MIPS SEGMENTADO Y GESTIÓN DE RIESGOS

Proyecto Optativo 1

REALIZADO POR:

Gregorio Largo Mayor NIA: 746621

Alonso Muñoz García NIA: 745016

a) Explicación breve del diseño realizado

Para este proyecto hemos realizado la implementación en vhdl del procesador MIPS, que posee 6 tipos de instrucción de las cuales, el load y el store poseen un auto incremento dependiendo del valor del inmediato en la instrucción. El formato de las instrucciones ha sido el mismo que el trabajo en clase en el tema 2. Este procesador además posee una unidad de anticipación de operandos, y una unidad de detención de instrucciones, de esta forma evitamos los riegos que se puedan producir en el código que generalmente solemos solventar colocando un numero de nops determinado, o reordenando el código de la forma más conveniente. Cabe destacar que con el diseño planteado se ha intentado optimizar lo máximo posible las instrucciones, sin alterar de forma muy significativa el formato del procesador que se nos entregó previamente por parte los profesores.

b) Hardware añadido y que función lógica desempeña

1) Modificaciones en la unidad de control

Las modificaciones que hemos realizado en la unidad de control han sido:

-Añadir una señal más, que hemos llamado FP_Mux, la cual es utilizada para la señal de control del multiplexor que decide si la señal de ALU_INT_out procede de la ALU o del sumador en coma flotante. Esta señal esta en todos los casos con valor '0', al requerir para esas instrucciones de la salida de ALU, y con valor '1' en la instrucción de suma en coma flotante (addFP) puesto que la salida la debemos de obtener del sumador en coma flotante.

- También hemos añadido algunas señales para la instrucción de suma con coma flotante, puesto que todas las señales estaban inicializadas a cero. Las señales que hemos puesto a uno en esta instrucción han sido las del RegDst, señal del multiplexor que elige el destino de la operación, se activa uno puesto que el destino depende de los bits 15-11 (Reg_Rd), al igual que en las operaciones aritméticas, y también hemos activado el bit de RegWrite, debido a que tras la suma se realiza una escritura en el banco de registros, al igual que en las operaciones de load y aritméticas.

2) Modificaciones en el Banco de registros EX

En el banco de registros Banco_EX, añadimos una nueva entrada y una nueva salida para el registro Rs (Reg_Rs), que es utilizado en la etapa de ejecución (EX) por la unidad de anticipación. El valor del registro RS determina el valor del bus_A, permitiendo el acceso de este a la unidad de anticipación permite ahorrar algunos ciclos en caso de ser necesario, para que otra instrucción sea capaz de utilizar el mismo registro pero actualizado. De esta forma podemos estar realizando algunas operaciones que requieren de registros que están en la etapa de memoria (Mem) o de escritura (WB) y que aún no se han actualizado en el banco de registros.

3) Modificaciones en la unidad de anticipación

En el caso de la unidad de anticipación hemos tenido que definir su procesamiento de acuerdo a la función que desempeña, puesto que esta unidad no nos la daban implementada. Por ello podemos diferenciar diferentes opciones, dependiendo de cuál sea la salida de esta. La salida de la unidad de anticipación son los bits de control de los multiplexores A y B. Estos se encargan de gestionar los valores que debería de tener el busA y el busB respectivamente. EN ambos multiplexores la entrada '0' es el estado del bus del banco de registros sin anticipar, la entrada '1' es el valor de la salida de la ALU en el ciclo anterior ya actualizado (Situado en la etapa Mem), la entrada '2' es el valor leído de memoria ya actualizado (Situado en la etapa WB), antes de que se escriba en el banco de registros. En ambos multiplexores poseen los mismos valores pero para sus respectivos buses. Por ello hemos definido diferentes casos:

Aplicado al bus_A y al registro Rs (Reg_Rs)

- En caso de que el registro Rs sea igual que el registro de destino de la etapa de ejecución (RW_Mem), y además la señal de control proporcionada por la unidad de control, RegWrite_Mem, en esa etapa posea el valor '1' entonces la salida de control del multiplexor A será '1', puesto que anticiparemos el valor de la salida de ALU_out en la etapa de memoria.
- En caso de que el registro Rs coincida con el registro destino de la etapa de escritura (RW_Mem), y la señal de control RegWrite_WB posea el valor '1', entonces la salida de control del multiplexor A será '2', puesto que anticiparemos de la etapa de escritura, el valor leído de memoria en el ciclo anterior.
- Por último, en caso de que no se produzca ninguna anticipación, la señal que establecerá la unidad de anticipación será de '0', que corresponde en este caso al bus_A leído del banco de registros en el ciclo anterior.

Lo mismo sucederá para el bus_B y el registro Rt (Reg_rt), posee exactamente las mismas comparaciones y otorga las mismas señales de control que el multiplexor A, pero en este caso a la entrada de control del multiplexor B.

4) Modificaciones en la unidad de detención

La unidad de detención principalmente se encarga de impedir que avance el MIPS cuando se están ejecutando algunas instrucciones que necesitan esperar a que terminen algunos procesos. Por ello se puede distinguir en la unidad de detención diferentes casos:

- En el caso de que se ejecute una instrucción de suma en coma flotante, esta requiere de varios ciclos en el periodo de ejecución por lo que debemos de parar de cargar datos, tanto en el banco de registros ID/EX (Banco_EX), como en el banco de registros IF/ID (Banco_ID), al parar el banco ID también paramos las cargas en el registro PC para evitar el avance de las instrucciones sin que se hayan ejecutado. Cuanto termina de ejecutarse la suma en coma flotante esta emite una señal, FP_done = '1', que significa que ya ha hecho la suma en coma flotante de manera correcta y puede volver a reanudarse el MIPS.

- En el caso de que estemos ejecutando una instrucción y la anterior a esta sea una instrucción de lectura de la memoria de datos, es decir un load, el cual guarda el valor leído en un registro que necesitamos para la instrucción a ejecutar, deberemos de pararel banco de registros ID y el PC, estableciendo una nop entre la instrucción que queremos ejecutar y el load previo a este. De esta manera permitimos que se pueda anticipar el valor en el siguiente ciclo de forma correcta. Este paso vale tanto como para el load como para la instrucción beq.

- Si estamos ejecutando una instrucción del tipo beq, y uno de los parámetros que necesita se está ejecutando deberemos de para el banco ID, deteniendo la instrucción y estableciendo tantas nops como fuese necesario. En caso de que fuese igual el registro destino que el de ejecución, con la restricción anterior nos valdría. En caso de que el registro que fuese igual estuviese en el tramo de memoria, deberíamos detener la instrucción beq un ciclo más en el banco ID y añadir una nop. En caso de que el salto se produzca de forma correcta, es decir que la señas PCSrc = '1', y que ninguno de los registros que se encuentran en las instrucciones anteriores, aun en ejecución, son parámetros del beq actual, entonces activamos la señal de Kill_IF reseteando el banco de registros ID y realizando el salto de forma correcta, a la dirección correspondiente.

- Si la instrucción, es un load, o un store y los registros requeridos por la instrucción son registros destino de las instrucciones anteriores en ejecución entonces debemos de detener el banco_ID. Debido a que tanto la instrucción load como store , necesitan leer del banco de registros para obtener los valores necesarios para las direcciones en memoria, añadiremos una nop hasta que no coincida ningún registro destino de los que están en ejecución con los utilizados por las instrucciones para evitar errores.

5) Modificaciones en el fichero de MIPS segmentado

En el MIPS segmentado hemos añadido una serie de cambios que vamos a empezar nombrando por orden desde la etapa de IF, lectura de la instrucción a ejecutar, hasta la estaba WB, de escritura.

- Hemos establecido que el load del registro PC depende de si la señal parar_ID, está activa o no, en caso de que esta señal este activa el PC no carga nada, por el contrario si esta desactivada realiza su carga de forma normal.

- El banco IF/ID, cuando la señal de Kill_ID está activa significa que se va a producir un salto, por lo que reseteamos el banco, por el contrario si esta desactiva su funcionamiento depende de la variable reset genérica. El load del banco IF/ID al igual que en el registro PC depende de la señal parar_ID, si esta activa paramos el banco si no lo está funcionamiento de forma normal.

- El load del banco ID/EX depende de la señal parar_EX, si esta activa significa que el load estará inactivo, es decir que no cargara ningún nuevo dato si no que mantendrá los que ya tiene, por el contrario si esta inactiva cargara valores de forma normal. Este caso principalmente se utiliza para la instrucción de coma flotante.
  El reset de este banco de registros, como he explicado antes depende de si queremos añadir nops o no, esperando a que alguna de las instrucciones anteriores se termine de

ejecutar puesto que su registro destino lo utilizamos. Esto sucede para el caso de la instrucción que tiene un load delante y que le tiene que dar un ciclo de ventaja, para la instrucción de tipo Beq, y para los loads y stores que deban esperar en la etapa ID.

- Establecemos un Mux pequeño a la entrada B de la ALU cuya señal de control es ALUSrc_EX, que solo está activa cuando es un load o un store por si la dirección lleva un incremento en el inmediato para la dirección en memoria, que debemos de leer o escribir. Cuando ALUSrc_EX vale '0', es decir no es un load o store significa que deja pasar la salida del Mux_B, por el contrario si vale '1' dejara pasar el inmediato extendido para sumarlo en la ALU, con el valor correspondiente obtenido de la dirección marcada en el banco de registros por el registro Rs. Los dos Multiplexores cuyas señales de control determina la unidad de anticipación, y cuyas entrada están explicada en el apartado de la unidad de anticipación, los situaremos delante de la ALU y del sumador de coma Flotante, puesto que serán sus entradas, en caso del MUX_B para la ALU dependerá del registro ALUSrc explicado antes.

- Para seleccionar si el valor que queremos como ALU_Out_Mem es el dado por la ALU o por el sumador de coma Flotante, ADDFP, establecemos un multiplexor FP_Mux, cuya señal de control si vale '0' deja pasar el valor de la ALU y por el contrario si vale '1' deja pasar el del sumador de coma flotante.

- El reset del banco EX/Mem depende de si se para el banco ID/EX o no, este solo se para como bien he dicho, en la instrucción de suma de coma flotante. Al estar parado debemos de añadir nops en el banco EX/Mem , tantas como ciclos necesite el sumador de coma flotante., por cada nop que debemos de establecer se establece un ciclo la señal de reset a uno.

c) Impacto en rendimiento de cada una de las modificaciones con respecto al procesador inicial

Cabe destacar con respecto al MIPS inicial, que no poseía ni unidad de anticipación, ni unidad de detención de riesgos, ni tenía la opción de incrementar los registros de dirección para el load y el store sin necesidad de hacer una operación aritmética previamente.

La unidad de anticipación nos permite ahorrar mucho tiempo, puesto que reduce de forma notable los ciclos, al reducir considerablemente las nops necesarias para almacenar los datos, al anticipar los datos a la etapa de ejecución en caso de ser necesarios

La unidad de detención, no reduce los ciclos pero nos permite establece un código con un menor número de instrucciones con respecto al procesador anterior, al no ser necesarias el establecimiento de la instrucción nop. Esta unidad directamente es capaz de determinar si es necesario añadir nops entre instrucciones, si es necesario esperar más o menos al ejecutar una instrucción, de forma automática sin que el usuario se vea obligado a establecer las nops en el código de programa.

El incremento del load y el store por medio del inmediato, a través de la ALU, nos permite ahorrar instrucción de incrementar el registro que utilizamos como puntero para obtener los datos de la memoria de datos. Gracias a esto por medio de cualquier registro podemos acceder de forma directa a una dirección en concreto de la memoria de datos con una instrucción de lectura tan solo, evitando operaciones aritméticas innecesarias y que aumentan el número de instrucciones del programa.

d) Pruebas realizadas para verificar que funciona correctamente

Los programas que hemos utilizado para los test incluyen un gran número de anticipaciones, detección de posibles riesgos y en ellos reflejamos todas las instrucciones que puede tener el procesador MIPS.
Algunos programas utilizan el inmediato para los load y store y otros no, esto lo hemos hecho para que se vea como un programa que no utiliza estos accesos para reducir instrucciones se puede hacer considerablemente largo con respecto al código.
Las pruebas las hemos colocado, en orden descendiente haciendo referencia cada una a un paso correspondiente del proyecto, aunque todas tengan anticipación y detención de riesgos. Cada programa presenta una pequeña de parte de datos, que son los datos que hemos añadido en la versión correspondiente a la memoria de datos, y una parte de código donde se presentan las instrucciones del código.
Las traducciones a hexadecimal, se pueden ver mejor en el fichero de notas adjuntos con el código del procesador.

Test_FP_V1.txt

```
.data
@0x0            4
@0x4            3e4ccccd
.code
000010 00000  00001  0000000000000000        @0x0   LW  R1, (R0)          R1 = 4
000010 00001  00000  0000000000000000        @0x4   LW  R0, (R1)          R0 = 3e4ccccd = 0.2
000001 00001  00001  00001 00000000000       @0x8   ADD R1, R1, r1        R1 = 8
100000 00000  00000  00010 00000000000       @0xC   ADDFP R2, R0, R0      R2 = 0.4
000011 00001  00010  0000000000000000        @0x10  SW  R2, (R1)          (R1) = @0x8 =
posicion2
```

Test_FP_V2.txt

```
.data
@0x0            4
@0x4            3e4ccccd
.code
000010 00001  00000   0000000000000000          @0x0   LW  R0, 4(R1)          R0 = 3e4ccccd = 0.2
100000 00000  00000   00010 00000000000          @0x4   ADDFP R2, R0, R0       R2 = 0.4
100000 00010  00010   00011 00000000000          @0x8   ADDFP R3, R2, R2       R3 = 0.8
000011 00001  00011   0000000000000000          @0xC   SW  R3, 8(R1)          (R1) = @0x8= posicion2
```

Test_ant_V1.txt

```
.data
@0      1       00000004
@1      5       00000005
@2      3       00000003

.code
000010 00000 00001 0000000000000000      @0x0   LW  R1, (R0)          R1 = 4
000010 00001 00010 0000000000000000      @0x4   LW  R2, (R1)          R2 = 5
000001 00001 00001 00000 00000000000     @0x8   ADD R0,R1,R1          R0 = 4 + 4 = 8
000010 00000 00011 0000000000000000      @0xC   LW  R3, (R0)          R3 = 3
000001 00011 00010 00100 00000000 001    @0x10  SUB R4, R3, R2        R4 = 3 - 5 = 2
000001 00000 00001 00000 00000000000     @0x14  ADD R0,R0,R1          R0 = 8 + 4 = 12
000011 00000 00100 0000000000000000      @0x18  SW  R4, (R0)          (R0) = @0xC = 2
```

Test_ant_V2.txt

```
.data
@0      1       00000004
@1              12300789
@2              FFFFFFFF

.code
000010 00001 00010 0000000000000100     @0x4    LW  R2, 4(R1)       R2 = 12300789
000010 00000 00011 0000000000000000     @0x8    LW  R3, 8(R0)       R3 = FFFFFFFF
000001 00011 00010 00100 00000000 010   @0xC    AND R4, R3, R2      R4 = 12300789 AND FFFFFFFF =
                                                                        12300789
000011 00000 00100 0000000000001100     @0x10   SW  R4, 12(R7)      (R0) = 0xC
```

Test_ant_V3.txt

```
.data
@0      1       00000004
@1      4       00000005
@2              3e4ccccd
.code
000010 00000 00001 0000000000000000     @0x0    LW  R1, (R0)        R1 = 4
000010 00001 00010 0000000000000000     @0x4    LW  R2, (R1)        R2 = 5
000001 00001 00001 00000 00000000000    @0x8    ADD R0,R1,R1        R0 = 4 + 4 = 8
000001 00010 00001 00011 00000000 001   @0xC    SUB R3,R2,R1        R3 = 5 - 4 = 1
000001 00011 00010 00100 00000000000    @0x10   ADD R4,R3,R2        R4 = 1 + 5 = 6
000010 00000  00101 0000000000000000    @0x14   LW  R5, (R0)        R2 = 0.2
100000 00101 00101 00110 00000000000    @0x18   ADDFP R6,R5,R5              R6 = 0.2 + 0.2 = 0.4
000001 00000  00001 00000 00000000000   @0x1C   ADD R0,R0,R1        R0 = 8 + 4 = 12
000011 00000 00110 0000000000000000     @0x20   SW  R6, (R0)        (R0) = 0xC
```

Test_beq_V1.txt

```
.data
@0              00000004
@1              00000008
.code
000010 00000 00001 0000000000000000          @0x0   LW  R1, (R0)          R1 = 4
000010 00001 00010 0000000000000000          @0x4   LW  R2, (R1)          R2 = 8
000001  00001   00001   00000 00000000000    @0x8   ADD R0,R1,R1          R0 = 4 + 4 = 8
000001 00011   00001   00011 00000000000  buc @0xC   ADD R3,R3,R1          R3 = R3 + 4
000100  00011   00010   0000000000000001    @0x10  BEQ R3, R2 #+1=sal    R3 = R2 salto al ADD
000100  00001   00001   1111111111111101    @0x14  BEQ R1,R1, #-3=buc    R1 = R1 salto a add R3
000001  00000   00000   00001 00000000000  sal @0x18  ADD R1,R0,R0          R0 = 8 + 8 = 16
000011  00111  00001   0000000001000000      @0x1C  SW  R1, 64(R7)         (R7) = 0 -> @64
```

Test_beq_V1.txt

```
.data
@0              00000004
.code
000010 00000   00001   0000000000000000      @0x0   LW  R1, (R0)          R1 = 4
000100 00001   00001   0000000000000010      @0x4   BEQ R1,R1 #+2=sal     ;Salto a sal
000001  00001   00001   00001 00000000000    @0x8   ADD R1,R1,R1          R1 = 4 + 4 = 8
000001  00001   00001   00001 00000000000    @0x10  ADD R1,R1,R1          R1 = 8 + 8 = 16
000001  00001   00001   00001 00000000000  sal @0x14  ADD R1,R1,R1          R1 = 4 + 4 = 8  O R1
000011  00001  00001   0000000000000000      @0x18  SW  R1, (R1)          (R1) = @0x8 = 8
```

e) Conclusiones y cuantificación de horas dedicadas por cada miembro.

Como se ha comentado anteriormente al principio de la memoria, lo que se ha buscado en este proyecto ha sido intentar mantener, la base del código lo más similar a la dada por los profesores. Todas las aportaciones por nuestra parte lo que han intentado ha sido aportar fluidez tanto en el procesamiento del código como en la reducción de ciclos. Esto se ha visto reflejado principalmente por la unidad de anticipación la que ha supuesto una reducción de ciclos, y la unidad de detección que otorga una reducción de instrucciones al no tener que tener ya en cuenta las nops.

Los principales problemas que han podido surgir han sido en ocasiones por un mal nombramiento de registros como sucedió cuando nos faltó el registro Rs que nos generó problemas, y también algunos pequeños errores que debimos pulir durante la depuración y ajustes del código.

Con respecto a la cuantificación de horas dedicadas al proyecto y por miembro, podemos decir:

- A la parte de toma de contacto con el simulador, y dedicación al primer paso del proyecto, la realización del esquema, ronda entorno a unas diez horas, en las cuales realizamos el esquema para facilitarnos de forma considerable el programar los cambios, y observamos y entendimos el código que nos entregaron los profesores.
- Al diseño del apartado dos le dedicaríamos en torno a 1 hora u hora y media, mientras pensamos de forma detenida los cambios que había que hacer y los posibles problemas que podían surgir.
- Al diseño del paso de tres la unidad de anticipación le dedicamos en torno a 1 hora u hora y media también puesto que no nos supuso un problema el ver de forma clara lo que debía de hacer. Como bien he comentado antes el problema que tuvimos fue ver la ausencia del registro Rs en el banco_EX que nos retrasó mínimamente.
- El paso tres fue algo mas costoso puesto que los diferentes casos debían de ser muy restrictivos para que no generasen problemas, la dedicación a esta parte fue de unas 2 horas.
- La depuración y ajustes costo entorno a unas 12 horas, contando la realización de numerosas pruebas, la creación de ficheros de prueba y los diferentes test. LA realizamos de forma conjunta en su mayoría.
- La memoria nos costó entorno a las tres horas debido a que cada vez que realizábamos un cambio o una modificación la anotábamos en un papel para ver las diferentes modificaciones realizadas, y facilitarnos así de forma muy significativa la realización de la memoria.