ITESO

Universidad Jesuita
de Guadalajara

PRÁCTICA 3

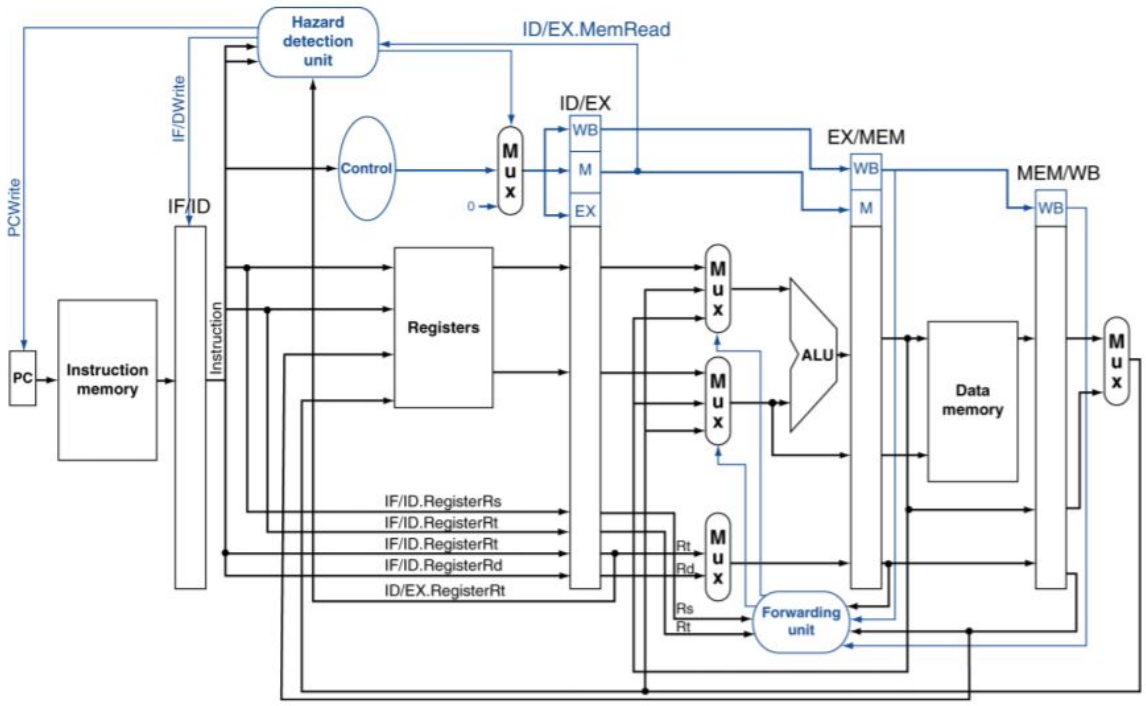
PipeLine

Lilia Arceli Lobato Martínez ie 706937
Jorge Karim Naciff Maldonnat ie708501

Introducción

En el salón de clase se vieron las mejoras en el rendimiento que ofrecen los procesadores con pipeline, es decir, que aprovechan el paralelismo a nivel de instrucción, donde dos o más instrucciones se ejecutan simultáneamente, cada una de ellas en diferente etapa de ejecución, aprovechando el hecho de que cada etapa usa una unidad diferente del hardware. También se mencionaros los diferentes riegos, o "hazards", que existen en los procesadores con pipeline, los métodos con los que se pueden mitigar, y el impacto que estos tienen en el desempeño del procesador.

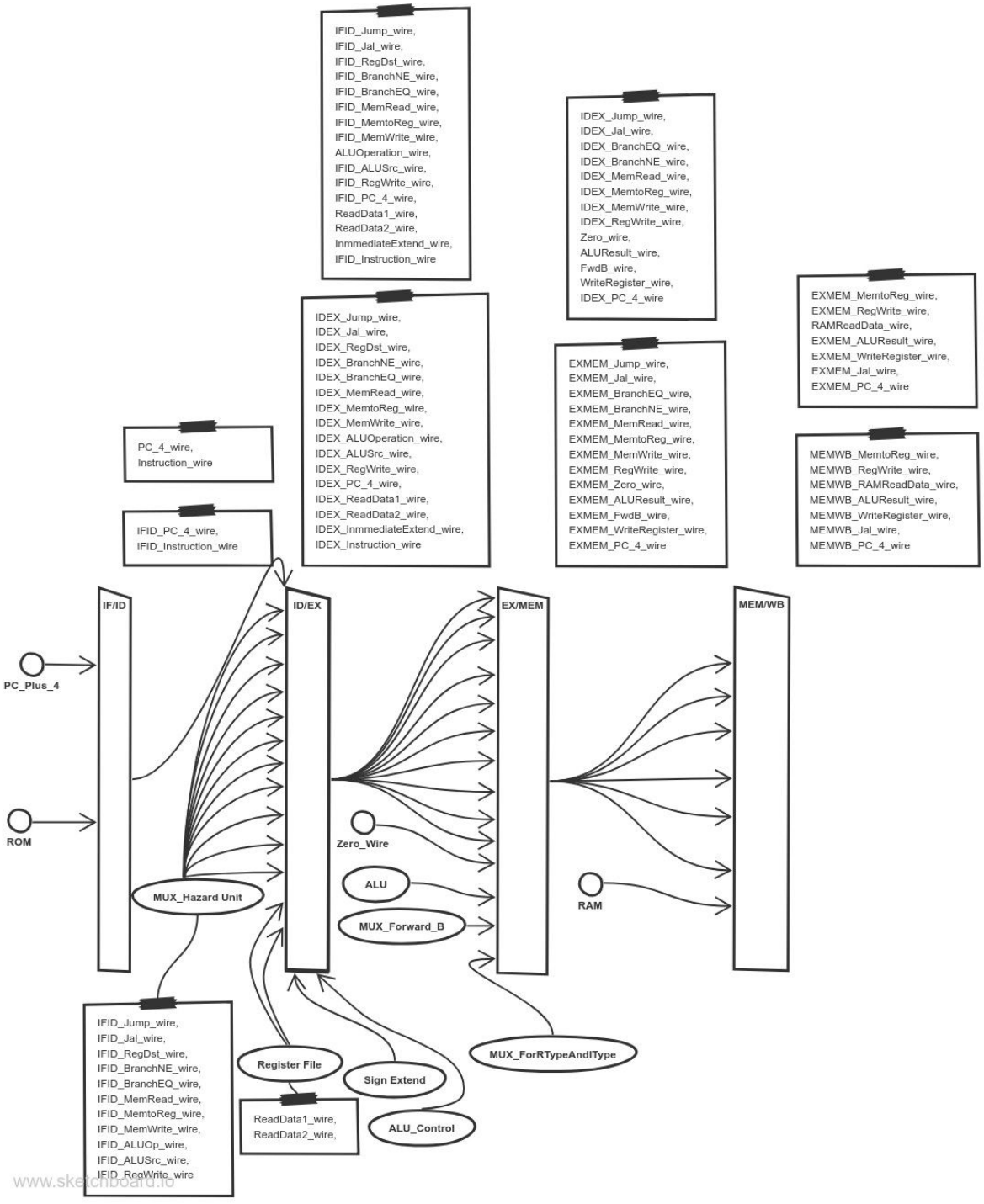
La siguiente figura esquematiza la estructura del procesador MIPS con pipeline visto en clase. Este esquemático no muestra todos los detalles de la implementación del procesador, pero sirve de guía. Le corresponde al alumno trabajar los detalles.



Las barras simbolizan los registros que será necesario agregar para guardar la información generada por una etapa en el ciclo n, y que servirá de entrada para la siguiente etapa en el ciclo n+1.

Las líneas azules simbolizan las señales de control, y la dirección del registro a escribir, que se propagan de etapa en etapa, de tal forma que cada etapa obtiene las señales de control que necesita.

Diagrama de Flujo



Decisiones Tomadas

Antes de iniciar cualquier cambio para generar un procesador en pipeline, cambiamos la forma en que implementamos la memoria RAM para que las direcciones se mostraran tal cual en MARS.

La práctica se dividió en tres secciones: Pipeline register, Forwarding Unit y Hazard Detection Unit.

Para el Pipeline register se creó un archivo PLRegister basado en Register, con este se instancian los registros del Pipe. Son 4 registros en total donde las entradas depende de la etapa en la que se encuentre y las salidas son utilizadas en la siguiente etapa.

Para el Forwarding Unit se crearon multiplexores de 3 a 1, basados en los mux de 2 a 1, con ellos se instancia el Forwarding Unit para A y B. Aparte de se codificó Forward_Unit con el código obtenido de las diapositivas vistas en clase.

La etapa más complicada fue el Hazard Detection Unit. Para implementar los Stalls utilizamos un mux 2 a 1 donde dependiendo de Stall_sel_wire se pasan 0s o los datos correspondientes. De igual forma se utilizó el código visto en clase para codificar Hazard_Unit.

Una de las decisiones más importantes que se tomaron fue codificar lo más posible, el problema fue que al meter la unidad de forwarding y los hazards, el código de las Torres de Hanoi se quedaba ciclado porque se sobre escribe una torre con las direcciones de retorno. Al intentar corregir el problema parece ser que viene del código de Hanoi así que... decidimos continuar con implementaciones más sencillas.

Módulos en Quartus

A continuación se muestra las implementaciones y módulos agregados:

Multiplexer3to1, usado para los mux de FwrA y FwrB

```
module Multiplexer3to1
#(
    parameter NBits=32
)
(
    input [1:0] Selector,
    input [NBits-1:0] MUX_Data0,
    input [NBits-1:0] MUX_Data1,
    input [NBits-1:0] MUX_Data2,

    output reg [NBits-1:0] MUX_Output

);

    always@(Selector,MUX_Data1,MUX_Data0,MUX_Data2) begin
       case (Selector)
          0: MUX_Output = MUX_Data0;
          1: MUX_Output = MUX_Data1;
          2: MUX_Output = MUX_Data2;

          default: MUX_Output =MUX_Data0;
       endcase


    end
```

PipeRegister, Notar que se agregó el input flush

```
module PipeRegister
#(
    parameter N=64
)
(
    input clk,
    input reset,
    input enable,
    input flush,
    input  [N-1:0] DataInput,


    output reg [N-1:0] DataOutput
);

always@(negedge reset or posedge clk) begin
    if(reset==0)
       DataOutput <= 0;
    else if (flush == 1)
       DataOutput <= 0;
    else  if(enable==1)
          DataOutput<=DataInput;
end

endmodule
```

HazarUnit, se ponen como inputs los wire selectores para cada tipo de salto y dependiendo de estos y los valores Rt y Rs se decide si se genera un Stall o no

```
module HazardUnit(
    input IDEX_MemRead,
    input jal,
    input jump,
    input branch,
    input jr,
    input [4:0] IDEX_Rt,
    input [4:0] IFID_Rs,
    input [4:0] IFID_Rt,

    output reg IFID_write,
    output reg stall_sel,
    output flush
);

 always @ (*)
begin


      //ForwardA
      if (IDEX_MemRead && ((IDEX_Rt == IFID_Rs) || (IDEX_Rt == IFID_Rt)))
         begin
         stall_sel <= 1'b1;
         IFID_write <= 1'b0;

         end
      else
         begin
         stall_sel <= 1'b0;
         IFID_write <= 1'b1;
         end

end

 assign flush = (jump || jal || branch || jr);

 endmodule
```

ForwardingUnit, el código está sacado de las diapositivas

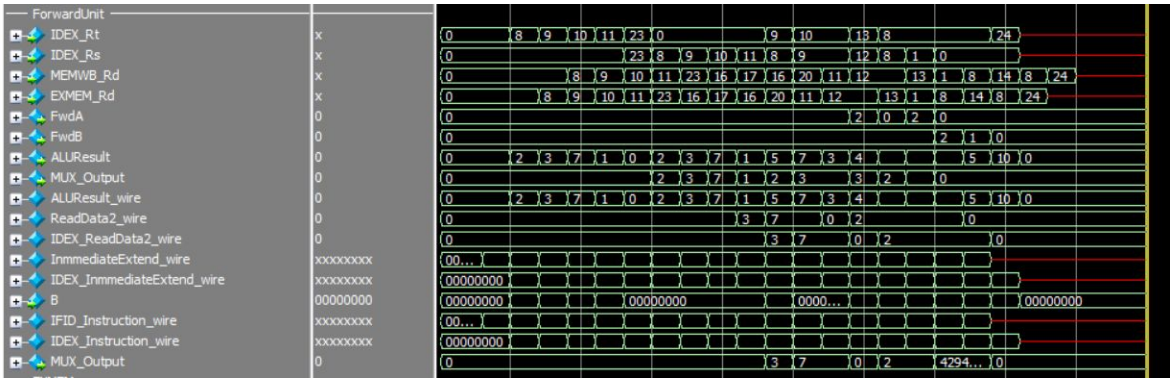
```
module ForwardUnit(
    input EXMEM_RegWrite,
    input MEMWB_RegWrite,
    input [4:0] IDEX_Rt,
    input [4:0] IDEX_Rs,
    input [4:0] MEMWB_Rd,
    input [4:0] EXMEM_Rd,
    output reg [1:0] FwdA,
    output reg [1:0] FwdB
);

    reg [1:0] Fwd_A=0;
    reg [1:0] Fwd_B=0;

 always @ (*)
begin
      //ForwardA
      if (EXMEM_RegWrite && (EXMEM_Rd != 0) && (EXMEM_Rd == IDEX_Rs))
         FwdA <= 2'b10;

      else  if (MEMWB_RegWrite && (MEMWB_Rd != 0) && (EXMEM_Rd != IDEX_Rs) && (MEMWB_Rd == IDEX_Rs))
         FwdA <= 2'b01;

      else
         FwdA <= 2'b00;

         //ForwardB
      if (MEMWB_RegWrite && (MEMWB_Rd != 0) && (EXMEM_Rd != IDEX_Rt) && (MEMWB_Rd == IDEX_Rt))
         FwdB <= 2'b01;

      else if (EXMEM_RegWrite && (EXMEM_Rd != 0) && (EXMEM_Rd == IDEX_Rt))
         FwdB <= 2'b10;

      else
         FwdB <= 2'b00;

 end

 endmodule
```

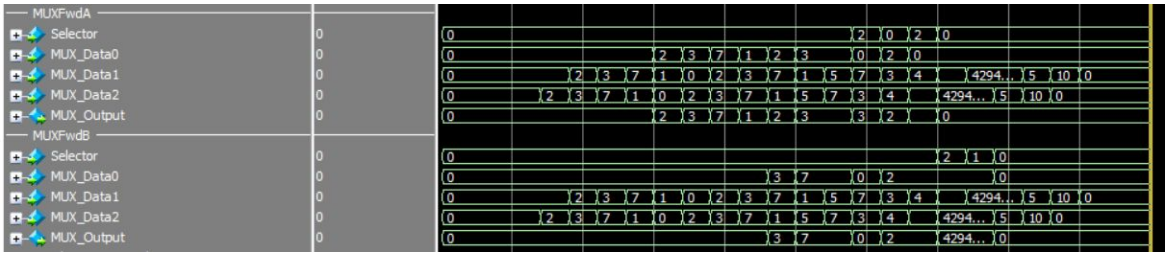
Para probar los Hazards y los Forwarding Units se utilizó el siguiente programa:

```
.text
# Initializations
addi $t0,$zero,2
addi $t1,$zero,3
addi $t2,$zero,7
addi $t3,$zero,1
addi $s7, $s7, 0

# Moving values
add $s0,$t0,$zero
add $s1,$t1,$zero
add $s0,$t2,$zero
add $s4,$t3,$zero

# Basic operations
add $t3,$t0,$t1
or  $t4,$t1,$t2
and $t4,$t1,$t2
addi $t5,$t4,1

# 2's complement of register $t0
nor $at,$t0,$t0
addi $t0,$at,1

# Move register $t0 to $t6
add $t6,$zero,$t0

# Swap registers $t1 and $t2
addi $t0,$zero,5
addi  $t8,$zero, 10
```

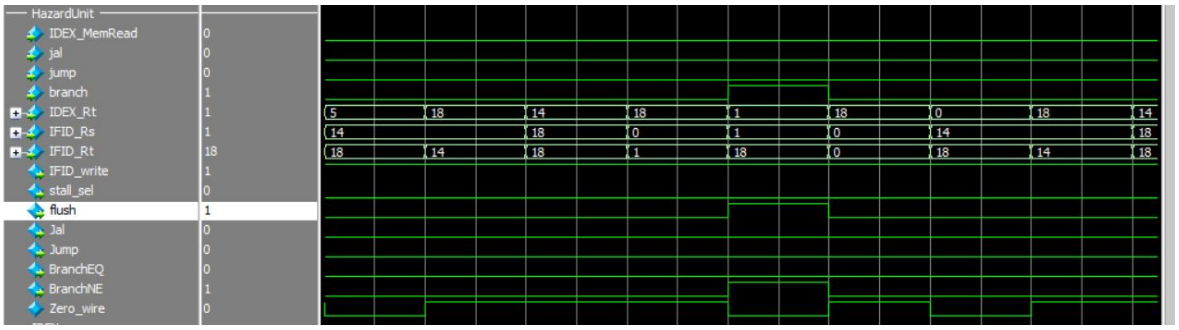
Podemos observar que en Moving Values, Basic Operations, Complement y Move utilizan el forwarding unit.

Forwarding Unit





Hazard Unit



Conclusiones

Lilia Lobato

Es una práctica sumamente complicada, no por los temas o los conceptos, sino por la facilidad con la que te puedes perder entre tantos cables. Es interesante pensar cómo se logran los pipeline de más de 5 etapas o con procesos en paralelo, la cantidad de cables a utilizar se multiplica.

Una mejora sobre mi proyecto sería ir guardando las versiones y trabajando sobre copias porque al momento de agregar al pipeline los forwarding units y el hazard detection unit el código de Hanoi tiene un error.