Introduction to Processor Architecture:

RISC-V Processor

Team No: **9**

Aryan Shrivastava - 2023102025

Prakhar Gupta - 2023112019

Somish Singh Nol - 2024122003

International Institute of Information Technology,
Hyderabad

Contents

1 Introduction

Processor architecture focuses on the design and implementation of processors to efficiently execute instructions. Among modern architectures, RISC-V is an open-source Reduced Instruction Set Computing (RISC) architecture that has become popular due to its simplicity, modularity, and extensibility. Its open nature allows for customization and innovation, making it a preferred choice for academic research, embedded systems, and high-performance computing applications.

1.1 RISC-V Instruction Set Architecture

RISC-V is a Load-Store architecture with a fixed instruction length of 32 bits in its base form. It supports several Instruction Formats, including:

- R-Type (Register-Register Format)
- I-Type (Immediate Format)
- S-Type (Store Format)
- B-Type (Branch Format)
- U-Type (Upper Immediate Format)
- J-Type (Jump Format)

The RISC-V Instruction Set Architecture consists of the following key instruction categories:

- **Arithmetic and Logical Instructions**: Perform mathematical and bitwise operations such as addition, subtraction, AND, OR, XOR, and shift operations.
- **Load and Store Instructions**: Handle memory access operations where data is transferred between registers and memory.
- **Control Transfer Instructions**: Manage program flow through conditional and unconditional jumps.
- **Floating Point Instructions**: Enable operations on floating-point registers, provided as an optional extension.
- **System Instructions**: Manage environment and privilege levels, including system calls and exception handling.

1.1.1 Instruction Formats

Each instruction in RISC-V follows a specific format based on its functionality:

- **R-Type**: Used for arithmetic and logical operations where all operands are registers (e.g., `add`, `sub`, `and`).
- **I-Type**: Used for operations involving immediate values, such as arithmetic with constants or load instructions (e.g., `addi`, `lw`).
- **S-Type**: Used for storing register values into memory (e.g., `sw`, `sh`).
- **B-Type**: Used for conditional branch instructions based on register comparisons (e.g., `beq`, `bne`).
- **U-Type**: Used for loading a 20-bit immediate value into the upper part of a register (e.g., `lui`, `auipc`).
- **J-Type**: Used for unconditional jumps (e.g., `jal`).

1.1.2 Instruction Categories

RISC-V instructions are divided into key categories based on their purpose:

- **Arithmetic and Logical Instructions**: Include basic mathematical operations such as addition (`add`), subtraction (`sub`), and bitwise operations like AND (`and`), OR (`or`), XOR (`xor`).
- **Load and Store Instructions**: Used for memory operations where data is loaded from or stored into memory. Examples include `lw` (load word), `sw` (store word), `lb` (load byte), and `sb` (store byte).
- **Control Transfer Instructions**: Handle jumps and branches, such as `beq` (branch if equal), `bne` (branch if not equal), `jal` (jump and link), and `jalr` (jump and link register).
- **Floating-Point Instructions**: Available in optional extensions (e.g., "F" for single-precision floating-point, "D" for double-precision). Example instructions include `fadd.s` (floating-point addition), `fmul.s` (floating-point multiplication), and `fdiv.s` (floating-point division).

- **System Instructions**: Used for system calls, privilege level management, and exception handling. Examples include `ecall` (environment call) and `ebreak` (breakpoint for debugging).

RISC-V's well-structured ISA, with its fixed instruction length and modular extensions, provides simplicity, efficiency, and scalability, making it a widely adopted architecture for embedded systems, high-performance computing, and academic research.

1.2 Processor Architecture Classification

Processor architecture can be classified into two main types: **Sequential** and **Pipelined**.

1.2.1 Sequential Processors

Sequential processors execute one instruction at a time in a step-by-step manner. Each instruction must be fully completed before the next instruction begins execution. This approach ensures simplicity in design and avoids complications such as instruction dependencies. However, it often results in lower performance because the processor remains idle while waiting for an instruction to complete before starting the next one.

1.2.2 Pipelined Processors

Pipelined processors, on the other hand, enhance performance by dividing instruction execution into multiple stages and overlapping the execution of different instructions. Instead of processing one instruction at a time, pipelining allows the processor to work on multiple instructions simultaneously at different execution stages. This significantly improves throughput and overall efficiency.

A typical instruction pipeline consists of several stages, such as:

- **Fetch**: Retrieving the instruction from memory.
- **Decode**: Decoding the instruction and determining the required operations.
- **Execute**: Performing the necessary computations or data manipulations.
- **Memory Access**: Accessing memory if needed (for load/store instructions).
- **Write Back**: Storing the result back into the register file.

By allowing multiple instructions to be in different stages simultaneously, pipelining increases instruction throughput. However, it also introduces challenges such as **pipeline hazards**, which can reduce efficiency.

1.2.3 Pipeline Hazards

Pipeline hazards occur when the execution of one instruction is affected by another instruction still in the pipeline. They can be classified into three types:

- **Structural Hazards**: Arise when two or more instructions require the same hardware resource at the same time.
- **Data Hazards**: Occur when an instruction depends on the result of a previous instruction that has not yet completed.
- **Control Hazards**: Happen when the pipeline needs to determine the next instruction to execute (e.g., branch instructions).

Various techniques, such as forwarding, pipeline stalling, and branch prediction, are used to mitigate these hazards and improve pipeline performance.

Pipelining is a fundamental concept in modern processor design, significantly enhancing execution speed and efficiency while introducing new challenges that must be carefully managed.

2 Arithmetic Logic Unit

We designed/constructed a Arithmetic Logic Unit that implements following operations

1. Addition : 64 bit-wise
2. Subtraction : 64 bit-wise
3. AND : 64 bit-wise
4. OR : 64 bit-wise

Arithmetic Logic Unit receives control signal ALUControl(4-bit value) and produce Result and Zero bit . Here Zero bit is set to 1'b1 when the result of Arithmetic Logic Unit is 64'd0

ALUControl	Operation
0000	AND
0001	OR
0010	Addition
0110	Subtraction

Table 1: ALUControl Decoding

Subtraction is implemented using 2's-complement logic . Which implies if Operation is A - B then A + (2's Complement(B)). 2's Complement of a 64-bit number can be achieved by adding 1'b1 to inverted number. So we simply Invert B and use addition logic with carry -in as 1'b1 .

3 Sequential Implementation

The sequential processor implementation is based on a single-cycle architecture where each instruction is fetched, decoded, executed, and its result written back before the next instruction is processed. Although this design is less efficient compared to pipelined designs, the sequential design serves as a baseline to ensure correct execution of instructions and a clear understanding of the architecture.

Key Components of the Sequential RISC-V Processor

- **Program Counter (PC):** Holds the address of the next instruction to fetch. Increments by the instruction size (typically 4 bytes) and updates for branch instructions.
- **Instruction Memory:** Stores the program instructions. Uses the PC to output the corresponding instruction during the fetch phase.
- **Register File:** Contains 32 registers (`x0` to `x31`), with `x0` hardwired to zero. Provides two read ports and one write port for operands and results.
- **Arithmetic Logic Unit (ALU):** Performs arithmetic (e.g., addition, subtraction) and logical (e.g., AND, OR) operations. Also computes effective addresses for load and store operations and generates a zero flag for branch decisions.
- **Data Memory:** Used for load (`ld`) and store (`sd`) instructions. The effective address calculated by the ALU is used for accessing the memory.
- **Control Unit:** Decodes instructions and generates the necessary control signals. It directs data flow, selects ALU operations, manages memory read/write, and handles branch decision logic.

Instruction Execution Flow

The sequential execution flow for each instruction in the processor consists of the following steps:

Step 1. Instruction Fetch:
The instruction at the address specified by the Program Counter (PC) is fetched from the Instruction Memory.

Step 2. Instruction Decode and Operand Fetch:
The fetched instruction is decoded by the Control Unit to determine the operation type (R-type, I-type, S-type, or B-type) and to extract operand registers and immediate values.
The required operands are read from the Register File. For branch instructions (e.g., `beq`), the operands are fetched for comparison.

Step 3. Execution:
The Arithmetic Logic Unit (ALU) performs the necessary arithmetic or logical operation:

- For **arithmetic/logic instructions** (e.g., `add, sub, and, or`), the ALU computes the result.
- For **load/store instructions** (`ld, sd`), the ALU calculates the effective memory address by adding the base register to the immediate offset.
- For the **branch instruction** (`beq`), the ALU checks if the operands are equal (by subtracting and verifying if the result is zero).

Step 4. Memory Access:
Depending on the instruction type:

- **Load (`ld`):** The computed effective address is used to read data from Data Memory.
- **Store (`sd`):** The computed effective address is used to write data from the Register File to Data Memory.
- For other instructions, no memory access is performed.

Step 5. Write-Back:
The result from the ALU or data fetched from memory (in the case of `ld`) is written back to the destination register in the Register File.

Step 6. PC Update:
The Program Counter (PC) is updated to the next instruction:

- Normally, the PC increments by 4.

– For branch instructions (e.g., `beq`), if the branch condition is met, the PC is updated with the computed branch target address.

4 Test for Sequential Implementation

4.1 Instruction Set

```
ld x4, 0(x0)
00000000000000000011001000000011

ld x5, 8(x0)
00000000100000000011001010000011

ld x6, 16(x0)
00000001000000000011001100000011

beq x4, x5, label1
00000000010000101000101001100011

add x5, x4, x5
00000000010100100000001010110011

beq x5, x6, label2
00000000010100110000011001100011

sub x0, x0, x0
00000000010000000000000000110011

add x0, x0, x0
00000000000000000000000000110011

sub x0, x0, x0
00000000010000000000000000110011

add x0, x0, x0
00000000000000000000000000110011

label2
sub x7, x6, x4
00000000010000110000001110110011
```

```
and x8, x7, x4
00000000010000111111010000110011

or x9, x4, x0
00000000000000010011001001011001 1

sd x9, 24(x0)
00000000011000000000110000100011
```

4.2 Flow Of Processes for this Example

Below is a step-by-step flow of the provided instructions. We assume:

- A 32-register file (`x0` through `x31`), each 64 bits wide.
- Instruction memory (IM) and data memory (DM) are each 512 bytes, byte-addressable, with addresses from 0 to 511.
- Each instruction is 4 bytes. The first instruction is at IM address 0, the second at IM address 4, etc.
- `x0` is always zero in RISC-V.

Instruction 1: `ld x4, 0(x0)`

- **Address in IM:** 0 (binary: `00000000000000000011001000000011`).
- **Action:** Loads a 64-bit value from DM address 0 into register `x4`.
- **Note:** Since `x0` is 0, the effective address is 0.

Instruction 2: `ld x5, 8(x0)`

- **Address in IM:** 4.
- **Action:** Loads a 64-bit value from DM address 8 into register `x5`.

Instruction 3: `ld x6, 16(x0)`

- **Address in IM:** 8.
- **Action:** Loads a 64-bit value from DM address 16 into register `x6`.

Instruction 4: `beq x4, x5, label`

- **Address in IM:** 12.
- **Action:** Compares `x4` and `x5`. If they are equal, branch to `label`; otherwise, continue to the next instruction.
- **Note:** The branch target (`label`) is encoded as an offset in the instruction. If not taken, PC advances by 4.
- ***NOTE*:** THE VALUES WE HAVE GIVEN AS TEST-CASE IN THAT THIS BRANCH OPERATON WILL WORK MEAN X5 AND X4 WILL BE NOT EQUAL

Instruction 5: `add x5, x4, x5`

- **Address in IM:** 16.
- **Action:** `x5` ← `x4` + `x5`.
- **Note:** This instruction only executes if the previous branch is *not* taken.

Instruction 6: `beq x5, x6, 20`

- **Address in IM:** 20.
- **Action:** Compares `x5` and `x6`. If they are equal, branch to `label2`; otherwise, continue.
- ***NOTE*:** THE VALUES WE HAVE GIVEN AS TEST-CASE IN THAT THIS BRANCH OPERATON WILL WORK MEAN X5 AND X6 WILL BE EQUAL

Instruction 7: `sub x0, x0, x0`

- **Address in IM:** 28.
- **Action:** `x0` ← `x0` - `x0` = 0.
- **Note:** Also a no-op, since `x0` is always 0.

Instruction 8: `add x0, x0, x0`

- **Address in IM:** 24.
- **Action:** `x0` remains 0 (adding 0 to 0).
- **Note:** Effectively a no-op.

Instruction 9: `sub x0, x0, x0`

- **Address in IM:** 28.
- **Action:** `x0` ← `x0` - `x0` = 0.
- **Note:** Also a no-op, since `x0` is always 0.

Instruction 10: `add x0, x0, x0`

- **Address in IM:** 24.
- **Action:** `x0` remains 0 (adding 0 to 0).
- **Note:** Effectively a no-op.
- ***NOTE*:** BRANCHED EXECUTION WILL BE START FROM BELOW INSTRUCTION.

Instruction 11: `sub x7, x6, x4`

- **Address in IM:** 32.
- **Action:** `x7` ← `x6` - `x4`.

Instruction 12: `and x8, x7, x4`

- **Address in IM:** 36.
- **Action:** `x8` ← `x7` `AND` `x4`.

Instruction 13: `or x9, x4, x0`

- **Address in IM:** 40.
- **Action:** `x9` ← `x4` `OR` `x0` = `x4`.

Instruction 14: `sd x9, 24(x0)`

- **Address in IM:** 44.
- **Action:** Stores the 64-bit value in `x9` to DM address 24.

Branch Behavior Depending on the values loaded into `x4`, `x5`, and `x6`:

- `beq x4, x5, label` may branch if `x4` = `x5`.
- `beq x5, x6, label2` may branch if `x5` = `x6`.

If a branch is taken, the PC jumps to the indicated label; otherwise, execution proceeds linearly to the next instruction address.

4.3 GTKwave for each stage

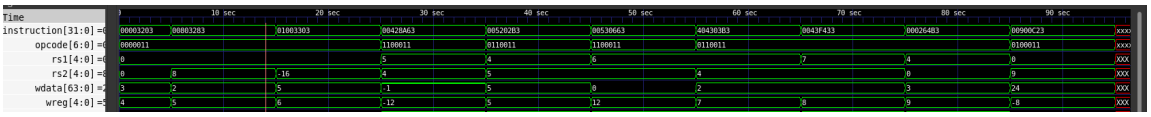

Figure 1: Fetch

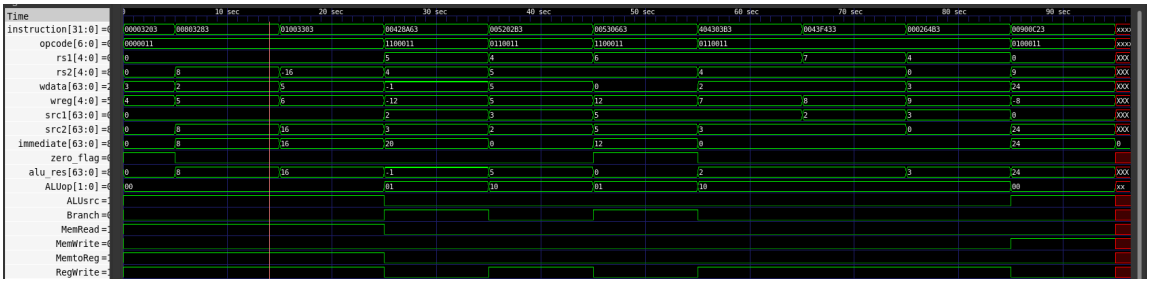

Figure 2: Decode Stage

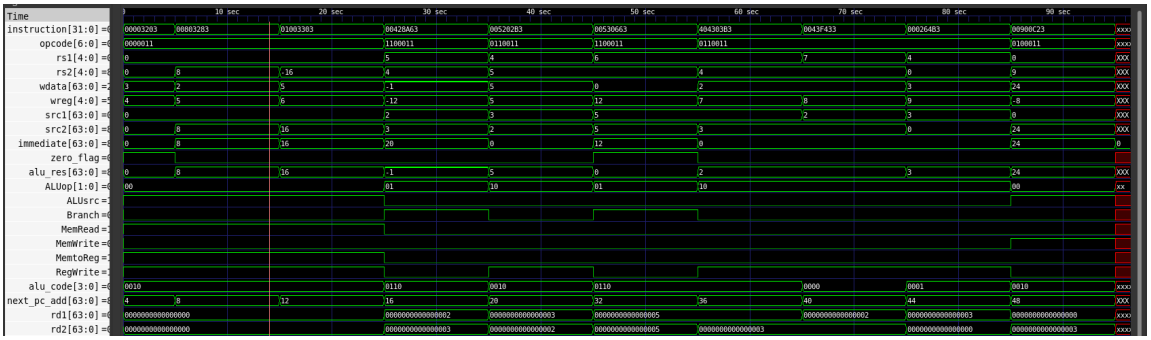

Figure 3: Execute Stage

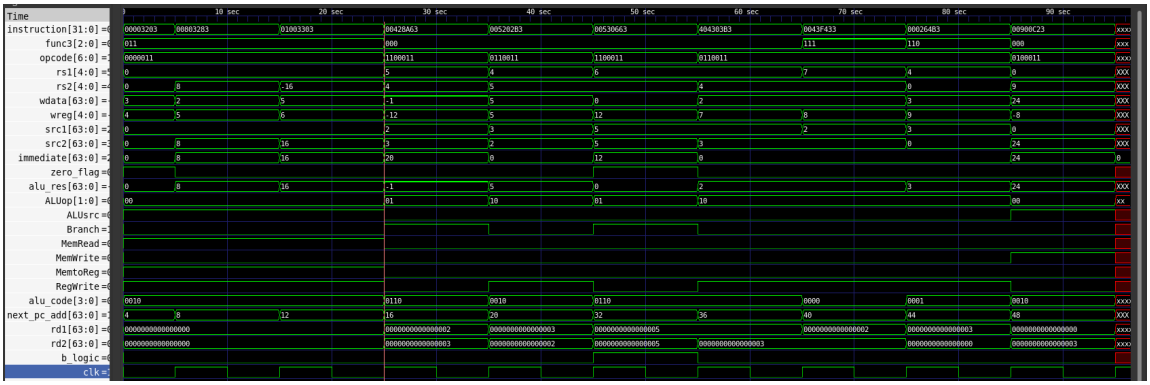

Figure 4: Full With Memory

5 Pipelining Implementation

In a pipelined processor, the execution of instructions is divided into several distinct stages. Each stage is responsible for a specific part of the instruction processing. Rather than waiting for one instruction to complete all stages before starting the next, multiple instructions can be processed concurrently, each at a different stage.

Stages in a Pipelined RISC-V Processor

- **IF: Instruction Fetch from Memory**
  - The processor retrieves the next instruction from memory using the current Program Counter (PC).
  - The PC is then incremented to point to the next instruction.
  - This stage may include simple branch prediction.
- **ID: Instruction Decode & Register Read**
  - The instruction is decoded to determine its type and required operations.
  - Register values needed for execution are read from the register file.
  - Control signals are generated for subsequent stages.
- **EX: Execute Operation or Calculate Address**
  - For arithmetic/logic instructions, the ALU performs the specified operation.
  - For memory instructions, the effective memory address is calculated.
  - For branch instructions, the branch condition is evaluated and the target address is computed.
  - This stage implements the core computation of the instruction.
- **MEM: Access Memory Operand**
  - For load instructions, data is read from memory at the calculated address.
  - For store instructions, data is written to memory at the calculated address.
  - For non-memory instructions, this stage is effectively bypassed.
- **WB: Write Result Back to Register**

  - Results from ALU operations or memory loads are written back to the destination register.
  - This completes the instruction execution.

Pipeline Registers in a 5-Stage RISC-V Pipeline

In a 5-stage pipeline, there are four sets of pipeline registers that store the intermediate information between stages:

- **IF/ID Register (Between Fetch and Decode)**
  - Stores the fetched instruction.
  - Holds the incremented PC value (PC+4).
  - May also store the PC of the current instruction (useful for branch calculations).
- **ID/EX Register (Between Decode and Execute)**
  - Stores register values read from the register file.
  - Contains immediate values.
  - Carries control signals for the Execute, Memory, and Write Back stages.
  - Holds the destination register addresses.
  - Contains function codes for the ALU.
  - Stores the PC value for branch/jump instructions.
- **EX/MEM Register (Between Execute and Memory)**
  - Holds the ALU result or computed address.
  - Contains data to be written to memory (for store instructions).
  - Stores the branch outcome (taken/not taken).
  - Carries control signals for the Memory and Write Back stages.
  - Retains the destination register address.
- **MEM/WB Register (Between Memory and Write Back)**
  - Stores data read from memory (for load instructions).

  - Holds the ALU result when no memory access is needed.
  - Carries control signals for the Write Back stage.
  - Retains the destination register address.

Example Content in Pipeline Registers

For an `add x3, x1, x2` instruction flowing through the pipeline:

- **IF/ID Register**
  - `Instruction: add x3, x1, x2` (32-bit binary).
  - `PC+4:` Address of the next instruction.
- **ID/EX Register**
  - `Register Values:` Contents of `x1` and `x2`.
  - `Control Signals:` ALU should perform addition, no memory operation, register write enabled.
  - `Destination Register: x3`.
  - `ALU Function: ADD`.
- **EX/MEM Register**
  - `ALU Result:` Sum of `x1` and `x2`.
  - `Control Signals:` No memory operation, register write enabled.
  - `Destination Register: x3`.
- **MEM/WB Register**
  - `Write Data:` Sum of `x1` and `x2` (same as the ALU result).
  - `Control Signals:` Register write enabled.
  - `Destination Register: x3`.

Importance of Registers in Handling Hazards

Pipeline registers are essential for implementing hazard detection and resolution:

- **Forwarding (Bypassing):** They store information needed to detect data dependencies. When a dependency is identified, forwarding logic can route data directly between pipeline stages, bypassing the register file.
- **Stalling:** Registers can hold their values when a stall is required, effectively freezing parts of the pipeline.
- **Flushing:** In the event of a branch misprediction, certain registers may be cleared or invalidated to flush out incorrect speculative instructions.

Instruction Execution Flow

- **Clock Cycle 1: Instruction Fetch (IF)**
  - The Program Counter (PC) provides the address of the instruction.
  - The instruction memory is accessed.
  - The instruction is fetched.
  - The PC is updated to point to the next instruction.
- **Clock Cycle 2: Instruction Decode (ID)**
  - The fetched instruction is decoded.
  - Necessary registers are read from the register file.
  - Immediate values are extended.
  - Control signals are generated.
  - Hazard detection is performed.
- **Clock Cycle 3: Execute (EX)**
  - The ALU executes the operation based on the opcode.
  - For R-type instructions: Performs register-to-register operations (e.g., add, sub).
  - For I-type instructions: Carries out register-immediate operations or address calculations.

  - For branch instructions: Evaluates the condition and computes the target address.
  - For jump instructions: Calculates the target address.

- **Clock Cycle 4: Memory Access (MEM)**
  - For load instructions: Data is read from memory using the address computed in the EX stage.
  - For store instructions: Data is written to memory.
  - For other instructions: This stage is simply passed through.

- **Clock Cycle 5: Write Back (WB)**
  - The results from the ALU or memory are written back to the destination register.
  - The instruction execution is completed.

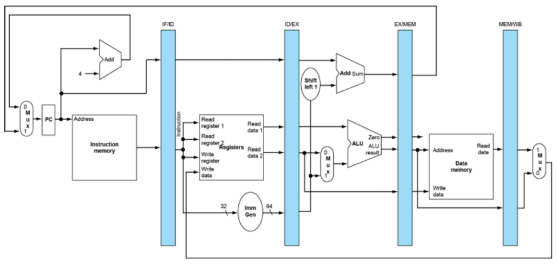

Figure 5: Pipeline Instruction Flow

5.1 GTKwave for each stage (PIPELINE)

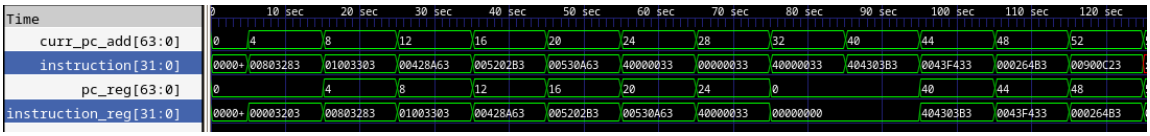

Figure 6: Fetch Unit (1st Stage)

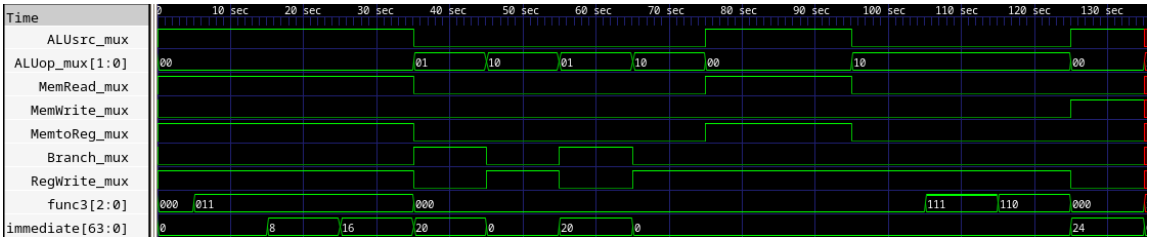

Figure 7: Decode Unit (2nd stage)

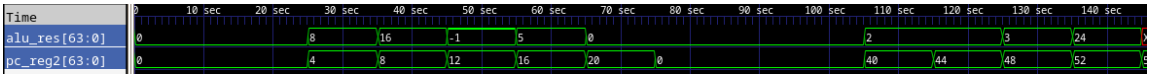

Figure 8: Execute (3rd stage)

Figure 9: MEM (4th stage)

Figure 10: Write Back (5th Stage)

6 Hazard Handling

- We have handled the Hazards in our processor which includes Load-use data hazard handling and Control hazard handling.
- For Load-Use Hazard, we are supposed to stall the processor for one cycle and then forwarding for the next cycle. So that we can correctly use the register data being loaded in the previous instruction.
- For Control Hazard handling, we will assume that all the branches are untaken and proceed with the next instruction.
- Then we will check whether we were supposed to take the branch or not.
- If we are supposed to take the branch then we will flush the IF/ID regsiter and ID/EX register and then load the correct instruction.

7 Contribution

7.1 Sequential Implementation

7.1.1 Aryan Shrivastava

- Control Unit Generation
- ALU control unit and ALU
- Program Counter correct updation and selection Logic

7.1.2 Prakhar Gupta

- Connection of Stage 1 and Stage 2.
- Creating instruction and data test file
- Immediate Generator

7.1.3 Somish Singh Nol

- Register File Unit
- Data Memory Unit
- Instruction Unit

7.2 Pipelined Implementation

7.2.1 Aryan Shrivastava

- Stage 5 - WB Register
- Hazard Detection - Load Use and Stalling
- Control Hazard handling
- Forwarding unit

7.2.2 Prakhar Gupta

- Complete Stage 1 and Stage 2
- Stage 1 - IF and Stage 2 - ID
- Register Implementation and Data Storage

7.2.3 Somish Singh Nol

- Complete Stage 3 and Stage 4
- Stage 3 - EX and Stage 4 - MEM
- Register Implementation and Data Storage