User Manual for 32-bit 5-Stage Pipelined MIPS Processor

Table of Contents

---

1. Introduction

Welcome to the user manual for the 32-bit 5-stage pipelined MIPS processor. This manual provides information on how to set up, use, and program the processor, including supported instructions, example code, and troubleshooting tips. The processor is designed for instruction-level parallelism and can be used in educational settings for teaching computer architecture concepts, or as a functional processor for FPGA-based applications.

---

2. Processor Overview

The 32-bit pipelined MIPS processor is designed with a 5-stage pipeline that consists of the following stages: - **Instruction Fetch (IF)** - **Instruction Decode (ID)** - **Execution (EX)** - **Memory Access (MEM)** - **Write Back (WB)**

This processor supports a subset of the MIPS32 instruction set and includes basic arithmetic, logical, and memory operations. Pipelining improves execution throughput, enabling simultaneous execution of multiple instructions.

---

3. Getting Started

3.1 System Requirements

To use this processor in simulation or FPGA synthesis, you need: - A computer with a Verilog-compatible simulator (e.g., ModelSim, VCS). - FPGA development tools (optional for hardware implementation, e.g., Xilinx Vivado or Intel Quartus). - Knowledge of Verilog HDL for writing and modifying testbenches. - Basic understanding of MIPS architecture and assembly programming.

3.2 Setting Up the Environment

1. **Download the Verilog Files**: Download the processor's Verilog implementation, including the modules for the ALU, control unit, hazard detection, forwarding, register file, and memories.
2. **Install the Simulator**: Install a Verilog simulator (ModelSim, GHDL, or other compatible tools) for testing and verifying the processor in software.
3. **Testbench Setup**: Use the provided testbench to initialize the processor, load instructions into memory, and run simulations.

---

4. Using the Processor

4.1 Processor Input and Output

- **Inputs**:
  - **Clock signal**: Drives the operation of the pipeline.
  - **Reset signal**: Resets the processor, clearing all registers and starting execution from the Program Counter (PC = 0).
  - **Instruction Memory**: Contains the program (sequence of instructions) to be executed.
  - **Data Memory**: Stores and retrieves data used during program execution.
- **Outputs**:
  - **Register values**: Final values of the general-purpose registers (R0 to R31).
  - **Program Counter**: The address of the instruction being executed.
  - **ALU results**: Outputs from the Arithmetic Logic Unit.

4.2 Loading and Running Programs

- **Step 1**: Write a MIPS assembly program, compile it into machine code (binary format), and load it into instruction memory.
- **Step 2**: Provide the data required by the program into data memory if needed.
- **Step 3**: Start the processor by providing the clock signal and monitor its outputs.

- **Step 4**: Once the program completes execution, check the register file and memory outputs to verify correct execution.

4.3 Supported Instructions

This processor supports a subset of the MIPS32 instruction set, including: - **Arithmetic**: `add`, `sub`, `and`, `or`, `slt` - **Memory Operations**: `lw` (load word), `sw` (store word) - **Control/Branch**: `beq` (branch on equal), `j` (jump)

For a detailed description of each instruction, see the Appendix (Section 7.1).

4.4 Instruction Execution Flow

Each instruction passes through five stages in the pipeline. At any given moment, up to five instructions are in different stages: - **IF Stage**: The next instruction is fetched from memory. - **ID Stage**: The instruction is decoded, and registers are read. - **EX Stage**: The ALU performs arithmetic or logical operations. - **MEM Stage**: Memory is accessed for load/store instructions. - **WB Stage**: The result is written back to the register file.

---

5. Programming Guide

5.1 Writing Assembly Code for the Processor

1. **Start with a program** written in MIPS assembly language. For example:

```
lw $t0, 0($s0)      # Load a word from memory
add $t1, $t0, $t2   # Add two registers
sw $t1, 4($s0)      # Store the result back into memory
```

2. **Assemble the program**: Use a MIPS assembler to convert the assembly code into binary machine code.

3. **Load the machine code**: Insert the machine code into the instruction memory of the processor.

5.2 Instruction Formats

- **R-type** (Register instructions):
  - Format: `opcode | rs | rt | rd | shamt | funct`
  - Example: `add $t0, $t1, $t2` performs `$t0 = $t1 + $t2`.
- **I-type** (Immediate instructions):
  - Format: `opcode | rs | rt | immediate`
  - Example: `lw $t0, 4($s0)` loads a word from memory address `$s0 + 4`.

- **J-type** (Jump instructions):
  - Format: `opcode | address`
  - Example: `j 1000` jumps to instruction address 1000.

5.3 Example Programs

Example 1: Simple Addition

```
addi $t0, $zero, 5  # Load immediate value 5 into $t0
addi $t1, $zero, 10 # Load immediate value 10 into $t1
add $t2, $t0, $t1   # Add $t0 and $t1, store result in $t2
```

Example 2: Memory Load/Store

```
lw $t0, 0($s0)      # Load word from address in $s0
addi $t1, $t0, 1    # Increment the value by 1
sw $t1, 0($s0)      # Store the result back to memory
```

---

6. Debugging and Troubleshooting

6.1 Common Issues and Solutions

- **Instruction not executing**: Ensure the instruction memory is properly loaded, and the clock is active.
- **Data hazards**: Check if forwarding logic is enabled to resolve register dependencies between instructions.
- **Control hazards**: Ensure branch instructions are correctly handled by the pipeline control logic.

6.2 Debugging Tips

- **Use a simulator with waveform analysis**: Observe the timing of register values, memory accesses, and control signals.
- **Step through the pipeline**: Run the simulation cycle by cycle to check each stage's outputs and pinpoint issues.
- **Test small programs**: Start with simple instructions (like `add`, `lw`, `sw`) to validate each module before running complex programs.

---

7. Appendix

7.1 Instruction Set Reference

Instruction	Operation	Format
`add`	Add two registers	R-type
`sub`	Subtract two registers	R-type
`and`	Bitwise AND	R-type
`or`	Bitwise OR	R-type
`slt`	Set if less than	R-type
`lw`	Load word from memory	I-type
`sw`	Store word to memory	I-type
`beq`	Branch if equal	I-type
`j`	Jump to address	J-type

7.2 Pipeline Timing Diagrams

(Include diagrams that show instruction progression through the pipeline stages over time.)

7.3 Glossary of Terms

- **PC (Program Counter)**: Holds the address of the next instruction to be fetched.
- **ALU (Arithmetic Logic Unit)**: Performs arithmetic and logical operations.
- **Register File**: Stores general-purpose registers used during instruction execution.
- **Hazard**: A situation where one instruction depends on the result of a previous instruction still in progress.
