CPE 480 Assignment 2
Multi-Cycle LOGICK

Mark Noblin, Tanner Wilkerson, CJ Vanderpool

***Abstract*—The goal of this assignment was to utilize the assembly code written in a previous assignment to develop and produce a multi-cycle processor design. The multi-cycle design should be able to execute most of the commands encoded with in the assembly using Verilog programming as a source for logic.**

I. General Approach

A. Assignment Specifications

The processor to be developed in this assignment needs to take care of executing a specific set of commands from the assembly encoding. Our group decided to use a fresh encoding that was more tailored to the needs of the assignment. The commands that this assignment focuses on executing are those that do not require the Logarithmic Number System, or LNS, that will be implemented later on. This means that the processor should be able to execute ad, an, or, no, eo, mi, sr, br, jr, li, si, co, lo, st, and sy. The encoding also has a macro developed for the la command, which was discussed in the last assignment. This operation, however, does not require anything from our processor as it simply tells the processor the proper combination of li and si to perform.

The advice given by the assignment is to stick to a top-down approach to designing the processor. Our group chose to do so, while also sticking to the ideology of breaking up our code chunks into seperate modules. This allows us to specify as much as possible what exactly verilog builds when the entire program is synthesized. It helps to avoid unecessary or messy synthesized blocks and helps in easing the stress of testing.

Our design makes use of a few very important structure blocks to achieve the needed functionality. We have an ALU that is capable of handling those operations that specifically require arithmetic to be performed. We also have a mainmem and regfile block that can not only read and write but perform shifts to the left and right as well. We a block currently being called the oracle that specifies the structure of the Finite State Machine, or FSM, used to determine the execution of instructions. Lastly, we have a few smaller blocks, such as the MDR, MAR, A, Y, and condition blocks that help in the movement and execution of data onto our bus.

The last step after designing all of this is our testing method. The first step in testing was to develop small code blocks capable of testing the logic of each of our instructions upon their execution using the operations we have available. For example, we might use ad to add two numbers and store them into memory, then use co to compare this to a predetermined result and see if this is correct. The testing program then would use a br operation to avoid a sy call that would halt the program if that result was incorrect. This provides a streamlined method for determining which operations function properly and which need work.

B. Code Implementation

II. Testing and Issues

A. Testing

B. Known Issues