AOC 2

Memoria proyecto 1

2018/2019

Jesús Aguas Acín -- 736935@unizar.es
Daniel Barceló Labuena -- 737070@unizar.es

Índice

1. Resumen

El proyecto realizado consiste en modificar el MIPS de 32 bits facilitado en Moodle, realizando todas las adiciones requeridas y completando todos los módulos necesarios para evitar cualquier tipo de riesgo de datos y de control.

El entorno de trabajo utilizado es el simulador de VDHL *ModelSim*. Los fuentes iniciales utilizados que contenían la estructura principal del MIPS fueron modificados atendiendo a los requisitos del guión del proyecto. Siguiendo 4 fases:

- **Fase 1**: Se corrigen errores en el código fuente inicial que impedían el correcto funcionamiento de la instrucción de salto **beq.**
- **Fase 2**: Se añaden dos nuevas instrucciones (**bne** y **la**) al repertorio de instrucciones del MIPS.
- **Fase 3**: Se corrigen riesgos de datos de tipo **RAW** (**R**ead **A**fter **W**rite) mediante la anticipación de operandos y la detención del pipeline.
- **Fase 4**: Se implementa el módulo de predicción dinámica de saltos para reducir los riesgos de control.

A lo largo de la memoria se puede encontrar un explicación en detalle de cada una de estas fases y de las pruebas que se han realizado para comprobar el correcto funcionamiento de cada funcionalidad añadida al MIPS.

2. Introducción

Para el desarrollo de este proyecto se utilizará el entorno de simulación ModelSim y se utilizará el lenguaje de descripción de hardware VHDL.

Como punto de partida, se utilizarán los fuentes en VHDL facilitados en *Moodle* correspondientes a un MIPS 32 con anticipación de operandos, que resuelve los saltos en la etapa ID, y escribe en el banco de registros en los flancos de bajada. Además incluye un predictor de saltos muy sencillo.

Algunos de los módulos están vacíos, y el proyecto consiste en completarlos para que todo funcione correctamente.

3. Objetivos

En el **Proyecto** se establecen como objetivos:

- Entender cómo funciona un procesador segmentado y cómo interactúa con memoria y con la entrada/salida
- Aprender a utilizar un entorno de desarrollo profesional
- Adquirir los conceptos básicos de depuración y verificación

Estos objetivos se completaran a los largo de 4 fases:

- Fase I: Preparación
- Fase II: Calentamiento (Añadir dos nuevas instrucciones, bne y la)
- Fase III: Gestión de riesgos de datos
- Fase 4: Gestión de riesgos de control

A continuación se detalla el desarrollo de las fases citadas, en el apartado de **Metodologia**.

4. Metodología

4.1 Fase I

Familiarización con el MIPS segmentado

Durante esta fase se ha estudiado el código fuente inicial, todos los componentes del procesador y su funcionamiento. Se han relacionado estos componentes con los estudiados en clase y a partir de allí se ha ejecutado el código de ejemplo.

Revisando ciclo a ciclo la ejecución se ha identificado un error por el que nunca se saltaba a la dirección especificada (BEQ).

El problema se encuentra en el fichero *MIPs_predictor_incompleto.vhd* , en el que la señal de control (PCSrc) del MUXPC se encuentra constantemente a 00, es decir, siempre PC = PC + 4, por lo que nunca se salta a la dirección señalada (DirSalto_ID).

Para solucionar este problema, se ha cambiado el valor de la señal PCSrc a 11 si Branch && Z. De esta manera, cuando la señal Branch proporcionada por la unidad de control (UC) esté a 1, significará que estamos frente a una instrucción de salto. A su vez, la señal Z nos indicará si los dos registros a los que accede el beq son iguales. Por tanto, cuando se trate de una instrucción de salto (Branch=1) y los dos registros contengan lo mismo (Z=1) se saltará a la dirección deseada (MUXPC=11).

4.2 Fase II

Añadir instrucciones bne y la

bne:

Salta si los registros no son iguales. Similar al BEQ, pero con la condición contraria.

Para añadir la instrucción ***bne*** se ha partido de la solución aportada en la fase I y se ha modificado la condición para realizar el salto, de modo que ahora para que sea efectivo un salto a otra instrucción del código se deberá cumplir la siguiente condición:

```
Saltar <= '1' when (Branch='1' AND Z='1' AND Op_code_ID="000100")
        else '1' when (Branch='1' AND Z='0' AND Op_code_ID="000101")
        else '0';
```

Esto quiere decir que se realizará el salto si la instrucción es ***beq*** y los dos registros son iguales o si la instrucción es ***bne*** y los dos registros son distintos.

A continuación se ha modificado la UC (Unidad de control), donde se ha codificado la operación "**bne"** (000101) con los flags de control idénticos a los utilizados para la instrucción **bne**.

Para comprobar el funcionamiento de la nueva instrucción se ha utilizado el código ejemplo facilitado en Moodle, donde se ha cambiado la instrucción "beq r0, r0, 0" por la instrucción "bne r0, r0, 0", y se tras simular la ejecución del código se ha comprobado que esta vez no se había tomado el salto, al ser r0 y r0 iguales. También se ha realizado otra simulación para comprobar que se realizaba el salto correctamente cuando el valor de los dos registros era distinto, "bne r0, r1, 0" y el resultado ha sido satisfactorio, el PC saltaba a la dirección especificada.

la:

la rt, inmed(rs):
rt ← rs + SignExt(inmed) , rs ← rs + SignExt(inmed), PC ← PC + 4

Para añadir la instrucción ***la*** (load address) se han aplicado los conocimientos estudiados en la Fase I del proyecto. Para llevar a cabo la introducción de esta instrucción se ha partido de la instrucción "**add**", que ya estaba implementada en los ficheros fuente proporcionados. Las diferencias entre estas dos instrucciones son las siguientes: en el "**add**" se suman dos registros (rs y rt) y el resultado se almacena en un tercer registro (rd); mientras que en el "**la**" se suma un registro (rs) con un inmediato de 16 bits(k) con el signo extendido hasta obtener un dato de 32 bits, y el resultado se almacena en un segundo registro (rt).

Para conseguir que la instrucción funcione correctamente solo hay que modificar la unidad de control, añadiendo una nueva entrada que codifica la instrucción "**la**" (0010000). Para esta entrada, hay que especificar las salidas que va a tener. Para que la instrucción sume el registro (rs) con el inmediato (k), hay que escribir un 1 en la salida ALUSrc para que el MUX de la alu que selecciona el segundo operando, elija el inmediato extendido (k) y no elija el registro (rt). También hay que escribir un 0 en la salida RegDst para que al escribir en el banco de registros lo haga en (rt) en vez de en (rd), escogiendo el registro deseado con el mux del banco de registros.

Para comprobar el funcionamiento de esta instrucción se ha partido del código de ejemplo proporcionado en las fuentes. El código que se ha ejecutado ha sido el siguiente:

```
08010000    LW  R1, 0(R0)
09020004    LW  R2, 4(R8)
00000000    nop
20250020    la r5, 32(r1)
```

Al finalizar la ejecución de las siguientes instrucciones, se comprueba el correcto funcionamiento de la instrucción tal y como se muestra en la foto 2.

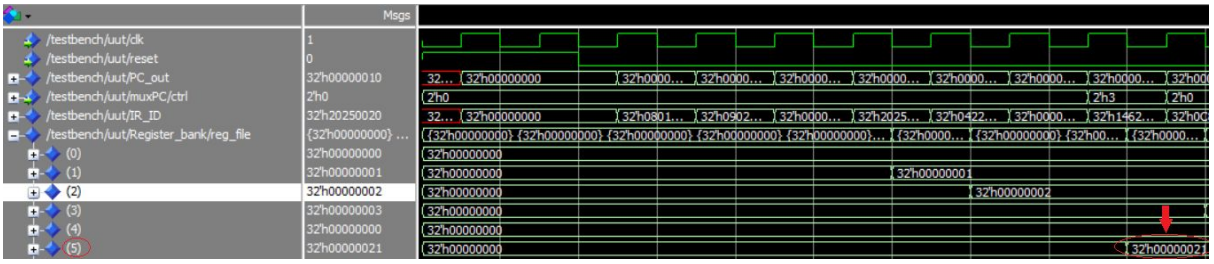


Foto 2

El resultado esperado es 33, ya que estamos sumando el contenido de r1(1) con 32, que escrito en hexadecimal es 0x21. Como se ve en la imagen, el registro r5 contiene al finalizar la ejecución de la instrucción el resultado deseado (0x21).

La "**nop**" que hay escrita en el código es necesaria para evitar un riesgo de load/uso (RAW), ya que en la primera instrucción estamos cargando un dato en r1(1) y este dato se escribe 2 ciclos más tarde(en la fase de WriteBack), por lo que si eliminamos la "**nop**", la instrucción "**la**" no cogería el dato de r1 correctamente, y cogería el dato que hubiera almacenado en el registro antes del load, haciendo que el funcionamiento fuera distinto al deseado.

4.3 Fase III

Gestión de riesgos de datos

En primer lugar se ha configurado la Unidad de anticipación (UA_incompleta.vhd) para que permita la anticipación de operandos de las etapas MEM y WB a la etapa EX y así solucionar dependencias que causan riesgo de lectura después de escritura (LDE), como por ejemplo, al guardar el resultado de una suma (***add***) en un registro que justo después se utiliza como operando de otra suma (***add***).
Los dos bits que salen de la UA son los bits de control del mux que controla la ALU:

```
Si (EX/MEM.RegWrite
and EX/MEM.Register_dst = ID/EX.RegisterRs)) Ant_A = 10

Si (EX/MEM.RegWrite
and EX/MEM.Register_dst = ID/EX.RegisterRt)) Ant_B = 10

Si (MEM/WB.RegWrite
and MEM/WB.Register_dst = ID/EX.RegisterRs)) Ant_A = 01

Si (MEM/WB.RegWrite
and MEM/WB.Register_dst = ID/EX.RegisterRt)) Ant_B = 01
```

Sin embargo, aún siguen existiendo riesgos de datos LDE en los casos de *load/uso* y en los saltos (***beq***/***bne***) que comparan registros en los que se acaba de almacenar datos. Para estos casos la única solución es la detención.

Se producirá un caso de load/uso cuando el registro en el que va a escribir la operación que está en la etapa EX(**ld**) coincida con alguno de los registros que va a utilizar la operación que está en la etapa ID. Cuando se produzca, se tendrá que detener la etapa IF y la etapa ID para que el dato obtenido de la memoria le dé tiempo(1 ciclo) a llegar a la etapa WB y así se podrá utilizar el dato en la siguiente instrucción mediante la anticipación de operandos.

- Para detener la etapa IF basta con hacer que el PC no aumente y se quede con el mismo valor hasta que se reanude el programa.
- Para detener la etapa ID hay que enviar a la etapa EX las señales que correspondan a una instrucción con el código de operación de una NOP. Para esto, se ha creado un MUX2_1 de 6 bits (ya que hay 6 señales de control). La señal de control de este MUX será la señal que indica si hay un riesgo de load/uso. De esta manera, cuando haya que parar, se enviarán las señales de las **nop**(000000), y cuando no, se mandarán las señales de la instrucción en decodificación.

Cuando una instrucción escribe en uno de los registros que la instrucción **beq** va a utilizar para comparar, se dará el caso de “instrucción productora -- beq consumidor”.
Si esta coincidencia se da cuando la productora está en EX se dice que es un riesgo a distancia 1 y habrá que parar 2 ciclos; mientras que si esta coincidencia se da en MEM se dice que es un riesgo a distancia 2 siendo necesario parar 1 ciclo para dar tiempo a que la productora escriba en el banco de registros. La anticipación de operandos no sirve para la instrucción **beq** porque esta operación se resuelve en la etapa ID. La detención se realizará de la misma forma que se hizo para el caso de “load-uso”.

En la fase de pruebas se ha detectado un error de anticipación de operandos en la instrucción **sw**. Esta instrucción no tiene anticipación de operandos para el registro que se utiliza para direccionar la memoria, así que se ha creado un mux que elige qué entrada utilizar para direccionar esta memoria, el dato anticipado por la unidad de control o el dato que sale del banco de registros. Para elegir uno u otro se ha mirado si la unidad de anticipación anticipaba un dato para el registro que sale por el Bus B, y si selecciona el dato anticipado, significa que tenemos que anticiparlo para nuestra dirección de memoria

```
ctrldirsSalto <= '0' when (MUX_ctrl_B="00")
                else '1';

mux_dirsaltoW: mux2_1 port map (Din0 => BusB_EX, DIn1 => Mux_B_out, ctrl => ctrldirsSalto, Dout => dirSalto);
```

De esta manera queda solucionada la anticipación y se puede utilizar perfectamente sin necesidad de detenerse ningún ciclo.

4.4 Fase 4

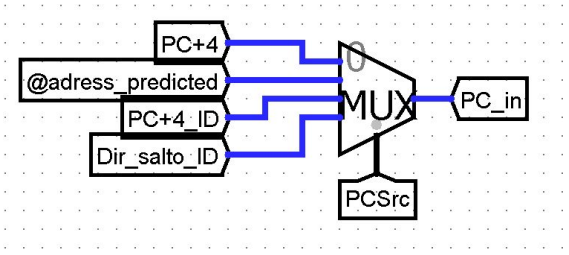
Gestión de riesgos de control

El objetivo de esta fase es diseñar la gestión del módulo de predicción dinámica de saltos para predecir la dirección del salto de la próxima instrucción de salto, y así reducir la penalización en los saltos y el número de detenciones del pipeline.

El primer paso será gestionar el MUX que controla las entradas del PC según el tipo de predicción que se haga. Para ello se establecerá un nuevo criterio con el que determinar los bits de control del MUX:

```
PCSrc <= "00" when (prediction = '0' AND predictor_error = '0')
      else "01" when (prediction = '1' AND predictor_error = '0')
      else "10" when (Saltar='0' AND predictor_error = '1')
      else "11";
```

Los bits de control que corresponden a las dos primeras entradas (“00” y “01”) se utilizarán según la predicción sea 1(saltar) o 0(continuar) y siempre que no se haya detectado ningún error en la predicción, en cuyo caso, si ha habido un error en la predicción de tipo *decission* o *address*, se cargará en el PC la instrucción que debía haberse cargado.



También se deberán identificar los errores de predicción, estos podrán ser de dos tipos, *decission error* o *address error*. Cada vez que se produzca alguno de estos dos errores se actualizará el módulo de predicción dinámica de saltos (update_predictor=’1’) y se utilizará como nueva predicción(prediction_in) la predicción contraria a la que causó el error.

- Se producirá un *decission error* cuando la predicción de saltar o no saltar sea distinta a la que se ha resuelto en la etapa ID de la instrucción de salto.
- Se producirá un *address error* cuando en una instrucción de salto, la dirección predicha sea distinta a la que se ha resuelto en la etapa ID de la instrucción de salto.

Una vez diseñada la gestión del módulo de predicción dinámica de saltos queda comprobar su efectividad en los bucles, su punto fuerte.
Para ello se ha utilizado como prueba un bucle infinito compuesto por un conjunto de nops y una instrucción de salto que vuelve a saltar a la dirección inicial indefinidamente. Con esto se ha podido comprobar como la primera vez falla, pero el resto de veces predice la dirección de salto correctamente.

4.5 Metodología y división del trabajo

El método de trabajo ha consistido principalmente en la reunión de los dos integrantes del grupo de forma presencial, a través de videollamadas, y dos tutorías con los profesores para aclarar dudas.

En cuanto a la división del trabajo, la mayoría del trabajo se ha realizado con los dos integrantes trabajando juntos para resolver cada una de las dificultades y problemas encontrados, excepto en el desarrollo de la memoria, donde se ha dividido el trabajo entre los integrantes del grupo. El motivo por el que se ha decidido trabajar en cada fase los dos miembros a la vez, es porque el guión exige que los dos miembros del grupo conozcan todas las modificaciones realizadas, por lo que es necesario que los dos integrantes vean y entiendan qué se ha modificado y por qué.

El total de horas estimadas dedicadas al trabajo es de **27 horas** por cada integrante del grupo, desglosadas en:

- Paso 1: **2** horas
- Paso 2: **2** horas
- Paso 3: **5** horas
- Paso 4: **3** horas
- Depuración y ajustes: **12** horas
- Memoria: **3** horas

5. Pruebas

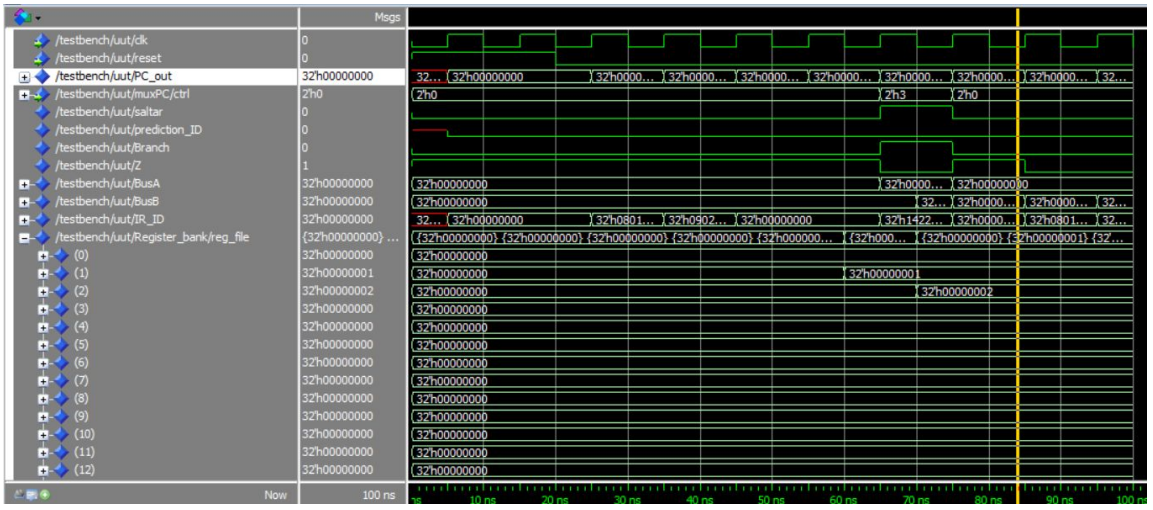
Todas las pruebas realizadas y descritas hasta el momento se han ido realizando a medida que se iba mejorando el MIPS en cada fase del proyecto.

A continuación, y con la versión final del MIPS, se pasarán una batería de pruebas para comprobar que satisface todas las funcionalidades especificadas en el guión del proyecto.

5.1 “bne” test

Ficheros: Test_Bne.asm y Test_Bne_No_Risks.vhd

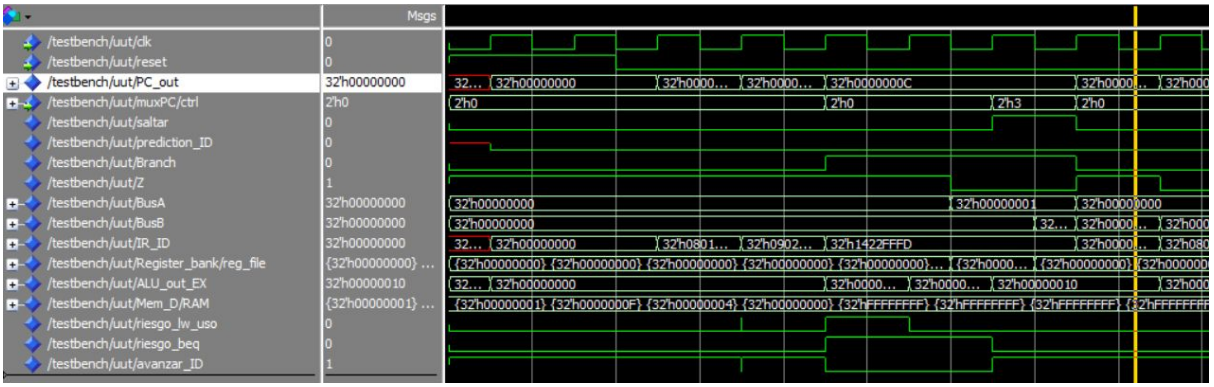
```
ini:
    LW R1, 0(R0);               //R1=0x1
    LW R2, 4(R0);               //R2=0xF
    NOP
    NOP
    BNE R1,R2,ini
```



En este test, simplemente se ha probado la ejecución de la instrucción **bne** sin riesgos de load/uso. El predictor predice que no hay que saltar pero luego rectifica y salta, por lo que funciona correctamente.

Ficheros: Test_Bne.asm y Test_Bne_With_Risks.vhd

```
ini:
    LW R1, 0(R0);                //R1=0x1
    LW R2, 4(R0);                //R2=0xF
    BNE R1,R2,ini
```

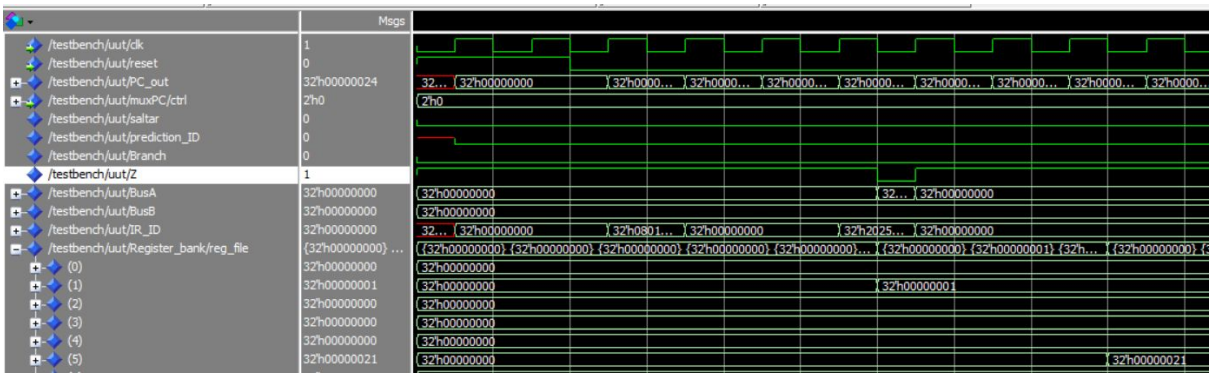


Este funcionamiento de este test es igual que el anterior, tiene el mismo resultado. En este se han eliminados las instrucciones **nop** y se ha utilizado la gestión de riesgos diseñada. Como se ve en la wave, la instrucción **bne** se detiene hasta que tiene sus operandos disponibles y luego se ejecuta, aunque al ser la primera instrucción de salto que se encuentra el predictor, predice que no hay que saltar y luego lo rectifica.

5.2 “la” test

Ficheros: test_la.asm y test_la_no_anti.vhd

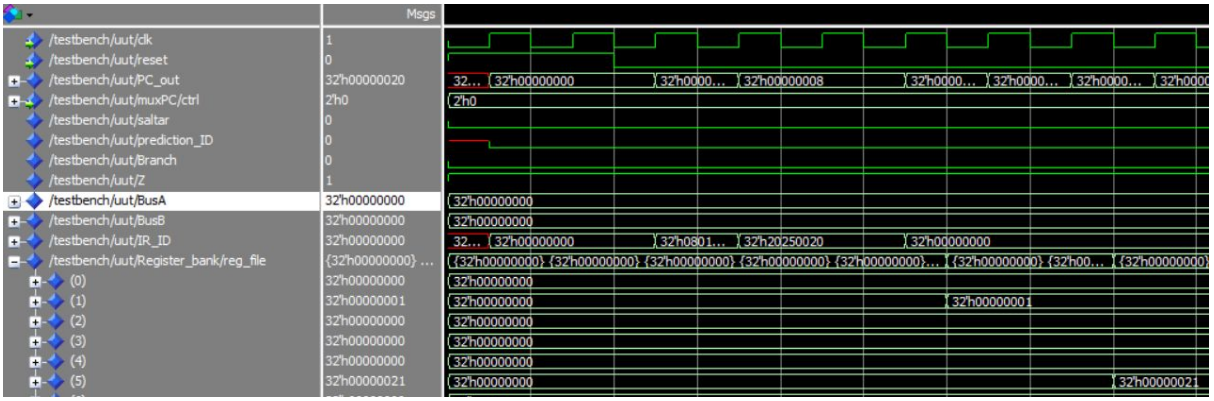
```
ini:
    LW R1, 0(R0)                //R1=0x1
    NOP
    NOP
    LA R5, 32(R1)               //R5=33
```



En este test se ha probado el funcionamiento de la nueva instrucción **la** sin anticipación y sin riesgos de load/uso. Como se observa en la wave, al final de la ejecución el registro r5 almacena 0x21, que es el resultado esperado.

Ficheros: test_la.asm y test_la_with_anti.vhd

```
ini:
    LW R1, 0(R0)                //R1=0x1
    LA R5, 32(R1)               //R5=33
```

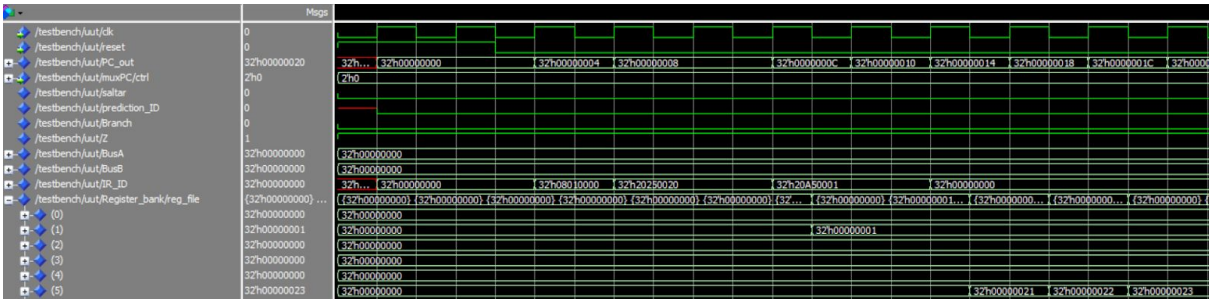


Se trata del mismo test que el anterior pero con anticipación y con riesgos. Se han eliminado las NOP y aparecen riesgos de load/uso, por lo que hay que detener un ciclo la instrucción **la**, hasta que la instrucción **lw** llega al ciclo de MEM y obtiene el dato para poder se anticipado mediante la unidad de anticipación al **la**.

5.3 ld-uso test

Ficheros: test_anticipacion.asm y test_anticipacion.vhd

```
ini:
    LW R1, 0(R0)                //R1=0x1
    LA R5, 32(R1)               //R5=33
    LA R5, 1(R5)                //R5=34
    LA R5, 1(R5)                //R5=35
```

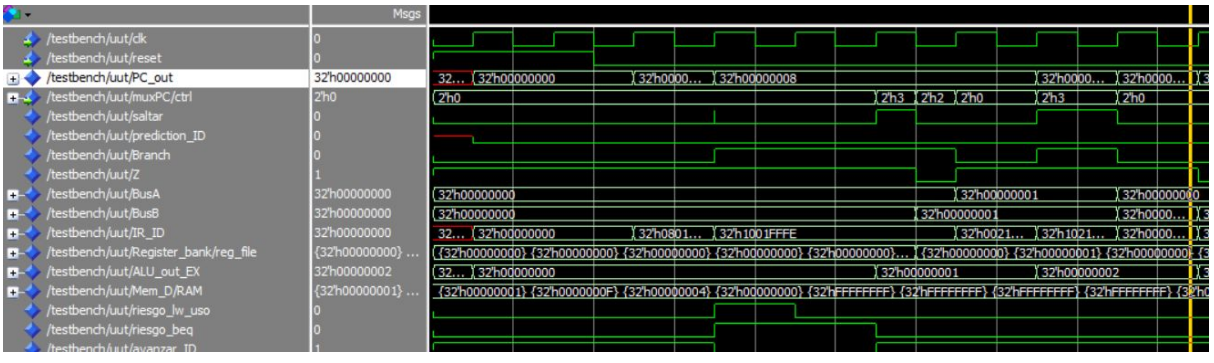


En este test, se prueba algo parecido al anterior. Probamos la eficiencia de resolver un problema de load/uso con riesgos y anticipación y tras esto se añaden instrucciones que utilizan los mismos registros pero funcionan correctamente por la unidad de anticipación.

5.4 productor-consumidor beq test

Ficheros: test_anticipacion.asm y test_anticipacion_not.vhd

```
ini:
    LW R1, 0(R0)                //R1=0x1
    BEQ R0, R1, ini
    BEQ R1, R1, ini
```

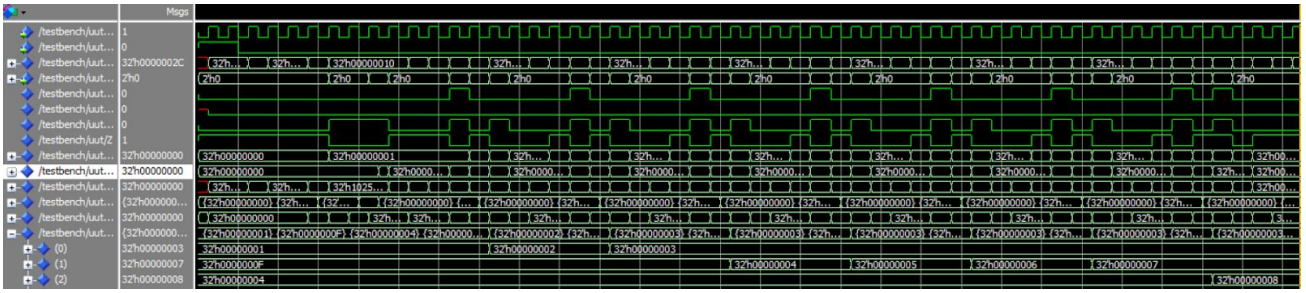


En este test se ha comprobado el correcto funcionamiento cuando se da un problema de load/uso en una instrucción **beq**. Como la instrucción **beq** no puede utilizar la unidad de anticipación, el problema se gestiona como un riesgo de un **beq** normal y corriente, deteniendo la instrucción 2 ciclos hasta que tiene sus operandos disponibles en el BR.

5.5 predictor de saltos test

Ficheros: test_predSaltos.asm y test_predsaltos_fuerzafallos.vhd

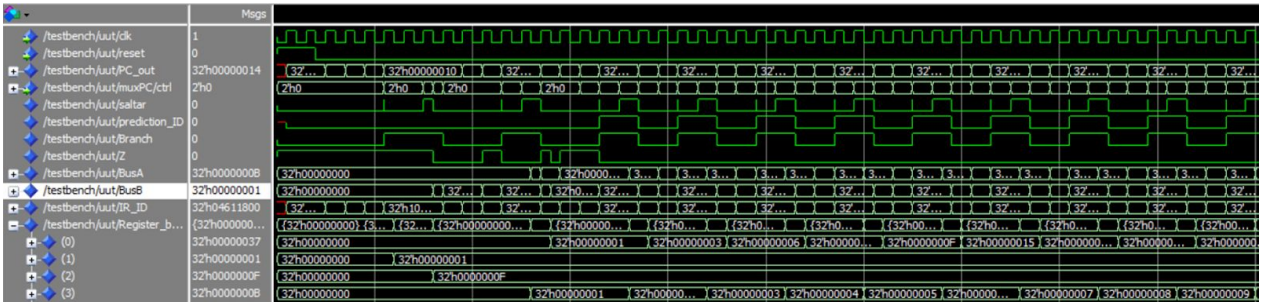
```
ini:
    LW R1, 0(R0)                //R1=0x1
    LA R2, 0(R1)                //R2=0x1
    LA R5, 8(R0)                //R5=8

buc:
    BEQ R1, R5, endbuc
    ADD R1, R2, R1
    SW  R1, 4(R1)
    BNE R5, R5, buc
endbuc:
        //R1= 0x8
```

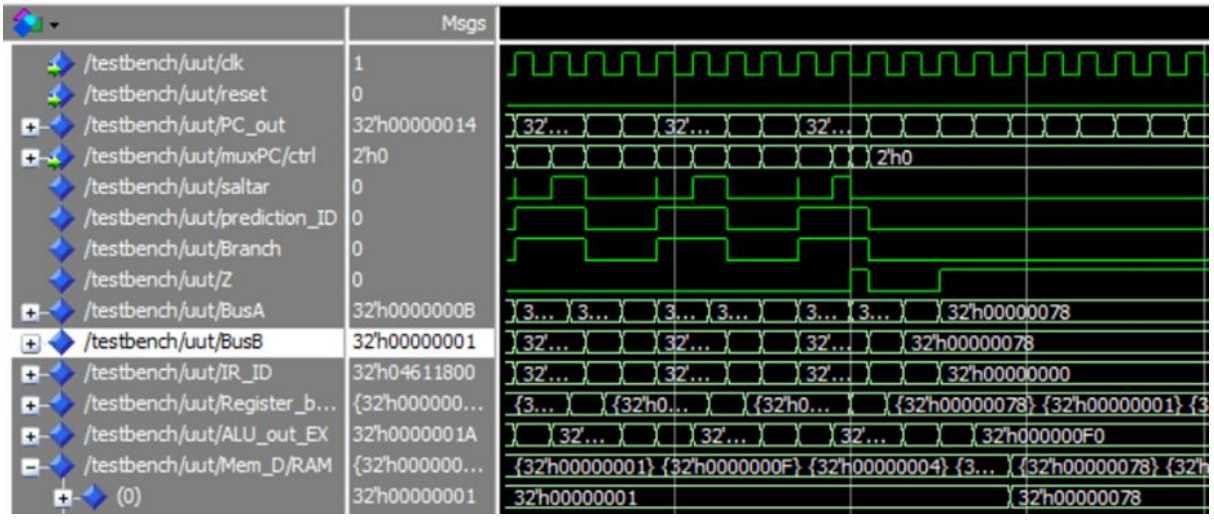


En este test se ha probado el predictor de fallos haciendo que falle cada vez que encuentra un salto para comprobar que siempre rectifica sus fallos y acaba saltando cuando debe. Como se ve en la ejecución en la wave, el resultado es el esperado y cada salto fallido se acaba recuperando correctamente.

Ficheros: test_predSaltos.asm y test_predSaltos.vhd

```
En hexadecimal:
ini:
    08010000        LW  R1, 0(R0)
    20030000        LA R3, 0(R0)
    08020004        LW R2, 4(R0)
    10620004        BEQ R3, R2, endbuc
buc:
    04611800        ADD R3, R3, R1
    04030000        ADD R0, R0, R3
    1462FFFD        BNE R3, R2, buc
endbuc:
    0C800000        SW R0, 0(R4)
```





En esta prueba final hemos querido hacer un programa que fuera útil y utilizara todo lo implementado anteriormente. El programa es un algoritmo que se encarga de sumar los n-1 primeros números naturales. Para introducir este n basta con escribirlo en la dirección @4 de memoria, y el resultado se almacenará en la dirección @0 de memoria. Tenemos un riesgo de load/uso con anticipación de operandos en las instrucciones 1 y 2, un riesgo de beq en las instrucciones 3 y 4; y un bucle para probar el correcto funcionamiento del predictor de saltos. En el ejemplo, se suman los 15 primeros números naturales, que dan como resultado 0x78==120, que queda almacenado al final de la ejecución en la dirección @0 de memoria.

6. Resultados

Impacto en el rendimiento

La primera mejora que se ha añadido ha sido la red de anticipación de operandos, que permite que el procesador funcione más rápido ya que no tiene que detener el pipeline (o introducir **nops**) por problemas de productor/consumidor RAW(Read after Write).
Se podrá ver mejor el impacto que tendrá en el número de instrucciones con un ejemplo:

Supongamos que tenemos una instrucción **add** que utiliza el registro **r1** para almacenar su resultado, y justo a continuación, otro **add** que utiliza también el registro **r1** para hacer una operación. Con la red de anticipación, se podría realizar una anticipación de operandos del registro **r1**, ahorrándonos la espera de 2 ciclos (EX Y MEM, no sería necesario esperar al ciclo WB ya que el procesador escribe en el BR en el flanco de bajada). Sin embargo, si no tuviésemos la red de anticipación y quisiéramos evitar los riesgos de datos tendríamos que introducir en el código, con planeación estática, 2 **nops** entre los dos **add.** Por lo que el impacto en el rendimiento con esta mejora sería evitar 2 instrucciones (y por lo tanto ciclos) menos por cada riesgo de datos. El CPI sería el mismo.
También se ha añadido la anticipación de operandos para el direccionamiento de la memoria RAM, lo que también nos ahorra dos ciclos de espera (No hace falta esperar a que la instrucción anterior escriba en el banco de registros de las etapas EX Y MEM).

La segunda mejora, también relativa al riesgo de datos, consiste en parar el pipeline el número de ciclos necesario (1 o 2) en el caso de productor-**beq** consumidor, el problema radica en que la instrucción de salto no puede beneficiarse de la red de anticipación. Por lo que, en cuanto al impacto en el rendimiento, al igual que en la anterior mejora, no sería necesario introducir 1 o 2 nops para evitar el riesgo, sin embargo el CPI aumentaría para este tipo de riesgos de datos, debido a los ciclos de detención del pipeline.

La tercera mejora es la gestión de un módulo de predicción dinámica de saltos, esto es una clara mejora en el rendimiento, ya que en el mejor de los casos(predicción correcta), no habría ningún ciclo de penalización en el salto. Antes de esta mejora, siempre se ejecutaba la siguiente instrucción al beq, por lo que era siempre necesario poner una nop, o en su defecto, detener el pipeline. Con esta mejora, en el peor de los casos, solo se detendrá el pipeline un ciclo si la predicción ha sido errónea. Por lo que el CPI será menor que antes de introducir esta mejora.

Como resultado de todas estas mejoras, serán necesarias 0 **nops** extra para evitar riesgos, por lo que el número de instrucciones totales para realizar una tarea será menor, y por lo tanto, los ciclos totales en realizar esa tarea. Además, con el módulo de predicción dinámica de saltos, los saltos que se predigan bien, durarán 1 ciclo menos que en el MIPS sin mejoras.