Lab 1 | Ejercicio 1 (Informe)

Integrantes:

- Emanuel Nicolás Herrador (DNI 44.898.601)
- Juan Bratti (DNI 44.274.011)

Materia: Arquitectura de Computadoras 2023

Introducción

Lo que llevamos a cabo en este ejercicio fue la modificación del microprocesador con pipeline, de tal forma que se incorpore la instrucción MOVZ sin que el funcionamiento de las instrucciones ya existentes se vea afectado. Detallaremos a continuación las modificaciones hechas.

Modificaciones Realizadas

La idea de las modificaciones es usar la estructura que originalmente se encontraba en los módulos brindados por la cátedra, simplemente haciendo que MOVZ se “aplique” en el **signext** y colocando las señales de control en base a las modificaciones que éste realiza (como por ejemplo, *escritura en registro*).

Ya respecto a las modificaciones en particular y detalle en el micro que realizamos, éstas se mencionan a continuación:

1. Imem

- Se agregó el código original brindado por la cátedra con las instrucciones *NOP* para evitar los hazards de dato y control, e instrucciones de tipo MOVZ para probar su correcto funcionamiento con diferentes *LSL*. Además, fue necesario agrandar el tamaño de *ROM* y del *address de imem:*

```
// Cambios en el address de imem
input  logic [  7 : 0] addr,

// Cambios en el tamaño del ROM
logic [N-1 : 0] ROM[0 : 255]
```

- Respecto al programa que se utilizó, se brinda más detalle en la sección *Programa de Verificación*.

2. Maindec

- Se agregó el caso para MOVZ para generar las *señales de control* correspondientes para que se ejecute correctamente. Para ello, lo que se hizo fue agregar el siguiente caso al *case*:

```
11'b110_1001_01??:
      {Reg2Loc, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp} = 9'b110100001; // MOVZ
```

- Respecto a las señales, se consideró lo siguiente:
  - **Reg2Loc** → 1 ⇒ Implica que se debe considerar el registro *Rd* (como referencia de la GreenCardLEGv8) → Es decir, para los bits [4..0]
  - **ALUSrc** → 1 ⇒ Implica que lo que se pasa como *valor B* a la ALU, es el *signImm_E*
  - **MemtoReg** → 0 ⇒ Implica que el valor que se quiere utilizar para escribir en el registro (hacer WriteBack), es el que sale como *resultado de la ALU*.
  - **RegWrite** → 1 ⇒ Implica que *se escribe en un registro*
  - **MemRead** → 0 ⇒ Implica que *no se lee de memoria*
  - **MemWrite** → 0 ⇒ Implica que *no se escribe en memoria*
  - **Branch** → 0 ⇒ Implica que *no se realiza un salto*
  - **ALUOp** → 01 ⇒ Implica que la ALU va a hacer *pass b* (para que pase el valor del inmediato que viene desde *signext*)
    - Este hace que en *aludec*, el *alucontrol* sea el mismo que el de CBZ (i.e., 4'b0111) ⇒ Para el *pass b*

3. Signext

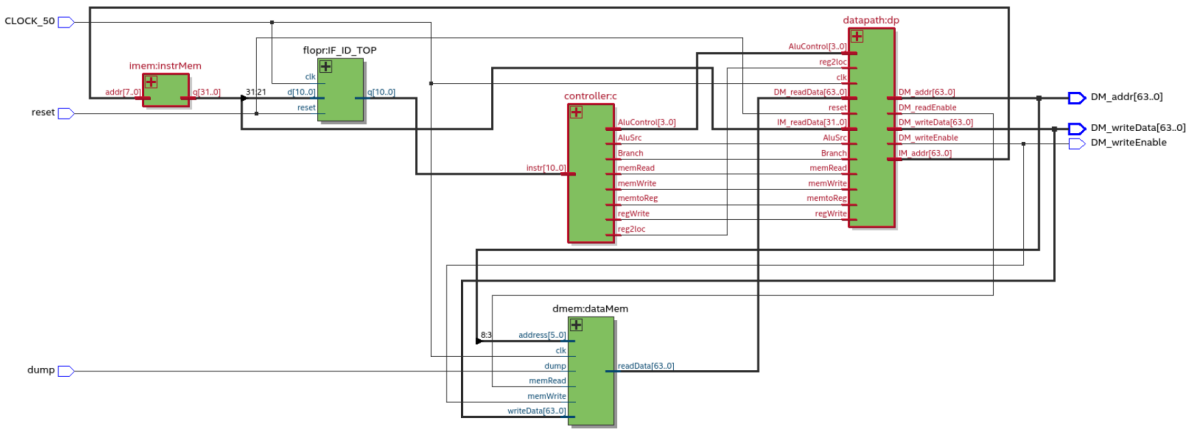
- En este módulo lo que agregamos fueron los casos para implementar las extensiones de signo correspondientes al hacer los *LSL* en la instrucción MOVZ.

```
// MOVZ:
11'b110_1001_0100: y = {{(N - 16) {1'b0}}, a[20 : 5]};  // LSL 00
11'b110_1001_0101: y = {{(N - 32) {1'b0}}, a[20 : 5], 16'b0};  // LSL 01
11'b110_1001_0110: y = {{(N - 48) {1'b0}}, a[20 : 5], 32'b0};  // LSL 10
11'b110_1001_0111: y = {{(N - 64) {1'b0}}, a[20 : 5], 48'b0};  // LSL 11
```

Diagrama Final

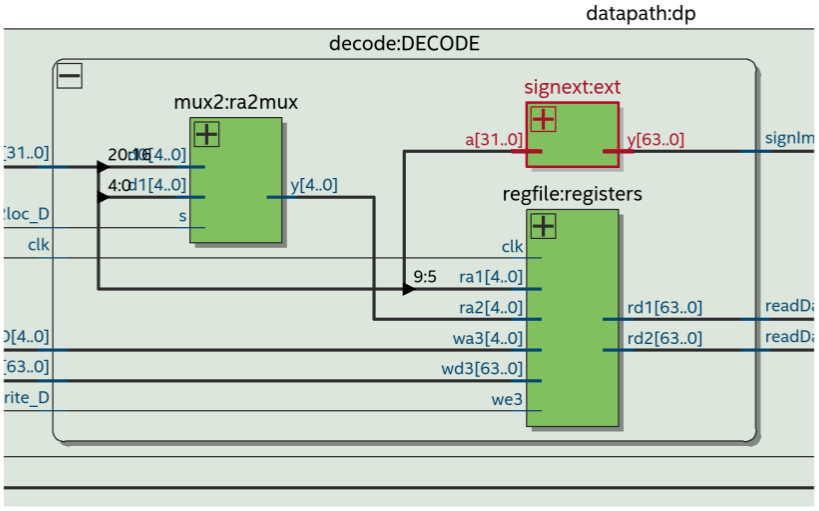
En el diagrama final, la idea es utilizar la estructura del micro que nos brindó la cátedra, sin agregar módulos nuevos, sino implementando toda la funcionalidad de MOVZ dentro de estos y con el uso de las señales de control. Por ello, entonces, el diagrama final es el mismo, con los módulos modificados marcados en rojo:

Paneo general de los cambios



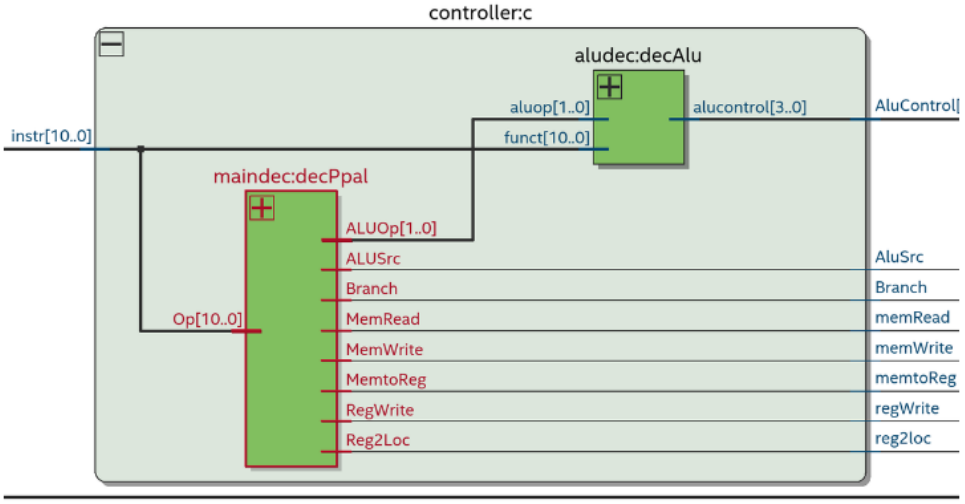
Están marcados en rojo los módulos que engloban los componentes que fueron modificados.

Cambios específicos en datapath



Esta marcado en rojo el componente de signext que fue modificado.

Cambios específicos de controller



Está marcado en rojo el componente maindec que fue modificado.

Programa de Verificación

El programa de verificación utilizado contiene tanto el programa original con las instrucciones *NOP* para evitar *hazards*, como las pruebas para MOVZ que ideamos, las cuales se encuentran en la parte final del test, donde tenemos un caso para cada tipo de *LSL*.

```
.text
    .org 0x0000

    STUR X1, [X0, #0] // MEM 0:0x1
    STUR X2, [X0, #8] // MEM 1:0x2
    STUR X3, [X16, #0] // MEM 2:0x3
    ADD X3, X4, X5
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X3, [X0, #24] // MEM 3:0x9
    SUB X3, X4, X5
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X3, [X0, #32] // MEM 4:0xFFFFFFFFFFFFFFFF
    SUB X4, XZR, X10
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X4, [X0, #40] // MEM 5:0xFFFFFFFFFFFFFFF6
    ADD X4, X3, X4
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X4, [X0, #48] // MEM 6:0xFFFFFFFFFFFFFFF5
    SUB X5, X1, X3
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X5, [X0, #56] // MEM 7:0x2
    AND X5, X10, XZR
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X5, [X0, #64] // MEM 8:0x0
    AND X5, X10, X3
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
```

```
    STUR X5, [X0, #72] // MEM 9:0xA
    AND X20, X20, X20
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X20, [X0, #80] // MEM 10:0x14
    ORR X6, X11, XZR
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X6, [X0, #88] // MEM 11:0xB
    ORR X6, X11, X3
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X6, [X0, #96] // MEM 12:0xFFFFFFFFFFFFFFFF
    LDUR X12, [X0, #0]
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    ADD X7, X12, XZR
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X7, [X0, #104] // MEM 13:0x1
    STUR X12, [X0, #112] // MEM 14:0x1
    ADD XZR, X13, X14
    STUR XZR, [X0, #120] // MEM 15:0x0
    CBZ X0, L1
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X21, [X0, #128] // MEM 16:0x0 (si falla CBZ =21)

L1: STUR X21, [X0, #136] // MEM 17:0x15
    ADD X2, XZR, X1
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR

L2: SUB X2, X2, X1
    ADD X24, XZR, X1
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X24, [X0, #144] // MEM 18:0x1 y MEM 19=0x1
    ADD X0, X0, X8
    CBZ X2, L2
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR

    STUR X30, [X0, #144] // MEM 20:0x1E
    ADD X30, X30, X30
    SUB X21, XZR, X21
        ADD XZR, XZR, XZR
    ADD X30, X30, X20
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    LDUR X25, [X30, #-8]
    ADD X30, X30, X30
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    ADD X30, X30, X16
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X25, [X30, #-8] // MEM 21:0xA (= MEM 9)

    // PRUEBAS PARA MOVZ

    MOVZ X29, #0xFFFF, LSL #0
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
```

```
    STUR X29, [X30, #0] // MEM 22: 0xFFFF

    MOVZ X29, #0xFFFF, LSL #16
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X29, [X30, #8] // MEM 23: 0xFFFF0000

    MOVZ X29, #0xFFFF, LSL #32
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X29, [X30, #16] // MEM 24: 0xFFFF00000000

    MOVZ X29, #0xFFFF, LSL #48
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X29, [X30, #24] // MEM 25: 0xFFFF000000000000

finloop: CBZ XZR, finloop
```

Los resultados obtenidos son los siguientes (*para leer de mem.dump*, recomendamos usar el script processor_tb_check.py que se encuentra en PipelinedProcessorPatterson-Modules/test-benches, el cual lo muestra de forma más “limpia”):