Report for CSC3050 Project 1

0. Introduction to the Usage of my Codes

All the codes are in the file *simulator.py.* Please run this file when testing. I have packed my codes together with the testcases provided on bb, each testcase contains one folder. I have made 6 copies for my codes, one copy for each testcase folder, (so that TAs can run each testcase directly). Another one is outside the testcase folders, as a backup.

I made some modifications on the directory of testcase folders. Originally, the correct dumping output (.bin) are in a separate folder. I drag them out so that now all files are under the same directory. **Please maintain this file structure when running** The ways to run the code is the same as that described in the project description,

python simulator.py *test*.asm *test*.txt *test*_checkpts.txt *test*.in *test*.out

Please note that the keyword *test* should be changed into the file name of the testcase. This command means that test files and source codes should be under same directory, so that's why I modify the structure of test folders.

1. How MIPS programs are executed in computers

Computers implement the MIPS program instructions by a machine cycle. Machine cycle is based on the fact that all of the codes are stored in the memory, and has their own address. This address can be retrieved by the program counter (PC), which is likely to a pointer in C, or the index of list in python. The machine cycle is described as below, as a loop.

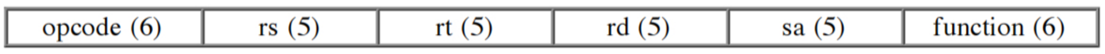
a. The computer loads the line of instruction that PC is "pointing at".
b. The computer decodes and executes the instruction loaded.
c. If the instruction involves branching or unconditional jump, the PC will go to the target address contained in the MIPS code. Else, it directly increment by 4, since every MIPS instruction are stored in unit of "word", which equals to 4 bytes in

memory. PC is maintained so that it always points to the next instruction to be implemented (then we start the new “cycle”, the cycle terminates unless we receive signals or we have reached the end of our program.

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



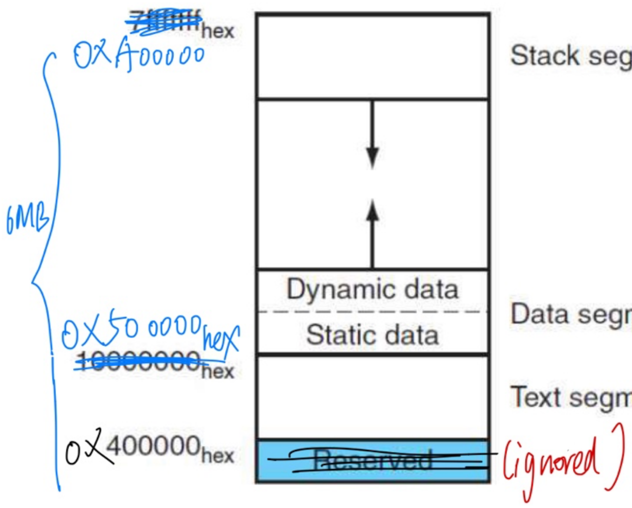
opcode (6)	rs (5)	rt (5)	rd (5)	sa (5)	function (6)

All R-type instructions use the same opcode 000000. Registers *rs*, *rt* corresponds to the first, and second register source operand, and register *rd* is the destination operand. *“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 links them together using the above sequence to form the machine code.

2. Implementation of MIPS simulator

2.1 Main Idea of Implementation

We assume that our codes only contain .data section and .text section. We first do memory simulation. In python, we simulate it as a list, each element of our list is a word, which will be discussed later. Below is a figure of what a real computer memory look like.



In this project, our memory is 6MB in size. It consists of text segment (1MB in size), the data segment and the stack segment. Our memory is assumed to start from byte address 0x400000 in hexadecimal, and the size of the text segment is designated to be 1MB. The data segment is right on top of the text segment, therefore, the data segment should start at byte address 0x500000 in hexadecimal. The data segment consists of static data and dynamic data. The dynamic data is also right on top of the static data, and it grows upward. The top of the static data needs to be stored since we need to implement the *sbrk()* system call, which involves the movement of the program break.

2.2 Data loading

We assume .data always appears before .text. 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 the list structure that we implement for memory. The data loading process stopped until a "*.text*" symbol is detected.

We give a brief introduction of how we implemented our memory list. As is required, the data is arranged piece by piece. A whole block of 4 bytes is assigned to a piece of data even it is not full. If the piece of data is more than 4 bytes, we will allocate another block of 4 bytes until it fits in all elements of this piece of data. Therefore, in memory simulation, we come of the idea of using a list, with each element of the list representing the information of 4 bytes. Therefore, the total size of our list is 6MB / 4 = 6 * (2 ^ 18).

To record the information stored in the data, we take recording the information of an English word as an example. The word "*Test*" should be recorded as an element of the list, since each letter contains one byte. The ASCII value for letter T, e, s, t are 84, 101, 115, 116. We stored the element and can dump it in little Endian, where the most significant bit should be at the rightmost position. We store the information of the word as a number concatenate by the ASCII code for each letter, as

84 + 101*(16^2) + 115*(16^4) + 116*(16^6) = 1953719636

The number 1953719636 then can be used to store in the memory list, containing information of the word "Test". The 4 ASCII codes can be shown if we dump it bytes by bytes using the *to_bytes()* instructions with parameter *byteorder = "little"*.

But we need to pay attention to the size of different data types. For example, in MIPS, each character for *.ascii* type occupies one byte, while each number in .word occupies 4 bytes, therefore, the value of a .word element is indeed an element of the memory list. Moreover, their storing way differs, for example, *.ascii* is stored by big Endian, while .half is stored by little Endian. To ensure the dumping result is correct, we need to adopt different ways to encapsulate them into a number as we did above.

Similarly, we can also store the machine code of the MIPS instruction in this way, by simply considering each binary 32-bit code as a number, which is also an element stored in the memory list.

2.3 MIPS instruction decoding and executing

After loading data, we can do the decoding and executing process of the MIPS code. In project 1, we have converted the MIPS instruction into binary code. We know that basically, there are 3 types of MIPS instructions, namely R-TYPE, I-TYPE and J-TYPE. Their structure is as below.

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

R-Type

opcode (6)	rs (5)	rt (5)	immediate (16)

I-Type

opcode (6)	target (26)

J-Type

We can decode them based on their structures and most importantly, their opcode and function code. As we can see, the *opcode* is always the first 6 bits of the binary machine code, the *rs* field is at position [6:10] the *rt* field at [11:15], the *rd* field is at [16:20], *sa* field at [21:25], *function code* field at [26:31], *immediate* field at [16:31], target field at [6:31]. We extract these fields in advance for further decoding.

Although the instructions may not contain all of the fields above, they can still be used to compare and judge what instruction this machine code represents. We basically judge them by their opcode and function code, and use an *if-elif* structure to go into the detailed implementation for each instruction. In the invoking of each function, we can then be able to use those fields for the parameters of our function execution, such as the detailed register *rs, rt* and *rd* number to retrieve operand numbers from and do the *add* operation.

The implementation of each function is done following the exact content of this MIPS code. We store the register in another list, independent from the 6MB memory we simulated. Element of the list at index *i* represent the data stored in register number *i*, and each element of the list again corresponds to actural data with size 4 bytes. Therefore, when implementing the add operation *add rs, rt, rd*, we simply do the below

Register_list[rd] = register_list[rs] + register_list[rt]

Where *Register_list[rd]* stores the data in register *rd*. Our register list consists of 35 elements, which are 32 general purpose registers and register *HI* and *Lo*, and register *PC*. Register HI and Lo are coprocessors used in division and multiplication, register PC contains the absolute byte address of the program counter. Every time after we finish the execution of an instruction, we need to increment the absolute value of pc, i.e., register_list[PC], by 4, unless there's direct jump or branch operations.

3 Data Storage of the variables

3.1 Data Storage for the memory

A list, with size 6MB / 4 = 6 * (2 ^ 18), detailed implementation is mentioned in section 2.2. The machine code is stored from index 0 of the list, the data is stored from index 0x100000 of the list

3.2 Data Storage for the memory

A list, with 35 elements. Element of the list at index *i* represent the data is stored in register number *i*, and each element of the list corresponds to the actual data with size of 4 bytes.

3.3 data storage for PC relative address

Since we are using list to simulate our memory, the absolute address of the PC (starting from 0x400000) is seldom needed (only used in register PC and *jalr* instruction) and cannot be used to index data or instructions from the memory. We may use to relative address for indexing, where

PC_relative = (PC_absolute – int("400000",16)) // 4

Therefore, without jumping or branching operations, after every instruction, PC_relative should be incremented by 1, while PC_absolute should be incremented by 4.

4. Python Environment

(Conda base) Python 3.9.15

The *requirements.yml* is also included in my submission file.