Project 2 Report

1. Main idea

The target of this project is to write a MIPS simulator that simulates the execution of a binary file. Specifically, this program will initiate the **memory (which is a list in this case)** and the **registers memory (which is a dictionary)**, store the static data from the input .asm file, load the machine code from the input .txt file, and start the execution according to each line of the machine code.

After finishing one instruction, the program counter will +4 to proceed to the next instruction (each instruction is 4 bytes, one item in the memory is 1 byte, so each instruction takes up 4 locations in the memory), or jump to certain address. The program ends when reach the end of .text section or the program terminates itself.

Note that there are other input files: a test.in file is read when calling syscall read operations, a test.out file is written by the program when doing syscall write/print operations, and a checkpoints.txt file contains the check points that the program needs to generate dump file for the current memory and registers.

2. Implementation

2.1 Initiation:

```
# allocate memory
memory_size = 10 * 1024 * 1024              # 6 MB
memory = ['00000000' for i in range(0, memory_size)]

# register values are integers
reg_mem = {
    "zero": 0, "at": 0, "v0": 0, "v1": 0,
    "a0": 0,   "a1": 0, "a2": 0, "a3": 0,
    "t0": 0,   "t1": 0, "t2": 0, "t3": 0,
    "t4": 0,   "t5": 0, "t6": 0, "t7": 0,
    "t8": 0,   "t9": 0, "s0": 0, "s1": 0,
    "s2": 0,   "s3": 0, "s4": 0, "s5": 0,
    "s6": 0,   "s7": 0, "k0": 0, "k1": 0,
    "gp": 0x508000,   "sp": 0xA00000-1, "fp": 0xA00000 -1, "ra": 0,
    "pc": 0x400000,  "hi": 0,   "lo": 0
}
```

To simulate a memory start from 0x400000, a full memory of 10 MB is created with the first 6MB left empty (in the form of str '00000000'), and the program counter (PC) is set to 0x400000. In this way, the ambiguity is eliminated.

2.2. Execution

In this part, the program first parses the instruction according to their op/funct code. After certifying the type of instruction (R, I, J), the program specifies the called registers by their register number and the operation according to the function code. Then, the program will evoke the corresponding function by looking up the dictionaries, with function codes as keys and function names as values. Parameters such as rd, rs, rt, immediate, target address are passed into the function.

Illustration for evoking functions:

```
# R type
if code_string.startswith('000000'):
    execute_time += 1
    rs = code_string[6:11]
    rt = code_string[11:16]
    rd = code_string[16:21]
    sa = code_string[21:26]
    func = code_string[26:]
    RFunctions[func](rs, rt, rd, sa)
```

```
RFunctions = {

    '100000': ADD, '100001': ADDU, '100100': AND, '011010': DIV,
    '011011': DIVU, '001001': JALR, '001000': JR, '010000': MFHI,
    '010010': MFLO, '010001': MTHI, '010011': MTLO, '011000': MULT,
    '011001': MULTU, '100111': NOR, '100101': OR, '000000': SLL,

    '000100': SLLV, '101010': SLT, '101011': SLTU, '000011': SRA,
    '000111': SRAV, '000010': SRL, '000110': SRLV, '100010': SUB,
    '100011': SUBU, '001100': SYSCALL, '100110': XOR

}
```

2.3 Function Specification

Each instruction is represented by a function. In this way, simulating execution equals to calling a series of functions by proper sequence. **The memory (list) and register value (dictionary) will be used and updated inside each function**, and the program counter (also in the dictionary) will +4 or change to the target address at the end of the function to maintain the execution.

For the arithmetic functions, since the program stores the decimal integer in the register memory, the program **strictly follows the signed/unsigned logic to store the value when doing operations** (However, in real CPU, there is no such concepts of signed/unsigned number, the CPU will the arithmetic operation from LSB to MSB and detect the overflow). For example, the program will turn any negative number (immediate) into positive when doing unsigned operation.

In the syscall function, **python os** is introduced to complete the I/O instructions.

2.4 type casting

In the program, all the data are stored in the memory are in the form of string, which contains an 8-digit binary number. To evaluation the data as characters, the program employs chr(int(binary str, base=2)) method, and to store the characters as 'binary string', it just uses bin(ord(char))[2:].zfill(8); to evaluate a date piece as decimal numbers (such as the string contains immediate number), it simply uses (str, 2).

In some cases, we need to convert a negative integer to binary string, the program just uses negative num & 0xffff to make it in 2's complement form of 16 digits, and similarly it can handle 8 digits or 32 digits. In order to transform a negative binary string, the program just employs two functions int_overflow() and str_signed_extend() as below:

```
def int_overflow(val):
    maxint = 2147483647
    if not -maxint-1 <= val <= maxint:
        print('Overflow Error!')
        val = (val + (maxint + 1)) % (2 * (maxint + 1)) - maxint - 1

    return val

def str_sign_extend(stri):
    if stri[0] == '1':
        num = int(stri, 2) - (2 ** (len(stri)))
    elif stri[0] == '0':
        num = int(stri, 2)
    else:
        print("str_sign_extend error! String is: ", stri)
        return
    return num
```

3. Summary

This project roughly finished the MIPS simulator, however, due to the time limit, I have to give up final implementation debugging. As a result, there are some functions that **are not supported** in the program, namely **lwr, lwl, swr, swl, and sbrk.** There can also exist more bugs in other functions, so that it cannot pass some test cases.