CSC3050 Assignment2 Report

Name: Xiang Fei

Student ID: 120090414

1. Design

1a. Overview

This project is to build a MIPS simulator, which simulates the execution according to the given MIPS assembly language code and the corresponding machine code. With the MIPS assembly language code, we can get the static data information. With the machine code, we can know which instructions we need to use. In the first part of the program, I construct a 6MB virtual memory. Then, I use nearly the same method as Project_1 to scan the MIPS file. In phase1, I regard the variable name in data segment as a label, and store the name and its address in the labelArray structure I used in Project_1. And then, In phase to, I scan the MIPS file again and store the value of the static variable. And I also read the machine code file and store the codes in the test segment of my virtual memory in phase2. Then is the simulator part. In this part, I execute the machine code using a machine cycle. First, I load the line of instruction PC is pointing at. Then, I increment the PC by 4 and run the loaded instruction. I repeat the above cycle until the program terminates. Last but not least, in this program, I also create memory and register file according to the checkpoints file to debugging and show the memory and register value of several different time.

1b. Important Data Model

- How to store the information of the static variables? I build a structure called LabelOj and create a LabelOj array.

```
struct LabelOj {char* name; int address;};
static LabelOj labelArray[1000];
```

The pointer **name** stores the name of the variable and the **address** stores the virtual address of the variable. And **labelArray** is an array consists of instances of LabelOj structure.

- How to map the file discriptor and the corresponding file? I use map structure.

```
static map<int,FILE*> file_list;
static map<char*,int> file_name;
```

file_list is a map from the file discriptor number to the corresponding file, and **file_name** is a map from the file name to the file discriptor number.

- I also create some important variable.

```
static char* MEMORY_START;
static const int TEXT_START = 0x400000;
static const int STATIC_START = 0x500000;
static const int STACK_START = 0xA00000;
static int *HI, *LO;
static char* PROGRAMCOUNTER;
static int **REGS;
static int* dynamic;
static int* DYNAMIC_START;
```

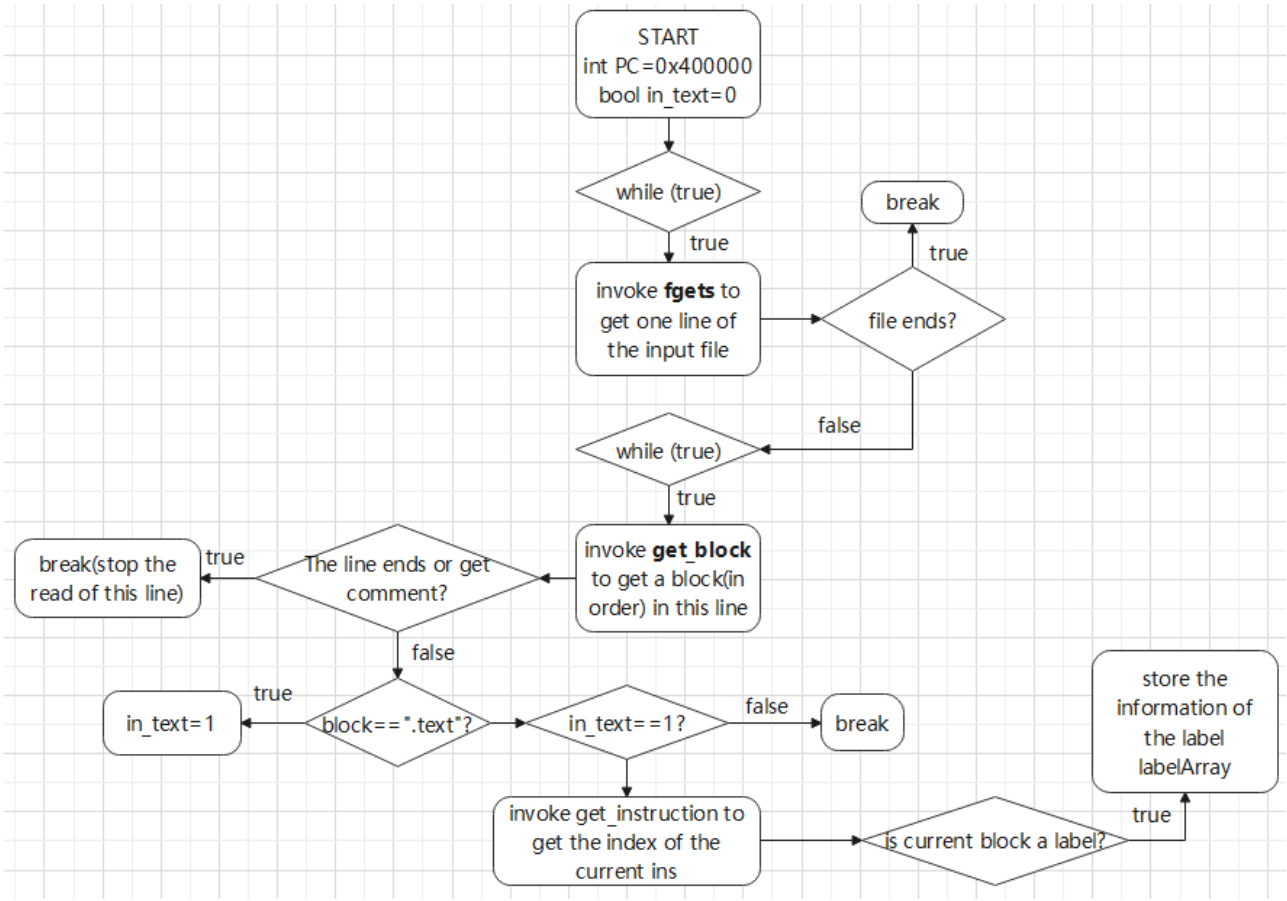
MEMORY_START is a pointer stores the start address I allocated for the virtual memory and minus 0x400000. Therefore, for any virtual memory address, I can get its real address by adding **MEMORY_START**. **TEXT_START** is the simulated address of the beginning of text segment. **STATIC_START** is the simulated address of the beginning of static data segment. **STACK_START** is the simulated address of the beginning of stack segment, and it grows downwards. **HI**, **LO**, **PROGRAMCOUNTER** are pointer for register HI, LO, PC respectively. **REGS** is a pointer to pointer, which works like an array of the 32 general purpose registers. **dynamic** points to the end of the dynamic data segment. **DYNAMIC_START** points to the end of the dynamic data segment.

2. Program Structure(Processing Logic)

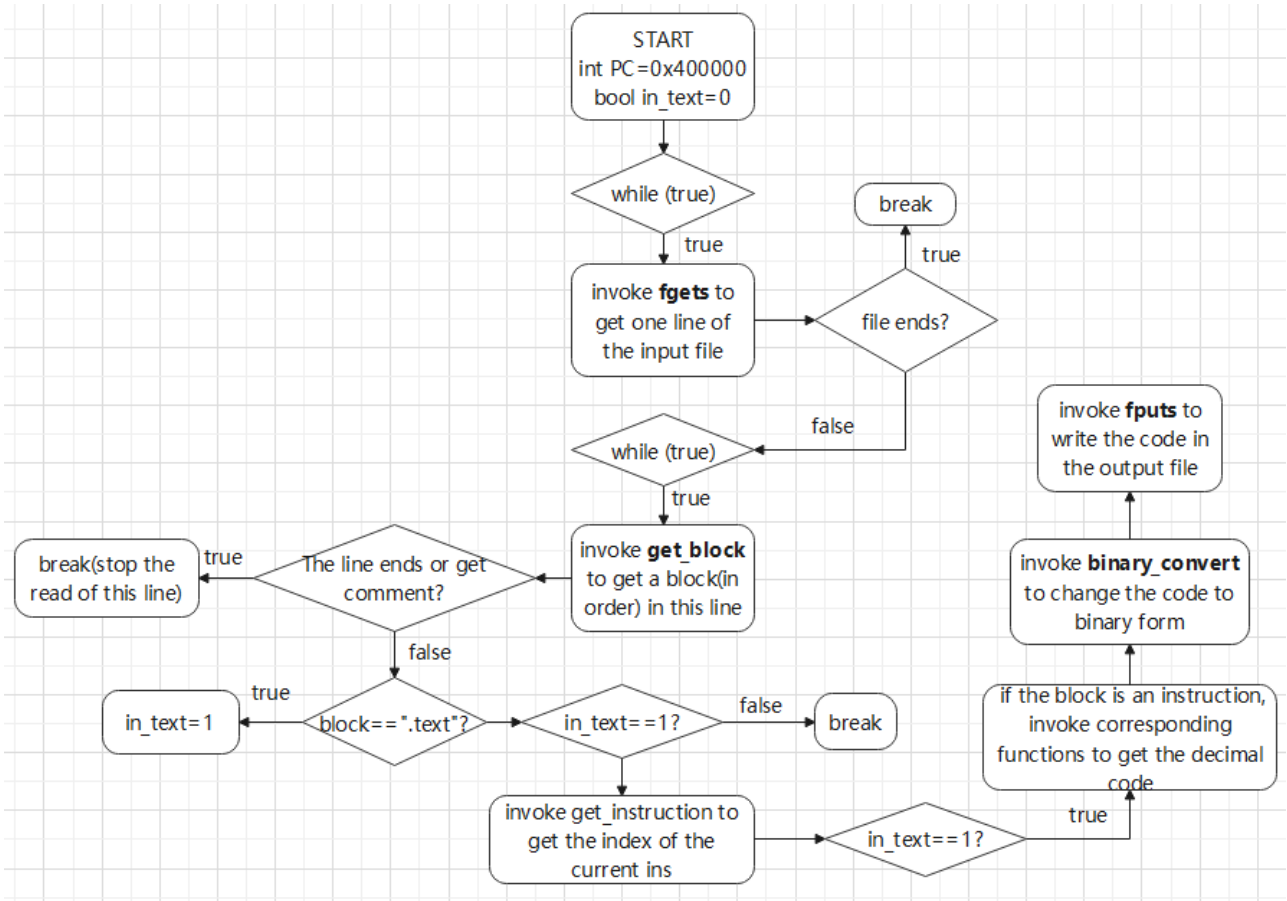
We need to simulate a virtual memory. Before we start, we need to allocate space for 32 registers as 32 pointers. Those pointers are out of the virtual memory we simulated. First, we set the PC to the beginning of the text, which is 0x400000 in virtual memory. And we have a one-to-one relation between real memory and virtual memory like this formula: $real_a = virtual_a + real start_a - 0x400000$ We should pay attention to the endian for storing data. For .ascii and .asciiz, we use Big Endian, for the remaining data type, we use Small Endian.

> The following figures are not used to show the technique details of my code, they are just used to show the structure or logic of the program. Just for easier understanding.

- Phase1



- Phase2



In fact, the above graph is the implementation of assembler, but the data scan logic in this project is nearly the same. For static data, we just regard them as labels and store them in the labelArray structure. One thing we need to do in .data is that after jumping to corresponding address, store the

variable correctly for it type (one of .word, .ascii, .asciiz, .bytes, .half) in memory where is corresponding to the current PC.

Then is the simulator part. In this part, I execute the machine code using a machine cycle. First, I load the line of instruction PC is pointing at. Then, I increment the PC by 4 and run the loaded instruction. I repeat the above cycle until the program terminates. Last but not least, in this program, I also create memory and register file according to the checkpoints file to debugging and show the memory and register value of several different time.

```
phase1(mips_file,instructions);
rewind(mips_file);
phase2(mips_file,code_file,instructions);
simulator(in_file,out_file);
```

Invoke the following part sequently like the above code, we done the implementation.

3. Important functions I construct

```
static char const *instructions[] = {

"add","addu","and","nor","or","sllv","slt","sltu","srav","srlv","sub","subu","xor"
,"jalr","div","divu","mult","multu","mfhi","mflo","mthi","mtlo","jr","sll","sra","
srl","syscall","addi","addiu","andi","slti","sltiu","xori","ori","bgez","bgtz","bl
ez","bltz","bne","beq","lb","lbu","lh","lhu","lw","sb","sh","sw","lwl","lwr","swl"
,"swr","lui","j","jal"};
```

For each instruction in the above array, I write a corresponding function. And I construct the following three functions to execute R, I, J types of instructions respectively.

```
inline void executeRins(unsigned int code);
inline void executeIins(int opcode, int code, int r_t);
inline void executeJins(int opcode, int code);
```