CSC3050 Project 2 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.

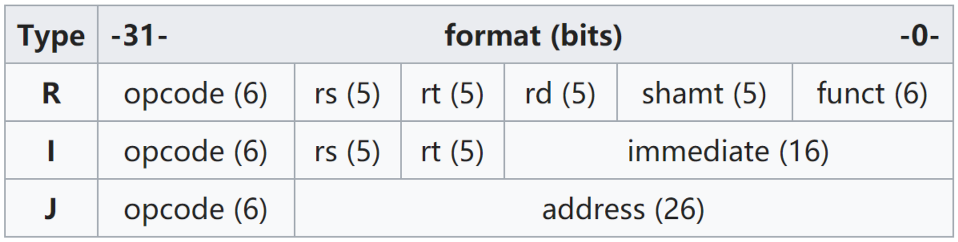
1.2 MIPS Instructions

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 program is called a MIPS Simulator, in which it will take a MIPS code input and simulate executing the code. For example, if an adder MIPS code is given, this program should prompt the user to enter two integers and output the correct answer.

2.2 Functions

The program executes the MIPS code just like how a machine does it.

1. Simulate memory. Allocate a block of memory of 6MB, which is used to simulate the main memory. Reserve 1MB for text section. Map the memory address of the block allocated to 0x00400000, which is a 32 bit address.

2. Allocate and maintain memory spaces to simulate the regular 32 registers. Correctly use them, including $\$fp$, $\$sp$, etc. Allocate 4 bytes for each register.

3. Maintain a PC, which indicates the next instruction to load.

4. Maintain a code section in the simulated memory, where the assembled MIPS machine

code is put.

5. Maintain a data section in the simulated memory, where the pre-defined data in MIPS code (.data section) is put. Located on top of text section. For dynamic data, it grows towards higher memory address. The first piece of data in .data section of MIPS code will be put at the lowest memory address of the data segment. The rule to put data is each piece of data is going to occupy the full block (4 bytes) even if it used only part of a block. For example, a string of length 11 is using 11 bytes in memory, and put at address 0x0. Then it used up two full blocks (8 bytes), and 3/4 of the third block, which starts at 0x8. The next piece of data should not be put at address 0xB, instead, it should be put at address 0xC.

6. Maintain a stack section in the simulated memory, which grows towards lower memory address.

7. Simulate what each instruction does using C/C++. This is done by writing a function for each instruction.

8. Simulate the execution and follow the machine cycle. Judge which instruction it is, call the function.

9. Support *syscall* instruction.

2.3 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 and execute them respectively. 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 all the text and data will be written into memory.

At last, the instructions will be executed respectively.

3.3 Time Complexity

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

M: The maximum operations 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_Project2.cpp MIPS_to_Binary.cpp

Header File: MIPS_to_Binary.h

4.2 Virtual System Configuration

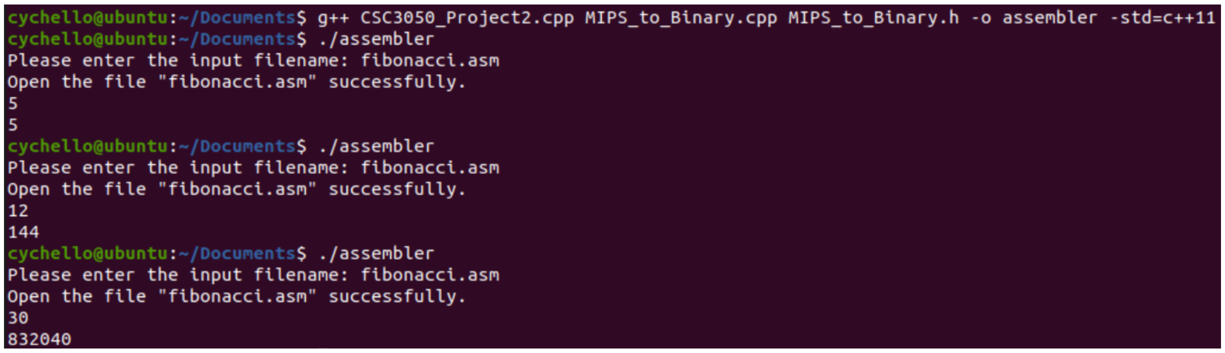
VMware Workstation version: VMware Workstation 15.5 Pro

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_Project2.cpp MIPS_to_Binary.cpp MIPS_to_Binary.h -o assembler -std=c++11

Execute: ./assembler



```
cychello@ubuntu:~/Documents$ g++ CSC3050_Project2.cpp MIPS_to_Binary.cpp MIPS_to_Binary.h -o assembler -std=c++11
cychello@ubuntu:~/Documents$ ./assembler
Please enter the input filename: fibonacci.asm
Open the file "fibonacci.asm" successfully.
5
5
cychello@ubuntu:~/Documents$ ./assembler
Please enter the input filename: fibonacci.asm
Open the file "fibonacci.asm" successfully.
12
144
cychello@ubuntu:~/Documents$ ./assembler
Please enter the input filename: fibonacci.asm
Open the file "fibonacci.asm" successfully.
30
832040
```

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 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.

5.1.3 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, S = 3 : The constants used to identify different funtions.

const int memory_size = 6 * 1024 * 1024 * sizeof(char) : The capacity of simulation memory.

const int reg_size = 34 * sizeof(int) : The capacity of simulation register.

const int text_address = 0x00400000 : The simulated head address of text section in the memory.

const int data_address = 0x00500000 : The simulated head address of data section in the memory.

const int data_type = 0, text_type = 1 : The constants used to identify different MIPS.

int MIPSnum, datanum : The number of the input instructions and data.

void* reg : The real head address of register section in the memory.

void* start : The real head address of text section in the memory.

void* dataend : The simulated end address of data section in the memory.

vector¡string¿ MIPS : A vector contains all instructions.

vector¡string¿ dataset : A vector contains all data.

5.2.2 Important Functions

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

void work() : Process all commands and simulate running.

5.3 Core Code

Load the data section into memory.

```
void load_data(void* head)
{
    int i, j, begin, end, space;
    char c;
    string type, str;

    for (i = 0; i < datanum; i++)
    {
    end = dataset[i].find(":");
    if (end != -1)
        dataset[i].erase(0, end + 1);
    dataset[i] = clear_invalid(dataset[i]);

    end = dataset[i].find('␣');
    if (end != -1)
```

```
{
type = clear_invalid(dataset[i].substr(0, end));
dataset[i].erase(0, end);
}
if (type == ".asciiz")
{
begin = dataset[i].find("\"");
end = dataset[i].find("\"", begin + 1);
str = dataset[i].substr(begin + 1, end - begin - 1);
str = str + '\0';
space = ceil((double)str.size() / 4.0) * 4;
memcpy(head, str.c_str(), str.size());
head = (void*)((long long)(head)+space);
}

if (type == ".ascii")
{
begin = dataset[i].find("\"");
end = dataset[i].find("\"", begin + 1);
str = dataset[i].substr(begin + 1, end - begin - 1);

space = ceil(str.size() / 4) * 4;
```

```
    for (j = 0; j < str.size(); j++)
    {
        c = str[j];
        memcpy(head, &c, sizeof(char));
        head = (void*)((long long)(head)+1);
    }
    head = (void*)((long long)(head)+space - str.size());
    }

    if (type == ".word")
    {
    begin = 0;
    end = dataset[i].find(",", begin);
    while (end != -1)
    {
    str = clear_invalid(dataset[i].substr(begin, end - begin));
    int tmp = str_to_int(str);
    memcpy(head, &tmp, sizeof(int));

    head = (void*)((long long)(head)+sizeof(int));

    begin = end + 1;
    end = dataset[i].find(",", begin);
    }
    str = clear_invalid(dataset[i].substr(begin, end - begin));
    int tmp = str_to_int(str);
    memcpy(head, &tmp, sizeof(int));

    head = (void*)((long long)(head)+sizeof(int));
    }

    if (type == ".half")
    {
    begin = 0;
    end = dataset[i].find(",", begin);
    while (end != -1)
    {
    str = clear_invalid(dataset[i].substr(begin, end - begin));
    short tmp = str_to_int(str);
    memcpy(head, &tmp, sizeof(short));

    head = (void*)((long long)(head)+sizeof(short));

    begin = end + 1;
    end = dataset[i].find(",", begin);
    }
    str = clear_invalid(dataset[i].substr(begin, end - begin));
    short tmp = str_to_int(str);
    memcpy(head, &tmp, sizeof(short));

    head = (void*)((long long)(head)+sizeof(short));
    }

    if (type == ".byte")
    {
    begin = 0;
    end = dataset[i].find(",", begin);
    while (end != -1)
    {
    str = clear_invalid(dataset[i].substr(begin, end - begin));
    char tmp = str_to_int(str);
    memcpy(head, &tmp, sizeof(int));

    head = (void*)((long long)(head)+sizeof(int));

    begin = end + 1;
```

```
        end = dataset[i].find(",", begin);
        }
        str = clear_invalid(dataset[i].substr(begin, end - begin));
        char tmp = str_to_int(str);
        memcpy(head, &tmp, sizeof(char));

        head = (void*)((long long)(head)+sizeof(char));
        }
        }
        dataend = head;
}
```