IPA Project: Team – 8

Design and Implementation of a RISC-V Processor in Verilog HDL

1. Introduction

This report presents the design details, features, and challenges encountered in the development of a RISC-V processor supporting a subset of instructions. The processor is designed to handle basic arithmetic, logical, memory, and branching operations. The architecture is tested using a set of predefined instructions to validate its functionality.

2. Processor Architecture

2.1 Sequential Implementation

The sequential implementation processes one instruction at a time, completing all five stages before fetching the next instruction. The five stages are:

2.1.1 Instruction Fetch (IF)

- The Program Counter (PC) fetches the instruction from memory.
- The instruction is passed to the next stage for decoding.
- The instruction Fetch module consists of two other modules, ‘PC adder’ and ‘Instruction memory’.
- The ‘PC adder’ module updates the PC value depending on the situations, like branching, or simply jumping to the next instruction in order.
- The ‘Instruction Memory’ module takes as input the PC value and gives as output the corresponding 32-bit instruction, fetching from the ‘instructions.txt’ file.

2.1.2 Instruction Decode (ID)

- The 32-bit instruction received from IF stage is decoded to extract the opcode, source, and destination registers.
- The Instruction Decode stage consists of three modules - ‘control’, ‘register file’, and ‘immediate gen’.

- The control unit generates the necessary control signals, depending on the 7-bit opcode.
- The register file is a set of registers, which takes as input the ‘reg_write’ signal, addressed of registers on which operations are to be done, and the register address on which data is to be written, and gives as output the values stored in the two source registers.
- The ‘immediate gen’ takes as input the 32-bit instruction, fetches the 12-bit input from the different indices of the instruction depending on the instruction type, and gives as output the sign extended 64-bit immediate.

2.1.3 Execution (EX)

- The ALU performs the required arithmetic or logical operation.
- The execute stage consists of the modules, ‘alu_control’ and ‘ALU’.
- The ‘alu_control’ takes as input the aluop, func3 and func7 and gives as output the 4 bit ‘alu_ctrl’ which tells the ALU what operation to perform on the incoming inputs.

2.1.4 Memory Access (MEM)

- Load (ld) and store (sd) instructions interact with data memory.
- The memory access stage consists of the module ‘data_memory’.
- The ‘data_memory’ updates the memory or fetches from the memory based on the control signals ‘mem_write’ and ‘mem_read’.

2.1.5 Write Back (WB)

- The result from the ALU or memory is written back to the register file.
- The write back stage is controlled by the signal ‘mem_read’.
- The next instruction begins execution after.

Since instructions are executed sequentially, there are no data hazards, but execution speed is limited by the single-instruction-at-a-time approach.

2.2 Pipelined Implementation

The pipelined implementation improves performance by overlapping the execution of multiple instructions using a five-stage pipeline:

2.2.1 Instruction Fetch (IF)

- The PC fetches an instruction from instruction memory.
- The instruction is stored in the IF/ID pipeline register for further processing.

- The PCWrite control signal determines whether PC adder should update the PC or stall it.

2.2.2 Instruction Decode (ID)

- The instruction is decoded to extract opcode, source, and destination registers.
- Register values are read from the register file.
- The control unit generates control signals for the execution stage.
- The hazard detection unit stalls the pipeline if a data hazard is detected.

2.2.3 Execution (EX)

- The ALU performs arithmetic and logical operations based on control signals.
- Forwarding unit handles data hazards by selecting correct input values for ALU.
- Branch target addresses are calculated for conditional jumps.

2.2.4 Memory Access (MEM)

- Load (ld) and store (sd) instructions access data memory.
- Memory hazards are managed by stalling or forwarding data.

2.2.5 Write Back (WB)

- The result from the ALU or memory is written back to the register file.
- The 'mem_to_reg' control signal determines whether the write-back should occur.

3. Supported Features

The processor supports the following operations:

- **Arithmetic Operations**: add, sub
- **Logical Operations**: and, or
- **Memory Operations**: ld, sd
- **Branching**: beq
- **Hazard Handling**:
    - Forwarding unit for resolving data hazards.
    - Hazard detection unit for stalling in case of load-use dependencies.
    - The branch prediction unit implements static predictions, always assuming beq to be false initially.

Forwarding:

- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

  } Fwd from EX/MEM pipeline reg

  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

  } Fwd from MEM/WB pipeline reg

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

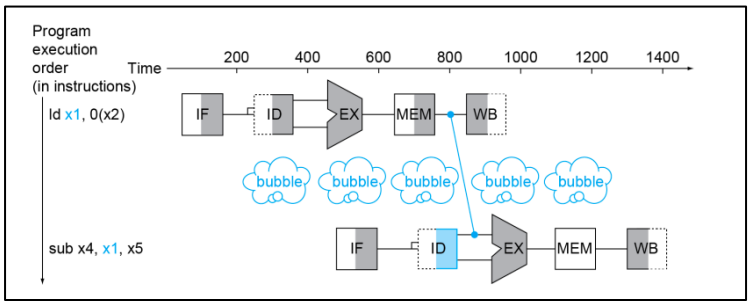
```
reg [63:0] forward_rs1;
reg [63:0] forward_rs2;

always @(*) begin
    if (forward_a == 2'b00)
        forward_rs1 = rs1_data_d2;
    else if (forward_a == 2'b10)
        forward_rs1 = alu_result_d3;
    else if (forward_a == 2'b01)
        forward_rs1 = write_data_d4;
    else
        forward_rs1 = rs1_data_d2;
end

always @(*) begin
    if (forward_b == 2'b00)
        forward_rs2 = rs2_data_d2;
    else if (forward_b == 2'b10)
        forward_rs2 = alu_result_d3;
    else if (forward_b == 2'b01)
        forward_rs2 = write_data_d4;
    else
        forward_rs2 = rs2_data_d2;
end
```

Hazard Detection:

We can't always avoid stalls by forwarding. If the required value is not computed when needed, we need to stall. We can't forward backward in time!



Branch Prediction:

In our pipelined implementation, branch prediction assumes that all branches are **not taken** by default. When a beq instruction enters the pipeline, the next instruction is fetched sequentially without waiting for the branch decision. The actual branch condition is evaluated in the **EX (Execution) stage**, where the ALU checks if the two source registers are equal. If the prediction was correct (branch not taken), execution continues normally. However, if the prediction was wrong (branch should be taken), all instructions fetched after the branch are **flushed** from the IF/ID, ID/EX and EX/MEM pipeline registers, and the PC is updated to the correct branch target. This results in a **control hazard**, introducing a stall as the pipeline refills from the correct address.

4. Simulation and Results

The processor was tested using an instruction sequence stored in instructions.txt. Simulation snapshots confirm correct execution of instructions and proper hazard management.

4.1 Sample Instruction Execution

4.1.1 Sequential

Instruction Set: The following instruction set was used to test the sequential implementation:

```
1. sd x2, 8(x0)
2. ld x3, 8(x0)
3. add x5, x1, x3
4. beq x0, x1, 8
5. beq x0, x0, 8
6. nop
7. nop
8. nop
9. add x8, x5, x7
(rest instructions are no operation instruction)
```

```
1  00000000001000000011010000100011
2  00000000100000000011000110000011
3  00000000001100001000001010110011
4  00000000000100000000010001100011
5  00000000000000000000010001100011
6  00000000000000000000000000000000
7  00000000000000000000000000000000
8  00000000000000000000000000000000
9  00000000011100101000010000110011
10 00000000000000000000000000000000
```

Initialisation: The register values and memory values were initialised as follows:

```
// test values initialization
registers[1] = 64'd1;
registers[2] = 64'd13;
registers[3] = 64'd3;
registers[4] = 64'd4;
registers[5] = 64'd5;
registers[6] = 64'd6;
registers[7] = 64'd7;
```

```
// initialize with some random test data

memory[0] = 64'd10;
memory[1] = 64'd20;
memory[2] = 64'd30;
memory[3] = 64'd40;
memory[4] = 64'd50;
```

Explanation

Let's see how the process is happening cycle by cycle through gtkwave.

We can clearly see that in each positive clock cycle, pc is getting updated and instruction is getting updated correspondingly.

We can see for the first instruction (sd x2, 8(x0)), rs1_addr = 0, rs2_addr = 2 which are correct, and rs1_data = 0, rs2_data = 13. Now we'll store this 13 in 8(x0) which is basically memory[1]. Also, for this instruction we can see that mem_write = 1 and alu_src = 1 and alu_op = 00, and immediate=8 which is representing the offset.

Now as the next positive clock cycle new pc becomes 4 so for second instruction (ld x3, 8(x0)), rs1_addr remains the same, since it's a load instruction we can see mem_read = 1, mem_to_reg = 1, reg_write = 1, since we are reading from memory (memory[1] = 13 here) and moving that value

into x3, we can see rd_addr = 3 and rd_data = 13 which is correct, also immediate = 8 which is representing the offset.

Now in the next positive clock cycle pc becomes 8 so for third instruction (add x5, x1, x3), we can see from gtkwave that registers address are correct and their data are also correct, now since, it is an add operation therefore reg_write = 1, alu_op = 10 and immediate = 0, we can see rd_data = 14 (rs1_data + rs2_data). Hence this instruction is also working perfectly.

Now in the next positive clock cycle pc becomes 12 so for fourth instruction (beq x0, x1, 8), we can see that rs1_addr = 0, rs2_addr = 1, which is correct. Branch =1 which is also correct. Value in x0 is 0, and value in x1 is 1, therefore the values are not same. So, alu_zero is 0 and no branching occurs, and next instruction(pc=pc+4) is executed.

Now in the next positive clock cycle pc becomes 16 so for fifth instruction (beq x0, x0, 8), we can see that rs1_addr = 0, rs2_addr = 0, which is corre ct. Branch =1 which is also correct. Value in x0 is 0, and value in x0 is 0, therefore the values are same. So, alu_zero is 1 and branching occurs. Immediate value is 8. As, branch and zero both are true in this instruction therefore now our pc should jump as pc+immediate*2 => (16 + 8*2 = 32) , and the 9th instruction instruction should run.

We can see in the next clock cycle pc is 32, and the instruction is (add x8, x5, x7). We can see that rs1_addr = 14, rs2_addr = 7, which is correct. and their data are also correct. Now since, it is an add operation therefore reg_write = 1, alu_op = 10 and immediate = 0, we can see rd_data = 21 (rs1_data + rs2_data). Hence this instruction is also working perfectly.

Hence, we can conclude that our **sequential implementation is working perfectly**!

Final Register Values

```
x0:                    0
x1:                    1
x2:                   13
x3:                   13
x4:                    4
x5:                   14
x6:                    6
x7:                    7
x8:                   21
x9:                    0
```

Final Memory Values

```
memory[0]:            10
memory[1]:            13
memory[2]:            30
memory[3]:            40
memory[4]:            50
```

From above register and memory values printed in the terminal, we can conclude our sequential implementation is working perfectly.

4.1.2 Pipeline

Initialisation: The register values and memory values were initialised as follows:

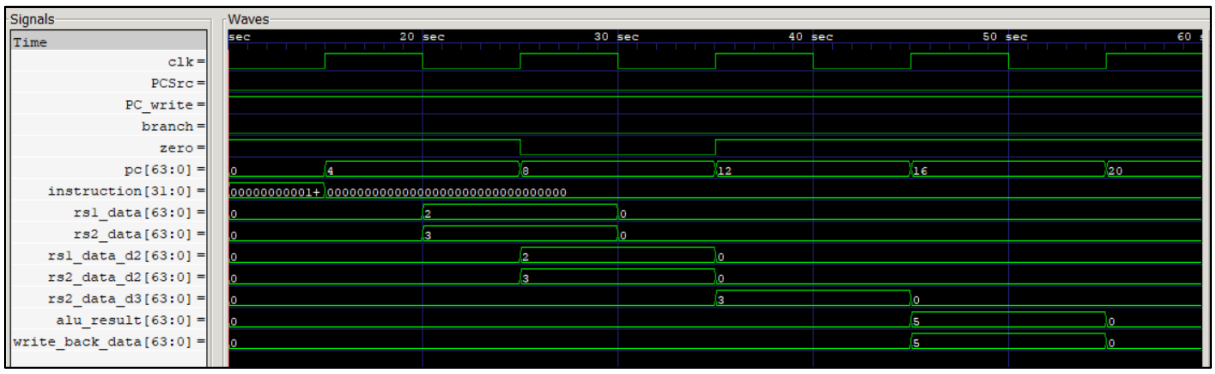
```
// test values initialization
registers[1] = 64'd1;
registers[2] = 64'd2;
registers[3] = 64'd3;
registers[4] = 64'd4;
registers[5] = 64'd5;
registers[6] = 64'd6;
registers[7] = 64'd7;
registers[8] = 64'd8;
```

```
// initialize with some random test data

memory[0] = 64'd10;
memory[1] = 64'd20;
memory[2] = 64'd30;
memory[3] = 64'd40;
memory[4] = 64'd50;
```

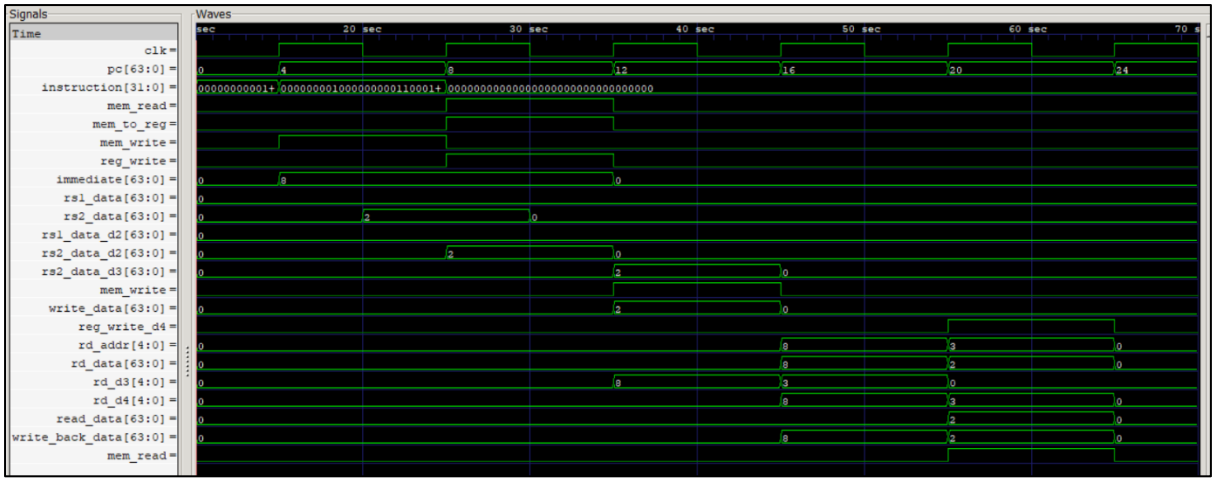
The following instruction sets were used to test the sequential implementation:

Instruction Set – 1: add x1 x2 x3
(basic add operation)



As we can see, the value in x1 gets updated to the addition (5) of the values stored in x2 and x3 (2 and 3 respectively). Thus, our **ADD instruction is working correctly.**
Similarly, we have verified that **SUB, AND, OR instructions are also working**.

Instruction Set – 2: sd x2 8(x0)
ld x3 8(x0)
(basic load and store operations)



The final register and memory values can be seen as follows:

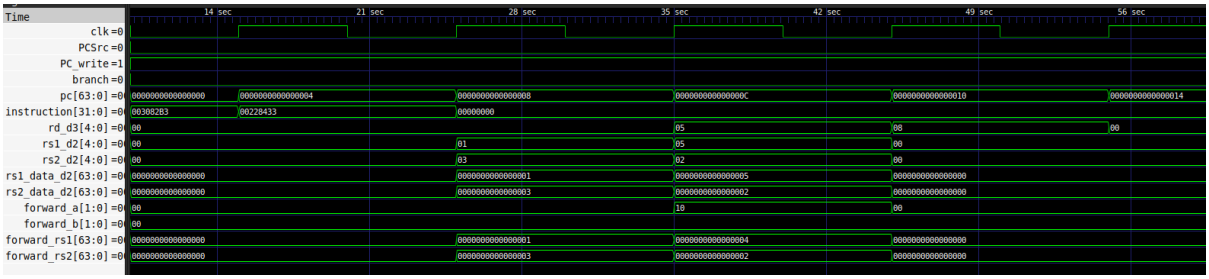
```
x0:                0
x1:                1
x2:                2
x3:                2
x4:                4
x5:                5
x6:                6
x7:                7
x8:                0
x9:                0
```

```
memory[0]:        10
memory[1]:         2
memory[2]:        30
memory[3]:        40
memory[4]:        50
```

This confirms that our **load and store operations are working correctly**.

Instruction Set – 3(a): add x5 x1 x3
add x8 x5 x2

(checking how data hazard is handled using forwarding (10): forwarding from ex/mem to id/ex)



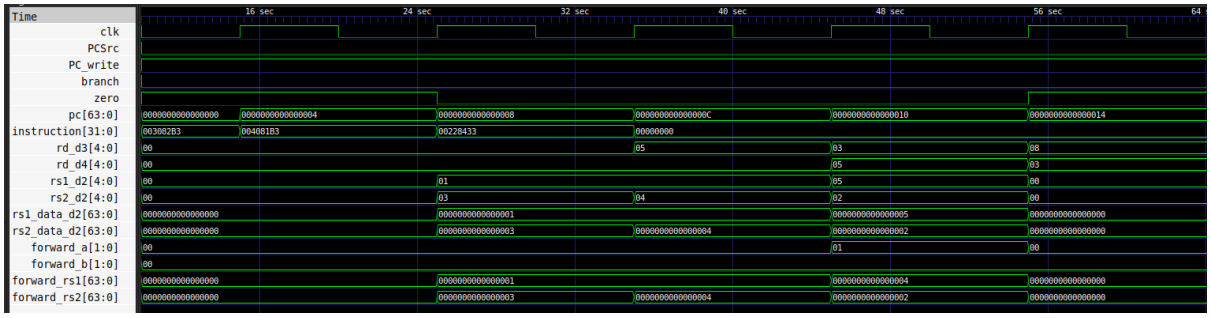
As we can see, rd_d3(which is the destination register after the ex/mem register) is same as rs1_d2(the source register after the id/ex register). So, forward_a becomes ‘10’ meaning we need to forward this value from the output of ex/mem pipeline to the execute stage so that the new value of x5(1+3=4) is used for the calculation of x8 which should now be 4+2=6.

Final register values:

```
x0:                    0
x1:                    1
x2:                    2
x3:                    3
x4:                    4
x5:                    4
x6:                    6
x7:                    7
x8:                    6
x9:                    0
```

Instruction Set – 3(b): add x5 x1 x3
add x3 x1 x4
add x8 x5 x2

(checking how data hazard is handled using forwarding (01): forwarding from mem/wb to id/ex)



As we can see, rd_d4(which is the destination register after the mem/wb register) is same as rs1_d2(the source register after the id/ex register). So, forward_a becomes '01' meaning we need to forward this value from the output of mem/wb pipeline to the execute stage so that the new value of x5(1+3=4) is used for the calculation of x8 which should now be 4+2=6.

Final register values:

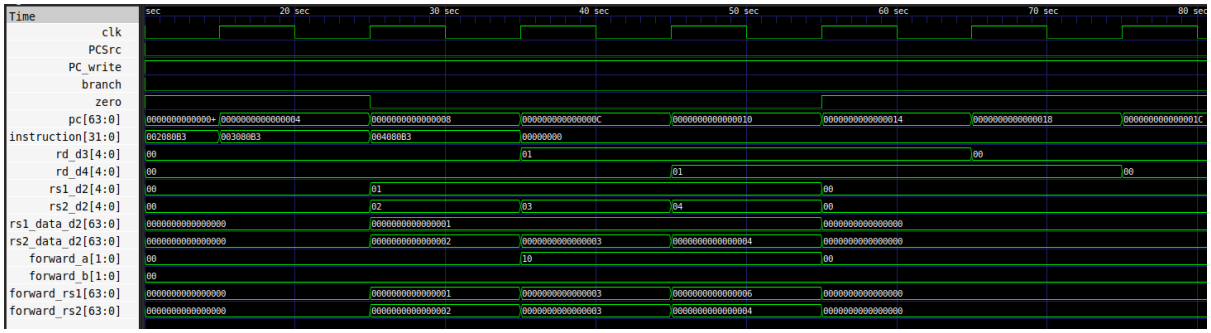
```
x1:                    1
x2:                    2
x3:                    5
x4:                    4
x5:                    4
x6:                    6
x7:                    7
x8:                    6
x9:                    0
```

This confirms that our **forwarding unit is working correctly.**

Instruction Set – 4: add x1 x1 x2
add x1 x1 x3

add x1 x1 x4

(checking how double data hazard is handled using forwarding)



While implementing the second instruction, we can see that we need to forward the updated value of x1 from first stage. So, forward is '10' and the updated value of source register x1 is 1+2=3 for the second instruction. Now, for the third instruction, the source register 1 is equal to both the output destination register of ex/mem pipeline and mem/wb pipeline. Priority is given to the output of the ex/mem pipeline, because it is the recently updated value. So, the value for x1 for third instruction is 3+3=6. Now when third instruction is complete, the updated x1 value is 6+4=10.

Final register values:

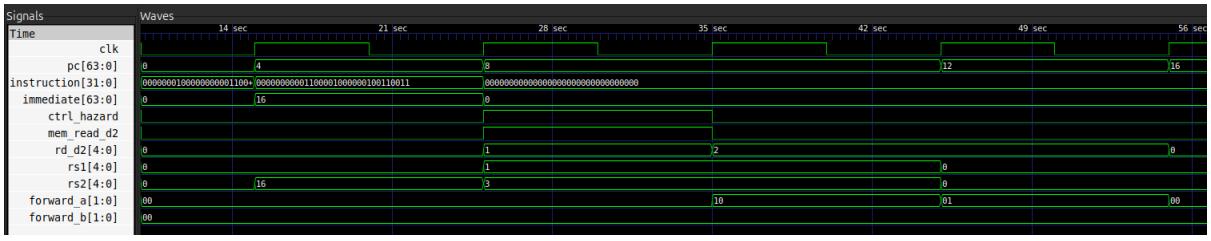
```
x0:                0
x1:               10
x2:                2
x3:                3
x4:                4
x5:                5
x6:                6
x7:                7
x8:                0
x9:                0
```

This confirms that **double data hazard is handled correctly using forwarding.**

Instruction Set – 5: ld x1, 16(x0)

add x2, x1, x3

(to check how load-data hazard is been handled by stalling {done by hazard detection unit} and forwarding {done by forwarding unit}).



The first instruction (ld x1, 16(x0)) will run without any problem. But while running second instruction load-data hazard will occur as we need

updated data of x1 from 16(x0) i.e., memory[2] which is not there, hence we need to stall then forward.
We can see from the gtkwave above that ctrl_hazard = 1 when load-data hazard condition (mem_read = 1 and id/ex rd = if/id rs) and stalling is happening we can see as pc didn't update during that time. And next forwardA is 01 representing the need of forward from mem/wb stage to id/ex stage. We can see forwardA is 10 as well but it is when stalling is happening so it doesn't matter because during stalling if/id and id/ex register will contain same corresponding values as if/id won't update then.

Final Register Values:

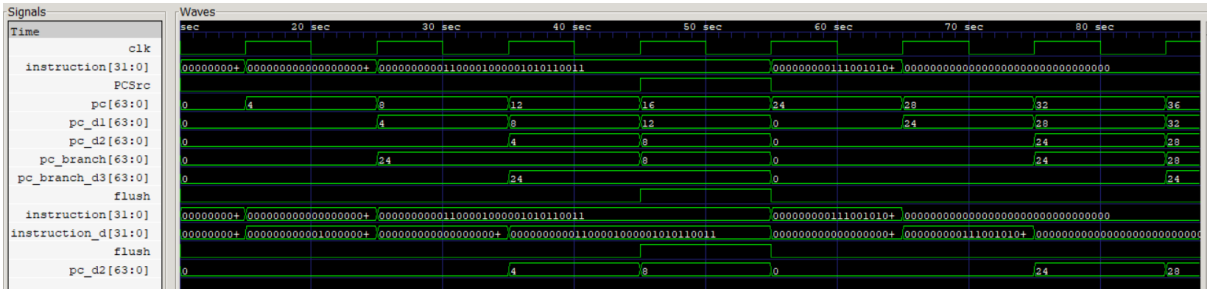
```
x0:                0
x1:               30
x2:               33
x3:                3
x4:                4
x5:                5
x6:                6
x7:                7
x8:                0
x9:                0
```

We can see the result is correct. Hence, **hazard detection unit is working perfectly.**

Instruction Set – 6:

```
beq x0, x1, 12
beq x0, x0, 10
add x5, x1, x3
add x5, x1, x3
add x5, x1, x3
add x5, x1, x3
add x8, x5, x7
```

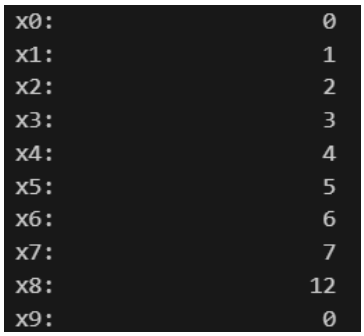
(basic branching operation that shows branch prediction and flushing when prediction is wrong)



In our pipelined implementation with branch prediction assuming all branches as not taken, the processor initially fetches and executes instructions sequentially. The first instruction, beq x0, x1, 12, is predicted

as not taken, so execution continues with beq x0, x0, 10. However, since x0 is always 0, this branch condition is true, meaning the processor must branch. Since it incorrectly predicted not to branch, it will have already fetched and started executing the subsequent add instructions. Once the branch is resolved in the EX stage, **the incorrect instructions in the pipeline are flushed**, and the PC is updated to the correct branch target (PC + 2*10). The flush logic is implemented using the PCSrc signal, which tells whether to branch or not. When PCSrc is high, the IF/ID, ID/EX, EX/MEM registers are flushed, PC is then updated, and then new instructions are fetched. The pipeline then resumes fetching from the correct address the instruction add x8, x5, x7, thus updating x8=5+7=12 finally.

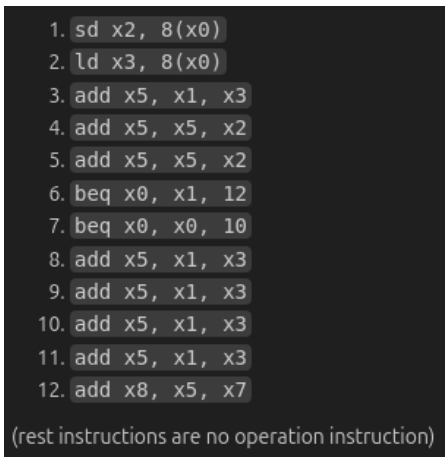
This confirms that our **branch prediction is working correctly**.

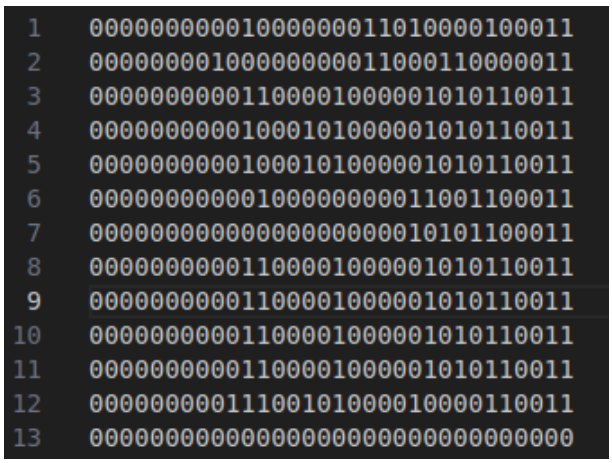


(updated x8 to 12)

FINAL INSTRUCTION SET:

Below is the final instruction set containing all types of instructions and hazards.





Explanation:

In instruction-2 we are loading memory[1] to x3, and then we are using z3 again in the next instruction before it is updated in the register memory. Hence, load-data hazard will occur. Which will be fixed by 1 stall then

forwarding. From line 3 to line 5, double-data hazard is happening which is also been handled in our implementation. Now in line 6 and line 7 branch control hazard will occur, which will be handled through static branch prediction i.e., we are predicting that branch result is always false and pc should update normally, but after 3 cycles we'll know whether our prediction was correct or not, if it was wrong, we'll flush previous ran instruction and jump to target address.

Hence, in this instruction all hazards are being handled.

The final result should be such that value of x8 becomes 14. Below is the gtkwave snapshot -

As we can see final write_back_data = 14 and rd_addr = 8 i.e., x8. Hence, x8 becomes 14.

Below are the final updated register file and memory file -

```
x0:                    0
x1:                    1
x2:                    2
x3:                    2
x4:                    4
x5:                    7
x6:                    6
x7:                    7
x8:                   14
x9:                    0
```

```
memory[0]:            10
memory[1]:             2
memory[2]:            30
memory[3]:            40
memory[4]:            50
```

Hence, we can say that our **pipeline implementation of RISC-V architecture is working properly.**

5. Challenges Encountered

Control Hazard Issues

- Tried implementing dynamic branch prediction (2 bit prediction), but were facing issues in updating the counters.

- We switched to static implementation for branch prediction assuming all branches as not taken.

Memory Accessing Issues

- We were having problem on deciding how to access the memory, where and how to store the base address. But then we figured out the correct way to implement it.

Write Back Issues

- We were having problem on basic wiring issues of write back stage to decode stage in pipeling implementation.

Clock Edge Issues

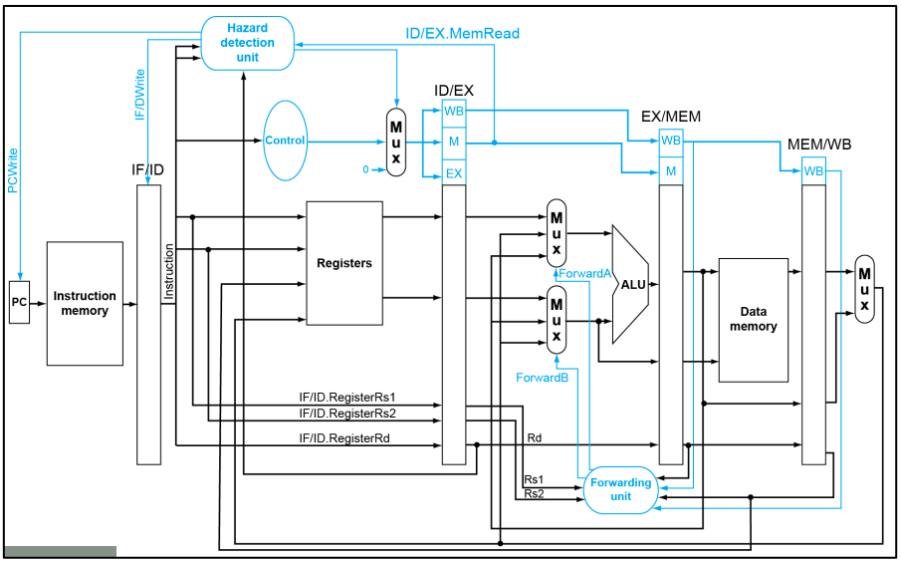
- We faced problem on in which clock edge should we do read operation and in which clock edge should we do write operation, then figured out to do read operation in -ve clock edge and write operation in +ve clock edge.

Assembly to Machine Converting Issues

- We tried to implement something which will take assembly code then convert it to machine code and then process it, but were unable to do it due to time constraints.

6. Conclusion

At the end we can say that our implementation of RISC-V ISA is working perfectly. We are able to handle all kind of hazards – Data Hazard, Double Data Hazard, Load-Data Hazard and at last Control Hazard (due to branch)



7. Contributions of Team Members

- **Deepak Pandey (2023102053):**

  Worked on how offset is working in the implementation; how we are accessing data from memory and register; how branching offset is working; implemented basic modules like – control unit, immediate generator, register file, memory access; did data hazard part i.e., forwarding unit; implemented pipeline register – if/id and id/ex.

- **Manas Inamdar (2023102052):**

  Worked on pc adder implementation; ALU control implementation, ex/mem pipeline register implementation, contributed in the final connection of pipelines and different modules, and generating instructions for test cases.

- **Gautam Gandhi (2023102059):**

  Implemented the instruction memory – fetching instructions from txt file and storing them in instruction memory; worked on the execute stage – created the ALU module; implemented the hazard detection unit; implemented the mem/wb pipeline register; contributed in the connection of different pipeline registers; also implemented control hazards including branch prediction and flushing of instructions when prediction is wrong.