EE480 Assignment 1: Assembling The Gr8BOnd

Implementor's Notes

Alex Dingus, Lisa Roberts, Trey King

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 by referring to reference materials and simply declaring it as an array of registers. Sorting out all the wires and registers within the main module took a lot of careful planning. We studied the TACKY equivalent a lot and went line by line to make sure we weren't forgetting anything, but we made sure to not copy and paste since the assignments are so different. We realized here that we had been including the carryout ALU bit both as a 17th bit on the ALU output as well as a separate 1-bit output, so we decided to remove the 17th bit from the ALU output to more easily distinguish between the two for our sake. Perhaps we had gotten caught up in the spirit of the TACKY project that we had been relying on as a reference material.

At this point we decided to include states as part of our design, even though we did not fully understand their functionality. We kind of blindly followed the TACKY processor's main module for a bit, before realizing we'd really need to sit down and sort out the states and groupings of instructions. At this point, our states were the Start state, Decoding instructions, and Executing instructions with the ALU. Our "ALU" is really more of a general module for resolving instructions. We took particular care with the ANY instruction, since it does not seem to be a super commonly found operator in the wild.

At this point we had been using covered to test the general functionality as we went along. Dumpfile and dumpvars became our friend. For the i2p and similar instructions, we opted to just set the ALU result to the input.

Next, we worked on getting the branch and branch-not-zero to work, just being careful to access all the correct values and to perform all the steps in the right order. We struggled with getting the WWW tester to work, experimenting with different testbenches and ways to set up the memory since we were not getting a full coverage report. We also struggled with various PC related issues, although it was difficult to figure out why testing wasn't working due to the lack of being able to test. Next we tackled the constant instructions, making sure to use the middle 8 bits of the instructions as the 8-bit constant. We continued to have mixed results with testing. We fixed the jr instruction to be more compatible with the assembler and also to feature a check to make sure the address isn't out of bounds. We went back and made sure to check for overflow in the addition of 8-bit integers, when we had previously only been checking for overflow with addition between 16-bit integers. We also needed to add overflow detection for multiplication. We weren't sure what to do when we detected an overflow, but we settled on displaying an error message and throwing up a halt. We continued to go instruction by instruction, checking each for correctness. We struggled with shifting left anytime the shift amount was negative.

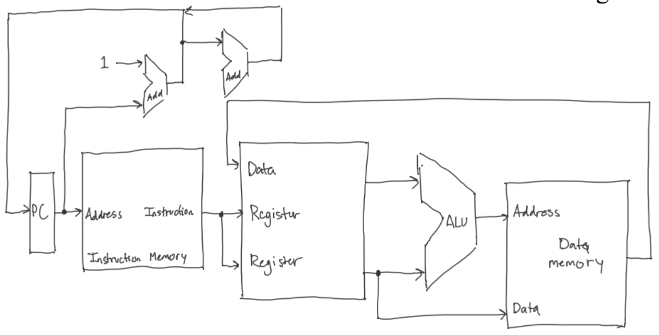

Figure 1