Autores: Abel Chils Trabanco (NIA: 718997) y Angel Cañal Muniesa (NIA: 716205).

En este trabajo se ha creado e implementado un autómata de tipo Mealy el cual funcionará como controlador de la memoria caché, así como también se ha modificado el procesador inicial para que se adaptase a la nueva interfaz al comunicarse con la memoria de datos y se han añadido 3 contadores de los diferentes tipos de paradas.

Descripción del diseño.

Modificación del procesador original

Para lograr que el procesador inicial siguiera funcionando correctamente al añadirle el módulo MD_mas_MC se ha tenido que modificar la unidad de detección de riesgos para hacer que en el caso de que la instrucción que esté en la etapa de memoria sea un load, save o sus variantes con postincremento, parase el procesador mientras la interfaz de la memoria no devuelva ready. Parar el procesador implica evitar que las instrucciones anteriores avancen y evitar que el pc aumente. En cambio, la última etapa, la que se encuentra en write back durante todos los ciclos que dure la parada se ejecutará, esto no afecta a la consistencia de los registros debido a que en el caso de que escriba la hará siempre sobre el mismo registro.

Para implementar los contadores se han utilizado por un lado las dos señales que salen desde el módulo de detección de riesgos y mandan a parar al procesado (stop por riesgo de datos y stop por memoria). Y por otro lado se ha usado la señal que implica un salto tomado.

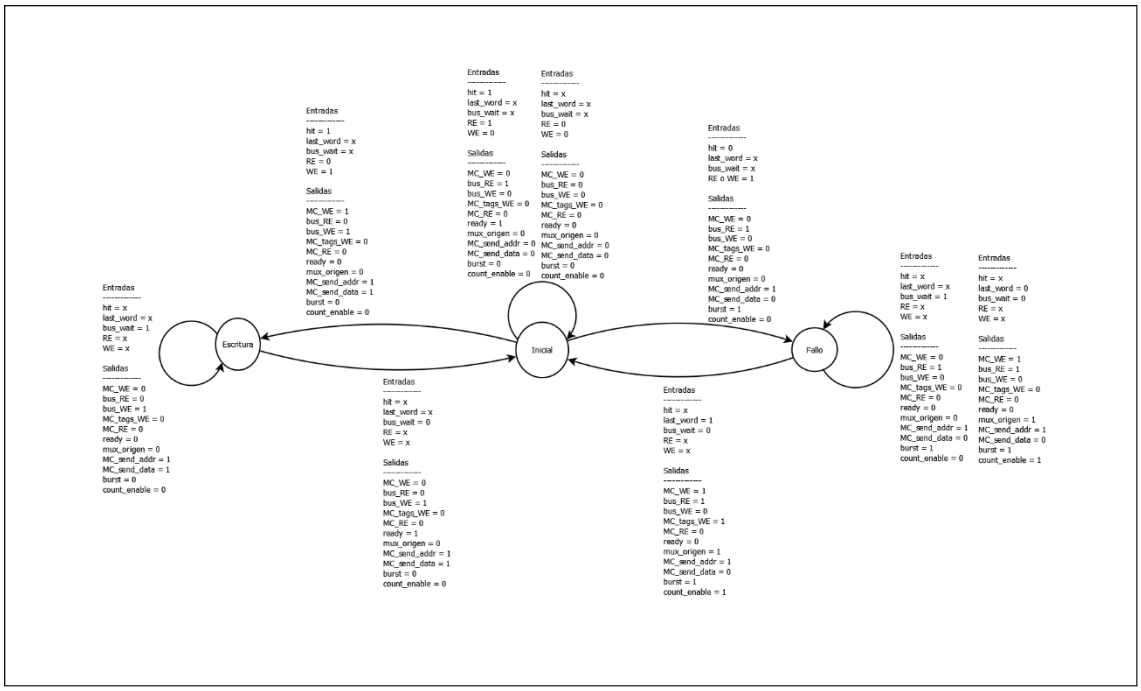
Cada señal aumentará un contador, pero solo lo hará uno a la vez, siendo el de memoria el que tiene preferencia sobre los otros dos, debido a que al ser la instrucción que está en memoria la que más adelantada está sería la causante de bloquear el avance del resto.

Esto por ejemplo evitaría que en situaciones como por ejemplo las siguiente secuencia de instrucciones lw r5,0(r0) - lw r4,0(r0) - add r4,r4, r4 , el contador cuente como riesgo de datos los ciclos que se está parado por el primer lw.

En el caso de un riesgo estructural, el cual solo se produciría con los saltos en el caso de tomarlo, en el momento en el que hay un salto tomado se aumenta en una unidad el contador, ya que un salto tomado detiene el procesador durante un ciclo.

Autómata controlador de la caché

Para el autómata que controla la caché se decidió crearlo de tipo Mealy en 3 estados, siendo este el siguiente (en los archivos entregados se incluye la misma imagen para que pueda ser ampliada):



Descripción del funcionamiento:

Se empieza desde el estado “Inicial”, en el caso de que no se produzca ni read ni write se quedará en el mismo estado.

En el caso de un acierto en lectura (hit) en el mismo ciclo se accede a la cache de manera combinacional y se devuelve el dato dando la señal ready al procesador, manteniéndose el autómata en el mismo estado.

En el caso de acierto en escritura, se escribe el dato en la cache y se ordena a memoria que lo guarde en el primer ciclo, y se mueve a un estado (Escritura) en el que se espera hasta que la memoria confirme que ha guardado el dato. En ese momento le se le da la señal ready al procesador y se vuelve al estado inicial.

Estando en el estado inicial, en el momento en el que hay un fallo ya sea de lectura o escritura se mueve al estado “Fallo” en el cual realiza la secuencia de fallo que consiste en traer el bloque desde memoria principal y en el momento en que llegue la última palabra se mueve a “Inicial” además de escribir el tag. En ese momento, la instrucción que antes falló, acertará en caché, por lo que a partir de ahí se trata con la misma secuencia que se realiza en los aciertos.

Pruebas de funcionamiento

Las pruebas realizadas se pueden agrupar en dos conjuntos, las primeras son las realizadas sobre el módulo mp_mc (archivos md_cache_1.vhd y md_cache_2.vhd) en las cuales se prueban todas las combinaciones de dos instrucciones que fallen o acierten ya sean de lectura o escritura.

En el propio archivo antes de cada señal se indica que producirá (acierto o fallo en lectura o escritura).

En estos dos bancos de pruebas también se prueban a realizar múltiples reemplazos, así como la utilización de todos los sets.

Luego a la hora de testear el procesador completo primero se pasaron las pruebas que se le pasaron al procesador anterior, para comprobar que todo seguía funcionando correctamente y que las nuevas modificaciones también funcionaban.

Descripción de estas pruebas (igual a la que incluimos en la anterior práctica):

Las 5 primeras pruebas se han pasado con la memoria ram llamada memoria_datos_1_2_3_4_5.vhd

Prueba nº1

sw_pos r1, 1(r1)

sw_pos r1, 1(r1)

add r1, r1, r1

add r1, r1, r1

add r1, r1, r1

lw r1, 1(r1)

lw_pos r0, 1(r1)

lw_pos r0, 1(r1)

lw r1, 0(r1)

lw r1, 0(r0)

lw_pos r0, 1(r1)

lw r1, 0(r0)

sw_pos r0, 1(r1)

sw_pos r0, 1(r1)

lw r0, 0(r1)

Prueba nº2

```
lw r1, 0(r0)
beq r1, r0, 0
beq r1, r0, 0
lw_pos r1, 4(r0)
beq r0, r1, 0
beq r0, r1, 0
```

Prueba nº3

```
lw r1, 0(r0)
nop
sw r1, 8(r0)
lw r1, 4(r0)
sw r1, 4(r1)
lw_pos r2, 1(r3)
nop
sw r2, 0(r3)
lw_pos r2, 1(r3)
sw r2, 0(r3)
```

Prueba nº4

```
add r1, r1, r1
nop
lw_pos r0, 1(r1)
add r1, r1, r1
lw_pos r0, 2(r1)
lw r0, 4(r3)
lw_pos r3, 4(r0)
sw_pos r4, 3(r4)
nop
lw_pos r3, 3(r4)
```

```
   sw_pos r0, 2(r1)
   lw_pos r0, 3(r1)
```

Prueba nº5

```
   lw r0, 0(r1)
   sw_pos r0, 3(r0)
   sw_pos r0, 3(r0)
   nop
   sw_pos r0, 3(r0)
   lw_pos r0, 2(r1)
   nop
   sw_pos r0, 2(r3)
   lw_pos r0, 2(r1)
   nop
   sw_pos r0, 3(r0)
```

Prueba nº6

```
   lw r1, 0(r0)
   lw r3, 4(r4)
b:  add r2, r2, r1
   add r0, r0, r3
   beq r0, r1, f
   beq r0, r0, b
f:   sw r2, 0(r1)
```

Almacena en una dirección leída de la posición 0x0 de la RAM el cuadrado de esa dirección por 4.

Prueba nº7

```
   lw_pos r1, r(r0)
   lw_pos r3, 4(r0)
b:  lw r4, 0(r0)
   add r2, r2, r4
   add r0, r0, r3
   beq r0, r1, f
   beq r0, r0, b
f:   sw r2, 0(r1)
```

Calcula la suma del vector cuya primera componente se encuentra en la posición 0x8 de memoria,

y ocupa tantos bytes como se lean de la posición 0x0 de la RAM. Esta suma la guarda en la palabra

que se encuentra a continuación del vector.

Prueba nº8

```
   lw_pos r1, 4(r0)
b:  lw_pos r3, 4(r0)
   add r2, r2, r3
   beq r0, r1, f
   beq r0, r0, b
f:   sw_pos r2, 0(r0)
```

Realiza lo mismo que el código anterior, pero aprovechando las instrucciones con postincremento.

Prueba nº9

```
   lw_pos r1, 4(r0)
   lw_pos r2, 4(r0)
   lw_pos r3, 4(r0)
```

```
   add r4, r2, r3
b:  lw_pos r5, 4(r1)
   sw_pos r5, 4(r2)
   beq r2, r4, f
   beq r2, r2, b
f:   sw r2, 0(r0)
```

Copia un vector v (en la posición 0x0 de la RAM hay un puntero a dicho vector) a un segundo vector w (su puntero se encuentra en la segunda palabra de la RAM).

Después se realizaron pruebas específicas que hacían un uso intensivo de la cache. El código de estas últimas pruebas está en ensamblador (p1.asm) junto a los bancos de pruebas (p1.vhd) y 3 RAMs (ram_p1_0.vhd, ram_p1_1.vhd y ram_p1_2.vhd) distintas que prueban un amplio espectro de direcciones, con fallos, aciertos y reemplazos tanto en lectura como en escritura.

Prueba específica general (accesos aleatorios a la memoria)

Código prueba 1:

```
lw2 r1, 4(r0)
lw2 r2, 4(r0)
lw2 r3, 4(r0)
lw2 r4, 4(r0)
lw2 r5, 4(r0)
lw2 r6, 4(r0)
lw2 r7, 4(r0)
lw2 r8, 0(r0)

#Guardar todas las direcciones leidas en esa direccion más 4.
sw r1, 4(r1)
sw r2, 4(r2)
sw r3, 4(r3)
sw r4, 4(r4)
```

```
sw r5, 4(r5)
sw r6, 4(r6)
sw r7, 4(r7)
sw r8, 4(r8)

#Leer y guardar datos con accesos aleatorios.
sw r1, 0(r1)
sw r2, 0(r2)
lw r3, 0(r3)
lw r4, 0(r4)
lw r5, 0(r5)
sw r6, 0(r6)
lw r7, 0(r7)
sw r8, 0(r8)
```

Datos prueba 1-a:

```
000000f0
00000130
000000d0
000001d0
00000160
000000b0
00000080
000001a0
```

Datos prueba 1-b:

```
000001e0
00000100
00000140
00000120
000000c0
```

000001c0

00000150

00000180

Datos prueba 1-c:

00000190

00000170

000000e0

00000090

00000110

000000a0

000001b0

00000180

Reparto del trabajo:

Aclaración: Durante el desarrollo del proyecto primero desarrollamos una versión inicial que incluía la sección "Avanzada", si bien el autómata era correcto al llevarlo a vhdl se debió de cometer algún error que producía un comportamiento incorrecto al pasar las pruebas. Y viendo que no quedaba mucho tiempo se decidió comenzar de nuevo reaprovechando lo que se pudiese con una versión que no incluyese la sección avanzada. En las horas dedicadas se diferencia el tiempo dedicado a esa versión inicial y a la versión que finalmente se entregó.

Horas Abel:

Desarrollo versión inicial:

Estudio del proyecto: hora y media

Autómata: hora y media

Implementación: 1 hora

Creación bancos de pruebas: 40 minutos

Testeo y corrección: 17 horas

Total: Aproximadamente 21 horas y media

Desarrollo versión entregada:

Autómata: 10 minutos

Implementación: 30 minutos

Testeo y corrección: 5 horas.

Memoria: 3 horas.

Total: Aproximadamente 8 horas y media

Comparación con las horas Abel con estimadas:

Estudio de las fuentes: Mis horas dedicadas son bastante similares a las que se estiman.

Autómata e implementación de este: Al desarrollar la versión inicial dediqué bastante tiempo ya que no estaba familiarizado con las señales que se utilizan. A la hora de realizar el de la versión que se entregó fue muy rápido ya que ya sabía que es lo que tenía que hacer el autómata y estaba muy familiarizado con todas las señales.

Testeo y corrección: En la versión inicial dediqué mucho tiempo a esta sección ya que fallaba en situaciones en las que no se esperaría que fallará y dediqué mucho tiempo siguiendo las señales para ver que producía los fallos y realizando sucesivas correcciones. En la versión que se entregó los fallos que se tenían eran fáciles de solucionar.

Memoria: Mis horas dedicadas son similares a las estimadas.

Horas Ángel:

Desarrollo autómata inicial: 3 horas

Puesta en común y mezcla de autómatas: 30 minutos

Implementación estados escritura en autómata y escritura en DIA: 1 hora

Escritura código para probar el proyecto: 3 horas

Desarrollo de dos programas Java (un "compilador" y un traductor de @): 3 hora

Entender el código nuevo (para depurar correctamente el primer código): 2 horas

Depuración del proyecto con los códigos del proyecto anterior y los nuevos: 7 horas

Total: Aproximadamente 14 horas y media

Primer intento de depuración del primer programa y solución de errores: 5 horas

Reescritura del primer programa y sus datos por múltiples fallos: 1,5 horas

Pruebas de los códigos del primer proyecto: 3 horas

Depuración del proyecto 2 con el programa correcto y los datos sin errores: 2 horas

Total: 24 horas

Comparación de horas Ángel con estimadas:

Estudio de fuentes: No se ha dedicado un tiempo excesivo a tal tarea.

Autómata e implementación de este: El desarrollo ha sido ágil, tanto las primeras versiones como las finales.

Testeo y corrección: Se ha dedicado un tiempo similar al que se presupone que se debía dedicar, y el tiempo extra viene de los fallos que han supuesto reescribir y la detección de los mismos