COMPUTER SYSTEM ARCHITECTURE

LAB REPORT #05

RISC-V Datapath Implementation

Prepared by:
SHAZIL 532263

Presented To:
LE USAMA SHAUKAT

NUST CEME
DEPT. COMPUTER ENGINEERING

TABLE OF CONTENT

RISC-V Datapath Implementation (I-Type and R-Type Instructions)

Objective

The aim of this lab is to design an accurate RISC-V Datapath capable of executing both Itype and R-type instructions.

Tools Required

Xilinix Vivado

RISC-V Datapath Overview

The microarchitecture of the RISC-V Datapath includes the following key components:

- Program Counter (PC)
- Instruction Memory (IMEM)
- Data Memory (DMEM)
- Register File
- Arithmetic Logic Unit (ALU)
- Immediate Generation Unit (ImmGen)

Task

Referring once more to the system-level diagram, your assignment is to design the Datapath illustrated below. The Datapath should be capable of handling all I-Type and R-Type instructions.

-

Modules and Codes (Top Module)

```
module Top#(
    parameter A = 32,
    parameter B = 76,
    parameter E = 32,
    parameter D = 8,
    parameter NUM REG = 32,
    parameter G = 6,
    parameter H = 32,
    parameter I = 7
)(
    input  logic clk,
    input  logic reset,
    output logic [E-1:0] f result
);

    logic [A-1:0] pc out;
    logic [E-1:0] instruction;

    logic we;
    logic [G-1:0] rsadd1, rsadd2, rdadd;
    logic [I-1:0] opcode;
    logic [E-1:0] wdata, reg result1, reg result2, alu result , mux res , imm res;
    logic [2:0] func3;
    logic [6:0] func7;
    logic [19:0] imm;
    logic aluSrc;

    PC #(A) pc inst (
        .clk(clk),
        .reset(reset),
        .add(pc out)
    );

    InsMemory #(B, E, D) insmemory inst (
        .addr(pc out),
        .dataR(instruction)
    );

    always comb begin
        if (instruction[6:0] == 7'd51) begin
            func7   = instruction[31:25];
            rsadd2  = instruction[24:20];
            rsadd1  = instruction[19:15];
            func3   = instruction[14:12];
            rdadd   = instruction[11:7];
            opcode  = instruction[6:0];
            imm     = ;
        end else begin
            func7   = ;
```

Program Counter (PC) Module

```
module PC #(parameter N = 32)
(
    input logic clk,
    input logic reset,
    output logic [N-1:0] add
);
    always_ff@(posedge clk)
    begin
        if(reset == 1’b1)
            add <= 0;
        else
            add <= add + 4;
    end
endmodule
```

Instruction Memory Module

```
module InsMemory #(parameter length = 76 , width = 32 , N = 8)
(
    input logic [width-1:0] addr,
    output logic [width-1:0] dataR
);

logic [N-1:0] InsMem [0:length-1];

initial begin
    $readmemh("IMdata.mem" , InsMem);
end

always_comb begin
    dataR = {InsMem[addr+3],InsMem[addr+2],InsMem[addr+1],InsMem[addr]};
end
endmodule
```

Register File Module

```
module Reg_file#(
    parameter DATA_WIDTH = 32,
    parameter NUM_REGS   = 32,
    parameter e =6
)(
    input logic clk,
    input logic we,
    input logic [e-1:0]rs1,
    input logic [e-1:0]rs2,
    input logic [e-1:0]rsw,
    input logic [DATA_WIDTH-1:0]dataw,
    output logic [DATA_WIDTH-1:0]data1,
    output logic [DATA_WIDTH-1:0]data2
);
logic [DATA_WIDTH-1:0] regfile [0:NUM_REGS-1];

initial begin
    $readmemh("rfdata.mem" , regfile);
end

initial begin
    $display("----- Initial Register File Contents -----");
    for (int i = 0; i < NUM_REGS; i++)
        $display("regfile[%0d] = %h", i, regfile[i]);
    $display("----------------------------------------");
end

always_ff@(posedge clk)
begin
    if(we && rsw !=0) begin
        regfile[rsw] <= dataw;
        $display("Time=%0t | Wrote %h to regfile[%0d]", $time, dataw, rsw);
    end
end

final begin
    $display("----- Final Register File Contents -----");
    for (int i = 0; i < NUM_REGS; i++)
        $display("regfile[%0d] = %h", i, regfile[i]);
    $display("--------------------------------------");
    $writememh("rfdata_final.mem", regfile);
end

assign data1 = regfile[rs1];
assign data2 = regfile[rs2];
endmodule
```

Immediate Generator Module

```
module ImmGen(
    input logic [11:0] imm,
    output logic [31:0] out
);

always_comb begin
    out = {{20{imm[11]}}, imm};
end
endmodule
```

MUX Module

```
module mux(
    input logic [31:0] ri,
    input logic [31:0] li,
    input logic sl,
    output logic [31:0] res
);
always_comb begin
    if(sl == 1’b0)
        res = ri;
    else
        res = li;
end
endmodule
```

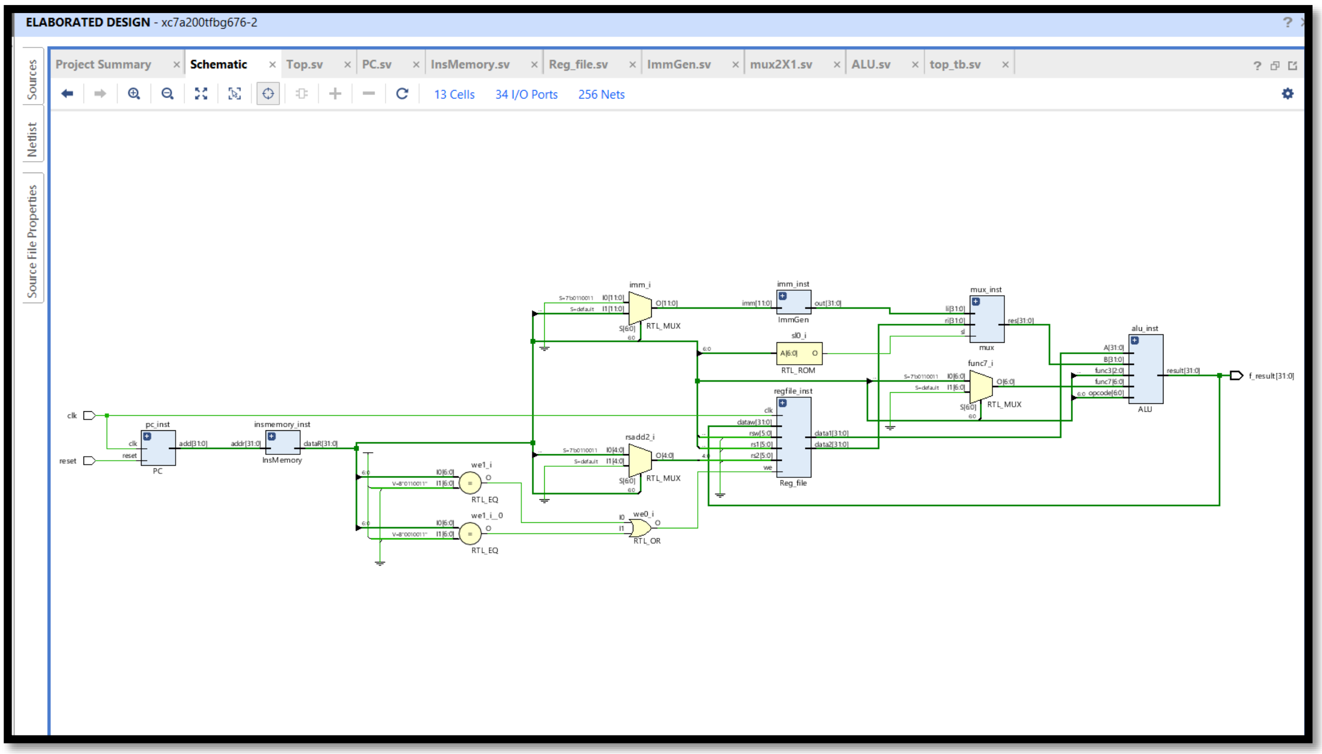
ALU Module

```
module ALU #(
    parameter WIDTH = 32
)(
    input  logic [WIDTH-1:0] A,
    input  logic [WIDTH-1:0] B,
    input  logic [6:0] opcode,
    input  logic [2:0] func3,
    input  logic [6:0] func7,
    output logic [WIDTH-1:0] result
);

always_comb begin
    result = '0;
    if (opcode == 7'b0110011) begin
        case (func3)
            3'b000: begin
                if (func7 == 7'b0000000)
                    result = A + B;
                else if (func7 == 7'b0100000)
                    result = A - B;
            end
            3'b111: result = A & B;
            3'b110: result = A | B;
            3'b100: result = A ^ B;
            3'b010: result = ($signed(A) < $signed(B)) ? 32'd1 : 32'd0;
            3'b011: result = (A < B) ? 32'd1 : 32'd0;
            3'b001: result = A << B[4:0];
            3'b101: begin
                if (func7 == 7'b0000000)
                    result = A >> B[4:0];
                else if (func7 == 7'b0100000)
                    result = $signed(A) >>> B[4:0];
            end
            default: result = 32'd0;
        endcase
    end else if (opcode == 7'b0010011) begin
        case (func3)
            3'b000: result = A + B;
            3'b111: result = A & B;
            3'b110: result = A | B;
            3'b100: result = A ^ B;
            3'b010: result = ($signed(A) < $signed(B)) ? 32'd1 : 32'd0;
            3'b011: result = (A < B) ? 32'd1 : 32'd0;
            3'b001: result = A << B[4:0];
            3'b101: begin
                if (func7 == 7'b0000000)
                    result = A >> B[4:0];
                else if (func7 == 7'b0100000)
                    result = $signed(A) >>> B[4:0];
            end
            default: result = 32'd0;
```

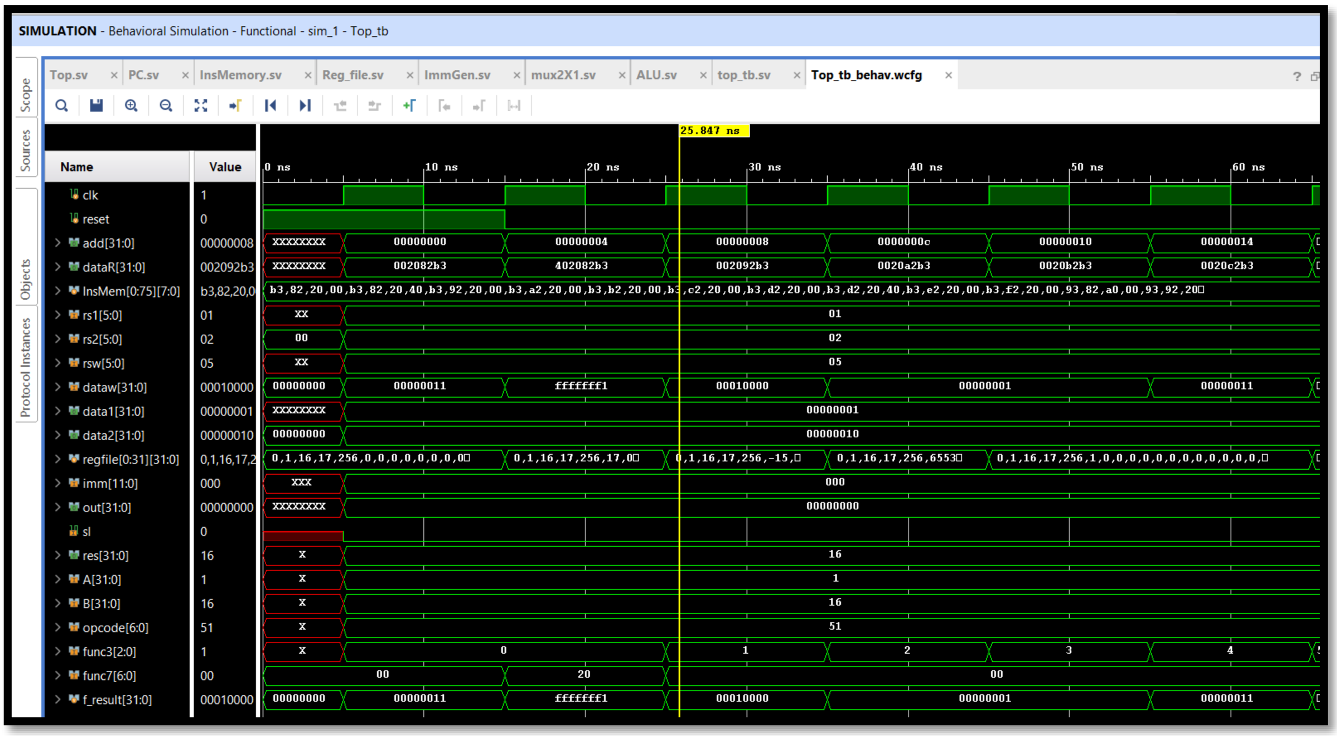
Top Testbench Module

```
module Top_tb;

    logic clk;
    logic reset;
    logic [31:0] f_result;

    Top dut (
        .clk(clk),
        .reset(reset),
        .f_result(f_result)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;
        reset = 1;
        #15 reset = 0;
        #200;
        $finish;
    end

endmodule
```

SCHEMATICS:



SIMULATIONS:



Name	Value
clk	1
reset	0
add[31:0]	00000008
dataR[31:0]	002092b3
InsMem[0:75][7:0]	b3,82,20,0
rs1[5:0]	01
rs2[5:0]	02
rsw[5:0]	05
dataw[31:0]	00010000
data1[31:0]	00000001
data2[31:0]	00000010
regfile[0:31][31:0]	0,1,16,17,2
imm[11:0]	000
out[31:0]	00000000
sl	0
res[31:0]	16
A[31:0]	1
B[31:0]	16
opcode[6:0]	51
func3[2:0]	1
func7[6:0]	00
f_result[31:0]	00010000

```
25.847 ns
0 ns   10 ns   20 ns   30 ns   40 ns   50 ns   60 ns
add[31:0]        XXXXXXXX  00000000  00000004  00000008  0000000c  00000010  00000014
dataR[31:0]      XXXXXXXX  002082b3  402082b3  002092b3  0020a2b3  0020b2b3  0020c2b3
InsMem[0:75][7:0]  b3,82,20,00,b3,82,20,40,b3,92,20,00,b3,a2,20,00,b3,b2,20,00,b3,c2,20,00,b3,d2,20,40,b3,e2,20,00,b3,f2,20,00,93,82,a0,00,93,92,20
rs1[5:0]         XX  01
rs2[5:0]         00  02
rsw[5:0]         XX  05
dataw[31:0]      00000000  00000011  fffffff1  00010000  00000001  00000011
data1[31:0]      XXXXXXXX  00000001
data2[31:0]      00000000  00000010
regfile[0:31][31:0]  0,1,16,17,256,0,0,0,0,0,0,0,0  0,1,16,17,256,17,0  0,1,16,17,256,-15,0  0,1,16,17,256,6553  0,1,16,17,256,1,0,0,0,0,0,0,0,0,0,0,0,0,0
imm[11:0]        XXX  000
out[31:0]        XXXXXXXX  00000000
res[31:0]        X  16
A[31:0]          X  1
B[31:0]          X  16
opcode[6:0]      X  51
func3[2:0]       X  0  1  2  3  4
func7[6:0]       00  20  00
f_result[31:0]   00000000  00000011  fffffff1  00010000  00000001  00000011
```

SHAZIL 532263

Conclusion

The processor design was successfully implemented and verified using Xilinx Vivado. Each module, including the Program Counter (PC), Instruction Memory, Register File, Immediate Generator, Multiplexer, ALU, and Top module, was developed and integrated to form a functional processor. The simulation results confirmed correct instruction decoding, data flow, and ALU operations for both R-type and I-type instructions.

This lab enhanced understanding of modular hardware design, hierarchy, and interconnection in processor architecture. It also improved practical skills in Verilog coding, simulation, debugging, and testbench creation. The project demonstrates a working foundation for a simple RISC-style processor capable of executing arithmetic and logical operations efficiently.