



THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC3050

COMPUTER ARCHITECTURE

Project1

Author:
Shi Wenlan

Student Number:
119010265

March 19, 2021

1 Introduction

I wrote a C++ program that can compile and execute assembly language files. This program reads MIPS file, reads syscall input from *.in* file, and outputs syscall output to *.out* file.

2 Big picture thoughts and ideas

My program would execute MIPS files in four steps: initializing virtual memory, converting *.text* part into machine code and storing it in the text segment of virtual memory, storing *.data* part in the static data segment of virtual memory, and reading the machine code in the text segment and executing instructions.

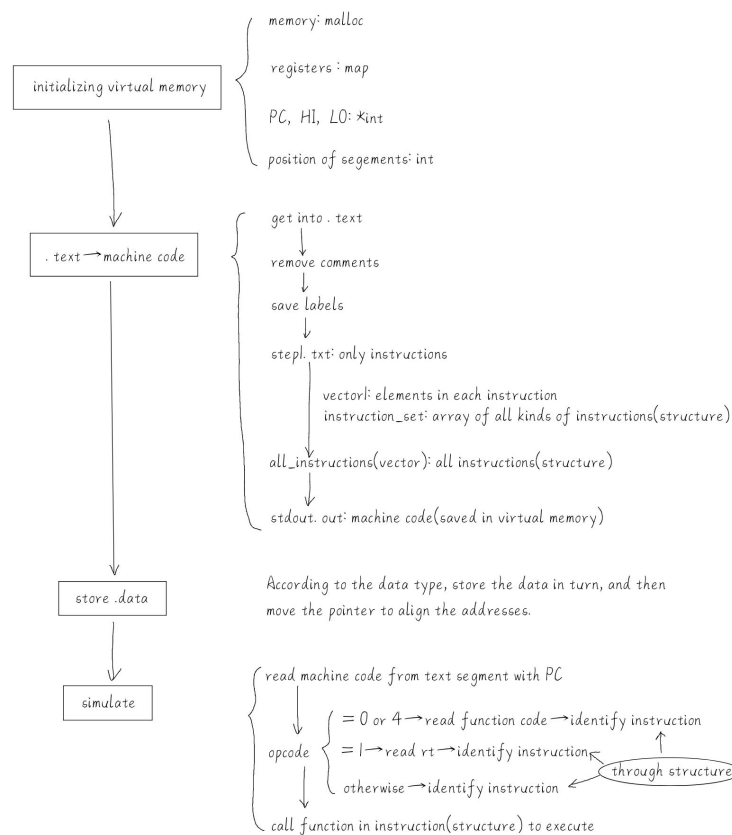


Figure 1: Big picture thoughts and ideas

In the first part, I used malloc to create a 6MB space in real memory for subsequent data storage. The integer pointer named *real_mem* pointed to the starting

point of this space, and stored the real address of the beginning of memory. Then, I used the array named *mem_address* to store the virtual address from 4MB to 10MB. And the virtual address storage of label in subsequent steps depends on this array to determine the virtual address. I also established a series of integer constants to mark the initial virtual address of each segment in virtual memory, and initialized the integer pointer PC to point to the initial virtual address of text segment. About registers initialization, I used two sets of maps, one was the one-to-one mapping between register names and register contents, and the other was the one-to-one mapping between register names and register addresses, so that the contents of designated registers could be accessed in subsequent processes. Except for \$sp, \$fp and \$gp, all other register contents were initialized to 0. In addition, for two special registers, HI and LO, I defined integer pointers outside the map range and initialize them to 0.

In the second part, the program would read asm file line by line, and confirmed to enter the *.text* part by recognizing *.text* and *.data*. After reading each line, it would check that whether it contained *#*. If it does, it means that there are comments in this line, and the comments after *#* should be deleted. Then, it would check that whether there was a colon. If there is, it means that there is a label in this line. The virtual address of the label in this line should be stored in the map named *labels* in one-to-one mapping with the name of the label. And the name of the label and colon in this line should be deleted. If this line is still not empty (it contains contents other than spaces and tabs), the rest is the instruction, which is passed into the file named *step1.txt*. After reading all the contents in asm file, *step1.txt* containing only instructions was obtained. Then, *step1.txt* was opened and read line by line. After reading each line, the components in the instruction separated by spaces, tabs, commas or brackets would be separated and stored in a vector. The first entry of the vector is the name of the instruction, which could be compared with *instruction_set* to find the corresponding instruction structure. Then other information stored in the vector would be read, and filled in the places where the values of the three parameters are stored in the structure in turn. Some instruction parameters are less than three, and the missing parts would use the initialized values. The processed instruction structure would be saved into a vector named *all_instructions*. The vector *all_instructions* containing all instruction information in asm file would be obtained after reading all contents in *step1.txt*. Then, each structure in *all_instructions* would be read in turn. After reading each structure, the string *base* composed of 32 zeros would be initialized, the decimal opcode stored in the structure would be converted into 6-bit binary and replace the first 6 bits of *base*. The format would be determined according to function code. if it is R-format, the last 6 bits of *base* would be replaced by function code, then the

names and values of three parameters would be read in turn, and the corresponding parts of *base* would be replaced by binary values. if it is I-format or J-format, the names and values of three parameters would be read directly in turn, and the corresponding parts of *base* would be replaced by binary values. The generated machine code would be saved into memory. After reading all the contents in *all_instructions*, the assembler part of asm file would be completed.

In the third part, the program would read the contents of asm file line by line again, this time it would enter the *.data* part, delete the comments of each line, and store the elements of each line in vector, which are similar to the second part. Each row of data would be read to obtain a vector, and the data in this vector would be read in turn. After reading the data type, the subsequent data were continuously stored in the static data segment according to the data type. If the address pointed by the pointer was not a multiple of 4 after storing the data, the pointer would be moved to a higher address which should be a multiple of 4 to ensure the address alignment for subsequent addressing operation. After storing all the data from *.data* section, the address pointed by the pointer at this moment would be marked as the starting address of dynamic data segment.

In the fourth part, I used PC pointer to point to the starting address of text segment, and read the machine code stored in the second part line by line through +4. After reading each line, the opcode would be identified first. If opcode is equal to 0 or 28, it indicates that it is an R-format instruction. The program would continue to identify function code, and then compare the two with *instruction_set* to determine the instruction structure corresponding to this line of machine code. If opcode is equal to 1, it indicates that it is a special I-format instruction. The program would continue to identify the value of rt, and then compare the two to *instruction_set* to determine the instruction structure. In other cases, the opcode corresponds to the instruction structure one by one, and the instruction structure can be determined by directly comparing *instruction_set*. Then, according to the parameter names recorded in the structure, the corresponding data would be sequentially read and stored in the structure, and the functions bound in the structure would be run. After reading all the machine code, the MIPS file simulation would be finished.

3 high level implementation ideas

About addressing: The 6MB memory created by *real_mem* can only be addressed according to word, so its unit is word. The *txt_end* I created is an integer variable, which marks the virtual address at the end of the current text segment,

and the unit is byte. When data needs to be stored at the current address, it is necessary to divide address by 4 before adding *real_mem*, so as to obtain the current real address. In the same way, *data_end* and *dynamic_end* respectively mark the virtual addresses at the end of the data segment and the dynamic data segment. In this way, I could replace byte at the specified position after reading a word and then save the new word back, thus completing byte-level operation. I created PC as an integer pointer, which, like *real_mem*, is addressed according to word, and its unit is word.

About string storage: When the sentence in the *.data* section is divided into elements, the characters are read by char. If the characters are double quotation marks, before the next double quotation mark is detected, all the characters read will be united as one element (excluding double quotation marks). During this period, pay attention to whether there is a backslash. If there is a backslash, the translation operation will be carried out according to the content of the next character. After reading the next double quotation mark, store this element in the vector, and continue reading characters to form the next element.

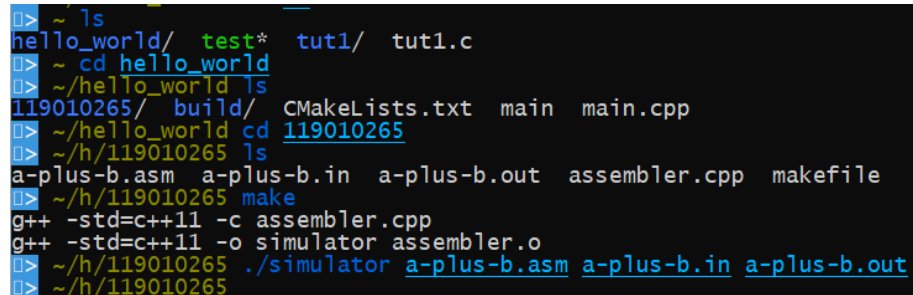
4 implementation details

I built a structure named *instruction*, which contained opcode, function code, instruction name, names of three parameters, values of three parameters, and function that used three parameters to execute instructions. The function code was used to identify the format of the instruction. The function code ≥ 0 indicates that the instruction is R-format, the function code = -1 indicated that the instruction is I-format, and the function code = -2 indicated that the instruction is J-format. All I-format instructions with opcode 1 have no rt input, and the position of rt stores a fixed number, which is used to identify instructions in combination with opcode. Therefore, the values of the three parameters were not all initialized to 0. In this case, the instruction would initialize the parameters at rt position to that specific number, for example, bgez is 1. Then, based on my structure, a 78-bit array *instruction_set* of instruction type was established. It contained the information of all instructions, including syscall, for subsequent cross-reference identification of instructions.

5 Operation manual

(1) Enter the folder containing *assambler.cpp* and *Makefile* in terminal. (2) Put *.asm* file and *.in* file for testing into this folder. (3) Enter *make* and press *enter*

button. (4) Enter *simulator*, *.asm*, *.in*, *.out* and press *enter* button.



```
~ ls
hello_world/ test* tut1/ tut1.c
~ cd hello_world
~/hello_world ls
119010265/ build/ CMakeLists.txt main main.cpp
~/hello_world cd 119010265
~/h/119010265 ls
a-plus-b.asm a-plus-b.in a-plus-b.out assembler.cpp makefile
~/h/119010265 make
g++ -std=c++11 -c assembler.cpp
g++ -std=c++11 -o simulator assembler.o
~/h/119010265 ./simulator a-plus-b.asm a-plus-b.in a-plus-b.out
~/h/119010265
```

Figure 2: operation example

6 Conclusion

This project made me fully understand the execution process of MIPS files, improved my proficiency in C++, and made me preliminarily understood the writing of makefile and the use of virtual machines.