IPA Project Report

Team : 19

- Gangineni Yaswanth
  - 2023102007
- Nethavath Praveen
  - 2023102013
- Kotha Phanindra Manoj Kumar
  - 2023102008

Sequential Implementation

Introduction

Processor architecture studies the Design and Implementation of Processors to execute instruc tions 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

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
-

- Control signals derived from instruction

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

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.

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 in struction 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.

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

ALU Control :

ALU Control	Operation
0000	AND
0001	OR
0010	Add
0110	Sub

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 .

Sequential Part

The sequential processor implementation is based on a single-cycle architecture where each in struction 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

1. **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.
2. **Instruction Memory:** Stores the program instructions. Uses the PC to output the corre sponding instruction during the fetch phase.
3. **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.
4. **Arithmetic Logic Unit (ALU):** Performs arithmetic (e.g., addition, subtraction) and log ical (e.g., AND, OR) operations. Also computes effective addresses for load and store opera

tions and generates a zero flag for branch decisions.

5. **Data Memory:** Used for load (ld) and store (sd) instructions. The effective address calcu lated by the ALU is used for accessing the memory.
6. **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 :

1. **Instruction Fetch:**

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

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.

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 ad dress 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).

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.

5. **Write-Back:**

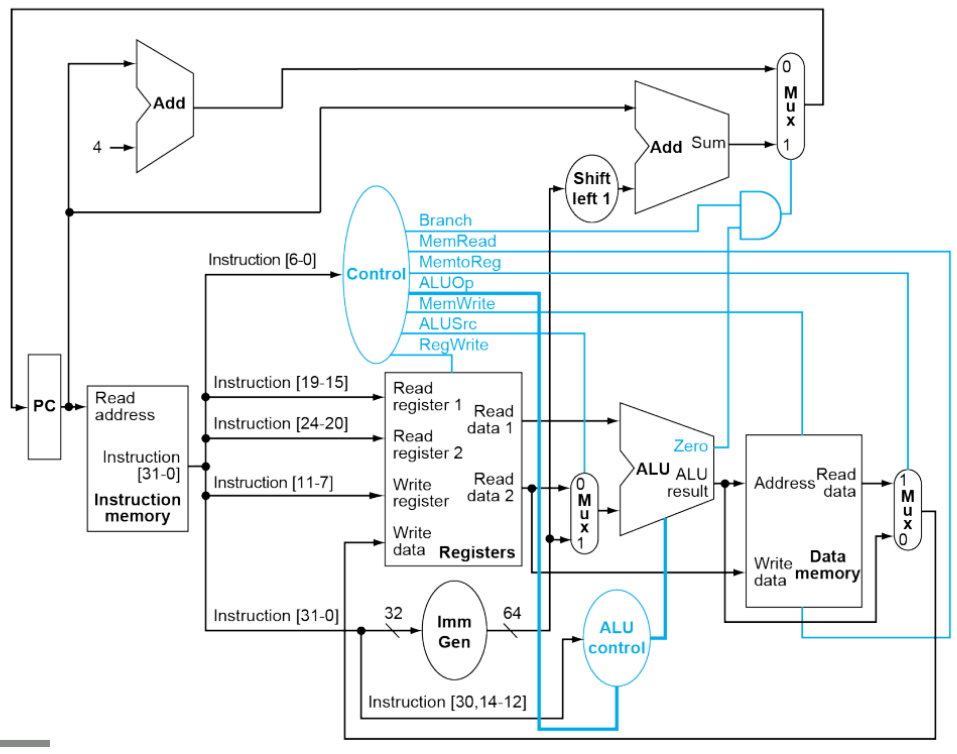
   The result from the ALU or data fetched from memory (in the case of ld) is written back to the destination register in the Register File.

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.



the above is the datapath for sequentional implementation

Explanation of each Module implemented in Verilog

Instruction Fetch (Stage 1)

1. Instruction_memory.v

Input: 64-bit pc value which represents the address of the memory

In the instruction memory module we created, the memory size is 1024 rows of 8-bit elements. Since each row represents 8 bits and an instruction is 32 bits, we need to fetch 4 elements from the array to retrieve a complete 32-bit instruction.

```
//this the main block block in the code to fetch the 32 bit instruction
always @(*) begin
      instruction = {memory[A3], memory[A2], memory[A1], memory[PC]};
  end
```

Instruction Decode(Stage-2)

1. Control Module:

It extracts the **opcode** from the instruction and determines the instruction type (R-type, I-type, S-type, B-type, etc.).

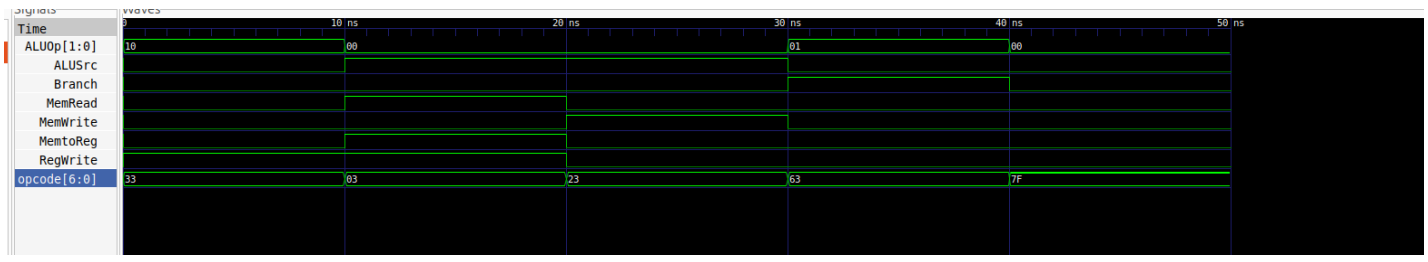
control.v

Inputs: Instruction (32 bits): The fetched instruction word whose opcode bits are used to generate control signals.

Outputs:

- **Register Write Enable (`RegWrite`)** – Determines whether the register file should be updated.
- **ALU Operation (`ALUOp`)** – Defines the operation the ALU should perform.
- **ALU Source (`ALUSrc`)** – Selects whether the second ALU operand comes from a register or immediate value.
- **Memory Read (`MemRead`)** – Indicates if data should be read from memory (for `ld` instructions).
- **Memory Write (`MemWrite`)** – Indicates if data should be written to memory (for `sd` instructions).
- **Memory to Register (`MemtoReg`)** – Selects whether the ALU result or memory data is written to a register.
- **Branch (`Branch`)** – Activates branching logic for conditional branches (`beq`).

gtkwave for sample testcases



register.v

Inputs:

- **Registers Declaration:**
  - `registers` is a 2D array that stores **32 registers** (`registers[31:0]`), each **64 bits wide**.
- **Register Initialization:**
  - `registers[0]` is always **0** (as per RISC-V convention).
  - Other registers have assigned values for testing.
- **Reading Registers:**
  - Two registers can be read simultaneously via **read_reg1** and **read_reg2**.
  - The values are **combinationally assigned** to `read_data1` and `read_data2`.
- **Writing to Registers:**
  - Writing occurs **on the positive clock edge**.
  - If `RegWrite` is **enabled** and the destination register is **not zero**, the value is updated.

ImmediateGenerator Module (immed.v)

The **Immediate Extractor Module** extracts and sign-extends the immediate values from **RISC-V instructions**. These immediate values are required for certain instruction types, such as **Load (LD), Store (SD), and Branch (BEQ)**.

-Functionality:

- The module takes a **32-bit instruction** as input.
- It extracts the **immediate value** based on the **opcode**.
- The extracted immediate is then **sign-extended to 64 bits** to match the processor's data width.

Inputs & Outputs:

Signal	Direction	Width	Description
`instruction`	Input	32 bits	The RISC-V instruction.
`imm64`	Output	64 bits	The extracted immediate value (sign-extended).

-Immediate Extraction Based on Instruction Type:

Instruction Type	Opcode	Immediate Extraction	Sign-Extension
Store (SD)	`7'b0100011`	`{instruction[31:25], instruction[11:7]}`	**From bit 31**
Load (LD)	`7'b0000011`	`{instruction[31:20]}`	**From bit 31**
Branch (BEQ)	`7'b1100011`	`{instruction[31:25], instruction[11:7]}`	**From bit 31**

- If the opcode does not match the supported types, the immediate value is set to **zero (64'd0)**.
- The extracted immediate values are used in **memory addressing (LD/SD)** and **branch offset calculations (BEQ)**.

Execution Stage(Stage-3):

-Alu Control

The **ALU Control module** determines the specific ALU operation based on the **ALUOp** signal and the **funct** field from the instruction. This module is a crucial part of the processor, as it ensures that the ALU performs the correct operation based on the instruction type.

Inputs and Outputs

Signal	Direction	Width	Description
`ALUOp`	Input	2 bits	Control signal from the main control unit that categorizes the instruction type.
`funct`	Input	4 bits	Function field from the instruction, which helps determine the exact ALU operation.
`ALUCtrl`	Output	4 bits	Control signal sent to the ALU to select the required operation.

1. The **ALUOp** signal is received from the **main control unit**.
2. The ALU Control module **decides the ALU operation** based on `ALUOp`:
   - If **ALUOp = 2'b00**, set `ALUCtrl = 4'b0010` (for load/store operations).
   - If **ALUOp = 2'b01**, set `ALUCtrl = 4'b0110` (for branch comparison).
   - If **ALUOp = 2'b10**, use `funct` to decide the exact operation:
     - `funct = 4'b1000` → Subtraction
     - `funct = 4'b0000` → Addition
     - `funct = 4'b0111` → AND
     - `funct = 4'b0110` → OR
3. The selected ALU operation is then sent as `ALUCtrl` to the **ALU module**.

It acts as a bridge between the **Control Unit** and the **ALU**.

-MUX:A **multiplexer (MUX)** is a combinational circuit that selects one of the multiple input signals and forwards it to the output based on a control signal (`sel`). In this case, the module is a **2-to-1 multiplexer** that selects between two **64-bit inputs (A and B)** based on the `sel` signal.

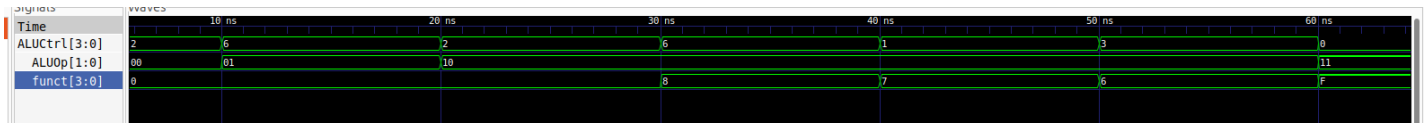
- If `sel = 0`, output `Y = A`
- If `sel = 1`, output `Y = B`
- The output `Y` is assigned using a **ternary operator (`?:`)**, which acts as a conditional selector.

Here we are using mux for selecting rs2 in some instruction types the rs2 will be selected from immediate and here the select line is AluSrc which selects second Alu operand comes from register.

-Execute(ALU Stage):

This ALU module performs arithmetic and logical operations on **64-bit** inputs based on a **4-bit control signal (`Alu_control`)**. It supports addition, subtraction, AND, OR operations, and outputs the result along with status flags (`zero` and `overflow`).

sample test cases



---

1. Inputs & Outputs

Signal	Direction	Width	Description
`a`	Input	64 bits	First operand
`b`	Input	64 bits	Second operand
`Alu_control`	Input	4 bits	Selects the ALU operation
`result`	Output	64 bits	Output of the ALU operation

`zero`	Output	1 bit	Flag set to 1 if `result` is zero
`overflow`	Output	1 bit	Flag set if overflow occurs in signed arithmetic

Functionalities

The ALU performs different operations based on the **Alu_control** input. Below are the supported operations:

A. Arithmetic Operations

1. **Addition (`4'b0010`)**
   - Adds `a` and `b` using `adder_64bit` module.
   - Overflow flag is set if there's an overflow.
   - Example:
     - `a = 5, b = 3` → `result = 8`
2. **Subtraction (`4'b0110`)**
   - Computes `a - b` using `Subtractor_64bit` module.
   - Sets `zero = 1` if the result is **0**.
   - Overflow flag is set if signed overflow occurs.
   - Example:
     - `a = 7, b = 7` → `result = 0, zero = 1`

B. Logical Operations

1. **Bitwise AND (`4'b0000`)**
   - Computes `result = a & b`.
   - Example:
     - `a = 1101, b = 1010` → `result = 1000`
2. **Bitwise OR (`4'b0001`)**
   - Computes `result = a | b`.
   - Example:
     - `a = 1101, b = 1010` → `result = 1111`

C. Default Operation

1. **Default Case (Unknown Operation)**
   - If an unsupported `Alu_control` value is provided, ALU sets `result = 0`, `overflow = 0`, and `zero = 0`.

3. Breakdown of Each Module in ALU

(A) `adder_64bit` Module

- **Performs 64-bit addition** using `FullAdder` instances.
- **Detects overflow** by comparing the carry-in and carry-out of the last bit.
- **Outputs:** `sum_result`, `sum_cout`, `sum_overflow`.

(B) `Subtractor_64bit` Module

- **Performs 64-bit subtraction** by adding `a` and `~b` (two's complement method).
- Uses **OverflowDetector** to check for overflow.
- **Outputs:** `diff_result`, `diff_cout`, `diff_overflow`.

(C) `And_64bit` Module

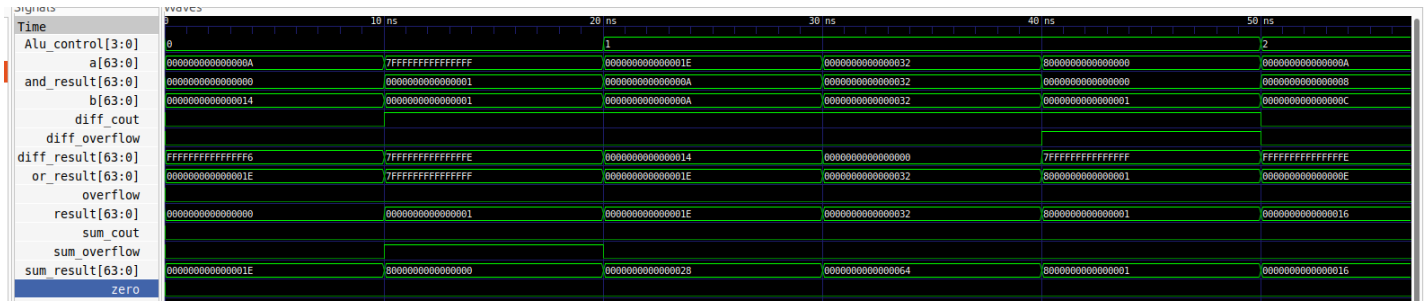
- **Performs bitwise AND** between `a` and `b`.

- Uses a **generate block** to apply AND operation to all 64 bits.

(D) `Or_64bit` Module

- **Performs bitwise OR** between `a` and `b`.
- Uses a **generate block** to apply OR operation to all 64 bits.

gtkwave for alu for sample testcases



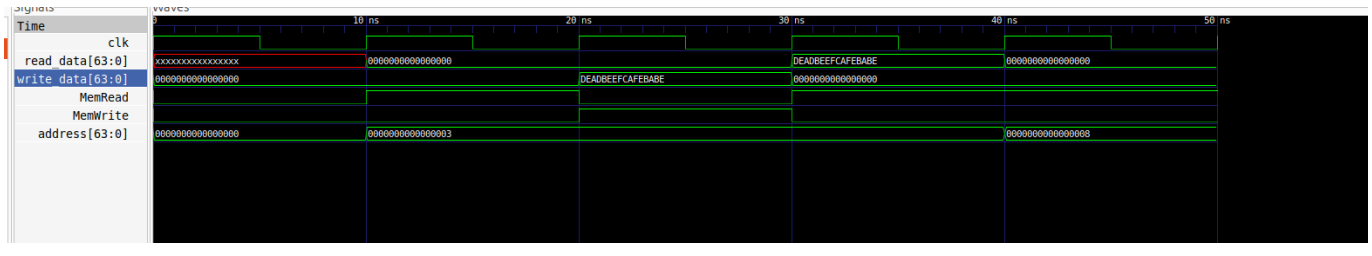
Data Memory .v(stage-4)

Data memory in a processor is used to store and retrieve data during program execution. It is a critical component of a processor's memory hierarchy and interacts with the control unit and arithmetic logic unit (ALU). In a RISC-V processor, data memory is used to implement **load (`ld`) and store (`sd`)** instructions, which allow the processor to read from and write to memory.

Functionality of the Data Memory Module

1. **Memory Storage**
   - The module defines a **256-entry memory array**, each holding a **64-bit word**. This represents a simple, small-scale memory for a processor.
2. **Inputs**
   - `MemRead`: A control signal that enables reading data from memory.
   - `MemWrite`: A control signal that enables writing data into memory.
   - `address [63:0]`: The memory address where the data is read from or written to. The lower 10 bits (`address[9:0]`) are used for indexing into the memory array (since 2^{10} = 1024, but only 256 locations are defined).
   - `write_data [63:0]`: The 64-bit data to be stored in memory when `MemWrite` is active.
3. **Outputs**
   - `read_data [63:0]`: The 64-bit data retrieved from memory when `MemRead` is active.
4. **Operations**
   - **Write Operation** (`MemWrite == 1`):
     - If `MemWrite` is asserted, the value in `write_data` is stored at the specified memory `address[9:0]`.
   - **Read Operation** (`MemRead == 1`):
     - If `MemRead` is asserted, the memory retrieves the value at `address[9:0]` and assigns it to `read_data`.
5. **Memory Initialization**
   - The `initial` block initializes memory locations to zero. In real hardware, memory is either preloaded with a program or initialized dynamically

gtk wave for sample test cases



Next PC

The **PC Next Module** (`new_pc`) is responsible for calculating the next program counter (`PC`) value in a **RISC-V processor**. It determines whether the processor should execute the next sequential instruction or jump to a branch target based on branch conditions.

Functionality of the `new_pc` Module

Inputs:

1. `Branch` (1-bit):
   - Indicates whether the instruction is a branch instruction.
2. `Zero` (1-bit):
   - Comes from the ALU, indicating whether the comparison resulted in zero (e.g., `beq` checks if two registers are equal).
3. `imm` (64-bit signed):
   - The immediate offset used for branch target computation. It represents how far to jump from the current `PC`.
4. `PC` (64-bit):
   - The current program counter value (address of the current instruction).

Outputs:

- `new_PC` (64-bit):
  - The next PC value, either the next sequential instruction (`PC + 4`) or a branch target address.
- **Calculate Sequential Next PC (`pc_plus_4`)**
  - Since RISC-V uses a **4-byte instruction width**, the next instruction is at `PC + 4`.

```
verilog
CopyEdit
assign pc_plus_4 = PC + 64'd4;
```

- **Compute Branch Target Address (`branch_target`)**
  - The branch target is calculated as `PC + (imm << 1)`.
  - RISC-V branch immediates are **signed values** in instruction encoding, and they are **word-aligned**, meaning they need to be left-shifted by 1 bit.

```
verilog
CopyEdit
assign branch_target = PC + (imm << 1);
```

- **Determine if a Branch is Taken (`branch_taken`)**
  - A branch is only taken if the instruction is a **branch** (`Branch` is 1) **AND** the condition (`Zero`) is met.

```
verilog
CopyEdit
assign branch_taken = Branch & Zero;
```

- **Select the Next PC (`new_PC`)**
  - If `branch_taken` is true, the new PC will be the branch target.
  - Otherwise, it moves to the next sequential instruction (`PC + 4`).

```
verilog
CopyEdit
always @(*) begin
    new_PC = branch_taken ? branch_target : pc_plus_4;
end
```

WB-(write Back Stage) (Stage-5)

Selecting the Write-Back Data (`final_write_data`)

```
wire signed [63:0] final_write_data;
mux write_data_mux(
    .A(ALU_result),
    .B(write_data1),
    .sel(MemtoReg),
    .Y(final_write_data)
);
```

- This **multiplexer (`mux`)** chooses between two sources for writing back to the register file:
  - `ALU_result` → The output of the ALU (for instructions like `add`, `sub`, `and`, `or`).
  - `write_data1` → Data from memory (for load instructions like `ld`).
  - `MemtoReg` (selector signal) decides:
    - `MemtoReg = 0`: Write ALU result.
    - `MemtoReg = 1`: Write memory load data.

2. Writing Data to Register File

```
verilog
CopyEdit
always @(posedge clk) begin
    if (RegWrite1 && current_instruction[11:7] != 5'b00000) begin
        register.registers[current_instruction[11:7]] <= final_write_data;
    end
end
```

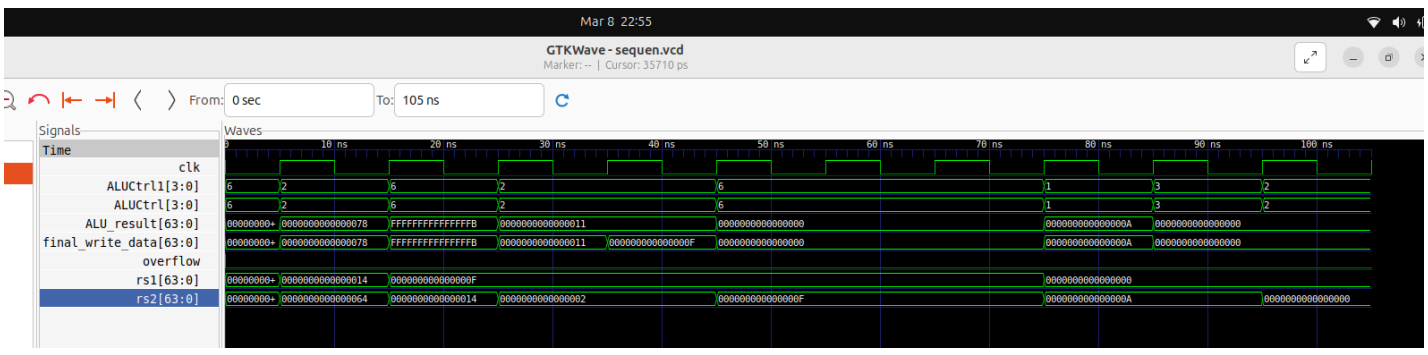
- `RegWrite1` **(control signal)**
  - When `RegWrite1` is **1**, it means we need to write data back to a register.
  - Used for **R-type (add, sub, and, or)** and **load (ld)** instructions.
  - Not used for `sd` (store) or `beq` (branch), as these don't write back.
- **Checking** `current_instruction[11:7] != 5'b00000`
  - Ensures that **register x0 is never written to** (RISC-V convention: `x0` is always 0).
- **Data is written to** `register.registers[current_instruction[11:7]]`
  - `current_instruction[11:7]` represents the **destination register (`rd`)** field.
  - The `final_write_data` (ALU result or memory data) is stored into this register.

txt:

```
        memory[0] = 8'h63;
        memory[1] = 8'h84;
        memory[2] = 8'h62;
```

```
        memory[3] = 8'h02;
        memory[4]=8'h7F;
        memory[5]=8'h00;
        memory[6]=8'h00;
        memory[7]=8'h00;
        memory[80] = 8'hB3;
        memory[81] = 8'h03;
        memory[82] = 8'h94;
        memory[83] = 8'h00;
        memory[84] = 8'h63;
        memory[85] = 8'h07;
        memory[86] = 8'h83;
        memory[87] = 8'hFE;
        memory[88] = 8'h23;
        memory[89] = 8'hB1;
        memory[90] = 8'h62;
        memory[91] = 8'h00;
        memory[92] = 8'h03;
        memory[93] = 8'hB4;
        memory[94] = 8'h22;
        memory[95] = 8'h00;
        memory[96] = 8'h63;
        memory[97] = 8'h0D;
        memory[98] = 8'h83;
        memory[99] = 8'hFE;
        memory[48] = 8'hB3;
        memory[49] = 8'h04;
        memory[50] = 8'h83;
        memory[51] = 8'h40;
        memory[52] = 8'h33;
        memory[53] = 8'hF5;
        memory[54] = 8'h44;
        memory[55] = 8'h00;
        memory[56] = 8'hB3;
        memory[57] = 8'hE1;
        memory[58] = 8'hA4;
        memory[59] = 8'h00;
        memory[60]=8'h7F;
        memory[61]=8'h00;
        memory[62]=8'h00;
        memory[63]=8'h00;

    registers[0]  = 64'h0000000000000000;
    registers[1]  = 64'h0000000000000001;
    registers[2]  = 64'd5;
    registers[3]  = 64'h0000000000000003;
    registers[4]  = 64'd10;
    registers[5]  = 64'd15;
    registers[6]  = 64'd15;
    registers[7]  = 64'h0000000000000000;
    registers[8]  = 64'd20;
    registers[9]  = 64'd100;
    registers[10] = 64'd0;
    registers[11] = 64'd0;
    registers[12] = 64'd16;
    registers[13] = 64'd19;
    registers[14] = 64'h0000000000000000;
    registers[15] = 64'h0000000000000000;
    registers[16] = 64'h0000000000000000;
    registers[17] = 64'h0000000000000000;
```

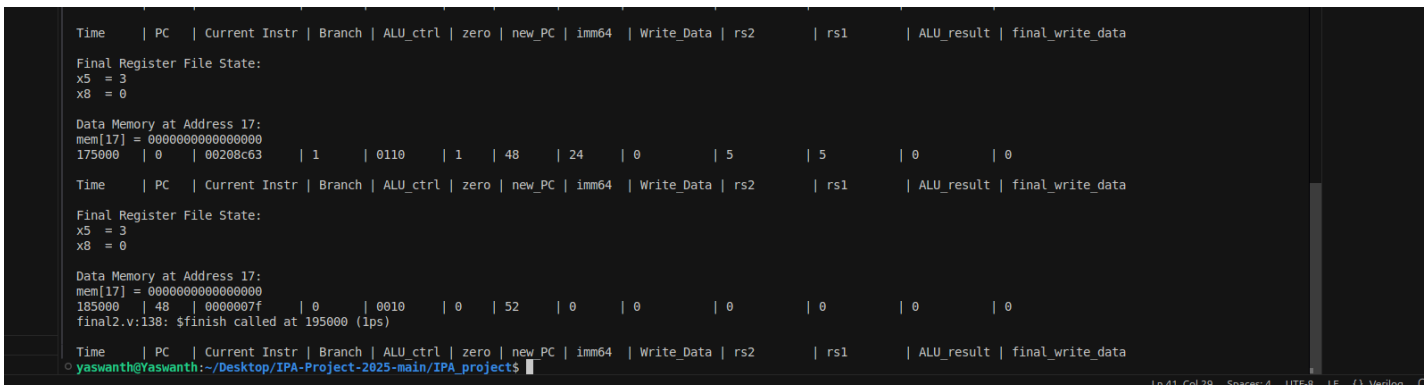
```
        registers[18] = 64'h0000000000000000;
        registers[19] = 64'h0000000000000000;
        registers[20] = 64'h0000000000000000;
        registers[21] = 64'h0000000000000000;
        registers[22] = 64'h0000000000000000;
        registers[23] = 64'h0000000000000000;
        registers[24] = 64'h0000000000000000;
        registers[25] = 64'h0000000000000000;
        registers[26] = 64'h000000000000001A;
        registers[27] = 64'h000000000000001B;
        registers[28] = 64'h000000000000001C;
        registers[29] = 64'h000000000000001D;
        registers[30] = 64'h000000000000001E;
        registers[31] = 64'h000000000000001F;

        //beq x6,x5,40 //00
        // exit // 4
        //sub x9 ×6 ×8 //48
        // or x10 ×4 ×9 //52
        // and x3 ×10 ×9 //56
        // exit //60
        //add x7 ×8 ×9 //80
        //beq x8 ×6 -18 //84
        //sd x6 2(x5) // 88
        // ld x8 2(x5)
        // beq x6 ×8 -4 //92
```

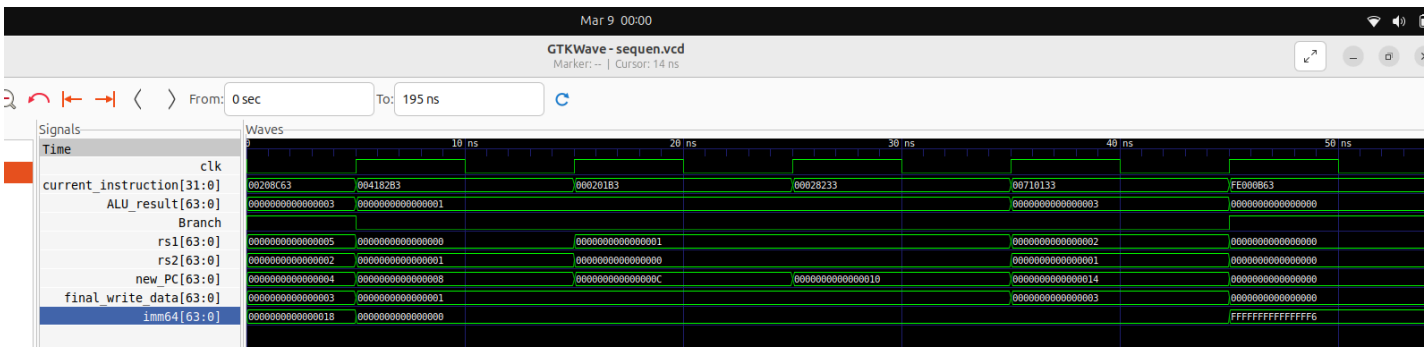


fibanocii txt

```
registers[0]  = 64'd0;
    registers[1]  = 64'd5;//n=5
     registers[2]  = 64'd2; //i=2
     registers[3]  = 64'd0;// fib(1)
     registers[4]  = 64'd1;//fib(2)
     registers[5]  = 64'd0;//result
   //fib(5)=0,1,1,2,3
   //so output x5=3
```

output:




Mar 9 00:00
GTKWave - sequen.vcd
Marker: -- | Cursor: 14 ns
From: 0 sec
To: 195 ns
Signals
Waves
Time
clk
current_instruction[31:0]
ALU_result[63:0]
Branch
rs1[63:0]
rs2[63:0]
new_PC[63:0]
final_write_data[63:0]
imm64[63:0]
10 ns
20 ns
30 ns
40 ns
50 ns
00208C63
004182B3
000201B3
00028233
00710133
FE000B63
0000000000000003
0000000000000001
0000000000000003
0000000000000000
0000000000000005
0000000000000000
0000000000000001
0000000000000002
0000000000000000
0000000000000002
0000000000000001
0000000000000000
0000000000000001
0000000000000000
0000000000000004
0000000000000008
000000000000000C
0000000000000010
0000000000000014
0000000000000000
0000000000000003
0000000000000001
0000000000000003
0000000000000000
0000000000000018
0000000000000000
FFFFFFFFFFFFFFF6

Pipelining Implementation

Team no ：19

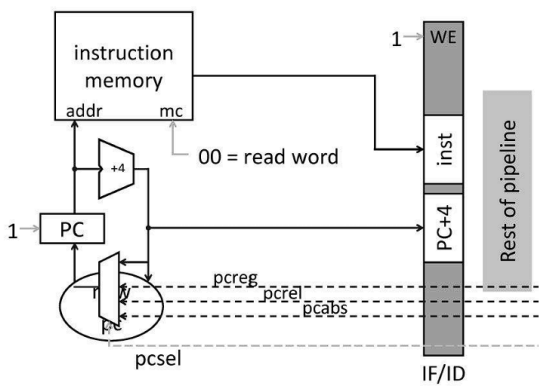
1. **Gangineni Yaswanth** - 2023102007
2. **Nethavath Praveen**-2023102013
3. **Kotha Phanindra Manoj Kumar** - 2023102008

Introduction

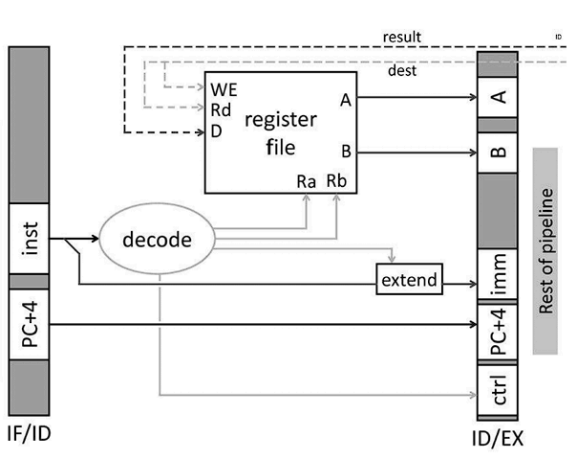
In the pipeline process implementation, the execution of instructions is broken down into several sequential stages, with each stage handling a different part of the instruction execution process. Instead of waiting for one instruction to complete all stages before starting the next, multiple instructions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

Stages in a Pipelined RISC-V Processor

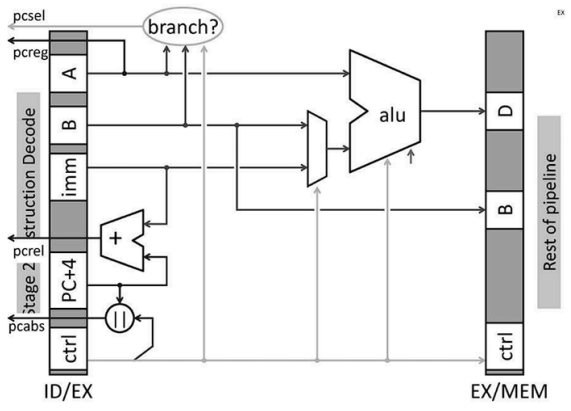
- **IF： Instruction Fetch from Memory**



- 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**

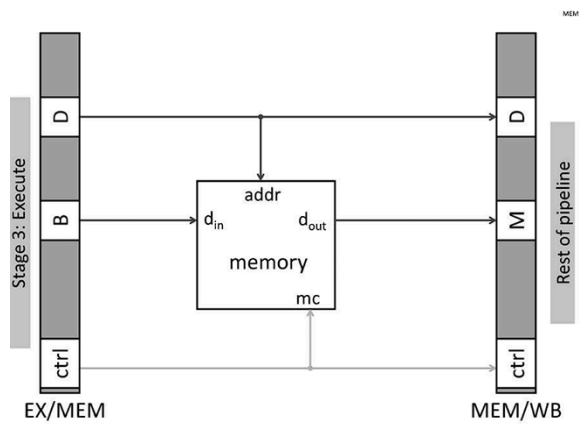


  - The instruction is decoded to determine its type and required operations.
  - Register values needed for execution are read from the register file.
  - Control signals are generated for subsequent stages.
- **EX: Execute Operation or Calculate Address**



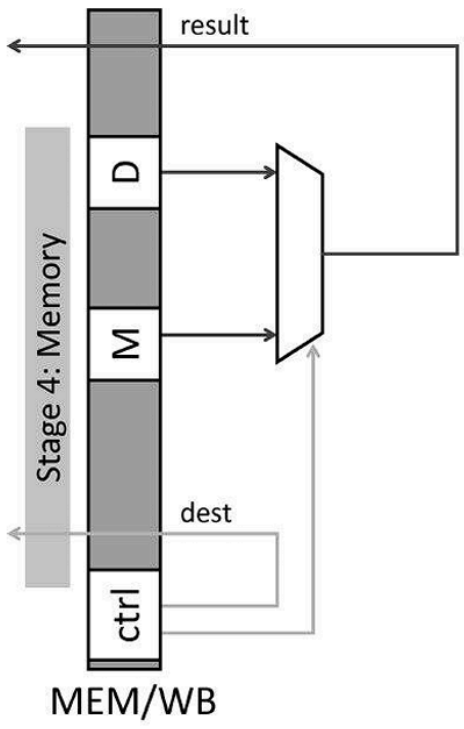
- For arithmetic/logic instructions: The ALU performs the specified operation.
- For memory instructions: The memory address is calculated.
- For branch instructions: The branch condition is evaluated and the target address is calculated.
- This stage implements the core computation of the instruction.

- **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 reg ister.
- This completes the instruction execution.

Pipeline Registers in a 5-Stage RISC-V Pipeline

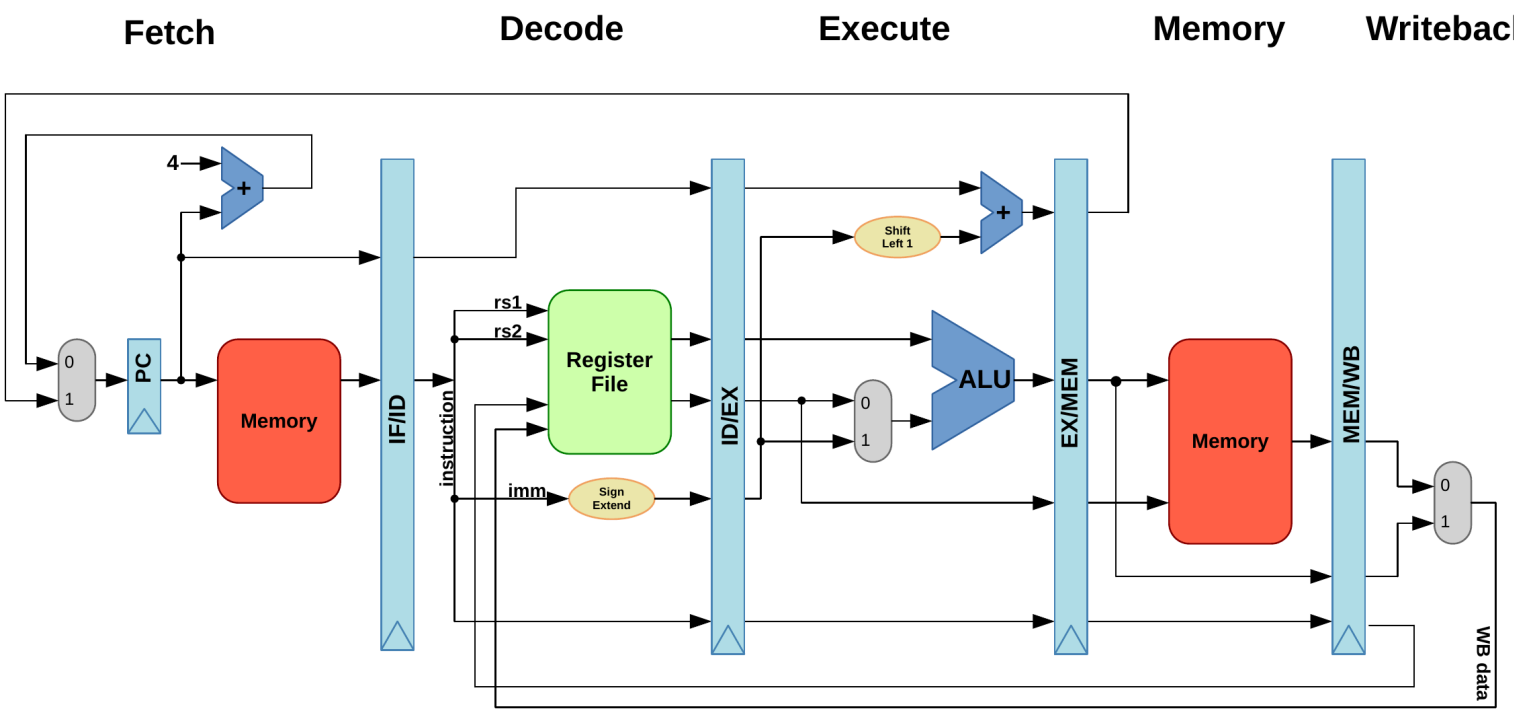
In a 5-stage RISC-V pipeline, there are four sets of pipeline registers:

- **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.

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 : A**fter 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

Cycle	Instruction 1	Instruction 2	Instruction 3	Instruction 4	Instruction 5
1	IF				
2	ID	IF			
3	EX	ID	IF		
4	MEM	EX	ID	IF	
5	WB	MEM	EX	ID	IF
6		WB	MEM	EX	ID
7			WB	MEM	EX
8				WB	MEM

Explanation of each new Module implemented in Verilog

Register between Stages

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.

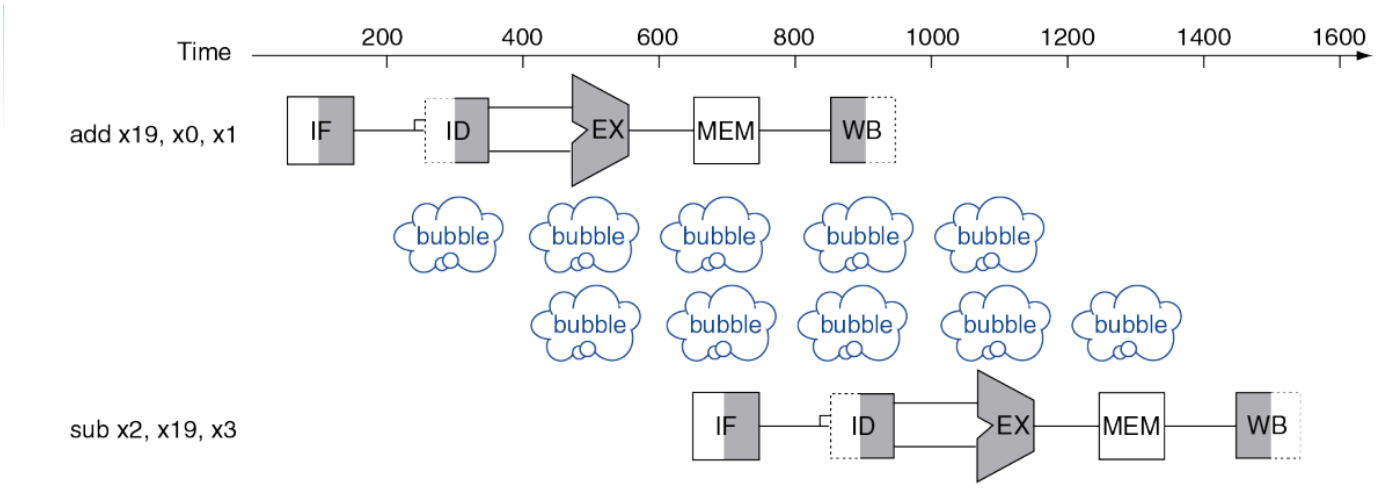
Outputs

- Used by Execute stage for ALU operations, branch calculations.
- Used by hazard detection unit (ID EX rd and ID EX MemRead)

Data Hazards

- **Cause:** An instruction depends on completion of data access by a previous instruction
- **Example:**

```
add  x19, x0, x1
sub  x2, x19, x3
```

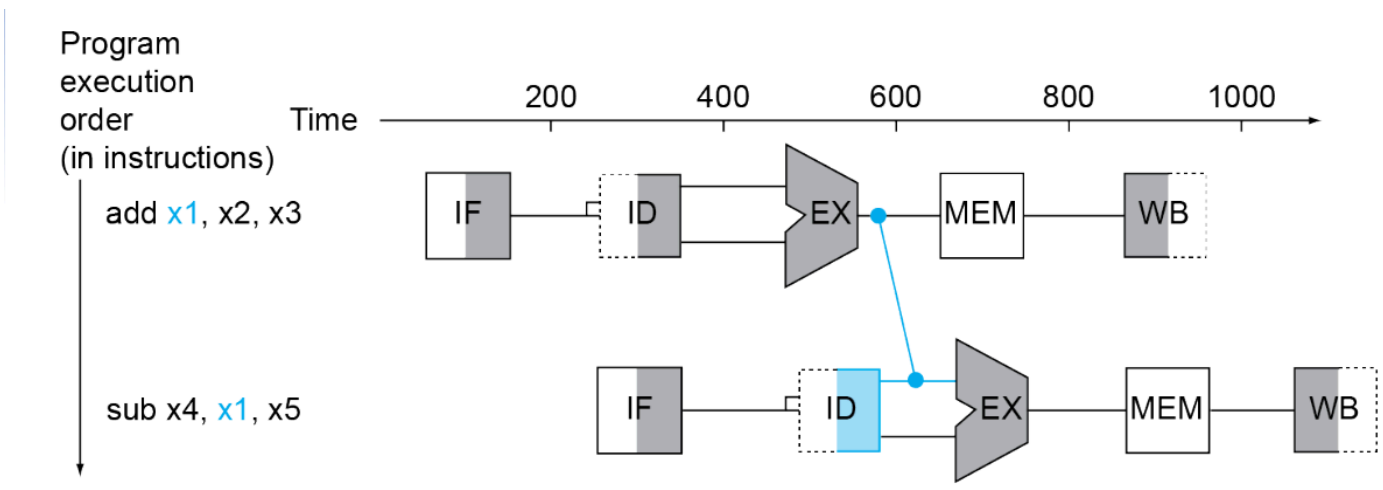


Solution:

Forwarding (Aka By Passing)

- Use result when it is computed
- Don’t wait for it to be stored in a register
- Requires extra connections in the data path

Stalling (Pipeline Stall)



- If forwarding is not possible, insert a **NOP (stall cycle)** to wait for the data.
- this case occurs when load use data hazard occurs.

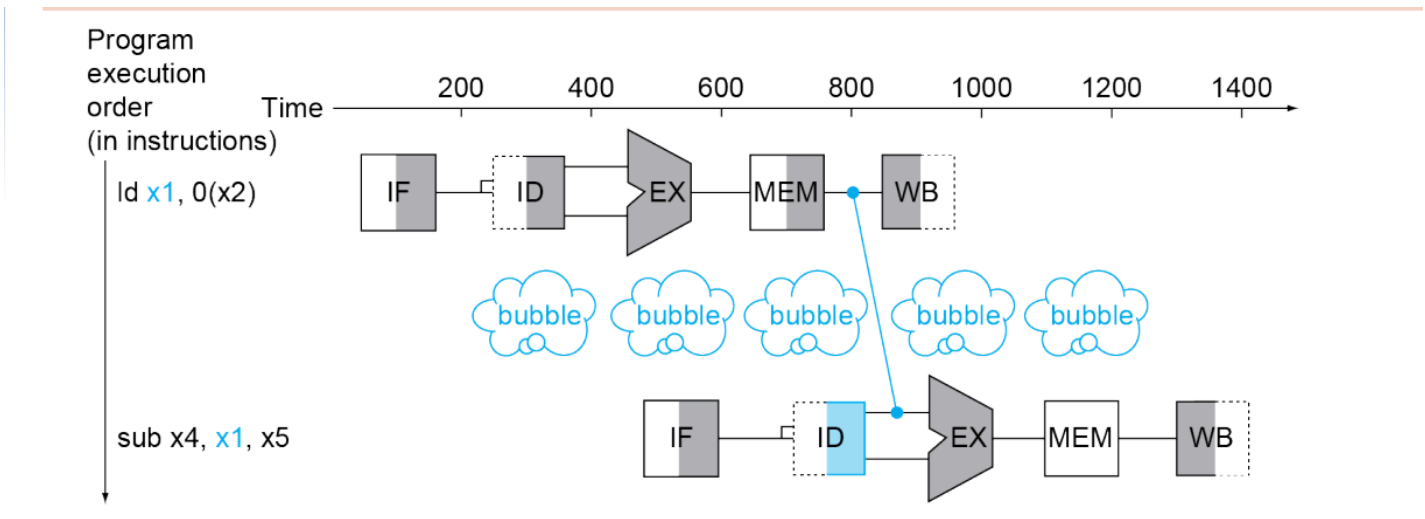
Load Use Data Hazard

occurs:

- Can’t always avoid stalls by forwarding
- If value not computed when needed
- Can’t forward backward in time!

Example:

```
ld x1,0(x2)
sub x4,x1,x5
```



- stall will occur we have to stop the the instruction for one cycle.

Control Hazards

- **Cause:** When the pipeline does not know **which instruction to fetch next**, usually caused by **branch instructions** (e.g., `beq`, `bne`, `jal`).
- **Example:**

```
BEQ x1, x2, LABEL  // Branch if x1 == x2
ADD x3, x4, x5     // Next instruction (may need to be flushed)
```

- If the **branch is taken**, the instruction already in the pipeline (`ADD x3, x4, x5`) **must be discarded**.
- This **wastes cycles** and reduces performance.

🔷 Solution:

Branch Prediction

- The processor predicts the **most likely path** and **speculatively executes instructions**.
- If the prediction is correct → No penalty.
- If the prediction is wrong → Flush the incorrect instructions and restart.

Forwarding Unit ：

```
module ForwardingUnit
(
    input wire [4:0] rs1,
    input wire [4:0] rs2,
    input wire [4:0] rd3,
    input wire [4:0] rd4,
    input wire WB3,
    input wire WB4,
    output reg [1:0] ForwardA,
    output reg [1:0] ForwardB
);

always @(*)
begin
    //No forwarding condn
    ForwardA = 2'b00;
    ForwardB = 2'b00;

    // Check if rd3 is the destination and should be forwarded
    if (WB3 && (rd3 != 0)) begin
        if (rd3 == rs1) ForwardA = 2'b10;
        if (rd3 == rs2) ForwardB = 2'b10;
```

```
    end

    // Check if rd4 is the destination and should be forwarded
    if (WB4 && (rd4 != 0)) begin
        if (rd4 == rs1) ForwardA = 2'b01;
        if (rd4 == rs2) ForwardB = 2'b01;
    end
end

endmodule
```

- **Inputs:**
  - `rs1`, `rs2`: Source registers 1 and 2 which are operands used in instructions.
  - `rd3`, `rd4`: Destination registers 3 and 4 which store results of executed instructions.
  - `WB3`, `WB4`: Signals indicating if there is a write-back (update) to registers rd3 and rd4.
- **Outputs:**
  - `ForwardA`, `ForwardB`: Control signals indicating if source registers rs1 and rs2 should be forwarded from rd3 or rd4.
- **Behavior:**
  - The `always @(*)` block continuously evaluates changes in inputs.
  - It initializes `ForwardA` and `ForwardB` to no forwarding (`2'b00`).
  - It checks if there's a write-back (`WB3`, `WB4`) and if the corresponding destination register (`rd3`, `rd4`) is not zero.
  - If true, it checks if the source registers (`rs1`, `rs2`) match the destination register (`rd3`, `rd4`) and updates `ForwardA` and `ForwardB` accordingly:
    - `2'b10` indicates forward from rd3,
    - `2'b01` indicates forward from rd4.

conflict :

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

Example :

ADD X3, X1, X2

ADD X5 ,X3,X4

Initially

X1=5, X2=10 ,X4=100

Output :

X5=115

Terminal Output :

```
--------------------------------------------------------------------------------
Time: 55000 | PC: 20 | Instruction: 00710133
--------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 5         | 0         | 4         | 0             | 0             |
--------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 1    | 0    | 1    | 0010        | 100             | 0               | 3     |
--------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 1    | 0    | 0                 | 115              | 5     | 1    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 1       | 1           | 0     | 0     | 24     | 0      | 0         | 12     |
| 0    |
--------------------------------------------------------------------------------
Registers[5]:
              0
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
Time: 65000 | PC: 24 | Instruction: 00000000
--------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 2         | 7         | 2         | 10            | 7             |
--------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 1    | 0    | 1    | 0010        | 115             | 0               | 4     |
--------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 1    | 0    | 0                 | 100              | 3     | 1    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 1       | 1           | 0     | 0     | 28     | 0      | 0         | 16     |
| 0    |
--------------------------------------------------------------------------------
Registers[5]:
            115
--------------------------------------------------------------------------------
```

Hazard Detection Unit

The **Hazard Detection Unit (HDU)** is a key component in a **pipelined processor** that ensures correct instruction execution by handling two types of pipeline hazards:

1. **Load-Use Data Hazard**
2. **Control Hazard (Branch Hazard)**

The **Hazard Detection Unit (HDU)** prevents incorrect execution in a pipelined RISC-V processor by:

1. **Detecting Load-Use Hazards** – Stalls the pipeline if a load instruction is followed by an instruction needing its result.
2. **Handling Control Hazards** – Flushes the pipeline when a branch is taken to avoid executing the wrong instructions.

```
module HazardDetectionUnit
(
    input wire ID_EX_MemRead,         // Load instruction in EX stage
    input wire [4:0] ID_EX_RegisterRd, // Destination register in EX stage
    input wire [4:0] IF_ID_RegisterRs1, // Source register 1 in ID stage
    input wire [4:0] IF_ID_RegisterRs2, // Source register 2 in ID stage
    input wire BranchTaken,           // Combined signal: Branch & ALUZero
```

```
    output reg Stall,       // Stall signal for load-use hazard
    output reg PCWrite,     // Control signal to stop PC update
    output reg IF_ID_Write, // Control signal to stop instruction fetch
    output reg Flush        // Control signal to flush incorrect instruction
);

always @(*) begin
    // Default values (no stall or flush)
    Stall = 0;
    PCWrite = 1;
    IF_ID_Write = 1;
    Flush = 0;

    // Handle Load-Use Hazard
    if (ID_EX_MemRead && (ID_EX_RegisterRd == IF_ID_RegisterRs1 || ID_EX_RegisterRd == IF_ID_RegisterRs2)) be
        Stall = 1;
        PCWrite = 0;
        IF_ID_Write = 0;
    end


    // Handle Control Hazard (Branch)
    if (BranchTaken) begin // Branch Taken
        Flush = 1;   // Flush IF/ID pipeline register
        PCWrite = 1; // Allow PC to update to branch target
        IF_ID_Write = 1; // Continue fetching
    end
end

endmodule
```

- **Inputs:**
  - `ID_EX_MemRead`: Indicates if the current instruction in the EX stage is a **load** instruction (e.g., `ld`).
  - `ID_EX_RegisterRd`: Destination register of the EX stage instruction.
  - `IF_ID_RegisterRs1`, `IF_ID_RegisterRs2`: Source registers of the instruction currently in the ID stage.
  - `BranchTaken`: Signal that becomes high when a branch is taken (based on branch condition & ALU result).
- **Outputs:**
  - `Stall`: Stops instruction execution to prevent a load-use hazard.
  - `PCWrite`: Controls whether the program counter (PC) should update.
  - `IF_ID_Write`: Controls whether the instruction fetch (IF/ID pipeline register) should update.
  - `Flush`: Clears the IF/ID pipeline register to remove incorrect instructions after a branch.

*By default, the pipeline continues execution with **no stall or flush**.*

A
Load-Use hazard occurs when an instruction needs data from a **load (`ld`) instruction** that is still in the EX stage.

A
control hazard occurs when a **branch (`beq`, `bne`) is taken**, but the **next instruction was already fetched** incorrectly.

Hazard Type	Detection Condition	Solution	Effect on Pipeline
Load-Use Hazard	ID_EX_MemRead+ID_EX_RegisterRd== `IF_ID_RegisterRs1`, `IF_ID_RegisterRs2`	Stall (`Stall = 1`)	1-cycle stall

Branch Hazard	`BranchTaken` ==1	Flush (`Flush = 1`)	Removes incorrect instruction

- If the instruction in the **EX stage** is a **load instruction (`MemRead` is high)** and its destination register (`ID_EX_RegisterRd`) matches either of the **source registers** (`IF_ID_RegisterRs1` or `IF_ID_RegisterRs2`) of the instruction in the **ID stage**, a **stall** is required to avoid incorrect data being used before it is available.
- **Actions Taken:**
  - `Stall = 1` : Stop execution of the current instruction.
  - `PCWrite = 0` : Halt program counter updates.
  - `IF_ID_Write = 0` : Stop fetching new instructions.
- If a **branch is taken (`BranchTaken` is high)**, it means the previous instruction decided to change the program counter (PC).
- The next instruction fetched is **incorrect**, so we must **flush it** from the pipeline.
- **Actions Taken:**
  - `Flush = 1` : Remove incorrect instruction from IF/ID pipeline register.
  - `PCWrite = 1` : Allow PC to update to branch target.
  - `IF_ID_Write = 1` : Allow fetching new instructions after branch

Load Hazard :

example testcase :

LD X8 , 10(X7)

ADD X9,X8,X10

initially X7=7, X10 =10

Data_memory[17]=16

Output : X9 = 10+16 =26

OUT PUT :

```
--------------------------------------------------------------------------------
Time: 65000 | PC: 20 | Instruction: 00710133
--------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 5         | 0         | 4         | 0             | 0             |
--------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 1    | 0    | 1    | 0010        | 1               | 0               | 3     |
--------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 1    | 0    | 0                 | 26               | 9     | 1    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 1       | 1           | 0     | 0     | 24     | 0      | 0         | 12     |
| 0    |
--------------------------------------------------------------------------------
Registers[9]:
              9
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
Time: 75000 | PC: 24 | Instruction: 00000000
--------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 2         | 7         | 2         | 2             | 7             |
--------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 1    | 0    | 1    | 0010        | 0               | 0               | 4     |
--------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 1    | 0    | 0                 | 1                | 3     | 1    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 1       | 1           | 0     | 0     | 28     | 0      | 0         | 16     |
| 0    |
--------------------------------------------------------------------------------
Registers[9]:
              26
--------------------------------------------------------------------------------
```

PC is Constant for one clock Cycle which indicates a stall :

```
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
Time: 25000 | PC: 8 | Instruction: 00a404b3
--------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 7         | 10        | 8         | 7             | 10            |
--------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 1    | 1    | 1    | 0010        | 0               | 0               | 0     |
--------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 0    | 0    | 0                 | 0                | 0     | 0    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 0       | 0           | 1     | 0     | 12     | 20     | 10        | 0      |
| 0    |
--------------------------------------------------------------------------------
Registers[9]:
              9
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
Time: 35000 | PC: 8 | Instruction: 00a404b3
--------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 7         | 10        | 8         | 7             | 10            |
--------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 1    | 1    | 1    | 0010        | 17              | 0               | 8     |
--------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 1    | 1    | 0                 | 0                | 0     | 0    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 1       | 1           | 0     | 0     | 12     | 20     | 10        | 0      |
| 0    |
--------------------------------------------------------------------------------
Registers[9]:
              9
--------------------------------------------------------------------------------
```

Control Hazard :

```
beq X3 , X5 , 30
ADD X3 , X2 , X1
ADD X5 , X3 , X4
EXIT:
AT 60th adress
SUB X10 ,X9 , X10
EXIT :
```

All the Reg Values are set 0 because Flush i s 1

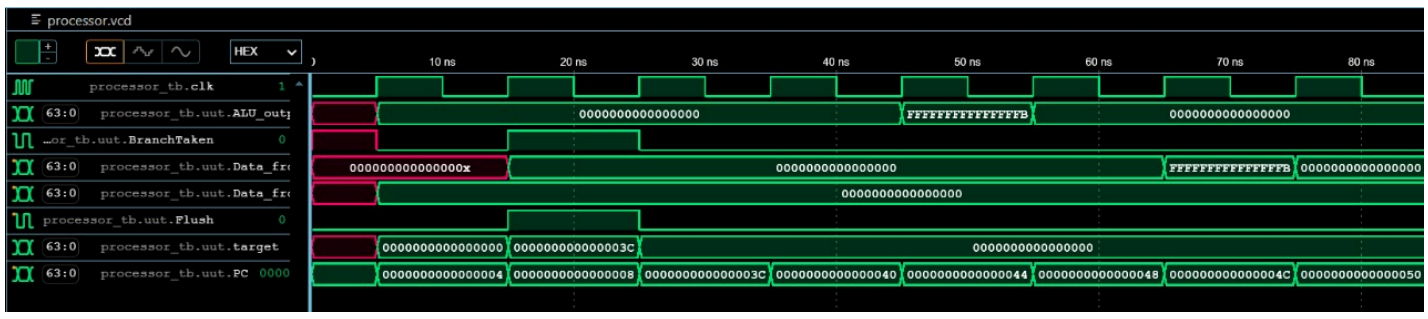
```
-------------------------------------------------------------------------------
Time: 25000 | PC: 8 | Instruction: 002081b3
-------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 3         | 5         | 30        | 0             | 0             |
-------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 0    | 0    | 1    | 0110        | 0               | 0               | 0     |
-------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 0    | 0    | 0                 | 0                | 0     | 0    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 1       | 1           | 0     | 1     | 60     | 60     | 30        | 0      |
| 0    |
-------------------------------------------------------------------------------
Registers[10]:
             10
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
Time: 35000 | PC: 60 | Instruction: 00000000
-------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 0         | 0         | 0         | 0             | 0             |
-------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 0    | 0    | 0    | 0000        | 0               | 0               | 30    |
-------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 0    | 0    | 0                 | 0                | 0     | 0    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 1       | 1           | 0     | 0     | 64     | 0      | 0         | 0      |
| 0    |
-------------------------------------------------------------------------------
Registers[10]:
             10
-------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
Time: 45000 | PC: 64 | Instruction: 40a48533
--------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 0         | 0         | 0         | 0             | 0             |
--------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 0    | 0    | 1    | 0010        | 0               | 0               | 0     |
--------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 0    | 0    | 0                 | 0                | 30    | 0    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 1       | 1           | 0     | 0     | 68     | 0      | 0         | 0   |
| 0    |
--------------------------------------------------------------------------------
Registers[10]:
             10
--------------------------------------------------------------------------------
```

OUTPUT :

```
--------------------------------------------------------------------------------
Time: 75000 | PC: 76 | Instruction: 00000000
--------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 0         | 0         | 0         | 0             | 0             |
--------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 0    | 0    | 1    | 0010        | 0               | 0               | 0     |
--------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 0    | 0    | 0                 | -5               | 10    | 1    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 1       | 1           | 0     | 0     | 80     | 0      | 0         | 68  |
| 0    |
--------------------------------------------------------------------------------
Registers[10]:
             10
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
Time: 85000 | PC: 80 | Instruction: 0000007f
--------------------------------------------------------------------------------
| IFID_rs1  | IFID_rs2  | IFID_rd   | rs1           | rs2           |
| 0         | 0         | 0         | 0             | 0             |
--------------------------------------------------------------------------------
| WB2  | M2   | EX2  | ALU_Control | ALU_result      | Data_to_write   | rd3   |
| 0    | 0    | 1    | 0010        | 0               | 0               | 0     |
--------------------------------------------------------------------------------
| WB3  | M3   | Data_from_memory  | Data_from_ALU    | rd4   | WB4  |
| 0    | 0    | 0                 | 0                | 0     | 0    |
| PCWrite | IF_ID_Write | Stall | Flush | new_pc | target | immed64_2 | PC3 |
| 1       | 1           | 0     | 0     | 84     | 0      | 0         | 72  |
| 0    |
--------------------------------------------------------------------------------
Registers[10]:
             -5
--------------------------------------------------------------------------------
```

GTKWAVE :



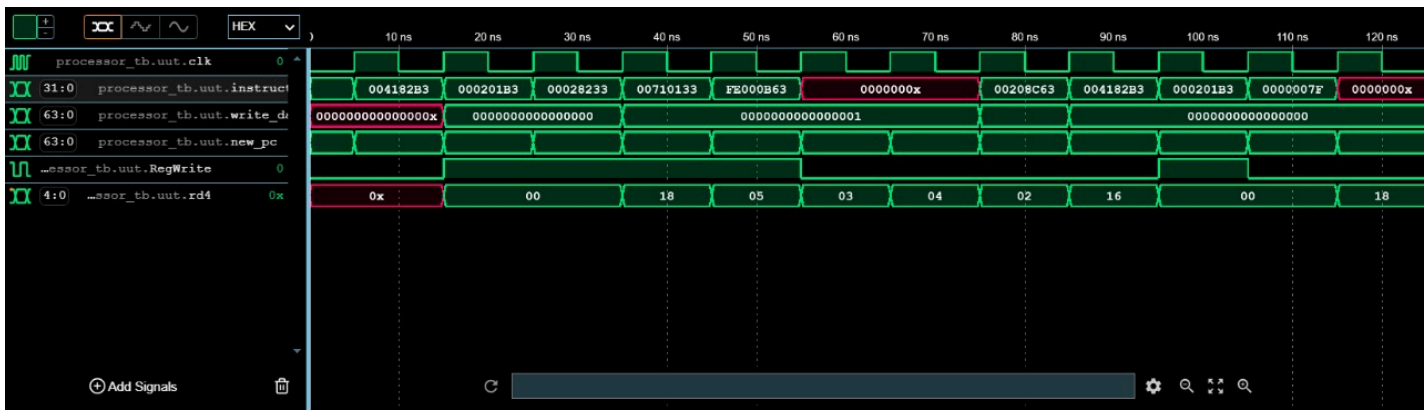
Overall Circuit :

example :

Fibonacci

input Registers :

```
registers[0]  = 64'd0;
registers[1]  = 64'd5;//n=5
registers[2]  = 64'd2; //i=2
registers[3]  = 64'd0;// fib(1)
registers[4]  = 64'd1;//fib(2)
registers[5]  = 64'd0;//result
registers[6]  = 64'd0;
registers[7]  = 64'd1;
registers[8]  = 64'd0;
registers[9]  = 64'd9;
registers[10] = 64'd10;
registers[11] = 64'd0;
registers[12] = 64'd0;
registers[13] = 64'd0;
registers[14] = 64'd0;
registers[15] = 64'd0;
registers[16] = 64'd0;
registers[17] = 64'd0;
registers[18] = 64'd0;
registers[19] = 64'd0;
registers[20] = 64'd0;
registers[21] = 64'd0;
registers[22] = 64'd0;
registers[23] = 64'd0;
registers[24] = 64'd0;
registers[25] = 64'd0;
registers[26] = 64'd0;
registers[27] = 64'd0;
registers[28] = 64'd0;
registers[29] = 64'd0;
registers[30] = 64'd0;
registers[31] = 64'd0;
```



OUTPUT :

The out 3 is stored in X5 .

```
registers[0]  = 64'd0;
registers[1]  = 64'd5;//n=5
registers[2]  = 64'd5; //i=5
```

```
    registers[3]  = 64'd1;// fib(1)
    registers[4]  = 64'd2;//fib(2)
    registers[5]  = 64'd3;//result
    registers[6]  = 64'd0;
    registers[7]  = 64'd1;
    registers[8]  = 64'd0;
    registers[9]  = 64'd9;
    registers[10] = 64'd10;
    registers[11] = 64'd0;
    registers[12] = 64'd0;
    registers[13] = 64'd0;
    registers[14] = 64'd0;
    registers[15] = 64'd0;
    registers[16] = 64'd0;
    registers[17] = 64'd0;
    registers[18] = 64'd0;
    registers[19] = 64'd0;
    registers[20] = 64'd0;
    registers[21] = 64'd0;
    registers[22] = 64'd0;
    registers[23] = 64'd0;
    registers[24] = 64'd0;
    registers[25] = 64'd0;
    registers[26] = 64'd0;
    registers[27] = 64'd0;
    registers[28] = 64'd0;
    registers[29] = 64'd0;
    registers[30] = 64'd0;
    registers[31] = 64'd0;
```

contributions :

- Gangineni Yaswanth
  - Sequential
    - Reg Block
    - MUX 2:1 , New PC
    - Wrapper
  - Pipeline
    - Load Use Hazard
    - Pipeline Wrapper
- Nethavath Praveen
  - Sequential
    - Alu , Alu Ctrl Block
    - Data Memory
    - Wrapper
  - Pipeline
    - Forwarding Unit

        - Pipeline Wrapper
- Kotha Phanindra Manoj Kumar
    - Sequential
        - Instruction Fetch
        - control Unit , Immediate
        - Wrapper
    - Pipeline
        - Branch Hazard
        - Pipeline Wrapper

Equal Contribution Towards Project Report , Testcases and Debuging .