Team 07: ELEC50010 Instr. Arch & Comp. : CPU Coursework

Contributors: Miguel Bragança, Liam Gallagher, Julio Castillejo Motta, Panagiotis Doumas

Architecture overview

This is a MIPS compatible, 32-bit, non-pipelined processor which communicates with memory by controlling an Avalon bus interface.

This bus-based interface means there is only one bus for both the program instructions and stack/data. The bus is split into 4 channels, each channel transmits an 8-bit number which represents a byte from a word being read from memory or a byte from a word that is being written to memory. These 4 channels are managed by the 4-bit signal “byteenable”. Each bit of this signal controls whether a channel is being used. “byteenable” is always 4’b1111 when reading an instruction or loading/storing a word but it is changed depending on loading/storing bytes instructions. Also, since byte addressing and “byteenable” are being used all addresses that are loaded into memory must be word aligned.

Furthermore, in accordance with the Avalon bus interface, a “waitrequest” flag must be considered. If this flag is low, then memory is ready to make a word sized transaction. Otherwise, the CPU must remain in its current state (stall cycle) and wait for “waitrequest” to be set to low. The “waitrequest” flag is set by memory.

At address “0x0000_0000” the CPU comes to a halt and the value is in register $v0 will be the output of the CPU and the address “0xBFC0_0000” is used for the reset vector, which is where the instruction region would start.

There are 32 registers in the register file:

- Register $0 always contains 0.
- Register $31 contains the return address for branches or jumps.
- Register $2 represents $v0 which is the output register.
- All other registers are General Purpose Registers (GPR).

The CPU makes use of the load/store architecture for memory instructions. The load instructions consume the most cycles to execute successfully. They need the Fetch state to update the memory address with the PC, they also need the Execute state to decode the instruction and do the address calculation for memory, finally it needs the Memory state to process the memory contents and write to the correct registers.

All instructions use the Fetch and the Execute states.

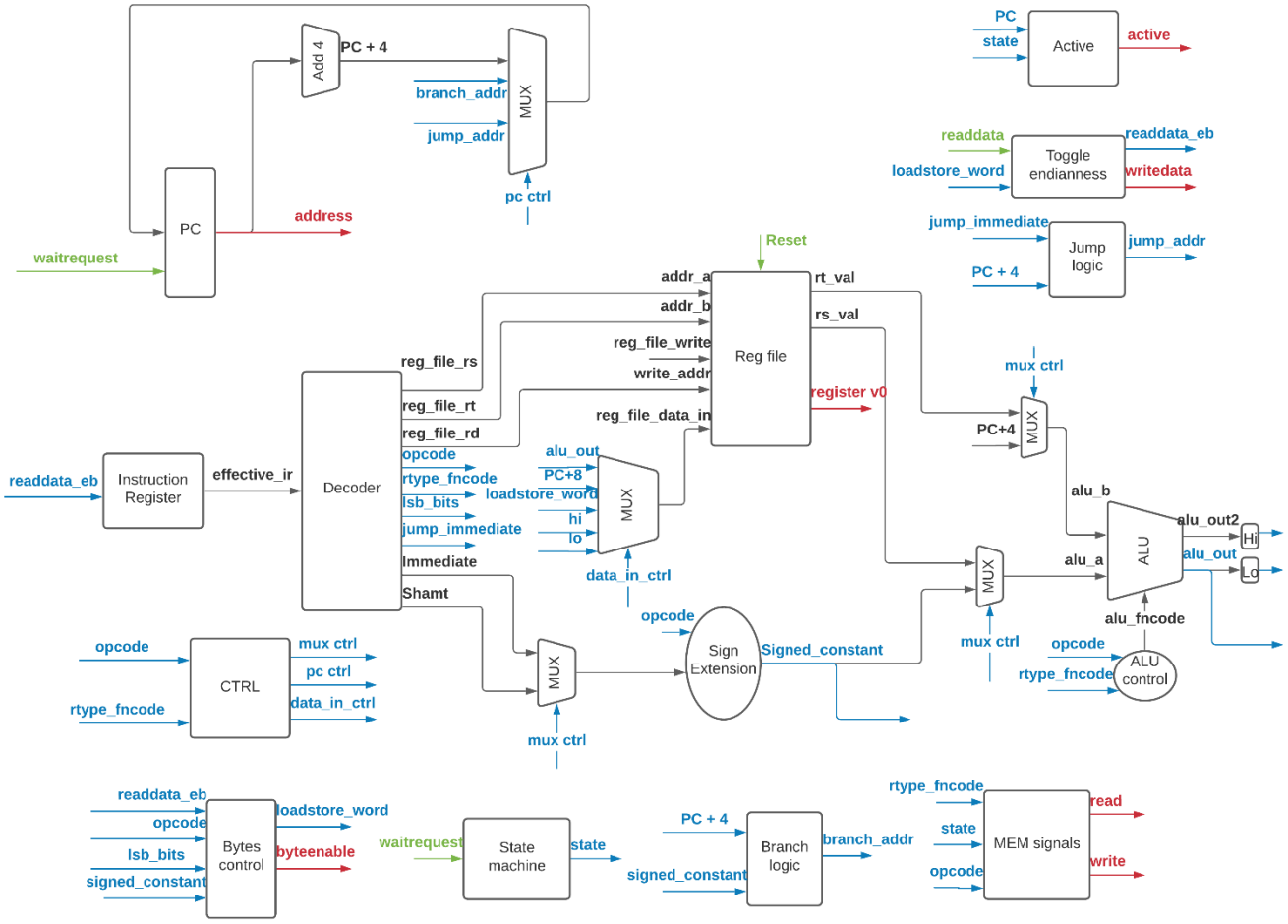


Figure 1: *High-level diagram of the CPU (with outputs being signaled in red, inputs in green and internal signals in blue). All sequential components are clocked.*

Design decisions:

In order to integrate DIV, DIVU, MULT and MULTU MIPS instructions we needed to generate two 32-bit values from the ALU. Thus, we had to choose between using an additional cycle to compute the second 32-bit output or adding an additional 32-bit output to the ALU. The latter choice makes the design of these instructions easier. As such, we chose two outputs from the ALU.

The usage of the *SystemVerilog*'s built-in arithmetic operators helped us simplify our code, keeping it clean and concise. This was of utmost importance due to the short time given to develop the CPU. However, a tradeoff to acknowledge is the non-optimized performance of the calculations given that combinatorial multiply and divide will likely add to the critical path greatly and add considerable area to the design.

The PC (Program Counter) branch slot implementation was simple, yet effective. It used a dedicated register, "pc_branch", allowing us to store the computed value of the branch address every time the program comes across a branching instruction. If the branch should be taken, it sets the "is_branch_delay"

flag. During the next instruction (which is in the branch delay slot), the PC is updated to the value stored in “pc_branch”. If the branch is not taken, the program continues running sequentially.

We chose not to pipeline the MIPS CPU as it was not required as a part of this assessed project for *Instruction Architectures and Compilers*. Due to the short timeframe given and difficulty of , the single purpose of this work was to create a fully functional MIPS CPU.

Testing:

Assembly

The CPU was tested by running assembly programs of varying complexity on the CPU. These were designed to verify the correct execution of a single MIPS I instruction. The testcases were aimed to trigger edge-cases by using boundary data. **Figure 2** shows a simple case for JR.

```
1      .text
2      .align  2
3      .globl  main
4      .ent    main
5  main:
6      .set    noreorder
7      jr $0
8  .end    main
```

Figure 2: *“jr” testcase*

Figure 2 is an assembly program that jumps to $0 which contains the halt address. All testcases contain an entry procedure named main in the .text section of the binary. ”.set noreorder” is present in each testcase to ensure the assembler does not automatically place a ”NOP” after an instruction which would cause a branch delay. This allowed branch delays to be tested for.

Testbench

The testbench creates an instance of the CPU module as well as two RAM modules, one for program memory and the other for the stack region. Each RAM had a capacity of “4096x4(RAM_SIZE)” byte words which is enough for simple testcase programs. The entire 4GB of addressable memory was not simulated as it would slow the simulation and take too much memory from the host system. The test bench had two compile-time parameters passed to it:

- The filename containing the assembled binary in a format readable by the testbench.
- The expected value of the General Purpose Register (GPR) $v0 at the end of program execution (when active is low).

With these, the testbench can load the binary into the program memory ram, run the simulation and verify that the value of $v0 output from the CPU matches its expected value. The testbench maps the output address of the CPU and its read write signals to the RAMs. If the address is in the program region, “0xBFC00000” to “0xBFC00000 + RAM_SIZE”, the read and write signals are propagated to the program region RAM. This is similarly so for the stack RAM region if the address is between “0x00000000” and “0x0000000 + RAM_SIZE”. **Figure 3** shows a high-level view of the testbench’s architecture with some signals omitted for clarity.

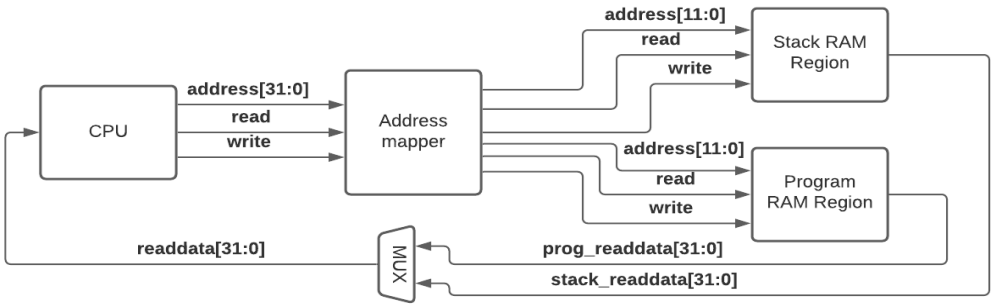
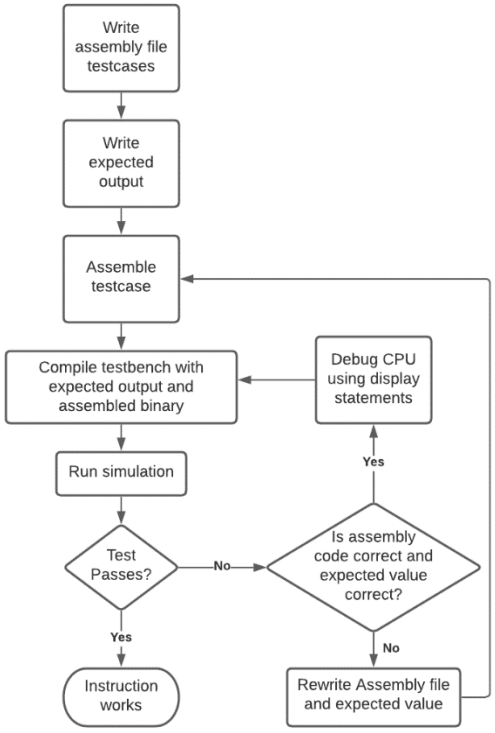


Figure 3: *High-level diagram of the testbench.*

Testing approach:



Tests were written in assembly and automatically assembled with a bash script using the ’mips-linux-gnu-gcc’ assembler. The script ensured the .text section of the executable was placed at address ”0xBFC00000”. This meant J and JAL instructions would be assembled correctly. This is because the assembler must be aware of the location of each instruction in order to calculate the jump address properly. Further, the testcase was assembled to little endian format so the testbench did not have to consider endianness. 'mips-linux-gnu-objcopy’ was used to remove section information from the executable. ’xxd’, a hex viewer, was used to view the bytes of the binary and put them in a file. Another bash script was made to compile the CPU and testbench with the expected output and assembled binary. The simulation is run, and the script reports a ”Pass” or a ”Fail” for each testcase. **Figure 4** details our testing and debugging process.

Figure 4: *Testing process flowchart.*