MIPS Pipeline CPU

Welcome to the MIPS Pipeline CPU Project in Verilog!
This project features a 5-stage pipelined processor inspired by the classic RISC architecture, complete with hazard detection, forwarding, and flush logic.
Explore the code, run the testbenches, and watch your MIPS CPU come to life!

Table of contents

Introduction

This project implements a simplified MIPS 5 stage pipelined CPU using Verilog. It covers all major pipeline stages—Instruction Fetch, Decode, Execute, Memory Access, and Write Back—along with essential mechanisms like hazard detection, data forwarding, and pipeline flushing. The goal is to simulate a functional and efficient instruction pipeline, similar to those used in real-world processors.

Project Overview and Features

This Verilog-based project simulates a 5 stage pipelined CPU. The pipeline consists of:

1. IF (Instruction Fetch)
2. ID (Instruction Decode)
3. EX (Execute)

4. MEM (Memory Access)
5. WB (Write Back)

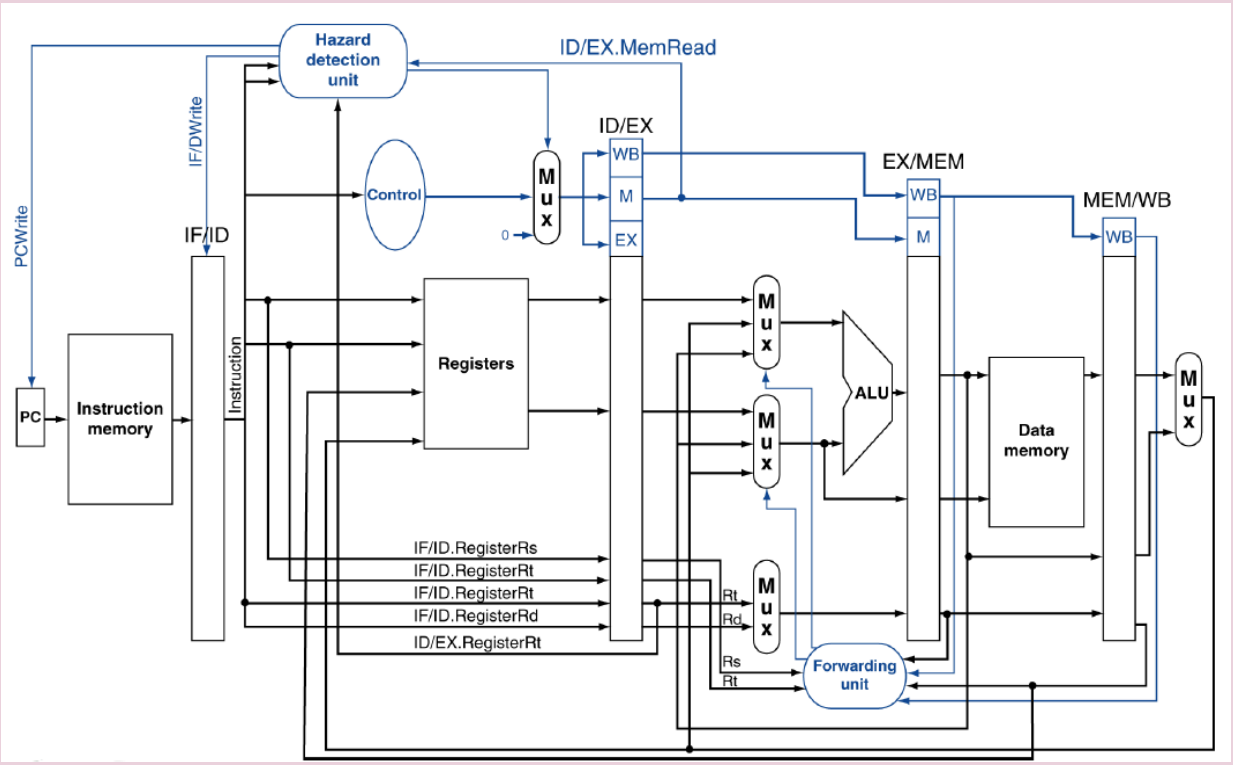
Each stage is connected through dedicated pipeline registers, allowing multiple instructions to be processed simultaneously. The design includes:

- A hazard detection unit to prevent data inconsistencies
- A forwarding unit to resolve data hazards efficiently
- A flush mechanism to handle control hazards from branches and jumps

To test the design, the project includes **two testbenches**; One for basic instruction execution and another focused on testing flush, stalls, and data hazards

Implementation

In the picture below, you can see the simple schematic of this pipeline processor :



Now let's take a look at how each module works.

InstructionMemory

This part stores the program instructions and based on the current PC, it pass the correct instruction to the pipeline processor.

DataMemory

It provides read and write access to data memory during the MEM stage, that helps us to implement lw and sw instructions.

RegisterFile

This module maintains 32 registers and is responsible for reading two register values and writing a result back to one register.

MainControl
It decodes the instruction’s opcode and generates control signals needed for each stage in the pipeline.

ALU

It performs required operations like addition, subtraction, AND, OR, etc., based on the control input.

ALUControl

This module determines the specific operation the ALU should perform, using the funct and ALUOp signal.

HazardDetectionUnit

It detects data hazards between instructions and, when necessary, stalls the pipeline and disables writing to the PC and pipeline registers and it freezes everything.

ForwardingUnit

To minimize stalls, this unit identifies data dependencies and forwards the required data from later stages (EX/MEM or MEM/WB) directly to the ALU inputs.

FlushUnit

This logic unit clears (resets) the instruction and control signals in the IF/ID pipeline registers when a mispredicted branch or jump is detected, to make sure that incorrect instructions do not proceed through the pipeline.

Pipeline Registers (IF/ID, ID/EX, EX/MEM, MEM/WB)

These registers hold the intermediate data and control signals between pipeline stages.

PipelineCPU module

This is the top-level module that connects all the other modules.

ProgramCounter

This module holds the address of the current instruction and updates it each clock cycle, either incrementing it or jumping to a target address in case of branches or jumps or remains unchanged in case of stall.

Now let's look at the datapath:

Instruction Fetch

The processor starts by looking at the Program Counter (PC) to see where it left off. It gets the next instruction from memory and sends it forward with the updated PC, ready for the next cycle.

Instruction Decode

Now the CPU tries to understand what the instruction is asking for. It breaks the instruction into parts, fetches the required values from the registers, and checks if it needs to handle any immediate numbers or prepare for a possible branch. At the same time, it creates all the control signals that the rest of the pipeline will need.

Execute

Here's the main part .If it's an arithmetic instruction, the ALU performs the calculation. If it's a memory instruction, this is where the address is figured out. And if the current instruction depends on the result of a previous one, the Forwarding Unit steps in to quickly deliver the data and avoid delays.

Memory Access

If the instruction involves reading from or writing to memory (like lw or sw), it happens here using the address calculated earlier. For other instruction types, this step is often just a pass-through.

Write Back

Finally, the result of the computation or the data fetched from memory gets written back into the register file. The instruction has now been completed.

Let's break it down by an example.Consider the instructions below. We want to follow the datapath.

```
add $t1, $t2, $t3

sub $t4, $t1, $t5
```

- **Clock 1 :**

  add enters IF stage .

- **Clock 2:**

  add enters ID so control signals are generated and $t2, $t3 are read from the register file.

  sub enters IF.

- **Clock 3:**

add enters EX and ALU calculates $t2 + $t3 and then result is ready by the end of this stage.

sub enters ID and control signals are generated, and operands $t1 and $t5 are read from the register file. But pay attention that $t1 is not updated.

- **Clock 4:**

  add enters MEM and ALU result is passed along.

  sub enters EX and needs $t1 which has just been calculated in add's EX stage).
  This is where the Forwarding Unit detects that $t1 was just written by add, and forwards the ALU result directly to the ALU input for sub so sub uses the correct value of $t1.

- **Clock 5:**

  add enters WB and the result is written back to $t1.

  sub enters MEM and the result of $t1 - $t5 is passed forward.

- **Clock 6:**

  sub enters WB and final result is written to $t4.

Challenges

Implementing the pipeline project has some tricky points. There are many differences between a single cycle processor and pipeline processor.

In the pipeline, we must consider parallelism so we should add a hazard unit in a case that we have data dependencies. The logic behind the hazard unit was a bit confusing. We believe that writing the HazardDetectionUnit, FlushUnit and ForwardingUnit was the hardest part of this project. Also this thing that we had to keep intermediate data was a new challenge for our group. On the paper, you know what the steps are, but it is not easy at all when you want to write a module for each part and then connect them with a main module.

Getting Started

To get started with the project, follow these steps:

1. Open an online verilog compile (we suggest you the one below: https://www.jdoodle.com/execute-verilog-online)
2. Copy “pipeline_final.v” file and then paste it instead of the welcome text in the online verilog compiler page.
3. If you want to test the basic instructions, you should comment the second testbench and if you want to test the flush, Stalls and data hazards, you should comment the first testbench.
4. Now click on the “Execute” button on the right up corner of the page.
5. Wait till the execution is done.
6. Now you can see the result in the output section.

   Pay attention that the testbenches in this project are self-checked. If you see a success message, it means that all of the instructions are working as expected.

Conclusion

This project demonstrates the design and simulation of a 5-stage pipelined MIPS processor in Verilog, complete with hazard detection, forwarding, and flushing. Through modular design and comprehensive testing, it provides a practical understanding of how real-world CPUs manage instruction-level parallelism and handle pipeline challenges. It serves as a solid foundation for deeper exploration into advanced processor architectures.