github-classroom[bot] GitHub Classroom Autograding Wo...		2 minutes ago 3
.github	GitHub Classroom Autograding Workflow	2 minutes ago
tests	Initial commit	3 minutes ago
.gitignore	Initial commit	3 minutes ago
README.md	Initial commit	3 minutes ago
hw2.asm	Initial commit	3 minutes ago
hw2_test.asm	Initial commit	3 minutes ago
munit.jar	Initial commit	3 minutes ago

README.md

Homework 2

Learning Outcomes

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

- Work with 1D Arrays.
- Write Functions and manage the call stack.
- Implement a basic encryption algorithm in assembly.

Getting Started

To complete this homework assignment, you will need the MARS simulator. Download it from Blackboard. 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-hw2-<username>`, where *username* is your GitHub username.

In this directory, you will find *hw2.asm*. This file has function stubs that you will need to fill up. 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 may not be graded. The directory also has a template test file ending with *_test.asm*. Use the file for preliminary testing. You can change the data section or the text section in these files to test different cases for each part (described later). The *tests* directory contain 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. The files in the *tests* directory should not be modified. If you do, you will receive no credit for the homework.

Note the hw2.asm file doesn't have a .data section. Do not add a .data section.

Don't forget to add you name and IDs at the top of hw2.asm. Follow the exact format, i.e, replace the hints with the correct information. You will be penalized if you do not follow the format.

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.

Always assume that memory and registers will have garbage data. When using memory or registers, it is your responsibility to initialize it correctly before using it. **You should enable the Garbage Data option under Settings in MARS to run your programs with garbage data in memory.**

How to read the test cases

As mentioned previously, the *tests* folder contains the test file *Hw2Test.java*. Each test is a Java function with a name prefixed with *verify_*. In each of these functions, you will find an assert statement that needs to be true for the test to pass. These asserts compare the expected result with the actual result returned by the function under test. The tests assume that the expected results will be in certain registers or labels as explained 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 *_test.asm* file and run the *_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

Block ciphers are a common encryption technique to encrypt secret messages and preserve their confidentiality. It works by breaking a messaging into blocks of size N. Each block is encrypted with a randomly generated key. The blocks are then combined to create an encrypted secret message. This encrypted message is called a block cipher. When the block cipher is shared with the intended person, they should be able to decrypt the secret message provided they have access to the key that was used to encrypt the message. To protect the confidentiality of the message, the key should only be shared with the intended party. Further, an encryption function `f` should be chosen such that `f(m,k) = f(c,k)`, where `m` is a secret message, `k` is a key, and `c` is a cipher. This means that the encryption function when applied to a message and a key produces a cipher and when applied to the same cipher and the same key produces the original message.

There are many block cipher algorithms. We will implement one of the simpler ones in this homework assignment. As per the algorithm we will implement, to encrypt a secret message will first divide it into blocks of size 4. Each block will be XOR-ed (chosen encryption function) with a randomly generated key of the same size. Each cipher block will be reversed and combined with other blocks to create a cipher message. It is possible, that the secret message is not a multiple of 4. When this happens, random characters should be appended to the end of the secret message to make the message length a multiple of 4. This padded secret message should then be encrypted using our block cipher algorithm.

We will implement a block cipher as described above by implementing functions as defined below. You can define additional functions if necessary. **Each function must follow the register conventions discussed in class**. Violating register conventions will result in loss of credit. A handy reference is available under *Course Documents -> Readings* in Blackboard. Read the document before starting the homework if you are unsure about the conventions.

Part 1: Substring

Implement a function *substr(str, lower, upper)* that takes 3 arguments in registers *$a0, $a1, $a2* and returns an integer in the register *$v0*. The arguments *str* is the base address of a null-terminated string and *lower* and *upper* are integer offsets. The function should extract the characters between offsets *lower* and *upper-1* (inclusive) and **replace** the string in *str* with the extracted characters. The new string should be null-terminated. The function returns 0 in *$v0* if the replacing is successful. It returns -1 in *$v0* if *lower* and *uppper* are out of bounds or are negative and the original string must not change.

Part 2: Encrypt A Block

Implement a function *encrypt_block(block, key)* that takes 2 arguments *block* in register *$a0* and *key* in register *$a1* and returns a signed integer in register *$v0*. the arguments *block* and *key* are strings of size 4. The return value in *$v0* is an integer obtained from a bitwise XOR operation on *block* and *key*. For example, if *block* is the string "unix" and *key* is the string "unit" then the integer in *$v0* should be 12 as the bitwise XOR of the strings "unix" and "unit" will result in the decimal number 12.

Part 3: Add A Block

Implement a function *add_block(dest, bindex, code)* that takes 3 arguments *dest*, *bindex*, and *code* in registers *$a0*, *$a1*, and *$a2* respectively and returns nothing. The argument *dest* is the base address of a string, *bindex* and *code* are integers. The function should reverse *code*, that is, start with the least significant byte and insert it into the appropriate block of *dest*. Assume that the string length at *dest* is a multiple of 4. Hence, the first block starts at offset 0, the second at offset 4 and so on. You can assume that *0 <= bindex <= (length(dest) - 4)/4*, where *length(dest)* is the length of string at *dest*. So, for example, if integer in *code* is `0000010100000010` and *bindex = 1* then block starting at address *dest+4* should have the string "2500". Notice how the LS byte is the first index of the string.

Part 4: Generate Key

Implement a function *gen_key(key, bindex)* that takes two arguments in registers *$a0* and *$a1* and returns nothing. The argument *key* is the base address of a string that will hold the cipher key used to encrypt and decrypt. The argument *bindex* is an integer that indicates the starting address of the block in *key*. You can assume that *0 <= bindex <= (length(key) - 4)/4*, where *length(key)* is the length of the string in *key*. Recall in a block cipher algorithm each block has its corresponding key. The function should generate a random collection of 4 characters and store them in the string *key* at an appropriate position as indicated by *bindex*. For example, if *bindex = 1* then the randomly generated characters should be stored in the string starting at address *key+4*. You can assume that each randomly generated character is an unsigned byte.

The following syscall in MARS is used to generate a random number within a particular range:

```
li $a1, N
li $v0, 42
syscall
```

Syscall 42 will generate a random number between 0 and N-1. To know more read about the syscall in the *Help* section in MARS.

Part 5: Encryption

Implement a function *encrypt(plaintext, key, cipher, nchars)* that takes 4 arguments in registers *$a0, $a1, $a2, $a3* and returns nothing. The argument *plaintext* is the base address of a string containing a message that will be encrypted, *key* is the base address of a string that will hold the cipher key used to encrypt, *cipher* is the base address of a string that will hold the encrypted cipher message, and *nchars* is the length of *plaintext*. The function should divide *plaintext* into blocks of size 4. It should also generate a key for each block and store it in appropriate positions in the *key* string. Finally, the function should encrypt each block with the corresponding key and store the cipher text in *cipher*. Note *nchars* may not be a multiple 4. If it is not a multiple of 4, then random characters should be added to the end of *plaintext* to make the length of *plaintext* a multiple of 4. You can assume that *cipher* will have the same length as that of *plaintext* (padded it necessary). You should call the functions defined in the previous parts to implement *encrypt*.

Part 6: Decrypt A Block

Implement a function *decrypt_block(cipherblock, keyblock)* that takes 2 arguments *cipherblock* in register *$a0* and *keyblock* in register *$a1* and returns a signed integer in register *$v0*. the arguments *cipherblock* and *keyblock* are strings of size 4. The return value in *$v0* is an integer obtained from a bitwise XOR operation on the *cipherblock* and the reversed *keyblock*. For example, if *cipherblock* is the string "us1u" and *keyblock* is the string "u5su" then the integer in *$v0* should be 1024 as the bitwise XOR of the strings "us1u" and "us5u" will result in the decimal number 1024.

Part 7: Decrypt

Implement a function *decrypt(ciphertext, key, nchars, plaintext)* that takes 4 arguments in registers *$a0, $a1, $a2, $a3* and returns nothing. The argument *plaintext* is the base address of a string containing the original message that is to be decrypted, *key* is the base address of a string that holds the cipher key that was used to encrypt, *ciphertext* is the base address of a string holds the encrypted cipher message, and *nchars* is the length of *ciphertext*. The function should divide *ciphertext* into blocks of size 4. Also, the function should decrypt each block with the corresponding key and store the original text in *plaintext*. Assume *nchars* is a multiple 4. You should call the functions defined in the previous parts to implement *decrypt*.

Submitting Code to GitHub

You can submit code to your GitHub repository as many times as you want till the deadline. Code submitted after the deadline will be subject to late penalties as defined in the syllabus. 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-hw2-<username>` (skip if you are already in this directory)

`$ git add hw2.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!

After you submit your code on GitHub. Enter your GitHub username in the Blackboard homework assignment and click on Submit. This will help us find your submission on GitHub.

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/Hw2Test.class hw2.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!