INTERNSHIP TASK -3

PIPELINE PROCESSOR DESIGN

- DESIGN A 4-STAGE PIPELINED PROCESSOR WITH BASIC INSTRUCTIONS LIKE ADD, SUB, AND LOAD

What is a 4-Stage Pipelined Processor?

A pipelined processor is one where multiple instructions are processed at different stages simultaneously, just like an assembly line. The 4 stages in our processor are:

1. **IF (Instruction Fetch)** – Fetch the instruction.
2. **ID (Instruction Decode)** – Decode the instruction and read registers.
3. **EX (Execute)** – Perform the operation (add, subtract, load).
4. **WB (Writeback)** – Write the result back to the register.

Basic Instructions

- **ADD R1, R2, R3**: Add values of `R2` and `R3`, store result in `R1`.
- **SUB R1, R2, R3**: Subtract `R3` from `R2`, store result in `R1`.
- **LOAD R1, 100(R2)**: Load value from memory address `R2 + 100` into `R1`.

Simple Processor Design

1. Registers

We need 32 registers (`R0` to `R31`) to store values.

2. Memory

A small memory space is needed to load data for the **LOAD** instruction.

3. Processor Stages

Here's a simple breakdown of each stage:

- **IF (Instruction Fetch)**:

    - Fetch an instruction from memory based on the **Program Counter (PC)**.
    - Increment PC for the next instruction.
- **ID (Instruction Decode)**:
    - Decode the instruction to understand what it needs to do (ADD, SUB, LOAD).
    - Fetch the necessary data from registers.
- **EX (Execute)**:
    - Perform the operation: addition, subtraction, or calculate the address for loading data.
- **WB (Writeback)**:
    - Write the result of the operation back to the register file.

Verilog Code Example for the Processor

1. Processor Module

```
module processor (
    input clk,       // Clock signal
    input reset      // Reset signal
);

    // Register file (32 registers, each 32 bits wide)
    reg [31:0] reg_file [31:0];

    // Program Counter
    reg [31:0] pc;

    // Pipeline registers (for each stage)
    reg [31:0] IF_ID_instr;  // Instruction fetched
    reg [31:0] ID_EX_instr, ID_EX_rs1, ID_EX_rs2;  // Decoded instruction and operands
    reg [31:0] EX_MEM_alu_result;  // Result from execution stage
```

```
reg [31:0] MEM_WB_result;  // Result for writeback

// Initialize Program Counter
initial begin
    pc = 32'b0;  // Start from instruction 0
end

// Instruction Fetch (IF) Stage
always @(posedge clk or posedge reset) begin
    if (reset) begin
        pc <= 32'b0;  // Reset PC to 0
    end else begin
        IF_ID_instr <= memory[pc];  // Fetch instruction from memory
        pc <= pc + 4;  // Move to next instruction
    end
end

// Instruction Decode (ID) Stage
always @(posedge clk or posedge reset) begin
    if (reset) begin
        ID_EX_instr <= 32'b0;
        ID_EX_rs1 <= 32'b0;
        ID_EX_rs2 <= 32'b0;
    end else begin
        ID_EX_instr <= IF_ID_instr;  // Pass instruction to next stage
        ID_EX_rs1 <= reg_file[IF_ID_instr[25:21]];  // Read register 1
        ID_EX_rs2 <= reg_file[IF_ID_instr[20:16]];  // Read register 2
    end
end
```

```
// Execute (EX) Stage
always @(posedge clk or posedge reset) begin
    if (reset) begin
        EX_MEM_alu_result <= 32'b0;
    end else begin
        case (ID_EX_instr[31:26])  // Check the opcode
            6'b000000: begin  // ADD/SUB
                if (ID_EX_instr[5:0] == 6'b100000)  // ADD
                    EX_MEM_alu_result <= ID_EX_rs1 + ID_EX_rs2;
                else if (ID_EX_instr[5:0] == 6'b100010)  // SUB
                    EX_MEM_alu_result <= ID_EX_rs1 - ID_EX_rs2;
            end
            6'b100011: begin  // LOAD
                EX_MEM_alu_result <= ID_EX_rs1 + ID_EX_instr[15:0];  // Address calculation
            end
        endcase
    end
end

// Memory (MEM) Stage
always @(posedge clk or posedge reset) begin
    if (reset) begin
        MEM_WB_result <= 32'b0;
    end else begin
        case (EX_MEM_alu_result[31:26])
            6'b100011: begin  // LOAD
                MEM_WB_result <= memory[EX_MEM_alu_result];  // Load from memory
            end
```

```
            default: begin  // ADD/SUB
                MEM_WB_result <= EX_MEM_alu_result;  // ALU result
            end
        endcase
    end
end

// Writeback (WB) Stage
always @(posedge clk or posedge reset) begin
    if (reset) begin
        reg_file[MEM_WB_instr[25:21]] <= 32'b0;  // Reset register file
    end else begin
        case (MEM_WB_instr[31:26])
            6'b000000: begin  // ADD/SUB
                reg_file[MEM_WB_instr[25:21]] <= MEM_WB_result;  // Write result back
            end
            6'b100011: begin  // LOAD
                reg_file[MEM_WB_instr[20:16]] <= MEM_WB_result;  // Write loaded value
            end
        endcase
    end
end

// Memory (for simplicity, we define a small memory here)
reg [31:0] memory [0:255];  // 256 words of memory

// Initializing some values in memory for testing
initial begin
    memory[0] = 32'b000000_00001_00010_00011_00000_100000;  // ADD R1, R2, R3
```

```
    memory[4] = 32'b000000_00001_00010_00100_00000_100010;  // SUB R1, R2, R4
    memory[8] = 32'b100011_00001_00010_0000000000000100;    // LOAD R1, 100(R2)
  end

endmodule
```

Explanation

- Register File:

  Contains 32 registers (R0 to R31), each 32 bits wide.

- Program Counter (PC):

  Points to the current instruction in memory.

- Pipeline Registers:

  IF_ID_instr: Stores the instruction fetched in the IF stage.

ID_EX_instr, ID_EX_rs1, ID_EX_rs2: Stores the instruction and operand values during the ID stage.

EX_MEM_alu_result: Stores the result of the EX stage (like ALU results).

MEM_WB_result: Stores the result of the MEM stage (like data read from memory).

- Instruction Execution:

The processor fetches, decodes, executes, and writes back the result of instructions in a 4-stage pipeline.

- Memory:

Simple memory with a few instructions for testing (ADD, SUB, LOAD).

To Test the Processor:

Initialize the PC and memory with some instructions (e.g., ADD, SUB, LOAD).

Simulate the processor in a Verilog simulator (like ModelSim or Vivado).

Observe the registers and memory after simulation to verify if the instructions were executed correctly.