Colin Grundey

ECE 2500

October 23, 2017

Project 1 Report

I used Python to create a MIPS assembler. First, I created 3 Python dictionaries that hold R-Type function codes, I-Type opcodes, and register numbers. This minimizes conditional statements for all of the different instructions by making it simpler to find the correct codes. The program begins with parsing branch labels in the assembly file for their names and locations and stores that information in a Python dictionary. This is necessary because an instruction may branch to a label that has not been reached yet, so a preliminary label search solves that issue.

Next, I parse the file once again to read the instructions line by line. To determine the type of instruction, I use the dictionaries with opcodes and function codes to see what set that instruction derives from. Once the instruction type is obtained, I begin to look at the operands. Certain instructions have unique formats, but both R-Type and I-Type have a set of similarly formatted instructions in that they have the same number of operands and they are in the same order. I begin with the common formatting across all instructions and then deal with the unique formats. Immediate values are converted appropriately to a binary string. Branching instructions use the dictionary of labels to find the correct displacement of PC. The dictionaries containing the opcodes and function codes serve as a lookup table for me to obtain the binary translation of codes and registers as a string. Once the given instruction and operands are translated to a binary string, I concatenate them all, creating the 32-bit instruction and then convert to hexadecimal. Then, the hexadecimal instruction is added to a list to later be output to an object file. The reason it is not written to the object file immediately is to avoid a partially written file when an error occurs. The program will wait until all parsing and conversions are finished to confirm there are no errors.

There are many places for errors to occur in the formatting of the assembly file, and my Python assembler is able to properly handle them and output a useful message upon exiting the program. Input and output files are given through command line arguments so there is error checking there to make sure the input file is valid and the output file has a valid object file extension. One error that may occur is an incorrect number of operands. I check this for each possible kind of instruction and record the instruction name for debugging. Another error that can happen is an unsupported instruction. I use the dictionaries at the beginning of the file with opcodes and function codes to determine if the instruction is supported. Similarly, my program looks for the registers in the register dictionary and labels for branching in the labels dictionary to determine if the given operands are valid.

Lastly, an object file is opened to be output to. I cycle through the list of hexadecimal instructions and output each one to the object file. The file will only be written if there are no errors from the assembling beforehand. It may overwrite if the given output file exists.