ISTANBUL TECHNICAL UNIVERSITY FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING

VLSI Circuit Design II
EHB425E
2022 Spring
CRN:20989

Instructor: Sıddıka Berna Örs Yalçın
Teaching Assistant: Yasin Fırat Kula

HOMEWORK - 8

Oğuzhan Vatansever
040170022
Muhammed Erkmen
040170049

1) Pipelining the processor

In this step, single cycle processor will be converted to a 5-stage pipelined RISC-V processor.

These stages are Instruction Fetch (IF) stage, Instruction Decode (ID) stage, Execution(EX) stage, Memory (MEM) stage and Writeback (WB) stage. Figure 1 shows the order of stages.

IF STAGE
ID STAGE
EX STAGE
MEM STAGE
WRITEBACK STAGE
Figure 1: Order of pipeline stages

Main reason in pipeline is decreasing the critical path delay. So we'll put pipeline registers between these stages to implement pipeline and we'll cary the informations which will be needed in next stages by those registers.We named our pipeline registers as IF/ID, ID/EX, EX/MEM and MEM/WB and put these registers between this stages as shown in Figure 2.

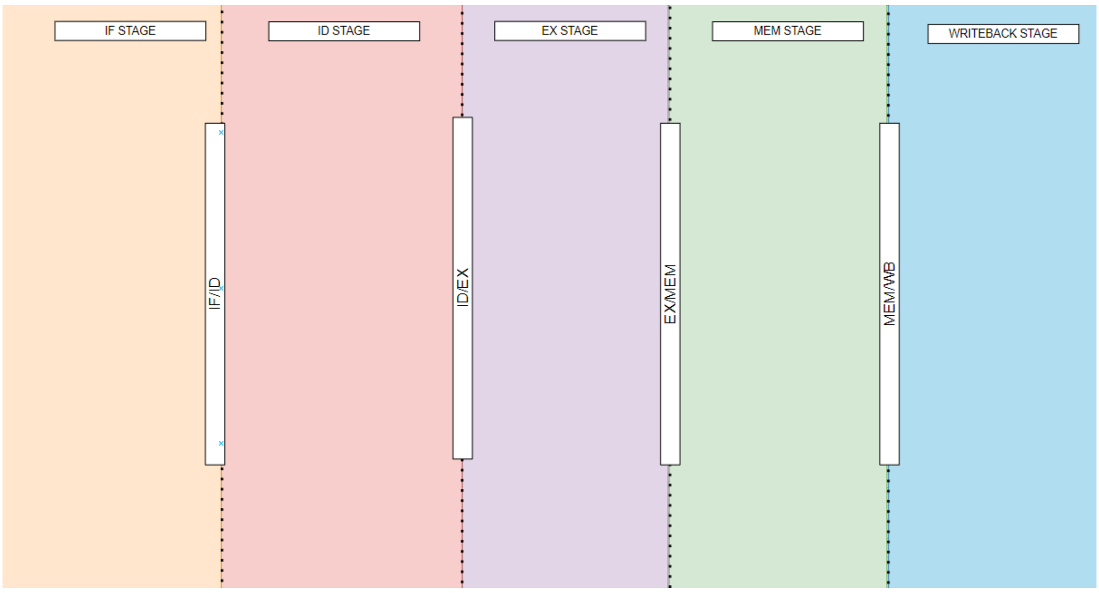


Figure 2: Pipeline registers between stages

In verilog code of the project, these registers are named as REG_IF_ID, REG_ID_EX, REG_EX_MEM, REG_MEM_WB.

Aim of Instruction Fetch (IF) stage is getting instruction from instruction memory. So we need to place instruction memory, program counter and an adder to calculate normal next value of PC which is PC+4 in this stage.

Main purpose of Instruction Decode (ID) stage is decoding the instruction and getting the needed datas. So in this stage we have to place register file, immediate generator and obviously instruction decoder.

Execution stage (EX) is the part which is calculations getting done. We make calculations in Functional Unit and make decisions in branch controller so these modules should be included in this stage. Also we calculate the PC+IMM values for some conditionary instructions. We decided to make it in this stage so an adder which is independent from FU is placed here.

Memory stage (MEM) is the part which includes memory operations. So we'll include data memory and load unit to this stage.

And in writeback (WB) stage we have a decision multiplexer.

After these informations, our diagram looks like in Figure 3 basically.

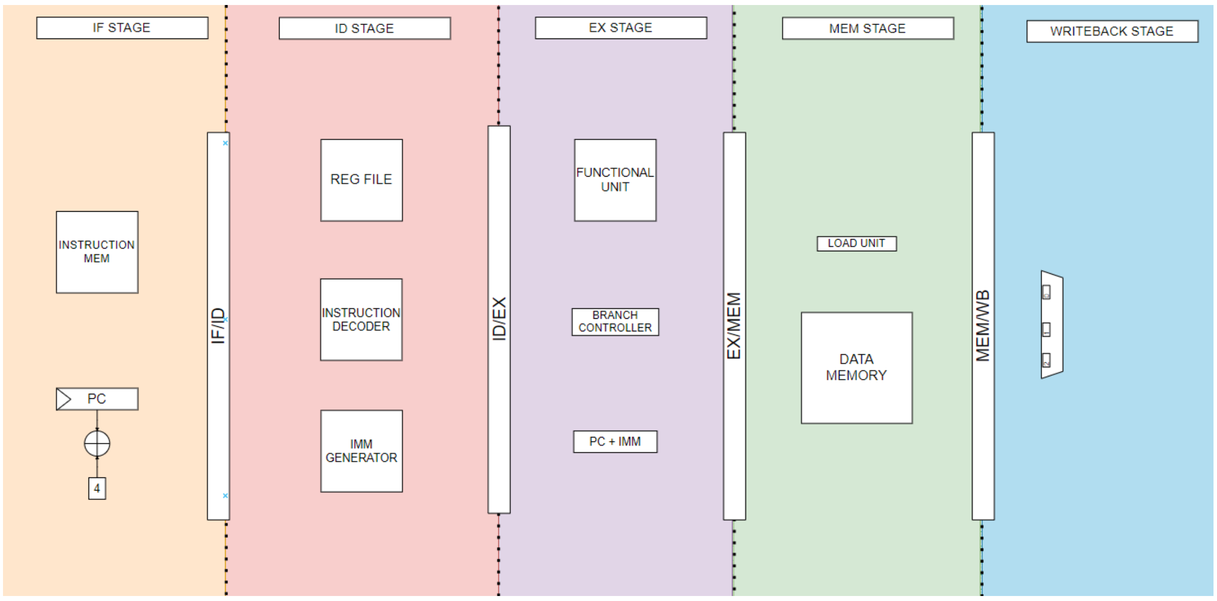
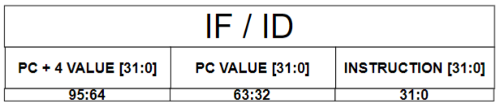


Figure 3: Basic diagram without connections

Now it is time to decide what to include to pipeline registers.

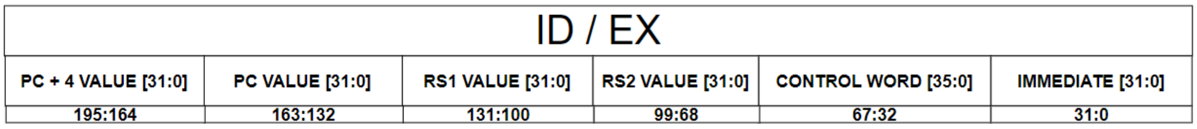
First of all, we carry our control word, PC value and PC + 4 value in every register.
IF/ID register is 96 bits in this part of assignment. Placement of values in this register is shown in Figure 4.



IF / ID		
PC + 4 VALUE [31:0]	PC VALUE [31:0]	INSTRUCTION [31:0]
95:64	63:32	31:0

Figure 4: IF/ID register mapping

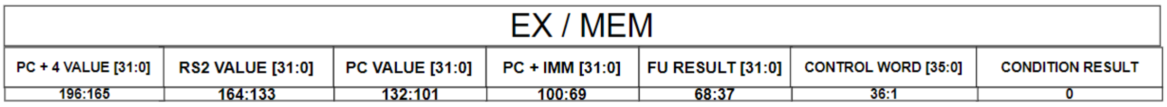
Then we carry PC and PC+4 values exactly same to ID/EX register. Also ID/EX register should store different values because in ID stage instruction has decoded,control word has created, immediate has generated and rs1 & rs2 values fetched. So this register should include these values too. Figure 5 shows the ID/EX register.



ID / EX					
PC + 4 VALUE [31:0]	PC VALUE [31:0]	RS1 VALUE [31:0]	RS2 VALUE [31:0]	CONTROL WORD [35:0]	IMMEDIATE [31:0]
195:164	163:132	131:100	99:68	67:32	31:0

Figure 5: ID/EX register mapping

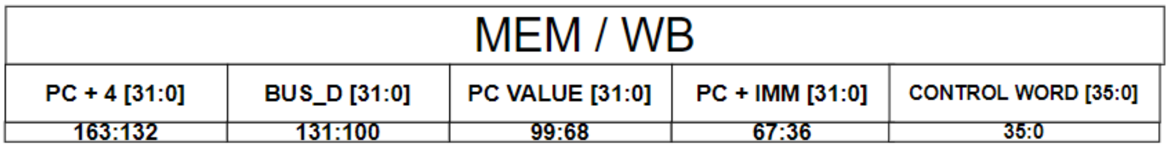
In EX stage, 4 value is generated. FU result value calculated by FU, immediate value created by immediate generator, condition result value operated by branch controller and PC+Immediate value calculated by independent adder. These values getting stored in EX/MEM register. Also we carry the PC+4 value, PC value, rs2 value for data in port, control word from ID/EX register to EX/MEM register. Figure 6 shows the EX/MEM register mapping.



EX / MEM						
PC + 4 VALUE [31:0]	RS2 VALUE [31:0]	PC VALUE [31:0]	PC + IMM [31:0]	FU RESULT [31:0]	CONTROL WORD [35:0]	CONDITION RESULT
196:165	164:133	132:101	100:69	68:37	36:1	0

Figure 6: EX/MEM register mapping

In MEM Stage, we'll have a value named BUS_D which is chosen between FU result and rd_out of data memory by Mux_D_Select. We get that value to MEM/WB register. We also carry control word, PC+4 and PC+Immediate values from EX / MEM register. Figure 7 shows the MEM/WB register mapping.



MEM / WB				
PC + 4 [31:0]	BUS_D [31:0]	PC VALUE [31:0]	PC + IMM [31:0]	CONTROL WORD [35:0]
163:132	131:100	99:68	67:36	35:0

Figure 7: MEM/WB register mapping

Now, we'll make connections between modules.

In IF stage, PC value should connected to address pin of Instruction Memory. Also we should select which PC is going to be next PC. For branch, JAL and JALR instructions We make this decision after EX stage. So there is an input which is selected by condition in EX Stage and that input goes to IF stage. Selection of this mux is EX/MEM register's branch | jal | jalr signal.

In ID stage, instruction inputs of instruction decoder and immediate generator is instruction from IF/ID register. Immediate generator gets immediate type input from instruction decoder. Register file gets rs1 and rs2 address values from instruction decoder and wr_din0, rd and wen0 signal comes from writeback stage.

Functional unit, branch controller and PC + Immediate adder are in EX stage. This stage has no input and outputs from other stage right now but that will change in the next questions.

Mem stage includes data memory and new pc output to IF stage. This stage has no input from other stages right now too but that will change either.

Writeback stage is the stage we send the address and data in to register file which placed in ID stage.

By making connections, question 1 of the sheet is done.

All connections and final view of our RISC-V pipelined core for question 1 can be seen in next page figure named Figure 8.

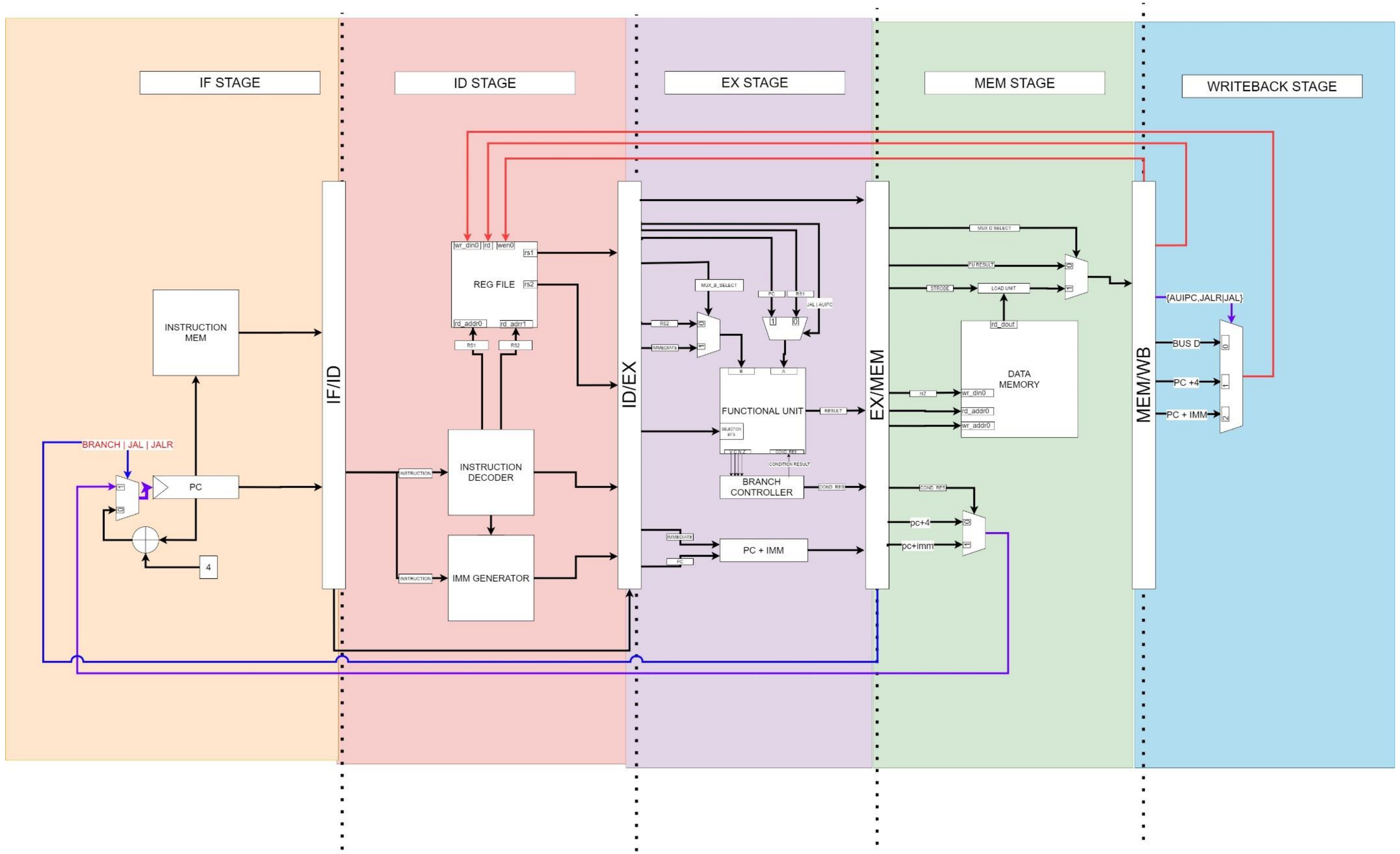


Figure 8. Final Schematic of question 1 pipelined RISC-V

Behavioral Simulation

We tested the assembly code we wrote for the previous homework and it did not worked properly because of hazards of the pipeline.

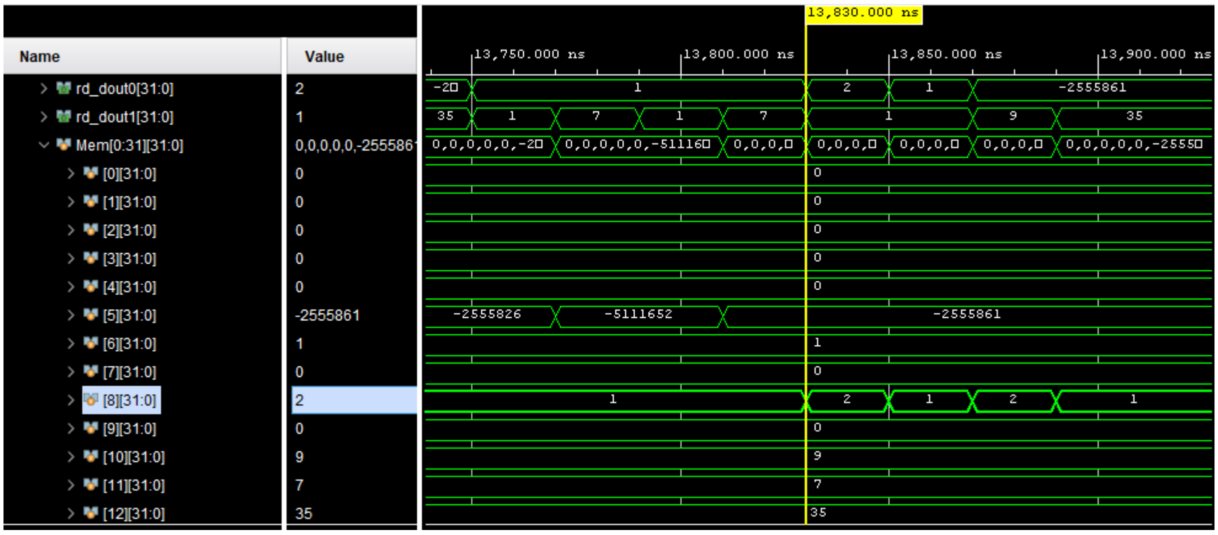
```
addi x10, x0, 9 #input a
addi x11, x0, 7 #input b
addi x12, x0, 35 #input n
addi x5,  x0, 0 #c = 0
addi x6,  x0, 31 #i = 31
addi x8,  x0, 1
sub  x7,  x0, x8
addi x21, x0, 1 #1
loop:
add x5, x5,  x5
blt x5, x12, cn #if c < n jump cn
sub x5, x5,  x12 #c = c - n
cn:
sll x8, x8,  x6
and x8, x8,  x11
srl x8, x8,  x6
bne x8, x21, ne
add x5, x5,  x10
ne:
blt x5, x12, cn2
sub x5, x5,  x12
cn2:
addi x8, x0, 1
sub  x6, x6, x21
bne  x6, x7, loop
```

We connected data memory and instruction memory in testbench and read the instructions with readmemb function.

```
`timescale 1ns / 1ps
module TOP_tb;

reg             clk = 0;
reg             rst = 0;
wire    [31:0]  Instruction;
wire    [31:0]  mem_data_in;
wire    [31:0]  mem_data_out;
wire    [31:0]  mem_address_out;
wire    [31:0]  PC;
wire            mem_we;
TOP TOP_i
(
    .clk                    ( clk                   ),
    .rst                    ( rst                   ),
    .Instruction            ( Instruction           ),
    .mem_data_in            ( mem_data_in           ),
    .mem_data_out           ( mem_data_out          ),
    .mem_address_out        ( mem_address_out       ),
    .PC                     ( PC                    ),
    .mem_we                 ( mem_we                )
);

data_memory     DMEM
(
    .clk                ( clk                   ),
    .rst                ( rst                   ),
    .rd_addr0           ( mem_address_out       ),
    .wr_addr0           ( mem_address_out       ),
    .wr_din0            ( mme_data_out          ),
    .we0                ( mem_we                ),
    .write_strobe       ( 2'b10                 ),
    .rd_dout0           ( mem_data_in           )
);
```

```
instruction_memory #
(
.WIDTH(32),
.DEPTH(128)
) ins_mem_i
(
    .clk        ( clk               ),
    .rst        ( rst               ),
    .rd_addr0   ( PC                ),
    .wr_addr0   (                   ),
    .wr_din0    (                   ),
    .we0        ( 1'b0              ),
    .rd_dout0   ( Instruction       )

);

always begin
    #10;
    clk = ~clk;
end

initial begin
    rst = 0;
    #25;
    rst = 1;
end

endmodule
```

Our assembly code for modular multiplication can be seen above. When we try to run this code in our pipelined processor, we got the following output:



Name	Value
rd_dout0[31:0]	2
rd_dout1[31:0]	1
Mem[0:31][31:0]	0,0,0,0,0,-2555861
[0][31:0]	0
[1][31:0]	0
[2][31:0]	0
[3][31:0]	0
[4][31:0]	0
[5][31:0]	-2555861
[6][31:0]	1
[7][31:0]	0
[8][31:0]	2
[9][31:0]	0
[10][31:0]	9
[11][31:0]	7
[12][31:0]	35

Figure 9: Simulation output of pipelined processor.

Simulation result shows the register file and we expected i in our assembly code (register 6) to be -1 when calculation is done. But when i = 1 result is calculated as -2555861 which is incorrect.

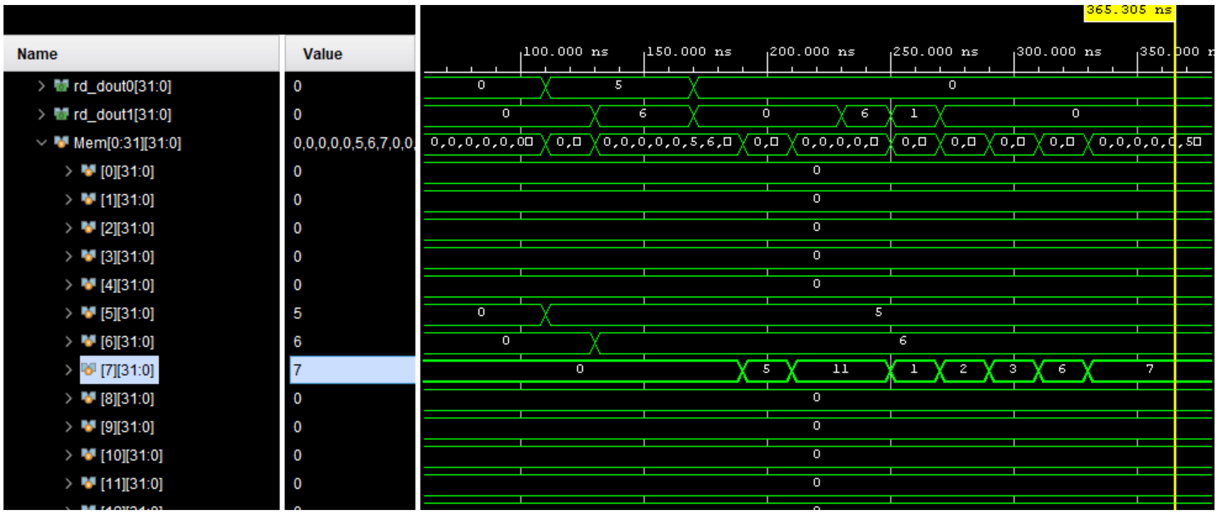
That mistake is caused by branch and register instructions.

- For register instructions: we need the values in the registers in the EX stage and if there are instructions that write to same registers in the MEM and WB stage, that means we use wrong values coming from registers (we use the old values).
- For branch instructions: we decide if branch will be taken or not in the EX stage, when branch instruction is moving in the pipeline instructions after the branch instruction is fetched too. When we decide to take the branch or not in EX stage, two instructions will be executed no matter branch is taken or not.

We will solve these hazards in the next parts. For understanding the mistakes we can use the assembly code below.

```
addi x5, x0, 5
addi x6, x0, 6
add x7, x5, x6
add x7, x5, x6
add x7, x5, x6
add x7, x5, x6
bne x5, x6, jump
addi x7, x0, 1
addi x7, x0, 2
addi x7, x0, 3
addi x7, x0, 4
addi x7, x0, 5
jump:
addi x7, x0, 6
addi x7, x0, 7
```

- This code should load x5 with the value 5 and x6 with the value 6. Then add them to find 11 and write it to register 7. Then it jumps the jump label (branch should be taken)
- Ideally, after the branch we should see register 7 becomes 6 and 7 after each other.



```
Name                Value
rd_dout0[31:0]      0
rd_dout1[31:0]      0
Mem[0:31][31:0]     0,0,0,0,0,5,6,7,0,0,
  [0][31:0]         0
  [1][31:0]         0
  [2][31:0]         0
  [3][31:0]         0
  [4][31:0]         0
  [5][31:0]         5
  [6][31:0]         6
  [7][31:0]         7
  [8][31:0]         0
  [9][31:0]         0
  [10][31:0]        0
  [11][31:0]        0
```

Figure 10: Pipeline hazard simulation output.

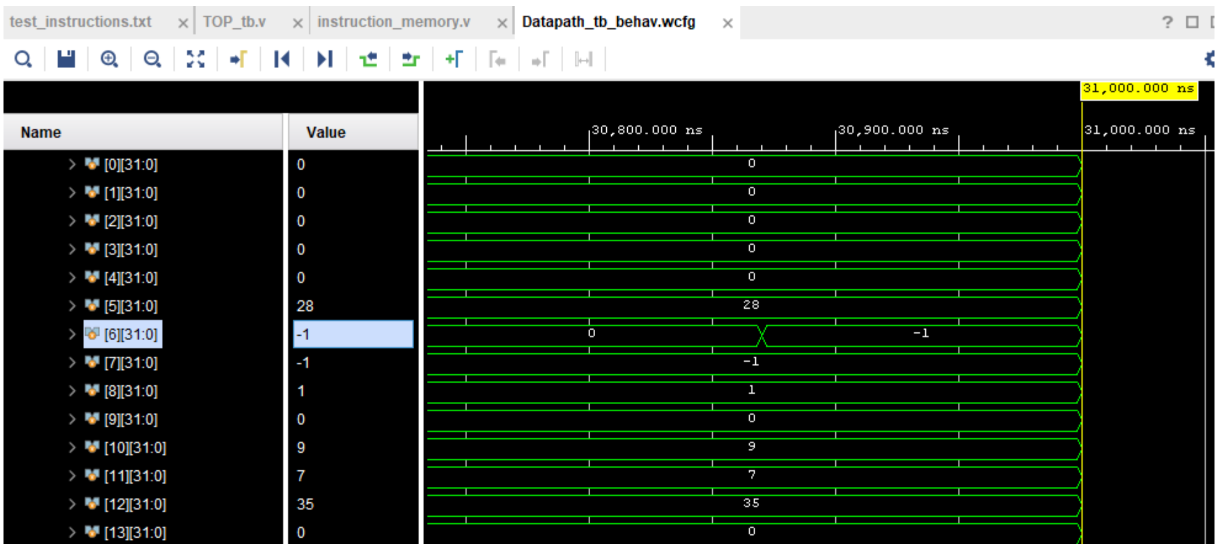
There were three add x7, x6, x5 instructions and as can be seen from the waveform the results are incorrect.

- First add instruction uses first value of the x5 = 0 and x6 = 0 and calculates the x6+x5 as 0.
- In the second add instruction same case is observed.
- In the third add instruction x5 is loaded and used as 5 but x6 is still in the WB stage so result is calculated as 5.
- In the fourth add instruction x6 and x5 is loaded with correct values and result is calculated as 11 which is correct.
- For the branch instruction case, we expected to only value 6 and 7 to be written into x7 but 1,2,3 is also written. That is because branch instruction is decided in EX stage and three instruction after the branch instruction will be executed no matter branch is taken or not.

For solving these problems in software we can add 3 NOP instruction after every branch and 5 NOP instruction after every dependent instruction. Dependent means destination register of the instruction is equal to source registers of the next instruction.

```
addi x10, x0, 9 #input a
addi x11, x0, 7 #input b
addi x12, x0, 35 #input n
addi x5,  x0, 0 #c = 0
addi x6,  x0, 31 #i = 31
addi x8,  x0, 1
#-----STALL-----
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
#---------------
sub  x7,  x0, x8
addi x21, x0, 1 #1
loop:
add x5, x5,  x5
#-----STALL-----
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
#---------------
blt x5, x12, cn #if c < n jump cn
#-----STALL-----
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
#---------------
sub x5, x5,  x12 #c = c - n
cn:
sll x8, x8,  x6
#-----STALL-----
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
#---------------
and x8, x8,  x11
#-----STALL-----
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
#---------------
srl x8, x8,  x6
#-----STALL-----
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
#---------------
bne x8, x21, ne
#-----STALL-----
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
#---------------
add x5, x5,  x10
#-----STALL-----
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
#---------------
#blt t0, a2, cn2
#sub t0, t0, a2
#cn2:
ne:
blt x5, x12, cn2
#-----STALL-----
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
#---------------
sub x5, x5,  x12
cn2:
addi x8, x0, 1
sub  x6, x6, x21
#-----STALL-----
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
#---------------
bne  x6, x7, loop
```

This code calculates mod35(7*9)=28.



Name	Value
[0][31:0]	0
[1][31:0]	0
[2][31:0]	0
[3][31:0]	0
[4][31:0]	0
[5][31:0]	28
[6][31:0]	-1
[7][31:0]	-1
[8][31:0]	1
[9][31:0]	0
[10][31:0]	9
[11][31:0]	7
[12][31:0]	35
[13][31:0]	0

Figure 11: Simulation result after nop instructions.

As we can see from the Figure 11, after adding the NOP instructions, result calculated correctly as 28 in x5.

2) Structural Hazards

In order to our system has seperated instruction memory and data memory, it has a structural hazard case which is when register file read address is same as write address. This is a hazard because one of the previous instruction should be writed this address in register file but it is not written yet. This hazard can be seen in Figure 12.

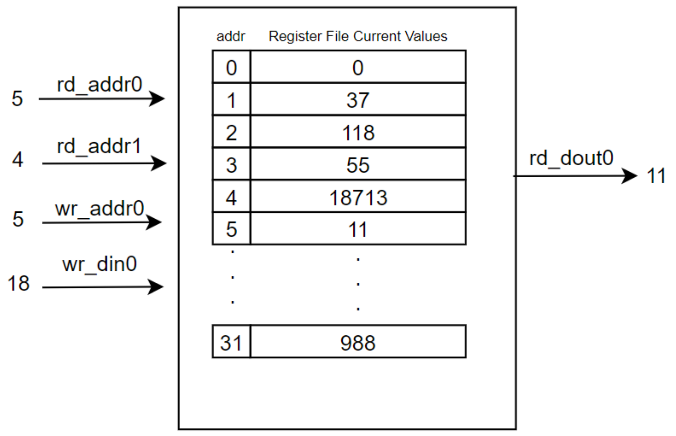


Figure 12: Structural Hazard Case

We solved that hazard by bypassing the wr_din0 and rd_dout0 / rd_dout1 and the condition is equality of rd_addr0 / rd_addr1 and wr_din0. Figure 13 shows the solution of this hazard.

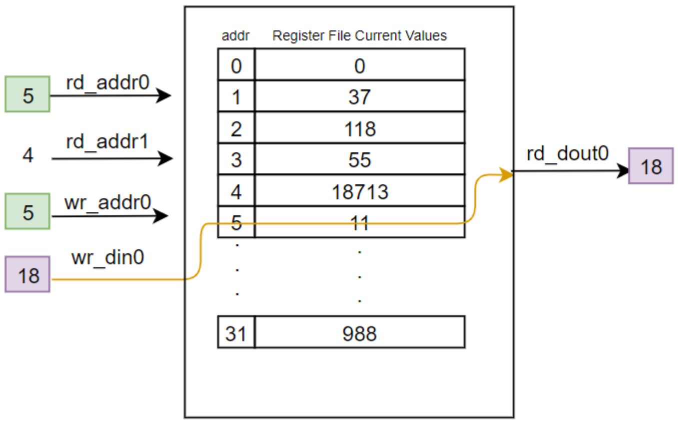


Figure 13: Structural Hazard Case Solution

Changes in the verilog code painted gray and can be seen below.

Fixed register file verilog code

```verilog
`timescale 1ns / 1ps


module register_file#(parameter WIDTH=32, parameter DEPTH=32)(

input                                   clk,
input                                   rst,
input   [$clog2(DEPTH)-1:0]             rd_addr0,
input   [$clog2(DEPTH)-1:0]             rd_addr1,
input   [$clog2(DEPTH)-1:0]             wr_addr0,
input   [WIDTH-1:0]                     wr_din0,
input                                   we_0,
output  [WIDTH-1:0]                     rd_dout0,
output  [WIDTH-1:0]                     rd_dout1

);

(* dont_touch = "true" *) reg [WIDTH-1:0] Mem [0:DEPTH-1];

assign rd_dout0 = (rd_addr0 == wr_addr0 & rd_addr0 != 5'd0 & we_0) ?
wr_din0 : Mem[rd_addr0];
assign rd_dout1 = (rd_addr1 == wr_addr0 & rd_addr1 != 5'd0 & we_0) ?
wr_din0 : Mem[rd_addr1];
integer i;

always @(posedge clk or negedge rst)
    begin
        if(!rst)
            begin
                for(i=0;i<DEPTH;i=i+1)  Mem[i] <= 'b0;
            end
        else
            begin
                if(we_0 & (wr_addr0 != 'd0))
                    begin
                            Mem[wr_addr0]      <=      wr_din0;
                    end
            end
    end
endmodule
```

Old register file code

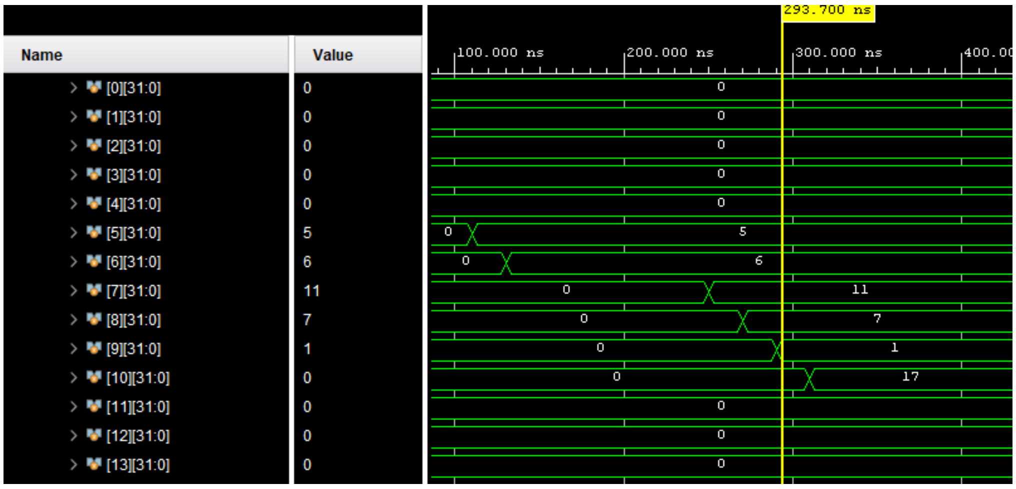
```verilog
`timescale 1ns / 1ps


module register_file#(parameter WIDTH=32, parameter DEPTH=32)(

input                                   clk,
input                                   rst,
input   [$clog2(DEPTH)-1:0]             rd_addr0,
input   [$clog2(DEPTH)-1:0]             rd_addr1,
input   [$clog2(DEPTH)-1:0]             wr_addr0,
input   [WIDTH-1:0]                     wr_din0,
input                                   we_0,
output  [WIDTH-1:0]                     rd_dout0,
output  [WIDTH-1:0]                     rd_dout1

);

(* dont_touch = "true" *) reg [WIDTH-1:0] Mem [0:DEPTH-1];

assign rd_dout0 = Mem[rd_addr0];
assign rd_dout1 = Mem[rd_addr1];
integer i;

always @(posedge clk or negedge rst)
    begin
        if(!rst)
            begin
                for(i=0;i<DEPTH;i=i+1)  Mem[i] <= 'b0;
            end
        else
            begin
                if(we_0 & (wr_addr0 != 'd0))
                    begin
                            Mem[wr_addr0]      <=      wr_din0;
                    end
            end
    end
```

Behavioral Simulation

After solving the structural hazards we used the code in the slides:

```asm
addi x5, x0, 5
addi x6, x0, 6
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
addi x0, x0, 0
add x7, x5, x6 #instruction X
or x8, x5, x6
slt x9, x5, x6
add x10, x7, x6 #instruction Y
```

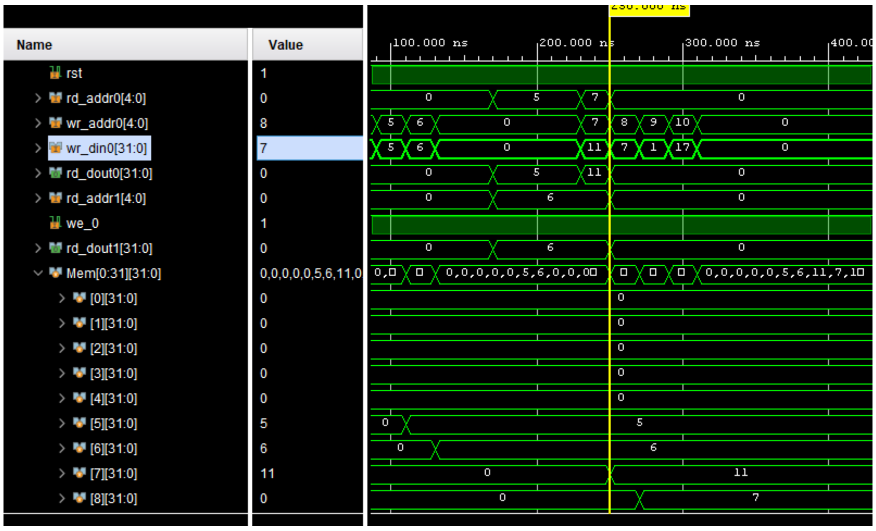
In this code, instruction X is writing the register file and at the same time instruction Y is reading that data. We could solve this by half cycle write before read but we chose to solve it by bypassing. We should see the value 17 in the register x10 at the end of the simulation.



Name	Value
[0][31:0]	0
[1][31:0]	0
[2][31:0]	0
[3][31:0]	0
[4][31:0]	0
[5][31:0]	5
[6][31:0]	6
[7][31:0]	11
[8][31:0]	7
[9][31:0]	1
[10][31:0]	0
[11][31:0]	0
[12][31:0]	0
[13][31:0]	0

Figure 14: Simulation result after solving structural hazards.

As we can see from the Figure 14, result is loaded into x10 as 17 which is correct.



Name	Value
rst	1
rd_addr0[4:0]	0
wr_addr0[4:0]	8
wr_din0[31:0]	7
rd_dout0[31:0]	0
rd_addr1[4:0]	0
we_0	1
rd_dout1[31:0]	0
Mem[0:31][31:0]	0,0,0,0,0,5,6,11,0
[0][31:0]	0
[1][31:0]	0
[2][31:0]	0
[3][31:0]	0
[4][31:0]	0
[5][31:0]	5
[6][31:0]	6
[7][31:0]	11
[8][31:0]	0

Figure 15: Register file bypassing.

In the Figure 15, at 250 ns (cursor), rd_addr0 = 7 (address that will be read) and wr_addr0 = 7 are equal and even in the register 7 (x7) has the value 0 at that time register file output rd_dout0 is equal to 11 which is wr_din0. That means register file gives the data to be written to the data out when we are reading from the same register when writing to that register.

3) Data Hazards – Read After Write (RAW) Hazards:

Topic of this subject is read after write hazards. This hazard happens because when an instruction is read after write, before write is completed read operation occurs from register file. There are 2 stage type of RAW hazards. First one is EX-ID stage hazard and second one is MEM-ID stage hazard. An example of EX-ID stage hazard can seen in Figure 16.

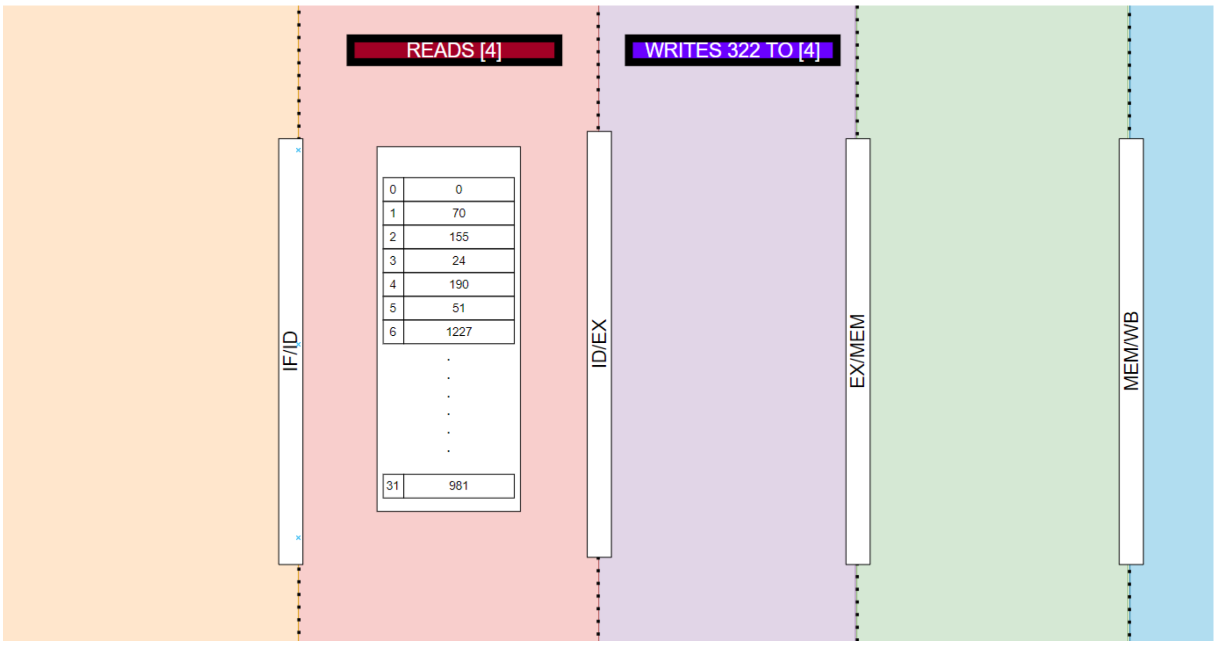


Figure 16. ID - EX Read after Write hazard type

In this case, purple instruction which is in EX stage should write decimal 322 which is going to calculated in functional unit to Reg[4]. After this instruction an instruction comes and wants to read Reg[4]. Single cycle core would write 322 to Reg[4] first and then reads 322 from Reg[4]. But in pipelined core, write operation has not been done and read Reg[4] operation will read 190 in this case.

To make figures clear, next case of this hazard is in the next page.

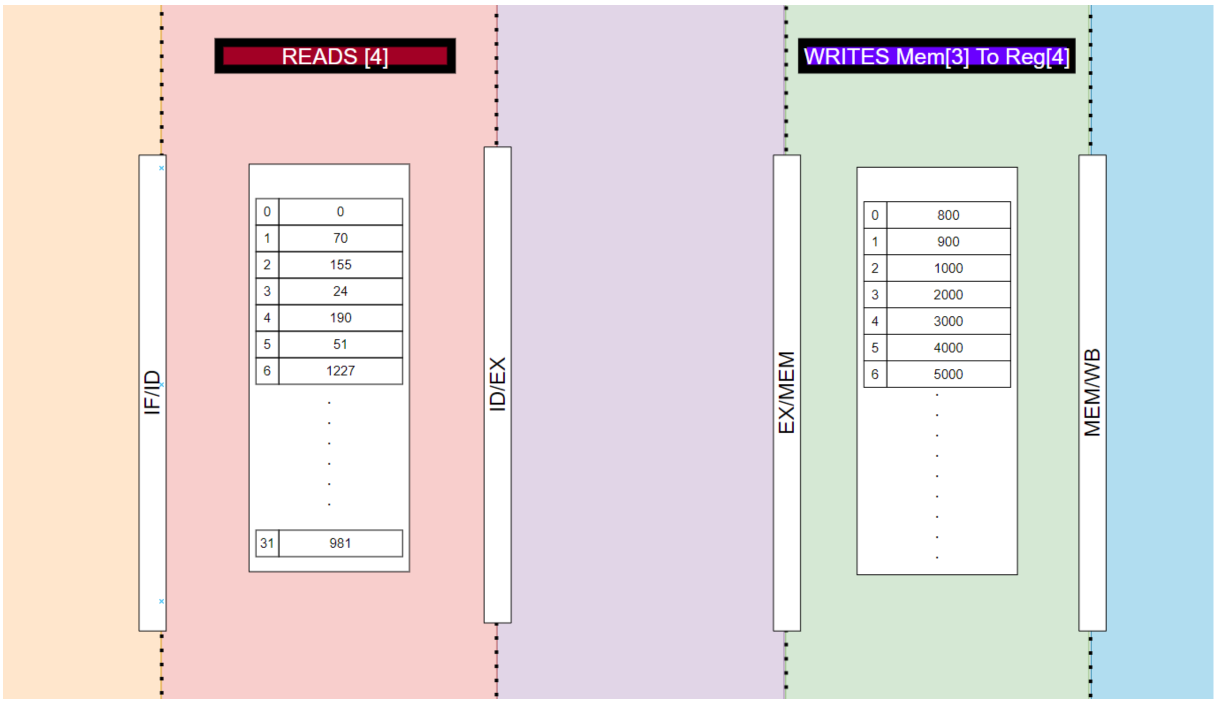


Figure 17. ID - MEM Read After Write Hazard

In that kind of case single cycle core will writes Mem[3] value which is decimal 2000 to Reg[4] in one cycle, makes the operation which is between read and write in other cycle, then reads Reg[4] as 2000.

But in pipelined design, Mem[3] value still has not written to Reg[4]. Because of that, read operation will read old Reg[4] value which is 190. That is the second type.

To solve this, we will use forwarding unit. This forwarding unit will get Reg_RW signal from control word in EX/MEM register and MEM/WB register and get rd addresses from both of these registers. This forwarding unit also gets RS2 address and RS1 address from ID/EX register to compare these values with RD values which are gotten from EX/MEM and MEM/WB. If there is an equality between these values, selection changes in multiplexers which is going to shown figures below.

Selection signal of forwarding unit is 4 bit wide. 2 bit is for equality in rs1 and previous rd, 2 bit is for equality in rs2 and previous rd. Forward A signal is for rs1 equality and forward B signal is rs2 equality.

When forward A signal is 2’b00, that means there is no need to forward any data.
When forward A signal is 2’b01, that means FU result in EX/MEM register should be forwarded to A input of FU in EX stage.
When forward A signal is 2’b10, that means BUS_D in MEM/WB register should be forwarded to A input of FU in EX stage.

For B input;
When forward B signal is 2’b00 that means there is no need to forward any data.
When forward B signal is 2’b01 that means FU result in EX/MEM register should be forwarded to B input of FU in EX stage.
When forward B signal is 2’b10 that means BUS_D in MEM/WB register should be forwarded to B input of FU in EX stage.

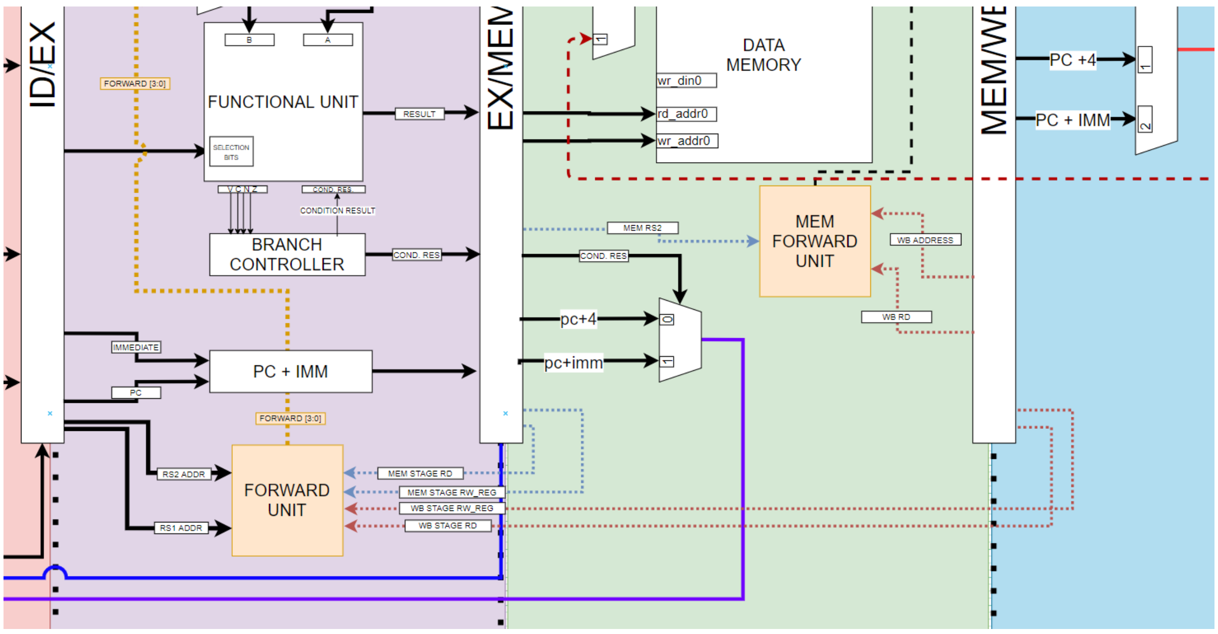


Figure 18. Forward unit inputs and output selection signal.

To make it clear, selection of the forward unit is 2 bits and painted orange and connected with dotted wires.

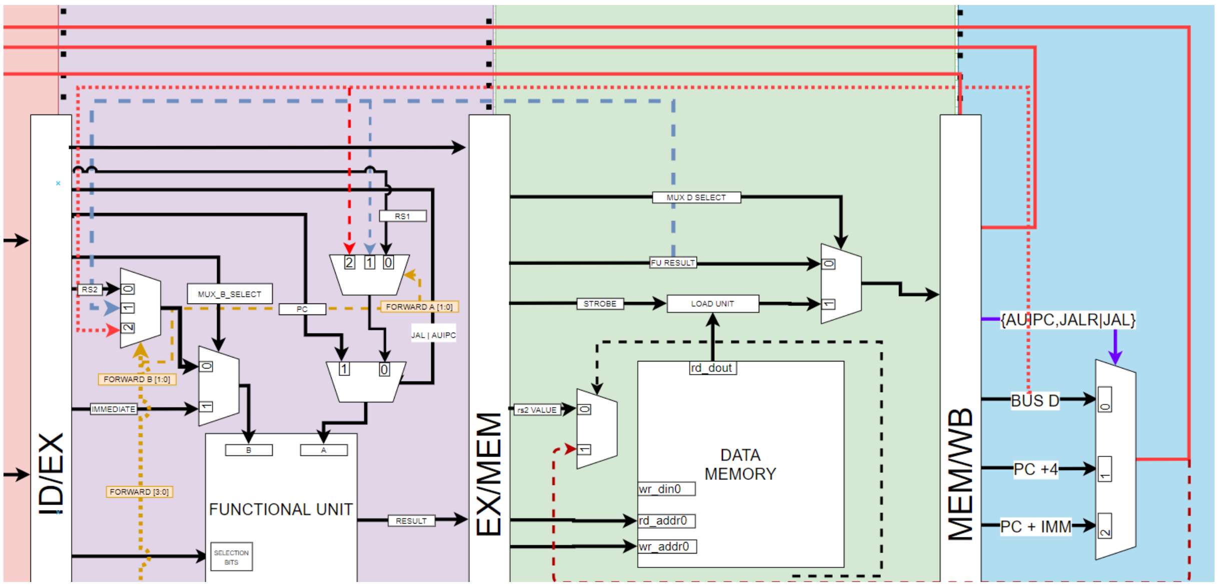


Figure 19. Forward unit selection bits with necessary multiplexers.

And there is another RAW hazard which occurs in memory to memory operations. Lets think about an instruction series as load a value from memory to an address to register file and then store this value in this address to data memory. In this case, single cycle core will make the write operation to register file, after that it will read the same address and will read the true value.But in pipelined system, that will cause an data hazard because it didn’t loaded to register file when it would try to store the same address. In Figure 20 that problem can be seen and Figure 21 shows the solution.

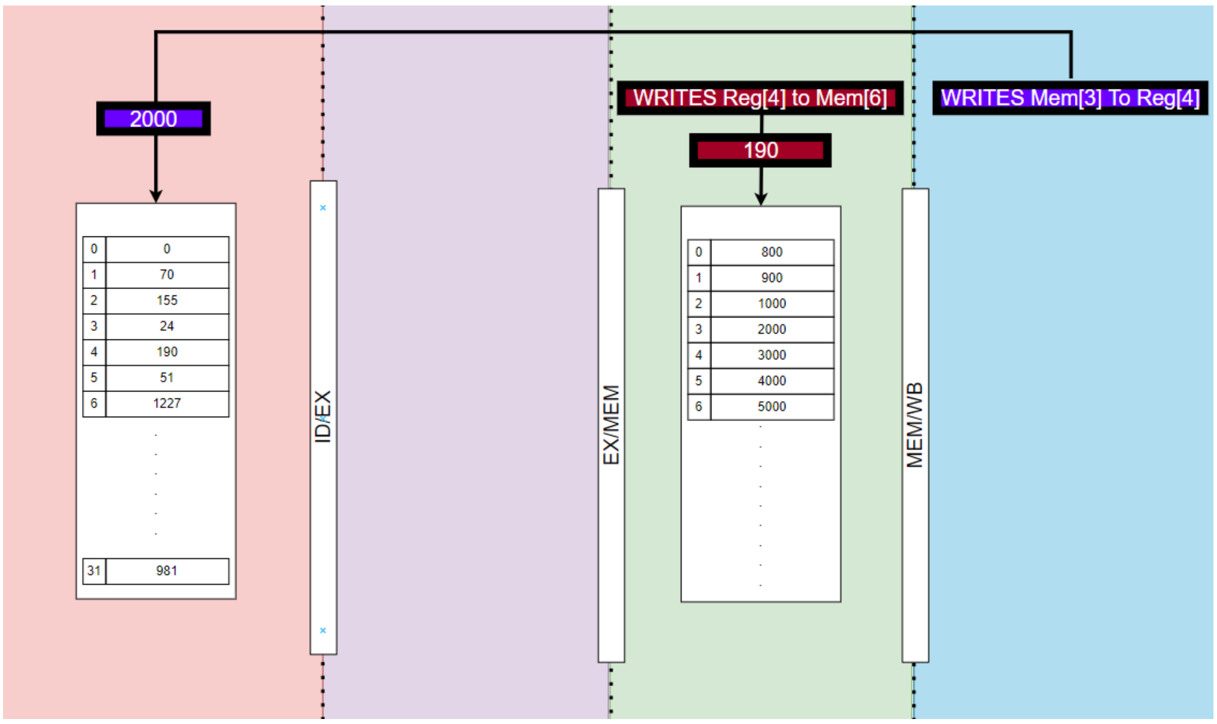


Figure 20. Memory to memory RAW hazard

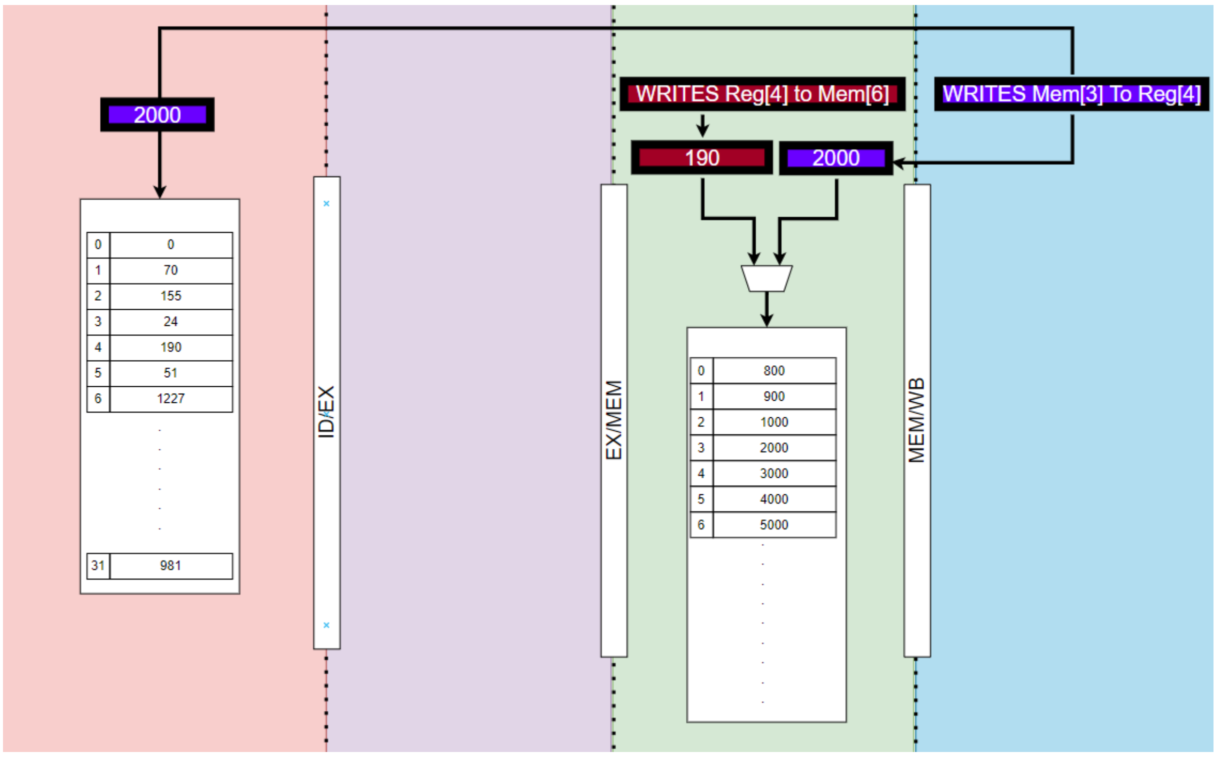


Figure 21. Memory to memory RAW hazard solution

By implementing this solution, we have another forwarding unit named Memory Forward Unit. This unit gets the Mem_RW signal and rd address from MEM/WB register and gets rs2 address from EX/MEM register. It compares rs2 address and rd address at the same time while looking is Mem_RW is 1. Then it generates a control signal which controls the input wr_din0 of data memory. If this signal is 1, this unit forwards the main line of WB stage to wr_din0, if it is 0 wr_din0 goes normal as rs2 value of EX/MEM register. Figure 22 shows that unit and connections.

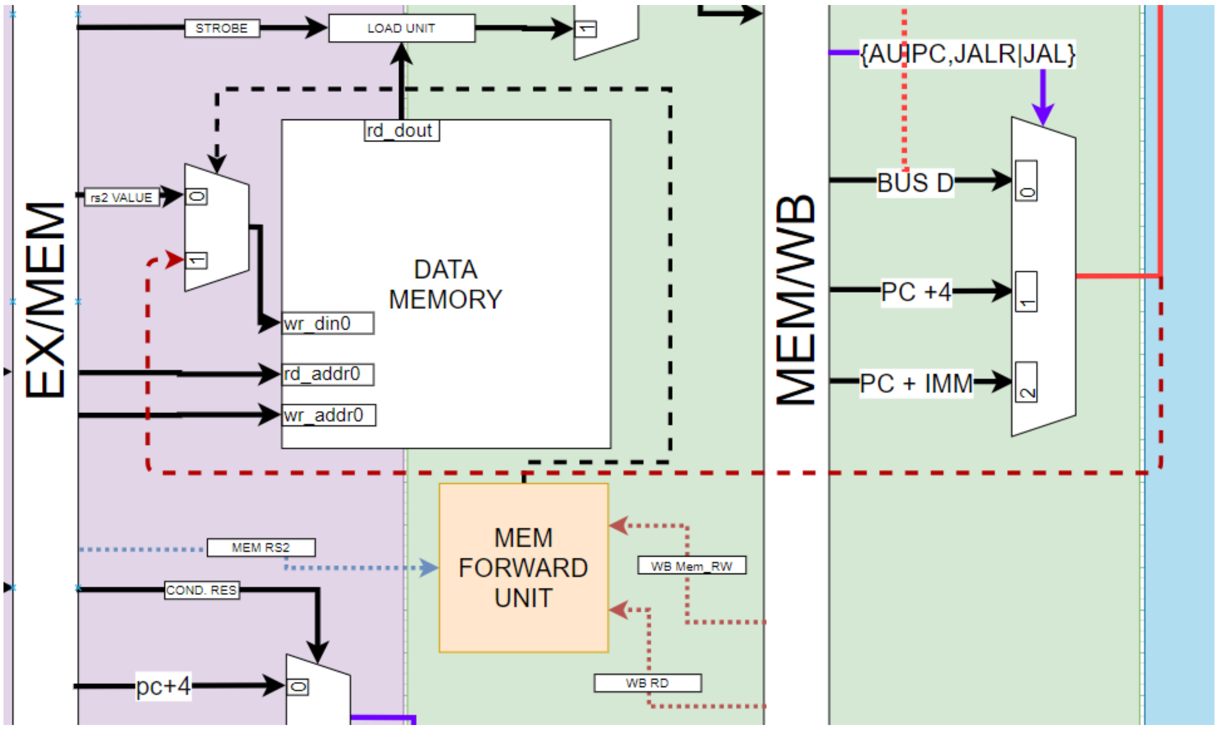


Figure 22: Memory Forward Unit Connections

We added 2 forwarding unit and their codes are below

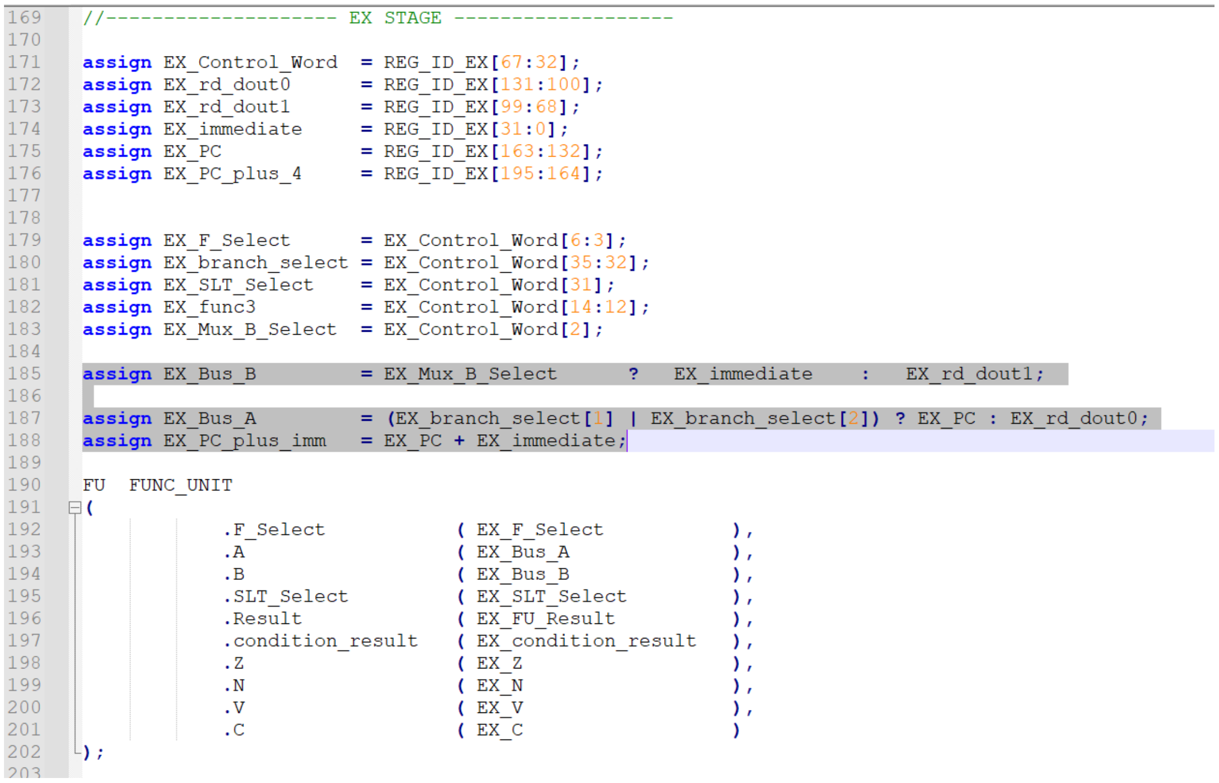
Memory Forwarding Unit

```
module mem_forward_unit(
    input        [4:0] MEM_RS2,
    input        [4:0] WB_RD,
    input              MEM_Mem_RW,
    output reg         Forward_MEM
    );


always @(*) begin

    if(MEM_Mem_RW && (MEM_RS2 ==
WB_RD))
        Forward_MEM = 1'b1;
    else
        Forward_MEM = 1'b0;
end

endmodule
```

Forwarding Unit

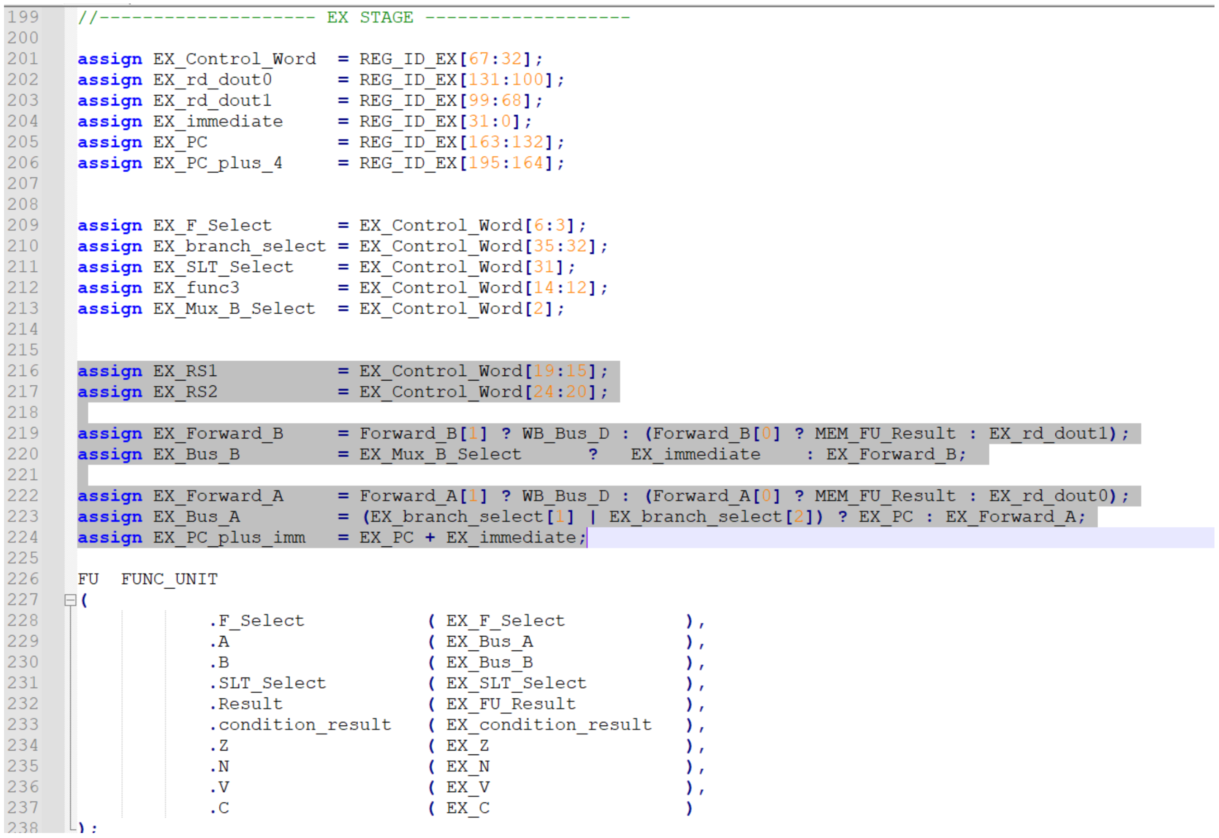
```
module forward_unit(
    input [4:0] MEM_RD,
    input [4:0] WB_RD,
    input [4:0] EX_RS1,
    input [4:0] EX_RS2,
    input MEM_load_enable,
    input WB_load_enable,
    output reg [1:0] Forward_A, Forward_B
    );

always @(*) begin

    if(MEM_load_enable && (MEM_RD != 0) && (MEM_RD ==
EX_RS1))
        Forward_A = 2'b01;
    else if(WB_load_enable && (WB_RD != 0) && (WB_RD ==
EX_RS1))
        Forward_A = 2'b10;
    else
        Forward_A = 2'b00;

    if(MEM_load_enable && (MEM_RD != 0) && (MEM_RD ==
EX_RS2))
        Forward_B = 2'b01;
    else if(WB_load_enable && (WB_RD != 0) && (WB_RD ==
EX_RS2))
        Forward_B = 2'b10;
    else
        Forward_B = 2'b00;
end

endmodule
```

For MEM to EX and WB to EX forwarding Figure 23 shows the first code which is not fixed for RAW hazards and Figure 24 shows the fixed code.



```verilog
//-------------------- EX STAGE -------------------

assign EX_Control_Word  = REG_ID_EX[67:32];
assign EX_rd_dout0      = REG_ID_EX[131:100];
assign EX_rd_dout1      = REG_ID_EX[99:68];
assign EX_immediate     = REG_ID_EX[31:0];
assign EX_PC            = REG_ID_EX[163:132];
assign EX_PC_plus_4     = REG_ID_EX[195:164];


assign EX_F_Select      = EX_Control_Word[6:3];
assign EX_branch_select = EX_Control_Word[35:32];
assign EX_SLT_Select    = EX_Control_Word[31];
assign EX_func3         = EX_Control_Word[14:12];
assign EX_Mux_B_Select  = EX_Control_Word[2];

assign EX_Bus_B         = EX_Mux_B_Select      ?   EX_immediate    :    EX_rd_dout1;

assign EX_Bus_A         = (EX_branch_select[1] | EX_branch_select[2]) ? EX_PC : EX_rd_dout0;
assign EX_PC_plus_imm   = EX_PC + EX_immediate;

FU  FUNC_UNIT
(
            .F_Select           ( EX_F_Select           ),
            .A                  ( EX_Bus_A              ),
            .B                  ( EX_Bus_B              ),
            .SLT_Select         ( EX_SLT_Select         ),
            .Result             ( EX_FU_Result          ),
            .condition_result   ( EX_condition_result   ),
            .Z                  ( EX_Z                  ),
            .N                  ( EX_N                  ),
            .V                  ( EX_V                  ),
            .C                  ( EX_C                  )
);
```

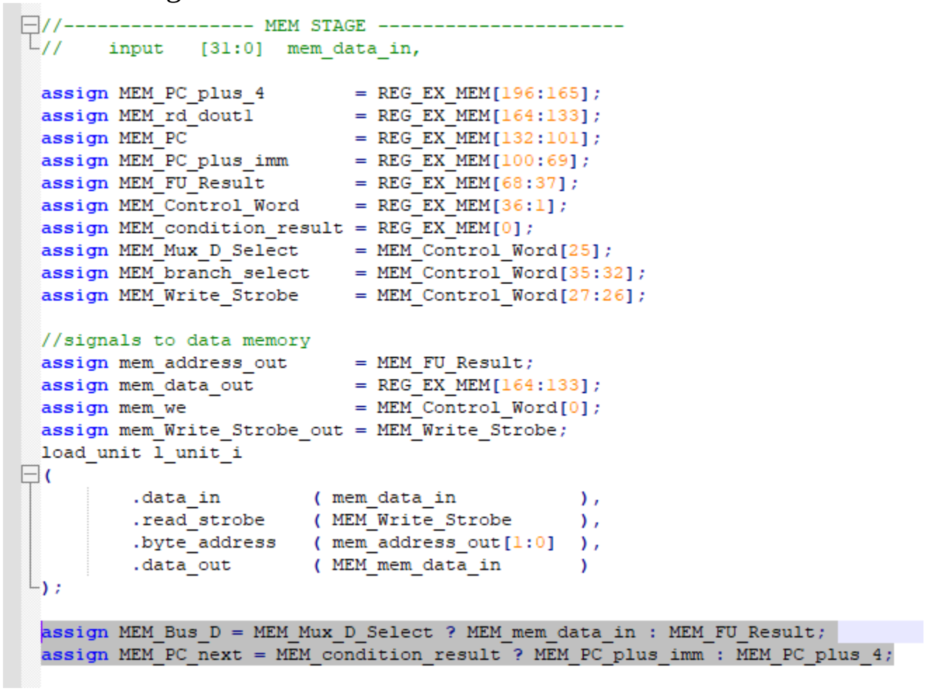
Figure 23: Without fixing MEM to EX and WB to EX RAW hazards



```verilog
//-------------------- EX STAGE -------------------

assign EX_Control_Word  = REG_ID_EX[67:32];
assign EX_rd_dout0      = REG_ID_EX[131:100];
assign EX_rd_dout1      = REG_ID_EX[99:68];
assign EX_immediate     = REG_ID_EX[31:0];
assign EX_PC            = REG_ID_EX[163:132];
assign EX_PC_plus_4     = REG_ID_EX[195:164];


assign EX_F_Select      = EX_Control_Word[6:3];
assign EX_branch_select = EX_Control_Word[35:32];
assign EX_SLT_Select    = EX_Control_Word[31];
assign EX_func3         = EX_Control_Word[14:12];
assign EX_Mux_B_Select  = EX_Control_Word[2];


assign EX_RS1           = EX_Control_Word[19:15];
assign EX_RS2           = EX_Control_Word[24:20];

assign EX_Forward_B     = Forward_B[1] ? WB_Bus_D : (Forward_B[0] ? MEM_FU_Result : EX_rd_dout1);
assign EX_Bus_B         = EX_Mux_B_Select      ?   EX_immediate    : EX_Forward_B;

assign EX_Forward_A     = Forward_A[1] ? WB_Bus_D : (Forward_A[0] ? MEM_FU_Result : EX_rd_dout0);
assign EX_Bus_A         = (EX_branch_select[1] | EX_branch_select[2]) ? EX_PC : EX_Forward_A;
assign EX_PC_plus_imm   = EX_PC + EX_immediate;

FU  FUNC_UNIT
(
            .F_Select           ( EX_F_Select           ),
            .A                  ( EX_Bus_A              ),
            .B                  ( EX_Bus_B              ),
            .SLT_Select         ( EX_SLT_Select         ),
            .Result             ( EX_FU_Result          ),
            .condition_result   ( EX_condition_result   ),
            .Z                  ( EX_Z                  ),
            .N                  ( EX_N                  ),
            .V                  ( EX_V                  ),
            .C                  ( EX_C                  )
);
```

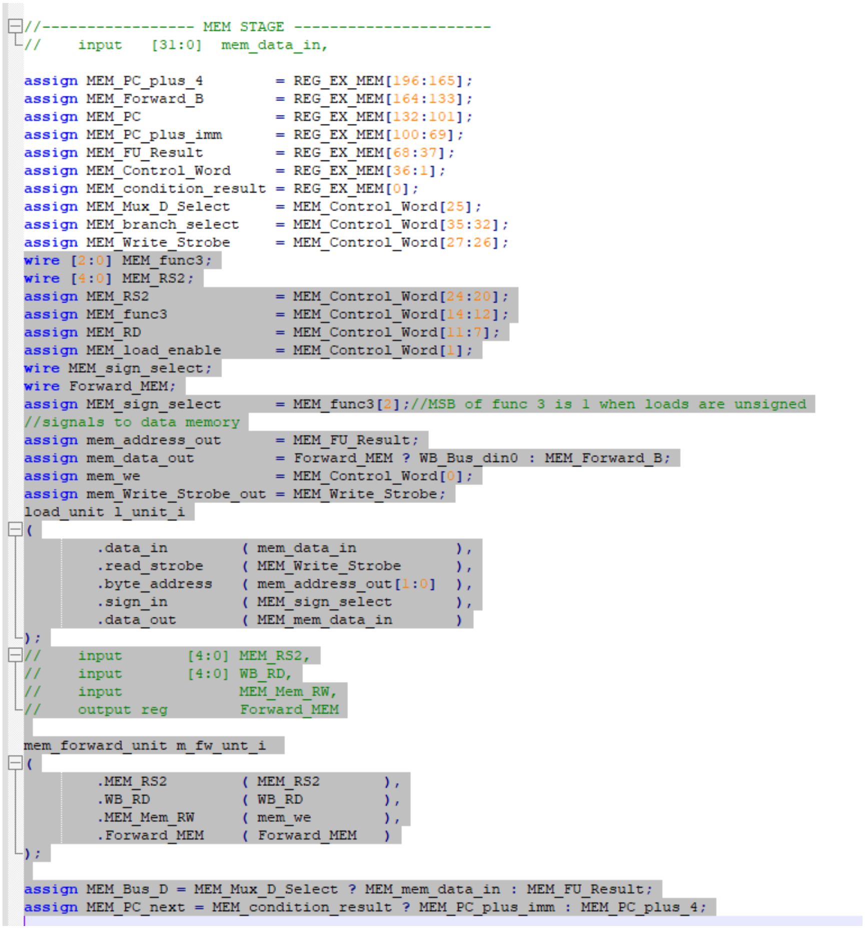
Figure 24: Fixing MEM to EX and WB to EX hazards with Forward Unit Signals

For memory to memory hazards which occurs between WB and MEM stages, figure 25 shows the previous code and figure 26 shows the fixed code.



```
//------------------ MEM STAGE ----------------------
//     input    [31:0]   mem_data_in,

assign MEM_PC_plus_4          = REG_EX_MEM[196:165];
assign MEM_rd_dout1           = REG_EX_MEM[164:133];
assign MEM_PC                 = REG_EX_MEM[132:101];
assign MEM_PC_plus_imm        = REG_EX_MEM[100:69];
assign MEM_FU_Result          = REG_EX_MEM[68:37];
assign MEM_Control_Word       = REG_EX_MEM[36:1];
assign MEM_condition_result = REG_EX_MEM[0];
assign MEM_Mux_D_Select       = MEM_Control_Word[25];
assign MEM_branch_select      = MEM_Control_Word[35:32];
assign MEM_Write_Strobe       = MEM_Control_Word[27:26];

//signals to data memory
assign mem_address_out        = MEM_FU_Result;
assign mem_data_out           = REG_EX_MEM[164:133];
assign mem_we                 = MEM_Control_Word[0];
assign mem_Write_Strobe_out = MEM_Write_Strobe;
load_unit l_unit_i
(
        .data_in          ( mem_data_in            ),
        .read_strobe      ( MEM_Write_Strobe       ),
        .byte_address     ( mem_address_out[1:0]   ),
        .data_out         ( MEM_mem_data_in        )
);

assign MEM_Bus_D = MEM_Mux_D_Select ? MEM_mem_data_in : MEM_FU_Result;
assign MEM_PC_next = MEM_condition_result ? MEM_PC_plus_imm : MEM_PC_plus_4;
```

Figure 25: Previous MEM stage code



```
//------------------ MEM STAGE ----------------------
//     input    [31:0]   mem_data_in,

assign MEM_PC_plus_4          = REG_EX_MEM[196:165];
assign MEM_Forward_B          = REG_EX_MEM[164:133];
assign MEM_PC                 = REG_EX_MEM[132:101];
assign MEM_PC_plus_imm        = REG_EX_MEM[100:69];
assign MEM_FU_Result          = REG_EX_MEM[68:37];
assign MEM_Control_Word       = REG_EX_MEM[36:1];
assign MEM_condition_result = REG_EX_MEM[0];
assign MEM_Mux_D_Select       = MEM_Control_Word[25];
assign MEM_branch_select      = MEM_Control_Word[35:32];
assign MEM_Write_Strobe       = MEM_Control_Word[27:26];
wire [2:0] MEM_func3;
wire [4:0] MEM_RS2;
assign MEM_RS2                = MEM_Control_Word[24:20];
assign MEM_func3              = MEM_Control_Word[14:12];
assign MEM_RD                 = MEM_Control_Word[11:7];
assign MEM_load_enable        = MEM_Control_Word[1];
wire MEM_sign_select;
wire Forward_MEM;
assign MEM_sign_select        = MEM_func3[2];//MSB of func 3 is 1 when loads are unsigned
//signals to data memory
assign mem_address_out        = MEM_FU_Result;
assign mem_data_out           = Forward_MEM ? WB_Bus_din0 : MEM_Forward_B;
assign mem_we                 = MEM_Control_Word[0];
assign mem_Write_Strobe_out = MEM_Write_Strobe;
load_unit l_unit_i
(
        .data_in          ( mem_data_in            ),
        .read_strobe      ( MEM_Write_Strobe       ),
        .byte_address     ( mem_address_out[1:0]   ),
        .sign_in          ( MEM_sign_select        ),
        .data_out         ( MEM_mem_data_in        )
);
//     input          [4:0] MEM_RS2,
//     input          [4:0] WB_RD,
//     input                MEM_Mem_RW,
//     output reg           Forward_MEM

mem_forward_unit m_fw_unt_i
(
        .MEM_RS2          ( MEM_RS2        ),
        .WB_RD            ( WB_RD          ),
        .MEM_Mem_RW       ( mem_we         ),
        .Forward_MEM      ( Forward_MEM    )
);

assign MEM_Bus_D = MEM_Mux_D_Select ? MEM_mem_data_in : MEM_FU_Result;
assign MEM_PC_next = MEM_condition_result ? MEM_PC_plus_imm : MEM_PC_plus_4;
```

Figure 26: Fixed MEM stage code

Behavioral Simulation

There are three cases for this hazard. First case is when dependency is between EX stage and MEM stage. For this case assembly code below will be simulated.

```
addi x5, x0, 5
addi x6, x0, 6
addi x7, x0, 7
addi x8, x0, 8
addi x9, x0, 9 #this instruction will be in MEM stage
add  x10, x9, x5 #when this instruction is in EX stage
```

In this code dependency is between EX and MEM stage and MEM.RD == EX.RS1 condition is true so forward unit should forward the data in the MEM stage to A input of the FU. Result should be 9 + 5 = 14 and stored into x10.

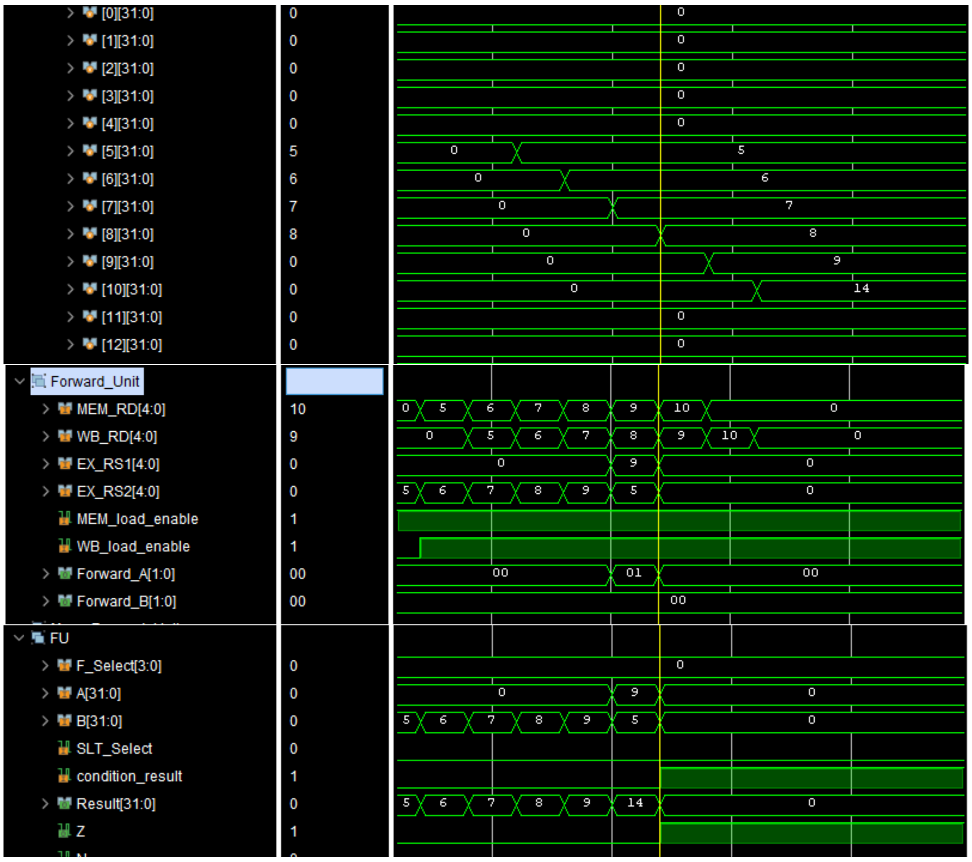


Figure 27: First case for the data hazards.

- As can be seen in the Figure 27, result is calculated and written into register file correctly (first image is register file).
- In Forward Unit we can see that, at the cursor, MEM_RD (destination write address of the instruction in the MEM stage) is 9 and EX_RS1 (source1 read address of the instruction in the EX stage) is 9, that means we should forward the data at MEM stage to the EX stage.
- For this forwarding operation, Forward_A = 01 and that signal is connected to the MUX at the input A of the FU.
- At that time (cursor time) x9 in the register file has the value 0 but in the A input of the FU we see the value 9 instead of value 0. That means forwarding is done correctly.

For the second case, dependency is between EX stage and WB stage.

```
addi x5, x0, 5
addi x6, x0, 6
addi x7, x0, 7
addi x8, x0, 8 #this instruction is at WB stage
addi x9, x0, 9
add x10, x8, x5 #when this instruction is in the EX stage
```

In this code dependency is between EX and WB stage and WB.RD == EX.RS1 condition is true so forward unit should forward the data in the WB stage to A input of the FU. Result should be 8 + 5 = 13 and stored into x10.

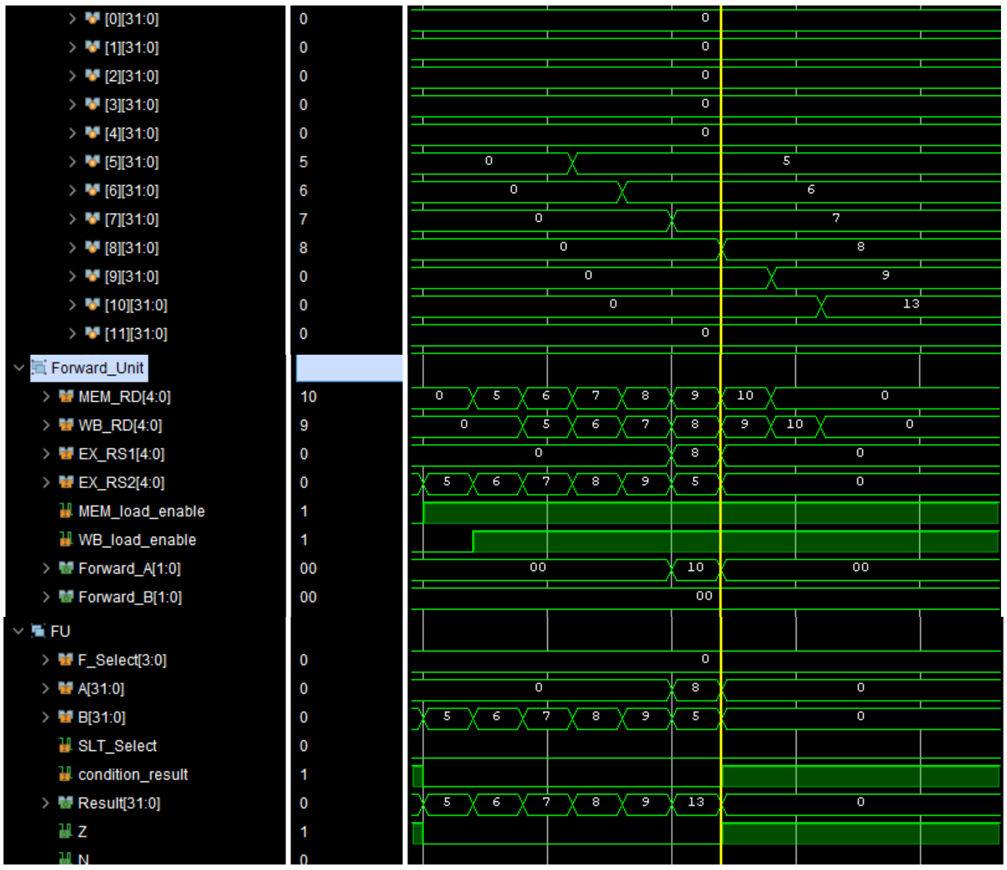


Figure 28: Second case for the data hazards.

- As can be seen in the Figure 28, result is calculated and written into register file correctly (first image is register file).
- In Forward Unit we can see that, at the cursor, WB_RD (destination write address of the instruction in the WB stage) is 8 and EX_RS1 (source1 read address of the instruction in the EX stage) is 8, that means we should forward the data at WB stage to the EX stage.
- For this forwarding operation, Forward_A = 10 and that signal is connected to the MUX at the input A of the FU. 10 means forward from WB.
- At that time (cursor time) x8 in the register file has the value 0 but in the A input of the FU we see the value 8 instead of value 0. That means forwarding is done correctly.

For the third case, dependency is between MEM stage and WB stage (memory to memory copy).

```
addi  x5,  x0,  5
sw    x5,  0(x0)
lw    x6,  0(x0)
sw    x6,  4(x0)
```

In this code dependency is between MEM and WB stage and WB.RD == EX.RS1 condition is true so forward unit should forward the data in the WB stage to wr_din0 input of the memory. That hazard happens when there is store instruction after the load or register write instruction and they use the same data. 5 should be stored in address 0 of memory and it should be loaded into x6, then x6 value (5) should be written to address 4 of memory.

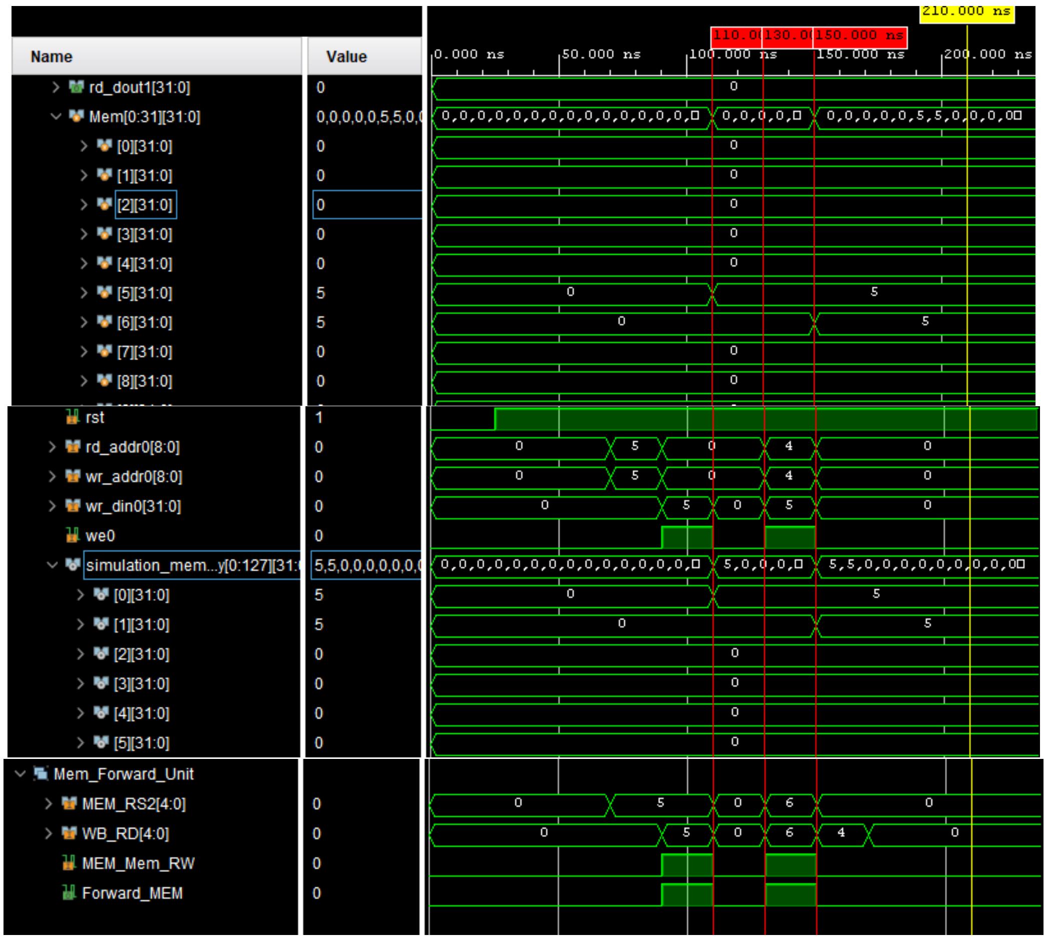


Figure 29: Third case for the data hazards.

- As can be seen in the Figure 29, result is calculated and written into register file and memory correctly. Here, first image is from register file, second image is from data memory and third image is from memory forward unit.
- In Forward Unit we can see that, at the cursor 1 (110ns), WB_RD (destination write address of the instruction in the WB stage) is 5 and MEM_RS2 (register whose contents will be written into memory) is 5, that means we should forward the data at WB stage to the input of the data memory at MEM stage. We can see that input of the data memory wr_din0 in the second image is 5 even though register 5 (register whose contents will be stored in memory) has the value 0. That means there is forwarding.

- For this forwarding operation, Forward_MEM = 1 and that signal is connected to the MUX at the input wr_din0 of the data memory. 1 means forward from WB.
- At the cursor 1 (110ns), value 5 written into address 0 of memory and addres 5 of register file. CPI = 1 in this example but memory write and register file write is done at the same time.
- At the cursor 3 (150ns), at the mem forward unit (bottom image in Figure 29), MEM_RS2 (register whose contents will be written into memory) is 6 and, WB_RD (destination write address of the instruction in the WB stage) is 6 so we should forward data at the WB stage to the data input of the memory. Forward_MEM = 1.
- At 150ns, even though register 6 has the value 0, value 5 is written into memory. That means store instruction after load or register instruction is working correctly. Load instruction after store instruction is also working correctly.

For the next simulation we used the cases above randomly and show that every RAW hazards is avoided with CPI = 1. RS1 and RS2 dependency is also included.

```
addi x5, x0, 5
sw   x5, 0(x0)
lw   x6, 0(x0)
sw   x6, 4(x0)
addi x6, x0, 3
add  x7, x6, x5
add  x8, x6, x7
add  x9, x8, x7
```

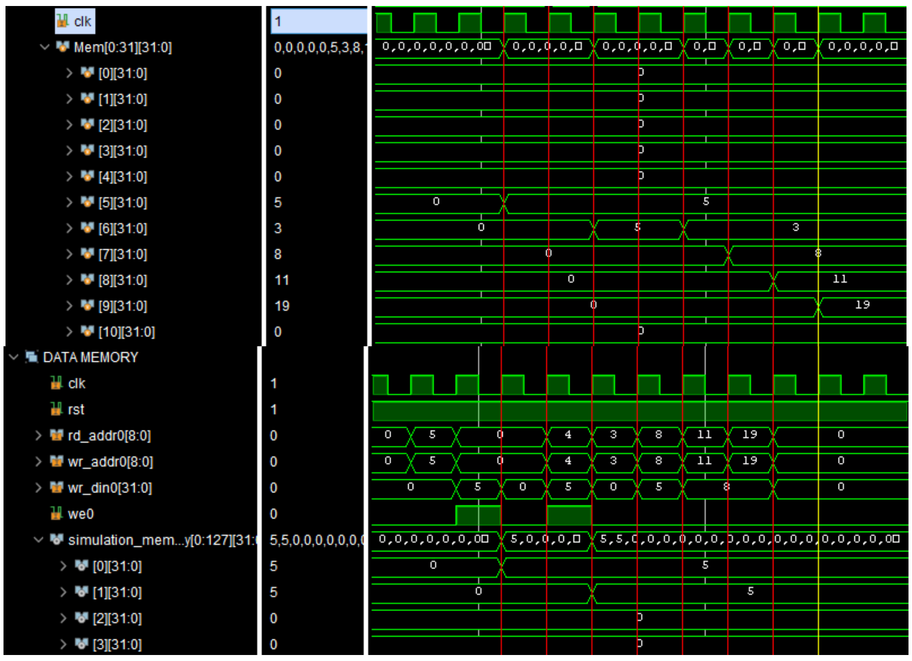


Figure 30: Register file and data memory waveform for RAW hazards.

- At marker 1, 5 value is stored in x5 and memory address 0.
- At marker 2, nothing happened because first two instructions are performed together at marker 1.
- At marker 3, value 5 stored in x6 and memory address 4.
- At marker 4, nothing happened because third and fourth instructions are performed together at marker 3.

- At marker 5, addi x6, x0, 3 is performed and 0 + 3 = 3 stored into x6.
- At marker 6, add x7, x6, x5 is performed and 3 + 5 = 8 stored into x7.
- At marker 7, add x8, x6, x7 is performed and 3 + 8 = 11 stored into x8.
- At marker 8, add x9, x8, x7 is performed and 11 + 8 = 19 stored into x9.

There are 8 instructions total and it takes 8 cycle to execute them so, CPI = 1. That means every load store or register instruction takes 1 cycle to complete.

4) Data Hazards – Load-Use Hazard:

In this case we need to solve load-use hazard. This hazard occurs because our forwarding units is not reaching to fix.

It occurs when an instruction comes which loads a value K to address X and after that instruction comes which uses value of address K. This value is a part of FU result which will be calculated. Figures below shows the problem and solution that we don't use.

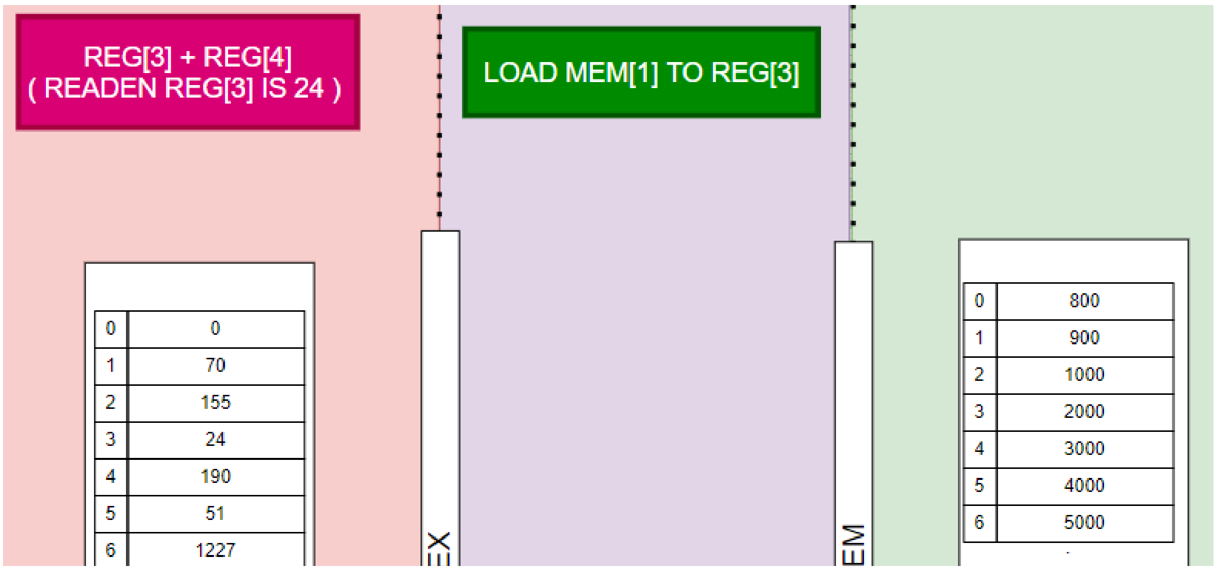


Figure 31: Use after a load instruction

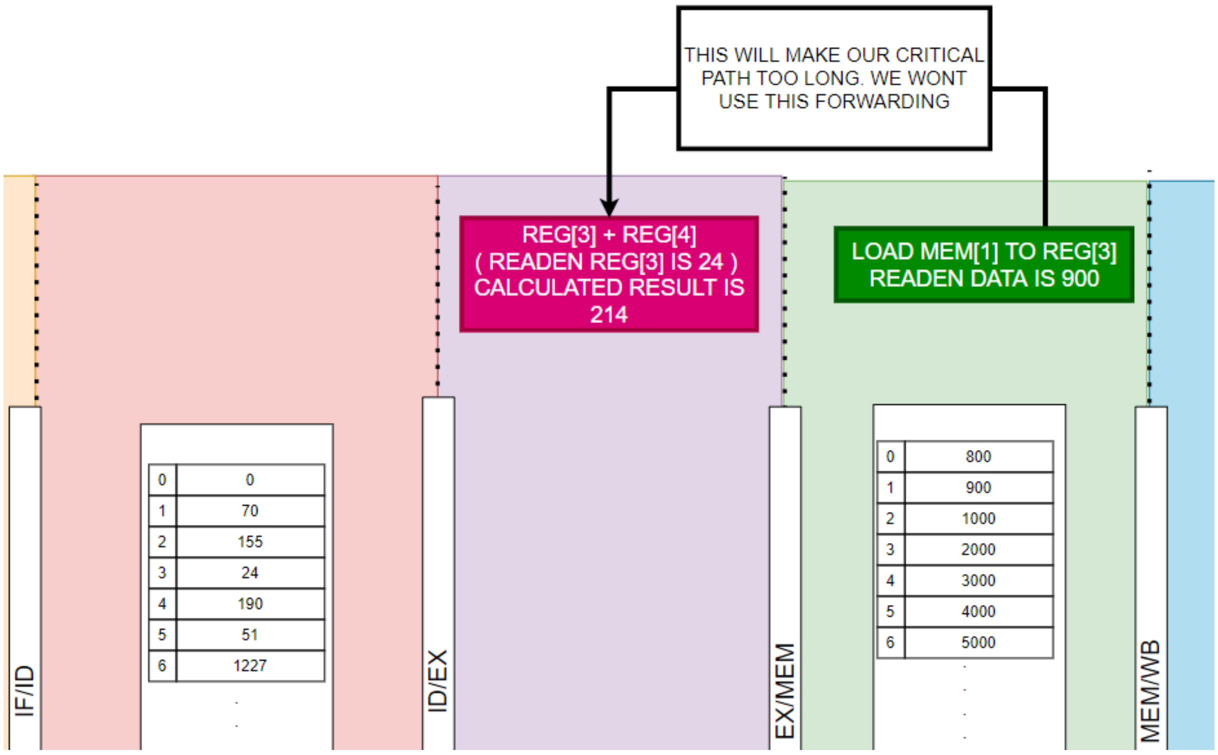


Figure 32: Use after a load instruction

If we use this forwarding, our critical path will be too long. So we choose to make a stall operation here. We created a hazard detection unit for this. This unit gets MUX_B_Select and MUX_D Select from ID/EX register's control word. We get these because in our design, only load operation make these signals both 1. Hazard detection unit gets RD address from ID/EX register too. That is for controlling is the rd and rs addresses are equal. It gets that rs addresses from IF/ID register as rs1 and rs2 addresses. It controls the equality of this registers, and if there is an equal condition. If there is an equality and a load operation, it will generate STALL signal.

Stall operation makes next PC as current PC. So it wont get a new instruction. In this stall cycle, IF/ID register will be goes to itself again to make it stop. We clear ID/EX register and other operations goes the next step as normal. Figures below show the operation.

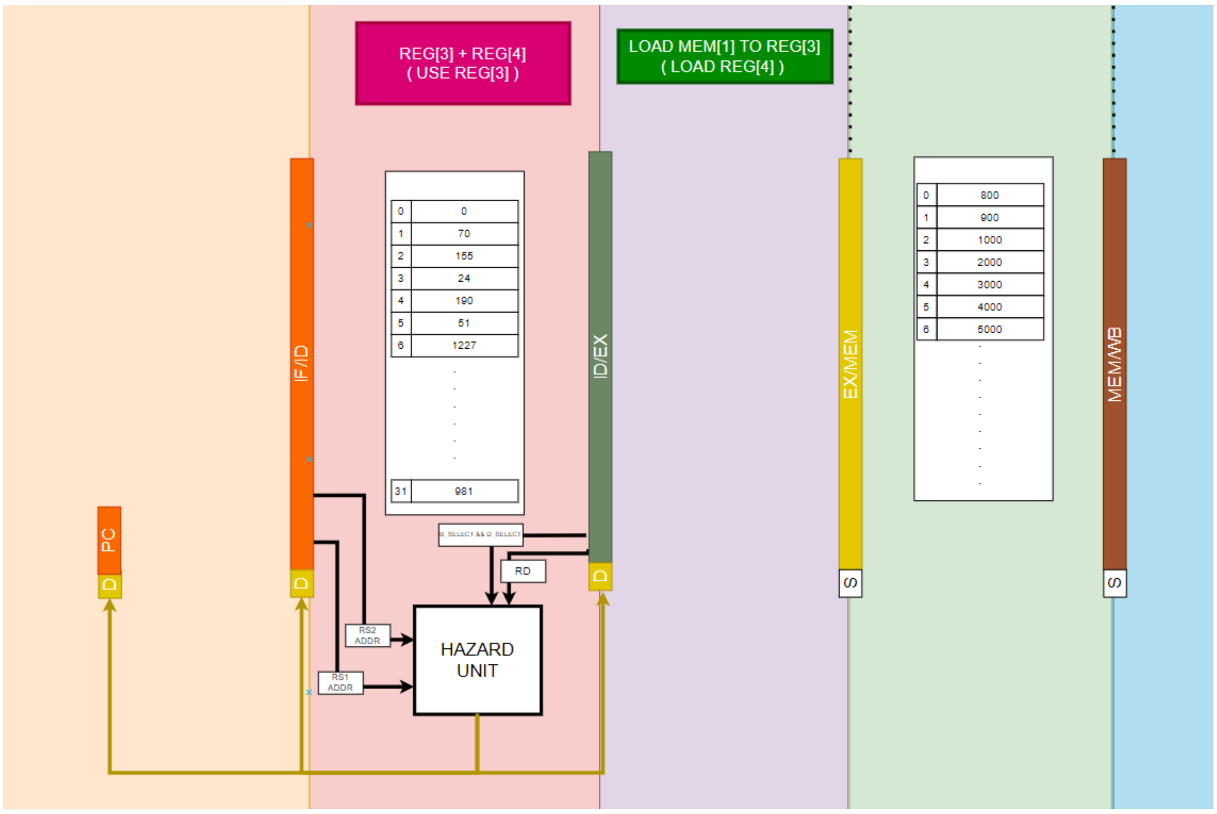


Figure 33. Stall signal from hazard unit.

As it can seen, in this instruction sequence load address and use address are same. Hazard unit detects that and sends stall signal. This stall signal has different decisions on registers. PC goes to current PC, IF/ID register doesn't change and ID/EX register gets NOP in next cycle. Figure 34 shows that clearly by looking registers' colors.

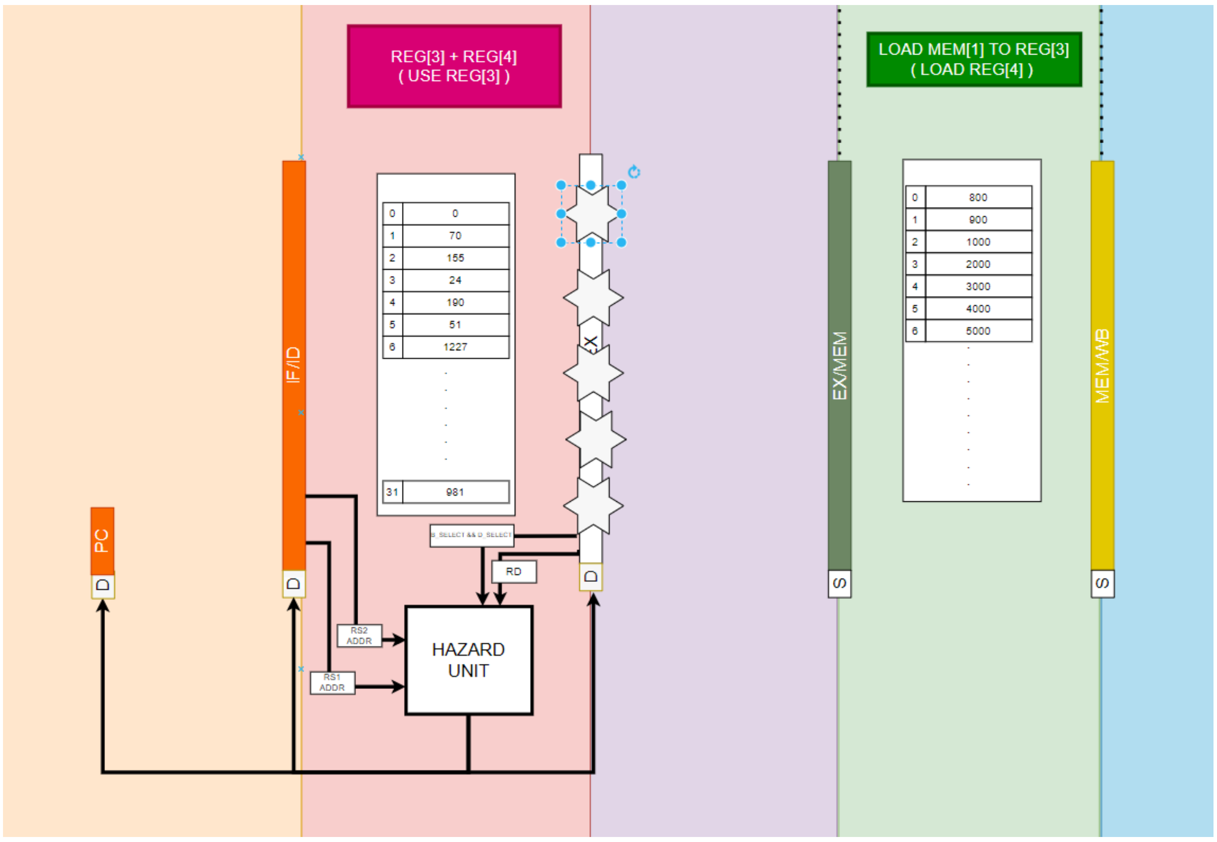


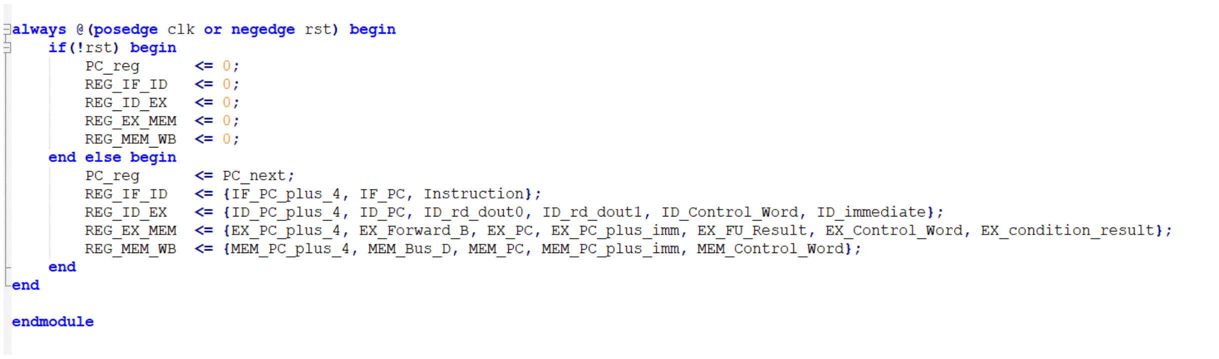
Figure 33. After stall signal

After doing that, our previous forwarding units solves the problem.

Hazard unit verilog code:

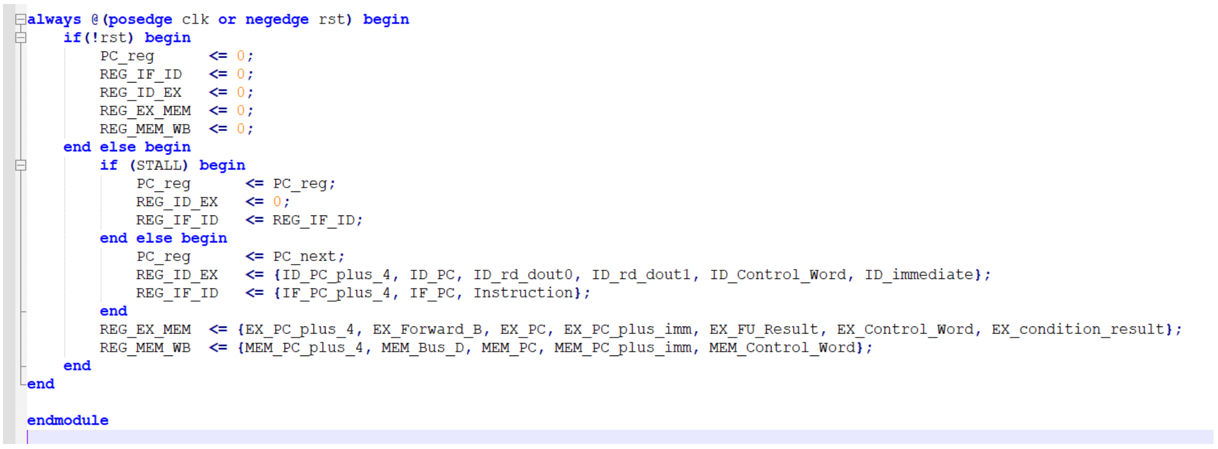
```
module hazard_detection_unit(
    input         EX_Mux_B_Select,
    input         EX_Mux_D_Select,
    input  [4:0]  EX_RD,
    input  [4:0]  ID_RS1,
    input  [4:0]  ID_RS2,
    output        stall
    );

wire MEM_Read;
assign MEM_Read          = EX_Mux_B_Select & EX_Mux_D_Select;
assign stall             = MEM_Read & (EX_RD == ID_RS1 | EX_RD == ID_RS2);

endmodule
```

These changes in verilog code shown in figures below.



```verilog
always @(posedge clk or negedge rst) begin
    if(!rst) begin
        PC_reg      <= 0;
        REG_IF_ID   <= 0;
        REG_ID_EX   <= 0;
        REG_EX_MEM  <= 0;
        REG_MEM_WB  <= 0;
    end else begin
        PC_reg      <= PC_next;
        REG_IF_ID   <= {IF_PC_plus_4, IF_PC, Instruction};
        REG_ID_EX   <= {ID_PC_plus_4, ID_PC, ID_rd_dout0, ID_rd_dout1, ID_Control_Word, ID_immediate};
        REG_EX_MEM  <= {EX_PC_plus_4, EX_Forward_B, EX_PC, EX_PC_plus_imm, EX_FU_Result, EX_Control_Word, EX_condition_result};
        REG_MEM_WB  <= {MEM_PC_plus_4, MEM_Bus_D, MEM_PC, MEM_PC_plus_imm, MEM_Control_Word};
    end
end

endmodule
```

Figure 34: Previous code



```verilog
always @(posedge clk or negedge rst) begin
    if(!rst) begin
        PC_reg      <= 0;
        REG_IF_ID   <= 0;
        REG_ID_EX   <= 0;
        REG_EX_MEM  <= 0;
        REG_MEM_WB  <= 0;
    end else begin
        if (STALL) begin
            PC_reg      <= PC_reg;
            REG_ID_EX   <= 0;
            REG_IF_ID   <= REG_IF_ID;
        end else begin
            PC_reg      <= PC_next;
            REG_ID_EX   <= {ID_PC_plus_4, ID_PC, ID_rd_dout0, ID_rd_dout1, ID_Control_Word, ID_immediate};
            REG_IF_ID   <= {IF_PC_plus_4, IF_PC, Instruction};
        end
        REG_EX_MEM  <= {EX_PC_plus_4, EX_Forward_B, EX_PC, EX_PC_plus_imm, EX_FU_Result, EX_Control_Word, EX_condition_result};
        REG_MEM_WB  <= {MEM_PC_plus_4, MEM_Bus_D, MEM_PC, MEM_PC_plus_imm, MEM_Control_Word};
    end
end

endmodule
```

Figure 35: Current code

Behavioral Simulation

We solved load use hazards by introducing one cycle NOP instruction if:

- There is an instruction that uses loaded data from memory after the load instruction.
- For example:
    - lw x5, 4(x0)
    - addi x5, x5, 3
- In that case we add one NOP instruction between these instructions.
- That operation can be thought as:
    - lw x5, 4(x0)
    - NOP
    - addi x5, x5, 3
- This time our previously designed forward unit forwards the data from WB stage to EX stage and hazard should be solved.

```
addi x5, x0, 5
addi x10, x0, 7
sw   x5, 4(x0)
addi x6, x5, 1
addi x7, x6, 1
lw   x8, 4(x0) #1 this is an load use hazards
and x11, x8, x10 #2 nop instruction should be inserted between #1 and #2
sub x12, x8, x6
add x13, x5, x8
```

This code does the operations below:

- x5 = 5
- x10 = 7
- stores x5 = 5 to memory address 4.
- X6 = x5 + 1 = 6
- X7 = x6 + 1 = 7
- X8 = 5 (data at memory address 4)
- NOP operation because of load use hazard
- X11 = x8 & x10 = 5 & 7 = 0101 & 0111 = 5
- X12 = x8 – x6 = 5 – 6 = -1
- X13 = x5 + x8 = 5 + 5 = 10

This code has read after write hazards too, so we will test the load and use hazards and RAW hazards together.

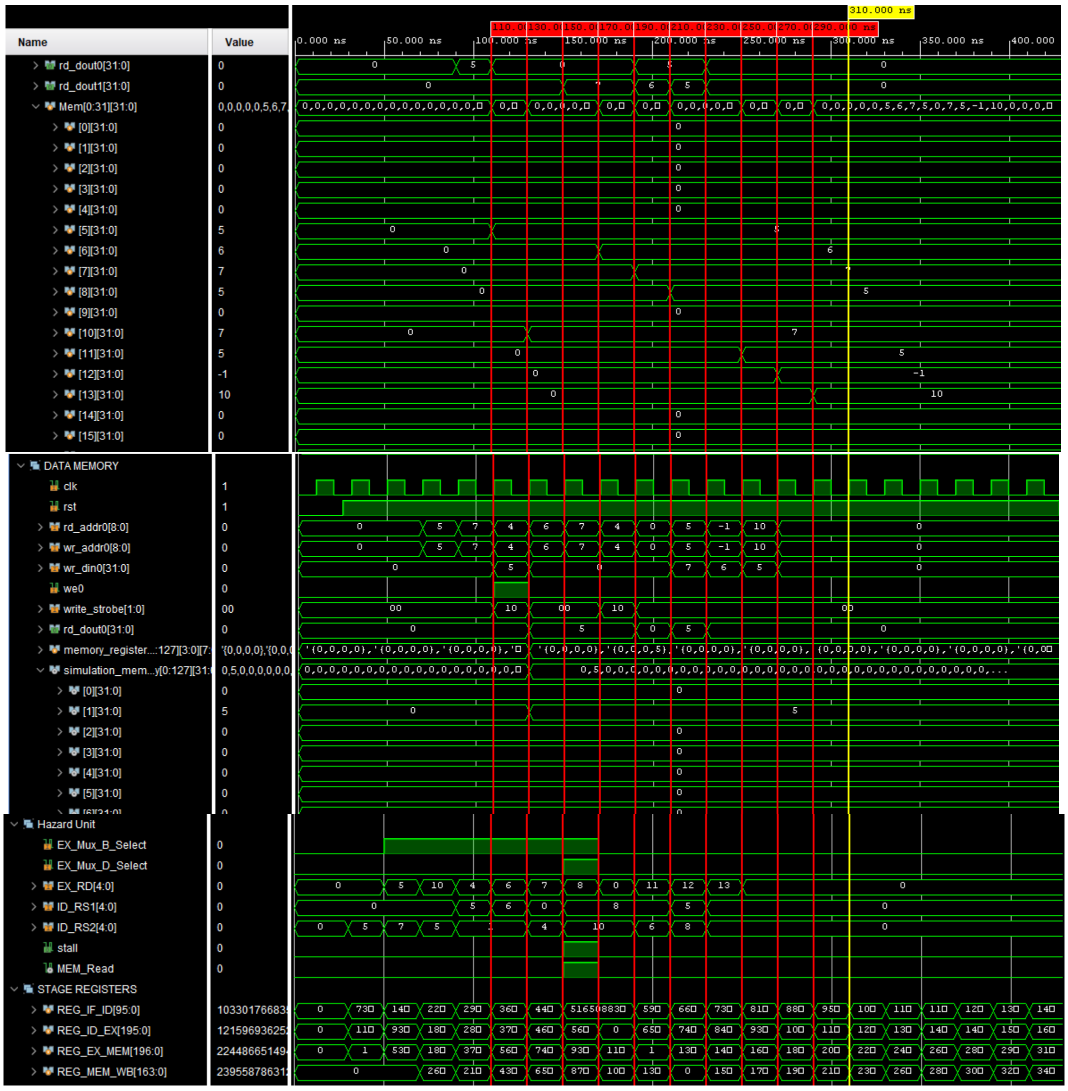


Figure 36: Load use hazards simulation.

In Figure 36, top image is register file, middle image is data memory, bottom image is hazard unit (unit that handles inserting the NOP instruction, stall) and stage registers.

- At marker 1 (110 ns), x5 = 5 is done.
- At marker 2 (130 ns), x10 = 7 and memory address 4 = 5, meaning addi x10, x0, 7, sw x5, 4(x0) done at the same time and correctly. These instructions done at the same time because when addi is in the WB stage sw is in the MEM stage. They write at the same time but in different stages.
- At marker 3 (150 ns), nothing is done because both memory write and register file write operations are done at the marker 2. That means CPI is still 1.
- At marker 4 (170 ns), x6 = 6 is done. At this time, lw x5, 4(x0) instruction is at the EX stage and and x11, x8, x10 instruction is in the ID stage. Hazard unit detects the load use hazards and gives the stall output as high. We implemented stalling as:
    - IF/ID register and PC stays the same
    - ID/EX register is made 0. (introducing nop)
    - At the next cycle normal pipeline register transfer operations continued. And NOP instruction is inserted between lw and and instructions.
- Notice that at the bottom image, at fourth marker REG_IF_ID register stays the same and REG_ID_EX becomes 0. That way we were able to insert NOP operation. NOP operation continues in the pipeline and can be seen in STAGE REGISTERS section in the Figure 36.

- At marker 5 (190 ns), x7 = 7 is done.
- At marker 6 (210 ns), x8 = 5 is done, register 8 is loaded with the value at memory address 4.
- At marker 7 (230 ns), nothing happens because of the NOP instruction that inserted.
- At marker 8 (250 ns), x11 = 5 is done.
- At marker 9 (270 ns), x11 = -1 is done.
- At marker 10 (290 ns), x12 = 11 is done.

That means inserted NOP operation makes CPI = 2 for load use hazard. That is when register to be loaded is used after loading for any instruction. Other instructions has CPI = 1 still.

5) Control Hazards - Jump:

In jump operation, there will be a control hazard which is becoming after instruction. We changed a little bit our design in this step. We carried PC + Imm adder to ID stage. We'll add flash unit for jump instruction. If it is JAL or JALR this unit will send flush signal to IF/ID. In this cycle, PC will be updated as expected in JAL or JALR instruction and next IF/ID register will be flushed. Then system will continue as normal. First we'll explain the expected work. Then we'll explain the hazards for JALR and how we fix it.

Control hazard in Jump instructions is, when it ended in ID stage, there will be a new instruction which is after the Jump instruction in Instruction Memory. But we don't want this instruction to execute because we want PC to jump immediate + pc or rs1 + pc value. To fix this, we created an hazard detection unit which flushes IF/ID stage in next cycle if there is a jump instruction. Figure 37 shows how the control hazard of jump is fixed. We also carried the PC+Immediate adder from EX stage to ID stage. So CPI is 2.

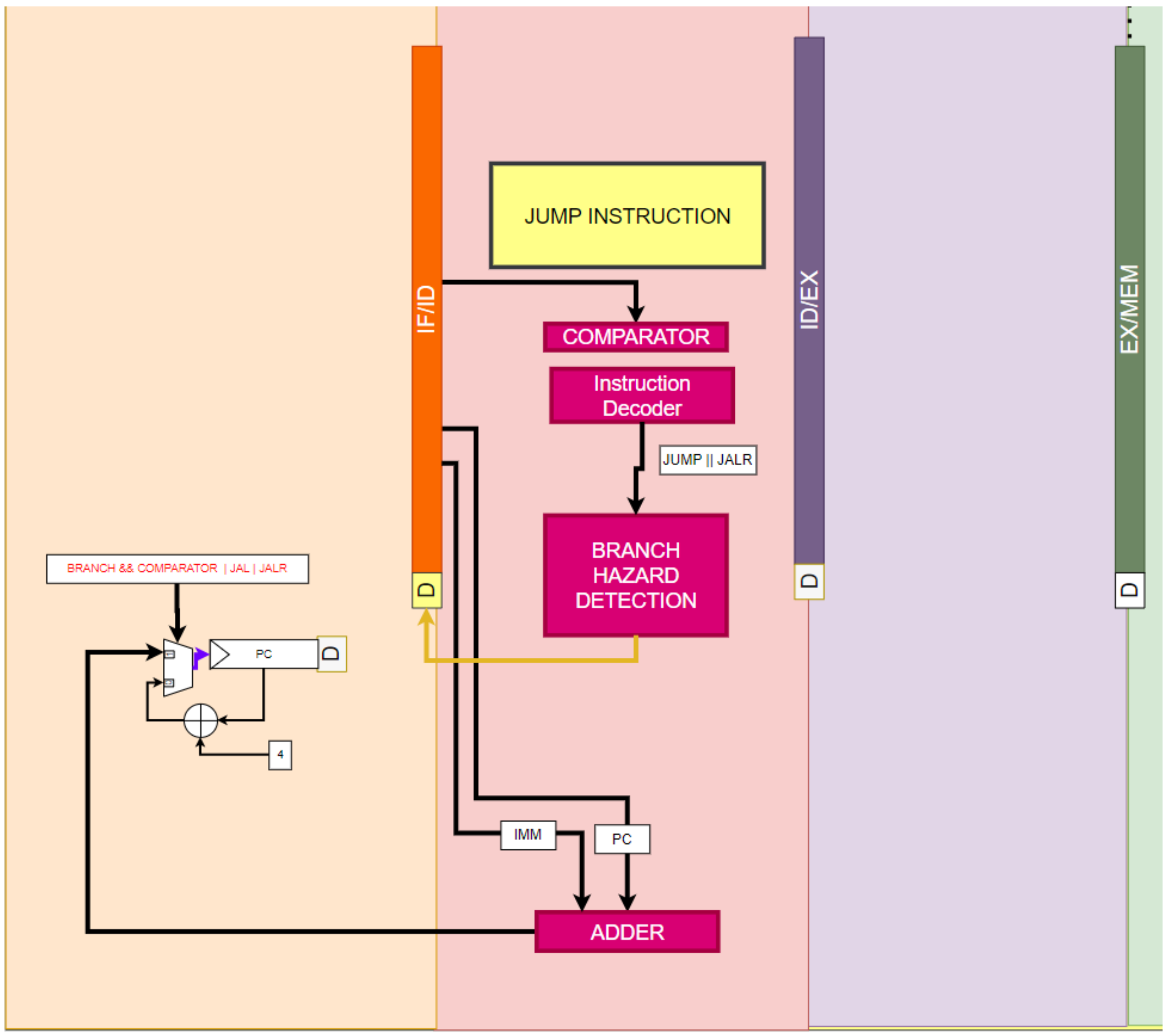


Figure 37. JUMP instruction in ID stage

Things will happen in next cycle is shown in figure 38.

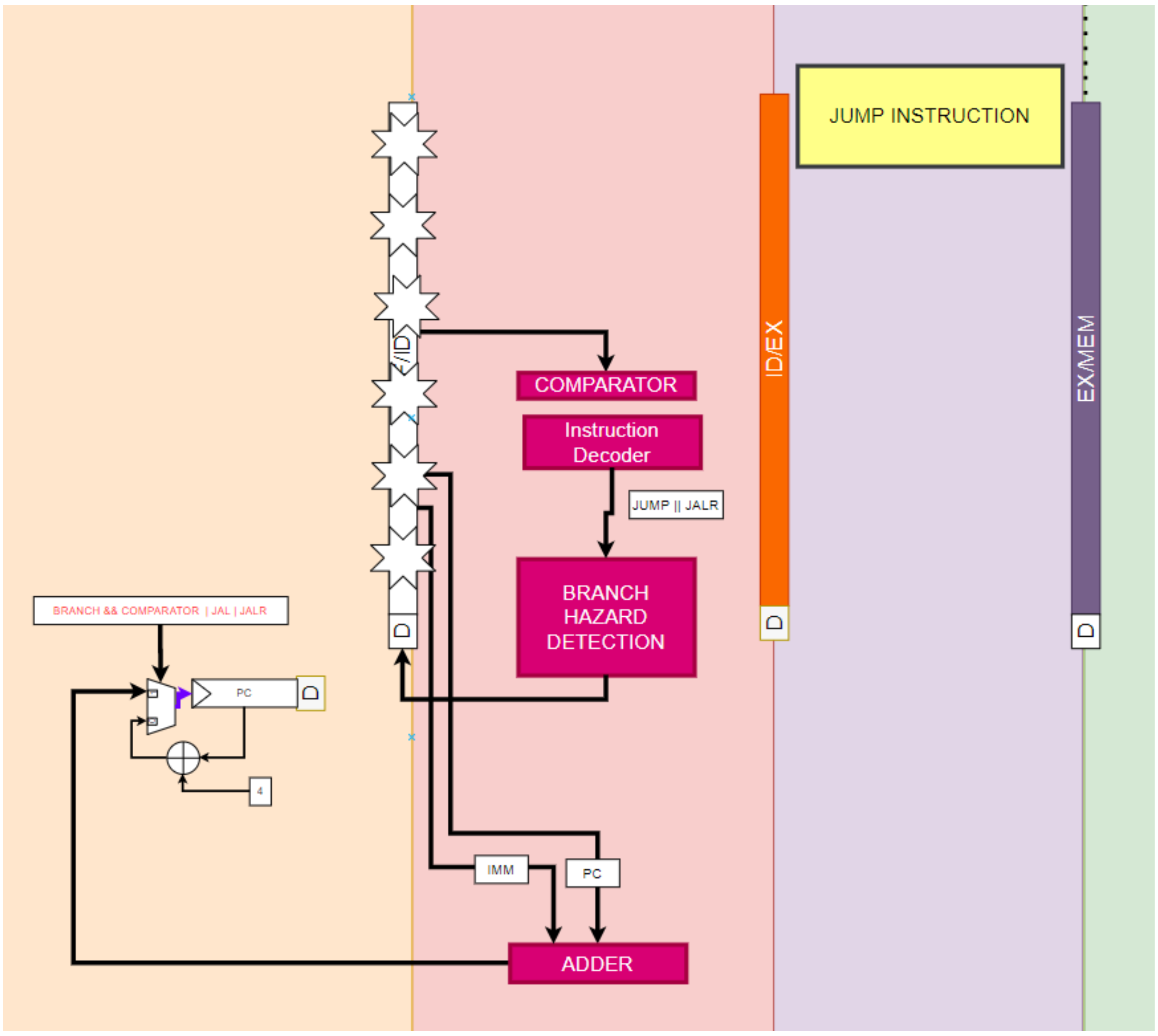


Figure 38. JUMP instruction in EX stage

So now PC is Jumped value and jump instruction goes through end in every cycle and instruction that should be flushed is flushed. This is how we fixed the jump control hazard.

But there is another hazard in JALR instruction. JALR instruction uses value on rs1 address in reg file to make addition with PC. But if there is an instruction before Jump which uses this rs1 address as rd address, there will be a data hazard. This should be fixed by forwarding or stall. By conditions, if the condition is on EX stage, there will be a stall operation, if condition is on MEM stage, we forward value through the adder. If condition is on WB stage, there is no problem because we bypassed it. Figurizing this is not going to show it clearly, but this operation is inculded in Branch Hazard Detection unit. Our proccessor can make this operations and we tested it. But our report is going to be too long and we did not include this waveform and testbench results.

*** Comparator input comes from register file, not from ID/IF register. Sorry for that figures ***

Added parts in verilog code:

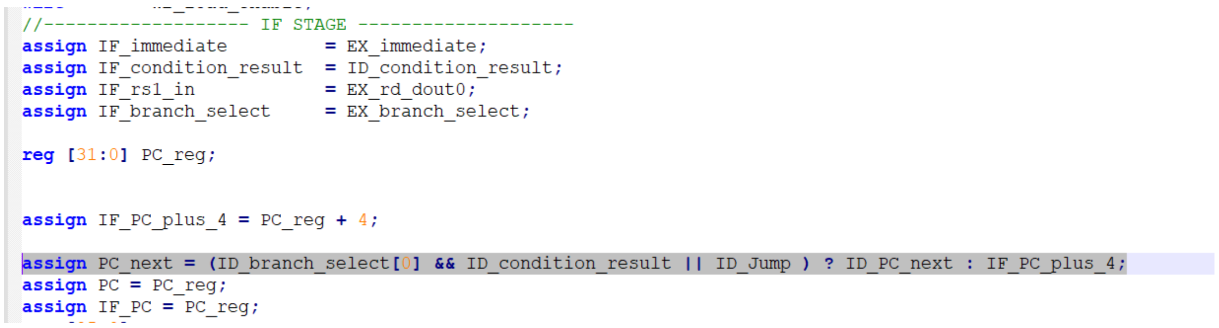
Comparator:

```
module comparator_unit(
    input [31:0] RS1,
    input [31:0] RS2,
    input        branch,
    input [2:0]  condition,
    output reg   condition_result
    );

localparam  BEQ     =           3'b000;
localparam  BNE     =           3'b001;
localparam  BLT     =           3'b100;
localparam  BGE     =           3'b101;
localparam  BLTU    =           3'b110;
localparam  BGEU    =           3'b111;
localparam  SLT     =           3'b010;
localparam  SLTU    =           3'b011;

wire EQ  = (RS1 == RS2);
wire NEQ = (RS1 != RS2);
wire LT  = $signed(RS1) < $signed(RS2);
wire LTU = $unsigned(RS1) < $unsigned(RS2);
wire GE  = $signed(RS1) >= $signed(RS2);
wire GEU = $unsigned(RS1) >= $unsigned(RS2);
always @(*) begin
    condition_result = 1'b0;
    case( condition )
        BEQ:        condition_result    = EQ;
        BNE:        condition_result    = NEQ;
        BLT, SLT:   condition_result    = LT;
        BGE:        condition_result    = GE;
        BLTU, SLTU: condition_result    = LTU;
        BGEU:       condition_result    = GEU;
    endcase
end
endmodule
```

Branch Hazard Detection:

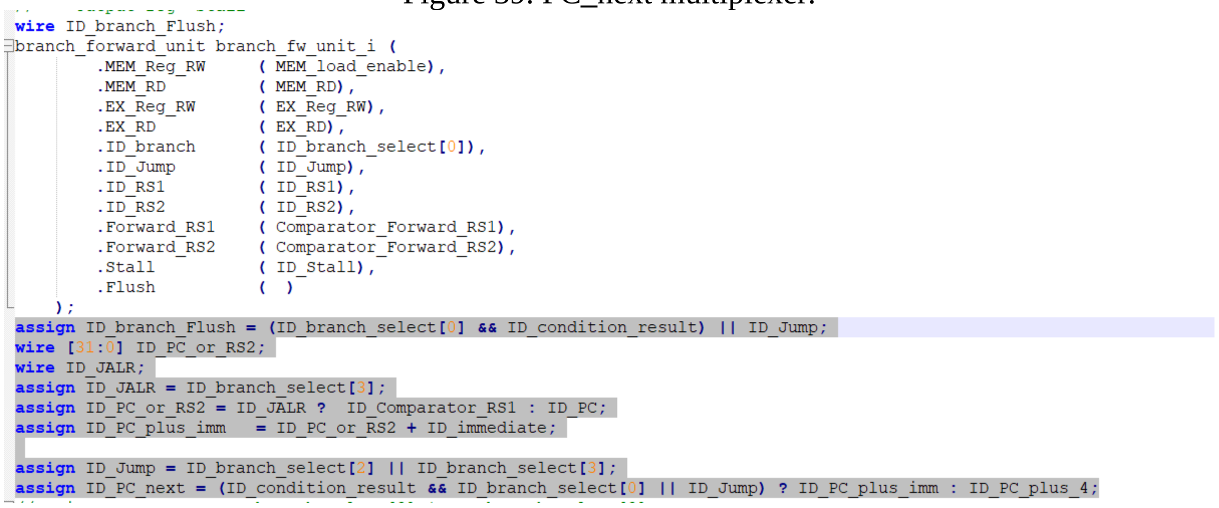
```
module branch_forward_unit(
    input MEM_Reg_RW,
    input [4:0] MEM_RD,
    input       EX_Reg_RW,
    input [4:0] EX_RD,
    input       ID_branch,
    input       ID_Jump,
    input [4:0] ID_RS1,
    input [4:0] ID_RS2,
    output reg  Forward_RS1,
    output reg  Forward_RS2,
    output reg  Stall,
    output reg  Flush
    );


always @(*) begin
    Forward_RS1 = 1'b0;
    Forward_RS2 = 1'b0;
    Stall       = 1'b0;
    //forwarding for RS1 input of the comparator
    if(MEM_Reg_RW && (ID_branch || ID_Jump) && MEM_RD != 5'd0 && MEM_RD == ID_RS1)
        Forward_RS1 = 1'b1;
    else
        Forward_RS1 = 1'b0;
    //forwarding for RS2 input of the comparator
    if(MEM_Reg_RW && (ID_branch || ID_Jump) && MEM_RD != 5'd0 && MEM_RD == ID_RS2)
        Forward_RS2 = 1'b1;
    else
        Forward_RS2 = 1'b0;
    //Stalling for EX stage
    if(EX_Reg_RW && (ID_branch || ID_Jump) && EX_RD != 5'd0 && (EX_RD == ID_RS1 || EX_RD == ID_RS2))
        Stall = 1'b1;
    else
        Stall = 1'b0;

end

endmodule
```

Changes in verilog codes are included in figures below.



```
//------------------- IF STAGE --------------------
assign IF_immediate         = EX_immediate;
assign IF_condition_result  = ID_condition_result;
assign IF_rs1_in            = EX_rd_dout0;
assign IF_branch_select     = EX_branch_select;

reg [31:0] PC_reg;


assign IF_PC_plus_4 = PC_reg + 4;

assign PC_next = (ID_branch_select[0] && ID_condition_result || ID_Jump ) ? ID_PC_next : IF_PC_plus_4;
assign PC = PC_reg;
assign IF_PC = PC_reg;
```

Figure 39. PC_next multiplexer.



```
wire ID_branch_Flush;
branch_forward_unit branch_fw_unit_i (
        .MEM_Reg_RW      ( MEM_load_enable),
        .MEM_RD          ( MEM_RD),
        .EX_Reg_RW       ( EX_Reg_RW),
        .EX_RD           ( EX_RD),
        .ID_branch       ( ID_branch_select[0]),
        .ID_Jump         ( ID_Jump),
        .ID_RS1          ( ID_RS1),
        .ID_RS2          ( ID_RS2),
        .Forward_RS1     ( Comparator_Forward_RS1),
        .Forward_RS2     ( Comparator_Forward_RS2),
        .Stall           ( ID_Stall),
        .Flush           (  )
    );
assign ID_branch_Flush = (ID_branch_select[0] && ID_condition_result) || ID_Jump;
wire [31:0] ID_PC_or_RS2;
wire ID_JALR;
assign ID_JALR = ID_branch_select[3];
assign ID_PC_or_RS2 = ID_JALR ?  ID_Comparator_RS1 : ID_PC;
assign ID_PC_plus_imm    = ID_PC_or_RS2 + ID_immediate;

assign ID_Jump = ID_branch_select[2] || ID_branch_select[3];
assign ID_PC_next = (ID_condition_result && ID_branch_select[0] || ID_Jump) ? ID_PC_plus_imm : ID_PC_plus_4;
```

Figure 40. Adder caried to ID stage and forward rs1 mux.

Behavioral Simulation

We solved jump hazard by calculating the target pc in ID stage and if jal or jalr instruction is detected, this calculated target is going to be next PC at the next clock edge. But instruction after the jump instruction is already fetched so we flush the instruction after the jump instruction.

For the JALR instruction we used assembly code below to test the processor.

```
addi x5, x0, 5   #0
addi x5, x0, 20 #4
addi x6, x0, 6   #8
jalr  x7, x5, 8    #12
addi x7, x0, 1    #16
addi x7, x0, 2    #20
addi x7, x0, 3    #24
addi x7, x0, 4    #28
addi x7, x0, 5    #32
addi x7, x0, 6    #36
```

This code does the following:

- X5 = 5
- X5 = 20
- X6 = 6
- Jumps to x5 + 8 = 28 and loads pc + 4 = 12 + 4 = 16 to x7
- If jump is performed correctly, we expect to not see the values 1, 2, 3 in the x7 register we expect to see 4, 5, 6 in x7 register.
- That is because we want to jump to pc = 28 from pc = 12.

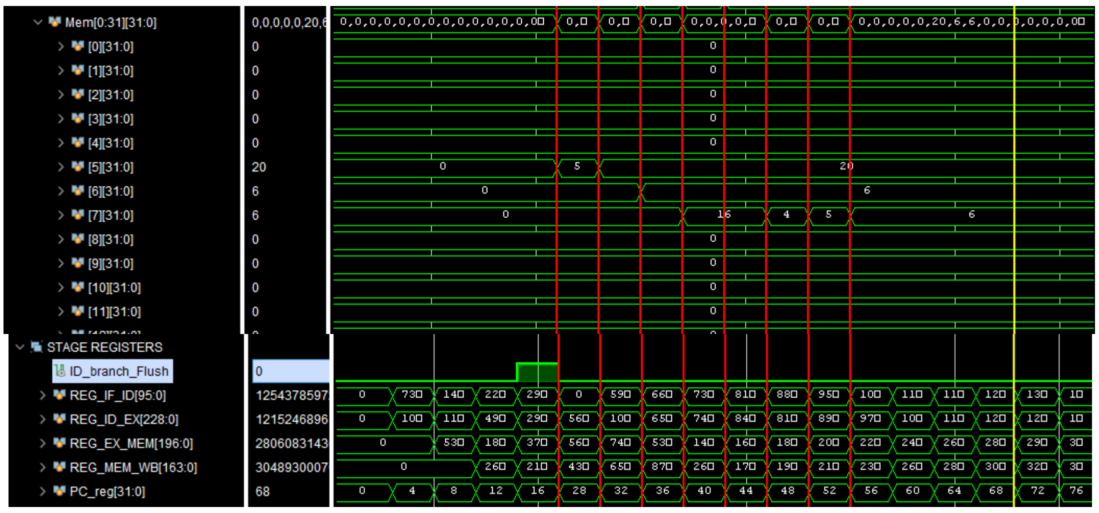


Figure 41: Simulation for JALR.

We can analyze the waveform for calculating the CPI for JALR. In Figure 41, top image is register file and bottom image is flush signal and stage registers. Flush signal becomes high when jalr or jump instruction comes to the ID stage.

- At marker 1, first instruction x5 = 5 reaches to WB stage and value 5 is written into register 5. At this time JALR instruction is in ID stage and that means ID_branch_flush signal (signal that tells the system jump instruction is arrived and at the next cycle instruction after the jump instruction needs to be flushed.) is high. At the marker 1

REG_IF_ID becomes 0, that means flushing the instruction after jump instruction and then going to correct address.
- At marker 2, x5 becomes 20.
- At marker 3, x6 becomes 6.
- At marker 4, x7 becomes 16 which is the return address for jump instruction. JALR instruction is at the address 12 of the instruction memory so return address 12 + 4 = 16 is loaded into x7.
- At marker 5, nothing happens because of the flush after jump instruction.
- At marker 6, x7 becomes 4
- At marker 7, x7 becomes 5
- At marker 8, x7 becomes6

Here, jump address calculation is done at ID stage and one instruction after the jump is flushed. So CPI = 2 for the JALR instruction. There are 7 instructions in the code and 6 of them has CPI = 1 and one JALR instruction has CPI = 2 so total of 8 cycles.

For the JAL instruction we used assembly code below to test the processor.

```
addi x5, x0, 5   #0
addi x5, x0, 20  #4
addi x6, x0, 6   #8
jal x7, jump     #12
addi x7, x0, 1   #16
addi x7, x0, 2   #20
jump:
addi x7, x0, 3   #24
addi x7, x0, 4   #28
addi x7, x0, 5   #32
addi x7, x0, 6   #36
```

This code does the following:
- X5 = 5
- X5 = 20
- X6 = 6
- Jumps to label jump at address 24 and loads pc + 4 = 12 + 4 = 16 to x7
- If jump is performed correctly, we expect to not see the values 1, 2 in the x7 register we expect to see 3, 4, 5, 6 in x7 register.
- That is because we want to jump to pc = 24 from pc = 12.

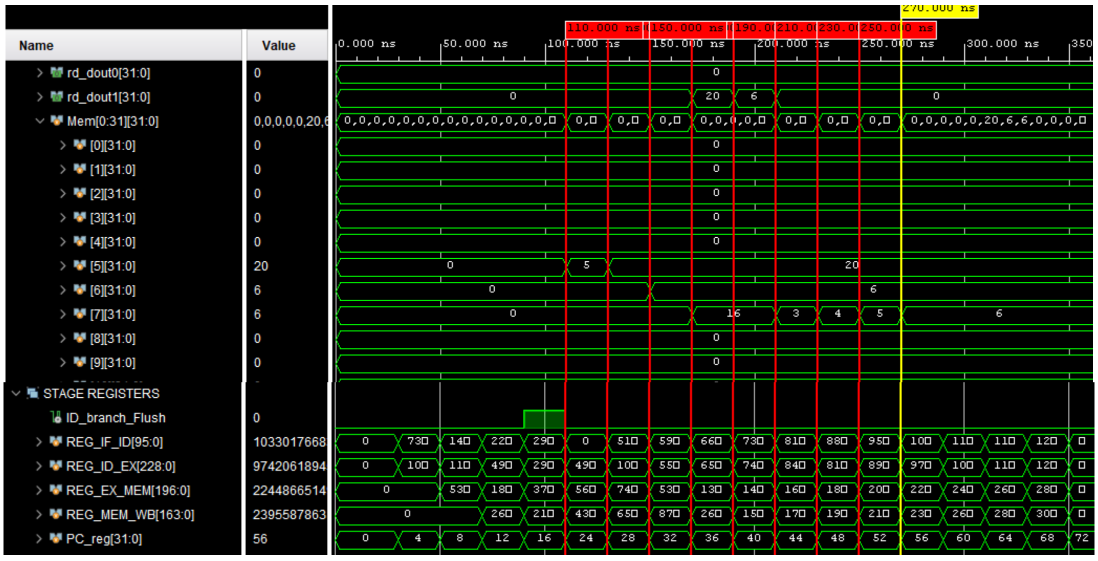


Figure 42: Simulation for JAL.

We can analyze the waveform for calculating the CPI for JAL. In Figure 42, top image is register file and bottom image is flush signal and stage registers. Flush signal becomes high when jalr or jal instruction comes to the ID stage.

- At marker 1, first instruction x5 = 5 reaches to WB stage and value 5 is written into register 5. At this time JAL instruction is in ID stage and that means ID_branch_flush signal (signal that tells the system jump instruction is arrived and at the next cycle instruction after the jump instruction needs to be flushed.) is high. At the marker 1 REG_IF_ID becomes 0, that means flushing the instruction after jump instruction and then going to correct address.
- At marker 2, x5 becomes 20.
- At marker 3, x6 becomes 6.
- At marker 4, x7 becomes 16 which is the return address for jump instruction. JAL instruction is at the address 12 of the instruction memory so return address 12 + 4 = 16 is loaded into x7.
- At marker 5, nothing happens because of the flush after jump instruction.
- At marker 6, x7 becomes 3
- At marker 7, x7 becomes 4
- At marker 8, x7 becomes 5
- At marker 9, x7 becomes 6

Here, jump address calculation is done at ID stage and one instruction after the jump is flushed. So CPI = 2 for the jump instruction. There are 9 instructions in the code and 8 of them has CPI = 1 and one JAL instruction has CPI = 2 so total of 10 cycles.

6) Control Hazards - Branch:

So in branch operations in pipeline, we made our branch operations lastly in ID stage. But as we declared in previous question, there is a control hazard which we should fix. We should flush the instruction which will come after branch instructions because we don't want this instruction to be ended. We will flush this instruction as jump control hazard.

As in JALR, in branch operation there are data hazards too. This hazards is about comparator inputs. If the rd of instruction in EX stage is equal to rs1 or rs2 of branch operation, we make stall. If the rd of instruction in MEM stage is equal to rs1 or rs2 of branch operation we make forwarding. If the rd of instruction in WB stage is equal to rs1 or rs2 of branch operation, there is no problem because we bypassed inside our register file.

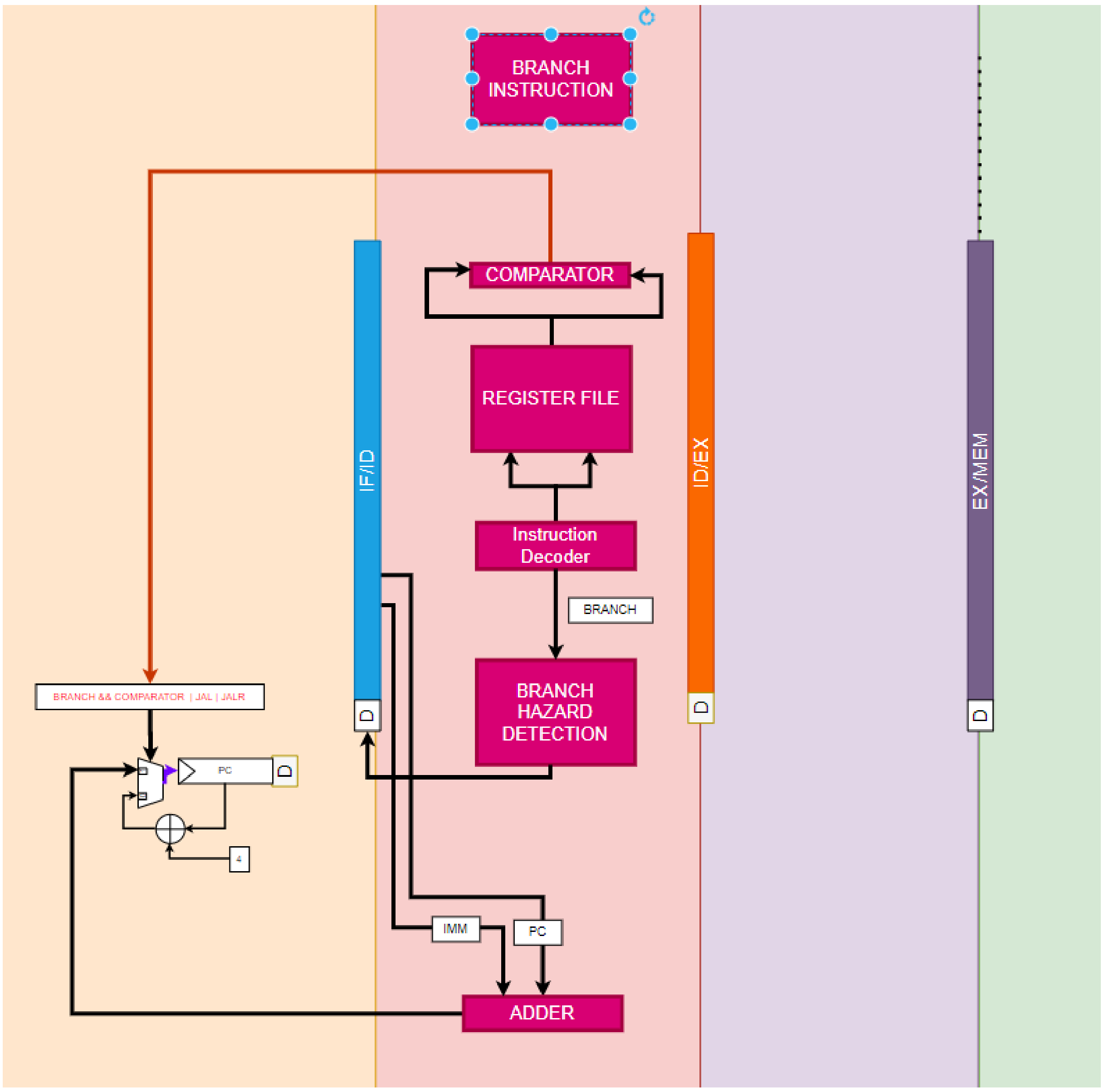


Figure 43. Branch instruction in ID stage

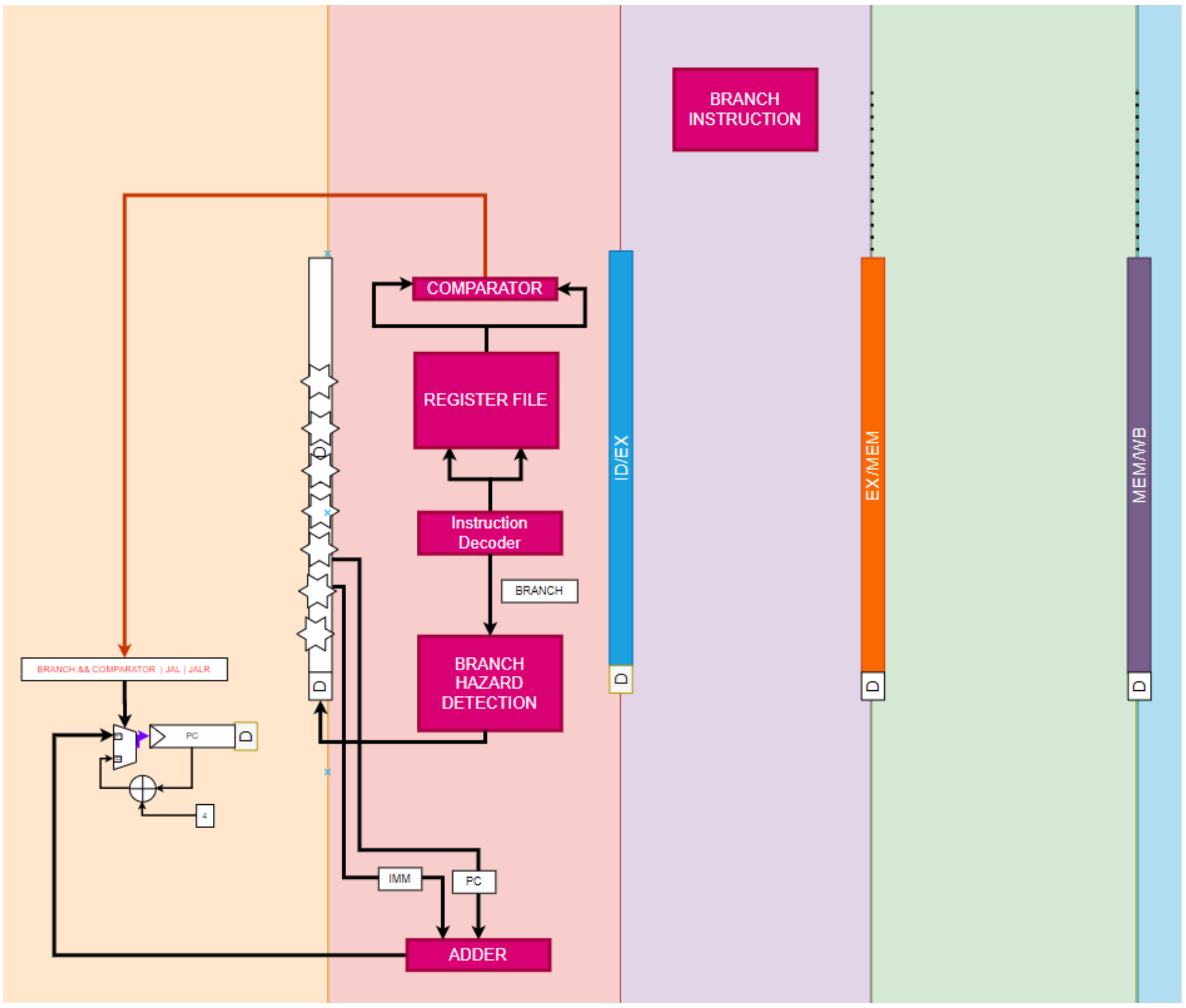


Figure 44. Branch instruction in EX stage, IF/ID flushed.

We make this flush operation in every branch case. But there are other hazards as we quoted. So for data hazards, we included them in branch hazard detection as well. So the difference in here is, we make forwarding operation by placing MUXes next to inputs of comparator.

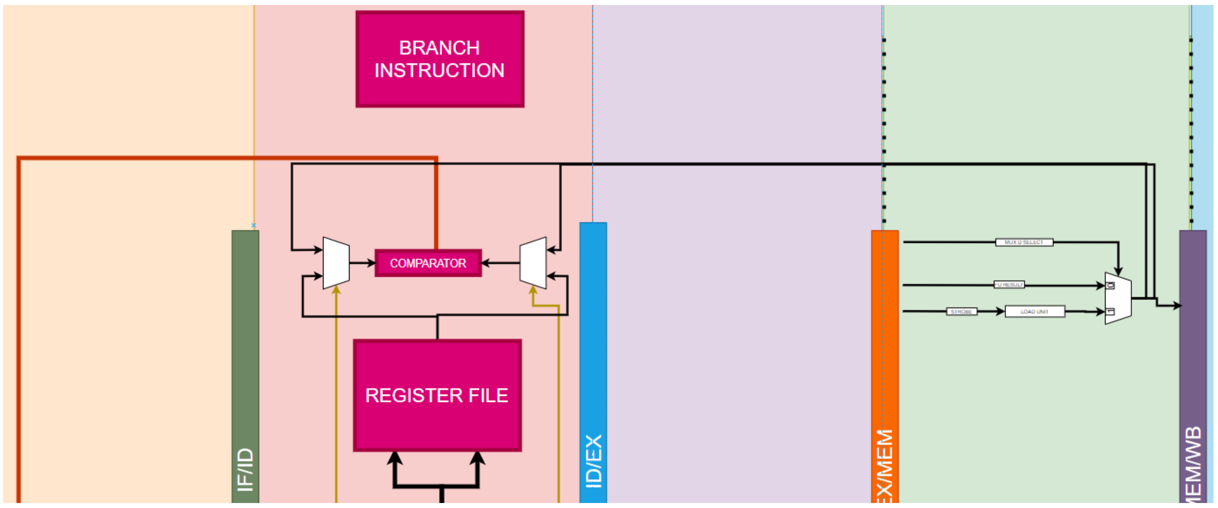


Figure 45. Branch instruction with MEM stage condition.

In figure 45 there is an instruction sequence which is in MEM stage there is an instruction which changes an register file address which branch instruction uses. So we forward this data through muxes next to comparator inputs.

Selection bits of comparator inputs are coming from branch hazard unit.

In the other cases we quoted about what is our solution. We implemented these solutions but we won't show these cases.

There is no changes in verilog codes in this part because we solved 5 and 6 together.

Behavioral Simulation

For testing the branch instructions we wrote the assembly code below, in this code branch taken cases and branch not taken cases are exists, if these cases are working correctly, all branch operations should work correctly. For comparator inputs, we used forwarding from MEM stage.

If there is a dependency between branch instruction at ID stage and register instruction at MEM stage, branch forward unit forwards the data from MEM stage to comparator inputs.

If there is a dependency between branch instruction at ID stage and WB stage, register file bypassing solves this hazard.

If there is a dependency between branch instruction at ID stage and EX stage, forwarding FU output to the comparator input increases critical path so much. Because of that we preferred to stall 1 cycle for ID stage EX stage dependency, after stalling branch instruction stays in the ID stage and dependency instruction at the EX stage goes to MEM stage and branch forward unit takes care of that hazard.

For the first and second case if branch is not taken, CPI becomes 1. For third case if branch is not taken CPI becomes 2 instead of 3 in this special case.

In the first two cases above CPI = 2 is accomplished. But in the third case as the lecture slides said stall is unavoidable, so for that special case we need one stall and one flush (this flush exists for all jump and branch operations and caused by making the decision and calculating the target in the ID stage) making the CPI = 3 for this special case. Other than forwarding FU output to the comparator there is no way to make CPI = 2 for this special case as explained in the lecture slides.

```
addi x5, x0, 5        #0
addi x5, x0, 20       #4
addi x7, x0, 13       #8
addi x6, x0, 6        #12
bne x7, x5, branch1   #16
addi x8, x0, 1        #20
addi x8, x0, 2        #24
addi x8, x0, 3        #28
branch1:
addi x8, x0, 4        #32
addi x8, x0, 5        #36
addi x8, x0, 6        #40
bge x5, x7, branch2   #44
addi x8, x0, 1        #48
addi x8, x0, 2        #52
addi x8, x0, 3        #56
branch2:
addi x8, x0, 7        #60
addi x8, x0, 8        #64
addi x8, x0, 9        #68
blt x5, x7, branch3   #72
addi x8, x0, 1        #76
addi x8, x0, 2        #80
addi x8, x0, 3        #84
branch3:
addi x8, x0, 10       #88
addi x8, x0, 11       #92
addi x8, x0, 12       #96
```

This code fills the registers with the values: x5 = 5, x5 = 20, x7 = 13, x6 = 6

- Then compares x7 and x5 and they are not equal so it takes the first branch and makes the pc 32.
- Second branch at #44, compares x5 >= x7 which is correct, so branch is taken, pc will be 60.
- Third branch at #72, compares x5 < x7 which is false, so branch is not taken, pc will be 76.
- For the sequence above if the code works correctly, we should see register 8 as:
- X8 = 4,5,6,7,8,9,1,2,3,10,11,12

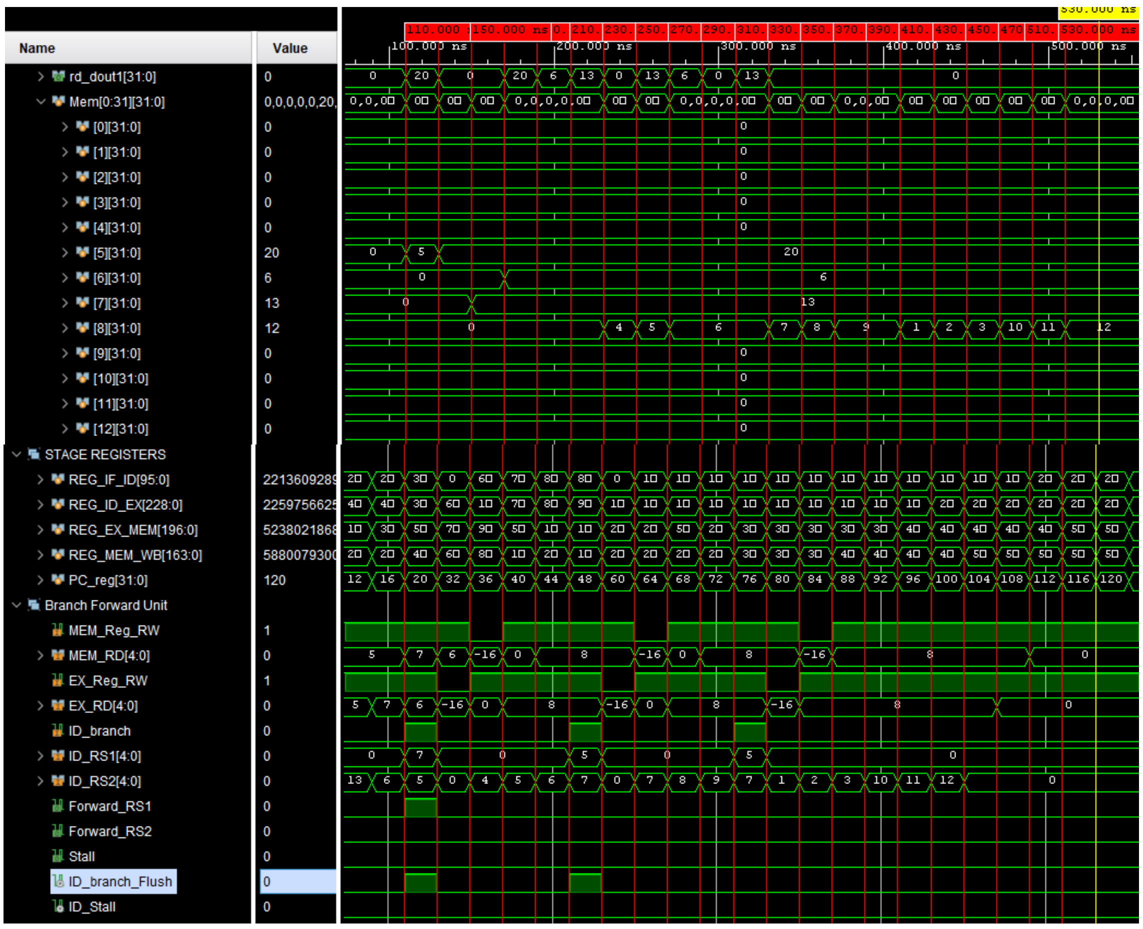


Figure 46: Branch hazards simulation.

In Figure 46, top image is register file and bottom image is stage registers and branch forward unit (for forwarding, stalling and flushing).

- At marker 1 (110 ns), x5 = 5. At this time first branch instruction is in ID stage and ID_branch_flush is 1 for flushing next instruction after the branch. This branch should be taken. Branch instruction is at address 16 and PC automatically becomes 20. After calculating the branch is going to be taken this instruction with address 20 is flushed and REG_IF_ID becomes 0. Then PC becomes 32 which is jump address. Forward RS1 is also 1 at this time because there is dependency between addi x7, x0, 13 and bne x7, x5, branch1 so RS1 data is forwarded from MEM stage.
- At marker 2 (130 ns), x5 = 20. Flushing happens
- At marker 3 (150 ns), x7 =13.
- At marker 4 (170 ns), x6 = 6.
- At marker 5 (190 ns), branch is calculated and executed.
- At marker 6 (210 ns), NOP instruction after the branch (caused by flushing) is executed. CPI for branch is 2.
- At marker 7 (230 ns), x8 = 4, another branch is in ID stage that means ID_branch_flush = 1. REG_IF_ID becomes 0. PC for second branch is 44 and jump address is 60, PC becomes 48 automatically and this instruction is flushed. Then PC becomes 60 and system jumped to the correct address. PC can be seen in PC_reg in waveform.
- At marker 8 (250 ns), x8 = 5.
- At marker 9 (270 ns), x8 = 6.
- At marker 10 (290 ns), second branch is executed.

- At marker 11 (310 ns), NOP instruction after the branch (caused by flushing) is executed. CPI for this branch is 2 because in the next marker (marker 12, 330 ns) executes x8 = 7 which is the target instruction after branch.
- At marker 13 (330 ns), x8 = 7.
- At marker 14 (350 ns), x8 = 8.
- At marker 15 (370 ns), x8 = 9.
- At marker 16 (390 ns), third branch is executed.
- At marker 17 (410), x8 = 1, this time branch is not taken so there is no flushing operation so, CPI = 1 for this branch because branch is not taken.
- After that x8 becomes 2, 3, 10, 11, 12 which is expected from the assembly code.

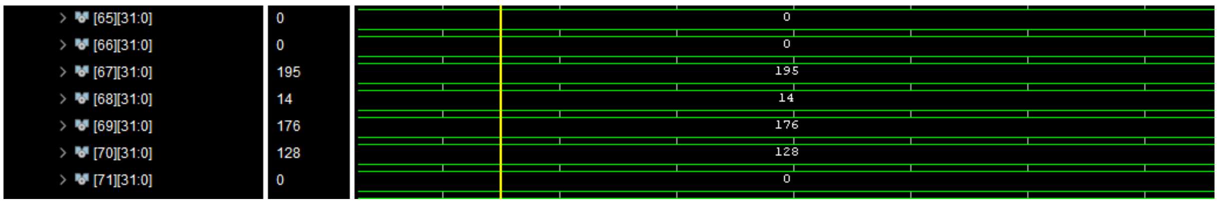
Branch instructions performed correctly with MEM ID dependency and CPI = 2. Non taken branches performed with CPI = 1.

Bubble Sort Behavioral Simulation

We need our instruction memory and data memory to read data from given bubble_sort.data and bubble_sort.instr. For doing that we changed readmemb function in our instruction memory and data memory to readmemh because hexadecimal format is given in the project files. After reading given data and instructions simulation results is given below.

In the question memory address of the data is given an memory address of the output is also given. Memory is byte addressible so, we should divide addresses to 4 to get word address

Input array:
Mem[0x10c = 268 = word67] = 195
Mem[0x110 = 272 = word68] = 14
Mem[0x114 = 276 = word69] = 176
Mem[0x118 = 280 = word70] = 128



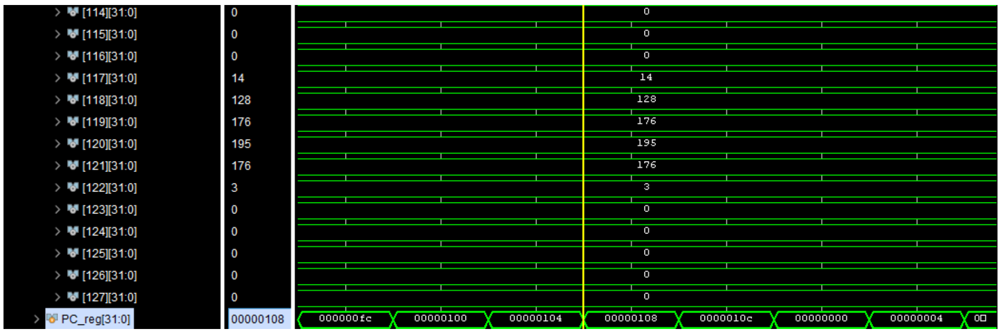
[65][31:0]	0	0
[66][31:0]	0	0
[67][31:0]	195	195
[68][31:0]	14	14
[69][31:0]	176	176
[70][31:0]	128	128
[71][31:0]	0	0

Figure 47: Input array in the memory.

As can be seen from the waveform, reading the given memory data is done correctly.

Expected Output array:
Mem[0x1d4 = 468 = word117] = 14
Mem[0x1d8 = 472 = word118] = 128
Mem[0x1d4 = 476 = word119] = 176
Mem[0x118 = 480 = word120] = 195

These values expected at corresponding addresses when PC becomes 0x108.



```
[114][31:0]   0
[115][31:0]   0
[116][31:0]   0
[117][31:0]   14
[118][31:0]   128
[119][31:0]   176
[120][31:0]   195
[121][31:0]   176
[122][31:0]   3
[123][31:0]   0
[124][31:0]   0
[125][31:0]   0
[126][31:0]   0
[127][31:0]   0
PC_reg[31:0]  00000108   000000fc  00000100  00000104  00000108  0000010c  00000000  00000004  00
```

Figure 48: Output array when PC becomes 0x108.

As can be seen from the waveform above, program sorted the array correctly. There are two extra values coming from c code but expected addresses in the question is filled with sorted values.

Bubble sort algorithm worked correctly in our pipelined RISCV processor.

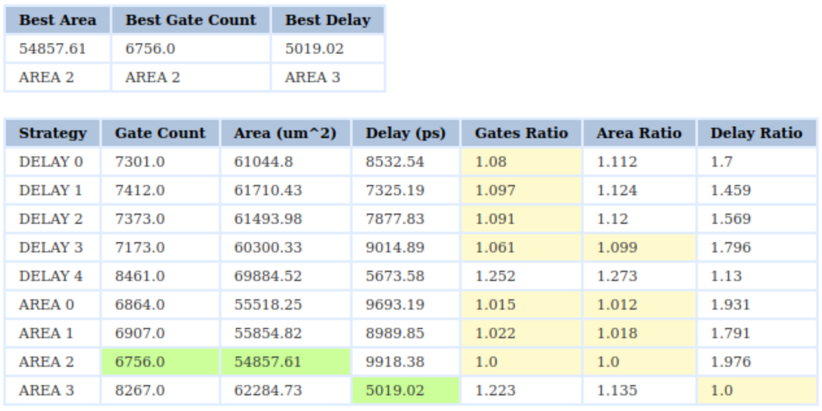
OpenLane Flow

```
$./flow.tcl -design TOP -init_design_config -add_to_designs
```

Config file for fast and error free flow:

```
{
   "DESIGN_NAME": "TOP",
   "VERILOG_FILES": "dir::src/*.v",
   "CLOCK_PORT": "clk",
   "CLOCK_PERIOD": 10,
   "DESIGN_IS_CORE": true,
   "ROUTING_CORES" : 8,
   "KLAYOUT_XOR_THREADS" : 8,
   "SYNTH_STRATEGY": "AREA 3",
   "FP_CORE_UTIL" : 30,
   "PL_TARGET_DENSITY" : 0.35
}
```

```
$./flow.tcl -design TOP -tag synth_explore -synth_explore
```



Best Area	Best Gate Count	Best Delay
54857.61	6756.0	5019.02
AREA 2	AREA 2	AREA 3

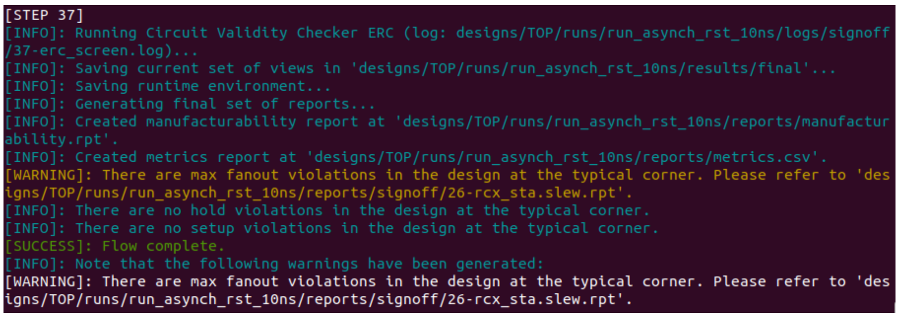
Strategy	Gate Count	Area (um^2)	Delay (ps)	Gates Ratio	Area Ratio	Delay Ratio
DELAY 0	7301.0	61044.8	8532.54	1.08	1.112	1.7
DELAY 1	7412.0	61710.43	7325.19	1.097	1.124	1.459
DELAY 2	7373.0	61493.98	7877.83	1.091	1.12	1.569
DELAY 3	7173.0	60300.33	9014.89	1.061	1.099	1.796
DELAY 4	8461.0	69884.52	5673.58	1.252	1.273	1.13
AREA 0	6864.0	55518.25	9693.19	1.015	1.012	1.931
AREA 1	6907.0	55854.82	8989.85	1.022	1.018	1.791
AREA 2	6756.0	54857.61	9918.38	1.0	1.0	1.976
AREA 3	8267.0	62284.73	5019.02	1.223	1.135	1.0

Figure 49: Synthesis exploration output.

To get speed focused performance, we chose AREA 3 strategy for synthesis and added the line below to our config file, then ran the flow.

"SYNTH_STRATEGY": "AREA 3"

```
$./flow.tcl -design TOP -tag run1
```

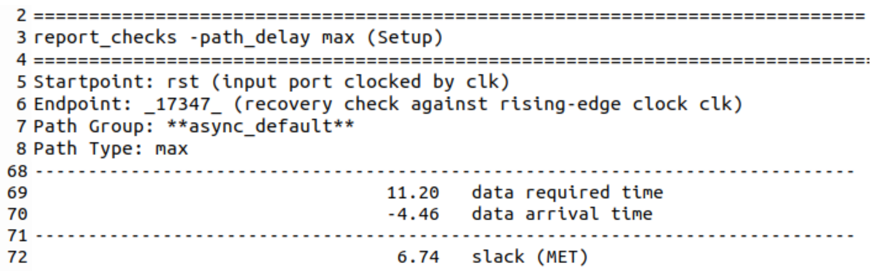


```
[STEP 37]
[INFO]: Running Circuit Validity Checker ERC (log: designs/TOP/runs/run_asynch_rst_10ns/logs/signoff
/37-erc_screen.log)...
[INFO]: Saving current set of views in 'designs/TOP/runs/run_asynch_rst_10ns/results/final'...
[INFO]: Saving runtime environment...
[INFO]: Generating final set of reports...
[INFO]: Created manufacturability report at 'designs/TOP/runs/run_asynch_rst_10ns/reports/manufactur
ability.rpt'.
[INFO]: Created metrics report at 'designs/TOP/runs/run_asynch_rst_10ns/reports/metrics.csv'.
[WARNING]: There are max fanout violations in the design at the typical corner. Please refer to 'des
igns/TOP/runs/run_asynch_rst_10ns/reports/signoff/26-rcx_sta.slew.rpt'.
[INFO]: There are no hold violations in the design at the typical corner.
[INFO]: There are no setup violations in the design at the typical corner.
[SUCCESS]: Flow complete.
[INFO]: Note that the following warnings have been generated:
[WARNING]: There are max fanout violations in the design at the typical corner. Please refer to 'des
igns/TOP/runs/run_asynch_rst_10ns/reports/signoff/26-rcx_sta.slew.rpt'.
```

Figure 50: OpenLane flow result.

Flow completed successfully, there are only max fanout violations and in the experiment sheet it is said that we can ignore fanout violations.

Maximum Clock Frequency



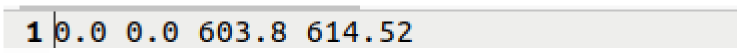
```
===========================================================================
report_checks -path_delay max (Setup)
===========================================================================
Startpoint: rst (input port clocked by clk)
Endpoint: _17347_ (recovery check against rising-edge clock clk)
Path Group: **async_default**
Path Type: max
---------------------------------------------------------------------------
                                  11.20   data required time
                                  -4.46   data arrival time
---------------------------------------------------------------------------
                                   6.74   slack (MET)
```

Figure 51: Longest path and its slack.

We ran the flow with 10 ns clock. In Figure 51, (/signoff/26-rcx_sta.max.rpt) longest path I our design can be seen. For calculating maximum frequency:

$$f_{max} = \frac{1}{clock\ period - smallest\ slack} = \frac{1}{(10 - 6.74)10^{-9}} = 306.748\ MHz$$

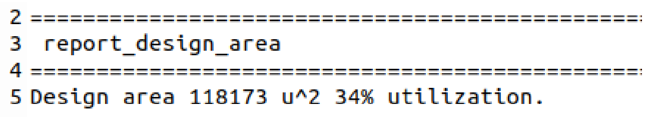
Total Die Area



```
0.0 0.0 603.8 614.52
```

Figure 52: Initial floorplan die area(reports/floorplan/3-initial_fp_die_area.rpt).

Die area is = 603.8 um * 614.52 um = 371047 um^2 = 0.371047 mm^2

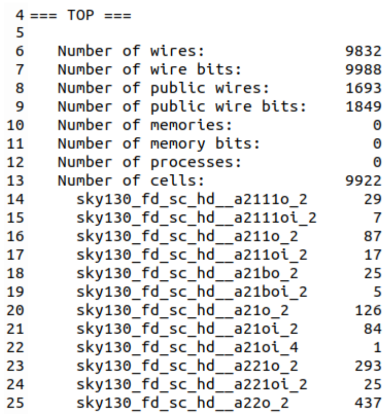


```
==============================================
 report_design_area
==============================================
Design area 118173 u^2 34% utilization.
```

Figure 53: Design area (reports/signoff/26-rcx_sta.area.rpt)

Design area is 118173 um^2 = 0.118173 mm^2

Total Cell Count

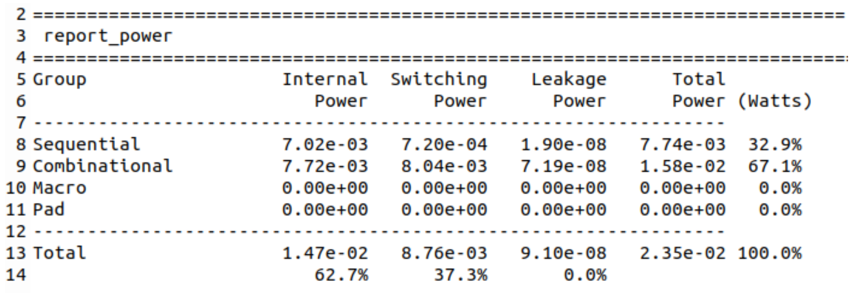


```
=== TOP ===

   Number of wires:               9832
   Number of wire bits:           9988
   Number of public wires:        1693
   Number of public wire bits:    1849
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:               9922
     sky130_fd_sc_hd__a2111o_2      29
     sky130_fd_sc_hd__a2111oi_2      7
     sky130_fd_sc_hd__a211o_2       87
     sky130_fd_sc_hd__a211oi_2      17
     sky130_fd_sc_hd__a21bo_2       25
     sky130_fd_sc_hd__a21boi_2       5
     sky130_fd_sc_hd__a21o_2       126
     sky130_fd_sc_hd__a21oi_2       84
     sky130_fd_sc_hd__a21oi_4        1
     sky130_fd_sc_hd__a221o_2      293
     sky130_fd_sc_hd__a221oi_2      25
     sky130_fd_sc_hd__a22o_2       437
```

Figure 54: Total cell count (reports/synthesis/1-synthesis.AREA_3.stat.rpt)

Total cell usage is 9922 cells.

Approximate Power Consumption



```
==========================================================================
 report_power
==========================================================================
Group                  Internal  Switching    Leakage      Total
                          Power      Power      Power      Power (Watts)
----------------------------------------------------------------
Sequential             7.02e-03   7.20e-04   1.90e-08   7.74e-03  32.9%
Combinational          7.72e-03   8.04e-03   7.19e-08   1.58e-02  67.1%
Macro                  0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
Pad                    0.00e+00   0.00e+00   0.00e+00   0.00e+00   0.0%
----------------------------------------------------------------
Total                  1.47e-02   8.76e-03   9.10e-08   2.35e-02 100.0%
                          62.7%      37.3%       0.0%
```

Figure 55: Power report (reports/signoff/26-rcx_sta.power.rpt).

Total power usage is 23.5 mW.

OpenRoad Gui

I opened the openroad gui and used the command below to see the chip after placement and routing.

```
$read_db designs/TOP/runs/run_asynch_rst_10ns/results/routing/TOP.odb
```

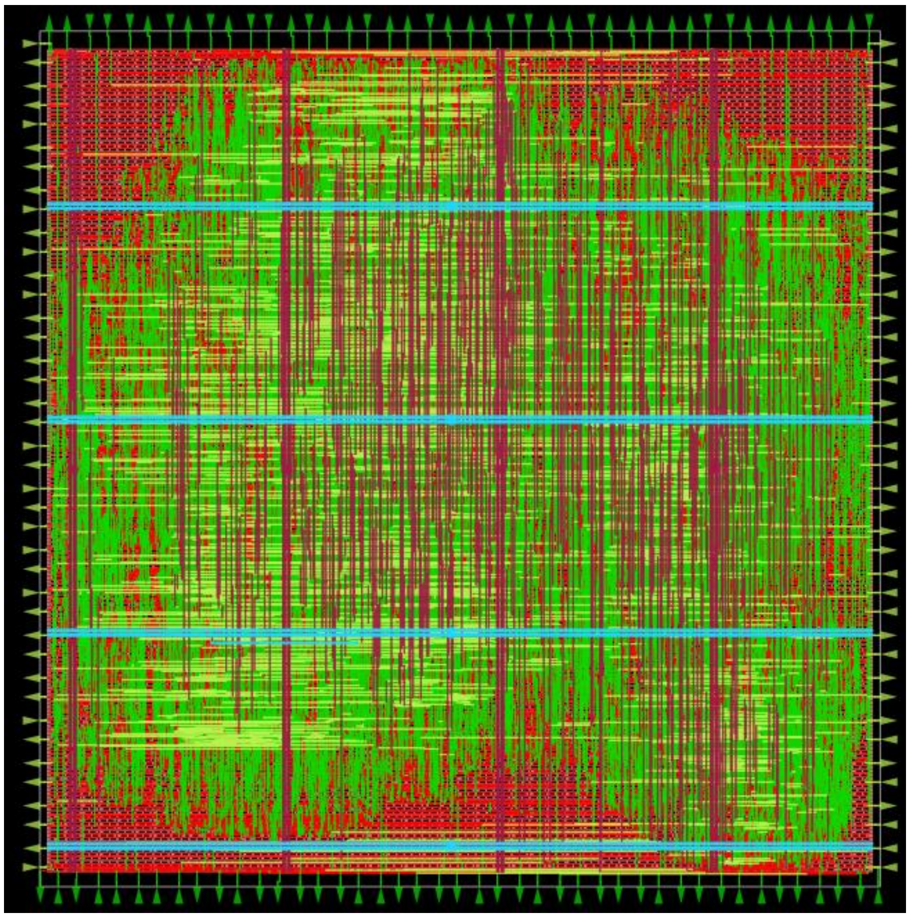


Figure 56: Post place and route layout.

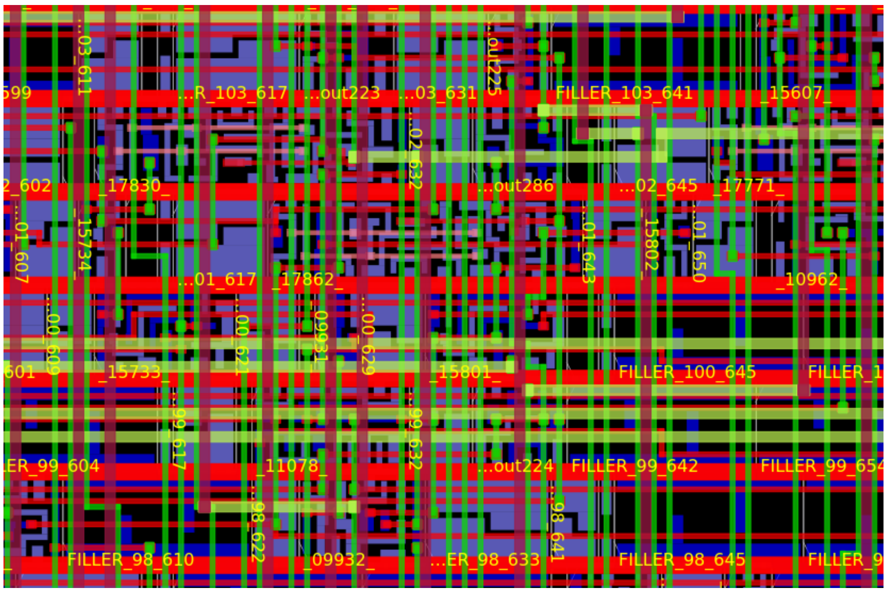


Figure 57: Zoomed layout with fillers and standart cells.

Project Files

In uploaded project files every part has its own name and OpenLane results are in the separate folder.

For readmemh and readmemb statements, their full path is used in the instruction_memory.v and data_memory.v, so for reproducing the given simulations correct path for test_instruction should be given in instruction_memory.v and data_memory.v. We did used the assembly codes given in the report and convert them to machine code with RARS (https://github.com/TheThirdOne/rars). We used more than one machine code for every part and we could not include every one of them as machine codes but their assembly codes are present in the report.