

MIPS CPU Instruction Architecture and Compilers Coursework Team 8

Design Decisions and Architecture

The objective of the design decisions in this unpipelined CPU was to achieve the highest CPI possible. Having four cycles per instruction makes the CPU more robust. These states are (FETCH, DECODE, EXEC and WB (WriteBack)). Whilst 5 states are possible, it is more efficient to have just 4 because this pushes the CPI down from 5 to 4. It is possible to implement a non-load-store instruction in three cycles. At start-up, the reset signal is pulsed. This causes the FSM to begin in WB to ensure that all values are correct in FETCH. The FSM state is dependent on the previous state, 'reset', 'waitrequest', and the signal 'halt' ('halt' goes high when the PC's address is 0). The 'HALT' state stops the CPU and makes it enter a stable state.

All instructions take 4 cycles (to avoid creating logical overhead): one cycle to FETCH the instruction, the next cycle to DECODE it and set up the control signals around the CPU. The ALU output is available during the EXEC stage so, data is sent to the RAM during store instructions. Finally, in WB, the address is set to PC and the RAM is told to read the next instruction. The register file is written to in WB and the correct PC value is updated (updating PC in WB makes it easier to deal with branch and jump instructions). In FETCH, the instruction is available to process.

Design choices were made to ensure that the CPU is extendable, scalable, and maintainable [1]. The CPU components have been split into the following components: control, registers, FSM. The PC, 'pc_next', ALU and the IR are all in the top-level sheet (reducing overhead code duplication and module instantiations), making the control path logic more legible. The FSM drives the logic of the CPU by ensuring that all components output their values out in the right state. By placing functionally independent components, in their own module sheet, the logic is made easier to understand. This makes the CPU's logic is easily extendable and maintainable.

One factor taken into consideration was the critical path of the ALU during addition and subtraction calculations. ALU dependent instructions were modelled without delays in the test benches to ensure that the clock speed is independent of the critical path. The bottleneck for the number of cycles per instruction is limited by memory access since it takes one cycle for the instruction/data to come out. The register enables are clocked components which ensures data, at most, flows through one component.

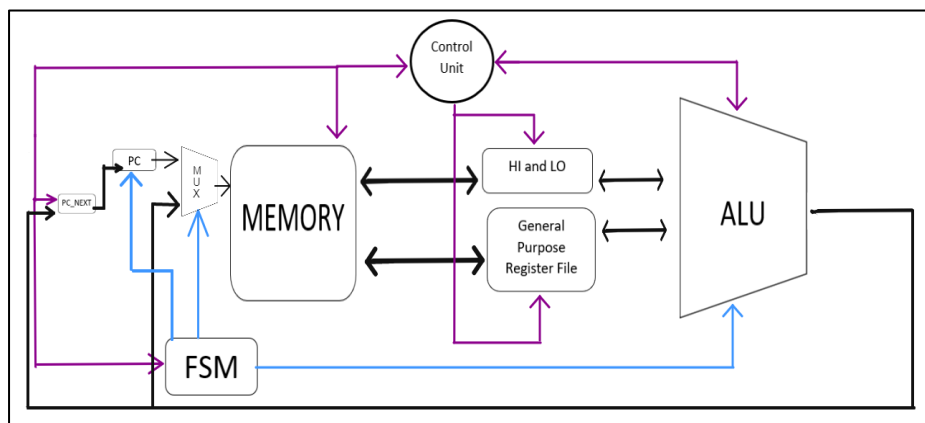


Figure 1: MIPS CPU diagram showing how data flows between components and how they interact with each other.

Some logic signals and registers have been added to smooth instruction processing. In FETCH, PC is set to PC_next, which contains the address after the delay slot instruction has happened or PC + 4.

The IR ensures the correct control flags

are still valid during the instruction cycle because the RAM data changes during the instruction cycle (data and instructions come out of the same RAM). Another signal, 'stall', has been used to help process signals. It goes high when there is a wait-request, or if the CPU is halted, or the CPU is not active. This ensures that the FSM does not change state when we are waiting for the RAM to complete its reading or writing.

Testing:

The CPU testbench aims to test the functionality of the CPU by testing each instruction individually. Then instructions are merged to form more complex test cases to assert multiple component interactions are correct. The output for some comparisons is stored into \$v0, which is bound to register 2 of the general-purpose register file. Some test cases are short (to test the basic correctness of the instructions) while some are longer to test for correctness in the context of a small program.

Most test cases run LW to load values into registers and, some use SW to load the values from the registers into memory locations. Test cases can check the content of \$v0 and make assertions about the content of some memory locations. At the end of every test case, JR \$zero is run to stop execution (by entering the HALT state) followed by a NOP (00000000).

The RAM, which has 2^{32} byte addresses, is used as an interface to the CPU. Large memories are not ideal for simulation purposes. Simulating a 4GB RAM is very slow on low-end computers, and they cannot run the simulation [2]. Instead, smaller RAMs can emulate the behaviour of larger RAMs since the CPU only sees the interface between itself and the memory. The RAM size has been decreased to increase the speed of simulations. But RAM was also split into data and instructions. Data covers memory ranges 0 – 0x1000. Instructions cover memory ranges 0xBFC00000 – 0xBFC01000, where 0xBFC00000 is the reset vector and the starting point of the instructions in the test cases. Assertions are made on the first 8 words of data RAM.

The RAM and CPU files are bound to the testbench (mips_cpu_bus_tb.v). The CPU, which has a clock period of 2, runs for the number of cycles defined by the number of instructions * 36, with 36 being the worst-case scenario CPI target given in the specification. This accommodates for the longest possible test cases of multiplication and division.

Along with the testbench and the bash script, a python script reads the config files in the config path and builds a Verilog testbench containing the information for the test case.

If there is a STDERR error, the Verilog file has failed to compile. If a test case fails, “fatal” or “error” occurs. If the CPU fails to execute in the maximum amount of time, the simulation will stop and give a fatal.

The config file ([Figure 3](#)) can generate waveforms, produce wait requests, change the timeout limit, add further comments about test cases, and display error messages in the terminal.

The following commands are for the addui instruction:

- success:good for addui (optional success comment)
- fail:fail for addui (optional fail comment)
- comment_display:false/true (print comment after execution)
- waveform:false/true (generate VCD files)
- verbose:false/true (to print out results in the terminal)
- waiting:false/true (pauses after each testcase to let the user see TB/RAM)
- waitrequest:false/true (produces pseudorandom waitrequests)
- CPI_LIMIT:400 (change when the TB times out)

Testing Approach:

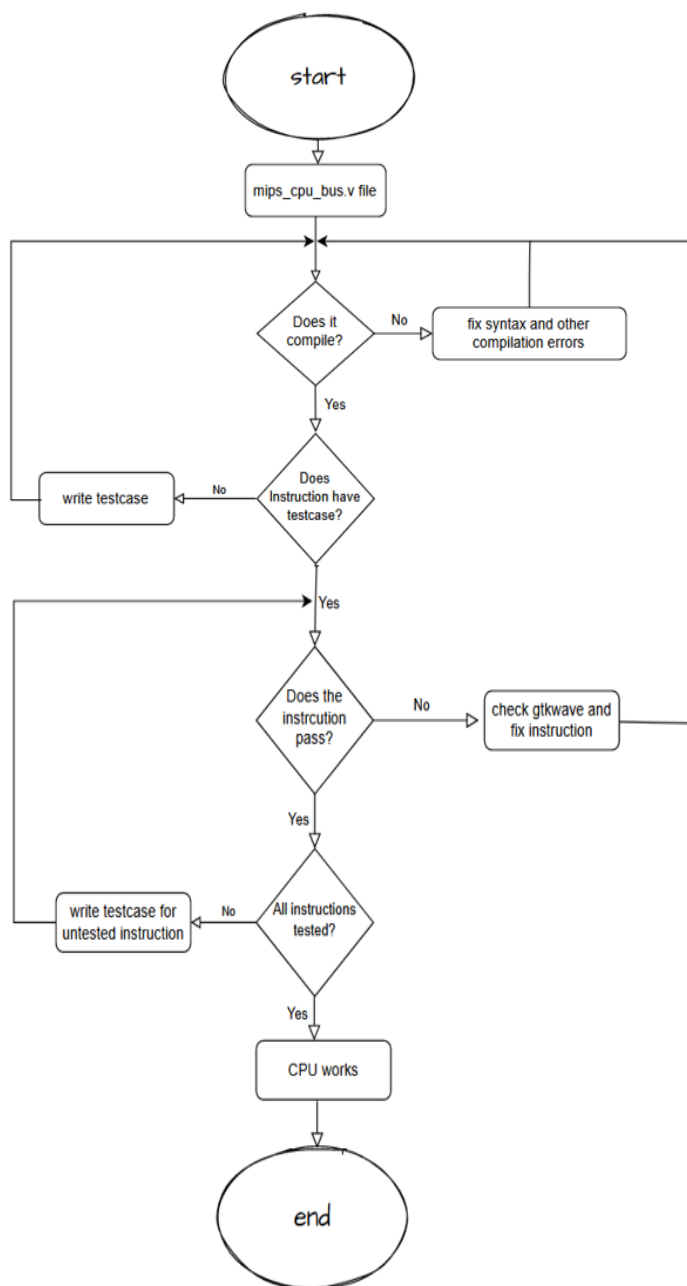


Figure 2: Flowchart showing testing method of instructions

Data and instructions were manually typed into the RAM in Hex inside the config files as shown in Figure 3. Outputs were checked using GTKWAVE and by asserting memory locations and register v0.

Figure 2 describes the testing approach. “Pass” refers to the test case (if the assertions, data and instructions are correct in the config files) and to the functionality of the instruction in the CPU.

LW and SW were tested first since they form the basis of the test cases for each instruction. At reset, all register values are 0. Because LW and SW are memory-based instructions, testing them individually first, then together, allowed to identify timing issues and how the different components interact.

JR \$zero is also tested because, after every instruction, JR \$zero is used to halt the CPU. This must be functional to ensure that test cases only fail due to the instruction itself and not the jump instruction.

By testing individual instructions, independent logical errors were easily found and eliminated. Multiplication and division instructions required MTHI and MTLO to store results in the HI and LO registers. To check whether those values are correct, MFHI and MFLO were also used to move the values out of HI and LO.

To convert the instructions from Assembly to Hex, the MIPS instruction converter was used [3].

```

CONFIG
id:1
name:s11

DATA
00
00
00
00
CF
FF
FF
FC
|
INSTR
8C
03
00
08

ASSERTION
assert(word0==32'h00000000);
assert(word1==32'hCFFFFFFC);
assert(word2==32'h00000004);
assert(word3==32'h00000000);
assert(word4==32'h00000000);
assert(word5==32'h00000000);
assert(word6==32'h00000000);
assert(word7==32'h00000000);
assert(register_v0==32'hCFFFFFFC);

COMMENT
lw $3, 0x8($0) (8C030008)
lw $1, 0x4($0) (8C010004)
sll $2, $1, 0 (00011000)
jr zero (00000008)
nop
  
```

Figure 3: Config File for SLL (just an example, actual data and instructions are not correct in this image)

Area and Timing Summary (Quartus)

Intel Quartus was used to determine the area and clock frequency of the CPU when synthesised into an Intel Cyclone IV E FPGA.

Area summary

Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	mips_cpu_bus_analysis
Top-level Entity Name	mips_cpu_bus_analysis
Family	Cyclone IV E
Total logic elements	2,824
Total registers	1155
Total pins	138
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0

Figure 4: Intel Quartus Area Summary obtained from fitter analysis.

As seen in Figure 4, the CPU uses 2824 logic elements out of 6227, (32%). By having little functional decomposition, a small number of modules, and good design decisions, this value is low. Good design decisions include few redundant signals. Most logic has been defined in the top-level sheet (the only exceptions being the FSM, control logic and registers).

Because the area needed to synthesise the CPU into the FPGA is small, the cost to physically implement it will be cheap. An example of a cost, that is very similar to the CPU described in this report is Intel's EP4CE6F17C6 (this uses 32% of the logic units but, the IO pins use 77%). In mips_cpu_bus.v, there is no separate ALU block, meaning there is less overhead. This explains why such a little proportion of the FPGA was used.

Timing Summary

Slow 1200mV 85C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	121.14 MHz	121.14 MHz	clk
2	131.79 MHz	131.79 MHz	waitrequest

Figure 5: Lower estimate for frequency at 85°C (1.2V)

The CPU had a frequency of 121.14MHz under the worst time-case scenario (conditions of 85 °C and 1.2V). This relatively high clock speed is explained by the fast nature of the MIPS architecture [5], the little use of redundant logic signals that prevented the formation of an excessively long critical path. These are best-effort timing results based on client provided information and can be tightened up once detailed platform timing is available.

Overall, the CPU performs relatively well when synthesised into an FPGA. This CPU has been designed for maximum performance in a Verilog Simulation, and it satisfies the highest possible CPI for a non-pipelined CPU. As the CPU avoids real-world performance checks, each instruction has been implemented individually. For example, addition and subtraction could have been implemented with one full-adder block. But for code legibility, it was done separately, increasing area and critical path.

The MIPS ISA specification is attached below [6].

References

- [1] Koen, 2019, Architecting For the -ilities, <https://towardsdatascience.com/architecting-for-the-ilities-6fae9d00bf6b> [online]
- [2] Weat, 2021, MIPS Store Word (sw) vs. Load Word (lw), <https://www.alpharithms.com/mips-store-word-sw-vs-load-word-lw-475521/> [online]
- [3] Aaron, 2020, MIPS Instruction Converter, <http://mipsconverter.com/instruction.html?optradio=on#> [online]
- [4] Mouser, 2021, Mouser Electronics <https://www.mouser.co.uk/ProductDetail/Intel-Altera/EP4CE6F17C6?qs=jblrfmjbeiGm%2FYQ7QYSfGg%3D%3D> [online]
- [5] Wikipedia, 2021, MIPS Architecture, https://en.wikipedia.org/wiki/MIPS_architecture [online]
- [6] Price, 1995, MIPS IV Instruction set <https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf> [online]