Datapath

1 Integrantes

Cristhoper Heredia
Alvaro Aguirre

2 Program Counter

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

```
module PC(clk,entrada,salida);
input clk;
input [31:0] entrada;
output reg [31:0] salida;
reg [31:0] contador;

initial
begin
contador = 32'h00000000;
end

always @(posedge clk)
begin
salida <= entrada;
end
endmodule
```

3 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
```

4 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(clk,pc,out);
input clk;
input [31:0] pc;
output reg [31:0] out;
reg [7:0] instrucciones [0:59];
initial begin
$readmemb("instrucciones.txt",IM);
end
always@(pc)
begin
  out <= {instrucciones[pc],instrucciones[pc+1],instrucciones[pc+2]
  ,instrucciones[pc+3]};
end
endmodule
```

5 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 2 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(Instruction,RegDst,Jump,Branch,MemRead,MemtoReg,ALUOp,MemWrite,
```

```
ALUSrc,RegWrite);

input [5:0] Instruction;
output reg RegDst,Branch,Jump,MemtoReg,ALUSrc,RegWrite;
output reg [1:0] ALUOp, MemRead, MemWrite;

always @(*)
begin
if(Instruction==6'b000000)//R-Type
  begin
  RegDst =1;
  Jump = 2'b00;
  Branch = 1'b0;
  MemRead = 2'b00;
  MemtoReg = 0;
  ALUOp = 2'b00;
  MemWrite = 2'b00;
  ALUSrc = 0;
  RegWrite = 1;
  end
else
  begin
  case(Instruction)
  6'b100011: //lw
  begin
    RegDst = 1;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b01;
    MemtoReg = 1;
    ALUOp = 2'b01;
    MemWrite = 2'b00;
    ALUSrc = 1;
    RegWrite = 1;
  end
  6'b101011: //sw
  begin
    RegDst = 1'bx;
    Jump = 2'b00;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'bx;
    ALUOp = 2'b01;
    MemWrite = 2'b01;
    ALUSrc = 1;
    RegWrite = 0;
```

```
end

6'b100000: //lb
begin
  RegDst = 1;
  Jump = 2'b00;
  Branch = 1'b0;
  MemRead = 2'b10;
  MemtoReg = 1;
  ALUOp = 2'b01;
  MemWrite = 2'b00;
  ALUSrc = 1;
  RegWrite = 1;
end

6'b101000: //sb
begin
  RegDst = 1'bx;
  Jump = 2'b00;
  Branch = 1'b0;
  MemRead = 2'b00;
  MemtoReg = 1'bx;
  ALUOp = 2'b01;
  MemWrite = 2'b10;
  ALUSrc = 1;
  RegWrite = 0;
end

6'b100001: //lh
begin
  RegDst = 1;
  Jump = 2'b00;
  Branch = 1'b0;
  MemRead = 2'b11;
  MemtoReg = 1;
  ALUOp = 2'b01;
  MemWrite = 2'b00;
  ALUSrc = 1;
  RegWrite = 1;
end

6'b101001: //sh
begin
  RegDst = 1'bx;
  Jump = 2'b00;
  Branch = 1'b0;
```

```
  MemRead = 2'b00;
  MemtoReg = 1'bx;
  ALUOp = 2'b01;
  MemWrite = 2'b11;
  ALUSrc = 1;
  RegWrite = 0;
end

6'b001111: //lui
begin
  RegDst = 0;
  Jump = 2'b00;
  Branch = 1'b0;
  MemRead = 2'b00;
  MemtoReg = 0;
  ALUOp = 2'b01; //REVISAR
  MemWrite = 2'b00;
  ALUSrc = 1;
  RegWrite = 1;
end

6'b001111: //andi
begin
  RegDst = 0;
  Jump = 2'b00;
  Branch = 1'b0;
  MemRead = 2'b00;
  MemtoReg = 0;
  ALUOp = 2'b01; //REVISAR
  MemWrite = 2'b00;
  ALUSrc = 1;
  RegWrite = 1;
end

6'b001111: //ori
begin
  RegDst = 0;
  Jump = 2'b00;
  Branch = 1'b0;
  MemRead = 2'b00;
  MemtoReg = 0;
  ALUOp = 2'b01; //REVISAR
  MemWrite = 2'b00;
  ALUSrc = 1;
  RegWrite = 1;
end
```

```
6'b001111: //beq
begin
  RegDst = 0;
  Jump = 2'b00;
  Branch = 1'b1;
  MemRead = 2'b00;
  MemtoReg = 0;
  ALUOp = 2'b01;
  MemWrite = 2'b00;
  ALUSrc = 1;
  RegWrite = 1;
end

6'b001111: //bneq
begin
  RegDst = 0;
  Jump = 2'b00;
  Branch = 1'b1;
  MemRead = 2'b00;
  MemtoReg = 0;
  ALUOp = 2'b01;
  MemWrite = 2'b00;
  ALUSrc = 1;
  RegWrite = 1;
end

6'b001111: //bgez
begin
  RegDst = 0;
  Jump = 2'b00;
  Branch = 1'b1;
  MemRead = 2'b00;
  MemtoReg = 0;
  ALUOp = 2'b01;
  MemWrite = 2'b00;
  ALUSrc = 1;
  RegWrite = 1;
end

6'b000010: //jump
begin
  RegDst = 1'bx;
  Jump = 2'b01;
  Branch = 1'b0;
  MemRead = 2'b00;
```

```
    MemtoReg = 1'bx;
    ALUOp = 2'bxx;
    MemWrite = 2'b00;
    ALUSrc = 1'bx;
    RegWrite = 0;
  end

  6'b000011: //jal
  begin
    RegDst = 1'bx;
    Jump = 2'b10;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'bx;
    ALUOp = 2'bxx;
    MemWrite = 2'b00;
    ALUSrc = 1'bx;
    RegWrite = 0;
  end

  6'b001000: //jr
  begin
    RegDst = 1'bx;
    Jump = 2'b11;
    Branch = 1'b0;
    MemRead = 2'b00;
    MemtoReg = 1'bx;
    ALUOp = 2'bxx;
    MemWrite = 2'b00;
    ALUSrc = 1'bx;
    RegWrite = 0;
  end
  endcase
  end
end
endmodule
```

6 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;

input [31:0] writedata;
input regwrite;

output reg [31:0] read_data1, read_data2;

reg [31:0]reg_set[0:31];

initial begin
$readmemb("register_set.txt", reg_set);
end
    always @(posedge clk)
begin
    read_data1 <= reg_set[readreg1];
    read_data2 <= reg_set[readreg2];
end

always @(negedge clk)
begin
  if(regwrite == 1'b1)
  begin
  reg_set[writereg] <= writedata;
  end
end

endmodule
```

7 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);
```

```
input [31:0] entr1, entr2;
input [3:0] alu_ctrl;

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

output reg [31:0] alu_result;

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
  endcase
end
endmodule
```

8 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 [1:0] aluOp;
input wire [5:0] func;
output reg [3:0] out;

always @(func or aluOp)
begin
case (aluOp)
 2'b00: //R type
 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

    2'b01: // I-type (addi, load y store)
 out <= 4'b0000; //ADD
    2'b10: //I-type (subi, branches y slt)
 out = 4'b0001; //SUB

endcase
end
```

```
endmodule
```

9 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
```

10 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
```

11 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
```

12 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
```

13 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
```