

CSC3050 Project 1 Report: Design of an Assembler

杨宇凡 117010349

1. Programming Language: Python

2. Basic Program's Logic:

Python read the content from the assemble language file, and then store it in a list. Every element in the list is the content of a single line. Then python deal with these elements: delete extra content (tab, space, line break, comments), pick up tags and their addresses and store them in a dictionary (a kind of data structure), and break up the string into several words and store them in a small list as the element of the big list. Then python judges the type of each line based on the first word of each line and give the corresponding machine code. At last write it to the output file.

3. Program Components: tester.py, phase1.py, phase2.py

✧ tester.py

This file is used for testing. It will ask the user to input names of test file, output file and expected output file. Then it activates phase2.py to do the compile job. After finishing, it will compare the content between output file and expected output file and told the user whether pass the test.

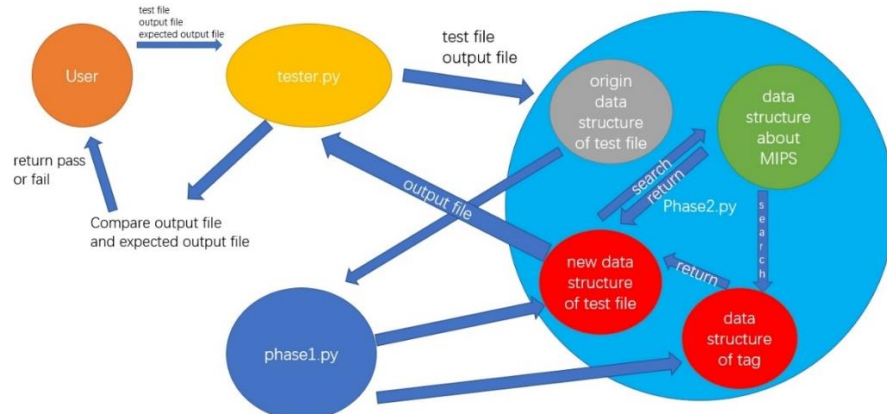
✧ phase1.py

This file store a function to deal with the content of each line of the original assemble file and then store each line and tags in proper data structure for the convenience of further searching and generating the machine code.

✧ phase2.py

This file is the main part of the program. It provides the clear data structures storing the characteristics of MIPS language. And based on function provided by phase1, it matches each row with the corresponding type and machine code structure, and then give the corresponding machine code. At last, it writes the machine code to the output file.

4. Program Overview:



5. Data Structures:

✧ label_dict {}: `label_dict=dict()`

label_dict is a dictionary which store the name of each tag and its corresponding address. Dictionary is a very useful data structure in python. Each element in the dictionary consists of two parts, one is called key and the other is the value. Each key match to its corresponding value. Therefore, this data structure is convenient for the search of the tag's address. (It does the same job as labelTable.c in C, so this part is dropped here.)

```
{'main': 0, 'begin': 1, 'loop': 3, 'finish': 8}
```

✧ R_functions {}:

```
R_functions={
    "sll":"000000","srl":"000010","sra":"000011","sllv":"000100",
    "srlv":"000110","srav":"000111","jr":"001000","jalr":"001001",
    "syscall":"001100","break":"001101","mfhi":"010000","mthi":"010001",
    "mflo":"010010","mtlo":"010011","mult":"011000","multu":"011001",
    "div":"011010","divu":"011011","add":"100000","addu":"100001",
    "sub":"100010","subu":"100011","and":"100100","or":"100101",
    "xor":"100110","nor":"100111","slt":"101010","sltu":"101011"
}
```

R_functions {} is a dictionary which stores all the R type operations as keys and their corresponding function codes as values.

✧ I_opcodes {}:

```
I_opcodes={
    "bltz":"000001","bgez":"000001","beq":"000100","bne":"000101",
    "blez":"000110","bgtz":"000111","addi":"001000","addiu":"001001",
    "slti":"001010","sltiu":"001011","andi":"001100","ori":"001101",
    "xori":"001110","lui":"001111","lb":"100000","lh":"100001",
    "lw":"100011","lbu":"100100","lhu":"100101","sb":"101000",
    "sh":"101001","sw":"101011","lwc1":"110001","swc1":"111001"
}
```

I_opcodes {} is a dictionary which stores all I type operations as keys and their corresponding opcodes as values.

✧ J_opcodes {}:

```
J_opcodes={
    "j":"000010", "jal":"000011"
}
```

J_opcodes {} is a dictionary which stores all J type operations as keys and their corresponding opcodes as values.

✧ Coprocessor_instructons {}:

```
Coprocessor_instructons={
    "add.s":["10000","000000"],"cvt.s.w":["10100","100000"],"cvt.w.s":["10000","100100"],
    "div.s":["10000","000011"],"mfc1":["00000","000000"],"mov.s":["10000","000110"],
    "mtc1":["00100","000000"],"mul.s":["10000","000010"],"sub.s":["10000","000001"]
}
```

Coprocessor_instructons {} is a dictionary which stores all coprocessor type operations as keys and their corresponding format and function codes as values.

✧ MIPS_Register {}:

```
MIPS_Register={
    "$zero":"00000", "$at":"00001", "$v0":"00010", "$v1":"00011",
    "$a0":"00100", "$a1":"00101", "$a2":"00110", "$a3":"00111",
    "$t0":"01000", "$t1":"01001", "$t2":"01010", "$t3":"01011",
    "$t4":"01100", "$t5":"01101", "$t6":"01110", "$t7":"01111",
    "$s0":"10000", "$s1":"10001", "$s2":"10010", "$s3":"10011",
    "$s4":"10100", "$s5":"10101", "$s6":"10110", "$s7":"10111",
    "$t8":"11000", "$t9":"11001", "$k0":"11010", "$k1":"11011",
    "$gp":"11100", "$sp":"11101", "$fp":"11110", "$ra":"11111"
}
```

MIPS_Register {} is a dictionary which stores all MIPS registers' names as keys and their corresponding addressing codes as values.

✧ Instruction Structure Lists:

```
rd_rt_sa_list=["sll","srl","sra"]
rd_rt_rs_list=["sllv","srlv","srav"]
rd_rs_rt_list=["add","addu","sub","subu","and","or","xor","nor","slt","sltu"]
rs_rt_list=["mult","multu","div","divu"]
rs_list=["jr","mthi","mtlo"]
rd_list=["mfhi","mflo"]
rs_label_list=["bltz","bgez","blez","bgtz"]
rs_rt_label_list=["beq","bne"]
rt_rs_im_list=["addi","addiu","slti","andi","ori","xori"]
rt_imrs_list=["lb","lh","lw","lbu","lhu","sb","sh","sw","lwcl","swcl"]
fd_fs_ft_list=["add.s","div.s","mul.s","sub.s"]
fd_fs_list=["cvt.s.w","cvt.w.s","mov.s"]
```

These lists classify the operations by their instruction structures. Each list contains the operations which follow the same instruction structure. For example, "sll", "srl", "sra" are all in rd_dt_sa_list because they all follow the structure of "rd, rt, sa".

```
sll    rd, rt, sa
srl    rd, rt, sa
sra    rd, rt, sa
```

✧ Unsigned_list(): `unsigned_list=["addiu","lbu","sltiu"]`

This list contains the operations which have to deal with the immediate number as unsigned number.

6. Functions:

✧ Label_tag (lines, label_dict):

```
def label_tag(lines,label_dict):
    for position in range(len(lines)-1,-1,-1):
        if lines[position].rstrip().replace("\t","").replace("\n","").replace(" ","")==":":
            del lines[position]
```



```

for position in range(0,len(lines)):
    line=lines[position]
    new_line=line.rstrip().replace("\t","").replace(","," ").replace("\n","")
    lines[position]=new_line
    if "#" in line:
        p1=new_line.index("#")
        new_line=new_line[:p1]
    if ":" in line:
        p2=new_line.index(":")
        label_dict[new_line[:p2]]=position
        new_line=new_line[p2+1:]
    new_line_list=new_line.split(" ")
    for i in range(len(new_line_list)-1,-1,-1):
        if new_line_list[i] == '':
            del new_line_list[i]
    lines[position]=new_line_list

```

This function has two parameters. The first input should be a list and the second input should be dictionary. The function does three things. Firstly, delete all the empty lines python reads. Secondly, delete all the “space”, “\t”, “\n”, “,” in each line’s string. Thirdly, delete the comments. Fourthly, pick up the tag in the line, delete them in the line and store their names and addresses separately as keys and values in the parameter label_dict {}.

✧ tobinary (number, length, sighed=0):

```

def tobinary(number,length,sighed=0):
    try:
        decimal=int(number)
    except:
        decimal=number
    if "-" in str(decimal):
        binary=str(bin(decimal))[3:]
    else:
        binary=str(bin(decimal))[2:]
    real_length=len(binary)
    if real_length<length:
        difference=length-real_length
        i=0
        while i<difference:
            binary="0"+binary
            i+=1
    if not sighed or not "-" in str(decimal):
        return binary
    else:
        i=length-1
        while i>0:
            if binary[i]=="1":
                break
            else:
                i-=1
        if i!=0:
            trans_binary=binary[:i]
            untrans_binary=binary[i:]
            trans_binary_list=list(trans_binary)
            for i in range(0,len(trans_binary_list)):
                if trans_binary_list[i]=="0":
                    trans_binary_list[i]="1"
                else:
                    trans_binary_list[i]="0"
            trans_binary="".join(trans_binary_list)
            binary=trans_binary+untrans_binary
        return binary

```

This function aims to transform an integer into signed or not signed binary format. It has three parameters. Number could be numbers in both integer or string format. Length is the length of the output binary number. Signed represents whether the output is signed or not signed, where “0” represents not signed and “1” represents signed. The default is 1. The function firstly judges whether the number is minus, then uses str and bin methods to get the positive binary string. If signed is equal to 0, it will return the binary string. Otherwise, if the string number is positive, it also directly returns the binary string, else if negative, it will do the 2’s complement and then return the value.

7. Procedure Details:

✧ tester.py:

```
test_file=input("Please enter the test file name:")
while not os.path.exists(test_file):
    test_file=input("Your input is wrong, please enter the correct name:")
output_file=input("Please enter the output file name:")
expectedoutput_file=input("Please enter the expected output file name:")
while not os.path.exists(expectedoutput_file):
    expectedoutput_file=input("Your input is wrong, please enter the correct name:")
with open("tem_file.txt",'w') as file_object:
    file_object.write(test_file+' '+output_file)
```

- Ask the user to input the file name, and check whether the file exist. If the file not exist, the program will ask the user to input again until correct. Then the program stores the two files’ name in a temporary txt file for phase2.py to load and use.

```
with open(output_file,"r") as file_object:
    output_lines=file_object.readlines()
with open(expectedoutput_file,"r") as file_object:
    expect_lines=file_object.readlines()

for i in range(0,len(output_lines)):
    line=output_lines[i]
    line=line.replace("\n","")
    output_lines[i]=line
for i in range(0,len(expect_lines)):
    line=expect_lines[i]
    line=line.replace("\n","").replace(" ", "").replace("\t","")
    expect_lines[i]=line

if output_lines == expect_lines:
    print("All Passed! Congratulations!")
else:
    print("You did something wrong!")
```

- Read the output file and the expected output file and store the contents separately in two lists. Then python delete “\n”, “\t” and space of each line which is elements in each list. Finally, python compare the two lists. If two lists are equal, then the user pass the test. Else, the user fails.

✧ phase2.py

```
with open("tem_file.txt") as file_object:
    tem_lines=file_object.readlines()
tem_lines=tem_lines[0].split(" ")
os.remove("tem_file.txt")
```

- From tem_file.txt get the test file name and the output file name, then delete it.

```
test_file=tem_lines[0]
output_file=tem_lines[1]
with open(test_file) as file_object:
    lines=file_object.readlines()
open(output_file,"w")
phase1.label_tag(lines,label_dict)
```

- Open test file, read the line from it and call the function in phase1.py to do the optimization of data structure to store the content in test file. Then use the open method in python: if the output file exist, clean it up ; else if not exist, create a new file.

```
for line in lines:
    if line[0] in R_functions.keys():
        opcode="000000"
        function_code=R_functions[line[0]]
        sa="00000"
        if line[0] in rd_rt_sa_list:
            rd=MIPS_Register[line[1]]
            rt=MIPS_Register[line[2]]
            sa=tobinary(line[3],5,1)
            rs="00000"
```

```
        machine_code=opcode+rs+rt+rd+sa+function_code
    elif line[0] in I_opcodes.keys():
        if line[0] in unsighed_list:
            sighed=0
        else:
            sighed=1
        opcode=I_opcodes[line[0]]
        if line[0] == "lui":
            rt=MIPS_Register[line[1]]
            im=tobinary(line[2],16,sighed)
            rs="00000"
```

```
    elif line[0] in J_opcodes.keys():
        opcode=J_opcodes[line[0]]
        if line[1].isdigit():
            target=tobinary(line[1],26)
        else:
            target=tobinary(label_dict[line[1]],26)
        machine_code=opcode+target
    elif line[0] in Coprocessor_instructions.keys():
        opcode="010001"
```

```

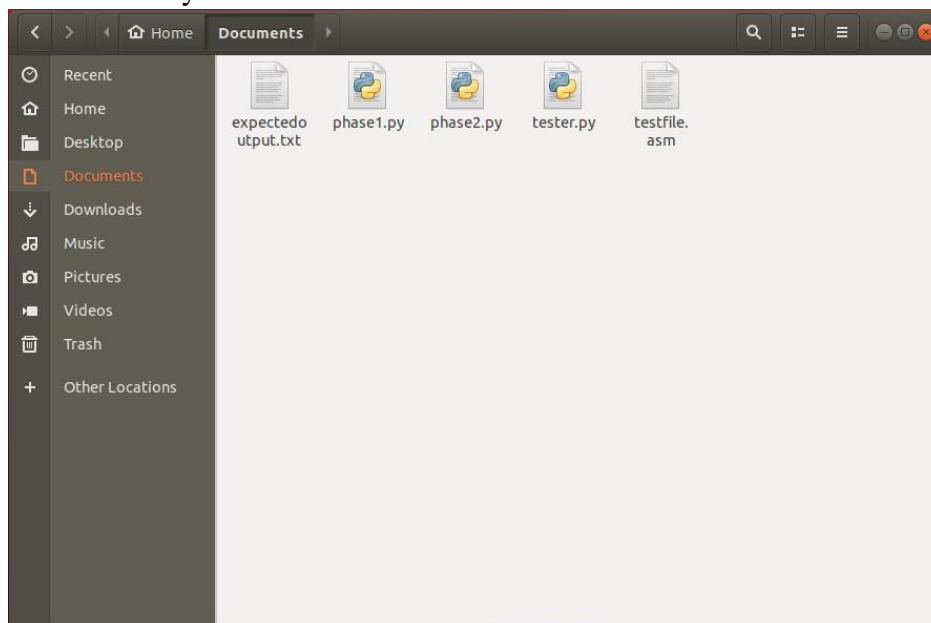
machine_code=opcode+format_code+ft+fs+fd+function_code
with open(output_file,'a') as file_object:
    file_object.write(machine_code+"\n")

```

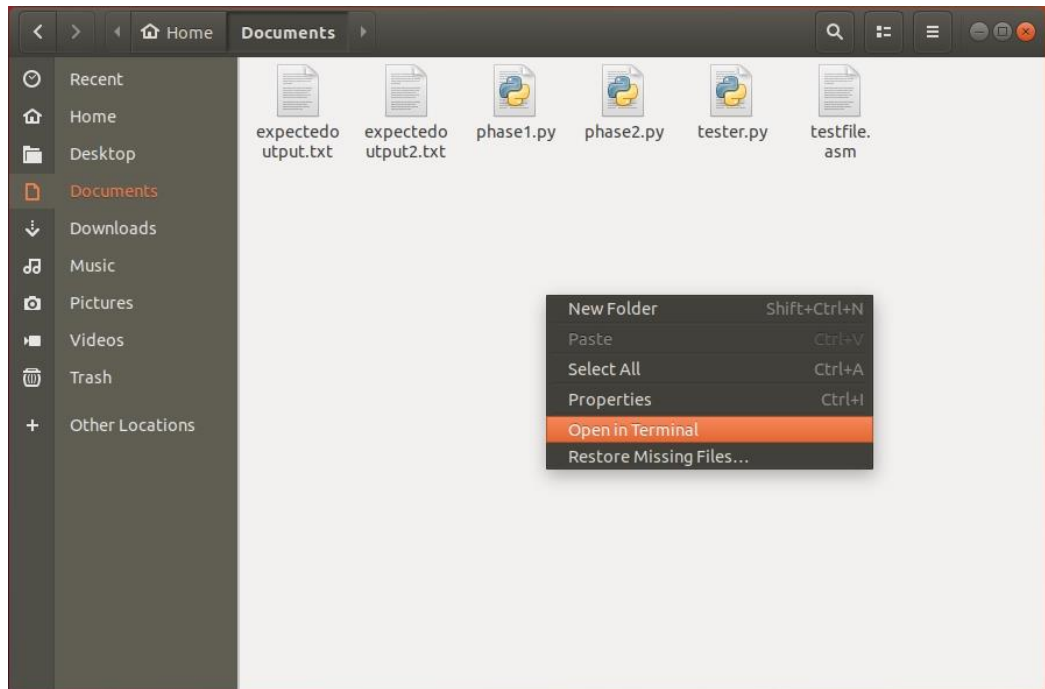
- These codes are some key parts of the for loop. The for-loop scan through each line of the test file and then match it with the machine code. The logic is like this. In each loop, firstly use the first word (line [0]) to make the judgement to determine which type of the institution it is because different types have different machine code structures. Then it uses the first word again to make sub judgement, determining what kind of the institution structure it belongs to. Then based on the structure, it uses line [1], line [2] (sometimes line [3]) as keys to search for corresponding codes in the dictionaries as mentioned before in data structure parts. Then use the string methods in python to combine part codes together to get the machine code. Finally, it writes the machine code to the output file, finishing one loop.

8. Testing Methods:

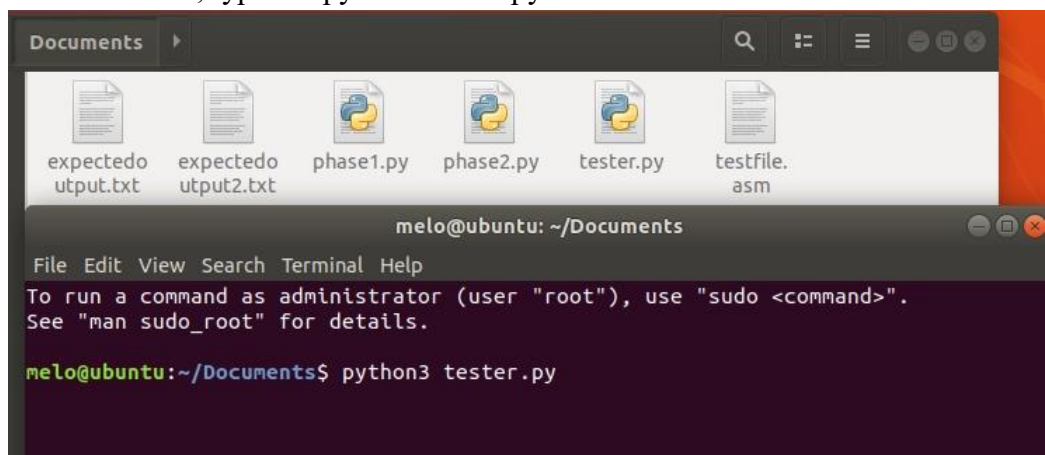
- ✧ System: Linux (Ubuntu 18.04 64bits)
- ✧ Environment requirements: install python 3 (python 2 is not supported)
- ✧ Steps:
 - i. Put tester.py, phase1.py, phase2.py, test file and expected output file in a single same directory:



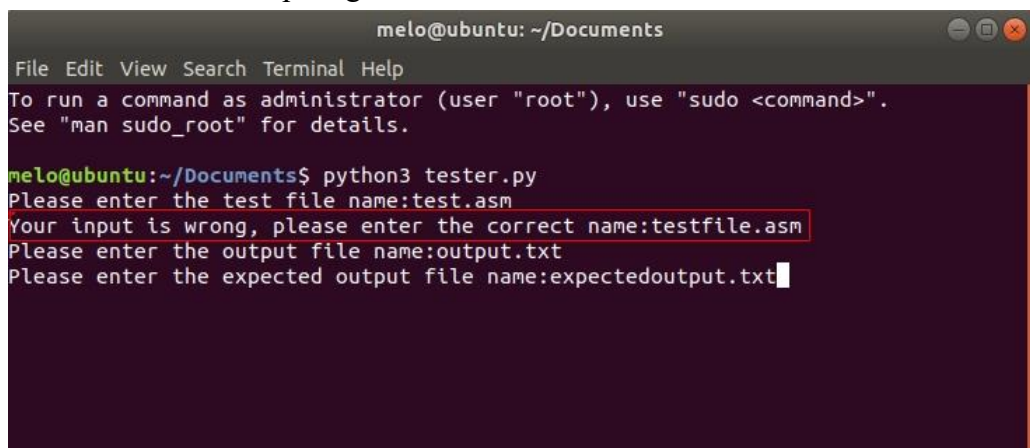
- ii. In the same directory you put the files, right-click and select “Open in terminal”, in order to limit the directory. (Notice: using the terminal for whole system or open terminal in other directory is not allowed, otherwise the program will not be able to find the matching files. If you insist to do so, you have to considering limit the directory by yourself.)



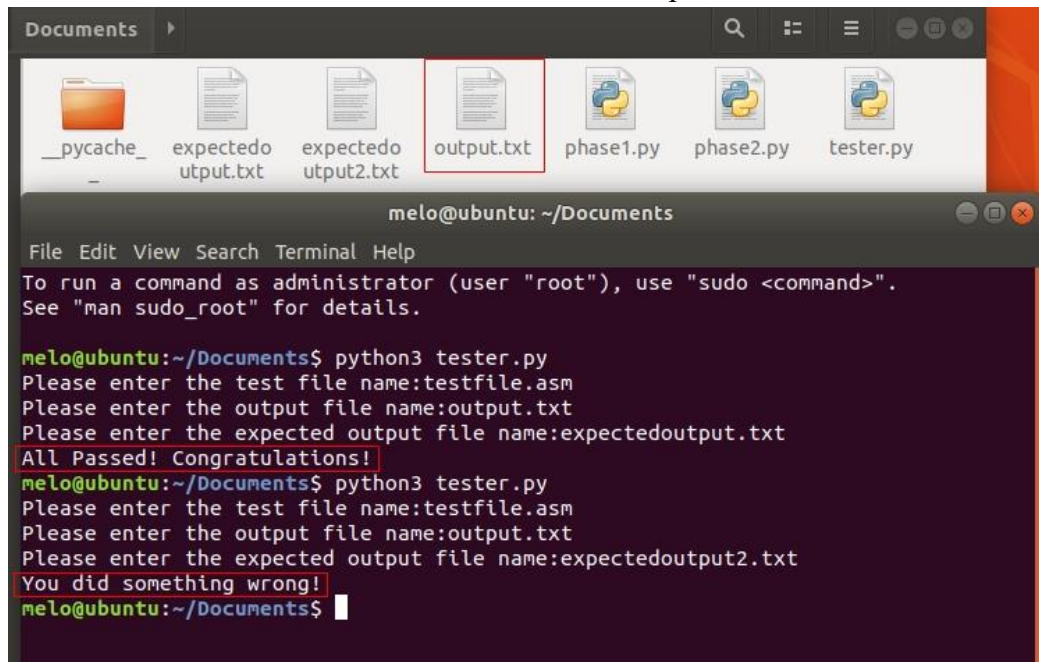
- iii. In the terminal, type in “python3 tester.py”.



- iv. Then the program will ask the user to input the names of test file, output file and expected output file. Please input them. The program will check the input. If it finds that the test file or expected output file not exists in the same directory, it will ask the user to input again until it is correct.



- v. Finally, the program will produce an output file, and print the comparison result on the screen to tell the user whether the assembler pass the test.



The screenshot shows a Linux desktop environment. At the top, a file manager window displays the contents of the 'Documents' directory. The files listed are: '_pycache_', 'expectedo utput.txt', 'expectedo utput2.txt', 'output.txt' (highlighted with a red box), 'phase1.py', 'phase2.py', and 'tester.py'. Below the file manager, a terminal window is open, showing the execution of a Python script named 'tester.py'. The terminal output is as follows:

```
melo@ubuntu: ~/Documents
File Edit View Search Terminal Help
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

melo@ubuntu:~/Documents$ python3 tester.py
Please enter the test file name:testfile.asm
Please enter the output file name:output.txt
Please enter the expected output file name:expectedoutput.txt
All Passed! Congratulations!
melo@ubuntu:~/Documents$ python3 tester.py
Please enter the test file name:testfile.asm
Please enter the output file name:output.txt
Please enter the expected output file name:expectedoutput2.txt
You did something wrong!
melo@ubuntu:~/Documents$
```