DIGITAL SYSTEM DESIGN LABORATORY

REPORT LAB 4

BASIC BUILDING BLOCKS OF SINGLE CYCLE MICROPROCESSOR

Name: Bùi Gia Bảo
ID: ITITIU22019

I. LAB OBJECTIVES

This Lab experiments are intended to implement basic building blocks of Single Cycle Microprocessor

II. DESCRIPTION

Single Cycle Microprocessor datapath to be implemented is in figure 2.1.

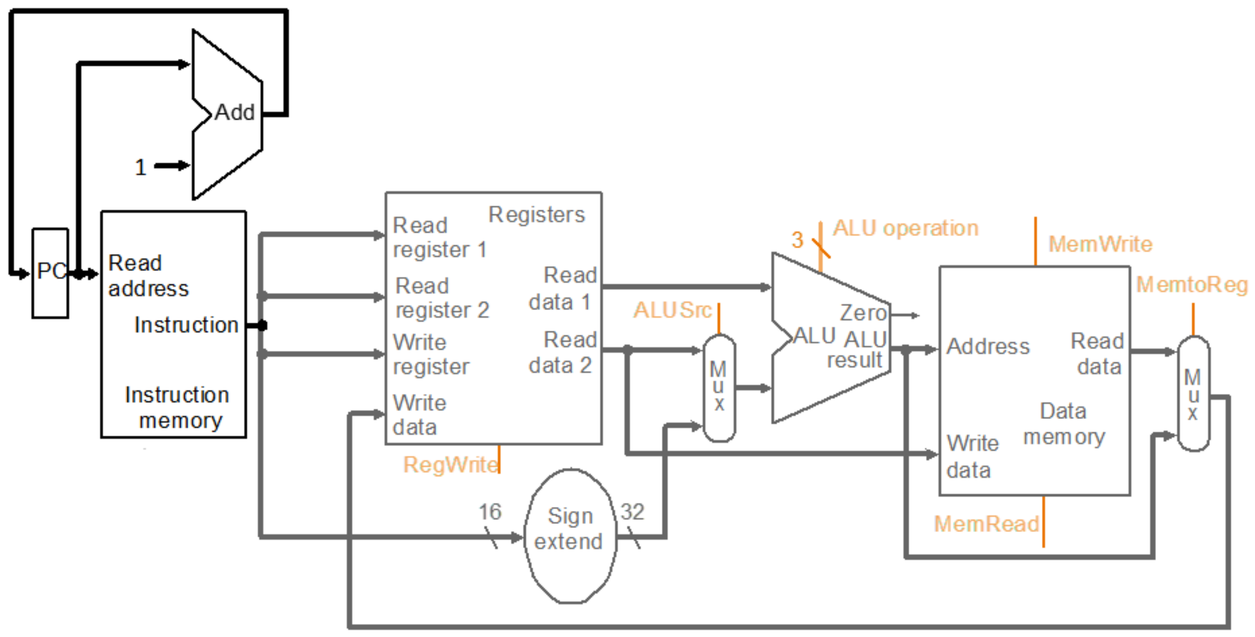
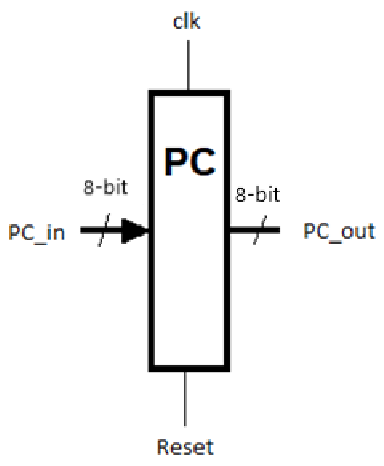


Figure 2.1: Single Cycle Microprocessor DataPath

III. LAB PROCEDURE

III.1 EXPERIMENT NO. 1

III.1.1 AIM: To implement Program Counter



III.1.2 CODE

```
module Program_Counter (clk, reset, PC_in, PC_out);
        input clk, reset;
        input [7:0] PC_in;
        output [7:0] PC_out;
        reg [7:0] PC_out;
        always @ (posedge clk or posedge reset)
        begin
                if(reset==1'b1)
                        PC_out<=8’b0;
                else
                        PC_out<=PC_in;
        end
endmodule
```

III.1.3 LAB ASSIGNMENT

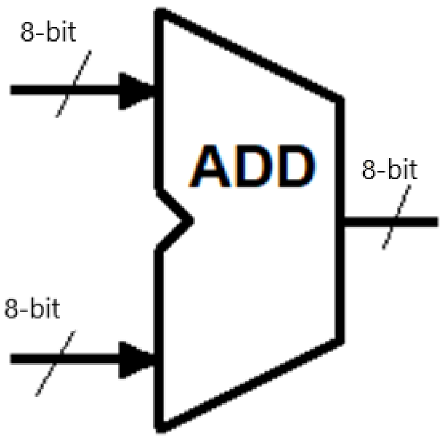
1. Write testbenches to verify above module and attach waveforms.
2. Write the Top level module to implement this module in FPGA KIT

```
module lab4_pc(SW,LEDG,LEDR);
        input[17:0] SW;
        output[7:0] LEDG;
```

```
	output[17:0] LEDR;
	assign LEDR=SW;
	Program_Counter (.clk(SW[0]), .reset(SW[1]), .PC_in(SW[9:2]),
.PC_out(LEDG[7:0]));
endmodule



module Program_Counter (clk, reset, PC_in, PC_out);
	input clk, reset;
	input [7:0] PC_in;
	output [7:0] PC_out;
	reg [7:0] PC_out;
	always @ (posedge clk or posedge reset)
	begin
		if(reset==1'b1)
			PC_out<=8'b0;
		else
			PC_out<=PC_in;
	end
endmodule
```

III.2 EXPERIMENT NO. 2

III.2.1 AIM: To implement 32 bit Adder



III.2.2 CODE

```
module Adder32Bit(input1, input2, out);

            input [7:0] input1, input2;
            output [7:0] out;
            reg [7:0]out;
            always@( input1 or input2)
            begin
                      out  <= input1 + input2;
            end
endmodule
```

III.2.3 LAB ASSIGNMENT

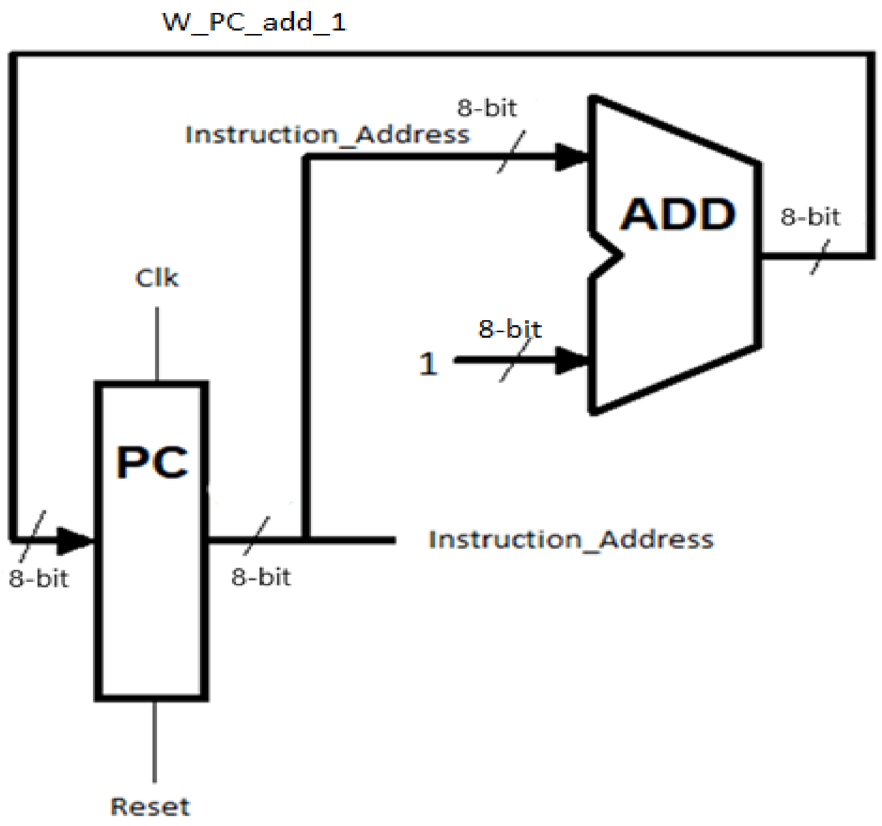
1. Write testbenches to verify above module and attach waveforms.
2. Write the Top level module to implement this module in FPGA KIT

```
module lab4_adder(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5);
           input[17:0] SW;
           output[7:0] LEDG;
           output[17:0] LEDR;

    output[0:6] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5;
    wire [7:0] A,B,SUM;
    assign B = SW[15:8];
    assign A = SW[7:0];
           Adder_8_Bit DUT(.input1(A), .input2(B), .out(SUM));

    HEX_7SEG_DECODE H0(.BIN(A[3:0]),   .SSD(HEX0));
    HEX_7SEG_DECODE H1(.BIN(A[7:4]),   .SSD(HEX1));
    HEX_7SEG_DECODE H2(.BIN(B[3:0]),   .SSD(HEX2));
    HEX_7SEG_DECODE H3(.BIN(B[7:4]),   .SSD(HEX3));
    HEX_7SEG_DECODE H4(.BIN(SUM[3:0]), .SSD(HEX4));
    HEX_7SEG_DECODE H5(.BIN(SUM[7:4]), .SSD(HEX5));

endmodule

module Adder_8_Bit(input1, input2, out);

 input [7:0] input1, input2;
 output [7:0] out;

 assign         out  = input1 + input2;
```

```
endmodule

module HEX_7SEG_DECODE(BIN, SSD);
  input [15:0] BIN;
  output reg [0:6] SSD;

  always begin
    case(BIN)
      0:SSD=7'b0000001;
      1:SSD=7'b1001111;
      2:SSD=7'b0010010;
      3:SSD=7'b0000110;
      4:SSD=7'b1001100;
      5:SSD=7'b0100100;
      6:SSD=7'b0100000;
      7:SSD=7'b0001111;
      8:SSD=7'b0000000;
      9:SSD=7'b0001100;
      10:SSD=7'b0001000;
      11:SSD=7'b1100000;
      12:SSD=7'b0110001;
      13:SSD=7'b1000010;
      14:SSD=7'b0110000;
      15:SSD=7'b0111000;
    endcase
  end
endmodule
```

III.3 EXPERIMENT NO. 3

III.3.1 AIM: To implement the datapath for PC = PC + 1



III.3.2 CODE

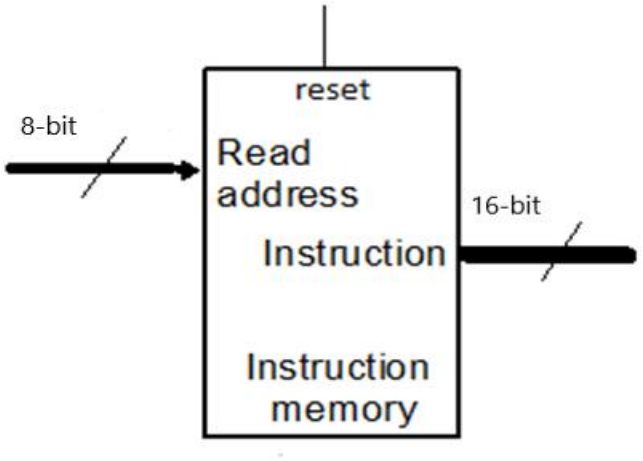
```
module
lab4_ex3(CLOCK_50,SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5);
        input  [17:0]         SW;
        input CLOCK_50;
        output[17:0]          LEDR;
        output[7:0]  LEDG;
        output[6:0]  HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
        wire clk_1Hz;
        clock_divider_1Hz div1 (
      .clk_in(CLOCK_50),
      .reset(SW[1]),         // SW[2] = reset
      .clk_out(clk_1Hz)
   );
        PC_add_1 DUT (.clk(clk_1Hz), .reset(SW[1]),
.Instruction_Address(LEDG[7:0]));

        hex7seg h4(LEDG[3:0], HEX4);
        hex7seg h5(LEDG[7:4], HEX5);

endmodule
```

```
module PC_add_1 (clk, reset, Instruction_Address);
        input clk, reset;
        output [7:0] Instruction_Address;

        wire   [7:0]  add_out;

        PC pc1(.clk(clk), .reset(reset), .PC_in(add_out),
.PC_out(Instruction_Address));
        ADD add1(.input1(Instruction_Address), .input2(8'b0000_0001),
.out(add_out));
endmodule

module hex7seg(
   input  wire [3:0] hex,
   output reg  [6:0] seg
);
   always @(*) begin
      case (hex)
         4'h0: seg = 7'b1000000;
         4'h1: seg = 7'b1111001;
         4'h2: seg = 7'b0100100;
         4'h3: seg = 7'b0110000;
         4'h4: seg = 7'b0011001;
         4'h5: seg = 7'b0010010;
         4'h6: seg = 7'b0000010;
         4'h7: seg = 7'b1111000;
         4'h8: seg = 7'b0000000;
         4'h9: seg = 7'b0010000;
         4'hA: seg = 7'b0001000;
         4'hB: seg = 7'b0000011;
         4'hC: seg = 7'b1000110;
         4'hD: seg = 7'b0100001;
         4'hE: seg = 7'b0000110;
         4'hF: seg = 7'b0001110;
         default: seg = 7'b1111111;
      endcase
   end
endmodule

module ADD(input1, input2, out);
```

```
        input [7:0] input1, input2;
        output [7:0] out;

        assign out = input1 + input2;

endmodule

module PC (clk, reset, PC_in, PC_out);
        input clk, reset;
        input [7:0] PC_in;
        output [7:0] PC_out;
        reg [7:0] PC_out;
        always @ (posedge clk or posedge reset)
        begin
                if(reset==1'b1)
                        PC_out<=8'b0;
                else
                        PC_out<=PC_in;
        end
endmodule



module clock_divider_1Hz (
    input  clk_in,
    input  reset,
    output reg clk_out
);
    reg [25:0] count;

    always @(posedge clk_in or posedge reset) begin
        if (reset) begin
            count   <= 26'd0;
            clk_out <= 1'b0;
        end else begin
            if (count == 26'd24_999_999) begin
                clk_out <= ~clk_out;
                count   <= 26'd0;
            end else begin
                count <= count + 1;
            end
         end
      end
endmodule
```

III.4 EXPERIMENT NO. 4

III.4.1 AIM: To implement the Instruction memory



III.4.2 CODE

```
module Instruction_Memory (read_address, instruction, reset);
        input reset;
        input [9:0] read_address;
        output [15:0] instruction;
        reg [15:0] Imemory [1024:0];

        Imemory[0] = 32'b00100000000010000000000000100000;
        //addi $t0, $zero, 32
        Imemory[1] = 32'b00100000000010010000000000110111;
        //addi $t1, $zero, 55
        Imemory[2] = 32'b00000001000010011000000000100100;
        //and $s0, $t0, $t1
        Imemory[3] = 32'b00000001000010011000000000100101;
        //or $s0, $t0, $t1
        Imemory[4] = 32'b10101100000100000000000000000100;
        //sw $s0, 4($zero)
        Imemory[5] = 32'b10101100000010000000000000001000;
        //sw $t0, 8($zero)
        Imemory[6] = 32'b00000001000010011000100000100000;
        //add $s1, $t0, $t1
```

```
Imemory[7] = 32'b00000001000010011001000000100010;
//sub $s2, $t0, $t1
Imemory[8] = 32'b00010010001100100000000000001001;
//beq $s1, $s2, error0
Imemory[9] = 32'b10001100000100010000000000000100;
//lw $s1, 4($zero)
Imemory[10]= 32'b00110010001100100000000001001000;
//andi $s2, $s1, 48
Imemory[11] =32'b00010010001100100000000000001001;
//beq $s1, $s2, error1
Imemory[12] =32'b10001100000100110000000000001000;
//lw $s3, 8($zero)
Imemory[13] =32'b00010010000100110000000000001010;
//beq $s0, $s3, error2
Imemory[14] =32'b00000010010100011010000000101010;
//slt $s4, $s2, $s1 (Last)
Imemory[15] =32'b00010010100000000000000000001111;
//beq $s4, $0, EXIT
Imemory[16] =32'b00000010001000001001000000100000;
//add $s2, $s1, $0
Imemory[17] =32'b00001000000000000000000000001110;
//j Last
Imemory[18] =32'b00100000000010000000000000000000;
//addi $t0, $0, 0(error0)
Imemory[19] =32'b00100000000010010000000000000000;
//addi $t1, $0, 0
Imemory[20] =32'b00001000000000000000000000011111;
//j EXIT
Imemory[21] =32'b00100000000010000000000000000001;
//addi $t0, $0, 1(error1)
Imemory[22] =32'b00100000000010010000000000000001;
//addi $t1, $0, 1
Imemory[23] =32'b00001000000000000000000000011111;
//j EXIT
Imemory[24] =32'b00100000000010000000000000000010;
//addi $t0, $0, 2(error2)
Imemory[25] =32'b00100000000010010000000000000010;
//addi $t1, $0, 2
Imemory[26] =32'b00001000000000000000000000011111;
//j EXIT
Imemory[27] =32'b00100000000010000000000000000011;
```

```
        //addi $t0, $0, 3(error3)
        Imemory[28] =32'b00100000000010010000000000000011;
        //addi $t1, $0, 3
        Imemory[29] =32'b00001000000000000000000000011111;
        //j EXIT
        end
endmodule
```

reset	Read_address	PC_out
1	32’b0	32'b00100000000010000000000000100000
0	32’b0	32'b00100000000010000000000000100000
0	32’b0000….. 0011	32'b00000001000010011000000000100101
0	32’b0000 …….0111	32'b00000001000010011001000000100010

III.4.3 LAB ASSIGNMENT

1. Write testbenches to verify above module and attach waveforms.
2. Write the Top level module to implement this module in FPGA KIT

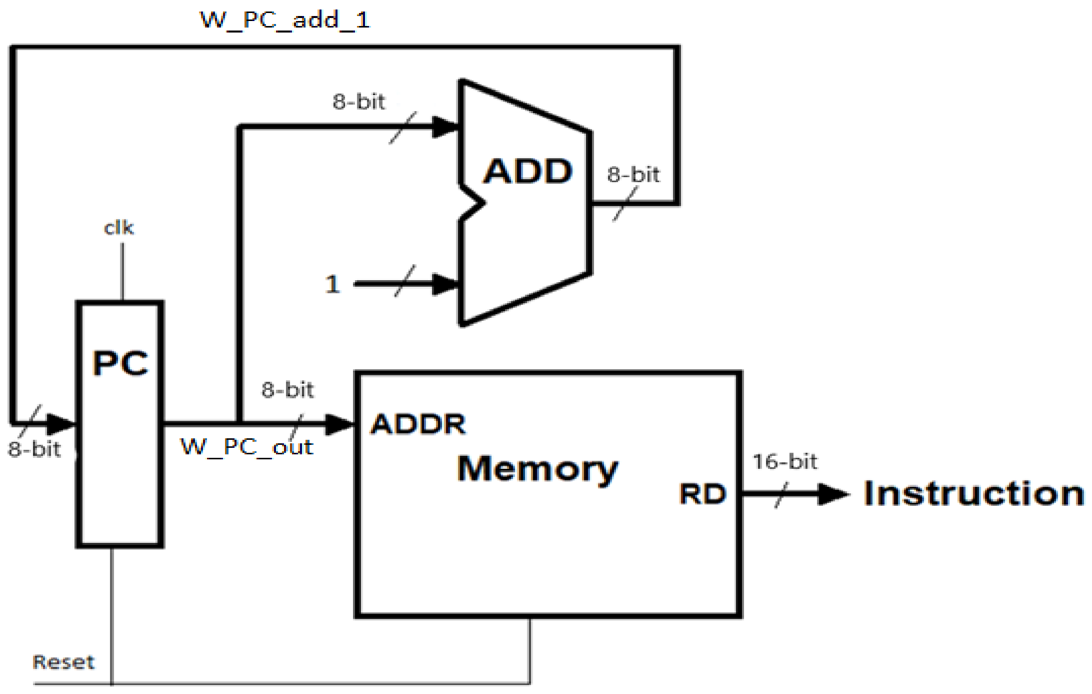
```
module lab4_ex4_v2(SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5);
        input[17:0] SW;
        output[7:0] LEDG;
        output[17:0] LEDR;
        output[0:6] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5;
        wire[31:0] w_out_instruction;

        assign LEDR[15:0] = SW[17] ? w_out_instruction[31:16] :
w_out_instruction[15:0];

        Instruction_Memory (.read_address(SW[5:0]),
.instruction(w_out_instruction));


endmodule



module Instruction_Memory (read_address, instruction);
input [7:0] read_address;
output [31:0] instruction;
reg [31:0] Imemory [63:0];
assign instruction=Imemory[read_address];
initial begin
Imemory[0] = 32'b00100000000010000000000000100000;
//addi $t0, $zero, 32
```

```
Imemory[1] = 32'b00100000000010010000000000110111;
//addi $t1, $zero, 55
Imemory[2] = 32'b00000001000010011000000000100100;
//and $s0, $t0, $t1
Imemory[3] = 32'b00000001000010011000000000100101;
//or $s0, $t0, $t1
Imemory[4] = 32'b10101100000100000000000000000100;
//sw $s0, 4($zero)
Imemory[5] = 32'b10101100000010000000000000001000;
//sw $t0, 8($zero)
Imemory[6] = 32'b00000001000010011000100000100000;
//add $s1, $t0, $t1
Imemory[7] = 32'b00000001000010011001000000100010;
//sub $s2, $t0, $t1
Imemory[8] = 32'b00010010001100100000000000001001;
//beq $s1, $s2, error0
Imemory[9] = 32'b10001100000100010000000000000100;
//lw $s1, 4($zero)
Imemory[10]= 32'b00110010001100100000000001001000;
//andi $s2, $s1, 48
Imemory[11] =32'b00010010001100100000000000001001;
//beq $s1, $s2, error1
Imemory[12] =32'b10001100000100110000000000001000;
//lw $s3, 8($zero)
Imemory[13] =32'b00010010000100110000000000001010;
//beq $s0, $s3, error2
Imemory[14] =32'b00000010010100011010000000101010;
//slt $s4, $s2, $s1 (Last)
Imemory[15] =32'b00010010100000000000000000001111;
//beq $s4, $0, EXIT
Imemory[16] =32'b00000010001000001001000000100000;
//add $s2, $s1, $0
Imemory[17] =32'b00001000000000000000000000001110;
//j Last
Imemory[18] =32'b00100000000010000000000000000000;
//addi $t0, $0, 0(error0)
Imemory[19] =32'b00100000000010010000000000000000;
//addi $t1, $0, 0
Imemory[20] =32'b00001000000000000000000000011111;
//j EXIT
Imemory[21] =32'b00100000000010000000000000000001;
```

```
//addi $t0, $0, 1(error1)
Imemory[22] =32'b00100000000100100000000000000001;
//addi $t1, $0, 1
Imemory[23] =32'b00001000000000000000000000011111;
//j EXIT
Imemory[24] =32'b00100000000100000000000000000010;
//addi $t0, $0, 2(error2)
Imemory[25] =32'b00100000000100100000000000000010;
//addi $t1, $0, 2
Imemory[26] =32'b00001000000000000000000000011111;
//j EXIT
Imemory[27] =32'b00100000000100000000000000000011;
//addi $t0, $0, 3(error3)
Imemory[28] =32'b00100000000100100000000000000011;
//addi $t1, $0, 3
Imemory[29] =32'b00001000000000000000000000011111;
//j EXIT
end
endmodule
```

III.5 EXPERIMENT NO. 5

III.5.1 AIM: To implement the Instruction memory datapath



III.5.2 CODE

```
module Instruction_Datapath (clk, reset, Instruction);
        input clk, reset;
        output [15:0] Instruction;

endmodule
```

III.5.3 LAB ASSIGNMENT

1. Write testbenches to verify above module and attach waveforms.
2. Write the Top level module to implement this module in FPGA KIT

```
module
lab4_ex5(CLOCK_50,SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,
HEX6,HEX7);
        input  [17:0]        SW;
        output[17:0]         LEDR;
        output[7:0]  LEDG;
        output[7:0]  HEX7,HEX6,HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;
        input CLOCK_50;
        wire clk_1Hz;
        wire [31:0] OUT;
        clock_divider_1Hz div1 (
      .clk_in(CLOCK_50),
      .reset(SW[0]),          // SW[2] = reset
      .clk_out(clk_1Hz)
   );

        Instruction_Datapath DUT (.clk(clk_1Hz),
.reset(SW[0]),.instruction_address(LEDG), .Instruction(OUT));

        wire   SEL;
        assign SEL = SW[17];
        assign LEDR = SEL ? OUT[15:0] : OUT[31:16];

        hex7seg h0(OUT[3:0], HEX0);
        hex7seg h1(OUT[7:4], HEX1);
        hex7seg h2(OUT[11:8], HEX2);
        hex7seg h3(OUT[15:12], HEX3);
        hex7seg h4(OUT[19:16], HEX4);
        hex7seg h5(OUT[23:20], HEX5);
        hex7seg h6(OUT[27:24], HEX6);
        hex7seg h7(OUT[31:28], HEX7);
```

```
endmodule

module Instruction_Datapath (clk, reset, instruction_address, Instruction);
        input clk, reset;
        output [31:0] Instruction;
        output[7:0]   instruction_address;

        PC_ADD DUT1(.clk(clk), .reset(reset),
.Instruction_Address(instruction_address));
        MEMORY DUT2(.read_address(instruction_address),
.instruction(Instruction), .reset(reset));
endmodule

module hex7seg(
   input  wire [3:0] hex,
   output reg  [7:0] seg
);
   always @(*) begin
      case (hex)
         4'h0: seg = 8'b11000000;
         4'h1: seg = 8'b11111001;
         4'h2: seg = 8'b10100100;
         4'h3: seg = 8'b10110000;
         4'h4: seg = 8'b10011001;
         4'h5: seg = 8'b10010010;
         4'h6: seg = 8'b10000010;
         4'h7: seg = 8'b11111000;
         4'h8: seg = 8'b10000000;
         4'h9: seg = 8'b10010000;
         4'hA: seg = 8'b10001000;
         4'hB: seg = 8'b10000011;
         4'hC: seg = 8'b11000110;
         4'hD: seg = 8'b10100001;
         4'hE: seg = 8'b10000110;
         4'hF: seg = 8'b10001110;
         default: seg = 8'b11111111;
      endcase
   end
endmodule
```

```
module PC_ADD (clk, reset, Instruction_Address);
        input clk, reset;
        output [7:0] Instruction_Address;

        wire   [7:0]  add_out;

        PC pc1(.clk(clk), .reset(reset), .PC_in(add_out),
.PC_out(Instruction_Address));
        ADD add1(.input1(Instruction_Address), .input2(8'b0000_0001),
.out(add_out));
endmodule

module PC (clk, reset, PC_in, PC_out);
        input clk, reset;
        input [7:0] PC_in;
        output [7:0] PC_out;
        reg [7:0] PC_out;
        always @ (posedge clk or posedge reset)
        begin
                if(reset==1'b1)
                        PC_out<=8'b0;
                else
                        PC_out<=PC_in;
        end
endmodule

module ADD(input1, input2, out);

        input [7:0] input1, input2;
        output [7:0] out;

        assign out = input1 + input2;

endmodule


module MEMORY (read_address, instruction, reset);
        input           reset;
        input  [7:0] read_address;
        outputreg     [31:0] instruction;
```

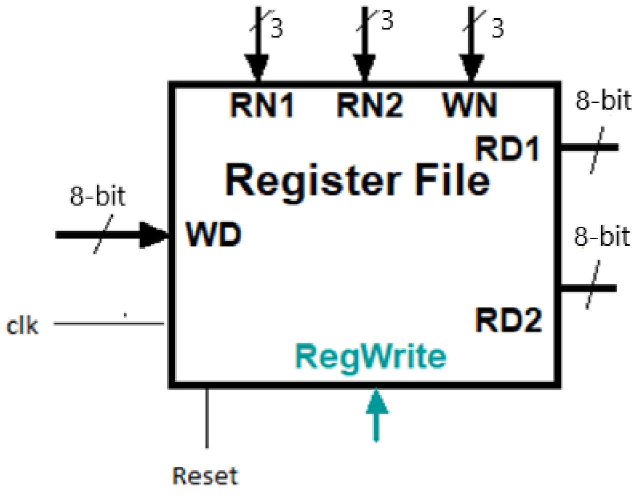
```
    reg [31:0] Imemory [1024:0];

    always @(*) begin
            Imemory[0] = 32'b00100000000010000000000000100000;
            //addi $t0, $zero, 32
            Imemory[1] = 32'b00100000000010010000000000110111;
            //addi $t1, $zero, 55
            Imemory[2] = 32'b00000001000010011000000000100100;
            //and $s0, $t0, $t1
            Imemory[3] = 32'b00000001000010011000000000100101;
            //or $s0, $t0, $t1
            Imemory[4] = 32'b10101100000100000000000000000100;
//sw $s0, 4($zero)
            Imemory[5] = 32'b10101100000010000000000000001000;
            //sw $t0, 8($zero)
            Imemory[6] = 32'b00000001000010011000100000100000;
            //add $s1, $t0, $t1
            Imemory[7] = 32'b00000001000010011001000000100010;
            //sub $s2, $t0, $t1
            Imemory[8] = 32'b00010010001100100000000000001001;
            //beq $s1, $s2, error0
            Imemory[9] = 32'b10001100000100010000000000000100;
            //lw $s1, 4($zero)
            Imemory[10]= 32'b00110010001100100000000001001000;
            //andi $s2, $s1, 48
            Imemory[11] =32'b00010010001100100000000000001001;
            //beq $s1, $s2, error1
            Imemory[12] =32'b10001100000100110000000000001000;
            //lw $s3, 8($zero)
            Imemory[13] =32'b00010010000100110000000000001010;
            //beq $s0, $s3, error2
            Imemory[14] =32'b00000010010100011010000000101010;
            //slt $s4, $s2, $s1 (Last)
            Imemory[15] =32'b00010010100000000000000000001111;
            //beq $s4, $0, EXIT
            Imemory[16] =32'b00000010001000001001000000100000;
            //add $s2, $s1, $0
            Imemory[17] =32'b00001000000000000000000000001110;
            //j Last
            Imemory[18] =32'b00100000000010000000000000000000;
            //addi $t0, $0, 0(error0)
```

```
			Imemory[19] =32'b00100000000010010000000000000000;
			//addi $t1, $0, 0
			Imemory[20] =32'b00001000000000000000000000011111;
			//j EXIT
			Imemory[21] =32'b00100000000010000000000000000001;
			//addi $t0, $0, 1(error1)
			Imemory[22] =32'b00100000000010010000000000000001;
			//addi $t1, $0, 1
			Imemory[23] =32'b00001000000000000000000000011111;
			//j EXIT
			Imemory[24] =32'b00100000000010000000000000000010;
			//addi $t0, $0, 2(error2)
			Imemory[25] =32'b00100000000010010000000000000010;
			//addi $t1, $0, 2
			Imemory[26] =32'b00001000000000000000000000011111;
			//j EXIT
			Imemory[27] =32'b00100000000010000000000000000011;
			//addi $t0, $0, 3(error3)
			Imemory[28] =32'b00100000000010010000000000000011;
			//addi $t1, $0, 3
			Imemory[29] =32'b00001000000000000000000000011111;
			//j EXIT
	end

	always @(*) begin
			if(reset)
				instruction = 31'b0;
			else
				instruction = Imemory[read_address];
	end
endmodule

module clock_divider_1Hz (
    input  clk_in,
    input  reset,
    output reg clk_out
);
    reg [25:0] count;

    always @(posedge clk_in or posedge reset) begin
        if (reset) begin
```

```
            count   <= 26'd0;
            clk_out <= 1'b0;
        end else begin
            if (count == 26'd24_999_999) begin
                clk_out <= ~clk_out;
                count   <= 26'd0;
            end else begin
                count <= count + 1;
            end
        end
    end
endmodule
```

III.6 EXPERIMENT NO. 6

III.6.1 AIM: To implement the Register File



III.6.2 CODE

```
module Register_File (read_addr_1, read_addr_2, write_addr, read_data_1, read_data_2,
write_data, RegWrite, clk, reset);
        input [2:0] read_addr_1, read_addr_2, write_addr;
        input [7:0] write_data;
        input clk, reset, RegWrite;
        output [7:0] read_data_1, read_data_2;

        reg [7:0] Regfile [7:0];
        integer k;
        assign read_data_1 = Regfile[read_addr_1];


        assign read_data_2 = Regfile[read_addr_2];




        always @(posedge clk or posedge reset)
        begin
                if (reset==1'b1)
                begin
                        for (k=0; k<8; k=k+1)
                        begin
                                Regfile[k] = 8'b0;
                        end
                end

                else if (RegWrite == 1'b1) Regfile[write_addr] = write_data;
        end

endmodule
```

reset	clk	read_addr_1	read_data_1	read_addr_2	read_data_2	RegWrite	write_addr	write_data
1	x	x	32'b0	x	32'b0	x	x	x
0	↑	32'd3	32'b0	32'd4	32'b0	1	32'b001	32'b0111
0	↑	32'd3	32'b0	32'd4	32'b0	1	32'b010	32'b1000
0	↑	32'd3	32'b0	32'd4	32'b0	1	32'b011	32'b1001
0	↑	32'd3	32'b0	32'd4	32'b0	1	32'b100	32'b1010
0	↑	32'd3	32'b0	32'b00	32'b0	1	32'b101	32'b1011
0	↑	32'b001	32'b0111	32'b101	32'b1011	0	x	x
0	↑	32'b011	32'b1001	32'b100	32'b1010	0	x	x
0	↑	32'b101	32'b1011	32'b101	32'b1011	0	x	x
0	↑	32'b100	32'b1010	32'b100	32'b1010	0	x	x

III.6.3 LAB ASSIGNMENT

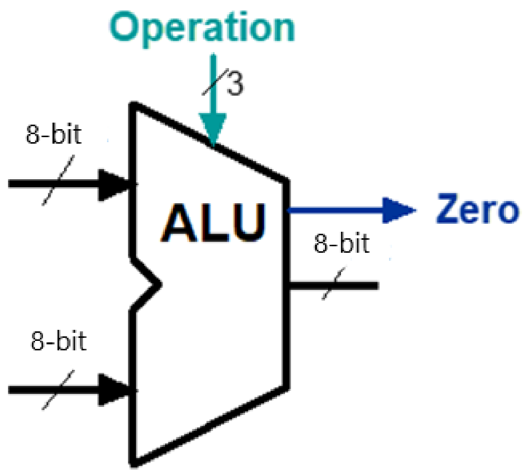
1. Write testbenches to verify above module and attach waveforms.
2. Write the Top level module to implement this module in FPGA KIT

```
module
lab4_ex6_v2(KEY,SW,LEDG,LEDR,HEX0,HEX1,HEX2,HEX3,HEX4,HEX5);
        input[3:0] KEY;
        input[17:0] SW;
        output[7:0] LEDG;
        output[17:0] LEDR;
      output[0:6] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5;
```

```
    wire [7:0] W_LED_HEX_1_0,W_LED_HEX_3_2;
      assign LEDR = SW;
    Register_File DUT(  .read_addr_1(SW[17:15]),
                                                .read_addr_2(SW[14:12]),
                                                .write_addr(SW[11:9]),
                                                .read_data_1(W_LED_HEX_1_0),
                                                .read_data_2(W_LED_HEX_3_2),
                                                .write_data(SW[8:1]),
                                                .RegWrite(SW[0]),
                                                .clk(KEY[0]),
                                                .reset(KEY[1]));

    HEX_7SEG_DECODE H0(.BIN(W_LED_HEX_1_0[3:0]), .SSD(HEX0));
    HEX_7SEG_DECODE H1(.BIN(W_LED_HEX_1_0[7:4]), .SSD(HEX1));
    HEX_7SEG_DECODE H2(.BIN(W_LED_HEX_3_2[3:0]), .SSD(HEX2));
    HEX_7SEG_DECODE H3(.BIN(W_LED_HEX_3_2[7:4]), .SSD(HEX3));

endmodule

module HEX_7SEG_DECODE(BIN, SSD);
  input [3:0] BIN;
  output reg [0:6] SSD;
  always begin
    case(BIN)
      0:SSD=7'b0000001;
      1:SSD=7'b1001111;
      2:SSD=7'b0010010;
      3:SSD=7'b0000110;
      4:SSD=7'b1001100;
      5:SSD=7'b0100100;
      6:SSD=7'b0100000;
      7:SSD=7'b0001111;
      8:SSD=7'b0000000;
      9:SSD=7'b0001100;
      10:SSD=7'b0001000;
      11:SSD=7'b1100000;
      12:SSD=7'b0110001;
      13:SSD=7'b1000010;
      14:SSD=7'b0110000;
      15:SSD=7'b0111000;
    endcase
  end
endmodule

module Register_File (read_addr_1, read_addr_2, write_addr, read_data_1,
read_data_2, write_data, RegWrite, clk, reset);
        input [2:0] read_addr_1, read_addr_2, write_addr;
        input [7:0] write_data;
        input clk, reset, RegWrite;
        output [7:0] read_data_1, read_data_2;

        reg [7:0] Regfile [7:0];
        integer k;

        assign read_data_1 = Regfile[read_addr_1];


        assign read_data_2 = Regfile[read_addr_2];

        always @(posedge clk or negedge reset)
        begin
                if (reset==1'b0)
                begin
                        for (k=0; k<8; k=k+1)
                        begin
                                Regfile[k] = 8'b0;
                        end
                end

                else if (RegWrite == 1'b1) Regfile[write_addr] = write_data;
        end

endmodule
```

III.7 EXPERIMENT NO. 7

III.7.1 AIM: To implement the ALU



III.7.2 CODE

```
module alu(
        input [2:0] alufn,
        input [7:0] ra,
        input [7:0] rb_or_imm,
        output reg [7:0] aluout,
        output reg zero);

        parameter       ALU_OP_ADD      = 3'b000,
                        ALU_OP_SUB      = 3'b001,
                        ALU_OP_AND      = 3'b010,
                        ALU_OP_OR       = 3'b011,
                        ALU_OP_NOT_A    = 3'b100,
                        ALU_OP_LW       = 3'b101,
                        ALU_OP_SW       = 3'b110,
                        ALU_OP_BEQ      = 3'b111;
        always @(*)
        begin
                case(alufn)
                        ALU_OP_ADD      : aluout = ra + rb_or_imm;
                        ALU_OP_SUB      : aluout = ra - rb_or_imm;
                        ALU_OP_AND      : aluout = ra & rb_or_imm;
                        ALU_OP_OR       : aluout = ra | rb_or_imm;
                        ALU_OP_NOT_A    : aluout = ~ ra;
                        ALU_OP_LW       : aluout = ra + rb_or_imm;
                        ALU_OP_SW       : aluout = ra + rb_or_imm;
                        ALU_OP_BEQ      : begin
                                            zero = (ra==rb_or_imm)?1'b1:1'b0;
                                            aluout = ra - rb_or_imm;
                                          end
                endcase
        end
```

endmodule

alufn	ra	rb_or_imm	aluout	zero
3'b000	32’d8	32’d2	32’d10	0
3'b001	32’d8	32’d2	32’d6	0
3'b010	32’b1000	32’b0010	32’b0000	0
3'b011	32’b1000	32’b0010	32’b1010	0
3'b100	32’b1000	32’d2	32’b0111	0
3'b101	32’d8	32’d2	32’d10	0
3'b110	32’d8	32’d2	32’d10	0
3'b111	32’d8	32’d2	32’d6	0
3'b111	32’d8	32’d8	32’d0	1

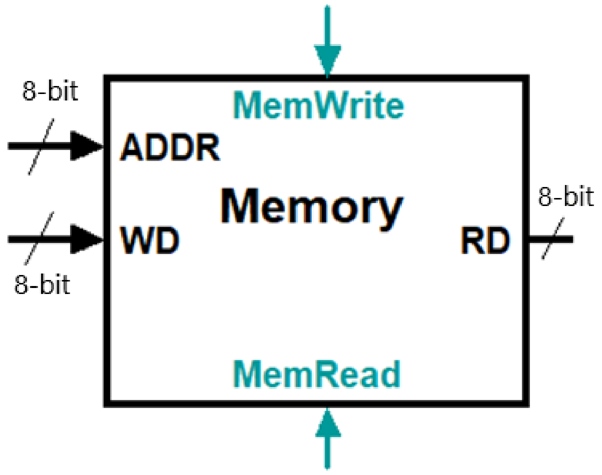
III.7.3 LAB ASSIGNMENT

1. Write testbenches to verify above module and attach waveforms.
2. Write the Top level module to implement this module in FPGA KIT

```
module lab4_ex7(LEDR, LEDG, SW);
        input  [17:0]         SW;
        output[17:0]          LEDR;
        output[7:0]   LEDG;
        assign LEDR =SW;

        alu(.alufn(SW[17:15]),
                .ra(SW[6:0]),
                .rb_or_imm(SW[14:7]),
                .aluout(LEDG[6:0]),
                .zero(LEDG[7]));


endmodule
```

III.8 EXPERIMENT NO. 8

III.8.1 AIM: To implement the 32 bit RAM



III.8.2 CODE

```
module Data_Memory (addr, write_data, read_data, clk, reset, MemRead, MemWrite);
        input [7:0] addr;
        input [7:0] write_data;
        output [7:0] read_data;
        input clk, reset, MemRead, MemWrite;
        reg [7:0] DMemory [7:0];
        integer k;
        assign read_data = (MemRead) ? DMemory[addr] : 8'bx;




        always @(posedge clk or posedge reset)
        begin
                if (reset == 1'b1)
                        begin
                                for (k=0; k<8; k=k+1)
                                  begin
                                     DMemory[k] = 8'b0;
                                  end
                        end
                else
                        if (MemWrite) DMemory[addr] = write_data;
        end
endmodule
```

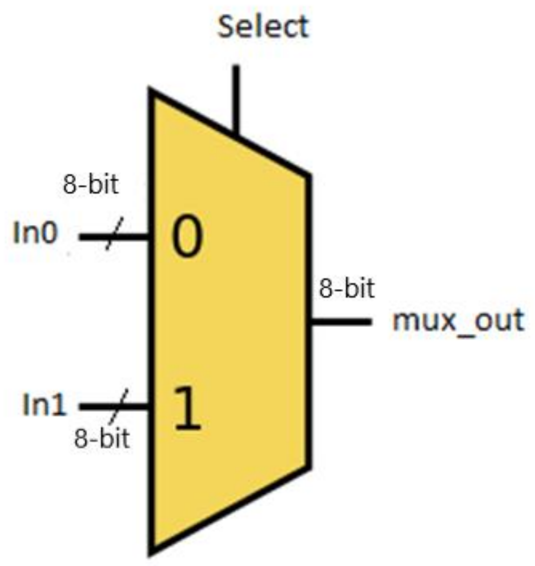
III.8.3 LAB ASSIGNMENT

1. Write testbenches to verify above module and attach waveforms.
2. Write the Top level module to implement this module in FPGA KIT

```
module lab4_ex8(SW,LEDG,LEDR);
        input  [17:0]          SW;
        output[17:0]           LEDR;
        output[7:0]   LEDG;
        assign LEDR = SW;

        Data_Memory (      .addr(SW[6:0]),
                                    .write_data(SW[13:7]),
                                    .read_data(LEDG[6:0]),
                                    .clk(SW[17]),
                                    .reset(SW[16]),
                                    .MemRead(SW[15]),
                                    .MemWrite(SW[14]));


Endmodule
```

III.9 EXPERIMENT NO. 9

III.9.1 AIM: To implement Multiplexer



III.9.2 CODE

```
module Mux_N_bit (in0, in1, mux_out, select);
        parameter N = 32;
        input [N-1:0] in0, in1;
        output [N-1:0] mux_out;
        input control;
        assign mux_out = select? in1: in0 ;
endmodule
```

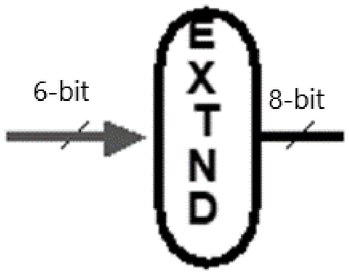
III.9.3 LAB ASSIGNMENT

1. Write testbenches to verify above module and attach waveforms.
2. Write the Top level module to implement this module in FPGA KIT

```
module lab4_ex9(SW, LEDR, LEDG);
        input[17:0] SW;
        output[7:0] LEDG;
        output[17:0] LEDR;


        Mux_N_bit (.in0(SW[7:0]), .in1(SW[15:8]), .mux_out(LEDG[7:0]), .select(SW[16]));

endmodule


module Mux_N_bit (in0, in1, mux_out, select);
        parameter N = 8;
        input select;
        input [N-1:0] in0, in1;
        output [N-1:0] mux_out;
        assign mux_out = select? in1: in0 ;
endmodule
```

III.10 EXPERIMENT NO. 10

III.10.1 AIM: To implement Sign_Extension



III.10.2 CODE

```
module Sign_Extension (sign_in, sign_out);
        input [15:0] sign_in;
        output [31:0] sign_out;
        assign sign_out[15:0]=sign_in[15:0];
```

```
    assign sign_out[31:16]=sign_in[15]?{16{1'b1}}:16'b0;
endmodule
```

III.10.3 LAB ASSIGNMENT

1. Write testbenches to verify above module and attach waveforms.
2. Write the Top level module to implement this module in FPGA KIT

```
module lab4_ex10(SW, LEDR,LEDG);
    input[17:0] SW;
    output[7:0] LEDG;
    output[17:0] LEDR;


    Sign_Extension (.sign_in(SW[4:0]), .sign_out(LEDG[7:0]));
endmodule


module Sign_Extension (sign_in, sign_out);
    input [4:0] sign_in;
    output [7:0] sign_out;
    assign sign_out[4:0]=sign_in[4:0];
    assign sign_out[7:5]=sign_in[4]?{3{1'b1}}:3'b0;
endmodule
```