IPA PROJECT-2025

Gandlur Valli
2023102068
IIIT Hyderabad
Hyderabad, India
gandlur.valli@students.iiit.ac.in

Priyanshi Jain
2023112021
IIIT Hyderabad
Hyderabad, India
priyanshi.jain@research.iiit.ac.in

Ritama Sanyal
2023112027
IIIT Hyderabad
Hyderabad, India
ritama.sanyal@research.iiit.ac.in

I. Aim

This project aims to design and implement a RISC-V-based processor using Verilog, focusing on a five-stage pipelined architecture to improve instruction throughput. The goal is to support fundamental RISC-V instructions (add, sub, and, or, ld, sd, beq) while addressing pipeline hazards like data and control hazards. Each pipeline stage is implemented as an independent module and verified before integration. A test bench is developed to validate the processor's correctness and efficiency. The project aims to demonstrate the benefits of pipelining and serve as a foundation for educational and experimental RISC-V processor designs.

II. Introduction

Modern processor architectures rely on pipelining to enhance performance. The RISC-V Instruction Set Architecture (ISA) provides a simplified, open-source design that allows for efficient implementation of both sequential and pipelined processor architectures. To achieve high instruction throughput while minimizing hazards, a well-structured pipeline with effective forwarding and hazard resolution mechanisms is essential.

This project focuses on the design and implementation of a RISC-V processor using Verilog, with two primary objectives:

1. Sequential Processor Implementation – A baseline design that executes one instruction at a time.

2. Pipelined Processor Implementation – A five-stage pipeline design that improves execution efficiency by eliminating structural, data, and control hazards.

The processor supports fundamental RISC-V instructions, including arithmetic (add, sub), logical (and, or), memory (ld, sd), and control (beq) operations. A modular approach is adopted, where each stage—Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB)—is designed as a separate module and tested independently before integration.

III. Instruction Fetch

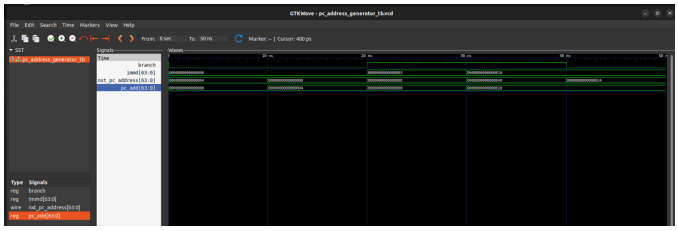
A. PC

The PC is a crucial component in a processor's instruction fetch stage. Its primary purpose is to determine the address of the next instruction to be executed. Typically, this involves incrementing the current PC value by a fixed amount (4 bytes) to point to the next instruction in memory. When the processor needs to execute a branch or jump instruction, this block calculates the target address. It combines the current PC value with an offset (immediate value) to determine where the program should jump.

Target Address $= \text{PC} + 2 * \text{immediate}$

Based on control signals, the block decides whether to use the sequential address or the calculated branch/jump address as the next PC value (there is a flag to do so).

```
VCD info: dumpfile pc_address_generator_tb.vcd opened for output.
Branch=0 | PC=0000000000000000 | Immediate=0000000000000000 | Next PC=0000000000000004
Branch=0 | PC=0000000000000004 | Immediate=0000000000000000 | Next PC=0000000000000008
Branch=1 | PC=0000000000000008 | Immediate=0000000000000003 | Next PC=000000000000000e
Branch=1 | PC=0000000000000020 | Immediate=0000000000000010 | Next PC=0000000000000040
Branch=0 | PC=0000000000000020 | Immediate=0000000000000010 | Next PC=0000000000000024
```



B. Instruction Memory

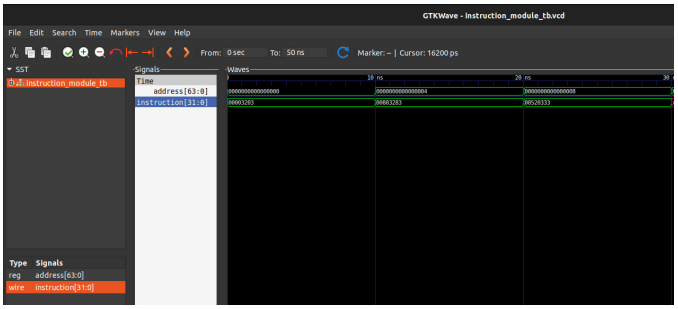
The Instruction Memory block is responsible for storing and supplying instructions to the processor. The Instruction Memory contains the program's instructions, which are fetched sequentially or based on control flow. The Program Counter (PC) points to the address of the next instruction to be fetched. The PC value is sent to the Instruction Memory. The Instruction Memory then retrieves the corresponding instruction and forwards it to the next stage, which is Instruction Decode.

```
VCD info: dumpfile instruction_module_tb.vcd opened for output.
Address=                  0, Instruction=00003203
Address=                  4, Instruction=00803283
Address=                  8, Instruction=00520333
```

IV. Instruction Decode

A. Register File

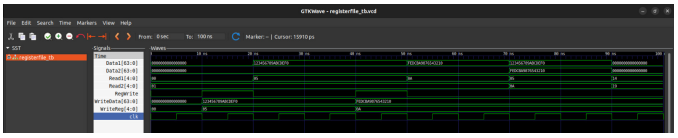
Contains general-purpose registers used by instructions.



The instruction specifies two source registers (Read register 1 and Read register 2), whose values are read and passed as Read data 1 and Read data 2.

If the instruction involves writing back a result (e.g., arithmetic or load instructions), the destination register is identified (Write register).

```
VCD info: dumpfile registerfile_tb.vcd opened for output.
Write to Reg5: 123456789abcdef0, Read from Reg5: 123456789abcdef0
Write to Reg10: fedcba9876543210, Read from Reg10: fedcba9876543210
Read Reg5: 123456789abcdef0, Read Reg10: fedcba9876543210
Read Reg20: 0000000000000000, Read Reg25: 0000000000000000 (Expecting 0s)
```



B. Immediate Generator

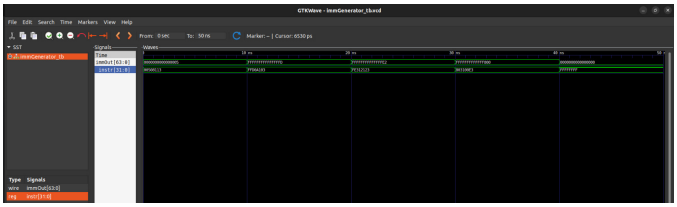
It is responsible for extracting and processing immediate values from the instruction. These immediate values are used in various operations, such as arithmetic, memory addressing, and branching. The instruction word contains specific bits that represent immediate values. The Immediate Generator extracts these fields based on the instruction format. After extraction, the immediate value is extended to 64 bits to match the word size of the processor. For certain instruction types (e.g., S-type or B-type), multiple fields from the instruction need to be concatenated or rearranged to form the complete immediate value. The generated immediate value is then forwarded to the subsequent stages.

```
VCD info: dumpfile immGenerator_tb.vcd opened for output.
I-Type: Instr=00000000010100001000000100010011, Immediate=0000000000000005
Load: Instr=11111111110100001010000100000011, Immediate=fffffffffffffffd
S-Type: Instr=11111110001100010010000100100011, Immediate=fffffffffffffffe2
B-Type: Instr=10000000001100010000000011100011, Immediate=ffffffffffffff800
Default: Instr=11111111111111111111111111111111, Immediate=0000000000000000
```

V. Execute

A. ALU Control

It is responsible for generating the appropriate ALU control signals based on the instruction type and specific function codes. It's a crucial component in a RISC-V processor's execution unit.



opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	XXXXXXXXXXX	add	0010
sd	00	store register	XXXXXXXXXXX	add	0010
beq	01	branch on equal	XXXXXXXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

Fig. 1. ALU Control

B. ALU

ALU is a fundamental component in digital systems, particularly in CPUs. It performs arithmetic and logical operations on binary numbers. This specific ALU is designed to handle 64-bit signed numbers and can perform four basic operations: AND, OR, addition, and subtraction.

The ALU includes logic to detect overflow conditions for addition and subtraction. Overflow occurs when the result of an operation is too large to be represented in the 64-bit signed format. The detection method differs for addition and subtraction:

For addition, overflow is said to occur when adding two numbers with the same sign produces a result with the opposite sign.

For subtraction, overflow occurs when subtracting a number from another with a different sign produces a result with a sign different from the first number.

The zero flag is set when the result of any operation is zero. This is useful in many computational scenarios, especially in comparison operations.

The zero flag is used to make decisions in conditional jump or branch instructions. For example, a "branch if zero" instruction uses this flag to determine whether to change the program counter.

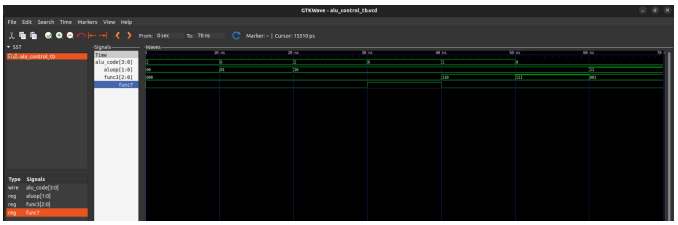
C. Immediate

In processors, this operation is commonly used in address calculations, such as determining the target address for branch instructions. For example, branch offsets are often word-aligned, meaning they are multiples of 4 bytes. Shifting left by 1 is part of aligning the offset for such calculations. Then, add the pc address to this value to the value. This gives the branch address.

VI. Memory Access

This stage is responsible for reading from and writing to memory during the execution of load and store instructions.

```
VCD info: dumpfile alu_control_tb.vcd opened for output.
aluop: 00, func3: 000, func7: 0 => alu_code: 0010
aluop: 01, func3: 000, func7: 0 => alu_code: 0110
aluop: 10, func3: 000, func7: 0 => alu_code: 0010
aluop: 10, func3: 000, func7: 1 => alu_code: 0110
aluop: 10, func3: 110, func7: 0 => alu_code: 0001
aluop: 10, func3: 111, func7: 0 => alu_code: 0000
aluop: 11, func3: 001, func7: 0 => alu_code: 0000 (Default case)
```



It has two main operations:

MemRead: Reads data from a specified memory address.

MemWrite: Writes data to a specified memory address.

For example:

During a load instruction:

The ALU computes the effective memory address using base address and offset.

The MemRead signal is activated, enabling data retrieval from the computed address.

The retrieved data is passed through the multiplexer (if MemtoReg is active) and written back into a register.

During a store instruction (e.g., sw in RISC-V):

The ALU computes the effective memory address.

The MemWrite signal is activated, enabling data to be written into the specified address. The Write Data input provides this data, sourced from a register.

VII. Write Back

The write-back stage is the final step in the instruction execution process. It ensures that the results of computations or data fetched from memory are written back to the appropriate register for future use. The multiplexer at the write-back stage determines the source of data to be written back to the registers.

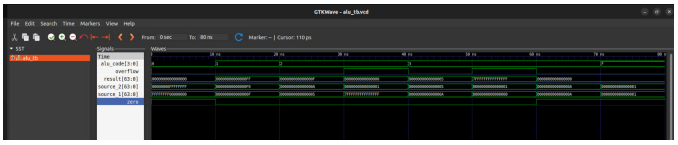
It selects between:

ALU Result: The output of the ALU, which is used for arithmetic or logical instructions (e.g., addition, subtraction).

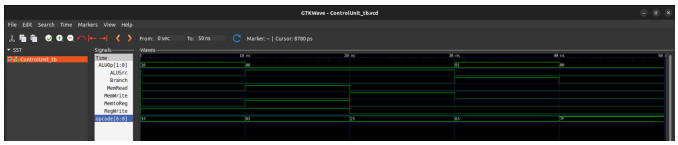
Read Data: The data fetched from memory during load instructions.

The selected data is written into one of the registers in this block.

```
VCD info: dumpfile alu_tb.vcd opened for output.
AND: ffffffff00000000 & 00000000ffffffff = 0000000000000000, Zero: 1, Overflow: 0
OR: 00000000000000ff | 00000000000000f0 = 00000000000000ff, Zero: 0, Overflow: 0
ADD:                  5 +                  10 =                  15, Zero: 0, Overflow: 0
ADD Overflow:  9223372036854775807 +                   1 = -9223372036854775808, Zero: 0, Overflow: 1
SUB:                 10 -                   5 =                   5, Zero: 0, Overflow: 0
SUB Overflow: -9223372036854775808 -                   1 =  9223372036854775807, Zero: 0, Overflow: 1
Zero Flag Test:                  10 -                  10 =                   0, Zero: 1, Overflow: 0
Default Case: ALU Code = 1111, Result = 0000000000000000, Zero: 1, Overflow: 0
```



```
VCD info: dumpfile ControlUnit_tb.vcd opened for output.
R-Type: opcode=0110011 | ALUSrc=0, MemtoReg=0, RegWrite=1, MemRead=0, MemWrite=0, Branch=0, ALUOp=10
Load: opcode=0000011 | ALUSrc=1, MemtoReg=1, RegWrite=1, MemRead=1, MemWrite=0, Branch=0, ALUOp=00
Store: opcode=0100011 | ALUSrc=1, MemtoReg=0, RegWrite=0, MemRead=0, MemWrite=1, Branch=0, ALUOp=00
Branch: opcode=1100011 | ALUSrc=0, MemtoReg=0, RegWrite=0, MemRead=0, MemWrite=0, Branch=1, ALUOp=01
Default: opcode=1111111 | ALUSrc=0, MemtoReg=0, RegWrite=0, MemRead=0, MemWrite=0, Branch=0, ALUOp=00
```



The specific register to be updated is identified by the Write Register input, which comes from the instruction's destination field.

The RegWrite control signal enables or disables writing to the registers.

VIII. Sequential Logic

The datapath with control for the sequential logic is shown in Fig.2.

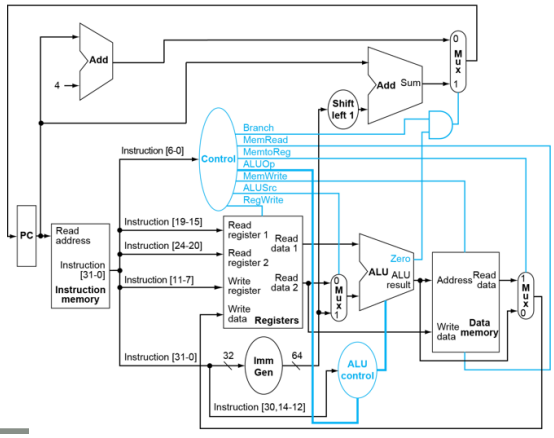


Fig. 2. Sequential Datapath

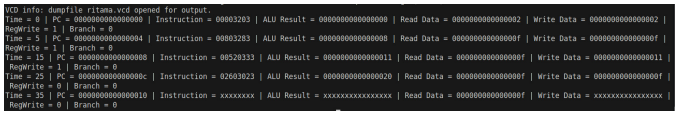


Fig. 3. Final output

A. Test Case

```
lw x4, 8(x6)     # Load word from memory
add x4, x6, x8   # Add registers x6 and x8, store i
lw x6, 5(x4)     # Load word from memory
add x6, x4, x6   # Add registers x4 and x6, store i
```

B. Test Case Breakdown

1. Load Word: `lw x4, 8(x6)`:

- **IF (Instruction Fetch)**: Fetch from memory.
  - **Blocks:** Instruction Memory, PC.
- **ID (Instruction Decode)**: Decode, read register `x6`.

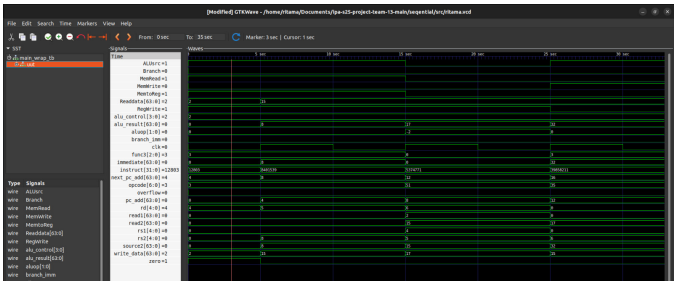


Fig. 4. Final gtkwave

  - **Blocks:** Control Unit, Registers, Imm Gen.
- **EX (Execute)**: Compute effective address `x6 + 8`.
  - **Blocks:** ALU, ALU Control.
- **MEM (Memory Access)**: Load value from calculated address.
  - **Blocks:** Data Memory.
- **WB (Write Back)**: Write loaded value to `x4`.
  - **Blocks:** Registers.

2. Addition: `add x4, x6, x8:`

- **IF**: Fetch from memory.
- **ID**: Decode, read `x6` and `x8`.
- **EX**: Compute `x6 + x8` using ALU.
- **MEM**: No memory access.
- **WB**: Write result to `x4`.

3. Load Word: `lw x6, 5(x4):`

- **IF**: Fetch from memory.
- **ID**: Decode, read `x4`.
- **EX**: Compute `x4 + 5`.
- **MEM**: Load value from calculated address.
- **WB**: Write to `x6`.

4. Addition: `add x6, x4, x6:`

- **IF**: Fetch from memory.
- **ID**: Decode, read `x4` and `x6`.
- **EX**: Compute `x4 + x6`.
- **MEM**: No memory access.
- **WB**: Write result to `x6`.

IX. Pipeline Logic

A. Introduction

Pipelining is a technique used in computer architecture to improve the efficiency and throughput of a processor by overlapping the execution of multiple instructions. It executes multiple instructions in parallel. Each instruction has the same latency.

B. Why pipelining?

Without pipelining, a processor executes one instruction at a time, completing all stages (fetch, decode, execute, memory access, write back) sequentially before starting the next instruction. This results in idle components during certain stages, leading to inefficiencies and delays.

With pipelining, multiple instructions are overlapped in execution, with different stages of the pipeline working simultaneously on different instructions. This parallelism ensures that the CPU components are utilized efficiently, reducing the delay between completed instructions and increasing the number of instructions executed per unit time. By breaking down complex operations into simpler steps distributed across pipeline stages, pipelining minimizes the processor's cycle time and maximizes throughput

C. Stages of Pipelining

There are five stages, namely:

- IF: Instruction fetch from memory
- ID: Instruction decode and register read
- EX: Execute operation or calculate address
- MEM: Access memory operand
- WB: Write result back to register

D. Pipeline Register

There is a need for registers between stages to hold information produced in previous cycle. Since we are implementing a five stage processor, we need 4 registers.

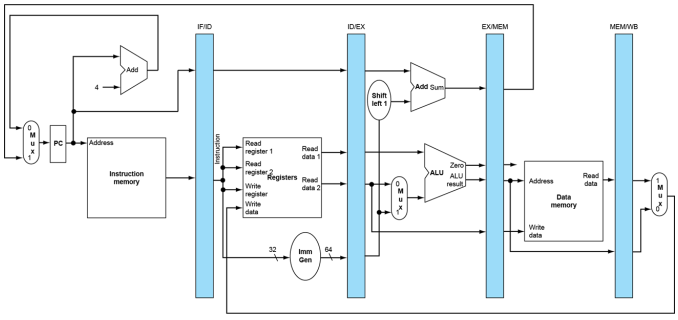


Fig. 5. Pipeline Registers

E. Data Hazard

Data hazards occur in pipelined processors when the execution of an instruction depends on the result of a previous instruction that is still in progress. These hazards can lead to incorrect computations or delays in execution if not properly managed. For example:

```
sub   x2, x1,x3
and   x12,x2,x5
or    x13,x6,x2
add   x14,x2,x2
sd    x15,100(x2)
```

Before x2 is written back in the sub instruction, it is being used in the 'and' instruction. This results in incorrect output. So, we use FORWARDING to fix this issue.

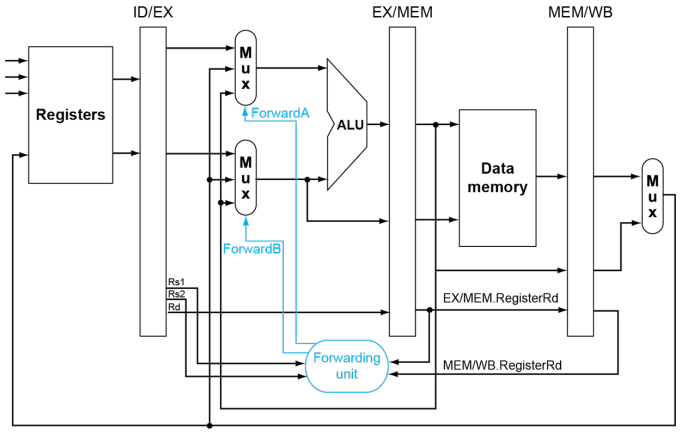


Fig. 6. Forwarding Paths

F. Forwarding Paths

Forwarding paths are used to resolve data hazards during instruction execution. Forwarding (or bypassing) is a technique employed to reduce stalls caused by dependencies between instructions. Below is an explanation of the forwarding paths depicted:

1) ***ForwardA Path****:*

- This path forwards data from the EX/MEM pipeline register (output of the ALU in the EX stage) directly to one of the inputs of the ALU in the current cycle.
- It is used when an instruction in the EX stage requires data that has been computed in the previous cycle but has not yet been written back to the register file.
- The forwarding unit detects that the destination register (Rd) of an instruction in EX/MEM matches the source register (Rs1) of the instruction in ID/EX and enables this path.

2) ***ForwardB Path****:*

- Similar to ForwardA, this path forwards data from the EX/MEM pipeline register or MEM/WB pipeline register directly to another input of the ALU in the current cycle.
- It is used when an instruction in the EX stage requires data that is either being computed in EX/MEM or is available after memory access in MEM/WB.
- The forwarding unit checks if the destination register (Rd) of an instruction matches the source register (Rs2) of another instruction and enables this path.

G. Control Hazards

A **control hazard** (or **branch hazard**) occurs in RISC-V pipelined architectures when the processor encounters a control instruction (such as a branch or jump) that affects the program flow. Since these instructions determine the next instruction to be fetched, any delay in resolving the branch decision can lead to pipeline stalls or incorrect instruction execution.

1) Causes of Control Hazards in RISC-V: Control hazards in RISC-V arise due to:

- **Branch Instructions** (e.g., `BEQ`, `BNE`, `BLT`, `BGE`): These instructions conditionally change the control flow based on register comparisons.
- **Jump Instructions** (e.g., `JAL`, `JALR`): These instructions unconditionally alter the program counter (PC).
- **Pipeline Depth**: In a classic 5-stage RISC-V pipeline (IF, ID, EX, MEM, WB), branch resolution typically happens in the Execution (EX) stage, causing a delay in knowing the correct target address.

2) Effects of Control Hazards: When a control hazard occurs in RISC-V, the pipeline may fetch incorrect instructions, leading to:

- **Pipeline Stalls**: The pipeline may need to halt instruction execution until the correct branch decision is made.
- **Flushing Mispredicted Instructions**: If an incorrect instruction is fetched due to a branch misprediction, it must be discarded.
- **Performance Degradation**: Unresolved control hazards increase the number of wasted cycles, reducing the overall instruction throughput.

H. Mitigation Techniques in RISC-V

To minimize control hazards, RISC-V processors use the following strategies:

- **Branch Prediction:** Modern RISC-V implementations may use dynamic branch prediction to guess the branch outcome and speculatively execute instructions.
- **Delayed Branching:** In some architectures, a **delay slot** is used, where an instruction after a branch is always executed before the jump takes effect.
- **Pipeline Flushing:** If a branch is mispredicted, the pipeline flushes the incorrect instructions and starts execution from the correct target.
- **Early Branch Resolution:** Some RISC-V designs resolve branch conditions earlier in the pipeline to reduce stalls.

X. TESTCASE

In a pipelined processor, five stages operate concurrently:

- **IF (Instruction Fetch)**: Retrieves an instruction from memory.
- **ID (Instruction Decode)**: Decodes the instruction and reads registers.
- **EX (Execute)**: Performs arithmetic operations or computes addresses.
- **MEM (Memory Access)**: Loads from or stores to memory.
- **WB (Write Back)**: Writes results back into the register file.

Each clock cycle a new instruction is fetched while previous instructions progress to subsequent stages. The following timeline explains the operation of each pipeline block over time.

A. *Test Case Explanation*

1) Assembly Instructions: The following assembly instructions were executed to test the 5-stage RISC-V pipeline:

```
     ld x1, 0(x0)
     ld x2, 8(x0)
     ld x3, 16(x0)
     add x4, x1, x2
     sub x5, x4, x3
     or x6, x4, x5
     sd x6, 24(x0)
     ld x7, 24(x0)
     beq x7, x6, label
     add x8, x0, x0
label:
     add x9, x0, x0
```

2) Stage-by-Stage Execution: The following is a detailed breakdown of the pipeline behavior for each instruction:

- **1. ld x1, 0(x0)**:
  - **IF**: Fetches the `ld` instruction.
  - **ID**: Decodes the instruction and reads `x0` (always 0).
  - **EX**: Calculates the memory address (`0 + 0 = 0`).
  - **MEM**: Reads the value `5` from memory address `0`.
  - **WB**: Writes `5` into `x1`.
- **2. ld x2, 8(x0)**:
  - **IF**: Fetches the `ld` instruction.
  - **ID**: Decodes the instruction and reads `x0`.
  - **EX**: Calculates the memory address (`0 + 8 = 8`).
  - **MEM**: Reads the value `10` from memory address `8`.
  - **WB**: Writes `10` into `x2`.
- **3. ld x3, 16(x0)**:
  - **IF**: Fetches the `ld` instruction.
  - **ID**: Decodes the instruction and reads `x0`.
  - **EX**: Calculates the memory address (`0 + 16 = 16`).
  - **MEM**: Reads the value `15` from memory address `16`.
  - **WB**: Writes `15` into `x3`.
- **4. add x4, x1, x2**:
  - **IF**: Fetches the `add` instruction.
  - **ID**: Decodes the instruction and reads `x1` and `x2`.
  - **EX**: Performs the addition (`5 + 10 = 15`).
  - **MEM**: No memory access (pass-through).
  - **WB**: Writes `15` into `x4`.
- **5. sub x5, x4, x3**:
  - **IF**: Fetches the `sub` instruction.
  - **ID**: Decodes the instruction and reads `x4` and `x3`.
  - **EX**: Performs the subtraction (`15 - 15 = 0`).
  - **MEM**: No memory access (pass-through).
  - **WB**: Writes `0` into `x5`.
- **6. or x6, x4, x5**:
  - **IF**: Fetches the `or` instruction.
  - **ID**: Decodes the instruction and reads `x4` and `x5`.
  - **EX**: Performs the bitwise OR (`15 | 0 = 15`).
  - **MEM**: No memory access (pass-through).
  - **WB**: Writes `15` into `x6`.
- **7. sd x6, 24(x0)**:
  - **IF**: Fetches the `sd` instruction.
  - **ID**: Decodes the instruction and reads `x6` and `x0`.
  - **EX**: Calculates the memory address (`0 + 24 = 24`).
  - **MEM**: Writes the value `15` into memory address `24`.
  - **WB**: No write-back (pass-through).
- **8. ld x7, 24(x0)**:
  - **IF**: Fetches the `ld` instruction.
  - **ID**: Decodes the instruction and reads `x0`.
  - **EX**: Calculates the memory address (`0 + 24 = 24`).
  - **MEM**: Reads the value `15` from memory address `24`.
  - **WB**: Writes `15` into `x7`.
- **9. beq x7, x6, label**:
  - **IF**: Fetches the `beq` instruction.
  - **ID**: Decodes the instruction and reads `x7` and `x6`.
  - **EX**: Compares `x7` and `x6` (`15 == 15`). Branch is taken.
  - **MEM**: No memory access (pass-through).
  - **WB**: No write-back (pass-through).
- **10. add x8, x0, x0**:
  - **IF**: Fetches the `add` instruction (skipped due to branch).
  - **ID**: Decodes the instruction (skipped).
  - **EX**: No operation (skipped).
  - **MEM**: No operation (skipped).
  - **WB**: No operation (skipped).
- **11. add x9, x0, x0**:
  - **IF**: Fetches the `add` instruction.
  - **ID**: Decodes the instruction and reads `x0`.
  - **EX**: Performs the addition (`0 + 0 = 0`).
  - **MEM**: No memory access (pass-through).
  - **WB**: Writes `0` into `x9`.

3) Data Hazards and Forwarding:

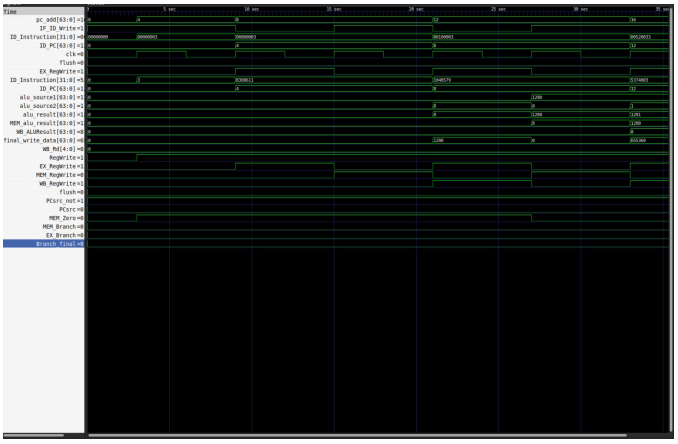
- **RAW Hazards**:
  - `add x4, x1, x2` → `sub x5, x4, x3`: Forwarding from EX/MEM to EX.
  - `sub x5, x4, x3` → `or x6, x4, x5`: Forwarding from MEM/WB to EX.
  - `sd x6, 24(x0)` → `ld x7, 24(x0)`: Forwarding from MEM/WB to EX.
- **Branch Handling**:
  - The `beq` instruction causes a branch to the `label`, skipping the `add x8, x0, x0` instruction.

4) Final Register and Memory State: The final register and memory states after executing the instructions are as follows:

- **Registers**:
  - `x1 = 5`
  - `x2 = 10`
  - `x3 = 15`
  - `x4 = 15`
  - `x5 = 0`
  - `x6 = 15`
  - `x7 = 15`

- `x8 = 0` (unchanged, branch skipped)
- `x9 = 0` (dummy instruction)

- **Memory**:
  - Address `0`: `5`
  - Address `8`: `10`
  - Address `16`: `15`
  - Address `24`: `15` (stor 'ed by `sd x6, 24(x0)`)



5) GTKWave Output:

XI. Challenges Faced

A. Pipeline Hazards

- Handling data hazards was very crucial for getting accurate results and we had to switch our logic multiple times for this.
- We added forwarding paths and stalling but even small mistakes caused incorrect results, leading to hours of debugging.

B. Instruction Synchronization

- Keeping data in sync across pipeline stages was way harder than expected.
- Misaligned or incorrectly initialized values caused errors in arithmetic and memory operations.

C. Memory Operations

- Implementing load and store instructions was not easy as it involved managing memory addresses while making sure everything is aligned.

D. Debugging Complexity

- Debugging the pipelined processor was the hardest among all. Whenever we tried fixing one error, a dozen more came up- errors came from any and every stage, and backtracking them required a ton of monitor print statements and simulation and waveform analysis using GTKWave.

References

1. RISC-V Lecture Slides
2. Computer Organization and Design-David A. Patterson, John L.Hennessy (RISC-V Edition)
3. LupLab: Open Source Gitlab based website used to get assembly codes from binary instructions

Contributions

This project was a collaborative effort, with each member contributing to different aspects of the RISC-V processor design:

- Gandlur Valli (2023102068): Worked on the Instruction Fetch and Instruction Decode stages, implementing the PC logic, instruction memory, and register file. Also contributed to debugging and testing.
- Priyanshi Jain (2023112021): Focused on the Execute and Memory Access stages, implementing ALU operations, control signals, and memory interactions. Assisted in verifying correct data flow across pipeline stages.
- Ritama Sanyal (2023112027): Developed the Write Back stage and Pipeline Integration, ensuring seamless data transfer between pipeline registers. Handled pipeline control logic and test case structuring.
- Pipelining Challenges (Collaborative Effort): Since pipelining was the most complex aspect, all three members worked together to implement hazard detection, forwarding, and stall handling to resolve data and control hazards.

Acknowledgments

We would like to extend our sincere gratitude to Prof. Deepak Gangadharan for his invaluable guidance throughout this project. His clear explanations of processor architecture during lectures, particularly pipelining and RISC-V instruction handling, helped us a lot in understanding the project and shaping our approach.

We are also deeply thankful to our Teaching Assistants for their patience, support, and timely assistance. They were always there to help us troubleshoot issues, refine our design, and provide constructive feedback.