Computer Architecture
Assignment-1

Group -19

Tanvi Behra	**2019B5A80989H**
Adithya K	**2019B4A80926H**
Ayaz Hussain	**2019B5A81108H**

- ❖ MIPS Single Cycle Datapath in Verilog
- ❖ Testing the circuit with R-format instruction.

Codes:

Module Alu_add.v

```
1  `timescale 1ns / 1ps
2
3  module alu_add(A,B,add_op);
4  input [31:0]A;
5  input [31:0]B;
6  output [31:0]add_op;
7  assign add_op=A+B;
8  endmodule
```

Module alu_control.v

```
1   `timescale 1ns / 1ps
2
3   module alu_control (aluop, func, alu_select);
4   input [1:0]aluop;
5   input [5:0]func;
6   output reg [4:0] alu_select;  //AINV,BINV,MuxSelects 2;
7
8   always@(aluop or func)
9   begin    if (aluop==2'b10) begin
10                  case(func)
11                          6'b100000: alu_select=4'b0010;  //add
12                          6'b100010: alu_select=4'b0110;  //sub
13                          6'b100100: alu_select=4'b0000;  //AND
14                          6'b100101: alu_select=4'b0001;  //OR
15                          6'b100110: alu_select=4'b1100;  //NOR
16                          6'b100111: alu_select=4'b1101;  //NAND
17                          6'b101010: alu_select=4'b0111;  //SLT
18                          default: alu_select=4'b0000;
19                  endcase
20                  end
21            else begin
22                  if(aluop==2'b00)    alu_select=4'b0010;     //add for LW,SW
23                  else if (aluop==2'b01) alu_select=4'b0110;  //sub for Beg
24                 end
25  end
26  endmodule
```

Module alu.v

```
`timescale 1ns / 1ps

module alu_1_bit (A, B, CIN, LESS, AINV, BINV, Opr, RESULT, COUT, ADD_R);

input A, B, CIN, LESS, AINV, BINV;
input [1:0] Opr;
output RESULT, ADD_R, COUT;

reg RESULT, COUT, ADD_R;

always @(*)

      begin  : combinational_logic
      reg RESULT_AND, RESULT_OR, RESULT_ADD_SUB, RESULT_SLT;
      reg Am, Bm; // Am is the A at the output of the mux controlled by AINV; Similarly Bm

        Am = AINV ?     ~A : A  ;
        Bm = BINV ?     ~B : B  ;


        RESULT_AND =    Am & Bm ;
        RESULT_OR  =    Am | Bm ;
        ADD_R = (Am ^ Bm ^ CIN) | (Am&Bm&CIN)   ;
        RESULT_ADD_SUB = ADD_R;
        RESULT_SLT = LESS;
        COUT   = ((Am & Bm) | (Bm & CIN) | (CIN & Am))      ;

        case (Opr)
                0   : // 2'b00 Logical AND Function
                    begin
                    RESULT = RESULT_AND;
                    end
                1   : // 2'b01 Logical OR Function
                    begin
                    RESULT = RESULT_OR;
                    end
                2   : // 2'b10 Arithmatic addition or subtract depending on the value of BINV
                    begin
                     RESULT = RESULT_ADD_SUB;
                    end
                3   : // 2'b11 for the SLT A, B instruction ( if A < B, RESULT =1)
                    begin
                     RESULT = RESULT_SLT;
                    end

          endcase
       end
endmodule //alu_1_bit
```

```
52  module alu(A, B, AINV, BNEG, Opr, RESULT, OVERFLOW, ZERO, COUT);
53
54  input [31:0] A, B;
55  input  AINV, BNEG;
56  input [1:0] Opr;
57
58  output[31:0] RESULT;
59  output OVERFLOW, ZERO, COUT;
60
61  wire COUT1, COUT2, COUT3, COUT4, COUT5,COUT6,COUT7,COUT8,COUT9,COUT10,COUT11,COUT12,COUT13,COUT1
62      COUT17,COUT18,COUT19,COUT20,COUT21,COUT22,COUT23,COUT24,COUT25,COUT26,
63      COUT27,COUT28,COUT29,COUT30,COUT31,COUT32;
64  wire BINV, SET, LESS_0,CIN;
65  wire [31:0] ADD_R;
66
67  assign BINV = BNEG;
68  assign CIN = BNEG;
69
70  assign COUT = COUT32;
71  assign OVERFLOW = COUT32 ^ COUT31;
72  assign SET = ADD_R[31];
73  assign LESS_0 =  OVERFLOW ? ~ SET : SET;
74  assign ZERO = (RESULT==0)?1:0;
75
76  // module alu_1_bit (A, B, CIN, LESS, AINV, BINV, Opr, RESULT, COUT, ADD_R);
77  alu_1_bit alu0 (A[0], B[0], CIN, LESS_0, AINV, BINV, Opr, RESULT[0], COUT1, ADD_R[0]);
78  alu_1_bit alu1 (A[1], B[1], COUT1, 1'b0, AINV, BINV, Opr, RESULT[1], COUT2, ADD_R[1]);
79  alu_1_bit alu2 (A[2], B[2], COUT2, 1'b0, AINV, BINV, Opr, RESULT[2], COUT3, ADD_R[2]);
80  alu_1_bit alu3 (A[3], B[3], COUT3, 1'b0, AINV, BINV, Opr, RESULT[3], COUT4, ADD_R[3]);
81  alu_1_bit alu4 (A[4], B[4], COUT4, 1'b0, AINV, BINV, Opr, RESULT[4], COUT5, ADD_R[4]);
82  alu_1_bit alu5 (A[5], B[5], COUT5, 1'b0, AINV, BINV, Opr, RESULT[5], COUT6, ADD_R[5]);
83  alu_1_bit alu6 (A[6], B[6], COUT6, 1'b0, AINV, BINV, Opr, RESULT[6], COUT7, ADD_R[6]);
84  alu_1_bit alu7 (A[7], B[7], COUT7, 1'b0, AINV, BINV, Opr, RESULT[7], COUT8, ADD_R[7]);
85  alu_1_bit alu8 (A[8], B[8], COUT8, 1'b0, AINV, BINV, Opr, RESULT[8], COUT9, ADD_R[8]);
86  alu_1_bit alu9 (A[9], B[9], COUT9, 1'b0, AINV, BINV, Opr, RESULT[9], COUT10, ADD_R[9]);
87
88  alu_1_bit alu10 (A[10], B[10], COUT10, 1'b0, AINV, BINV, Opr, RESULT[10], COUT11, ADD_R[10]);
89  alu_1_bit alu11 (A[11], B[11], COUT11, 1'b0, AINV, BINV, Opr, RESULT[11], COUT12, ADD_R[11]);
90  alu_1_bit alu12 (A[12], B[12], COUT12, 1'b0, AINV, BINV, Opr, RESULT[12], COUT13, ADD_R[12]);
91  alu_1_bit alu13 (A[13], B[13], COUT13, 1'b0, AINV, BINV, Opr, RESULT[13], COUT14, ADD_R[13]);
92  alu_1_bit alu14 (A[14], B[14], COUT14, 1'b0, AINV, BINV, Opr, RESULT[14], COUT15, ADD_R[14]);
93  alu_1_bit alu15 (A[15], B[15], COUT15, 1'b0, AINV, BINV, Opr, RESULT[15], COUT16, ADD_R[15]);
94  alu_1_bit alu16 (A[16], B[16], COUT16, 1'b0, AINV, BINV, Opr, RESULT[16], COUT17, ADD_R[16]);
95  alu_1_bit alu17 (A[17], B[17], COUT17, 1'b0, AINV, BINV, Opr, RESULT[17], COUT18, ADD_R[17]);
96  alu_1_bit alu18 (A[18], B[18], COUT18, 1'b0, AINV, BINV, Opr, RESULT[18], COUT19, ADD_R[18]);
97  alu_1_bit alu19 (A[19], B[19], COUT19, 1'b0, AINV, BINV, Opr, RESULT[19], COUT20, ADD_R[19]);
98
99  alu_1_bit alu20 (A[20], B[20], COUT20, 1'b0, AINV, BINV, Opr, RESULT[20], COUT21, ADD_R[20]);
100 alu_1_bit alu21 (A[21], B[21], COUT21, 1'b0, AINV, BINV, Opr, RESULT[21], COUT22, ADD_R[21]);
101 alu_1_bit alu22 (A[22], B[22], COUT22, 1'b0, AINV, BINV, Opr, RESULT[22], COUT23, ADD_R[22]);
102 alu_1_bit alu23 (A[23], B[23], COUT23, 1'b0, AINV, BINV, Opr, RESULT[23], COUT24, ADD_R[23]);
103 alu_1_bit alu24 (A[24], B[24], COUT24, 1'b0, AINV, BINV, Opr, RESULT[24], COUT25, ADD_R[24]);
104 alu_1_bit alu25 (A[25], B[25], COUT25, 1'b0, AINV, BINV, Opr, RESULT[25], COUT26, ADD_R[25]);
105 alu_1_bit alu26 (A[26], B[26], COUT26, 1'b0, AINV, BINV, Opr, RESULT[26], COUT27, ADD_R[26]);
106 alu_1_bit alu27 (A[27], B[27], COUT27, 1'b0, AINV, BINV, Opr, RESULT[27], COUT28, ADD_R[27]);
```

```
 alu_1_bit alu20 (A[20], B[20], COUT20, 1'b0, AINV, BINV, Opr, RESULT[20], COUT21, ADD_R[20]);
 alu_1_bit alu21 (A[21], B[21], COUT21, 1'b0, AINV, BINV, Opr, RESULT[21], COUT22, ADD_R[21]);
 alu_1_bit alu22 (A[22], B[22], COUT22, 1'b0, AINV, BINV, Opr, RESULT[22], COUT23, ADD_R[22]);
 alu_1_bit alu23 (A[23], B[23], COUT23, 1'b0, AINV, BINV, Opr, RESULT[23], COUT24, ADD_R[23]);
 alu_1_bit alu24 (A[24], B[24], COUT24, 1'b0, AINV, BINV, Opr, RESULT[24], COUT25, ADD_R[24]);
 alu_1_bit alu25 (A[25], B[25], COUT25, 1'b0, AINV, BINV, Opr, RESULT[25], COUT26, ADD_R[25]);
 alu_1_bit alu26 (A[26], B[26], COUT26, 1'b0, AINV, BINV, Opr, RESULT[26], COUT27, ADD_R[26]);
 alu_1_bit alu27 (A[27], B[27], COUT27, 1'b0, AINV, BINV, Opr, RESULT[27], COUT28, ADD_R[27]);
 alu_1_bit alu28 (A[28], B[28], COUT28, 1'b0, AINV, BINV, Opr, RESULT[28], COUT29, ADD_R[28]);
 alu_1_bit alu29 (A[29], B[29], COUT29, 1'b0, AINV, BINV, Opr, RESULT[29], COUT30, ADD_R[29]);

 alu_1_bit alu30 (A[30], B[30], COUT30, 1'b0, AINV, BINV, Opr, RESULT[30], COUT31, ADD_R[30]);
 alu_1_bit alu31 (A[31], B[31], COUT31, 1'b0, AINV, BINV, Opr, RESULT[31], COUT32, ADD_R[31]);

 endmodule

```

Module control_unit.v

```
`timescale 1ns / 1ps

module control_unit(opcode, regdist, jump, branch, memread, memwrite, memtoreg,alusrc,regwrite, aluop );
input [5:0] opcode;
output reg regdist, jump, branch, memread, memwrite, memtoreg, alusrc, regwrite;
output reg [1:0]aluop;

always@(opcode)
begin
    case (opcode)
    6'b000_000:  begin                                  //R- TYPE instructions
        regdist=1; jump=0; branch=0; memread=0;
        memwrite=0; memtoreg=0; alusrc=0; regwrite=1;
        aluop=2'b10;
                end
    6'b100_011:  begin                                  //LW instruction - LOAD
        regdist=0; jump=0; branch=0; memread=1;
        memwrite=0; memtoreg=1; alusrc=1; regwrite=1;
        aluop=2'b00;
                end
    6'b101_011:  begin                                  //SW instruction - STORE
        regdist=0; jump=0; branch=0; memread=0;
        memwrite=1; memtoreg=0; alusrc=1; regwrite=0;
        aluop=2'b00;
                end
    6'b000_100:  begin                                  //BEQ instruction - Branch if equal
        regdist=0; jump=0; branch=1; memread=0;
        memwrite=0; memtoreg=0; alusrc=0; regwrite=0;
        aluop=2'b01;
                end
    6'b000_010:  begin                                  //JUMP instruction - j target address
        regdist=0; jump=1; branch=0; memread=0;
        memwrite=0; memtoreg=0; alusrc=0; regwrite=0;
        aluop=2'b00;
                end
    default:     begin
        regdist=0; jump=0; branch=0; memread=0;
        memwrite=0; memtoreg=0; alusrc=0; regwrite=0;
        aluop=2'b00;
                end
    endcase
end
endmodule
```

Module data_memory.v

```
1  `timescale 1ns / 1ps
2
3  module data_memory(clk,reset,add,write_data,read_data,memwrite,memread);
4  input clk,reset,memwrite,memread;
5  input [31:0]add;
6  input [31:0]write_data;
7  output [31:0]read_data;
8  reg [31:0] DMemory [31:0]; //can be extended to 2 power 32-1 :0
9  integer k;
10 wire [31:0] shifted_addr;
11 //assign shifted_addr={add[31:2],2'b0};
12 assign shifted_addr=add[31:2];
13 assign read_data = (memread) ? DMemory[add] : 32'bx;
14
15 always @(posedge clk or posedge reset)
16     begin
17         if (reset == 1)
18             begin
19                 for (k=0; k<64; k=k+1) begin
20                     DMemory[k] = 32'b0;
21                 end
22             DMemory[0]=32'b0000_0000_0000_0000_0000_0000_0000_0000;
23             DMemory[1]=32'b0000_0000_0000_0000_0000_0000_1010_0101;
24             DMemory[2]=32'b0000_0000_0000_0000_0000_0000_0000_0001;
25             DMemory[3]=32'b0000_0000_0000_0000_0000_0000_0000_0010;
26             DMemory[4]=32'b0000_0000_0000_0000_0000_0000_0000_0011;
27             DMemory[5]=32'b0000_0000_0000_0000_0000_0000_0000_0100;
28             end
29         else
30             if (memwrite) DMemory[add] = write_data;
31     end
32
33 endmodule
```

Module instruction_mem.v

```
`timescale 1ns / 1ps

module instruction_mem(read_add, instruction, reset);
input reset;
input [31:0] read_add;
output [31:0] instruction;
reg [31:0]Imemory [31:0]; //can be upto 2 power 32-1 : 0 as well
integer k;
wire [31:0] abs_read_add;
//assign abs_read_add={read_add[31:2],2'b0};
assign abs_read_add={read_add[31:2]};
assign instruction = Imemory[abs_read_add];

always@(posedge reset)
begin
        for (k=0; k<1023; k=k+1) begin
            Imemory[k] = 32'b0;
        end

//Hardccoding the instructions initially.
Imemory[0]=32'b000000_00001_00010_00011_00000_100100;       //RTYPE - AND  AND $3, $1, $2;
Imemory[1]=32'b000000_00001_00010_00100_00000_100101;       //RTYPE - OR   OR  $4, $1, $2;
Imemory[2]=32'b000000_00001_00010_00101_00000_100110;       //RTYPE - NOR  NOR $5, $1, $2;
Imemory[3]=32'b000000_00001_00010_00110_00000_100111;       //RTYPE - NAND NAND $6,$1, $2;
Imemory[4]=32'b000000_00001_00010_00111_00000_100000;       //RTYPE - ADD  ADD $7, $1, $2;
Imemory[5]=32'b000000_00001_00010_01000_00000_100010;       //RTYPE - SUB  SUB $8, $1, $2;
Imemory[6]=32'b000000_00001_00010_01001_00000_101010;       //RTYPE - SLT  SLT $9, $1, $2;
Imemory[7]=32'b000000_00010_00001_01010_00000_101010;       //RTYPE - SLT  SLT $10, $2, $1;

Imemory[8]=32'b100011_00000_01011_00000000_00000100;       //ITYPE - LW   lw $11, 0004($0);
Imemory[9]=32'b101011_00000_01011_00000000_00000100;       //ITYPE - SW   sw $11, 0004($0);
end
endmodule
```

Module mux32bit.v

```
`timescale 1ns / 1ps

module mux_32bit(ip1,ip2,op,s);
input [31:0]ip1;
input [31:0]ip2;
output [31:0]op;
input s;
assign op=s?ip1:ip2;

endmodule
```

Module program_counter.v

```
1   `timescale 1ns / 1ps
2
3   module program_counter(clk, reset, PC_in, PC_out);
4   input clk, reset;
5   input [31:0] PC_in;
6   output reg [31:0] PC_out;
7   always @ (posedge clk, posedge reset)
8       begin   if(reset==1)
9                   PC_out<=0;
10              else
11                  PC_out<=PC_in;
12      end
13  endmodule
```

Module Register file.v

```
1   `timescale 1ns / 1ps
2
3
4   module register_file(read_add_1,read_add_2,write_add,write_data,read_data_1,read_data_2,reg_write,clk,reset);
5   input [4:0] read_add_1;
6   input [4:0] read_add_2;
7   input [4:0] write_add;
8   input [31:0]write_data;
9   output [31:0] read_data_1;
10  output [31:0] read_data_2;
11  input reg_write,clk,reset;
12
13  reg [31:0] registers [31:0];
14  integer k;
15
16  assign read_data_1=registers[read_add_1];
17  assign read_data_2=registers[read_add_2];
18
19  always@(posedge clk, posedge reset)
20  begin
21          if (reset==1'b1)
22          begin
23              for (k=0; k<32; k=k+1)
24              begin
25                  registers[k] = 32'b0;
26              end
27              //initializing some regsiters to few values.
28              registers[0]=32'b0000_0000_0000_0000_0000_0000_0000_0000;
29              registers[1]=32'b0000_0000_0000_0000_0000_0000_1010_0101;
30              registers[2]=32'b1111_1111_1111_1111_1111_1111_0101_1010;
31
32          end
33
34          else if (reg_write == 1'b1) registers[write_add] = write_data;
35      end
36
37  endmodule
```

Module shift_left2_branch.v

```
1  `timescale 1ns / 1ps
2
3  module shift_left2_branch(add_in, add_out);
4  input [31:0] add_in;
5  output [31:0] add_out;
6  assign add_out[31:0]={add_in[29:0],2'b00};
7  endmodule
```

Module shift_left2_jump.v

```
1  `timescale 1ns / 1ps
2
3  module shift_left2_jump(add_in,add_out);
4  input [25:0] add_in;
5  output [27:0] add_out;
6  assign add_out[27:0]={add_in[25:0],2'b00};
7  endmodule
```

Module sign_extend.v

```
1  `timescale 1ns / 1ps
2
3
4  module sign_extend(sign_in, sign_out);
5  input [15:0] sign_in;
6  output [31:0] sign_out;
7  assign sign_out[15:0]=sign_in[15:0];
8  assign sign_out[31:16]=sign_in[15]?16'b1111_1111_1111_1111:16'b0;
9  endmodule
```

Module single cycle cpu.v

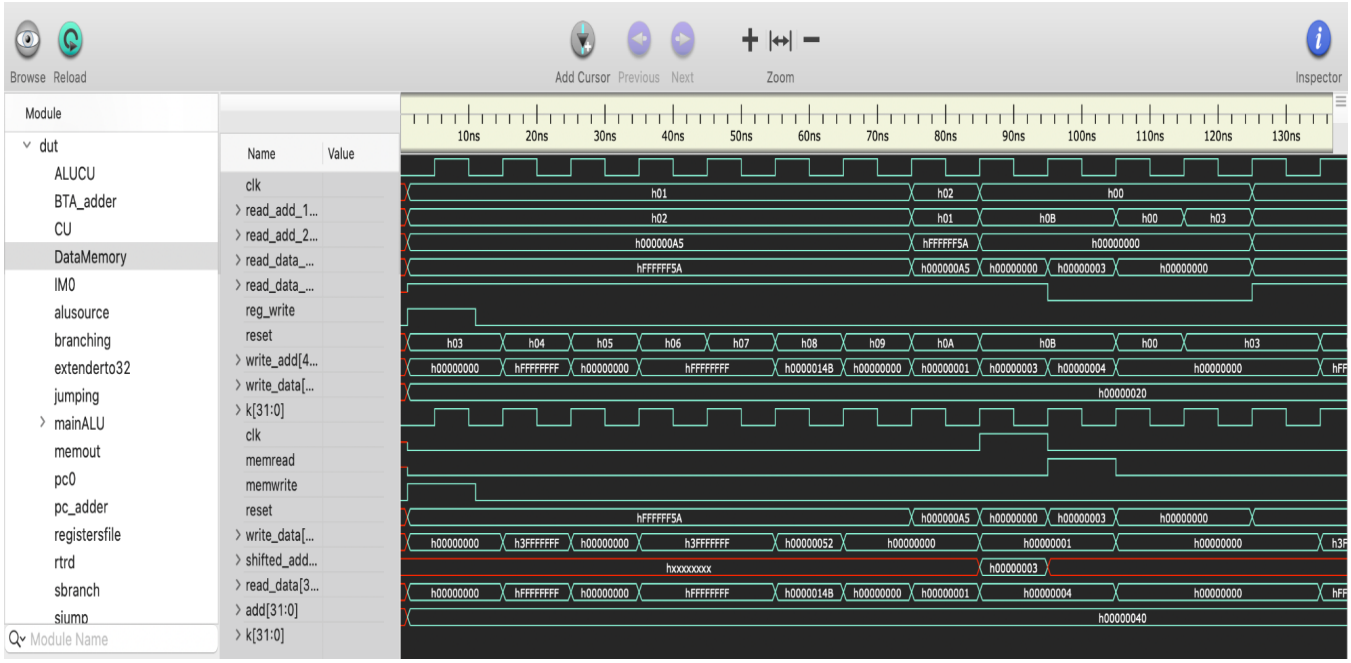
```
`timescale 1ns / 1ps

module SingleCycleCPU(clk, reset);
input clk,reset;

wire [31:0]PC_next;            //next address accessing the Instructions
wire [31:0]PC;              //address accessing the instruction in this cycle
wire [31:0]Instruction;     //the 32 bit instruction in MIPS
wire [31:0]write_addr;      //select the Rg or Rt
wire [31:0]write_data;      //writing to the DM registers
wire [31:0]reg_data1,reg_data2; //data from DM registers
wire regdist, jump, branch, memread, memwrite, memtoreg,alusrc,regwrite;    //CPU CU signals
wire [1:0]aluop;                //for ALU control from CU
wire [3:0]alu_select;           //selecting the operation on the ALU
wire [31:0]extendedfield;       //when 16bit is extended, this is used.
wire [31:0]alusource2;   //ALU'2 32 bit input - selected from reg_data2 or extendedfield
wire [31:0]aluresult;   //result after operation in ALU
wire overflow_detect,cout;     //an extra function...currently no use in the design
wire zero;                      //=1 if result is 32'b0
wire [31:0]dm_readdata;         //read data from the Data Memory
wire [31:0]PC_4;            //PC<=PC+4;
wire [31:0]offsetadd;       // Here the offset is ready to be added to the PC+4
wire [31:0]addjump;         //PC+4 + Offset ready
wire [31:0]PC_J;            // addjump or PC_4
wire [27:0]jumpaddP;        //jumpaddress almost ready
program_counter     pc0(.clk(clk), .reset(reset), .PC_in(PC_next), .PC_out(PC));
instruction_mem     IM0(.read_add(PC), .instruction(Instruction), .reset(reset));
mux_32bit           rtrd(.ip1(Instruction[15:11]), .ip2(Instruction[20:16]), .op(write_addr), .s(regdist)); //s=1, op=ip1;
register_file       registersfile(.read_add_1(Instruction[25:21]),.read_add_2(Instruction[20:16]),
                        .write_add(write_addr),.write_data(write_data),.read_data_1(reg_data1),
                        .read_data_2(reg_data2),.reg_write(regwrite),.clk(clk),.reset(reset));
sign_extend         extenderto32(.sign_in(Instruction[15:0]), .sign_out(extendedfield));
mux_32bit           alusource(.ip1(extendedfield),.ip2(reg_data2),.op(alusource2),.s(alusrc));
alu                 mainALU(.A(reg_data1),. B(alusource2), .AINV(alu_select[3]),
                            .BNEG(alu_select[2]), .Opr({alu_select[1],alu_select[0]}),
                            .RESULT(aluresult), .OVERFLOW(overflow_detect), .ZERO(zero), .COUT(cout));
data_memory         DataMemory(.clk(clk),.reset(reset),.add(aluresult),.write_data(reg_data2),
                       .read_data(dm_readdata),.memwrite(memwrite),.memread(memread));
mux_32bit           memout(.ip1(dm_readdata),.ip2(aluresult),.op(write_data),.s(memtoreg));
control_unit        CU(.opcode(Instruction[31:26]), .regdist(regdist), .jump(jump),
                       .branch(branch), .memread(memread), .memwrite(memwrite),
                       .memtoreg(memtoreg), .alusrc(alusrc), .regwrite(regwrite), .aluop(aluop) );
alu_control         ALUCU(.aluop(aluop), .func(Instruction[5:0]), .alu_select(alu_select));
alu_add             pc_adder(.A(PC),.B(32'd4),.add_op(PC_4));
shift_left2_branch  sbranch(.add_in(extendedfield), .add_out(offsetadd));
alu_add             BTA_adder(.A(PC_4),.B(offsetadd),.add_op(addjump));
mux_32bit           branching(.ip1(addjump),.ip2(PC_4),.op(PC_J),.s(branch&&zero));
shift_left2_jump    sjump(.add_in(Instruction[25:0]),.add_out(jumpaddP));
mux_32bit           jumping(.ip1({PC_4[31:28],jumpaddP}),.ip2(PC_J),.op(PC_next),.s(jump));

endmodule
```

TESTBENCH

Module singlecyclecpu.v

```
1  `timescale 1ns / 1ps
2
3  `include "SingleCycleCPU.v"
4  `include "alu_add.v"
5  `include "alu_control.v"
6  `include "alu.v"
7  `include "control_unit.v"
8  `include "data_memory.v"
9  `include "instruction_mem.v"
10 `include "mux_32bit.v"
11 `include "program_counter.v"
12 `include "register_file.v"
13 `include "shift_left2_branch.v"
14 `include "shift_left2_jump.v"
15 `include "sign_extend.v"
16
17 //Tanvi Behra  2019B5A80989H
18 //Adithya K    2019B4A80926H
19 //Ayaz Hussain 2019B5A81108H
20
21 module singlecyclecpu_tb(   );
22
23 reg clk,reset;
24 SingleCycleCPU dut(clk, reset);
25
26 initial
27 begin
28 clk=0;
29 reset=0;
30 #1 reset=1;
31 #10 reset=0;
32 #110 $finish;
33 end
34 always begin
35  #5 clk=~clk;
36  end
37
38 initial begin
39         $dumpfile("dump.vcd");
40         $dumpvars(0, dut);
41     end
42 endmodule
43
```

OUTPUT


Browse
Reload
Add Cursor
Previous
Next
Zoom
Inspector
Module
dut
ALUCU
BTA_adder
CU
DataMemory
IM0
alusource
branching
extenderto32
jumping
mainALU
memout
pc0
pc_adder
registersfile
rtrd
sbranch
siumo
Module Name
Name
Value
clk
read_add_1...
read_add_2...
read_data_...
read_data_...
reg_write
reset
write_add[4...
write_data[...
k[31:0]
clk
memread
memwrite
reset
write_data[...
shifted_add...
read_data[3...
add[31:0]
k[31:0]
10ns
20ns
30ns
40ns
50ns
60ns
70ns
80ns
90ns
100ns
110ns
120ns
130ns