LE/EECS 2021 4.00   Computer Organization

MIPS SIMULATOR LAB REPORT

Michael Williams 211087798

Section E

October 27 2014

The work in this report is my own. I have read and understood York University academic dishonesty policy and I did not violate the senate dishonesty policy in writing this report.

ABSTRACT

In this lab, we used our understandings of the fundamentals of MIPS and assembly code to create our very own MIPS simulator, taking input from a file, and executed the instructions within it, just as the program xspim would. The purpose was to test our knowledge regarding assembly code and to use that knowledge to generate a java program, capable of doing some of the same functions. Although what we produced doesn’t work or function exactly like other simulators, or quite as efficiently, it still does complete the specified tasks. Doing this also gave insight as to how difficult some of simulating MIPS is, as well as how easy other proportions become once you acquire a firm understanding of how the system works. For the most part, the code was rather simple, consisting of for loops, if statements and while loops: the basics. It became more of a thorough check in deciphering the specifications of each individual instruction and then figuring out how and where to apply these specified instructions. Along the way I personally found that it was a lot harder to get started and simply understand the basis and the idea of implementation than it was to actually implement. In conclusion, a MIPS simulator was created, designed to perform the specified tasks and more, as efficiently as I could, while maintaining user friendly input solutions for file extraction, while using as little code as possible.

EQUIPMENT

-laptop

-java eclipse

-putty

-xming

-xspim

-gedit

METHODS/PROCEDURES

At the beginning, the program proved to be incredibly difficult. I had very little understanding of even how to begin programming a simulation. After doing some research and reading parts of the text, it started becoming clear to me. The first thing I did, upon understanding the fundamentals to creating a MIPS simulator, was to draw up some notes and a very rough, step to step pseudocode, attached to the back of the lab. The first part of the code describes my path of action: First, was to take the input from the user itself. This was obviously essential and the best starting point. This meant stopping at certain points, as to let the program then later decipher and execute the instruction, and looking at what the actual sent line was, either a model version number, or the actual hex line to be executed. Next, was to decipher what the actual coding meant. To do this, I had to essentially take the hex code sent, convert to binary, and then break the binary code into various bits of different sizes, dependent on the type of instruction being made, that being op, rs, rt, rd, shamt, funct, and the constant/address. The most important ones at this point were op and funct as they would be the 2 values to distinguish both the type of instruction sent (I-type or R-type), as well as the specific R-function, if it were an R-type instruction. In coding, this simply meant checking to see if the op code was 0. If it was then it was an R-type instruction. Else, it was an I-type instruction. This is roughly shown on the back of the lab, in the chart. Next, was simply executing the instructions itself. Because I knew what each part of the now binary code meant, and what registers it applied to, it was just a simple task of taking values from registers, or the constant/address and storing values into new registers, all specified by the code. At least, that is, after converting the binary to actual decimal so that the program would know which registers to store it into. This probably could have been done without converting to decimal, but would have also required more coding along the way, something I aimed to reduce as much as possible. After using the op and funct codes to figure out exactly what operation was to be conducted, it was easy to use this information, along with if statements, to isolate each of the unique instructions, and specifically adhere to it. Last, but certainly not least, was printing out the registers. Doing this was fairly simple. I just created a set of register clones, in which the values were all converted back to and treated as hexadecimal strings. This, along with a little formatting to specify the size of the output and a series of print statements, got the job done quite well, and as wanted by the professor. It should be noted that in the coding, for a better understanding, as well as to keep proper coding properties and efficiency, the entirety of the program is broken into 3 methods. The first being the main method, is used to take in a file from the user. The other 2, called from the main method, simply decipher the code and apply the instructions, and print out the registers in the matter specified. I figured this was best, as to not clutter coding up and make it increasingly difficult to read and understand.

RESULTS

The result shown below are comprised of both the input and the output generated by my program. For input, I used the example given by the prof.

INPUT

0

26310005

26520007

001290c0

02329821

0271a024

OUTPUT

$0:0x00000000 $1:0x00000000 $2:0x00000000 $3:0x00000000

$4:0x00000000 $5:0x00000000 $6:0x00000000 $3:7x00000000

$8:0x00000000 $9:0x00000000 $10:0x00000000 $11:0x00000000

$12:0x00000000 $13:0x00000000 $14:0x00000000 $15:0x00000000

$16:0x00000000 $17:0x00000005 $18:0x00000038 $19:0x0000003d

$20:0x00000005 $21:0x00000000 $22:0x00000000 $23:0x00000000

$24:0x00000000 $25:0x00000000 $26:0x00000000 $27:0x00000000

$28:0x00000000 $29:0x00000000 $30:0x00000000 $31:0x00000000

DISCUSSION

There wasn’t much need to interpret or analyse the results above. The explanation in the above methods/procedure section as well as the notes made in the results section should more than suffice to give the reader a proper understanding of how the program works, and what the results mean, given that they understand how to decipher hexadecimal, and use the chart at the back of the lab as reference to what each broken op code or funct code represents, the op and funct code being the first and last 6 bits of the hexadecimal code, when converted to binary, respectively. However, in terms of difficulty, there is some discussion to be had. In terms of programming this code, it was incredibly difficult when started, mostly because I had no idea where to begin, or how to properly use or interpret the hexadecimal code provided. However, after reading the textbook a bit, I acquired a better understanding of what the code actually meant. Further reading would show me the specific op and funct codes for individual instructions, as well as the registers used and needed in the instruction itself. Once I understood this, the program became fairly easy. It consisted mainly of just creating many if statements to adhere to the many possible combinations. I did come into more trouble when trying the bonus instructions, as these required storing and loading values, which meant making something that could hold these values, just as described in the textbook, and provided a space to use as reference to memory. However, the solution I made works well. The only other problem I faced was when I was printing the registers out, for 2 reasons. First was because I wanted to use a more programmer efficient method of printing out the registers. However, this would require more if statements, which would result in increasing the time to actually execute the program itself. Hence, it would actually be less efficient. In the end, although not the best looking, the most efficient choice was to just hardcode print statements for each of the registers. The second issue was converting the registers to hexadecimal without creating a set of clone registers to do so. Again, for the sake of efficiency, I found it better to just make the clone, instead of converting them to Hexadecimal within the real registers, as this would create complications, as well as the need to re-convert them back to integer value in order to work properly with the remainder of the program.

CONCLUSION

In conclusion, although the lab provided me with a lot of trouble in the beginning, I developed and understanding of the essential information needed, and I achieved everything I sought to accomplish with this lab, as well as the bonus instructions. With the exception of the beginning, and a few minor problems along the way, creating the MIPS simulator did not provide me with too much trouble. In the end, a MIPS simulator was created, designed to perform the specified tasks and more, as efficiently as I could, while maintaining user friendly input solutions for file extraction, while using as little code as possible.

APPENDIX

package mipsSimulator;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class MIPS {

static int register[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

//creates an array of 32 registers, all initialized at 0

static int memory[] = new int[50];

public static void main(String [] args){

//System.out.println(System.getProperty("user.dir"));

Scanner x = new Scanner(System.in);

System.out.println("Enter file name \nMake sure you include extension as well.");

System.out.println("And make sure file is in directory " + System.getProperty("user.dir"));

String fileName = x.nextLine();

//The above will take a file from user, assuming it is in the requested format

int mode = 0; //determines mode

int line = 0; //used to distinguish first line from all others

File file = new File(fileName);

try {

Scanner input = new Scanner(file);

while(input.hasNextLine()){

if(line == 0){

mode = input.nextInt();

input.nextLine();

line ++;

}//at first line take mode value for later

else{

execute(input.nextLine());//will send Line over for steps 2 and 3

if(mode == 1){

printReg();

}//at this point if mode = 1, print out all registers

}// else loop

}//while loop to traverse through file

} catch (FileNotFoundException e1) {

System.out.println("File not found.");

// e1.printStackTrace();

}//catch if file is not found

//the try catch statement above will read the file, 1 line at a time

if(mode == 0){

printReg();

}//At this point if mode = 0, print out all registers, else do nothing.

//System.out.println(mode);

}//main method/tester. This main method focuses on taking the input from the user, and sending

//each line over to another method to convert it to binary and execute instructions.

//After that it is returned here and the registers are printed out.

public static void execute(String hex){

String num = Long.toBinaryString(Integer.parseInt(hex, 16));

String mask = "00000000000000000000000000000000";

String binary = mask + num; //binary now contains the binary, 32-bit version of hex.

binary = binary.substring(binary.length() - 32, binary.length()); // take the right-most 32 digits

// System.out.println(binary);

// String op = binary.substring(0, 6);

//String rs = binary.substring(6, 11);

//String rt = binary.substring(11, 16);

//String rd = binary.substring(16, 21);

//String shamt = binary.substring(21, 26);

//String funct = binary.substring(26);

//String address = binary.substring(16);

int op = Integer.parseInt(binary.substring(0, 6), 2);

int rs = Integer.parseInt(binary.substring(6, 11), 2);

int rt = Integer.parseInt(binary.substring(11, 16), 2);

int rd = Integer.parseInt(binary.substring(16, 21), 2);

int shamt = Integer.parseInt(binary.substring(21, 26), 2);

int funct = Integer.parseInt(binary.substring(26), 2);

int address = Integer.parseInt(binary.substring(16), 2);

//System.out.println(op);

if( op == 0){

//System.out.println(funct);

if(funct == 0){

register[rd] = register[rt] << shamt;

//System.out.println(rd);

//System.out.println(rt);

//System.out.println(shamt);

//System.out.println(register[rd]);

//System.out.println("sll");

}//if funct == sll

else if(funct == 2){

register[rd] = register[rt] >>> shamt;

// System.out.println(register[rd]);

// System.out.println("srl");

}//if funct == srl

else if(funct == 33){

register[rd] = register[rs] + register[rt];

//System.out.println("addu");

}//if funct == addu, whats the difference between this and add?

else if(funct == 32){

register[rd] = register[rs] + register[rt];

// System.out.println("add");

}//if funct == add

else if(funct == 36){

register[rd] = register[rs] & register[rt];

// System.out.println("and");

}//if funct == and

else if(funct == 39){

register[rd] = ~(register[rs] | register[rt]);

// System.out.println("nor");

}//if funct == nor

else if(funct == 42){

int x;

if(register[rs] < register[rt]){x = 1;}

else{x = 0;}

register[rd] = x;

// System.out.println("slt");

}//if funct == slt

else if(funct == 43){

int x;

if(register[rs] < register[rt]){x = 1;}

else{x = 0;}

register[rd] = x;

// System.out.println("sltu");

}//if funct == sltu, whats the difference between slt and sltu?

else if(funct == 12){

System.out.println("Program ended by syscall");

//stop everything at this point

return;

}//if syscall

else{

// System.out.println("funct value in " + hex + "\nwas not recognized. No action will be done");

}//if funct code is not recognized

}//if op == 0

else if(op == 9){

register[rt] = register[rs] + address;

// System.out.println("addiu");

}//if op is addiu

else if(op == 10){

int x;

if(register[rs] < address){x = 1;}

else{x = 0;}

register[rt] = x;

// System.out.println("slti");

}// if op is slti

else if(op == 15){

register[rt] = (address << 16) | 0;

// System.out.println("lui");

}//if op is lui

else if(op == 32){

int x = address + rs;

if(x > memory.length - 1){

int[] mem2=new int[memory.length + x];

for(int i = 0; i < memory.length; i++){

mem2[i] = memory[i];

}//put all members of memory into mem2

memory = mem2;

} //if memory array is too small, make it bigger

register[rt] = memory[x];

// System.out.println("lb");

}//if op is lb

else if(op == 35){

int x = address + rs;

if(x > memory.length - 1){

int[] mem2=new int[memory.length + x];

for(int i = 0; i < memory.length; i++){

mem2[i] = memory[i];

}//put all members of memory into mem2

memory = mem2;

} //if memory array is too small, make it bigger

register[rt] = memory[x];

// System.out.println("lw");

}// if op is lw/ Does not account for loading individual byte locations

else if(op == 36){

int x = address + rs;

if(x > memory.length - 1){

int[] mem2=new int[memory.length + x];

for(int i = 0; i < memory.length; i++){

mem2[i] = memory[i];

}//put all members of memory into mem2

memory = mem2;

} //if memory array is too small, make it bigger

register[rt] = memory[x];

// System.out.println("lbu");

}// op is lbu

else if(op == 40){

int x = address + register[rs];

if(x > memory.length - 1){

int[] mem2=new int[memory.length + x];

for(int i = 0; i < memory.length; i++){

mem2[i] = memory[i];

}//put all members of memory into mem2

memory = mem2;

} //if memory array is too small, make it bigger

memory[x] = register[rt];

// System.out.println("sb");

}//if op is save byte

else if(op == 43){

int x = address + rs;

if(x > memory.length - 1){

int[] mem2=new int[memory.length + x];

for(int i = 0; i < memory.length; i++){

mem2[i] = memory[i];

}//put all members of memory into mem2

memory = mem2;

} //if memory array is too small, make it bigger

memory[x] = register[rt];

// System.out.println("sw");

}//if op is sw

else{

System.out.println("The op value in " + hex + "\nwas not recognized. No action will be performed");

}//if op is not recongnized

}//execute will take the hex string sent from the user. This method combines steps 2 and 3.

//It will decode the hex string into binary, and break the binary code into its instructions

//After that it will execute the said instructions.

public static void printReg(){

String reg[] = new String[32];

for(int i = 0; i < 32; i ++){

reg[i] = Integer.toHexString(register[i]);

while(reg[i].length() < 8)

reg[i] = "0" + reg[i];

}

System.out.print("$0:0x" + (reg[0]) + " ");

System.out.print("$1:0x" + (reg[1]) + " ");

System.out.print("$2:0x" + (reg[2]) + " ");

System.out.print("$3:0x" + (reg[3]) + " ");

System.out.println();

System.out.print("$4:0x" + (reg[4]) + " ");

System.out.print("$5:0x" + (reg[5]) + " ");

System.out.print("$6:0x" + (reg[6]) + " ");

System.out.print("$7:0x" + (reg[7]) + " ");

System.out.println();

System.out.print("$8:0x" + (reg[8]) + " ");

System.out.print("$9:0x" + (reg[9]) + " ");

System.out.print("$10:0x" + (reg[10]) + " ");

System.out.print("$11:0x" + (reg[11]) + " ");

System.out.println();

System.out.print("$12:0x" + (reg[12]) + " ");

System.out.print("$13:0x" + (reg[13]) + " ");

System.out.print("$14:0x" + (reg[14]) + " ");

System.out.print("$15:0x" + (reg[15]) + " ");

System.out.println();

System.out.print("$16:0x" + (reg[16]) + " ");

System.out.print("$17:0x" + (reg[17]) + " ");

System.out.print("$18:0x" + (reg[18]) + " ");

System.out.print("$19:0x" + (reg[19]) + " ");

System.out.println();

System.out.print("$20:0x" + (reg[20]) + " ");

System.out.print("$21:0x" + (reg[21]) + " ");

System.out.print("$22:0x" + (reg[22]) + " ");

System.out.print("$23:0x" + (reg[23]) + " ");

System.out.println();

System.out.print("$24:0x" + (reg[24]) + " ");

System.out.print("$25:0x" + (reg[25]) + " ");

System.out.print("$26:0x" + (reg[26]) + " ");

System.out.print("$27:0x" + (reg[27]) + " ");

System.out.println();

System.out.print("$28:0x" + (reg[28]) + " ");

System.out.print("$29:0x" + (reg[29]) + " ");

System.out.print("$30:0x" + (reg[30]) + " ");

System.out.print("$31:0x" + (reg[31]) + " ");

System.out.println();

System.out.println("========================================");

}//prints all registers

}//MIPS CLASS