Digital Design Verification

Lab

22 – Handling Data Hazards

Submitted by:

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

Date:
Nov 5, 2025

NUST Chip Design Centre (NCDC), Islamabad, Pakistan

RISC-V Data and Control Paths

The pipelined data and control paths designed so far are shown in the figure below.

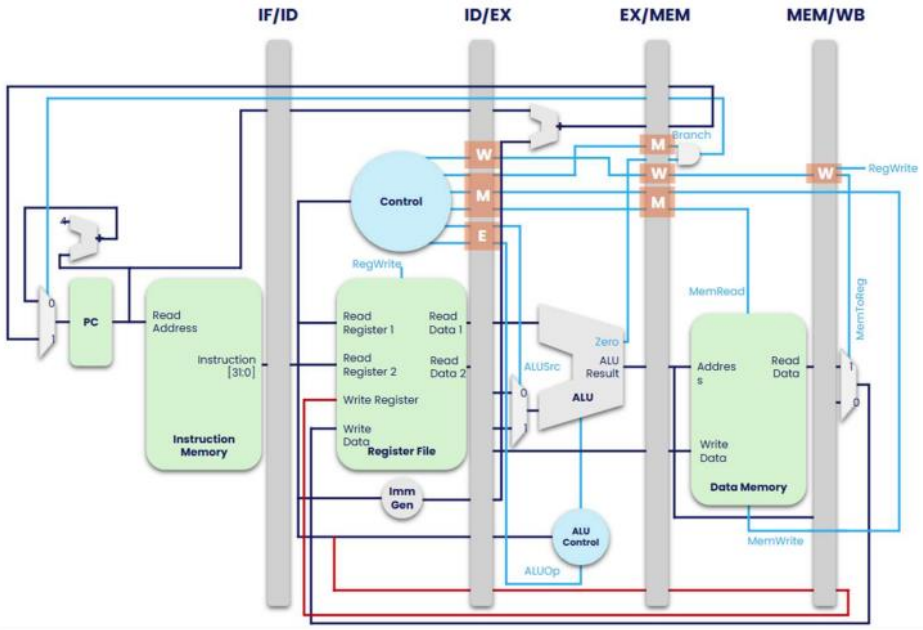


Figure 1: Pipelined Datapath

Pipelining: Data Hazards

In pipelined design, data, control and structural hazards can occur, which may disrupt the operation of processor leading to unexpected behavior. Let us discuss data hazards step-by-step in detail for sake of clarity.

Data Hazards: Forwarding to EX Stage

In the pipelined design, only one data hazard can occur, i.e. Read After Write (RAW).

Consider the following assembly codes,

Example 1	Example 2

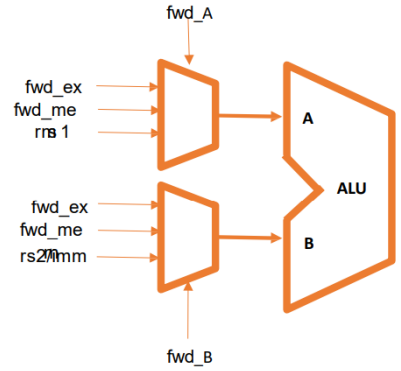
`add x10,x20,x21`
`add x11,x22,x23`
`add x12,x24,x25`
`addi x13,x12,1`
	`add x10,x20,x21`
	`add x11,x22,x23`
	`add x12,x24,x25`
	`addi x13,x11,1`

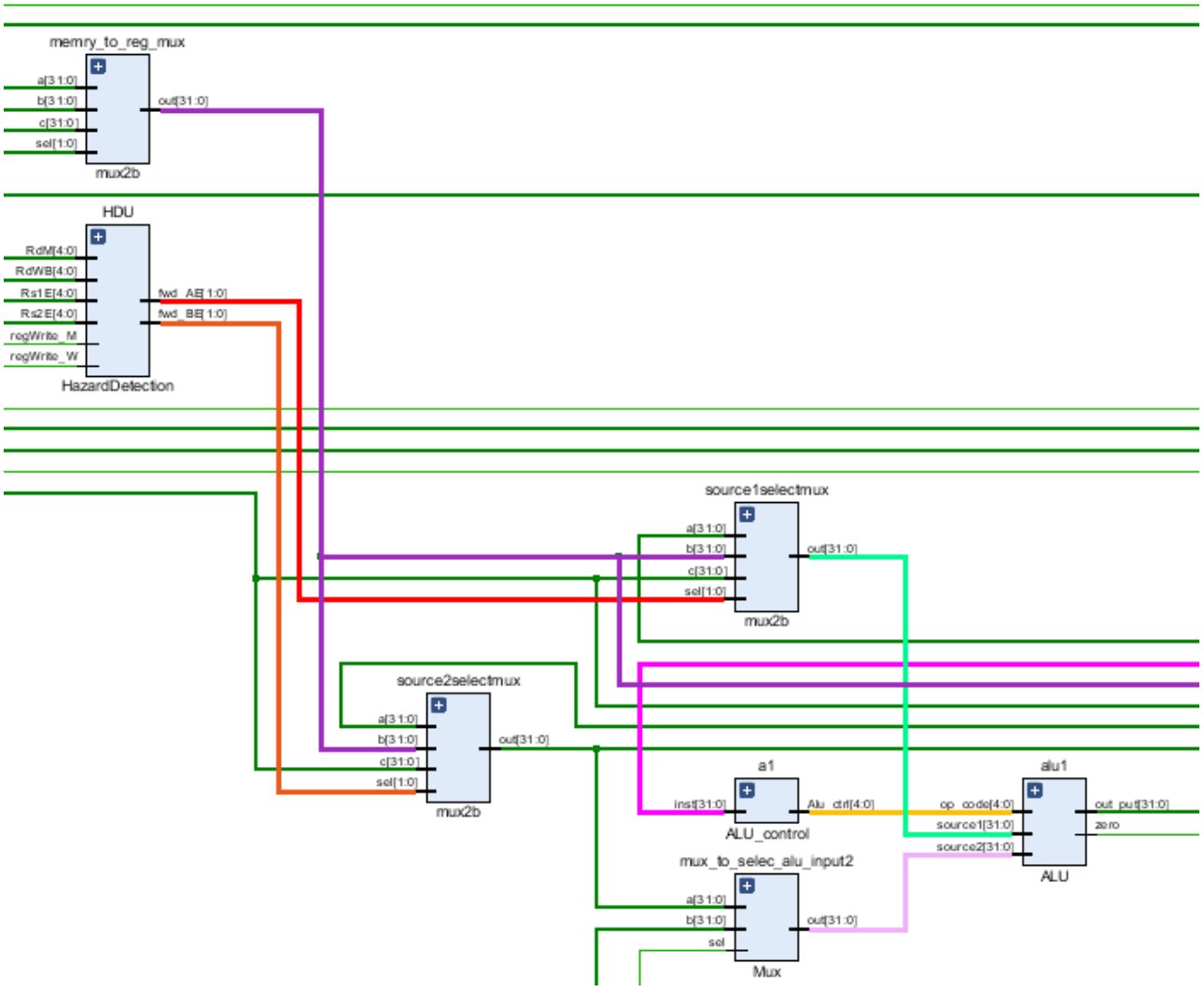
In the first example, the ***third instruction*** clashes with the ***highlighted instruction*** due to register **x12.** As the third instruction reaches MEM stage, highlighted instruction reaches EX Stage and requires the updated value of **x12**. Hence, we need to forward the operand from EX/MEM Pipeline Register to EX Stage.

A similar case occurs in the second example and the data needs to be forwarded from MEM/WB Register to EX Stage.

The resulting logic could be designed with an additional multiplexor immediately before ALU.

Control signals from the Hazard detection unit control the input to the ALU:





Two forwarding control signals “fwd_AE” and “fwd_BE” indicate whether forwarding is needed and from which stage.

The forwarding signals are generated by checking the conditions listed in the table below.

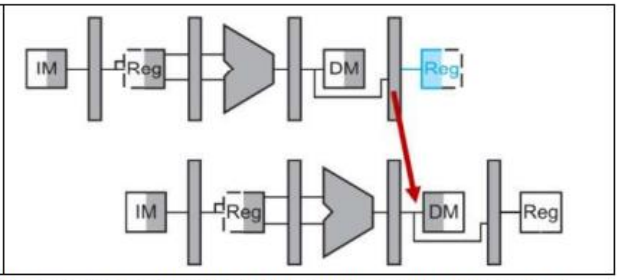
Address Comparison Conditions	Zero Check	RegWrite Check	Additional Conditions
EX/MEM.RegisterRd == ID/EX.RegisterRs1	Rd != x0	EX/MEM.RegWrite=1	NIL
EX/MEM.RegisterRd == ID/EX.RegisterRs2			
MEM/WB.RegisterRd == ID/EX.RegisterRs1		ME/WB.RegWrite=1	EX Hazard Must Not Exist
MEM/WB.RegisterRd == ID/EX.RegisterRs2			

```
//
always_comb begin
    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
    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
```

Data Hazards:

Forwarding to MEM Stage In consecutive load-store instructions, as shown in the example code below.

```
lw x1,offset(x2)
sw x1,offset(x3)
```

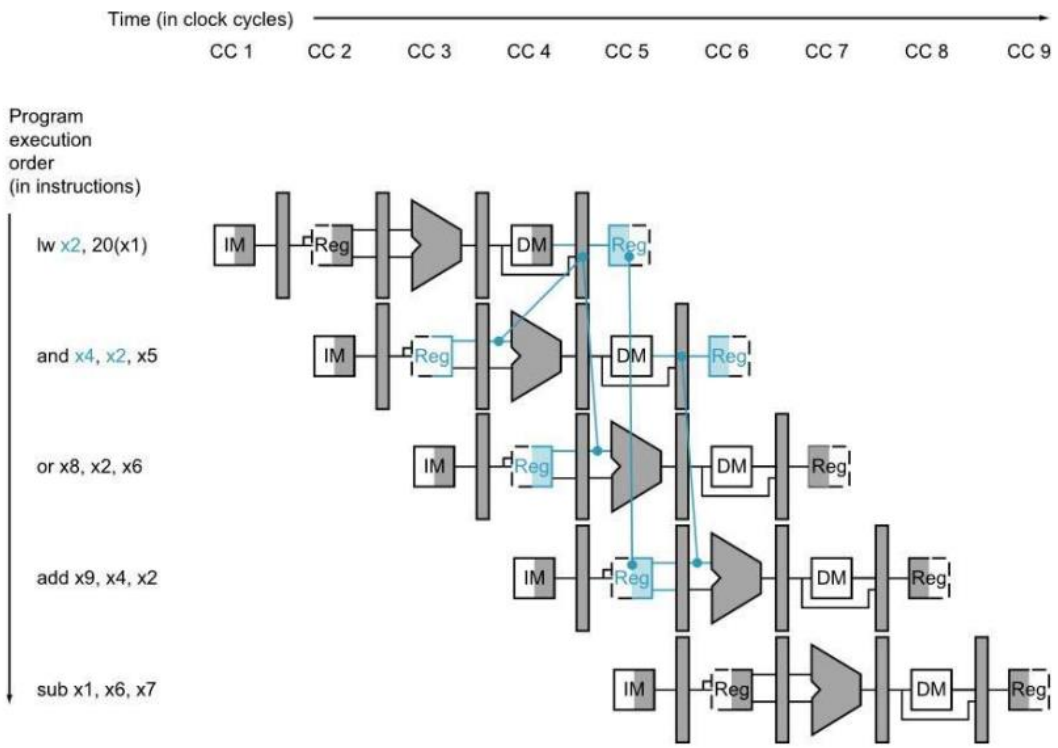


The data from load stage can be forwarded to the store instruction through multiplexors as before with an additional forwarding logic.

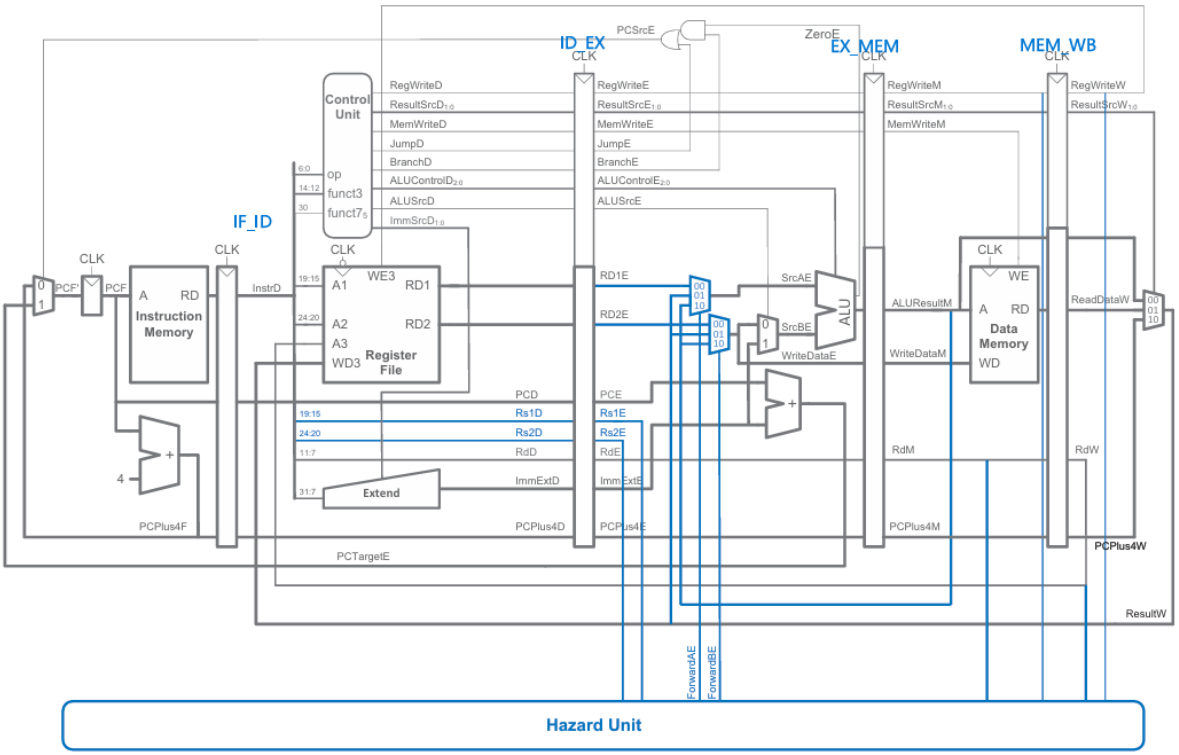
Data Hazards: **Stalling**

In certain situations, it becomes necessary to stall the pipeline. An example is given in the figure below, where data from load instruction is immediately required by “and” instruction. It is not possible to forward, because the “and” instruction would already have passed through execute stage one cycle before the data from load is available. Hence, we must stall.

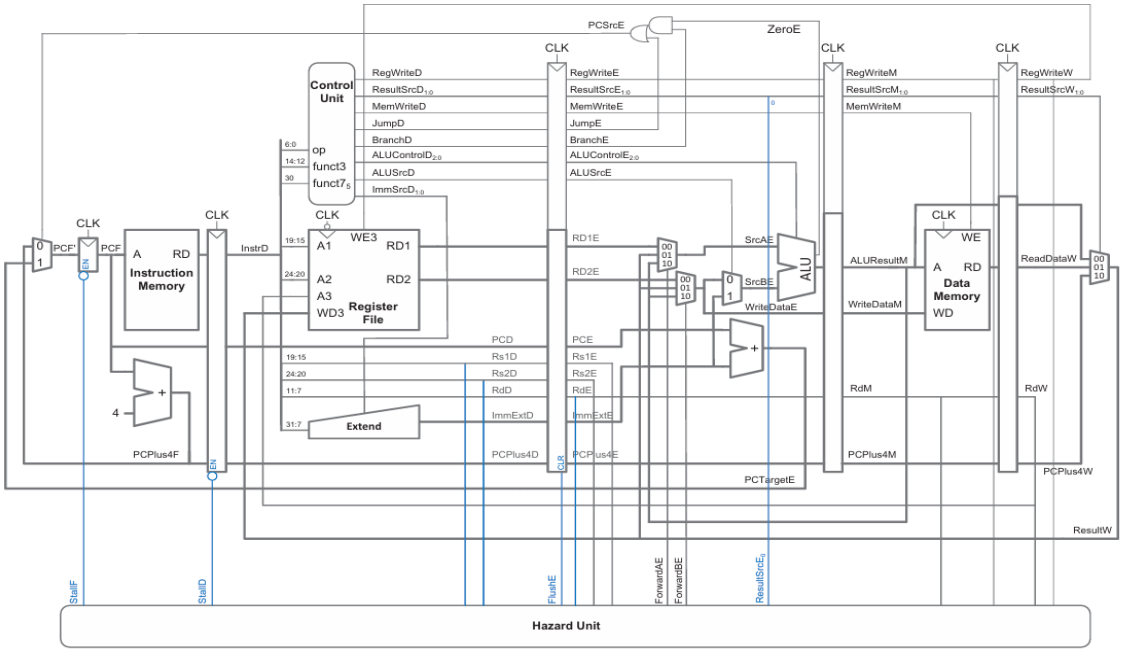
```
//stalls logic
always_comb begin
        if(((ResultSrcE == 1'b1) && !MemWrD && ((Rs1D == RdE) || (Rs2D == RdE)))) begin
                StallF = 1'b1;
                StallD = 1'b1;
                lwstall = 1'b1;
         end
        else begin
                StallF = 1'b0;
                StallD = 1'b0;
                lwstall = 1'b0;
         end
end

 //sw after lw
always_comb begin
       if(ResultSrcWB && ((Rs2M == RdWB)))
                wrdata_sel = 1;
       else
                wrdata_sel = 0;
end
```



Architecture Followed: Forwarding

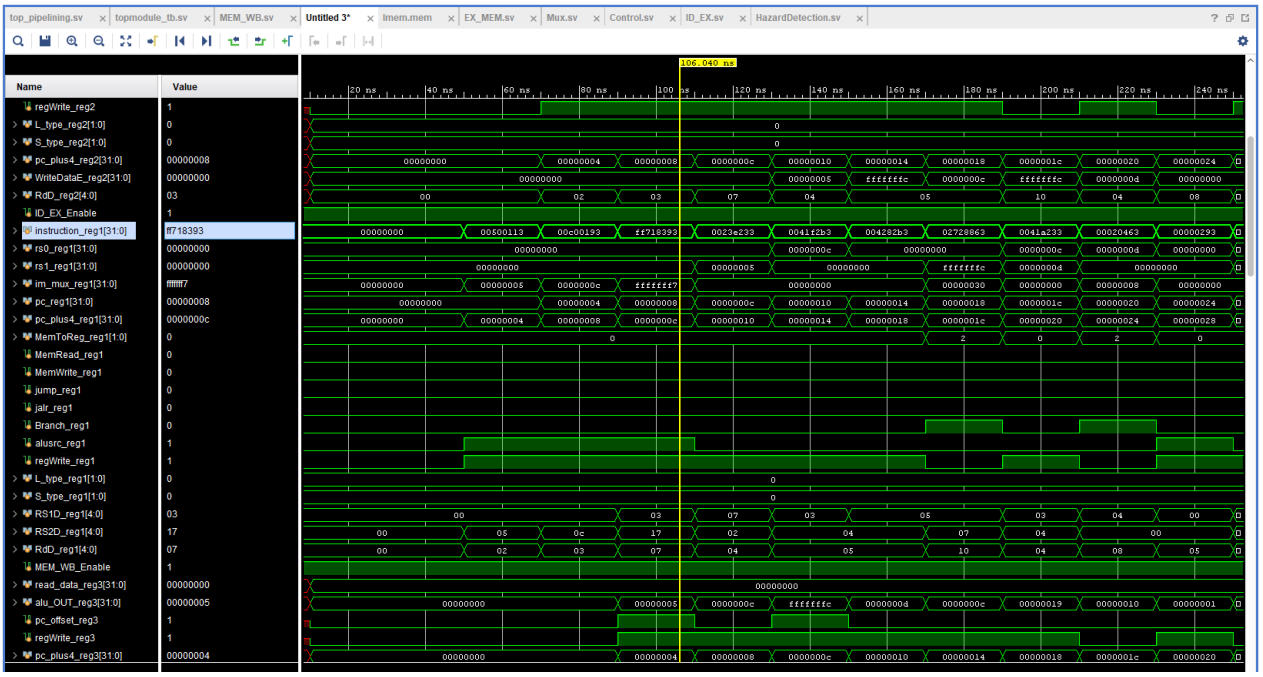


Architecture Followed: Stalling

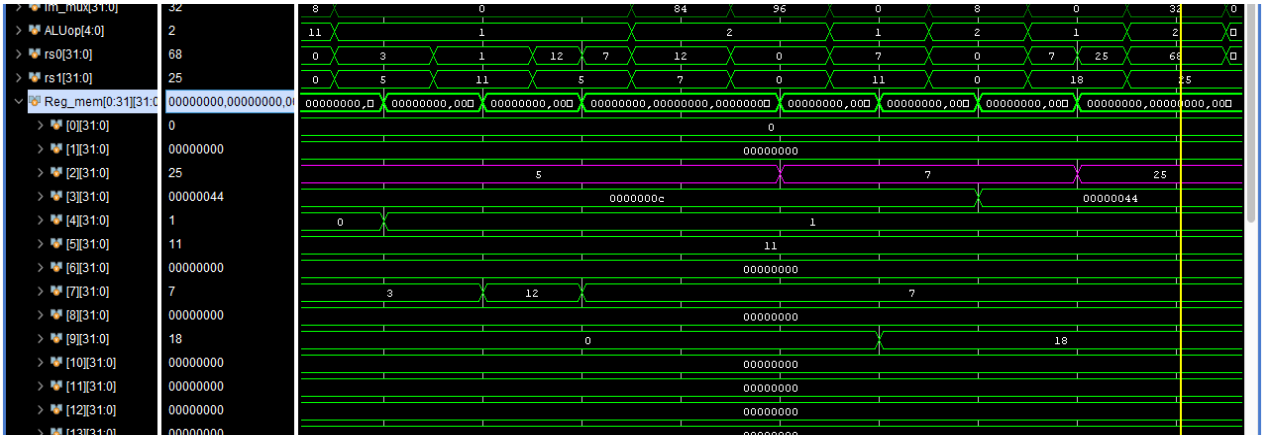


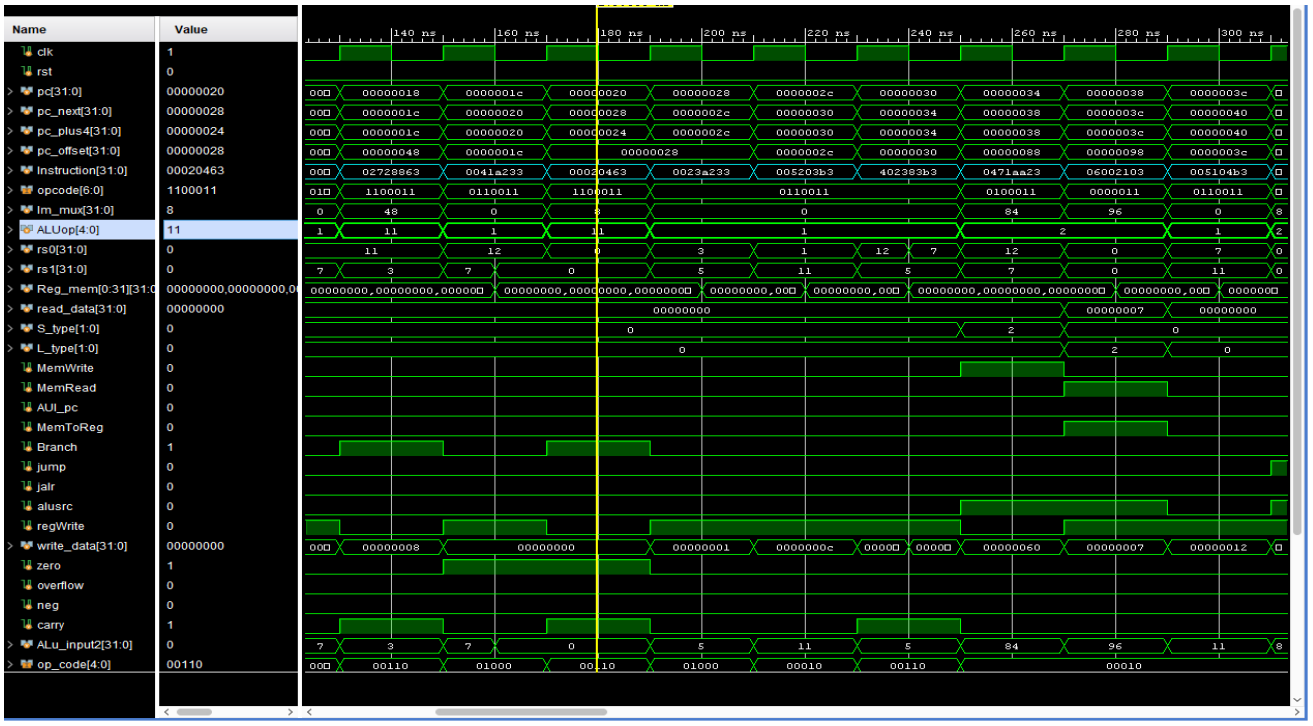
$$lwStall = ResultSrcE_0 \;\&\; ((Rs1D == RdE) \mid (Rs2D == RdE))$$
$$StallF = StallD = FlushE = lwStall$$

Simulation: outputs

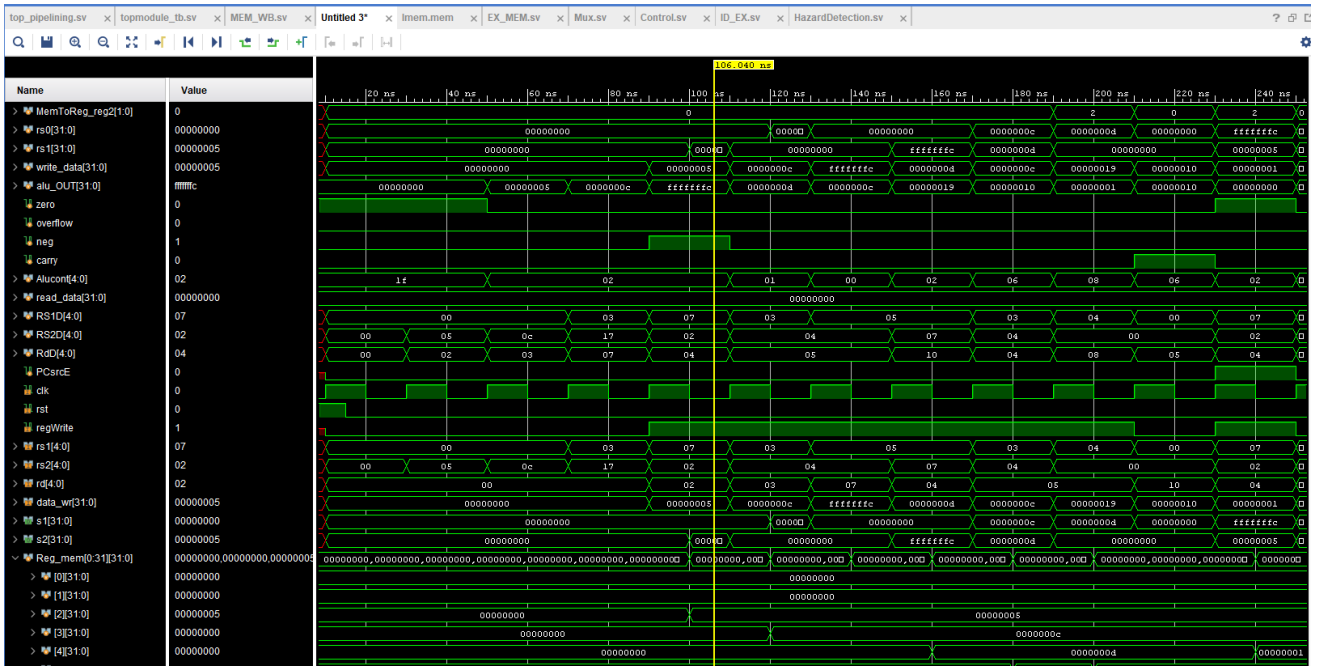


Values written to register




Name
Value
clk 1
rst 0
pc[31:0] 00000020
pc_next[31:0] 00000028
pc_plus4[31:0] 00000024
pc_offset[31:0] 00000028
Instruction[31:0] 00020463
opcode[6:0] 1100011
Im_mux[31:0] 8
ALUop[4:0] 11
rs0[31:0] 0
rs1[31:0] 0
Reg_mem[0:31][31:0]
read_data[31:0] 00000000
S_type[1:0] 0
L_type[1:0] 0
MemWrite 0
MemRead 0
AUI_pc 0
MemToReg 0
Branch 1
jump 0
jalr 0
alusrc 0
regWrite 0
write_data[31:0] 00000000
zero 1
overflow 0
neg 0
carry 1
ALu_input2[31:0] 0
op_code[4:0] 00110
140 ns
160 ns
180 ns
200 ns
220 ns
240 ns
260 ns
280 ns
300 ns

Sim Time: 1 us


top_pipelining.sv
topmodule_tb.sv
MEM_WB.sv
Untitled 3*
imem.mem
EX_MEM.sv
Mux.sv
Control.sv
ID_EX.sv
HazardDetection.sv
Name
Value
MemToReg_reg2[1:0]
rs0[31:0]
rs1[31:0]
write_data[31:0]
alu_OUT[31:0]
zero
overflow
neg
carry
Aluconf[4:0]
read_data[31:0]
RS1D[4:0]
RS2D[4:0]
RdD[4:0]
PCsrcE
clk
rst
regWrite
rs1[4:0]
rs2[4:0]
rd[4:0]
data_wr[31:0]
s1[31:0]
s2[31:0]
Reg_mem[0:31][31:0]
[0][31:0]
[1][31:0]
[2][31:0]
[3][31:0]
[4][31:0]
106.040 ns