Lab 1 | Ejercicio 2 (Informe)

Integrantes: Herrador Emanuel Nicolás, Bratti Juan

Materia: Arquitectura de Computadoras 2023

Introducción

Lo que llevamos a cabo en este ejercicio fue la implementación de los bloques de **detección de hazards** (HDU), **forwarding** (FU) y la generación de stalls en el microprocesador.

Modificaciones Realizadas

Las modificaciones principales en el micro fueron las siguientes:

1. Datapath

- **Modificaciones para data hazards** (nos referimos a stalls porque tenemos forwarding hecho)
    - Agregamos la señal *data_hazard*, la cual indica en qué momento generar *stall*. Ésta es calculada en la *HDU* dentro del módulo DECODE.
    - Se pasa la señal *data_hazard* a FETCH con el objetivo de mantener el PC en su valor. Al igual que esto, el registro `IF_ID` es deshabilitado para que no sufra cambios en éste clock.
    - Agrupamos las señales de control en `ID_EX_control_signals` con el objetivo de setearlas en 0 en caso de un *data hazard* para el cual tengamos que aplicar *stall*. El reseteo se hace antes de colocar los valores de señales en el registro `ID_EX`.
    - El *data_hazard*, tal y como fue mencionado en el enunciado, se devuelve como output del *datapath* para que *processor_arm* pueda deshabilitar en éste clock la parte del

registro `IF_ID`.

- **Modificaciones para forwarding**
  - Agregamos los valores `IF_ID_rn` y `IF_ID_rm` de los registros de la instrucción, para setearlos en la etapa de DECODE, agregarlos al registro `ID_EX` y de ahí poder pasarlo al módulo de EXECUTE para realizar forwarding.
  - Del mismo modo que en el anterior punto, se agregaron las señales para:
    - EX_MEM_regWrite → viene del registro EX_MEM
    - EX_MEM_rd → viene del registro EX_MEM
    - EX_MEM_aluResult → viene del registro EX_MEM
    - MEM_WB_regWrite → viene del registro MEM_WB
    - MEM_WB_rd → viene del registro MEM_WB
    - MEM_WB_aluResult → es el writeData3 calculado en la etapa *writeback*

    para calcular el forwarding necesario a hacer (tipo de forwarding).

2. Processor_arm

- Se crea la señal *data_hazard* la cual se calcula en el *datapath* y sirve en éste módulo para deshabilitar el registro `IF_ID_TOP` en caso de que se tenga que generar un *stall*.

3. Decode

- Como se mencionó en el datapath, para el forwarding lo que hacemos es calcular en este módulo cuáles son los registros `Rn` y `Rm`, y guardarlos en `IF_ID_rn` y `IF_ID_rm` para su posterior uso en la forwarding unit.
- También dentro del módulo de DECODE se hace uso de la HDU para calcular cuándo vamos a tener un *data hazard* y necesitamos hacer *stall*.

4. Hazard Detection Unit (HDU)

- Lo que hicimos en este nuevo módulo, es simplemente setear el bit de *data_hazard* si y sólo si se cumple la siguiente condición:

```
if  (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRn1) or
      (ID/EX.RegisterRd = IF/ID.RegisterRm2)))
      stall the pipeline
```

Ésta condición fue establecida y colocada en base a lo que vimos en la cátedra y lo visto en el libro de *Patterson* en la sección 4.7 “Data Hazards: Forwarding vs Stalling”.

5. Execute

- Dadas las señales que el *datapath* nos pasa para que podamos calcular con la forwarding unit qué tipo de forwarding se hace para cada entrada de la ALU, lo que hacemos es calcular efectivamente el tipo de forwarding y usar los valores obtenidos de forward A y forward B para definir qué valor es el que vamos a considerar como input de la ALU.
  - Respecto a ello, el input A de la ALU (el que se encuentra usualmente arriba en los diagramas) consideramos que los valores entre los cuales se elige son:
    - readData1_E
    - MEM_WB_aluResult
    - EX_MEM_aluResult
  - Del mismo modo, para el input B de la ALU, lo que consideramos es:
    - readData2_E
    - MEM_WB_aluResult
    - EX_MEM_aluResult

    con la diferencia que aquí sigue estando el *mux2* dependiente de la *AluSrc*, el cual decide si consideramos alguno de los valores mencionados arriba o el inmediato *signImm_E*
  - Para esto generamos un nuevo módulo simple *mux4*.

6. Forwarding Unit

- Lo que hicimos en este nuevo módulo es simplemente definir las condiciones para determinar qué tipo forwarding se toma para cada entrada de la ALU.

```
always_comb begin
    forwardA = 2'b00;
    forwardB = 2'b00;

    // Forward A (EX, MEM)
    if (EX_MEM_regWrite && (EX_MEM_rd !== 31) && (EX_MEM_rd === ID_EX_rn1)) forwardA = 2'b10;
    else if (MEM_WB_regWrite && (MEM_WB_rd !== 31) && (MEM_WB_rd === ID_EX_rn1)) forwardA = 2'b01;

    // Forward B (EX, MEM)
    if (EX_MEM_regWrite && (EX_MEM_rd !== 31) && (EX_MEM_rd === ID_EX_rm2)) forwardB = 2'b10;
    else if (MEM_WB_regWrite && (MEM_WB_rd !== 31) && (MEM_WB_rd === ID_EX_rm2)) forwardB = 2'b01;
  end
```

  - Las condiciones que seguimos son las siguientes:

FORWARDING

1. *EX hazard:*

```
if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 31)
and (EX/MEM.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 10

if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 31)
and (EX/MEM.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 10
```

2. MEM hazard

```
if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 31)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31)
        and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRn1))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 01

if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 31)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31)
        and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRm2))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 01
```

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

FIGURE 4.54 The control values for the forwarding multiplexors in Figure 4.53. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

Éstas condiciones fueron establecidas y colocadas en base a lo que vimos en la cátedra y lo visto en el libro de *Patterson* en la sección 4.7 “Data Hazards: Forwarding vs Stalling”.

7. Flip-Flop D with Enable

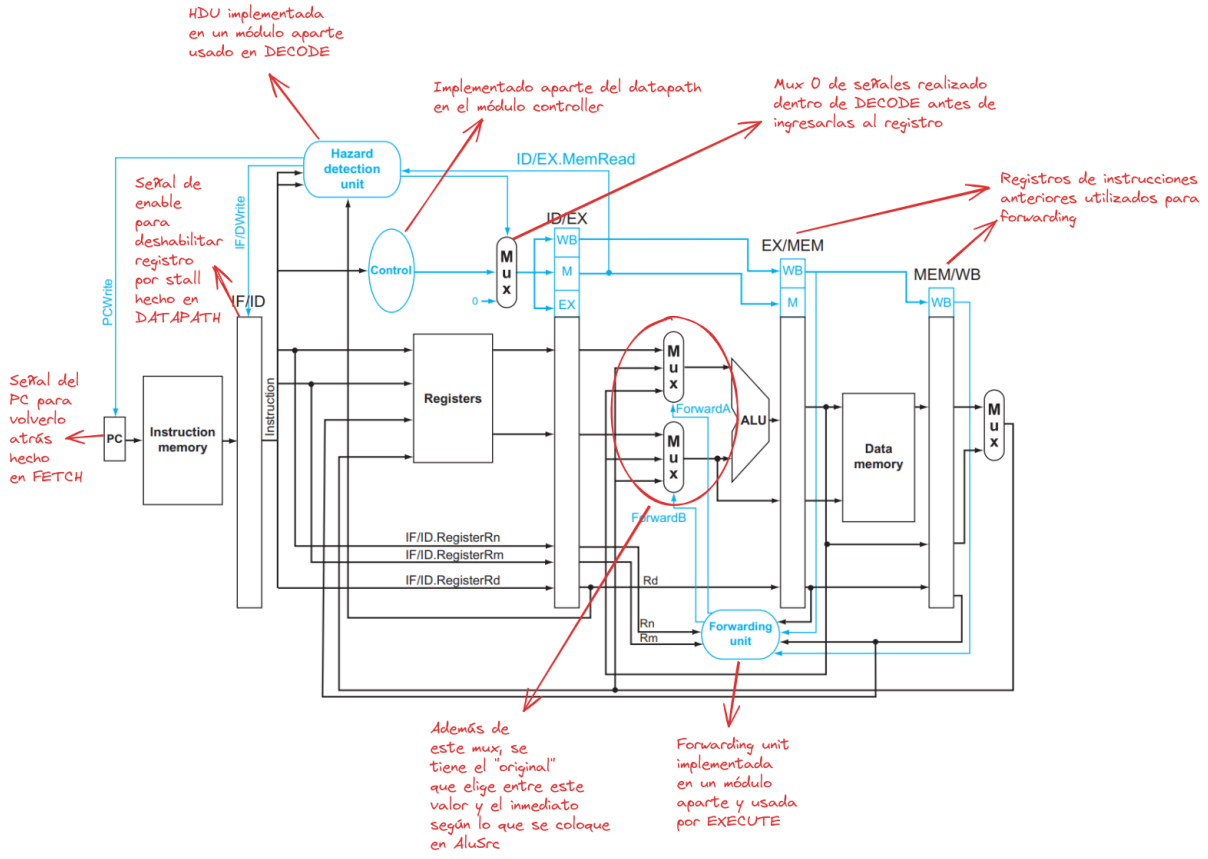
- La modificación que se realizó en este módulo simple fue hecha con el objetivo de que los registros del pipeline utilicen la señal de enable para poder generar correctamente los *stalls* en los casos que sea necesario.
- La implementación se hizo en función de lo explicitado en el enunciado del ejercicio:

```
always_ff @(posedge clk or posedge reset)
    if (reset) begin
      q <= '0;
```

```
    end else if (enable) begin
      q <= d;
    end
```

Diagrama Final

En el siguiente diagrama podemos ver la estructura general de nuestro micro tal y como la plantea el libro de *Patterson*, ya que la implementación fue realizada tomando como referencia. Motivo de ello, también se marcará en rojo cada módulo agregado con su correspondiente implementación.



Programa de Verificación

El programa de verificación que usamos para testear que funcione correctamente nuestra implementación del micro con forwarding-stall contiene los siguientes casos:

- Pruebas iniciales → se corrobora que funcione correctamente el forwarding y el stall.

```
MOVZ X1, #0xFFFF, LSL #0
STUR X1, [X0, #0xB0] // MEM 22:0xFFFF

MOVZ X1, #0xAAAA, LSL #0
STUR X1, [X0, #0xB8] // MEM 23:0xAAAA
```

```
LDUR X2, [X0, #0xB0]

ADD X3, X2, X2
STUR X3, [X0, #0xC0] // MEM 24:0x1fffe

// Reiniciar registros modificados

MOVZ X1, #0x1, LSL #0
MOVZ X2, #0x2, LSL #0
MOVZ X3, #0x3, LSL #0
```

- Pruebas de MOVZ → son las mismas pruebas definidas en el ejercicio 1.

```
MOVZ X29, #0xFFFF, LSL #0
STUR X29, [X0, #0xC8] // MEM 25: 0xFFFF

MOVZ X29, #0xFFFF, LSL #16
STUR X29, [X0, #0xD0] // MEM 26: 0xFFFF0000

MOVZ X29, #0xFFFF, LSL #32
STUR X29, [X0, #0xD8] // MEM 27: 0xFFFF00000000

MOVZ X29, #0xFFFF, LSL #48
STUR X29, [X0, #0xE0] // MEM 28: 0xFFFF000000000000

// Reiniciar registros modificados

MOVZ X29, #0x1D, LSL #0
```

- Código brindado por la cátedra → se dejan sólo las instrucciones *NOP* para evitar hazards de control.

```
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
STUR X3, [X0, #24] // MEM 3:0x9
SUB X3, X4, X5
STUR X3, [X0, #32] // MEM 4:0xFFFFFFFFFFFFFFFF
SUB X4, XZR, X10
STUR X4, [X0, #40] // MEM 5:0xFFFFFFFFFFFFFFF6
ADD X4, X3, X4
STUR X4, [X0, #48] // MEM 6:0xFFFFFFFFFFFFFFF5
SUB X5, X1, X3
STUR X5, [X0, #56] // MEM 7:0x2
AND X5, X10, XZR
STUR X5, [X0, #64] // MEM 8:0x0
AND X5, X10, X3
STUR X5, [X0, #72] // MEM 9:0xA
AND X20, X20, X20
STUR X20, [X0, #80] // MEM 10:0x14
ORR X6, X11, XZR
STUR X6, [X0, #88] // MEM 11:0xB
ORR X6, X11, X3
STUR X6, [X0, #96] // MEM 12:0xFFFFFFFFFFFFFFFF
LDUR X12, [X0, #0]
ADD X7, X12, XZR
STUR X7, [X0, #104] // MEM 13:0x1
```

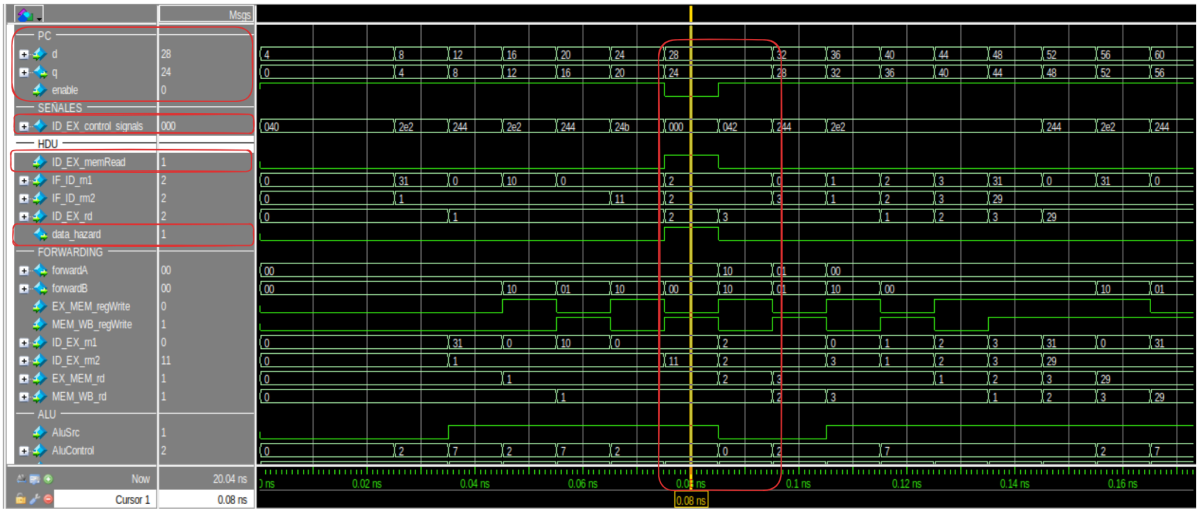
```
    STUR X12, [X0, #112] // MEM 14:0x1
    ADD XZR, X13, X14
    STUR XZR, [X0, #120] // MEM 15:0x0
    CBZ X0, L1
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
        ADD XZR, XZR, XZR
    STUR X21, [X0, #128] // MEM 16:0x0(si falla CBZ =21)

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

L2:
    SUB X2, X2, X1
    ADD X24, XZR, X1
    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 X30, X30, X20
    LDUR X25, [X30, #-8]
    ADD X30, X30, X30
    ADD X30, X30, X16
    STUR X25, [X30, #-8] // MEM 21:0xA (= MEM 9)
```

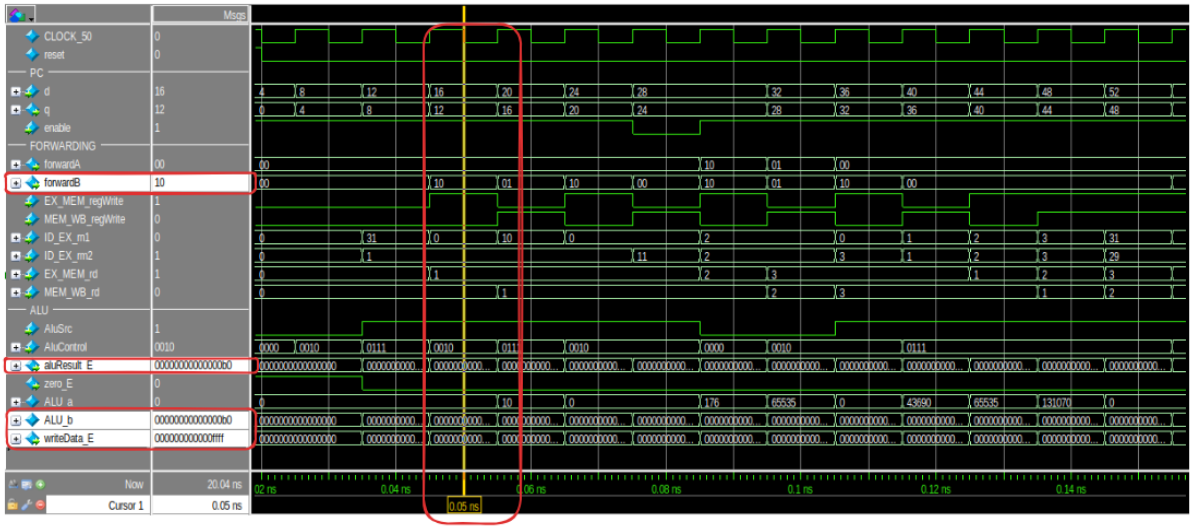
Comportamiento de stalls, FU y HDU visualizados en la Wave

Stalls



- En este momento, tenemos en etapa de WB a `STUR X1, [X0, #0xB8] // MEM 23:0xAAAA`, lo cual puede verse dado que **DM_addr** = 23 * 8 y **DM_writeData** = 0xAAAA
- Si usamos esto para guiarnos temporalmente, sabemos que en MEM está `LDUR X2, [X0, #0xB0]`, mientras que en EXECUTE se encontraría (si no tuviéramos stalls ni forwarding), el `ADD X3, X2, X2`.
  Motivo de ello, como una dependencia de dato de tipo RAW con LDUR debe generar un stall, esto se puede visualizar en los siguientes hechos:
  - El PC no aumenta, por lo que se queda en 24 (el valor q del flip flop D)
  - La HDU detecta hazard
  - Las señales de control se ponen en 0

Forwarding



- En este caso se va a analizar la situación donde tenemos:
  - `MOVZ X1, #0xFFFF, LSL #0` → está en MEM
  - `STUR X1, [X0, #0xB0] // MEM 22:0xFFFF` → está en EXECUTE
- Motivo de ello podemos visualizar que el registro en donde tenemos dependencia de dato es el `X1` dado que `ID_EX_rm2` = `EX_MEM_rd`.
- Como el forwarding viene desde MEM, entonces para el STUR lo que se hace es pasarle el valor que obtuvo la ALU en el ciclo anterior. Es decir, en este caso se le pasa 0xFFFF. Ésto se puede visualizar dado que `writeData_E` se encuentra en 0xFFFF.
  - En este caso, en particular se considera que el forwarding se realiza para el input b de la ALU dado que el registro que necesitamos es el `Rt`.