Report for CSC3050 Project 1

0. Introduction to the Usage of my Source Codes

The file *"scan_label.py"* generally scans and stores the addresses of the labels appear in the code (generally serves as the function of *"phase1.py"*). I use a python dictionary to store the labels and their corresponding addresses ("label table").

The file *"process_mips.py"* generally converts the MIPS code into machine code after the scanning for label address (generally serves as the function of *"phase2.py"*).

The file *"tester.py"* is the tester that prompts users for input and generates output. **Please run this file when testing. I made some modifications on the tester provided so as to make it compatible to my programs.**

1. How a MIPS assembler works

MIPS is an assembly language, which is a low-level programming language that each line of code corresponds to one machine instruction. The MIPS assembler generally converts MIPS codes into executable machine code.

MIPS instruction is composed of R-Type, I-Type and J-Type instruction, each corresponding to one encoding format.

1.1 R-Type instruction

R-Type instruction contains all instructions that do not require an immediate value, target offset, memory address displacement, or memory address to specify an operand. This includes arithmetic and logic with all operands in registers, shift instructions, and register direct jump instructions.

The MIPS assembler converts an R-Type operation into the sequence of

opcode (6)	rs (5)	rt (5)	rd (5)	sa (5)	function (6)

All R-type instructions use the same opcode 000000. Registers *rs*, *rt* corresponds to the first, second register source operand, and register *rd* is the destination

operands. *"Shamp"* means the shift amount, and *"function"* is the function code of the corresponding instruction. MIPS assembler converts each component to its binary code and link them together using the above sequence to form the machine code.

1.2 I-Type Instruction

I-Type instructions include instructions with an immediate operand, branch instructions, and load/store instructions. Branch instructions involves the handling of PC-relative address.

The MIPS assembler converts an I-Type operation into the form of

All opcodes except 000000, 00001x, and 0100xx are used for I-type instructions. *"Immediate"* is the offset of address. I-Type instructions have a 16-bit immediate operand, a branch target offset, or a displacement for a memory operand. For a branch target offset, the *immediate* field contains the signed difference between the address of the following instruction and the target label, with the two low order bits dropped. The dropped bits are always 0 since instructions are word-aligned. MIPS assembler converts each component to its binary code and link them together in the above sequence to form the machine code.

1.3 J-Type Instruction

J-Type instruction consists of the 2 direct jump instructions. The only J-type instructions are the jump instructions *j* and *jal*. These instructions require a 26-bit coded absolute address field to specify the target of the jump. The coded address is formed from the bits at positions 27 to 2 in the binary representation of the address. The bits at positions 1 and 0 are always 0 since instructions are word-aligned.

The MIPS assembler converts an R-Type operation into the form of

When a J-type instruction is executed, a full 32-bit jump target address is formed

by concatenating the high order four bits of the PC (the address of the instruction following the jump), the 26 bits of the *"target"* field, and two 0 bits. MIPS assembler converts each component to its binary code and concatenates them together in the above sequence to form the machine code.

2. Implementation of MIPS Assembler

2.1 Main Idea of Implementation

The main task of the program is to convert each line of MIPS code into machine code, based on the principles introduced above. The input MIPS program only contains *".data"* and *".text"* section, we then need to store the content of .data in certain data format, and converts MIPS codes into machine code.

The implementation involves **scanning for 2 times** of the whole program. In the first scan (written in *scan_label.py*), we store the data in the *".data"* and store all the labels and their corresponding address appeared in the *".text"* section for later use. In the second scan (written in *tester.py*), we process the instructions line by line, using the labels we obtain in our first scan.

2.2 Scan for the First Time

First, we need to identify each section, we read line by line and ignore comments and empty lines when reading each line. If a *'.data'* symbol is detected in the line, then in the reading of the next line, we need to store the data information in a list structure (*pandas* Dataframe is also OK), until a "*.text*" symbol is detected.

If we detect *'.text'* in the line, then in the next read, we need to prepare for recording the labels since the I-type and J-type instruction involves labels processing. After we read the line and ignore the comment, we use *split()* function to split the line according to the empty space between the MIPS code. Then the line is converted to a list, with each element being one component of the MIPS code. If the first element of the list is not a MIPS instruction, it means that it is a label which will be used later, we need to store it in a dictionary (variable "*label_dict*" in the source code), together with

its relative address (variable "*address_index*" in the source code) as value. The relative address is calculated in the following way.

We assume the index of the first MIPS instruction (excluding label) line to be 0, and we scan line by line.

If we detect that the line is a MIPS instruction (by checking whether the first or second element is in the MIPS Instruction set list, since the label can be in the same line with the MIPS instruction), we add 1 to the "address_index" at the end of each reading loop.

Else, it means a label is detected, we don't need to increment the "address_index", since the label itself has no address.

In this way, we manage to store the *"address_index"* of each label in the dictionary.

2.3 Scan for the Second Time

In the second scan, we convert each MIPS code to machine code. This time, we start the converting process from *".text"* section and also, we again need to track the *"address_index"* of each instruction in the way we mentioned above, in case we need to do branching or unconditionally jump to a certain label.

If we meet the I-type instructions that involves label addressing and branching, we detect the label part of the instruction code and access its address index stored in the dictionary ("*label_dict*") we constructed in the first scan. The immediate address offset of such branching instructions is

Immediate = ("address_index" of the label needed) – ("address_index" of the line of this instruction) – 1

If we meet the J-type instruction that involves direct address targeting. Since the address of instruction starts from 0x400000 in this project, we have

Target = 100000 + ("address_index" of the label)

It's easy to deal with other cases which does not involves label addressing. We can simply determine which instruction it is, obtain the number of registers, opcode, function code, convert them into binary, and then concatenate them together with certain ways into machine code.

2.4 Data Storage of the variables

2.4.1 Data Storage for the ".data" section

Originally, I plan to store the data in a Dataframe provided by "pandas" module, but I am not sure whether TA has "pandas" module in python, so finally I did not do so, since the program will pose an error and won't run if no "pandas" module is detected. Instead, the data will be stored in a 2-dimensional list (variable "data_list"). (But I reserve the code for constructing a "pandas" Dataframe, the grader can check it if he/she needs)

2.4.2 Data Structure of the Relative Address of Labels

They are stored in the dictionary (variable "*label_dict*") in the form of key-value pair during the first scan process. Every time we met a line whose first element is not a MIPS instruction code (meaning that it must be a label), we add them into this dictionary.

2.4.3 Data Structure to Store Other Variables

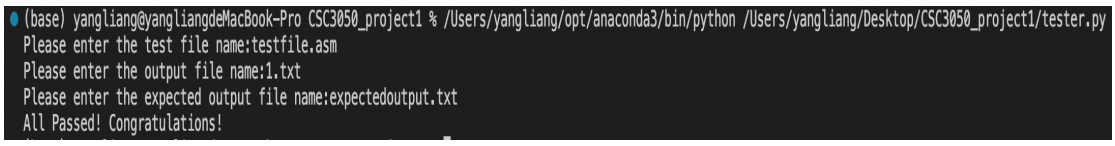
All MIPS codes instruction is stored in the form of lists.

Register's symbols and its corresponding number is stored in a dictionary.

Opcodes are stored in a dictionary.

Function codes are stored in a dictionary.

3. Code Demo



Please run the code in *tester.py*.

Please enter the files one by one as instructed.

First, enter the test file name, such as *"testfile.asm"*. It is the *".asm"* testcase file of MIPS code.

Next, enter the output file name, such as *"1.txt"*, which should be the *".txt"* file for the program to write its output to. A new file will be generated if the file name does

not exist.

Finally, enter the file name of the expected correct output, such as *“expectedoutput.txt”*.

The tester will compare line by line the output file and the expected correct output to check whether the translation is correct.

Then the feedback will be given.