MIPS CPU Data Sheet – Team 4

Ziyad Ansari, Luca Chammah, Jian Fu Eng, Weihan Huang, Derek Lai, Chris Myers

December 2021

1 Overview

An Avalon bus interface [1] compatible 32-bit big-endian MIPS1 architecture [4] CPU was designed and implemented in SystemVerilog. Initially, a five-instruction Harvard architecture CPU was built as a foundation. The CPU was then expanded to 42 instructions before making the transition from two-cycle Harvard to Bus architecture.

Figure 1 portrays the block diagram of the Bus CPU:

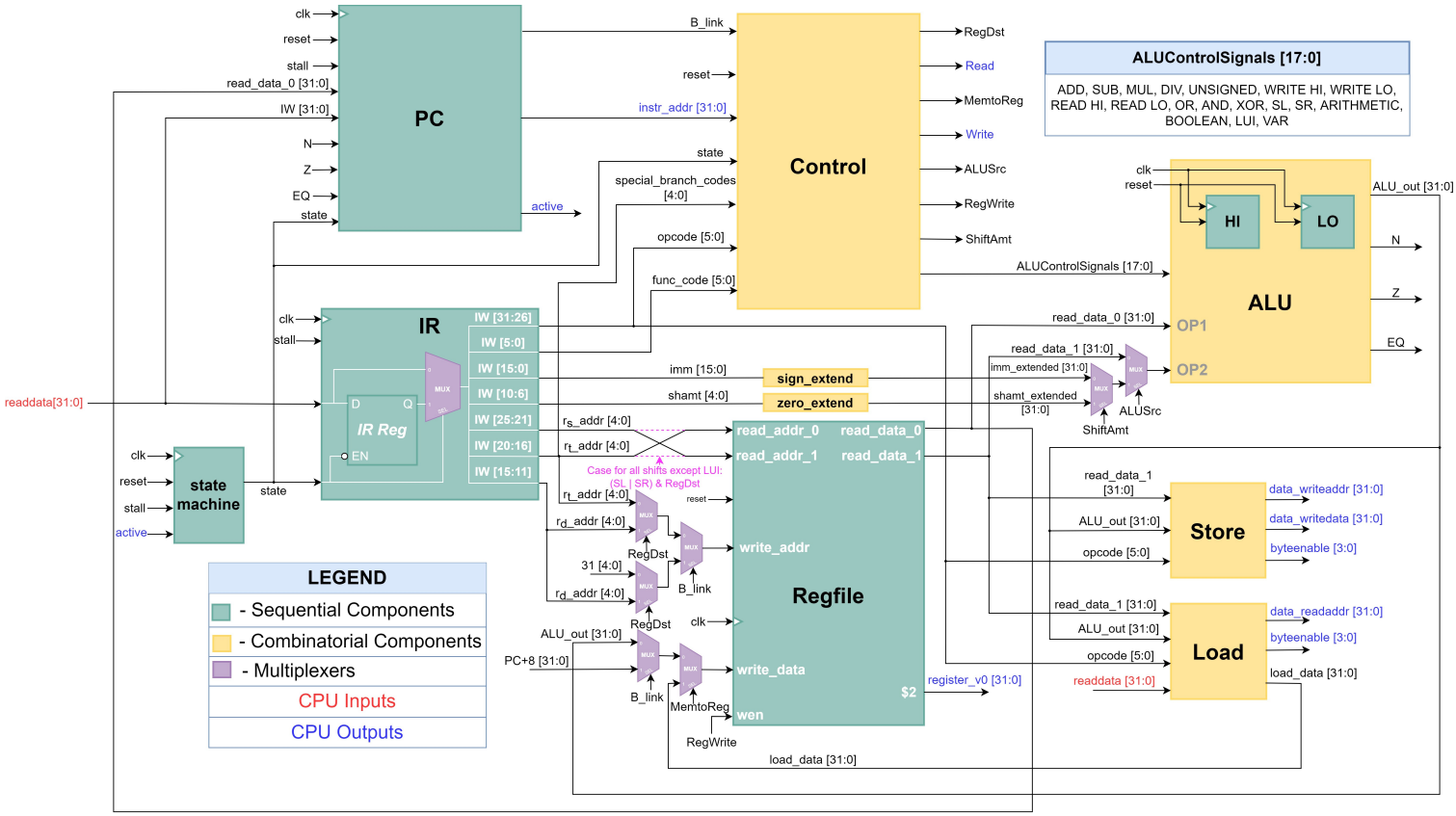

Figure 1: MIPS CPU Block Diagram

2 Microarchitecture

2.1 Design Decisions

1. **CPU Architecture:** The first CPU built was a single-cycle Harvard CPU. This later became two-cycle in order to facilitate the instruction `FETCH` cycle that the Intel Avalon Bus required.

Blocks used in the Harvard architecture were designed to enable ease of transition to the bus architecture. Multiplexers were then added to select the appropriate address to be fed to the Intel Avalon Bus.

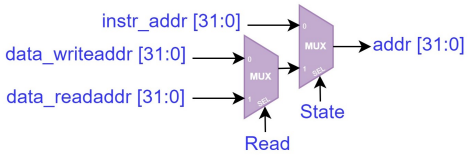

Figure 2: Bus Address Selection

2. **Harvard to Avalon Bus Modifications:**
   - To handle the random `waitrequest` input from the Bus, the state machine was modified to include stalls in order to pause the CPU. During stalls, the CPU preserves its state until the end of the stall.
   - Since `readdata` is only valid in the cycle after `FETCH`, a register was added in the IR to hold the `instruction_word` during the `EXEC` cycle. Additional logic was also added to ensure that load instructions only writeback to the RegFile if the `readdata` is valid.
   - Load/store blocks handle generation of a word-aligned address as well as the correct `byteenable`, `data_writedata` and `load_data` for a given load/store instruction. They also perform appropriate bit-slicing operations.
   - To make the CPU compatible with the Avalon Bus interface, the endianness of `readdata` and `writedata` were reversed at the top-level module of the CPU.

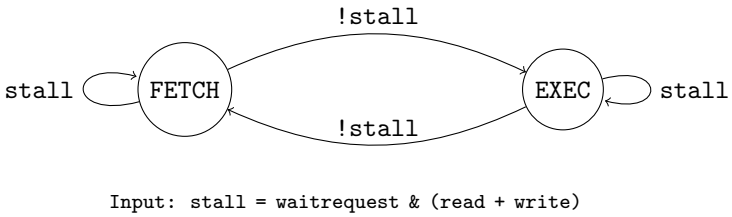

Figure 3: Bus Architecture State Machine

3. **PC:** The PC consists of two blocks:
   (a) Branch control block: This block takes in the `instruction_word` and the `N, Z, EQ` flags to determine if a jump is to be executed. The block also generates an output `B_link`, which triggers the writeback of `PC + 8` into the designated link register. Handling all the control logic for branch instructions within this block allows for easier debugging of PC-modifying instructions.
   (b) PC block: This block computes the next instruction address to be executed using the output from the branch control block. It also handles the execution of address `0x0` by halting the CPU accordingly and dropping the CPU `active` output. To implement the branch delay slot, two registers hold the contents of the previous `instruction_word` and `read_data_0` (the contents of `rs`). This allows the PC block to compute the address to jump to following the branch delay slot.
4. **Control Block:** The ALU and MUX control lines were unified into a single Control Block. This is an instruction decoder whose inputs are different fields of the `instruction_word` and whose outputs are the correct control lines for that instruction.

3 Testing

To construct a comprehensive testing suite, a custom MIPS assembler was developed using Python to run MIPS assembly on the CPU. Four different tests were developed as shown in Figure 4 to vary the `waitrequest` signal in order to simulate the many possibilities of unpredictable memory

access by other peripherals in a real computer. To determine a pass or fail for a given instruction, the top level test bench waits for the negative edge of the CPU active output and calls `$assert()` to determine if register `$v0` is equal to an expected value.

`$fatal()` statements were also added to ensure that the CPU tested is compliant with the Avalon Bus Interface Specifications.

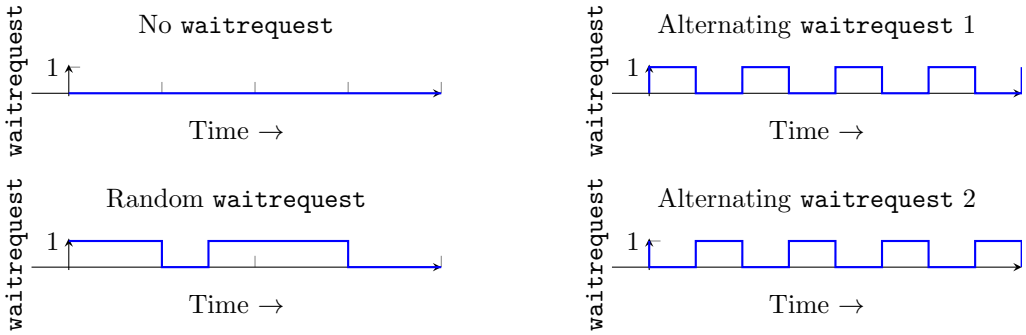

Figure 4: Different `waitrequest` test cases

3.1 Approach

The testing pipeline of the test suites is shown in Figure 5 below.

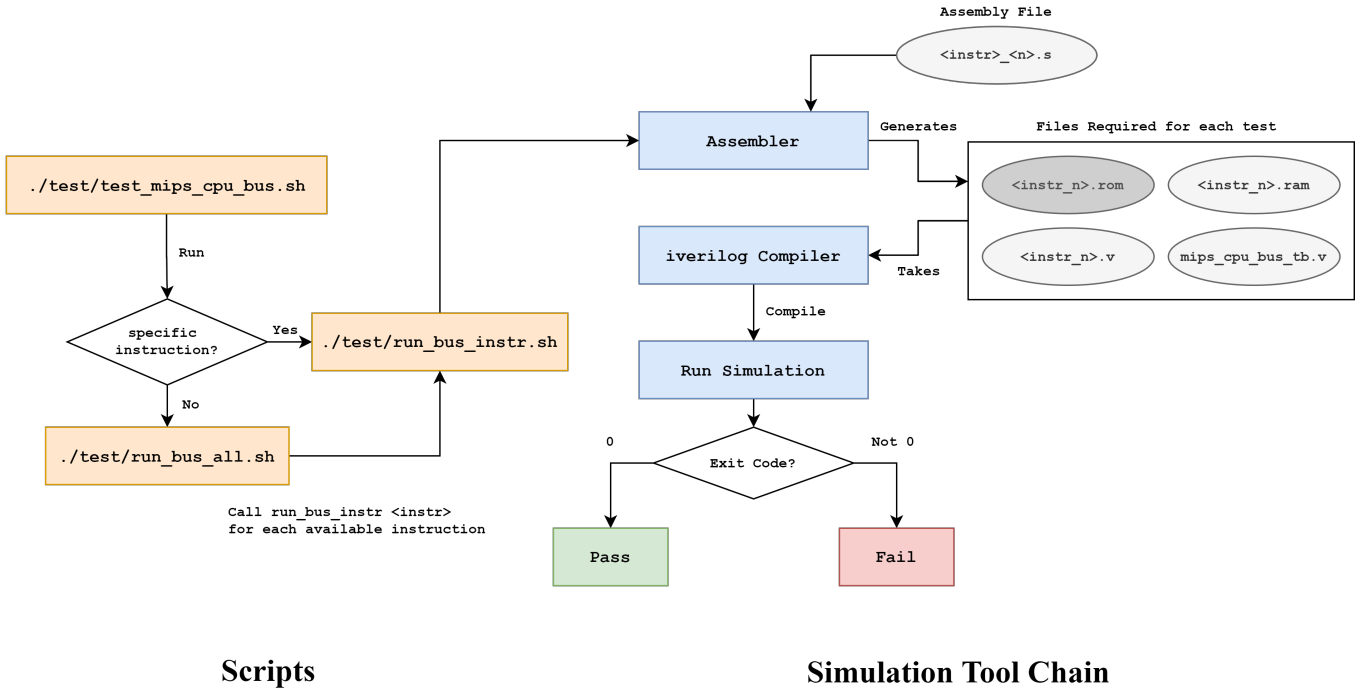

Figure 5: Test Suite Flow Chart

3.2 Assembler & Tools

A configurable two-pass assembler was built to improve automation of the testing process. The ability to generate Verilog files was added to the assembler, making the asserted result configurable from within the assembly file. This meant that only an assembly file had to be written and a single script had to be executed to test whether an instruction works. The assembler would proceed to generate the required files shown in Figure 5. Then the scripts would compile, simulate and report the results to the console.

3.3 Scripts

`test_mips_cpu_bus`	This script is used as a wrapper so that the test benches complied to the specification provided.
`run_bus_instr`	Requires a path to the top level module CPU, and an instruction to test. This script handles the entire compilation, simulation, testing tool chain.
`run_bus_all`	Short script which handles finding all tests which have been written and calls `./run_bus_instr` on each instruction it finds.

4 Evaluation

4.1 Quartus Performance

The CPU's performance was evaluated using Intel's Quartus Prime Lite tool [2], "Cyclone IV E 'Auto'" variant.

Metric	Measurement
Maximum Frequency	7.62 MHz
Total logic elements	8923/15498 (58%)
Total registers	1186
Total pins	138 / 344

Table 1: Quartus Flow Summary

Given the limited time for optimisation and focus on functionality, the CPU performs up to its expectations. However, possible optimisations can be made for both area and timing as listed below.

4.2 Optimisation for Timing

1. **Critical Path:** The longest combinatorial path limiting the clock frequency of the CPU is from the state machine through the control logic to the ALU. If pipelining was implemented, the critical path will be shortened to enable a higher clock-rate. However, pipelining creates data and control hazards which can impact efficiency as stalls will have to be inserted to deal with these hazards.
2. **Forwarding & Branch Prediction Units:** To further increase the IPS of the CPU, forwarding [3] and branch prediction [5] units can be included in our CPU. This reduces the impact of data and control hazards on the CPU.
3. **CPI:** Implement a cache that contains blocks of RAM so that `waitrequest` will not result in as many stalls.

4.3 Optimisation for Area

1. **ALU:** Remove the comparator in the ALU and use subtraction operations to set `N, Z` flags instead.
2. **PC:** To implement the branch delay slot, the PC consists of 2 32-bit registers, containing `read_data_0_prev` and `instruction_word_prev` to compute the branch address in the branch delay slot. Instead, the branch address can be computed in the `EXEC` of the branch instruction and can be stored in a single 32 bit register for access in the branch delay slot.
3. **Control:** Look for better patterns in MIPS opcode and function code to reduce the number of control outputs.

References

[1] Intel. "Avalon® Interface Specifications". In: (May 2021). URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf.

[2] Intel. "Quartus Prime Lite Edition". In: (). URL: https://fpgasoftware.intel.com/?edition=lite.

[3] Univeristy of Iowa. "Pipelining". In: (). Referenced for Data Forwarding. URL: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiMv7K4nOv0AhUQ0EAHcC2DmgQFnoECB8QAQ&url=http%5C%3A%5C%2F%5C%2Fhomepage.divms.uiowa.edu%5C%2F~ghosh%5C%2F2-9-06.pdf&usg=AOvVaw31Ciyagon5Qnwy1YBOp57U.

[4] Charles Price. "MIPS IV Instruction Set". In: (Sept. 1995). URL: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjajf6vo-j0AhXNMMAKHft2CVUQFnoECAQQAQ&url=https%5C%3A%5C%2F%5C%2Fwww.cs.cmu.edu%5C%2Fafs%5C%2Fcs%5C%2Facademic%5C%2Fclass%5C%2F15740-f97%5C%2Fpublic%5C%2Fdoc%5C%2Fmips-isa.pdf&usg=AOvVaw1HyNEe_cE0OesaBp179DL9.

[5] Techopedia. "Branch Prediction". In: (). URL: https://www.techopedia.com/definition/18062/branch-prediction.