IPA PROJECT REPORT

TEAM MATE-1:

NAME: NUNNA SRI ABHINAYA

ROLL NO: 2023102071

TEAM MATE-2:

NAME: ANUMULA VENKATA SAI SREE SAHITHI

ROLL NO: 2023112002

TEAM MATE-3:

NAME: KEERTHI SEELA

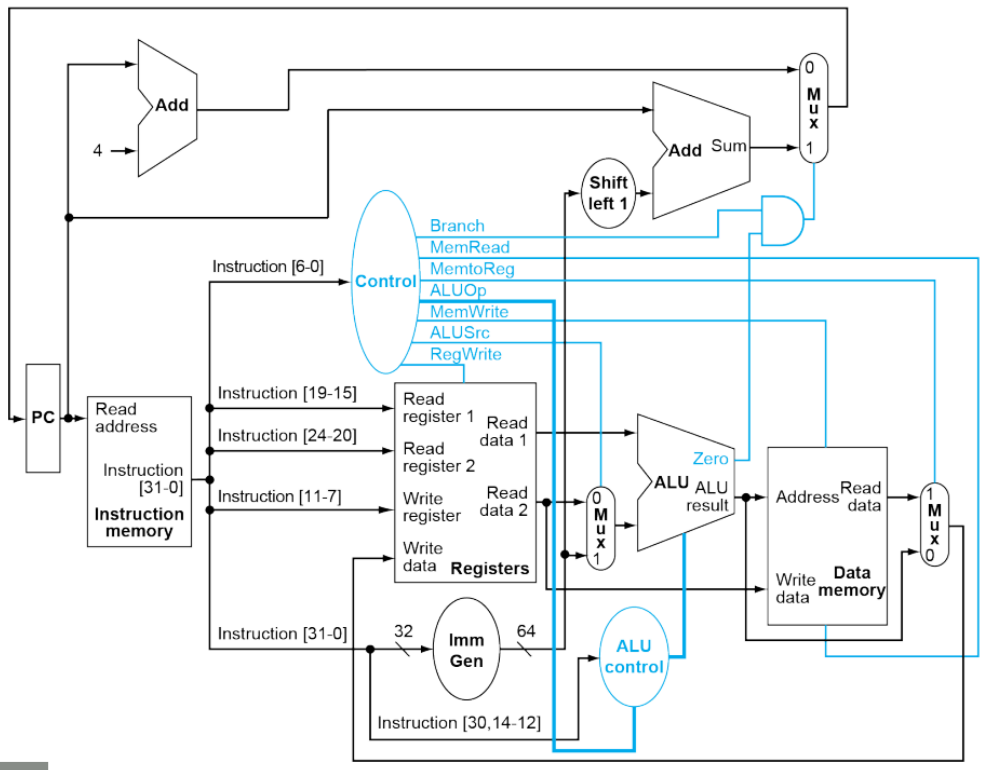
ROLL NO: 2023102012

PROCESSOR ARCHITECTURE DESIGN USING RISC-V

The processor is designed in Verilog and incorporates both **sequential** and **pipelined** architectures. These architectures define how instructions are executed and how efficiently the processor handles multiple instructions simultaneously.

SEQUENTIAL PART:

1.DIAGRAM:



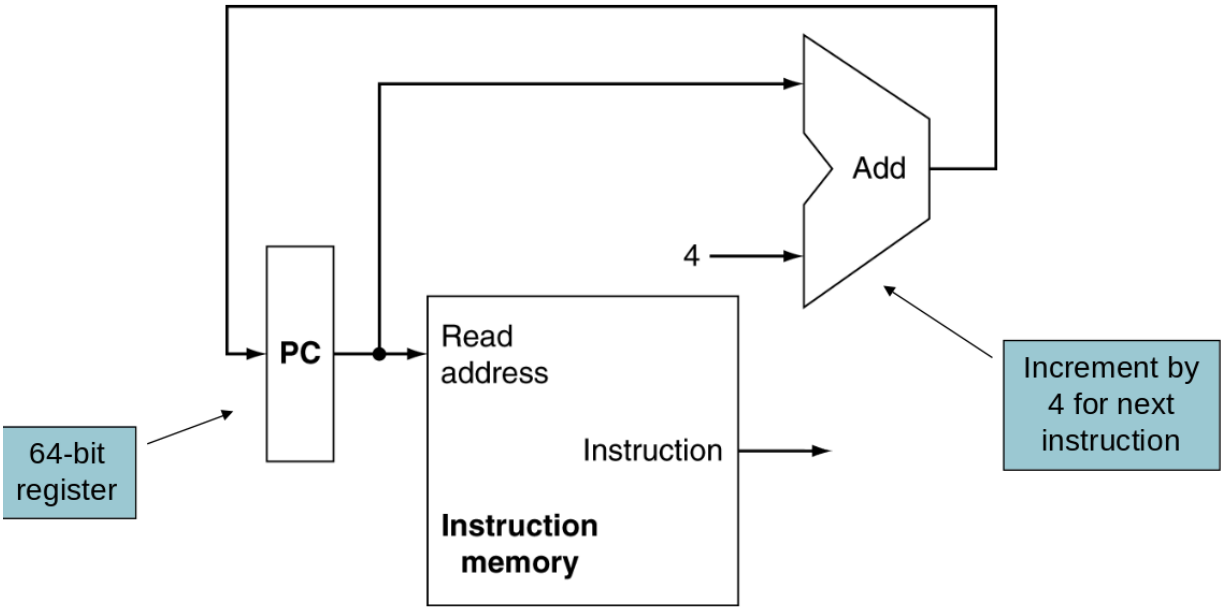
In the sequential-based processor design, instructions are executed in a **single-cycle** manner, meaning that each instruction undergoes the entire cycle of **fetch, decode, execute, and write-back** before moving to the next instruction. The processor operates on a **positive-edge clock**, meaning updates to the **program counter (PC)** and registers occur at the rising edge of the clock signal. This approach simplifies control logic but may lead to inefficiencies since each instruction must complete before the next one begins.

At every positive clock edge, the processor fetches the next instruction from memory and processes it in a sequential manner. Each instruction undergoes the complete execution cycle—fetch, decode, execute, memory access, and write-back—before the next instruction begins. There is no overlap between instructions, ensuring that each instruction completes fully within a single clock cycle before moving to the next one. This approach simplifies control and eliminates pipeline hazards, though it limits overall performance compared to pipelined architectures.

2.A simple Processor is divided into 4 parts:

FETCH:

PC UPDATE:



In RISC-V, the **Program Counter (PC) update** depends on the instruction type. Typically, the PC is incremented by **4** for standard 32-bit instructions (`PC = PC + 4`). However, for **control transfer instructions** such as **branches (`beq` , `bne` , etc.)** the PC is updated based on the computed target address instead of a fixed increment. For **branches**, the PC update depends on the condition being met. If the branch is taken, the PC jumps to the target address; otherwise, it continues sequentially.

The primary goal of the Fetch stage is to retrieve the instruction from memory using the Program Counter (PC) as the memory address. The Instruction Memory provides the 32-bit instruction. The instruction bits are used in later stages, as shown. Because it is a RISC-V, a fixed 4 is added to the PC and the result is sent

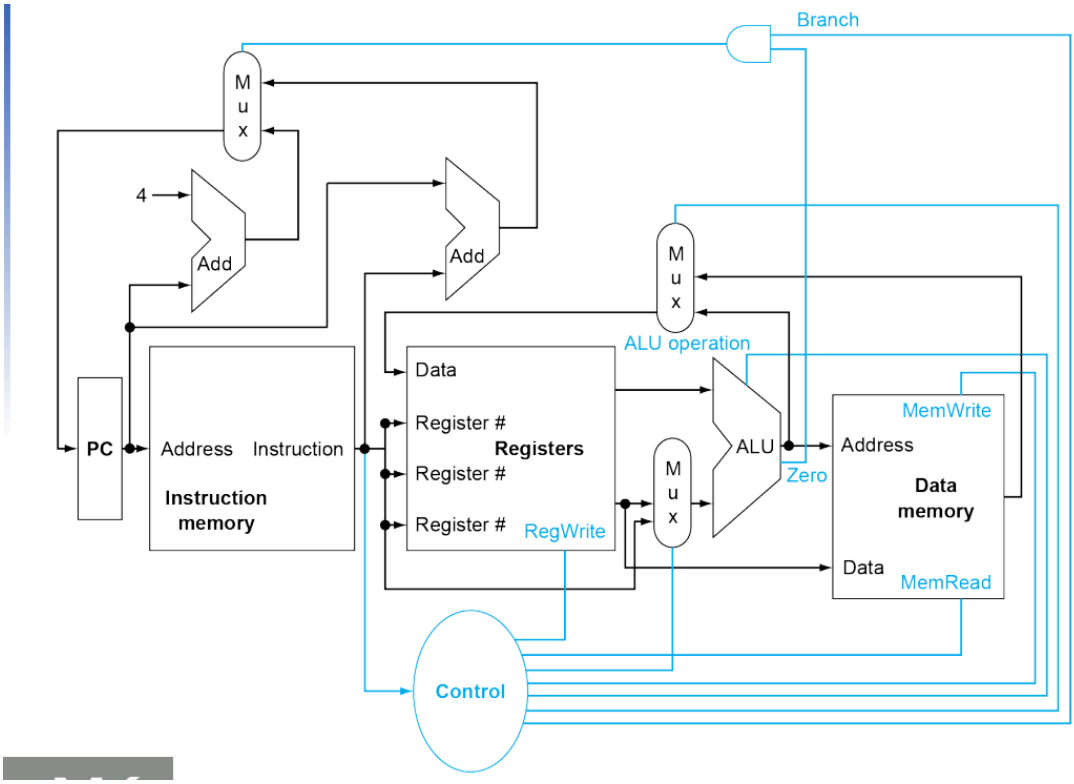
to the PC. In essence, the Fetch stage aims to retrieve the current instruction and compute the address for the next instruction assuming that it is a sequential execution .

In our code we implemented instruction decoding inside fetch

Challenges Encountered:

When a branch needs to go back to a previous instruction, we forgot to handle the negative offset correctly. Because of this, the program counter (PC) didn't update properly, causing the wrong instruction to be fetched.

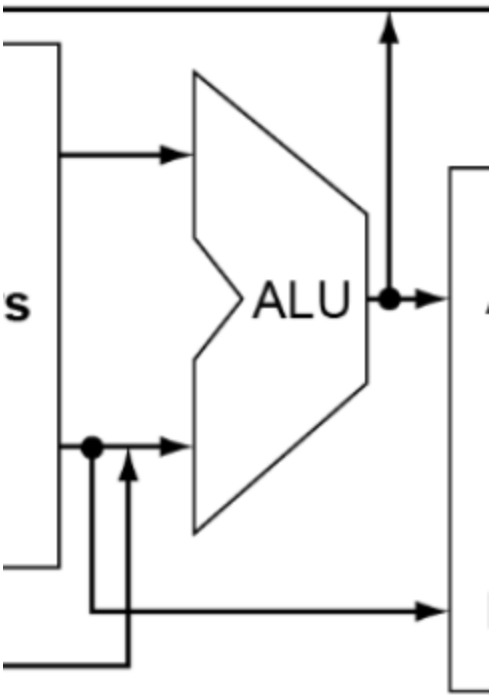
DECODE AND CONTROL:



The Decode stage reads up to two operands from the Register File, obtaining values `Read data 1` and/or `Read data 2`. It reads registers specified by instruction fields

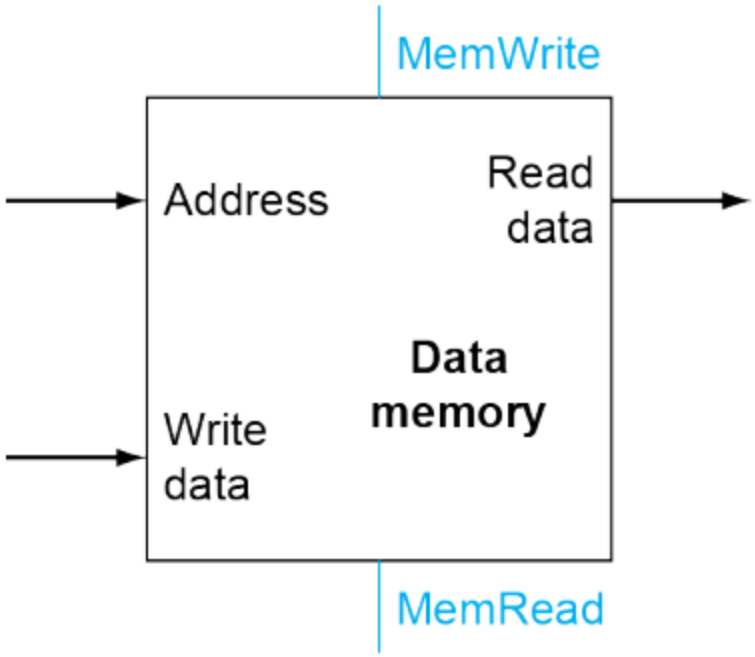
`Instruction[19-15]` (rs1) and `Instruction[24-20]` (rs2). The `Imm Gen` block generates the immediate value as well and is sent to Execute.

EXECUTE:



In the Execute stage, the Arithmetic Logic Unit (ALU) performs the operation specified by the instruction's `ALUop` control signal, computes effective memory addresses, or calculates the branch target. The resulting value is termed `ALU result`. Additionally, the *Branch* instruction performs a comparison, outputting `Zero`, to determine if the jump will be performed or the next instruction must follow up.

MEMORY:



The Memory stage may write data to memory if MemWrite is asserted, or it may read data from memory if MemRead is asserted, which results in the value `Read data`.

WRITE BACK:

The Write Back stage writes the result to the register file, which is specified by the `Instruction[11-7]` and if the `RegWrite` control signal is asserted, where the data is taken from either the ALU result or the Read Data.

PC UPDATE:

The PC is set to the address of the next instruction which is determined by the result of the Excute and Control Unit in order to take the appropriate address. If Branch is not asserted, then the PC takes the PC+4. If Branch is asserted, then it will take the `Add Sum` in the Execute block.

1)FETCH AND INSTRUCTION DECODING:

First stage in sequential implementation.

In the **RISC-V fetch stage**, the **Program Counter (PC)** holds the memory address of the instruction that needs to be executed next. This address is sent to **Instruction Memory**, which retrieves the instruction stored at that location, it is array storing 256 instructions each 32 bits value. Once retrieved, the instruction is passed to the **Control Unit** and other components for decoding and execution.

INPUTS:

- `instr` (32-bit Instruction) – The fetched instruction
- `branch_taken` (1-bit) – Control signal indicating if a branch should be taken
- `branch_target` (64-bit) – The target address for a branch/jump

OUTPUTS:

- `opcode` (7-bit) – Operation code identifying the instruction type
- `rd` (5-bit) – Destination register (if applicable)
- `rs1` (5-bit) – Source register 1
- `rs2` (5-bit) – Source register 2 (only for certain instructions)
- `funct3` (3-bit) – Function field for instruction classification
- `funct7` (7-bit) – Additional function field (used for R-type)
- `imm` (64-bit) – Immediate value extracted from the instruction

IMMEDIATE GENERATION:

1.R-FORMAT:(NOT REQUIRED)

```
funct7  rs2  rs1  funct3  rd  opcode
[6:0]  [4:0] [4:0]  [2:0] [4:0]  [6:0]
```

2.LOAD:

```
imm[11:0]  rs1  funct3  rd  opcode
[11:0]   [4:0]  [2:0]  [4:0]  [6:0]
```

```
IMMEDIATE:
assign imm_out = {{52{imm[11]}}, imm[11:0]};
```

3.STORE:

```
imm[11:5]  rs2  rs1  funct3  imm[4:0]  opcode
[6:0]    [4:0]  [4:0]  [2:0]    [4:0]  [6:0]

IMMEDIATE:
assign imm_out = {{52{imm[11]}}, imm[11:5], imm[4:0]};
```

4.BEQ:

```
imm[12]  imm[10:5]  rs2  rs1  funct3  imm[4:1] imm[11] opcode
[1]     [6:0]    [4:0]  [4:0]  [2:0]   [3:0] [1]    [6:0]

IMMEDIATE:
assign imm_out = {{51{imm[12]}}, imm[12], imm[10:5], imm[4:1], 1'b0};
```

After fetching, the **PC must be updated** to point to the next instruction. Normally, it is incremented by **4** for standard 32-bit instructions. In cases where the fetched instruction is a **branch or jump**, the PC is updated to a different address rather than simply incrementing. The new address is calculated based on the branch offset or jump target, ensuring the correct instruction sequence is followed.

The **Control Unit** plays a crucial role in this stage by enabling **memory access (**`MemRead`**)** and determining whether the fetched instruction is compressed.

Control & Next Address Computation

1. `nextp` (64-bit) – The next program counter value (`PC+4` or `branch_target` if a branch is taken)

2. `instr_valid` (1-bit) – Indicates whether the fetched instruction is valid (based on known opcodes)
3. `imem_error` (1-bit) – Memory access error flag

For a **normal sequential execution**, if the PC is at **0x1000** and fetches a **32-bit instruction**, the next PC will be **0x1004**. However, for a **branch instruction** like `beq` at **0x3000**, if the branch condition is met and the offset is **+16**, the PC will jump to **0x3020** instead of the next sequential instruction at **0x3004**.

2)DECODE AND WRITE BACK:

The Decode and Write Back stages are conceptually combined since they both interact heavily with the register file(this is done in main code). The Write Back operation occurs only at the positive edge of the clock, while the decode logic responds continuously to input changes.

INPUTS:

- `opcode` (7-bit) – Identifies the instruction type
- `rs1`, `rs2`, `rd` (5-bit each) – Source and destination registers
- `funct3` (3-bit) – Further classifies operations
- `funct7` (7-bit) – Further classifies operations for R-type
- `imm` (64-bit) – Immediate value

OUTPUTS:

- `ALU_src` – Determines if ALU operand comes from register (`0`) or immediate (`1`)
- `Mem_to_Reg` – Determines if the value to be written comes from memory (`1`) or ALU (`0`)
- `Reg_Write` – Enables register writing (`1`)
- `Mem_Read` – Enables memory read (`1`)

- `Mem_Write` – Enables memory write (`1`)
- `Branch_en` – Enables branch execution (`1`)
- `ValA` – First operand for ALU (from register)
- `ValB` – Second operand for ALU (from register or immediate)
- `opcode_out, rs1_out, rs2_out, rd_out, funct3_out, funct7_out, imm_out` – Pass-through decoded instruction fields

The RISC-V register file features multiple ports: two read ports (connected to 'Read register 1' and 'Read register 2' in the image) and one write port ('Write register' in the image). Each port has an address (register number) and a 64-bit data connection . For read operations, the register addresses come from `Instruction[19:15]` (rs1) and `Instruction[24:20]` (rs2), specifying which registers to read. These produce `Read data 1` and `Read data 2` as outputs. The write port uses `Instruction[11:7]` (rd) for its address, receiving data from either the ALU result or Data Memory output.

R-FORMAT:

- **Control Signals:**
  - `Reg_Write = 1` → Write result to register `rd`
  - `ValA = registers[1]` → Read value from `rs1`
  - `ValB = registers[2]` → Read value from `rs2`
- **Expected Behavior:**
  - ALU performs addition using values of `x1` and `x2`
  - The result is written to register `x3`

LOAD:

- **Instruction Meaning:** `ld x8, offset(x7)` → Load value from memory at address `x7 + offset`
- **Control Signals:**

    - `ALU_src = 1` → Use `imm` as second operand
    - `Mem_to_Reg = 1` → Data from memory
    - `Mem_Read = 1` → Enable memory read
    - `Reg_Write = 1` → Write result to `rd`
    - `ValA = registers[7]`
    - `ValB = imm`
- **Expected Behavior:**
    - Compute memory address: `x7 + imm`
    - Load memory value into `x8`

STORE:

- **Instruction Meaning:** `sd x10, offset(x9)` → Store `x10` in memory at `x9 + offset`
- **Control Signals:**
    - `ALU_src = 1` → Use `imm` as second operand
    - `Mem_Write = 1` → Enable memory write
    - `ValA = registers[9]`
    - `ValB = registers[10]`
- **Expected Behavior:**
    - Compute memory address: `x9 + imm`
    - Store `x10` value at that address

BEQ:

- **Instruction Meaning:** `beq x11, x12, offset` → If `x11 == x12`, branch to `PC + offset`
- **Control Signals:**
    - `Branch_en = 1` → Enable branching
    - `ValA = registers[11]`
    - `ValB = registers[12]`
- **Expected Behavior:**

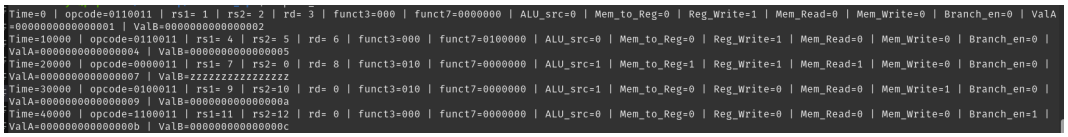
- Compare `x11` and `x12`
- If equal, update PC to `PC + imm`

WRITE-BACK:

- **When MemtoReg = 1** → The data from memory (from the MEM stage) is written to the register file (for `ld` instructions).
- **When MemtoReg = 0** → The result from the ALU (from the EX stage) is written back (for `add`, `sub`, `and`, `or` instructions).

If **RegWrite = 1**, the selected data is written into the register file.

The RISC-V register file contains 32 registers (x0-x31), each storing 64-bit integers, with register x0 permanently set to zero. The stack pointer (x2, also known as `sp`) starts at a high value to prevent negative values during push operations, which could cause memory access errors.



3)EXECUTE:

INPUT:

Signal	Width	Description
opcode	7 bits	RISC-V opcode to determine instruction type
rs1	5 bits	Source register 1 index
rs2	5 bits	Source register 2 index
rd	5 bits	Destination register index
funct3	3 bits	Function field for instruction classification

funct7	7 bits	Additional function field for R-type instructions
imm	64 bits	Immediate value for I, S, and B-type instructions

OUTPUT:

Signal	Width	Description
result	64 bits	Output result from ALU
carry_alu	1 bit	Carry-out flag from ALU
overflow_alu	1 bit	Overflow flag from ALU
zero_flag	1 bit	Zero flag (1 if result is 0

- If `funct3 = 3'b000` and `funct7 = 7'b0000000`, perform **ADD** using `bit64_adder` where `result = A + B`.

- If `funct3 = 3'b000` and `funct7 = 7'b0100000`, perform **SUB** using `bit64_subtractor` where `result = A - B`.

- If `funct3 = 3'b001` and `funct7 = 7'b0000000`, perform **Shift Left Logical (SLL)** using `shift_sll` where `result = A << B[5:0]`.

- If `funct3 = 3'b010` and `funct7 = 7'b0000000`, perform **Set Less Than (SLT)** using `compare_bitwise_64_signed` where `result = (A < B) ? 1 : 0`.

- If `funct3 = 3'b011` and `funct7 = 7'b0000000`, perform **Set Less Than Unsigned (SLTU)** using `compare_bitwise_64` where `result = (A < B) ? 1 : 0` (unsigned comparison).

- If `funct3 = 3'b100` and `funct7 = 7'b0000000`, perform **Bitwise XOR (XOR)** using `xor_gate` where `result = A ^ B`.

- If `funct3 = 3'b101` and `funct7 = 7'b0000000`, perform **Shift Right Logical (SRL)** using `shift_srl` where `result = A >> B[5:0]`.

- If `funct3 = 3'b101` and `funct7 = 7'b0100000`, perform **Shift Right Arithmetic (SRA)** using `shift_sra` where `result = A >>> B[5:0]`.

- If `funct3 = 3'b110` and `funct7 = 7'b0000000`, perform **Bitwise OR (OR)** using `or_gate` where `result = A | B`.

- If `funct3 = 3'b111` and `funct7 = 7'b0000000`, perform **Bitwise AND (AND)** using `and_gate` where `result = A & B`.

For **Load Instructions** (`opcode = 7'b0000011`):

- If `funct3 = 3'b011`, perform **Load (LD)** using `bit64_adder` where `result = A + sign-extended imm`.

For **Store Instructions** (`opcode = 7'b0100011`):

- If `funct3 = 3'b011`, perform **Store (SD)** using `bit64_adder` where `result = A + sign-extended imm`.

For **Branch Instructions** (`opcode = 7'b1100011`):

- If `funct3 = 3'b000`, perform **Branch Equal (BEQ)** using `bit64_subtractor` where `zero_flag = (A == B) ? 1 : 0`.

The Execute stage performs the operation specified by the instruction. It receives `Read data 1` and either `Read data 2` or an immediate value from the `Imm Gen`. The `ALU control` determines the ALU operation based on `Instruction[30,14-12]`. The ALU computes the `ALU result`. For branches, the ALU also calculates the branch target. The `Branch`

signal depends on the instruction and `Zero` signal, and it controls whether the PC is updated with the branch target. The outputs are `ALU result` and signal for when branch should be taken.

If there is a overflow in addition and subtraction then overflow is flagged.INLCUDED IN OUR CODE

4)MEMORY:

INPUTS:

- `rs1 [63:0]` – Base register address (signed)
- `rs2 [63:0]` – Register storing the value to be written (signed)
- `op [6:0]` – Opcode (determines if the instruction is Load or Store)
- `f7 [6:0]` – Funct7 field (not directly used in load/store operations)
- `f3 [2:0]` – Function code (determines load/store type, e.g., LD or SD)
- `imm12 [11:0]` – 12-bit signed immediate offset (used for address calculation)
- `clk` – Clock signal (synchronous memory operations)
- `memread` – Memory Read Enable (1 for read operation)
- `memwrite` – Memory Write Enable (1 for write operation)
- `mem2reg` – Control signal (decides whether to use memory output)
- `alures [63:0]` – ALU result (not used directly in this module)

OUTPUT:

- `rd [63:0]` – Destination register (loaded value when `memread=1`)
- `address_error` – Flag indicating out-of-bounds memory access

In the **Memory stage** of RISC-V, data is read from or written to memory for **load** (`ld`, `lw`, `lh`, `lb`) and **store** (`sd`, `sw`, `sh`, `sb`) instructions. The memory address is computed using a base register and an immediate offset. If the instruction is a **load**, the data at the calculated address is read and passed to the next stage. If it is a **store**, the computed address is used to write data from a register into memory. Control signals like `MemRead` and `MemWrite` determine whether a read or write should occur, and `MemtoReg` decides if the memory output is written back to registers. Errors such as out-of-bounds memory access or simultaneous read/write operations result in program termination. This stage interacts with the Fetch and Write-back stages to ensure smooth execution of memory-related instructions.

Memory Read (`memread = 1`)

- If `f3 = 3'b011` (indicating **LD instruction**), the memory at `addr` is read into `rd`.
- If `addr` is **out of range**, `rd = 64'hX` and `address_error = 1`.
- If valid, `rd = memory[word_addr]`.

Memory Write (`memwrite = 1`)

- If `f3 = 3'b011` (indicating **SD instruction**), `rs2` is stored at `addr`.
- If `addr` is **out of range**, the memory is **not updated**, and `address_error = 1`.
- If valid, `memory[word_addr] = rs2`.

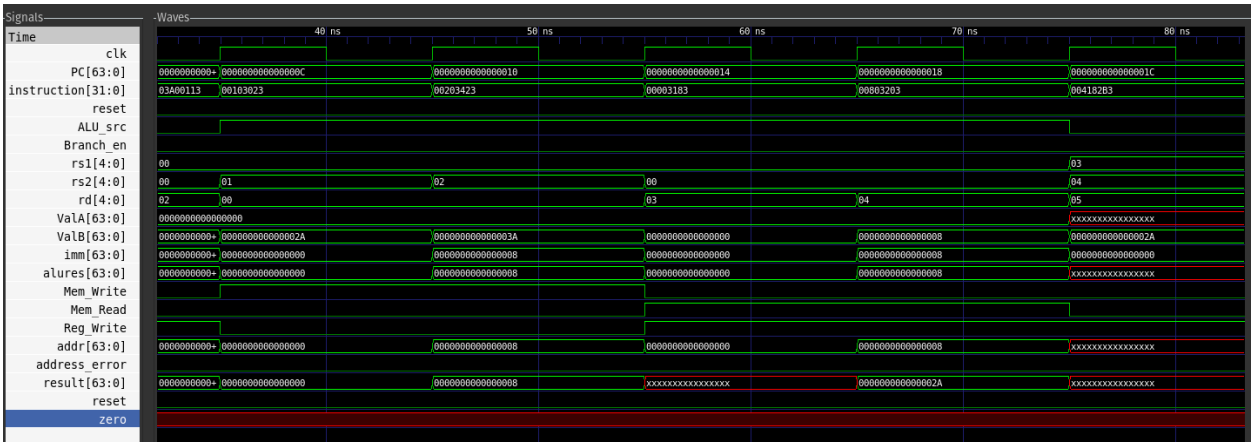
3.SEQUENTIAL TIMING:(RECHECK)

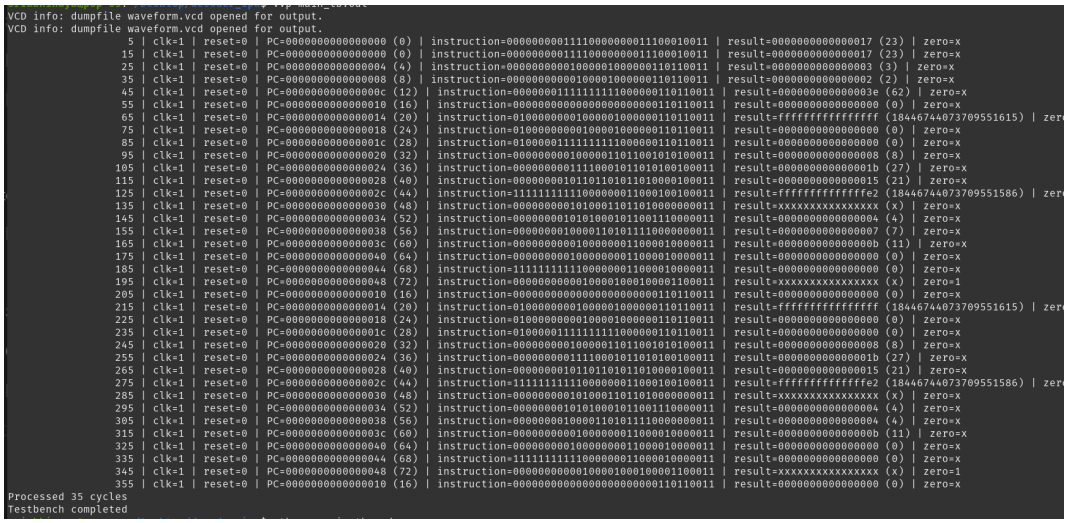
- Each instruction is executed in a single clock cycle in a single-cycle implementation, while in a pipelined design, different stages execute across multiple cycles.

- Combinational logic does not require explicit clock control; values propagate as soon as inputs change.
- In Verilog, this is denoted by `always @(*)`. Instruction memory is typically modeled as a combinational unit since it is only read during the fetch stage.
- The **PC, control registers, data memory, and register file** are updated at the **positive edge of the clock cycle**, ensuring synchronized execution.
- The **PC is updated every clock cycle**, either sequentially (`PC = PC + 4`) or based on branch/jump conditions.
- Conditional control registers update only when integer or floating-point operations are executed.
- Data memory is written only for **store instructions (`sw`, `sd`)**, maintaining memory consistency.
- By the **no reading back** principle, new state values are computed every cycle but only take effect at the **positive edge of the clock transition**.

4.TESTCASES

Screenshot from 2025-03-08 22-12-49.png





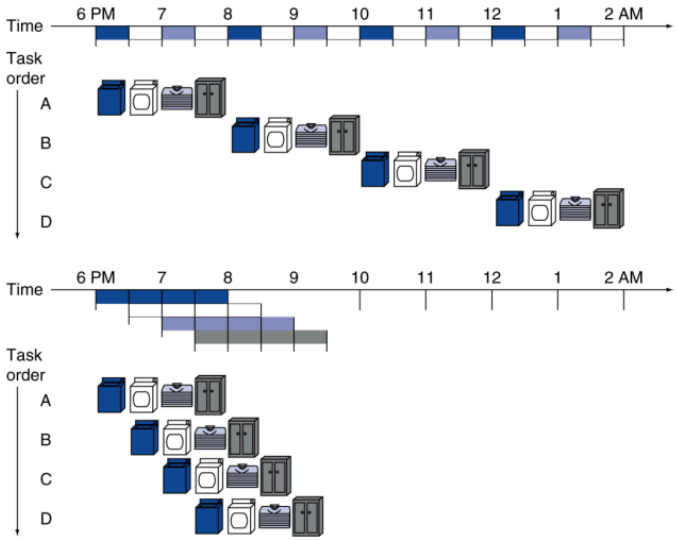
5.CHALLENGES FACED

- **Incorrect PC Update for Branches**
  - The **Program Counter (PC)** is responsible for keeping track of which instruction to execute next.
  - When a **branch instruction** was used to jump to a different instruction (especially to a previous one), the processor did not correctly calculate the **negative offset** (the amount by which the PC should move backward).
  - This mistake caused the processor to **fetch the wrong instruction**, leading to incorrect execution.
- **Sign Extension Errors**
  - Some instructions use **immediate values** (small numbers embedded directly in the instruction) for calculations.
  - In certain cases, these immediate values need to be **sign-extended** (converted from a smaller bit-width to a larger one while preserving their positive or negative sign).
  - The processor did not correctly handle the **sign bit**, which caused errors in instruction execution.

- **Branch Condition Errors**
  - Branch instructions (like **BEQ - Branch if Equal**) check if two values are the same before deciding whether to jump to a different instruction.
  - The logic that determines **whether to take the branch or not** was not always working correctly.
  - This caused the program to **jump when it shouldn't** or **not jump when it should**, leading to incorrect program execution.

PIPELINE IMPLEMENTATION:

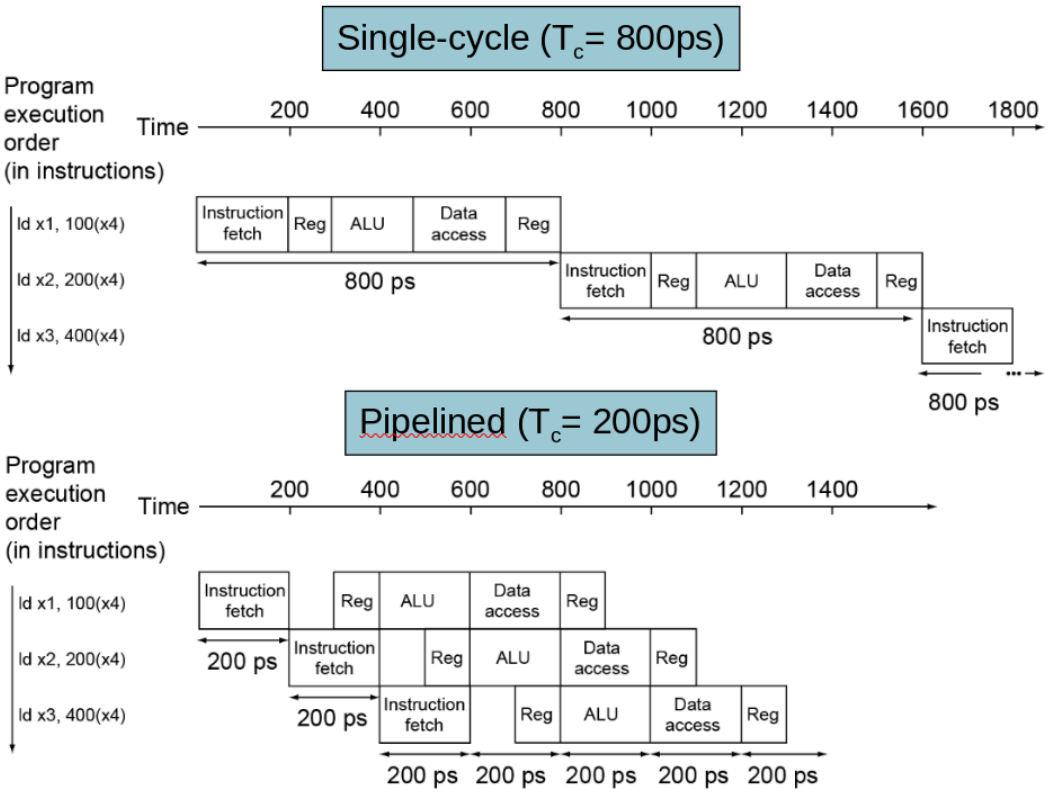
1.PIPELINING ANALOGY:



- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

pelining

(in processor)



$$T_{pipelined} = \frac{T_{non-pipelined}}{N_{stages}}$$

In pipelining:

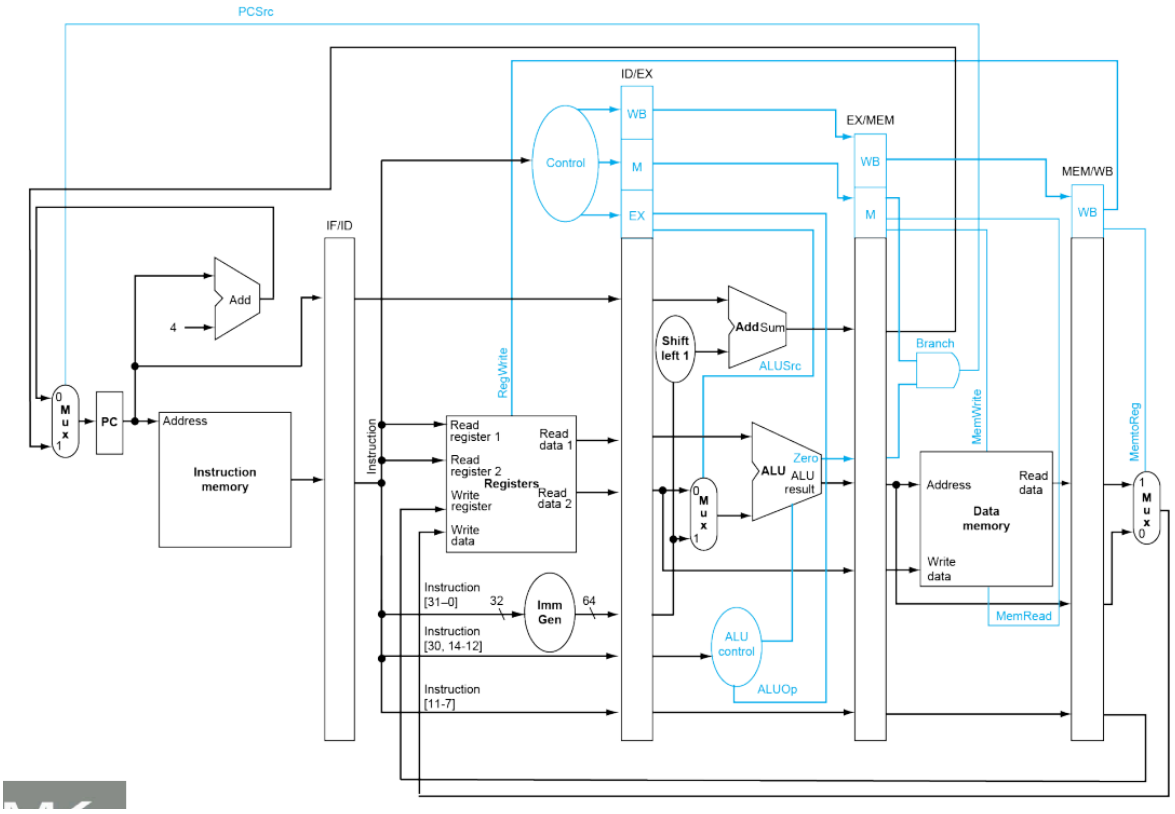
- **Latency:** The time for a *single* instruction to complete execution. Pipelining doesn't reduce individual instruction latency—it may actually *increase* it slightly due to pipeline overhead (latch delays, etc.). Each instruction must still pass through all stages.
- **Throughput:** The *number* of instructions completed per unit of time (instructions per second). Pipelining *significantly increases* throughput by processing multiple instructions simultaneously at different stages. In an ideal pipeline, one instruction completes every clock cycle, substantially improving execution speed.

2.COMPARISON:

	SINGLE CYCLE(Tc=800ps)	PIPELINED(Tc=200ps)
Latency (Time to execute one instruction)	800ps	1000ps (5 × 200ps)
Throughput (Instructions per second)	1.25×10^9	5×10^9
Speedup	1× (Baseline)	4× Higher Throughput

Change	Sequential	Pipelined Architecture
PC Update	In SEQ, **PC update happens at the end** of the clock cycle	PC is updated at the beginning of the cycle using a **separate IF/ID pipeline register**.
Pipeline Registers	No pipeline registers exist.	**Added pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB)** to store intermediate data between stages.
Instruction Execution	One instruction is fetched, decoded, executed, and written back **before** the next starts.	Multiple instructions are in different **pipeline stages simultaneously**.
Clock Cycle Duration	The clock cycle must be **long enough** to accommodate the slowest stage.	Each stage has its own **fixed duration**, allowing **shorter clock cycles**.
Hazard Handling	No hazards because instructions execute one by one.	**Data, control, and structural hazards** require solutions like **forwarding, stalls, and branch prediction**.

3.PIPELINED IMPLEMENTATION:



Sequential execution processes one instruction at a time, completing all stages before starting the next instruction. In contrast, pipelining overlaps instruction execution by dividing it into multiple stages with **pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB)**. This reduces instruction latency and increases throughput. However, pipelining introduces **hazards** (data, control, and structural), requiring techniques like **forwarding, stalls, and branch prediction** to maintain efficiency.

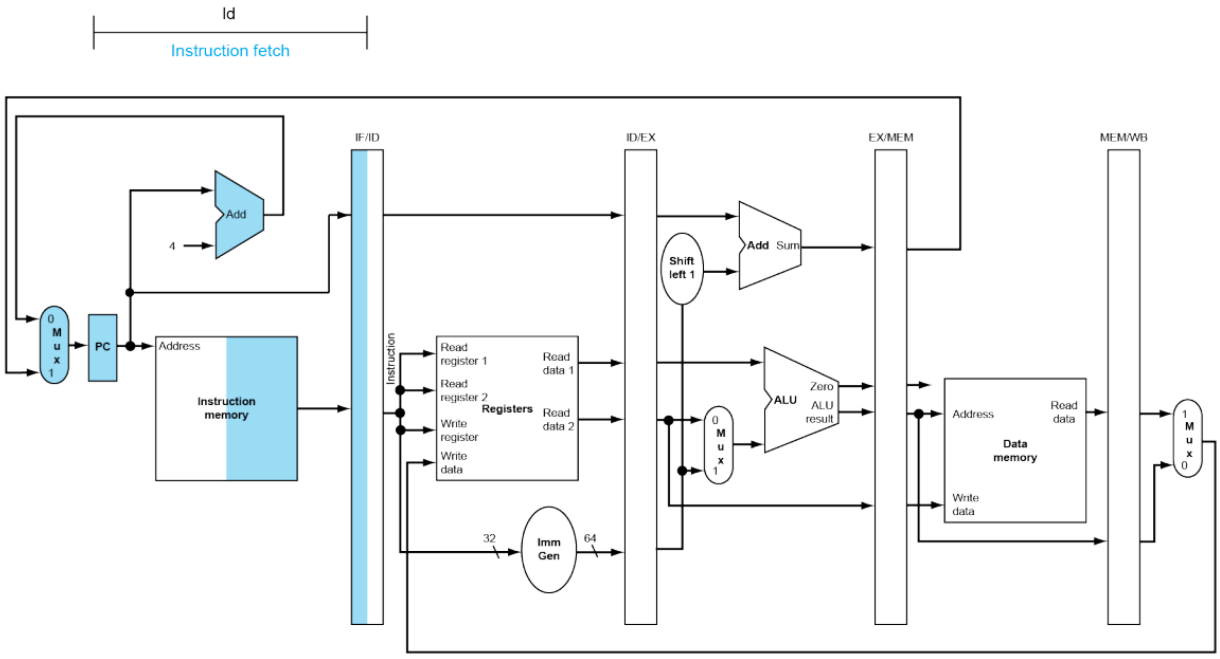
Pipeline Stages and Corresponding Registers:

Stage	Pipeline register	stored information
Instruction Fetch (IF)	IF/ID	PC, instruction
Instruction Decode (ID)	ID/EX	Register values, decoded instruction, immediate values, control signals
Execute (EX)	EX/MEM	ALU result, destination register, control signals

Memory Access (MEM)	MEM/WB	Memory read data, ALU result, destination register
Write Back (WB)	Output	Register file update

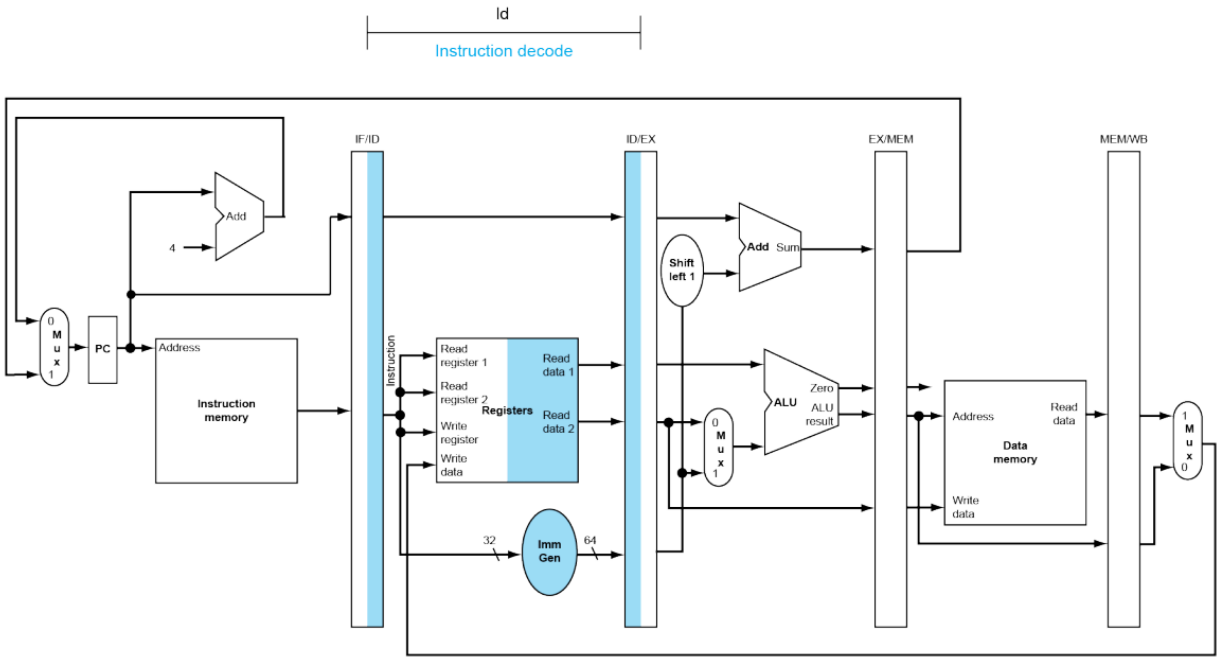
4.PIPELINE STAGES IMPLEMENTATION;

i)PC SELECTION AND FETCH STAGE:



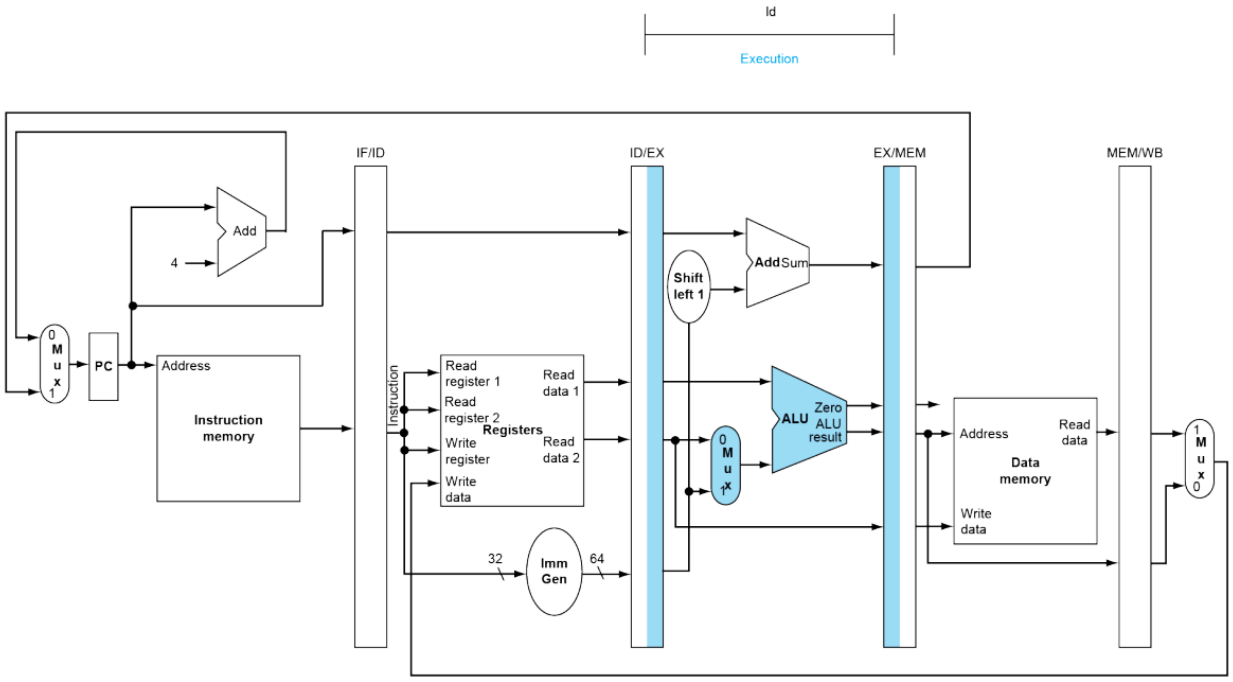
In the **RISC-V fetch stage**, the **Program Counter (PC)** holds the memory address of the instruction that needs to be executed next. This address is sent to **Instruction Memory**, which retrieves the instruction stored at that location, it is array storing 256 instructions each 32 bits value. Once retrieved, the instruction is passed to the **Control Unit** and other components for decoding and execution.Same as sequential.

ii) DECODE AND WRITEBACK:



In the **decode (ID) and write-back (WB) stages** of the RISC-V pipeline, the register identifiers **rs1** and **rs2** are used to fetch source operands from the **register file** during the decode stage. The **destination register (rd)** is determined in the write-back stage. The **pipeline register ID/EX** holds the decoded instruction, including register values read from the register file. Forwarding is handled using selection logic to avoid data hazards. The **Sel+Fwd A** block ensures correct operand selection by choosing between values from previous pipeline stages such as EX/MEM, MEM/WB, or directly from the register file. The priority of forwarding follows the sequence from the earliest stage to the latest, ensuring the most recent value is used.

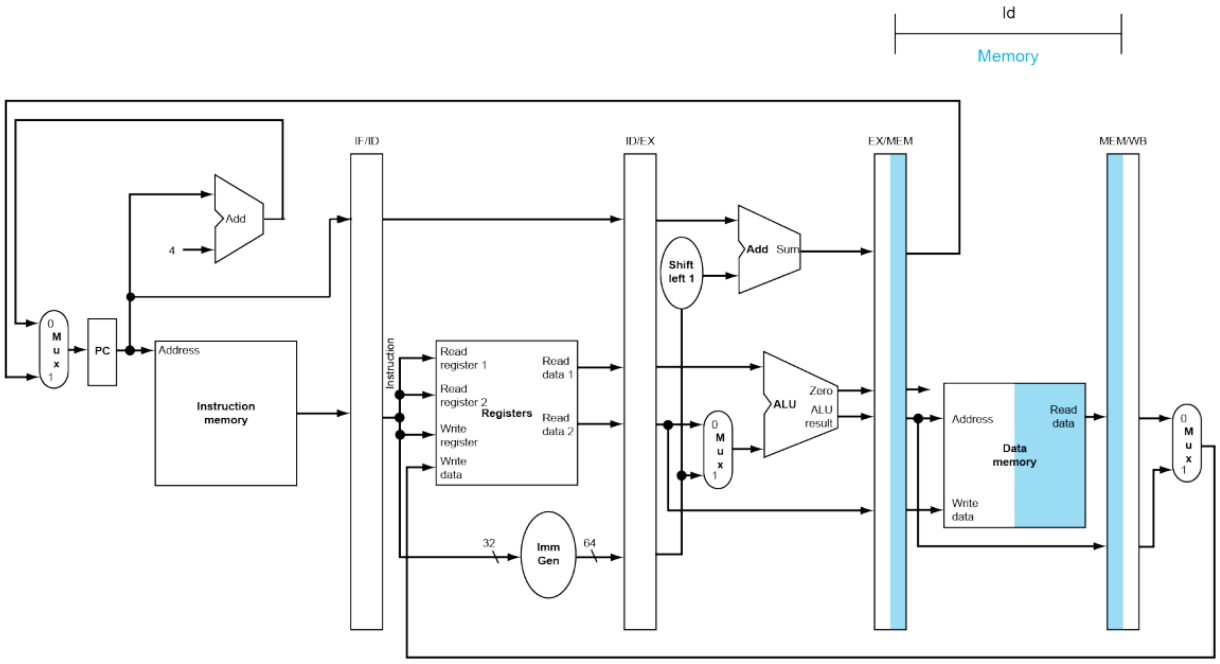
iii)EXECUTE;



The `ALU_Top` module is a **pipelined ALU execution unit** designed for RISC-V architecture. This module takes in opcode, register addresses (`rs1`, `rs2`, `rd`), function codes (`funct3`, `funct7`), an immediate value (`imm`), and the ALU source selection signal (`alusrc`). It first passes the second operand (`ValB` or `imm`) through a **multiplexer (`mux2to1`)**, allowing the ALU to choose between register-based and immediate-based operations. The selected value is then fed into the ALU for computation, producing results such as the final output (`result`), carry and overflow flags, and a zero flag for conditional execution. Additionally, the module outputs **pipeline register values** (`opcode_out`, `rs1_out`, `rs2_out`, etc.), ensuring data is correctly propagated through the pipeline stages.

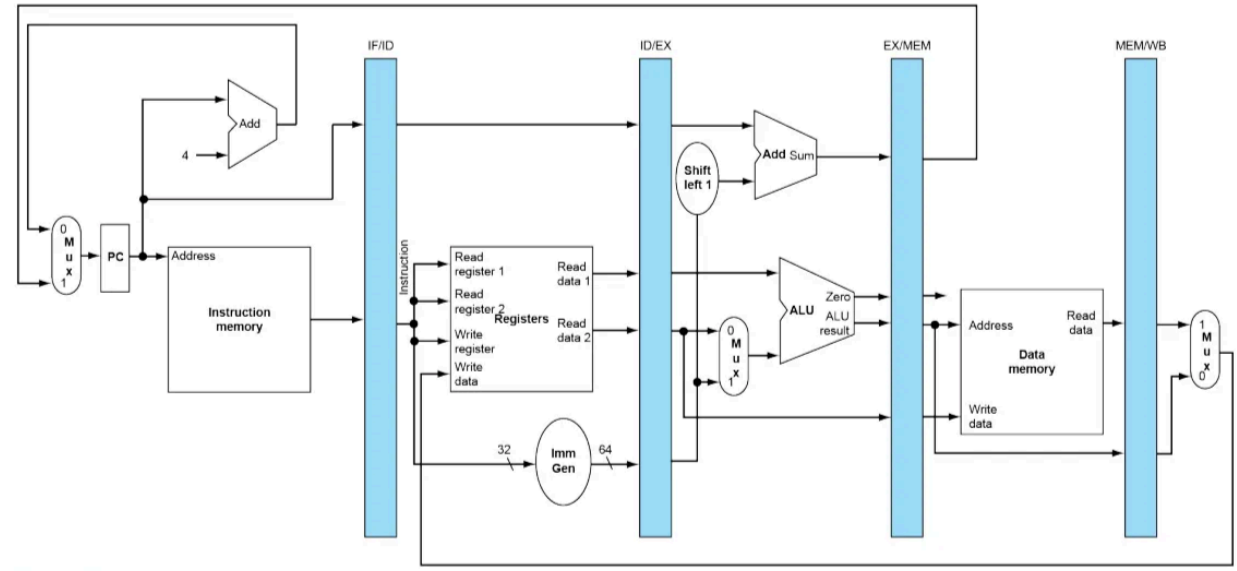
Unlike a **non-pipelined ALU**, which executes operations sequentially within a single cycle, this version is optimized for **pipeline execution**, ensuring intermediate results and control signals are stored for use in later stages. The **presence of pipeline registers** allows simultaneous processing of multiple instructions, preventing data hazards and improving instruction throughput. By separating **operand selection (ID stage)** from **execution (EX stage)** and forwarding results efficiently, this design enables faster and more efficient instruction execution in a **multi-stage pipeline processor**.

IV)MEMORY:



The `riscv_ld_sd` module implements the **load (LD) and store (SD) instructions** in a pipelined RISC-V processor. Unlike a non-pipelined design, where memory access and execution happen sequentially in a single cycle, this version follows a **pipelined execution model**, improving instruction throughput. In this design, the effective memory address (`addr`) is **precomputed in the Execute (EX) stage** using an ALU and then passed as an input to this module. The **memory operations (read/write) occur in the Memory (MEM) stage**, and the loaded value is written back in the **Write-Back (WB) stage**. A **boundary check** ensures that memory accesses stay within valid address ranges, setting an `address_error` flag if an invalid address is encountered. This pipelined approach **separates address computation from memory access**, allowing the processor to execute multiple instructions concurrently. Additionally, pipeline registers hold intermediate values, enabling efficient execution while handling data dependencies. This design ensures a significant **performance improvement** by preventing stalls and overlapping multiple instruction executions.

PIPELINE REGISTERS:

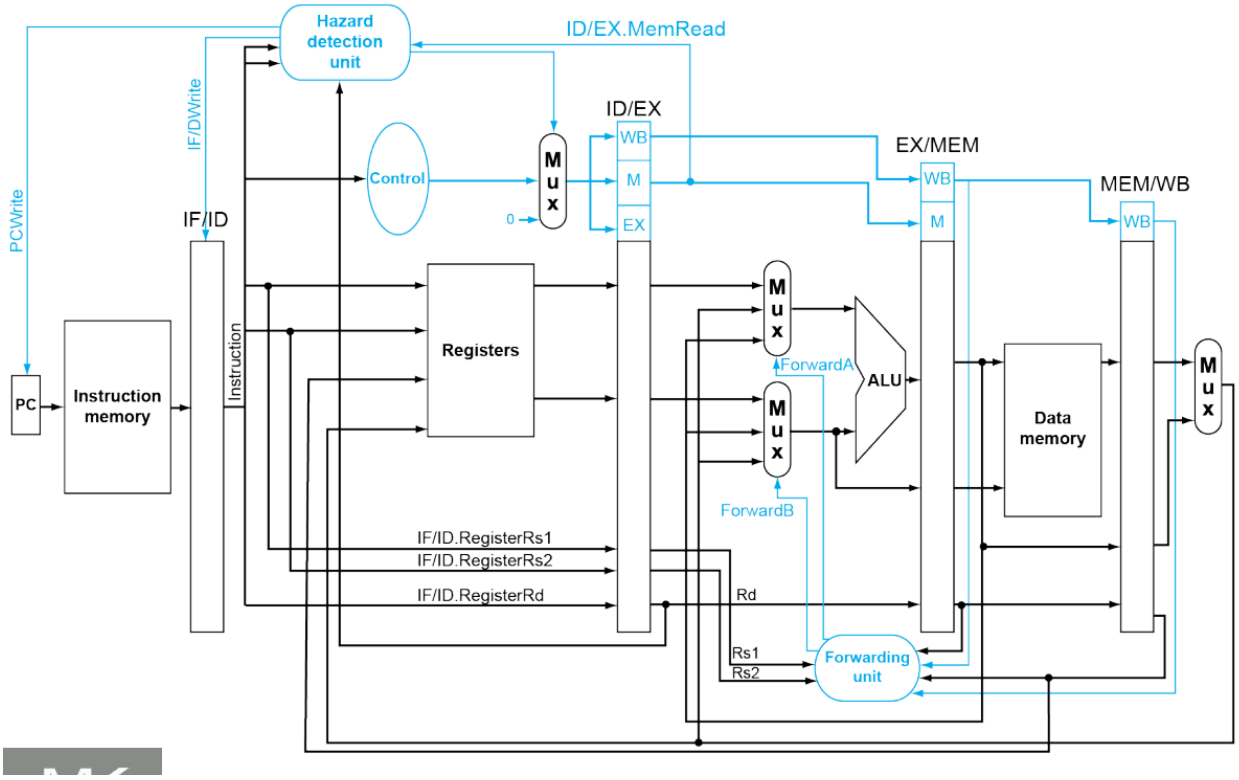


- **IF/ID Register**: Stores the fetched instruction and program counter (PC) for the decode stage.
- **ID/EX Register**: Holds register values, immediate values, and control signals for the execute stage.
- **EX/MEM Register**: Stores the ALU result and memory access control signals for the memory stage.
- **MEM/WB Register**: Holds data read from memory or ALU results for the write-back stage.

CHALLENGES:

- The branch conditions were not always evaluated correctly, leading to incorrect branching behavior.
- **Memory Access Issues:** Ensuring that memory accesses were properly synchronized to avoid out-of-bounds errors or conflicting memory read/write operations.

5.PIPELINING WITH FORWARDING AND HAZARD DETECTION:



HAZARD DETECTION:

1. Introduction

In pipelined processor architectures, instruction execution is divided into multiple stages to improve throughput. However, certain dependencies between instructions can lead to hazards that disrupt the smooth flow of execution. One critical type of hazard is the **load-use hazard**, which occurs when an instruction in the **ID/EX** stage is performing a memory load, and the next instruction in the **IF/ID** stage depends on that loaded value. To prevent incorrect execution, a **Hazard**

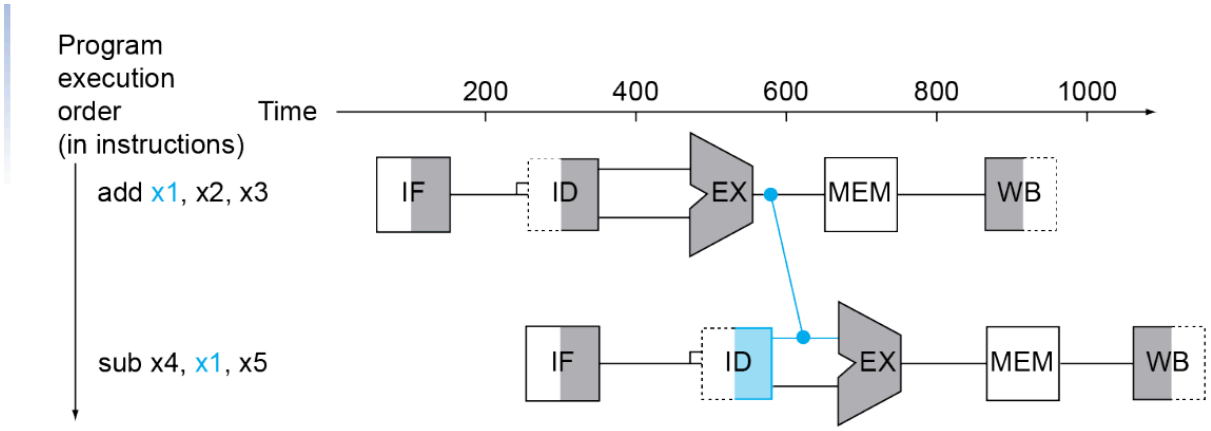
Detection Unit (HDU) is implemented to stall the pipeline when such dependencies are detected.

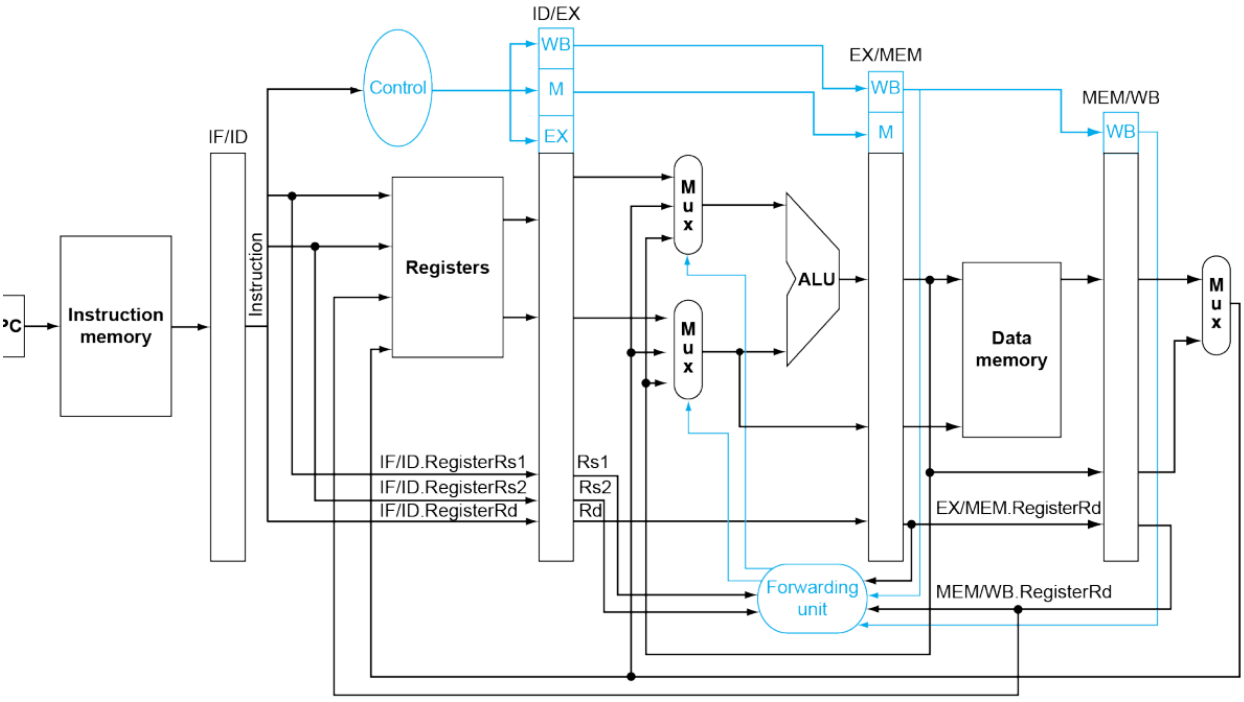
2. Role of the Hazard Detection Unit

The Hazard Detection Unit (HDU) is responsible for identifying load-use hazards and ensuring correct instruction execution by introducing stalls when necessary. It monitors the pipeline stages and halts the progression of instructions when a hazard is detected.

Hazard detection is a crucial component in pipelined architectures to ensure correct instruction execution. By identifying load-use hazards and introducing stalls when required, the **Hazard Detection Unit** prevents incorrect computation and maintains the integrity of the processor's operations. Although stalling impacts performance, it is necessary to avoid data corruption and ensure correct program execution. Advanced techniques like **forwarding** and **out-of-order execution** can further optimize pipeline efficiency and reduce stall penalties in modern processors.

FORWARDING:


Program
execution
order
(in instructions)
Time
200
400
600
800
1000
add x1, x2, x3
sub x4, x1, x5
IF
ID
EX
MEM
WB
IF
ID
EX
MEM
WB


ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
WB
M
WB
IF/ID
Instruction
Instruction memory
Registers
M u x
ALU
Data memory
M u x
M u x
IF/ID.RegisterRs1
IF/ID.RegisterRs2
IF/ID.RegisterRd
Rs1
Rs2
Rd
EX/MEM.RegisterRd
Forwarding unit
MEM/WB.RegisterRd

Forwarding (also called bypassing) is used in the pipelined processor to resolve data hazards without stalling execution. When an instruction in the EX stage requires an operand that is yet to be written back to the register file, forwarding allows the operand to be taken from a later pipeline stage instead of waiting for the register write to complete.

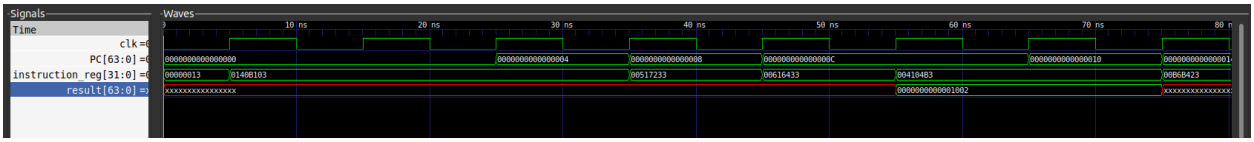
In the given Verilog design, forwarding is implemented through **pipeline_forward.v**, which detects dependencies between instructions. Specifically, forwarding is applied in the **Execute (EX) stage**, where the ALU receives forwarded values (`ValA_forwarded`, `ValB_forwarded`) instead of directly using `ValA_idex` and `ValB_idex`. These forwarded values come from either the **EX/MEM stage (most recent)** or the **MEM/WB stage (older but still valid)** if the operand matches a destination register (`rd`) from a previous instruction.

For example, if an `add` instruction writes to `x3` and a subsequent `sub` instruction requires `x3` before it is written back, the forwarding unit detects the dependency and provides the correct value from an earlier pipeline stage. This ensures correct execution without introducing stalls, improving pipeline efficiency.

- **EX/MEM Forwarding (ForwardA = 10 or ForwardB = 10)**
  - If the previous instruction (EX/MEM stage) writes to a register needed by the current instruction (ID/EX stage), forward the ALU result.
- **MEM/WB Forwarding (ForwardA = 01 or ForwardB = 01)**
  - If an instruction two stages ahead (MEM/WB stage) writes to a register needed by the current instruction, forward the data from memory or the earlier ALU result.
- **No Forwarding (ForwardA = 00 or ForwardB = 00)**
  - If there is no dependency, use the value directly from the register file.

OUTPUTS FROM PIPELINED PROCESSOR WITH FORWRDING HAD BEEN INCLUDED IN GIT HUB(*processor_withforward_terminal_output.txt*)

STALLING:



Stalling (Pipeline Stall) - Brief Explanation

A **stall** (or **pipeline stall**) happens when the processor has to **pause execution** because an instruction depends on data or control information that is **not yet available**. This causes a delay in instruction execution, reducing pipeline efficiency.

Types of Stalls

1. **Data Hazard Stall**

- Happens when an instruction **needs data** from a previous instruction that hasn't finished yet.
- Example:

```
add x3, x1, x2   # x3 = x1 + x2
sub x4, x3, x5   # Needs x3, but it's still being calculated
```

- Since `x3` is still in the pipeline, the CPU **pauses execution** until it's ready.
- **Control Hazard Stall**
  - Happens in **branch instructions** where the next instruction is **uncertain**.
  - Example:

```
beq x1, x2, LABEL   # If x1 == x2, branch to LABEL
add x5, x6, x7      # Should this execute or not?
```

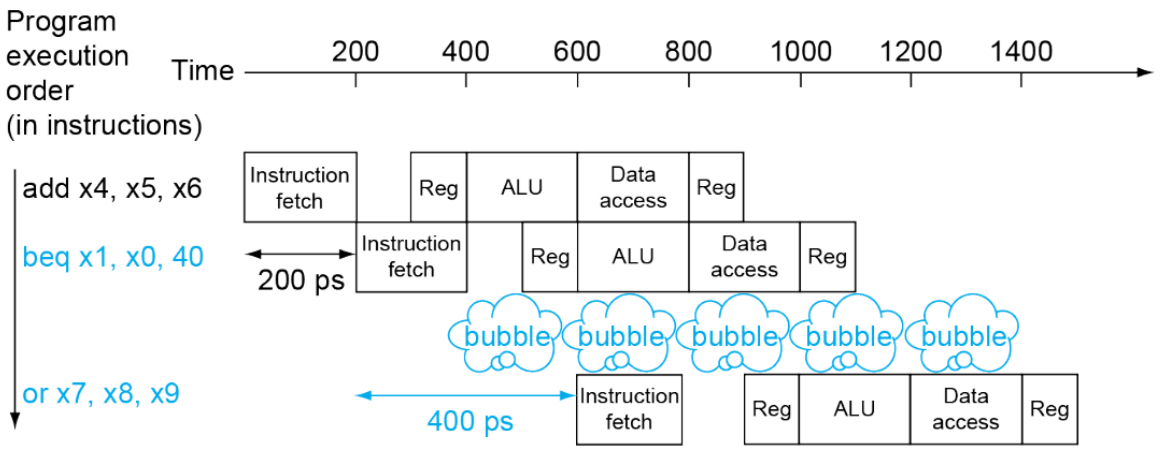
Structural Hazard Stall

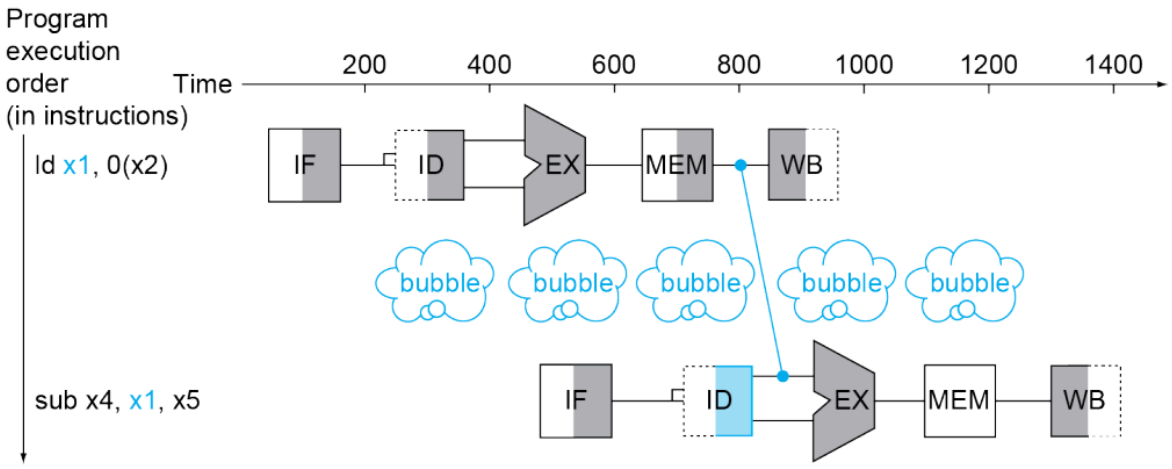
- Occurs when two instructions **compete for the same hardware resource** (e.g., memory or ALU).
- Example: If instruction fetch and data memory access need the same memory in a single-cycle processor, a stall is required

Solutions to Stalling

1. **Forwarding (Bypassing):** Instead of waiting for a register to be updated, the CPU **directly forwards** the computed value from an earlier pipeline stage.
2. **Pipeline Flushing:** In control hazards, if a wrong branch prediction occurs, instructions in the pipeline are **discarded** (flushed) to correct execution flow.
3. **Stall Insertion (Bubble):** The CPU **pauses execution** for a cycle or more, introducing a **bubble** in the pipeline.

4. **Branch Prediction:** The CPU guesses the outcome of a branch to **keep the pipeline busy** and avoid stalls.





CONTRIBUTIONS:

SAHITHI:

MEMORY IN BOTH

EXECUTE IN BOTH

TESTBENCH FOR SEQUENTIAL

EX/MEM

PIPELINING_WRAPPER

SEQUENTIAL TESTBENCH

SRI ABHINAYA:

REPORT

CONTROL IN BOTH SEQUENTIAL & PIPELINING

REGISTER DECODE IN BOTH

FORWARDING UNIT

MEM_WB BLOCK

WRITE BACK

KEERTHI:

FETCH IN BOTH

INSTR DECODER IN BOTH

IF/ID

ID/EX

SEQUENTIAL WRAPPER

PIPELINING PROCESSOR TESTBENCH

STALLING