Sam Barbosa

CS 2200

28 October 2023

# Creating A More Efficient LC-2200

The LC-2200 is a slow machine using, on average, around 6 cycles to compute a single instruction. According to the provided simulation, running the provided "pow.s" assembly file took (0x1B9B) 7,067 clock cycles. From this initial run, I felt like I could create a faster computer. This report documents my efforts to enhance the performance of this little computer.

Initially, I attempted a small improvement by adding a dual-ported register file with two buses. While this did make the LC-2200 slightly faster, the calculated speedup was only 1.22, which fell short of any significant improvement for extra credit. Deciding to pursue a more substantial enhancement, I opted to redesign the LC-2200 to incorporate a pipeline architecture. Although I anticipated the complexity of this implementation, I couldn't identify any other incremental changes to boost speed.

Firstly, I determined the necessary data paths for each instruction by iteratively adding datapaths as needed and registers based on their data requirements. For instance, ADD/NAND/OR all required similar datapaths as they write to registers and perform a single operation in the execute stage.

A challenge arose when introducing wires for branch instructions, which required computing A minus B and determining the PC value to jump to (if branching was necessary). Following the initial pipeline concept, the branch instruction would stall the pipeline to perform both steps. However, I resolved this issue by adding a second ALU to compute the new PC value, eliminating the need for stalling. A multiplexer was introduced to pass through the old PC value if no branch occurred and the new PC value if a branch occurred. Additionally, I ensured that opcodes not related to branches automatically signaled no branch. I incorporated functionality for JALR to always output a branch. I then configured the PC to either take its old value plus one or the calculated branch instruction, providing the LC-2200 with branch prediction functionality, which proved superior to waiting for the branch instruction to compute and load the correct instructions. The initial branch prediction, however, was simple, assuming no branching would occur.

After establishing all the necessary datapaths for instructions, I created state machines for each stage, opting for circuits instead of a ROM with microcode due to the perceived simplicity of each stage. Using the 4 bits of the opcode as input, I designed circuits to output the necessary control signals for selecting how data should travel down the datapath. For example, in the ID/RR stage, I needed control signals to choose which register to read from and output to A and B, as well as which register the instruction should eventually write to (if applicable).

Subsequently, I introduced data forwarding to the pipeline. This involved adding inputs for the outputs and registers of the EXE, MEM, and WB stages into the ID/RR stage. If a register that needed to be read matched any in the EXE, MEM, or WB stages, the read value would be replaced with the corresponding output. Determining the precedence of stages was challenging, but after some trial and error, I concluded that the most recent calculation, i.e., the EXE stage, should take precedence.

Therefore, if all inputted stages had the same register as a register that needed to be read, the output from the EXE stage would be used as it represented the most recent calculation.

In the end, running the "pow.s" assembly file on my pipelined LC-2200 took (0x064A) 1610 cycles. Using the speedup equation, speedup A over B = execution time on processor A/execution time on processor B, I found that the speedup from the old LC-2200 to the pipelined processor was 4.38, as 7,067/1,610 = 4.38. (Note: if one were to run my LC-2200, they should add the hex to the RAM in both the IF and MEM stages). This speedup shows that by pipelining the LC-2200 datapath, we can make it way more efficient.