Arquitectura de Computadoras

Informe Laboratorio ARMv8 en SV

Integrantes

- Andruskiewitsch, Igor
- Lucero, Rodrigo
- Orellana, Leonardo

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 pasa porque las instrucciónes estan compartiendo un mismo registro, entonces cuando el procesador ejecuta la primera instrucción, la siguiente instrucción tendría que esperar a que la primera escriba, pero como no espera, entonces toma el valor viejo del registro.

Se realizaron los siguientes cambios :

Código original :

```
#Cargar 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]

# Lectura de memoria en los registros
8  -  ldur x7,[x0]
9  -  ldur x8,[x0,#8]
10 -  ldur x9,[x0,#16]
11 -  ldur x10,[x0,#24]
12 -  ldur x11,[x0,#32]
13 -  ldur x12,[x0,#40]
14 -  ldur x13,[x0,#48]

# Sumatoria de los registros

15 -  sub x14,x14,x14
16 -  cbz x14, Loop
17 -  sub x15,x0,x1
18 -  Loop: add x15,x0,x1

# Carga del resultado de la sumatoria en memoria
19 -  stur x15, [x0,#56]
```

Resultado Original (primeros 7 registros del mem dump):

```
0000000000000000
0000000000000001
```

```
0000000000000002
0000000000000003
0000000000000004
0000000000000005
0000000000000006
000000000000000F
0000000000000000
```

Lo que se puede apreciar en el código original, es que exiten dependencia de datos, en la cual generan data hazard y esto pasa en las siguientes instrucciones :

Hazar de Control (HC) : X14 15(MEM) 16(EXE)

```
15 - SUB x14,x14,x14
16-  CBZ X14, Loop
```

Hazar de datos (HD) :

```
18 - ADD X15, X0, X1
19 - STUR X15, [X0, #56]
```

Hazar de datos (HD) :x15 18(WB) 19(MEM)

Entonces, para poder arreglar este problema, se agrega los NOPS necesarios para poder arreglar estas dependencias:

Código modificado:

```
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]

# Lectura de memoria en los registros
8  -  ldur x7,[x0]
9  -  ldur x8,[x0,#8]
10 -  ldur x9,[x0,#16]
11 -  ldur x10,[x0,#24]
12 -  ldur x11,[x0,#32]
13 -  ldur x12,[x0,#40]
14 -  ldur x13,[x0,#48]

# Sumatoria de los registros

15 -  sub x14,x14,x14
16 -  cbz x14, Loop
17 -  sub x15,x0,x1
18 -  Loop: add x15,x0,x1
19 -  NOP
20 -  NOP

# Carga del resultado de la sumatoria en memoria
21 -  stur x15, [x0,#56]
```

Resultado:

```
0000000000000000
0000000000000001
0000000000000002
```

```
0000000000000003
0000000000000004
0000000000000005
0000000000000006
0000000000000001
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.

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:

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

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 es el siguiente :

```
0000000000000004
0000000000000004
0000000000000008
0000000000000009
```

Para que el resultado sea el correcto, se tienen que implentar la cantidad de NOP's necesarios para que todas las instrucciones puedan acceder al valor más reciente del registro. Entonces el código quedaría así :

Código Modificado :

```
SUB X23, X1, X1
ADD X0, XZR, X4
NOP
NOP
ADD X1, X0, X4
NOP
NOP
ADD X2, X1, X4
NOP
NOP
ADD X3, X2, X4
STUR X0, [X23,0]
STUR X1, [X23, #8]
STUR X2, [X23, #16]
STUR X3, [X23, #24]
```

El nuevo resultado es el siguiente :

```
0000000000000004
0000000000000008
000000000000000C
0000000000000010
```

Ejercicio 3

En este ejercicio vamos agregar instrucciones que no estan implementada en el microprocesador y que son las siguientes: ADDI y CBNZ

ADDI

Como el procesador no reconoce la instruccion ADDI, la idea es poder modificar el ADD para que pueda reconocer números que sean inmediatos. Para ello, lo que se tuvo que hacer es modificar las siguientes entidades :

aludec.sv

Como este módulo decide la operación que va a realizar la alu, dependiendo del aluop obtenido del maindec y de su opcode, tuvimos que modificar para que en el caso del opcode del ADDI, envíe una señal de control de suma.

maindec.sv

En este módulo, de entre otras cosas que especificamos (AluOp, ...) la más importante es ALUSrc que decide de donde va a tomar el operando la ALU.

signext.sv

En este módulo agregamos la extensión de signo del inmediato del ADDI.

CBNZ

signext.sv

En este módulo simplemente tomamos el inmediato necesario de la instrucción y hacemos la extesión de signo a 64 bits.

maindec

Este módulo está encargado de especificar distintos comportamientos de las instrucciones, por lo que agregamos una salida de un bit, cuya funcionalidad es especificar que si está prendida, significa que hay que realizar un Branch if Not Zero, también modificamos el nombre de la antigua salida Branch por BranchZero, ya que esta ahora sirve únicamente para el CBZ.

controller.sv

En éste módulo sólamente tomamos la nueva salida de maindec y la devolvemos.

datapath.sv

Como el datapath está encargado de comunicar datos entre las stages en pipeline, tuvimos que hacer varias modificaciones, la idea del bit BranchNotZero es usarlo en el módulo *memory*, que decide si se va a realizar un branch, por lo tanto teníamos que recuperarlo del *controller* y enviarlo a través del pipeline a memory. Para esto tuvimos que modificar las palabras **ID_EX** y **EX_MEM** que se guardaban en los pipeline, agregándoles el bit, que se le asigna en el módulo *controller*. No hizo falta modificar el último stage (**MEM_WB**) ya que en ese momento ya se ejecutó el módulo *memory*.

memory.sv

Como éste módulo decidía si se debía hacer un branch con el inmediato, tuvimos que modificar la lógica para que considere el bit de BranchNotZero.

processor_arm.sv

Acá definimos BranchNotZero y se linkea el resultado del *controller* con el *datapath*.

Aca podemos apreciar la modificación en todo el datapath :