



PROCESSOR ARCHITECTURE AND APPLICATIONS

Mini Project

report on:

Design and Implementation of a Simple 5-stage Pipelined RISC V Processor and Comparing with NOP Handling using Verilog

Gagan R (251090420018)

MTech in Electronics Engineering (Microelectronics)

Dept. of Electronics and Communication Engineering

October/2025

Under the Supervision of:

Dr. G. Subramanya Nayak

ABSTRACT:

The design and implementation of a 5-stage pipelined RISC (Reduced Instruction Set Computer) processor aim to demonstrate the principles of instruction-level parallelism through hardware modeling in Verilog HDL. This project focuses on creating a simplified MIPS-like pipeline architecture consisting of five main stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). The pipeline allows multiple instructions to be processed concurrently, improving throughput and efficiency compared to a single-cycle design. Each stage is implemented as a sequential process synchronized by the system clock, with dedicated pipeline registers for inter-stage data flow. The processor executes a limited set of instructions (LW, SW, ADD, SUB, ADDI) and uses a small register file and memory module for demonstration. Simulation results, obtained using a Verilog testbench and waveform analysis, verify the correct functioning of instruction flow across stages. The project provides valuable insights into pipelined processor design, timing synchronization, and data flow control within RISC architectures.

Table of Contents:

INTRODUCTION:	3
OBJECTIVE:	3
METHODOLOGY:	4
RESULTS:	6
CONCLUSION:	8
REFERENCES:	9

INTRODUCTION:

Processor pipelining is a key concept in modern computer architecture, enabling overlapping execution of multiple instructions to enhance performance. Instead of completing one instruction before starting another, a pipelined architecture divides instruction execution into several stages, allowing each stage to operate on a different instruction concurrently. This approach significantly increases instruction throughput without increasing the clock frequency, thus making efficient use of hardware resources.

In this project, a simple 5-stage pipelined RISC processor is designed and implemented using Verilog Hardware Description Language (HDL). The design follows the classical MIPS pipeline model, which includes the following stages: **Instruction Fetch (IF)**, **Instruction Decode (ID)**, **Execution (EX)**, **Memory Access (MEM)**, and **Write Back (WB)**. Each stage is separated by pipeline registers that store intermediate data and control signals. The main idea is to simulate real-world processor behavior on a smaller scale to understand how pipelining improves performance and what challenges arise, such as data hazards, structural hazards, and control hazards.

The implemented processor supports a basic set of instructions — **Load Word (LW)**, **Store Word (SW)**, **Add (ADD)**, **Subtract (SUB)**, and **Add Immediate (ADDI)** — which are sufficient to demonstrate instruction flow and basic arithmetic/logic operations. The Verilog testbench drives the processor with a predefined instruction sequence and clock signal, and outputs are observed at each stage to verify the correct operation. This pipelined design emphasizes conceptual clarity over complexity, making it an excellent educational model for learning about instruction-level parallelism, register file management, and memory interfacing.

The project contributes to understanding how a RISC-based pipelined architecture can efficiently process instructions, highlighting the importance of synchronization and data forwarding in real-world CPU design. Future enhancements may include hazard detection units, forwarding logic, and branch prediction mechanisms to make the design more robust and closer to an actual processor implementation.

OBJECTIVE:

To design, simulate, and analyse a **5-stage pipelined RISC processor** in Verilog HDL, demonstrating instruction flow through **IF–ID–EX–MEM–WB** stages and comparing with pipeline behaviour with **NOP Handling**.

METHODOLOGY:

The processor design is divided into modular stages corresponding to the pipeline phases. Each stage performs a specific subset of operations and passes the processed data to the next stage through dedicated pipeline registers.

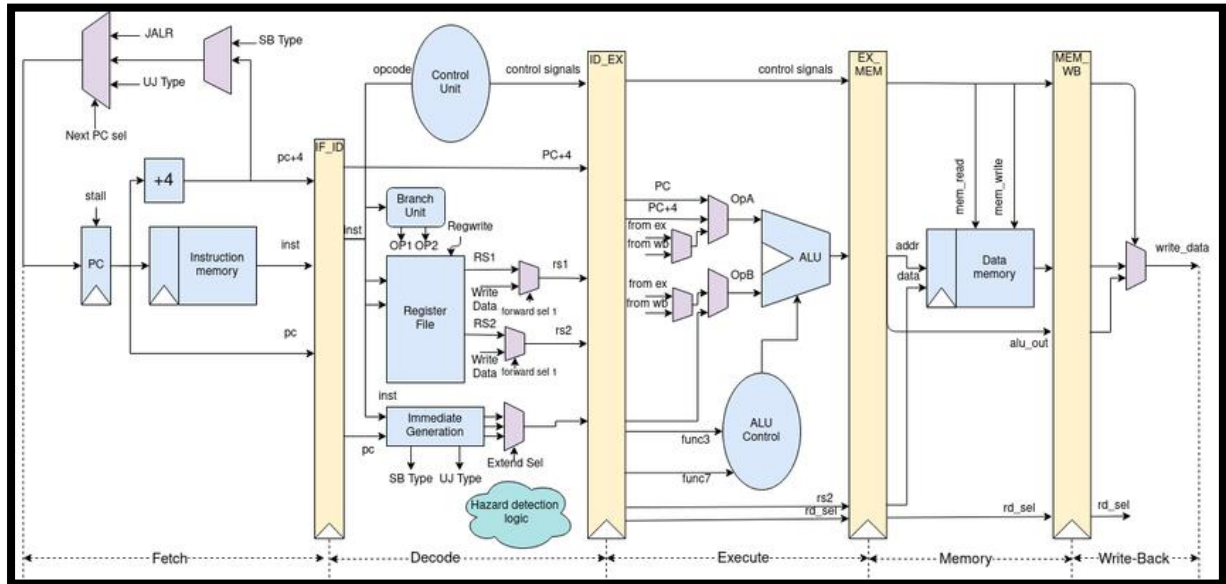


Figure 1: Block of Simple 5-Stage Pipeline

1. **Instruction Fetch (IF):** In this stage, the processor fetches an instruction from the instruction memory based on the current program counter (PC). The PC is incremented after every clock cycle. The fetched instruction and associated data are stored in the **IF/ID pipeline register**.
2. **Instruction Decode (ID):** The fetched instruction is decoded to determine the operation type and operands. In this simplified model, decoding primarily involves identifying the opcode. The corresponding data from the register file is also read in this stage and stored in the **ID/EX pipeline register**.
3. **Execution (EX):** The ALU performs the arithmetic or logical operation as specified by the instruction (e.g., ADD, SUB, ADDI). The result is then stored in the **EX/MEM pipeline register** for use in the memory or write-back stages.
4. **Memory Access (MEM):** In this stage, data memory operations are performed. For load (LW) instructions, data is read from memory, whereas for store (SW) instructions, data is written to memory. Other instruction types simply pass their ALU results to the next stage. The results are stored in the **MEM/WB pipeline register**.

5. **Write Back (WB):** The final stage writes the computed result back to the register file. For simplicity, all arithmetic and load instructions write to a single destination register (e.g., R1).

A **Verilog testbench** generates the clock and reset signals and displays the content of each pipeline stage and register file during simulation. The design is verified using a waveform viewer (e.g., GTKWave) to ensure that instructions progress correctly through each stage, demonstrating overlapping execution.

Algorithm for “Simple 5-Stage Pipeline (Basic Version)”:

1. Initialization

- Register file (R0–R7) is preloaded with values equal to register indices.
- Instruction memory stores 5 example instructions: LW, ADD, SUB, SW, ADDI.

2. Fetch (IF Stage)

- Instruction at current **PC** is fetched and stored in **IF/ID pipeline register**.
- PC increments by 1.

3. Decode (ID Stage)

- Instruction opcode is passed to the **ID/EX register**.
- No hazard checking or operand decoding is implemented (conceptual).

4. Execute (EX Stage)

- Based on opcode: perform ALU operations (ADD, SUB, ADDI) or address calculation for LW/SW.
- Result stored in **EX/MEM register**.

5. Memory (MEM Stage)

- For LW: read from memory using address in EX/MEM register.
- For SW: write to memory.
- Output passed to **MEM/WB register**.

6. Write Back (WB Stage)

- Write result back to **R1** (simplified single-destination model).

7. Repeat until all 5 instructions are fetched; then insert NOPs.

Algorithm for “Improved Simple Pipeline with NOP Handling”:

1. Initialization

- Registers preloaded same as Version-1.
- Instruction memory extended to 8 slots with **extra NOPs (3'b111)** after the 5 instructions.

2. Fetch (IF Stage)

- If $PC < 8$, fetch instruction; otherwise, insert NOP.
- Prevents garbage values after last instruction.

3. Decode (ID Stage)

- Transfers fetched instruction and data into the **ID/EX register**.
- Propagates NOPs properly to drain pipeline.

4. Execute (EX Stage)

- ALU performs ADD, SUB, ADDI, address calculations.
- Unused stages for NOPs perform no operation.

5. Memory (MEM Stage)

- Handles LW/SW as before, but ensures stable propagation of NOPs.

6. Write Back (WB Stage)

- Results written back to R1, identical to previous model.

7. Observation

- Register and stage outputs updated every clock cycle for monitoring.

RESULTS:

✓ Simple 5-Stage Pipeline:

- The pipeline correctly executes a sequence of 5 instructions (LW, ADD, SUB, SW, ADDI) through the stages **IF → ID → EX → MEM → WB**.
- Each clock cycle introduces a new instruction, and after 5 cycles, all stages are active (full pipeline). However, once the last instruction is fetched, undefined or garbage instruction values appear because no NOPs are inserted after the program ends. Register R1 updates correctly during the **Write Back** stage, confirming successful data flow.

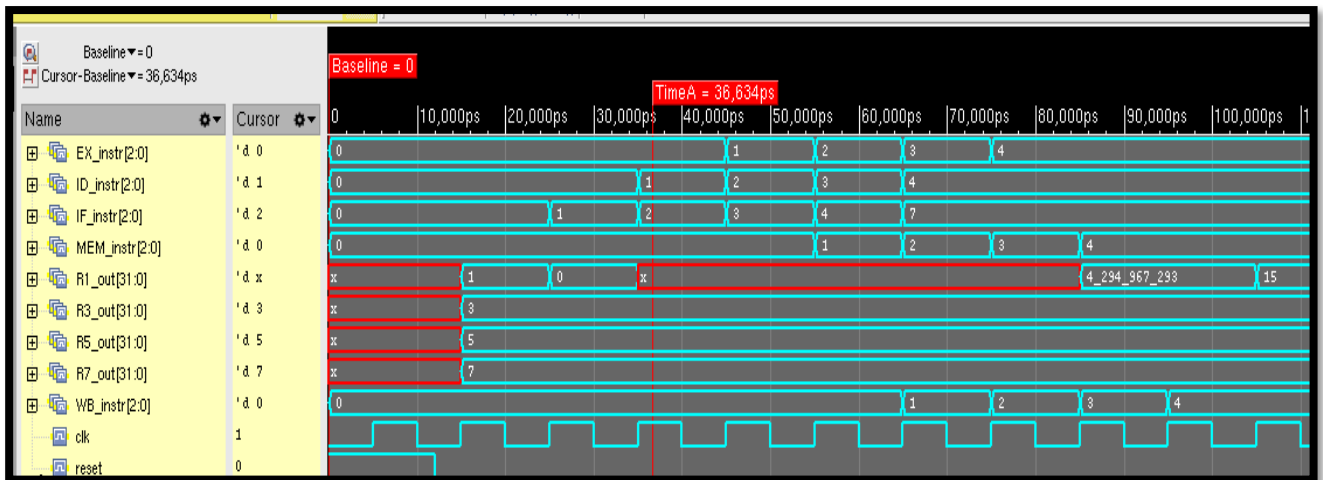


Figure 1: waveform of Simple 5-Stage Pipeline

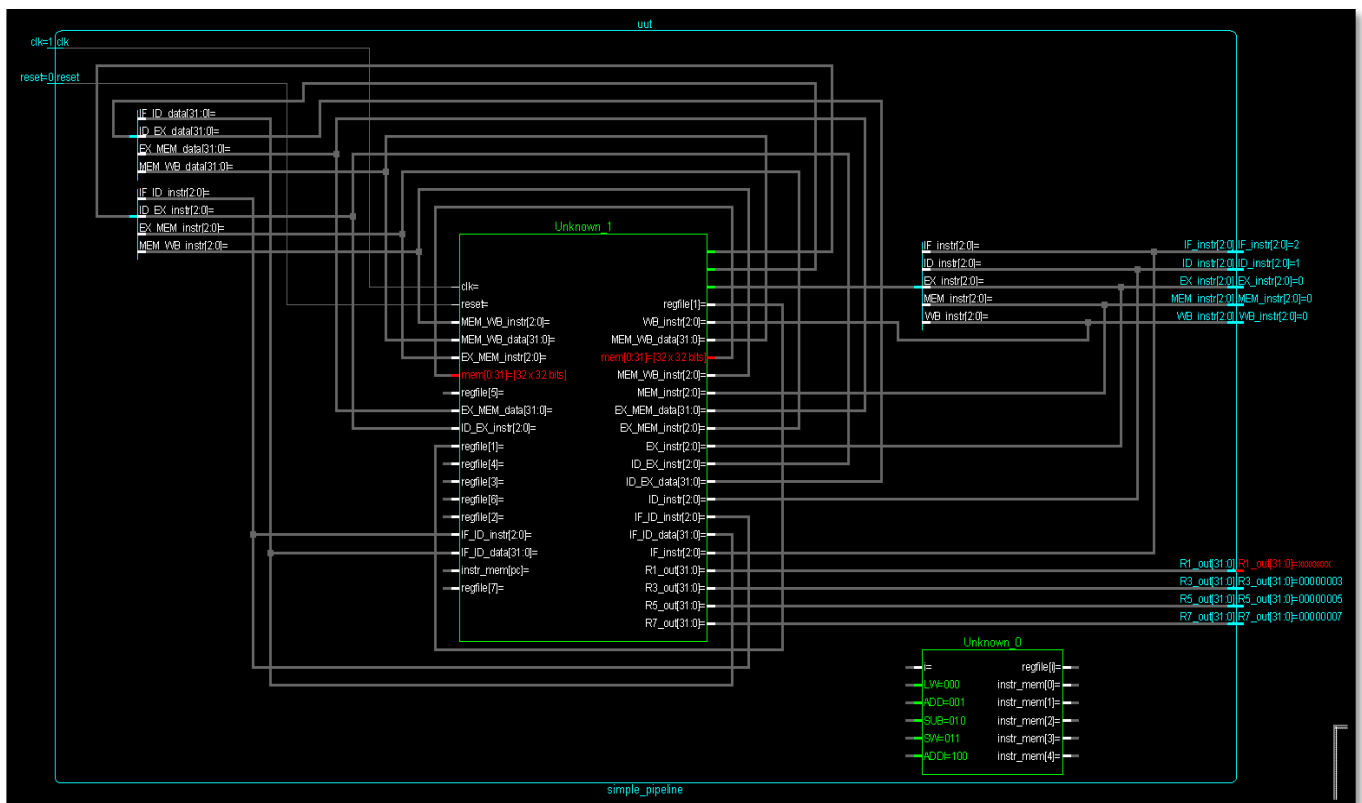


Figure 2: circuit of the Simple 5-Stage Pipeline

✓ Improved Simple Pipeline with NOP Handling:

- The processor behaves similarly but includes **NOP instructions (3'b111)** to automatically fill remaining cycles after the last instruction & This ensures smooth pipeline draining — no undefined instructions occur after program completion.

- The instruction flow is continuous, and once the last instruction reaches **WB**, NOPs occupy all stages, stabilizing the pipeline output & Register outputs remain stable throughout, and final values match expected computation.

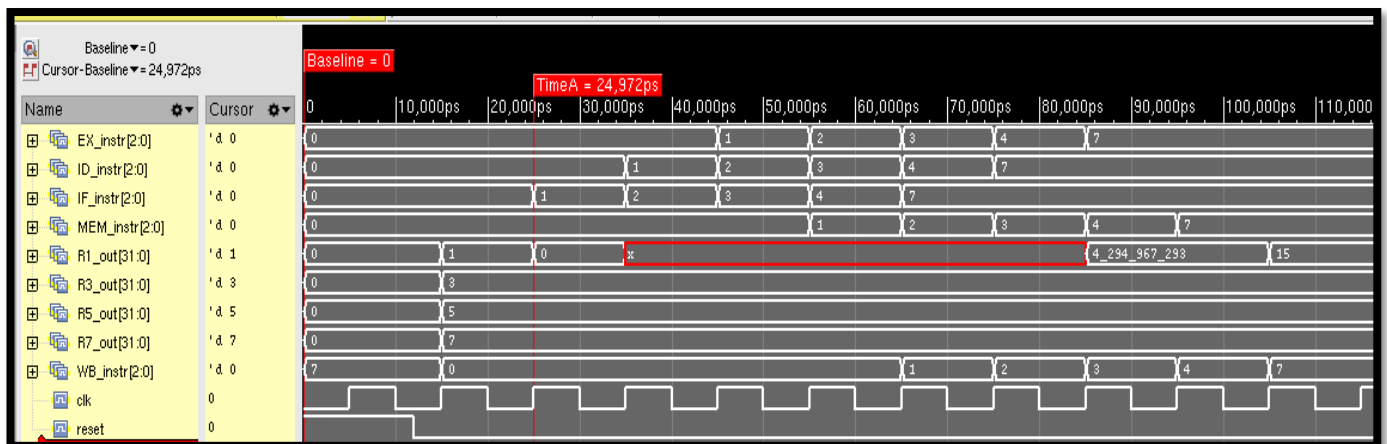


Figure 1: waveform of Simple Pipeline with NOP Handling

CONCLUSION:

Both pipeline designs successfully demonstrate the operation of a 5-stage pipelined RISC processor. The basic pipeline correctly illustrates instruction overlapping but ends abruptly after the last instruction, while the improved version with NOP handling ensures clean pipeline flushing and stable simulation. Overall, both achieve the core goal of showcasing pipelined instruction execution, with the improved model offering a more accurate and robust pipeline behavior.

Comparison table:

Feature	Simple 5-Stage Pipeline	With NOP Handling
Instruction Memory Size	5	8 (with NOP padding)
NOP Handling	No (last stage undefined)	Yes (automatic NOP insertion)
Pipeline Flush	Implicit	Explicit (using NOP)
Data Hazards	Not handled	Not handled (same)
Register Updates	Simplified to R1	Simplified to R1
Simulation Stability	Moderate (garbage after last instr.)	Stable (pipeline drains properly)

REFERENCES:

- [1] Patterson, D. A., & Hennessy, J. L. *Computer Organization and Design: The Hardware/Software Interface*. RISC-V Edition. Morgan Kaufmann. (The classic textbook on CPU design, including pipelining).
- [2] Hennessy, J. L., & Patterson, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann. (Provides in-depth analysis of pipelining and advanced techniques).
- [3] Palnitkar, S. *Verilog HDL: A Guide to Digital Design and Synthesis*. Prentice Hall. (A *standard* reference for learning and using Verilog for digital design).
- [4] "MIPS Architecture." Wikipedia. https://en.wikipedia.org/wiki/MIPS_architecture. (For *general* overview of the MIPS instruction set and pipeline).