EE480 Assignment 1: Assembling The Gr8BOnd

Implementor’s Notes

Alex Dingus

agdi224@uky.edu

*ABSTRACT*

The project was to design a multi-cycle processor implementing the Gr8BOnd instruction set.

*GENERAL APPROACH*

The first thing we did was a lot of staring at reference materials and sample code. We wanted to find a Verilog code on how to implement MIPS, because we worked with MIPS so much in 380. The first code we sat down and wrote was the `define code that specifies how long a word is, how long an opcode is, etc. Then we converted all the opcodes into hex and copied those over into our code. We elected to use the provided sample solution to Assignment 1, as none of us felt confident about our pseudo-instructions.

One of our first top-down approaches was to physically draw a map of what kind of modules we would need, as seen in figure 1. Then we went instruction by instruction, making sure we had what we would need. In the midst of that process, we started to think about and discuss the input and outputs of each module, using a Verilog implementation of MIPs as a reference (<https://www.fpga4student.com/2017/01/verilog-code-for-single-cycle-MIPS-processor.html>).

The first module we decided to take a closer look at was the ALU. We looked at the MIPS ALU control Verilog code and thought about how we would pass along bits of the opcode to give to the ALU. We recognized that the ALU would need two inputs of 16 bits each. We again heavily relied on another article from the same site (<https://www.fpga4student.com/2017/06/Verilog-code-for-ALU.html>) for the basic design of our ALU, of course changing the amount of bits in the inputs and outputs to 16 bits and changing the opcode being fed into the ALU to 8-bits so that we could just copy over the whole opcode to the ALU. Our general process here was to literally go line-by-line through the MIPS ALU and determine which lines to change and which lines to essentially copy over. We went instruction-by-instruction and defined what the ALU would need to do for the more basic instructions such as addi and muli, using the Gr8BOnd reference material as a fairly one-to-one resource for how to tell the ALU what to do in each instruction. We also included single register instructions such as not and negi in the ALU, basically everything that doesn’t need the PC or memory. We tried to test the ALU using a modified version of the testbench from that website, which helped us identify a few typos. The shift instruction needed more attention. This is also where we switched to sharing code using GitHub instead of just Google Drive. It turns out the Shift Left was working fine, but for some reason we had been using an 8-bit register for the ALU result, which we rectified. Another issue we had was just getting used to working with the syntax of `define code. We were getting errors because we had been including brackets in the initial definitions as well as when we called them later. Our initial ALU testing just consisted of running through each group of instructions via for statement with arbitrary values to check for super obvious common-sense errors.

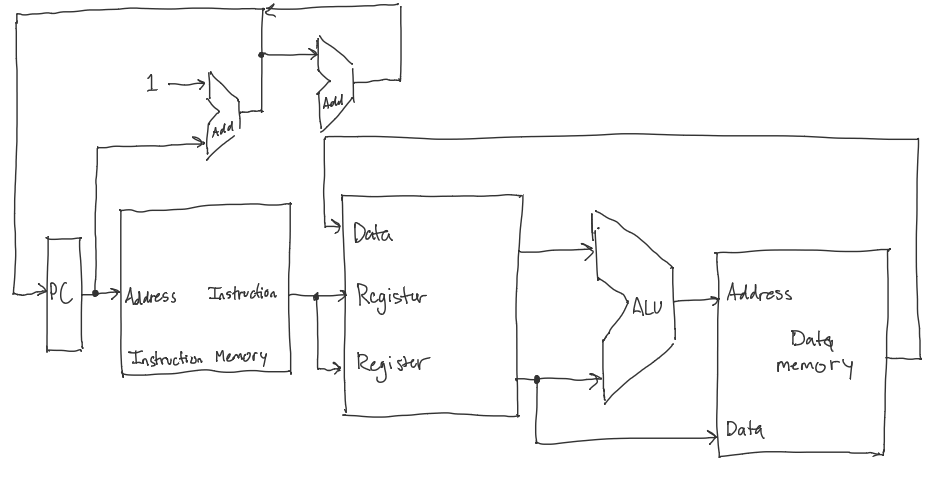
Next we tackled memory

Figure 1