Intro to Processor Architecture COURSE PROJECT

TEAM-4

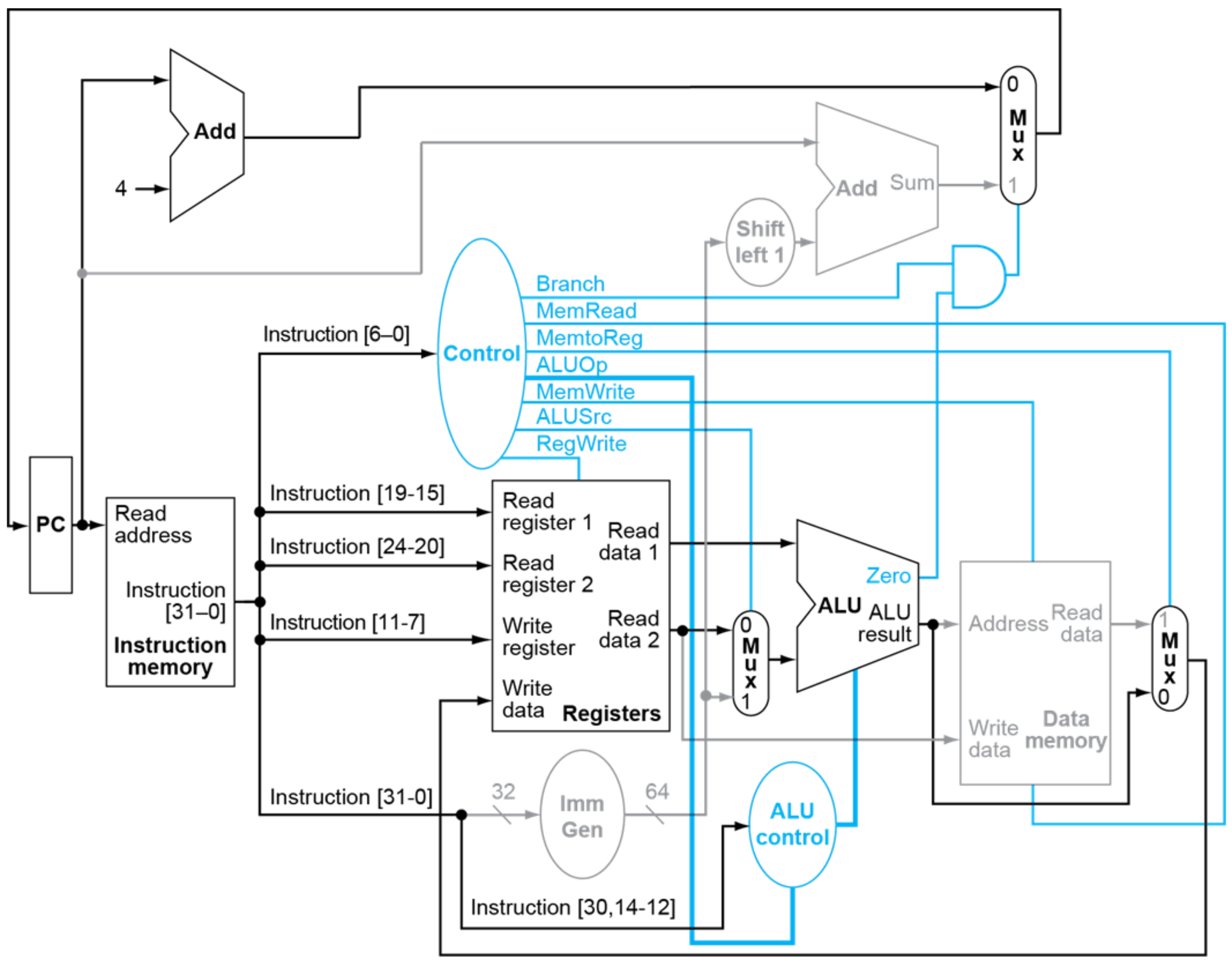
DATAARNOOR SINGH-2023102047

DHIMANT BHUVA-2023102063

JAL PARIKH-2023102066

PART-1

SEQUENTIAL IMPLEMENTATION



COMPONENTS IN SEQUENTIAL IMPLEMENTATION:

- Program Counter
- Instruction Memory
- Register File
- Immediate Generator
- Control Unit
- ALU Control
- ALU Result
- Data Memory

1. Program Counter (PC)

Function:

The **Program Counter (PC)** keeps track of the address of the current instruction being executed. It is responsible for sequential instruction execution and updating the instruction memory with the correct address.

Working:

1. At the start of each cycle, the **PC outputs an address**.
2. The **Instruction Memory** fetches the instruction from this address.
3. The PC is then updated:
    - For sequential execution: **PC = PC + 1**
    - For branch/jump: **PC = PC + Immediate**
4. This new value is stored in the **PC register**, ensuring the processor fetches the next instruction in the following cycle.

Importance:

- Ensures the **flow of execution** by keeping track of instruction addresses.
- Enables **sequential execution** and supports **branching/jumping**.
- Serves as an essential input to **instruction memory**.

2. Instruction Memory

Function:

The **Instruction Memory** stores the machine code of the program and provides the instruction corresponding to the address given by the **PC**.

Working:

1. The **PC outputs an address**.
2. The **Instruction Memory reads the instruction** stored at this address.
3. This instruction is then sent to the **decoder** for processing.

Importance:

- It is the **source of program execution**, storing all instructions.
- Acts as an **interface between the control unit and PC**.

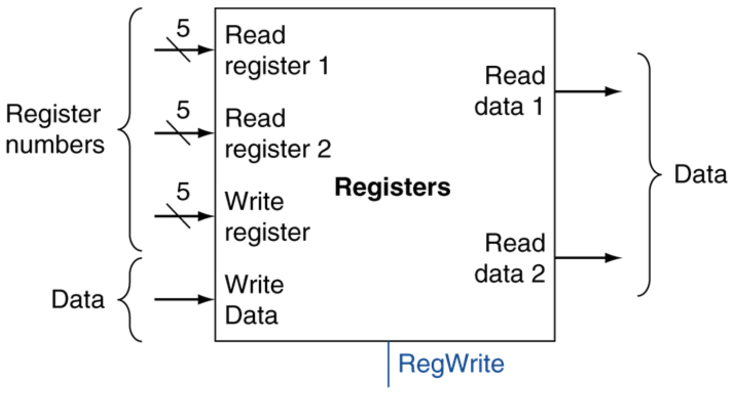
3. Register File

Function:

The **Register File** holds **32 registers (`x0` to `x31`)** in the processor. It allows reading two registers (`rs1`, `rs2`) and writing to one (`rd`).

Working:

1. The instruction decoder extracts:
    - **Source registers**: `rs1`, `rs2`
    - **Destination register:** `rd`
2. The register file provides the **values of `rs1` and `rs2`.**



3. After the ALU computes the **result**, it is **written back to `rd`**.

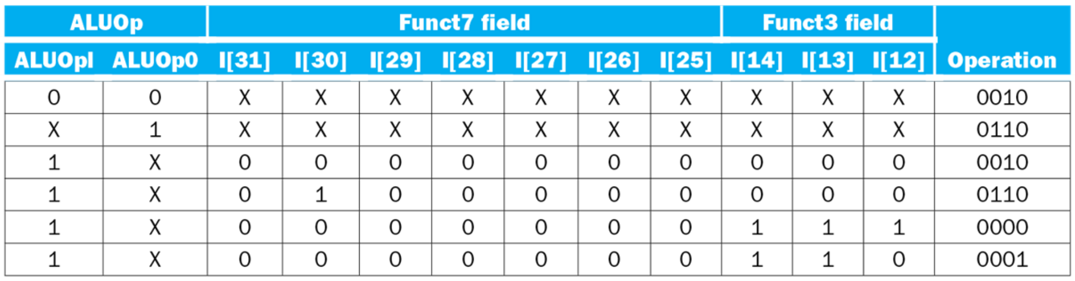
Importance:

- Enables **fast access** to frequently used values.
- Acts as a **temporary storage** for computations.
- Plays a key role in **data dependency management**.

4. Control Unit

Function:

The **Control Unit** is the **brain** of the processor. It **decodes the instruction** and generates the necessary **control signals** to direct data flow and operations.



ALUOp		Funct7 field							Funct3 field			
ALUOpl	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

Working:

1. **Decodes the opcode** from the instruction.
2. **Generates control signals**:
   - `RegWrite` → Enables writing to the register file.
   - `ALUSrc` → Chooses between a register or immediate for ALU input.
   - `MemRead` → Enables memory reading for `lw`.
   - `MemWrite` → Enables memory writing for `sw`.
   - `Branch` → Activates for branch instructions.
   - `ALUOp` → Determines ALU operation.

Importance:

- Ensures **correct execution** of each instruction.
- Directs signals to various processor components.
- Plays a **central role in instruction execution**.

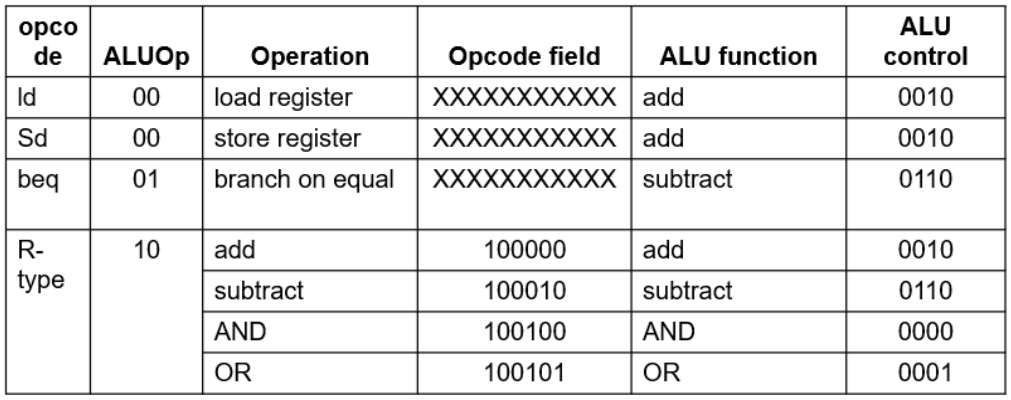
5. ALU Control

Function:

The **ALU Control Unit** determines **which operation** the ALU should perform based on the **ALUOp** signal and the function bits (`funct3`, `funct7`) of the instruction.

Working:

1. Takes `ALUOp` signal from the **Control Unit**.
2. Reads `funct3` and `funct7` from the instruction.
3. **Decides the specific ALU operation**:
   - `funct3 = 000`, `ALUOp = 00` → ADD
   - `funct3 = 000`, `ALUOp = 01` → SUB
   - `funct3 = 100`, `ALUOp = 10` → XOR



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

Importance:

- Ensures correct **arithmetic and logic execution**.
- Allows **modular ALU operation selection**.

6. Arithmetic Logic Unit (ALU)

Function:

The **ALU** performs all **arithmetic and logical operations** such as addition, subtraction, AND, OR, XOR, and comparisons.

Working:

1. Takes two inputs (`rs1`, `rs2` or immediate).
2. Performs the operation specified by the **ALU Control**.
3. Outputs:
   - Computed result.
   - Zero flag (`Z`) if the result is zero.

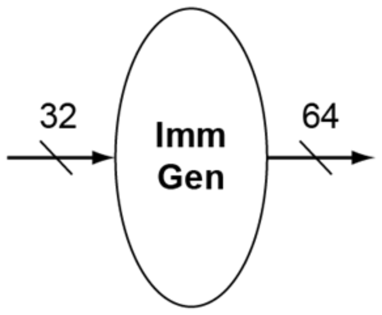
Importance:

- Central to **data processing** in the processor.
- Enables computations needed for **program execution**.

7. Immediate Generator

Function:

The **Immediate Generator** extracts immediate values from instructions and **sign-extends** them to 32-bit format.



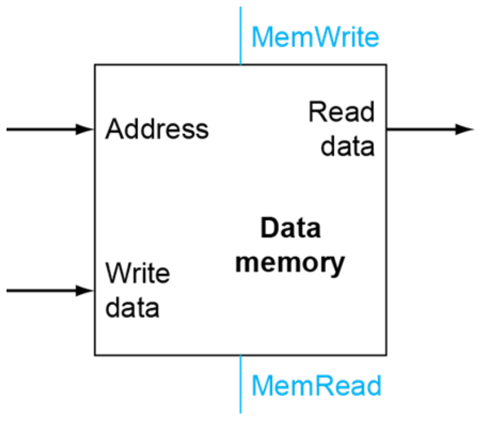
Working:

1. Detects **instruction type (I, S, B)**.
2. Extracts **immediate bits** and **sign-extends** them.
3. Provides this **32-bit immediate** to the ALU or branch logic.

Importance:

- Enables **immediate-based operations** (`addi`, `ld`, `sd`).
- Supports **jump/branch calculations**.

8. Data Memory



Function:

The **Data Memory** component stores **data values** and allows **memory read (`ld`) and memory write (`sd`) operations**.

Working:

1. **For `ld x1, 8(x2)`:**
   - The ALU computes the address: `Mem[8 + x2]`.
   - Data is **read** from memory.
   - The result is written to `x1`.
2. **For `sd x1, 8(x2)`:**
   - The ALU computes the address: `Mem[8 + x2]`.
   - The value in `x1` is **stored at that address**.

Importance:

- Essential for **load (`ld`) and store (`sd`) instructions**.
- Acts as **primary storage** in processor memory hierarchy.

The project involves implementing the following instructions:

- **ADD** (Addition)
- **SUB** (Subtraction)
- **AND** (Bitwise AND)
- **OR** (Bitwise OR)
- **LD** (Load)
- **SD** (Store)

- **BEQ** (Branch if Equal)

1. ADD (Addition)

Format (R-Type Instruction)

ADD rd rs1 rs2

- Adds values from registers `rs1` and `rs2`, and stores the result in `rd`.

Opcode	funct3	funct7	Description
`0110011`	`000`	`0000000`	ADD

Execution Steps:

1. **Instruction Fetch:**
    - PC provides the instruction address to **Instruction Memory**.
    - Instruction Memory fetches the **ADD instruction**.
    - PC increments by 4 (`PC = PC + 1`).
2. **Instruction Decode:**
    - The Control Unit **decodes** the instruction.
    - Determines that the operation is **ADD**.
    - Sets control signals:
        - `RegWrite = 1`, `ALUOp = 10` (R-type operation).
3. **Register Read:**
    - Register File **reads values** from `rs1` and `rs2`.
4. **ALU Execution:**
    - ALU **performs addition**:

Result = rs1 + rs2

5. **Write Back:**
    - Result is **written back** to register `rd`.

2. SUB (Subtraction)

Format (R-Type Instruction)

SUB rd rs1 rs2

- Subtracts values from `rs1` and `rs2`, and stores the result in `rd`.

Opcode	funct3	funct7	Description
`0110011`	`000`	`0100000`	SUB

Execution Steps (Same as ADD except ALU operation)

1. **Fetch**: PC fetches the instruction.
2. **Decode**: Control Unit recognizes it as SUB.
3. **Register Read**: Register values are fetched.
4. **ALU Execution**:
   - ALU performs subtraction:

     Result = rs1 − rs2

5. **Write Back**: Result is stored in `rd`.

3. AND (Bitwise AND)

Format (R-Type Instruction)

AND rd rs1 rs2

- Performs a **bitwise AND** operation between `rs1` and `rs2` and stores the result in `rd`.

Opcode	funct3	funct7	Description
`0110011`	`111`	`0000000`	AND

Execution Steps (Same as ADD with ALU operation change)

1. **Fetch**: PC fetches instruction.
2. **Decode**: Control Unit identifies AND instruction.
3. **Register Read**: Values from `rs1` and `rs2` are read.
4. **ALU Execution**:
   - ALU performs bitwise AND:

Result = rs1 & rs

5. **Write Back**: Result is stored in `rd`.

4. OR (Bitwise OR)

Format (R-Type Instruction)

OR rd, rs1, rs2

- Performs a **bitwise OR** operation between `rs1` and `rs2` and stores the result in `rd`.

Opcode	funct3	funct7	Description
`0110011`	`110`	`0000000`	OR

Execution Steps (Same as AND, but with OR operation)

1. **Fetch**: PC fetches instruction.
2. **Decode**: Control Unit identifies OR instruction.
3. **Register Read**: Values from `rs1` and `rs2` are read.
4. **ALU Execution**:
   - ALU performs bitwise OR:

Result = rs1 | rs2

5. **Write Back**: Result is stored in `rd`.

5. LD (Load Double Word - ld)

Format (I-Type Instruction)

LD rd, offset(rs1)

- Loads a word from memory at `rs1 + offset` into `rd`.

Opcode	funct3	Description
`0000011`	`010`	LOAD

Execution Steps

1. **Fetch**: PC fetches instruction.
2. **Decode**: Control Unit identifies it as **LOAD**.
3. **Register Read**: Reads `rs1` from Register File.
4. **Immediate Generation**:
    - Immediate Generator extracts **offset**.
5. **Address Calculation**:
    - ALU computes memory address:

Address = rs1 + OffsetAddress

6. **Memory Access**:
    - Data Memory reads data from the computed address.
7. **Write Back**:
    - Loaded data is stored in `rd`.

6. STORE (Store Double Word - sd)

Format (S-Type Instruction)

SD rs2, offset(rs1)

- Stores a word from `rs2` into memory at address `rs1 + offset`.

Opcode	funct3	Description
`0100011`	`010`	STORE

Execution Steps

1. **Fetch**: PC fetches instruction.
2. **Decode**: Control Unit identifies **STORE**.
3. **Register Read**: Reads values of `rs1` and `rs2`.
4. **Immediate Generation**:
   - Immediate Generator extracts **offset**.
5. **Address Calculation**:
   - ALU computes memory **address** as:

Address = rs1 + OffsetAddress

6. **Memory Write**:
   - Data Memory writes the value of `rs2` to this address.

7. BEQ (Branch if Equal)

Format (B-Type Instruction)

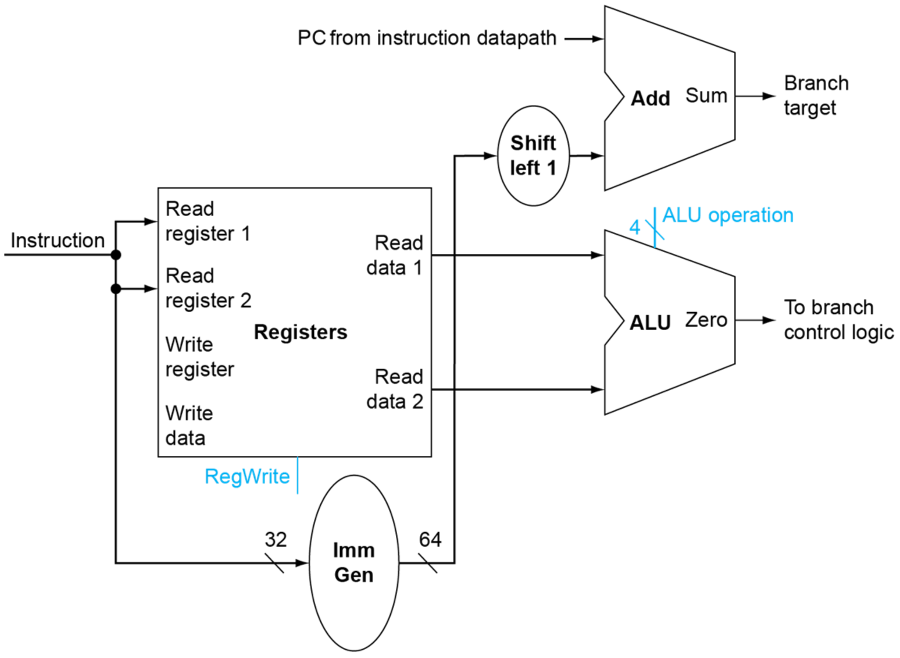
BEQ rs1, rs2, offset

- If `rs1 == rs2`, the PC jumps to `PC + offset`.

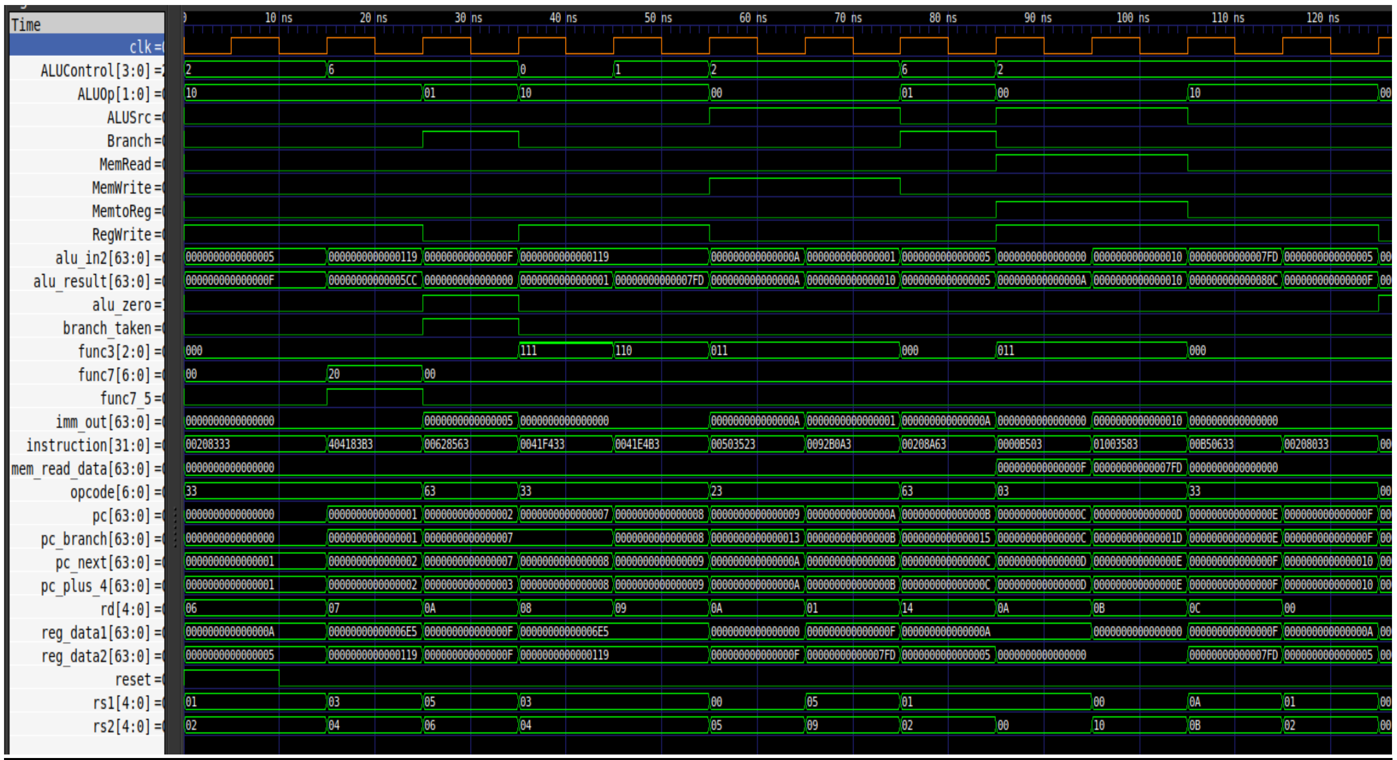
Opcode	funct3	Description
`1100011`	`000`	BEQ

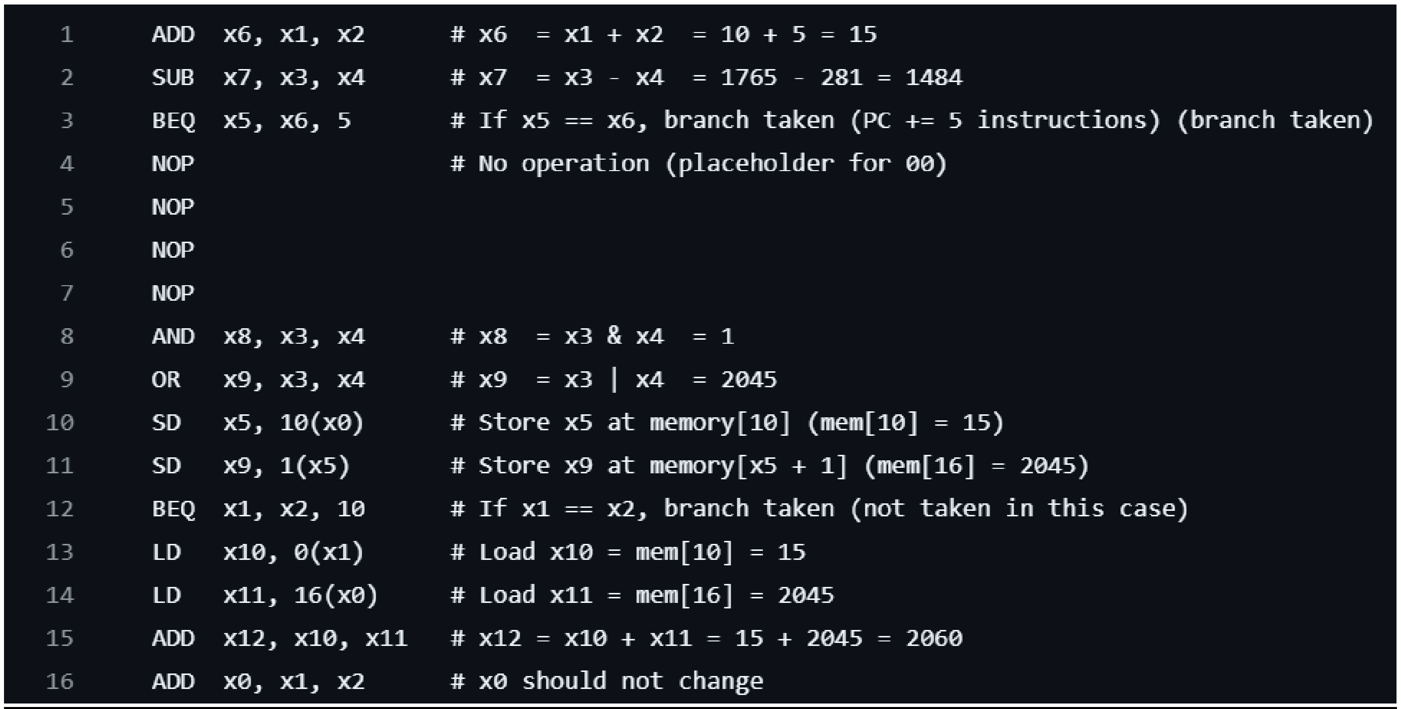
Execution Steps

1. **Fetch**: PC fetches instruction.
2. **Decode**: Control Unit identifies **BEQ**.
3. **Register Read**:
   Reads `rs1` and `rs2`.
4. **ALU Execution**:
   - **ALU compares values: rs1 − rs2**
5. **Branch Decision**:
   - If `Result == 0`, branch is taken:
   - **PC = PC + Offset**
   - Otherwise, **PC = PC + 1** (next instruction).



SIMULATIONS:





```
ADD  x6, x1, x2      # x6  = x1 + x2  = 10 + 5 = 15
SUB  x7, x3, x4      # x7  = x3 - x4  = 1765 - 281 = 1484
BEQ  x5, x6, 5       # If x5 == x6, branch taken (PC += 5 instructions) (branch taken)
NOP                  # No operation (placeholder for 00)
NOP
NOP
NOP
AND  x8, x3, x4      # x8  = x3 & x4  = 1
OR   x9, x3, x4      # x9  = x3 | x4  = 2045
SD   x5, 10(x0)      # Store x5 at memory[10] (mem[10] = 15)
SD   x9, 1(x5)       # Store x9 at memory[x5 + 1] (mem[16] = 2045)
BEQ  x1, x2, 10      # If x1 == x2, branch taken (not taken in this case)
LD   x10, 0(x1)      # Load x10 = mem[10] = 15
LD   x11, 16(x0)     # Load x11 = mem[16] = 2045
ADD  x12, x10, x11   # x12 = x10 + x11 = 15 + 2045 = 2060
ADD  x0, x1, x2      # x0 should not change
```

PROBLEMS IN SEQUENTIAL IMPLEMENTATION:

1. Low Performance Due to Long Clock Cycle

Problem:

- In a sequential processor, **each instruction completes in one clock cycle**.
- The clock cycle duration is determined by the **slowest instruction** (e.g., `LD` takes longer due to memory access).
- Even simple operations (`ADD`, `SUB`) must wait for the full clock cycle, reducing overall speed.

Impact:

- Processor speed is **limited by the slowest instruction**.
- Leads to **underutilization** of faster components.

Example:

- `ADD x1, x2, x3` requires only **register read and ALU execution**.
- `LD x1, 8(x2)` requires **register read, ALU computation, memory access, and register write-back**.
- Since the processor runs in a **single clock cycle**, every instruction is **forced to wait** for the slowest operation.

2. Inefficient Use of Hardware

Problem:

- The ALU, Register File, and Memory are **active only for a small fraction** of each clock cycle.
- The **Control Unit, ALU, and Memory sit idle** when not in use.

Impact:

- **Wasted resources** during each cycle.

- **Low efficiency**, since different units are not used simultaneously.

Example:

- In `LD`, the **ALU is active** only for address calculation but **remains idle** during memory access.
- In `BEQ`, **Memory is unused** but still consumes power.

3. High Power Consumption

Problem:

- Every instruction **utilizes the entire processor**, even if only a few components are needed.
- **Clock cycle is long**, meaning components stay active longer than necessary.

Impact:

- **Unnecessary power wastage**, leading to higher energy consumption.
- Affects **battery life** in embedded systems.

4. Instruction Dependency (Data Hazards)

Problem:

- Some instructions **depend on the result** of a previous instruction.
- In sequential execution, **each instruction must wait for the previous one to complete**.

Impact:

- **Delays execution** of dependent instructions.
- Reduces instruction throughput.

Example:

```
ADD x1, x2, x3      # x1 = x2 + x3
```

`SUB x4, x1, x5` # Needs x1, but must wait for ADD `to finish`

- The `SUB` instruction **cannot execute** until `ADD` writes its result to `x1`.

5. No Parallel Execution (No Pipelining)

Problem:

- A sequential processor **executes one instruction at a time**.
- No ability to **overlap** execution of different instructions.

Impact:

- **Slower execution**, since only **one instruction runs per cycle**.
- **Pipelined processors** can complete multiple instructions at once.

6. Larger Memory Access Time (LD & SD are Slow)

Problem:

- Load (`LD`) and Store (`Sd`) instructions access **external memory**, which is **slower than registers.**
- The entire processor must **wait for the memory operation** to complete.

Impact:

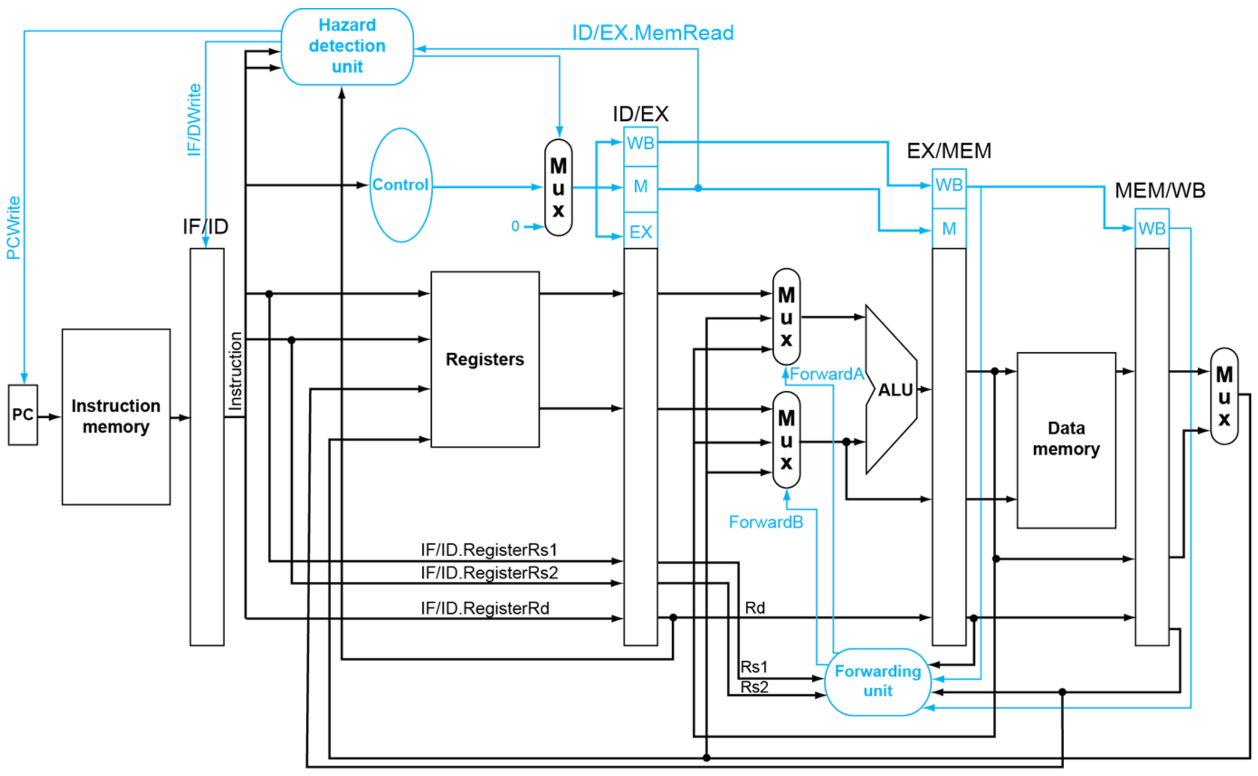
- **Slows down execution**, since memory latency is high.
- Affects performance of **load-heavy programs**.

Example:

```
LD x1, 8(x2)  # Memory access takes longer than ALU
operations

ADD x3, x1, x4  # Must wait for LD to complete
```

PART-2
PIPELINING IN PROCESSORS



What is Pipelining?

Pipelining is a **technique used in modern processors** to improve execution speed by **overlapping** multiple instructions. Instead of completing one instruction at a time, a pipelined processor **starts executing the next instruction before the previous one finishes**, thereby **increasing throughput**.

A **5-stage pipeline** consists of the following stages:

1. Instruction Fetch (IF)

Function:

- Fetches the **next instruction** from memory using the **Program Counter (PC)**.
- Increments the PC to point to the **next instruction**.

Steps:

1. **PC sends address** to instruction memory.
2. **Instruction memory provides instruction** at that address.
3. **PC is updated** to `PC + 1` (for 32-bit instruction width).

Hardware Components Used:

- **Program Counter (PC)**
- **Instruction Memory**
- **Adder (for PC + 1 calculation)**

2. Instruction Decode (ID)

Function:

- **Decodes the instruction** to identify the operation type.
- Reads **register operands (`rs1`, `rs2`)** from the **Register File**.
- Extracts **immediate values** (if required).
- Generates **control signals** for execution.

Steps:

1. The **opcode** is used to determine the instruction type.
2. The **Control Unit generates control signals**.
3. The **Register File fetches values** from registers.
4. The **Immediate Generator** extracts **immediate values** (for `LD`, `SD`, `BEQ`).

Hardware Components Used:

- **Control Unit**
- **Register File**
- **Immediate Generator**

3. Execute (EX)

Function:

- Performs **arithmetic/logic operations** using the **ALU**.
- Computes **memory addresses** for `LD` and `SD`.
- Evaluates **branch conditions** (`BEQ`).

Steps:

1. **ALU Control Unit** decides the ALU operation based on `funct3` and `funct7`.
2. **ALU executes the operation**:
   - `ADD/SUB/AND/OR`
   - Address computation for `LD/SD`
   - Comparison for `BEQ`
3. If `BEQ`, **branch condition is checked**.

Hardware Components Used:

- **ALU**

4. Memory Access (MEM)

Function:

- Reads from or writes to **Data Memory** (for `LD` and `SD`).
- If a **load (`LD`)**, the value from memory is read.
- If a **store (`SD`)**, the value from `rs2` is stored into memory.

Steps:

1. **For Load (`LD`)**:
    - ALU **computes address**.
    - **Memory reads data** from that address.
2. **For Store (`SD`)**:
    - ALU **computes address**.
    - **Memory writes `rs2` data** to that address.

Hardware Components Used:

- **Data Memory**
- **Memory Control Logic**

5. Write Back (WB)

Function:

- Writes the final **ALU result or memory data** back into the **Register File**.

Steps:

1. If the instruction is **arithmetic (`ADD`, `SUB`, etc.)**,
    - The ALU result is **written back** into the register file (`rd`).
2. If the instruction is **load (`LD`)**,
    - The value read from memory is **stored in `rd`**.

Hardware Components Used:

- **Register File**
- **Multiplexer (to select ALU result or memory data)**

Pipeline Hazards (Execution Delays Due to Dependencies)

Hazards prevent the smooth execution of instructions in the pipeline. There are three types:

1. Structural Hazards (Hardware Conflict)

- Occurs when **two instructions need the same resource** (e.g., ALU, Memory) at the same time.
- **Solution:** Use **separate instruction & data memory (Harvard Architecture)** or **multi-port memory**.

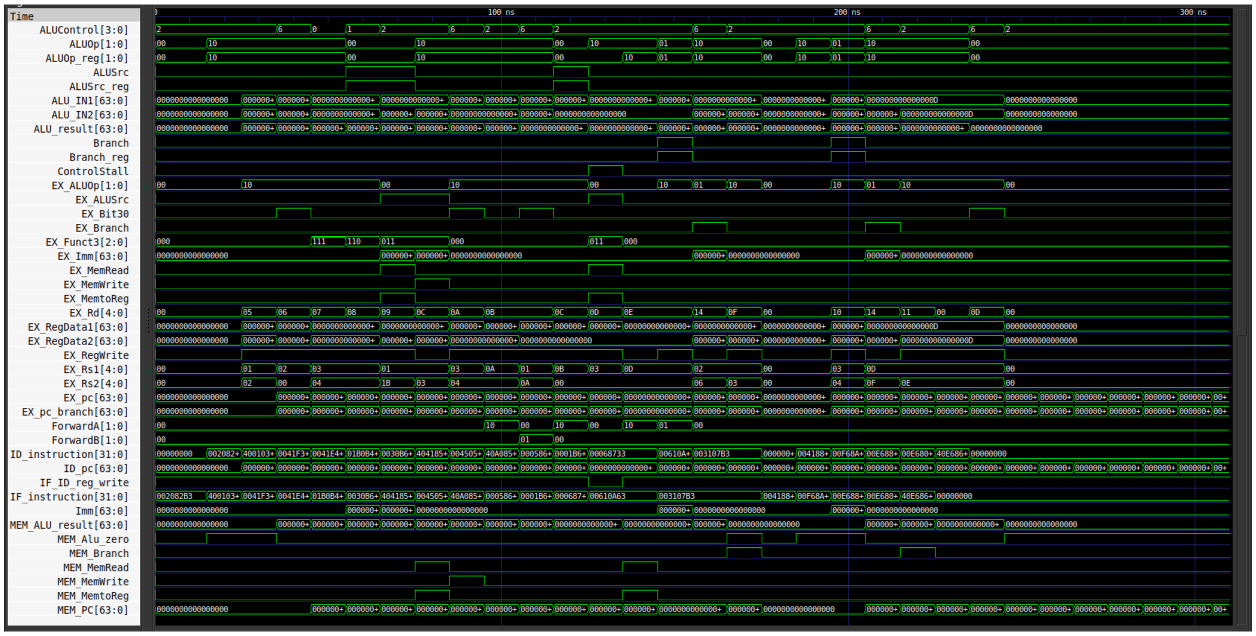
2. Data Hazards (Instruction Dependency Issues)

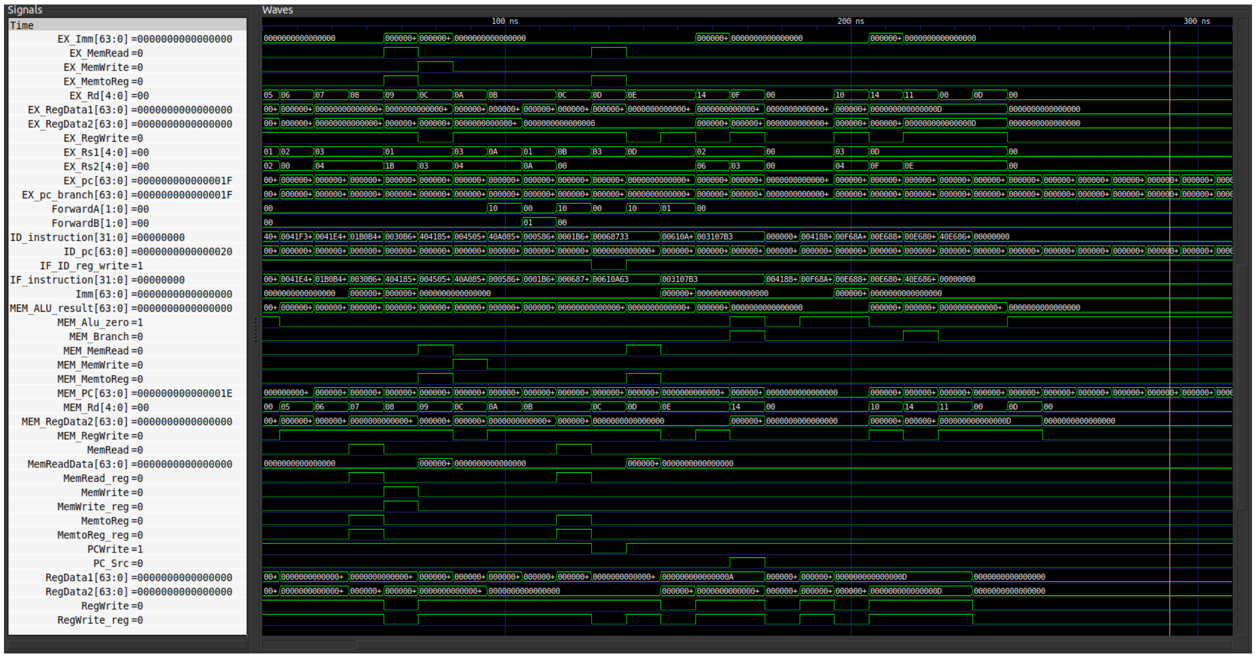
- Happens when an instruction **depends on the result of a previous instruction** that hasn't completed yet.
- **Solution:** Use **data forwarding** or introduce **pipeline stalls** (bubbles).

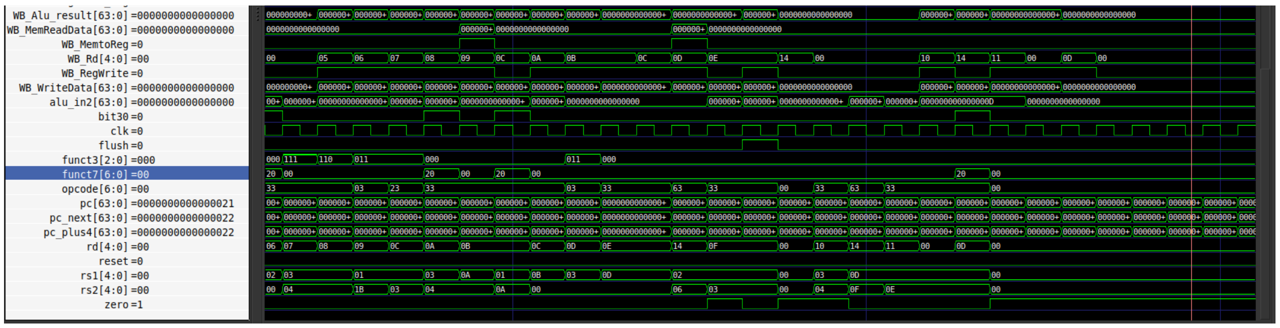
3. Control Hazards (Branching Issues)

- Occurs with **branch instructions (`BEQ`, `JAL`)**, as the processor does not know the next instruction to fetch.
- **Solution:** Use **branch prediction** or **delayed branching** to minimize stalls.

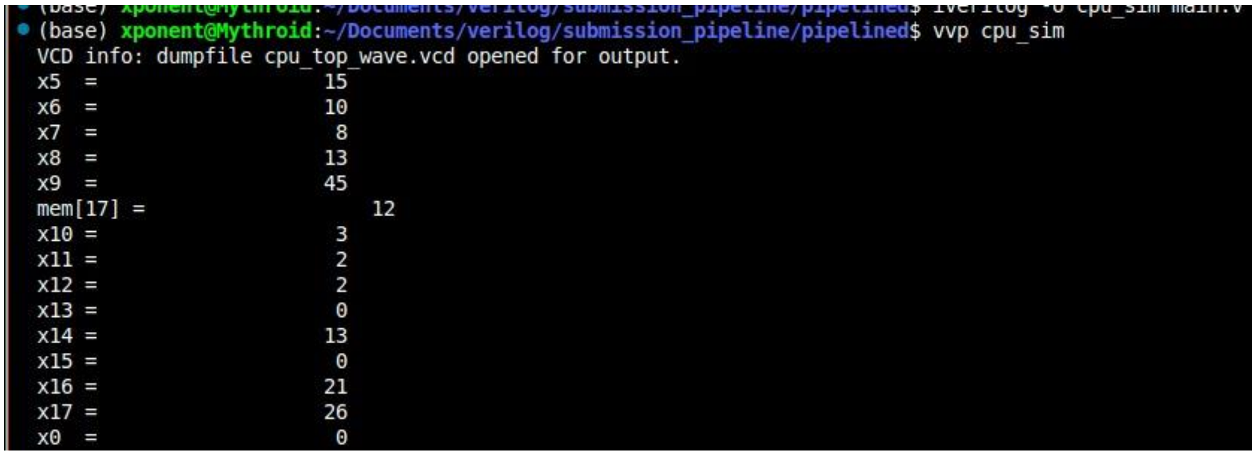
SIMULATION:


Time
100 ns
200 ns
300 ns
ALUControl[3:0]
ALUOp[1:0]
ALUOp_reg[1:0]
ALUSrc
ALUSrc_reg
ALU_IN1[63:0]
ALU_IN2[63:0]
ALU_result[63:0]
Branch
Branch_reg
ControlStall
EX_ALUOp[1:0]
EX_ALUSrc
EX_Bit30
EX_Branch
EX_Funct3[2:0]
EX_Imm[63:0]
EX_MemRead
EX_MemWrite
EX_MemtoReg
EX_Rd[4:0]
EX_RegData1[63:0]
EX_RegData2[63:0]
EX_RegWrite
EX_Rs1[4:0]
EX_Rs2[4:0]
EX_pc[63:0]
EX_pc_branch[63:0]
ForwardA[1:0]
ForwardB[1:0]
ID_instruction[31:0]
ID_pc[63:0]
IF_ID_reg_write
IF_instruction[31:0]
Imm[63:0]
MEM_ALU_result[63:0]
MEM_Alu_zero
MEM_Branch
MEM_MemRead
MEM_MemWrite
MEM_MemtoReg
MEM_PC[63:0]

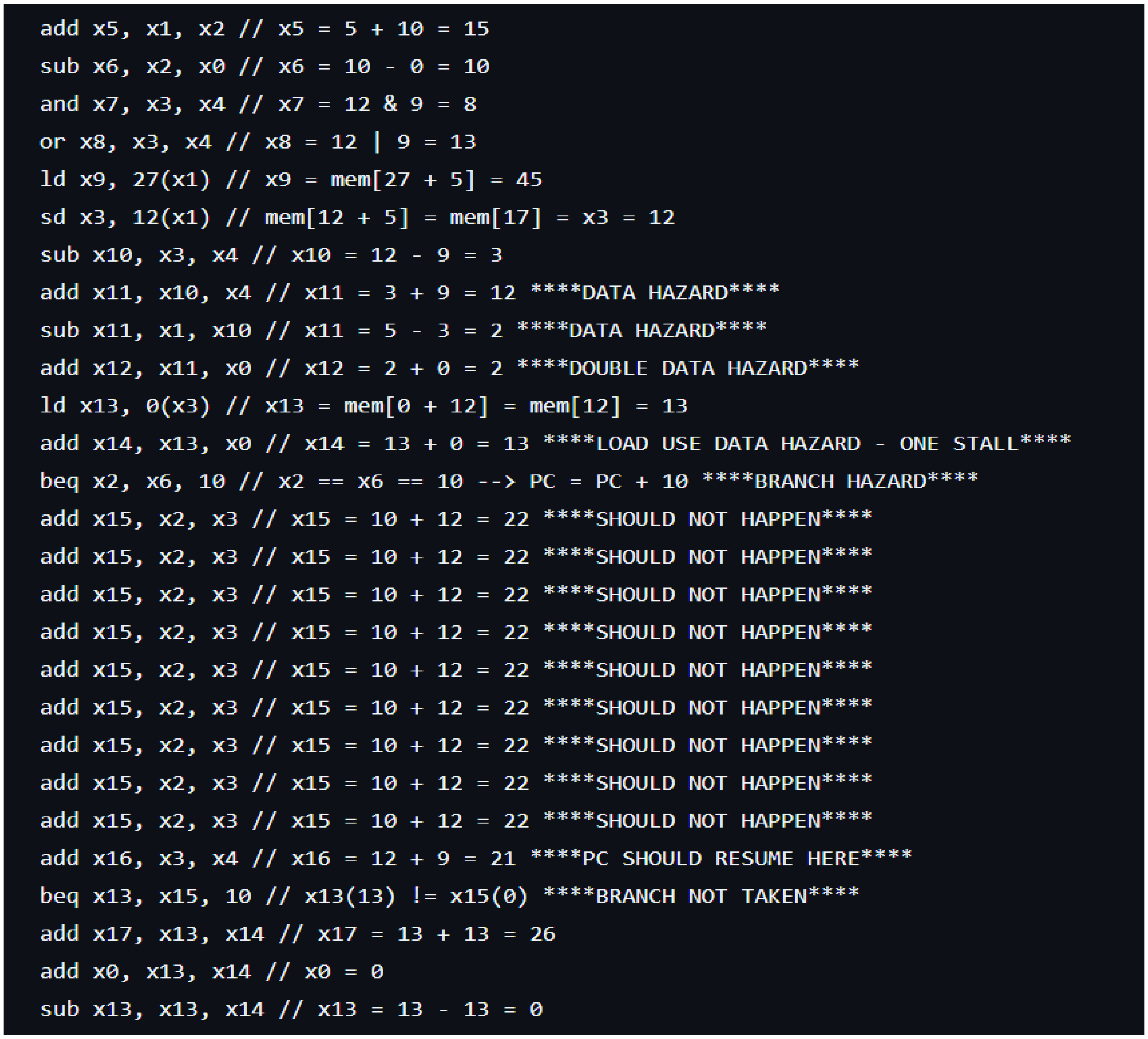

Signals
Waves
Time
100 ns
200 ns
300 ns
EX_Imm[63:0] =0000000000000000
EX_MemRead =0
EX_MemWrite =0
EX_MemtoReg =0
EX_Rd[4:0] =00
EX_RegData1[63:0] =0000000000000000
EX_RegData2[63:0] =0000000000000000
EX_RegWrite =0
EX_Rs1[4:0] =00
EX_Rs2[4:0] =00
EX_pc[63:0] =000000000000001F
EX_pc_branch[63:0] =000000000000001F
ForwardA[1:0] =00
ForwardB[1:0] =00
ID_instruction[31:0] =00000000
ID_pc[63:0] =0000000000000020
IF_ID_reg_write =1
IF_instruction[31:0] =00000000
Imm[63:0] =0000000000000000
MEM_ALU_result[63:0] =0000000000000000
MEM_Alu_zero =1
MEM_Branch =0
MEM_MemRead =0
MEM_MemWrite =0
MEM_MemtoReg =0
MEM_PC[63:0] =000000000000001E
MEM_Rd[4:0] =00
MEM_RegData2[63:0] =0000000000000000
MEM_RegWrite =0
MemRead =0
MemReadData[63:0] =0000000000000000
MemRead_reg =0
MemWrite =0
MemWrite_reg =0
MemtoReg =0
MemtoReg_reg =0
PCWrite =1
PC_Src =0
RegData1[63:0] =0000000000000000
RegData2[63:0] =0000000000000000
RegWrite =0
RegWrite_reg =0



```
WB_Alu_result[63:0] =0000000000000000
WB_MemReadData[63:0] =0000000000000000
WB_MemtoReg =0
WB_Rd[4:0] =00
WB_RegWrite =0
WB_WriteData[63:0] =0000000000000000
alu_in2[63:0] =0000000000000000
bit30 =0
clk =0
flush =0
funct3[2:0] =000
funct7[6:0] =00
opcode[6:0] =00
pc[63:0] =0000000000000021
pc_next[63:0] =0000000000000022
pc_plus4[63:0] =0000000000000022
rd[4:0] =00
reset =0
rs1[4:0] =00
rs2[4:0] =00
zero =1
```



```
(base) xponent@Mythroid:~/Documents/verilog/submission_pipeline/pipelined$ iverilog -o cpu_sim main.v
(base) xponent@Mythroid:~/Documents/verilog/submission_pipeline/pipelined$ vvp cpu_sim
VCD info: dumpfile cpu_top_wave.vcd opened for output.
x5  =                    15
x6  =                    10
x7  =                     8
x8  =                    13
x9  =                    45
mem[17] =                    12
x10 =                     3
x11 =                     2
x12 =                     2
x13 =                     0
x14 =                    13
x15 =                     0
x16 =                    21
x17 =                    26
x0  =                     0
```



```
add x5, x1, x2 // x5 = 5 + 10 = 15
sub x6, x2, x0 // x6 = 10 - 0 = 10
and x7, x3, x4 // x7 = 12 & 9 = 8
or x8, x3, x4 // x8 = 12 | 9 = 13
ld x9, 27(x1) // x9 = mem[27 + 5] = 45
sd x3, 12(x1) // mem[12 + 5] = mem[17] = x3 = 12
sub x10, x3, x4 // x10 = 12 - 9 = 3
add x11, x10, x4 // x11 = 3 + 9 = 12 ****DATA HAZARD****
sub x11, x1, x10 // x11 = 5 - 3 = 2 ****DATA HAZARD****
add x12, x11, x0 // x12 = 2 + 0 = 2 ****DOUBLE DATA HAZARD****
ld x13, 0(x3) // x13 = mem[0 + 12] = mem[12] = 13
add x14, x13, x0 // x14 = 13 + 0 = 13 ****LOAD USE DATA HAZARD - ONE STALL****
beq x2, x6, 10 // x2 == x6 == 10 --> PC = PC + 10 ****BRANCH HAZARD****
add x15, x2, x3 // x15 = 10 + 12 = 22 ****SHOULD NOT HAPPEN****
add x15, x2, x3 // x15 = 10 + 12 = 22 ****SHOULD NOT HAPPEN****
add x15, x2, x3 // x15 = 10 + 12 = 22 ****SHOULD NOT HAPPEN****
add x15, x2, x3 // x15 = 10 + 12 = 22 ****SHOULD NOT HAPPEN****
add x15, x2, x3 // x15 = 10 + 12 = 22 ****SHOULD NOT HAPPEN****
add x15, x2, x3 // x15 = 10 + 12 = 22 ****SHOULD NOT HAPPEN****
add x15, x2, x3 // x15 = 10 + 12 = 22 ****SHOULD NOT HAPPEN****
add x15, x2, x3 // x15 = 10 + 12 = 22 ****SHOULD NOT HAPPEN****
add x15, x2, x3 // x15 = 10 + 12 = 22 ****SHOULD NOT HAPPEN****
add x16, x3, x4 // x16 = 12 + 9 = 21 ****PC SHOULD RESUME HERE****
beq x13, x15, 10 // x13(13) != x15(0) ****BRANCH NOT TAKEN****
add x17, x13, x14 // x17 = 13 + 13 = 26
add x0, x13, x14 // x0 = 0
sub x13, x13, x14 // x13 = 13 - 13 = 0
```

CONTRIBUTIONS:

All the project was made by equal contributions by all the team members together. Dhimant majorly focused on pipelined implementation, forwarding and pipelining registers, Jal did control hazards, Dataarnoor and Jal focused on sequential implementation and stalling in pipelining. Dataarnoor made the report. Dataarnoor did sequential testing and Jal and Dhimant did testing of pipelined implementation.