|  |
| --- |
| **DIGITAL SYSTEM DESIGN LABORATORY** |
| **LAB 5** |

**BASIC BUILDING BLOCKS OF SINGLE CYCLE MICROPROCESSOR**

### 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

|  |
| --- |
| **Top module** |
| module lab5\_ex1(SW,LEDG,LEDR);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;  assign LEDR=SW;  Program\_Counter(.clk(SW[9]),.reset(SW[8]),.PC\_in(SW[7:0]),.PC\_out(LEDG));    endmodule |
| **Testbench** |
| module test\_Program\_Counter;  reg t\_clk, t\_reset;  reg [7:0] t\_PC\_in;  wire [7:0] t\_PC\_out;    // Instantiate Device Under Test (DUT)  Program\_Counter DUT (  .clk(t\_clk),  .reset(t\_reset),  .PC\_in(t\_PC\_in),  .PC\_out(t\_PC\_out)  );    // Clock Generation  always #5 t\_clk = ~t\_clk;    // Test Sequence  initial begin  // Initialize signals  t\_clk = 0;  t\_reset = 1;  t\_PC\_in = 8'b00000000;    // Monitor changes  $monitor($time, " reset=%b | clk=%b | PC\_in=%b | PC\_out=%b |", t\_reset, t\_clk, t\_PC\_in, t\_PC\_out);    // Reset sequence  #10 t\_reset = 0;    // Random input tests  repeat(20) begin  #10 t\_PC\_in = $random;  end    // Additional reset test  #10 t\_reset = 1;  #10 t\_reset = 0;    // Finish simulation  #50 $finish;  end  endmodule |
|  |
|  |

### III.2 EXPERIMENT NO. 2

##### III.2.1 AIM: To implement 32 bit Adder

##### Diagram Description automatically generated

**III.2.2 CODE**

module Adder8Bit(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

|  |
| --- |
| **Top module** |
| module lab5\_ex2(SW,LEDR,LEDG);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;    assign LEDR=SW;    Adder8Bit(.input1(SW[17:10]),.input2(SW[7:0]),.out(LEDG));  endmodule |
| **Testbench** |
| `timescale 1ns/1ps  module Adder8Bit\_tb;  reg [7:0] input1, input2;  wire [7:0] out;    Adder8Bit uut (  .input1(input1),  .input2(input2),  .out(out)  );  initial begin  // Test Case 1  input1 = 8'd15;  input2 = 8'd20;  #10;  $display("Test Case 1: input1 = %d, input2 = %d, out = %d", input1, input2, out);  // Test Case 2  input1 = 8'd100;  input2 = 8'd28;  #10;  $display("Test Case 2: input1 = %d, input2 = %d, out = %d", input1, input2, out);  // Test Case 3  input1 = 8'd254;  input2 = 8'd1;  #10;  $display("Test Case 3: input1 = %d, input2 = %d, out = %d", input1, input2, out);  // Test Case 4  input1 = 8'd0;  input2 = 8'd0;  #10;  $display("Test Case 4: input1 = %d, input2 = %d, out = %d", input1, input2, out);    $finish;  end  endmodule |
|  |
|  |

### III.3 EXPERIMENT NO. 3

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

##### Diagram Description automatically generated

**III.3.2 CODE**

module PC\_ADD\_1(clk, reset, instruction\_address);

input clk, reset;

output [7:0] instruction\_address;

wire [7:0]W\_PC\_add\_1;

Program\_Counter C1(.clk(clk), .reset(reset), .PC\_in(W\_PC\_add\_1), .PC\_out(instruction\_address));

Adder32Bit C2(.input1(8'b1), .input2(instruction\_address), .out(W\_PC\_add\_1));

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

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.3.3 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

|  |
| --- |
| **Top module** |
| module lab5\_ex3(SW,LEDR,LEDG);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;    assign LEDR=SW;    PC\_ADD\_1(.clk(SW[0]),.reset(SW[1]),.instruction\_address(LEDG));  endmodule |
| **Testbench** |
| module testbench\_PC\_add\_1;  reg clk, reset;  wire [7:0] instruction\_address;  PC\_ADD\_1 DUT(  .clk(clk),  .reset(reset),  .instruction\_address(instruction\_address)  );  always #5 clk = ~clk;  initial begin  clk = 0;  reset = 1;  #5 reset = 0;  $monitor($time, " reset=%b, clk=%b, instruction\_address=%b", reset, clk, instruction\_address);  #50 reset=1;  #5 reset=0;    #100 $finish;  end  endmodule |
|  |
|  |

### III.4 EXPERIMENT NO. 4

##### III.4.1 AIM: To implement the Instruction memory

##### Diagram Description automatically generated

**III.4.2 CODE**

module Instruction\_Memory (read\_address, instruction, reset);

input reset;

input [7:0] read\_address;

output [31:0] instruction;

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

integer k;

// I-MEM in this case is addressed by word, not by byte

assign instruction = Imemory[read\_address];

always @(posedge reset)

begin

for (k=0; k<32; k=k+1)

begin

// here Out changes k=0 to k=16

Imemory[k] = 32'b0;

end

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

|  |
| --- |
| **Top module** |
| module lab5\_ex4(SW,LEDR,LEDG,HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);  input [17:0] SW;  output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;  output [17:0] LEDR;  output [7:0] LEDG;  wire [31:0] W\_LED\_SEG;  assign LEDR = SW;  hex\_ssd (W\_LED\_SEG[3:0], HEX0);  hex\_ssd (W\_LED\_SEG[7:4], HEX1);  hex\_ssd (W\_LED\_SEG[11:8],HEX2);  hex\_ssd (W\_LED\_SEG[15:12],HEX3);  hex\_ssd (W\_LED\_SEG[19:16],HEX4);  hex\_ssd (W\_LED\_SEG[23:20],HEX5);  hex\_ssd (W\_LED\_SEG[27:24],HEX6);  hex\_ssd (W\_LED\_SEG[31:28],HEX7);    Instruction\_Memory (.read\_address(SW[7:0]),.instruction(W\_LED\_SEG),.reset(SW[8]));  endmodule  module hex\_ssd (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 |
| **Testbench** |
| module testbench\_Instruction\_Memory;  reg reset;  reg [7:0] read\_address;  wire [31:0] instruction;  Instruction\_Memory IM (  .reset(reset),  .read\_address(read\_address),  .instruction(instruction)  );  initial begin    $monitor("Time=%0d reset=%b read\_address=%b instruction=%h", $time, reset, read\_address, instruction);  reset = 0;  read\_address = 8'b00000000;  // Test Case 1  #5 reset = 1;  #10 reset = 0;  // Test Case 2  #10 read\_address = 8'b00000000; // Address 0  #10 read\_address = 8'b00000001; // Address 1  #10 read\_address = 8'b00000010; // Address 2  #10 read\_address = 8'b00000011; // Address 3  #10 read\_address = 8'b00000100; // Address 4  #10 read\_address = 8'b00000101; // Address 5  #10 read\_address = 8'b00000110; // Address 6  #10 read\_address = 8'b00000111; // Address 7  #10 read\_address = 8'b00011101; // Address 29    // End simulation  #50 $finish;  end  endmodule |
|  |
|  |

### III.5 EXPERIMENT NO. 5

##### III.5.1 AIM: To implement the Instruction memory datapath

##### Diagram Description automatically generated

**III.5.2 CODE**

module data\_path\_instruction\_mem(reset,clk,read\_address,instruction);

input reset,clk;

output [7:0] read\_address;

output [31:0] instruction;

wire [7:0] W\_PC\_add\_1;

Program\_Counter C1(.clk(clk), .reset(reset), .PC\_in(W\_PC\_add\_1), .PC\_out(read\_address));

Adder8Bit C2(.input1(8'b1), .input2(read\_address), .out(W\_PC\_add\_1));

Instruction\_Memory C3(.reset(reset),.read\_address(read\_address), .instruction(instruction));

endmodule

module Instruction\_Memory (reset,read\_address, instruction);

input reset;

input [7:0] read\_address;

output [31:0] instruction;

reg [31:0] Imemory [255:0];

integer k;

// I-MEM in this case is addressed by word, not by byte

assign instruction = Imemory[read\_address];

always @(posedge reset)

begin

for (k=0; k<32; k=k+1)

begin

// here Out changes k=0 to k=16

Imemory[k] = 32'b0;

end

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

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

module Adder8Bit(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.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

|  |
| --- |
| **Top module** |
| module lab5\_ex5(SW,LEDR,LEDG,HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0);  input [17:0] SW;  output [0:6] HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0;  output [17:0] LEDR;  output [7:0] LEDG;  wire [31:0] W\_LED\_SEG;  assign LEDR = SW;  hex\_ssd (W\_LED\_SEG[3:0], HEX0);  hex\_ssd (W\_LED\_SEG[7:4], HEX1);  hex\_ssd (W\_LED\_SEG[11:8],HEX2);  hex\_ssd (W\_LED\_SEG[15:12],HEX3);  hex\_ssd (W\_LED\_SEG[19:16],HEX4);  hex\_ssd (W\_LED\_SEG[23:20],HEX5);  hex\_ssd (W\_LED\_SEG[27:24],HEX6);  hex\_ssd (W\_LED\_SEG[31:28],HEX7);    data\_path\_instruction\_mem(.reset(SW[17]),.clk(SW[16]),.read\_address(LEDG[7:0]),.instruction(W\_LED\_SEG));  endmodule  module hex\_ssd (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 |
| **Testbench** |
| module data\_path\_instruction\_mem\_tb;  reg clk, reset;  wire [7:0] read\_address;  wire [31:0] instruction;  data\_path\_instruction\_mem uut (  .reset(reset),  .clk(clk),  .read\_address(read\_address),  .instruction(instruction)  );  always begin  #5 clk = ~clk;  end  initial begin  clk = 0;  reset = 0;    #5 reset = 1;  #10 reset = 0;  #20;  #30 reset = 1;  #10 reset = 0;  #40;  #50;  $finish;  end  initial begin  $monitor("At time %t, reset = %b, read\_address = %b, instruction = %b",  $time, reset, read\_address, instruction);  end  endmodule |
|  |
|  |

### III.6 EXPERIMENT NO. 6

##### III.6.1 AIM: To implement the Register File

##### Diagram, schematic Description automatically generated

**III.6.2 CODE**

module Register\_File(

input [2:0] read\_addr\_1, read\_addr\_2, write\_addr,

input [7:0] write\_data,

input RegWrite, clk, reset,

output reg [7:0] read\_data\_1, read\_data\_2

);

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

integer k;

always @(\*) begin

read\_data\_1 = Regfile[read\_addr\_1];

read\_data\_2 = Regfile[read\_addr\_2];

end

always @(posedge clk or posedge reset) begin

if (reset) begin

for (k = 0; k < 8; k = k + 1) begin

Regfile[k] = 8'b0;

end

end else if (RegWrite) begin

Regfile[write\_addr] = write\_data;

end

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

|  |
| --- |
| **Top module** |
|  |
| **Testbench** |
| `timescale 1ns / 1ps  module tb\_Register\_File;  reg [2:0] read\_addr\_1, read\_addr\_2, write\_addr;  reg [7:0] write\_data;  reg clk, reset, RegWrite;  wire [7:0] read\_data\_1, read\_data\_2;    Register\_File uut (  .read\_addr\_1(read\_addr\_1),  .read\_addr\_2(read\_addr\_2),  .write\_addr(write\_addr),  .write\_data(write\_data),  .RegWrite(RegWrite),  .clk(clk),  .reset(reset),  .read\_data\_1(read\_data\_1),  .read\_data\_2(read\_data\_2)  );    always #5 clk = ~clk; // Toggle clock every 5ns    initial begin  // Initialize signals  clk = 0;  reset = 1; // Reset is initially 1  RegWrite = 0;  write\_addr = 3'b000;  write\_data = 8'b0;  read\_addr\_1 = 3'b000;  read\_addr\_2 = 3'b000;  // Print simulation results  $monitor("Time: %0d | Clk: %b | Reset: %b | RegWrite: %b | Read Addr 1: %03b | Read Data 1: %02x | Read Addr 2: %03b | Read Data 2: %02x | Write Addr: %03b | Write Data: %02x",  $time, clk, reset, RegWrite, read\_addr\_1, read\_data\_1, read\_addr\_2, read\_data\_2, write\_addr, write\_data);    // Initial Reset  #10 reset = 0; // Release reset after 10ns  // Write sequence  #10 RegWrite = 1; write\_addr = 3'b001; write\_data = 8'h07; // Write 7 (binary 0111) to register 001  #10 write\_addr = 3'b010; write\_data = 8'h08; // Write 8 (binary 1000) to register 010  #10 write\_addr = 3'b011; write\_data = 8'h09; // Write 9 (binary 1001) to register 011  #10 write\_addr = 3'b100; write\_data = 8'h0A; // Write A (binary 1010) to register 100  #10 write\_addr = 3'b101; write\_data = 8'h0B; // Write B (binary 1011) to register 101  // Disable writing and start reading  #10 RegWrite = 0;  // Read from registers after writes  #10 read\_addr\_1 = 3'b001; read\_addr\_2 = 3'b101; // Read from registers 001 and 101  #10 read\_addr\_1 = 3'b011; read\_addr\_2 = 3'b100; // Read from registers 011 and 100  #10 read\_addr\_1 = 3'b101; read\_addr\_2 = 3'b101; // Read from register 101 twice  #10 read\_addr\_1 = 3'b100; read\_addr\_2 = 3'b100; // Read from register 100 twice  // End the simulation after a final check  #10 $finish;  end  endmodule |
|  |
|  |

### III.7 EXPERIMENT NO. 7

##### III.7.1 AIM: To implement the ALU

##### Diagram, schematic Description automatically generated

**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

|  |
| --- |
| **Top module** |
| module lab5\_ex7(SW,LEDR,LEDG,HEX7, HEX6);  input [17:0] SW;  output [0:6] HEX7, HEX6;  output [17:0] LEDR;  output [7:0] LEDG;  wire [7:0] W\_LED\_SEG;  assign LEDR = SW;  hex\_ssd (W\_LED\_SEG[3:0],HEX6);  hex\_ssd (W\_LED\_SEG[7:4],HEX7);    alu(.aluop(SW[17:15]),.ra(SW[12:7]),.rb\_or\_imm(SW[5:0]),.aluout(W\_LED\_SEG),.zero(LEDG[7]));  endmodule  module hex\_ssd (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 |
| **Testbench** |
| module tb\_alu;  // Inputs  reg [7:0] ra, rb\_or\_imm;  reg [2:0] aluop;  // Outputs  wire [7:0] aluout;  wire zero;  // Instantiate the ALU  integer i;  alu test\_unit (  aluop, // ALU Operation Select  ra, // ALU 8-bit Input A  rb\_or\_imm, // ALU 8-bit Input B  aluout, // ALU 8-bit Output  zero // Zero Flag Output  );  initial begin  // Monitor values  $monitor("Time=%0t | ra=%d | rb\_or\_imm=%h | aluop=%d | aluout=%d | zero=%b",  $time, ra, rb\_or\_imm, aluop, aluout, zero);  // Initial values  ra = 8'd3;  rb\_or\_imm = 8'd2;  aluop = 3'b0;  // Loop to test ALU operations  for (i = 0; i <= 7; i = i + 1) begin  aluop = aluop + 3'b1; // Increment ALU operation  #10; // Delay for each operation  end  // Change inputs for another test  ra = 8'd5;  rb\_or\_imm = 8'd3;  aluop = 3'b0; // Test an operation with new inputs  #10;  end  endmodule |
|  |
|  |

### III.8 EXPERIMENT NO. 8

##### III.8.1 AIM: To implement the 32 bit RAM

##### Diagram Description automatically generated

**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

|  |
| --- |
| Top module |
| module lab5\_ex8(SW,LEDR,LEDG);  input [17:0]SW;  output [17:0]LEDR;  output [7:0]LEDG;    assign LEDR=SW;    Data\_Memory(.addr(SW[7:0]),.write\_data(SW[15:8]),.read\_data(LEDG),.clk(SW[16]),.reset(SW[17]),.MemRead(1),.MemWrite(1));  endmodule |
| Testbench |
| `timescale 1ns / 1ps  module tb\_data\_memory;  // Inputs  reg [7:0] addr;  reg [7:0] write\_data;  reg clk;  reg reset;  reg MemRead;  reg MemWrite;  // Outputs  wire [7:0] read\_data;  // Instantiate the Data\_Memory module  Data\_Memory uut (  .addr(addr),  .write\_data(write\_data),  .read\_data(read\_data),  .clk(clk),  .reset(reset),  .MemRead(MemRead),  .MemWrite(MemWrite)  );  // Clock generation  always #5 clk = ~clk;  initial begin  // Initialize Inputs  clk = 0;  reset = 0;  MemRead = 0;  MemWrite = 0;  addr = 0;  write\_data = 0;  // Apply reset  $display("Applying reset...");  reset = 1;  #10;  reset = 0;  // Write data to memory  $display("Writing data to memory...");  addr = 8'h01;  write\_data = 8'hA5;  MemWrite = 1;  #10;  MemWrite = 0;  addr = 8'h02;  write\_data = 8'h3C;  MemWrite = 1;  #10;  MemWrite = 0;  // Read data from memory  $display("Reading data from memory...");  addr = 8'h01;  MemRead = 1;  #10;  $display("Read data at address 0x01: %h", read\_data);  addr = 8'h02;  #10;  $display("Read data at address 0x02: %h", read\_data);  // Read from uninitialized address  $display("Reading from uninitialized address...");  addr = 8'h03;  #10;  $display("Read data at address 0x03: %h", read\_data);  // Apply reset and verify memory reset  $display("Resetting memory...");  reset = 1;  #10;  reset = 0;  addr = 8'h01;  #10;  $display("Read data at address 0x01 after reset: %h", read\_data);  addr = 8'h02;  #10;  $display("Read data at address 0x02 after reset: %h", read\_data);  $finish;  end  endmodule |
|  |
|  |

### III.9 EXPERIMENT NO. 9

##### III.9.1 AIM: To implement Multiplexer

##### Chart, diagram Description automatically generated

**III.9.2 CODE**

module Mux\_N\_bit(in0, in1, mux\_out, select);

parameter N = 8;

input [N-1:0] in0, in1;

output [N-1:0] mux\_out;

input select;

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

|  |
| --- |
| Top module |
| module lab5\_ex9(SW,LEDG,LEDR);  input [17:0] SW;  output [17:0] LEDR;  output [7:0] LEDG;    assign LEDR =SW;    parameter N = 8;  Mux\_N\_bit(.in0(SW[N-1:0]),.in1(SW[2\*N-1:N]),.mux\_out(LEDG[N-1:0]),.select(SW[17]));  endmodule |
| Testbench |
|  |
|  |
|  |

### III.10 EXPERIMENT NO. 10

##### III.10.1 AIM: To implement Sign\_Extension

##### Diagram Description automatically generated

**III.10.2 CODE**

module Sign\_Extension (sign\_in, sign\_out);

input [5:0] sign\_in;

output [7:0] sign\_out;

assign sign\_out[5:0]=sign\_in[5:0];

assign sign\_out[7:6]=sign\_in[5]?2'b11:2'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

|  |
| --- |
| Top module |
| module lab5\_ex10(SW,LEDG,LEDR);  input [17:0] SW;  output [17:0] LEDR;  output [7:0] LEDG;    assign LEDR =SW;    Sign\_Extension (.sign\_in(SW[5:0]),.sign\_out(LEDG));  endmodule |
| Testbench |
|  |
|  |
|  |

**III. LAB REPORT GUIDELINES**

Students write up a report on the Verilog HDL implementation experiment projects created in this lab. The lab report should include Verilog code for the module under test, Verilog test bench code and a truth table results, and example data input and output to validate the experiment. Simulation Result in form of Simulation Capture Screen.