Proyecto I AOC II

Nestor Monzón Gonzalez 735418
Jose Manuel Sánchez Aquilué 759267

Explicación de nuestro diseño y hardware añadido.

A la hora de diseñar esta versión del Mips nos hemos ceñido en esencia a las instrucciones del enunciado.

Con el propósito de incorporar las instrucciones load address y bne fue necesario modificar la unidad de control e incluso añadir un nuevo flag. Como nuestro diseño puede anticipar operandos y predecir saltos cuenta con una serie de señales en la ruta de datos y una unidad de anticipación. Cada señal sigue su una lógica que hace posible el funcionamiento del mips.

En primer lugar, al añadir dos instrucciones hizo falta cambiar la unidad de control, detallando los flags que se activan en cada instrucción nueva. Del mismo modo, añadimos un flag a la unidad de control llamando 'BNE' cuyo valor es igual a uno si y sólo si es una instrucción bne. Se utiliza para diferenciar las condiciones de salto de beq y bne.

A la hora de implementar el predictor de salto fue necesario añadir una nueva señal, saltar, en la etapa ID. Esta es igual a uno cuando se ha de saltar. Por lo tanto, evalúa el flag Z y el BNE.

Otro cambio destacable fue la fuente del mux pc, que en este caso debe permitir la salida uno y dos. Con esta mejora, cuando el bit de predicción vale uno; es decir cuando la instrucción leída tiene la misma etiqueta que la almacenada y la última vez que evaluó un salto con esa etiqueta se saltó; se tiene que saltar antes de evaluar el salto. En otras palabras, si la salida del predictor de saltos es uno, la fuente del mux pc vale 1 y pc contiene la dirección del predictor. Puede ocurrir que la dirección predicha sea incorrecta, en cuyo caso hay que tener en cuenta la señal de error de predicción, que detallaré más adelante. Además, en caso de fallo de predicción, en función de si el flag saltar, nombrado anteriormente, es igual a cero o a uno, la fuente del pc será dos o tres (respectivamente). En el resto de situaciones se tomará la primera salida del mux pc y pc contendrá la dirección de la instrucción siguiente.

Al haber discernido entre dos tipos de errores de predicción, existen dos señales, una para cada error. El error de dirección se da cuando la dirección a la que se ha saltado es distinta a la que se tenía que haber ido. Para determinar si se ha producido ese error, tengo que comparar la dirección del predicha con la del salto. Además se tiene que haber dado la situación de que haya saltado, con lo cual necesito una puerta lógica para determinar si se dan las dos condiciones. Por otra parte, se puede producir un error de decisión, cuando he saltado y no debía saltar o cuando no he saltado y debía saltar. Con lo cual he de comparar esa nueva señal saltar con predictio_ID, si son distintas, se habrá producido un error de decisión. Finalmente, determinamos que se ha dado un error de predicción si alguna de esas dos

señales de error descritas anteriormente es igual a uno y si estoy en una instrucción de salto (Branch=1).

Cuando la predicción es errónea actualizo el predictor. Es decir, conecto la señal de error a el bit de actualización del predictor de salto. Cuando actualizo debo guardar la dirección de salto de esta etapa (DirSaltoo_ID) y como bit de predicción la señal saltar.

En cuanto a la unidad de detención, hemos añadidos dos señales, una para cada tipo.

Para los riesgos de load_uso, hemos definido en primer lugar la señal RegRead, que determina si la instrucción en la etapa de Decodificación necesita leer del BR. En nuestro caso, todas las instrucciones, excepto “nop” y “la” activan esta señal. Aunque intentamos añadirla a la UC, nos dio problemas que no conseguimos solucionar, con lo que finalmente optamos por añadirla en la ruta de datos a partir del código de la operación. A nivel de hardware, esto ha supuesto la adición de una puerta XNOR (de igualdad) de seis entradas para cada señal, y una OR de seis entradas (que agrupe las salidas de dichas puertas).

Para los riesgos de salto, se ha definido la señal “Salto”, que es 1 únicamente para las operaciones “beq” y “bne”. Para calcularla, también se ha comparado el código de operación, con un coste similar al de RegRead, aunque con solo dos XNORs. Aunque intentamos utilizar la señal de Branch, que finalmente codifica la misma información y en el mismo ciclo, también presentó problemas a la hora de realizar las simulaciones.

El cálculo de las propias señales de riesgos también tiene un montaje similar en cuanto a hardware, ya que en ambos casos requiere una puerta AND para cada una (con 3 entradas en ambos casos), y tres comparadores en las entradas de las mismas.

Finalmente, estas señales se unen en un conjunto de puertas OR, que dan lugar a las señales riesgo_lw_uso, riesgo_beq, y su combinación, avanzar_ID. Esta última, a su vez, determina la detención enviando a través de la pipeline o bien el código de operación que le llega, si no hay riesgo, o bien el de una “nop”, en caso contrario.

Por otra parte, en la Unidad de Anticipación se ha añadido la lógica que compara los registros destino y de lectura, así como la etapa en la que se escribirá para determinar el origen apropiado del dato buscado (es decir, lo que será la señal de control de los MUXes A y B). Para esto, se han necesitado cuatro puertas AND de dos entradas, en las cuales se sitúan ocho comparadores. Por último, cada una las dos señales de control de los MUXes supone a su vez un MUX de dos bits cuyas señales de control dependen a su vez de las salidas de las ANDs.

Así, la UA selecciona como entrada de RA (o RB) el del resultado de la instrucción en etapa MEM si necesita ese dato (corto_MEM). Si no se cumple esa condición y sí la del corto_WB, se tomara el dato de la etapa WB y, si no se da ninguna de estas, simplemente tomará el dato de salida del BR.

Impacto en rendimiento de la gestión de los riesgos.

A la hora de evaluar el rendimiento hemos tenido utilizado los códigos de ejemplo que nos dieron.

El primero consta de 8 instrucciones por iteración y su CPI es igual a uno.

```
08010000    LW  R1, 0(R0)
09020004    LW  R2, 4(R8)
00000000    nop
00000000    nop
04221800    ADD R3, R1, R2
00000000    nop
1063FFF9    beq r0, r0, dir0
0C830004    SW  R3, 4(R4)
```

El segundo, que da los mismos resultados que el primero pero solo es soportado por el mips modificado, cuenta con 5 instruciones y su CPI 7/5.

```
08010000    LW  R1, 0(R0)
09020004    LW  R2, 4(R8)
04221800    ADD R3, R1, R2
0C830004    SW  R3, 4(R4)
1063FFFB    beq r3, r3, dir0;
```

Por Tanto, el tiempo de ejecución del primero es de 80*NI ns, siendo NI el número de iteraciones. Mientras que en la segunda versión es de 70*NI (como se puede observar en la imagen más abajo). En suma, hemos la mejora de rendimiento es del 14%.

Hemos observado el rendimiento de la nuestra versión del mips es mayor, a pesar de que, como es habitual, el CPI ha aumentado al eliminar las nops.

Pruebas realizadas.

Prueba la.

Cuando se creó la instrucción load address, antes de pasar a la siguiente, la probamos con el siguiente código.

```
0     08010000    LW    R1, 0(R0); r1=1
4     09020004    LW    R2, 4(R8); r2=2
8     00000000    nop
C     00000000    nop
10    20410010    LA    R1, 16(R2); r1=0x12
```

Las pruebas concluyeron de manera satisfactoria, cada registro contenía el valor esperado en el momento esperado.

Ejemplo nops.

A la hora de probar el predictor de saltos, decidimos hacer las primeras pruebas con instrucciones nops. De ese modo, en caso de error, podíamos determinar si se trataba de un fallo del detector de riesgos o del predictor.

El primer código que pusimos a prueba fue el que nos dieron de ejemplo.

```
08010000    LW  R1, 0(R0)
09020004    LW  R2, 4(R8)
00000000    nop
00000000    nop
04221800    ADD R3, R1, R2
00000000    nop
1063FFF9    beq r0, r0, dir0
0C830004    SW  R3, 4(R4)
```

Naturalmente, no funcionó en la primera ejecución. Se produjeron un par de fallos desdeñables que fueron corregidos de inmediato. Finalmente, el resultado fue el esperado, en la primera iteración se cargó la instrucción ulterior al salto, se actualizó el predictor y la orden store fue sustituida por una nop.En el resto de iteraciones se saltaba antes de evaluar el salto.

Prueba salto1.

Para concluir las pruebas nos servimos de un código que compuesto por un bucle con un condicional en su interior. **Es necesario cargar unos determinados valores en la memoria de datos, de lo contrario el programa no se ejecuta satisfactoriamente. (Memoria_RAM_pruebaSalto1)**

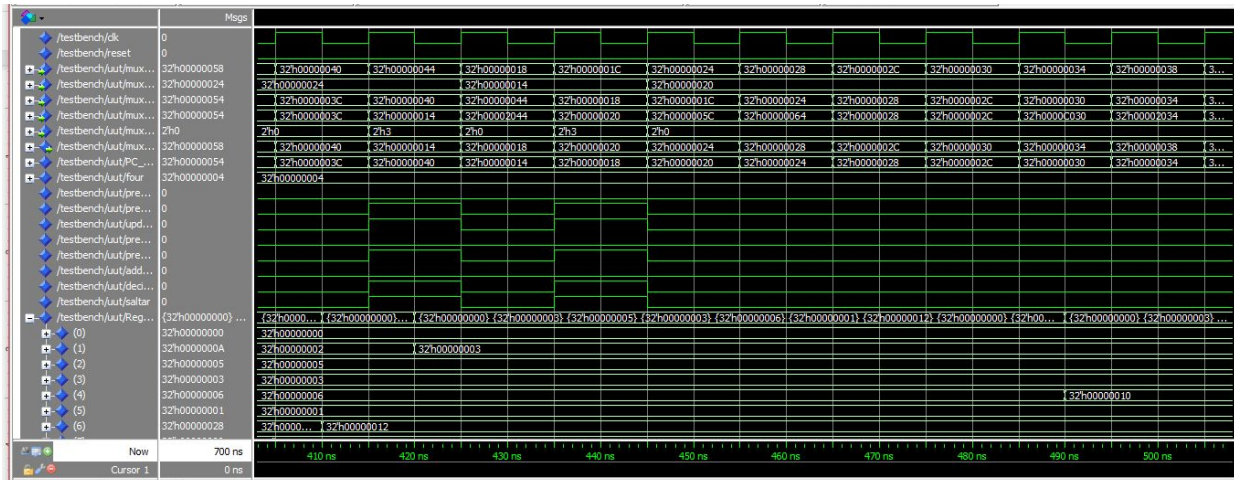
Código en c.

```
i=0
j=0
n=0
do{
        if(i=3){
                j=16;
        }
        else{
                j=6;
        }
        n+=j;
        I++;
}while(i<5)
i+= 5;
// n=40 ^ i=10
```

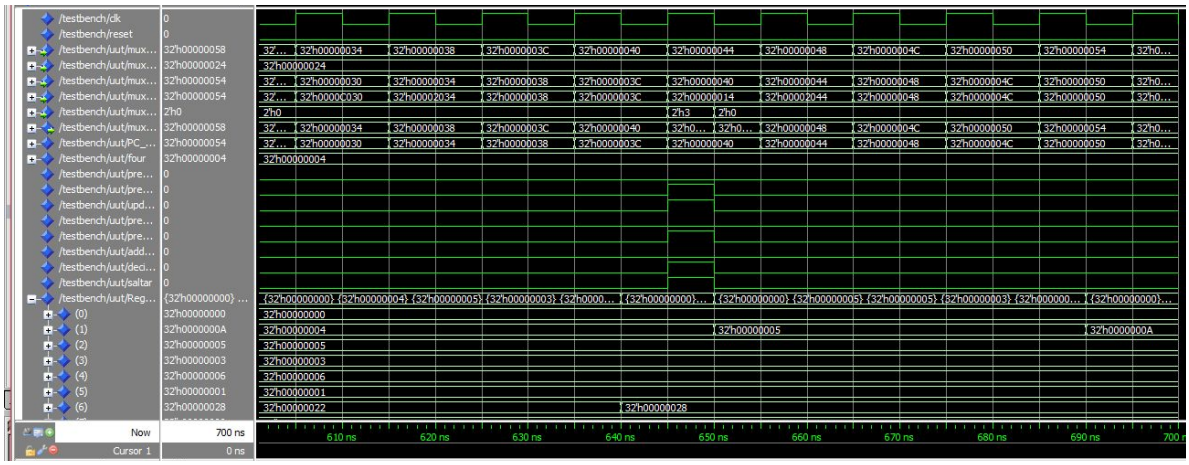
Código en ensamblador.

```
0       08010000    LW      R1, 0(R0); r1=i=0
4       09020004    LW      R2, 4(R8); r2=5
8       09230008    LW      R3, 8(R9); r3=3
C       0945000C    LW      R5, C(R10); r5=1
10      00000000    nop
14      10230002    beq R1, R3, else
18      08040010    LW      R4, 10(R0); r4=6
1C      10000001    beq R0, R0, end
20      20040010    LA  R4, 10(R0);
24      00000000    nop
28      00000000    nop
2C      04C43000    ADD R6, R4, R6
30      04250800    ADD R1, R5, R1
34      00000000    nop
38      00000000    nop
3C      1422FFF5    bne R1, R2, while
40      04220800    ADD    R1, R2,R1
```

Como hay dos saltos diferentes intercalados, el predictor está continuamente leyendo etiquetas distintas y actualizandose. Por tanto, no llega a predecir ningún salto.



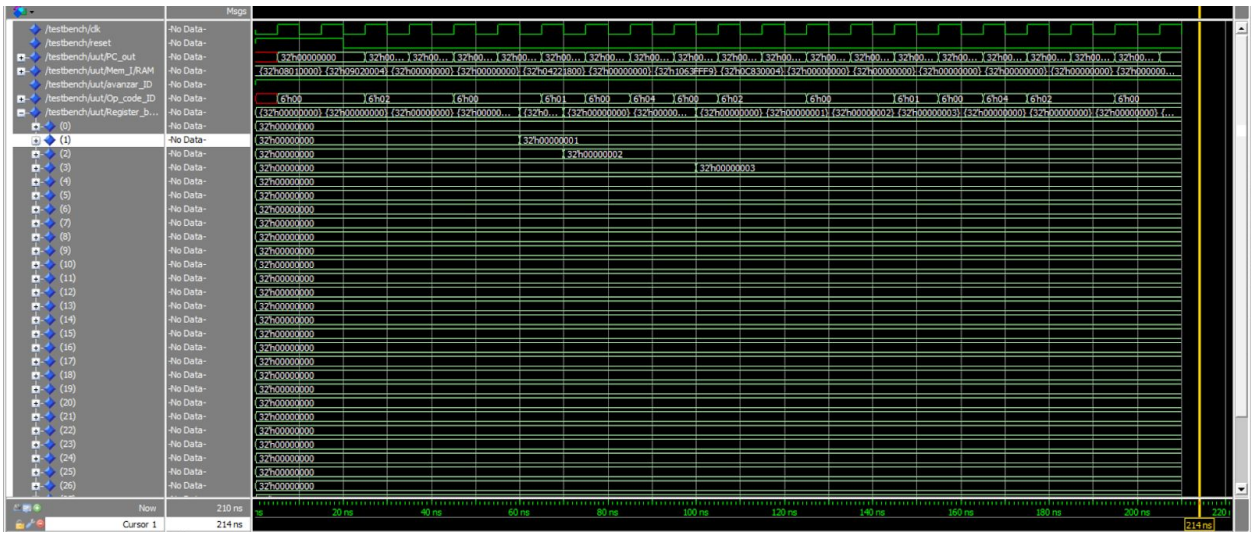
Gracias a este código de ejemplo pudimos detectar un fallo de implementación. En nuestra primera versión saltaban errores de predicción al ejecutar instrucciones distintas a los saltos. Finalmente fue corregido y como resultado se obtuvo lo esperado, el registro r6 con 40 y r1 con 10.



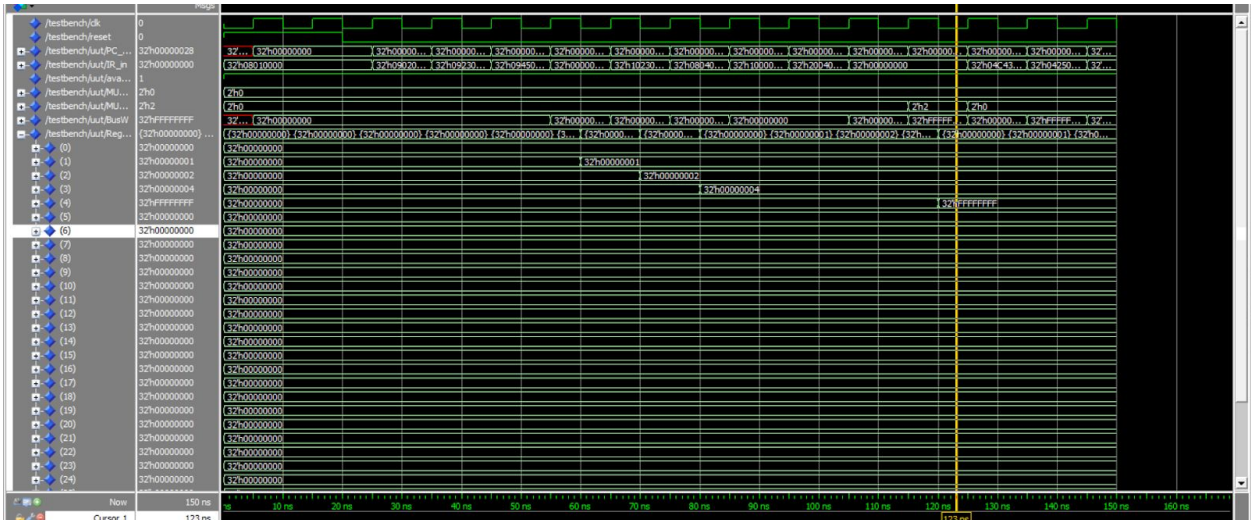
Si bien hemos tenido en cuenta que se pueda producir un error de dirección, en el módulo que nos dieron la memoria tiene direcciones de 7 bits y la etiqueta que almacena el predictor es de 8. De modo que, es imposible que se llegue a dar un error de ese tipo. Si contasemos con una memoria más grande, sí que podría darse la situación de que dos saltos compartieran una misma etiqueta pero cada uno tuviese su propia dirección de destino. De ser así, el predictor detectaría el error y enmendaría el fallo de predicción.

Prueba de anticipación:

Primera simulación conjunta (con la parte 4 corregida y la 3 tras implementar la primera versión de la anticipación de operandos), con la MI original (**ejemplo_nops**):



Mismo código, pero con la MI de la prueba salto1:



El problema era que la UA activaba la entrada 10 del mux (la que hace la anticipación desde la etapa WB de dos instrucciones antes) basada en:

Corto_B_WB <= '1' when (Reg_Rt_EX = RW_WB and RegWrite_WB = '1') else '0';

Así, la instrucción ‘la’ que se ejecuta dos ciclos antes, presenta precisamente el RegWrite_WB = ‘1’. Se debe tratar a ‘la’ como instrucción aritmética en el sentido de que ya tiene disponible su resultado en la etapa EX. Para esto, hemos decidido utilizar la señal de la UC MemtoReg, que controla el MUX a la salida de la Memoria de Datos, ya que solo es ‘1’ en el caso que nos interesa, cuando la instrucción es ‘lw’.

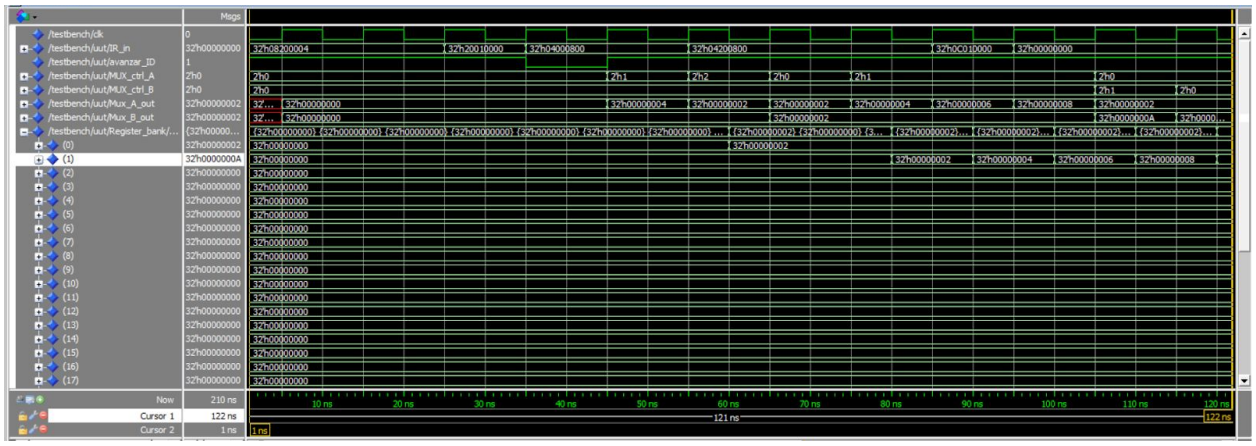
Tras solucionar este problema, nos dimos cuenta de que una nueva señal introducida en la UC, RegRead, que es ‘1’ solo si la instrucción pertinente necesita leer del BR, no era manipulada y transmitida por las etapas correctamente, cayendo en un bucle infinito que hacía imposibles las simulaciones en ModelSim.

La solución que tomamos finalmente fue analizar el propio código de la operación en la etapa ID. Aunque es una solución mucho menos flexible y escalable, ya que cada nueva instrucción que se quiera añadir a la UC supondrá un cambio en esta señal, tuvimos que optar por esta ya que no encontramos la raíz del problema de la señal.

Prueba del programa de anticipación y detenciones:

Programa en ensamblador y binario (separado por campos):

```
000010 00001 00000 0000000000000100        lw r0, 4(r1) ; r0=@(4+r1)=
001000 00000 00001 0000000000000000        la r1, 0(r0) ; r1=@4=0 (detencion)
000001 00000 00000 00001 00000 000000      add r1, r0, r0 ; r1 = 4+4 = 8
000001 00001 00000 00001 00000 000000      add r1, r1, r0 ; r1= 12
000001 00001 00000 00001 00000 000000      add r1, r1, r0 ;          16
000001 00001 00000 00001 00000 000000      add r1, r1, r0 ;   = 20 (0x14)
000011 00000 00001 0000000000000000     sw  r1, 0(r0)  ; @r0 = @4 = 20 (0x14)
```

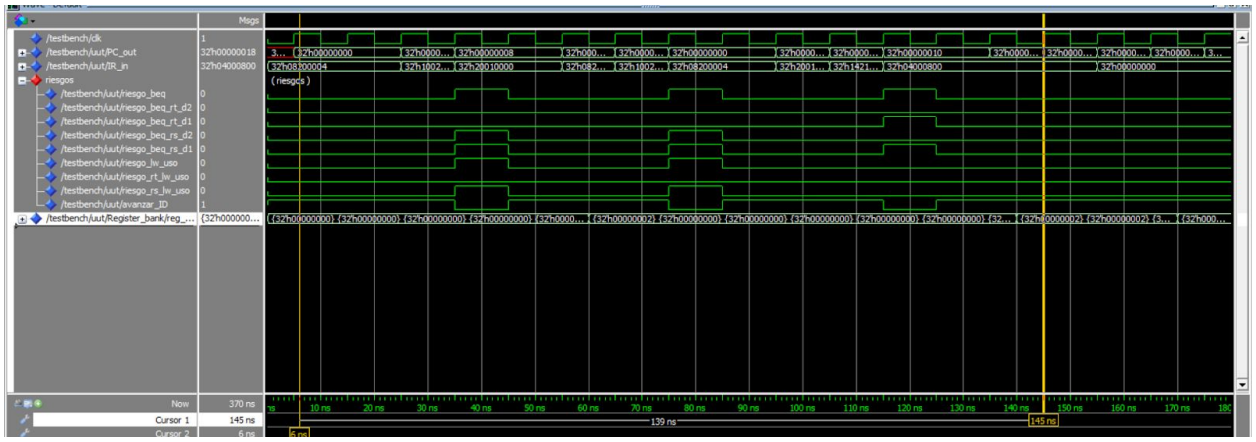


En este caso, una vez arreglado el problema con RegRead, observamos que la detención necesaria entre lw y la se produce (avanzarID=’0’), y que las anticipaciones también llevan al resultado adecuado. En este caso, por el contenido de la memoria de datos, se suma 2 cinco veces en r1, con lo que queda r1=0xA, y se guarda correctamente en la posición de memoria apropiada @r0, en este caso @2.

Pruebas de detenciones con saltos:

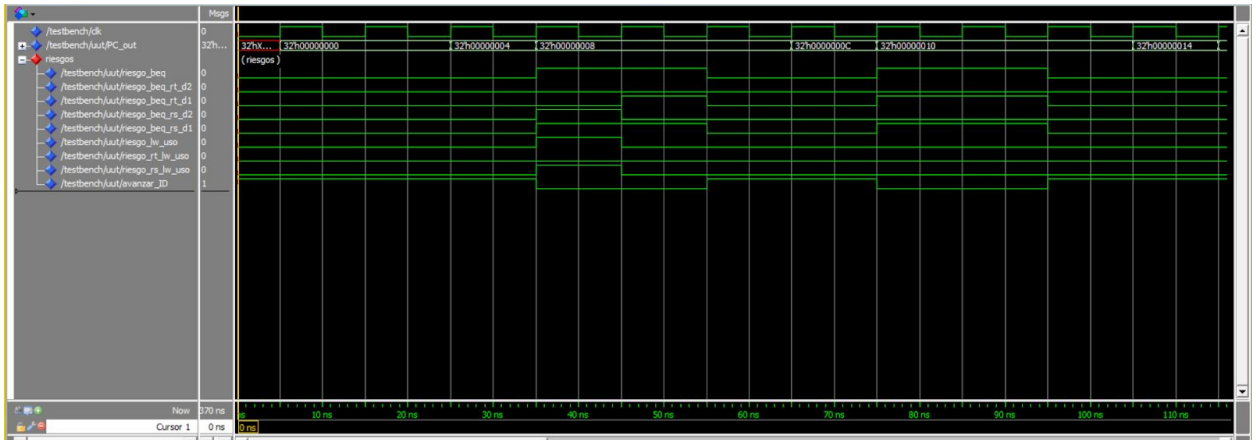
Programa:

000010 00001 00000 0000000000000100	lw r0, 4(r1) ; r0=2
000100 00000 00010 1111111111111110	beq r0, r2, dir4 ; doble detención
001000 00000 00001 0000000000000000	la r1, 0(r0) ; r1=r0=2 (detencion)
000101 00001 00001 1111111111110000	bne r1,r1,dir0
000001 00000 00000 00001 00000 000000	add r1, r0, r0 ; r1 = r0+r0



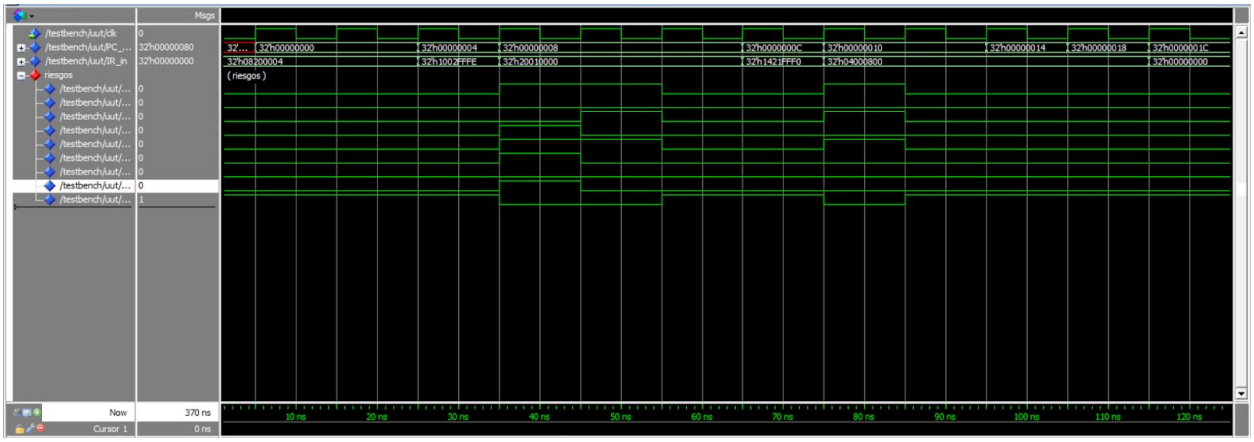
En este punto, observamos que había un problema con la detección de detenciones de distancia 1: pasaba de marcar un riesgo a distancia 2 a ninguno, cuando evidentemente debía seguir quedando a distancia 1. Esto se debía a que no teníamos en cuenta el caso en que la instrucción en etapa M no tuviera disponible el hasta esa misma etapa (caso de las lw).

Tras añadir la lógica que identifica estas situaciones en las señales de riesgos de beq, el diagrama queda como sigue:



Como se puede apreciar, ahora las señales identifican satisfactoriamente los casos, resultando en una bajada a su vez de la señal avanzar_ID, y la correspondiente detención de dos ciclos esperada. No obstante, un error ha producido que se detenga dos ciclos también tras

la instrucción ’la’, siendo que ya tiene disponible su dato en la etapa M. Una vez arreglado esto, el diagrama es el apropiado:



Resumen de nuestras aportaciones.

José Manuel.

Fecha.	Tiempo dedicado.	Tipo de tarea.
15/4/19	2h	Ver los videos, instalar el programa y familiarizarme con él.
19/4/19	30m	Implementar instrucción la.
19/4/19	30m	Comprobar que funciona instrucción la.
20/4/19	3h	Empezar con el predictor de saltos.
21/4/19	3h	Probar predictor de saltos.
23/4/19	45m	Esbozar un borrador de la memoria.
24/4/19	1h 30 min	Pasar a limpio los resultados de las pruebas.

Nestor

Fecha.	Tiempo dedicado.	Tipo de tarea.
18/4/19	4h	Ver los videos e instalar el programa, conseguir que funcionara (tuve problemas de todo tipo)
18/4/19	3h	Completar la parte 1. Por algún motivo el PC no avanzaba en las simulaciones.
20/4/19	2h	Implementar bne.
20/4/19	1h	Reparto de tareas, José parte 4 y yo la 3.
22/4/19	2h	Pruebas de 'bne' conjuntas con 'la' de José.
25/4/19	2h	Diseño y primera implementación de la anticipación.
26/4/19	5h	Depuración de la UA, diseño e implementación de la lógica de detenciones
27/4/19	4h	Pruebas y depuración de las detenciones
27/4/19	5h	Pruebas en limpio, corrección de errores generales, memoria
27/4/19	2h	Terminar memoria