Explanation for the whole project

The project should be run on a linux system. Just use the makefile included to compile the project. Then use the following instructions to run the program.

```
./simulator test.asm test.txt test_checkpoints.txt test.in test.out
```

test.asm: An input test file of assembly codes, which is used for static data loading.

test.txt: An input file for assembled binary codes, which is used for binary code loading.

test_checkpoints.txt: An input file indicating where memory and registers' snapshots should be dumped. This file is just for testing the correctness of your program and grading. **See document** `Checkpoints specifications.pdf` **for more details**.

test.in: An input file storing inputs for some read-related I/O operations (read_int, read_char, read_string).

test.out: The name of the output file storing the outputs for print-related I/O operations.

For other files included in this zip, functions.cpp is responsible for all the functions of the project, main.cpp calls those functions and perform the assembling process.

My understanding for the MIPS simulator

MIPS simulator is a self-contained simulator that runs MIPS32 programs. It reads and executes assembly language programs. Mips simulator also provides a simple debugger and minimal set of operating system services.

The implementation of the project

The main function of the project is to build a MIPS simulator which simulate the execution of a binary file. To complete this goal, a few steps need to be done:

1. **Memory simulation & Register simulation**
2. **Read data/text in to the right place**
3. **Implement the function of every machine code in the mips instruction list**
4. **Use a machine cycle to perform the simulation**

(1) Memory simulation & Register simulation

I choose to use dynamic array to implement the memory and the register. The memory is set to array of char to do the bit-level manipulation. The register is set to array of int to access it directly by index. The size of memory is 6mb and the size of the register is 4*35 bytes. All the memory and registers are initialized to 0 except for fp, sp and gp.

It is important to notice that it is 6mb memory I am really simulating. But the virtual memory is actually 10mb in size. So there should be a mapping between the real mem and the virtual mem. In my proj, I record the start address of the 6mb memory as char* SixMbMemory and every time I refer to the virtual memory, I use "SixMbMemory-0x400000+*VirtualMemory*". The *VirtualMemory* refer to the simulated address in the virtual address.

(2) To read data/text to the memory the following steps are taken:

1. use a fstream to open the .asm file

2. use getline to read the file line by line. If reach "data:" then start reading the data until the "text:" is found for the first time. This process filter all the unnecessary information like spaces by using a , label for data out and only read in the useful information like the content of the data and its type. In this process, the ascii and asciiz type need to have different treatment

compared with other data type because the existing of " ", which will make the filtering process take different strategy to deal with.

It is important to notice that getline will transform the \n inside of a string into \\n this is a deadly bug but really hard to notice. My solution for it is to do the transformation again, if in the string find a char '\' followed by a 'n' then replace them with '\n'.

3.After the unnecessary information is filtered, then read all the filtered data into the memory in the same order as it was in the .asm file according to their type. The ascii and asciiz type take the big endian format and the other types take the small endian format. All the type will be aligned into a multiple of 4 bytes even it is not. The start of the static data is 0x500000 in the virtual memory and the dynamic data pointer is set to the end of the static data.

4. Then read the text into the memory. In the common MIPs simulator, it usually read in the assembly code and transform it in to machine code and then read it into the memory. In this project, to ensure the correctness, ta provide the machine code for us. So this project do not do the assembly process but only read in the machine code into the memory. The start of the machine code is at the 0x400000 in the virtual memory. Each line of code take 4bytes.

(3) Implement the function of every machine code in the mips instruction list

The machine codes are divided into 3 type, R, I and J.

The R types all start with 000000. I use this character to distinguish R type. For the elementary instructions in R type, they are distinguished according to the last 6 bit. For their functions, I refer to the appendix A of the textbook to implement them. Syscalls are distinguished according to the v0 value. I implement the open,read and write by directly calling the linux api, which can be

found by googling “man read/write/open”. The other syscalls are implemented according to the site:

https://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html.

For the I type, their first six bit are different. So just use this feature to distinguish them.

The J type have different first six bit too. So the same way as I type.

The different part of the machine code has different meaning. So it is important to correctly cut them and convert the binary into decimal. I use substr to cut and bitset to convert the binary information into decimal information.

(4) Use a machine cycle to perform the simulation

To inplement the machine cycle, I use a while loop. The loop will continue as long as the syscall exit is not called, which will change the value of a flag and break the loop.

Inside the machine cycle:

Because the machine code is stored byte by byte in the memory. The first thing to do is to fetch them out into a 32bit string machine code. To do this, I use bitset<8> to convert each byte to an 8bit binary string and connect the 4 byte together to get a 32-bit machine code string. After that, increment the PC by 4 in order to fetch the next instruction. Then use the *process* function to do what the machine code is meant to do (which already implemented in (3))