IPA Project Report:

RISC-V Processor with Hazard Handling

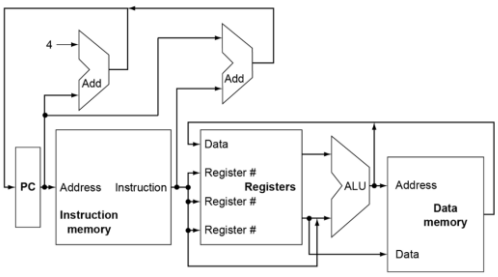
Bhargav (2023112014), A. karthik(2023102009), N.Dinesh(2023102002)

IIIT Hyderabad

Abstract**—***This report presents a Verilog-based RISC-V processor with sequential and pipelined designs that resolve hazards, validated via simulation***.***

Keywords**—***RISC-V, Verilog, Processor Architecture, Pipelined Design, Sequential Design, Pipeline Hazard.***

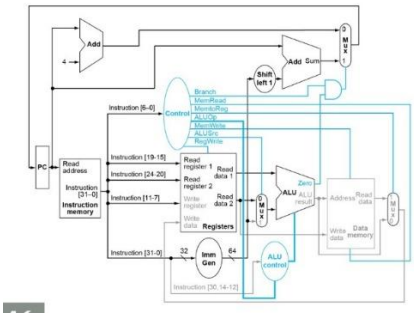
I. INTRODUCTION



This project implements a RISC-V processor using Verilog with two designs: a sequential version that establishes basic functionality and a pipelined version that employs a five-stage pipeline to resolve hazards and enhance performance. Key RISC-V instructions (e.g., add, sub, and, or, ld, sd, beq) are supported and validated through simulations.

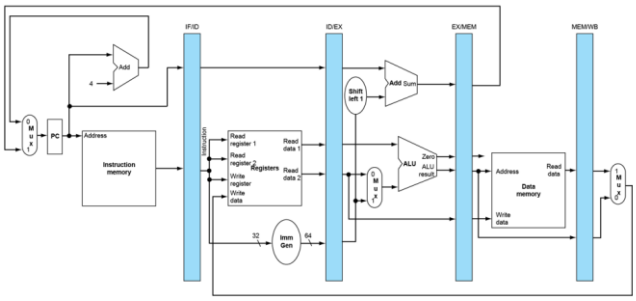
he design process emphasized modular development—each stage was independently tested before integrating into the overall system. This approach helped address challenges such as managing data and control hazards while ensuring synchronized control signals across the pipeline. The report details the architectural decisions, design challenges, and verification techniques used to meet the project requirements, along with suggestions for future enhancements.

II. SYSTEM OVERVIEW (MAIN IDEA)



Our design is based on a traditional approach, following textbook methodologies for both sequential and pipelined processor architectures. The core modules include:

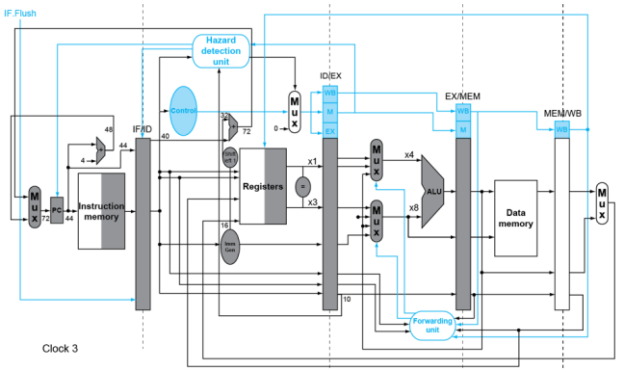
- **Program counter**: Maintains the address of the current instruction. It is updated every clock cycle to point to the next instruction in memory.
- **Instruction fetch:** Retrieves instructions from memory using the address provided by the PC. This unit ensures that the proper instruction is delivered for decoding and execution.
- **Register** File: Implements a standard set of 32 registers for storing operands and intermediate results. This module supports simultaneous reads and writes, as required by RISC-V instructions.
- **ALU**: Executes arithmetic and logical operations on operands from the register file. It processes operations such as add, sub, and, or, which are fundamental to the instruction set.
- **Data memory:** Handles load and store operations, providing access to data required during execution. It works in tandem with the ALU to ensure that memory operations are correctly executed.



To transition from the sequential to the pipelined design, we divided the processor into five stages by inserting four pipeline registers. The stages are as follows:

1. **Instruction Fetch (IF)**
2. **Instruction Decode (ID)**
3. **Execute (EX)**
4. **Memory (MEM)**
5. **Write Back (WB)**

This modification enables multiple instructions to execute simultaneously, improving throughput but introducing hazards. To ensure correct execution, we implement hazard handling mechanisms.



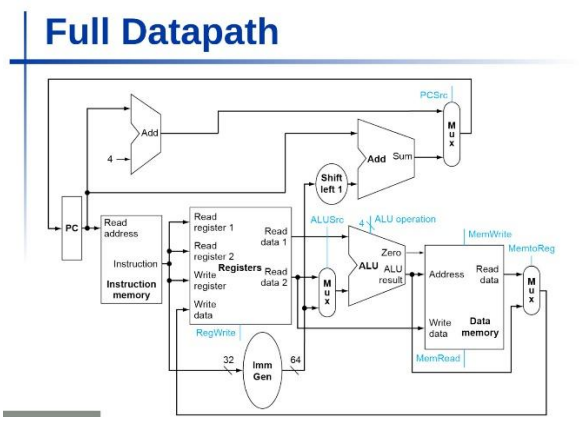
- **Data Hazards:** We implemented forwarding techniques to resolve data hazards. This allows the results of computations to be directly transferred between pipeline stages without waiting for the register file update.
- **Control Hazards:** Control hazards, primarily caused by branch instructions, are managed using a hazard detection unit. Along with appropriate stalling (and flushing, if necessary), this ensures that the pipeline executes the correct sequence of instructions.

By following a traditional processor design methodology, we ensured that our implementation aligns with textbook principles while maintaining clarity and efficiency.

III. IMPLEMENTATION

Our processor was implemented in Verilog using a modular design, ensuring that each core component was developed, tested, and integrated systematically. The implementation began with the **sequential processor**, which laid the foundation for the **pipelined architecture**.

Sequential Processor Implementation:



The **sequential processor** follows a straightforward execution model where each instruction **fully completes all five stages** (**Fetch, Decode, Execute, Memory, Write-Back**) before the next one begins. Since there is **no instruction overlap**, execution is simple and avoids hazards, but performance is significantly reduced.

At each **positive clock edge**, the processor:

1. **Instruction Fetch (IF)** – Fetches the instruction from memory using the **Program Counter (PC)** and updates the PC.
2. **Instruction Decode (ID)** – Decodes the instruction, extracts registers, immediate values, and generates control signals.
3. **Execute (EX)** – The **ALU** performs arithmetic, logic, or branch calculations based on the instruction type.
4. **Memory Access (MEM)** – If it's a **load (ld)**, data is retrieved from memory. If it's a **store (sd)**, a value is written to memory.
5. **Write-Back (WB)** – Writes results (from ALU or memory) back to the **Register File** if needed.

Impact of Load (ld) Instructions

Load instructions take longer due to memory access delays, creating a bottleneck. As a result:

- **Execution speed is limited by the slowest instruction** (usually ld).
- **All instructions must wait**, making execution even slower.
- Unlike pipelining, where multiple instructions execute in parallel, the **sequential processor stalls** until the load completes.

Need for Pipelining

To overcome these inefficiencies, **pipelining** is introduced, allowing multiple instructions to progress simultaneously in different stages. This **reduces waiting time** and **increases throughput**, significantly improving performance over the sequential design.

Pipelining Implementation:

To improve efficiency, we transitioned to a five-stage pipelined design by inserting four pipeline registers at key points:

1. IF/ID Register: Stores the instruction and PC value after fetch.
2. ID/EX Register: Holds decoded operands, control signals, and instruction type.

3. EX/MEM Register: Stores ALU results and memory access control signals.
4. MEM/WB Register: Latches memory data and ALU results for final write-back.

Each stage Implementations:

1. Program Counter (PC):

Purpose:

The **Instruction Fetch (IF) stage** is responsible for retrieving the instruction from memory based on the **Program Counter (PC)**. This stage sets up the instruction pipeline by supplying the processor with new instructions every clock cycle.

Detailed Functions:

1. **Fetch Instruction from Memory**
   - The **PC** is sent to the instruction memory, which retrieves the corresponding instruction.
   - The fetched instruction is a **32-bit machine code value** that represents an operation for the processor.
2. **Compute Next PC**
   - By default, the next instruction is located at PC + 4 since RISC-V uses **32-bit fixed-length instructions**.
   - If a branch instruction (beq) is detected and the condition is satisfied, the PC is updated to the **branch target address** instead.
3. **Store Instruction in IF/ID Pipeline Register**
   - The fetched instruction and PC + 4 are stored in the **IF/ID pipeline register** for the next stage.

Inputs and Outputs

Input Signal	**Description**
PC	Stores the address of the instruction being fetched.
Instruction Memory	Contains all program instructions in machine code.
Branch Target Address	Address to jump to if a branch is taken.

Output Signal	**Description**
Instruction	The fetched 32-bit instruction from memory.
Next PC	PC + 4 (default) or the branch target address if PCSrc = 1.

Internal Operations:

1. **Fetch the instruction** from memory using the PC value.
2. **Compute Next PC** as PC + 4 unless a branch is taken.
3. **Store results in the IF/ID register** for use in the next pipeline stage.

Pipeline Register (IF/ID) Contents

Signal	**Stored Value**
Instruction	The fetched instruction.
PC + 4	Address of the next instruction.

2. Instruction Decode (ID) Stage

Purpose:

The **Instruction Decode (ID) stage** is responsible for:

- ➔ **Decoding the fetched instruction** to extract opcode, function fields, and register addresses.
- ➔ **Generating the immediate value** for instructions that require it.
- ➔ **Reading operands from the register file** for execution.
- ➔ **Generating control signals** for the execution, memory access, and write-back stages.

This stage ensures that all required information is prepared before forwarding it to the **Execute (EX) stage** for processing.

Key Modules in the ID Stage:

Module Name	Function
Instruction Decoder	Extracts opcode, function codes, and register addresses from the instruction.
Register File	Stores and provides values of all general-purpose registers.
Immediate Generator (imm_gen_all_types)	Generates sign-extended immediate values for different instruction types.
Control Unit (control_alucontrol)	Generates necessary control signals based on the instruction type.
ID/EX Pipeline Register	Stores decoded instruction data and control signals for the next stage.

Working Mechanism:

1. **Instruction Decoding:**
   - The instruction is **split into multiple fields** based on its format:
     - Opcode: Defines instruction type (R-type, I-type, Load, Store, Branch, etc.).
     - Rs1, Rs2, Rd: Identifies register locations used in the instruction.
     - Funct3, Funct7: Determines the ALU operation in R-type and some I-type instructions.
2. **Immediate Value Generation:**
   - The **Immediate Generator (imm_gen_all_types)** extracts and sign-extends the **immediate value** based on the instruction format.
   - Supports different instruction types:
     - **I-type:** Uses bits [31:20] as an immediate value.
     - **S-type (Store):** Combines bits [31:25] and [11:7].
     - **B-type (Branch):** Uses scattered bits from the instruction and sign-extends them.
3. **Operand Fetching from Register File:**
   - The **Register File** reads the values of **Rs1 and Rs2**, making them available for the next stage.
4. **Control Signal Generation:**
   - The **Control Unit (control_alucontrol)** generates signals that determine:
     - **ALU Operation:** Specifies which computation the ALU should perform.
     - **Memory Read/Write:** Indicates if the instruction accesses memory.
     - **Branch Control:** Determines if the instruction is a branch.
     - **ALU Source Selection:** Chooses between using register Rs2 or an immediate value.
5. **Pipeline Register Update (ID/EX):**
   - The decoded values, operands, control signals, and immediate values are stored in the **ID/EX pipeline register** for use in the EX stage.

Control Signal Decision Table (From control_alucontrol)

Opcode	Instruction Type	ALUOp	MemRead	MemWrite	RegWrite	ALUSrc
0110011	R-type (add, sub, and, or, xor, sll)	10	0	0	1	0
0000011	Load (ld)	00	1	0	1	1
0100011	Store (sd)	00	0	1	0	1
0010011	I-type ALU (addi, slti, ori,	10	0	0	1	1

Opcode	Instruction Type	ALU Op	MemRead	MemWrite	RegWrite	ALUSrc
	xori, andi)					
1100011	Branch (beq)	01	0	0	0	0

Pipeline Register (ID/EX) Contents:

Signal	Stored Value
Read Data 1	First source register value (Rs1).
Read Data 2	Second source register value (Rs2).
Immediate	Sign-extended immediate value (if applicable).
Control Signals	Execution, memory access, and write-back controls.

3.Execute (EX) Stage

Purpose:

The **Execution Unit** (EX stage) is responsible for performing **arithmetic, logical, and shift operations** based on the instruction type. It forms a crucial part of the **ALU operation** in the pipeline, where computations are executed before memory access and write-back.

This unit determines the result based on the **operation type (op signal)** and selects the second operand (b or imm) based on the **ALU source control signal (idex_alusrc)**.

Inputs and Outputs:

Signal	Type	Description
a	Input	First operand (Rs1 value from register file).
b	Input	Second operand (Rs2 value from register file, unless ALUSrc is set).
op	Input	4-bit control signal selecting the operation type.
imm	Input	Immediate value (used when idex_alusrc = 1).
idex_alusrc	Input	Control signal: **1 = use immediate (imm)**, **0 = use register (b)**.
result	Output	64-bit computation result (sent to MEM stage).

Key Modules in the Execution Unit:

Module Name	Function
ADD	Performs 64-bit addition with carry output.
SUB	Performs 64-bit subtraction.
AND_64_bit	Computes bitwise AND operation.
OR_64_bit	Computes bitwise OR operation.
XOR_64_bit	Computes bitwise XOR operation.
shift_logical_right (SLR)	Performs logical right shift.
shift_logical_left (SLL)	Performs logical left shift.
shift_arithmetic_right (SRA)	Performs arithmetic right shift.
set_less_than (SLT)	Compares two signed values and sets result accordingly.
set_less_than_unsigned (SLTU)	Compares two unsigned values and sets result accordingly.

Working Mechanism:

1. **Operand Selection (MUX Logic for ALU Source Selection)**
    - The second operand is chosen using idex_alusrc:
        - If idex_alusrc = 1, operand **operb = imm** (immediate).
        - If idex_alusrc = 0, operand **operb = b** (register value).
2. **Execution of ALU Operations**

- Based on the op control signal, the corresponding ALU operation is performed.
- The case structure selects the appropriate operation result:
  - 0000 → Addition (ADD_OUT)
  - 0001 → Subtraction (SUB_OUT)
  - 0010 → AND (AND_OUT)
  - 0011 → OR (OR_OUT)
  - 0100 → XOR (XOR_OUT)
  - 0101 → Shift Logical Right (SLR_OUT)
  - 0110 → Shift Logical Left (SLL_OUT)
  - 0111 → Shift Arithmetic Right (SRA_OUT)
  - 1000 → Set Less Than (SLT_OUT → 1 if a < b, else 0)
  - 1001 → Set Less Than Unsigned (SLTU_OUT → 1 if a < b in unsigned mode, else 0)

3. **Result Output and Formatting (SLT and SLTU Handling)**
   - Since SLT and SLTU are single-bit comparisons, they are converted to a **64-bit format** by extending them with 63 leading zeros.

ALU Operation Decision Table:

Operation (op)	**ALU Function**	**Result**
0000	ADD	a + operb
0001	SUB	a - operb
0010	AND	a & operb
0011	OR	a | operb
0100	XOR	a ^ operb
0101	Shift Right (Logical)	a >> operb[5:0]
0110	Shift Left (Logical)	a << operb[5:0]
0111	Shift Right (Arithmetic)	a >>> operb[5:0]
1000	Set Less Than (SLT)	1 if (a < operb) else 0
1001	Set Less Than Unsigned (SLTU)	1 if (a < operb) (unsigned)

Purpose:

- Performs **ALU computations** for arithmetic, logical, and memory address calculations.
- Determines **branch conditions** and **target addresses**.
- Uses **forwarding logic** to resolve data hazards.

Detailed Functions:

1. **Select ALU Operands**
   - Chooses between register values and immediate values.
   - Uses forwarding if necessary.
2. **ALU Execution**
   - Performs operation (ADD, SUB, AND, OR, etc.) based on ALU control signals.
3. **Branch Decision**
   - If the instruction is a beq, the **Zero Flag** determines if the branch is taken.

Inputs and Outputs

Input Signal	**Description**
Read Data 1	First operand.
Read Data 2	Second operand (or immediate).
ALU Control	Determines which operation to execute.

Output Signal	Description
ALU Result	Computed result from the ALU operation.
Zero Flag	1 if the result is zero (used for branching).

Pipeline Register (EX/MEM) Contents

Signal	Stored Value
ALU Result	Computed value from ALU.
Zero Flag	1 if the branch condition is met.
Control Signals	Memory access and write-back actions.

.

4 .Memory Access (MEM) Stage

Purpose:

The **Memory Access (MEM) stage** is responsible for handling **load (ld) and store (sd) instructions**, ensuring that data is correctly transferred between registers and memory. It interacts with the **data memory unit**, determines whether to read or write data, and forwards the results to the **Write-Back (WB) stage**.

This stage ensures that memory operations occur without disrupting pipeline execution and minimizes hazards caused by memory dependencies.

Key Modules in the MEM Stage:

Module Name	Function
Data Memory Unit	Reads data from memory for load instructions and writes data for store instructions.
Memory Control Logic	Determines whether a memory operation is a **read or write**.
Pipeline Register (MEM/WB)	Stores the memory read data or ALU result for the next stage.

Working Mechanism:

1. **Memory Operation Decision (Control Logic)**
   - The **control signals from the EX stage (idex_memread, idex_memwrite)** determine whether the instruction accesses memory.
   - If idex_memread = 1, the instruction is a **load**, and data is retrieved from memory.
   - If idex_memwrite = 1, the instruction is a **store**, and the value from Rs2 is written to memory.
2. **Memory Read Operation (Load Instruction - ld)**
   - If the instruction is a **load**, the **memory unit fetches data** from the computed address (ALU result).
   - The data is then **stored in the MEM/WB pipeline register** for the Write-Back stage.
3. **Memory Write Operation (Store Instruction - sd)**
   - If the instruction is a **store**, the value in Rs2 is **written into the memory** at the address computed by the ALU.
   - No data is forwarded to the Write-Back stage since store instructions do not update registers.
4. **Pipeline Register Update (MEM/WB)**
   - The **memory-read value (for loads)** or **ALU result (for arithmetic instructions)** is stored in the pipeline register for use in the WB stage.

Control Signal Decision Table (From EX Stage)

Opcode	Instruction Type	MemRead	MemWrite	Action in MEM Stage
0000011	Load (ld)	1	0	**Read from memory** at ALU-computed address.
0100011	Store (sd)	0	1	**Write to memory** at ALU-computed address.
0110011	R-type (add, sub, or, etc.)	0	0	**No memory operation, forward ALU result.**

Opcode	Instruction Type	MemRead	MemWrite	Action in MEM Stage
1100011	Branch (beq)	0	0	**No memory operation, branch handled in EX.**

Pipeline Register (MEM/WB) Contents:

Signal	Stored Value
Memory Read Data	Data fetched from memory (for load instructions).
ALU Result	Computed result from ALU (for arithmetic instructions).
Destination Register (Rd)	Register where data will be written (for loads).
Control Signals	Determines write-back behavior.

5. Write-Back (WB) Stage

Purpose:

- **Writes final results back into the register file** so future instructions can use them.
- **Ensures correct register updates** after computation or memory access.

Key Modules in WB Stage:

Module Name	Function
MUX (Write-Back Selection)	Selects between memory-read data and ALU result.
Register File	Stores computed results for future instructions.
Control Logic for Register Write-Back	Ensures only valid data is written to registers.

How It Works:

1. **Write-Back Selection (MUX for WB)**
   - If the instruction was an ld (load), the **MUX selects Memory Read Data**.
   - Otherwise, the **MUX selects ALU Result** for arithmetic/logical operations.
2. **Register File Update**
   - The selected value is **written back into the Register File** at location Rd.
   - The **Control Logic** ensures write-back occurs only if RegWrite = 1.
3. **Final Instruction Completion**
   - After this stage, the instruction is **fully executed**, and its result is available for use in future operations.

Pipeline Register (WB) Contents

Signal	Stored Value
Final Register Value	Data written to the destination register.

Hazard Handling Implementation:

While pipelining improves instruction throughput, it introduces hazards that can disrupt the normal flow of execution. These hazards arise due to dependencies between instructions and must be managed using dedicated hardware units to ensure correct execution.

1.Implementation of Forwarding Unit (Data Hazard Resolution):

Purpose:

The **Forwarding Unit** is designed to resolve **data hazards** in a pipelined processor by forwarding the correct data to the **Execute (EX) stage** when an instruction depends on a result that is still in the pipeline. This prevents unnecessary stalls, improving performance by enabling **data to be used as soon as it is available** instead of waiting for it to be written back to the register file.

Inputs and Outputs:

Signal	Type	Description
id_ex_Rs1	Input	Source register 1 from the **ID/EX** stage.
id_ex_Rs2	Input	Source register 2 from the **ID/EX** stage.

Signal	Type	Description
ex_mem_rd	Input	Destination register from the **EX/MEM** stage (result currently in EX/MEM).
mem_wb_rd	Input	Destination register from the **MEM/WB** stage (result currently in MEM/WB).
ex_mem_Regwrite	Input	Indicates if the EX/MEM stage is writing back to a register.
mem_wb_Regwrite	Input	Indicates if the MEM/WB stage is writing back to a register.
forwardA	Output	Control signal for selecting the correct operand for **Rs1** in EX stage.
forwardB	Output	Control signal for selecting the correct operand for **Rs2** in EX stage.

Working Mechanism:

The Forwarding Unit operates using **combinational logic** to check if an instruction in the **EX/MEM or MEM/WB stages** is writing a value that is needed by the **ID/EX stage**. The logic ensures that operands in the EX stage receive the most recent value without waiting for the **Write-Back (WB) stage**.

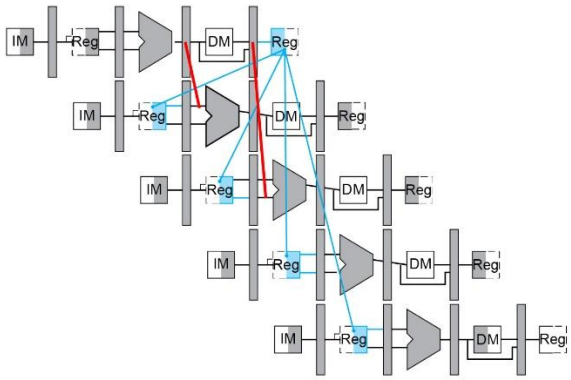
1. **EX Hazard (Forward from EX/MEM Stage - Priority 1):**
   - If the instruction in EX/MEM is writing back (ex_mem_Regwrite = 1) and the destination register (ex_mem_rd) matches either source register (id_ex_Rs1 or id_ex_Rs2), **forwarding occurs from the EX/MEM stage**.
   - forwardA = 2'b10 → Use EX/MEM value for Rs1.
   - forwardB = 2'b10 → Use EX/MEM value for Rs2.

2. **MEM Hazard (Forward from MEM/WB Stage - Priority 2):**
   - If the instruction in MEM/WB is writing back (mem_wb_Regwrite = 1), but there is **no conflicting EX/MEM forward**, then MEM/WB forwarding is applied.
   - forwardA = 2'b01 → Use MEM/WB value for Rs1.
   - forwardB = 2'b01 → Use MEM/WB value for Rs2.

3. **Default Case (No Forwarding Needed):**
   - If no data hazards exist, both forwardA and forwardB remain **2'b00**, meaning the values are read from the register file normally.

Forwarding Conditions

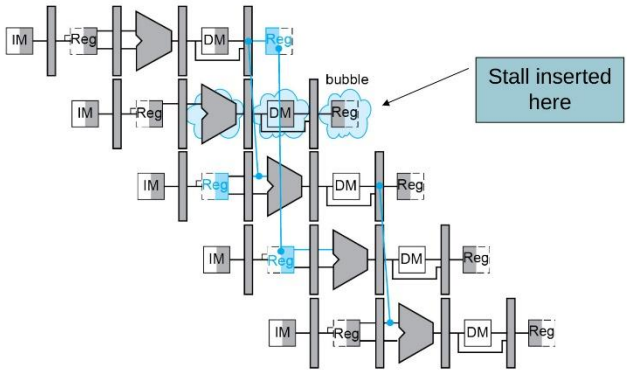
Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX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



Advantages of the Forwarding Unit:

- ➔ **Prevents pipeline stalls** by allowing dependent instructions to use the latest values immediately.
- ➔ **Reduces execution delays** caused by waiting for data to be written to the register file
- ➔ **Ensures correct program execution** by selecting the correct operand dynamically.

2.Implementation of the Hazard Detection Unit



Purpose:

The **Hazard Detection Unit** is responsible for **detecting load-use hazards** in a pipelined processor and **inserting stalls** when necessary. A **load-use hazard** occurs when an instruction in the **EX stage** is performing a **load (ld) operation**, and the next instruction in the **ID stage** requires the loaded value. Since the value has not yet been written back to the register file, executing the next instruction immediately would result in **incorrect data** being used.

This unit **generates a stall signal**, pausing the pipeline to allow the data to be correctly loaded before execution continues.

Inputs and Outputs:

Signal	Type	Description
idex_memread	Input	Indicates that the instruction in the EX stage is a **load (ld)** instruction.
idex_rd	Input	Destination register of the instruction in the EX stage (the register being loaded).
ifid_rs1	Input	First source register of the instruction in the ID stage.
ifid_rs2	Input	Second source register of the instruction in the ID stage.
stall	Output	Stall signal to control the pipeline (1 = stall, 0 = continue execution).

Working Mechanism:

The **Hazard Detection Unit** continuously checks whether the instruction in the **EX stage** is performing a **memory read (ld)** and whether the next instruction (currently in the **ID stage**) depends on that loaded value.

1. **Default Behavior (No Stall Needed)**
   - If the EX stage is **not** performing a memory read (idex_memread = 0), no stall is required (stall = 0).
   - If the instruction in the ID stage does **not** use the destination register of the EX stage, no stall is required.
2. **Detecting a Load-Use Hazard**
   - If the instruction in **EX is a load (idex_memread = 1)** and
   - The destination register (idex_rd) is being used by the **ID stage instruction (ifid_rs1 or ifid_rs2)**,
   - Then the processor **inserts a stall (stall = 1)** to wait for the data to be available.
3. **Inserting a Stall**
   - A stall prevents the **IF and ID stages from advancing**, allowing the load instruction to complete.
   - The processor automatically prevents incorrect execution by **inserting a bubble** (NOP instruction) in the pipeline.

Hazard Detection Decision Table:

Condition	Stall Signal (1 = Stall, 0 = No Stall)	Reason
idex_memread = 0	0	No load instruction in EX, so no hazard.
idex_memread = 1 & idex_rd ≠ ifid_rs1 & idex_rd ≠ ifid_rs2	0	Load in EX, but ID instruction doesn't

Condition	Stall Signal (1 = Stall, 0 = No Stall)	Reason
		need the loaded value.
idex_memread = 1 & idex_rd == ifid_rs1	1	Load in EX, and ID instruction needs the loaded value for Rs1.
idex_memread = 1 & idex_rd == ifid_rs2	1	Load in EX, and ID instruction needs the loaded value for Rs2.

Advantages of the Hazard Detection Unit:

- ➔ **Prevents incorrect instruction execution** by ensuring the correct data is available before use.
- ➔ **Minimizes pipeline stalls** by only stalling when absolutely necessary.
- ➔ **Ensures smooth pipeline flow** without requiring forwarding in cases where the data is not yet available.

3.Implementation of the Branch Hazard Unit

Purpose:

The **Branch Hazard Unit** is responsible for handling **control hazards** in a pipelined processor. A **control hazard** occurs when a **branch (beq) instruction** is encountered, and the processor must determine whether to take the branch or continue sequential execution. Since the **branch decision is made in the Execute (EX) stage**, the instructions **already in the pipeline might be incorrect** if a branch is taken.

The **Branch Hazard Unit ensures correct execution** by:

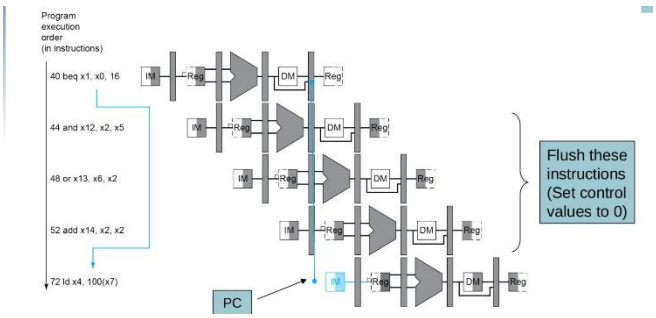
-> **Flushing incorrect instructions from the pipeline** when a branch is taken.

-> **Updating the PC source to select the correct next instruction** (either PC + 4 or the branch target).

Inputs and Outputs:

Signal	Type	Description
branch	Input	Indicates that a branch instruction is being executed (from EX stage).
zero	Input	ALU Zero flag: **1 if branch condition is met**, **0 otherwise**.
clk	Input	Clock signal to synchronize operations.
flush_ifid	Output	Signal to **flush** the IF/ID pipeline register if a branch is taken.
flush_idex	Output	Signal to **flush** the ID/EX pipeline register if a branch is taken.
pc_src	Output	Selects **next PC value**: **0 → PC + 4**, **1 → Branch Target Address**.

Working Mechanism:



The **Branch Hazard Unit continuously monitors the EX stage** to determine if a branch should be taken.

1. **Default Execution (No Branch Taken)**
   - branch = 0 or zero = 0:
   - The processor continues executing instructions sequentially (PC + 4).
   - flush_ifid = 0, flush_idex = 0, pc_src = 0.
2. **Branch Taken (Flush Required)**
   - If branch = 1 **and** zero = 1:
   - The processor **flushes IF/ID and ID/EX pipeline registers** to remove incorrectly fetched instructions.
   - pc_src is set to **1**, selecting the **branch target address** as the next PC value.

3. **Flushing the Pipeline**
    - **Flushing ensures that no incorrect instructions are executed** after a branch decision is made.
    - The pipeline registers are cleared, and a **NOP (No Operation)** instruction is inserted to prevent errors.

Branch Decision Table:

Branch Signal	Zero Flag	Flush IF/ID	Flush ID/EX	PC Source (pc_src)	Action Taken
0	0	0	0	0	Continue sequential execution (PC + 4).
0	1	0	0	0	Continue sequential execution (PC + 4).
1	0	0	0	0	Branch condition not met, continue normally.
1	1	1	1	1	**Branch taken → Flush pipeline, update PC to branch target.**

Advantages of the Branch Hazard Unit:

- ➔ **Prevents incorrect execution** by removing instructions that should not be executed after a branch.
- ➔ **Minimizes wasted cycles** by making the branch decision as early as possible.
- ➔ **Ensures correct program flow** by dynamically updating the PC to the correct address.

IV. Block-Level Overview

1. **cpu_top (Top Module)** – Connects all pipeline stages, memory, and control units, managing instruction execution across cycles.
2. **instr_mem (Instruction Memory)** – Stores and fetches instructions based on the Program Counter (PC).
3. **control_alucontrol (Control Unit)** – Decodes the instruction and generates control signals for ALU, memory, and pipeline execution.
4. **imm_gen_all_types (Immediate Generator)** – Extracts and sign-extends immediate values from instructions based on format type.
5. **execution (ALU Execution Unit)** – Performs arithmetic, logical, and shift operations based on control signals.
6. **data_memory (Data Memory Unit)** – Reads from and writes to memory for load (ld) and store (sd) instructions.
7. **forwarding_unit (Forwarding Unit)** – Resolves data hazards by forwarding register values from later pipeline stages.
8. **HazardDetectionUnit (Hazard Detection Unit)** – Detects load-use hazards and inserts stalls when needed.
9. **branch_hazard_unit (Branch Hazard Unit)** – Flushes incorrect instructions when a branch is taken to prevent control hazards.
10. **Pipeline Registers (IF/ID, ID/EX, EX/MEM, MEM/WB)** – Store intermediate instruction data between pipeline stages.
11. **Register File (Registers in cpu_top)** – Holds general-purpose registers and updates them during write-back.
12. **pc (Program Counter)** – Tracks the current instruction address and updates based on sequential or branch execution.

V . Possible Improvements

While our processor successfully implements a **sequential and pipelined RISC-V design with hazard resolution**, there are several ways to further optimize its performance and efficiency.

1. Branch Prediction (1-bit or 2-bit Predictors):
Instead of stalling on branches, we can implement a **branch predictor**:

- **1-bit predictor:** Remembers the last branch outcome (taken/not taken).
- **2-bit predictor:** Uses a counter to reduce mispredictions.

2. Deeper Pipelining
Extending beyond a **five-stage pipeline** (e.g., 7 or 10 stages) can increase clock speed but requires better hazard handling.

4. Out-of-Order Execution:
Instead of executing instructions sequentially, an **out-of-order execution model** schedules ready instructions first, minimizing idle cycles.

4. Cache Implementation
Adding **instruction and data caches** speeds up memory access and reduces delays in load/store operations.

VI. RESULTS

Sequential:

Testcases:

1. Branching:

If condition satisfied :

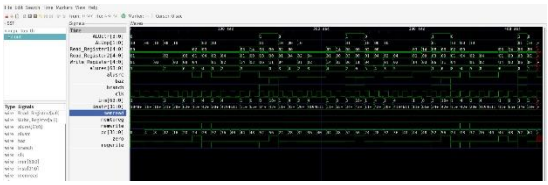
Instructions:

```
memory[0] = 32'b0000000_10000_00000_000_00001_0010011; // addi x1, x0, 16   (Initialize x1 = 1)
memory[1] = 32'b0000000_10000_00000_000_00010_0010011; // addi x2, x0, 16  (Initialize x1 = 1)
memory[2] = 32'b0000000_00001_00010_000_00100_1100011; // beq x1, x2, loop (branching example)
memory[3] = 32'b0100000_01001_01101_000_01000_0110011; // sub x8, x13, x9
memory[4] =32'b0000000_01001_01101_000_01000_0110011; // add x8, x13, x9
```

Output:

```
PS C:\Users\DELL\documents> vvp hi.out
Time = 0 | x8=          8|x1=          16|x2=          2|zero=0,branch=0
Time = 15 | x8=         8|x1=          16|x2=          16|zero=0,branch=0
Time = 25 | x8=         8|x1=          16|x2=          16|zero=1,branch=1
Time = 35 | x8=         22|x1=         16|x2=          16|zero=0,branch=0
```

Gtkwave:



If condition not satisfied:

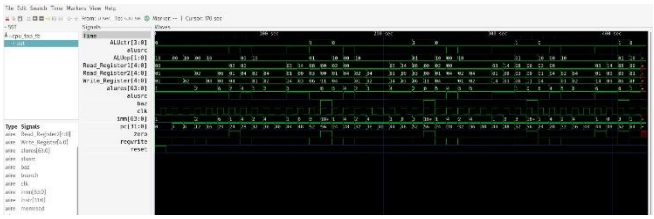
Instructions:

```
memory[0] = 32'b0000000_10000_00000_000_00001_0010011; // addi x1, x0, 16   (Initialize x1 = 1)
memory[1] = 32'b0000000_10001_00000_000_00010_0010011; // addi x2, x0, 17  (Initialize x1 = 1)
memory[2] = 32'b0000000_00001_00010_000_00100_1100011; // beq x1, x2, loop (branching example)
memory[3] = 32'b0100000_01001_01101_000_01000_0110011; // sub x8, x13, x9
memory[4] =32'b0000000_01001_01101_000_01000_0110011; // add x8, x13, x9
```

Output :

```
PS C:\Users\DELL\documents> vvp hi.out
Time = 0 | x8=          8|x1=          16|x2=          2|zero=0,branch=0
Time = 15 | x8=         8|x1=          16|x2=          17|zero=0,branch=0
Time = 25 | x8=         8|x1=          16|x2=          17|zero=1,branch=1
Time = 35 | x8=         4|x1=          16|x2=          17|zero=0,branch=0
```

Gtkwave:

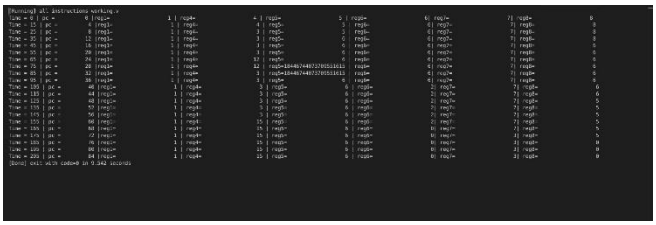


2. Working of and,or,xor,add,sub,ld ad sd:

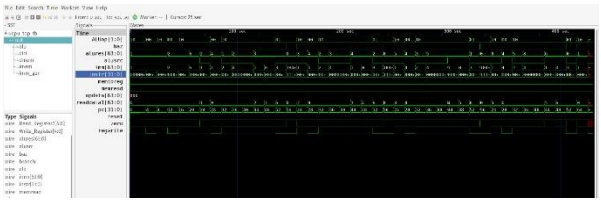
Instructions:

```
memory[0]  = 32'b0000000_00001_00000_000_00001_0010011; // addi x1, x0, 1   (Initialize x1 = 1)
memory[1]  = 32'b0000000_00001_00000_010_00001_0100011; // sd x1, 0(x0)      (Store Fib(1) = 1)
memory[2]  = 32'b0000000_00001_00010_000_00100_0110011; // add x4, x1, x2
memory[3]  = 32'b0100000_00011_01001_000_00101_0110011; // sub x5, x9, x3
memory[4]  = 32'b0000000_00100_00000_010_01000_0100011; // sd x4, 8(x0)
memory[5]  = 32'b0000000_00101_00000_010_10000_0100011; // sd x5, 16(x0)
memory[6]  = 32'b0000000_01001_00011_000_00100_0110011; // add x4, x9, x3
memory[7]  = 32'b0100000_00010_00001_000_00101_0110011; // sub x5, x1, x2

memory[8]  = 32'b0000000_01000_00000_010_00100_0000011; // ld x4, 8(x0)
memory[9]  = 32'b0000000_10000_00000_010_00101_0000011; // ld x5, 16(x0)

memory[10] = 32'b0000000_00101_00100_111_00110_0110011; // and x6, x4, x5
memory[11] = 32'b0000000_00101_00100_110_00111_0110011; // or x7, x4, x5
memory[12] = 32'b0000000_00101_00100_100_01000_0110011; // xor x8, x4, x5
memory[13]  = 32'b0000000_01111_00000_000_01101_0010011; // addi x13, x0, 15
memory[14]  = 32'b0000000_01111_00000_000_01110_0010011; // addi x14, x0, 15
memory[15] = 32'b0000000_01101_01110_000_00100_1100011; // beq x14, x13, loop (branching example)
memory[16] = 32'b0000000_01111_00000_000_01111_0010011; // addi x15, x0, 15
memory[17] = 32'b0000000_00010_00001_111_00110_0110011; // and x6, x1, x2
memory[18] = 32'b0000000_00010_00001_110_00111_0110011; // or x7, x1, x2
memory[19] = 32'b0000000_00010_00010_100_01000_0110011; // xor x8, x2, x2
memory[20] = 32'b0000000_11000_00000_010_01000_0100011; // sd x8, 24(x0)
memory[21] = 32'b0000000_11000_00000_010_01001_0000011; // ld x9, 24(x0)
```

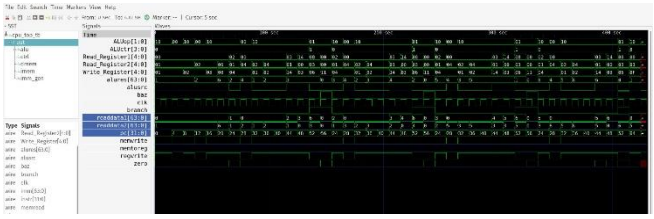
Output :



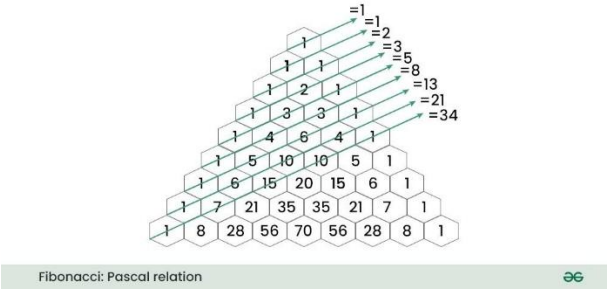
Gtkwave:



Gtkwave:



3. Fibonacci sequence:



Instructions:

```
memory[0]  = 32'b0000000_00001_00000_000_00001_0010011; // addi x1, x0, 1   (Initialize x1 = 1)
memory[1]  = 32'b0000000_00001_00000_010_00001_0100011; // sd x1, 0(x0)      (Store Fib(1) = 1)
memory[2]  = 32'b0000000_00001_00000_000_00010_0010011; // addi x2, x0, 1   (Initialize x2 = 1)
memory[3]  = 32'b0000000_00010_00000_010_00010_0100011; // sd x2, 8(x0)      (Store Fib(2) = 1)
memory[4]  = 32'b0000000_00010_00000_000_00011_0010011; // addi x3, x0, 2   (Counter x3 = 2)
memory[5]  = 32'b0000000_01010_00000_000_01000_0010011; // addi x8, x0, 10  (Loop limit N=3)
memory[6]  = 32'b0000000_00001_00010_000_00100_0110011; // add x4, x1, x2   (Fib(n) = x1 + x2)
memory[7]  = 32'b0000000_00100_00000_010_00100_0100011; // sd x4, 16(x0)     (Store Fib(n))
memory[8]  = 32'b0000000_00010_00000_000_00001_0110011; // add x1, x2, x0    (x1 = x2)
memory[9]  = 32'b0000000_00100_00000_000_00010_0110011; // add x2, x4, x0    (x2 = x4)
memory[10]  = 32'b0000000_00100_00000_000_00010_0110011; // add x2, x4, x0    (x2 = x4)
memory[11] =32'b0000000_00001_00011_000_10100_0010011; // addi x20, x3, 1   (Increment counter)
memory[12] =32'b0000000_00000_10100_000_00011_0010011; // addi x3, x20, 0   (Increment counter)
memory[13] = 32'b0000000_00011_01000_000_00110_1100011; // beq x3, x~8, exit (Exit if x3 == N)
memory[14] = 32'b1111111_00000_00000_000_10001_1100011; // beq x0, x0, -32   (Jump back to loop)
memory[15] = 32'b0000000_00000_00000_000_00000_0000000; // nop
memory[16] = 32'b0000000_00001_00000_000_01111_0010011; // addi x15, x0, 1
```

Output:

```
PS C:\Users\DELL\documents> vvp h1.out
Time = 0     | output=                1
Time = 65     | output=                2
Time = 155     | output=                3
Time = 245     | output=                5
Time = 335     | output=                8
Time = 425     | output=               13
Time = 515     | output=               21
Time = 605     | output=               34
Time = 695     | output=               55
PS C:\Users\DELL\documents>
```

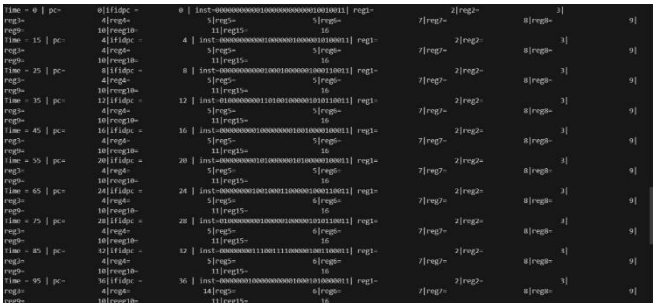
Pipelining (without hazard handling):

1. All types of data hazards:

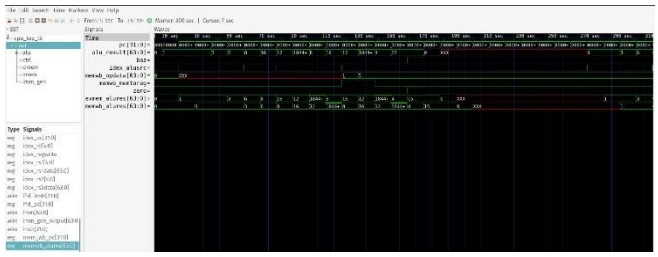
Instructions:

```
ld x5 0(x2)
ld x6 1(x2)
add x4 x5 x6
ld x7 2(x2)
add x4 x4 x7
ld x8 3(x2)
add x4 x4 x8
ld x9 4(x2)
add x4 x4 x9
sd x4 0(x3)
ld x10 0(x3)
add x10 x10 x10
sd x10 0(x3)
```

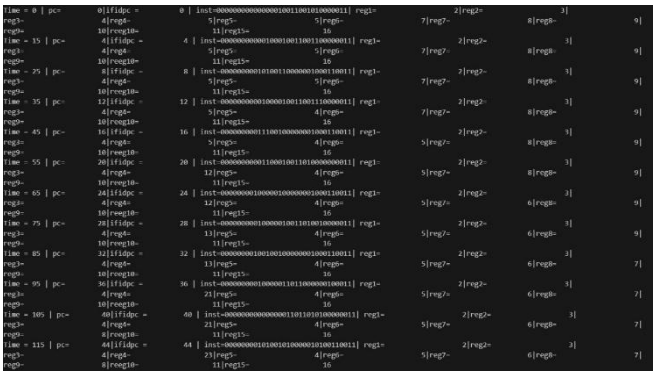
Outputs:



Gtkwaves:



Outputs:

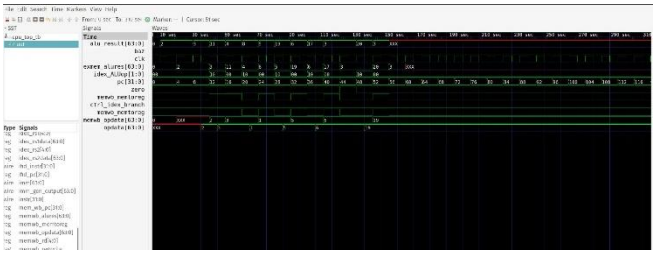


2.Double data hazard:

Instructions:

```
00000000000001000000000000010010011   # addi x1, x0, 0      # x1 = 0
00000000000001000001000001010001101   # lui x2, 0x80        # Load upper immediate, x2 = 0x80000
00000000000001000110000001011001101   # lui x3, 0x10        # Load upper immediate, x3 = 0x10000
01000000001101001000001011001101      # lui x5, 0x20000     # Load upper immediate, x5 = 0x20000
00000000001000000000100100001000101   # addi x8, x0, 4      # x8 = 4
00000000001010000000101000001000101   # addi x10, x0, 5     # x10 = 5
00000001001000001100000001011001101   # lui x4, 0x24000     # Load upper immediate, x4 = 0x24000
01000000000001000001000000001011001   # lui x9, 0x10000     # Load upper immediate, x9 = 0x10000
00000001111101111000000100110000111   # beq x31, x31, label # Branch always (infinite loop)
00000001000000000000100000100000011   # lb x4, 0(x0)        # Load byte from address 0 into x4
00000000001010010011100001101110011   # and x3, x10, x5     # x3 = x10 & x5
00000000001010010011000011111110011   # or x3, x10, x5      # x3 = x10 | x5
00000000001010010010000100001110011   # xor x4, x10, x5     # x4 = x10 ^ x5
00000000011110000000000001101001001   # addi x6, x0, 15     # x6 = 15
00000000011110000000000001110010011   # addi x7, x0, 15     # x7 = 15
00000000011100111000000100000100011   # sw x7, 0(x9)        # Store x7 at address x9
00000000011110000000000001111001001   # addi x8, x0, 15     # x8 = 15
00000000001000000111100001101110011   # and x3, x1, x2      # x3 = x1 & x2
00000000001000000111000011111110011   # or x3, x1, x2       # x3 = x1 | x2
00000000001000001010000100001110011   # xor x4, x1, x2      # x4 = x1 ^ x2
00000000010000000000111400000100011   # sw x8, 8(x0)        # Store x8 at address 8
00000000110000000000110100100000011   # lh x9, 24(x0)       # Load halfword from address 24 into x9
00000000110010000000100101011111111   # jalr x10, 25(x0)    # Jump and link register (to address 25)
```

Gtkwave:



Outputs:

3.Instructions :

```
00000000000000010011001010000011   # ld x5, 0(x1)     # Load from address in x1 to x5
00000000000010001001100110000011   # ld x6, 1(x1)     # Load from address in x1+1 to x6
00000000010100110000001000110011   # add x4, x6, x5   # x4 = x6 + x5
00000000001000010011001110000011   # ld x7, 2(x1)     # Load from address in x1+2 to x7
00000000001110010000000100011011   # add x4, x4, x7   # x4 = x4 + x7
00000000001100010011010000000011   # ld x8, 3(x1)     # Load from address in x1+3 to x8
00000000100000010000000100011011   # add x4, x4, x8   # x4 = x4 + x8
00000000010000010011010100000011   # ld x9, 4(x1)     # Load from address in x1+4 to x9
00000000010010010000000100011011   # add x4, x4, x9   # x4 = x4 + x9
00000000010000011011000000100011   # sd x4, 4(x3)     # Store x4 at address x3+4
00000000000000011011010100000011   # ld x10, 0(x3)    # Load from address in x3 to x10
00000000101001010000001010110011   # add x10, x10, x10  # x10 = x10 + x10
00000000101000011011000000100011   # sd x10, 4(x1)    # Store x10 at address x1+4
```

Pipelining (hazard handling):

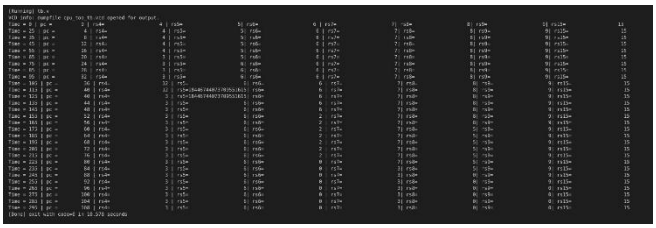
Testcases:

1. **Working of all operations (and,or,add,sub,ld,sd)**

Instructions:

```
memory[0]  = 32'b0000000_00001_00000_000_00001_0010011; // addi x1, x0, 1   (Initialize x1 = 1)
memory[1]  = 32'b0000000_00001_00000_010_00001_0100011; // sd x1, 0(x0)     (Store Fib(1) = 1)
memory[2]  = 32'b0000000_00001_00010_000_00100_0110011; // add x4, x1, x2
memory[3]  = 32'b0100000_00011_01001_000_00101_0110011; // sub x5, x9, x3
memory[4]  = 32'b0000000_00100_00000_010_01000_0100011; // sd x4, 8(x0)
memory[5]  = 32'b0000000_00101_00000_010_10000_0100011; // sd x5, 16(x0)
memory[6]  = 32'b0000000_01001_00011_000_00100_0110011; // add x4, x9, x3
memory[7]  = 32'b0100000_00010_00001_000_00101_0110011; // sub x5, x1, x2

memory[8] = 32'b0000000_01000_00000_010_00100_0000011; // ld x4, 8(x0)
memory[9] = 32'b0000000_10000_00000_010_00101_0000011; // ld x5, 16(x0)

memory[10] = 32'b0000000_00101_00100_111_00110_0110011; // and x6, x4, x5
memory[11] = 32'b0000000_00101_00100_110_00111_0110011; // or x7, x4, x5
memory[12] = 32'b0000000_00101_00100_100_01000_0110011; // xor x8, x4, x5
memory[13]  = 32'b0000000_01111_00000_000_01101_0010011; // addi x13, x0, 15
memory[14]  = 32'b0000000_01111_00000_000_01110_0010011; // addi x14, x0, 15
memory[15] = 32'b0000000_01101_01110_000_00100_1100011; // beq x14, x13, loop (branching example)
memory[16] = 32'b0000000_01111_00000_000_01111_0010011; // addi x15, x0, 15
memory[17] = 32'b0000000_00010_00001_111_00110_0110011; // and x6, x1, x2
memory[18] = 32'b0000000_00010_00001_110_00111_0110011; // or x7, x1, x2
memory[19] = 32'b0000000_00010_00010_100_01000_0110011; // xor x8, x2, x2
memory[20] = 32'b0000000_11000_00000_010_01000_0100011; // sd x8, 24(x0)
memory[21] = 32'b0000000_11000_00000_010_01001_0000011; // ld x9, 24(x0)
```

Output:



2. Handling double data hazard:

Instruction:

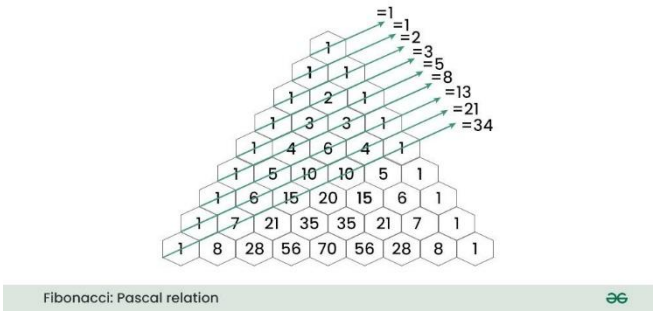
```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

Output:

```
[Running] tb.v
VCD info: dumpfile cpu_top_tb.vcd opened for output.
Time = 0| rs1=                    1
Time = 45| rs1=                   3
Time = 55| rs1=                   6
Time = 65| rs1=                  10
[Done] exit with code=0 in 10.06 seconds
```

Gtkwave:

3. Fibonacci sequence:



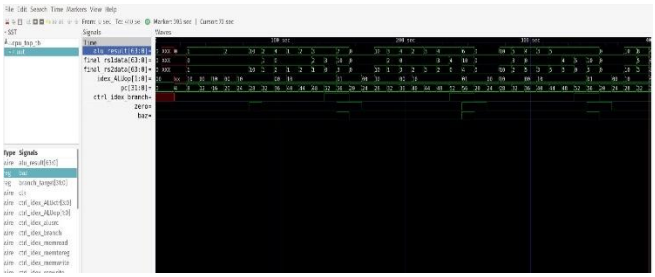
Instructions:

```
memory[0]= 32'b0000000_00001_00000_000_00001_0010011;  //addi x1, x0, 1   (Initialize x1 = 1)
memory[1]= 32'b0000000_00001_00000_010_00001_0100011;  //sd x1, 0(x0)     (Store Fib(1) = 1)
memory[2]= 32'b0000000_00001_00000_000_00010_0010011;  //addi x2, x0, 1   (Initialize x2 = 1)
memory[3]= 32'b0000000_00010_00000_010_00010_0100011;  //sd x2, 8(x0)     (Store Fib(2) = 1)
memory[4]= 32'b0000000_00010_00000_000_00011_0010011;  //addi x3, x0, 2   (Counter x3 = 2)
memory[5]= 32'b0000000_01010_00000_000_01000_0010011;  //addi x8, x0, 10   (Loop limit N=6)
memory[6]= 32'b0000000_00001_00010_000_00100_0110011;  //add x4, x1, x2   (Fib(n) = x1 + x2)
memory[7]= 32'b0000000_00100_00000_010_00100_0100011;  //sd x4, 16(x0)    (Store Fib(n))
memory[8]= 32'b0000000_00010_00000_000_00001_0110011;  //add x1, x2, x0   (x1 = x2)
memory[9]= 32'b0000000_00100_00000_000_00010_0110011;  //add x2, x4, x0   (x2 = x4)
memory[10]= 32'b0000000_00001_00011_000_10100_0010011;  //addi x20, x3, 1   (Increment counter)
memory[11]= 32'b0000000_00000_10100_000_00011_0010011;  //addi x3, x20, 0   (Update counter)
memory[12]= 32'b0000000_00011_01000_000_00110_1100011;  //beq x3, x8, exit  (Exit if x3 == N)
memory[13]= 32'b1111111_00000_00000_000_10001_1100011;  //beq x0, x0, -32   (Jump back to loop)
memory[14]= 32'b0000000_00000_00000_000_00000_0000000;  //nop
memory[15]= 32'b0000000_00001_00000_000_01111_0010011;  //addi x15, x0, 1   (Final instruction)
```

Output:

```
[Running] tb.v
VCD info: dumpfile cpu_top_tb.vcd opened for output.
Time = 0|fibonocci sequence=                    1
Time = 105|fibonocci sequence=                  2
Time = 205|fibonocci sequence=                  3
Time = 305|fibonocci sequence=                  5
Time = 405|fibonocci sequence=                  8
Time = 505|fibonocci sequence=                 13
Time = 605|fibonocci sequence=                 21
Time = 705|fibonocci sequence=                 34
Time = 805|fibonocci sequence=                 55
[Done] exit with code=0 in 12.464 seconds
```

Gtkwave:

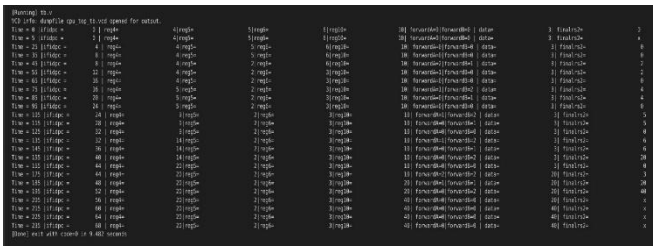


Handling All data hazards:

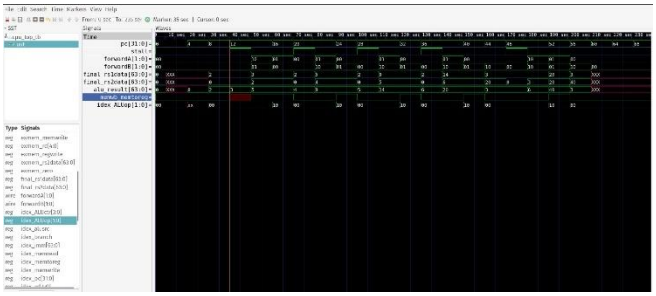
Instructions:

```
ld x5 0(x2)
ld x6 1(x2)
add x4 x5 x6
ld x7 2(x2)
add x4 x4 x7
ld x8 3(x2)
add x4 x4 x8
ld x9 4(x2)
add x4 x4 x9
sd x4 0(x3)
ld x10 0(x3)
add x10 x10 x10
sd x10 0(x3)
```

Output:



Gtkwave:



VI. Conclusion:

This report presents the design and implementation of a RISC-V processor**,** incorporating both sequential and pipelined execution models**.** The sequential implementation, while simple and free from hazards, suffers from performance limitations as each instruction must complete before the next begins. This is particularly evident in **load (**`ld`**)** instructions, which take longer to execute and dictate the overall execution speed.

To overcome these limitations, the pipelined **implementation** improves performance by allowing multiple instructions to execute simultaneously in different stages of the pipeline. However, this introduces **hazards**—data, control, and structural—which require forwarding, hazard detection, and branch handling mechanisms to ensure correct execution. The report discusses the Forwarding Unit,

Hazard Detection Unit, and Branch Hazard Unit**,** which work together to resolve these hazards effectively.

By integrating these techniques, the pipelined processor achieves better instruction throughput and improved **execution efficiency** while maintaining correct program behavior. This project demonstrates the fundamental trade-offs between design complexity and performance, highlighting the necessity of hazard management in modern processor architectures

VII. Contributions:

1. Bhargav:

- Designed the Control Unit for both sequential and pipeline execution and hazard handling.
- Implemented the Program Counter (PC) for instruction sequencing.
- Developed the Instruction Fetch (IF) stage for fetching instructions from memory.
- Developed the Load-Use Hazard Detection Unit to manage pipeline stalls.
- Managed integration and modularity in both sequential and pipelined implementations.
- Generated test cases and instructions to verify correct execution.
- Generating all Testbenches for each module.

2. Karthik:

- Implemented **Data Memory (MEM) stage** for load (ld) and store (sd) operations for both sequential and pipeline processor .
- Designed the Immediate Generator for handling immediate values in instructions for both sequential and pipeline processor.
- Developed the Write-Back (WB) stage for updating register values for both sequential and pipeline processor.
- Developed the Forwarding Unit to resolve data hazards efficiently.
- Generated test cases and debugging the processor.
- Conducted testing and debugging the testbenches to ensure correct execution.

3.Dinesh

- Implemented the Instruction Decode (ID) stage.
- Designed the Register Memory for storing and retrieving operand values.
- Developed the ALU Execution (EX) stage for performing arithmetic and logic operations.
- Implemented the Branch Hazard Unit to handle control hazards in the pipeline.
- Prepared the final report, documenting the design, implementation, and results.