Lab 3

Bryan Werth, Joseph Lee, Serena Chen

CPU Architecture

The single cycle CPU is controlled by a program counter (PC) indexing instruction memory, which outputs a 32-bit instruction word. Instruction Decoder separates the 32-bit instruction word into usable parts which are used to inform other blocks in the CPU. The CPU controller uses the opcode and function code from the instruction decoder to generate the correct control signals. Register File serves as local memory, storing the results of operations performed elsewhere in the CPU and feeding stored data into the CPU as inputs. Data Memory and the ALU on the output of Register File are used to perform operations dictated by the instruction. The output of this part of the CPU is optionally stored into local memory. The program counter, which is just a D flip-flop, is updated from three possible values produced using the previous PC value as well as an optional immediate or address from the instruction. Branch if not equal is implemented using the zero output of the main arithmetic ALU on the output of the Register File. If the zero output is low when bne is the selected operation, the mux with the zero output driving the select input (bottom left) passes the jump address as an output to inform the next program counter value.

For the RTL, we used the MIPS instruction reference[1]. The RTL for each implemented instruction is reflected in our design, but we made one change to the JAL instruction. We found that storing PC+8 in the jump register caused the program to skip an instruction when it jumped back to the address in jump register. We were not sure why the RTL required us to store PC+8, so to match the behavior of the Mars assembler, we stored PC+4 in the jump register.

[1] http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

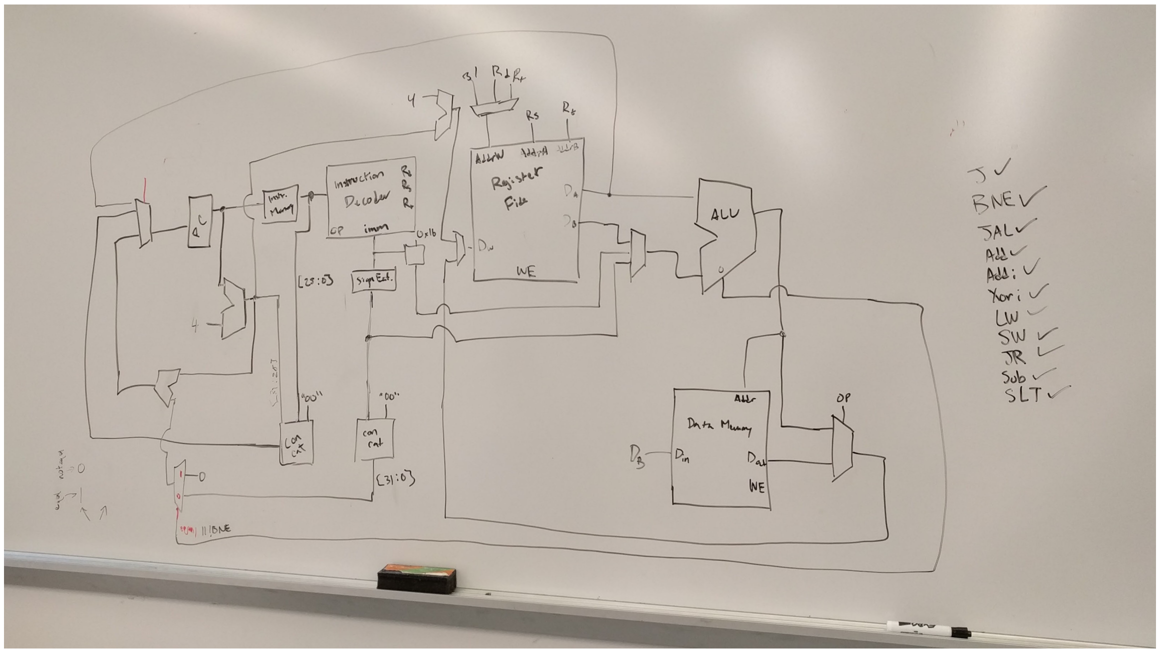

Figure 1: The block diagram for our single cycle CPU.

Test Cases

For the modules we added from other labs, we simply copied over the old test benches. One exception is for *datamemory.v* and *instruction_memory.v*. Our previous lab and Ben's notes did not include a test bench, and we wanted to make sure that the memory was being written to and read from correctly, so we wrote our own test bench.Our test strategy for these modules was pretty straightforward. We essentially just tried writing and reading from different addresses and ensured that writeEnable was functioning properly by providing input without asserting writeEnable and then performing a read to make sure that the data at that address was not written to.

We wrote test benches for the new files introduced in the lab:

For the instruction decoder, we simply checked the 3 instruction types, and made sure that the decoder broke down the instruction properly and had the correct data in the relevant fields.

For the sign extender, we had 2 test cases - one where the sign was 1, and the other where the sign was 0. The rest of the digits were random. We ensured that the random string was consistent and that the sign was extended properly.

For the CPU itself, we wrote a program that woud just make the cpu, and print out the registers after some large amount of steps. The registers would have to be manually inspected (we could have automated that but it seemed silly since we were switching the programs that were being run on the cpu and there were 32 registers to check).

Performance/Area Analysis

Unfortunately due to lack of time, we were not able to really explore performance and area extensively. We did try running Vivado synthesis to get an estimate, but the numbers that the synthesis reported were unreasonable. Part of the problem is that our implementation of instruction memory is non-synthesizable. We tried synthesizing just the CPUcontroller which returned more reasonable results, but still did not seem quite right. It said that our design would utilize 8 LUTs and around 100 IO pins.

Work Plan Reflection

Overall, we were fairly optimistic with our work timeline. We initially planned on finishing the single cycle CPU for the Friday before the lab was due, but we did not finish our first pass implementation until the next Tuesday. We did finish the test cases, assembly assembly test file for Monday, but that meant we were two days late on the test cases. Debugging the first pass implementation and polishing our final implementation also took longer than expected. We finished this the Monday after break. As such, we did not implement a pipelined CPU. In the future, we need to be more diligent about keeping to our deadlines while better understanding the significant time commitment required to debug a complicated low-level design.