NATIONAL UNIVERSITY OF SCIENCE AND TECHNALOGY

DEPARTMENT OF COMPUTER AND SOFTWARE ENGINEERING

Branch Instructions Datapath and Control Logic

Lab Report # 10

Course Instructor: *Dr Shahid Ismail*

Lab Instructor: *Usama Shoukat*

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

Due Date: 25-Nov-2025
Submission Date: 25-Nov-2025

TABLE OF CONTENT

INTRODUCTION:

Branches

There are several types of branch instructions having B-Type Encoding and listed in the table below

op	funct3	funct7	Type	Instruction	Description	Operation
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA

When the branch condition becomes true the processor jumps to the label listed in the instruction. Essentially, label is encoded as an immediate value that specifies the offset from current value of program counter**. i.e. PC = PC + Offset (Label).**

It might seem evident, but for sake of clarity we explain that arithmetic logic unit (ALU) is busy computing program counter address, and we need to add an additional unit known as the branch comparator for comparing the branch conditions or we can do within the ALU Unit

OBJECTIVE:

The objective of this lab is to Extend the previously designed Datapath to include branch controls and datapath.

SOFTWARE\TOOL USED:

Xilinx Vivado

Task:

Extend your design to include all branch type instructions and simulate it (Test the logic) and finally Your Processor now must execute all the instructions (R,I,LW,SW, Branch) datapath and Control Logic.

OUTPUT:

PC MODULE:

```
22
23  module PC #(parameter N = 32)
24  (
25      input logic clk,
26      input logic reset,
27      output logic [N-1:0] add
28  );
29      always_ff@(posedge clk)
30      begin
31      if(reset == 1'b1)
32          add <= 0;
33      else
34          add <= add+4 ;
35      end
36  endmodule
37
```

INSTRUCTION MEMORY

```
20  ////////////////////////////////////////////////////////////////////////////////
21  module InsMemory #(parameter length = 116 , width = 32 , N = 8)
22       (
23       input logic [width-1:0] addr,
24       output logic [width-1:0] dataR
25        );
26        logic [N-1:0] InsMem [0:length-1];
27        initial
28        begin
29        $readmemh("IMdata.mem" , InsMem);
30        end
31
32        always_comb
33        begin
34          dataR = {InsMem[addr+3],InsMem[addr+2],InsMem[addr+1],InsMem[addr]};
35        end
36  endmodule
37
```

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:

```
23  module ImmGen(
24      input  logic [31:0] instr,
25      output logic [31:0] imm
26  );
27  logic [11:0] imm_I;
28  logic [11:0] imm_S;
29  logic [20:0] imm_J;
30  assign imm_I = instr[31:20];
31  assign imm_S = {instr[31:25], instr[11:7]};
32  assign imm_J = {instr[31], instr[19:12], instr[20], instr[30:21], 1'b0};
33  always_comb begin
34      case (instr[6:0])
35          7'b0000011: imm = {{20{imm_I[11]}}, imm_I};      // LOAD
36          7'b0100011: imm = {{20{imm_S[11]}}, imm_S};      // STORE
37          7'b1101111: imm = {{11{imm_J[20]}}, imm_J};      // JAL
38          7'b1100111: imm = {{20{imm_I[11]}}, imm_I};      // JALR
39          default:    imm = {{20{imm_I[11]}}, imm_I};      // I-type
40      endcase
41  end
```

ALU MODULE:

```
24  )(
25      input  logic [WIDTH-1:0] A,
26      input  logic [WIDTH-1:0] B,
27      input  logic [3:0] opcode,
28      output logic [WIDTH-1:0] result,
29      output logic zeroFlag
30  );
31      always_comb begin
32  case(opcode)
33      4'b0000:
34          result = A + B;
35      4'b0001:
36          result = A - B;
37      4'b0010:
38          result = A & B;
39      4'b0011:
40          result = A | B;
41      4'b0100:
42          result = A ^ B;
43      4'b0101:
44          result = ($signed(A) < $signed(B)) ? 32'd1 : 32'd0;
45      4'b0110:
46          result = (A < B) ? 32'd1 : 32'd0;
47      4'b0111:
48          result = A << B[4:0];
49      4'b1000:
50          result = A >> B[4:0];
51      4'b1001:
52          result = $signed(A) >>> B[4:0];
53      default:
54          result = 32'd0;
55  endcase
56  end
57          assign zeroFlag = (result == 32'b0);
```

CONTROL UNIT:

```
20  //////////////////////////////////////////////////////////////////////////////////
21  module control_unit(
22      input logic [6:0]opcode,
23      output logic regwrite,
24      output logic alusrc,
25      output logic memread,
26      output logic memwrite,
27      output logic memtoreg,
28      output logic branch,
29      output logic [1:0] aluop
30  );
31      always_comb
32      begin
33      regwrite = 1'b0;
34      alusrc   = 1'b0;
35      memread  = 1'b0;
36      memwrite = 1'b0;
37      memtoreg = 1'b0;
38      branch   = 1'b0;
39      aluop    = 2'b00;
40
41      case(opcode)
42        7'b0000011:  //lw
43        begin
44      regwrite = 1'b1;
45      alusrc   = 1'b1;
46      memread  = 1'b1;
47      memwrite = 1'b0;
48      memtoreg = 1'b1;
49      branch   = 1'b0;
50      aluop    = 2'b00;
51          end
52
53        7'b0110011:
```

```
83        7'b1100011:
84          begin
85      regwrite = 1'b0;
86      alusrc   = 1'b0;
87      memread  = 1'b0;
88      memwrite = 1'b0;
89      memtoreg = 1'b0;
90      branch   = 1'b1;
91      aluop    = 2'b01;
92          end
93          7'b1100111: //jalr
94      begin
95      regwrite = 1'b1;
96      alusrc   = 1'b1;
97      memread  = 1'b0;
98      memwrite = 1'b0;
99      memtoreg = 1'b0;
100     branch   = 1'b1;
101     aluop    = 2'b10;
102     end
103         7'b1100111: //jal
104     begin
105     regwrite = 1'b1;
106     alusrc   = 1'bX;
107     memread  = 1'b0;
108     memwrite = 1'b0;
109     memtoreg = 1'b0;
110     branch   = 1'b1;
111     aluop    = 2'bXX;
112     end
113         endcase
114 end
115 endmodule
116
```

ALU_CONTROL:

```
module alucontrol(
    input  logic [1:0] op,
    input  logic [2:0] x,
    input  logic       y,
    output logic [3:0] out
);

always_comb begin
    case(op)
        2'b00: out = 4'b0000;
        2'b01: out = 4'b0001;
        2'b10: begin
            case(x)
                3'b000: out = y ? 4'b0001 : 4'b0000;
                3'b010: out = 4'b0101;
                3'b011: out = 4'b0110;
                3'b100: out = 4'b0100;
                3'b110: out = 4'b0011;
                3'b111: out = 4'b0010;
                3'b001: out = 4'b0111;
                3'b101: out = y ? 4'b1001 : 4'b1000;
                default: out = 4'b0000;
            endcase
        end
        default: out = 4'b0000;
    endcase
end

endmodule
```

TOP MODULE:

```
20  //////////////////////////////////////////////////////////////////////////////////
21  module Top2#(
22      parameter A = 32,
23      parameter B = 116,
24      parameter E = 32,
25      parameter D = 8,
26      parameter NUM_REG = 32,
27      parameter G = 5,
28      parameter H = 32,
29      parameter I = 7
30  )(
31      input  logic clk,
32      input  logic reset
33  );
34
35      logic [A-1:0] pc_out , pc_in;
36      logic [E-1:0] instruction;
37      logic we;
38      logic [G-1:0] rsadd1, rsadd2, rdadd;
39      logic [I-1:0] opcode;
40      logic [E-1:0] reg_result1, reg_result2, alu_result , mux_res , imm_res , memresult;
41      logic [2:0] func3;
42      logic [6:0] func7;
43      logic [3:0] op;
44      logic aluSrc;
45      logic memwrite;
46      logic memread;
47      logic memtoreg;
48      logic branch;
49      logic [A-1:0]adderResult;
50      logic [1:0] signal;
51
52      PC pc_inst (
53          .clk(clk),
```

```
PC pc_inst (
    .clk(clk),
    .reset(reset),
    .pc_in(pc_in),
    .add(pc_out)
);

InsMemory #(B, E, D) insmemory_inst (
.addr(pc_out),
.dataR(instruction)
);

Decoder decoder_inst (
.instruction(instruction),
.opcode(opcode),
.rdadd(rdadd),
.func3(func3),
.rsadd1(rsadd1),
.rsadd2(rsadd2),
.func7(func7)
);
```

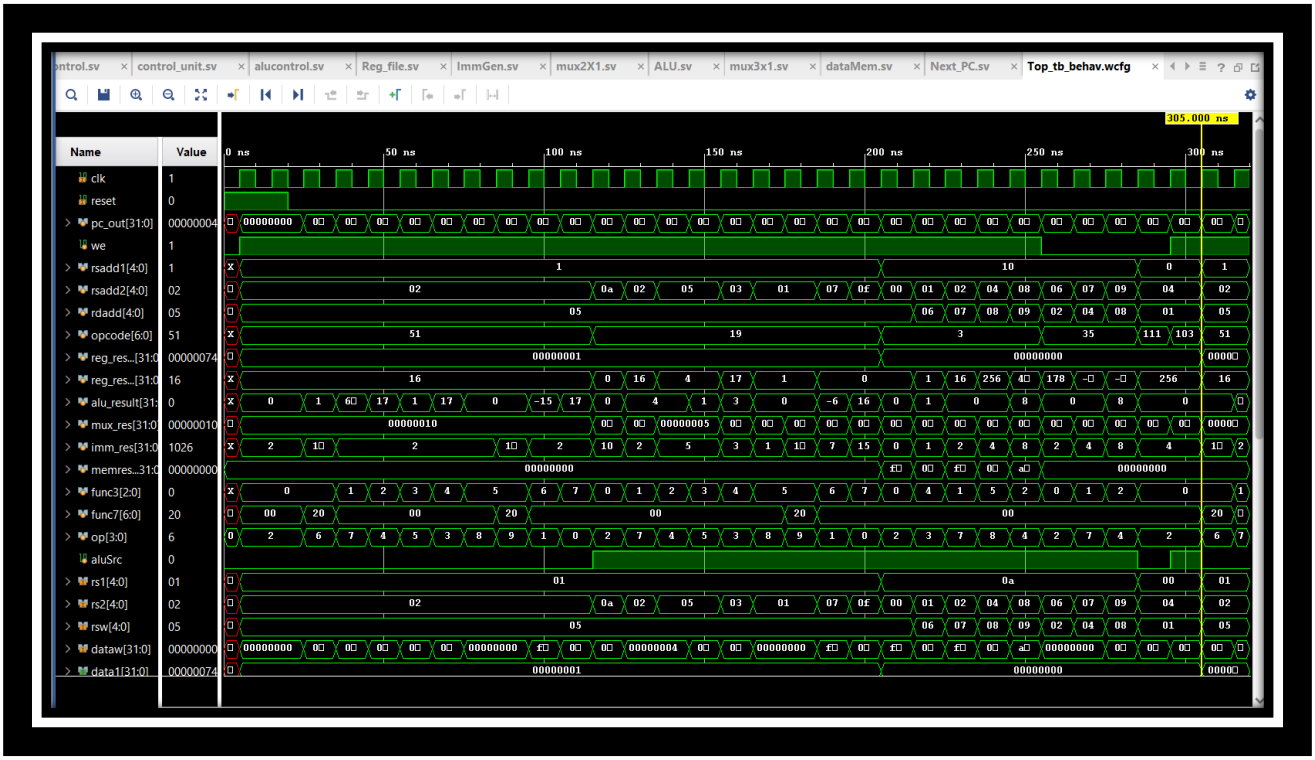
```
74      top_control_alu control_unit (
75          .opcode(opcode),
76          .funct3(func3),
77          .funct7(func7[5]),
78          .alu_ctrl(op),
79          .regwrite(we),
80          .alusrc(aluSrc),
81          .memread(memread),
82          .memwrite(memwrite),
83          .memtoreg(memtoreg),
84          .branch(branch)
85
86      );
87      Reg_file #(E, NUM_REG, G) regfile_inst (
88          .clk(clk),
89          .we(we),
90          .rs1(rsadd1),
91          .rs2(rsadd2),
92          .rsw(rdadd),
93          .dataw(adderResult),
94          .data1(reg_result1),
95          .data2(reg_result2)
96      );
97      ImmGen imm_inst (
98          .instr(instruction),
99          .imm(imm_res)
100     );
101
102     mux mux_inst (
103         .ri(reg_result2),
104         .li(imm_res),
105         .sl(aluSrc),
106         .res(mux_res)
107      );
```

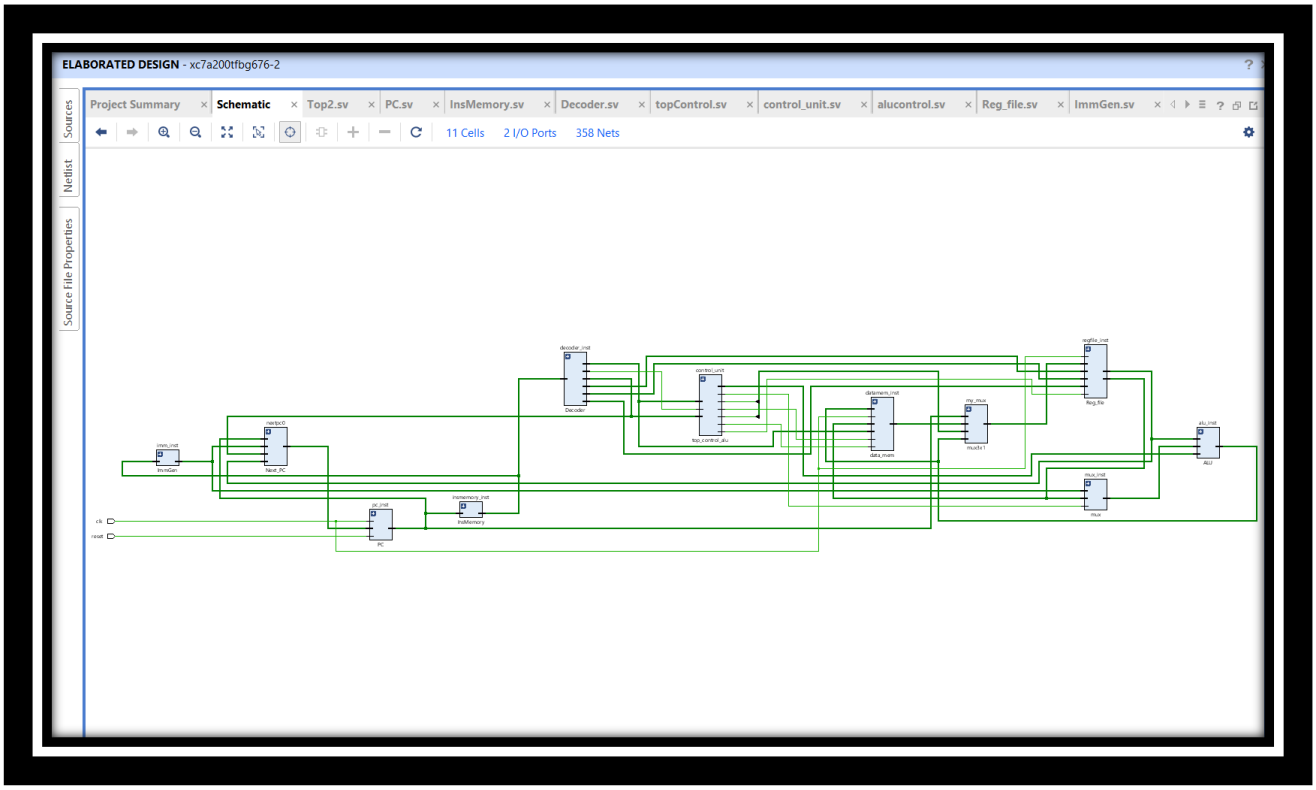
```
101
102     mux mux_inst (
103         .ri(reg_result2),
104         .li(imm_res),
105         .sl(aluSrc),
106         .res(mux_res)
107      );
108
109     ALU #(H) alu_inst(
110         .A(reg_result1),
111         .B(mux_res),
112         .opcode(op),
113         .result(alu_result)
114     );
115
116    assign  signal = {branch , memtoreg} ;
117
```

```
always_comb begin
    branch_taken = 0;
    if (branch) begin
        case(func3)
            3'b000: if (zeroFlag) branch_taken = 1;
            3'b001: if (!zeroFlag) branch_taken = 1;
            3'b100: if (alu_result == 1) branch_taken = 1;
            3'b101: if (alu_result == 0) branch_taken = 1;
            3'b110: if (alu_result == 1) branch_taken = 1;
            3'b111: if (alu_result == 0) branch_taken = 1;
        endcase
    end
end

assign mux_sig = branch_taken;
    mux branch_mux (
    .ri(add1),
    .li(add2),
    .sl(mux_sig),
    .res(pc_in)
);
endmodule
```

TOP_TB_FILE:

```
module Top_tb;

    logic clk;
    logic reset;

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

    always #5 clk = ~clk;

    initial begin
        clk = 0;
        reset = 1;

        #20 reset = 0;

        #350;

        $finish;
    end

endmodule
```


control_unit.sv
alucontrol.sv
Reg_file.sv
ImmGen.sv
mux2X1.sv
ALU.sv
mux3x1.sv
dataMem.sv
Next_PC.sv
Top_tb_behav.wcfg
305.000 ns
Name
Value
0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns
clk
reset
pc_out[31:0]
we
rsadd1[4:0]
rsadd2[4:0]
rdadd[4:0]
opcode[6:0]
reg_res...[31:0]
reg_res...[31:0]
alu_result[31:0]
mux_res[31:0]
imm_res[31:0]
memres...31:0]
func3[2:0]
func7[6:0]
op[3:0]
aluSrc
rs1[4:0]
rs2[4:0]
rsw[4:0]
dataw[31:0]
data1[31:0]


ELABORATED DESIGN - xc7a200tfbg676-2
Project Summary
Schematic
Top2.sv
PC.sv
InsMemory.sv
Decoder.sv
topControl.sv
control_unit.sv
alucontrol.sv
Reg_file.sv
ImmGen.sv
11 Cells
2 I/O Ports
358 Nets
Sources
Netlist
Source File Properties

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 writeback 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.