

CSC3050 Project 2 Report: Design of a MIPS Simulator

杨宇凡 117010349

1. Programming Language: Python
2. Basic Logic:
 - 1) Design proper data structures to represent register, memory and PC pointer.
 - 2) Design functions to imitate MIPS functions, which does operations on the registers and memory.
 - 3) Design function to imitate MIPS syscall, which enables the program to get the input from user, print the output and malloc memory to store data.
 - 4) From the user get the machine code file name, read it and store the instructions to the memory.
 - 5) According to the PC to get the corresponding instructions stored in memory, and execute it by the functions designed before. After the execution of the function, PC will be changed, and then the next instruction will be executed until the PC points to empty memory or it is over the memory instruction size.
3. Data Structure Explanations:

✧ Register:

[illegible]

The 32 normal registers are represented by a register list, and the functions use the index of each register to get and modify the data stored in each register. For example, `register_list [2]` represents `$v0` register, and `register_list [31]` represents `$ra` register. It is very convenient. `$zero` register is initialized by 32 bits “0”, and all other registers are initialized by empty string.

There are two special register, Lo and Hi, which are listed separately and be initialized by empty string.

✧ PC Pointer:

PC=0

PC pointer is initialized by 0, and it is a global variable and will be changed

after each function is executed.

✧ Memory:

```
instruction_length=2**12
memory_size=2**20
memory=[]
for i in range(0,memory_size):
    memory.append("")
```

Memory is represented by a very large list. The memory size is 1MB, and there are 4KB is divided to store instructions. The content in each memory address is initialized by empty string. The index of every element in the list represents the address of the memory.

4. Functions Explanation:

✧ Universal Functions:

```
#This function returns the 2's complement of the given binary string
def twos_complement(binary): ...

#This function transform the binary to decimal
#The last parameter decides whether to a signed integer or not signed
def binaryToInt(binary,signed=0): ...

#This function transform the decimal to binary
def intToBinary(number,length): ...

#This function can extend the binary to the expected length
#The last parameter decides whether the extension is signed or unsigned
def extend(binary,length,signed): ...

#This function can directly do the adding between two binary strings
#The third parameter decides whether there's an overflow check
#The last parameter decides whether need to do signed completion
def binary_add(add1,add2,overflow_check,signed_completion=1): ...
```

These functions are widely applied in the MIPS functions.

Twos_complement do the 2's complement of a binary string for sub operation, binaryToInt transform a binary to integer, and the signed parameter decides whether the binary is signed or unsigned, intToBinary transform an integer to a binary with the expected length, extend function can extend a binary to the expected length which supports both signed and unsigned methods, and binary_add can directly add two binary strings together without any transformation. intToBinary and binary_add support overflow check, which

can find most overflow situations during the execution of MIPS program.

✧ MIPS Functions:

```
def MIPS_add(rd,rs,rt): ...  
  
def MIPS_addu(rd,rs,rt): ...  
  
def MIPS_sub(rd,rs,rt): ...  
  
def MIPS_subu(rd,rs,rt): ...  
  
def MIPS_mult(rs,rt): ...  
  
def MIPS_multu(rs,rt): ...  
  
def MIPS_div(rs,rt): ...  
  
def MIPS_divu(rs,rt): ...  
  
def MIPS_mflo(rd): ...  
  
def MIPS_mfhi(rd): ...  
  
def MIPS_mtlo(rs): ...  
  
def MIPS_mthi(rs): ...  
  
def MIPS_and(rd,rs,rt): ...  
  
def MIPS_or(rd,rs,rt): ...  
  
def MIPS_nor(rd,rs,rt): ...  
  
def MIPS_xor(rd,rs,rt): ...  
  
def MIPS_sll(rd,rt,sa): ...  
  
def MIPS_srl(rd,rt,sa): ...  
  
def MIPS_sra(rd,rt,sa): ...
```

These functions execute the corresponding MIPS functions just as they are named. They mainly change the data stored in register list, memory list and PC pointer. For those jumping or linking functions, PC will go the specified #address, and PC will increase 4 in other functions

✧ Syscall:

```
def MIPS_syscall():
    global PC
    v0=binaryToInt(register_list[2])
    a0=register_list[4]
    a1=register_list[5]
    #print integer stored in a0 register
    if v0==1:
        print(binaryToInt(a0,1))
    #print integer stored in memory, the memory address was stored in a0 register,
    #and until the terminated character
    if v0==4:
        string=""
        address=binaryToInt(a0,0)
        count=0
        while not binaryToInt(memory[address+count],0)==10:
            string=string+chr(binaryToInt(memory[address+count],0))
            count+=1
        print(string)
    #get the input integer from user, and then read and store it in v0 register
    if v0==5:
        integer=input("Please enter an integer:")
        while not integer.replace("-", "").isdigit():
            integer=input("Your input is wrong, please enter again!")
        register_list[2]=intToBinary(integer,32)
```

.....

```
    if v0==9:
        allocate_size=binaryToInt(a0,0)
        allocate_memory=[1]
        check_list=[0]
        while check_list!=allocate_memory:
            allocate_memory=[]
            check_list=[]
            address=random.randint(instruction_length,memory_size-allocate_size)
            for i in range(0,allocate_size):
                check_list.append("")
                allocate_memory.append(memory[address+i])
            register_list[2]=intToBinary(address,32)
```

.....

This program supports 1,4,5,8,9,10,11,12 syscalls. These syscalls could be divided into three types. Type 1 asks the user to input something, and store it to the corresponding register or memory. Type 2 print out the data stored in some register or memory. Type 3 is syscall 9, which is like the malloc in C and allocate a piece of memory for using. In python, we use random function to allocate a piece of memory, and it will check whether this piece of memory is empty automatically. If not, it will allocate the memory again until it is empty.

5. Procedure Explanations:

✧ Write Instructions into Memory:

```
#get the machinecode file name from user, and open it
input_file_name=input("Please enter the machinecode file name:")
while not os.path.exists(input_file_name):
    input_file_name=input("Please enter correct name:")
input_file=open(input_file_name,"r")

#Modify and examine the input file
#if there's no problem, write the instructions into memory
#for further execution
count=0
for line in input_file:
    line=line.replace("\n","").replace("\t","").replace(" ", "")
    if (not line.isdigit() or len(line)!=32) and line!="":
        print("The input file is wrong.")
        os._exit(0)
    if not line=="":
        memory[count]=line[0:8]
        memory[count+1]=line[8:16]
        memory[count+2]=line[16:24]
        memory[count+3]=line[24:32]
    count+=4
```

In this procedure, the program will first ask the user to input the file name of machine code file name. Then it will modify each line in the machine code file and examine whether the file contains the machine file. If not, the program will exit directly. Otherwise, it will write the instructions into the memory in order for further execution.

✧ Execution:

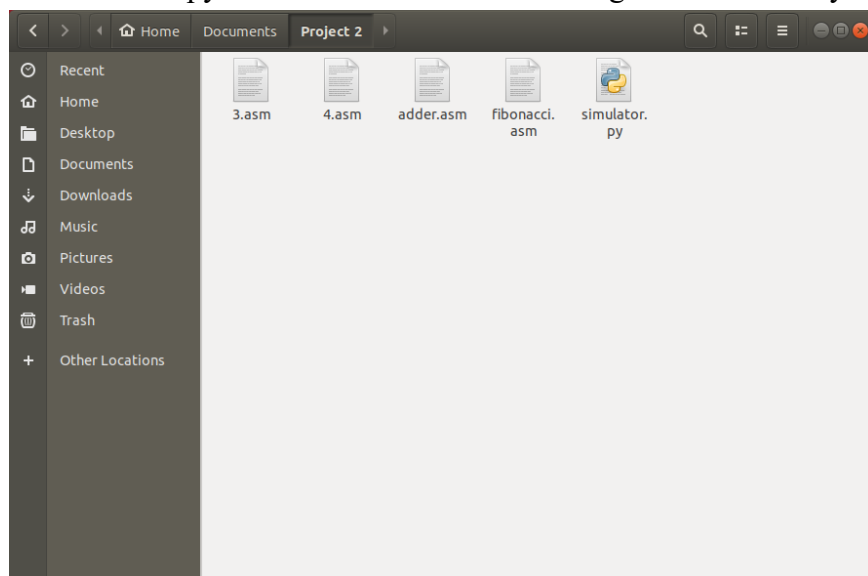
```
while not (memory[PC]==" " or PC>=instruction_length):
    #get corresponding instruction
    instruction=memory[PC]+memory[PC+1]+memory[PC+2]+memory[PC+3]
    opcode=instruction[0:6]
    #classify the instruction according to the opcode
    if opcode in R_opcodes:
        rs=instruction[6:11]
        rt=instruction[11:16]
        rd=instruction[16:21]
        sa=instruction[21:26]
        function_code=instruction[26:32]
        #Do the corresponding operation according to function code
        if function_code=="000000":
            MIPS_sll(rd,rt,sa)
```

For the execution part, it will first check whether the PC pointer is within the legal range. If the memory PC point to is empty or out of the instruction range, the program will stop. Then it will use the address from PC pointer to get the instruction, and then classify the instruction to different group according to the opcode, and then execute the corresponding function related to opcode or function code. Then one execution is finished, it will enter the while loop again according to the value of PC pointer.

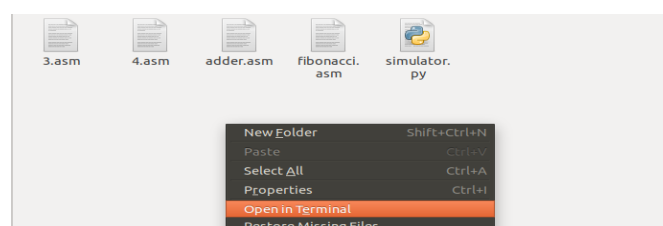
6. Testing Methods:

- ✧ System: Linux (Ubuntu 18.04 64bits)
- ✧ Environment requirements: install python 3 (python 2 is not supported)
- ✧ Steps:

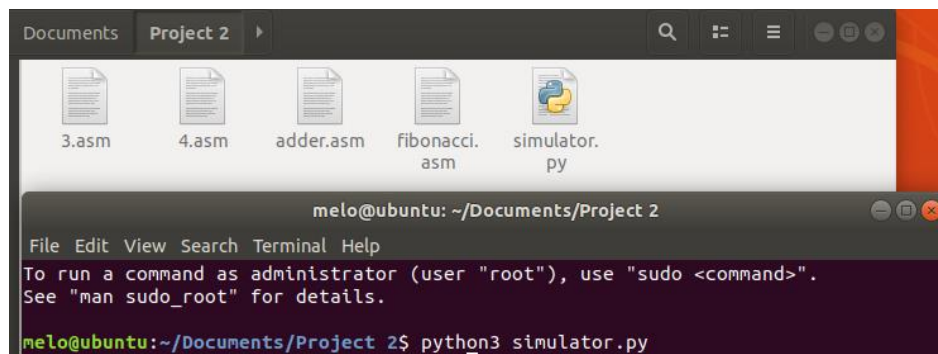
i. Put simulator.py and machine code file in a single same directory:



- ### 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.)

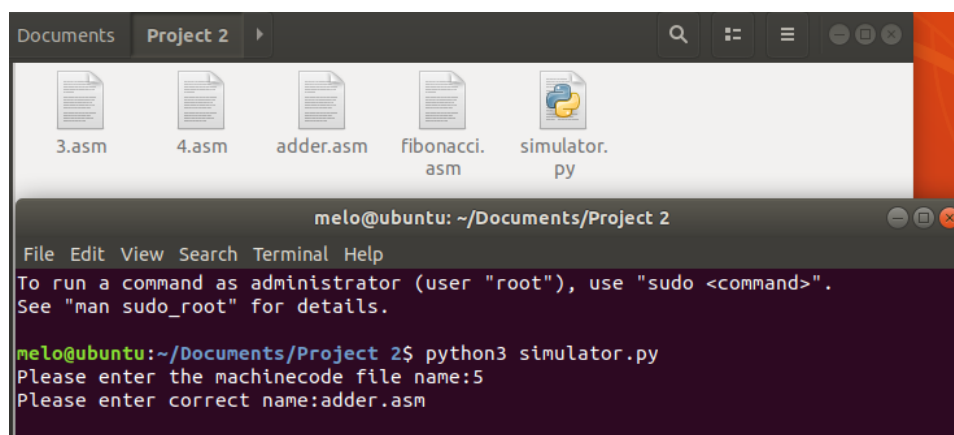


- iii. In the terminal, type in “python3 simulator.py”.



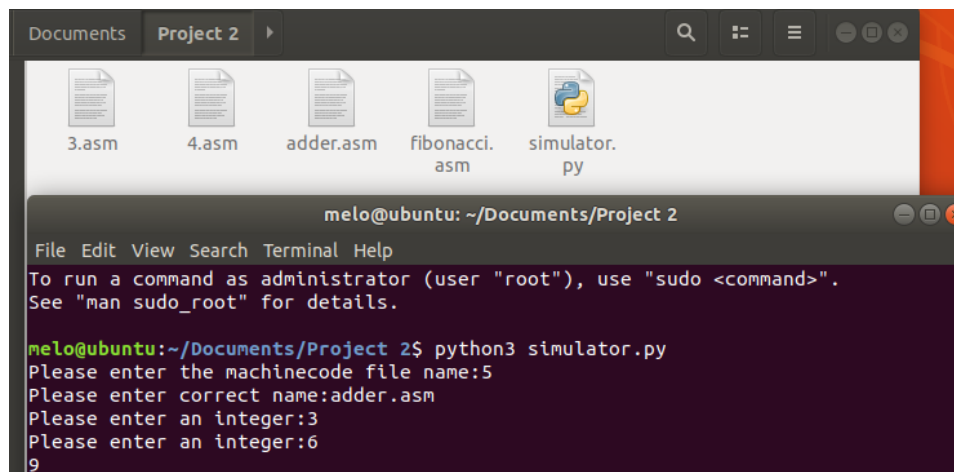
```
melo@ubuntu: ~/Documents/Project 2
File Edit View Search Terminal Help
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.
melo@ubuntu:~/Documents/Project 2$ python3 simulator.py
```

- iv. Enter the correct machine code file name, if the input file name is wrong and the program can't find it in the same directory, it will ask the user to input the machine code file name again.



```
melo@ubuntu: ~/Documents/Project 2
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/Project 2$ python3 simulator.py
Please enter the machinecode file name:5
Please enter correct name:adder.asm
```

- v. Finally, it will execute the corresponding instructions.



```
melo@ubuntu: ~/Documents/Project 2
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/Project 2$ python3 simulator.py
Please enter the machinecode file name:5
Please enter correct name:adder.asm
Please enter an integer:3
Please enter an integer:6
9
```