CSC3050 Assignment1 Report

Name: Xiang Fei

Student ID: 120090414

1. Design

1a. Overview

This project is to build a MIPS assembler, which translates the MIPS code(assembly language code) to machine code. In this specific program, we need to consider 55 instructions, of which there are 27 R type instructions, 26 I type instructions and 2 J type instructions. Different types of instruction have different translating rules. Therefore, I create different structures for each type of instructions to store the corresponding information. And I divide the big problem into two parts. The first part is phase1, which read the mips files line by line(just care about the .text segment and ignore all of the comments) and store the label name and its address (here is the absolute address) in a **LabelOj** array created by my self. The second part is phase2, which scan the file for the second time line by line and identify all the three type of instructions, then translate them to machine code according to different rules. For lines with labels, we can handle them refer to the information we stored in phase1. And finally, we need to create a output machine code file.

1b. Important Data Model

- How to store the information of the labels? 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 label and the **address** stores the absolute address of the label. And **labelArray** is an array consists of instances of LabelOj structure.

- store the name of all the 55 instructions we need to consider with an array of pointers to char.

```
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"};
```

- The structures of three types of instructions and the corresponding arrays to store the information of each instruction.

```
struct R_type_ins
{
    int opcode,funct;
};
struct I_type_ins{
    int opcode,rt;
};
struct J_type_ins{
    int opcode;
};
```

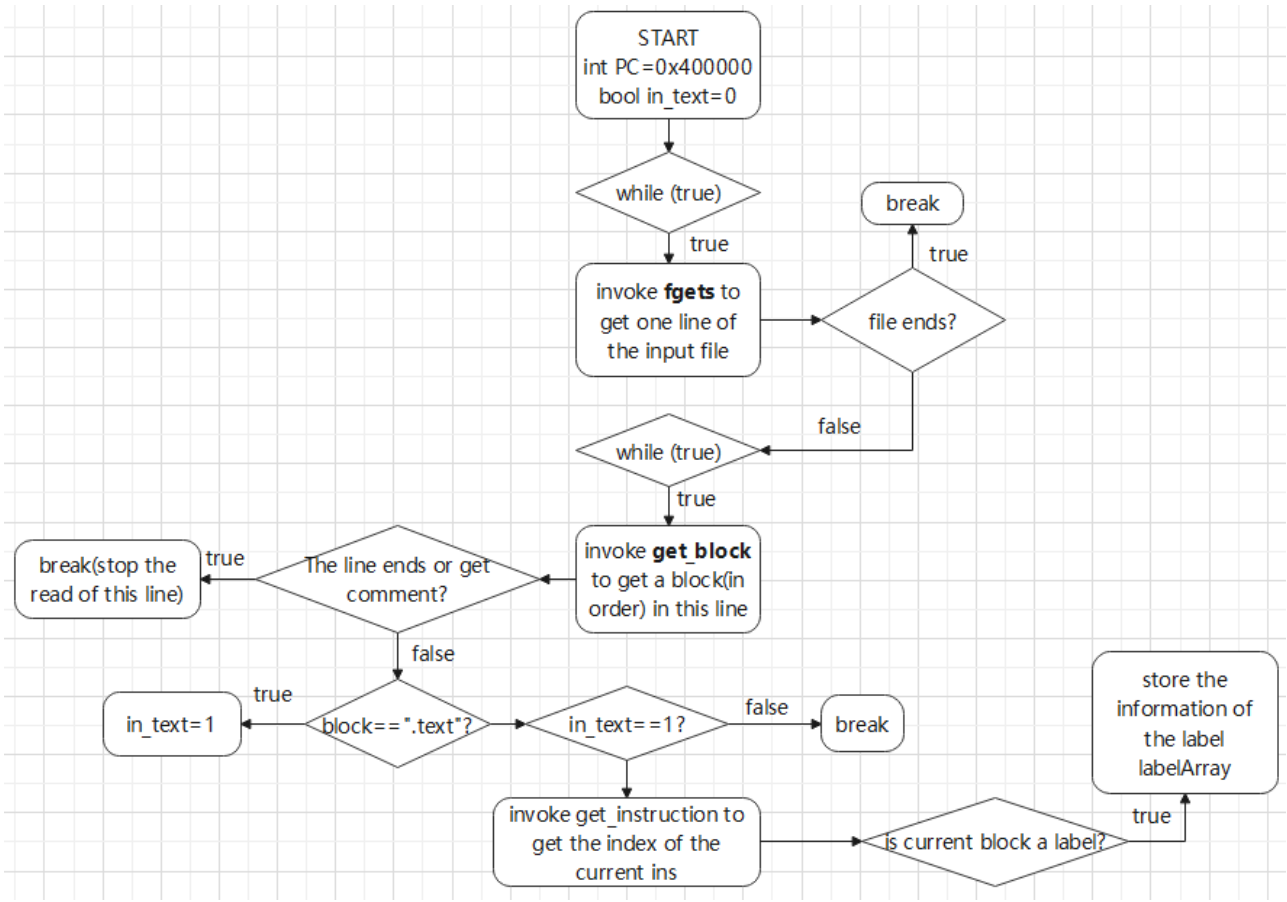
The meaning of each fields is obvious. For arrays, I just use J type as an example:

```
static J_type_ins J_ins[2] = {
    {0x02}, //j label
    {0x03}, //jal label
};
```

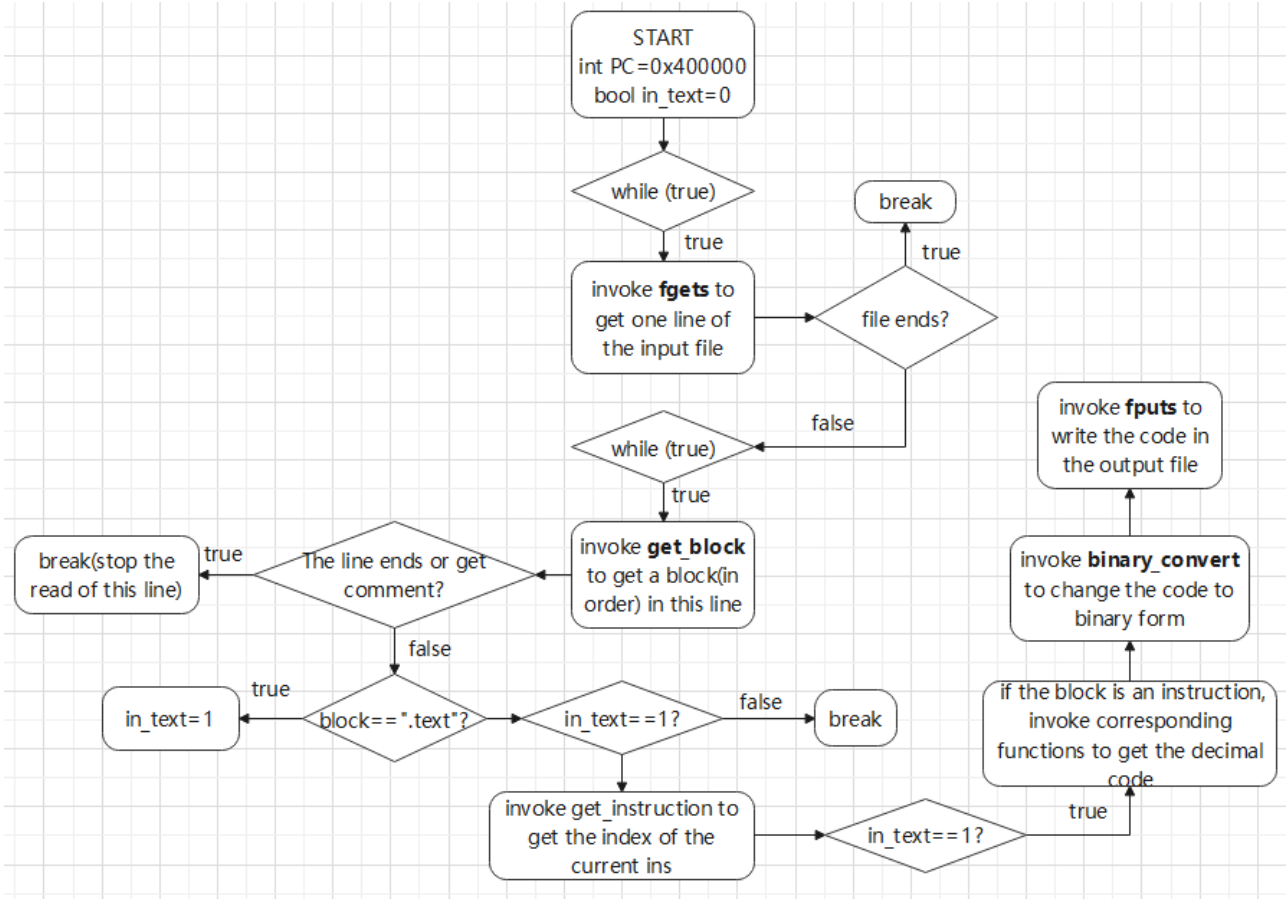
1c. Program Structure(Processing Logic)

> 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 two phases. Just for easier understanding.

- Phase1



- Phase2



2. Implementation of Small Parts

2a. Divide the Big Problem into Several Small Problem

In order to handle the big project, I divide the big problem into several small problem. First, I construct some important functions. Then, I use these functions and the processing logic to finish the two phases, which are used to store label information and generate the machine code respectively.

2b. Important functions I construct

```
inline int get_regNum(char *reg_name);
inline int get_label(char *label_name);
inline void append_label(char *block,int PC);
inline char *get_block(char *start_ptr,char **end_ptr,char const *invalid);
inline int get_instruction(char *block,char const **instructions);
void phase1(FILE *in_file,char const **instructions);
inline int R_ins_code(int reg_num, char* start_ptr, char** end_str);
inline int I_ins_code(int reg_num,char* start_ptr,char** end_ptr,int PC);
inline int J_ins_code(int reg_num, char* start_ptr, char** end_ptr);
inline string binary_convert(int x);
void phase2(FILE* in_file, FILE* out_file, char const **instructions);
```

Above are functions I created. Phase1 and Phase2 are constructed by these functions and the processing logic. I just choose some very important functions to show and explain the code (I have bolded these

functions).

- get_regNum This function is used to get the register number according to the name of the register.
- get_label This function is used to get the index of the specific label stored in the label array by compare the label's name. If the name is not in the label array, then return 0.
- append_label This function is used to insert the label we find currently into the label array.
- **get_block** This function is used to get the block in one line consequently.

```
// get the effective block of the MIPS code
inline char *get_block(char *start_ptr,char **end_ptr,char const *invalid) {
    int length;
    char *block,*p1,*p2;

    length = strspn(start_ptr,invalid); // find the first valid one after the
invalid character
    p1 = start_ptr+length;
    p2 = strpbrk(p1,invalid); // get the first invalid character after the valid
ones
    if (p2 == NULL){
        return(NULL);
    }
    length = p2-p1;
    *end_ptr = p2+1;
    block = (char*)malloc(length+1);
    if ((length==0)||(block==NULL)) {
        return NULL;
    }
    memcpy(block,p1,length);
    block[length] = '\0';
    return block;
}
```

The strspn function is used to find the first valid character after the invalid ones. The strpbrk function is used to get the first invalid character after the valid ones. By using these two function, we can easily get a block(you can also understand it as token) in the current line. If the line ends, return NULL.

- get_instruction This function is used to check whether a block is an instruction (like lw, add), if so, get the index of the instruction in the list of all instructions (a array I said before).
- binary_convert This function is used to convert the decimal code to binary string form.
- **R_ins_code** This function is used to get the code of R instructions.

```
// get the R instruction code
inline int R_ins_code(int reg_num,char* start_ptr,char** end_str) {
    char *reg;
    int op,rs,rt,rd,shamt,funct;
    unsigned int code;
    op = R_ins[reg_num].opcode;
    funct = R_ins[reg_num].funct;
    if (reg_num <= 12) {  // ins rd,rs,rt
```

```
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rd = get_regNum(reg);
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rs = get_regNum(reg);
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rt = get_regNum(reg);
            *end_str = start_ptr;
            code = funct+(rd << 11)+(rt << 16)+(rs << 21)+(op << 26);
        }else if (reg_num == 13) {  // ins rd, rs
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rd = get_regNum(reg);
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rs = get_regNum(reg);
            *end_str = start_ptr;
            code = funct+(rd << 11)+(rs << 21)+(op << 26);
        }else if ((reg_num >= 14)&&(reg_num <=17)) { // ins rs,rt
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rs = get_regNum(reg);
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rt = get_regNum(reg);
            *end_str = start_ptr;
            code = funct+(rt << 16)+(rs << 21)+(op << 26);
        }else if ((reg_num == 18)||(reg_num == 19)) {  // ins rd
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rd = get_regNum(reg);
            *end_str = start_ptr;
            code = funct+(rd << 11)+(op << 26);
        }else if ((reg_num >= 20)&&(reg_num<=22)){   // ins rs
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rs = get_regNum(reg);
            *end_str = start_ptr;
            code = funct+(rs << 21)+(op << 26);
        }else if ((reg_num >= 23)&&(reg_num <=25)) {  // ins rd,rt,sa
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rd = get_regNum(reg);
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            rt = get_regNum(reg);
            reg = get_block(start_ptr,&start_ptr,",\n\t ");
            sscanf(reg,"%d",&shamt);
            *end_str = start_ptr;
            code = funct+(shamt << 6)+(rd << 11)+(rt << 16)+(op << 26);
        }else{  // syscall
            code = funct+(op << 26);
        }
        return code;
    }
```

In fact, I divide the R instructions into several groups according to the structure of the mips code. You can see it clearly in comment of the above code. I get the value of the section like op,rs,rt,rd,funct respectively by consequently get the blocks in the line. Then, I shift the bit and add them together to get the int code. To get the code of I instructions and J instructions follow the same logic and method. So I don't show them in the report. You can see the details by seeing my code.

2c. Part1(Phase1):

```
void phase1(FILE *in_file,char const **instructions){
    int PC = 0x400000;
    char currentLine[257];
    int tempPC = PC;
    int length;
    bool in_text = 0;
    char *block,*spt,*thisLine;

    while (true){
        for (int i = 0;i < 257;i++){
            currentLine[i] = '\0';
        }
        thisLine = fgets(currentLine,256,in_file);
        if (thisLine == NULL){
            break;
        }
        length = strlen(currentLine);
        currentLine[length] = '\n';
        spt = currentLine;
        while (true){
            block = get_block(spt,&spt,",\n\t ");
            if ((block==NULL)||(*block=='#')) {
                free(block);
                break;
            }

            if(!strcmp(block,".text")){
                in_text = 1;
            }

            if(!in_text){
                break;
            }

            int ins_i = get_instruction(block, instructions);
            if (ins_i != -1){
                PC+=4;
            }else if (!strcmp(block,".text")){
                PC = tempPC;
            }

            if (strstr(block,":")){
                int len_block = strlen(block);
                block[len_block-1] = '\0';
                append_label(block, PC);
            }
        }
    }
}
```

The function is used to implement the phase1. I use two loop, The first is to read the input files line by line, the second one is to read the current line block by block (blocks are tokens, such as add, $t1$,t2, these are all blocks). And we stop the second loop until the line ends or we see comment or we are not under .text segment, stop the first loop until the file ends. We begin to analyze blocks since the block .text has appeared (change the boolean variable in_text to true). Then, if the block is a label(has :), then store the name and the address of the label to the label array.

2d. Part2(Phase2):

```
void phase2(FILE* in_file, FILE* out_file, char const **instructions) {
    int PC = 0x400000;
    char currentLine[257];
    int tempPC = PC;
    int length;
    bool in_text = 0;
    char *block, *spt, *thisLine;
    string str_code;

    while (true) {
        for (int i = 0; i < 257; i++) currentLine[i] = '\0';
        thisLine = fgets(currentLine, 256, in_file);
        if (thisLine == NULL){
            break;
        }
        length = strlen(currentLine);
        currentLine[length] = '\n';
        spt = currentLine;
        while (true) {
            block = get_block(spt,&spt,",\n\t ");
            if ((block == NULL)||(*block == '#')) {
                free(block);
                break;
            }

            if(!strcmp(block,".text")){
                in_text = 1;
            }

            if(!in_text){
                break;
            }

            int ins_i = get_instruction(block, instructions);
            if (ins_i != -1){
                PC += 4;
            }else if (!strcmp(block,".text")) {
                PC = tempPC;
            }

            if (in_text) {
                unsigned int code;
                if (ins_i != -1) {
```

```
                    if (ins_i <= 26){
                        code = R_ins_code(ins_i,spt,&spt);
                    }else if ((ins_i >= 27)&&(ins_i <= 52)){
                        code = I_ins_code(ins_i, spt, &spt, PC);
                    }else if ((ins_i >= 53)&&(ins_i <= 54)){
                        code = J_ins_code(ins_i, spt, &spt);
                    }
                    str_code = binary_convert(code);
                    fputs(str_code.c_str(),out_file);
                    fputs("\n",out_file);
                }
            }
        }
    }
    fclose(out_file);
}
```

The function is used to implement the phase2. The code in the previous lines is very similar to phase1. We just need to talk about the code after if(in_text) condition (which means the lines are under .text). If the block is a instruction, then use the index of it in the instruction array to determine the type of the instruction. And then, we get the code by the functions R_ins_code(or I_ins_code, or J_ins_code) I said in the previous part. Finally, I convert the int code to binary string form and use fputs to write it in the output file. I repeat line by line, and get the complete machine code.