Introduction to Processor Architecture:

RISC-V Processor

Team : **17**

Aasrith Reddy Vedanaparti - 2023102031

Aditya Peketi - 2024122001

Akshat Puneet - 2024122005

International Institute of Information Technology
Hyderabad

Contents

1 Introduction

Processor architecture studies the Design and Implementation of Processors to execute instructions efficiently. Among Modern architectures, RISC-V is an open-source, Reduced Instruction Set Computing (RISC) Architecture that has gained popularity for its simplicity, modularity, and extensibility.

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
- I-Type
- S-Type
- B-Type
- U-Type
- J-Type

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

- **Arithmetic and Logical Instructions**: Addition, subtraction, Bit-wise operations.
- **Load and Store Instructions**: Memory access operations.
- **Control Transfer Instructions**: Conditional and unconditional jumps.
- **Floating Point Instructions**: Operations on floating-point registers (optional extension).
- **System Instructions**: Environment and privilege level management.

1.2 Processor Design

Processor architecture can be classified into two main types: **Sequential** and **Pipelined**.
Sequential Processors execute one instruction at a time, while Pipelined Processors divide the instruction execution into multiple stages and overlap the execution of different instructions. Pipelined Processors can achieve higher performance and efficiency than sequential processors. Still, they also introduce challenges such as pipeline hazards, which occur when the execution of one instruction depends on the outcome of another instruction still in the pipeline.

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 Explanation of each Module implemented in Verilog

4.1 Fetch.v

This Verilog module implements the *fetch* stage of a RISC-V-like processor. It consists of three main modules:

1. `PC` (Program Counter)
2. `Inst_Memory` (Instruction Memory)
3. `Fetch` (Top-level fetch stage)

Below is a brief description of each module, including its inputs, outputs, and functionality.

1. PC Module

- **Inputs:**
  - `Clk`: Clock signal
  - `reset`: Active-high reset signal
  - `nxt_pc`: Next PC value to be loaded on the rising edge of the clock
- **Output:**
  - `PC`: Current PC value (64-bit)
- **Functionality:**
  - On reset, `PC` is initialized to `0`.
  - On each rising edge of the clock, if `reset` is not asserted, `PC` is updated with the value of `nxt_pc`.
  - This module effectively holds the address of the current instruction to be fetched.

2. Inst_Memory Module

- **Input:**
  - `address` (64-bit): Address from which to fetch the instruction
- **Output:**
  - `instruction` (32-bit): Instruction word stored at the given address
- **Functionality:**
  - Implements a simple memory array `memory[255:0]`, each location storing a 32-bit instruction.
  - Several instructions (e.g., ADD, SUB, AND, OR, BEQ, LD, SD) are pre-loaded in the `initial` block.
  - The instruction is selected by `address[63:2]`, effectively ignoring the lower two bits (since instructions are word-aligned).
  - Outputs the 32-bit instruction for the requested address.

3. Fetch Module (Top-Level)

- **Inputs:**
  - `Clk`: Clock signal
  - `reset`: Active-high reset signal
  - `pc_next` (64-bit): The next PC value to be used
- **Outputs:**

  - `PC` (64-bit): Current PC value (exposed externally)
  - `instruction` (32-bit): The instruction fetched from memory at the current PC

- **Functionality:**
  - Instantiates the `PC` module (`Prog_counter`) to hold the current instruction address.
  - Instantiates the `Inst_Memory` module (`imem`) to read out the 32-bit instruction from the memory array.
  - Connects `PC` to the `address` port of the instruction memory.
  - Continuously assigns the outputs:
    * `PC` ← `instruction_addrs` (the PC output from the `PC` module)
    * `instruction` ← `fetch_out` (the instruction read from memory)

Overall Flow

1. On every clock cycle, the `PC` module updates the current PC (unless reset is asserted).
2. The `Inst_Memory` module uses the current PC value to fetch the corresponding 32-bit instruction.
3. The `Fetch` module outputs both the `PC` and the fetched `instruction` to the rest of the processor (e.g., decode stage).

This completes the fetch stage, where each cycle, one new instruction is fetched based on the updated program counter.

4.2 Decode.v

This verilog module provides the logic for the *decode* stage of a RISC-V-like processor and includes four modules: `Control`, `Register_file`, `ImmediateGenerator`, and `Decode` (the top-level decode module). Below is an explanation of each module, including their inputs, outputs, and functionality.

1. Control Module

File: `Control.v`

- **Inputs:**
  - `Instruction` (32 bits): The fetched instruction word whose opcode bits are used to generate control signals.
- **Outputs:**
  - `Branch`: Asserted for branch instructions (`beq`).
  - `MemRead`: Enables memory read for load instructions.
  - `MemtoReg`: Selects whether data comes from memory or ALU to write back to registers.

  - `ALUOp` (2 bits): Encodes the desired ALU operation (e.g., `00` for add, `01` for subtract, etc.).
  - `MemWrite`: Enables memory write for store instructions.
  - `ALUSrc`: Chooses between register operand or immediate operand for ALU.
  - `RegWrite`: Enables register file writes for instructions that produce a register result.

- **Functionality:**
  - Decodes the `opcode` (bits `[6:0]`) from the 32-bit instruction.
  - Sets all control signals to default values, then updates them based on the recognized instruction type:
    - `r_type` (`0110011`): ALUOp is `10`, `RegWrite` is asserted.
    - `ld` (`0000011`): Memory read is asserted, ALUSrc is set to use an immediate, etc.
    - `sd` (`0100011`): Memory write is asserted, ALUSrc is set to use an immediate.
    - `beq` (`1100011`): Branch is asserted, ALUOp is set to `01`.
  - A default case sets all signals to zero if the opcode is unrecognized.

2. Register_file Module

File: `Register_file.v`

- **Inputs:**
  - `clk`: Clock signal.
  - `RegWrite`: Control signal indicating whether to write to the register file.
  - `rs1`, `rs2` (5 bits each): Indices of source registers to read.
  - `rd` (5 bits): Index of the destination register to write.
  - `write_data` (64 bits): Data to be written into register `rd` (if `RegWrite` is asserted).

- **Outputs:**
  - `read_data1` (64 bits): Contents of register `rs1`.
  - `read_data2` (64 bits): Contents of register `rs2`.

- **Functionality:**
  - Implements an array of 32 registers, each 64 bits wide.
  - `x0` (register 0) is always zero.
  - On negative clock edge, if `RegWrite` is asserted and `rd` is not zero, the `write_data` is stored into register `rd`.
  - Initializes register values from a file `registers.txt` if it exists; otherwise, it creates the file and writes initial values to it.
  - Continuously outputs the values of registers `rs1` and `rs2` as `read_data1` and `read_data2`.

3. ImmediateGenerator Module

File: `ImmediateGenerator.v`

- **Input:**
  - `Instruction` (32 bits): The fetched instruction whose immediate field is to be extracted.
- **Output:**
  - `imm_value` (64 bits): The sign-extended immediate value derived from the instruction.
- **Functionality:**
  - Examines the `opcode` bits (`[6:0]`) to determine the instruction format (I-type, S-type, B-type).
  - Extracts and sign-extends the relevant bits of the instruction to form a 64-bit immediate:
    - `I_Type` / `I_Type_ld`: Bits `[31:20]`.
    - `S_Type`: Bits `[31:25]` and `[11:7]`.
    - `B_Type`: Combines bits `[31]`, `[7]`, `[30:25]`, and `[11:8]`, plus a trailing 0.
  - For unrecognized opcodes, `imm_value` is set to zero.

4. Decode Module (Top-Level)

File: `Decode.v`

- **Inputs:**
  - `Clk`: Clock signal.
  - `RegWrite`: Control signal indicating whether the destination register should be written.
  - `Instruction` (32 bits): The instruction to be decoded.
  - `write_data` (64 bits): Data to be written to the register file if `RegWrite` is asserted.
- **Outputs:**
  - `A` (64 bits): Value read from source register `rs1`.
  - `B` (64 bits): Value read from source register `rs2`.
  - `C` (64 bits): Immediate value generated by the `ImmediateGenerator`.
- **Functionality:**
  - Extracts the register fields `rd`, `rs1`, `rs2` from the `Instruction`.
  - Instantiates the `Register_file` module to read the values of `rs1` (`A`) and `rs2` (`B`), and to write to `rd` if `RegWrite` is set.
  - Instantiates the `ImmediateGenerator` module to produce the sign-extended immediate `C`.

Overall Decode Stage Flow

1. The `Decode` module reads the source registers (`rs1`, `rs2`) and generates the immediate value based on the instruction format.
2. The `Control` signals (not shown directly in `Decode.v`) guide whether the processor will use the register outputs (`A`, `B`) or the immediate (`C`) in subsequent stages (e.g., ALU input).
3. If `RegWrite` is asserted, the `write_data` is stored into register `rd` on the negative edge of the clock.

4.3 Execute.v

1 . Execute

- **Inputs:**
  - `A` : First operand for ALU operations.
  - `B` : Second operand for ALU operations (register value).
  - `C` : Immediate value used in ALU operations or branch calculations.
  - `PC`: Program counter value used for calculating target addresses.
  - `funct3`: Function code used to determine ALU operations.
  - `funct7`: Extended function code for specific ALU operations.
  - `ALUOp` : Control signal defining the ALU operation type.
  - `ALUSrc`: Select signal to determine if the second ALU operand comes from `B` (register) or `C` (immediate).
- **Outputs:**
  - `Zero` : High when ALU output is zero (used for branch decisions).
  - `Target`: Computed branch target address.
  - `ALU_result` : Computed result of the ALU operation.
- **Functionality:**
  - `ALU Control Computation:`
    - The ALU control unit determines the ALU operation based on `ALUOp`, `funct7`, and `funct3`.
  - `Operand Selection:`
    - A 2-to-1 multiplexer (`MUX_21`) selects between register value `B` and immediate value `C` based on `ALUSrc`.
    - The selected value is passed as the second operand to the ALU.
  - `ALU Computation:`
    - The ALU performs the operation specified by `ALUcontrol` and produces `result`.
    - It also generates the `zero` signal (`Zo`) to indicate if the result is zero.

- `Branch Address Calculation:`
  - The immediate value `C` is shifted left by one using the `Shiftleft` module.
  - A `full_adder` computes the target branch address as:

$$\texttt{Solution} = \texttt{PC} + (\texttt{C} \ll 1)$$

- `Final Output Assignment:`
  - `ALU_result` stores the ALU computation result.
  - `Target` stores the computed branch address.
  - `Zero` is updated based on the ALU operation result.

2 . MUX_21

- `Inputs:`
  - `B` : Register value that may be selected as the second ALU operand.
  - `C` : Immediate value that may be selected as the second ALU operand.
  - `ALUSrc`: Control signal that determines whether `B` or `C` is selected.
- `Outputs:`
  - `Y` : Selected output based on `ALUSrc`, either `B` (register) or `C` (immediate).
- `Functionality:`
  - `Multiplexer Operation:`
    - If `ALUSrc` is high (`1`), the immediate value `C` is selected as the output.
    - If `ALUSrc` is low (`0`), the register value `B` is selected as the output.
  - `Selection Logic:`
    - The selection is implemented using an `always` block that continuously monitors changes in inputs.
    - The output `Y` is updated based on the value of `ALUSrc`.

3 . Shift left

- `Inputs:`
  - `C` : Input value to be shifted left by one bit.
- `Outputs:`
  - `S` : Output value after shifting `C` left by one bit.
- `Functionality:`
  - `Bit wise Left Shift:`
    - The input `C` is shifted left by one bit.

    - Each bit of `C` is moved to the next higher index in `S`.
    - The least significant bit (`S[0]`) is set to `0`.
  - `Loop-Based Implementation:`
    - A `for` loop iterates from bit index 0 to 62.
    - Each bit at position `j` in `C` is assigned to position `j+1` in `S`.
    - The most significant bit (`S[63]`) is implicitly set to 0.
  - `Combinational Logic:`
    - The operation is performed in an `always` block sensitive to any changes in `C`.
    - The shifting is done purely combinationally, without clock dependency.

4 . alu_control

- `Inputs:`
  - `ALUOp` : 2-bit control signal that defines the type of ALU operation.
  - `Funct7` : 7-bit function code used for distinguishing between different ALU operations.
  - `Funct3` : 3-bit function code used to determine the specific ALU operation.
- `Outputs:`
  - `Operation` : 4-bit control signal specifying the ALU operation.
- `Functionality:`

ALU Control	Function
0000	AND
0001	OR
0010	add
0110	subtract

Table 2: ALU control signals and corresponding functions

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

Table 3: 2-bit ALUOp derived from opcode

4.4 Data Memory .v

Data_memory

- **Inputs and Outputs**
  - The module takes `Clk`, `address`, `Data_write`, `MemWrite`, and `MemRead` as inputs.
  - It produces `Data_read` as output. The `address` specifies the memory location, while `MemRead` and `MemWrite` control read and write operations.
- **Functionality**
  - Memory is implemented as an array of 1024 bytes, initialized to zero.
  - Write operations occur on the rising edge of `Clk` when `MemWrite` is high, storing 64-bit data in little-endian format.
  - Read operations occur asynchronously when `MemRead` is high, retrieving 64-bit data from memory.
- **Behavior**
  - If `MemWrite` is enabled, data is stored byte by byte across consecutive addresses.
  - When `MemRead` is high, data is reconstructed from memory. If neither signal is active, `Data_read` remains zero.

4.5 Next PC

Next_PC Module

- **Inputs**
  - **Branch** (1-bit) – Control signal indicating whether branching is enabled.
  - **Zero** (1-bit) – Flag indicating if a branch condition is met.
  - **PC** (64-bit) – Current program counter value.
  - **Target** (64-bit) – Target address for branching.
- **Output**
  - **next_pc** (64-bit) – The computed next PC value.
- **Internal Signals**
  - **w1** – Result of AND operation between `Branch` and `Zero`.
  - **carry** – Carry-out from the full adder computing `PC + 4`.
  - **PC_4** – Incremented program counter (`PC + 4`).
  - **Solution** – Final selected next PC value.
- **Functionality**

- **Branch Decision:** The signal `w1` is computed using an AND gate:

$$w1 = \text{Branch} \wedge \text{Zero}$$

  This ensures branching occurs only when both conditions are met.

- **Incrementing PC:** A full adder computes:

$$\text{PC\_4} = \text{PC} + 4$$

  This represents the next instruction in sequential execution.

- **Selecting the Next PC:** A 2-to-1 multiplexer selects between `PC + 4` and `Target` based on `w1`:

$$\text{Solution} = \begin{cases} \text{Target}, & \text{if } w1 = 1 \\ \text{PC} + 4, & \text{otherwise} \end{cases}$$

5 Test for Sequential Implementation

5.1 Provided Instruction Set

ADD x1, x2, x3
`0000000 00011 00010 000 00001 0110011`
ADD x4, x5, x6
`0000000 00101 00110 000 00100 0110011`
SUB x7, x4, x1
`0100000 00001 00100 000 00111 0110011`
AND x8, x5, x6
`0000000 00110 00101 111 01000 0110011`
SW x13, 16(x10)
`0000000 01101 01010 011 10000 0100011`
Stores word at memory address x10 + 16
LD x7, 16(x10)
`000000010000 01010 011 00111 0000011`
Loads word from memory address x10 + 16 into x7
OR x8, x7, x9
`0000000 00111 01001 110 01000 0110011`
AND x5, x5, x6
`0000000 00110 00101 111 00101 0110011`
ADD x20, x20, x3
`0000000 10100 00011 000 10100 0110011`
ADD x20, x20, x21
`0000000 10100 10101 000 10100 0110011`
ADD x20, x20, x22
`0000000 10100 10110 000 10100 0110011`
BEQ x23, x22, +4
`000000 10111 10110 000 00100 1100011`
Branches to the instruction at offset 8 if x23 = x22
AND x20, x5, x6
`0000000 00110 00101 111 10110 0110011`
(Not executed due to BEQ instruction)
SUB x26, x25, x24
`0100000 11000 11001 000 11010 0110011`
SW x22, 16(x10)
`0000000 10110 01010 011 10000 0100011`
Stores word x22 at memory address 16 + x10
SW x20, 16(x10)
`0000000 10100 01010 011 10000 0100011`
Stores word x20 at memory address 16 + x10
LD x7, 16(x10)
`000000010000 01010 011 00111 0000011`
Loads word 16 + x10 from memory address into x7

5.2 Provided Register values(Initial)

```
0  0000000000000000
1  0000000000000001
2  0000000000000002
3  0000000000000003
4  0000000000000004
5  0000000000000005
6  0000000000000006
7  0000000000000007
8  0000000000000008
9  0000000000000009
10 0000000000000002
11 000000000000000b
12 000000000000000c
13 0000000000000006
14 0000000000000009
15 000000000000000f
16 0000000000000010
17 0000000000000011
18 0000000000000000
19 0000000000000000
20 0000000000000001
21 0000000000000009
22 000000000000000a
23 000000000000000a
24 0000000000000001
25 0000000000000000
26 0000000000000000
27 0000000000000000
28 0000000000000000
29 0000000000000000
30 0000000000000000
31 0000000000000000
```

5.3 Obtained Register values after Execution

After each Instructions **expected outputs** are

Instruction Executed	Register Edited	Value printed
Instruction - 1	x1	5
Instruction - 2	x4	11
Instruction - 3	x7	6
Instruction - 4	x8	4
Instruction - 5	–	Store
Instruction - 6	x7	6
Instruction - 7	x8	15
Instruction - 8	x5	4
Instruction - 9	x20	4
Instruction - 10	x20	13
Instruction - 11	x20	23
Instruction - 12	–	Branch
Instruction - 13	–	Not Executed
Instruction - 14	x26	-1
Instruction - 15	–	Store
Instruction - 16	–	Store
Instruction - 17	x7	23

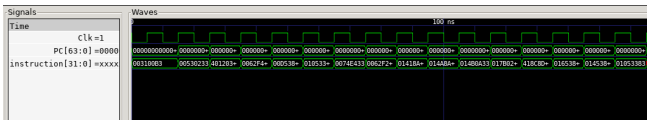
But there are some registers edited multiple times like x7 , x20 .
Finally the obtained register values after execution are shown below .

```
0  0000000000000000
1  0000000000000005
2  0000000000000002
3  0000000000000003
4  000000000000000b
5  0000000000000004
6  0000000000000006
7  0000000000000017
8  000000000000000f
9  0000000000000009
10 0000000000000002
11 000000000000000b
12 000000000000000c
13 0000000000000006
14 0000000000000009
15 000000000000000f
16 0000000000000010
17 0000000000000011
18 0000000000000000
19 0000000000000000
20 0000000000000017
```

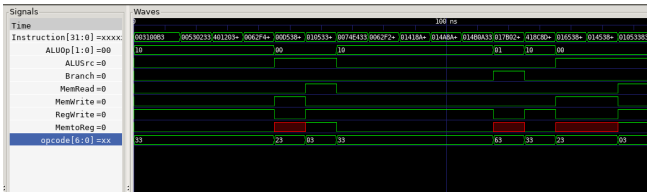
```
21 0000000000000009
22 000000000000000a
23 000000000000000a
24 0000000000000001
25 0000000000000000
26 ffffffffffffffff
27 0000000000000000
28 0000000000000000
29 0000000000000000
30 0000000000000000
31 0000000000000000
```

5.4 Output plots Obtained after each Block

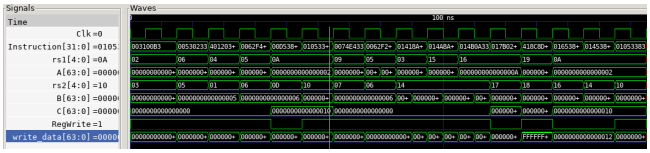
5.4.1 Fetch Module



5.4.2 Control Module



5.4.3 Decode Module



5.4.4 Execute Module

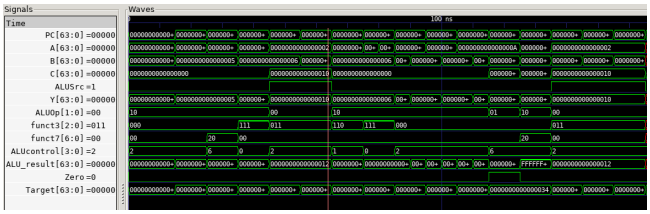
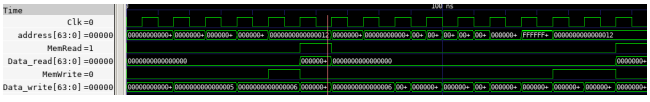
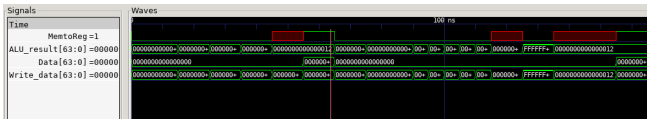


Figure 1: ***Stick Diagram***

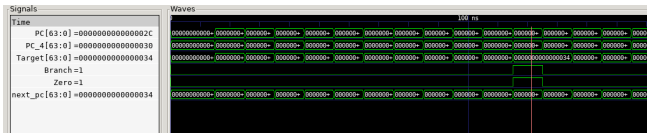
5.4.5 Memory Module



5.4.6 WriteBack Module



5.4.7 NextPC Update Module



6 Pipelining Implementation

In the pipeline process implementation, the execution of instructions is broken down into several sequential stages, with each stage handling a different part of the instruction execution process. Instead of waiting for one instruction to complete all stages before starting the next, multiple instructions can be in different stages of execution simultaneously.

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 memory address is calculated.
  - For branch instructions: The branch condition is evaluated and the target address is calculated.
  - 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 RISC-V pipeline, there are four sets of pipeline registers:

- **IF/ID Register (Between Fetch and Decode)**
  - Stores the fetched instruction.
  - Stores the incremented PC value (PC+4).
  - May store the PC of the current instruction (for branch calculation).
- **ID/EX Register (Between Decode and Execute)**
  - Stores register values read from the register file.
  - Stores immediate values.
  - Stores control signals for Execute, Memory, and Writeback stages.
  - Stores register destination addresses.
  - Stores function codes for the ALU.
  - Stores the PC value (for branch/jump instructions).
- **EX/MEM Register (Between Execute and Memory)**
  - Stores ALU result/computed address.
  - Stores data to be written to memory (for store instructions).
  - Stores branch result (taken/not taken).
  - Stores control signals for Memory and Writeback stages.
  - Stores register destination address.
- **MEM/WB Register (Between Memory and Writeback)**
  - Stores data read from memory (for load instructions).
  - Stores ALU result that bypassed memory.
  - Stores control signals for the Writeback stage.
  - Stores register destination address.

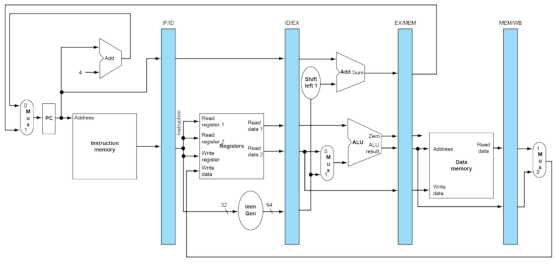


Figure 2: Registers in Pipelined RISC V Processor

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 add, 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 ALU result).
  - Control signals: Register write enabled.
  - Destination register: `x3`.

Importance of Registers in Handling Hazards

Pipeline registers are critical for implementing hazard detection and resolution:

- **Forwarding (Bypassing)** The pipeline registers hold information needed to detect data dependencies. When a dependency is detected, forwarding logic can route data from one pipeline register directly to a functional unit, bypassing the register file.
- **Stalling** Pipeline registers can be controlled to hold their values (not update) when a stall is required, effectively freezing parts of the pipeline.
- **Flushing** After a branch misprediction, certain pipeline registers can be cleared/invalidated to flush incorrect speculative instructions.

Instruction Execution Flow

- **Clock Cycle 1: Instruction Fetch (IF)**
  - Program Counter (PC) provides address of instruction.
  - Instruction memory is accessed.
  - Instruction is fetched.
  - PC is updated to point to the next instruction.
- **Clock Cycle 2: Instruction Decode (ID)**
  - Instruction is decoded.
  - Required registers are read from the register file.
  - Immediate values are extended.
  - Control signals are generated.
  - Hazard detection occurs.
- **Clock Cycle 3: Execute (EX)**
  - ALU performs operation based on opcode.
  - For R-type: Register-register operation (add, sub, etc.).
  - For I-type: Register-immediate operation or address calculation.
  - For branches: Condition evaluation and target address calculation.
  - For jumps: Target address calculation.
- **Clock Cycle 4: Memory Access (MEM)**
  - For load instructions: Memory is read using address from EX stage.
  - For store instructions: Data is written to memory.
  - For other instructions: Stage is passed through.
- **Clock Cycle 5: Write Back (WB)**
  - Results from ALU or memory are written back to the destination register.
  - Instruction execution is now complete.

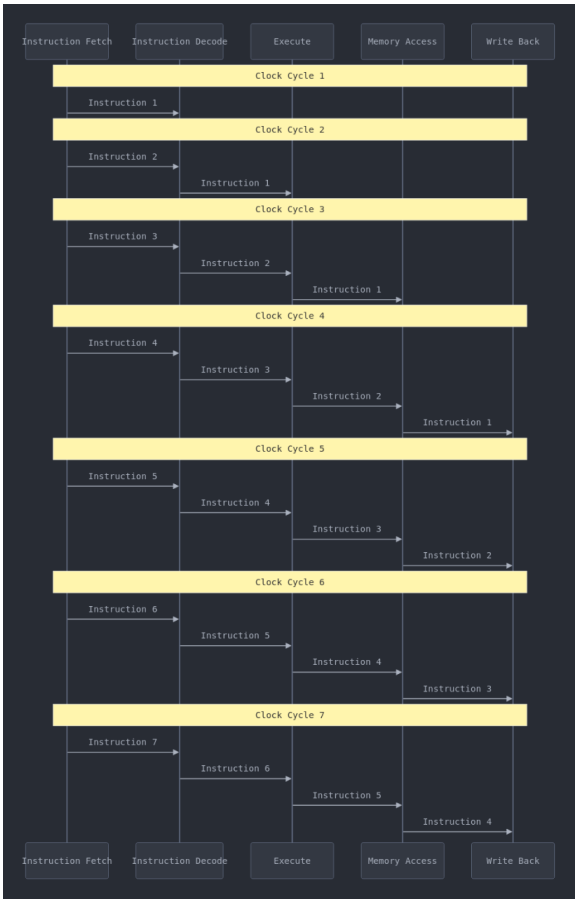


Figure 3: Pipeline Instruction Flow

7 Explanation of each new Module implemented in Verilog

1. Register between Stages

File: `Register_file.v`

IF/ID Pipeline Register

This register captures the output of the Fetch stage to pass to the Decode stage.

Register Components

- **IF_ID_Instruction (32 bits)**: Holds the fetched instruction.
- **IF_ID_PC (64 bits)**: Holds the current program counter value.

Clock and Control Behavior

- Triggered on the negative edge of the clock (@(negedge Clk)).
- **Reset behavior**: Clears both registers to zero when reset is high.
- **Flush behavior**: When Flush signal is high (branch taken), both registers are cleared to zero.
- **Stall behavior**: When Stall signal is high (hazard detected), maintains previous values.

Inputs

- **Instruction**: Current instruction from instruction memory.
- **PC**: Current program counter value.
- **Stall**: Signal from hazard detection unit.
- **Flush**: Signal from control hazard unit.

Outputs

- **IF_ID_Instruction**: Used by Decode stage to extract register addresses and immediates.
- **IF_ID_PC**: Used for branch target calculation.

ID/EX Pipeline Register

This register captures the output of the Decode stage to pass to the Execute stage.

Register Components

Data Registers:

- **ID_EX_Instruction (32 bits)**: Passed instruction.
- **ID_EX_PC (64 bits)**: Program counter value.
- **ID_EX_A (64 bits)**: Data from first source register (Rs1).
- **ID_EX_B (64 bits)**: Data from second source register (Rs2).
- **ID_EX_C (64 bits)**: Immediate value.
- **ID_EX_rd (5 bits)**: Destination register address.

Control Signals:

- **ID_EX_ALUOp (2 bits)**: ALU operation type.
- **ID_EX_ALUSrc (1 bit)**: Selects between register or immediate for ALU input.
- **ID_EX_MemRead (1 bit)**: Memory read enable.
- **ID_EX_MemWrite (1 bit)**: Memory write enable.
- **ID_EX_MemtoReg (1 bit)**: Selects between ALU result or memory data for writeback.
- **ID_EX_RegWrite (1 bit)**: Register write enable.
- **ID_EX_Branch (1 bit)**: Branch instruction indicator.

Clock and Control Behavior

- Triggered on the negative edge of the clock (@(negedge Clk)).
- **Reset behavior**: All registers cleared to zero.
- **Flush behavior**: All registers cleared to zero (effectively inserting a NOP).
- **Stall behavior**: Data registers maintain values, but control signals are disabled (set to 0).

Inputs

- **IF_ID_Instruction**: Instruction from IF/ID register.
- **IF_ID_PC**: Program counter from IF/ID register.
- **A, B**: Register file outputs (read data).
- **C**: Immediate value generated by decode.
- Control signals from control unit.

Outputs

- Used by Execute stage for ALU operations, branch calculations.
- Used by hazard detection unit (**ID_EX_rd** and **ID_EX_MemRead**).

EX/MEM Pipeline Register

This register captures the output of the Execute stage to pass to the Memory stage.

Register Components

Data Registers:

- **EX_MEM_Instruction (32 bits)**: Passed instruction.
- **EX_MEM_ALU_result (64 bits)**: Result from ALU operation.
- **EX_MEM_Target (64 bits)**: Calculated branch target address.
- **EX_MEM_B (64 bits)**: Value to be stored in memory (if store instruction).
- **EX_MEM_Zero (1 bit)**: Zero flag from ALU (for branch decision).

Control Signals:

- **EX_MEM_MemRead (1 bit)**: Memory read enable.
- **EX_MEM_MemWrite (1 bit)**: Memory write enable.
- **EX_MEM_MemtoReg (1 bit)**: Memory to register signal.
- **EX_MEM_Branch (1 bit)**: Branch instruction indicator.
- **EX_MEM_RegWrite (1 bit)**: Register write enable.

Clock and Control Behavior

- Triggered on the negative edge of the clock (@(negedge Clk)).
- **Reset behavior**: All registers cleared to zero.
- No explicit flush or stall handling (relies on previous stage's handling).

Inputs

- **ID_EX_Instruction**: Instruction from ID/EX register.
- **ALU_result**: Result from ALU.
- **Target**: Branch target address.
- **ID_EX_B**: Data for store instructions.
- **Zero**: ALU zero flag.
- Control signals from ID/EX register.

Outputs

- Used by Memory stage for memory operations.
- Used for data forwarding (**EX_MEM_ALU_result**).
- **EX_MEM_Branch** and **EX_MEM_Zero** used for branch decision.

MEM/WB Pipeline Register

This register captures the output of the Memory stage to pass to the WriteBack stage.

Register Components

- **MEM_WB_Instruction (32 bits)**: Passed instruction.
- **MEM_WB_ALU_result (64 bits)**: ALU result forwarded from Execute stage.
- **MEM_WB_Data (64 bits)**: Data read from memory.

Control Signals:

- **MEM_WB_RegWrite (1 bit)**: Register write enable.
- **MEM_WB_MemtoReg (1 bit)**: Select between ALU result and memory data.

2. Forwarding Unit

File: `Forwarding_Unit.v`

Inputs

- **ID_EX_RegisterA/B**: Register numbers (rs1/rs2) from the instruction currently in the ID/EX pipeline stage.
- **EX_MEM_Instruction, MEM_WB_Instruction**: Complete instruction words in later pipeline stages.
- **EX_MEM_RegWrite, MEM_WB_RegWrite**: Control signals indicating if the instructions in those stages will write to a register.

Outputs

- **ForwardA**: 2-bit control signal for the first ALU input multiplexer.
- **ForwardB**: 2-bit control signal for the second ALU input multiplexer.

Operation

The forwarding unit extracts destination register numbers (rd) from instructions in later pipeline stages. The unit then makes forwarding decisions in a combinational logic block (`always @(*)`):

Default Condition

- No forwarding occurs (**ForwardA/B** = `2'b00`).

First Priority: Forward from EX/MEM Stage

- If the instruction in EX/MEM will write to a register.
- The destination register is not x0 (zero register).
- The destination register matches the source register needed in ID/EX.
- Then, **ForwardA/B** = `2'b10` (most recent result).

Second Priority: Forward from MEM/WB Stage

- If the instruction in MEM/WB will write to a register.
- The destination register is not x0.
- The destination register matches the source register needed in ID/EX.
- No higher priority forwarding from EX/MEM.
- Then, **ForwardA/B** = `2'b01`.

Conflict Resolution

- In case both EX/MEM and MEM/WB can forward to the same register, prioritize EX/MEM (Taking Care of Double Data Hazard).

3. Hazard Detection Unit

File: `Hazard_Detection_Unit.v`

Overview

The Hazard Detection Unit (HDU) is responsible for detecting and handling load-use data hazards. These hazards occur when an instruction attempts to use data that is still being loaded from memory by the immediately preceding instruction.

Inputs

- **ID_EX_MemRead**: Control signal indicating if the instruction in the ID/EX stage is a load instruction.
- **ID_EX_RD**: Destination register number for the instruction in the ID/EX stage.
- **IF_ID_Rs1**: First source register number for the instruction in the IF/ID stage.
- **IF_ID_Rs2**: Second source register number for the instruction in the IF/ID stage.

Output

- **stall**: Signal that indicates whether the pipeline should stall.

Operation

The Hazard Detection Unit operates using a combinational logic block to detect load-use hazards.

Stall Condition

The unit asserts the stall signal when:

- There is a load instruction in the ID/EX stage (**ID_EX_MemRead** is high), **AND**
- The destination register of that load instruction matches either source register of the instruction in the IF/ID stage:
  - ID_EX_RD = IF_ID_Rs1, or
  - ID_EX_RD = IF_ID_Rs2

If both conditions are met, the **stall** signal is asserted to pause the pipeline execution and allow the required data to become available.

4. Control Hazard Unit

File: `Control_Hazard_beq.v`

Overview

The Control Hazard Unit is responsible for detecting and handling control hazards that occur when a branch instruction changes the flow of execution. This module specifically handles the **BEQ (Branch if Equal)** instruction.

Inputs

- **Branch**: Control signal indicating that a branch instruction is in the Execute stage.
- **Zero**: ALU Zero flag indicating if the branch condition (equality) is met.

Output

- **Flush**: Signal to flush the pipeline when a branch is taken.

Operation

The Control Hazard Unit uses combinational logic to determine when to flush the pipeline.

Flush Condition

The unit asserts the **Flush** signal when:

- The instruction is a branch (**Branch** is high), **AND**
- The branch condition is satisfied (**Zero** is high, meaning the compared values are equal).

This specific implementation applies to **BEQ (Branch if Equal)** instructions, where the branch is taken when two registers contain equal values (**Zero** = 1).

8 Test for Pipeline Implementation

8.1 Provided Instruction Set

ADD x1, x2, x3
`0000000 00011 00010 000 00001 0110011`
ADD x4, x5, x6
`0000000 00101 00110 000 00100 0110011`
SUB x7, x4, x1
`0100000 00001 00100 000 00111 0110011`
AND x8, x5, x6
`0000000 00110 00101 111 01000 0110011`
SW x13, 16(x10)
`0000000 01101 01010 011 10000 0100011`
Stores word at memory address x10 + 16
LD x7, 16(x10)
`000000010000 01010 011 00111 0000011`
Loads word from memory address x10 + 16 into x7
OR x8, x7, x9
`0000000 00111 01001 110 01000 0110011`
AND x5, x5, x6
`0000000 00110 00101 111 00101 0110011`

ADD x20, x20, x3
`0000000 10100 00011 000 10100 0110011`
ADD x20, x20, x21
`0000000 10100 10101 000 10100 0110011`
ADD x20, x20, x22
`0000000 10100 10110 000 10100 0110011`
BEQ x23, x22, +4
`000000 10111 10110 000 00100 1100011`
Branches to the instruction at offset 8 if x23 = x22
AND x20, x5, x6
`0000000 00110 00101 111 10110 0110011`
(Not executed due to BEQ instruction)
SUB x26, x25, x24
`0100000 11000 11001 000 11010 0110011`
SW x22, 16(x10)
`0000000 10110 01010 011 10000 0100011`
Stores word x22 at memory address 16 + x10
SW x20, 16(x10)
`0000000 10100 01010 011 10000 0100011`
Stores word x20 at memory address 16 + x10
LD x7, 16(x10)
`000000010000 01010 011 00111 0000011`
Loads word 16 + x10 from memory address into x7

8.2 Provided Register values(Initial)

```
0  0000000000000000
1  0000000000000001
2  0000000000000002
3  0000000000000003
4  0000000000000004
5  0000000000000005
6  0000000000000006
7  0000000000000007
8  0000000000000008
9  0000000000000009
10 0000000000000002
11 000000000000000b
12 000000000000000c
13 0000000000000006
14 0000000000000009
15 000000000000000f
16 0000000000000010
17 0000000000000011
18 0000000000000000
```

```
19 0000000000000000
20 0000000000000001
21 0000000000000009
22 000000000000000a
23 000000000000000a
24 0000000000000001
25 0000000000000000
26 0000000000000000
27 0000000000000000
28 0000000000000000
29 0000000000000000
30 0000000000000000
31 0000000000000000
```

8.3 Obtained Register values after Execution

After each Instructions **expected outputs** are

Instruction Executed	Register Edited	Value printed
Instruction - 1	x1	5
Instruction - 2	x4	11
Instruction - 3	x7	6
Instruction - 4	x8	4
Instruction - 5	–	Store
Instruction - 6	x7	6
Instruction - 7	x8	15
Instruction - 8	x5	4
Instruction - 9	x20	4
Instruction - 10	x20	13
Instruction - 11	x20	23
Instruction - 12	–	Branch
Instruction - 13	–	Not Executed
Instruction - 14	x26	-1
Instruction - 15	–	Store
Instruction - 16	–	Store
Instruction - 17	x7	23

But there are some registers edited multiple times like x7 , x20 .
Finally the obtained register values after execution are shown below .

```
0  0000000000000000
1  0000000000000005
2  0000000000000002
3  0000000000000003
4  000000000000000b
5  0000000000000004
```

```
6  0000000000000006
7  0000000000000017
8  000000000000000f
9  0000000000000009
10 0000000000000002
11 000000000000000b
12 000000000000000c
13 0000000000000006
14 0000000000000009
15 000000000000000f
16 0000000000000010
17 0000000000000011
18 0000000000000000
19 0000000000000000
20 0000000000000017
21 0000000000000009
22 000000000000000a
23 000000000000000a
24 0000000000000001
25 0000000000000000
26 ffffffffffffffff
27 0000000000000000
28 0000000000000000
29 0000000000000000
30 0000000000000000
31 0000000000000000
```

8.4 Output plots Obtained after each Block

8.4.1 IF Stage

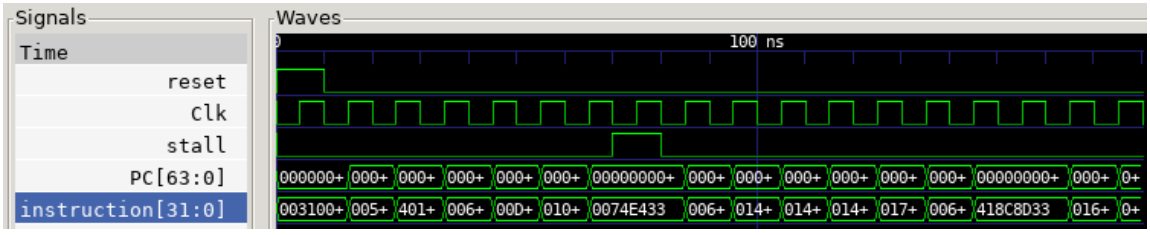


Figure 4: Instruction Fetch Stage

8.4.2 Control Block

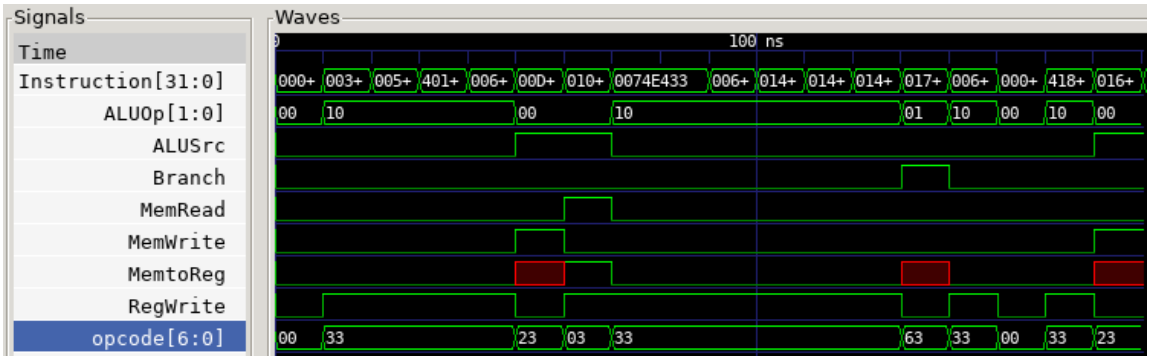


Figure 5: Control Block

8.4.3 ID Stage

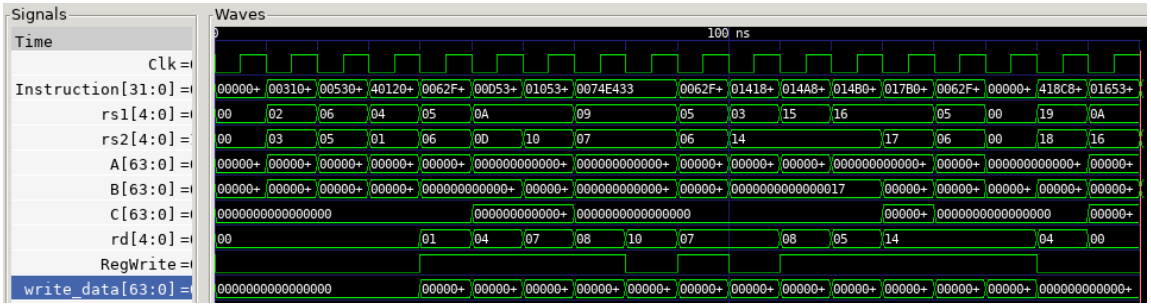


Figure 6: Instruction Decode Stage

8.4.4 MEM Stage

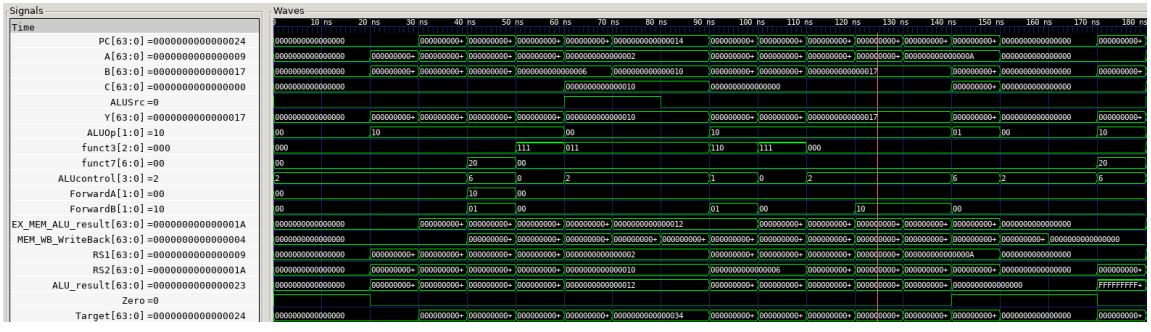


Figure 7: MEM Stage

8.4.5 WB Stage

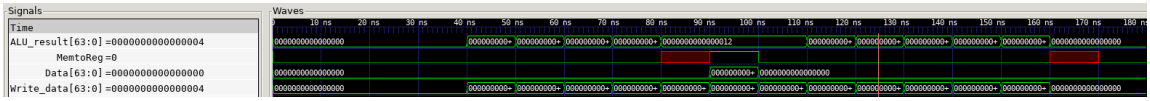


Figure 8: Write Back Stage

8.4.6 PC Update Stage

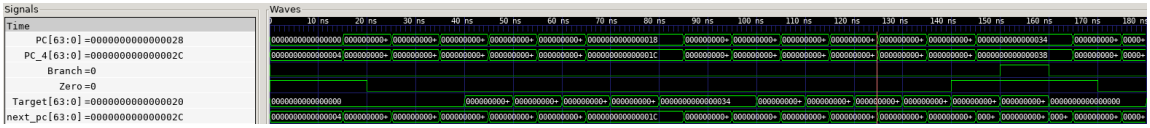


Figure 9: PC Update

8.4.7 Hazard Detection Unit for Load Use

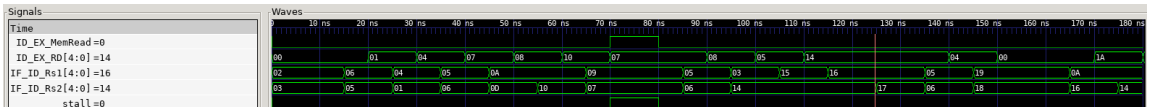


Figure 10: Hazard Detection Unit

8.4.8 Control Hazard Unit

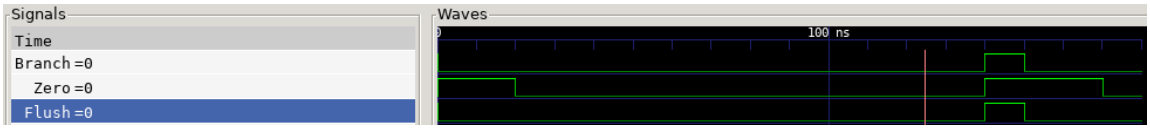


Figure 11: Control Hazard unit

8.4.9 EXE Stage

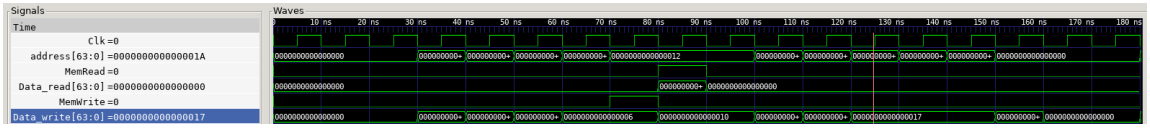


Figure 12: Enter Caption

9 Contribution :

Aasrith Reddy

Sequential :

- Execute file
- Immediate Generator
- Wrapper file

Pipelining :

- Registers between blocks and its update
- Wrapper file

Aditya Peketi

Sequential :

- Fetch file
- Decode file

Pipelining :

- Load-use hazard unit
- Flushing for branching
- Wrapper file

Akshat Puneet

Sequential :

- Next Pc file
- Control block
- Memory block

Pipelining :

- Forward unit

All decoding in errors did in group.
All equally contributed to Report .