PROYECTO 1:

MIPS SEGMETADO Y GESTIÓN DE RIESGOS

Arquitectura y Organización de Computadores II

Abril 2019

Sergio García Esteban 755844

Irene Fumanal Lacoma 758325

Contenido

1- Explicación breve de nuestro diseño

En este proyecto, trabajamos con el procesador MIPS de 32 bits estudiado en clase. El código VHDL proporcionado inicialmente implementa dicho procesador para que realice la escritura en el banco de registros en los flancos de bajada.

Debemos incluir tres componentes más. En primer lugar, la unidad de anticipación, que nos permitirá trabajar con los resultados de las instrucciones que escriben en el banco de registros antes de que lo hagan. En segundo lugar, la unidad de detención, para detener la pipeline los ciclos necesarios. Por último, un predictor dinámico de saltos, cuya función es predecir que ocurrirá lo mismo que en eventos pasados, en concreto, lo mismo que la vez anterior.

Estos tres elementos, nos permiten gestionar los riesgos pertenecientes a nuestro MIPS segmentado, tanto de datos como de control. En el diseño, la unidad de anticipación y el predictor de saltos, aparecen como componentes en el código, y su implementación está en el fichero fuente correspondiente. La unidad de detención no tiene su propio fichero fuente, sino que lo hemos implementado directamente en el código, ya que solo era necesario controlar señales.

2- Hardware añadido y que función lógica realiza

2.1- Introducción

El primer paso realizado para llevar a cabo este proyecto fue estudiar el código proporcionado del procesador y comprender cómo funciona. Para ello, identificamos sus componentes y ejecutamos el código de ejemplo ciclo a ciclo. Realizando esta acción nos dimos cuenta de que la instrucción de salto “beq” no realizaba el salto nunca.

Planteamos una solución al salto, tarea que nos facilitó la incorporación de una de las dos instrucciones propuestas para ser añadidas. Posteriormente, incluimos las dos instrucciones planteadas en el enunciado y resolvimos los riesgos de datos y de control que se producen en nuestro MIPS.

2.2- Instrucciones añadidas

A continuación, explicamos en qué consisten las nuevas instrucciones y como fue llevada a cabo su implementación.

1. Load address (la): carga una dirección en un registro. Puede confundirse con el load normal (lw) pero su principal diferencia es que no accede a memoria. Se expresa de la siguiente forma:
   la rt, inmed(rs):
   rt <= rs + SignExt(inmed) , PC <= PC + 4
2. Nuevo salto (bne): salta si los registros no son iguales, funciona de forma parecida al beq, pero con la condición contraria.

Cuando incorporamos instrucciones, el primer paso a completar es definir el formato de instrucción y el código de operación que le corresponde a cada una. En nuestro el código del nuevo load será ‘001000’ y el del salto ‘000101’. El formato de instrucción de ambas coincidirá con el del beq y el lw.

- 6 bits para el código de operación

- 5 bits para el registro rs y otros 5 bits para el rt

- 16 bits para codificar el inmediato

Para desarrollar la implementación de la primera instrucción comentada, modificamos la unidad de control, añadiendo su código de operación 001000, y las señales que requiere la instrucción en cada una de las etapas. Las dos salidas cuyo valor será igual a ‘1’ en la Unidad de Control serán ‘ALUSrc’ y ‘RegWrite’.

‘ALUSrc’ es la señal de control del multiplexor ‘muxALU_src’. Este multiplexor es el encargado de seleccionar el segundo operando que entra a la ALU, que en este caso será el inmediato con signo extendido. ‘RegWrite’ deberá estar a ‘1’ porque la instrucción escribe en el banco de registros la suma del inmediato y el contenido del registro. Es importante remarcar que las señales de la memoria de datos deben de estar a ‘0’, tanto la de escritura como la de lectura, ya que esta instrucción no requiere ningún tipo de acceso a memoria.

Para la nueva instrucción de salto realizamos dos pasos. El primero fue la modificación de la Unidad de Control, de manera similar al Load Address, y el

segundo, la manipulación de la señal de control PCSrc utilizada en el código principal del MIPS.

En la UC, añadimos el código de operación de la nueva instrucción y las señales que necesitará a lo largo de su ejecución. Incorporamos una nueva salida llamada 'Branch_new' cuyo objetivo es similar al de 'Branch' utilizada en el beq, es decir, indica que el código de operación que entra a la unidad pertenece al bne. En el resto de las instrucciones estará a '0' y el resto de las salidas para bne también.

PCSrc es la señal de control del multiplexor muxPC, que es el encargado de seleccionar la dirección de la siguiente instrucción que cargará en PC. Esta señal la utilizamos para las dos instrucciones de salto. En la siguiente imagen podemos observar el código añadido para su control.

```
PCSrc <= "00" when ((Z='0' AND Branch='1') OR (Z='1' AND Branch_new='1') OR
                    (Branch='0' AND Branch_new='0')) else
         "11" when ((Z='1' AND Branch='1') OR (Z='0' AND Branch_new='1')) else
         "10";
```

En este punto del proyecto las únicas entradas del multiplexor que nos interesan son '00' y '11', cuyo valor corresponde a la dirección de la siguiente instrucción en caso de que el salto no se realice (PC + 4) y en caso de que sí. La primera entrada será seleccionada siempre que la instrucción no sea de salto, cuando sea beq y el contenido de los registros no sea el mismo, o cuando sea bne y suceda lo contrario, que el contenido de los registros si que sea equivalente. La segunda entrada mencionada será seleccionada cuando la instrucción sea beq y el contenido de los registros sea igual, y bne, cuando sea distinto.

Cuando las dos instrucciones estuvieron implementadas, realizamos pruebas sencillas para verificar su funcionamiento, comentadas en el apartado de pruebas.

2.3- Riesgos de datos.

Se deben a la limitación hardware y ocurren cuando no se dispone de un dato concreto cuando se necesita. Cuando hablamos de riesgos, hablamos también de las dependencias, que son propiedades del código y que cuando estas existen, puede haber o no haber riesgos.

En nuestro MIPS la dependencia que si que nos produce un riesgo es la conocida como "productor-consumidor". El riesgo causado es el de lectura después de escritura. Se produce en dos casos: consumidor a distancia 1, es decir, la instrucción que nos sigue lee nuestro registro destino, y a distancia 2, que será la siguiente a la que nos sigue.

Para solucionar los riesgos lectura después de escritura desarrollamos la unidad de anticipación.

La unidad de anticipación nos proporcionará los datos necesarios para nuestra instrucción en la etapa de la ALU, aunque todavía no hayan sido escritos en el banco de registros. Para ello, sus salidas serán las señales de control de los multiplexores Mux_A y Mux_B situados a la entrada de la ALU, uno para seleccionar cada operando.

Para comprobar si la unidad de anticipación debe proporcionarnos un dato que no ha sido escrito todavía, tendremos en cuenta las instrucciones que están ejecutándose en la etapa de memoria y de escritura. Para cada uno de los casos que mostramos a continuación, la unidad de anticipación asignara un valor a las señales de control de los multiplexores comentados.

- Reg_Rs_EX==RW_MEM and Reg_Write_MEM==1 Mux_ctrl_A='01'
- Reg_Rt_EX==RW_MEM and Reg_Write_MEM==1 Mux_ctrl_B='01'
- Reg_Rs_EX==RW_WB and Reg_Write_WB==1 Mux_ctrl_A='10'
- Reg_Rt_EX==RW_WB and Reg_Write_WB==1 Mux_ctrl_b='10'

En el caso del lw, la unidad de anticipación no resuelve todos los riesgos de LDE. El problema del lw es que su resultado, es decir, lo que se va a escribir en el banco, lo conocemos después de leer en la memoria de datos. Es un caso especial que denominamos "load-uso" y para resolverlo debemos realizar detenciones del pipeline.

No es el único caso en el que necesitaremos detener la pipeline, en el caso del beq y del bne también debemos hacerlo, ya que son instrucciones que en nuestro MIPS segmentado se resuelven en la etapa de decodificación.

En el caso del load uso, debemos tener en cuenta la señal MemRead_EX, ya que necesitamos saber que la instrucción anterior era un load y por lo tanto lee de memoria de datos. Además, haremos las comparaciones con los registros de nuestra instrucción y el destino del load. Es importante remarcar que comprobamos que la operación no es una NOP, porque si el registro destino del load fuese r0, haría detenciones innecesarias.

En el load, bastaba con detener un ciclo la pipeline, pero en los saltos debemos hacer dos detenciones. Para ello, haremos comparaciones con las instrucciones que estén en fase de la ALU y en memoria, así cubrimos los riesgos en las dos distancias posibles. En este caso hay que comprobar que la instrucción que produce riesgo y que está delante del beq escriba en el banco de registros. Esta comprobación es importante porque si no se realiza, podría hacer detenciones innecesarias con la instrucción sw. Debemos asegurar que la instrucción tiene la codificación de uno de los dos saltos.

2.3- Riesgos de control.

En nuestro procesador los saltos se resuelven en la etapa ID y por lo tanto la penalización máxima es de 1 ciclo. Implementamos un predictor dinámico de saltos basado en eventos pasados que guardará la parte menos significativa de la dirección de la instrucción de salto, la dirección de salto y si fue o no tomado la última vez.

En la etapa IF se comprueba el predictor, si en el predictor no hay información previa, predecimos no tomado y en el registro PC se cargará la siguiente instrucción del código (PC+4). Lo mismo ocurrirá si hay información en el predictor y la última vez el salto no se tomó. En el caso de que tenga registrado que sí se tomó, el registro PC cargará la dirección a la que apunta el salto.

En la etapa ID se resuelve el salto, y tenemos que comprobar si la predicción fue correcta y solucionarlo en el caso contrario. Ha habido un error si el predictor tomó la decisión contraria o si decidió saltar pero se saltó a una dirección incorrecta.

Tras detectar el fallo se carga en PC el valor correcto y se anula la instrucción cargada cambiando su código de operación por el de una NOP. Además se actualizará el predictor con los valores obtenidos al resolver el salto.

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

3- Impacto en rendimiento de la gestión de los riesgos

La implementación de la unidad de anticipación nos ha permitido evitar paradas del pipeline en las instrucciones con dependencias productor/consumidor. Sin la UA cada vez que se detectara la dependencia, el pipeline sufriría una parada de 2 ciclos hasta que la instrucción tuviera disponible los datos en el banco de registros. Con la UA mejoramos el rendimiento ahorrando esos 2 ciclos de parada, a excepción de los LD-USO, que antes provocarían 2 ciclos de parada y ahora únicamente 1, y los saltos, los cuales se resuelven en ID y necesitan la parada de 2 ciclos de pipeline para utilizar los datos del banco de registros.

En cuanto a la gestión de riesgos de control, con respecto a saltos retardados que tendrían 1 ciclo de penalización siempre, el predictor intenta ahorrar el máximo número de fallos posibles. Esto se traduce en 1 ciclo ahorrado por cada acierto. Dependerá de cada programa y funciona muy bien en bucles, en los que solo obtendría un fallo la primera vez que salta (predice NT) y otro fallo en la última iteración (predice T).

4- Pruebas realizadas

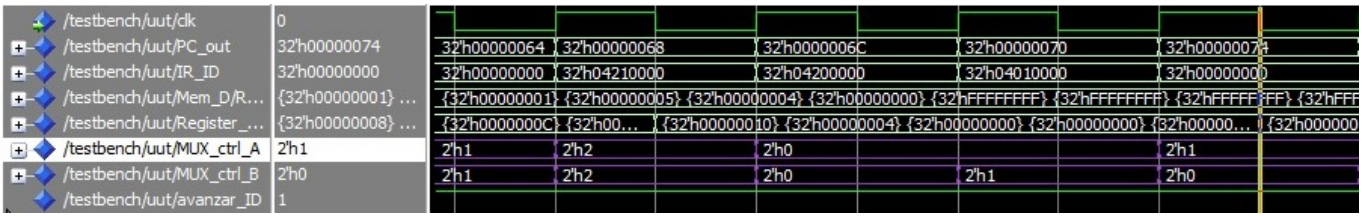
Hemos realizado 3 programas distintos para probar nuestro procesador. Cada uno está diseñado para cubrir el mayor número de casos y asegurarnos de que su funcionamiento es siempre el esperado.

El primero de ellos está diseñado para comprobar que se generan correctamente las señales que controlan los multiplexores de entrada de la ALU, controlados por la Unidad de Anticipación diseñada por nosotros.

```
Prueba 1            MUX_ctrl_A/MUX_ctrl_B (en etapa EX de la instrucción)

LA r0, 5(r0)        X
nop                 X
nop                 X
ADD r1, r0, r0      X
ADD r1, r0, r0      0/0
nop                 X
nop                 X
SW r0, 0(r0)        X
ADD r0, r0, r0      0/0
nop                 X
nop                 X
ADD r0, r0, r0      X
LW r0, 0(r1)        0/X (inmed va a la entrada b de la alu)
nop                 X
nop                 X
ADD r0, r0, r0      X
LW r1, 0(r0)        1/X
nop                 X
nop                 X
ADD r0, r1, r1      X
ADD r0, r1, r0      0/1
nop                 X
ADD r0, r0, r1      2/0
nop                 X
nop                 X
ADD r0, r1, r1      X
ADD r0, r1, r0      0/1
ADD r0, r0, r1      1/0
```

Después de ejecutar el programa en nuestro procesador, observamos que las señales generadas son las esperadas.



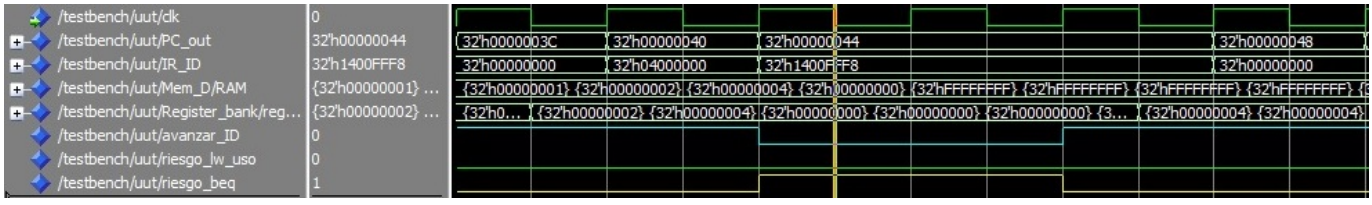
En la captura podemos ver la última prueba del código, el cursor apunta a la etapa EX de la última instrucción y observamos que funciona correctamente.

La segunda prueba está diseñada para poner a prueba la Unidad de detención que hemos desarrollado. Intenta cubrir el mayor número de casos para comprobar si el pipeline se detiene en los momentos precisos y el numero correcto de ciclos.

```
Prueba 2            Ciclos de parada del pipeline

LA r0, 5(r0)
nop
nop
LW r1, 0(r0)
ADD r0, r0, r0
nop
nop
LW r0, 0(r1)
ADD r0, r0, r0       1
nop
nop
ADD r1, r0, r0
BNE r0, r0, dir0
nop
nop
ADD r0, r0, r0
BNE r0, r0, dir0     2
nop
nop
ADD r0, r0, r0
nop
BNE r0, r0, dir0
nop
nop
LW r0, 0(r1)
nop
ADD r0, r0, r0
nop
nop
ADD r0, r0, r0
nop
nop
BNE r0, r0, dir0
```

Tras ejecutarlo en nuestro MIPS observamos que los resultados obtenidos son los esperados.



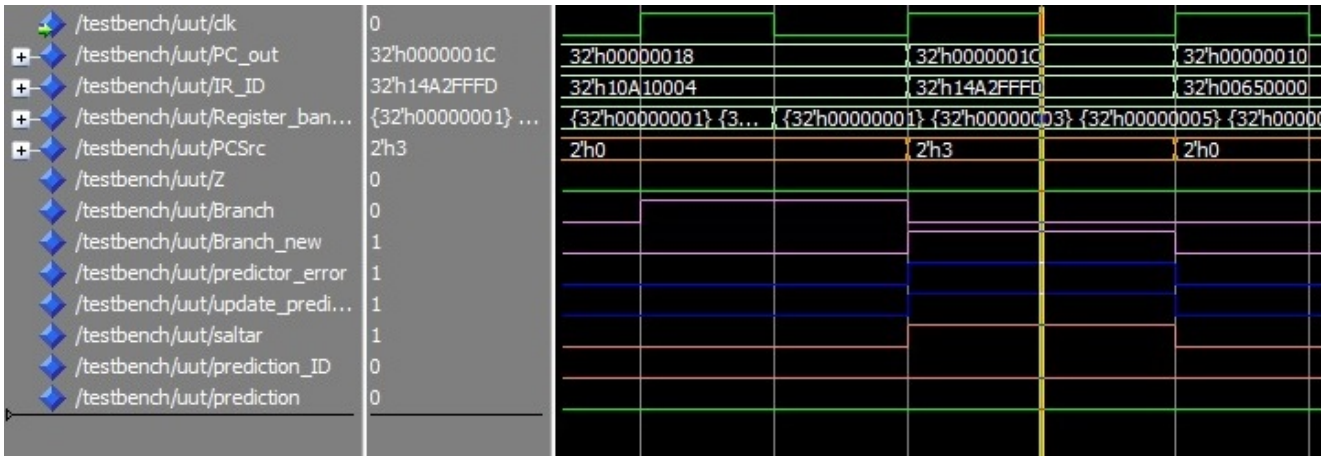
En la captura aparece el cuarto caso del código, como esperábamos, la unidad de detención se encarga de que el pipeline se detenga durante 2 ciclos.

La tercera prueba tiene el objetivo de probar a fondo el predictor de saltos, hemos diseñado un programa que incluye dos bucles anidados y una subrutina.

```
Prueba 3

LA r0, 1(r0)
LA r1, 3(r1)
LA r2, 5(r2)
ADD r4, r4, r0
ADD r5, r5, r0
BEQ r5, r1, dir10
BNE r5, r2, dir4
LA r5, 0(r3)
BNE r4, r1, dir3
BEQ r0, r0, dir13
ADD r6, r4, r5
SW r6, 64(r6)
BEQ r0, r0, dir6
BEQ r0, r0, dir13
```

Al ejecutarlo hemos comprobado que el predictor predice según lo previsto, actualiza su información correctamente en cada fallo y los resultados son los esperados.



En la captura podemos ver la primera instrucción que salta, el predictor se equivoca, actualiza sus datos, anula la instrucción cargada sustituyendo su código de operación por el de una NOP y carga en el PC la dirección de salto calculada en ID.

5- Aportaciones y cuantificación de horas

La instalación del simulador, el estudio del código proporcionado y la comprensión de su funcionamiento fue realizada conjuntamente. Esta tarea nos llevó aproximadamente 4 horas y media.

Cuando ya nos familiarizamos con los fuentes VHDL, repartimos el trabajo. La distribución de este dependió de la disponibilidad de cada ambos componentes del grupo durante el periodo vacacional de Semana Santa.

Irene, se dedicó a realizar y depurar los pasos 2 y 3. El tiempo dedicado al paso 2 fue de una hora aproximadamente. No resultó muy complicada su comprensión ya que la instrucción bne es muy similar al beq, por lo tanto, su incorporación fue bastante sencilla. El tiempo dedicado al diseño de la parte 3, fueron dos horas y media aproximadamente.

Irene, dedicó aproximadamente 5 horas para la depuración y corrección de fallos en el diseño propuesto inicialmente. Esta tarea consistió en realizar pruebas bastante sencillas para comprobar el funcionamiento correcto.

Sergio, se dedicó a implementar el predictor dinámico de saltos y realizar las pruebas finales para terminar de verificar el proyecto.

Para el paso 4, Sergio dedico 1 hora y media para el diseño. La depuración de esta parte y las 3 pruebas finales le llevaron 5 horas.

Finalmente, las horas dedicadas a la elaboración y revisión de la memoria fueron 4 horas, dos por cada uno de los integrantes.