# LAB 6: VIGENERE CIPHER

**MINIMUM SUBMISSION REQUIREMENTS:**

- Lab6.asm in the lab6 folder
- README.txt in the lab6 folder
- Commit and Push your repo
- A Google form submitted with the Commit ID taken from your GITLAB web interface verifying that the above files are correctly named in their correct folders
- All of the above must be completed by the lab due date

**LAB OBJECTIVE:**

In this lab, you will learn to write subroutines (functions), and to write code that complies with a standardized interface. You will also gain additional experience in accessing arrays.

As engineering projects become increasingly complex, it becomes necessary to develop some parts of the project independently. These parts must work together, even if they are not written by the same people or at the same time. It is crucial to carefully adhere to the specifications and standards that other pieces of code rely on. This is a key idea of software engineering, and is called *encapsulation*. That is, you can call code with no idea how it is written underneath as long as it adheres to the standard. In this lab, you will write code that interfaces with pieces of software that we have written (some of which you will not see).

**LAB PREPARATION:**

Read this document in its entirety, carefully.

Read *Introduction to MIPS Assembly Language Programming* by Charles Kann, Chapter 5. Note that Kann uses the word "subprograms" to refer to what we call "subroutines" or "functions."

Read the description of Vigenere ciphers on Wikipedia, or find some other suitable source of information.

**LAB SPECIFICATIONS:**

In this lab, you will write a set of MIPS32 subroutines and some test code for those subroutines. These subroutines will perform various encryption and decryption algorithms related to the Vigenere Cipher.

You will write the following four subroutines (though you are encouraged to write additional subroutines if you find it helpful):

 

```
# Subroutine EncryptChar
#     Encrypts a single character using a single key character.
# input:          $a0 = ASCII character to encrypt
#                 $a1 = key ASCII character
# output:         $v0 = Vigenere-encrypted ASCII character
# Side effects:   None
# Notes:          Plain and cipher will be in alphabet A-Z or a-z
#                 key will be in A-Z

# Subroutine DecryptChar
#     Decrypts a single character using a single key character.
# input:          $a0 = ASCII character to decrypt
#                 $a1 = key ASCII character
# output:         $v0 = Vigenere-decrypted ASCII character
# Side effects:  None
# Notes:          Plain and cipher will be in alphabet A-Z or a-z
#                 key will be in A-Z

# Subroutine EncryptString
#     Encrypts a null-terminated string of length 30 or less,
#     using a keystring.
# input:          $a0 = Address of plaintext string
#                 $a1 = Address of key string
#                 $a2 = Address to store ciphertext string
# output:         None
# Side effects:   String at $a2 will be changed to the
#                       Vigenere-encrypted ciphertext.
#                 $a0, $a1, and $a2 may be altered

# Subroutine DecryptString
#     Decrypts a null-terminated string of length 30 or less,
#     using a keystring.
# input:          $a0 = Address of ciphertext string
#                 $a1 = Address of key string
#                 $a2 = Address to store plaintext string
# output:         None
# Side effects:   String at $a2 will be changed to the
#                       Vigenere-decrypted plaintext
#                 $a0, $a1, and $a2 may be altered
```

The Vignere cipher has the following properties:

- It operates over the alphabet of ASCII letters: [ABCDE…..XYZ] and [abcd….xyz]
- When encrypting/decrypting strings, it should ignore any other ASCII characters in the plaintext (spaces, punctuation, unprintable characters, etc.). The key should not advance over these characters.
- The keystring will only contain uppercase characters in the alphabet, [A-Z].
- Encryption and decryption are inverse operations. Any message encrypted with key can be decrypted with the same key, returning to the original plaintext.

Your code should follow these additional specifications:

- Your subroutines should have no side effects besides those listed in the spec. In particular:
  - They should not perform ANY syscalls.
  - External memory should NOT be altered unless described in the spec.
  - All $s registers should have the same contents at the end of your subroutine as they held at the beginning. (Your code may reserve memory for its own use, if required)
- Your Lab6.asm file must be usable by another MIPS program that uses the directive ".include Lab6.asm".
  - Do NOT use the label "main:" in your file
  - Return properly from every subroutine
  - Every label in Lab6.asm should start with double underscores (e.g. "__Endloop:"). "EncryptString", "DecryptString", "EncryptChar", and "DecryptChar" are the only exceptions.
- Your "EncryptString" and "DecryptString" subroutines should call "EncryptChar" and "DecryptChar", respectively.
- Your "EncryptString" and "DecryptString" should not encrypt more than 30 printable characters. Target strings (the strings that store results) should be null-terminated.

You should not run Lab6.asm. Instead, we provide you with a script, test_Lab6.asm. This script should be placed in the same folder as Lab6.asm

Lab6.asm should contain *NO* syscalls. Instead, the calling code will manage the system input and output.

As always, your code should be clean and readable. Refer to Lab4 to see the standard of readability we expect. In addition, we expect that your functions will have the headers containing their specifications (you may copy-and-paste from this document if you like).

We will test your code using a much more thorough battery of tests than contained in test_Lab6.asm. You should modify test_Lab6.asm to run your own tests, or generate your own testing code. You are encouraged to commit your own test code, but we will not check it as part of the grading process.

Here are some examples of the results of encryption and decryption that we expect:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Plaintext: | N | Z | A | f | NZA | DEAD | FOOBAR | UPPERlower | Hello World! | Long |
| Key: | B | B | A | C | BBA | BEEF | KEY | B | GOODBYE | SHORT |
| CipherText: | O | A | A | h | OAA | EIEI | PSMLEP | VQQFSmpxfs | Nszop Usxzr! | Dvbx |

**LAB STRATEGY:**

As usual, an incremental development strategy is best for complicated projects. Here is a good strategy:

1. Start by figuring out how to use the test code. Generate some additional tests of your own.
2. Write pseudocode for each of these functions.
3. Write the EncryptChar function to handle very simple test cases ("A" and "B"). Test it very carefully. Then add code to handle a harder test case ("a" and "B"). Test it carefully. Continue increasing the complexity in this way. That is, develop a little, test it, develop a little more, test, etc.
4. Once EncryptChar works to your satisfaction, write EncryptString to work on simple cases ("ABC" and "BBB"). Test it very carefully. Then add code to handle a harder test case ("ABC" and "B"). Test it carefully. Continue increasing the complexity in this way.
5. Once you are satisfied with EncryptChar and EncryptString, write the decryption versions. Test those carefully as well.

6. Finally, write additional test code. Think outside the box.

**USING THE TEST CODE:**

Subroutines must interact with other code. To emphasize this, we have written some code that will call your code: test_Lab6.asm. To use it, put it in the same folder as Lab6.asm, and run test_Lab6.asm in MARS. test_Lab6.asm calls EncryptChar and EncryptString, so Lab6.asm must, at a minimum, contain those labels.

Read the calling code carefully. It uses several MARS and MIPS techniques that you may be unfamiliar with. Ask questions if parts of this code do not make sense to you.

Notice that test_Lab6.asm contains a preprocessor macro at the very end, ".include Lab6.asm". This is how MARS includes your code in the assembly process. Upon assembly, you will see that your code has been "pasted" into the Text Segment, and that the Data Segment contains data from both Lab6.asm and test_Lab6.asm. (This is part of why you must be cautious about creating side effects in your code!)

As you progress in the lab, we expect you to generate your own test code. You may modify test_Lab6.asm as you wish. You may also make your own versions of the code inside it, and name them whatever you wish. Consider other approaches to writing test code that could simplify the testing process.

Testing is an extremely important skill in engineering, so treat this as an opportunity to develop your professional technique.

**SUBROUTINES:**

To interact correctly with the testing code, your code must use MIPS calling conventions. Information is passed into the subroutine using $a registers, and information is returned from the subroutine using the $v registers. The spec describes the specific role of each register. Furthermore, $s registers are meant to be preserved when subroutines are called, so their values are unchanged from the value it contained after the subroutine returns.

After a subroutine is complete, the $pc must point to the address of the instruction after the calling instruction (that is, the program counter should "return" to the calling code). Use "jal" to store the address of the return instruction in $ra, and "jr $ra" to return to the appropriate instruction.

Note that calling a subroutine inside of another subroutine will overwrite $ra, so you must "save" original values of $ra to return to the correct address. You may use memory local to the subroutine, an appropriate register, or put the $ra values on the stack, as you wish.

**ENCRYPTING AND DECRYPTING CHARACTERS:**

To encrypt characters according to the rules of the Vigenere cipher, you must treat the alphabet as a field of numbers on which you can do arithmetic (that is, translate [A-Z] to [0-25] and do arithmetic on the integers). This approach should be familiar to you from Lab5, though you must take extra measures to ensure that encrypted/decrypted characters remain within the cipher's alphabet.

EncryptChar and DecryptChar should never take inputs that are not upper or lowercase letters. You will need to handle upper and lowercase letters slightly differently. Be especially careful about encrypting uppercase letters and decrypting lowercase letters, so that uppercase letters do not encrypt to lowercase letters.

**ENCRYPTING AND DECRYPTING STRINGS:**

To encrypt or decrypt a string, you must iterate along the string, and handle each character independently. Again, this should be familiar to you from Lab5. In this case, however, you must iterate over three strings simultaneously.

Once the characters to operate on have been detected, you must call EncryptString or DecryptString as appropriate. EncryptString and DecryptString should not handle encryption arithmetic.

The Vigenere cipher, as we have defined it, has special rules to handle punctuation. This can be handled easily with well-structured code, so make sure you spend plenty of time on your pseudocode.

**LAB WRITEUP:**

Insert your Pseudocode that accurately reflects the structure of your code as a block comment at the top of your Lab6.asm file.

Be sure your write-up (in the README.txt file) contains:

- Appropriate headers
- Describe what you learned, what was surprising, what worked well and what did not
- Answer the following questions:
  1. What additional test code did you write? Why? Did it work?
  2. What happens when you attempt encryption with a keystring that has illegal characters (for example, "NotALegalKey"? Why?
  3. How would you write a recursive subroutine (a subroutine that calls itself)?
  4. According to MIPS calling conventions, arguments should be passed in the $a registers. There are only 4 of these registers. What could you do to pass more than 4 arguments into a subroutine?

To alleviate file format issues we want lab reports in plain text. Please ensure that your README.txt is a plain text file and CHECK IT ON GITLAB to make sure it is readable.

**BUMPS AND ROAD HAZARDS:**

Start this lab early. Ask questions.

Your success or failure in this lab will be determined to a large extend by the quality of your test code. The ability to run a variety of tests with little effort makes debugging much quicker. Do not treat your test code like an afterthought!

Subroutines may appear complex at first, but if you use them effectively, they actually reduce complexity. For example, calling EncryptChar inside of EncryptString may seem like brain-twister, but the result is that you do not need to handle encryption arithmetic inside EncryptString. This way, you can split the debugging of EncryptString into smaller pieces. If EncryptString has an error, test EncryptChar to see if the error is there. This can dramatically speed up the debugging process.

You may find it useful to create your own subroutines (For example, a subroutine that determines if a character is uppercase, lowercase, or other is very handy). This is encouraged! Just make sure you use the appropriate headers for your subroutines, so we understand what you did.