Arquitectura de Computadoras

Informe Laboratorio ARMv8 en SV

Ejercicio 1: Integración y Test Básico

Se realizó la integración del procesador con Pipeline a los módulos definidos en el Ejercicio 1 del Práctico 1, luego se probó el Test Básico del Práctico 1 (el cargado en el módulo **imem**).

Como el procesador no tiene **Hazard Control**, el resultado obtenido no es el esperado. Esto se debe a que el programa intenta leer de los registros, datos que en la memoria todavía no se terminaron de escribir. // TODO: esta frase está mal formulada

Para obtener el resultado esperado, se realizaron los siguientes cambios al código original:

Código Original:

```
0       # Carga de registros en memoria
1               stur x0,[x0]
2               stur x1,[x0,#8]
3               stur x2,[x0,#16]
4               stur x3,[x0,#24]
5               stur x4,[x0,#32]
6               stur x5,[x0,#40]
7               stur x6,[x0,#48]
8
9       # Lectura de memoria en los registros
10              ldur x7,[x0]
11              ldur x8,[x0,#8]
12              ldur x9,[x0,#16]
13              ldur x10,[x0,#24]
14              ldur x11,[x0,#32]
15              ldur x12,[x0,#40]
16              ldur x13,[x0,#48]
17
18      # Sumatoria de los registros
19              sub x14,x14,x14
20              cbz x14, Loop
21              sub x15,x0,x1
22      Loop:
23              add x15,x0,x1
24
25      # Carga del resultado de la sumatoria en memoria
26              stur x15, [x0,#56]
```

Resultado Original (primeros 7 registros del mem dump)**:**

```
0 0000000000000000
1 0000000000000001
2 0000000000000002
3 0000000000000003
4 0000000000000004
5 0000000000000005
6 0000000000000006
7 000000000000000F
8 0000000000000000
```

Código modificado:

```
0       # Carga de registros en memoria
1               stur x0,[x0]
2               stur x1,[x0,#8]
3               stur x2,[x0,#16]
4               stur x3,[x0,#24]
5               stur x4,[x0,#32]
6               stur x5,[x0,#40]
7               stur x6,[x0,#48]
8
9       # Lectura de memoria en los registros
10              ldur x7,[x0]
11              ldur x8,[x0,#8]
12              ldur x9,[x0,#16]
13              ldur x10,[x0,#24]
14              ldur x11,[x0,#32]
15              ldur x12,[x0,#40]
16              ldur x13,[x0,#48]
17
18      # Sumatoria de los registros
19              sub x14,x14,x14
20              cbz x14, Loop
21              sub x15,x0,x1
22      Loop:
23              add x15,x0,x1
24      # Instrucciones NOP
25              nop
26              nop
27      # Carga del resultado de la sumatoria en memoria
28              stur x15, [x0,#56]
```

Resultado:

```
0 0000000000000000
1 0000000000000001
2 0000000000000002
3 0000000000000003
4 0000000000000004
5 0000000000000005
6 0000000000000006
7 0000000000000001
8 0000000000000000
```

Análisis del código nuevo:

Como podemos ver en las evidencias anteriores, el resultado obtenido en memoria no era el esperado. En la posición de memoria 7 vemos que se cargó el valor 0xF, cuando esperábamos 0x1.

Esto se debe a que la instrucción stur x15, [x0,#56], depende de la instrucción add x15,x0,x1, como los registros están inicializados cada uno con su índice (en este caso el X15 tiene el valor 0xF), cuando hace el STUR, todavía no se guardó el resultado del ADD y usa el valor viejo del registro (0xF en vez de 0x1).

Al agregar instrucciones NOP entre estas instrucciones, logramos que el resultado en memoria sea el esperado. En particular debemos agregar únicamente 2 NOP, ya que modificamos el módulo **regfile** para que se pueda leer un registro antes que haya sido efectivamente escrito, y esto le permite a la etapa decode, usar el valor que la instrucción anterior está escribiendo.

Por otro lado, no hizo falta agregar NOPS entre la Carga de registros en Memoria y la Lectura de estos, ya que no ocurrían Data Hazards.

Aclaración:
La modificación en el código se realizó para tener el mismo resultado en memoria, pero el estado de los registros no se mantiene como en el micro sin Pipeline, para lograr esto hay que agregar nops entre la carga de los registros en memoria y la lectura de esta memoria en los registros.

Análisis cantidad de ciclos de CLK:

Antes de comenzar el análisis, es importante aclarar que la cantidad de ciclos de clock del **TestBench** al momento de correr el código era de 80 ciclos. Cada ciclo de clock de 10ps, el TestBench especificaba 20ps en reset, 800ps de ejecución y 20ps de Memory Dump.

Para realizar el análisis fuimos modificando el valor de los *ps* de ejecución para ver si había algún cambio.

Vimos que el código original tiene 19 instrucciones, el código modificado agrega dos instrucciones más (dos NOPS), luego tiene 21 instrucciones. Pero una de las instrucciones no se ejecuta, ya que se hace un BRANCH que la evita.

Concluimos que necesitamos en total 24 ciclos de clock (5 para la primera instrucción y 19 que se van sumando con las instrucciones siguientes) en el micro con Pipeline. Es decir que necesitamos por lo menos *24 * 10ps = 240ps* de ejecución para terminar correctamente el programa.

Por otro lado, si usamos Single Cycle Processor, no podemos paralelizar la ejecución de las instrucciones, pero no necesitamos agregar las instrucciones NOP, luego tenemos 18 instrucciones (no contamos la sub x15,x0,x1, que no se ejecuta debido al branch).
En total necesitamos 18 ciclos de clock, porque cada instrucción se corre en un único ciclo. Por lo tanto necesitamos por lo menos 18 * *10ps = 180ps* para terminar correctamente la ejecución del programa.

Ejercicio 2

Antes de comenzar el análisis del funcionamiento del programa, tuvimos que modificar el código, ya que no cumplía con la ISA, porque usaba el registro XZR para el STUR.
Realizamos el siguiente cambio para que la compilación sea exitosa:

```
0       SUB X23, X1, X1
1       ADD X0, XZR, X4
2       ADD X1, X0, X4
3       ADD X2, X1, X4
4       ADD X3, X2, X4
5       STUR X0, [X23, #0]
6       STUR X1, [X23, #8]
7       STUR X2, [X23, #16]
8       STUR X3, [X23, #24]
```

El resultado esperado de la ejecución del código es el siguiente y fue verificado con la ejecución del programa en el micro sin Pipeline:

```
0 0000000000000004
1 0000000000000008
2 0000000000000010
3 000000000000000C
```

El código realiza lo siguiente:

- Guarda el valor de X4 en X0, como cada registro se inicializa con su índice, luego X0 = 4.
- Suma el valor de X4 con X0 y lo guarda en el registro X1, luego X1 = 8.
- Suma el valor de X4 con X1 y lo guarda en el registro X2, luego X2 = 12.
- Suma el valor de X4 con X2 y lo guarda en el registro X3, luego X3 = 16.
- Finalmente guarda los valores de los registros desde el X0 hasta el X3 en las primeras 4 direcciones de memoria del bloque de memoria 0.

El resultado esperado en un micro con Pipeline, pero sin control de Hazards es el siguiente:

```
0 0000000000000004
1 0000000000000004
2 0000000000000005
3 0000000000000006
```