Digital Design Verification

Lab

22 – Handling Control Hazards

Submitted by:

Name:	**Muhammad Farhan Shah**
Instructor:	**Dr Imran**

Date:
Nov 7, 2025

NUST Chip Design Centre (NCDC), Islamabad, Pakistan

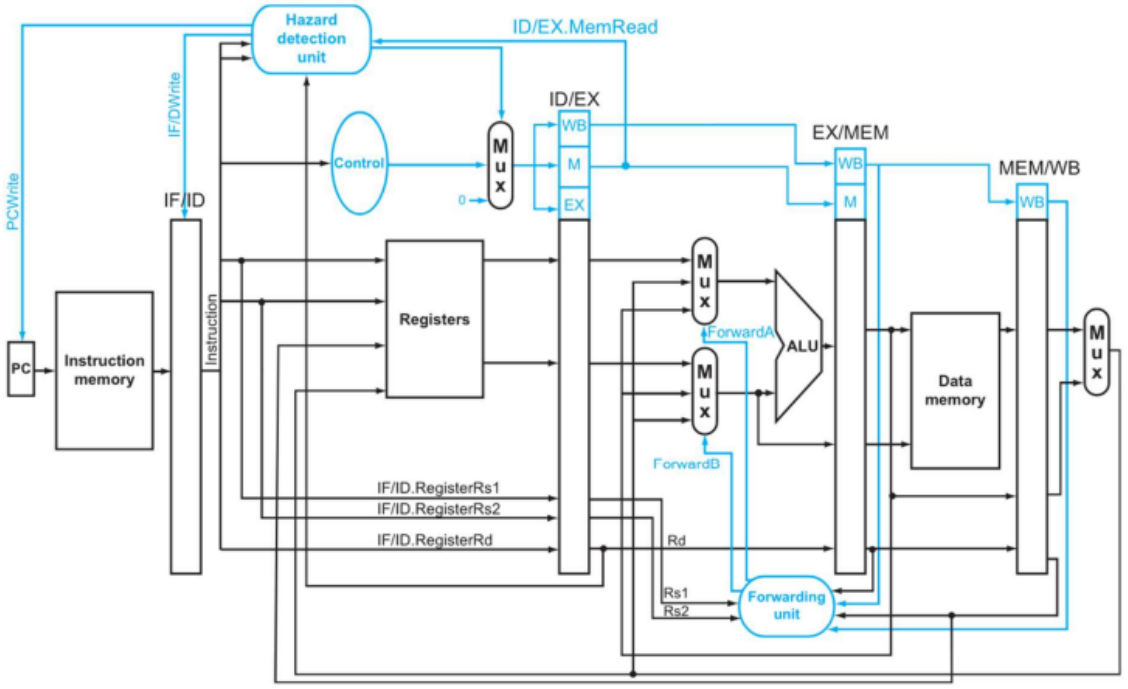
Objective

The objective of this lab is to

Append the design with additional logic to handle control hazards.

RISC-V Data and Control Paths

A rough estimate of pipeline datapath with hazard detection and forwarding is shown in the figure below.

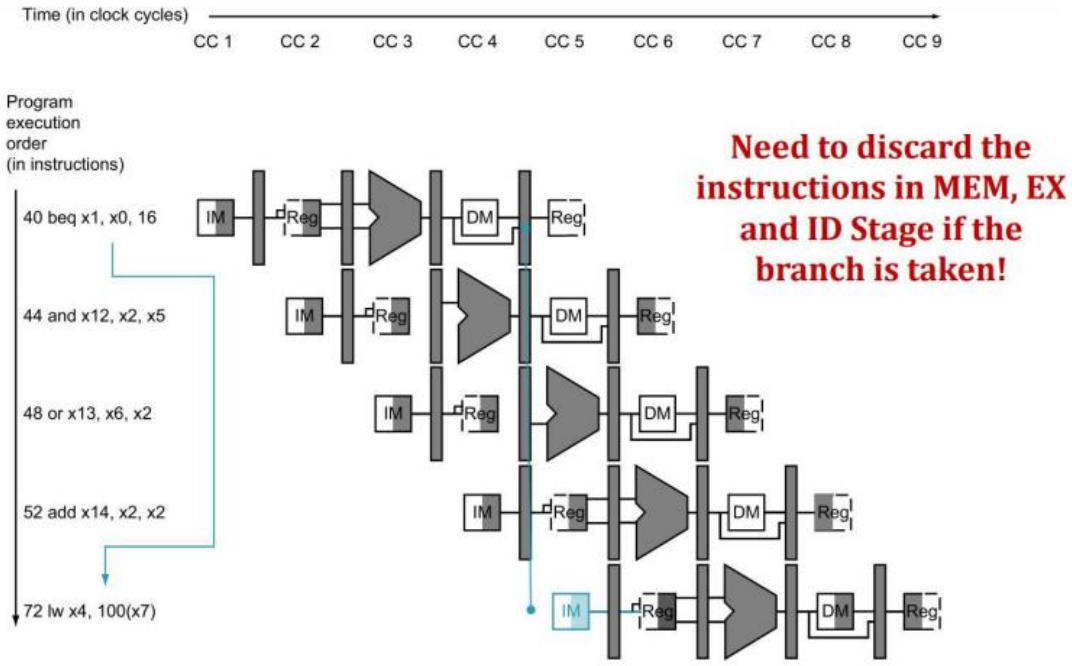


Pipelining: Control Hazards

A control hazard may occur when the flow of execution of a processor becomes unexpected. In the pipeline designed previously, a control hazard may occur in a branch instruction as shown in the figure below. Branches cause control hazards because branch decision is delayed, and the processor continues execution until the decision has been made.

In other words, a branch predictor exists with the assumption that branch is not taken. In the case that branch needs to be taken, the processor must flush all the instructions following the branch instruction. To flush, you must disable register and memory writes, however, you may not disable writing of pipeline registers. This can be done in a similar manner to disabling RegWrite in Data Hazards.

Specifically, the data needs to be flushed in IF/ID and ID/EX Pipeline Registers



\

Logic to tackle control hazards:

Forward to solve data hazards when possible[1]:

if $((Rs1E == RdM)\ \&\ RegWriteM)\ \&\ (Rs1E\ != 0)$ then
$ForwardAE = 10$
else if $((Rs1E == RdW)\ \&\ RegWriteW)\ \&\ (Rs1E\ != 0)$ then
$ForwardAE = 01$
else $ForwardAE = 00$

Stall when a load hazard occurs:

$lwStall = ResultSrcE_0\ \&\ ((Rs1D == RdE)\ |\ (Rs2D == RdE))$
$StallF = lwStall$
$StallD = lwStall$

Flush when a branch is taken or a load introduces a bubble:

$FlushD = PCSrcE$
$FlushE = lwStall\ |\ PCSrcE$

Code:

```
        //control hazards logic
      // assign StallF = Stall_F || PCSrcE_Sel;
      always_comb
      begin
        flush_D=0;
        if(PCSrcE) flush_D = 1;
        else flush_D = 0;
      end
       always_comb
           begin
             flush_E=0;
             if(lw_stall | PCSrcE) flush_E = 1;
             else flush_E = 0;
           end
```

Logic totally is implemented inside the Hazard control unit :

```
module HazardDetection(
  input  logic [4:0] Rs1E, Rs2E,RdE,//Rs2M,       // source regs in Execute
  input  logic [4:0] Rs1D, Rs2D, RdM, RdWB,    // destination regs
  input  logic regWrite_M, regWrite_W,PCSrcE,
  input logic ResultSrcE,//ResultSrcWB
  output logic [1:0] fwd_AE, fwd_BE,  // forwarding controls
  output logic flush_E,flush_D, stall_D, stall_F//wrdata_sel
);
logic lw_stall;
          always_comb begin
         fwd_AE = 2'b00;
     if ((Rs1E == RdM) & regWrite_M & (Rs1E != 0))
```

```
        fwd_AE = 2'b01;   // forward from MEM stage
    else if ((Rs1E == RdWB) & regWrite_W & (Rs1E != 0))
        fwd_AE = 2'b10;   // forward from WB stage
    else
        fwd_AE = 2'b00;
end


always_comb begin
    fwd_BE = 2'b00;
    if ((Rs2E == RdM) && regWrite_M && (Rs2E != 0))
        fwd_BE = 2'b01;   // forward from MEM stage
    else if ((Rs2E == RdWB) && regWrite_W && (Rs2E != 0))
        fwd_BE = 2'b10;   // forward from WB stage
    else
        fwd_BE = 2'b00;
end


     //stalls and flush logic
always_comb begin
            stall_F = 1'b1;
            stall_D = 1'b1;
            lw_stall = 1'b1;
        if(((ResultSrcE == 1'b1) && ((Rs1D == RdE) || (Rs2D == RdE)))) begin
                stall_F = 1'b0;
                stall_D = 1'b0;
```

```
                lw_stall = 1'b0;
            end
            else begin
                stall_F = 1'b1;
                stall_D = 1'b1;
                lw_stall = 1'b1;
            end
        end

//      //sw after lw
//      always_comb begin
//          wrdata_sel = 0;
//          if(ResultSrcWB && ((Rs2M == RdWB)))
//              wrdata_sel = 1;
//          else
//              wrdata_sel = 0;
//      end
        //control hazards logic
      // assign StallF = Stall_F || PCSrcE_Sel;
      always_comb
      begin
        flush_D=0;
        if(PCSrcE) flush_D = 1;
        else flush_D = 0;
      end
```

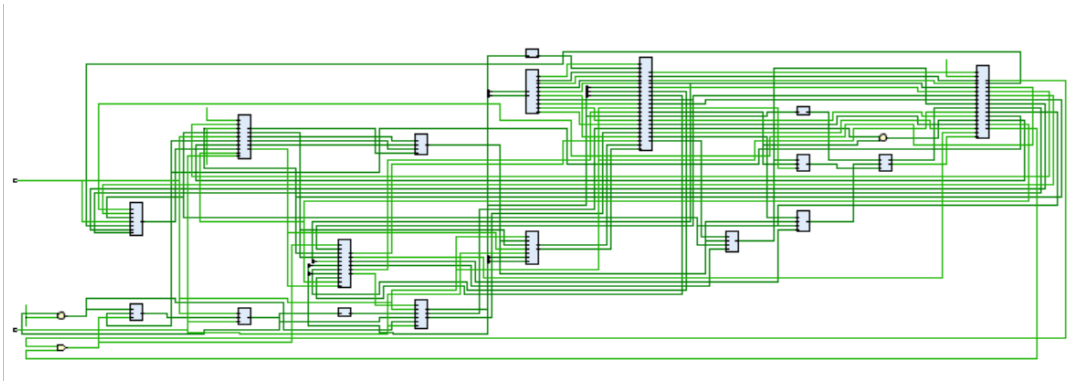
```
    always_comb
      begin
       flush_E=0;
       if(lw_stall | PCSrcE) flush_E = 1;
       else flush_E = 0;
      end
endmodule
```

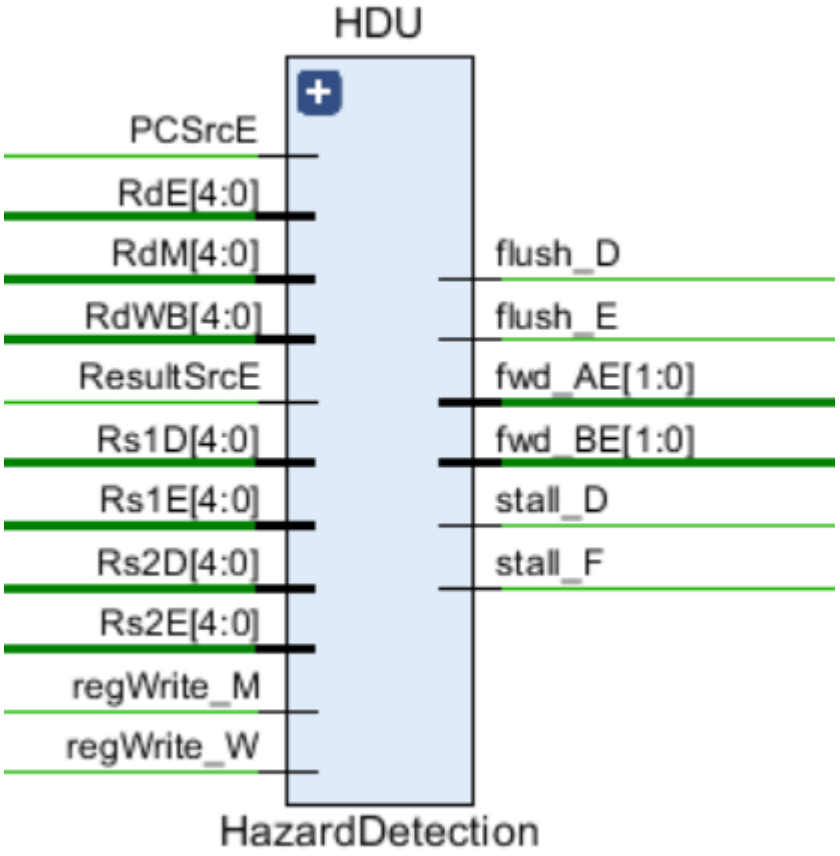
Instantiation:

```
HazardDetection HDU (
    // Source and dest register numbers
    .Rs1E(RS1D_reg1),
    .Rs2E(RS2D_reg1),
    .RdM(RdD_reg2),     // rd from Execute (ID/EX)
    .regWrite_M(regWrite_reg2),     // reg write from exmem
    .RdWB(RdD_reg3),    // rd from WriteBack (MEM/WB)
    .regWrite_W(regWrite_reg3),
    .RdE(RdD_reg1),
    .Rs1D(RS1D),
    .Rs2D(RS2D),
    .PCSrcE(PCsrcE),
    .ResultSrcE(MemToReg_reg1[0]),

    // Outputs
    .fwd_AE(fwd_AE),
    .fwd_BE(fwd_BE),
    .flush_D(rst_D),
    .flush_E(rst_E),
    .stall_F(IF_ID_Enable),
    .stall_D(ID_EX_Enable)
);
```

Elaborated Design:

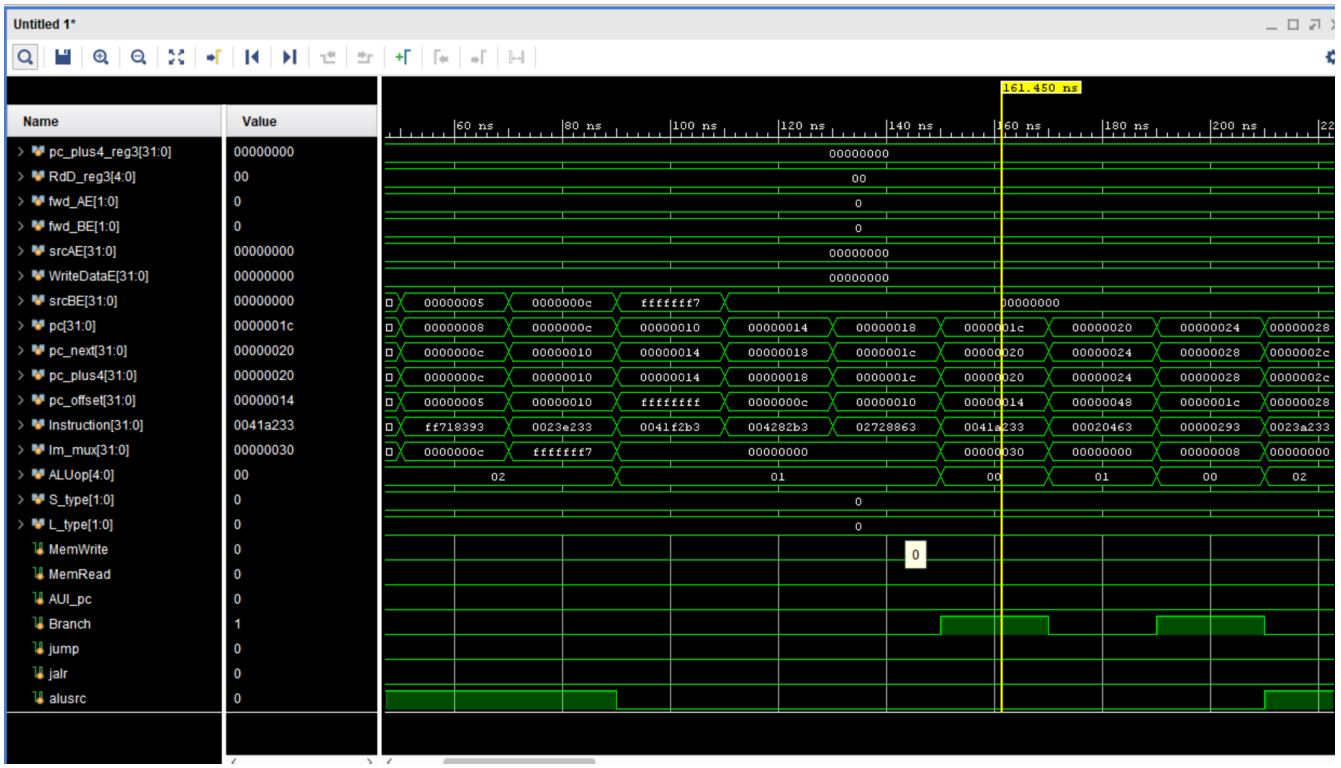


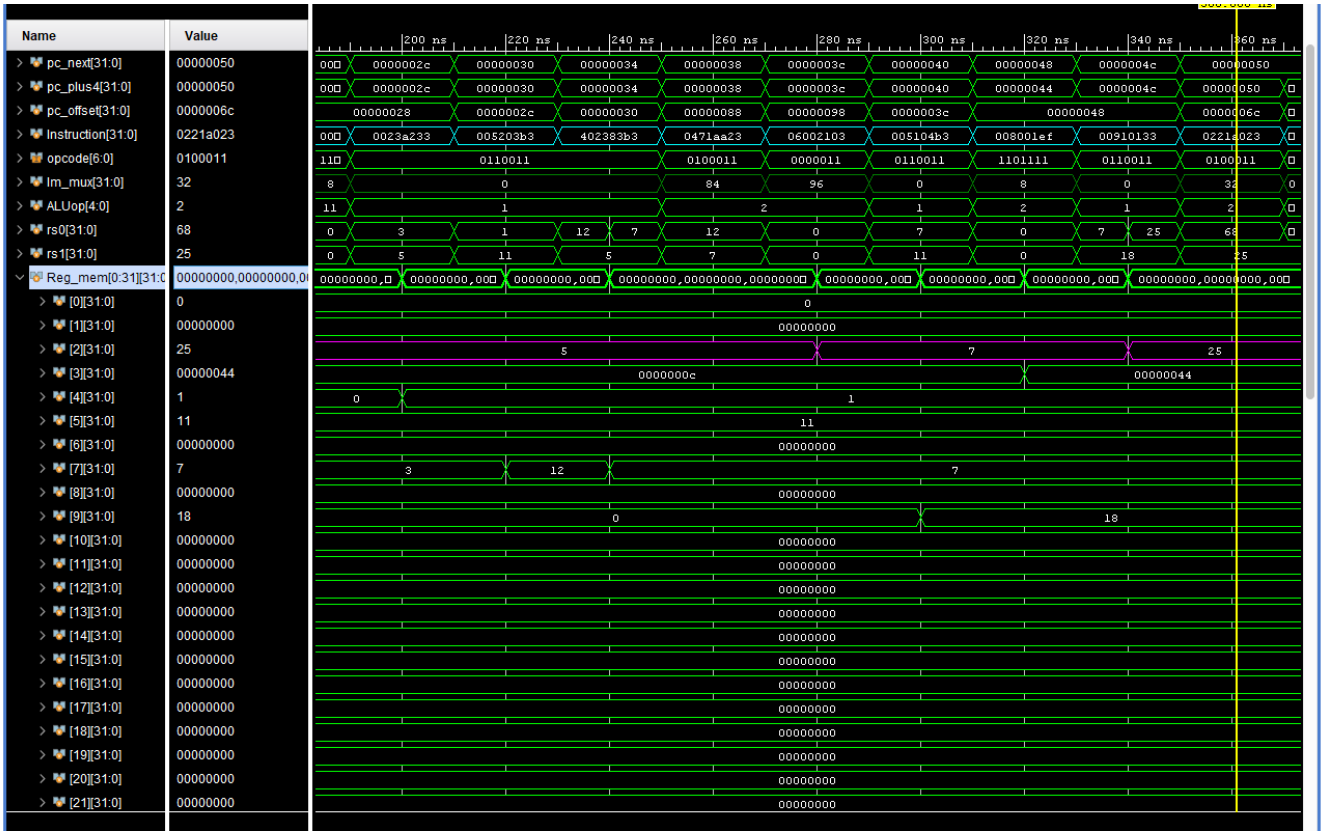


Tester File:

```
#
# Test the RISC-V processor:
#   add, sub, and, or, slt, addi, lw, sw, beq, jal
# If successful, it should write the value 25 to address 100
#       RISC-V Assembly         Description                 Address   Machine Code
main:   addi x2, x0, 5          # x2 = 5                    0         00500113
        addi x3, x0, 12         # x3 = 12                   4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3         8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7         C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4       10        0041F2B3
        add  x5, x5, x4         # x5 = 4 + 7 = 11           14        004282B3
        beq  x5, x7, end        # shouldn't be taken        18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0         1C        0041A233
        beq  x4, x0, around     # should be taken           20        00020463
        addi x5, x0, 0          # shouldn't execute         24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5) = 1          28        0023A233
        add  x7, x4, x5         # x7 = (1 + 11) = 12        2C        005203B3
        sub  x7, x7, x2         # x7 = (12 - 5) = 7         30        402383B3
        sw   x7, 84(x3)         # [96] = 7                  34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 7             38        06002103
        add  x9, x2, x5         # x9 = (7 + 11) = 18        3C        005104B3
        jal  x3, end            # jump to end, x3 = 0x44    40        008001EF
        addi x2, x0, 1          # shouldn't execute         44        00100113
end:    add  x2, x2, x9         # x2 = (7 + 18) = 25        48        00910133
        sw   x2, 0x20(x3)       # [100] = 25                4C        0221A023
done:   beq  x2, x2, done       # infinite loop             50        00210063
```

Simulation Output:





Conclusion:

In this project, we successfully implemented a pipelined RISC-V processor with a focus on handling data hazards, control hazards, and control signal propagation across the pipeline stages. The use of pipeline registers (IF_ID, ID_EX, EX_MEM, MEM_WB) allowed instruction execution to overlap, improving throughput compared to a single-cycle design. Data hazards were resolved through forwarding paths and hazard detection logic, ensuring correct operand selection without unnecessary stalls. Control hazards were addressed by generating branch decisions in the execution stage and applying flush signals to maintain instruction correctness. Additionally, proper control signal propagation across pipeline stages guaranteed consistent behavior for memory, ALU, and register operations.

Overall, the design demonstrates how hazard management and control flow are essential for maintaining correctness in a pipelined architecture while exploiting parallelism for performance gains. This implementation provides a strong foundation for further optimizations, such as branch prediction, exception handling, and support for a wider instruction set.