Nicholas Jones

Jason Chin

Computer Architecture and Organization

4/16/2014

Final Project Report

This report describes the implementation and design of a single cycle MIPS processor with a reduced instruction set of add, addi, beq, j, lw, sw, and HLT (end the program). It should be known that neither Nick nor Jason had any prior experience with Verilog or computer architecture prior to the start of the project. Nick is a Computer Scientist, so while the coding processes were familiar in many ways, the unfamiliar non-sequential nature of the implementation, and the ability for modules to acquire data without being explicitly told to made the implementation much more difficult than previously expected. Jason is a mechanical engineer, so while he has also had a bit of experience programming, this was new to him as well.

In the initial stages of the project, we decided to use GitHub, a revision control system that would allow us to easily share code without meeting in person or passing emails back and forth. Additionally, throughout the project we communicated over email and google chat. Our next step was to figure out which parts of the CPU we already had, what we needed to make, and what we could modify. Luckily, there were only a few parts that needed to be built from scratch. After making the necessary additions and modifications, we began working on the CPU structure. This part mainly involved following the MIPS CPU diagram, which was provided with the project. For each of the parts in the diagram we created a corresponding module in the CPU. We then connected these together with the necessary wires. Designing the CPU module structurally—as compared to behaviorally—made things much easier to understand, since it resembled the diagram we were basing the design off. Finally, after putting all the pieces together we discovered that our timing was off. Since some modules in the processor rely on data from other parts, we needed to delay certain operations until all the data was available. For the most part, we were able to isolate the delays to the control module, since all other data paths relied on it to begin processing.

We implemented the two important control modules (control and aluCtrl) in very similar ways. For control we provided two inputs: the 6bit opcode and a 1 bit reset signal. The posedge of the reset sets all the control signals to their default states. The opcode is parsed using a case structure. For each type of instruction recognized we set the control values accordingly. If the opcode is not recognized then the control signal are all default to zero. For output, we gave it twelve output values—one for each control signal plus a stop signal controlled by the new HLT instruction. These control signals control the functionality for the rest of the CPU. aluCtrl functions in much the same way. We provide it the ALU opcode, an addI signal (Is the instruction addI?), and a 6bit instruction code and it uses a case structure to set the single 4bit output, operation. As in the control module, if the inputs do not match a known operation, the operation is set to zero.

The final product of our CPU is fully functional. It performs 100 instructions in 100 cycles, which results in a CPI of 1. The register contents are as described in section four of the project specification:

R0: 0x00000000	R8: 0x00000003	R16: 0x00000000	R24: 0x0000001B
R1: 0x00000000	R9: 0x00000006	R17: 0x00000000	R25: 0x00000024
R2: 0x00000020	R10: 0x00000009	R18: 0x00000000	R26: 0x00000000
R3: 0x00000012	R11: 0x0000000C	R19: 0x00000000	R27: 0x00000000
R4: 0x00000000	R12: 0x0000000F	R20: 0x00000000	R28: 0x00000000
R5: 0x00000000	R13: 0x00000012	R21: 0x00000000	R29: 0x00000000
R6: 0x00000000	R14: 0x00000015	R22: 0x00000000	R30: 0x00000000
R7: 0x00000000	R15: 0x00000018	R23: 0x00000000	R31: 0x00000000