NATIONAL UNIVERSITY OF SCIENCE AND TECHNALOGY

DEPARTMENT OF COMPUTER AND SOFTWARE ENGINEERING

Lab#8: RISC-V Datapath Implementation (Loads and Stores)

Lab Report # 08

Course Instructor: *Dr Shahid Ismail*

Lab Instructor: *Usama Shoukat*

Sr. No.	Student's Name	CMS ID
01	SHAZIL	532263

Due Date: 10-Nov-2025
Submission Date: 9-Nov-2025

TABLE OF CONTENT

INTRODUCTION:

Loads (I-Type)

The load instructions are encoded as I-Type Instructions.

$imm_{11:0}$	rs1	funct3	rd	op	**I-Type**

A list of load instructions is given in the table below:

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)

Load instructions operate similarly to add instructions that use immediate operands. The address is calculated within the ALU and then sent to the data memory, which uses it to access the required data from the memory array.

Stores (S-Type)

The store instructions are encoded in a special instruction type known as the S-Type (bear in mind that S does not stand for Special). The encoding of S-Type is as follows:

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**

It is slightly different from the I-Type. The destination register operand is replaced with the immediate value while the source operand (rs2) is brought back, leading to breaking down of immediate into two fields. Breaking down ensures uniformity in the fields (rs1, rs2, rd, etc.). A list of store instructions is given in the table below:

op	funct3	funct7	Type	Instruction	Description	Operation
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	$[Address]_{7:0}$ = $rs2_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	$[Address]_{15:0}$ = $rs2_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	$[Address]_{31:0}$ = rs2

Memory Alignment

To simplify hardware design, the data memory is restricted to accessing 32 bits at a time. Therefore, both the data read (dataR) and data write (dataW) signals are always 32 bits wide, regardless of the specific load or store instruction being executed.

Additionally, many RISC-V memory systems impose an alignment requirement, allowing memory accesses only at addresses that are multiples of 4. In RISC-V, memory operations must occur at *aligned* addresses, meaning:

1. **lw/sw (word access)** — must occur at addresses that are multiples of 4.
2. **lh/sh (half-word access)** — must occur at addresses that are multiples of 2.
3. **lb/sb (byte access)** — can occur at any address.

Unaligned accesses result in undefined behavior.

For **load instructions**, specific bytes or half-words are extracted from the 32-bit word and then sign- or zero-extended based on the instruction type.

For **store instructions**, data smaller than a word is aligned to the correct byte positions, and a masking mechanism is applied to prevent overwriting unintended memory locations (using separate read/write control bits).

OBJECTIVE:

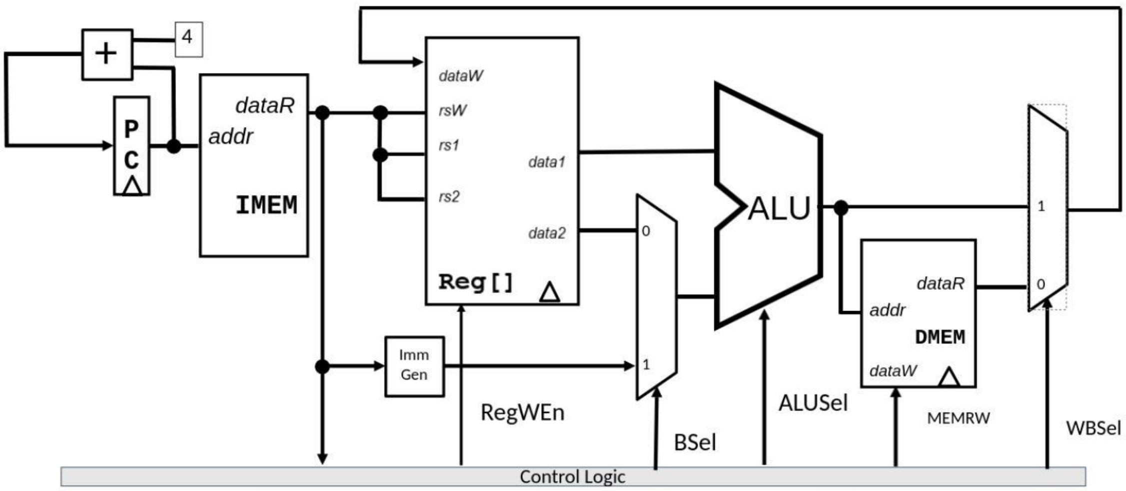
The objective of this lab is to Extend the previously designed Datapath to include loads and store.

SOFTWARE\TOOL USED:

Xilinx Vivado

LAB TASK:

Extend the previous design to include load and store instructions in your datapath. After extending the design your datapath may look along the lines of this figure:



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 InsMemory #(parameter length = 108 , 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
```

DATA MEMORY:

```
module data_mem #(
    parameter DEPTH = 1024
)(
    input  logic        clk,
    input  logic        mem_read,
    input  logic        mem_write,
    input  logic [2:0]  funct3,
    input  logic [31:0] addr,
    input  logic [31:0] dataW,
    output logic [31:0] dataR
);

    logic [7:0] mem [0:DEPTH-1];

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

    logic [31:0] word;
    logic [31:0] write_word;
    logic [31:0] result;
```

```
    initial begin
        $readmemh("DataMemory.mem", mem);
    end

    logic [31:0] word;
    logic [31:0] write_word;
    logic [31:0] result;

    always_comb begin
        word = { mem[addr+3], mem[addr+2], mem[addr+1], mem[addr] };

        case(funct3)
            3'b000: result = {{24{mem[addr][7]}}, mem[addr]};
            3'b100: result = {24'b0, mem[addr]};

            3'b001: result = {{16{word[15]}}, word[15:0]};
            3'b101: result = {16'b0, word[15:0]};

            3'b010: result = word;

            default: result = 32'h0;
        endcase

        if(mem_read)
            dataR = result;
        else
            dataR = 32'h0;
    end
```

```
    always_ff @(posedge clk)
    begin
        if(mem_write)
         begin
            case(funct3)
                3'b000: mem[addr] <= dataW[7:0];

                3'b001: begin
                    mem[addr]   <= dataW[7:0];
                    mem[addr+1] <= dataW[15:8];
                end

                3'b010: begin
                    mem[addr]   <= dataW[7:0];
                    mem[addr+1] <= dataW[15:8];
                    mem[addr+2] <= dataW[23:16];
                    mem[addr+3] <= dataW[31:24];
                end
            endcase
        end
    end

    final begin
        $writememh("DataMemory.mem", mem);
    end

endmodule
```

REGISTER FILE:

```
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

   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

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

IMMEDIATE GENERATOR:

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

    logic [6:0] opcode;
    assign opcode = instr[6:0];

    always_comb begin
        case (opcode)
            7'b0000011:
                imm = { {20{instr[31]}}, instr[31:20] };

            7'b0010011:
                imm = { {20{instr[31]}}, instr[31:20] };

            7'b0100011:
                imm = { {20{instr[31]}}, instr[31:25], instr[11:7] };

            default:
                imm = 32'h0;
        endcase
    end
endmodule
```

ALU MODULE:

```
module ALU #(
    parameter WIDTH = 32
)(
    input  logic [WIDTH-1:0] A,
    input  logic [WIDTH-1:0] B,
    input  logic [3:0] opcode,
    output logic [WIDTH-1:0] result
);

    always_comb
    begin
    case(opcode)
        4'b0000:
        begin
        result = A & B;
         end
        4'b0001:
        begin
        result = A | B;
         end
        4'b0010:
        begin
        result = A + B;
         end
        4'b0011:
        begin
        result = A - B;
         end
    endcase


    end
endmodule
```

TOP MODULE:

```
module Top#(
    parameter A = 32,
    parameter B = 108,
    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 , memresult;
    logic [2:0] func3;
    logic [6:0] func7;
    logic [11:0] imm;
    logic [3:0] op;
    logic aluSrc;
    logic memwrite;
    logic memread;
    logic memtoreg;
    logic branch;
```

```
    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     = 0;
    end
   else if (instruction[6:0] == 7'd19)
     begin
        func7   = 0;
        rsadd2  = 0;
        rsadd1  = instruction[19:15];
        func3   = instruction[14:12];
        rdadd   = instruction[11:7];
        opcode  = instruction[6:0];
        imm     = instruction[31:20];
    end
    else if (instruction[6:0] == 7'd3)
    begin
        func7 = 0;
        rsadd2 = 0;
        rsadd1 = instruction[19:15];
        func3 = instruction[14:12];
        rdadd = instruction[11:7];
        opcode = instruction[6:0];
        imm = instruction[31:20];
    end
    else
    begin
```

```
    begin
        func7 = instruction[31:25];
        rsadd2 = instruction[24:20];
        rsadd1 = instruction[19:15];
        func3 = instruction[14:12];
        rdadd = 0; // no rd in S-type
        opcode = instruction[6:0];
        imm = {instruction[31:25], instruction[11:7]};
    end
end

    top_control_alu control_unit (
        .opcode(opcode),
        .funct3(func3),
        .funct7(func7[5]),
        .alu_ctrl(op),
        .regwrite(we),
        .alusrc(aluSrc),
        .memread(memread),
        .memwrite(memwrite),
        .memtoreg(memtoreg),
        .branch(branch)

    );
    Reg_file #(E, NUM_REG, G) regfile_inst (
        .clk(clk),
        .we(we),
        .rs1(rsadd1),
        .rs2(rsadd2),
        .rsw(rdadd),
        .dataw(wdata),
        .data1(reg_result1),
        .data2(reg_result2)
    );
```

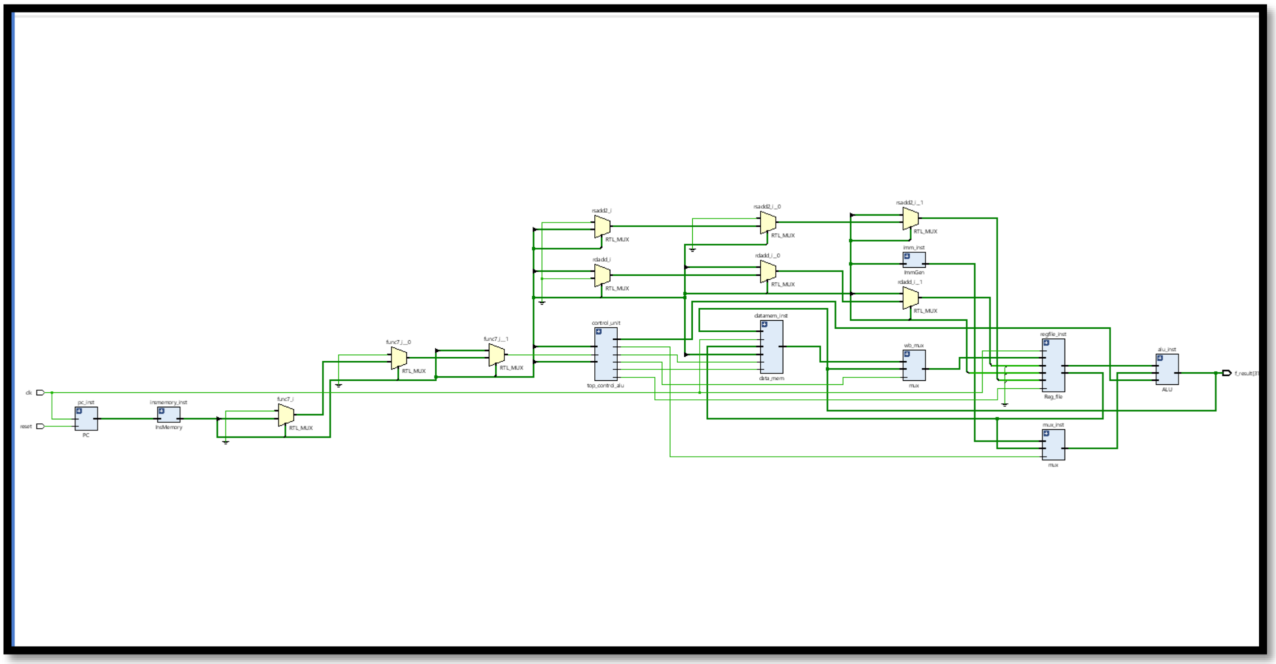
```
    ImmGen imm_inst (
        .instr(instruction),
        .imm(imm_res)
    );

    mux mux_inst (
        .ri(reg_result2),
        .li(imm_res),
        .sl(aluSrc),
        .res(mux_res)
     );

    ALU #(H) alu_inst(
        .A(reg_result1),
        .B(mux_res),
        .opcode(op),
        .result(alu_result)
    );


    data_mem #(1024) datamem_inst (
    .clk(clk),
    .mem_read(memread),
    .mem_write(memwrite),
    .funct3(func3),
    .addr(alu_result),
    .dataW(reg_result2),
    .dataR(memresult)
);
```

```
    ALU #(H) alu_inst(
        .A(reg_result1),
        .B(mux_res),
        .opcode(op),
        .result(alu_result)
    );


    data_mem #(1024) datamem_inst (
    .clk(clk),
    .mem_read(memread),
    .mem_write(memwrite),
    .funct3(func3),
    .addr(alu_result),
    .dataW(reg_result2),
    .dataR(memresult)
);

    mux wb_mux (
    .ri(alu_result),
    .li(memresult),
    .sl(memtoreg),
    .res(wdata)
);
    assign f_result = alu_result;

endmodule
```

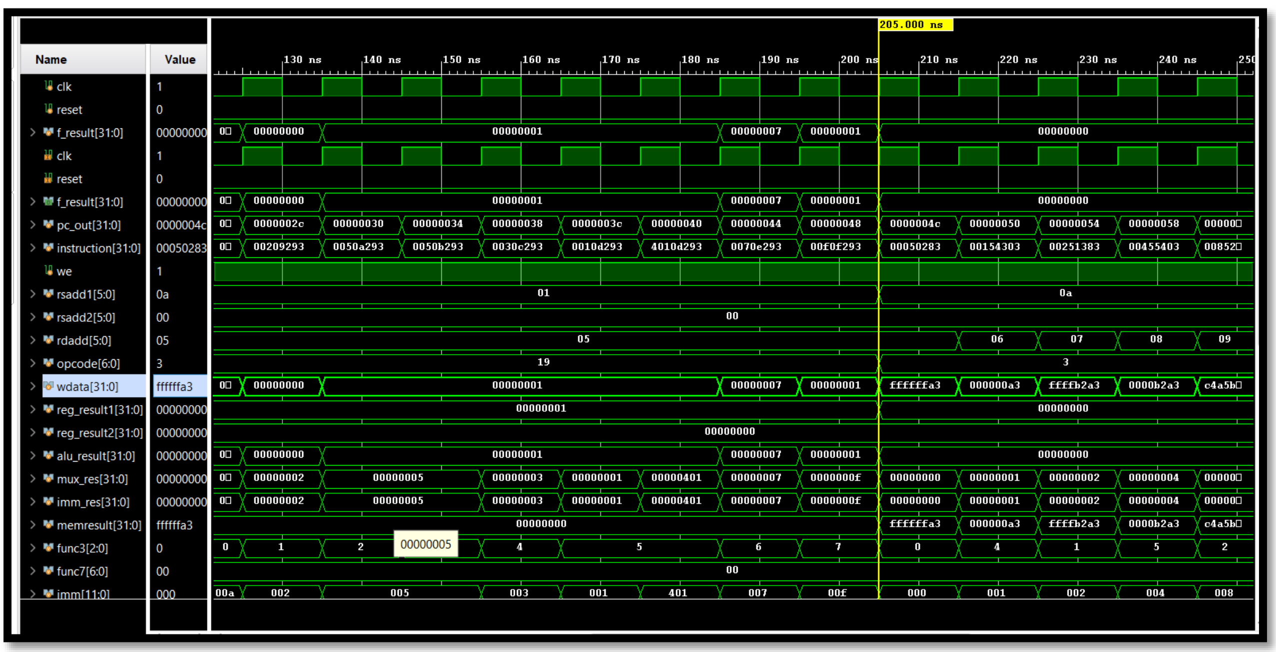
TOP TESTBENCH FILE:

```
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;

        #20 res loading tooltips...

        #300;

        $finish;
    end

endmodule
```

SCHEMATICS DIAGRAM:



SIMUALTIONS:



CONCLUSION:

This lab built a working data path that handles register operations, immediate arithmetic, and memory access. You added load and store support. You decoded S-type and I-type formats. You generated the correct immediate values. You selected between register and immediate operands. You used the ALU to calculate memory addresses. You wrote data to memory on store and read data from memory on load. You routed data back to the register file through a controlled write-back path.

You confirmed correctness with simulation. You observed register updates and memory changes. You verified that control signals match instruction types.

This lab improved your understanding of how instructions translate into hardware operations. You learned how decoding, control logic, memory access, and write-back integrate into one continuous flow in a processor.