Escuela de Ingeniería y Arquitectura Universidad Zaragoza

MIPS SEGMENTADO Y GESTIÓN DE RIESGOS

AUTORES:
ALEJANDRO TERRÓN ÁLVAREZ, NIP:761069
DIEGO MARCO BEISTY, NIP:755232

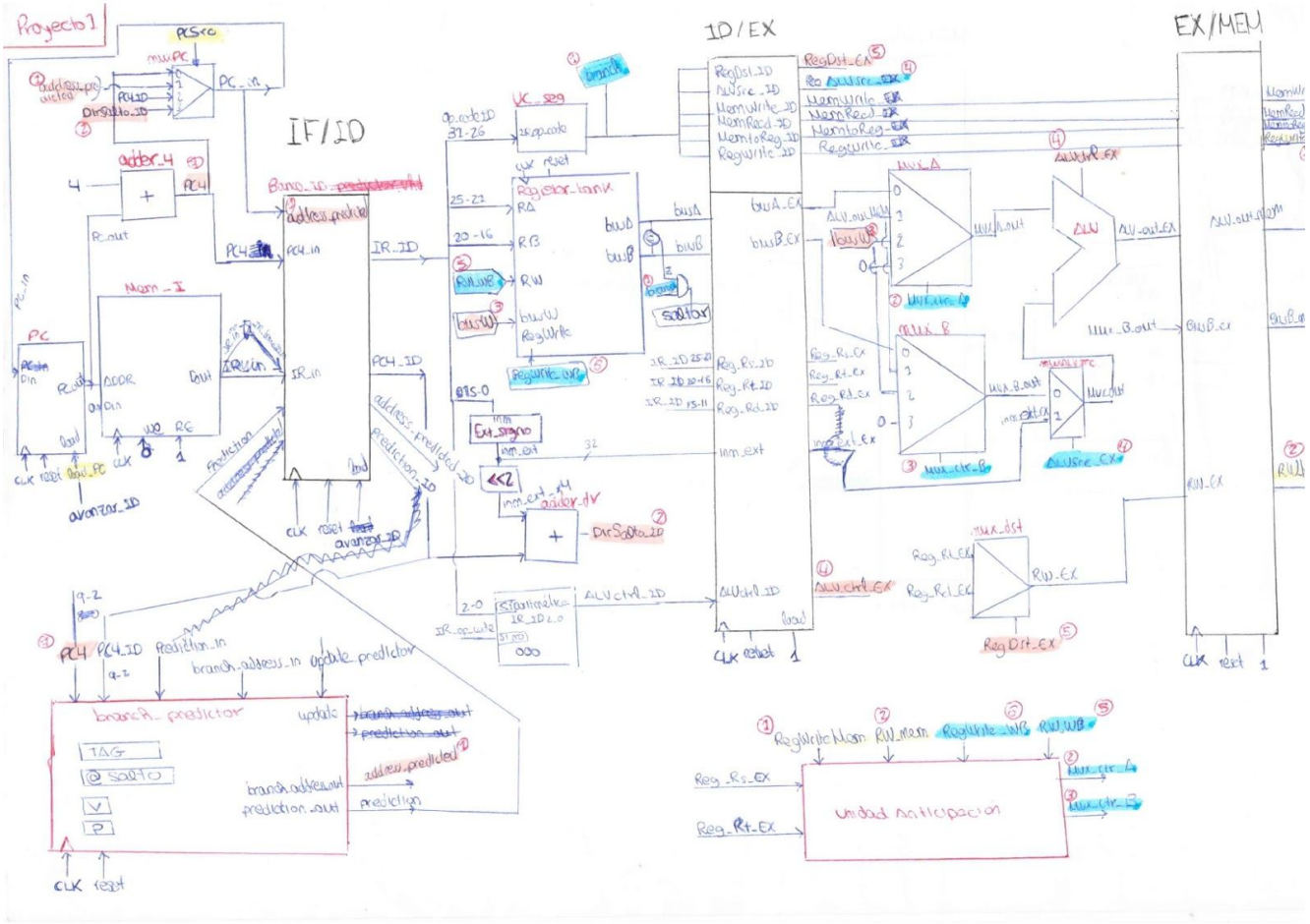
ÍNDICE

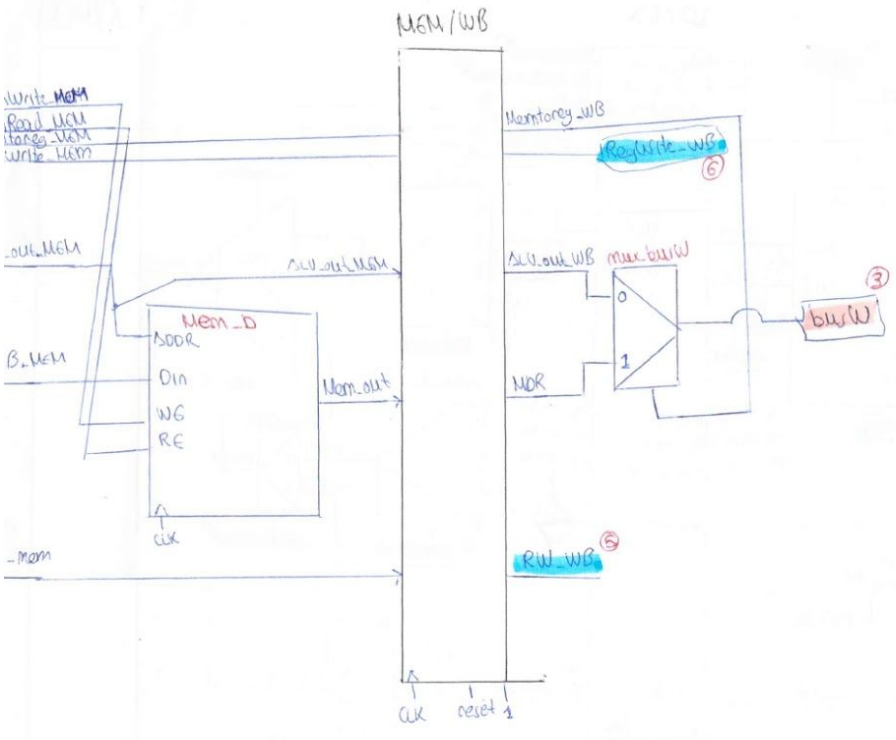
RESUMEN

El objetivo de este proyecto es aprender a trabajar con un lenguaje de descripción de hardware y un entorno de simulación basado en bancos de prueba y cronogramas, tomando como caso de estudio el mismo procesador MIPS de 32 bits utilizado en clase con anticipación de operandos, que resuelve los saltos en la etapa ID, y escribe en el banco de registros en los flancos de bajada.

Paso 1: Trabajo inicial

Para el desempeño de este trabajo se decidió trasladar el modelo proporcionado en lenguaje VHDL a papel para así poder visualizar las interconexiones entre los distintos módulos que se nos daban, posteriormente nos decidimos a tomar contacto con el entorno de trabajo a través de los videos disponibles en moodle y la información que obtuvimos a través de algunas búsquedas en internet.





Antes de las modificaciones:

Codificación de las operaciones:

NOP	000000
ADD	000001
SW	000010
LW	000011
BEQ	000100

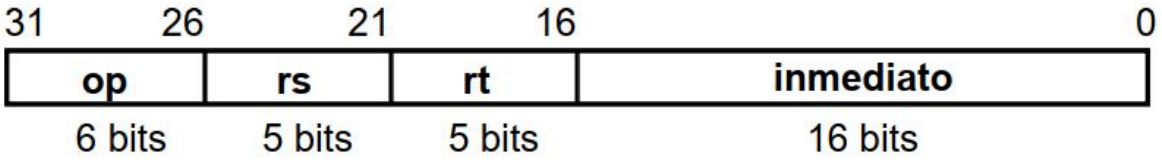
Señales de control de la UC:

	Branch	RegDst	ALUSrc	Mem Write	Mem Read	Memto Reg	RegWrite	Update_Rs	FP_add
NOP	0	0	0	0	0	0	0	0	0
ADD	0	1	0	0	0	0	1	0	0
LW	0	0	1	0	1	1	1	0	0
SW	0	0	1	1	0	0	0	0	0
BEQ	1	0	0	0	0	0	0	0	0

Paso 2: Añadir nuevas instrucciones

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

BNE:
bne rs, rt, inmed si (rs /= rt) entonces (PC ← PC + 4 + 4·SignExp(inmed))
en otro caso PC ← PC + 4



Codificación:

LA	001000
BNE	000101

Señales de control de la UC:

	Branch	RegDst	ALUSrc	Mem Write	Mem Read	Memto Reg	RegWrite	Update_Rs	FP_add
LA	0	0	1	0	0	0	1	0	0
BNE	1	0	0	0	0	0	0	0	0

Para este apartado en el caso del BNE hemos añadido una señal en el MIPS segmentado llamada saltar2 que es análoga a saltar solo que esta se activa con Branch=1 y Z=0 es decir salta cuando los datos son distintos.

```
Saltar <= Branch AND Z;
Saltar2 <= Branch AND NOT Z; -- empleada por la instrucción BNE
```

Paso 3: Gestión de riesgos de datos

Los riesgos de gestión de datos que se pueden dar a los largo de la ejecución de las instrucciones, surgen cuando la instrucción actual depende del resultado de instrucciones previas que aún están en proceso. Para ello hemos implementado la unidad de anticipación y la unidad de detención, se encuentran ambas en los ficheros UA_Mips.vhd y UD_Mips.vhd

```
entity UA is
    Port(
            Reg_Rs_EX: IN  std_logic_vector(4 downto 0);
            Reg_Rt_EX: IN  std_logic_vector(4 downto 0);
            RegWrite_MEM: IN std_logic;
            RW_MEM: IN  std_logic_vector(4 downto 0);
            RegWrite_WB: IN std_logic;
            RW_WB: IN  std_logic_vector(4 downto 0);
            MUX_ctrl_A: out std_logic_vector(1 downto 0);
            MUX_ctrl_B: out std_logic_vector(1 downto 0)
        );
    end UA;
```

La unidad de anticipación (**UA**) es la que se encarga, como su nombre indica, de anticipar un dato a posteriores instrucciones en otras etapas donde lo necesiten utilizar con el valor actualizado antes de que este se guarde en el banco de registros.

La salida de esta unidad son los bits de control de los multiplexores “muxA” y “muxB”, situados en la etapa EX. En ambos multiplexores, la entrada “0” es el estado del bus de salida del banco de registros leído en la etapa anterior. La entrada “1” corresponde a la salida de la ALU del ciclo anterior con el valor actualizado del operando. La entrada “2” corresponde a la salida de la memoria de datos o salida de la ALU de dos ciclos anteriores. Hemos implementado 3 casos diferentes para esta lógica en el fichero UA_Mips.vhd, iguales para ambos muxes. Aplicado al “muxA”:

➤ Caso en que registro rs de la instrucción que se encuentra en la etapa ID es igual que el registro RW de la instrucción que se encuentra en la etapa MEM y además el flag RegWrite en la etapa MEM está activo. Hay anticipación y la salida del “muxA” será “1”.

```
"01" when RegWrite_MEM='1' AND RW_MEM=Reg_Rs_EX else
```

➤ Caso en que registro rs de la instrucción que se encuentra en la etapa ID es igual que el registro RW de la instrucción que se encuentra en la etapa WB y además el flag RegWrite en la etapa WB está activo. Hay anticipación y la salida del “muxA” será “2”.

```
"10" when RegWrite_WB='1' AND RW_WB=Reg_Rs_EX else
```

➢ Caso en que no se produce ninguna anticipación. La salida del “muxA” será “0”.

```
"00";
```

```
entity UD is
    Port ( Reg_Rs_ID: in STD_LOGIC_VECTOR (4 downto 0);--registro rs de etapa ID
           Reg_Rt_ID: in STD_LOGIC_VECTOR (4 downto 0);--registro rt de etapa ID
           RW_MEM: in  STD_LOGIC_VECTOR (4 downto 0);--registro destino etapa MEM
           RW_EX: in  STD_LOGIC_VECTOR (4 downto 0);--registro destino etapa EX
           MemRead_EX: in STD_LOGIC;--Señal control lectura memoria etapa EX
           RegWrite_MEM: in STD_LOGIC;--Señal control escritura memoria etapa MEM
           RegWrite_EX: in STD_LOGIC;--Señal control escritura memoria etapa EX
           Op_code_ID: in  STD_LOGIC_VECTOR (5 downto 0);--Código operación etapa ID
           avanzar_ID: out STD_LOGIC);
```

La unidad de detención (**UD**) es la que se encarga de parar la ejecución de la instrucción que se encuentra actualmente en la fase de decodificación así como de parar la actualización del PC para evitar la entrada de nuevas instrucciones. Hemos implementado la unidad en un módulo a parte llamado UD_Mips.vhd. Esta unidad complementa a la unidad de anticipación para evitar los siguientes riesgos de datos:

➢ Caso en que tenemos una NOP en fase ID. La unidad de detección no realiza ninguna detención.

```
'1' when Op_code_ID=NOP else -- si se está ejecutando una NOP, no se produce dentenciones
```

➢Caso en que tenemos una instrucción en fase ID y la anterior corresponde a una instrucción de lectura, es decir un load. El registro leído de memoria por el load es guardado en el registro que necesita la instrucción actual para ejecutar.

La unidad de de detención para el banco de registros IF/ID y el PC e inserta una NOP en la fase ID estableciendo una distancia de un ciclo más entre ambas instrucciones. De esta manera permitimos que se anticipe el valor del registro producido por el load en la fase MEM de forma correcta.

```
'0' when Reg_Rs_ID=RW_EX AND MemRead_EX='1' else -- caso Rs en etapa ID, dependencia con load en etapa EX
'0' when Reg_Rt_ID=RW_EX AND MemRead_EX='1' else -- caso Rt en etapa ID, dependencia con load en etapa EX
```

➢ Caso en que tenemos un BEQ o BNE en etapa ID y la instrucción en el ciclo anterior (etapa EX) o la instrucción en los dos ciclos anteriores (etapa MEM) va a escribir en un registro del banco de registros que está leyendo actualmente la instrucción de salto BEQ o BNE. Cómo estas instrucciones no usan la red de anticipación porque realizan todas sus operaciones en la etapa ID, la instrucción productora de la dependencia tiene que llegar a la etapa WB y escribir el nuevo valor del registro en el banco de registros.

La unidad de detención para el banco de registros IF/ID e inserta una NOP en la fase ID. De esta forma permitimos que las instrucciones con dependencia en etapa EX o MEM llegen a la etapa WB y actualicen el banco de registros.

```
'0' when Op_code_ID=BEQ AND Reg_Rs_ID=RW_EX AND RegWrite_EX='1' else  --caso BEQ y dependencia Rs etapa ID con instrucción en etapa EX
'0' when Op_code_ID=BEQ AND Reg_Rt_ID=RW_EX AND RegWrite_EX='1' else   --caso BEQ y dependencia Rt etapa ID con instrucción en etapa EX
'0' when Op_code_ID=BNE AND Reg_Rs_ID=RW_EX AND RegWrite_EX='1' else --caso BNE y dependencia Rs etapa ID con instrucción en etapa EX
'0' when Op_code_ID=BNE AND Reg_Rt_ID=RW_EX AND RegWrite_EX='1' else --caso BNE y dependencia Rt etapa ID con instrucción en etapa EX
'0' when Op_code_ID=BEQ AND Reg_Rs_ID=RW_MEM AND RegWrite_MEM='1' else    --caso BEQ y dependencia Rs etapa ID con instrucción en etapa MEM
'0' when Op_code_ID=BEQ AND Reg_Rt_ID=RW_MEM AND RegWrite_MEM='1' else    --caso BEQ y dependencia Rt etapa ID con instrucción en etapa MEM
'0' when Op_code_ID=BNE AND Reg_Rs_ID=RW_MEM AND RegWrite_MEM='1' else  --caso BNE y dependencia Rs etapa ID con instrucción en etapa MEM
'0' when Op_code_ID=BNE AND Reg_Rt_ID=RW_MEM AND RegWrite_MEM='1' else  --caso BNE y dependencia Rt etapa ID con instrucción en etapa MEM
```

Paso 4: Gestión de riesgos de control

```
entity branch_predictor is
 Port (     clk : in  STD_LOGIC;
            reset : in  STD_LOGIC;
            -- Puerto de lectura se accede con los 8 bits menos significativos de PC+4 sumado en IF
            PC4 : in  STD_LOGIC_VECTOR (7 downto 0);
            branch_address_out : out  STD_LOGIC_VECTOR (31 downto 0); -- dirección de salto
            prediction_out : out  STD_LOGIC; -- indica si hay que saltar a la dirección de salto o no
            -- Puerto de escritura se envía PC+4, la dirección de salto y la predicción, y se activa
            PC4_ID:  in  STD_LOGIC_VECTOR (7 downto 0); -- Etiqueta: 8 bits menos significativos del
            prediction_in : in  STD_LOGIC; -- predicción
            branch_address_in : in  STD_LOGIC_VECTOR (31 downto 0); -- dirección de salto
          update:  in  STD_LOGIC); -- da la orden de actualizar la información del predictor
end branch_predictor;
```

El branch_predictor tiene como señales de entrada el PC4 que corresponde al PC actual, el PC4_ID corresponde al estado del PC4 en la fase anterior, prediction_in es un bit que indica si realmente quería saltar o no la instrucción, branch_address_in dirección de salto calculada en la etapa ID y update_predictor bit que indica si se debe actualizar el predictor o no.

Como señales de salida tiene address_predicted que es la predicción de la dirección en la que se va a realizar el salto y prediction bit que predice si salto tomado o no.

El comportamiento de este módulo es el siguiente:

➢ Si clk=1 y reset=1 las señales del registro se ponen a cero
➢ Si clk=1 y update=1 se actualiza la información del registro con la información proveniente de las señales de entrada.

Por otra parte se saca continuamente la dirección almacenada en el registro a través del address_predicted y se evalúa de manera continua una sentencia la cual decide si por prediction saca un cero o un uno dependiendo de que PC4(9-2) y tag sean iguales así como que la información almacenada en el registro sea válida y la información de si el último salto a sido tomado

A partir de este módulo que se nos proporciona definimos los posibles tipos de errores con los que nos podíamos encontrar.

➢ Error decission_error: Se produce cuando el predictor hace salto tomado en etapa fetch de la instrucción de salto y en la etapa ID de la instrucción de salto se comprueba que el salto era no tomado. También se produce cuando el predictor hace salto no tomado en etapa fetch de la instrucción de salto y en la etapa ID de la instrucción de salto se comprueba que el salto era tomado.

Decission_error está activo cuando que IR_ID(31-26) sea una instrucción BEQ o BNE y address_predicted_ID sea una dirección distinta de DirSalto_ID.

```
'1' when IR_ID(31 downto 26)="000100" AND saltar /= prediction_ID else --caso BEQ y predicción errónea
'1' when IR_ID(31 downto 26)="000101" AND saltar2 /= prediction_ID else --caso BNE y predicción errónea
```

En cualquier otro caso, decission_error vale “0”.

➢ address_error: Se produce cuando dos instrucciones de salto distintas tienen la misma etiqueta y su dirección de salto es distinta.

Hemos considerado que este error nunca se puede llegar a producir puesto que las memorias tienen 7 bits de dirección y el tag almacenado en el predictor es de 8 bits, por lo que nunca se produciría un desbordamiento en el tag que hiciese que distintos PC4 tuvieran una misma etiqueta.

Pese a ello, hemos implementado la siguiente solución:

- Address_error se activaría si el predictor hace salto tomado(PC4=tag) y la instrucción en ID es una instrucción de salto y address_predicted_ID es distinto de DirSalto_ID.

```
'1' when prediction_ID='1' AND IR_ID(31 downto 26)="000100" AND (address_predicted_ID /= DirSalto_ID) else
'1' when prediction_ID='1' AND IR_ID(31 downto 26)="000101" AND (address_predicted_ID /= DirSalto_ID) else
```

Con los errores ya establecidos implementamos la lógica de la señal de control PCsrc del multiplexor “muxPC”, que se encarga de gestionar la actualización del PC con la dirección correspondiente. Sus entradas son:

“00” si se predice salto no tomado y no hay fallo de predicción.
“01” si se predice salto tomado y no hay error de predicción.
“10” si se predice salto tomado y es salto no tomado.
“11” si hay address_error o se predice salto no tomado y es salto tomado.

```
"00" when (prediction = '0' AND predictor_error = '0') else
"01" when (prediction = '1' AND predictor_error = '0') else
"10" when (decission_error = '1' AND prediction_ID='1') else
"11" when (address_error = '1' OR (decission_error = '1' AND prediction_ID='0'));
```

Impacto en rendimiento

El impacto en el rendimiento ha ido aumentando conforme terminábamos de implementar y depurar los módulos.

Los riesgos de datos aparecen cuando existe algún tipo de dependencia entre instrucciones que se ejecutan de forma segmentada. Para solventar este problema, que anteriormente se realizaba interponiendo instrucciones NOP en las situaciones que existia algun cierto riesgo retrasando así la ejecución tantos ciclos como NOP había, hemos implementado las unidades de anticipación y detención cuya combinación causa que en situaciones en las que existe dependencia con una instrucción que no se un LW no se pierda ningún ciclo, en el caso del LW si se encuentra seguidas las ejecuciones solo se tendría que detener la ejecución un ciclo a diferencia de las dos NOP que se introducían de la otra manera. Por todo ello el rendimiento del procesador aumenta.

Los riesgos de control se han solucionado a través de un predictor dinámico que en el mejor de los casos puede predecir el salto en un solo ciclo, este caso se da en los bucles, produciendo que el predictor acierte numerosas veces rebajando los CPI y por lo tanto reduciendo el tiempo de ejecución. Por supuesto también tiene contras, estos se producen cuando el predictor falla lo cual produce que este se actualice e incluso que tenga que recalcular la siguiente instrucción a ejecutar. En conclusión como he señalado, si se ejecuta un programa que hace numerosas iteraciones el rendimiento se verá afectado positivamente pero por el contrario si solo existen algunos saltos aislados puede llegar a repercutir severamente en el rendimiento.

Pruebas

test_la.vhd

```
000010  00010   00001   0000000000000001    LW  R1, 0(R2)   0841 0000
000010  00100   00010   0000000000000011    LW  R2, 0(R4)   0882 0000
001000  00010   00001   0000000000000011    LA  R1, 3(R2)   2041 0003
000011  00001   00010   0000000000000000    SW  R2, 0(R1)   0C22 0000
```

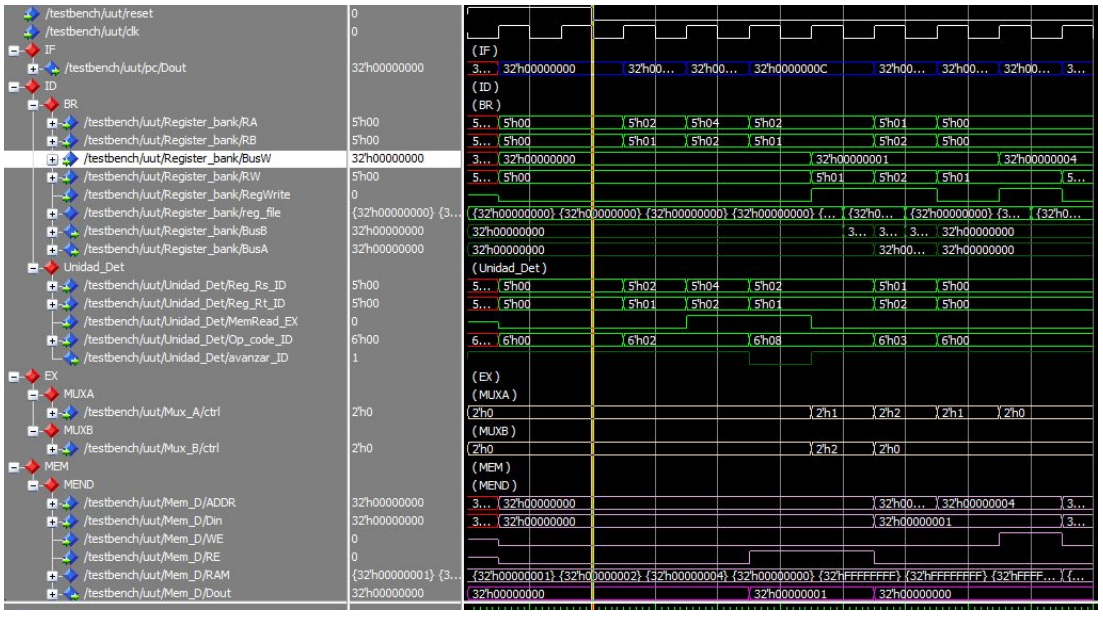
LW
LW ↓ caso 1
LA ↓ caso 2
SW

Este test comprueba los casos:

1. <u>Instrucción LA en fase ID dependiente de instrucción LD en fase EX</u>
   - En el ciclo 4 de la simulación el LA está en etapa ID y necesita leer el registro R2 pero LW todavía está en EX sin haber producido la actualización del registro por lo que avanza se pone a 0 hasta que este está disponible (1 ciclo).
   - En el ciclo 6, LA está en EX y anticipa el operando R2 ya producido por LW que está en WB.
     La entrada del muxA es 2 porque anticipa operando Rs.
     La entrada del "muxB" es 0 porque no anticipa nada.

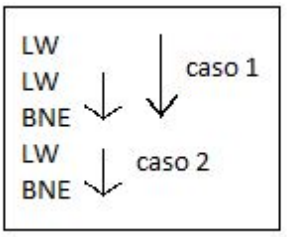
2. <u>Instrucción SW en fase ID dependiente de instrucción LA en fase EX</u>
   - En el ciclo 6 de la simulación el SW está en etapa ID y necesita leer el registro R1 pero ADD todavía está en EX sin haber producido la actualización del registro
   - En el ciclo 7, SW está en EX y anticipa el operando R1 ya producido por LA que está en MEM.
     La entrada del muxA es 1 porque anticipa operando Rs.
     La entrada del "muxB" es 0 porque no anticipa nada.

3. <u>Funcionamiento nueva instrucción</u>
   - En el ciclo 8 de la simulación, LA está en etapa WB y almacena el valor calculado (4) a través de BusW en RW (1) en la segunda parte de dicho ciclo, cuando RegWrite=1



test_bne.vhd

```
000010  00010  00001  0000000000000000        LW   R1, 0(R2)    0841 0000
000010  00100  00011  0000000000000000        LW   R3, 0(R4)    0883 0000
000101  00001  00011  1111111111111101        bne r1, r3,dir0 1423 FFFD
000010  00100  00011  0000000000000100        LW   R3, 4(R4)    0883 0004
000101  00001  00011  1111111111111011        bne r1, r3,dir0 1423 FFFB
```

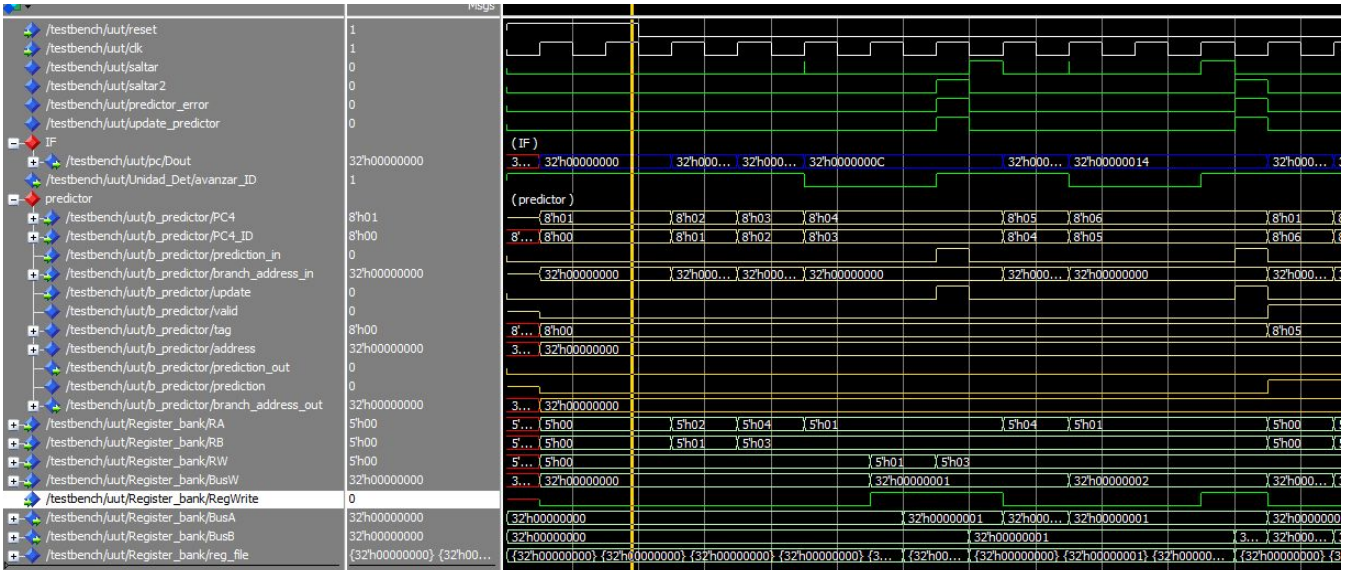


Este test comprueba los casos:

1. <u>Instrucción BNE en fase ID dependiente de instrucción LD en fase MEM y EX.</u>
   - En el ciclo 4 de la simulación el BNE está en etapa ID y necesita leer el registro R3 y R1 pero un LW todavía está en MEM y el otro en EX sin haber producido la actualización del registro.
   - En el ciclo 6 , BNE está en ID y toma el operando R3 y R1 ya producido por los LW y almacenados en el BR en sus respectivos WB.

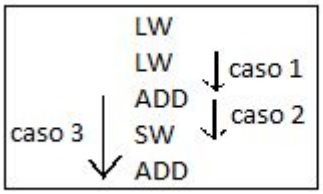
2. <u>Instrucción BNE en fase ID dependiente de instrucción LD en fase EX.</u>

3. <u>Funcionamiento nueva instrucción</u>
   - En el ciclo 6 de la simulación, BNE no toma el salto ya que ambos registros tiene el mismo valor como se puede ver en los buses A y B.
   - En el ciclo 10 de la simulación, BNE toma el salto debido a que como muestran busA y busB los valores son distintos por lo que saltar2 =1.



test_anticipacion1.vhd

```
000010  00010   00001   0000000000000000        LW  R1, 0(R2)   0841 0000
000010  00100   00011   0000000000000000        LW  R3, 0(R4)   0883 0000
000001  00011   00001   00001   00000   000000  ADD R1, R3, R1  0461 0800
000011  00100   00001   0000000000000000        SW  R1, 0(R4)   0C81 0000
000001  00011   00001   00001   00000   000000  ADD R1, R3, R1  0461 0800
```

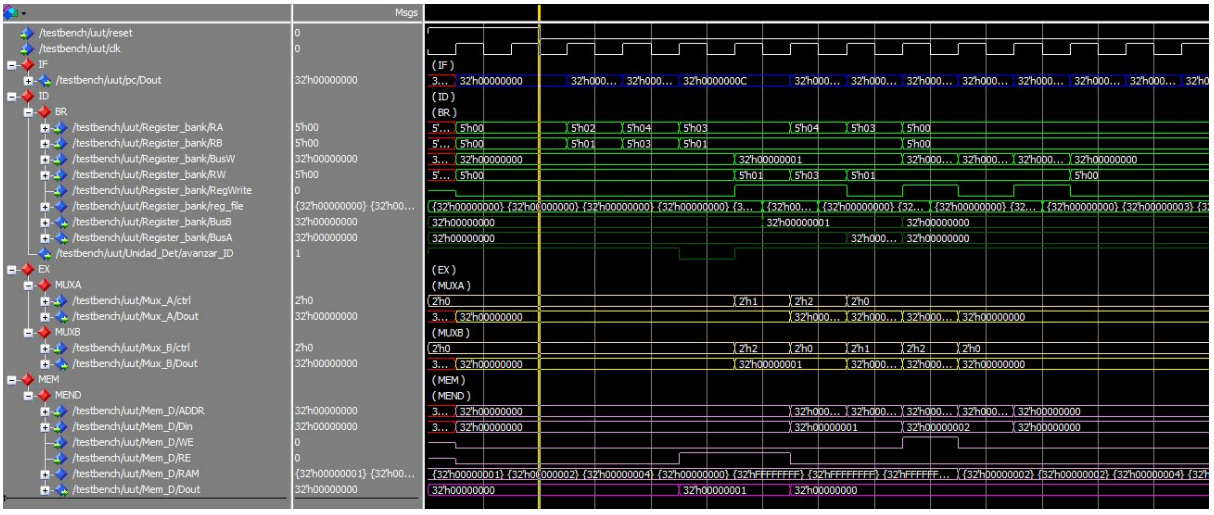


Este test comprueba los casos:

1. <u>Instrucción ADD en fase ID dependiente de instrucción LD en fase MEM y EX</u>
   - En el ciclo 4 de la simulación el ADD está en etapa ID y necesita leer el registro R1 y R3 pero los LW todavía están en MEM y EX sin haber producido la actualización del registro R0.
   - En el ciclo 6, ADD está en EX y anticipa el operando R3 ya producido por LW que está en WB.
     La entrada del muxA es 2 porque anticipa operando Rs.
     La entrada del "muxB" es 0 porque no anticipa nada.

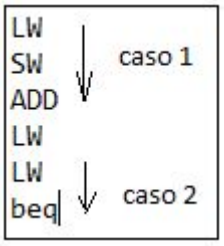
2. <u>Instrucción SW en fase ID dependiente de instrucción ADD en fase EX</u>
   - En el ciclo 6 de la simulación, SW está en etapa ID y necesita leer del registro R1 pero ADD todavía está en EX sin haber producido la actualización del registro.
   - En el ciclo 7 de la simulación, SW está en etapa EX y anticipa el operando R1 ya producido por ADD que está en MEM.
     La entrada del muxA es 1 porque anticipa operando Rs.
     La entrada del "muxB" es 0 porque no anticipa nada.

3. <u>Instrucción ADD en fase ID dependiente de instrucción ADD en MEM</u>
   - En el ciclo 8 de la simulación, ADD esta en etapa EX y necesita leer del registro R1 pero ADD todavía está en MEM sin haber producido la actualización del registro así que anticipa el registro.
     La entrada del muxA es 0 porque no existe dependencia con SW.
     La entrada del "muxB" es 1 porque no anticipa RT



test_anticipacion2.vhd

```
000010  00010   00000   0000000000000000        LW R0, 0(R2)     0840 0000
000011  00011   00001   0000000000000000        SW  R1, 0(R3)    0C61 0000
000001  00000   00010   00001   00000   000000  ADD R1, R0, R2   0402 0800
000010  00101   00100   0000000000000100        LW  R4, 4(R5)    08A4 0004
000010  00010   00000   0000000000000000        LW  R0, 0(R2)    0840 0000
000100  00000   00000   1111111111111010        beq r0, r0,dir0 1000 FFFA
```



Este test comprueba los casos:

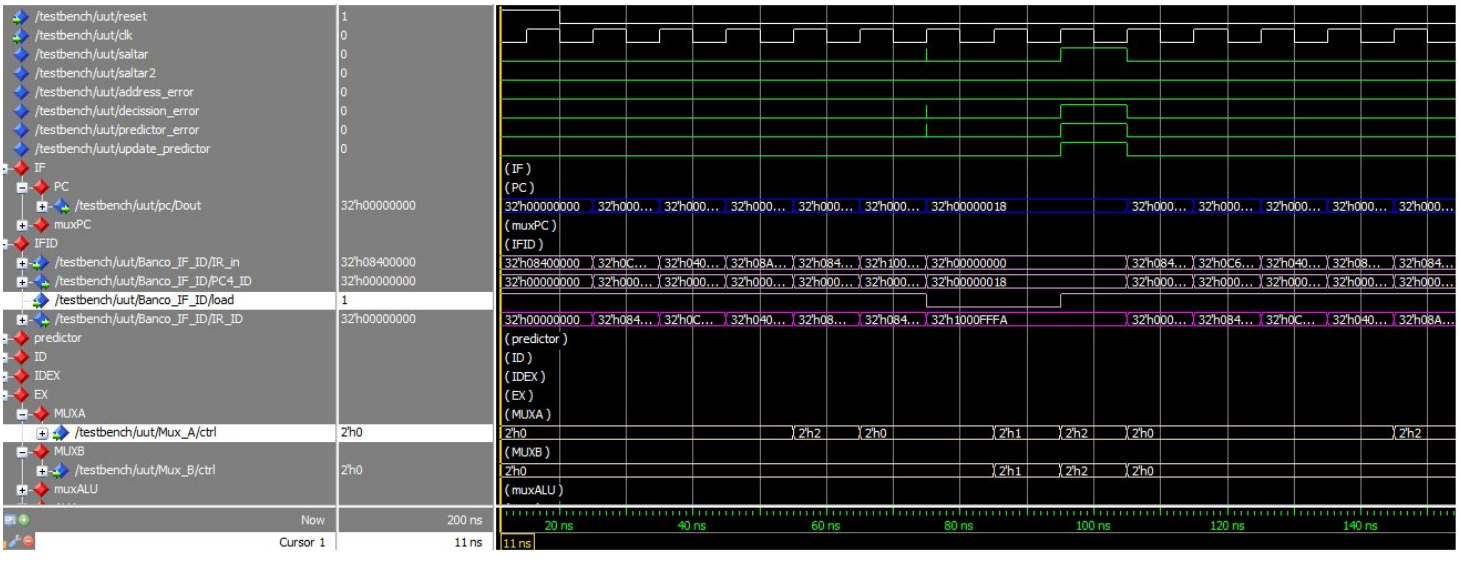
1. <u>Instrucción ADD en fase ID dependiente de instrucción LD en fase MEM</u>
   - En el ciclo 4 de la simulación el ADD está en etapa ID y necesita leer el registro R0 pero LW todavía está en MEM sin haber producido la actualización del registro R0.
   - En el ciclo 5, ADD está en EX y anticipa el operando R0 ya producido por LW que está en WB.

     La entrada del muxA es 2 porque anticipa operando Rs.

     La entrada del "muxB" es 0 porque no anticipa nada.

<u>2.Instrucción BEQ en fase ID dependiente de instrucción LD en fase EX</u>

   - En el ciclo 7 de la simulación, BEQ está en etapa ID y necesita leer del registro R0 pero LW todavía está en EX sin haber producido la actualización del registro R0. Se produce primera detención.
   - En el ciclo 8 de la simulación, BEQ está en etapa ID y necesita leer del registro R0 pero LW todavía está en MEM sin haber producido la actualización del registro R0. Se produce segunda detención.
   - En el ciclo 9 de la simulación, LW está en etapa WB y actualiza R0 en el banco de registros. BEQ lee el registro R0 ya actualizado.



test_predictor1.vhd

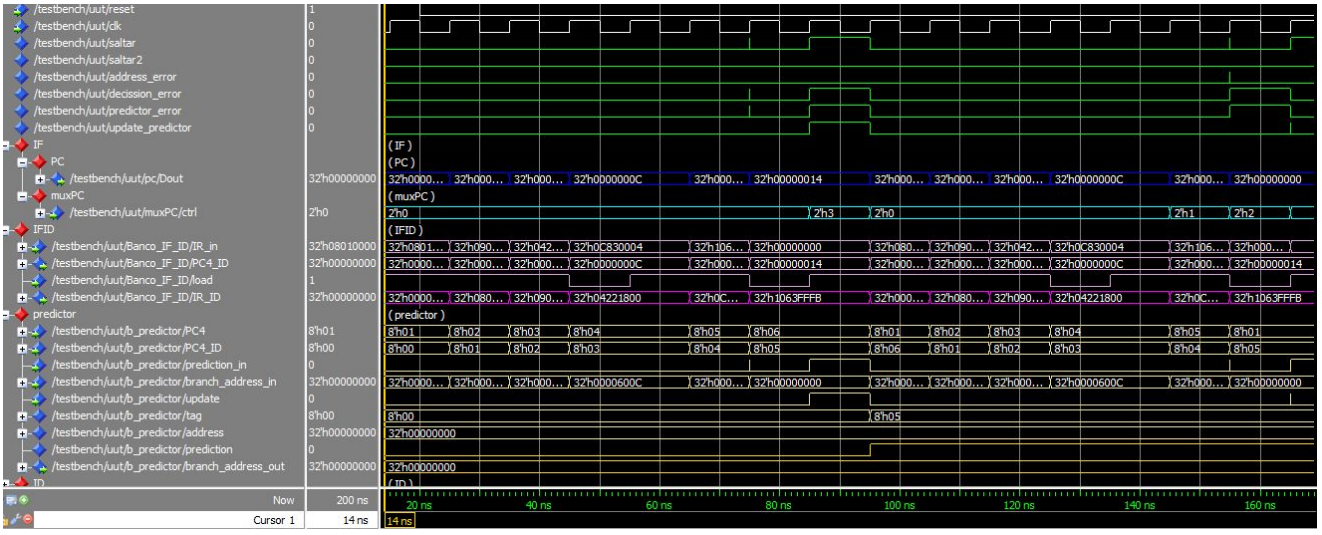
```
000010  00000   00001   0000000000000000        LW  R1, 0(R0)    0801 0000
000010  01000   00010   0000000000000100        LW  R2, 4(R8)    0902 0004
000001  00001   00010   00011   00000   000000  ADD R3, R1, R2   0422 1800
000011  00100   00011   0000000000000100        SW  R3, 4(R4)    0C83 0004
000100  00011   00011   1111111111111011        beq r3, r3,dir0 1063 FFFB
```

Este test comprueba los casos:

1. <u>Predictor predice salto NT de forma errónea</u>
   - En el ciclo 6 de la simulación, la instrucción BEQ está en fase IF, el predictor predice salto NT. La señal PCsrc vale “00”, y la entrada del “muxPC” corresponde a PC4
   - En el ciclo 7 de la simulación, la instrucción BEQ está en fase ID, debido a la dependencia con el ADD que está en fase MEM, se produce una detención hasta tener el registro R3 actualizado.
   - En el ciclo 8 de la simulación, la instrucción BEQ sigue en fase ID y se activa el decission_error, porque saltar vale “1”, avanzar_ID vale “1” y la predicción en el ciclo anterior (prediction_ID), vale “0”.
     Por lo tanto se actualiza el predictor de saltos con predicción a “1”.
     La señal PCsrc vale “3”, y la entrada del “muxPC” corresponde a DirSalto_ID.
     Además se mandará una NOP a la fase ID en el ciclo siguiente para eliminar la instrucción anterior errónea.

<u>2.Predictor predice salto T de forma correcta</u>

   - En el ciclo 14 de la simulación, (segunda iteración). La instrucción BEQ vuelve a estar en la etapa IF, el predictor predice salto T. La señal PCsrc vale “1” y la entrada del “muxPC” corresponde a address_predicted.
   - En el ciclo 15 de la simulación, la instrucción BEQ está en fase ID debido a la dependencia con el ADD que está en fase MEM, se produce una detención hasta tener el registro R3 actualizado.
   - En el ciclo 16 de la simulación, la instrucción BEQ sigue en la fase ID, no se activa ningún error. Por lo tanto la señal PCsrc vale “0”, y la entrada del “muxPC” corresponde a PC4.



test_predictor2.vhd

```
001000  00010   00001   0000000000000001        LA  R1, 1(R2)   2041 0001
001000  00010   00101   0000000000000010        LA  R5, 2(R2)   2045 0002
000001  00001   00010   00010   00000   000000  ADD R2, R1, R2  0422 1000
000011  00001   00011   0000000000000000        SW  R3, 0(R1)   0C23 0000
000101  00010   00101   1111111111111101        bne r2, r5,dir3 1445 FFFD
000010  00000   00001   0000000000000000        LW  R1, 0(R0)   0801 0000
```

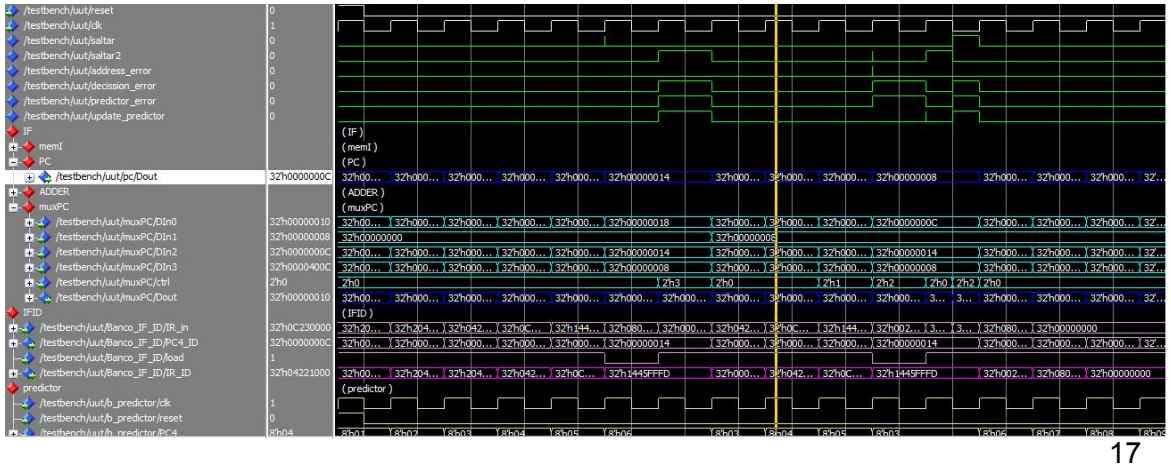
Este test comprueba los casos:

1. Predictor predice salto NT de forma errónea

- En el ciclo 5 de la simulación, la instrucción BNE está en fase IF, el predictor predice salto NT. La señal PCsrc vale “00”, y la entrada del “muxPC” corresponde a PC4
- En el ciclo 6 de la simulación, la instrucción BNE está en fase ID, debido a la dependencia con el ADD que está en fase MEM, se produce una detención hasta tener el registro R2 actualizado.
- En el ciclo 7 de la simulación, la instrucción BNE sigue en fase ID y se activa el decission_error, porque saltar2 vale “1”, avanzar_ID vale “1” y la predicción en el ciclo anterior (prediction_ID), vale “0”.
  Por lo tanto se actualiza el predictor de saltos con predicción a “1”.
  La señal PCsrc vale “3”, y la entrada del “muxPC” corresponde a DirSalto_ID.
  Además se mandará una NOP a la fase ID en el ciclo siguiente para eliminar la instrucción anterior errónea.

2.Predictor predice salto T de forma errónea

- En el ciclo 10 de la simulación, la instrucción BNE está en fase IF, el predictor predice salto T. La señal PCsrc vale “01”, y la entrada del “muxPC” corresponde a address_predicted.
- En el ciclo 11 de la simulación, la instrucción BNE está en fase ID, debido a la dependencia con el ADD que está en fase MEM, se produce una detención hasta tener el registro R2 actualizado.
- En el ciclo 12 de la simulación, la instrucción BNE sigue en fase ID y se activa el decission_error, porque saltar2 vale “0”, avanzar_ID vale “1” y la predicción en el ciclo anterior (prediction_ID), vale “1”.
  Por lo tanto se actualiza el predictor de saltos con predicción a “0”.
  La señal PCsrc vale “2”, y la entrada del “muxPC” corresponde a PC4_ID..
  Además se mandará una NOP a la fase ID en el ciclo siguiente para eliminar la instrucción anterior errónea.



APORTACIONES Y TIEMPO DEDICADO

Diego Marco

Trabajo inicial	8h
Añadir nuevas instrucciones	2h
Gestión de riesgo de datos	2h
Gestión de riesgo de control	4h
Depuración, ajustes y bancos de prueba	30h
Redacción memoria	6h
TOTAL:	**52h**

Alejandro Terrón

Trabajo inicial	3h
Añadir nuevas instrucciones	2h
Gestión de riesgo de datos	2h
Gestión de riesgo de control	4h
Depuración, ajustes y bancos de prueba	30h
Redacción memoria	6h
TOTAL:	**47h**

En el inicio tomamos la decisión de dibujar el mips en un papel para visualizar todas las conexiones y no equivocarnos, lo que nos llevó bastante tiempo. Añadir las instrucciones y la gestión de riesgos de datos no nos supuso muchos problemas, por lo que no tardamos mucho en terminar estos apartados. La gestión de riesgos de control la implementamos mal al principio y eso nos supuso aumentar el tiempo debugeando hasta que pudimos arreglarlo. Al depurar varios programas pensados por nosotros nos salieron varios fallos que nos costó identificar y solucionar. Los bancos de pruebas también nos llevó tiempo para revisar ciclo a ciclo que se ejecutaban correctamente y que la gestión de riesgos de datos y de control funcionaba. Finalmente nos pusimos a redactar la memoria añadiendo fragmentos de código para contrastar los datos que os damos, y en los bancos de prueba añadimos una captura de las señales más importantes para cada caso.