Project Report

CSC 3050 MIPS Simulator

Wei Wu

118010335

March 20, 2020

The School of Science and Engineering

香 港 中 文 大 學 (深 圳)
The Chinese University of Hong Kong, Shenzhen

1. Understandings of the Project

This project is about writing a simulator for the MIPS CPU. Briefly speaking, the project can be divided into two parts. The first part is using a MIPS assembler to assemble an input MIPS assembly language file, which is basically the same as Project 1. The second part is using a MIPS simulator to simulate the memory, registers, and execution. Here, static data come from the original MIPS assembly language file, while instructions (machine code) come from the first part.

2. The Implementations of the Project

2.1. The Big Picture Idea

In general, the MIPS simulator needs to simulate the memory, registers, and execution using high-level language.

In the very beginning, the simulator calls the assembler to assemble the code.

In terms of memory, the simulator firstly allocates 6MB of memory to be the simulated main memory. From bottom to top, the main memory is divided into text section (1MB), static data section, dynamic data section, and stack. Since we are running the simulator on a 64-bit personal computer, the actual memory address is 64-bit long. As MIPS is a 32-bit architecture, we need to map each 64-bit address to a 32-bit address.

In this project, the 32 general registers, $hi, and $lo are simulated. Each of them is allocated 4 bytes of memory. Note that the registers are independent of the 6MB main memory.

Before execution, the simulator needs to load the machine code to the text section, and then the static data to the static data section. Then, it initializes the registers and the PC.

Following the machine cycle, the simulator fetches one instruction from the memory, increases PC by 4, decodes the instruction, and then executes the instruction. A specific function is written for each instruction (including syscall).

2.2. Problems and Solutions

2.2.1. How to compile a C++ program using command lines?

Using the command line "g++ -std=c++11 main.cpp MIPS.cpp ins.cpp data.cpp -o simulator" in the terminal.

2.2.2. How to get the input filename and the output filename?

In C++, it is feasible and convenient to get the input and output filenames from the command line. To be more specific, two parameters, int argc and char** argv, are added to the main() function. This way, after compiling, the user can use a command line of the form "xxx input.asm output.txt" to specify the input and output filenames. Here, "input.asm" corresponds to argv[1], while "output.txt" corresponds to argv[2]. Moreover, since there are always one input filename and one output filename, argc should always be 3. Otherwise, an error message will be generated.

2.2.3. How to allocate and maintain 6MB in the heap to simulate the main memory?

First, use malloc() to allocate 6MB of memory in the heap. In MIPS, the smallest addressing unit is 1 byte. Therefore, we use uint8_t * to store the memory address. The implementation is as follows:

```
uint8_t * base_addr = (uint8_t *) malloc(SIM_MEM_SIZE);
```

2.2.4. How to map a 64-bit memory address to a 32-bit memory address?

Since the main memory is only 6MB large, a 32-bit address is sufficient. After malloc(), we map the 64-bit `base_addr` to the 32-bit 0x00400000. Whenever we need to access a 32-bit memory address, we calculate the difference between this address and 0x00400000. Then, add this difference to `base_addr` to obtain the corresponding 64-bit address.

2.2.5. How to allocate and maintain memory space for each register?

Still, we can use malloc() to allocate 4 bytes for each register. Since each time a register is loaded from or stored to, all the 4 bytes are visited, we use int32_t * to store the memory address.

For ease of visiting the registers, the addresses of the 32 general registers are stored in a vector<int32_t *>.

2.2.6. How to load the instructions into the memory?

When using input file stream to deliver the instructions, each line is a string. Therefore, we must convert each string into a 32-bit unsigned integer. Then, we can load these unsigned integers into memory using a pointer.

2.2.7. How to load static data to memory?

For those .asm files containing a .data section, we need to deal with the data line by line. Each line is basically of the form “name: .type data #comment”. According to the data type, the program calls the corresponding function to load the data into the static data section using a pointer. Note that each piece of data must occupy one or more full blocks (4 bytes), even if it uses only part of a block.

2.2.8. How to manage dynamic data?

After loading all the static data, record the address of the top of the static data section. In other words, maintain a pointer to the top of the dynamic data. Whenever a piece of dynamic data is input through syscall, store it in the memory using the pointer.

2.2.9. After decoding an instruction, how to call the corresponding function?

We can build a map<int, void*> from the opcode/funct of the instruction to the corresponding function. To implement this map, we have to define a new type first. For example,

```
typedef void (*rFunc)(int32_t *, int32_t *, int32_t *, int);
```

Then, define the map as follows:

```
static map<int,rFunc> rFuncMap;
```

In this way, we can call the corresponding function simply using one line of code:

```
rFuncMap[funct](rs, rt, rd, shamt);
```

2.2.10. How to do sign-extension and zero-extension?

Zero-extension is for an unsigned integer. For a 16-bit immediate, convert it to uint16_t. Sign-extension is for a signed integer. For a 16-bit immediate, convert it to a int16_t.

2.2.11. How to detect overflow?

For ADD, ADDI, and SUB, use a buffer of type int64_t to store the result. Then compare it with INT_MIN and INT_MAX. For DIV, overflow occurs only when (dividend = INT_MIN and divisor = -1) or (divisor = 0).

2.2.12. How to deal with overflow?

Terminate the program.

2.2.13. How to deal with trap?

Terminate the program.

2.2.14. How to implement syscall 13-16?

First, create a new fstream object in the heap. Then, use the fstream to open the input file. Next, store the address of the fstream in a static vector<fstream *>. Meanwhile, store the corresponding index (starting from 0) in register $a0. Later we can find that fstream in the vector<fstream *> using that index.

Before terminating the program, we must free() those fstream pointers.

2.2.15. What to do before terminate the program?

Free() all the memory space allocated for the main memory, the registers, and the fstream objects. This is crucial.

2.3. Data Structures

2.3.1.
```
typedef void (*rFunc)(int32_t *, int32_t *, int32_t *, int);
typedef void (*iFunc)(int32_t *, int32_t *, int);
typedef void (*jFunc)(uint32_t);
static map<int,rFunc> rFuncMap;
static map<int,iFunc> iFuncMap;
static map<int,jFunc> jFuncMap;
```

The map from opcode/funct to the corresponding function.

2.3.2. `vector<int32_t *> regs;`

The collection of the 32 general registers.

2.3.3. `vector<fstream*> fileDescriptor;`

The vector to store fstream*, which correspond to files opened by SYSCALL 13-16.

2.4. Functions

2.4.1. `int main(int argc, char** argv)`

This function takes the input filename and the output filename from command lines. Then,

0. Call the MIPS assembler

1. Simulate the memory

2. Simulate the registers

3. Maintain a PC

4. Maintain a code section

5. Maintain a data section
6. Maintain a stack section
7. Simulate the execution

2.4.2. void **initRegs**(uint32_t stack_bottom)

This function initializes the registers, such as $zero, $sp.

2.4.3. void **terminate**(bool success)

This function firstly free() all the memory space allocated for the main memory, the registers, and the fstream objects. Then it terminates the program, meanwhile outputs the current memory address.

2.4.4. void **buildFuncMap**()

This function builds the function map for R-type, I-type, and J-type instructions.

2.4.5. uint32_t **stringToUnsigned**(string code)

This function is used to convert a string containing machine code to a 32-bit unsigned integer. For instance, "00111100000001000000000001010000".

2.4.6. int **loadText**(uint8_t * base_addr, istream & input)

This function loads the machine code to the text section of the simulated memory.

2.4.7. string **binaryToString**(uint32_t value)

This function converts a 32-bit unsigned integer to a string.

2.4.8. void **fetchIns**()

The function fetches one instruction from the memory pointed by PC, and then increases PC by 4.

2.4.9. void **decodeIns**(string ins)

This function decodes the instructions, and sorts the instructions into R-type, I-type, and J-type.

2.4.10. int **stringToSigned**(string code)

This function converts a string to a signed integer. Note that code[0] corresponds to the sign bit.

2.4.11. void iType(string ins)

This function decodes an I-type instruction and call the corresponding function.

2.4.12. void **rType**(string ins)

This function decodes an R-type instruction and call the corresponding function

2.4.13. void **jType**(string ins)

This function decodes a J-type instruction and call the corresponding function.

2.4.14. uint32_t * loadData(ifstream & source, uint32_t * data_addr)

This function loads the static data to the memory. The source is a MIPS assembly language file.

2.4.15. bool **overflow**(int64_t value)

Overflow detection for ADD, ADDI, and SUB.

2.4.16. void **trap**()

This function is called whenever a trap exception is raised. It terminates the program.

2.4.17. void **r_SYSCALL**(int32_t * rs, int32_t * rt, int32_t * rd, int shamt)

This function handles syscalls.

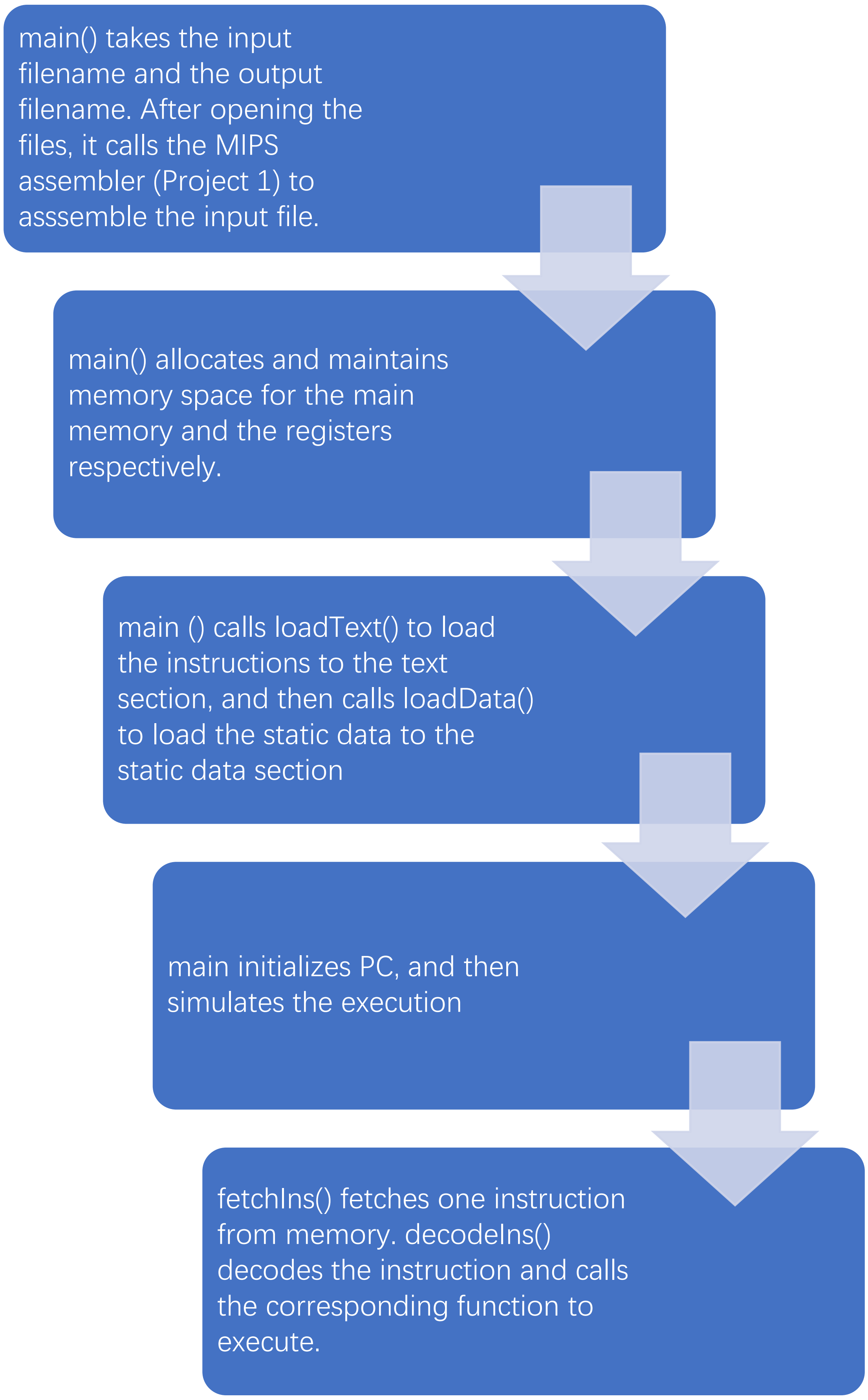
2.5. Discussion & Further Improvement

In this project, I mainly focus on how to simulate the MIPS CPU. If the input assembly language code is totally correct, the program runs nicely. However, in real life, there might be all kinds of errors in the input file. To make things easier, the assembler and the simulator should be able to identify the types and locations of errors. In this project, I did write some code to identify some of the possible errors, but not all of them. Thus, to make the simulator more robust, I think I should implement more of error detection.

Moreover, I used mainly software approaches to implement the simulator. For example, the hardware overflow detection is comparing the last two carry-outs. While I just used a 64-bit integer to store the result, and then compare it with INT_MIN and INT_MAX. However, the result should be the same.

Besides, since the whole project is procedure-oriented, the functions have a lot of parameters. I think the project can be changed into object-oriented, where each instruction is treated as an object. Then, parameters like rt, rs, rd can be changed into data fields, functions like ADD, SUB, DIV, can be changed into methods.

2.6. Simplified Procedures


main() takes the input filename and the output filename. After opening the files, it calls the MIPS assembler (Project 1) to asssemble the input file.
main() allocates and maintains memory space for the main memory and the registers respectively.
main () calls loadText() to load the instructions to the text section, and then calls loadData() to load the static data to the static data section
main initializes PC, and then simulates the execution
fetchIns() fetches one instruction from memory. decodeIns() decodes the instruction and calls the corresponding function to execute.

3. Testing

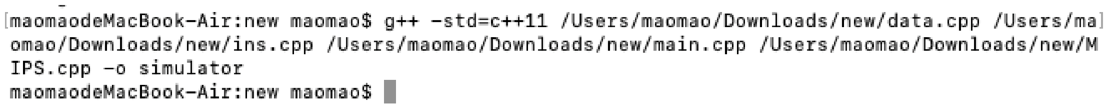
3.1. Environment:

macOS
g++
C++11

Note:
To compile:
g++ -std=c++11 main.cpp MIPS.cpp data.cpp ins.cpp -o simulator
To execute:
simulator input.asm output.txt
Acknowledgement:
The testing environment (MacBook) is provided by Mr. Yu Mao.

3.2. Result:

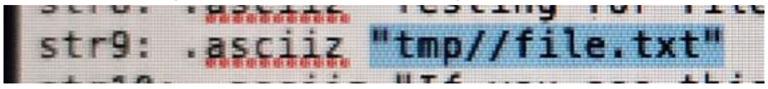
Compiling:



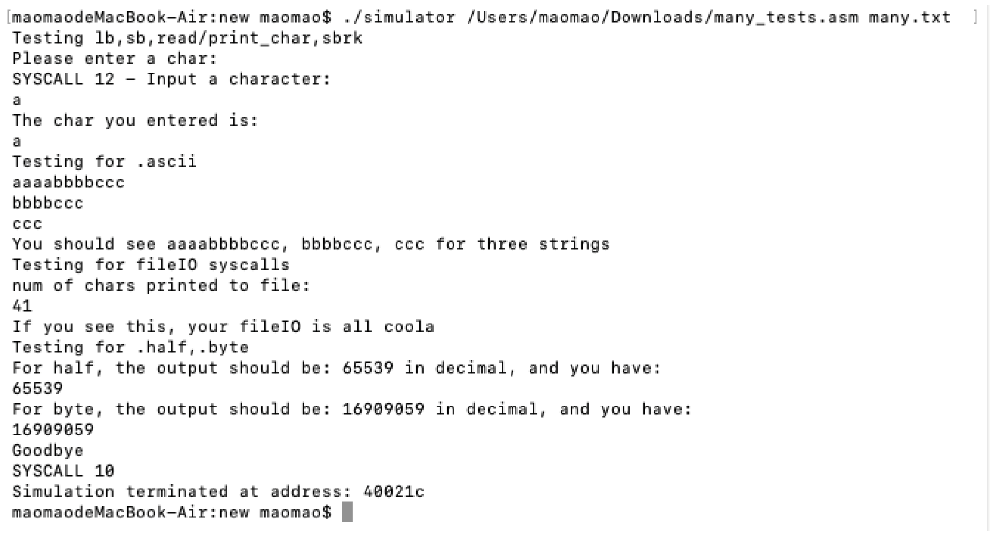
```
[maomaodeMacBook-Air:new maomao$ g++ -std=c++11 /Users/maomao/Downloads/new/data.cpp /Users/ma]
omao/Downloads/new/ins.cpp /Users/maomao/Downloads/new/main.cpp /Users/maomao/Downloads/new/M
IPS.cpp -o simulator
maomaodeMacBook-Air:new maomao$
```

Executing:

1. **Manytests.asm**
   Note: the original filename is changed.

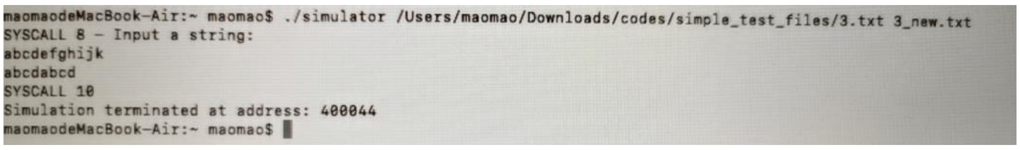


```
str9: .asciiz "tmp//file.txt"
```



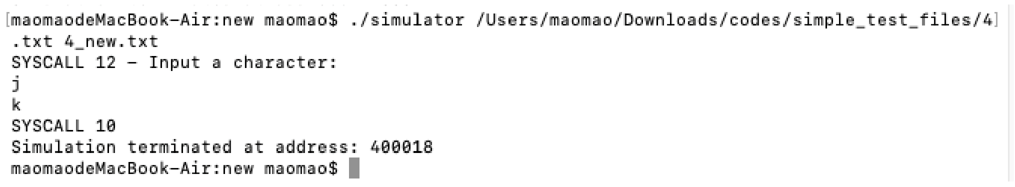
```
[maomaodeMacBook-Air:new maomao$ ./simulator /Users/maomao/Downloads/many_tests.asm many.txt ]
Testing lb,sb,read/print_char,sbrk
Please enter a char:
SYSCALL 12 - Input a character:
a
The char you entered is:
a
Testing for .ascii
aaaabbbbccc
bbbbccc
ccc
You should see aaaabbbbccc, bbbbccc, ccc for three strings
Testing for fileIO syscalls
num of chars printed to file:
41
If you see this, your fileIO is all coola
Testing for .half,.byte
For half, the output should be: 65539 in decimal, and you have:
65539
For byte, the output should be: 16909059 in decimal, and you have:
16909059
Goodbye
SYSCALL 10
Simulation terminated at address: 40021c
maomaodeMacBook-Air:new maomao$
```

2. 3.asm



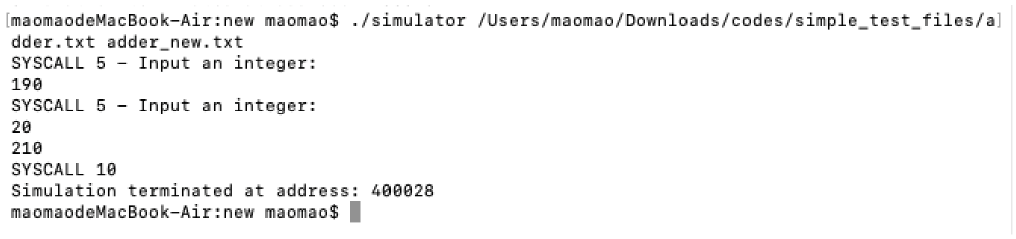
```
maomaodeMacBook-Air:~ maomao$ ./simulator /Users/maomao/Downloads/codes/simple_test_files/3.txt 3_new.txt
SYSCALL 8 - Input a string:
abcdefghijk
abcdabcd
SYSCALL 10
Simulation terminated at address: 400044
maomaodeMacBook-Air:~ maomao$
```

3. 4.asm



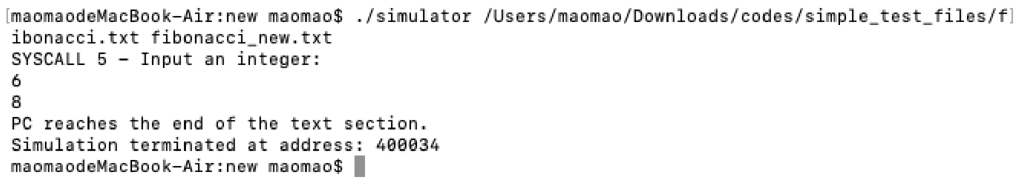
```
[maomaodeMacBook-Air:new maomao$ ./simulator /Users/maomao/Downloads/codes/simple_test_files/4]
 .txt 4_new.txt
 SYSCALL 12 - Input a character:
 j
 k
 SYSCALL 10
 Simulation terminated at address: 400018
 maomaodeMacBook-Air:new maomao$
```

4. Adder.asm



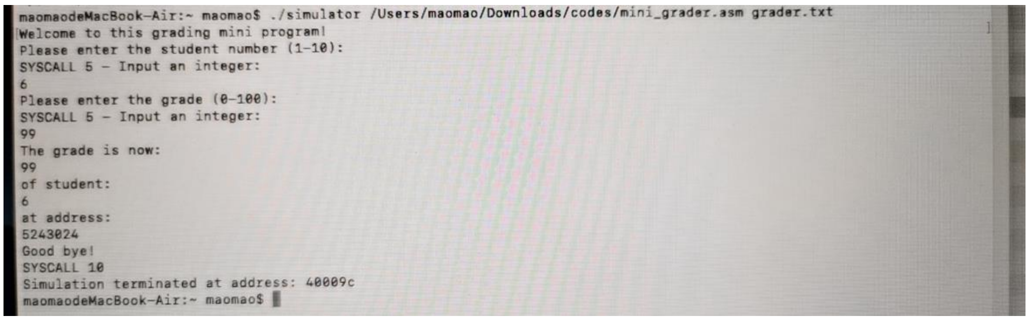
```
[maomaodeMacBook-Air:new maomao$ ./simulator /Users/maomao/Downloads/codes/simple_test_files/a]
 dder.txt adder_new.txt
 SYSCALL 5 - Input an integer:
 190
 SYSCALL 5 - Input an integer:
 20
 210
 SYSCALL 10
 Simulation terminated at address: 400028
 maomaodeMacBook-Air:new maomao$
```

5. Fibonacci.asm



```
[maomaodeMacBook-Air:new maomao$ ./simulator /Users/maomao/Downloads/codes/simple_test_files/f]
 ibonacci.txt fibonacci_new.txt
 SYSCALL 5 - Input an integer:
 6
 8
 PC reaches the end of the text section.
 Simulation terminated at address: 400034
 maomaodeMacBook-Air:new maomao$
```

6. Mini_grader.asm



```
maomaodeMacBook-Air:~ maomao$ ./simulator /Users/maomao/Downloads/codes/mini_grader.asm grader.txt
[Welcome to this grading mini program!
Please enter the student number (1-10):
SYSCALL 5 - Input an integer:
6
Please enter the grade (0-100):
SYSCALL 5 - Input an integer:
99
The grade is now:
99
of student:
6
at address:
5243024
Good bye!
SYSCALL 10
Simulation terminated at address: 40009c
maomaodeMacBook-Air:~ maomao$
```