Program 1 report

陈启旭 120090643

1. Big picture thoughts and ideas

   An assembler takes a single assembly language source file and converts it into an object file with machine instructions.

   There are two primary aspects to the translating process:

   1. The first step is to locate memory regions with labels to find the relationship between symbolic names and addresses. (First scan)

   2. The second step is to convert each assembly statement into a legal binary instruction by combining the numeric equivalents of opcodes, register specifiers, and labels. (Second scan)

2. High level implementation ideas

   a) Tester.py

      The tester.py file will read the name of the test file, output file and expected output file. Name of the test file, output file will be stored in the file of tem_file.txt.

   b) First scan (phase 1.py):

      Read the assembly file into the program by lines

      i. Cope with the string:

         A. Delete all the "\t" and "\n" and all the blank in the end of each line

         B. Find where "#" locates and delete all the characters after this location (delete all the comments)

         C. Delete all the blank space

         D. Delete all the texts between ".data' and '.text' (including ".data" and ".text")

         E. Cope with line break as labels can be followed by a line of code, or can have its own line. The label and its following line saved in the same line regardless of whether line break between the label and the line exists.

      ii. Save the label:

         A . Find where ":" locates, then store the label and its corresponding address.

   c) Second Scan (phase2.py)

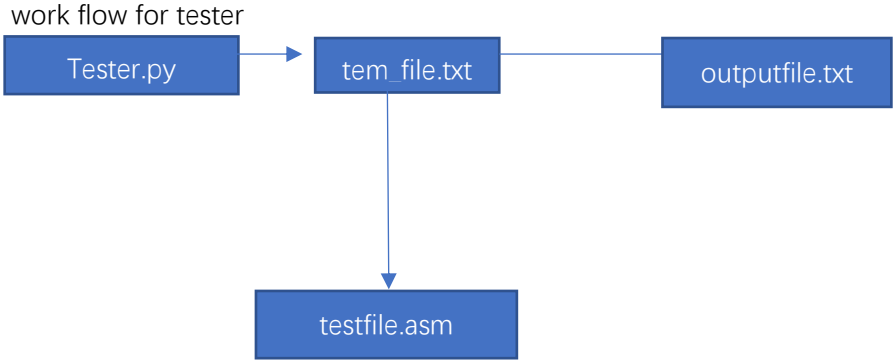
      Read through the tem_file.txt to get the name of the test file and output file.

      During the process of the second scan, phase2 matches each row to the appropriate type and machine code structure (the type and machine code structure have been stored in phase2.), then returns the matching machine code and store all the machine codes in one data structure.

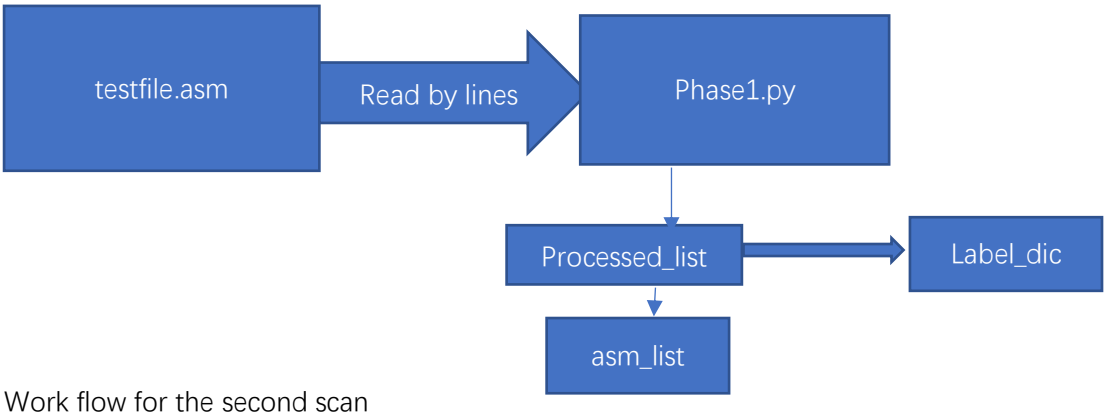
      Finally, the machine code is written to the output file.

   d) workflow for the whole process

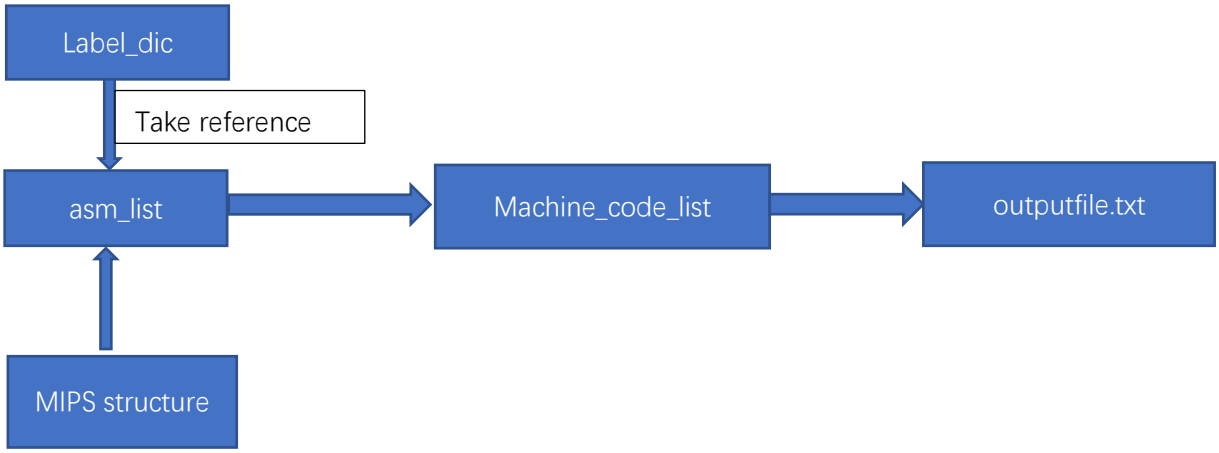
      work flow for tester



work flow for the first scan



Work flow for the second scan



3. The implementation details

a) Data structure

Processed_list: a python list containing each single line of the input file as an element (Without comment and blank) but still contains the label

asm_list: a python list containing each single line of the input file as an element (without comment, blank and label)

Label_dic: a python dictionary with the label as the key while its address as the value (the basic address is 0x400000, which is 4194304 in decimal. The address in the dictionary is the labels' location in Processed_list.)

MIPS structure: 13 dictionaries containing different types of operation and its corresponding opcodes or funct code. For example:

#i instructions concerning arithmetic operations

i_opcode_dic_nb={ "addi":"001000","addiu":"001001","andi":"001100", "ori":"001101","xori":"001110","slti":"001010","sltiu":"001011",}

Register_list: a list storing all the registers' name, from 0 to 31.

Machine_code_list: a list containing all the binary codes derived.

b) Some special details

3.2.1 how to cope with line break

The program provides a temporary list called temp_list which store the all the lines except lines with":" in it. The program then gets the length of the list.

We then define a "position" pointing to the start of the temp_list.

Append all the lines into processed list, if the prior element ends with ":", then concatenate it with the next element as a single element. The "position" moves by two. Else, moves by one.

Loop until "position" points to the end of temp_list.

3.2.2 how the asm_list takes reference of the MIPS structure and label_dic

The phase 2 will read through every element of the asm_list.

PC will start from 4194304(0x400000 in decimal) initially

A. Every time phase2 lift out an element of asm_list, the program will convert the element into a list.

   Example:

   Convert **"addi $t1, $t1, 2"** to ["addi"," $t1"," $t1","2"]

   A special point during this process is that if the program simply use split (","), then, "$t1,2" will become "$t12". The method to solve this problem is load the "addi" into a list named split_line and delete the "addi" from the original string. Then we delete all the blanks contained in the left part of the string. Next, we split the string into a new list using split (","). The code looks like:

```
elements = element.replace(oper,"")
elements = [x.strip() for x in elements.split(",")]
```

We concatenate split_line and the new list to form the list split_list.

B. We read through every element in the split_list, and find the corresponding codes.

   For the first element, like "addi", we search through the 13 dictionaries to find its corresponding opcodes. We then search through register_list to find the number of the register. We translate the number into binary code.

   We can find corresponding codes of rs, rt, rd in this way.

   For example:

```
elif split_line[0] in i_opcode_dic_nb.keys():
        opcodes = i_opcode_dic_nb[split_line[0]]
        rs = reg2code(split_line[2])
        rt = reg2code(split_line[1])
        immediate = decimal2code(split_line[3],16)
        code = opcodes+rs+rt+immediate
```

   The function **reg2code()** will translate the name of the registers to the corresponding codes

   The function **deciaml2code(str ,n)** will translate a decimal number in string form to n bits binary code(if the decimal string has negative sign before it , the function will transform it into 2's complement)

   For assembly codes with operations of jumping to specific label, this program has two functions to generate the immediate codes for the machine codes according to the current addresses of the current line and the target line.

The

C. We add 4 to PC every time after reading one split_line to update the address.
D. The program appends the generated machine code to the machine code list.

4. How to test the program
   A. Open the “tester.py”
   B. Enter the name of the file according to the instructions:

```
D:\Python\python code\mips assembler\CSC3050_P1>C:/Users/陈/AppData/Local/Programs/Python/Python38-32/python.exe "d:/Python/python
Please enter the test file name:testfile4.asm
Please enter the output file name:outputfile.txt
Please enter the expected output file name:expectedoutput4.txt
['add', '$s0', '$s1', '$s2']
```

C. Output in IDE

The output will print out all the machine code, the location of the file storing the machine codes, whether the test success or fails.

```
Please enter the test file name:testfile.asm
Please enter the output file name:outputfile.txt
Please enter the expected output file name:expectedoutput.txt
['10001111110010011111111001110000', '10001111110010100000000110010000', '10001111110010110000000000000000', '10101111110010011111111110011100
00', '10101111110010100000000110010000', '10101111110010110000000000000000', '10001011110010011111111001101111', '10011011110010100000000110
010011', '10101011110010110000000000000001', '10111011110010111111111111111111', '10000011100010001111111111111011', '10100011100010001101001011011000']
the machine code is now stored in outputfile.txt
All Passed! Congratulations!
```

If the test fail, the output will print out in which line the code of output is different from the expected output.

```
Please enter the test file name:testfile4.asm
Please enter the output file name:output.txt
Please enter the expected output file name:expectedoutput4.txt
['00100000000100000000000000110111', '00100000000010010000000000000000', '00100000000010100000000000000001', '00100000000010110000000000000
00', '00010001001100000000000000000100', '00000001001010100101100000100000', '00100001010010010000000000000000', '00100001011010100000000000
000000', '00001000000100000000000000000100', '10101100100010110000000000000000']
the machine code is now stored in outputfile.txt
You did something wrong!
4
8
```