ISA MIPS Datasheet - Team 14

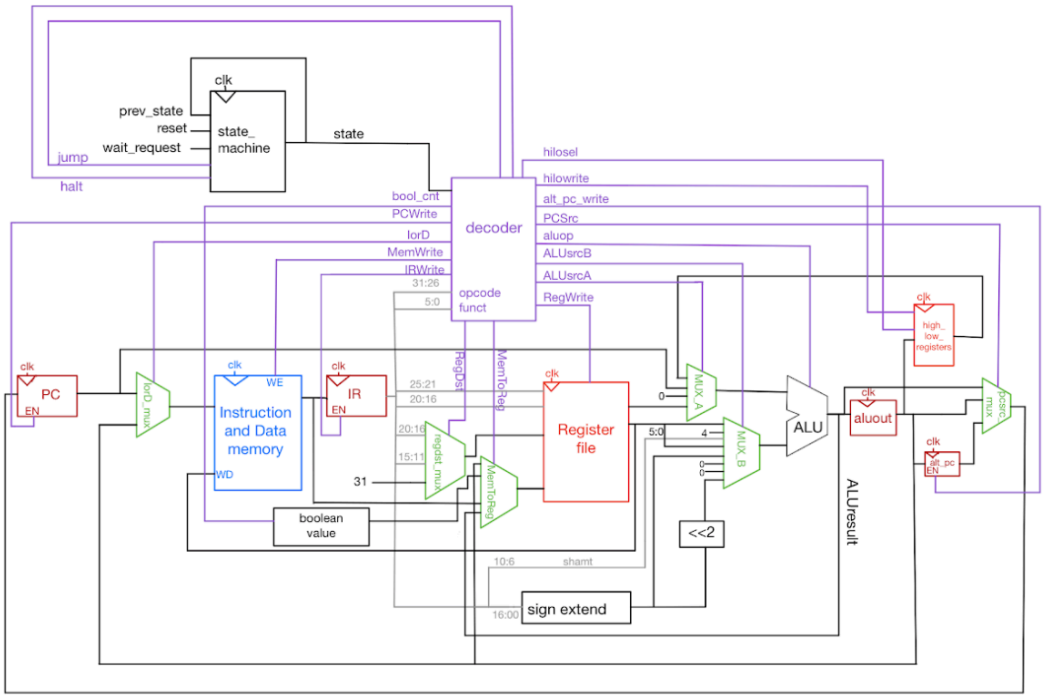
Specifications:

The specifications for this CPU design project are to create a working synthesizable MIPS-compatible CPU as well as the corresponding infrastructure required to thoroughly test it or other CPUs. It must implement 50 instructions from a 32-bit little-endian MIPS1, specified in MIPS ISA Specification (Revision 3.2). Furthermore, the CPU must have a maximum CPI of 36, and the test bench should take fewer than 10 minutes to run. Within the specification there is a choice between implementing either a bus-based memory interface or a harvard interface.

The CPU is designed and written in the hardware description language System Verilog, with synthesizable code that can be written to a Cyclone IV FPGA. The CPU has a CPI of 5, with most instructions taking exactly 5 cycles. The CPU is also implemented using a bus interface in order to be more realistic.

CPU Design:

The overall CPU architecture can be seen below:



Design decisions:

One of the most important decisions is how the CPU is controlled, which is largely a result of the state machine. As a result, the state machine was one of the first things to be designed, and it influenced the implementation of all instructions.

The CPU typically operates in five states: FETCH, DECODE, EXEC1, EXEC2, WRITEBACK.

- In FETCH, the instruction currently stored in the address of the PC is read from memory and wired into the IR. The PC is also incremented by 4 to be ready for the next instruction.
- In DECODE, the instruction from the memory is decoded by the decoder, and the required registers are selected from the register file.
- In EXEC1, the registers and potentially the immediate are operated on by the ALU according to the ALUOP provided by the decoder. This result is both immediately available in the wire ALUresult, as well as being stored in the ALUout register to be available next cycle.
- In EXEC2, the result of ALU operation will be used to either write to the memory or the destination register or to select the address of the memory data to be read from if the instruction is a load.
- In WRITEBACK, if the instruction is a load then the data from the calculated address will be used to write to the destination register.

There is one additional state used to implement the branch delay slot, ALTDECODE.

- ALTDECODE is very similar to the DECODE state, except that the value of the PC will be updated with the destination address of any jump or successful branch. This state replaces the DECODE state, and is used when the previous instruction was a jump or successful branch.

There are three additional states used in the startup and halting of the CPU, specifically they are: RESET, START, END.

- In RESET, the CPU will loop to this state as long as the reset signal is held high. In this state all registers are set to 0. Whenever the reset signal goes low, active will be turned high and the CPU will go into the START state.
- In START, the PC will be loaded with the reset vector so that the instruction can be ready in the first FETCH cycle.
- In HALT, the CPU will loop to HALT as well as set the active signal to be low. This state will be entered when the instruction at memory address zero would be executed.

Specific instruction implementations

The branch and jump instructions are all implemented via the use of an alternate PC register in order to facilitate the branch delay slot. If any given instruction is a branch (or a jump) the pc will be incremented normally in the FETCH state. When the instruction is determined to be a jump or a successful branch, the destination will be stored into the alternate PC register and a “branch” signal will be set high. The state machine receives this signal and uses the ALTDECODE state, in which the value from the alternate PC is loaded into the PC and the jump signal is set low.

The HILO register is implemented as a separate register file with 2 control signals: hilosel, and hilowrite. Hilowrite indicates whether the register should be written to and hilosel selects whether the register that should be written to or read from is the HI or LO register. The input to this register comes from the ALUout register, and the output of this register is multiplexed into the A input of the ALU.

The various load instructions (LB, LBU, LH, LHU, LWL, LWR) are all implemented via a masking block after the data output of the memory. This block is controlled by the decoder and the final 2 bits of the address fed into memory. It subsequently shifts the data to the correct format. Similarly, for SB and SH, there is a mask that eliminates the most significant bits of the writedata, which is also controlled by the decoder. The byte enable signal is controlled by a block that decodes the last 2 bits of the address to identify which byte or half word address is being stored to.

Timing & Area Analysis

The maximum frequency generated by the Quartus timing analysis was 8.49 MHz for a slow silicon 1200mV FPGA chip at 0°C. This speed was subsequently analysed further by isolating the ALU within the CPU. When conducting the timing analysis on solely the ALU surrounded by registers buffering the in/out pins, the maximum frequency achieved was 9.72MHz. This indicates that a large amount of the delay is due to the implementation of the ALU and its relatively large effect on the critical path. This is due to the fact that both multiplication and division operations in the ALU are implemented as single cycle, as opposed to multi cycle. This obviously ensures a low CPI however the maximum frequency has clearly been limited. If a higher frequency were desired the ALU could be changed to implement the multiplication and division over multiple cycles to lower the critical path.

Analysis & Synthesis Resource Usage Summary

	Resource	Usage
1	Estimated Total logic elements	9,416
2		
3	Total combinational functions	8390
4	⌄Logic element usa…ber of LUT inputs	
1	-- 4 input functions	4702
2	-- 3 input functions	3161
3	-- <=2 input functions	527
5		
6	⌄Logic elements by mode	
1	-- normal mode	5781
2	-- arithmetic mode	2609
7		
8	⌄Total registers	1231
1	-- Dedicated logic registers	1231
2	-- I/O registers	0
9		
10	I/O pins	142
11		
12	Embedded Multiplier 9-bit elements	16
13		
14	Maximum fan-out node	clk~input
15	Maximum fan-out	1231
16	Total fan-out	33409
17	Average fan-out	3.37

The area analysis for the CPU can be seen on the right.
These values are for an implementation on an EP4CE15F23C6 chip, which the Quartus timing & area analysis determined to be the optimal Cyclone IV FPGA for the CPU design.

The Test-bench

To simplify testing, bash scripts are used to automate necessary operations. These test scripts compile and run programs to simulate what the CPUs should output to the ram as well as the $v0 register. By comparing the CPU output to that of the simulated output, the test scripts will determine whether the CPU has performed its instructions correctly and output the result of this comparison.

To test all instructions and the functionality of the CPU, only the script test_mips_cpu_bus.sh will have to be run, which can either run all test-cases for a given CPU or alternatively, run a selected group of test-cases if an instruction name is supplied as an additional argument. The script will run with the following input format:

bash ./test/test_mips_cpu_bus.sh [CPU_source_path] [instruction_name(optional)]

For more detailed testing, the lower level scripts test_temp_w.sh and test_temp.sh can be used, which work identically to test_mips_cpu_bus.sh, but take in the test-case name instead to run a single specific test case:

bash ./test/test_temp.sh [CPU_source_path] [testcase_name]

Specifically, the scripts feed the MIPS assembly test-cases into the assembler which translates them into the machine code. This machine code is subsequently passed into both the CPU and the simulator for testing. Both the CPU and Simulator will then output the final contents of the memory, the contents of the register $v0, and the list of memory addresses that were accessed. To make the analysis of the memory easier, only non-zero memory is outputted along with its address, while the non-outputted memory contents are assumed to be null. If the memory contents, register $v0 and the list of addresses accessed are identical then the test will be considered a “Pass” and if they are different it will be considered a “Fail”. The format of the final output is as follows:

addiu_1 addiu Pass

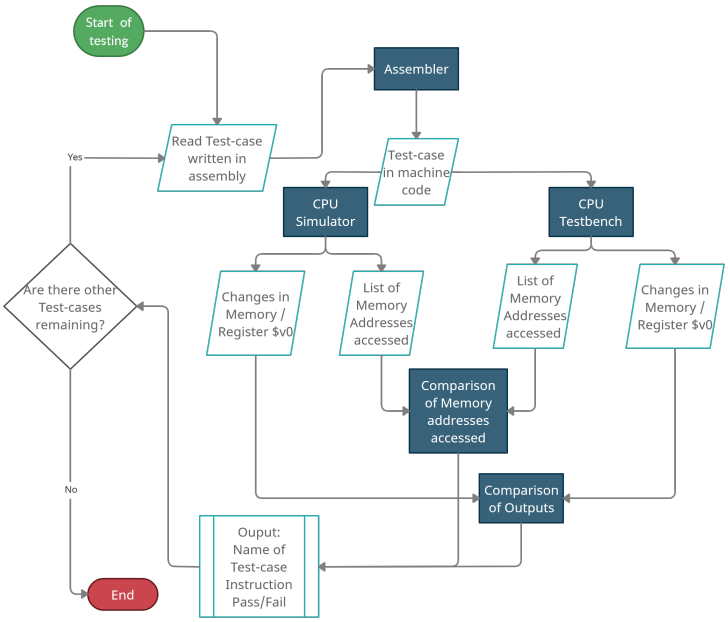
Where the first, second and third column refers to the test case ID, instruction being tested and result of the test respectively.

The difference between the *test_temp_w.sh* and *test_temp.sh* scripts is that the former manually toggles the wait-request every cycle whilst the tests are running. The tests being run are identical and this script serves to ensure that the wait-request can be high at any given cycle and not affect the functionality of the CPU.

Test flow Design:

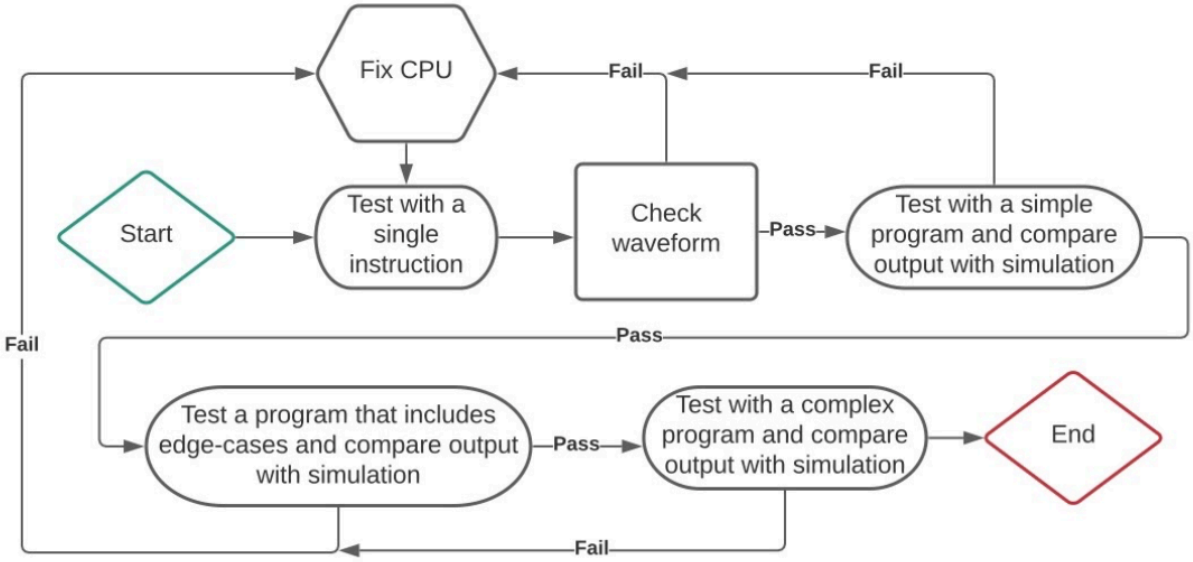
The test_bench was created with the intention to be as simple as possible to use. Any test-case can be run by adding the assembly code to the *./test/test-case* folder. This will be used by the test script according to the diagram on the right to indicate whether or not the CPU can pass this test-case.

This test process is enabled by the help of two components - the assembler and the simulator, which translates the test-cases written in MIPS assembly into a reference output, which is compared to the CPU output to determine the outcome of the test-cases.



The CPU Testing Process

When testing the CPU, the first instructions to be tested were ADDIU, SW, LW, and JR. There were several reasons for this: specifically these instructions were relatively simple to implement, they allow the CPU to halt, they allow the cpu to both read and write from memory and they allow for simple manipulation of the values held by various registers. As a result, most of the final test-cases utilise these 4 instructions in combination with the actual instruction being tested.



As can be seen in the diagram above, the CPU is tested in an iterative method. The first step is to create a test-case with just the individual instruction, the waveform of this would subsequently be checked to ensure that it is what was expected, and if there are any differences, whether it is correct or incorrect. Subsequently this instruction would be used in a simple test case that would allow the CPU to actually halt, and compared with the simulator. Once again the waveform of these results regardless of pass or fail are manually checked to ensure that both the CPU and simulator behaved as expected. If there are any discrepancies the test case would be manually reexamined and the CPU (or simulator) would subsequently be changed if required. After this has been done for each instruction, more complicated test cases which are intended to catch any edge cases will be run on the CPU. At this point the waveform would only be examined if the test case failed. The fault would subsequently be fixed in the CPU. Finally, these instructions are included in more complex tests that are intended to more accurately represent the typical usage of a CPU.

Final Test Cases

The final test cases included are intended to test all levels of functionality of the instruction. For example, the test-case for *ADDIU_1.asm.txt* is the first test case that checks the functionality of the MIPS addiu instruction. The test-cases *ADDIU_2.asm.txt*, *ADDIU_3.asm.txt*, etc, further check the functionality of addiu by applying different instruction operands, targeting operands that trigger edge-cases in the functionality of the instruction. Specifically for addiu, this can be checking the immediate in the addiu to be 0 or the largest unsigned 16-bit number (65,535). Other edge-cases may also include adding from and storing the result to the same register, adding non-zero immediates to $0, etc.

This methodology of creating test cases that specifically target edge cases was applied to the entirety of the instructions within the MIPS1 instruction set. This ensures that all instructions are thoroughly checked and are tested for as many edge cases as possible. Finally included are the more complex test-cases designed to test common program structures such as loops, function calls, recursions, structs, etc. For example, JALR_99.asm.txt calls two functions which create a linspace vector of variable size and interval and then applies a linear function to its values.