Take-Home Quiz #5

CSEE W3827 - Fundamentals of Computer Systems Fall 2020

1. Due Date

You should upload your code in a file named uni.s to Courseworks before 11:59PM EST on 11/19. Submit your work well ahead to allow time to manage any unexpected outages or events. Late submissions will not be accepted, but repeat submissions and Courseworks's renaming of your file to uni-1.s, uni-2.s, etc. are fine.

2. Collaboration and Academic Honesty

This take-home assignment is replacing an in-class quiz. Like a quiz, it is open-book and open-note. You are not to discuss it with anyone other than current course staff or to seek solutions on the web. If you have any questions about whether a resource is acceptable, ask course staff privately on piazza. Students who do not abide by this policy will be subject to academic and disciplinary penalties.

As on a quiz, staff will answer questions about the problems, but will not help you solve them. You may pose clarifying questions to the staff via private questions on piazza. Please make those posts private to all instructors.

3. Overview

In this assignment, you will build a simple code breaker. The bad guys are sending encrypted messages, and your job is to crack the code and reveal the message. Fortunately, the bad guys don't know very sophisticated coding methods, and have chosen to code their message as follows:

A message stored in memory is *N* words long. Each 32-bit word of the message contains 4 bytes or characters. The **key**, *K*, used to encrypt the message, is an *N* word quantity that repeats the same byte value over and over again. The **key byte** refers to the byte that is repeated over and over again 4N times, and the **key word** refers to the 4-byte quantity that repeats the key byte four times. For example, for a key byte 0x01, the key word would be 0x01010101. 0x01020102 would not be a key word, because the byte values vary.

To produce the encrypted message, the bad guys subtract the 32*N*-bit key from the original 32*N*-bit message string. This subtraction treats both values as unsigned 32*N*-bit quantities with the highest order bits appearing at the lowest memory addresses. The encrypted message is followed by an unencrypted 0-word (i.e., 0x00000000) that indicates the end of the message. The bad guys always choose a key such that the encrypted text itself contains no 0-word.

To decrypt the message, you must try different values for the byte that forms the key. For each value, perform an unsigned add with each word in the key, excluding the 0-word.

How do you know when you've correctly decrypted the message?

In the unencrypted message, the bad guys use only capital letters A (65 or 0x41[1] in hex) through Z (90 or 0x5A in hex) and the character @ (byte value 64 or 0x40 in hex) in place of a space. Thus, an unencrypted message contains only bytes with values between 64 and 90. A valid word consists of four valid bytes, and a valid message is a sequence of valid words. Messages decrypted with the wrong key are very unlikely to be valid, so a decryption key that produces a valid message is a strong candidate for the correct key!

Your decrypter will:

- Iterate over all possible keys, trying each one.
- Try a key by performing the multi-word addition, and check if the decrypted message is valid.
- If the message is valid, print the message along with the successful key byte that was used to decrypt the message. Note that there may be more than 1 valid decryption, so we will print them all. However, there should not be many, and the correct message should be easy to determine when you read it.

Note that MIPS does not have a routine to automatically perform this 32*N*-bit addition, especially with the length of the message not initially known, and not when the value is placed in memory from high order to low order. Hence, you will need to build a procedure that performs the addition by starting at the lowest order word (at the end of the encrypted string), adds that word to the key word, and returns both the sum and whether there was a carry out. Next, the second lowest order word must be added to the word-size key and any carry out from the previous addition, and so on, repeating the process until you reach the front of the encrypted string. You must perform this operation as a recursive procedure.

4. Coding Details

The assignment is broken down into three functions that work together to test a key. We recommend completing them in the order they are listed here.

wordDecrypt (2 pts)

Our first procedure, called wordDecrypt, takes 3 arguments: $a0, $a1, and $a2, which are an encrypted word of the message, a key word, and a word representing the incoming carry value of either 0 or 1, respectively. wordDecrypt adds the three inputs together as unsigned integers. It returns a pair of values, $v0 and $v1, which are the sum and the carry out value, again either 0 or 1, respectively.

MIPS has a somewhat confusing mechanism for obtaining the overflow result from an addition. Since we are summing unsigned quantities, we can detect overflow when the sum is smaller than key. So, if $a0 + $a1 + $a2 < $a1, then carry out should be 1, otherwise it should be 0.

isCandidate (3 pts)

Our next procedure, called isCandidate, takes in a single argument, $a0, and returns a single result $v0 which equals 1 if $a0 contains all valid bytes, and is 0 otherwise. This procedure can be written as follows:

(a) Fill a register with a mask with a value of 255, and AND the mask with the input word. The result of the AND is the lowest-order byte of the word.

[1] Note that hexadecimal numbers are prefaced with a '0x' - this is also done within SPIM.

(b) Check to see if this value is between 64 and 90 inclusive. If not, return 0. If so, check the next byte.

(c) Shift the decrypted word 8 bits, and repeat (a), (b), and (c) to check the next byte in the word. After checking all 4 bytes, if they are all valid, return 1.

addAndVerify (3 pts)

Lastly, write a recursive procedure, addAndVerify that performs the decryption and checks if the decryption is valid. addAndVerify takes an address in $a0 for the encrypted string, an address in $a1 for the decrypted string, and a key word *K* in $a2. addAndVerify (eventually, after performing all recursions) returns in $v0 the value of 1 if the decrypted message is a valid, and 0 otherwise. addAndVerify will use $v1 during the recursion to store the carry bit (when applicable, see below). Since it's a recursive procedure, it should operate by reading the word at the address indicated by $a0:

Base case: if the value read in equals 0, this indicates the end of the encrypted message. Write a 0-word to the destination address, so that the decrypted string will be null-terminated as well. Set $v0 to 1, $v1 to 0 (no carry) and return.

Recursive case: the value read in is non-0. We need to determine the carry from the lower-order addition first. Recursively call addAndVerify with the addresses $a0 and $a1 incremented (by 4!), and with the same key in $a2. When the recursive call returns, you will have in $v0 whether the suffix of the message is valid and in $v1 the result of the carry (if the suffix was valid). If the suffix is valid ($v0 = 1), take the carry result, the non-0 value read in at the beginning of the function invocation, and the key, and pass them to wordDecrypt. Pass the result of the addition (in $v0) to isCandidate. When isCandidate returns, the values in $v0 and $v1 are ready to be returned from this function invocation. Before returning, make sure to restore all preserved registers, popping everything off the stack that was pushed during this function invocation.

5. Additional Tips

Comment your code, particularly when the behavior is not self-evident. You will have an easier time debugging if your code is well-commented.

To troubleshoot, try implementing and checking wordDecrypt and isCandidate first.

6. Testing

As in PS8 and PS9, we have supplied scaffolding.s.

Do not move or delete the separator lines, and put all of your code for each problem after the function name label and before the next separator.

As usual, the body of main has a number of test function calls for you to comment/uncomment according to the tests you wish to run. After these tests, main has a loop that will try all keys and crack an encrypted message.

Since the addAndVerify depends on working implementations of wordDecrypt and isCandidate, we will test addAndVerify only if the first two operate correctly.

The addAndVerify function is the most challenging, so keep this in mind when budgeting your time. We cannot emphasize enough that you should start early.

8. When you are done and your code works

Congratulations! You have now officially doubled your Nerd-Cred. You can now brag to your family and friends that you can code in Assembly Language. Should you brag? We leave that up to you...