CSC3050 Project 1

March 6, 2022

Build a MIPS assembler

Student ID: 120090266

Student Name: Feng Yutong

This assignment represents my own work in accordance with University regulations.

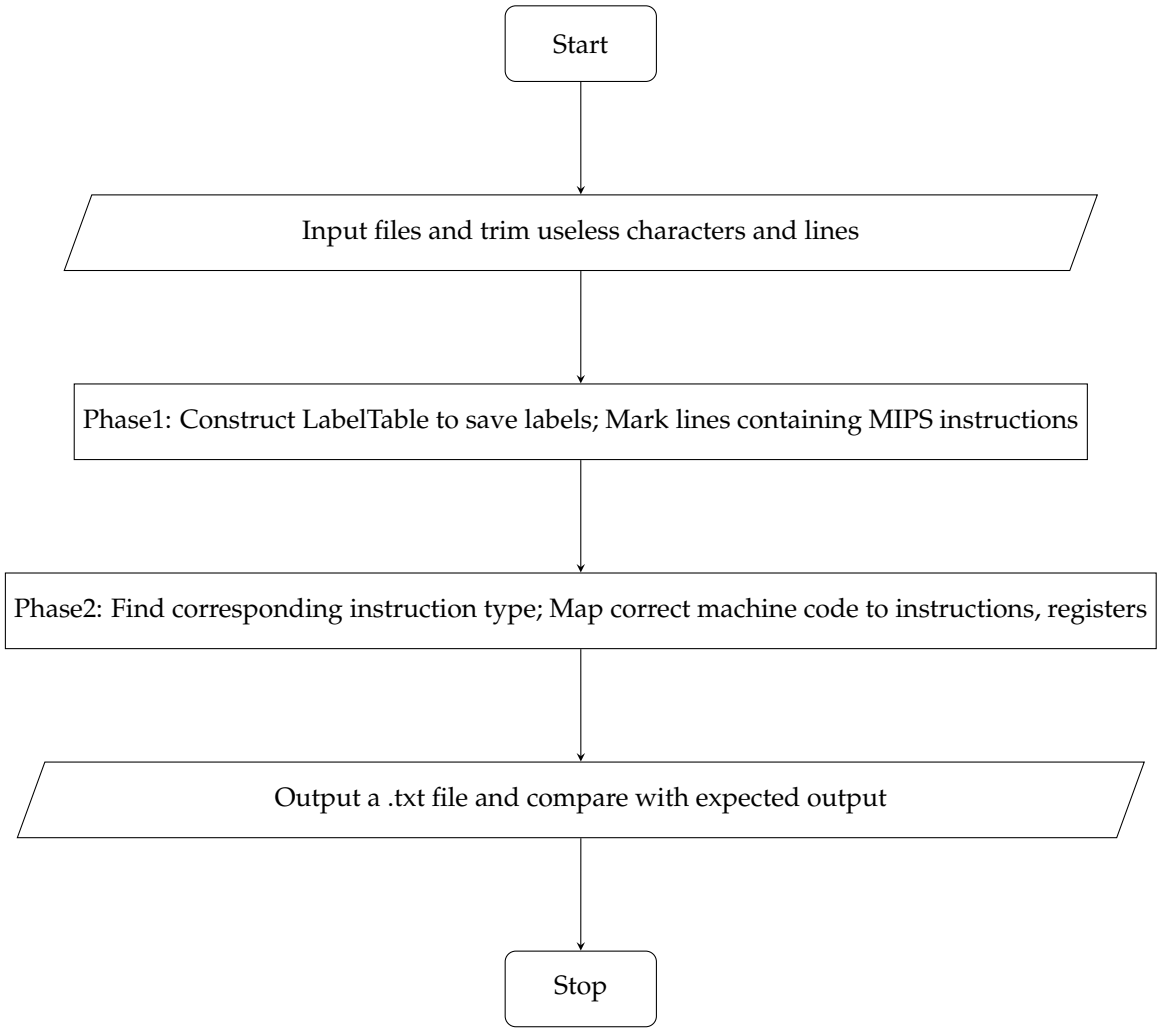
Signature: Feng Yutong

1 Overview

The task of this project is to implement a MIPS assembler. It translates MIPS code to its corresponding machine code. The MIPS assembly language has three types of instruction in general: R-type, I-type and J-type. Each instruction and register has their own 01 sequence. Therefore, the main idea is to implement a function that can realize this one-to-one mapping relationship. There are several challenges of this task:

1. Find a proper data struture to realize the mapping relation.
2. Identify and classify the type of MIPS.
3. Ignore the comments after the line of code or in a whole line.
4. Store the address of the label and compute absolute and relative address.
5. Deal with empty spaces, tabs and empty lines.

The big picture thoughts and ideas are as followed:



2 High Level Implementation Ideas

2.1 Phase 1

In Phase1, the code reads input file and does preprocessing for Phase2.

Construct LabelTable to save labels

The LableTable is implemented in LabelTable.cpp. It uses map of STL to realize the one-to-one mapping relation between instructions, registers, labels and their corresponding machine code or address. Since all types of MIPS code have fixed format and length, every returned result from LabelTable is in the format of string type. So it also has function that converts decimal number to binary number, digit to string.

Mark lines containing MIPS instructions

When reading input file line by line, the code marks lines containing MIPS instruction. Those unmarked lines e.g., empty lines, comments and labels, will not be processed during Phase2. At the same time, the code finds labels and stores them.

2.2 Phase 2

Phase2 is the main part of processing.

Find corresponding instruction type

The code first finds the instruction and its corresponding opcode. According to the opcode (See Table 1), the code calls function get_R, get_I, get_J to do more specific process.

Table 1: Classification of instruction type

type	opcode
R-Type	000000
I-Type	All opcodes except 000000, 00001x, and 0100xx
J-Type	00001x

Map correct machine code to instructions, registers

The main idea is to 'merge like terms'. Instructions are classified again by the number and tpye of registers. Fields including rs, rt, rd, sa, function, immediate and targets are all initialized with 0. If one instruction has certain field, the field will be replaced by machine code stored in map or calculated result. Then connect each fields into one line and output.

3 Implementation Details

3.1 Phase 1

LabelTable.cpp:

Function ' get ' (get machine code of instructions and registers):

Use ' map<string, string> op ' to store the the opcode.

Use ' map<string, string> m ' to store the machine code of registers, label address, function code of instructions in R type.

Function ' get_addr ' (calculate the address of label):

Use ' map<string, int> ad ' to store the line number of label. Thus, absolute and relative address can be found through easy calculation.

Use <bitset> to convert decimal number to binary number. One of its benefits is that it can fix the number of digit.

```
string LabelTable::dec2bin(int x, int l){
        if(l == 16){ // for immediate
                bitset<16> bs2(x);
                return bs2.to_string();
        }
        if(l == 5){ // for shamft
                bitset<5> bs2(x);
                return bs2.to_string();
        }
        if(l == 26){ // for target
                bitset<26> bs2(x);
                return bs2.to_string();
        }
}
```

Phase1 function:

Use ' trim ' function to trim spaces in the head and tail of the line.

```
void Trim(string & str){
        string blanks("\f\v\r\t\n ");
        str.erase(0,str.find_first_not_of(blanks));
        str.erase(str.find_last_not_of(blanks) + 1);
}
```

Use ' find(":") ' and ' find("#") ' to locate labels and comments.

Use ' substr ' and ' add ' function to store labels in LabelTables.

Use ' bool is_mip ' array to record whether certain line needs processing in Phase2.

3.2 Phase 2

Phase2 function:

get_R, get_I: Classify instructions by the number and tpye of registers following it and deal with them perspectively (See Table 2). Use two pointers 'st' and 'ed' to substr the line and look up them in map. Then place them in their fields (default fields are filled with 0s). Constant immediate are converted to 01 string by function ' dec2bin '.

Table 2: Example of classification of I type

instructions	number of like terms	fileds (order matters)	others
bgez, bgtz, blez, bltz	2	rs, immediate	rt of bgez is 00001
lui	2	rt, immediate	
beq, bne	3	rs, rt, immediate	
ori, xori, addi, addiu, andi, slti, sltiu	3	rt, rs, immediate	
rest instructions	3	rt, ts, immediate	

get_I, get_J: Call ' get_addr(string label, bool is_relative, int current_line_number) ' to get 01 string of address. (relative = 0 for get_J, relative = 1 for get_I)

```
string LabelTable::get_addr(string key, bool rel, int now){
        // key: label, rel:is_relative, now: current_line_number
        if(rel == 1){
                int t = ad[key] - now - 1;
                return dec2bin(t, 16);
        }
        else{
                int t = (0x400000 + ad[key]* 4) >>2;
                return dec2bin(t, 26);
        }
}
```

4 Result

4.1 Complie and run

To run the program, move to the folder of tester.cpp and enter following instructions in terminal:

```
$ make
$ ./tester input_file output_file expected_output
```

The success information is "ALL PASSED! CONGRATS :) ". The error information is " YOU DID SOMETHING WRONG :("

4.2 Example

Here provides a sample input and its running result. This example covers most special cases.

Input file:

```
.data
# not used data
HELLO: .ascii   "hello, world\n" ##      hello #
LENGTH: .word     13
.text


__builtin_memcpy_aligned_large:
addi    $t7, $a2, -4  # junk
# junk #
blez  $t7,                __builtin_memcpy_bytes
lw      $t0,     0($a1)
sw      $t0,     0    ($a0)
addi       $a2,     $a2, -4    ## lalala ###
addiu     $a1, $a1,      4
addiu $a0, $a0, 4


j       __builtin_memcpy_aligned_large
__builtin_memcpy_bytes:beq    $a2, $zero, __builtin_memcpy_return
lbu    $t0, 0($a1)
sb      $t0, 0(    $a0     )
addi  $a2, $a2, -1
addiu $a1, $a1, 1


addiu $a0, $a0, 1
j          __builtin_memcpy_bytes
```

```
29 __builtin_memcpy_return:    jr     $ra
30 __builtin_memcpy:
31 addi        $t7,     $a2, -4
32 blez      $t7,     __builtin_memcpy_bytes
33 xor     $t8    , $a0,  $a1
34
35
36 andi      $t8    , $t0,     3
37 subu  $t1,               $zero,           $a0
38 andi  $t1,               $t1, 3
39 __builtin_memcpy_prepare:
40 beq  $t1, $zero, __builtin_memcpy_check
41 lbu        $t0    , 0(     $a1                      )
42 sb     $t0, 0($a0)
43
44
45 addi    $a2, $a2, -1
46 addi  $t1, $t1, -1
47 addiu $a1, $a1, 1
48 addiu    $a0,   $a0, 1
49 j      __builtin_memcpy_prepare
50
51 __builtin_memcpy_check:
52 beq      $t8,   $zero,     __builtin_memcpy_aligned_large
53 __builtin_memcpy_unaligned_large: addi                    $t7, $a2, -4
54 blez  $t7, __builtin_memcpy_bytes
55 lwl               $t0, 0($a1)
```

Output file:

```
1 00100000110011111111111111111100
2 00011001111000000000000000000110
3 10001100101010000000000000000000
4 10101100100010000000000000000000
5 00100000110001101111111111111100
6 00100100101001010000000000000100
```

```
00100100100001000000000000000100
00001000000100000000000000000000
00010000110000000000000000000110
10010000101010000000000000000000
10100000100010000000000000000000
00100000110001101111111111111111
00100100101001010000000000000001
00100100100001000000000000000001
00001000000100000000000000001000
00000011111000000000000000001000
00100000110011111111111111111100
00011001111000001111111111110110
00000000100001011100000000100110
00110001000110000000000000000011
00000000000001000100100000100011
00110001001010010000000000000011
00010001001000000000000000000111
10010000101010000000000000000000
10100000100010000000000000000000
00100000110001101111111111111111
00100001001010011111111111111111
00100100101001010000000000000001
00100100100001000000000000000001
00001000000100000000000000010110
00010011000000001111111111100001
00100000110011111111111111111100
00011001111000001111111111100111
10001000101010000000000000000000
```