# CSC3050 Assignment 2 Report (120090211)

## Overview

This C++ program aims to simulate the computer execution. There are 5 inputs for the programs, including a MIPS instruction file, a machine code file, a checkpoint file, an input file, and an output file. The program will first convert all MIPS instructions to machine codes. Then it will compare the machine codes with the correct machine code file. After that, the program will store all the .text section and .data section into the allocated area. (text starts from 0x400000, data starts from 0x500000) The program will read .text word by word and does the required instruction.

## Processing Logic:

To solve the problems, I divide the whole program into 7 parts.

1) Translate the .text part into machine codes (Assignment 1)
2) Store text

In this part, I only focus on the .text section, discarding other parts. Read machine codes stored in .txt file (argv[2]) line by line. Convert it into a 32-bit integer, and be stored in the assigned location with the int* real_mem. (like real_mem[i++]=num)

3) Store data

In this part, I only focus on the .data section, discarding other parts. Read the .asm file line by line. Firstly, I discard all the redundant spaces at the beginning of the string or the end of the string and discard the comments at the same time. Next, split the whole line into two parts. Use the first part to specify the type of the data. Using an int index to record the next place to store.

If the type is ascii or asciiz:
I put the second part into a queue<char>. Then, pop the character one by one and store them into the static section of the memory. If a '\' appears, the character to be stored will be defined by the latter element in the queue.
For example, if the next element is n, then I only store one '\n' in the memory. Particularly, if the type is asciiz, I will add an '\0' at the end of the string.
For ascii and asciiz, I use big-endian to store the data.

If the type is word/half/byte:
I use ',' to split the second part into several strings. For each split string, I convert it into 32/16/8 bits and store them into the memory using an int pointer real_mem.
For example, to store a half 4 into the memory is like, real_mem[index+(0x500000-0x400000)/2]= 4 (set the size to 16 bits)
All data pieces are 4B, so this way of storage is safe even the sizes of half, byte, word are different.

For word/half/byte, I use little-endian to store the data.
The index will be returned to calculate the address of the pointer for the dynamic section.

4) Set checkpoint set

First, open the checkpoints file. Next, read the file line by line and store them into a set<int>. The number stored in the set determines when the program will dump the content of memory and registers.

Further specifications will be explained in the implementation details.

5) Initialization

Firstly, set 6MB for memory, 32*4B for registers file, 4B for pc, lo, hi.
Secondly, reset all the bytes in 5 parts above to 0.
Thirdly, assign pointers using the name of each register (zero, at…ra) to the corresponding space in the registers file. (For example, ki=&real_reg[27])
Lastly, assign the data in the space pointed by the gp pointer to 0x508000, assign the data in the space pointed by the fp and sp pointer to 0xa00000, assign the data in the space pointed by the pc pointer to 0x400000.

More specifications about memory allocation and the use of pointers will be explained in the implementation details.

6) Working cycle

The working cycle is a while(true) loop. For each cycle, an int pointer ptr will record the current data in the space pointed by pc. (i.e. pc[0]). Next, subtract 0x400000 from pc[0] and divide the result by 4. This number indicates the line number of the current instruction. If the number appears in the checkpoint set<int>, then the content of the memory and registers file will be dumped after the execution of the current instruction. Then, add 4 to pc[0], indicating the address containing the next instruction if none of jalr, jr, jal, j is called. After that, execute the current instruction and dump the memory and registers file if needed.

7) Execution

In this part, I first split the 32-bit machine code into several parts using bitset. A function named "cut" can help to get a fraction of the whole machine code starting from any index and ending at any index you want. In this way, the number representing opcode, rs, rt, rd, sa, immediate, address, and so on are obtained. (For example, "cut" line[31:26] to bitset<6> opcode.)
First turn the bitset opcode to unsigned long type, in the hope of specifying which instruction will be executed.

If the number is 0, this is a R type instruction. Go on specifying the exact instruction using bitset<6> function (i.e. line[5:0]). Particularly, if the number converted from bitset function is 12, it is a syscall instruction, the integer stored in v0 register will tell

what instruction to be executed.
If the number is 2, this is the "j" instruction.
If the number is 3, this is the "jal" instruction.
If the number is some integers other than 0,2,3, then it is a I type instruction. We can know the type of instruction directly from the number.

After knowing what instruction we are going to execute, other bitsets are used to get numbers indicating the needed registers, immediate numbers, or the virtual address. The numbers will be matched to the exact pointers using a map. The matched pointers are used to get the numbers stored in the registers file or the memory to do some calculation, loading, storing, etc.
To manipulate byte and half, char pointer and short pointer are used.
For syscall instructions, some of them (open, read, write, close, etc.) are directly implemented with Linux API.

## Implementation details:

Data memory:
Use the melloc function and memset function to assign the memory.
An int pointer (real_mem), a char pointer (chp) and a short pointer (shp) point at the beginning of the memory section in order to get/store data from/into the memory.
The conversion from virtual address to the real address is subtracting the virtual address by 0x400000 and divide by 2 (half) or 4 (byte).

Register file:
Use the melloc function and memset function to assign the register file. Every 4 bytes is seen as a specific register pointed by an int pointer. A map<int, int*> is used to match an integer number to that int pointer. For example, 31 for ra register. This will be used in the execution part of the working cycle.

Dump memory:
If the line number of current instruction appears in the checkpoint set. After the execution, the fwrite function will be called to dump the content into a .bin file.