EE126 Lab7, Fall17

Michelle Chan and Ashton Stephens, Tufts University

December 12, 2017

Introduction

Although programming languages consist of syntax and semantics reminiscent of natural languages, all data in computers are encoded in binary bits that hold the state of either '1' or '0' with arbitrary meaning attached. A process of translating languages into machine code is needed and often performed by other kinds of programs.

Often, a high-level language may be compiled into an assembly language, the lowest-level of abstraction from binary. From assembly, an assembler translates assembly code into machine code in binary format readable to computers.

MIPS is an instruction set architecture (ISA) introduced in 1985 often used in embedded systems and university courses. As a reduced instruction set arcitecture (RISC), most assembly instructions in this ISA are simple such that the cycles per instruction are kept low.

In this course, a 5-stage pipeline was built in VHDL. This MIPS 32-bit assembler was built to help translate MIPS assembly instructions into machine code that can be easily loaded into instruction memory in VHDL.

Specifications

This MIPS-32 assembler reads from a file specified as a command line argument or from stdin if a file is absent. The output is printed on stdout and can be piped into a file.

This assembler recognizes all 32 MIPS-32 registers and labels. A list of supported instructions, including pseudo-instructions, are clearly listed in the "opcode_code.h" file provided.

Implementation

On the surface, the assembler conducts two passes where the first pass handles labels and the second pass translates each assembly line of code.

Tables

There were three tables used to lookup information used in the implementation.

1. The mneumonic table maps a mneumonic to a corresponding opcode.
2. The register table maps the register name to its number in memory.
3. The pseudoinstruction table maps a pseudoinstruction to the instructions it expands into along with the order to pass in registers.

First Pass

Each line of assembly code in the program is read line-by-line and stored internally to be read again from the second pass. At each line, the first word is checked for a label and the word address is incremented the appropriate amount. Labels are stored as the key to its word address value. The word address that is tracked always increments by one on a MIPS instruction because there exists the invariant that each instruction is one word long. The first word is also checked against the internal pseudoinstruction map to increment more than one word forward.

Second Pass

In the second pass, the mneumonic and operands of each line are parsed into strings. Using the current word address and parsed strings, the assemble method is called to translate into machine code.

Internally, the translation looks up the mneumonic to get either an Instruction type or Pseudoinstruction type internally defined as structs containing all the necessary information for assembling. In "struct Mnemonic_func," the information stored includes the opcode (or funct field), the instruction format (R-type, I-type, or J-type), and the syntax group. Each syntax group referred to a unique order in which the operands for an assembly instruction were translated and placed into the machine instruction.

In the case that the mneumonic maps to a pseudoinstruction, the list of instructions that it expands to is returned along with the appropriate order of operands for each instruction. Using this, the instructions are referenced in the map again.

After the translation, the binary is printed to stdout and the number of words to increment the program counter is also added into the word address.

Results

The assembler was tested using the test assembly programs from previous labs for testing the 5-stage MIPS processor. The output in binary was decoded into MIPS assembly again and compared with the original input to check for the functionality of this assembler.

The files "lab3.asm" and "lab6.asm" tested the functionality of the label handling. Additionally, these programs as well as "lab4.asm" and "lab5.asm" included each instruction format, so the functionality of processing R-type, I-type, and J-type instructions was also tested.

Although this assembler recognizes a larger MIPS ISA than just the instructions that were tested, the implementation for each insuction format was the same. Then, testing each format using the programs provided was sufficient coverage for verifying functionality.

The files "pseudo1.asm" and "pseudo2.asm" tested functionality of pseudoinstructions. The first file tests all pseudoinstructions included in the lookup table. The second file tests the integration of pseudoinstructions with a program with other instructions and labels.

Two reference output files containing the programs in hex are provided based off the files with pseudoinstructions. They are used to check for differences with the output of the assembler. Although the assembler currently outputs ASCII binary, that output can be converted to hex using a script. For this assembler, the tests used were passed.

The use of lookup tables enabled the opcode, syntax group, and potential labels to each be found in constant time efficiency.

Conclusion

Improvements to this implementation of the assembler include adding more instructions and macro capabilities.

The set of instructions and pseudoinstructions can be easily expanded in this implementation by adding to the lookup table. The infrastructure for implementing macros is also available from the struct that defines a portable instruction. Then, a macro would be implemented like a pseudoinstruction in which the list of instructions to be run are set by the program instead of stored as constants in the lookup table.