Kinza Fatima

5-stage pipeline RV32I

A pipeline in computer architecture is a technique used to improve the throughput of a processor by breaking down the execution process into several stages. Each stage performs a part of the instruction execution process, allowing multiple instructions to be processed simultaneously but at different stages of execution.

Why Use Pipelining?

1. **Increased Throughput**: By overlapping the execution of multiple instructions, a pipelined processor can achieve higher instruction throughput compared to a non-pipelined processor.
2. **Efficiency**: Pipelining helps in utilizing the CPU's resources more effectively by keeping different stages of the pipeline busy with different instructions.
3. **Reduced Latency**: Although each instruction takes the same amount of time to execute as in a non-pipelined processor, the overall time to execute a sequence of instructions is reduced.

The 5 Stages of a Pipeline

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

1. **Instruction Fetch (IF)**: The instruction is fetched from memory.
2. **Instruction Decode (ID)**: The fetched instruction is decoded, and the required operands are read from the register file.
3. **Execution (EX)**: The instruction is executed, and any required operations (like arithmetic or logical operations) are performed.
4. **Memory Access (MEM)**: If the instruction involves memory access (e.g., load or store), the memory operation is performed.
5. **Write Back (WB)**: The result of the execution or memory access is written back to the register file.

Implementing a 5-Stage Pipeline in Chisel

1. **Design Each Stage**: Implement each stage of the pipeline as a separate module in Chisel. These modules will be interconnected to form the complete pipeline.
2. **Pipeline Registers**: Introduce pipeline registers between each stage to hold intermediate results and control signals. This allows the stages to operate independently.
3. **Control Logic**: Develop control logic to manage the flow of instructions through the pipeline and handle hazards (like data hazards, control hazards, and structural hazards).

4. **Hazard Detection and Handling**: Implement mechanisms to detect and handle hazards. This includes techniques like stalling, forwarding (bypassing), and branch prediction to ensure the pipeline operates correctly.
5. **Integration**: Connect all stages and ensure that data flows correctly from one stage to the next. Verify that the pipeline handles various instruction types and scenarios effectively.

A basic structure:

```
class Pipeline extends Module {

 val io = IO(new Bundle {
   // Define inputs and outputs
 })

 // Stage 1: Instruction Fetch
 val if_stage = Module(new IFStage())
 // Connect IFStage to the rest of the pipeline

 // Stage 2: Instruction Decode
 val id_stage = Module(new IDStage())
 // Connect IDStage to IFStage and other stages

 // Stage 3: Execution
 val ex_stage = Module(new EXStage())
 // Connect EXStage to IDStage and other stages

 // Stage 4: Memory Access
 val mem_stage = Module(new MEMStage())
 // Connect MEMStage to EXStage and other stages

 // Stage 5: Write Back
 val wb_stage = Module(new WBStage())
 // Connect WBStage to MEMStage and other stages
}
```

1. Define the Instruction Execution Stages

You need to break down the instruction execution into several stages. A typical multi-cycle processor might use the following stages:

1. **Instruction Fetch (IF)**: Fetch the instruction from instruction memory.
2. **Instruction Decode (ID)**: Decode the instruction and read registers.
3. **Execution (EX)**: Perform the arithmetic or logical operations, or calculate addresses.
4. **Memory Access (MEM)**: Access data memory if needed.
5. **Write Back (WB)**: Write results back to the register file.

2. Design the Control Unit

The control unit generates control signals to manage each stage of instruction execution. You need to design a finite state machine (FSM) or control logic to handle the different stages and transitions between them. The FSM will control the flow of instructions through the stages and manage the necessary operations in each stage.

3. Create the Data Path

The data path in a multi-cycle processor includes various components that interact based on control signals:

- **Program Counter (PC)**: Holds the address of the next instruction.
- **Instruction Memory**: Stores the instructions.
- **Register File**: Holds general-purpose registers.
- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logical operations.
- **Data Memory**: Stores data for load/store operations.
- **MUXes**: Multiplexers are used to select inputs to various components based on control signals.

4. Implement the Stages

For each stage of the instruction execution, implement the required functionality:

- **IF Stage**: Increment the PC, fetch the instruction from memory, and update the PC.
- **ID Stage**: Decode the instruction, read the registers, and prepare control signals.
- **EX Stage**: Perform the ALU operation or address calculation.
- **MEM Stage**: Access data memory for load/store instructions.
- **WB Stage**: Write the result back to the register file.

5. Manage Data Hazards

In a multi-cycle processor, you need to handle data hazards that occur when instructions depend on the results of previous instructions. Implement forwarding (bypassing) and stalling techniques to manage these hazards.

6. Update the Control Signals

Ensure that the control signals correctly coordinate the operations in each stage. The control unit should manage signals for ALU operations, memory read/write, and register file read/write operations.

Top file of 5-Stage pipeline

```
class Top extends Module {
 val io = IO(new Bundle {
   val in = Input(SInt(32.W))
   val out = Output(SInt(32.W))
 })

 // Instantiate pipeline stage modules
 val fetch     = Module(new IF("/path/to/instruction/file"))
 val decoder   = Module(new ID)
 val execute   = Module(new EX)
 val memory    = Module(new MA)
 val writeBack = Module(new WB)

 // Pipeline registers
 val F_D_ins    = RegNext(fetch.io.instruction)
 val F_D_pc     = RegNext(fetch.io.pc_out)
 val F_D_pc4    = RegNext(fetch.io.pc_4out)

 val D_EX_regrs1    = RegNext(decoder.io.regrs1)
 val D_EX_regrs2    = RegNext(decoder.io.regrs2)
 val D_EX_aluOp     = RegNext(decoder.io.aluOP)
 // ... (other Decode to Execute pipeline registers)

 val EX_out         = RegNext(execute.io.out)
 // ... (other Execute to Memory pipeline registers)

 val MA_readData    = RegNext(memory.io.readData)

 // Stage connections
 decoder.io.instruction := F_D_ins
```

```
 execute.io.fnct3 := D_EX_branchfun3
 execute.io.x1 := D_EX_regrs1
 // ... (other Execute connections)

 memory.io.addr := EX_out.asUInt
 // ... (other Memory connections)

 writeBack.io.aluResult := EX_out
 // ... (other WriteBack connections)

 // Final output
 io.out := execute.io.out
}
```