|  |
| --- |
| **DIGITAL SYSTEM DESIGN LABORATORY** |
| **REPORT LAB 4** |

**BASIC BUILDING BLOCKS OF SINGLE CYCLE MICROPROCESSOR**

Nguyen Gia Cat Tuong ITITIU21117

### 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

### III.2 EXPERIMENT NO. 2

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

##### Diagram Description automatically generated

**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.

module Adder32Bit\_tb;

reg [7:0] input1;

reg [7:0] input2;

wire [7:0] out;

// Instantiate the Adder32Bit module

Adder32Bit uut (

.input1(input1),

.input2(input2),

.out(out)

);

initial begin

// Initialize inputs

input1 = 0;

input2 = 0;

// Apply test vectors

#10 input1 = 8'h15; input2 = 8'h10; // 21 + 16 = 37

#10 input1 = 8'hFF; input2 = 8'h01; // 255 + 1 = 0 (overflow)

#10 input1 = 8'h7F; input2 = 8'h01; // 127 + 1 = 128

#10 input1 = 8'hA5; input2 = 8'h5A; // 165 + 90 = 255

#10 input1 = 8'h00; input2 = 8'h00; // 0 + 0 = 0

#10 $finish;

end

initial begin

$dumpfile("Adder32Bit\_tb.vcd");

$dumpvars(0, Adder32Bit\_tb);

end

endmodule

2. Write the Top level module to implement this module in FPGA KIT

module TopLevel(

input [7:0] sw1, // Assume sw1 is connected to input1 through switches

input [7:0] sw2, // Assume sw2 is connected to input2 through switches

output [7:0] led // Assume led is connected to output out through LEDs

);

// Instantiate the Adder32Bit module

Adder32Bit adder (

.input1(sw1),

.input2(sw2),

.out(led)

);

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 (

input clk,

input reset,

output reg [7:0] Instruction\_Address

);

always @(posedge clk or posedge reset) begin

if (reset)

Instruction\_Address <= 8'd0;

else

Instruction\_Address <= Instruction\_Address + 1;

end

endmodule

**III.3.3 LAB ASSIGNMENT**

1. Write testbenches to verify above module and attach waveforms.

`timescale 1ns / 1ps

module tb\_PC\_add\_1;

// Inputs

reg clk;

reg reset;

// Outputs

wire [7:0] Instruction\_Address;

// Instantiate the Unit Under Test (UUT)

PC\_add\_1 uut (

.clk(clk),

.reset(reset),

.Instruction\_Address(Instruction\_Address)

);

initial begin

// Initialize Inputs

clk = 0;

reset = 1;

#10;

reset = 0;

#1000;

reset = 1;

#10;

reset = 0;

#1000;

$finish;

end

always #5 clk = ~clk; // Clock generator

endmodule

2. Write the Top level module to implement this module in FPGA KIT

module top\_module (

input clk,

input reset,

output [7:0] Instruction\_Address

);

PC\_add\_1 pc\_add\_1\_inst (

.clk(clk),

.reset(reset),

.Instruction\_Address(Instruction\_Address)

);

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 [15:0] instruction;

reg [15:0] Imemory [7:0];

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<16; k=k+1)

begin

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

Imemory[k] = 16'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.

`timescale 1ns / 1ps

module tb\_Instruction\_Memory;

reg reset;

reg [7:0] read\_address;

wire [15:0] instruction;

Instruction\_Memory uut (

.reset(reset),

.read\_address(read\_address),

.instruction(instruction)

);

initial begin

// Initialize Inputs

reset = 0;

read\_address = 0;

// Apply Reset

#10 reset = 1;

#10 reset = 0;

// Read Instructions at different addresses

#10 read\_address = 8'd0;

#10 read\_address = 8'd1;

#10 read\_address = 8'd2;

#10 read\_address = 8'd3;

#10 read\_address = 8'd4;

#10 read\_address = 8'd5;

#10 read\_address = 8'd6;

#10 read\_address = 8'd7;

// Finish simulation

#10 $finish;

end

initial begin

$monitor("Time: %0d, Reset: %b, Read Address: %d, Instruction: %b",

$time, reset, read\_address, instruction);

end

endmodule

2. Write the Top level module to implement this module in FPGA KIT

module Top\_Level (

input wire reset,

input wire [7:0] read\_address,

output wire [15:0] instruction

);

// Instantiate the Instruction Memory

Instruction\_Memory imem (

.reset(reset),

.read\_address(read\_address),

.instruction(instruction)

);

// Other components can be instantiated here and connected

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 Instruction\_Datapath (

input clk,

input reset,

output reg [15:0] Instruction

);

always @(posedge clk or posedge reset) begin

if (reset) begin

Instruction <= 16'b0; // Reset instruction to zero

end else begin

Instruction <= 16'hABCD; // Dummy instruction

end

end

endmodule

**III.5.3 LAB ASSIGNMENT**1. Write testbenches to verify above module and attach waveforms.

`timescale 1ns / 1ps

module tb\_Instruction\_Datapath;

reg clk;

reg reset;

wire [15:0] Instruction;

// Instantiate the module

Instruction\_Datapath uut (

.clk(clk),

.reset(reset),

.Instruction(Instruction)

);

// Clock generation

always begin

#5 clk = ~clk; // 10ns period clock

end

initial begin

// Initialize signals

clk = 0;

reset = 1;

// Apply reset

#10 reset = 0;

// Observe for a few clock cycles

#50;

// Finish simulation

$stop;

end

initial begin

$dumpfile("tb\_Instruction\_Datapath.vcd");

$dumpvars(0, tb\_Instruction\_Datapath);

end

endmodule

2. Write the Top level module to implement this module in FPGA KIT

module Top (

input clk,

input reset,

output [15:0] Instruction

);

// Instantiate the Instruction\_Datapath module

Instruction\_Datapath datapath (

.clk(clk),

.reset(reset),

.Instruction(Instruction)

);

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 (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];

always @(read\_data\_1 or Regfile[read\_data\_1])

begin

if (read\_data\_1 == 0) read\_data\_1 = 0;

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

end

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

always @(read\_data\_2 or Regfile[read\_data\_2])

begin

if (read\_data\_2 == 0) read\_data\_2 = 0;

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

end

always @(posedge clk or posedge reset) // Ou combines the block of reset into the block of posedge clk

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.

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),

.clk(clk),

.reset(reset),

.RegWrite(RegWrite),

.read\_data\_1(read\_data\_1),

.read\_data\_2(read\_data\_2)

);

initial begin

// Initialize signals

clk = 0;

reset = 1;

RegWrite = 0;

read\_addr\_1 = 0;

read\_addr\_2 = 0;

write\_addr = 0;

write\_data = 0;

// Apply reset

#10 reset = 0;

#10 reset = 1;

#10 reset = 0;

// Write some data

#10 RegWrite = 1; write\_addr = 3'd1; write\_data = 8'h77;

#10 clk = 1; #10 clk = 0; // Generate clock pulse

#10 RegWrite = 1; write\_addr = 3'd2; write\_data = 8'h88;

#10 clk = 1; #10 clk = 0;

#10 RegWrite = 1; write\_addr = 3'd3; write\_data = 8'h99;

#10 clk = 1; #10 clk = 0;

// Read the data back

#10 RegWrite = 0; read\_addr\_1 =

2. Write the Top level module to implement this module in FPGA KIT

module Top\_Module (

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

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

input [2:0] write\_addr,

input [7:0] write\_data,

input clk,

input reset,

input RegWrite,

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

output [7:8] read\_data\_2

);

// Instantiate the Register File

Register\_File reg\_file (

.read\_addr\_1(read\_addr\_1),

.read\_addr\_2(read\_addr\_2),

.write\_addr(write\_addr),

.write\_data(write\_data),

.clk(clk),

.reset(reset),

.RegWrite(RegWrite),

.read\_data\_1(read\_data\_1),

.read\_data\_2(read\_data\_2)

);

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.

`timescale 1ns / 1ps

module alu\_tb;

// Inputs

reg [2:0] alufn;

reg [7:0] ra;

reg [7:0] rb\_or\_imm;

// Outputs

wire [7:0] aluout;

wire zero;

// Instantiate the ALU

alu uut (

.alufn(alufn),

.ra(ra),

.rb\_or\_imm(rb\_or\_imm),

.aluout(aluout),

.zero(zero)

);

initial begin

// Test cases

// Test ADD

alufn = 3'b000; ra = 8'd8; rb\_or\_imm = 8'd2; #10;

$display("ADD: %d + %d = %d, zero = %b", ra, rb\_or\_imm, aluout, zero);

// Test SUB

alufn = 3'b001; ra = 8'd8; rb\_or\_imm = 8'd2; #10;

$display("SUB: %d - %d = %d, zero = %b", ra, rb\_or\_imm, aluout, zero);

// Test AND

alufn = 3'b010; ra = 8'b00001000; rb\_or\_imm = 8'b00000010; #10;

$display("AND: %b & %b = %b, zero = %b", ra, rb\_or\_imm, aluout, zero);

// Test OR

alufn = 3'b011; ra = 8'b00001000; rb\_or\_imm = 8'b00000010; #10;

$display("OR: %b | %b = %b, zero = %b", ra, rb\_or\_imm, aluout, zero);

// Test NOT\_A

alufn = 3'b100; ra = 8'b00001000; rb\_or\_imm = 8'd2; #10;

$display("NOT\_A: ~%b = %b, zero = %b", ra, aluout, zero);

// Test LW

alufn = 3'b101; ra = 8'd8; rb\_or\_imm = 8'd2; #10;

$display("LW: %d + %d = %d, zero = %b", ra, rb\_or\_imm, aluout, zero);

// Test SW

alufn = 3'b110; ra = 8'd8; rb\_or\_imm = 8'd2; #10;

$display("SW: %d + %d = %d, zero = %b", ra, rb\_or\_imm, aluout, zero);

// Test BEQ (not equal case)

alufn = 3'b111; ra = 8'd8; rb\_or\_imm = 8'd2; #10;

$display("BEQ (not equal): %d - %d = %d, zero = %b", ra, rb\_or\_imm, aluout, zero);

// Test BEQ (equal case)

alufn = 3'b111; ra = 8'd8; rb\_or\_imm = 8'd8; #10;

$display("BEQ (equal): %d - %d = %d, zero = %b", ra, rb\_or\_imm, aluout, zero);

end

endmodule

2. Write the Top level module to implement this module in FPGA KIT

module top\_level(

input [2:0] alufn,

input [7:0] ra,

input [7:0] rb\_or\_imm,

output [7:0] aluout,

output zero,

input clk, // Assuming a clock input is needed

input rst // Assuming a reset input is needed

);

// Instantiate the ALU

alu my\_alu (

.alufn(alufn),

.ra(ra),

.rb\_or\_imm(rb\_or\_imm),

.aluout(aluout),

.zero(zero)

);

// Additional logic can be added here if needed

// for example, interfacing with other peripherals,

// handling reset, clock management, etc.

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.

module Data\_Memory\_tb;

// Testbench signals

reg [7:0] addr\_tb;

reg [7:0] write\_data\_tb;

wire [7:0] read\_data\_tb;

reg clk\_tb, reset\_tb, MemRead\_tb, MemWrite\_tb;

// Instantiate the Data\_Memory module

Data\_Memory uut (

.addr(addr\_tb),

.write\_data(write\_data\_tb),

.read\_data(read\_data\_tb),

.clk(clk\_tb),

.reset(reset\_tb),

.MemRead(MemRead\_tb),

.MemWrite(MemWrite\_tb)

);

// Clock generation

initial begin

clk\_tb = 0;

forever #5 clk\_tb = ~clk\_tb;

end

// Testbench procedure

initial begin

// Initialize signals

reset\_tb = 1;

MemRead\_tb = 0;

MemWrite\_tb = 0;

addr\_tb = 0;

write\_data\_tb = 0;

// Apply reset

#10 reset\_tb = 0;

// Write data to memory

#10 addr\_tb = 8'h01; write\_data\_tb = 8'hAA; MemWrite\_tb = 1; MemRead\_tb = 0;

#10 MemWrite\_tb = 0;

// Read data from memory

#10 addr\_tb = 8'h01; MemRead\_tb = 1;

#10 MemRead\_tb = 0;

// Write another data

#10 addr\_tb = 8'h02; write\_data\_tb = 8'hBB; MemWrite\_tb = 1;

#10 MemWrite\_tb = 0;

// Read new data

#10 addr\_tb = 8'h02; MemRead\_tb = 1;

#10 MemRead\_tb = 0;

// Apply reset and check memory clearance

#10 reset\_tb = 1;

#10 reset\_tb = 0;

// Check if memory is cleared

#10 addr\_tb = 8'h01; MemRead\_tb = 1;

#10 MemRead\_tb = 0;

#10 addr\_tb = 8'h02; MemRead\_tb = 1;

#10 MemRead\_tb = 0;

// End simulation

#50 $finish;

end

// Monitor the changes

initial begin

$monitor("Time=%0d, reset=%b, addr=%h, write\_data=%h, read\_data=%h, MemRead=%b, MemWrite=%b",

$time, reset\_tb, addr\_tb, write\_data\_tb, read\_data\_tb, MemRead\_tb, MemWrite\_tb);

end

endmodule

2. Write the Top level module to implement this module in FPGA KIT

module Top\_Module (

input clk, // System clock

input reset, // Reset signal

input MemRead, // Memory Read signal

input MemWrite, // Memory Write signal

input [7:0] addr, // Address for memory access

input [7:0] write\_data, // Data to be written

output [7:0] read\_data // Data to be read

);

// Instantiate the Data\_Memory module

Data\_Memory data\_memory\_inst (

.addr(addr),

.write\_data(write\_data),

.read\_data(read\_data),

.clk(clk),

.reset(reset),

.MemRead(MemRead),

.MemWrite(MemWrite)

);

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 = 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

### 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.

module Mux\_N\_bit\_tb;

// Parameter

parameter N = 32;

// Testbench signals

reg [N-1:0] in0\_tb, in1\_tb;

wire [N-1:0] mux\_out\_tb;

reg select\_tb;

// Instantiate the Mux\_N\_bit module

Mux\_N\_bit #(.N(N)) uut (

.in0(in0\_tb),

.in1(in1\_tb),

.mux\_out(mux\_out\_tb),

.select(select\_tb)

);

// Testbench procedure

initial begin

// Initialize signals

in0\_tb = 32'hAAAAAAAA;

in1\_tb = 32'h55555555;

select\_tb = 0;

// Apply different select values

#10 select\_tb = 0;

#10 select\_tb = 1;

#10 select\_tb = 0;

#10 select\_tb = 1;

// Change inputs and reapply select values

#10 in0\_tb = 32'h12345678;

in1\_tb = 32'h87654321;

select\_tb = 0;

#10 select\_tb = 1;

#10 select\_tb = 0;

#10 select\_tb = 1;

// End simulation

#10 $finish;

end

// Monitor changes

initial begin

$monitor("Time=%0d, select=%b, in0=%h, in1=%h, mux\_out=%h",

$time, select\_tb, in0\_tb, in1\_tb, mux\_out\_tb);

end

endmodule

2. Write the Top level module to implement this module in FPGA KIT

module Top\_Module (

input clk, // System clock (not used in this simple MUX)

input reset, // Reset signal (not used in this simple MUX)

input select, // Select signal for MUX

input [31:0] in0, // First input to MUX

input [31:0] in1, // Second input to MUX

output [31:0] mux\_out // Output of MUX

);

// Instantiate the Mux\_N\_bit module

Mux\_N\_bit #(.N(32)) mux\_inst (

.in0(in0),

.in1(in1),

.mux\_out(mux\_out),

.select(select)

);

endmodule