poder observar el correcto funcionamiento de las sumas. Como aparece en la imagen, el programa no se detiene nunca y guarda los valores correctos los correspondientes registros. En

r0 almacena un 2 (r1+r2), en r4 un 3 (r3+r0), en r6 un 4 (r4+r5) y por último en r7 un 8 (r6+r6). Esto se consigue gracias a la unidad de anticipación que realiza los correctos cortocircuitos para anticipar el resultado de la suma anterior sin tener que detener el programa.

7.3 Tercera prueba

En esta prueba se utiliza la instrucción de suma en coma flotante. El programa ha sido implementado con la finalidad de afirmar que los riesgos de datos están perfectamente controlados, y por consiguiente, que la unidad de anticipación y la unidad de detención funcionan correctamente. Posteriormente se ha plasmado las instrucciones que conforman el programa y un breve comentario de que se está ejecutando

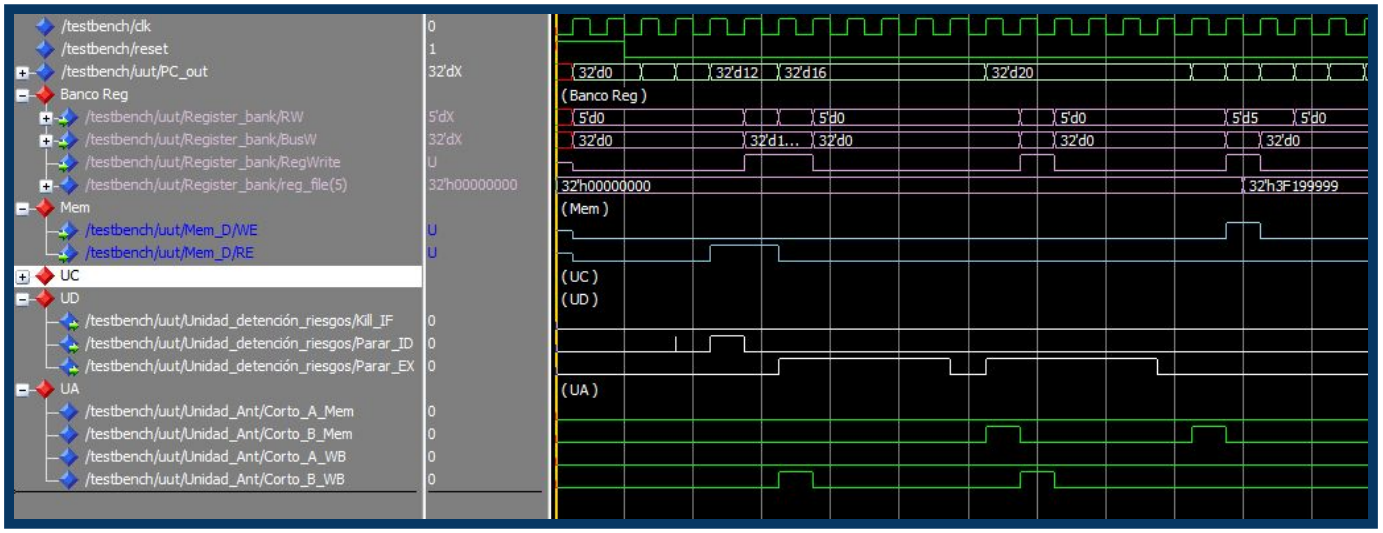
Instrucción	Descripción
LW r1,(r0)	Carga en el registro r1 el contenido de la dirección de memoria apuntada por r0
LW r2,(r3)	Carga en el registro r2 el contenido de la dirección de memoria apuntada por r3
ADDFP r4,r1,r2	Guarda el r4 el resultado de sumar en coma flotante los registros r1 y r2
ADDFP r5,r2,r4	Guarda el r5 el resultado de sumar en coma flotante los registros r2 y r4
SW r5,(r0)	Guarda r5 en la dirección de memoria apuntada por el registro r0

Se han mezclado ambos formatos de instrucciones, y por ello, la codificación de las instrucciones se refleja en tablas distintas.

Instrucción	Op	Rs	Rt	K	Hexadecimal
LW r1 , (r0)	000010	00000	00001	0000000000000000	08010000
LW r2 , (r3)	000010	00011	00010	0000000000000000	08620000
SW r5 , (r0)	000011	00000	00101	0000000000000000	0C050000

Instrucción	Op	Rs	Rt	Rd	Shamt	Funct	Hexadecimal
ADDFP r4 , r1, r2	100000	00001	00010	00100	00000	000000	80222000

ADDFP r5, r2 , r4	100000	00010	00100	00101	00000	000000	80442800



En primer lugar, se puede comprobar que la unidad de detección realiza 3 detenciones. La primera, de un solo ciclo, es para darle tiempo al load para guardar el dato cargado en memoria en r2, ya que la instrucción siguiente lo quiere utilizar. Las otras dos detenciones suceden porque la suma en coma flotante tarda unos ciclos, por lo que se debe detener el programa hasta que termine correctamente la suma.

También se puede observar que entre suma y suma, y después de cargar el segundo dato de memoria, no hay ninguna detención, ya que la unidad de anticipación realiza un cortocircuito. Una vez acabadas las dos sumas se realiza una escritura en la memoria con el resultado de las dos sumas. El contenido inicial de la memoria en la dirección 0 es 3E4CCCCD (0.2), y como aparece en la onda *reg_file(5)*, el registro r5, vale 3F199999 (0.6 aproximadamente), por lo tanto es correcto.

7.4 Cuarta prueba

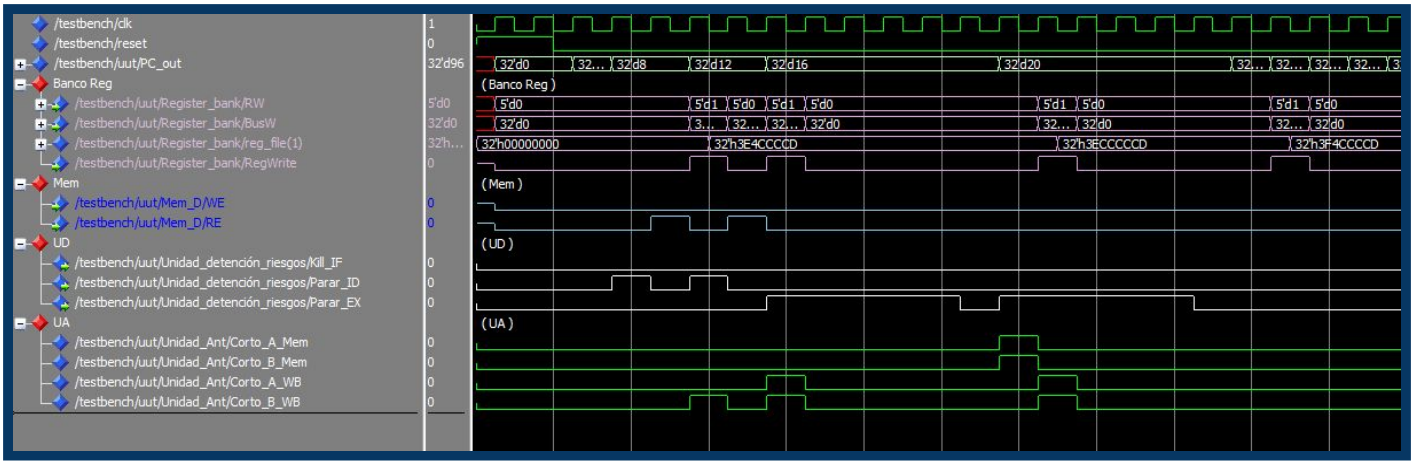
Este programa es parecida a la anterior, pero con unas ligeras modificaciones.

Instrucción	Descripción
LW r1,(r0)	Carga en r1 el contenido de la dirección de memoria apuntada por r0
LW r1,(r0)	Carga en r1 el contenido de la dirección de memoria apuntada por r0
ADDFP r1,r1,r1	Guarda en r1, el resultado de sumarse en coma flotante consigo mismo
ADDFP r1,r1,r1	Guarda en r1, el resultado de sumarse en coma flotante consigo mismo

A continuación se muestra la codificación de las instrucciones

Instrucción	Op	Rs	Rt	K	Hexadecimal
LW r1 , (r0)	000010	00000	00001	0000000000000000	08010000

Instrucción	Op	Rs	Rt	Rd	Shamt	Funct	Hexadecimal
ADDFP r1, r1 , r1	100000	00010	00001	00001	00000	000000	80442800



Como se puede observar, es muy parecida a la anterior, solo que en este caso realiza 4 detenciones, debido a que el segundo LOAD utiliza un registro que va a modificar el primero, aunque no sería necesario. La modificación más importante es

la de las instrucciones addfp, que son iguales, para comprobar el completo funcionamiento de la unidad de anticipación. En este caso, cumple su función perfectamente.

El valor inicial de la memoria es igual que la de la prueba anterior, por lo que el resultado en *reg_file(1)*, que sería r1, es 3F4CCCCD (0.8), por lo que ha sumado correctamente.

7.5 Quinta prueba

En este programa se ha puesto a prueba cómo funcionan varias anticipaciones de operandos con instrucciones de suma en coma flotante. De esta manera, se tiene absoluta certeza de que el hardware destinado a realizar operaciones en coma flotante funciona bien, es decir, que no existe riesgo estructural, y que las anticipaciones que deben realizarse para evitar los riesgos de datos también se efectúan bien. En relación a lo anterior, se muestra el código del programa cuyo funcionamiento se ha sido testeado junto con una explicación de su comportamiento

Instrucción	Descripción
LW r1,(r0)	Carga en r1 el contenido de la dirección de memoria apuntada por r0
ADDFP r1,r1,r1	Guarda en r1, el resultado de sumarse en coma flotante consigo mismo
LW r1,(r0)	Carga en r1 el contenido de la dirección de memoria apuntada por r0
ADDFP r1,r1,r1	Guarda en r1, el resultado de sumarse en coma flotante consigo mismo
STR r1,(r3)	Guarda el resultado de r1 en la dirección de memoria apuntada por r3

Nótese que al igual que ocurría con pruebas anteriores, en este programa también se mezclan instrucciones de memoria con operaciones aritmético-lógicas. Por este motivo, la codificación se ha reflejado como en los casos anteriores por separado.

Codificación de las instrucciones :

Instrucciones de gestión de memoria

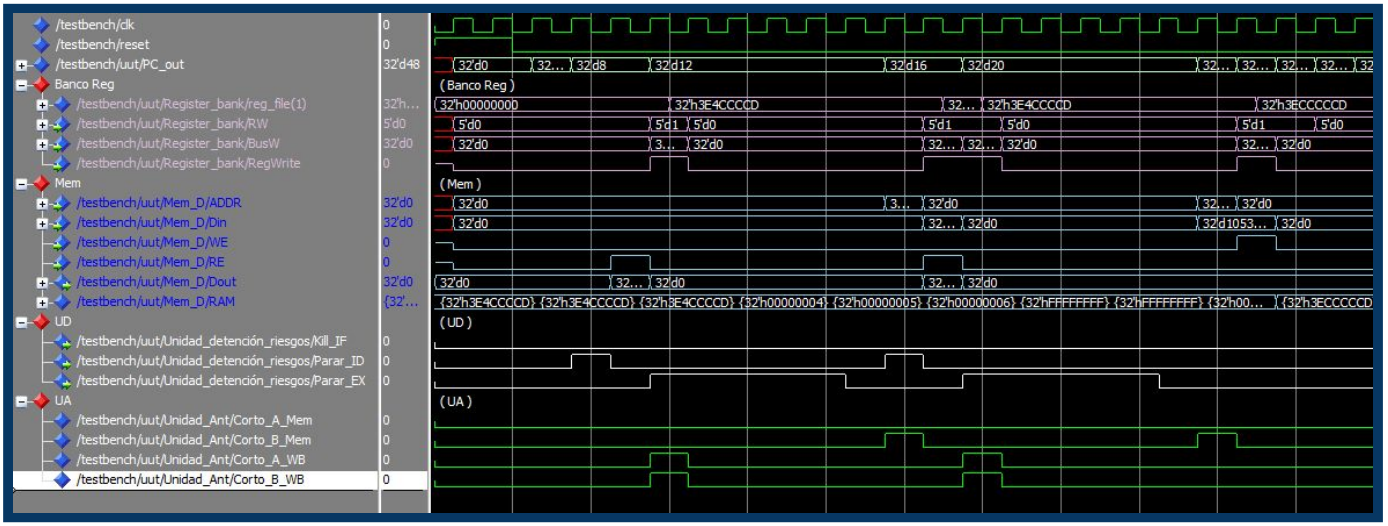
En este programa se puede ver que existen dos instrucciones de gestión de memoria, concretamente los LW, que se repiten dos veces, por lo que sólo se ha reflejado una única vez

Instrucción	Op	Rs	Rt	K	Hexadecimal
LW r1 , (r0)	000010	00000	00001	0000000000000000	08010000
SW r1 , (r3)	000011	00001	00011	0000000000000000	0C610000

Instrucciones aritmético-lógicas

Si observamos el programa, es claramente evidente que la instrucción aritmético-lógica de suma en coma flotante se repite en dos ocasiones, por lo que la codificación es la misma y por tanto sólo la reflejamos una vez.

Instrucción	Op	Rs	Rt	Rd	Shamt	Funct	Hexadecimal
ADDFP r1, r1 , r1	100000	00001	00001	00001	00000	000000	80210800



En la dirección 0 de memoria se encuentra el valor 3E4CCCCD (0.2), para comprobar la suma en coma flotante. En primer lugar, se observa que la unidad de detención realiza 4 detenciones, dos de un ciclo por los loads y otros dos más largos por los addfps, por las razones explicadas en pruebas anteriores.

En el banco de registros se realizan cuatro escrituras diferentes, el primero del load, luego el del addfp, inmediatamente después uno del load (descartando el valor de la suma en addfp y volviendo a su valor de 0.2) y por último el del segundo addfp. No hay más detenciones de las necesarias entre los loads y addfps gracias a la unidad de anticipación. Como se puede ver en la imagen, el último valor de *reg_file(1)*, que

sería r1, es de 3ECCCCCD (0.4), y que posteriormente se escribe ese valor en memoria, por lo que se realiza correctamente el programa.

7.6 Sexta prueba

Esta es la última prueba que se realiza, una prueba completa que realiza un bucle para sumar 4 números guardados en memoria (1,2,3 y 4, cuya suma debe dar A). Una vez sumados los 4 números en r5, guarda en la posición siguiente de la memoria el resultado y entra en un bucle infinito donde se queda en su posición.
A continuación se muestra el código del programa y la codificación de las instrucciones

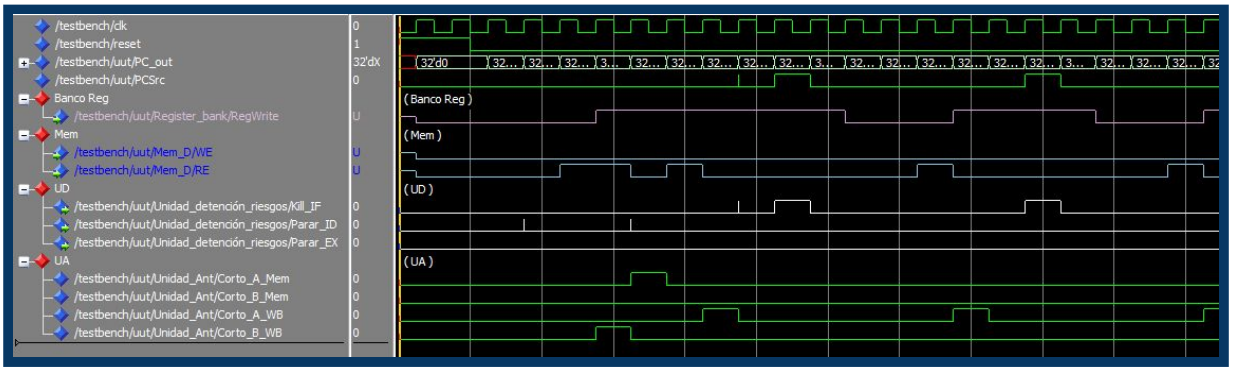
Instrucción
LW r1,(r0)
LW r4, (r0)4
ADD r3,r3,r1
LW r2,(r3)4
ADD r0,r0,r4
ADD r5,r5,r2
ADD r3,r3,r1
BEQ r1,r0,#1
BEQ r1,r1,#-6
SW r5,(r3)4
BEQ r1,r1,#-1

Codificación de las instrucciones:

Instrucción	Op	Rs	Rt	K	Hexadecimal
LW r1 , (r0)	000010	00000	00001	0000000000000000	08010000
LW r4 , (r2)4	000010	00010	00100	0000000000000100	08440004
LW r2 , (r3)4	000010	00011	00010	0000000000000100	08620004
BEQ r0 ,r1, #1	000100	00000	00001	0000000000000001	10010001
BEQ r1 ,r1, #-6	000100	00001	00001	1111111111111010	1021FFFA
SW r5 ,(r3)4	000011	00011	00101	0000000000001000	0C650004

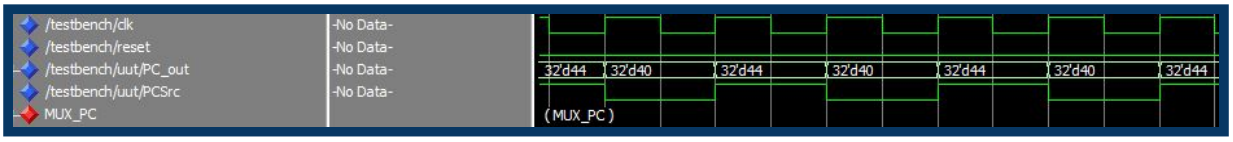
BEQ r1 ,r1 ,#-1	000100	00001	00001	1111111111111111	1021FFFF

Instrucción	Op	Rs	Rt	Rd	Shamt	Funct	Hexa
ADD r3 , r3 , r1	00001	00011	00001	00011	00000	000000	04611800
ADD r0 , r0 , r4	00001	00000	00100	00000	00000	000000	04040000
ADD r5 , r2 , r5	00001	00010	00101	00101	00000	000000	04452800



El programa empieza haciendo dos loads seguidos (no se detiene ya que escriben en dos registros distintos) para cargar los valores iniciales, 4 en r1 que limita el número de bucles y 1 en r4 para sumar al contador de bucles, que será r0. Entrando al bucle hay dos anticipaciones por parte de la UA, el primero por el primer ADD del programa y el segundo por el primer LOAD del bucle, ya que comparten operandos. Una vez en el bucle, se repiten siempre las mismas detenciones y anticipaciones: una anticipación por el segundo ADD del bucle y una detención por el BEQ.

El bucle finaliza cuando ha repetido 5 ciclos, donde se queda en un bucle infinito de saltos a la misma instrucción como señal de que ha llegado al final.



Los registros y la memoria quedan de esta forma al finalizar el programa:

Como podemos observar, en la memoria al final de los números que debe sumar se ha guardado un A, la suma de los 4 dígitos anteriores. Por otro lado, en los registros se puede observar que r0 ha llegado a 4 (el máximo número de ciclos), r3 ha llegado a 14 (guardaba las direcciones de donde cargar los números y donde guarda el resultado final) y r5 vale A, que es el resultado de la suma. Por lo tanto, queda demostrado que funciona correctamente el programa.

8. Planificación del trabajo

En este último apartado se muestra una planificación de las tareas realizadas y del tiempo aproximadamente en horas que ha contado hacer cada una de ellas

Tarea	Duración
Comprensión de simulador ModelSim:	cuatro horas
Riesgos estructurales	una hora
Riesgos de datos	cuatro horas
Riesgos de control	una hora
Debuggear y depuración de fallos	quince horas
Realización de la memoria	siete horas