DATASHEET Team 01 MIPS CPU

1. Overview

This CPU implements a subset of the MIPS IV ISA revision 3.2, specifically MIPS I instructions. It operates using 32x32 bit registers and follows the basic principles of a bus architecture (instructions and data share the same interface to the CPU). As in the given specification, the CPU uses the Intel Avalon memory mapping interface allowing for an overall, more flexible solution. This is a 32 bit, big endian system and there is no support for double word instructions or instructions involving floating point numbers.

2. CPU Architecture

The core components of the CPU are the ALU, controller, instruction register, program counter and register file (see Figure 1):

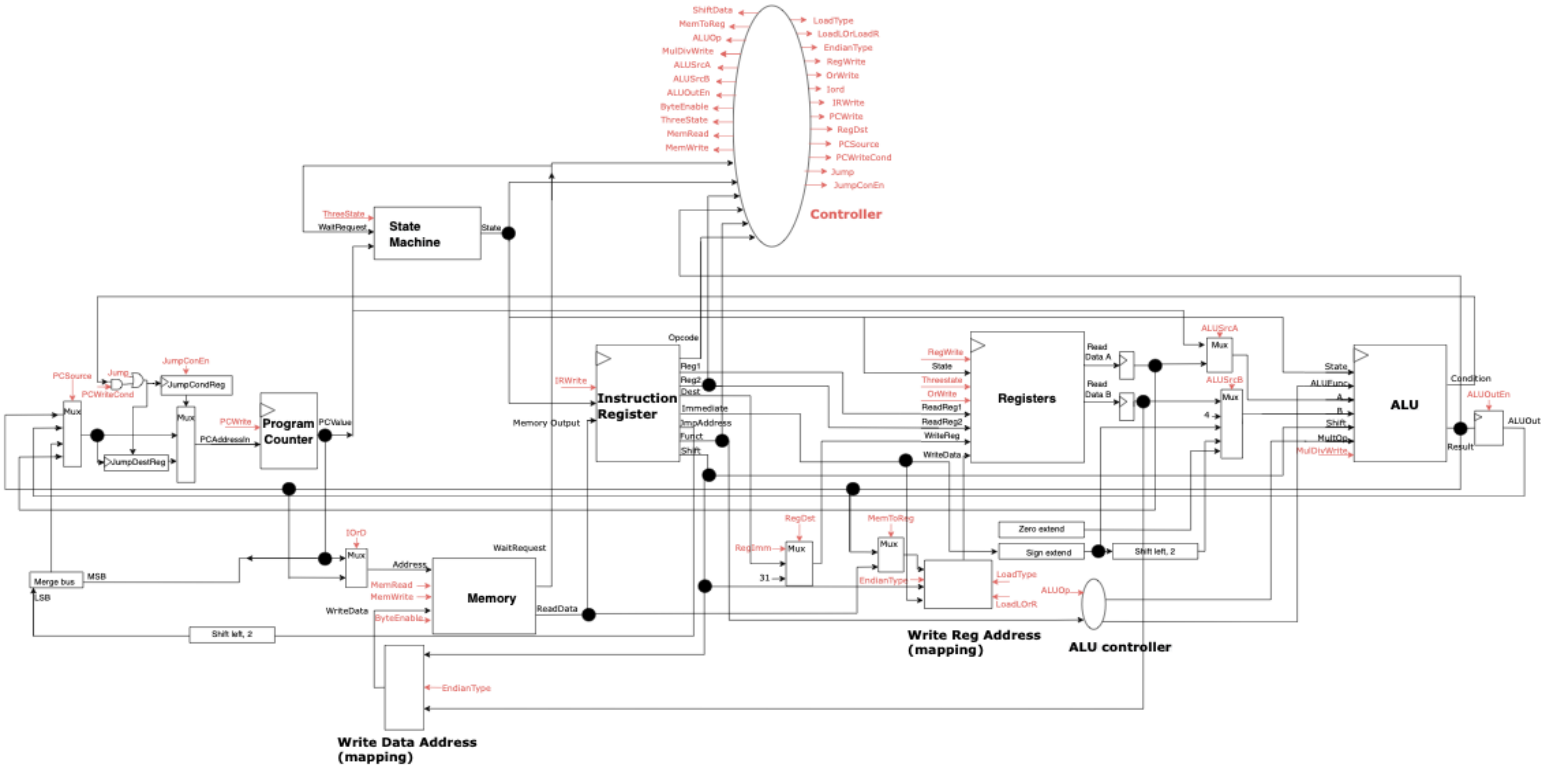


Figure 1: Top level diagram of CPU

2.1 State machine

This block is mostly responsible for synchronising the CPU, including when there are wait requests and cycles of shorter length. It outputs to the controller and instruction register.

2.2 Controller

The controller uses the 32 bit instruction from the instruction register to provide the other blocks and top level with the information they need to carry out their respective instructions. This includes write enables, shifts and the ALUOp which is processed separately by the ALU controller.

2.3 Program counter
The program counter is used to indicate the address of the next instruction to be executed. This starts at 0xBFC00000 as this is the specified instruction start address. Address 0 is the halt address, if the program counter is set to zero, the system halts. When it is enabled, it adds 4 to the current address. The program counter is connected to a set of registers that ensure that the branching is executed.

2.4 Instruction register
The instruction register is responsible for breaking up instructions into the opcode, register addresses, immediates, jump addresses, function code and shifts. It provides the majority of the information required by the CPU to execute a given instruction. Consequently, its outputs feed into the controller, register file, and ALU.

2.5 Registers
This is where the 32 registers are. The register file is 2-read 1-write enabling up to 3 operand instructions to be executed within a single cycle. They feed into another single register (register A and register B), which holds the output for another cycle until it can be fed into the ALU or elsewhere in the CPU.
Registre v0 is included as an output solely for testing purposes and is the main register used to verify the CPU. However, since v0 is not part of the CPU design, it is not included in the diagram.

2.6 Write register address and write data address
These two blocks represent two sets of multiplexers. They are included for mapping purposes, rearranging the the order of the bytes depending on the endian type, load type, and other related signals from the controller. This is needed because the CPU is little endian but designed to use the big endian system for its memory. Although the blocks differ in their implementation, they both serve to normalise the data.

2.7 Arithmetic logic unit and ALU controller
The ALU block performs the arithmetic and logic operations in the CPU and is controlled by a separate controller. This reduces the number of outputs in the CPU controller and creates a level of abstraction for the ALU itself. It also consists of one sub-block, mult/div, which contains the hi and lo registers that are updated in each clock cycle. The sub-block is the only part of sequential logic in the ALU block, which is otherwise combinatorial.

2.8 Top-level multiplexers
The CPU has multiplexers around the top level module that are controlled by the controller block to give the correct inputs to each module.

2.9 Sign extending, shifts, and zero extending
This takes place in the top level module and creates the inputs for some top level multiplexers. They are manipulations of one block's signal to another, such as the immediate from the instruction register being manipulated into 3 different values for ALUsrcB.

3. Design implementation

The solution drew inspiration from David A. Patterson and John L. Hennessy's "Computer Organization and Design - fifth edition" [1]. Within the text there is a simplified version of a MIPS CPU, which we adapted to fit our own specifications and contain the required instruction support.

The key features in our implementation include:

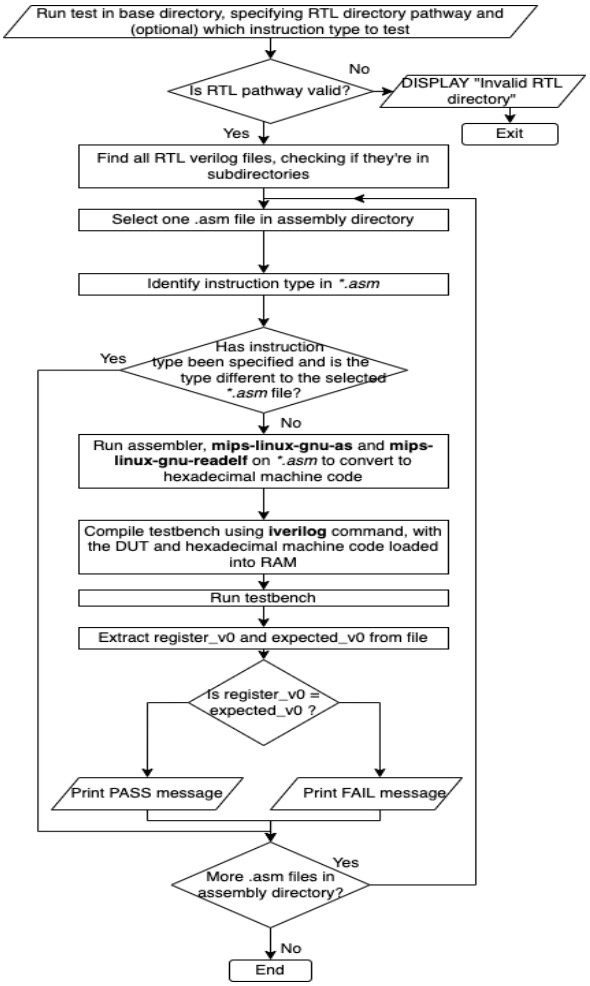
- In SystemVerilog, it's not possible to simulate a 32-bit address RAM. To circumvent this, this CPU works with a 4096x32 RAM and simulates the address that directly maps to the corresponding location in the MIPS RAM. However, there is another level of complexity introduced because our CPU has a little endian interface whilst the memory is big-endian. To take care of this, there are two mapping blocks (one

connected to the register file for the load instructions and another connected to the RAM for the store instructions) to ensure the memory-CPU interactions are accurate.

- When implementing the various load and store instructions, it is more efficient to create a control signal – shiftdata – which is set to modulo 4 of the memory address being accessed during load/store instructions. With this value, the CPU can shift the data going to and from memory to fit with the offset of the load/store instruction.
- For the instructions LWL and LWR, we added a special orwrite signal to the register file module, which allowed for the data from memory to not overwrite parts of the current data in the destination register, as specified in the ISA. When orwrite is high, the register file module ors the shifted data from memory with a combination of parts of the current data and zeros to correctly write in the data. This combination is decided by the value of shiftdata.
- Mult/Div integration into ALU - the mult/div block, also known as the MDU, is built into the ALU block as this helps to focus all mathematical operations into one area. This decision was prompted by ease of design as the integration allowed for a single code to be sent to the ALU controller, then deciphers what is required of its child module. This reduces the number of (top level) controller outputs.
- Using the Avalon interface meant that the waitrequest pin could go high at any time. It was important to ensure correct stalls in the correct cycles so as to not cause any unnecessary propagation delays.
- To implement the jump/branch delay slot, the CPU uses the signal identifying the jump/branch should take place to store the jump/branch destination at the end of the jump/branch instruction. This signal is then fed into a flip-flop which controls a MUX that switches the program counter to take its next value from the stored destination rather than pc+4 during the fetch of the instruction in the delay slot.

4. Testing approach and testbench

Firstly, in order to test this MIPS CPU, the testbench needs a test RAM to load the instructions into.



The assembly directory contains assembly code, which is used to test each instruction from the reduced ISA. The aim of this was both to test normal functionality, and also edge cases in order to holistically assess the correctness of a given design. Furthermore, in this implementation, some instructions depend on the success of other instruction tests. For example, a particular BLEZ test utilises a SUBU instruction to provide a negative input to the BLEZ instruction.

The assembly code is then assembled into machine code (hexadecimal) and then loaded into RAM. Instead of converting each one manually (or writing a C++ conversion program), we used a linux package called *mips-linux-gnu-as*. This ensures that the conversion is accurate. Also, the results directory stores the value of register_v0 in a separate text file for ease of access later.

For the jump and branch instructions, checking the next value in the program counter would be ideal, however, there is a simpler approach: extra instructions, such as ADDIU, are put before and after the branch / on jump target address, to deduce if the CPU executes these instructions properly. This means that these tests check whether instructions are skipped correctly as opposed to directly checking the program counter.

Figure 2: Flowchart of the testbench script

A script was made to automate the testing of instructions. The diagram on the left shows how the test script works.

Firstly, the script is called from the source directory: *test/test_mips_cpu_bus.sh [Directory] [Optional Instruction]* The directory refers to the *rtl* folder which may contain the verilog files or another subfolder called *mips_cpu*. The CPU files may be spread among these directories. The second argument allows the user to specify if only a single instruction type should be tested. If nothing is entered, by default, all instructions are tested.

The assembly programs are stored in *test/assembler/assembly*, where *mips-linux-gnu-as* is called to convert the *TESTNAME.asm* files into hexadecimal machine code. The machine code is then loaded into our *mips_cpu_avalon_RAM.v* and our test bench compiles and runs the MIPS CPU. After, the result of register_v0 is written to a text file and stored in a folder called *compiled_results*. A comparison is then made against a predetermined value stored in the *results* folder, if the text files contain the same value for register_v0, the test case passes. If the test case fails, the obtained and correct value for register_v0 is output for debugging purposes.

5. Evaluation

Using the timing and compilation tools in Intel's Quartus prime with Cyclone IV E and device EP4CE30F23C6, we were able to determine the total area that our solution would use:

Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimated Total logic elements	10,330
2		
3	Total combinational functions	9306
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	5003
2	-- 3 input functions	3942
3	-- <=2 input functions	361
5		
6	Logic elements by mode	
1	-- normal mode	6883
2	-- arithmetic mode	2423
7		
8	Total registers	1285
1	-- Dedicated logic registers	1285
2	-- I/O registers	0
9		
10	I/O pins	138

Simulating our design with the slow model under the realistic conditions of 1200mV at 85 degrees Celsius, we were able to determine a maximum clock speed of: **8.33MHz.**

Without timing optimisations, this is a strong start towards the approximate clock speed of 1GHz of the MIPS32 74k processor.

References

[1]

D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, Fifth edition. Elsevier/Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, 2014. ; IBSN: 978-0-12-407726-3