MIPS Data Sheet: Team 02

Overall Architecture of CPU

Summary of Architecture

We implemented the MIPS ISA (Price, C. (1995, September). MIPS IV Instruction Set (Version 3.2) [Program documentation]) with a bus architecture that is compliant with Intel's Avalon bus interface as to make it possible to synthesize our CPU onto an FPGA.

The two main goals we had in designing this CPU were maximizing understandability (with the hopes of making further additions as well as open sourcing) and minimizing latency (as we wanted to make a good CPU).

Because we were constrained by assessment specifications to make an unpipelined CPU, we decided to make a 3-cycle CPU to maximize the number of instructions per cycle by minimizing the number of cycles. Furthermore, we used shortcuts in Verilog which utilize large area but small time such as r = a*b for multiplication – the Verilog synthesis libraries choose the most optimized hardware for the application (Nelson, V. P. (n.d.). Lecture presented at ELEC 4200 in Auburn University, Alabama.). From Patterson and Hennessy (CPU Time = Instruction / Program * IPC^-1 * Clock Rate^-1): the only other metric than IPC the system architecture can improve is clock rate and that happens on more of an FPGA design level (Hennessy, J. L., & Patterson, D. A. (2011). Computer Architecture: A Quantitative Approach. Amsterdam: Elsevier Morgan Kaufmann.).

To maximize understandability, we avoided the use of confusing and potentially incorrect control logic based off analysing singular bits of machine code and instead opted for having a multitude of control signals for each instruction from a centralized decoder. And all enables (memory write enables, register read enables, or what have you) are dependent on logic based off the current clock cycle in the CPU. Because of this, we have scalability; adding instructions will be quite trivial. Moreover, it will be quite easy to change instructions to the users' preferences – because we have a separate control signal for each instruction, it is easy to create and implement our own new instructions bespoke to the application. Furthermore, open-source developers who look at our CPU will be able to read, easily understand our architecture, design methodologies, and make changes.

Diagram of Architecture

Below is a systems-level representation of our architecture which gives some intuition for the flow of data.

- The blue arrows represent the current cycle or clock being inputted into each respective block
- The red arrows represent the necessary control inputs (formatted as instruction codes) into each block

- The green arrows represent the program counter behaviour and how the next instruction data travels

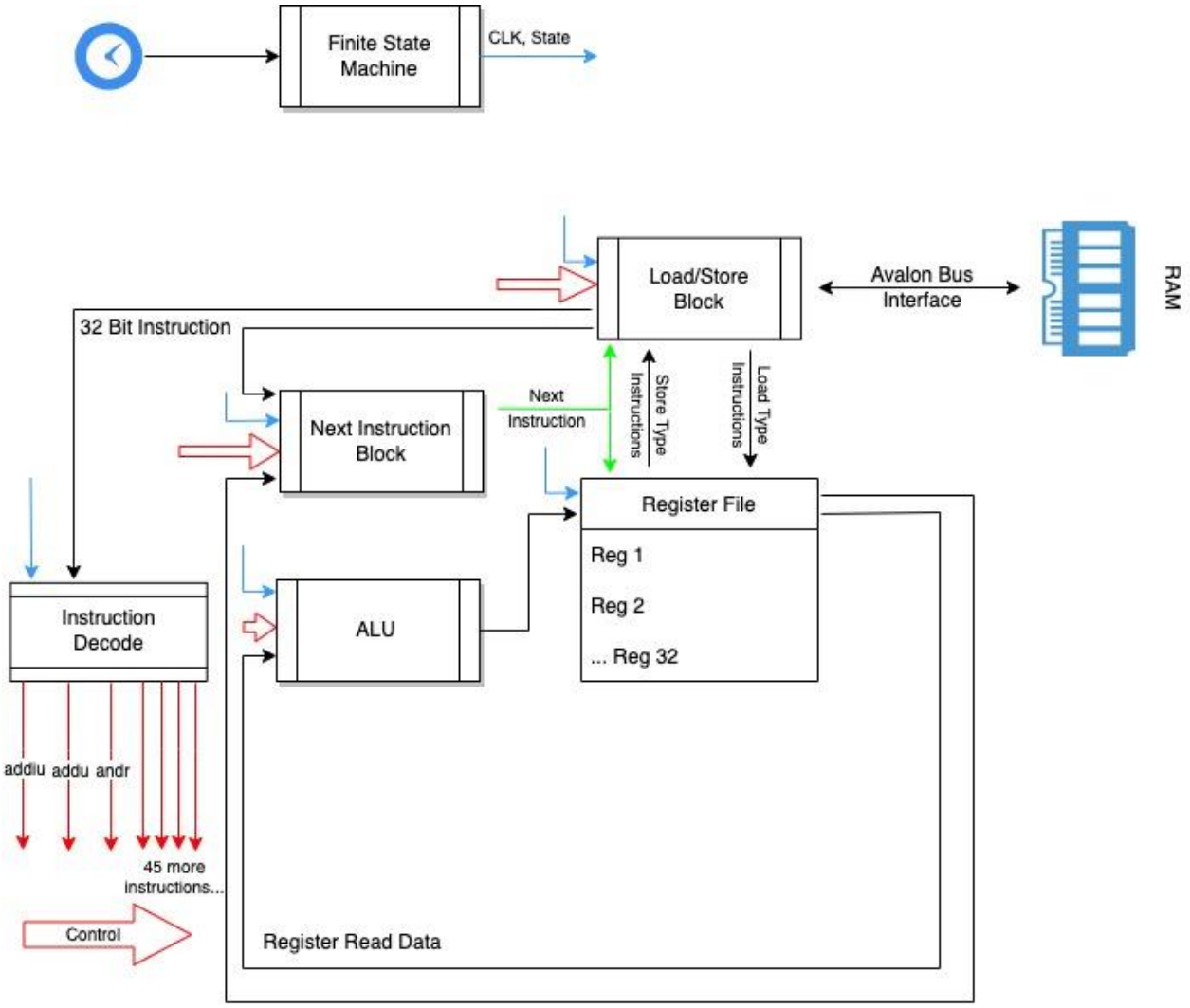

Figure 1 – A Systems Level Architecture of our MIPS CPU

Summary of Each Module

ALU

The ALU implements all arithmetic and logical instructions in the CPU. The inputs include immediate constant for I-type instructions, and 2 register read inputs for both R-type and I-type instructions as well as the signed versions of those read inputs. It also contains each ALU instruction as an input – only one control signal should be 1 at any given time. It also contains a HI and LO register which are used in multiply instructions as well as MTHI and MTLO instructions. It operates using an immediate extension module (which extends immediate modules), multiplexers selecting each control signal, and logic which produces the necessary ALU output. Furthermore, when the ALU calculates an output, it turns on the register file write enable to write the output value into one of the 32 registers.

Decoder

The instruction decode block takes inputs for the machine code of each instruction, as well as wait requests. The wait requests call the CPU to do a stall if a load or store instruction requires it. Aside from these stalls, the decode block simply decodes each instruction into its instruction code (for example ADDIU, or J) as well as its operands (including shift amounts) – all as control signals.

Next Instruction

Next Instruction was a module that determined the value of the Program Counter register was based on the current value of the Program Counter and the incoming instruction. It fundamentally is a modified counter, counting by 4 (as each instruction is 4 bytes), and only deviated on instructions that had a branch or a jump to another value.
These branch instructions are always conditional, meaning that to branch to the location you want a condition must be met regarding the values of the operands rs and rt, meanwhile jump instructions in this ISA always jumped. Since branch instructions allow for conditions, this allows you to implement loops. For instructions which link, it gives the next instruction to a register, and it provides the next instruction to the load/store block.
Next Instruction also contains the finite state machine which determines which cycle the CPU is currently in. Our CPU is a 3 cycle CPU as even though not all instructions require 3 cycles, it is easier to implement by just having the 2 cycle instructions do nothing for the third cycle.

Load Store

The load store block interfaces with memory and registers so that other blocks do not have to. Like the name suggests, the block does the heavy lifting for load and store instructions, and it also allows for byte addressing, which is used for many load and store instructions to load certain bytes.

RAM

The RAM has 32-bit memory words and is not strictly included in our CPU, it exists as a file purely for testing purposes. It is a single read single write RAM and it has an enable to decide when to write the 32-bit write data.

Register File

It is a 32-bit register file with 32 addresses as outlined in the MIPS ISA. It takes control inputs from the next instruction block, the ALU block as well as the load/store block for write enablement; furthermore, it takes input byte enable data for load byte operations (for example LB or LBU). Finally, it outputs the two registers that are being read at any given instruction.

Testing Suite

General Approach for Testing

Overall, we use a comprehensive testing method that goes through each instruction in the MIPS Instruction Set Architecture by having incrementally more complex testbenches. This exhaustively tests the outputs at the end of every testbench to ensure the correctness of the CPU. This is done by having over 100 testbenches which we run sequentially using a bash script and give information automatically as to which testbenches fail.

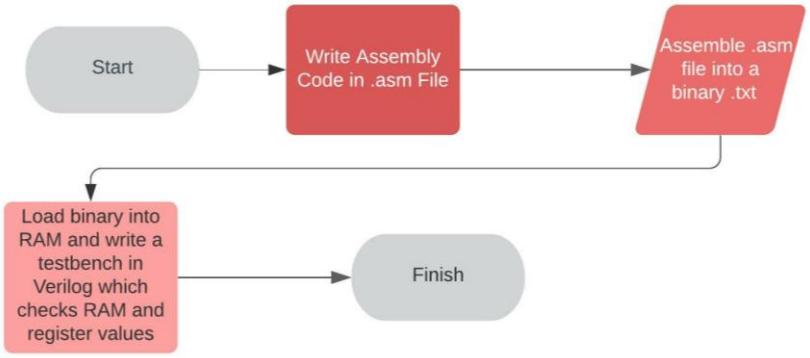

Figure 2: Testbench Generation Flowchart

For each testbench, the above diagram depicts the methodology that we use to engineer each testbench.

Flow-chart describing testing flow/approach

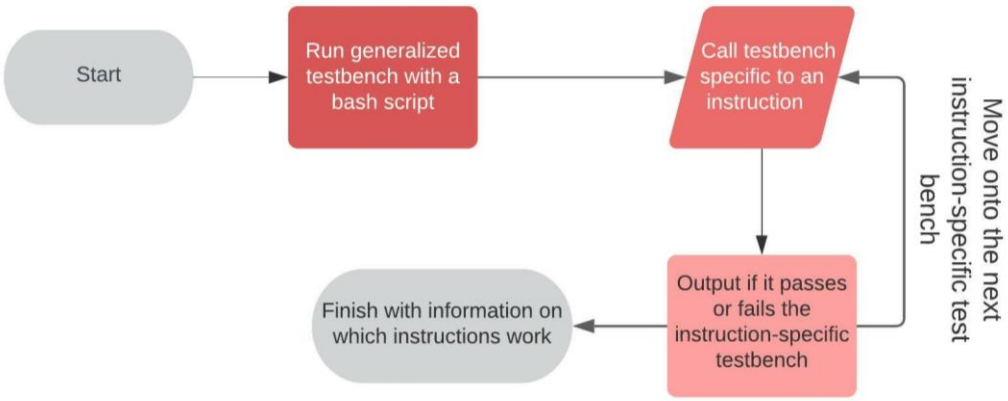

Figure 3: Generalized Testbench

In terms of testing each instruction, we have created a few testbenches specific to each instruction of various length and we check the terminal value (as different implementations of the CPU can use different numbers of cycles, so the only point we can verify is the final one) of registers and RAM to verify correctness by checking our answers with by hand and by emulator. Overall, by having such an extensive suite of testbenches, we can have confidence that the CPU indeed works correctly, and our test suite is applicable for other CPUs as well.

References

Hennessy, J. L., & Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Amsterdam: Elsevier Morgan Kaufmann.

Nelson, V. P. (n.d.). Lecture presented at ELEC 4200 in Auburn University, Alabama. Retrieved from https://www.eng.auburn.edu/~nelsovp/courses/elec4200/VHDL/Verilog_Overview_4200.pdf Lecture Slides

Price, C. (1995, September). MIPS IV Instruction Set (Version 3.2) [Program documentation]. Retrieved from https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf