MIPS-Simulator

Introduction

In this project, I finish building a program, MIPS simulator, to execute MIPS (assembly language) file with C++/C, and I understand the procedure of MIPS program's execution in the machine. The program takes MIPS file (*.asm) as input. The program firstly translates each instruction into machine code (Assembler), and then, distinguishes and executes the machine code (Simulator).

File Structure

```
|—include
        |—assembler.h
|—src
        |—assembler.cpp
        |—simulator.cpp
|—test
        | —***
|—Makefile
|—Report.pdf
```

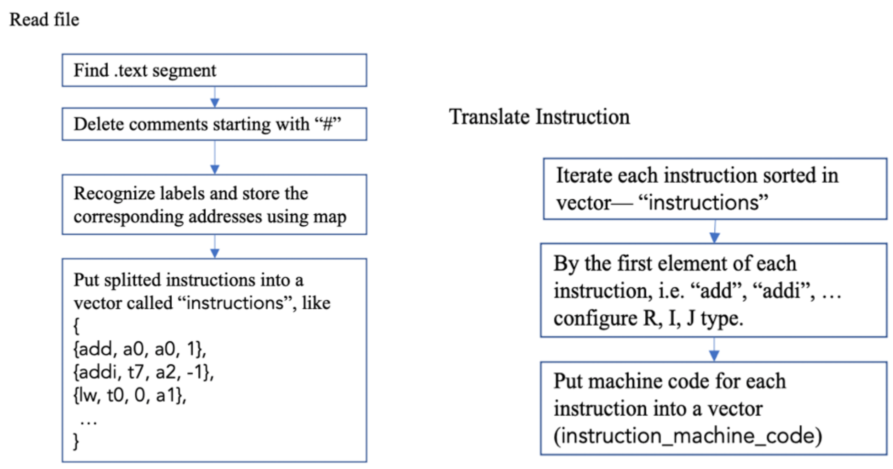
Assembler

1. Purpose

   Read MIPS file and translate ".text" segment into machine code.

2. Implementation. (`class Assembler`)

   2.1 Overview


Read file
Find .text segment
Delete comments starting with "#"
Recognize labels and store the corresponding addresses using map
Put splitted instructions into a vector called "instructions", like
{
{add, a0, a0, 1},
{addi, t7, a2, -1},
{lw, t0, 0, a1},
…
}
Translate Instruction
Iterate each instruction sorted in vector— "instructions"
By the first element of each instruction, i.e. "add", "addi", … configure R, I, J type.
Put machine code for each instruction into a vector (instruction_machine_code)

   2.2 Problems and Solutions (Implementation details)

a) Multi-spaces

⇨ Solution: A function `split()` is used to ignore spaces.

b) Handle different situations of label's position, i.e., in different line or same line with instruction.

⇨ Solution: Using PC (program counter) to record the absolute address of each instruction. Only if the line contains a valid instruction, the PC will increase by 4. The PC always points to the next instruction. Thus, no matter where the label is, the corresponding address is point to the same instruction.

c) Translate different format instructions

⇨ Solution: Define functions

```
void translate_R_type(int op, int rs, int rt, int rd, int shamt, int funct);
void translate_I_type(int op, int rs, int rt, int addr);
void translate_J_type(int op, int addr);
```

Each instruction has its own op code and the code of executed registers can be obtained from `unordered_map<string, int> regMap`, e.g. {"at", 0}

d) Translate machine code of label.

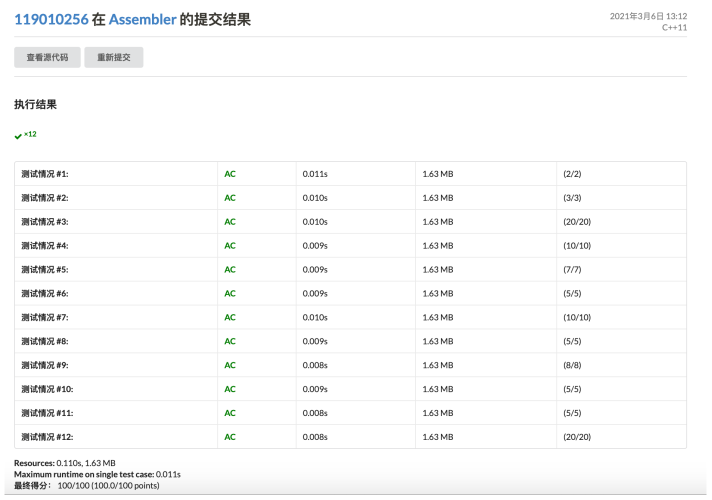
⇨ Solution: J-format and I-format has different addressing mode.

For J-format, except for `op` code (6 bits), the `relative address` (26 bits) is also required for translating machine code. Firstly, determine `j label` or `j address`. If `j label`, get the absolute address from map. Merge the `op` and `address[27:2]`.

For I-format, the machine code should include offset. For example, `bne $s0, $s1, Exit`, since `pc+4+offset*4 = absolute address of Exit`, `offset` can be calculated. For `bne` with offset (not label), offset is obvious.

3. Tests and Results

The assembler is successfully tested via OJ platform. The result can be seen as following:



119010256 在 Assembler 的提交结果

2021年3月6日 13:12
C++11

执行结果

✓ ×12

测试情况 #1:	AC	0.011s	1.63 MB	(2/2)
测试情况 #2:	AC	0.010s	1.63 MB	(3/3)
测试情况 #3:	AC	0.010s	1.63 MB	(20/20)
测试情况 #4:	AC	0.009s	1.63 MB	(10/10)
测试情况 #5:	AC	0.009s	1.63 MB	(7/7)
测试情况 #6:	AC	0.009s	1.63 MB	(5/5)
测试情况 #7:	AC	0.010s	1.63 MB	(10/10)
测试情况 #8:	AC	0.009s	1.63 MB	(5/5)
测试情况 #9:	AC	0.008s	1.63 MB	(8/8)
测试情况 #10:	AC	0.009s	1.63 MB	(5/5)
测试情况 #11:	AC	0.008s	1.63 MB	(5/5)
测试情况 #12:	AC	0.008s	1.63 MB	(20/20)

Resources: 0.110s, 1.63 MB
Maximum runtime on single test case: 0.011s
最终得分： 100/100 (100.0/100 points)

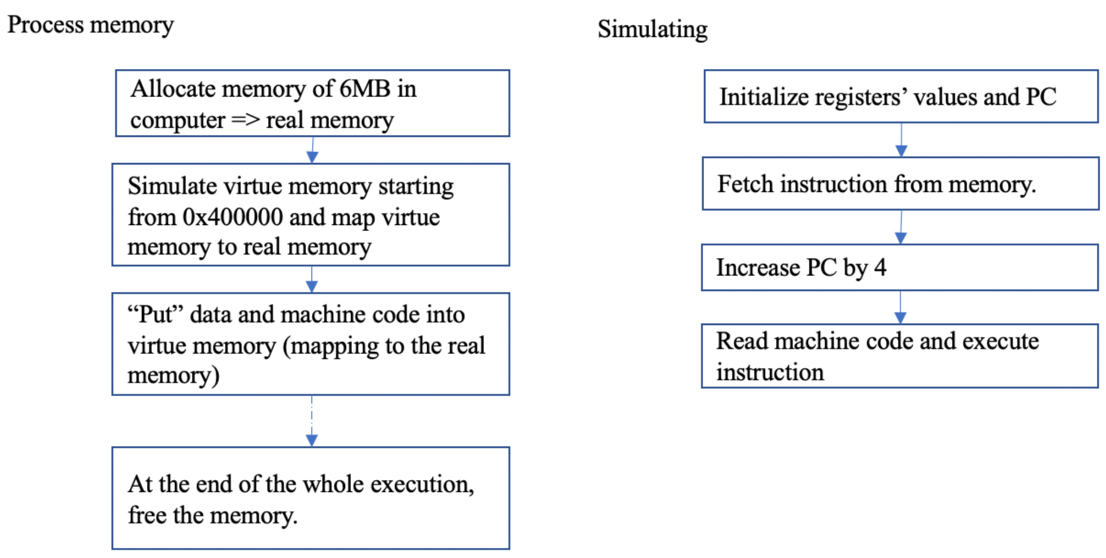
Figure 1: Test result on OJ platform

Simulator

1. Purpose

Simulator put data from .data segment and machine code that translated from .text segment into virtue memory. By reading machine code, the simulator can properly execute the input MIPS file. (*Note: all outputs write to *.out.)

2. Overview


Process memory
Allocate memory of 6MB in computer => real memory
Simulate virtue memory starting from 0x400000 and map virtue memory to real memory
"Put" data and machine code into virtue memory (mapping to the real memory)
At the end of the whole execution, free the memory.
Simulating
Initialize registers' values and PC
Fetch instruction from memory.
Increase PC by 4
Read machine code and execute instruction

3. Problems and solutions (Implementation details)

   a) Memory declaration relating to putting different type of data.

      ⇨ Solution: Considering the smallest addressable "byte", a **pointer** pointing to a char can be used to access the corresponding memory. The virtue memory can map to real memory via function

```
char *mapMem(u_int32_t virtual_mem)
```

b) Put data with different types

⇨ Solution: Define functions

```
void readData(string filename);  // only handle .data segment
void putData(string mode, string line);
```

In `readData`, the data type and data are recognized.

In `putData`, `mode` is received as "ascii", "asciiz", "half", "word", "byte". By recording the current virtue memory that going to being put with data, the data will put into virtue memory by mapping to real memory.

`byte` and `char` can be easily put into memory. However, the pointer should be cast to a pointer to `int16_t` to put `half` or to `int32_t` to put `word`.

c) Simulating `viod readInstruction();`

⇨ Solution: PC counter is initialized as 0x400000, it always contains the memory address of the next instruction to be executed. During one machine cycle, the instruction firstly fetched from memory at the address stored in PC, and PC will increase by 4. Then, the instruction (in machine code) will be decoded into different control signals. The op (with func) code determines the format (R, I, J) of that instruction and what the exact instruction is. After knowing the information of the instruction, the instruction can execute along with register and data memory. Finally, send and write the result back to memory.

d) Sign extension relating to machine code (offset / immediate)

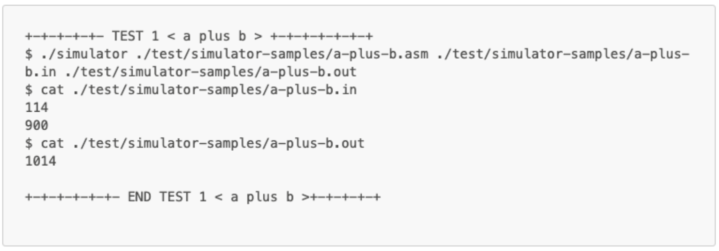
⇨ Solution: For example, `addi $a2, $a2, -4` (machine code: 0x20c6fffc). There will be cast-rejection between 16bits and 32bits. To avoid this, I take lowest 16bits, i.e., 0xfffc. Sign extension function check the 16th bit, if that is 1, the output will be 0xfffffffc, otherwise, it is also 0xfffc (zero extension).

e) Debugging

⇨ Solution: With aid of MARS, firstly, check whether the machine code is correct or not. Then, check whether the PC is performing in correct order or not. If problems still exist, check function’s implementation.

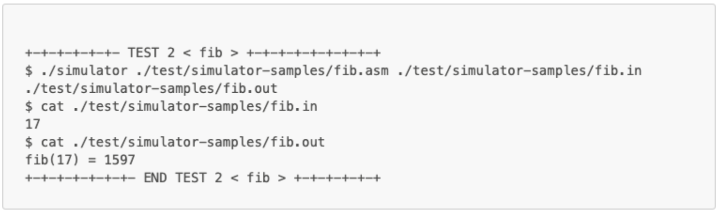
4. Tests and Results

   The simulator is successfully tested through three tests. The following instruction can be executed to show the test results.

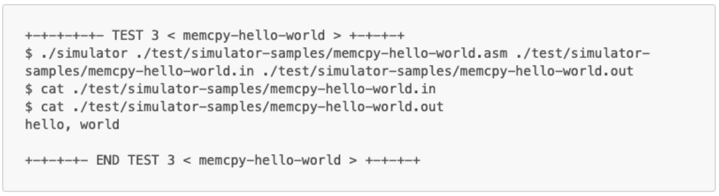


```
+-+-+-+-+- TEST 1 < a plus b > +-+-+-+-+-+-+
$ ./simulator ./test/simulator-samples/a-plus-b.asm ./test/simulator-samples/a-plus-
b.in ./test/simulator-samples/a-plus-b.out
$ cat ./test/simulator-samples/a-plus-b.in
114
900
$ cat ./test/simulator-samples/a-plus-b.out
1014

+-+-+-+-+-+- END TEST 1 < a plus b >+-+-+-+-+
```



```
+-+-+-+-+-+- TEST 2 < fib > +-+-+-+-+-+-+-+-+
$ ./simulator ./test/simulator-samples/fib.asm ./test/simulator-samples/fib.in
./test/simulator-samples/fib.out
$ cat ./test/simulator-samples/fib.in
17
$ cat ./test/simulator-samples/fib.out
fib(17) = 1597
+-+-+-+-+-+-+- END TEST 2 < fib > +-+-+-+-+-+
```



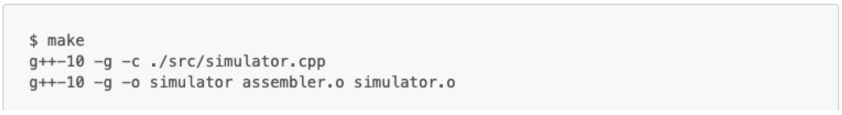
```
+-+-+-+-+- TEST 3 < memcpy-hello-world > +-+-+-+
$ ./simulator ./test/simulator-samples/memcpy-hello-world.asm ./test/simulator-
samples/memcpy-hello-world.in ./test/simulator-samples/memcpy-hello-world.out
$ cat ./test/simulator-samples/memcpy-hello-world.in
$ cat ./test/simulator-samples/memcpy-hello-world.out
hello, world

+-+-+-+- END TEST 3 < memcpy-hello-world > +-+-+-+
```

Conclusion

Through this project, I have a better understanding of numerous instructions and computer architecture. The assembly language is firstly translated by assembler to machine language that can be understood and executed by machine. Then, during one machine cycle, PC indicates where to fetch instructions in memory. The instruction determines the action of registers and data memory. For further implementation, the data memory can be improved to be well-organized. Besides, pipeline can be used to improve the operation.

Usage



```
$ make
g++-10 -g -c ./src/simulator.cpp
g++-10 -g -o simulator assembler.o simulator.o
```