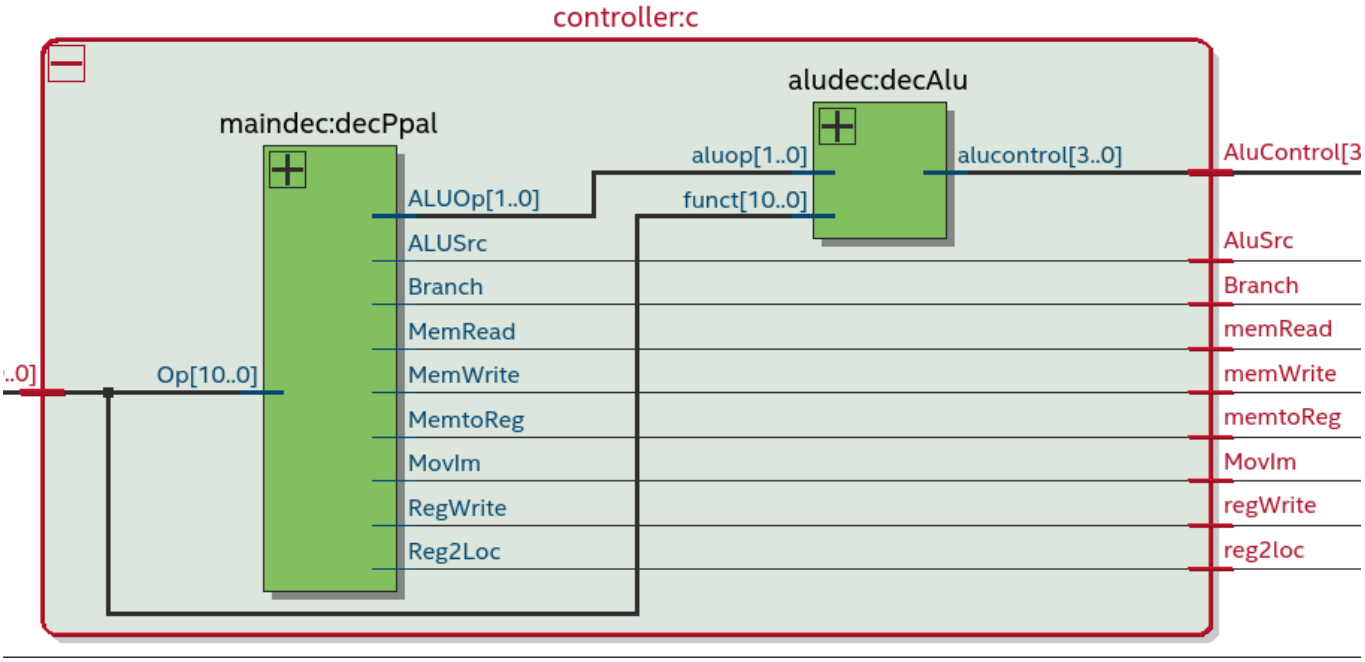
MOVZ ej 1

introducimos cambios en los modulos controler y datapath

Controler

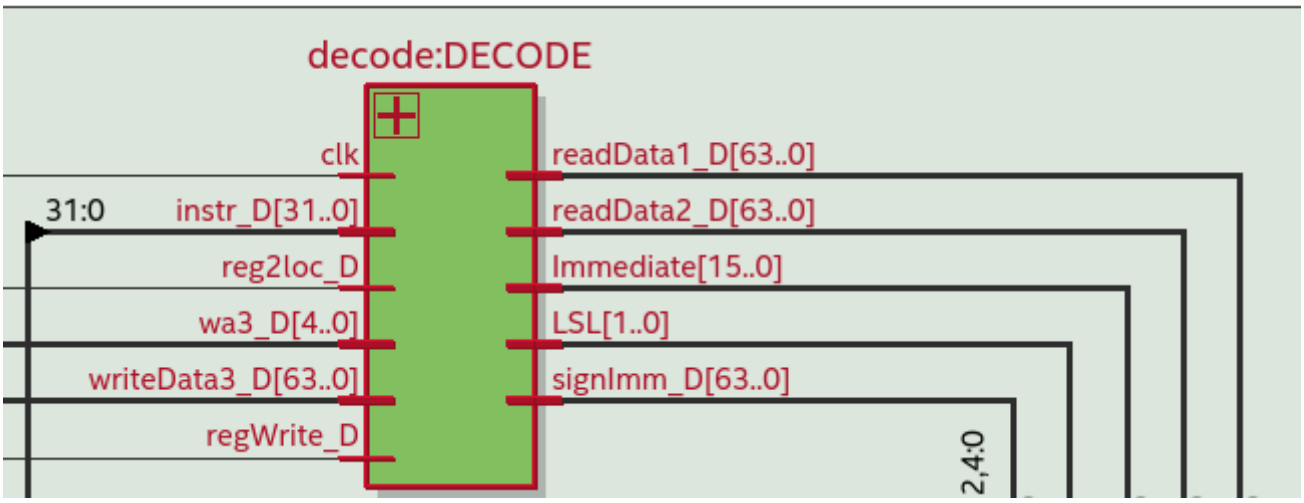
La primera modificacion realizada fue en maindec.sv es decir en el bloque controler agregando una guarda que chequea el opcode de MOVZ y setea una nueva señal de control MovIm en uno, señal que se usa para utilizar en la etapa de writeBack el resultado de la intruccion tipo IM calculado en el nuevo modulo mov_im
De esta manera podemos reconocer los 9 bits del opcode de MOVZ y setear las señales para leer el registro RD y escribirlo



(nueva señal MovIm)

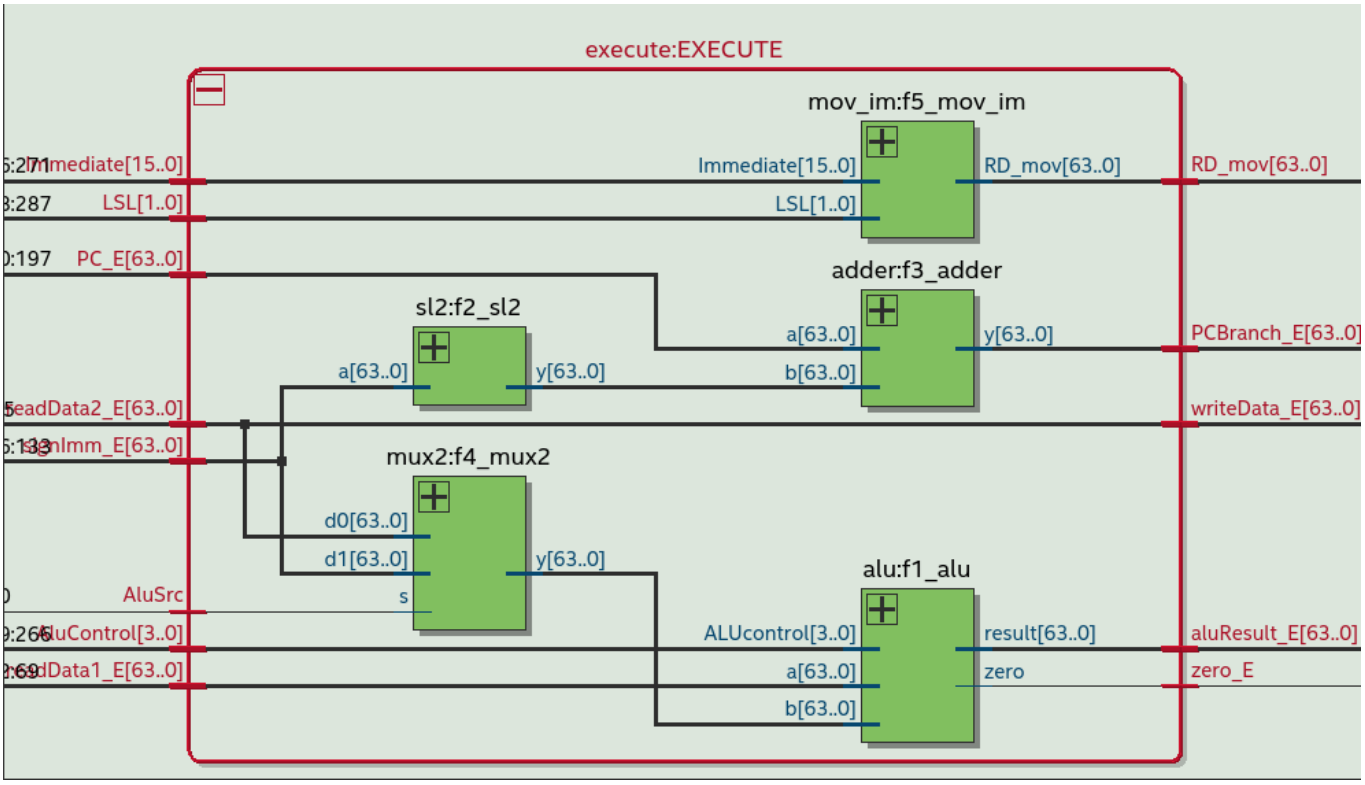
Datapath

Decode



Se agregaron 2 señales a este modulo a partir de la instruccion el modulo decode es el encargado de discriminar los bits que corresponden al inmediato que se quiere guardar en el registro de destino y los bits correspondientes al shift dentro del registro (LSL)
Estas señales pasan por el flipflop ID_EX

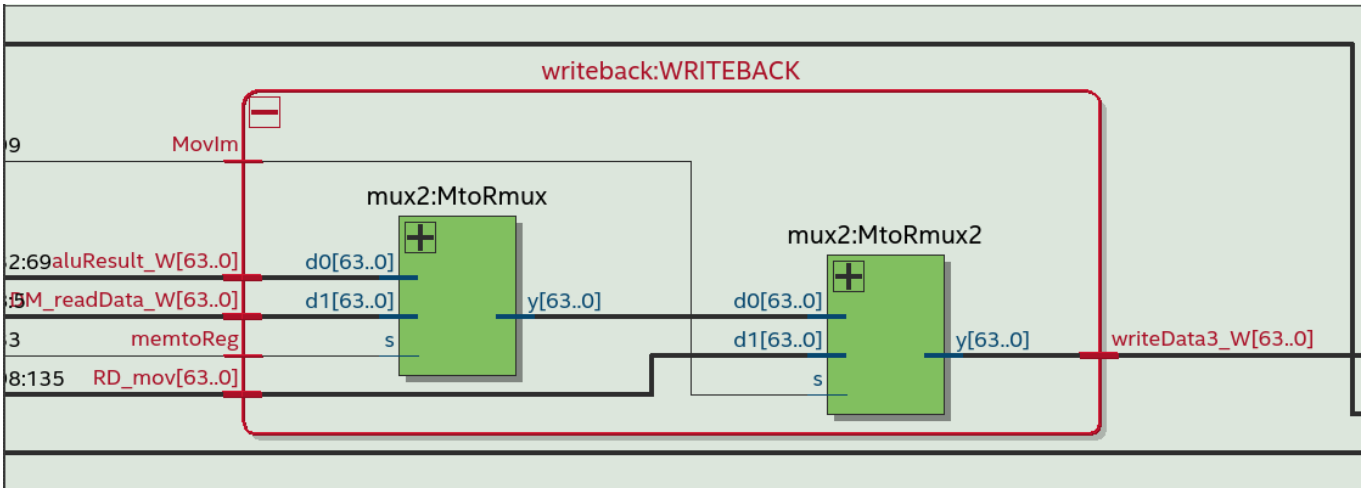
Execute



Se agrego el nuevo modulo mov_im que es el encargado de generar el resultado correspondiente de shiftear el inmediato 0, 16, 32 o 64 bits de acuerdo al valor dado en LSL, el resultado de esta operacion en guardado en RD_mov para su posterior uso, este resultado es tambien guardado en los flipflop EX_MEM y MEM_WB

WriteBack

A este modulo se le agrego un nuevo multiplexor que chequea si la señal MovIm esta en 1 uno o no, si la señal esta en 1 significa que la instruccion ejecutada es un MOVZ y deja pasar el valor de RD_mov como writeData3 para que este sea escrito sobre el registro correspondiente, en caso contrario el funcionamiento es el mismo que antes de la modificación



Codigo de prueba

Utilizamos el siguiente codigo

```
.text
        .org 0x0000

STUR X1, [X0, #0]
STUR X2, [X0, #8]
STUR X3, [X16, #0]
ADD X3, X4, X5
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X3, [X0, #24]
SUB X3, X4, X5
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X3, [X0, #32]
SUB X4, XZR, X10
ADD XZR, XZR, XZR
```

```
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X4, [X0, #40]
ADD X4, X3, X4
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X4, [X0, #48]
SUB X5, X1, X3
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X5, [X0, #56]
AND X5, X10, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X5, [X0, #64]
AND X5, X10, X3
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X5, [X0, #72]
AND X20, X20, X20
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X20, [X0, #80]
ORR X6, X11, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X6, [X0, #88]
ORR X6, X11, X3
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X6, [X0, #96]
LDUR X12, [X0, #0]
ADD XZR, XZR, XZR
```

```
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD X7, X12, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X7, [X0, #104]
STUR X12, [X0, #112]
ADD XZR, X13, X14
STUR XZR, [X0, #120]
CBZ X0, L1
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X21, [X0, #128]
L1:STUR X21, [X0, #136]
ADD X2, XZR, X1
ADD XZR, XZR, XZR
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
ADD XZR, XZR, XZR
STUR X24, [X0, #144]
ADD X0, X0, X8
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
CBZ X2, L2
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X30, [X0, #144]
ADD X30, X30, X30
ADD XZR, XZR, XZR
```

```
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
SUB X21, XZR, X21
ADD X30, X30, X20
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
LDUR X25, [X30, #-8]
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD X30, X30, X30
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD X30, X30, X16
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
ADD XZR, XZR, XZR
STUR X25, [X30, #-8]
MOVZ X26, #48830, LSL 0
MOVZ X0, #0, LSL  0
MOVZ X17, #43966, LSL 16
MOVZ X18, #49354, LSL  32
MOVZ X19, #4660, LSL 48
STUR X26, [X0, #208]
STUR X17, [X0, #216]
STUR X18, [X0, #224]
STUR X19, [X0, #232]
finloop: CBZ XZR, finloop
```

Como se puede ver al final del codigo utilizamos varias instrucciones MOVZ una despues de la otra para chequear que el pipeline este funcionando correctamente y guardamos los valores que escriben estas instrucciones en las posiciones 26 a 29 de la memoria

en memdump obtenemos el siguiente resultado para esas posiciones de memoria

```
26 000000000000BEBE
27 00000000ABBE0000
```

```
28 0000C0CA00000000
29 1234000000000000
```

lo que demuestra que MOVZ esta funcionando correctamente, en cuanto al resto de posiciones de memoria de memDump el comportamiento es exactamente el esperado en la consigna

HDU ej 2

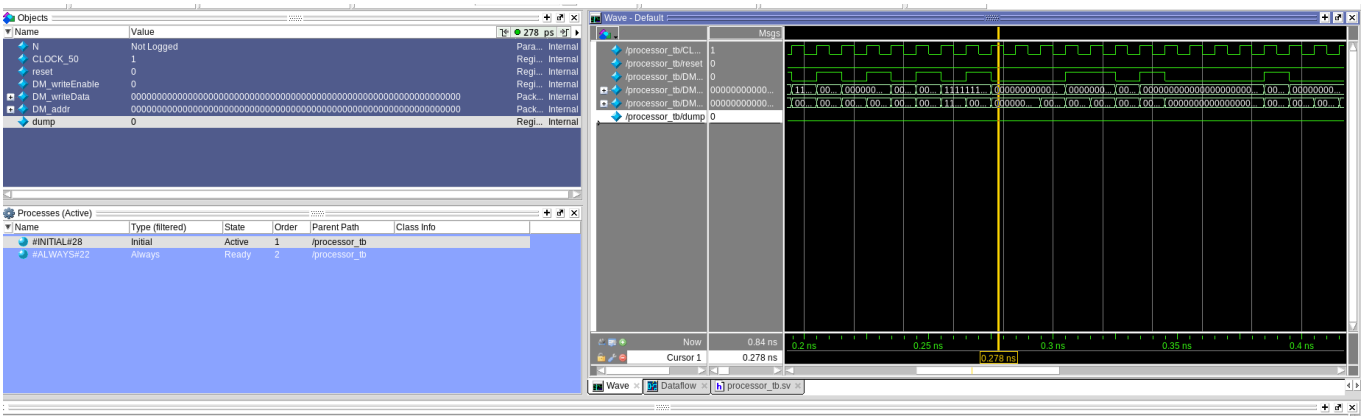
Agregamos el modulo de hdu el cual contiene la logica e implementacion del hazard detection unit, el cual es usado dentro del datapath.
Modificamos ademas el modulo del procesador para poder hacer display de estas señales y poder testearlas.

HDU

El módulo `hdu` (Hazard Detection Unit) en este contexto es responsable de detectar riesgos (hazards) en el pipeline del procesador, específicamente en las etapas de ejecución y memoria.

Vamos a analizar detalladamente el código del módulo `hdu` y su implementación en el archivo `datapath`.

Módulo `hdu`:



(ejemplo de stall)

```
module hazard_detection_unit
        (input logic idex_memread,
        input logic [4:0] idex_rd, ifid_rn1, ifid_rm2,
        output logic stall);

        assign stall = idex_memread && ((idex_rd === ifid_rn1) || (idex_rd
=== ifid_rm2));
```

```
endmodule
```

- **Entradas:**
  - `ifid_rn1` y `ifid_rm2`: Registros de destino de la instrucción previa en la etapa de memoria.
  - `idex_rd`: Registro de destino de la instrucción previa en la etapa de ejecución.
  - `idex_memread`: Indica si la instrucción en la etapa de ejecución requiere una lectura de memoria.
- **Salidas:**
  - `PCWrite` y `IF_D_Write`: Control de escritura en el contador de programa y en la etapa de decodificación, respectivamente.
  - `stall`: Indica si se ha detectado algún tipo de riesgo.

Implementación en `datapath`:

```
        hazard_detection_unit HAZARD (.idex_memread(qID_EX[264]),
.idex_rd(qID_EX[4:0]),

.ifid_rn1(ra1_D), .ifid_rm2(ra2_D), .stall(stall_signal));
```

- El módulo `hdu` se instancia en el bloque `datapath` y se conectan las señales correspondientes.
- `ra1_D` y `ra2_D` son los registros de destino de la instrucción previa en la etapa de memoria.
- `qID_EX[4:0]` es el registro de destino de la instrucción previa en la etapa de ejecución.
- `qID_EX[264]` indica si la instrucción en la etapa de ejecución requiere una lectura de memoria.
- `stall` es la salida que indica si se ha detectado algún tipo de riesgo.

Módulo `flopre`:

```
module flopre #(parameter N = 64)(
    input logic enable,
    input logic clk, reset,
    input logic [N-1:0] d,
```

```
    output logic [N-1:0] q
);

    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            q <= 'b0;
        end else
            if (enable) begin
                q <= d; // actualizar valor.
            end
        end

endmodule
```

- Este módulo implementa un registro (`flopre`), que es similar a un flip-flop pero con una habilitación (`enable`).
- La entrada `d` se carga en el registro en el flanco positivo del reloj (`posedge clk`) cuando `enable` es 1.
- En caso de un reset, el registro se reinicia a 0.

Este tipo de registros con habilitación se utilizan comúnmente para controlar el flujo de datos en un pipeline, permitiendo o deteniendo el avance de los datos según sea necesario. En el contexto de `datapath` , este tipo de registros se utilizan para gestionar la propagación de datos a través del pipeline del procesador, controlando posibles riesgos y asegurando un funcionamiento correcto.

Módulo `forwarding_unit` :

Este mecanismo asegura un flujo de datos eficiente y evita esperas innecesarias, mejorando el rendimiento del pipeline y permitiendo una ejecución más rápida de las instrucciones en una arquitectura de CPU ARM.

```
module forwarding_unit
    (input logic exmem_regwrite, memwb_regwrite,
    input logic  [4:0] idex_rn1, idex_rm2, exmem_rd, memwb_rd,
    output logic [1:0] forwardA, forwardB);

    logic [1:0] forA, forB;

    always_comb
        begin
```

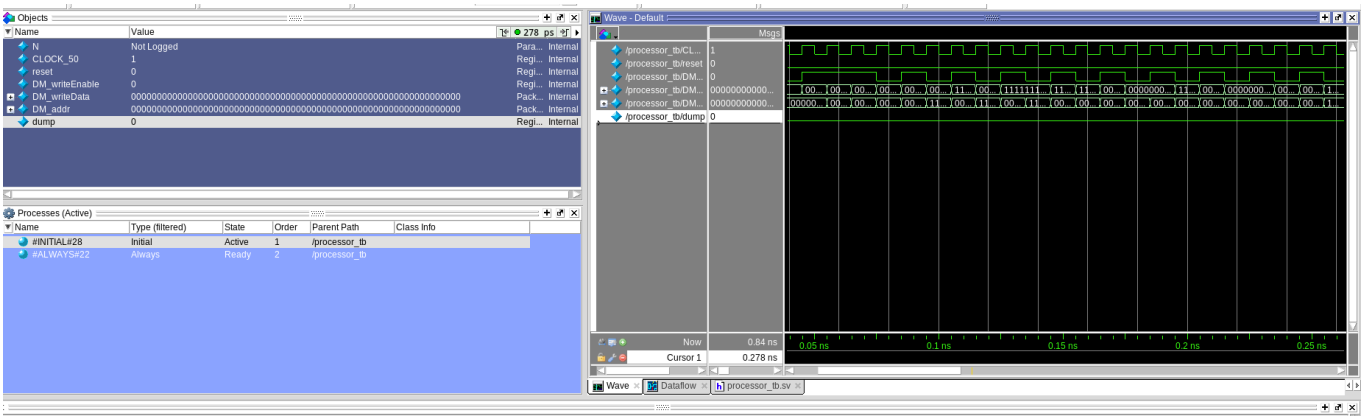
```
        /* EX hazard rn1 && MEM hazard rn1 */
            if (exmem_regwrite && exmem_rd !== 31 && exmem_rd === idex_rn1)
                forA = 2'b10;
            else if (memwb_regwrite && memwb_rd !== 31 && memwb_rd ===
idex_rn1) //! Distinto a como estaba en el libro
                forA = 2'b01;
            else
                forA = 2'b00;
        /* EX hazard rm2 && MEM hazard rm2 */
            if (exmem_regwrite && exmem_rd !== 31 && exmem_rd === idex_rm2)
                forB = 2'b10;
            else if (memwb_regwrite && memwb_rd !== 31 && memwb_rd ===
idex_rm2)
                forB = 2'b01;
            else
                forB = 2'b00;

            forwardA = forA;
            forwardB = forB;
        end

endmodule
```

Forwarding Unit en el Datapath

El módulo `forwarding_unit` es crucial en el datapath de una CPU ARM, ya que se encarga de manejar la propagación de datos a través del pipeline, específicamente para superar riesgos de datos (data hazards). Los data hazards ocurren cuando una instrucción depende de los resultados de una instrucción anterior que aún no ha completado su ejecución.



Funcionamiento:

- **Entradas:**
  - `memwb_regwrite`: Señal que indica si la instrucción en la etapa de escritura de memoria (MEM) está escribiendo en un registro.
  - `exmem_regwrite`: Señal que indica si la instrucción en la etapa de ejecución (EX) está escribiendo en un registro.
  - `memwb_rd`: Identificador del registro de destino en la etapa de escritura de memoria.
  - `exmem_rd`: Identificador del registro de destino en la etapa de ejecución.
  - `idex_rn1`: Identificador del registro fuente (Rn) en la etapa de decodificación.
  - `idex_rm2`: Identificador del registro fuente (Rm) en la etapa de decodificación.
- **Salidas:**
  - `ForwardA` y `ForwardB`: Estas señales indican las fuentes de datos para las operaciones en la etapa de ejecución (EX). Son salidas de dos bits que pueden tener los siguientes valores:
    - `2'b00`: No hay forwarding para esta fuente de datos.
    - `2'b01`: Forwarding desde la etapa de escritura de memoria (MEM) hacia la etapa de ejecución (EX).
    - `2'b10`: Forwarding desde la etapa de ejecución (EX) hacia la etapa de ejecución (EX).

Uso en el Contexto de una CPU ARM:

1. **Forwarding A (`ForwardA`):**
   - `2'b00`: No hay forwarding. La fuente de datos proviene del registro especificado por `ID_EX_RegisterRn`.
   - `2'b01`: Forwarding desde la etapa de escritura de memoria (MEM) hacia la etapa de ejecución (EX). Se usa cuando hay una dependencia de datos y la instrucción en MEM acaba de escribir en el registro especificado por `ID_EX_RegisterRn`.
   - `2'b10`: Forwarding desde la etapa de ejecución (EX) hacia la etapa de ejecución (EX). Se utiliza cuando la instrucción en EX acaba de escribir en el registro especificado por `ID_EX_RegisterRn`.
2. **Forwarding B (`ForwardB`):**
   - Similar a Forwarding A pero aplicado al registro especificado por `ID_EX_RegisterRm`.

Uso en la Prevención de Riesgos:

- **Raw Hazard (Riesgo de Lectura después de Escritura):**
  - **Escritura en MEM (ForwardA):** Permite que una instrucción en la etapa de ejecución (EX) acceda a datos escritos por una instrucción anterior en la etapa de escritura de memoria (MEM).

    - **Escritura en EX (ForwardB):** Permite que una instrucción en la etapa de ejecución (EX) acceda a datos escritos por una instrucción anterior en la etapa de ejecución (EX).
- **Control Hazard (Riesgo de Control):**
    - **memwb_regwrite:** Si la instrucción en MEM está escribiendo en un registro, indica que hay un resultado que puede ser necesario para instrucciones futuras.