joy-courses / cse220-hw1-WENSHIHAOO Private

Code Issues Pull requests Actions Projects Security

main cse220-hw1-WENSHIHAOO / README.md Go to file ...

github-classroom Initial commit Latest commit b4329e0 yesterday History

1 contributor

145 lines (78 sloc) | 17.2 KB Raw Blame

Homework 1

Learning Outcomes

After completion of this assignment, you should be able to:

- Implement non-trivial bit manipulation algorithms.
- Work with fixed-length and null-terminated strings.
- Write programs that require conditional execution.

Getting Started

To complete this homework assignment, you will need the MARS simulator. Download it from Blackboard under *Course Documents*. You can write your programs in the MARS editor itself. You can choose to use other text editors if you are not comfortable with the MARS editor. At any point, if you need to refer to instructions click on the *Help* tab in the MARS simulator.

Read the rest of the document carefully. This document describes everything that you will need to correctly implement the homework and submit the code for testing.

You should have already setup Git and configured it to work with SSH. If you haven't then do Homework 0 first!

The first thing you need to do is download or clone this repository to your local system. Use the following command:

```
$ git clone <ssh-link>
```

After you clone, you will see a directory of the form *cse220-hw1-*, where *username* is your GitHub username.

In this directory, you will find two asm files: *hw1.asm* and *hw1_test.asm*. The file *hw1.asm* is where you will write your code in. At the top of the file you will find hints to fill your full name, NetID, and SBU ID. Please fill them up accurately. This information will be used to collect your scores from GitHub. If you do not provide this information, your submission will not be graded. The *hw1_test.asm* file is a template that has been provided for preliminary testing. You can change the values in the labels *args* and *n* to test different cases for each part (described later). You must run *hw1_test.asm* to run *hw1.asm*. The *tests* directory contains the test cases for this homework. You can use the test cases as specifications to guide your code. Your goal should be to pass all the tests. If you do so, then you are almost guaranteed to get full credit. None of the files in the *tests* directory should be modified. If you do, you will receive no credit for the homework.

Note the hw1.asm file has a .data section. Do not modify this section.

You should start writing code in hw1.asm at the label start_coding_here. Nothing before this label should change, except for filling in your name and IDs.

Assembling and Running Your Program in MARS

To execute your MIPS programs in MARS, you will first have to assemble the program. Click on the *assemble* option in the *Run* tab at the top of the editor. If the instructions in your program are correctly specified, the MARS assembler will load the program into memory. You can then run the program by selecting the *Go* option in the same *Run* tab. To debug your program, add breakpoints. This is done after assembling the program. Select the *execute* tab, you will see the instructions in your program. Each instruction will have a checkbox associated with it. Clicking on the checkbox will add a breakpoint, that is, when the program is run, control will stop at that instruction allowing you to inspect the registers and memory up to that point. The execute tab will show you the memory layout in the bottom pane. The right hand pane shows list of registers and their values.

Note in order to run hw1.asm, you will have to assemble and run hw1_test.asm.

How to read the test cases

As mentioned previously, the *tests* folder contains a test file. The contains a test to verify a specific feature. Each test is a Java function with a name prefixed with *verify_*. In each of these functions, you will find one or more assert statements that need to be true for the test to pass. These asserts compare the expected result with the result in registers *$a0* or/and *$a1* in your program. The tests assume that the expected results will be in these registers as specified in the later parts of this readme. In each test, we will execute your program by calling the *run* function along with the necessary arguments required to test that particular feature. If a test fails, the name of the failing test will be reported with an error message. You can use the error message from the failing test to diagnose and fix your errors. Another way is to look at the inputs of the failed test case and plug them into appropriate labels in *hw1_test.asm* and run *hw1_test.asm* to debug your program.

Do not change any files in the tests directory. If you do then you won't receive any credit for this homework assignment.

Problem Specification

Write a MIPS program to do the following:

- Convert a numeric string to a signed decimal number.
- Decode I-type instructions.
- Convert a hexadecimal string to a floating point string in the 32-bit IEEE-754 format.
- Decode an input string to verify a player's hand in a card game called Loot.

We will implement the above mentioned requirements as described in the following parts.

Part 1: Validate Arguments

For the first part, we will validate the arguments that are expected as input by the program. By program, we mean the file *hw1.asm*. The program expects exactly 2 arguments as input. Assume that the no. of arguments provided as input to it are stored in the *num_args* label (see .data section in hw1.asm). If the no. of arguments provided as input is not 2 then your program must print the error message in the label **args_err_msg** and terminate.

The first argument is a null-terminated string with exactly 2 characters, including the null terminator. If the first argument is more than 2 characters long, then print error message in label *invalid_arg_msg* and terminate. Assume that the base address of the first argument's string is in the label *arg1_addr*. Recall that the base address of a string is the address of the first character in the string.

The first argument must be either "D", "O", "S", "T", "I", "F", or "L". Otherwise, print error message in label *invalid_arg_msg* and terminate.

Once this basic validation is complete we will move on to the next parts. Notice that the base address of the first argument to the program is in the label *arg1_addr* and the base address of the second argument is in the label *arg2_addr*.

Part 2: String to Decimal

In this part, we will convert a numeric string provided as input to the program and print the corresponding signed decimal number.For this operation, the user is expected to provide "D" as the first argument and a null-terminated string as the second argument where every character in the string is a digit [0-9]. If the numeric string is a negative number then the first character of this string will be the character '-'. For example, the two arguments could be "D" and the string "123" or it could be "D" and "-123". Recall that you can access the arguments from the labels *arg1_addr* and *arg2_addr*.

If the second argument contains invalid characters, that is, characters not in [-,0-9], then print the error message in the label *invalid_arg_msg* and terminate.

If the second argument is valid, then convert the string into a signed decimal number, print it and terminate the program. You can implement this conversion in a number of ways. One way to implement the conversion is to think about how decimal numbers are represented. For example, consider the numeric string "123". The decimal representation of this string can be obtained by extracting each digit (from the left) and multiplying it by 10 and adding it to the next digit. Repeating this calculation till the rightmost digit and then adding the rightmost digit will give us the desired unsigned decimal number. So, the numeric string "123" can be converted into the unsigned decimal number 123 using the computation ((1*10)+2)*10 + 3. You can generalize this with any N-digit numeric string. You can follow a similar algorithm with negative numbers with a small change to account for the negative sign.

Part 3: Decode I-Type Instructions

In this part, we will decode the different parts of an I-Type instruction from a hex string. Recall, an I-type instruction in MIPS has the following format:

- The 6 most significant bits for the *opcode*.
- The next 5 bits for the *rs* register.
- The next 5 bits for the *rt* register.
- The least 16 bits for the *immediate*.

Assume that the string provided as the second argument to our program is a hexadecimal encoding (also called hex string) of an I-type instruction. A valid hex string must begin with the characters "0x" and must contain valid hexadecimal characters [0-9] and [A-F] (uppercase). The length of the hex string excluding the characters "0x" must be at least 1 and at most 8. If the hex string has less than 8 hexadecimal digits then you can assume the upper hex digits are 0. For example, "0x8A" should be the same as "0x0000008A". If the hex string is not valid as per the above constraints then print the message in the label *invalid_arg_msg* in *hw1.asm* and terminate.

If the hex string is valid then, we will proceed to extract the fields, opcode, rs, rt, and immediate, of an I-type instruction from it. For example, assume that the second argument is the hexadecimal string “0x30E9FFFC“. The program must first convert the string to a 32-bit binary representation. For the string “0x30E9FFFC“, the 32-bit binary value will be:

0011 0000 1110 1001 1111 1111 1111 1100

Next, the program should reorganize the bits as per the operation provided in the first argument.

- **Opcode**. If the first argument is the character ‘O’, then the program should print the opcode field and terminate. In the example that we are discussing, the opcode field is the first 6 bits of the binary value (read left to right) since the number of bits in the opcode field of an I-type instruction is 6. Hence, the opcode should be 001100, which is 12 in decimal. The program should print this decimal value. You can assume that the opcode will always be unsigned. For example, for hex string “0x30E9FFFC“, the program should print 12 and terminate.

- **Source Register**. If the first argument is the character ‘S’, then the program should print the rs field and terminate. In the running example, the rs field is 5 bits of the binary value starting from 7th bit to the 11th bit (inclusive; read left to right). Hence, the rs field should be 00111, which is 7 in decimal. The program should print this decimal value. You can assume that the rs field will always be unsigned.

- **Destination Register**. If the first argument is the character ‘T’, then the program should print the rt field and terminate. So for the hexadecimal encoding 0x30E9FFFC, the rt field is 5 bits of the corresponding binary value starting from 12th bit to the 16th bit (inclusive; read left to right). Hence, the rt field should be 01001, which is 9 in decimal. The program should print this decimal value. You can assume that the rt field will always be unsigned.

- **Immediate**. If the first argument is the character 'I', then the program should print the immediate field and terminate. Assuming that the hexadecimal encoding provided as the second argument is 0x30E9FFFC, the immediate field is the last 16 bits of the corresponding binary value. Hence, the immediate should be 1111 1111 1111 1100. We will assume that the 16-bit immediate will always be signed. Since in this example, the MSB of the immediate is 1, the corresponding decimal value will be negative. In this case, it will be -4.

Part 4: Hex String to 32-bit Floating Point in IEEE 754

In this part, we will convert a string in hexadecimal to a 32-bit floating point number in IEEE 754 format. The first argument must be "F". The second argument must be an 8-character string with valid hexadecimal digits [0-9A-F]. If the second argument does not have valid hex digits or does not have exactly 8 characters then print the error message in the label *invalid_arg_msg* and terminate.

If both arguments are valid, then check if the hex string is a special floating-point number. A floating-point number is special if the hex string is one of the following:

- **00000000** or **80000000**. Zero; in this case print the message in the label *zero* and terminate.
- **FF800000**. -Inf; in this case print the message in the label *inf_neg* and terminate.
- **7F800000**. +Inf; in this case print the message in the label *inf_pos* and terminate.
- **7F800001** thru **7FFFFFFF**. Nan; in this case print the message in the label *nan* and terminate.
- **FF800001** thru **FFFFFFFF**. Nan; in this case print the message in the label *nan* and terminate.

If the hex string is *not* a special floating-point number, then calculate the 127-biased exponent and the 23-bit fractional part. Store the exponent in register $a0 and the 23-bit fractional part in a string prepended by the sign and the normalized 1. This string should be stored in the label *mantissa* and must be null-terminated. Store the base address of label *mantissa* in register $a1 and terminate. For example, if the hex string is "F4483B47" then set the exponent 105 in register $a0 and the signed normalized mantissa, "-1.10010000011101101000111", in the label *mantissa*. The base address of this string should be set in register $a1 before terminating. Notice the negative sign and 1 prepended to the fractional part ".1001...". The sign is negative since the MSB in the hex string is 1. For positive numbers, do not attach the + sign. For example, the hex string "42864000" is a positive number and will have the exponent 6 in register $a0 and the normalized mantissa string "1.00001100100000000000000" with its base address set to $a1.

Part 5: Verify Hand in The Loot Card Game

Loot is a multi-player card game. In this game each player has a collection of merchant ships and pirate ships. Merchant ships carry gold coins. A player can attack merchant ships with their own pirates or defend their own merchant ships. The goal is for each player is to collect as many merchant ships as possible. In this part, we do not need to develop the game. We only need to verify a player's hand, that is, the collection of cards held by a player at a particular instant of time. According to the rules of the game, a player must have exactly 6 cards in their hand. These cards can be merchant ship or pirate ship cards. A merchant ship card has a digit [3-8] associated with it, which indicates the gold coins in a merchant ship. A pirate ship card has a digit [1-4] associated with it to indicate the strength of a pirate ship.

To verify a player's hand, a user will have to provide two arguments to the program. The first argument is the null-terminated string "L" and the second a string encoding the player's hand. The encoding has a special notation. The merchant ships will be indicated the letter 'M' followed by a digit to indicate the value of the card. The pirate ships will be indicated by the letter 'P' followed by a digit to indicate the strength of the pirate ship. For example, a valid second argument will be "M6M4P3P2M4M5". This means that there are four merchant ships with values 6,4,4, and 5 and two pirate ships with strength values 3 and 2. This string may not be null-terminated. If the arguments are invalid then then print the error message in the label *invalid_arg_msg* and terminate.

If the provided arguments are valid, then the program should create an *unsigned binary representation* of the form N1N2, where N1 is the no. of merchant ships and N2 is the no. of pirate ships in the player's hand. The program should print the decimal equivalent of the binary representation and terminate. In the example above, the player has 4 merchant ship cards and 2 pirate ship cards in their hand. This should result in the unsigned binary representation `100010`. Hence, the decimal value `34` should be printed.

Submitting Code to GitHub

You can submit code to your GitHub repository as many times as you want till the deadline. After the deadline, any code you try to submit will be rejected. To submit a file to the remote repository, you first need to add it to the local git repository in your system, that is, directory where you cloned the remote repository initially. Use following commands from your terminal:

`$ cd /path/to/cse220-hw1-<username>` (skip if you are already in this directory)

`$ git add hw1.asm`

To submit your work to the remote GitHub repository, you will need to commit the file (with a message) and push the file to the repository. Use the following commands:

`$ git commit -m "<your-custom-message>"`

```
$ git push
```

Every time you push code to the GitHub remote repository, the test cases in the *tests* folder will run and you will see either a green tick or a red cross in your repository just like you saw with homework0. Green tick indicates all tests passed. Red cross indicates some tests failed. Click on the red cross and open up the report to view which tests failed. Diagnose and fix the failed tests and push to the remote repository again. Repeat till all tests pass or you run out of time!

Running Test Cases Locally

It may be convenient to run the test cases locally before pushing to the remote repository. To run a test locally use the following command:

```
$ java -jar munit.jar tests/<test-file>.class hw1.asm
```

Remember to set java in your classpath. Your test cases may fail if you do not have the right setup. If you do not have the right setup it is most likely because you did not do homework 0 correctly. So, do homework 0 first and then come back here!