Pipeline CPU Design and Architecture

1. Objective

The primary objective of this project was to design and simulate a basic pipelined CPU architecture using Verilog. The CPU includes instruction fetch (IF), decode (ID), execute (EX), memory access (MEM), and write-back (WB) stages with pipeline registers in between. The system supports basic instructions like `LOAD`, `OUT`, and special operations like `ADD`, `SUB`, `COMPARE`, and conditional `JUMP`, using a 2-bit opcode structure.

2. Program Instruction Design

Instruction Encoding

- **Opcode**: 2 bits
    - `00`: LOAD
    - `01`: OUT
    - `10`: SPECIAL (ADD, SUB, COMPARE, JUMP)

Extended Functionality

- Under SPECIAL (`10`), the next two bits (`Rx`) are used as a **function specifier**:
    - `00`: ADD
    - `01`: SUB
    - `10`: COMPARE
    - `11`: JUMP

Operand Encoding

- `Ry` (last 3 bits) is used to specify the second operand register.
- The first operand is stored in a **default register (R1)**.

Sample Instruction Flow

1. `LOAD` value1 into `R1`
2. `LOAD` value2 into `R2`
3. `COMPARE` R1 with R2
4. Based on flags:
    - If R1 > R2: `ADD` R1 and R2
    - If R1 < R2: `SUB` R1 and R2
    - If R1 == R2: `SUB` (result is 0)
5. Output the result

3. Module Description

3.1 Program Counter

- Increments the address by 1 on every rising clock edge.
- Determines the next instruction to be fetched from instruction memory.

3.2 Instruction Memory

- Stores a predefined set of machine instructions.
- Addressed by the program counter.

3.3 Control Unit

- Decodes the opcode and function bits.
- Controls:
    - Register writing
    - Memory access
    - Output enabling
    - ALU source selection

3.4 Register File

- Contains 8 general-purpose registers (R0 - R7).
- Two outputs are used to fetch operands.

3.5 Data Memory

- A simple RAM.
- Used for LOAD operations and memory manipulation.

3.6 ALU (Arithmetic Logic Unit)

- Performs ADD, SUB, COMPARE based on control signals.
- Outputs result to memory/write-back.

3.7 Output Module

- Used to show final output.
- Controlled by `out_en` signal.

3.8 Flags Module

- Sets Zero and Negative flags after operations.

3.9 Pipeline Registers

- IF_ID, ID_EX, EX_MEM, MEM_WB
- Transfers values between pipeline stages every clock pulse.

4. Hardware Design Considerations

Comparator Implementation

- Comparison done via **subtraction** in ALU.
- **Flags interpretation**:
    - `Zero = 1`: R1 == R2
    - `Negative = 1`: R1 < R2
    - `Zero = 0 & Negative = 0`: R1 > R2

Logic-Based Control

- Use of NOT, NAND, AND gates to generate conditional jump controls based on flags.

ALU Function MUX Concept

- An alternative we considered:
    - All operations (ADD, SUB, COMPARE, MULTIPLY) done in parallel.
    - MUX chooses the correct result based on `Rx`.
    - Downside: high power consumption.
    - We optimized it by enabling only one ALU operation using control logic.

Opcode Size Optimization

- Stuck to **2-bit opcode** to reduce instruction memory bit width.
- Used `Rx` as a function selector instead of adding extra opcode bits.

5. Pipeline Design Choices

- Implemented full 5-stage pipeline.
- Registers inserted between stages.
- Clock-driven transitions.
- No hazard detection or resolution yet (for simplicity).

6. Output Routing

- Output is always taken from the result of the ALU.
- Stored in a designated register before enabling output.
- Allows consistent interfacing with output modules.

7. Alternatives Explored

- Using three-register instruction format: `ADD R3, R1, R2`
    - Discarded due to opcode bit width increase.
- Using **default destination registers**
    - Chosen method due to bit efficiency.
- Using **MUX to handle parallel outputs**

    - Considered but discarded for power optimization.

8. Future Enhancements

1. **Hazard Detection Unit**
2. **Branch Prediction**
3. **Immediate Instructions**
4. **Custom Instruction Set Expander Tool**
5. **Python-based Assembler GUI**
6. **FPGA Deployment (e.g., Basys3, Nexys A7)**
7. **Memory-Mapped I/O**
8. **Interrupt Handling**

- (like 7-segment or LED)

9. Hardware Implementation Overview

To implement this pipelined CPU on actual hardware, you'd need to build or program logic blocks corresponding to each module you designed in Verilog. This includes:

- Program Counter (PC)
- Instruction Memory (ROM)
- Register File
- ALU
- Data Memory (RAM)
- Control Unit
- Pipeline Registers
- Output Display

- Clock generator

10. Key Hardware Components

a. Registers (Pipeline and General-Purpose)

- Type: 8-bit D-type Flip-Flop ICs or custom registers using FFs on FPGA
- Example:
    - Discrete: 74HC574 (Octal D-type Flip-Flop with 3-state output)
    - Modular Register Chips: 74HC173 (4-bit D register with enable)
    - FPGA-based: Registers are implemented using flip-flops in the logic fabric (e.g., on Xilinx Artix-7)

b. ALU Implementation

- ALU can be built using:
    - 7483 – 4-bit binary full adder (you'll need two for 8-bit addition)
    - 74LS181 – 4-bit ALU; combine for larger width
    - Or build logic using gates inside an FPGA

c. Instruction Memory & Data Memory

- Instruction Memory → Treated as a ROM
    - Can use EEPROM chips like AT28C256 (32KB EEPROM)
    - Or build using block ROM inside FPGA
- Data Memory → Treated as RAM
    - SRAM Chip: CY7C199 (32K x 8 SRAM)
    - On-FPGA: Block RAM using BRAM in Vivado

d. Control Unit

- Implemented using combinational logic or a ROM-based decoder

- EasilyOSRAM, Lite-On
- Driver: MAX7219 if you want serial control

b. LEDs for binary output

- For quick debug or binary output
- Can use current-limiting resistors + generic LEDs (5mm)

11. Clock Generation

- Use 555 timer for basic clock or
- Use Crystal Oscillator Modules (1MHz, 10MHz, etc.)
- FPGAs have built-in PLLs and global clock networks

12. Platform Recommendation

To simplify, all of this is best implemented on:

- Basys 3 or Nexys A7 FPGA Board (Xilinx Artix-7)
    - Built-in: 7-segment display, switches (input), LEDs (output), clock, USB-JTAG
    - You can directly upload Verilog code via Vivado and simulate it on-chip

13. Power Supply

- 3.3V or 5V Regulated Supply
    - Can use LM7805 voltage regulator
    - FPGA dev boards have USB-powered regulators

Summary of Recommended Components

Function	Component Type	Example
Registers	D-Flip Flop IC	74HC574
ALU	ALU Chip or FPGA Logic	74LS181 or FPGA
ROM	EEPROM	AT28C256
RAM	SRAM	CY7C199
Display	7-Segment Display	Kingbright SC56-11
LED Output	LEDs + Resistors	Generic Red LEDs
Clock	Oscillator	Crystal 10 MHz or 555 timer
Control Logic	FPGA or GAL/PLA	FPGA recommended

- mapped onto FPGA using LUTs
- Could use GAL/PLA if making discrete logic design

e. Pipeline Registers

- Can be implemented using:

    - Multiple 74HC574 ICs for individual pipeline stages
    - Or coded directly in FPGA using `always @(posedge clk)` logic

14. Output Display Options

a. 7-Segment Display (for output)

- Common Cathode or Common Anode 7-segment displays
- Connect through a BCD to 7-segment decoder IC like 74LS47
- Vendors:
    - Kingbright SC56-11 (commonly used 7-seg)
    - SparkFun 7-Segment Display

15. Conclusion

This project successfully simulates a simplified pipelined CPU with core functionalities like instruction fetch, decode, execution, memory access, and output. It supports conditional operations and is designed with low hardware overhead in mind. The pipeline registers make it a step closer to real processor designs. Extensions toward hazard handling, GUI-based tools, and hardware implementation are planned in future work.

16. References

- Verilog HDL by Samir Palnitkar
- Vivado Documentation (Xilinx)
- ISAs of MIPS and RISC-V (inspiration for design principles)