CSC3050 Project 1 Report

118010045 Cui Yuncong

March 2019

Contents

1 Background

1.1 MIPS Architecture

MIPS is a load/store architecture (also known as a register-register architecture); except for the load/store instructions used to access memory, all instructions operate on the registers. There are multiple versions of MIPS: including MIPS I, II, III, IV, and V; as well as five releases of MIPS32/64 (for 32- and 64-bit implementations, respectively). The early MIPS architectures were 32-bit only; 64-bit versions were developed later.

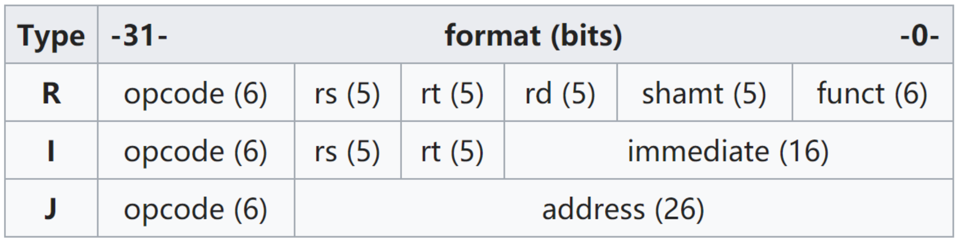
1.2 MIPS Instructions

MIPS I has thirty-two 32-bit general-purpose registers (GPR). Register \$0 is hardwired to zero and writes to it are discarded. Register \$31 is the link register. For integer multiplication and division instructions, which run asynchronously from other instructions, a pair of 32-bit registers, HI and LO, are provided. There is a small set of instructions for copying data between the general-purpose registers and the HI/LO registers.

The program counter has 32 bits. The two low-order bits always contain zero since MIPS I instructions are 32 bits long and are aligned to their natural word boundaries.

Instructions are divided into three types: R, I and J. Every instruction starts with a 6-bit opcode. In addition to the opcode, R-type instructions specify three registers, a shift amount field, and a function field; I-type instructions specify two registers and a 16-bit immediate value; J-type instructions follow the opcode with a 26-bit jump target.

The following are the three formats used for the core instruction set:



Type	-31-		format (bits)			-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Figure 1: MIPS Instruction Format

2 Project Description

2.1 Object

This project is an assembler for MIPS assembly language. Basically, the program takes an input of MIPS assembly language file (, assembles it, then generates a simple output file. The output file contains machine code for each instruction. The file should be able to run on the MIPS Simulator.

2.2 Assembler Syntax

Comments in assembler files begin with a sharp sign (#). Everything from the sharp sign to the end of the line is ignored.

Identifiers are a sequence of alphanumeric characters, underbars (_), and dots(.) that do not begin with a number. Instruction opcodes are reserved words that cannot be used as identifiers. Labels are declared by putting them at the beginning of a line followed by a colon.

Numbers are base 10 by default. If they are preceded by 0x, they are interpreted as hexadecimal. Hence, 256 and 0x100 denote the same value. Strings are enclosed in double quotes ("). Special characters in strings follow the C convention.

3 Project Analysis

3.1 General Thought

Generally, MIPS assembly language has three kinds of instructions, which are R, I and J format. Different kinds instruction has different formats to write; therefore, the the general idea of the project is that separate the instructions into specific kinds. The instructions that are classified in the same group can be use a uniform method to process.

3.2 Specific Steps

The program firstly preprocesses the instructions, deletes redundant invalid characters (white space, tab, etc.) to convenience follow-up work.

Then the instructions will be divided into R, I and J format to be processed respectively. The R and I instructions are more complicate than J. Therefore R and I instructions will be respectively divided into 7 types to be processed.

Next, the program extract the function name and parameters in each instruction. The function name can be used to distinguish which type this instruction is. Different types will be processed differently.

At last, all the results will be outputted in order.

3.3 Time Complexity

N: The total number of instructions is denoted as N.

M: The maximum characters in one instruction is denoted as M.

The time complexity is $O(N) + O(NM) = O(NM)$.

Since the M is a constant. The overall time complexity is $O(N)$.

4 Test

4.1 File Details

Source Code Filename: CSC3050_Project1.cpp

Input Filename: testfile1.asm testfile2.asm

Standard Output Filename: output1.txt output2.txt

Output Filename: output.txt

4.2 Virtual System Configuration

VMware Workstation version: VMware Workstation 15.5 Pro

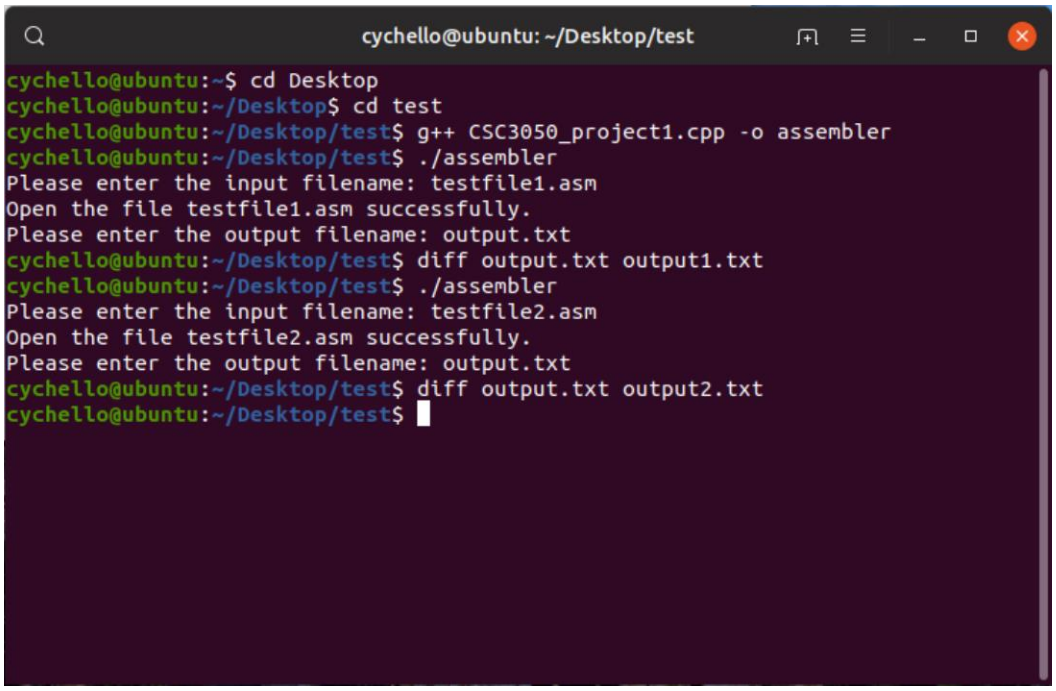
System: Linux version 5.0.0-31-generic (buildd@lcy01-amd64-010) (gcc version 8.3.0 (Ubuntu 8.3.0-6ubuntu1)) #33-Ubuntu SMP Mon Sep 30 18:51:59 UTC 2019

4.3 Compile and Execute

Compile: g++ CSC3050_Project1.cpp -o assembler

Execute: ./assembler

Compare: diff $filename1$ $filename2$ (standard output and practical output)



```
cychello@ubuntu:~$ cd Desktop
cychello@ubuntu:~/Desktop$ cd test
cychello@ubuntu:~/Desktop/test$ g++ CSC3050_project1.cpp -o assembler
cychello@ubuntu:~/Desktop/test$ ./assembler
Please enter the input filename: testfile1.asm
Open the file testfile1.asm successfully.
Please enter the output filename: output.txt
cychello@ubuntu:~/Desktop/test$ diff output.txt output1.txt
cychello@ubuntu:~/Desktop/test$ ./assembler
Please enter the input filename: testfile2.asm
Open the file testfile2.asm successfully.
Please enter the output filename: output.txt
cychello@ubuntu:~/Desktop/test$ diff output.txt output2.txt
cychello@ubuntu:~/Desktop/test$
```

Figure 2: Running Result in Ubuntu

5 Solution

5.1 Technique

5.1.1 Input File Path Check

Program will prompt the user to input the filename to read in. If the user input the wrong input filename, the program will prompt ”Cannot open the file” and the user can input filename again.

To realize this function, the program use a while loop to detect the failure when opening the file . Once the failure exists in the input stream, the program use .clear() to clear the error in the stream and ask the user to input filename again.

5.1.2 STL vector

The number of instructions are unknown to the program before. Vector, a data structure in library STL, is very useful to handle this problem. The capacity of a vector is dynamic growth. It does not need a fixed capacity like an array. Therefore, the program uses a vector to store the MIPS instructions.

5.1.3 STL map

Map, a data structure in library STL, is very useful to deal with the complicated instructions. This data structure can establish an one-to-one relationship between opcode and functions. It makes searching respective opcode faster and easier.

5.1.4 Switch and Case

Switch-case is used to process multiple branch situation. The instructions have many types, so that using if-else will make the code complicated and hard to debug.

5.2 Implementation

5.2.1 Parameters (Global Variable)

const int R = 0, I = 1, J = 2 : the constants used to identify different funtions.

const int Rnum = 32, Inum = 36, Jnum = 2, Renum = 32: the numbers of funtions.

int MIPSnum: the number of the input instructions.

vector<string> MIPS: a vector contains all the input instructions.

map<string, int> tags: a map contains all the labels.

map<string, string> Reg: a map contains all the Register name and numbers.

map<string, string> Rm: a map contains all the R format functions and relative code.

map<string, string> Im: a map contains all the I format functions and relative code.

map<string, string> Jm: a map contains all the J format functions and relative code.

string Register[N]: a string array contains all the Register names.

string Register_Code[N]: a string array contains all the Register codes.

string Rtype[N]: a string array contains all the R-format function names.

string Rtype_Code[N]: a string array contains all the R-format function codes.

string Itype[N]: a string array contains all the I-format function names.

string Itype_Code[N]: a string array contains all the I-format function codes.

string Jtype[N]: a string array contains all the J-format function names.

string Jtype_Code[N]: a string array contains all the J-format function codes.

vector<string> Rtype1 ... Rtype7: the classification of 7 types R-format instructions.

vector<string> Itype1 ... Itype7: the classification of 7 types T-format instructions.

5.2.2 Functions

int Type(string func) : Distinguish the R, I and J format instructions.

bool invalid(char ch) : Delete the invalid characters.

string clear_invalid(string c) : Erase invalid characters in the string.

string int_to_string(int n) : Convert a int variable to a string.

string decimal_to_binary(string num, int bit) : Convert a decimal number to a fixed-bit binary number.

int Rclassify(string func) : Classify R format commands.

string Rcommand(string c) : Process single R format command in MIPS.

int Iclassify(string func) : Classify I format commands.

string Icommand(string c, int line) : Process single I format command in MIPS.

string Jcommand(string c) : Process single J format command in MIPS.

string command(string c, int line) : Process single command in MIPS.

void init() : Read a MIPS file. Preprocess the commandes for transformation. Load in the function and register code.

void work() : Process all commands and output results.

5.3 Core Code

Process single R format command in MIPS.

```
string Rcommand(string c)
{
    int begin = 0, end = c.find('␣');
    // Extract the function name.
    string func = clear_invalid(c.substr(begin, end - begin));
    string op;
    vector<string> op_code;
    string mach = "00000000000000000000000000000000";
    mach.replace(26, 6, Rm[func]);

    begin = end + 1;
    end = c.find(',', begin);

    while (end != string::npos)      // Extract the parameters.
    {
        op = clear_invalid(c.substr(begin, end - begin));
        if (op[0] == '$')
            op_code.push_back(Reg[op]);
        else
            op_code.push_back(decimal_to_binary(op, 5));

        begin = end + 1;
        end = c.find(',', begin);
    }

    op = clear_invalid(c.substr(begin, c.size() - begin));
    if (op[0] == '$')
        op_code.push_back(Reg[op]);
    else
        op_code.push_back(decimal_to_binary(op, 5));

    switch (Rclassify(func))
    // Process the commands with different types in R format.
    {
```

```
    case 1:
        mach.replace(6, 5, op_code[1]);
        mach.replace(11, 5, op_code[2]);
        mach.replace(16, 5, op_code[0]);
        break;
    case 2:
        mach.replace(6, 5, op_code[0]);
        mach.replace(11, 5, op_code[1]);
        break;
    case 3:
        mach.replace(6, 5, op_code[2]);
        mach.replace(11, 5, op_code[1]);
        mach.replace(16, 5, op_code[0]);
        break;
    case 4:
        mach.replace(6, 5, op_code[0]);
        mach.replace(16, 5, op_code[1]);
        break;
    case 5:
        mach.replace(6, 5, op_code[0]);
        break;
    case 6:
            mach.replace(16, 5, op_code[0]);
            break;
        case 7:
            mach.replace(11, 5, op_code[1]);
            mach.replace(16, 5, op_code[0]);
            mach.replace(21, 5, op_code[2]);
            break;
        }

        return mach;
}
```

Process single I format command in MIPS.

```
string Icommand(string c, int line)
{
    int begin = 0, end = c.find('␣'), left, right;
    // Extract the function name.
    string func = clear_invalid(c.substr(begin, end - begin));
    vector<string> op;
    string mach = "00000000000000000000000000000000";
    begin = end + 1;
    end = c.find(',', begin);
```

```cpp

    while (end != string::npos)      // Extract the parameters.
    {
        op.push_back(clear_invalid(
            c.substr(begin, end - begin)));
        begin = end + 1;
        end = c.find(',', begin);
    }

    op.push_back(clear_invalid(
        c.substr(begin, c.size() - begin)));

    // Process the commands with different types in I format.
    switch (Iclassify(func))
    {
    case 1:
        mach.replace(0, 6, Im[func]);
        mach.replace(6, 5, Reg[op[1]]);
        mach.replace(11, 5, Reg[op[0]]);
        mach.replace(16, 16, decimal_to_binary(op[2], 16));
        break;
    case 2:
        mach.replace(0, 6, "000001");
        mach.replace(6, 5, Reg[op[0]]);
        mach.replace(11, 5, Im[func]);
        mach.replace(16, 16, decimal_to_binary(op[1], 16));
        break;
    case 3:
        mach.replace(0, 6, Im[func]);
        mach.replace(11, 5, Reg[op[0]]);
        mach.replace(16, 16, decimal_to_binary(op[1], 16));
        break;
    case 4:
        mach.replace(0, 6, Im[func]);
        mach.replace(6, 5, Reg[op[0]]);
        mach.replace(11, 5, Reg[op[1]]);
        mach.replace(16, 16, decimal_to_binary(
            int_to_string(tags[op[2]] - line - 1), 16));
        break;
    case 5:
        mach.replace(0, 6, Im[func]);
        mach.replace(6, 5, Reg[op[0]]);
        mach.replace(16, 16, decimal_to_binary(
            int_to_string(tags[op[1]] - line - 1), 16));
            break;
        case 6:
            mach.replace(0, 6, "000001");
            mach.replace(6, 5, Reg[op[0]]);
            mach.replace(11, 5, Im[func]);
            mach.replace(16, 16, decimal_to_binary(
                int_to_string(tags[op[1]] - line - 1), 16));
            break;
        case 7:
            mach.replace(0, 6, Im[func]);
            mach.replace(11, 5, Reg[op[0]]);
            left = op[1].find('(', 0);
            right = op[1].find(')', 0);
            mach.replace(6, 5, Reg[clear_invalid(
                op[1].substr(left + 1, right - left - 1))]);
            mach.replace(16, 16, decimal_to_binary(
                clear_invalid(op[1].substr(0, left)), 16));
            break;
    }
    return mach;
}
```

Process single J format command in MIPS.

```cpp
string Jcommand(string c)
{
    int begin = 0, end = c.find('␣');
    // Extract the function name.
    string func = clear_invalid(c.substr(begin, end - begin));
    string op, jline;
    string mach = "00000000000000000000000000000000";

    mach.replace(0, 6, Jm[func]);

    begin = end + 1;
    end = c.size();
    op = clear_invalid(c.substr(begin, end - begin));
    jline = int_to_string(tags[op]);

    mach.replace(6, 26, decimal_to_binary(jline, 26));

    return mach;
}
```