Team10: IAC MIPS Coursework Documentation

Design Overview

This MIPS processor was designed and developed with Compatibility, Flexibility, Modularity, and Robustness in mind. The CPU is MIPS-compliant and could be integrated into various existing integrated circuits. Components of the CPU were implemented in separate modules, which enables future modifications to components without interfering with other functionalities. The design of the CPU was logically partitioned which allows for methodical implementation and maintenance. The CPU is also rigorously examined for all supported instructions and features.

The processor is designed to operate efficiently with security processors, IoT and embedded devices, small sensors, and smart home products. Combining highly efficient data processing functionality with low-power and simple design satisfy many markets demand for today's internet of things.

MIPS CPU Architecture

MIPS

This CPU is a single cycle 32-bit big-endian MIPS 1 CPU. MIPS has a Reduced Instruction Set Computer (RISC) design, which utilizes a small, highly optimized set of instructions. Although the instructions per program is higher, the cycles per instruction is less than the traditional CISC approach[1]. This implementation utilizes 50 instructions of the MIPS1 category in accordance with the MIPS IV Instruction Set (Revision 3.2 September, 1995), including Load/Store instructions, ALU instructions, Jump/Branch instructions, and other instructions such as MTHI and MTLO. MIPS functionalities such as the branch delay slot was also implemented.

Harvard Architecture

The processor uses the Harvard Architecture which provides a separate instruction interface and data memory interface. It assumes that it is connected to a little-endian memory interface, with a combinatorial read-only instruction memory and a data memory with combinatorial read and 1-cycle writes. Hence, it can access both interfaces at the same time and offers a better performance than a traditional von-Neumann architecture.

The processor will begin executing instructions at the reset vector 0xBFC00000 and continues to operate until it attempts to execute the instruction at address 0. This is defined as the Halt behavior of the CPU, at which the CPU will be inactive (active signal de-asserted) and the register_v0 output provides the value of register 2 in the register file.

Design Decisions

The variance in manipulation methods of data within the CPU is taken care of by separate modules, instead of an integrated block. This decision was made to allow for a more structured development and debugging process, as well as provide versatility for future modifications.

---

[1] https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

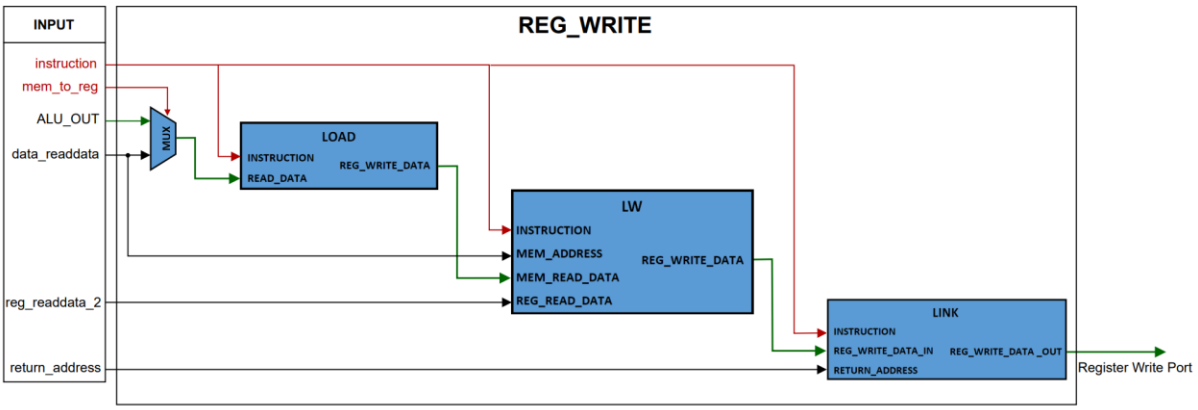

Figure 1 The 'Collective' Register Write Block

As an example, the "collective module" that deals with the manipulation of the data written into the registers is comprised of a cascade of modules/multiplexers. The modules use the instruction input as an active high control signal to decide whether to change the input data according to specific instructions (ex. LW changes the data during the execution of LWR or LWL instructions). For instructions that do not require data manipulation (ex. ALU instructions), the data would be pass-through as shown in green. This design style is reflected in the handling of signals throughout this processor, such as the ALU_B input.

Such design allows for a coherent understanding for collaboration in the development process. The implementation also provides a clear update path for future applications of more extensive instructions or potential modifications to current features. This reflects our design requirements of flexibility and modularity.

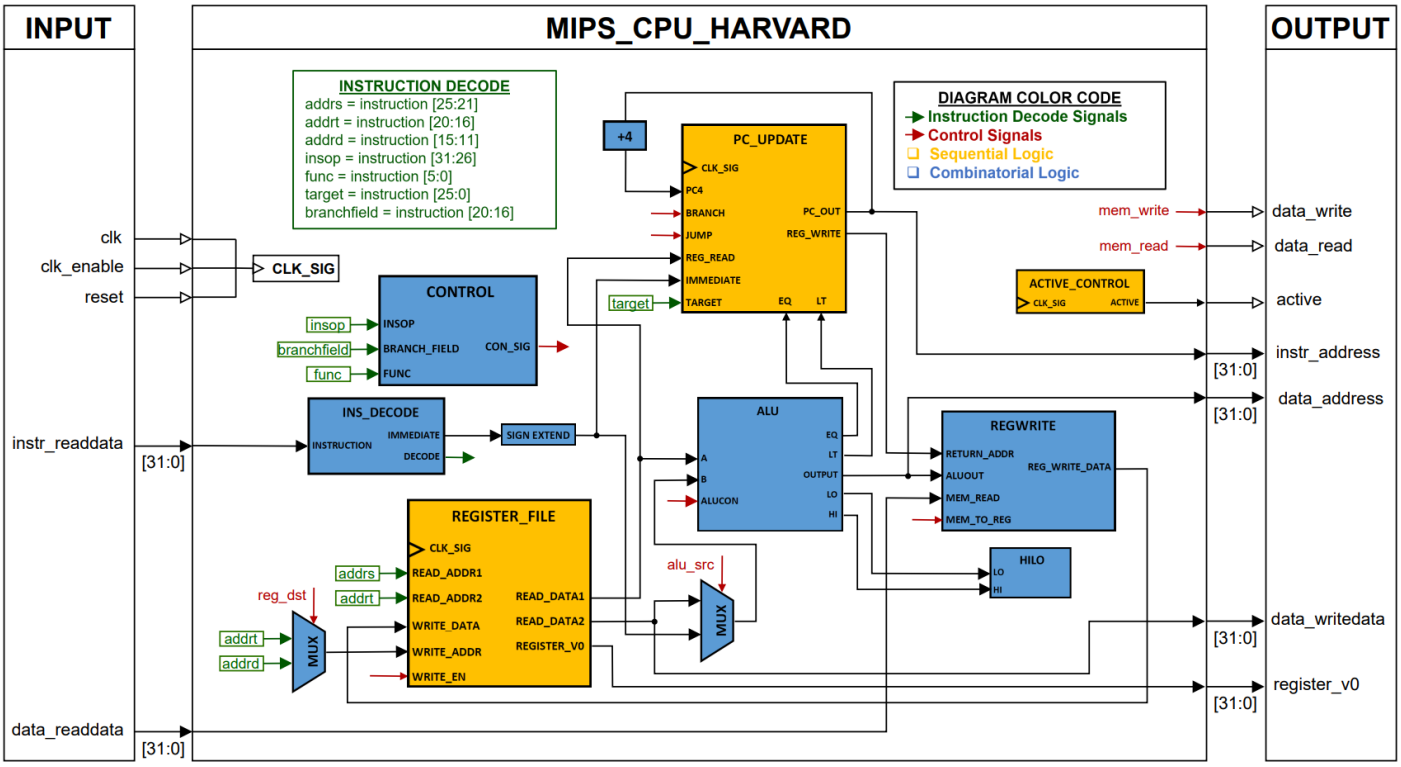

Figure 2 CPU Architecture Schematic

Cyclone IV E 'Auto': Testing Results

Flow Summary:

Flow Status:	Successful - Thu Dec 16 16:00:06 2021
Quartus Prime Version:	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Top-level Entity Name:	mips_cpu_harvard
Family:	Cyclone IV E
Total logic elements:	8,498 / 15,408 (55 %)
Total registers:	1059
Total pins :	198 / 344 (58 %)
Total virtual pins:	0
Total memory bits:	0 / 516,096 (0 %)
Embedded Multiplier 9-bit elements:	16 / 112 (14 %)
Total PLLs :	0 / 4 (0 %)
Device:	EP4CE15F23C6
Timing Models:	Final

Fmax	Restricted Fmax	Clock Name
3.96 MHz	3.96 MHz	Instr_readdata [24]
4.01 MHz	4.01 MHz	Instr_readdata [2]
35.04 MHz	35.04 MHz	Instr_readdata [10]
83.53 MHz	83.53 MHz	clk

Figure 3 Slow 1200mV 85C Model Fmax Summary

*Thus, our timing analysis suggests a conservative lower-bound clock rate of 4MHz.

Testing and Verification: Testbenches and Testcases

Testcases Sequence

Testcases are designed to test individual instructions separately, so in the case of failure, only the components responsible for executing that instruction needs to be debugged. However, dependencies exist within the instruction set where testcases for certain instructions rely on other instructions to initialize or manipulate data.

The hierarchy of the MIPS1 instructions and features are shown in Figure 4, where the lower-placed instructions depend on the higher-placed instructions or features. The sequence of testcases reflect this as well.

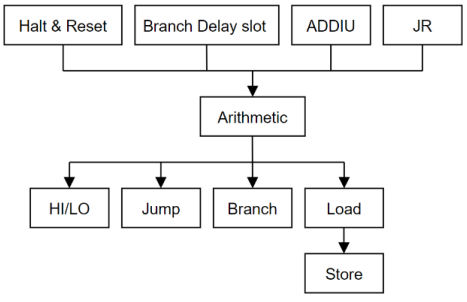

Figure 4 Hierarchy of Instructions and Features

Testbench Design

The testcases are sequentially categorized into the following types and are tested in order (Figure 5):

0 – Halt and Testing Utils:	Basic Instructions (ADDIU, JR) + Features (Branch delay slot, Halt, Register v0)
1 – Arithmetic and Load:	All ALU instructions + Load Instructions
2 – Branch/Jump and Store:	All Branch Instructions + All Jump Instructions + Store Instructions
3 – Functions:	Realistic functions such as if and while to test multiple instructions

The 0-Halt and Testing Utils tests lay the foundation for all other testbenches since all the other instructions rely on ADDIU and JR to operate and halt.

Typically, multiple testcases for each instruction are implemented to test both basic functionality and edge cases. For Conditional Branches, all possible comparisons between the two operands are included and tested. For ALU instructions, some examples of testing edge cases are: testing whether the CPU treated data as signed or unsigned, with or without overflows and underflows, and divide by zero.

Instead of having a centralised assembler that decodes assembly code into machine words, a testbench containing binary machine words at the respective instruction addresses is created for every testcases (i.e. a virtual instruction memory). The main reasoning behind this is to reduce the run-time for rapid testing. This also grants the test-bench robustness since it eliminates a point of failure, but in return sacrifices some degree of versatility when integrating new test cases.

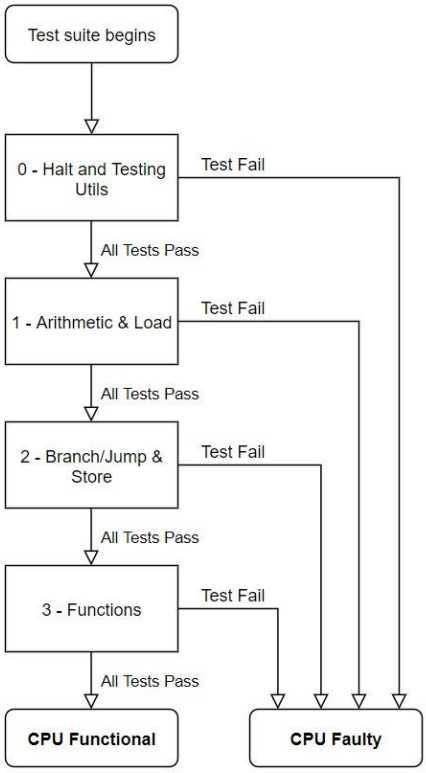

Figure 5 Testbench Sequence

The testbenches are designed to output a 'Fail' status by default. Only when the active signal is low and the register_v0 output meets the expected value (stored in directory /test/5-Reference) will the testbench return a 'Pass' status. When the CPU is halted, the value of register_v0 will be printed to an output file and compared with the expected output. The comparison result will then be printed to stdout indicating whether the testcase passed or failed.

To test Load/Store instructions, a physical data memory is put into the test directory containing 16 pre-configured bytes to test the Load instructions. The store instruction test cases would first store data into the data memory, then load them back to the registers to check if the CPU works as intended.

A shell script is created to automate the testing process. User can either test all the testcases for a specific instruction or test all instructions if not specified.