TASK-3

Designing a 4-stage pipelined processor that can execute basic instructions like ADD, SUB, and LOAD involves understanding both the hardware components and the pipeline stages needed to process these instructions efficiently. A pipeline allows multiple instructions to be in different stages of execution simultaneously, increasing throughput by overlapping instruction cycles.

**1. Pipelining Overview**

In a pipeline, the execution of an instruction is split into several stages, where each stage performs a part of the overall operation. A 4-stage pipeline has four distinct stages, which allow the processor to process four instructions at a time (each in a different stage).

**Pipeline Stages:**

* **Stage 1: IF (Instruction Fetch)**: Fetch the instruction from memory.
* **Stage 2: ID (Instruction Decode)**: Decode the instruction and read registers.
* **Stage 3: EX (Execute)**: Execute the operation (arithmetic or address calculation).
* **Stage 4: MEM (Memory Access/Write-back)**: Access memory (for load/store operations) or write results back to the register file (for arithmetic operations).

Now, let’s explain how this works for basic instructions like **ADD**, **SUB**, and **LOAD**.

**2. Detailed Design of the 4-Stage Pipeline**

**Stage 1: Instruction Fetch (IF)**

* **Operation**: The instruction is fetched from memory using the Program Counter (PC). The PC points to the memory location where the instruction is stored.
* **Components**:
  + **Instruction Memory**: Stores the program instructions.
  + **Program Counter (PC)**: Holds the address of the next instruction to be fetched.
  + **PC Incrementer**: Increments the PC to point to the next instruction.

**Example**: If the current PC points to address 0x1000, the processor fetches the instruction at that memory location.

**Stage 2: Instruction Decode (ID)**

* **Operation**: The instruction is decoded to determine the operation type (ADD, SUB, LOAD, etc.). The source registers are read, and control signals are generated.
* **Components**:
  + **Control Unit**: Decodes the opcode and generates the control signals for the next stages (e.g., register file read/write, ALU operation).
  + **Register File**: Holds the values for the registers. The register file is read during this stage for the operands.
  + **Sign-Extend Unit**: For certain instructions (like LOAD or ADD with an immediate), the immediate value is sign-extended to match the operand size.
  + **ALU**: (Though used later) Can generate addresses or perform some operations.

**Example**: For the instruction ADD R1, R2, R3:

* Decode: Operand registers R2 and R3 are read from the register file.
* The control unit sets up the signal for an ALU operation (addition).

**Stage 3: Execute (EX)**

* **Operation**: The arithmetic operation (such as addition or subtraction) is performed by the ALU, or the address for a memory operation is calculated.
* **Components**:
  + **ALU (Arithmetic Logic Unit)**: Performs the actual arithmetic operations (ADD, SUB).
  + **Address Generation Unit**: If the instruction involves a memory operation (such as LOAD), this unit calculates the effective address.

**Example**:

* For ADD R1, R2, R3: The ALU adds the values from R2 and R3, and the result will be passed to the next stage.
* For LOAD R1, 100(R2): The effective memory address is calculated by adding R2 and the immediate value 100.

**Stage 4: Memory Access/Write-back (MEM)**

* **Operation**:
  + **For arithmetic instructions (ADD/SUB)**: The result is written back into the register file.
  + **For memory instructions (LOAD)**: The data from memory is read and written back into the register file.
* **Components**:
  + **Data Memory**: In the case of LOAD, this is accessed to retrieve data at the calculated address.
  + **Register File**: The result from the ALU or memory is written back to the destination register.

**Example**:

* For ADD R1, R2, R3: The result is written back into register R1.
* For LOAD R1, 100(R2): The data fetched from memory at the address 100 + R2 is written into R1.

**3. Example Instruction Flow**

Let’s walk through how these stages work for three example instructions in a pipeline:

**Instruction 1: ADD R1, R2, R3**

* **Cycle 1 (IF)**: Fetch the instruction ADD R1, R2, R3 from memory.
* **Cycle 2 (ID)**: Decode the instruction, read R2 and R3 from the register file.
* **Cycle 3 (EX)**: The ALU performs the addition of R2 and R3.
* **Cycle 4 (MEM)**: The result is written back to R1 in the register file.

**Instruction 2: SUB R4, R5, R6**

* **Cycle 1 (IF)**: Fetch the instruction SUB R4, R5, R6.
* **Cycle 2 (ID)**: Decode the instruction, read R5 and R6 from the register file.
* **Cycle 3 (EX)**: The ALU performs the subtraction of R5 - R6.
* **Cycle 4 (MEM)**: The result is written back to R4 in the register file.

**Instruction 3: LOAD R7, 100(R8)**

* **Cycle 1 (IF)**: Fetch the instruction LOAD R7, 100(R8).
* **Cycle 2 (ID)**: Decode the instruction, read R8 from the register file, and compute the address 100 + R8.
* **Cycle 3 (EX)**: The effective address 100 + R8 is computed.
* **Cycle 4 (MEM)**: Data is fetched from memory at the address 100 + R8, and it is written to R7.

**4. Hazards in Pipelining**

In pipelined processors, there are several types of hazards that can occur, which affect the performance:

* **Data Hazards**: When an instruction depends on the result of a previous instruction.
  + **Example**: If the ADD R1, R2, R3 instruction is followed by SUB R4, R1, R5, the SUB needs the value of R1 which is still in the MEM stage.
  + **Solution**: Data forwarding (or bypassing) allows the result of an instruction to be used immediately without waiting for it to be written back.
* **Control Hazards**: Occur due to branch instructions (like BEQ, BNE), which can change the PC and cause the pipeline to fetch incorrect instructions.
  + **Solution**: Branch prediction can be used to guess the branch outcome before it is known, or branch delays can be inserted.
* **Structural Hazards**: Occur when two instructions require the same resource, like access to memory.
  + **Solution**: Ensuring that separate functional units exist for memory access and ALU operations can help.

**5. Pipeline Diagram Example**

This is a basic representation of how instructions flow through the pipeline:

markdown

Copy

Cycle: 1 2 3 4

------------------------------------------------

ADD | IF | ID | EX | MEM |

------------------------------------------------

SUB | | IF | ID | EX |

------------------------------------------------

LOAD | | | IF | ID | EX |

------------------------------------------------

In the diagram above:

* The first instruction (ADD) goes through all four stages.
* The second instruction (SUB) enters the pipeline at cycle 2, and the third (LOAD) enters at cycle 3.

Each stage is doing its part of the process simultaneously for different instructions, increasing throughput.

**Conclusion**

In this design of a 4-stage pipelined processor, we have a straightforward architecture that supports basic arithmetic operations (ADD, SUB) and memory access (LOAD). The pipeline allows overlapping instruction execution, but we must manage hazards like data, control, and structural hazards to keep the processor running efficiently.

This design forms the backbone of many modern processors, and with optimizations like forwarding, branch prediction, and hazard detection, it can be extended to handle more complex instruction sets and multi-stage pipelines.

Would you like more details on any particular part of the design or need clarification on certain concepts?