NATIONAL UNIVERSITY OF SCIENCE AND TECHNALOGY

DEPARTMENT OF COMPUTER AND SOFTWARE ENGINEERING

Lab#12: Adding the Control Unit to the 5-Stage Pipelined RISC-V Processor

Lab Report # 12

Course Instructor: *Dr Shahid Ismail* **Lab**

Instructor: *Usama Shoukat*

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

Due Date: 10-Dec-2025
Submission Date: 06-Dec-2025

TABLE OF CONTENT	
1)	Introduction
2)	Objectives
3)	Software or Equipments
4)	Lab tasks
5)	Outputs
6)	Conclusion

INTRODUCTION:

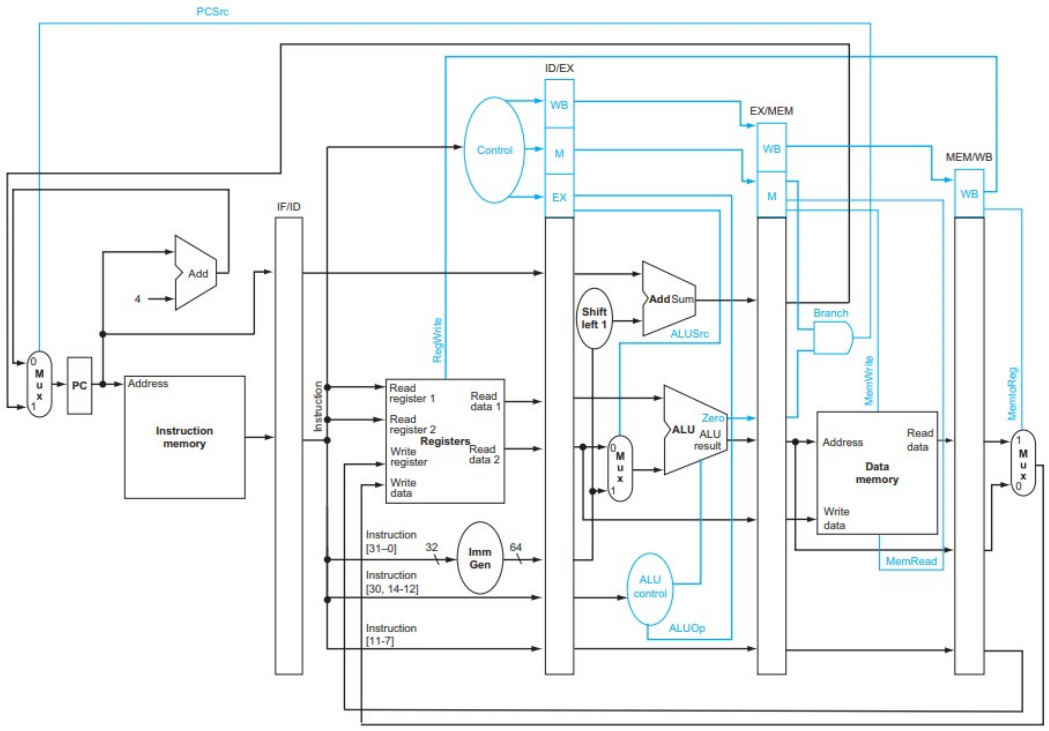
. Understanding the Role of the Control Unit

The Control Unit is responsible for generating all control signals required to execute an instruction. These signals include:

- **RegWrite**
- **ALUSrc**
- **MemRead**
- **MemWrite**
- **MemtoReg**
- **Branch**
- **ALUop**

In a pipelined CPU, these signals must be generated **in the ID stage** and then passed forward through pipeline registers.

2. Where Does the Control Unit Go? (Very Important)



Look at the pipeline diagram mentioned above:

1. Instruction is fetched in **IF**
2. Instruction is latched in **IF/ID register**
3. Instruction is decoded in **ID → This is where opcode becomes available**
4. Control Unit uses the opcode to generate control signals

Note: Therefore, the Control Unit must be placed in the ID stage, after the IF/ID register.

3. OBJECTIVE:

The objective of this lab is to add control to already created datapath of Pipeline the Design to Enhance the Number Instructions Executed Per Cycle

SOFTWARE\TOOL USED:

Xilinx Vivado

What Students Must Do in Lab

- Place the control unit in the ID stage after IF/ID register
- Extract opcode from instruction
- Connect opcode to the control unit
- Generate control signals (RegWrite, ALUop, etc.)

- Add these control signals to the ID/EX pipeline register
- Pass required signals forward into EX/MEM and MEM/WB registers
- Use the EX/MEM/WB versions of the control signals in their stages
- Verify with simple instructions (ADD, LW, SW, BEQ)

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:

```
)(
    input  logic [WIDTH-1:0] A,
    input  logic [WIDTH-1:0] B,
    input  logic [3:0] opcode,
    output logic [WIDTH-1:0] result,
    output logic zeroFlag
);
    always_comb begin
case(opcode)
    4'b0000:
        result = A + B;
    4'b0001:
        result = A - B;
    4'b0010:
        result = A & B;
    4'b0011:
        result = A | B;
    4'b0100:
        result = A ^ B;
    4'b0101:
        result = ($signed(A) < $signed(B)) ? 32'd1 : 32'd0;
    4'b0110:
        result = (A < B) ? 32'd1 : 32'd0;
    4'b0111:
        result = A << B[4:0];
    4'b1000:
        result = A >> B[4:0];
    4'b1001:
        result = $signed(A) >>> B[4:0];
    default:
        result = 32'd0;
endcase
end
        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
```

Pipelined Registers Module:

```
module PipelinedRegister_File #(parameter N = 64)
(
    input logic clk,
    input logic reset,
    input logic [N-1:0] in,
    output logic [N-1:0] out
 );
 always_ff@(posedge clk or posedge reset)
 begin
    if(reset)
    begin
    out <= 0;
    end
    else
    begin
    out <= in;
    end
 end
endmodule
```

TOP MODULE:

```
module Top#(
    parameter A = 32,
    parameter B = 182,
    parameter E = 32,
    parameter D = 8,
    parameter NUM_REG = 32,
    parameter G = 5,
    parameter H = 32,
    parameter I = 7
)(
    input  logic clk,
    input  logic reset
);

    logic [A-1:0] pc_out , pc_in;
    logic [E-1:0] instruction;
    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 [3:0] op;
    logic regWrite, aluSrc, memwrite, memread, memtoreg, branch, zeroFlag, branch_taken;
    logic [1:0] aluop;
    logic [A-1:0] add1 , add2 , shift;

    logic [63:0] IF_ID_out;
    logic [144:0] ID_EX_out;
    logic [109:0] EX_MEM_out;
    logic [70:0] MEM_WB_out;
```

```
logic [Z-1:0] IF_ID_out;
logic [Y-1:0] ID_EXE_out;
logic [X-1:0] EXE_MEM_out;
logic [W-1:0] MEM_WB_out;

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)
);

PipelinedRegister_File #(Z) IF_ID (
    .clk(clk),
    .reset(reset),
    .in({pc_out, instruction}),
    .out(IF_ID_out)
);

Decoder decoder_inst (
    .instruction(IF_ID_out[31:0]),
    .opcode(opcode),
    .rdadd(rdadd),
    .func3(func3),
    .rsadd1(rsadd1),
    .rsadd2(rsadd2),
    .func7(func7)
);
```

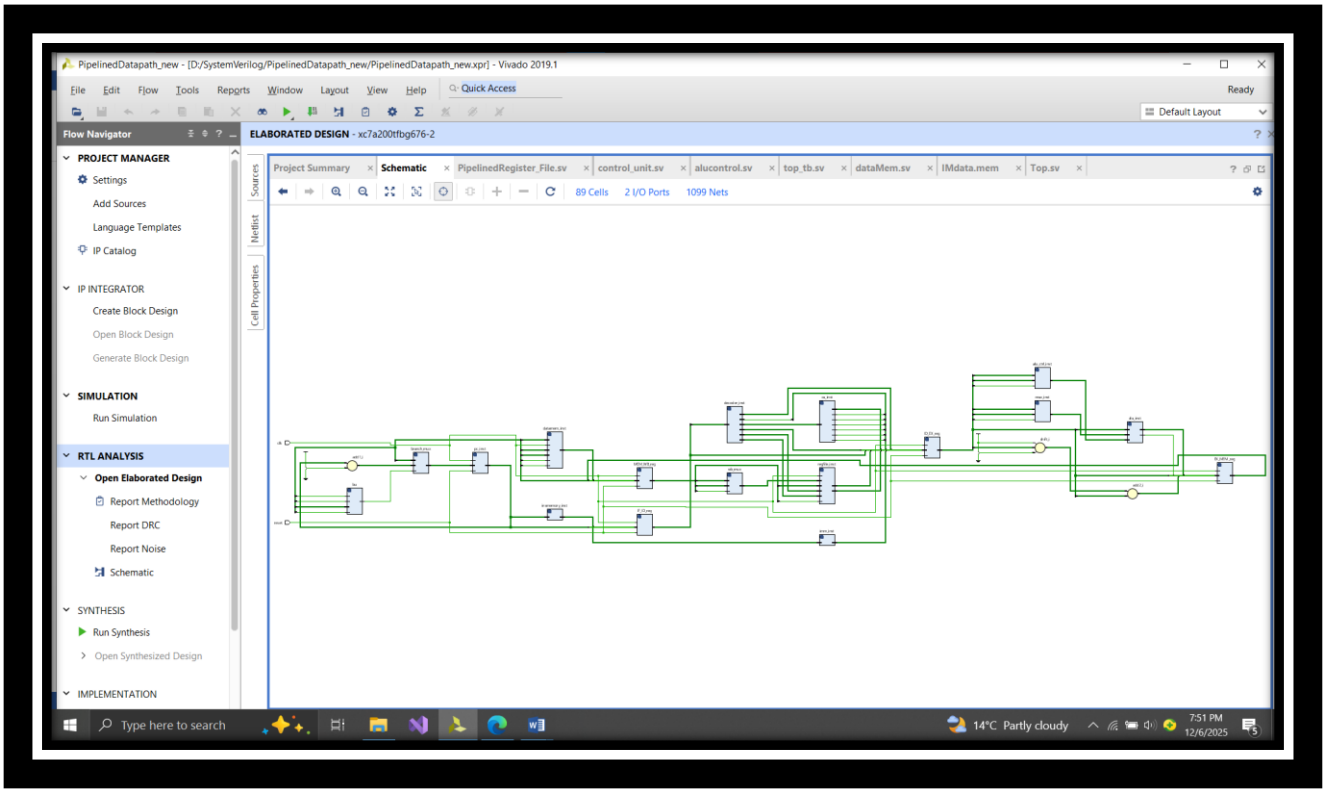
```
PipelinedRegister_File #(Y) ID_EXE (
    .clk(clk),
    .reset(reset),
    .in({
        IF_ID_out[63:32],
        reg_result1,
        reg_result2,
        imm_res,
        IF_ID_out[30],
        IF_ID_out[14:12],
        IF_ID_out[11:7],
        we, aluSrc, memread, memwrite, memtoreg, branch, op
    }),
    .out(ID_EXE_out)
);

mux mux_inst (
    .ri(ID_EXE_out[72:41]),
    .li(ID_EXE_out[104:73]),
    .sl(ID_EXE_out[138]),
    .res(mux_res)
);

ALU #(H) alu_inst(
    .A(ID_EXE_out[136:105]),
    .B(mux_res),
    .opcode(ID_EXE_out[142:139]),
    .result(alu_result),
    .zeroFlag(zeroFlag)
);

assign shift = ID_EXE_out[40:9] << 1;
assign add1 = IF_ID_out[63:32] + 4;              // PC+4
```

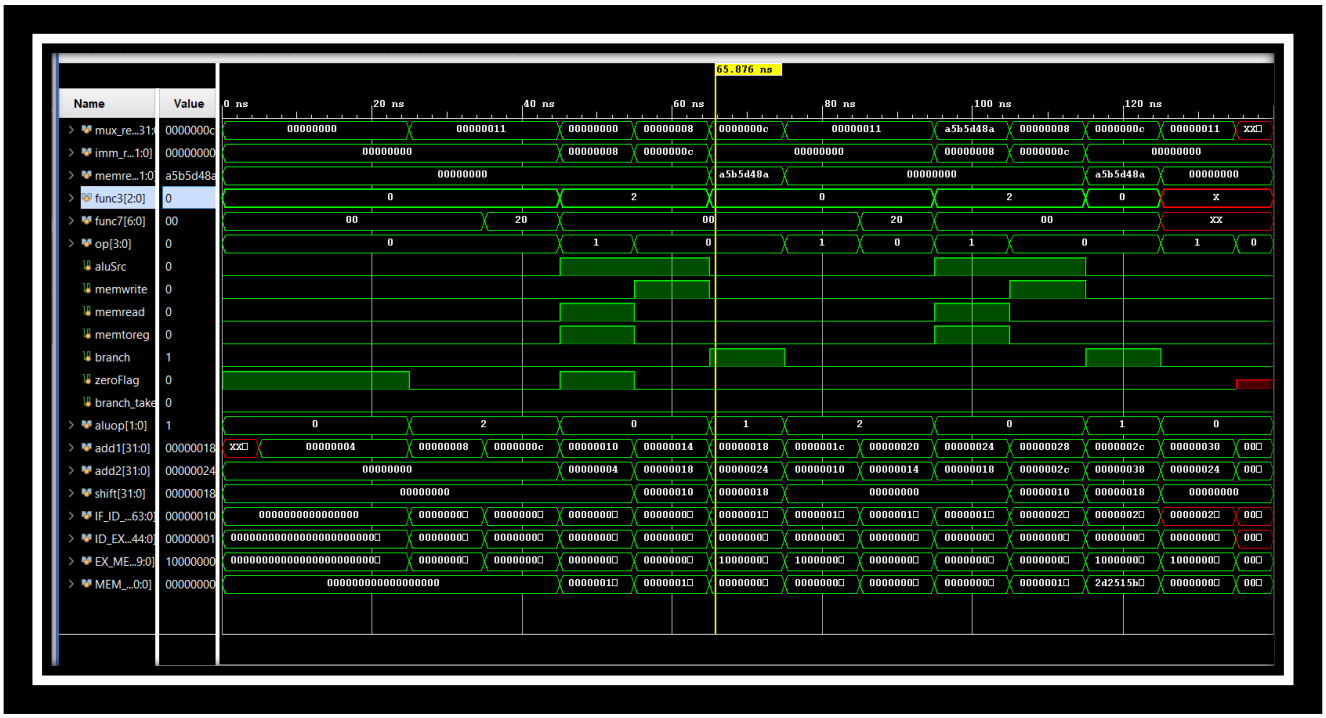
```
    PipelinedRegister_File #(X) EXE_MEM (
        .clk(clk),
        .reset(reset),
        .in({
            add2,                // branch target
            alu_result,          // ALU result
            ID_EXE_out[72:41],   // reg_result2 (for store)
            ID_EXE_out[12:8],    // rd
            ID_EXE_out[142:137] // control: {branch, memtoreg, memwrite, memread, we}
        }),
        .out(EXE_MEM_out)
    );

    data_mem #(1024) datamem_inst (
        .clk(clk),
        .mem_read(EXE_MEM_out[3]),
        .mem_write(EXE_MEM_out[2]),
        .funct3(func3),
        .addr(EXE_MEM_out[101:70]),
        .dataW(EXE_MEM_out[69:38]),
        .dataR(memresult)
    );

    PipelinedRegister_File #(W) MEM_WB (
        .clk(clk),
        .reset(reset),
        .in({
            EXE_MEM_out[101:70], // ALU result
            memresult,           // memory read
            EXE_MEM_out[37:33],  // rd
            EXE_MEM_out[1:0]     // control: {memtoreg, we}
        }),
        .out(MEM_WB_out)
    );
```

```
    mux wb_mux (
        .ri(MEM_WB_out[68:37]),
        .li(MEM_WB_out[36:5]),
        .sl(MEM_WB_out[0]),
        .res(wdata)
);
    BranchUnit bu(
        .branch(EXE_MEM_out[0]),
        .func3(func3),
        .zeroFlag(zeroFlag),
        .alu_result(EXE_MEM_out[101:70]),
        .branch_taken(branch_taken)
    );

    mux branch_mux (
        .ri(add1),
        .li(add2),
        .sl(branch_taken),
        .res(pc_in)
    );

endmodule
```

TOP TB FILE:

```
module Top_tb;
    logic clk;
    logic reset;
    Top dut (
        .clk(clk),
        .reset(reset)
    );
    initial clk = 0;
    always #5 clk = ~clk;
    initial begin
        reset = 1;
        #20 reset = 0;

        #500;
        $finish;
    end
    initial begin
        $dumpfile("Top_tb.vcd");
        $dumpvars(0, Top_tb);
    end
    always @(posedge clk) begin
        if (!reset) begin
            $display("Time: %0t | IF/ID PC: %0h | Instruction: %0h",
                     $time, dut.IF_ID_out[63:32], dut.IF_ID_out[31:0]);
            $display("Time: %0t | ID/EXE rs1: %0h | rs2: %0h | imm: %0h | rd: %0d",
                     $time, dut.ID_EXE_out[136:105], dut.ID_EXE_out[72:41], dut.ID_EXE_out[104:73], dut.ID_EXE_out[12:8]);
            $display("Time: %0t | EXE/MEM ALU result: %0h | branch target: %0h | rd: %0d",
                     $time, dut.EXE_MEM_out[101:70], dut.EXE_MEM_out[111:102], dut.EXE_MEM_out[37:33]);
            $display("Time: %0t | MEM/WB rd: %0d | WB data: %0h",
                     $time, dut.MEM_WB_out[37:33], dut.MEM_WB_out[36:5]);
            $display("-----------------------------------------------------------");
        end
    end
```





CONCLUSION:

Now our processor fetches instructions, decodes them, executes ALU operations, accesses memory, and writes results back. The pipeline moves data across stages every clock. You saw that wrong PC updates and wrong pipeline slicing can stop instructions from advancing. You confirmed that instruction memory must be initialized. You saw that each stage must get correct control bits or the ALU and memory give wrong outputs. The lab showed you how to trace signals, how to check pipeline flow, and how to validate each stage cycle by cycle. You learned how to use a testbench to watch PC, instructions, register values, ALU outputs, and write back results. The main value is that you now understand how a pipelined datapath works, how pipeline registers hold state, and how control signals must align with data.