Adam Rankin
10/14/19

ECE 2500 Project 1

Introduction

The goal of this project was to build a high-level language MIPS core instruction assembler. The assembler takes human readable MIPS assembly code and converts it to 32-bit machine code in hex. It takes a file with a '. s' extension and outputs a file with the same name and a '.obj' extension.

Approach

The language I chose for this project was python. I chose python because it makes string manipulation, list traversing, and exception catching very easily. Additionally, dictionaries in python are very easy to use and manipulate. Since performance isn't a big worry, the overhead of using python is not a factor. My overall approach was to build one dictionary containing all of the commands as keys, and their type and op/function code as the value, and another dictionary of registers and their integer values. This way it would be easy to build out the line in binary and easily convert to hex using python built-ins.

Data Structures

Upon initialization of the myAssembler class, the two data structures mentioned above are built. They are called instructions and registers, respectively. If the instruction is I-type, the instruction, opcode is stored, and if the instruction is j-type, the function code is stored. After the file is read in and initialized, the assembler calls another function which builds a data structure for the labels.

Label Handling

Label names were preprocessed into a dictionary of their name, and their integer line number. This way and branch offsets can easily be calculated later, by just referencing the label map data structure build at the beginning of assembly.

Instruction Parsing

Instruction parsing was handled in two parts based on the instruction type, with the exception of the jr command, which was handled alone. The simpler j-type instructions were parsed for their arguments and then the binary instruction string was built from the existing data structures. If the last argument was a shift amount, then that was converted into a binary string and concatenated accordingly. I-type instructions are slightly more complicated. If the instruction used pointer notation, that was checked first. If it did then the immediate was parsed out from the instruction, while if not, the immediate was set the first instruction. Then, if the immediate was negative it was sign extended 16 bits and added to the binary instruction. Once a 32 bit binary string was created, python allows a conversion to an int, followed by a cast to hex. This was zero-filled to 8 bits to create the instructions.

How to use myAssembler

myAssmbler can be called from the command line as follows: './myAssmbler *.s'. or using python3. The only requirements are that assembler.py is in the same directory, as it contains the

assembler class, and that the file exists and ends with the ‘ .s’ extension. This is highlighted step by step in readme.txt.

Conclusion

Overall, I actually really enjoyed this project because I love programming in general. It additionally helped me learn a lot about how an actual assembler may put code together, and learn more about MIPS in general.