Lab 5

x86 Addressing Modes Lab

Due: Week of March 8, before the start of your lab section

This is an individual-effort project. You may discuss concepts and syntax with other students, but you may discuss solutions only with the professor and the TAs. Sharing code with or copying code from another student or the internet is prohibited.

In this assignment, you will practice creating the source and destination for x86 assembly language instructions. **You should be able to complete this lab assignment during lab time.**

The instructions are written assuming you will edit and run the code on your account on the *csce.unl.edu* Linux server. If you wish, you may edit the code in a different environment; however, you probably will not be able to assemble and link an executable on MacOS nor on Windows.

Scenario

You've settled into a comfortable routine at the Pleistocene Petting Zoo. While your job isn't quite as exciting as that of the sabre-toothed tigers' dentist, it still has something new and interesting almsot every day.

Archie announces that he heard that hand-crafted assmebly code can be faster than high-level language code. You try to explain that while this may have been true decades ago, modern optimizing compilers generate code faster than what a typical programmer can achieve with assembly code. Archie doesn't believe you and instists that you write the zoo's new cipher program in x86 assembly code.

In this assignment, you won't have to write the entire functions in assembly language; most of that has already been done for you. Instead, there are ten lines of assembly code that you will need to complete, demonstrating an understanding of x86 operands and memory addressing modes. The assembly code has a few optimizations, so some of it may not be immediately recognizable.

1 Caesar Cipher Function

The first of the three functions is the Caesar Cipher itself. Here is the equivalent C code:

```
 1 char *caesar_cipher(char *destination, const char *text, int key) {
 2     char *target = destination;
 3     int reduced_character;
 4     while (*text) {
 5         if ('A' <= *text && *text <= 'Z') {
 6             reduced_character = *text - 'A';
 7             reduced_character = (reduced_character + key + 26) % 26;
 8             *target = (char)(reduced_character + 'A');
 9         } else {
10             *target = *text;
11         }
12         target++;
13         text++;
14     }
15     *target = '\0';
16     return destination;
17 }
```

1.1 Subtract Using an Immediate Operand

In this task, you will insert the assembly instruction that corresponds to line 6 and part of line 7 in the function's C code. Specifically, we are combining the `-'A'` from line 6 with the `key + 26` to create a temporary variable equal to `key + 26 - 'A'`. The `key` value is initially in register `%edx`; since we will not need this value again, the assembly instruction can safely overwrite `%edx`.

Find the line in *addressinglab.s* that says
`##### PLACE INSTRUCTION FOR TASK 1.1 ON NEXT LINE #####`
On the next line, insert a `subl` instruction that subtracts the immediate value `39` from the contents of register `%edx` and places the difference in register `%edx`.

Do not delete the `##### PLACE INSTRUCTION...` comment, and do not delete or modify any other instructions.

1.2 Add the Contents of Two Registers

You can now add the value computed in Task 1.1 to the copy of the character previously pointed to by the `text` pointer, as part of line 7. The character is in the `%al` register, and the value from Task 1.1 is in register `%edx`. In this task, you will first sign-extend the one-byte character so that it can be added to the 32-byte value, and then you will perform the addition.

Find the line in *addressinglab.s* that says
`##### PLACE INSTRUCTIONS FOR TASK 1.2 ON NEXT TWO LINES #####`
On the next line, insert a `movsbl` instruction. This instruction is like a `mov` instruction except

that it *s*ign-extends a *b*yte so that it becomes a *l*ong word (using the IA32 nomenclature that a "long word" occupies 4 bytes). The source for this instruction is the `%al` register, and the destination is `%eax`.

On the next line after that, insert an `addl` instruction to add the contents of `%eax` and `%edx`, leaving the sum in `%eax`.

Do not delete the `##### PLACE INSTRUCTIONS...` comment, and do not delete or modify any other instructions.

1.3 Store a Value in Memory

The next assembly instruction you will introduce combines part of line 8 and line 10. Register `%al` will hold either `(char)(reduced_character+'A')` from line 28 or it will hold the character previously pointed to by `text`. This character needs to be placed in memory at the address pointed to by the `target` pointer. Register `%r8` holds that address.

Find the line in *addressinglab.s* that says

```
##### PLACE INSTRUCTION FOR TASK 1.3 ON NEXT LINE #####
```

On the next line, insert a `movb` instruction to copy the character. The source for the instruction is the `%al` register. The destination is the memory location pointed to by the `%r8` register; that is, you will need to dereference the address in `%r8`.

Do not delete the `##### PLACE INSTRUCTION...` comment, and do not delete or modify any other instructions.

1.4 Load a Value from Memory

Now you will load the next character to be enciphered. This character is pointed to by the `text` pointer, and this pointer is in register `%rsi`. You will place the character in the 32-bit register `%eax`.

Find the line in *addressinglab.s* that says

```
##### PLACE INSTRUCTION FOR TASK 1.4 ON NEXT LINE #####
```

On the next line, insert a `movzbl` instruction to copy the character. `movzbl` is like `movsbl`, except that it *z*ero-extends the byte. The source for the instruction is the memory location pointed to by `%rsi`, and the destination is register `%eax`.

Do not delete the `##### PLACE INSTRUCTION...` comment, and do not delete or modify any other instructions.

Check Your Progress

You have now completed the **`caesar_cipher`** function. We strongly recommend that you confirm that this function works before moving on to the remaining tasks. Generate the executable with the command:

```
gcc -std=c99 -Wall -o addressinglab addressingdriver.c addressinglab.s
```

Run the program with a few manual tests. For example, "ZEBRA" with the key `1` enciphers to "AFCSB". (Note that the **`caesar_cipher`** function will only encipher uppercase letters.) If the function does not perform correctly go back and double-check each of the six instructions you placed in it.

2 Sentence to Uppercase Function

The second of the three functions is converts lowercase letters in a sentence to uppercase letters. Here is the equivalent C code:

```
1 char *sentence_to_uppercase(char *destination, const char *sentence) {
2     unsigned long length = strlen(sentence);
3     for (int i = 0; i < length; i++) {
4         destination[i] = (char)toupper(sentence[i]);
5     }
6     return destination;
7 }
```

Both tasks for this function are for line 4.

2.1 Index Arrays as the Source

Your next task is to convert lowercase letters to uppercase letters. As an efficiency gain, the repeated calls to **`toupper`** on line 4 have been replaced with a lookup table. The base address for the `sentence` array is in `%r15`; the base address for the lookup table is in `%rcx`, and the loop index `i` is in `%rbx`.

Find the line in *addressinglab.s* that says
`##### PLACE INSTRUCTIONS FOR TASK 2.1 ON NEXT TWO LINES #####`
On the next line, insert a `movsbq` instruction. `movsbq` is like `movsbl` except that it sign-extends the byte so that it becomes a *q*uad word (using the Intel64 nomenclature that a "quad word" occupies 8 bytes). The source is a location in the **`sentence`** array: use the indexed addressing mode. The base address is `%r15`; the index is `%rbx`, and each array element is 1 byte. The destination for this instruction is `%rdx`.

On the next line after that you will use the **`toupper`** lookup table. To preserve the illusion that the program called the **`toupper`** function, the lookup table stores integers (**`toupper`**'s specification states that it returns an **`int`**). Insert a `movzbl` instruction. The source is a location in the lookup table array: use the indexed addressing mode. The base address is `rcx`; the index is the character from the previous instruction in `%rdx`, and each array element is 4 bytes in size. The destination for this instruction is `%ecx`.

Do not delete the `##### PLACE INSTRUCTIONS...` comment, and do not delete or modify any other instructions.

2.2 Index an Array as the Destination

The other part of line 4 is casting the integer from Task 2.1 to a **`char`** and storing it in the `destination` array. The array's base address is in `%r14`, and as before the loop index is in

`%rbx`.

Find the line in *addressinglab.s* that says

`##### PLACE INSTRUCTION FOR TASK 2.2 ON NEXT LINE #####`

On the next line, use a `movb` instruction to move the lower 8 bits of the integer into the `destination` array. The instructions's source is `%cl`, and the destination is a location in the array. The array's base address is `%r14`; the index is `%rbx`, and each array element is 1 byte.

Do not delete the `##### PLACE INSTRUCTION...` comment, and do not delete or modify any other instructions.

Check Your Progress

You have now completed the **`sentence_to_uppercase`** function. We strongly recommend that you confirm that this function works before moving on to the remaining tasks. In *addressingdriver.c*'s **`main`** function, un-comment this line:

```
capitalized_plaintext = sentence_to_uppercase(capitalized_plaintext, plai
```

Then generate the executable with the command:

```
gcc -std=c99 -Wall -o addressinglab addressingdriver.c addressinglab.s
```

Run the program with a few manual tests. For example, "ZebrA" with the key `1` enciphers to "AFCSB", and the deciphered text will be "ZEBRA". If the function does not perform correctly go back and double-check each of the three instructions you placed in it.

3 Cipher Validation Function

The third function validates that the plaintext and ciphertext are a valid pair by confirming that they are both the length specified by the package's `sentence_length` field and that the inverse of the package's `key` field will decipher the ciphertext back to the plaintext. (See *addressinglab.h* for the field details of the `cipher_package` structure.) Here is the equivalent C code:

```
bool validate_cipher(struct cipher_package *package) {
    bool is_valid = (strlen(package->plaintext) ==
       package->sentence_length);
    is_valid = is_valid && (strlen(package->ciphertext) ==
       package->sentence_length);
    char *deciphered_text = malloc(MAXIMUM_INPUT_LENGTH);
    deciphered_text = caesar_cipher(deciphered_text,
       package->ciphertext, -(package->key));
    is_valid = is_valid && !strncmp(package->plaintext,
       deciphered_text, package->sentence_length);
    return is_valid;
}
```

3.1 Access Fields in a Struct

Your final task is to position arguments into the correct registers for the call to **`caesar_cipher`** in line 5. The `deciphered_text` pointer has already been placed in the correct register; you do not need to take care of that. Each of the other two arguments is a field in the `cipher_package` structure. The base address for `package` is in `%rbx`.

Find the line in *addressinglab.s* that says
`##### PLACE INSTRUCTIONS FOR TASK 3.1 ON NEXT TWO LINES #####`
On the next line, use a `movq` instruction to copy `package->ciphertext` into `%rsi`. Use the displacmenet addressing mode in the source operand. The base address is `%rbx`, and the `ciphertext` field is positioned 8 bytes after the base address.

On the next line after that, you need to place `-(package->key)` into `%edx`. Register `%edx` contains the value `0`, so we can generate `-(package->key)` by subtracting `package->key` from the content of `%edx` and placing the result in `%edx`. Insert a `subl` instruction to accomplish this. Use displacement mode addressing to access `package->key`, which is 20_{10} bytes after the base address that is in `%rbx`.

Do not delete the `##### PLACE INSTRUCTIONS...` comment, and do not delete or modify any other instructions.

Check Your Program

You have now completed the **`validate_cipher`** function, and with it, the Pleistocene Petting Zoo's cipher program. In *addressingdriver.c*'s **`main`** function, un-comment all of the lines. Generate the executable with the command:

```
gcc -std=c99 -Wall -o addressinglab addressingdriver.c addressinglab.s
```

Run the program with a few manual tests. All cipher packages generated by the **`main`** function are valid, and so any input you use should produce the report "Cipher package is valid." If **`validate_cipher`** does not perform correctly go back and double-check each of the two instructions you placed in it.

Turn-in and Grading

When you have completed this assignment, upload *addressinglab.s* to Canvas.

This assignment is worth 10 points.

_____ **+1** Task 1.1: Subtract Using an Immediate Operand

_____ **+2** Task 1.2: Add the Contents of Two Registers

_____ **+1** Task 1.3: Store a Value in Memory

_____ **+1** Task 1.4: Load a Value from Memory

_____ **+2** Task 2.1: Index Arrays as the Source

_____ **+1** Task 2.2: Index an Array as the Destination

_____ **+2** Task 3.1: Access Fields in a Struct