Single cycle MIPS processor

Team 3 :

1- Rana Wagdy
2- Omnia Mohammed
3- Youmna Mohammed
4- Kerolos Youssef

CODE:

Top module:

```
module
MIPS(clk,reset,register_data1,register_data2,ALU_out,resu
lt,A1,A2,A3);
  input clk,reset;
  output [31:0]
register_data1,register_data2,ALU_out,result;
  output [4:0] A1,A2,A3;
```

Signals:

```
  wire [31:0]
pc_branch,pc_jump,pc_plus4,pc,pc_before_clk,final_pc;//
PC wires
  wire [31:0]int_four;//adder
  wire [31:0] int31;
  assign int31=32'd31;
  assign int_four=32'b100;
  wire [31:0] instruction;//instruction memory
  wire [4:0]write_to_register;//register file
  wire[31:0]
regfile_WD,register_RD1,register_RD2;//register_file
  wire[31:0]extended_imm,extended_imm_by_four;
//extension
  wire[1:0]mem_to_reg,reg_dest,alu_op;//control unit
  wire
jump,mem_write_en,reg_write_en,branch,ori_ctrl,alu_src,
zero,pc_source,sourceB;
  wire[2:0] alu_control;//alu
  wire[31:0]
alu_output,mem_data_out,data_result;//result
  //***************************
connection****************************************
  Mux_2 PCSource(pc_plus4, pc_branch, pc, pc_source);
  assign pc_jump[31:28]=pc_plus4[31:28];
  Mux_2 PCJump(pc, pc_jump, pc_before_clk, jump);
  PC final_PC (final_pc,pc_before_clk,clk,reset);
  adder PCPlus4 (final_pc,int_four,pc_plus4);
  instr_mem instruction_mem(final_pc,instruction);
  Shift_Left_Two_Jump shifter1 (instruction[25:0],
pc_jump[27:0]);
  extension extend_address(instruction[15:0],
extended_imm,ori_ctrl);
  Shift_Left_Two
shifter2(extended_imm_by_four,extended_imm);
  adder
PCBranch(extended_imm_by_four,pc_plus4,pc_branch);
 controlunit
main_decoder(instruction[31:26],instruction[5:0],
mem_write_en,mem_to_reg,branch,alu_src,reg_dest,reg_
write_en,alu_op,jump,ori_ctrl);
 aludec decoder
(instruction[31:26],instruction[5:0],alu_op,alu_control);
  Mux_2 Source_B(register_RD2, extended_imm, sourceB,
alu_src);
  alu ALU(register_RD1,
sourceB,alu_control,alu_output,zero);
  assign pc_source=branch && zero;
  Data_Memory memory
(mem_data_out,clk,mem_write_en,alu_output,register_R
D2);
```

```
  Mux_3
resulted_data(alu_output,mem_data_out,pc_plus4,
data_result, mem_to_reg);

  register_file RegisterFile (instruction[25:21],
instruction[20:16], write_to_register, register_RD1,
register_RD2,data_result,reg_write_en, clk);

  Mux_3 register_dst(instruction[20:16],
instruction[15:11],int31, write_to_register, reg_dest);


Endmodule
```

Program counter:

```
module PC(out,in,clk,reset);

  output [31:0] out;

  input [31:0] in ;

  input clk,reset;

  reg [31:0] out;


  initial begin

    out =32'b0;

  end


  always@(posedge clk or posedge reset)
  begin
    if (reset == 1'b1)
      out<=32'b0;
    else
      out <= in;
  end
endmodule
```

Register file:

```
module register_file (A1, A2, A3, RD1, RD2, WD3, WE3, 
clk);
        input [4:0] A1, A2, A3;
        input [31:0] WD3;
        input clk, WE3;
        output [31:0] RD1, RD2;

        reg [31:0] regfile [31:0];
        initial
        begin
          regfile[0]=32'b0;
        end
        assign RD1 = regfile[A1];
        assign RD2 = regfile[A2];

        always @(posedge clk )
        begin
                if (WE3 == 1'b1) regfile[A3] = WD3;
        end
endmodule
```

ALU:

```
module alu
(

input [31:0] srcA , srcB,
input [2:0] alucontrol,
output reg [31:0] aluresult,
output reg zero

);

always @(*) begin
```

```
  case (alucontrol)
   3'b010: aluresult <= srcA + srcB; // add
   3'b110: aluresult <= srcA + (~srcB) + 1; // sub
   3'b000: aluresult <= srcA & srcB; // and
   3'b001: aluresult <= srcA | srcB; // or
        3'b111: aluresult <=((srcA[31] != srcB[31]) &&
srcA<srcB ) ? 1 : 0; // slt
   default: aluresult <=0;
     endcase
end

always @(aluresult) begin
if(aluresult==0)
    begin
        zero<=1;
    end
else begin
        zero<=0;
end
```

```
end
endmodule
```

ALU decoder:

```
module aludec(
   input [5:0]opcode,
   input [5:0]funct,
   input [1:0]aluop,
   output reg [2:0] alucontrol

   );
   always @(*)
   begin
      case(aluop)
         2'b00:alucontrol<=3'b010;//ADD_LW_SW_ADDI
         2'b01:alucontrol<=3'b110;//SUB_BEQ
         2'b10:case(funct) //R_Type
               6'b100000:alucontrol<=3'b010;//ADD
               6'b100010:alucontrol<=3'b110;//SUB
               6'b100100:alucontrol<=3'b000;//AND
               6'b100101:alucontrol<=3'b001;//OR
```

```
                6'b101010:alucontrol<=3'b111;//SLT
                default:alucontrol<=3'bxxx;
              endcase
   2'b11:case(opcode)
             6'b001000:alucontrol<=3'b010;//ADDI
          6'b001101:alucontrol<=3'b001;//ORI
          default:alucontrol<=3'bxxx;//NOT Used
           endcase
          endcase


      end
endmodule
```

Control unit:

```
module controlunit
(
 input [5:0] opcode,
 input [5:0] funct, //we describe main decoder not ALU
decoder so this variable is not used


 output reg mem_write,
 output reg [1:0]mem_toreg,

 output reg branch,

 output reg alu_src,

 output reg[1:0] reg_dst,

 output reg reg_write,

 output reg [1:0] alu_op,

 output reg jump,

 output reg logic


 );


 always@(*)

 begin

 if(opcode==6'b000011)

   begin

    mem_toreg[1]=1'b1;

    reg_dst[1]=1'b1;

   end

 else begin

   mem_toreg[1]=1'b0;
   reg_dst[1]=1'b0;
 end
 if (opcode==6'b001101)
   logic=1'b1;
 else logic=1'b0;

 case(opcode)

    6'b000000: //R_Type

           begin

                  reg_dst[0]=1;

                  alu_src=0;

                  mem_toreg[0]=0;

                  reg_write=1;
```

```
        mem_write=0;

        branch=0;

        alu_op=2'b10;

        jump=0;

    end

6'b100011: //lw

    begin

        reg_dst[0]=0;

        alu_src=1;

        mem_toreg[0]=1;
```

```
          reg_write=1;

          mem_write=0;

          branch=0;

          alu_op=2'b00;

          jump=0;

     end

6'b101011: //sw

     begin

          reg_dst[0]=1'bx;

          alu_src=1;
          mem_toreg[0]=1'bx;

          reg_write=0;

          mem_write=1;

          branch=0;

          alu_op=2'b00;

          jump=0;

     end

6'b000010: //jump

     begin

          reg_dst[0]=1'bx;
```

```
        alu_src=1'bx;

        mem_toreg[0]=1'bx;

        reg_write=0;

        mem_write=0;

        branch=1'bx;

        alu_op=2'bxx;

        jump=1;

    end

6'b000011: //jal

    begin
```

```
            reg_dst[0]=1'b0;

            alu_src=1'bx;

            mem_toreg[0]=1'b0;

            reg_write=1;

            mem_write=0;

            branch=1'bx;

            alu_op=2'b00;

            jump=1;

        end

    6'b000100: //beq
```

```
begin

    reg_dst[0]=1'bx;

    alu_src=0;

    mem_toreg[0]=1'bx;

    reg_write=0;

    mem_write=0;

    branch=1;

    alu_op=2'b01;

    jump=0;

end
```

```
6'b001000: //addi

      begin

            reg_dst[0]=0;

            alu_src=1;

           mem_toreg[0]=0;

            reg_write=1;

            mem_write=0;

            branch=0;

            alu_op=2'b11;
            jump=0;

        end

    6'b001101: //ori

        begin

            reg_dst[0]=0;

            alu_src=1;

            mem_toreg[0]=0;

            reg_write=1;

            mem_write=0;

            branch=0;
            alu_op=2'b11;

            jump=0;

        end


    default://look @ function

        begin

            reg_dst[0]=1;

            alu_src=0;

            mem_toreg[0]=0;

            reg_write=1;

            mem_write=0;
```

```
              branch=0;

              alu_op=2'b10;

          end

 endcase
end
endmodule
```

Data memory:

```
module Data_Memory(output [31:0] Read_Data,input
clk,WE,output reg[31:0] ALU_Result,output
reg[31:0]Write_Data);
  reg [31:0] Memory[63:0];
  //wire [31:2] memory_add;

  assign Read_Data=Memory[ALU_Result[31:2]];
  always @(posedge clk)
  begin
  if(WE)
  Memory[ALU_Result[31:2]]<=Write_Data;
  end
endmodule
```

Instruction memory:

```
module instr_mem        // a synthesisable rom
implementation
 (
     input    [31:0]    pc,
     output wire   [31:0]        instruction
 );
     wire [4: 0] rom_addr = pc[5: 1];
     reg [31:0] rom[21:0];
     initial
     begin
               rom[0] <=
32'b00100000000010000000000000100000; //addi $t0,
$zero, 32
               rom[1] <=
32'b00100000000010010000000000110111; //addi $t1,
$zero, 55
```

```
            rom[2] <=
32'b00000001000010011000000000100100; //and $s0,
$t0, $t1
            rom[3] <=
32'b00000001000010011000000000100101; //or $s0, $t0,
$t1
            rom[4] <=
32'b10101100000100000000000000000100; //sw $s0,
4($zero)
            rom[5] <=
32'b00000001000010011001000000100010; //sub $s2,
$t0, $t1
            rom[6] <=
32'b10101100000010000000000000001000; //sw $t0,
8($zero)
            rom[7] <=
32'b00000001000010011001000000100010; //sub $s2,
$t0, $t1
            rom[8] <=
32'b00010010001100100000000000001001; //beq $s1,
$s2, error0
            rom[9] <=
32'b10001100000100010000000000000100; //lw $s1,
4($zero)
```

```
            rom[10] <=
32'b00110010001100100000000001001000; //andi $s2,
$s1, 48

            rom[11] <=
32'b00010010001100100000000000001001; //beq $s1,
$s2, error1

            rom[12] <=
32'b00000010010100011010000000101010; //slt $s4, $s2,
$s1 (Last)

            rom[13] <=
32'b00000010001000001001000000100000; //add $s2,
$s1, $0

            rom[14] <=
32'b00001100000000000000000000001110; //jal last;

            rom[15] <=
32'b00100000000010000000000000000000; //addi $t0,
$0, 0(error0)

            rom[16] <=
32'b00100000000010010000000000000000; //addi $t1,
$0, 0

            rom[17] <=
32'b00100000000010000000000000000001; //addi $t0,
$0, 1(error1);
```

```
            rom[18] <=
32'b00100000000010010000000000000001; //addi $t1,
$0, 1 ;
            rom[19] <=
32'b00110100000010000000000000000000; //ori $t0, $0,
0(error0)
            rom[20] <=
32'b00001000000000000000000000011111; //j EXIT;
     end
     assign instruction = (pc[31:0] != 0 )? pc[31:0]:32'b0;
 endmodule


module Shift_Left_Two(output [31:0] Output,input[31:0]
Input);
  assign Output={Input[29:0] ,2'b00};
endmodule


module Shift_Left_Two_Jump (shift_in, shift_out);
        input [25:0] shift_in;
        output [27:0] shift_out;
        assign shift_out[27:0]={shift_in[25:0],2'b00};
endmodule
```

```
//###############################################
#########################
module Mux_3 (input0, input1,input2, mux_out, control);
        input [31:0] input0, input1,input2;
        output reg [31:0] mux_out;
        input[1:0]control;
        always@(*) begin
        if (control[1])
           mux_out=input2;
        else if (control[1]!=1)
          mux_out=control[0]?input1:input0;
end
endmodule
//###############################################
###############################
module Mux_2 (input0, input1, mux_out, control);
        input [31:0] input0, input1;
        output [31:0] mux_out;
        input control;
        assign mux_out=control?input1:input0;
endmodule
```

```
//################################################
################################
module adder (input1,input2,out);
  input [31:0] input1, input2;
        output [31:0] out;
        assign out=input1+input2;
endmodule
//################################################
##
module extension (imm, sign_imm,logic);
 input [15:0] imm;
 input logic;
 output reg[31:0] sign_imm;
 always@(*)begin
 sign_imm[15:0]=imm[15:0];
 if(logic)  sign_imm[31:16]=16'b0;
 else

sign_imm[31:16]=imm[15]?16'b1111111111111111:16'b0;
end
endmodule
```