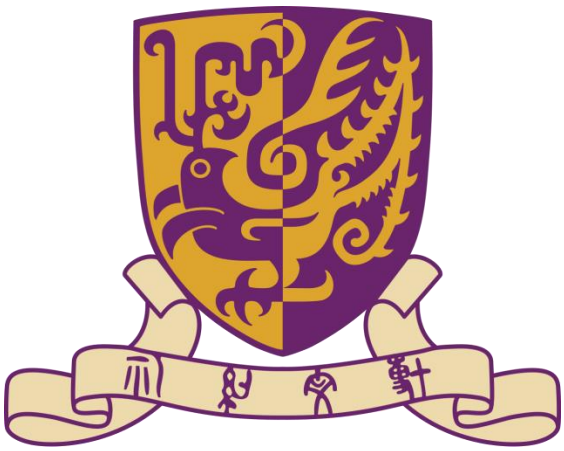


THE CNINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC3050

Computer Architecture

# Report for Project 2

*Author:*
Li Jingyu 李璟瑜

*Student Number:*
118010141

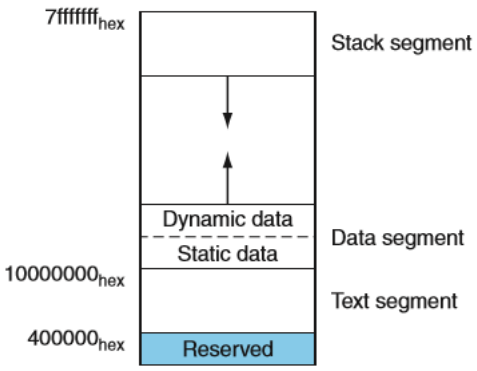
March 23, 2020

# Contents

# 1 Understanding

Project 2 asks us to design a MIPS simulator, which takes the machine code file generated from project 1 and simulates the execution. Data section .data will be added. We also need to simulate a 6MB memory to store the code, static data and dynamic data, which is the most significant part in this project. The program can be used to execute many other MIPS programs to realize different functions.

1) **Simulate memory:**




We need to allocate 6MB to simulate the memory. From the bottom: 1MB for text segment: store the 32-bit instruction code (transform to a signed integer here) ; x MB for pre-defined static data (in .data) ; y MB for dynamic data, which is the heap in our simulation program (used in sbrk – malloc in C), this part is just on the static data part (continuous); z MB for stack segment, which stores the local variables during execution (we don't need to care about, just let the MIPS programmer to handle it), the address of this part goes down. 1+x+y+z = 6

Besides, we need to map the 64-bit real base address to 32-bit 0x00400000 in order to test and manage the code more conveniently (actually the map creates lots of trouble…).

**2) Registers:**

Simulate registers. Since each register stores 32-bit, which just equals to the length of (int), so an integer array of size 34 (with hi &lo) is sufficient to represent them. There are mainly two kinds of data in registers: int and address (pointer *, represented as integer).

**3) PC:**

Program counter, which points to the current instruction and moves to the next instruction after execution (can be changed by instructions like beq). PC is just like a pointer of the machine code (int *).

**4) Code/Data/Stack section:**

As mentioned above, the 1MB code section is used to store the 32-bit machine code (saved as an integer). Just use a int * to store it. The data section is to store the .data before .text code. There are 5 types of data required:

.ascii – no terminated string .asciiz – null terminated string

.word –int .half - short .byte - char

The data must occupy a whole word (4 bytes, 1 line). For example, if a .asciiz is 14 bytes with the \0, it should occupy 16 bytes. Data section is located on top of the code section. Stack section is on the top of the 6MB memory and grows down.

**5) Simulate instruction & execution:**

For this part, extract the 32-bit code from text section first, decode it, and execute the corresponding functions. (Here I use switch to select execution).

70+13 functions in total. (memory operation load/store and syscall about file are more difficult)

**6) Syscall:**

Basic function for lower level operation and I/O (UI: print/read). There are corresponding functions in C (e.g. sbrk – malloc; open; read; read_.. - scanf).

# 2 Logic and Details

Process: read the file – generate two files (text and data) – simulate memory - store the code text and data – extract the instruction and decode – execution – end of the program.

1) **Translation**: This is what project 1 does. Enter the input file name and output file name, scan the MIPS codes and generate a machine code file. The .data part will also be dealt with here. (One more function: store the 5 types of data in the static data part, again the tricky string operation…)

```
void DataScan(int * &static_data_pointer,string& Ifilex_data)
```

2) **Memory Simulation**: Project 2 begins. At first I define a global integer array to simulate the 34 32-bit registers: `int reg[34]={0};`

Then allocate a 6MB memory using malloc, get a pointer to the base address.

Then I set a long long type to map the real address to 0x00400000, which is frequently used in the following instructions:

```
long long real_to_fake_memory_diff = (long long) 0x00400000 - (long long)
(memory_base_address);
```

Then define a int * PC, from base of the memory. Point to the first instruction. Other pointer pointing to different parts of the memory are also allocated. (static_data_pointer, text_pointer, $sp,$fp,$gp). Store the code into text section.

3) **Execution**: While loop (END: PC points to 0, no instruction left). Extract the integer from text section, change it to 32-bit, split it into different parts (opcode, rs, rt, rd, shamt, funct, imm, target……) Switch the 6-bit opcode first, if opcode is 0, then it must be R format instruction / syscall, then switch the5-bit funct and go to different R functions/ syscall (another switch in syscall) – just like instruction hierarchy. Else if opcode is not 0, then it will go to I / J format. Each time the PC will +1 first, but there may be instructions which adjust PC. Free the memory in the end. (I would say it is really hard to understand, simulate and debug each instruction. There are always tiny bugs.)

Actually it is procedure-oriented instead of object-oriented. Just follow the step and work out the details.

Key words: Step by step. Divide and conquer.

Header File:

#include "memoryandexecution.h" : simulate the memory, read the code in text segment, decode the instructions and execute (main part of project 2)

#include "scanAndOutput.h" : Scan the original MIPS code and generate a machine code .txt file and a data file .txt (main part of project 1)

#include "stringOperation.h": Some function to deal with strings (split and delete blank) and change the binary form to decimal form (vice versa)

#include "RegisterAndInstructions.h" : some information of registers. (mainly for project 1)

There are also detailed comments in the program.

# 3 How to run the program

```
whale@ubuntu:~$ g++ -std=c++11 main.cpp memoryandexecution.cpp RegisterAndInstru
ctions.cpp scanAndOutput.cpp stringOperation.cpp -o simulator
whale@ubuntu:~$ ./simulator
Enter the MIPS input file name: many_tests.asm
Enter the machine code output file name: output.txt
```

Compile the code:

g++ -std=c++11 main.cpp memoryandexecution.cpp RegisterAndInstructions.cpp scanAndOutput.cpp stringOperation.cpp -o simulator

./simulator

Follow the prompt to input the file names (I/O).

The input file includes the original MIPS codes. Name of the output file can be set arbitrarily (as .txt).

By the way, the program will generate an immediate file called “whale.txt” and a data file called “whale_data.txt”, which store the instructions and data being filtered. The file can be ignored. But if you have interest, you can have a check ☺.

Result of the above simulation (many_test.asm):

```
Testing lb,sb,read/print_char,sbrk
Please enter a char:
Enter a character: t
The char you entered is:
t
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 cool
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
Current address: 0x40021c
Program exits with status 0, goodbye!
```

More results (mini_grader.asm):

```
Enter the MIPS input file name: mini_grader.asm
Enter the machine code output file name: o.txt
Welcome to this grading mini program!
Please enter the student number (1-10):
Enter an integer: 5
Please enter the grade (0-100):
Enter an integer: 90
The grade is now:
90
of student:
5
at address:
5243020
Good bye!
Current address: 0x40009c
Program exits with status 0, goodbye!
```

Fibonacci.asm:

```
whale@ubuntu:~$ ./simulator
Enter the MIPS input file name: fibonacci.asm
Enter the machine code output file name: ou.txt
Enter an integer: 25
75025
```

3.asm:

```
whale@ubuntu:~$ ./simulator
Enter the MIPS input file name: 3.asm
Enter the machine code output file name: p.txt
Enter a string of length 3:hahahahaha
hah
Current address: 0x400044
Program exits with status 0, goodbye!
```

# 4 Difficulty

Debugging:

1) Fall through of switch: nested switch sentences. Forget to add break.

2) Typo mistake. rs → rt  add "+" → "-"

3) Load byte: register should use int32_t to receive instead of int8_t
4) Transformation between real address and fake address
5) Int * pointer + 1 = address + 4
6) Order of *(int *) xxx + c …

……

Memory Calculation:

1MB = 1024KB = 1024 * 1024 bytes = 1024 * 1024 * 8 bits

File operation:

It really annoyed me a lot… I cannot find the functions used in that example on bb even though I included many strange header files. So I finally used the "fstream" in C++ to simulate the file syscall (the easiest way is to use the lower level function: open() read() write() to simulate). The open mode in fstream (ios::in | ios::out | ios:: trunc) really confused me. It took me a long time to realize that the syscall open function (13) should include two situations: create a file which does not exist; open a file which already exists and R/W. And I use a conditional sentence to deal with it.

The number of instructions:

70 instructions + 13 syscall functions: we need to think of the logic and write every function, much more time-consuming and difficult than project 1. It feels like writing 80+ little programs. Actually I think the simulation of instructions is the most difficult part of this project instead of the simulation of the memory (just about many pointers and the transformation of the real and fake address). TA Handsome just said "No longer than 5 lines for each instruction… It is the easy

part…” Really?

# 5 Feeling

In a way, I learn and practice a lot and make great progress in this project.

This is the largest coding project I have ever experienced. I just felt that I was struggling for the final week. I spent the whole week understanding everything, discussing, asking for help, coding, and testing…… I cannot sleep well.

It really costs me a great amount of time to write 1500+ lines of code.

However, I understand more deeply about the memory and machine cycle in this project. I was more and more familiar with the use of pointers (address). I started to feel that I was exactly a student from CS, that’s good.

TA Handsome gave us lots of hints and assistance during the job. I remembered that on the last tutorial about project 2, I asked many questions and TA Handsome answered them one by one. I focused on every minute of the tutorial and understand many things about the project. This may be the most efficient tutorial I have ever had! Besides, he even marked some information of the code in the test file (e.g. address), which is really helpful when testing the code.

I also learned a lot from wechat group (sometimes very interesting). There are indeed many “dalaos”.

This design of this project is really elaborate. We just made use of what we learned in the lecture and applied the theory in practice. Really good!

That’s all.