HW #5

CSEE W3827 - Fundamentals of Computer Systems Spring 2022

Prof. Rubenstein
Due 3/25/22, 5pm

Topic: MIPS programming

1 Overview

In this homework, 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 dont know very sophisticated coding methods, and have chosen to code their message as follows:

A message stored in memory is N words long. Each word consists of 4 bytes or characters, totaling 32 bits. The **key**, K, used to encrypt the message, is an N word quantity that repeats the same byte value over and over again. Note that in hex, a pair of literals represents a byte, so $00000000\cdots$ could be a key with byte 00, or $01010101\cdots$ with byte 01, or $02020202\cdots$ with byte 02, and so on. However, 01020102 could not be a key because not all bytes share the same value. The **key byte** refers to the byte that is repeated over and over again in the $4N$-bit key, and the **key word** refers to the 4-byte quantity that repeats the key byte four times.

To produce the encrypted message, the bad guys subtract the $32N$-bit key from the original $32N$-bit message string, treating both as unsigned $32N$-bit quantities with the highest order bits appearing at the lowest memory addresses. The encrypted message is followed by an unencrypted 0-word (all bits in the word are 0), so that the encrypted message end can be determined. The bad guys always choose a key so 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 (except the 0-word).

How do you know when youve correctly decrypted the message?

In the unencrypted message, the bad guys only use 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 that lie between 64 and 90. When a character lies within this ASCII range of 64 through 90, we say the character (byte) is **valid**. A word formed of 4 valid bytes is called a valid word, and a message formed from valid words is called a valid message. Messages decrypted with the wrong key are very unlikely to be valid, so a decryption key that produces a valid word is a strong candidate for the correct key!

You are tasked with building a decrypter which must do the following:

- Iterate over the 255 possible keys (skip byte 00, as this doesnt change the value).
- For each key value, perform the multi-word addition, and check if the decrypted message is valid.
- If the checker determines the entire message is valid, print the valid message to the screen, as well as the corresponding byte that was used to decrypt the message. Note that there may be more than 1 valid decryption, so 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 $32N$-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 word-size key, and returns both the result and whether there was an overflow (i.e., a carry). Next, the second lowest order word must be added to the word-size key, as well as to the carry (if there was overflow) from the previous word. This process repeats until we reach the front of the encrypted string. You will perform this operation as a recursive procedure.

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

2 Coding Details

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

   MIPS unfortunately has a somewhat confusing mechanism for obtaining the overflow result from an addition. Since we are summing unsigned quantities, the sum results in an overflow if it is less than the key. For our purposes, if $a0 + $a1 + $a2 $<$ $a1, then there was an overflow and the returned carry should equal 1. Otherwise the returned carry should equal 0.

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

   (a) Set a register to be a *mask*, with a value of 255 and AND the mask with the input word ($a0). The resulting value equals the lowest-order byte of the word.

   (b) Check to see that this value is between 64 and 90 (inclusive). If not, return 0. If so, check the next byte. After checking all 4 bytes, if they are all valid, return 1.

   (c) Checking the "next" byte is done by shifting the decrypted word 8 bits.

3. Next, write a recursive procedure, **AddAndVerify** that performs the decryption, and checks if the decryption is valid. AddAndVerify takes as input an address in $a0 for the encrypted string, an address in $a1 for the decrypted string, and a key K in $a2. AddAndVerify (eventually, after performing all recursions) returns in $v0 the value of 1 if the decrypted message is a valid. Otherwise, $v0 is set to 0. AddAndVerify will also use $v1 during the recursive process to store the carry bit. Since its a recursive procedure, it should operate as follows:

   (a) First, dont forget to push necessary values onto the stack (e.g., return address and saved registers, and maybe the argument registers since their values might change) before modifying the registers.

   (b) Read the word stored at the address indicated by $a0.

   (c) **Base case**: if the value read equals 0, this indicates the end of the encrypted message. Copy the 0 value into the destination address (so that the decrypted string will be null-terminated as well, needed for printing by SPIM), set $v0 to 1, $v1 to 0 (no carry) and return.

   (d) **Non-base case**: the value read 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 $a2. When the recursive call returns, you will have in $v0 whether the suffix of the message is valid and in $v1 (if the suffix was valid) the result of the carry. If the suffix is valid ($v0 = 1), take the carry result, the non-0 value read at the beginning of the procedure, and the key and pass all three to WordDecrypt. Pass the result of the addition (in $v0) to IsCandidate. When IsCandidate returns, the values in $v0 and $v1 are ready for the outer-nesting call of AddAndVerify. Before returning, remember to pop the appropriate items from the stack you pushed on so that the return goes to the right address, and so that the outer-nest has values in any saved registers appearing unmodified.

4. Finally, create a procedure **main** which starts with the key word 01010101 (recall 00000000 is not a key), and calls AddAndVerify with the key word and base addresses of the encrypted string and destination memory. If AddAndVerify returns a 1 in $v0, then the key produced a valid message, so print the message and the key word value. Try this for all key words, stopping after $FEFEFEFE$ (it wont be $FFFFFFFF$ and we dont want to use $FFFFFFFF$ because of how we are computing overflow).

   (a) To iterate over the keys: one possibility is to form the key word from the key byte with a series of shifts and adds. However, there is an easier way which computes the next key word directly from the last. Think about how this might be possible.

3 Using SPIM

You can find SPIM software and documentation by Googling, but a tutorial for the course will be made available shortly. Read the tutorial also. Some things you need to do (e.g., printing to screen) are easy to do in SPIM, but were not covered in class.

4 Additional Suggestions

- Comment your code. This way if there are mistakes, the grader can more easily understand what it was you were trying to do.
- A file, `template.s` will be made available online that you can use to start your code. It has the encrypted message built in with label `EncryptedPhrase`, and 100 words of memory reserved to store decryption attempts with label `DecryptionSpace`. The decryption space should be reused for each key value tried.
- Remember to have your code print the decoded message and the key that was associated with the message.
- To troubleshoot, try checking your inner procedures first. MIPS code is provided to test your procedures. `TestWordDecrypt.s` can be used to test your `WordDecrypt` procedure. Once thats right, you can use `TestIsCandidate.s` to test your `IsCandidate` procedure with your correct `WordDecrypt` procedure. Then `testAddAndVerify.s` can make sure `AddAndVerify` is working properly, along with your already correct `IsCandidate` and `WordDecrypt`.
- Make use of SPIMs abilities to step through your code one instruction at a time and to stop execution via breakpoints. Also, make use of the fact that you can view register values as the code is running to verify things seem to be working properly.
- Do not copy other peoples code! There are places where everyones code should look somewhat the same, but also places where there should be substantial differences.

5 Grading Info

Please coordinate with your P-Credit TA how to best submit your work.

6 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...