Datapath

1 Integrantes

Cristhoper Heredia
Alvaro Aguirre begin document

2 MODULO DATAPATH

En el modulo de DATAPATH llamo a todos los elementos que el contiene en orden segun el datapth que hemos diseñado.

```
‘include"pc_4.v"
‘include"Adder.v"
‘include"Alu.v"
‘include"Alu_control.v"
‘include"and.v"
‘include"Control.v"
‘include"Data_Memory.v"
‘include"InstructionMemory.v"
‘include"mux_2_1_5bits.v"
‘include"mux2_1.v"
‘include"ProgramCounter.v"
‘include"Register_file.v"
‘include"Shift_left_Branch.v"
‘include"Shift_left_Jump.v"
‘include"SignExtend.v"
module DATAPATH(clk,reset);
input clk;//funca
input reset;
//PROGRAM COUNTER:
wire [31:0] address_final;//funca
wire [31:0] Out_PC;//Out_PC=salida del program counter
//INSTRUCTION MEMORY:
wire [31:0] Instruction;//la instruccion
//CONTROL:
wire RegDst,Jump,MemtoReg,ALUsrc,RegWrite,Branch;//del Control
wire [1:0] MemRead, MemWrite;//del Control
```

```
wire [3:0] ALUOP;
//REGISTER FILE:
wire [4:0] writereg;
wire [4:0] readreg1,readreg2;
wire [31:0] read_data1,read_data2;
//SIGN EXTEND:
wire [31:0] sign_extended;
//ALU CONTROL:
wire [3:0] alucontrol;
//ALU
wire [31:0] mux_alu;
wire [31:0] alu_result;
wire zero;//verifica si es branch
//shift jump
wire [31:0] Jump_address;
//shift Branch
wire [31:0] shift_left_branch;
wire [31:0] branch_pc;
wire [31:0] mux_branch_out;
//DATA MEMORY:
wire [31:0] writedata;
wire zero_to_mux;
wire [31:0] DM_out;
wire [31:0] DM_mux;
//FETCH:
PC #(32)call_pc(.clk(clk),.reset(reset),.d(address_final),.q(Out_PC));
InstructionMemory call_IM(.pc(Out_PC),.out(Instruction));
adder_pc call_adder_pc(.pc(Out_PC),.pc_add(address_final));
//DECODE:
Control call_Control(.clk(clk),.Instruction(Instruction[31:26]),.RegDst(RegDst),
```

```
.Jump(Jump),.Branch(Branch),.MemRead(MemRead),
.MemtoReg(MemtoReg),.ALUOp(ALUOP),.MemWrite(MemWrite),.ALUSrc(ALUsrc),
.RegWrite(RegWrite));//llamando al control
mux2_1_5 call_mux2_1_5bits(.a(Instruction[20:16]),.b(Instruction[15:11]),
.sel(RegDst),.out(writereg));
Register_File call_RF(.clk(clk),
.readreg1(Instruction[25:21]),.readreg2(Instruction[20:16]),.writereg(writereg),
.writedata(writedata),.read_data1(read_data1),.read_data2(read_data2),
.regwrite(RegWrite));
SignExtend call_Signextend(.a(Instruction[15:0]),.b(sign_extended));
mux2_1 mux_antes_del_alu(.a(sign_extended),.b(read_data2),.sel(1'b1),
.out(mux_alu));
ALU_Control call_alu_control(.aluOp(ALUOP),.func(Instruction[5:0]),
.out(alucontrol));
//EXECUTE
ALU call_ALU(.entr1(read_data1),.entr2(mux_alu),.alu_ctrl(alucontrol),
.alu_result(alu_result),.zero(zero));
//////////////////
Shift_Left_Jump call_shift_jump(.imm(Instruction[25:0]),.PC(Out_PC[31:28])
,.jump(Jump_address));
Shift_Left_Branch call_shift_branch(.imm(sign_extended),
.branch_address(shift_left_branch));
//el sign_extend es el unsigned que sale del sign_extend
Adder call_adder(.a(Out_PC),.b(shift_left_branch),.y(branch_pc));
And call_and(.a(Branch),.b(zero),.out(zero_to_mux));//Corre perfecto
mux2_1 call_mux2_1_branch(.a(Out_PC),.b(branch_pc),.sel(zero_to_mux),
.out(mux_branch_out));
Data_Memory call_data_memory(.clk(clk),.address(alu_result),
.memwrite(MemWrite),.writedata(read_data2),.read_data(DM_out),
.memread(MemRead));
```

```
mux2_1 call_mux_data_memory(.a(DM_out),.b(alu_result),.sel(MemtoReg),
.out(DM_mux));
always @ (posedge clk)
begin
#2;$display("%d,%b,%d,%b,%b,%b,%b,%b,%d",Out_PC,Instruction,
Instruction[25:21],read_data1,mux_alu,alu_result,zero,zero_to_mux,branch_pc);
end

endmodule
```

3 Program Counter

El Program Counter tiene como inputs clock y entrada, que tiene 32 bits..

```
module PC #(parameter WIDTH=8)(input clk, reset,input [WIDTH-1:0] d,
output reg [WIDTH-1:0] q);//Program Counter d=next y q=actual
always @ (posedge clk, posedge reset)
if (reset)
q <= 0;
else
q <= d;
endmodule
```

4 PC ADDER

El input para este módulo es pc de 32 bits y su output es la dirección

de la siguiente instrucción (pc_add) de 32 bits. Luego de declarar ambas variables usamos un always para sumarle 4 a la dirección actual y poder pasar a la siguiente en el próximo clock cycle.

```
module adder_pc(pc,pc_add);
input [31:0] pc;
output reg [31:0] pc_add;
always @(*)
begin
    pc_add <= pc + 4;
end
endmodule
```

5 INSTRUCTION MEMORY

Los inputs son clock, pc de 32 bits y el output out de 32 bits. Luego creamos un reg "instrucciones" de 96 filas y 8 columnas en donde agrupamos todas las instrucciones que se realizarán. Finalmente, el output será la instrucción de 32 bits recogida byte por byte (4 filas). Por eso obtenemos instrucciones[pc] hasta instrucciones[pc+3] que coge las 4 filas deseadas. pc va aumentando gracias al ADDER antes explicado.

```
module InstructionMemory(pc,out);//solo recibe
input [31:0] pc;
output reg [31:0] out;
reg [7:0] instrucciones [0:91];
integer i;
initial begin
    $readmemb("instrucciones.txt",instrucciones);
```

```
end
always@(*)
begin
    out <= {instrucciones[pc],instrucciones[pc+1],instrucciones[pc+2],
    instrucciones[pc+3]};
end
endmodule
```

6 CONTROL

Los inputs del módulo son los últimos 6 bits de la instrucción e indican qué tipo de instrucción es: R, I o J. Los inputs son las señales que el control habilitará para realizar la instrucción. Jump tiene 2 bits porque hay tres tipos de instrucciones J y sirven para diferenciar cuál tipo es la instrucción. Lo mismo pasa con MemRead. Este cable se prende cuando se usa una instrucción load. Como hay tres tipos de loads se usan 2 bits. Lo mismo pasa con las tipo store. ALUOp tiene 4 bits ya que así sabremos si es R-type o I-type, y los bits le indicarán al ALUOp qué instrucción debe ejecutar el ALU.

```
 module Control(clk,Instruction,RegDst,Jump,Branch,MemRead,MemtoReg,ALUOp
,MemWrite,ALUSrc,RegWrite);
input clk;
input [5:0] Instruction;
output reg RegDst,Branch,Jump,MemtoReg,ALUSrc,RegWrite;
output reg [1:0] MemRead, MemWrite;
output reg [3:0] ALUOp;
```

```
always @(*)
begin
if(Instruction==6'b000000)//R-Type
  begin
  RegDst =1'b1;
  Jump = 2'b00;
  Branch = 1'b0;
  MemRead = 2'b00;
  MemtoReg = 1'b0;
  ALUOp = 4'b0000;
  MemWrite = 2'b00;
  ALUSrc = 1'b0;
  RegWrite = 1'b1;
  end
else if(Instruction==6'b100011)//lw
  begin
    RegDst = 1'b1;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b01;
    MemtoReg = 1'b1;
    ALUOp = 4'b0100;
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
    RegWrite = 1'b1;
  end
else if(Instruction==6'b101011)//sw
  begin
    RegDst = 1'bx;
    Jump = 2'b00;
```

```
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'bx;
    ALUOp = 4'b0100;
    MemWrite = 2'b01;
    ALUSrc = 1'b1;
    RegWrite = 1'b0;
  end
else if(Instruction==6'b100000)//lb
  begin
    RegDst = 1'b1;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b10;
    MemtoReg = 1'b1;
    ALUOp = 4'b0100;
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
    RegWrite = 1'b1;
  end
else if(Instruction==6'b101000)//sb
  begin
    RegDst = 1'bx;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'bx;
    ALUOp = 4'b0100;
    MemWrite = 2'b10;
    ALUSrc = 1'b1;
```

```
    RegWrite = 1'b0;
  end
else if(Instruction==6'b100001)//lh
  begin
    RegDst = 1'b1;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b11;
    MemtoReg = 1'b1;
    ALUOp = 4'b0100;
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
    RegWrite = 1'b1;
  end
else if(Instruction==6'b101001)//sh
  begin
    RegDst = 1'bx;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'bx;
    ALUOp = 4'b0100;
    MemWrite = 2'b11;
    ALUSrc = 1'b1;
    RegWrite = 1'b0;
  end
  else if(Instruction==6'b001010)//slti
  begin
    RegDst = 1'b0;
    Jump = 2'b00;
```

```
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'b0;
    ALUOp = 4'b0010;
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
    RegWrite = 1'b1;
  end
else if(Instruction==6'b001111)//lui
  begin
    RegDst = 1'b0;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'b0;
    ALUOp = 4'b0001; //REVISAR
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
    RegWrite = 1'b1;
  end
else if(Instruction==6'b001100)//andi
  begin
    RegDst = 1'b0;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'b0;
    ALUOp = 4'b1100;
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
```

```
    RegWrite = 1'b1;
  end
else if(Instruction==6'b001101)//ori
  begin
    RegDst = 1'b0;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'b0;
    ALUOp = 4'b1110;
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
    RegWrite = 1'b1;
  end
else if(Instruction==6'b001000)//andi
  begin
    RegDst = 1'b0;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'b0;
    ALUOp = 4'b0100;
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
    RegWrite = 1'b1;
  end
else if(Instruction==6'b000100)//beq
  begin
    RegDst = 1'b0;
    Jump = 2'b00;
```

```
    Branch = 1'b1;
    MemRead = 2'b00;
    MemtoReg = 1'b0;
    ALUOp = 4'b0101;
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
    RegWrite = 1'b1;
  end
else if(Instruction==6'b000101)//bneq
  begin
    RegDst = 1'b0;
    Jump = 2'b00;
    Branch = 1'b1;
    MemRead = 2'b00;
    MemtoReg = 1'b0;
    ALUOp = 4'b0111;
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
    RegWrite = 1'b1;
  end
else if(Instruction==6'b000111)//bgez
  begin
    RegDst = 1'b0;
    Jump = 2'b00;
    Branch = 1'b1;
    MemRead = 2'b00;
    MemtoReg = 1'b0;
    ALUOp = 4'b1111;
    MemWrite = 2'b00;
    ALUSrc = 1'b1;
```

```
    RegWrite = 1'b1;
  end
else if(Instruction==6'b000010)//jump
  begin
    RegDst = 1'bx;
    Jump = 2'b01;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'bx;
    ALUOp = 4'bxxxx;
    MemWrite = 2'b00;
    ALUSrc = 1'bx;
    RegWrite = 1'b0;
  end
else if(Instruction==6'b000011)//jal
  begin
    RegDst = 1'bx;
    Jump = 2'b10;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'bx;
    ALUOp = 4'bxxxx;
    MemWrite = 2'b00;
    ALUSrc = 1'bx;
    RegWrite = 1'b0;
  end
else if(Instruction==6'b001000)//jr
  begin
    RegDst = 1'bx;
    Jump = 2'b11;
```

```
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'bx;
    ALUOp = 4'bxxxx;
    MemWrite = 2'b00;
    ALUSrc = 1'bx;
    RegWrite = 1'b0;
  end
end
endmodule
```

7 REGISTER FILE

El módulo Register_File tiene como input 2 sources (readreg1, readreg2), un destination(writereg) y un writedata. Como output tiene read_data 1 y 2. Creamos un reg de 32 filas y 32 columnas llamado register_set. Ahí ponemos todas las variables y sus valores respectivos. Luego creamos un always y inicializamos los outputs usando los bits de los inputs readreg 1 y 2. Finalmente, en el negedge clk se implementa la etapa de write back siempre y cuando este cable se prenda.

```
module Register_File(clk, readreg1, readreg2, writereg, writedata,
read_data1, read_data2, regwrite);
input clk;
input [4:0] readreg1, readreg2, writereg; // 2 sources y 1 destination
input [31:0] writedata; // resultado (writeback)
input regwrite; // input de Control
output reg [31:0] read_data1, read_data2;
reg [31:0]reg_set[0:31]; // tabla de las variables
```

```
initial begin
$readmemb("register_set.txt", reg_set);
end
//se lee en posedge
always @(posedge clk)
begin
  read_data1 <= reg_set[readreg1];
  read_data2 <= reg_set[readreg2];
end
/*
assign read_data1 = reg_set[readreg1];
assign read_data2 = reg_set[readreg2];
*/

//se escribe en negedge
/*
always @ (negedge clk)
  if (regwrite) reg_set[writereg] <= writedata;
*/

endmodule
```

8 ALU

Los inputs del ALU son las entradas a las cuales se les ejecuta alguna operación. alu_ctrl le avisa cuál de esta acción se realiza, alu_result es el output con el resultado y zero se usa para las instrucciones de tipo

branch. Dentro del always está implementado el código del ALU que realiza instrucciones lógicas y aritméticas.

```
module ALU(entr1,entr2,alu_ctrl,alu_result,zero);
//entradas al alu
input [31:0] entr1, entr2;
//orden de la instruccion
input [3:0] alu_ctrl;

output reg zero;
initial begin
zero = 1'b0;
end

output reg [31:0] alu_result;
/*
ISA-1:
add,addi,sub,and,andi,nor,ori,or,slti,slt.
ISA-2:

*/

always@(*)
begin
  case(alu_ctrl)
    //ADD
    4'b0000:
     alu_result= entr1+entr2;
    //SUB
    4'b0001:
```

```
 alu_result = entr1 - entr2;
//AND
4'b0010:
 alu_result = entr1 & entr2;
//NOR
4'b0011:
    alu_result = ~(entr1 | entr2);
//OR
4'b0100:
    alu_result = entr1 | entr2;
//SLT
4'b0101:
begin
if(entr1>entr2)
    alu_result = 1'b1;
else
    alu_result = 1'b0;
end
//BEQ
4'b0110:
begin
if(entr1==entr2)
    zero = 1'b1;
else
    zero = 1'b0;
end
//BNQ
4'b0111:
begin
if(entr1==entr2)
```

```
      zero = 1'b0;
    else
      zero = 1'b1;
    end
    //BGEZ
    4'b1111:
    begin
   if(entr1>=entr2)
      zero = 1'b1;
    else
      zero = 1'b0;
    end

  endcase
end
endmodule
```

9 ALU CONTROL

Los inputs son ALUOp, func (en caso sea un R-type y le avisa al ALU qué operación ejecutar) y el output out. En caso el ALUOp sea 00, entonces el ALU CONTROL tendrá como output la operación que ejecutará el ALU dependiendo de \func". Si recibe 01, ejecutará suma, ya que este tipo de instrucción es store o load y lo que hacen es sumar el offset con el base address. Si fuera 10 se ejecutaría la resta, ya que en este caso se pide la instrucción slti, subi o branches.

```
 module ALU_Control(aluOp,func,out);
input wire [3:0] aluOp;
input wire [5:0] func;
output reg [3:0] out;
```

```
always @(*)
begin
case (aluOp)
 4'b0000: //R type
 begin
 case (func)
 6'b100000: out <= 4'b0000; //ADD
 6'b100010: out <= 4'b0001; //SUB
    6'b100100: out <= 4'b0010; //AND
 6'b100111: out <= 4'b0011; //NOR
    6'b100101: out <= 4'b0100; //OR
 6'b101010: out <= 4'b0101; //SLT
 endcase
    end


    4'b0100: // I-type (addi, load y store)
 out <= 4'b0000; //ADD
    4'b1010: // I-type (subi)
 out <= 4'b0001; //SUB
 4'b0010: // I-type (slti)
 out <= 4'b0101; // SLT
 4'b1100: // I-type (andi)
 out <= 4'b0010; // AND
 4'b1110: // I-type (ori)
 out <= 4'b0100; // OR
 4'b1111: // I-type (bgez)
 out <= 4'b1111;
 4'b0101: // I-type (beq)
```

```
 out <= 4'b0110;
 4'b0111: // I-type (bneq)
 out <= 4'b0111;


endcase
end

endmodule
```

10 DATA MEMORY

Para este módulo creamos un array en donde están los valores que serán storeados" y loadeados". El input memread y memwrite indican si la instrucción es store o load respectivamente. Tienen dos bits porque hay tres tipos de store y load instructions. Dentro del always, en caso sea un loadword, se va llenando el read_data byte por byte empezando por array[address] hasta array[address+3]. En caso sea loadbyte, los 8 bits menos significantes se llenan con el valor pedido, los demás con 1's. Para loadhalfword hacemos lo mismo, pero para los 16 primeros bits y los demás 1's. Lo mismo es para storeword, storehalfword y storebyte.

```
module Data_Memory(clk, address, memwrite, writedata, read_data, memread);
input clk;
input [31:0] address;
input [1:0] memwrite;
input [1:0] memread;
input [31:0] writedata;
output [31:0] read_data;
```

```
reg[7:0]array[0:39];

wire clk;
wire [31:0] address;
wire [31:0] writedata;
wire [1:0] memread;
wire [1:0] memwrite;
reg [31:0] read_data;

initial
  begin
    $readmemb("array.txt", array);
  end

always @(*) //REVISAR
begin
  if (memread == 2'b01) //lw
  begin
 read_data[31:24] <= array[address];
 read_data[23:16] <= array[address+1];
 read_data[15:8]  <= array[address+2];
 read_data[7:0]  <=  array[address+3];
  end
  if(memread == 2'b10) //lb
  begin
     read_data[7:0] <= array[address+3];
 read_data[31:8] <= 24'hFFFFFF;
  end
  if(memread == 2'b11) //lh
  begin
```

```
     read_data[7:0] <= array[address+3];
 read_data[15:8] <= array[address+2];
 read_data[31:16] <= 16'h0000;
  end
end

always@(negedge clk) //REVISAR
begin
  if (memwrite == 2'b01) //sw
  begin
 array[address] <= read_data[31:24];
 array[address+1] <= read_data[23:16];
 array[address+2] <= read_data[15:8];
 array[address+3] <= read_data[7:0];
  end
  if(memwrite == 2'b10) //sb
  begin
    array[address+3] <= read_data[7:0];
 array[address+2] <= 8'hFF;
 array[address+1] <= 8'hFF;
 array[address] <= 8'hFF;
  end
  if(memwrite == 2'b11) //sh
  begin
    array[address+3] <= read_data[7:0];
 array[address+2] <= read_data[15:8];
 array[address+1] <= 8'h00;
 array[address] <= 8'h00;
  end
end
```

```
endmodule
```

11 AND

Este módulo sirve para una branch instruction y el output irá como selector al mux.

```
module And(a,b,out);
input wire a,b;
output out;
assign out = a & b;
endmodule
```

12 SHIFT LEFT BRANCH

Este módulo usa la función sll para operar con el input imm de 32 bits.

```
module Shift_Left_Branch(imm,branch_address);
input [31:0] imm;
output reg [31:0] branch_address;
always@(*)
begin
  branch_address <= imm << 2;
end
endmodule
```

13 SHIFT LEFT JUMP

Este módulo recibe un input de 26 bits (imm) y otro de 4 bits (PC) para botar un output jump" de 32 bits, que es la dirección de la instrucción a la que se quiere saltar". Dentro del always concatenamos los bits para formar el output jump de 32 bits.

```
module Shift_Left_Jump(imm, PC, jump);
input[25:0] imm;
input[3:0] PC;
output reg[31:0] jump;
reg[1:0] shift;

initial begin
shift = 2'b00;
end

always@(*)
begin
  jump = {{PC},{imm},{shift}};
end

endmodule
```

14 SIGN EXTEND

Este módulo concatena los 16 bits que se recibe como input "a" y se añaden otros 16 para así crear el output \b" de 32 bits.

```
module SignExtend(a,b);
input[15:0] a;
output reg [31:0] b;
always@(*)
begin
 b = {{16{a[15]}},{a}};
end
endmodule
```

Para la realizacion del pipeline incluimos modulos entre los stages para poder asi enviarle todas las señales al data hazard.

15 Fetch-Decode

```
module F_D (input clk,input rst,input STALL,input [31:0] PCIn,input [31:0] instructionIn

always @ (posedge clk) begin
  if (rst)
  begin
    PC <= 0;
    instruction <= 0;
  end
  else
  begin
    if (STALL)
    begin
    //no pasa nada
    end
    else
```

```
        begin
          instruction <= instructionIn;
          PC <= PCIn;
        end
      end
    end
    endmodule
```

16 DECODE-EXECUTE

```
module D_EX(input clk,rst,input [4:0] src_1,input [4:0] src_2,input [4:0] dest_in,input [31:
input [31:0] Reg2_in,input [31:0] out_mux_reg_2_o_imm_in,input Branch_in,
input Jump_in,
input MemtoReg_in,input [1:0] MemRead_in,input [1:0] MemWrite_in,
input [3:0] alu_control_out_in,input [31:0] Jump_address_in,
input [31:0] shift_left_branch_in,output reg [4:0] src_1_out,output reg [4:0] src_2_out,outp
output reg [31:0] Reg1_out,output reg [31:0] Reg2_out,
output reg [31:0] out_mux_reg_2_o_imm_out,output reg Branch_out,
output reg Jump_out,
output reg MemtoReg_out,output reg [1:0] MemRead_out,
output reg [1:0] MemWrite_out,
output reg [3:0] alu_control_out_out,output reg [31:0] Jump_address_out,
output reg [31:0] shift_left_branch_out);


always @(posedge clk)
begin
    if (rst)
        begin
```

```
    src_1_out<=0;
    src_2_out<=0;
    Reg1_out<=0;
    Reg2_out<=0;
    dest_out<=0;
    out_mux_reg_2_o_imm_out<=0;
    Branch_out<=0;
    Jump_out<=0;
    MemtoReg_out<=0;
    MemRead_out<=0;
    MemWrite_out<=0;
    alu_control_out_out<=0;
    Jump_address_out<=0;
    shift_left_branch_out<=0;
    end
else
    begin
    src_1_out<=src_1;
    src_2_out<=src_2;
    Reg1_out<=Reg1_in;
    Reg2_out<=Reg2_in;
    dest_out<=dest_in;
    out_mux_reg_2_o_imm_out<=alu_control_out_in;
    Branch_out<=Branch_in;
    Jump_out<=Jump_in;
    MemtoReg_out<=MemtoReg_in;
    MemRead_out<=MemRead_in;
    MemWrite_out<=MemWrite_in;
    alu_control_out_out<=alu_control_out_in;
    Jump_address_out<=Jump_address_in;
```

```
        shift_left_branch_out<=shift_left_branch_in;
        end
end
endmodule
```

17 EXECUTE-MEMORY

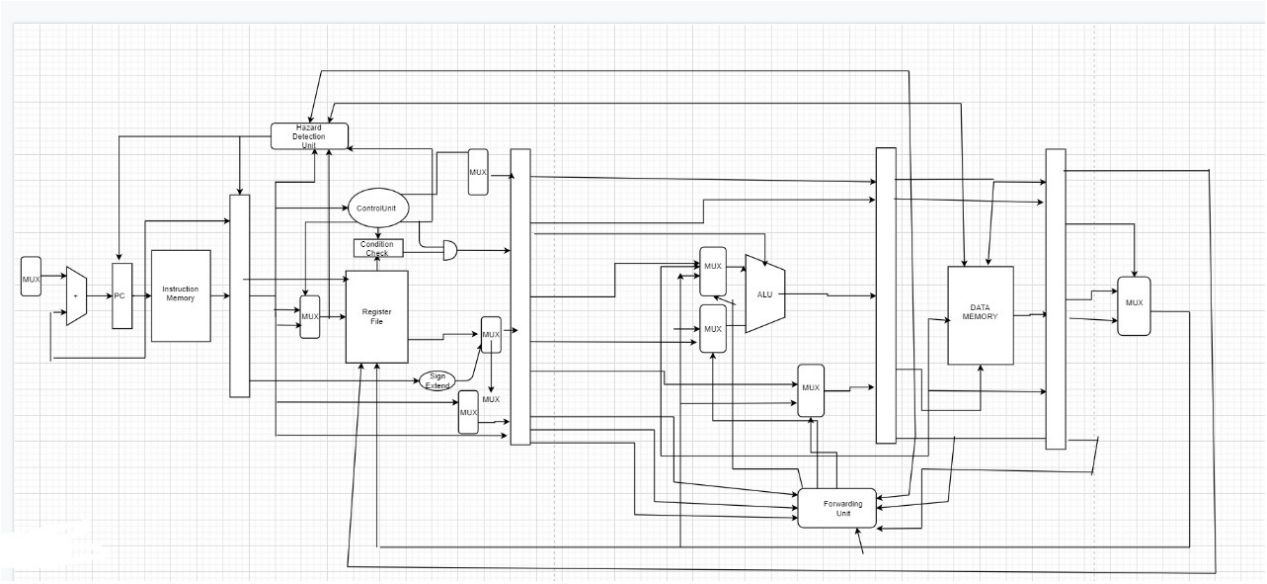
```
    module EX_MEM(input clk,input rst,input [4:0] dest,input [1:0] memwrite,input memtoreg,i
output reg [4:0] dest_out,output reg [1:0] memwrite_out,output reg [1:0] memread_out,output

always @(posedge clk)
begin
if (rst) begin
    dest_out<=0;
    memwrite_out<=0;
    memread_out<=0;
    memtoreg_out<=0;
end
else
begin
    dest_out<=dest;
    memwrite_out<=memwrite;
    memread_out<=memread;
    memtoreg_out<=memtoreg;
end

end

endmodule
```

18 Data hazard y Forwarding

DATA HAZARD: En este modulo verificamos las dependencias porque comparamos los sources con el dest y verificamos si existe la dependencia RAW entonces si existe se prende el hazard detected y dependiendo de nuestras instrucciones ejecutara o forwarding o stall.

```
module hazard_detection(input forward_EN,input alu_src,input [4:0] src1_ID,input [4:0] src2_
input mem_to_reg_d_e,input [4:0] dest_e_m,input mem_to_reg_e_m,output hazard_detected);

wire src2_is_valid, exe_hazard, mem_hazard, hazard;

assign src2_is_valid = alu_src;//si es inmediate o no

assign exe_hazard = mem_to_reg_d_e && (src1_ID == dest_d_e || (src2_is_valid && src2_ID == d
assign mem_hazard = mem_to_reg_e_m && (src1_ID == dest_e_m || (src2_is_valid && src2_ID == d

assign hazard = (exe_hazard || mem_hazard);//si alguno esta prendido hay dependencia


assign hazard_detected = (forward_EN==1'b0) ? hazard : hazard;

endmodule
```

DISEÑO :


Hazard Detection Unit
MUX
ControlUnit
Condition Check
MUX
PC
Instruction Memory
MUX
Register File
MUX
Sign Extend
MUX
MUX
MUX
MUX
ALU
MUX
DATA MEMORY
MUX
Forwarding Unit