CSE222 – Proj HW 2

Due 11:59pm check black board for dates

In this assignment you will be creating functions. The goal is to understand argument passing, returning values, and the role of register conventions. The theme of the assignment is around basic steganography and will give you good practice manipulating arrays and strings in assembly language.

You **must** implement all the functions in the assignment as defined. It is OK to define additional helper functions of your own in hw2.asm .

⚠ **You must follow the MIPS calling and register conventions (Textbook Page 330, Table 6.3). If you do not, you WILL lose points.**

Table 6.3 Preserved and nonpreserved registers

Preserved	Nonpreserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Return address: `$ra`	Argument registers: `$a0–$a3`
Stack pointer: `$sp`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

⚠ Do not submit a file with the functions/labels **main** or **_start** defined. You will obtain a **ZERO** for the assignment if you do this.

ℹ If you are having difficulties implementing these functions, write out the pseudo code or implement the functions in a more comfortable language first. Once you understand the algorithm and what steps to perform, then translate it to MIPS.

Getting started

- Download `hw2.zip` from piazza in the homework section of resources.
  - This file contains `hw2.asm` and `main.asm` which you need for the assignment.
- At the top of your hw2.asm program in comments put your name and id.

```
# Homework #2
# name: MY_NAME
# sbuid: MY_SBU_ID
```

How to test your functions

To test your functions simply open the provided `hw2.asm` and `main.asm` in MARS. Next just assemble `main.asm` and run the file. Mars will take the contents of the file referenced with the `.include` at the end of the file and add the contents of your file to the main file before assembling it. Once the contents have been substituted into the file, Mars will then assemble it as normal.

`main.asm` tests each one of the functions with one of the sample test cases. You should modify this file to test your functions with more test cases. Your assingment will not be graded using these tests.

Any modifiations to main.asm will not be graded, as you will only submit your `hw2.asm` file via sparky. Make sure that all code require for implementing your functions (`.text` and `.data`) are included in the `hw2.asm` file!

> ⚠ It is highly advised to write your own main programs (individual files) to test each of your functions thoroughly.
> ⚠ Make sure to initialize all of your values within your functions! Never assume registers will hold any particular value!
>
> ℹ All strings passed to functions for grading will be Null terminated.

Part 1: String Manipulation

Functions

The goal is to implement string functions which will serve as helper functions in Part 2. Create the following functions: `toUpper` , `length2Char` , and `strcmp` .

toUpper function

```
/**
 * Converts all lowercase letters in null-terminated
 * string to uppercase in the original string. Leave
 * all other characters unchanged.
 * Do not make a copy of the string.
 *
 * @param string Start address of Null-terminated string.
 * @return Returns the address of the string passed in.
 */
public abstract char[] toUpper(char[] string);
```

Examples:

string: `Computer Science is fun.\0`
resultant string: `COMPUTER SCIENCE IS FUN.\0`

string: `I'll be back!!!!!!!!!\0`
resultant string: `I'LL BE BACK!!!!!!!!!\0`

string: `!@#$%^&*(\0`
resultant string: `!@#$%^&*(\0`

> ℹ `\0` is the NULL ASCII character, not 2 separate characters in the string. The NULL character **WILL NOT** print to the screen but is shown in these examples for clarity in explanation.

length2Char function

```
/**
 * Determines the length of a string until the specified
 * terminator character or '\0' is found.
 *
 * @param string Null-terminated string.
 * @param terminator Address of Terminator character
 * @return Returns the number of chars prior to terminator
 * character or end of string.
 */
public abstract int length2Char(char[] string, char[]
terminator);
```

Examples:

string: `Computer Science is fun.\0`
terminator: `S`
length: 9

string: `I'll be back!\0`
terminator: `T`
length: 13

string: `\0`
terminator: `\0`
length: 0

string: `I'll be back!\0`
terminator: `'`
length: 1

> ℹ `\0` is the NULL ASCII character, this isn't 2 separate characters in the string.

strcmp function

Our implementation of `strcmp` is a variation of the standard string compare function, which includes a `length` argument.

When the `length` argument is set to `0` , the `strcmp` function will compare the two strings in their entirety to determine a match. The function will return two values: (i) the

number of characters within the string which did match (not including the '\0' terminator) and (ii) 1 or 0, indicating whether or not the strings matched (1 = strings matched, 0 = strings did not match).

When `length < 0`, the argument is invalid. No comparisons of the strings are made and the function should return (0,0).

When `length > 0`, the function must check to see if the argument `length` is greater than the number of characters in `str1` or `str2`. If the length is larger than either string, then the argument is invalid. No comparisons of the strings are made and the function should return (0,0).

When `length > 0` and is shorter than the number of characters in `str1` or `str2`, the argument is valid. The function should perform exactly `length` number of comparisons from the start of `str1` and `str2`. If `length` comparisons between the strings match, return (length, 1). If fewer than `length` characters match between the two strings, return (number of matching characters, 0).

```
/**
 * Performs case-sensitive string comparison with a character
 * comparison limit.
 *
 * @param str1 Starting address of string one to compare.
 * @param str2 Starting address of string two to compare.
 * @param length The number of character comparisons to
 * be performed from the start of each string. A value
 * of 0 means to compare strings until '\0'. A negative value
 * is considered invalid and results in error.
 * @return int Returns number of matching characters before
 * returning (completion or error), not including '\0'.
 * @return int Returns 1 if all compared characters in
 * str1 match str2, 0 if otherwise (invalid arg, error, etc).
 */
public abstract (int,int) strcmp(char[] str1, char [] str2, int
length);
```

ℹ The return type **(int, int)** is a "made-up" return type specifying that the function returns two integer values. Remember that in MIPS you have two registers (`$v0`, `$v1`) which are designated to store return values from a function.

Examples:

str1: `MIPS!!\0`
str2: `MIPS - Millions.of.Instruction.Per.....Second\0`
length: 4
return: 4, 1

str1: `MIPS!!\0`
str2: `MIPS!!\0`
length: 0
return: 6, 1

str1: `MIPS!!\0`
str2: `MIPS!!\0`
length: 10
return: 0, 0

str1: `MIPS!!\0`
str2: `MIPS!! - Millions.of.Instruction.Per.....Second\0`
length: 7
return: 0, 0

str1: `MIPS!!\0`
str2: `MIPS - Millions.of.Instruction.Per.....Second\0`
length: 6
return: 4, 0

str1: `MIPS!!\0`
str2: `MIPS\0`
length: 0
return: 4, 0

Part 2: Fractionated Morse Cipher Encryption

In this part we will perform cipher encryption of a plaintext message using the Fractionated Morse Cipher. The Fractionated Morse cipher begins by converting a plaintext message to morse code. Then it enciphers fixed size blocks of morse code back to letters. Morse code has representations for punctuation, numbers, and space which enables these characters to also be encrypted.

We will use the following sequences of "dashes and dots" to represent a set of letters, symbols and numbers in morse code.

Char	Pattern	Char	Pattern	Char	Pattern	Char	Pattern
A	.-	N	-.	.	.-.-.-	1	.----
B	-...	O	---	,	--..--	2	..---
C	-.-.	P	.--.	:	---...	3	...--
D	-..	Q	--.-	"	.-..-.	4	....-
E	.	R	.-.	'	.----.	5	.....
F	..-.	S	...	!	-.-.--	6	-....
G	--.	T	-	?	..--..	7	--...
H	....	U	..-	@	.--.-.	8	---..
I	..	V	...-	-	-....-	9	----.
J	.---	W	.--	;	-.-.-.	0	-----
K	-.-	X	-..-	(	-.--.		
L	.-..	Y	-.--	)	-.--.-		
M	--	Z	--..	=	-...-		

Each of the patterns have been defined in the .data section in the `hw2.asm` file. To reduce the complexity to lookup each of these patterns, the array `MorseCode` has been defined. Each element in the array is the base address of a pattern. The patterns are ordered in the array as the characters are in the ASCII table, beginning with the character `!`.

For example, MorseCode[0] holds the base address of the null-terminated string for `!`, which is `-.-.--`

Symbols which appear in the ASCII table between the characters `!` and `Z` which do not have a morse code encoding are specified with empty strings in the `.data` section. They are included in the MorseCode Array to simplify the logic for lookup.

> ℹ Uppercase characters and symbols which appear in the Morse Code table above will be translated. All characters not in this table, **INCLUDING LOWERCASE characters**, MUST be ignored by this function!

Examine the `.data` section of `hw2.asm` to understand the `MorseCode` array and its corresponding strings.

Plaintext to Morse Code

Begin by creating a function to encode a plaintext message into morse code.

```
/**
 * Converts a plaintext message to morse code. The
 * character 'x' is used to denote a "stop" after
 * their respective morse code sequences. The
 * characters 'xx' are used to denote a
 * "stop" (space) between words and at the end of the message.
 * If the encoding morse code sequence is longer than the size
 * provided, truncate the morse code sequence.
 * At most (size-1) characters of morse code should be stored.
 *
 * Created morse code sequence must be null-terminated.
 *
 * Characters which do not have morse code patterns should
 * be skipped over.
 *
 * @param plaintext Starting address of Null-terminated
 * string to convert to morse code. String is assumed to
 * contain uppercase characters. Lowercase characters will
 * be ignored.
 * @param mcmsg Starting address of allocated space to store
 * morse code sequence.
 * @param size Total number of bytes allocated to store
 * morse code sequence and '\0' terminator. size < 1
 * is invalid, return (0,0).
 * @return int Returns length of morsecode, including
 * '\0'.
 * @return int Returns 1 if plaintext message was
 * completely and correctly encoded into morsecode,
 * 0 otherwise (not enough space, error, etc).
 */
public abstract (int,int) toMorse(char[] plaintext, char []
mcmsg, int size);
```

ℹ If the morse code sequence is longer than the size of the buffer provided, truncate the morse code sequence to fit with the Null-terminator. This means, at most (size-1) characters of morse code should be stored.
ℹ All created morse code sequence **MUST BE** null-terminated.

ℹ You may use the `MorseCode` label defined in the .data section within your function.
ℹ The return type **(int, int)** is a "made-up" return type specifying that the function returns two integer values. Remember that in MIPS you have two registers (`$v0` , `$v1`) which are designated to store return values from a function.

Examples:

plaintext: `MIPS!!\0`
size: 35
mcmsg: `--x..x.--.x...x-.-.--x-.-.--xx\0`
return: 31, 1

plaintext: `MIPS!!\0`
size: 30
mcmsg: `--x..x.--.x...x-.-.--x-.-.--x\0`
return: 30, 0

plaintext: `MIPS!!\0`
size: 10
mcmsg: `--x..x.--\0`
return: 10, 0

plaintext: `Mips!!\0`
size: 30
mcmsg: `--x-.-.--x-.-.--xx\0`
return: 19, 1

plaintext: `MIPS Is Fun!\0`
size: 35
mcmsg: `--x..x.--.x...xx..xx..-.x-.-.--xx\0`
return: 34, 1

plaintext: `\0`
size: 10
mcmsg: `\0`
return: 1, 1

Generate Key from Phrase

To encrypt/decrypt the morse code message a secret key shared between the sender and receiver is required. This key must be a mixed alphabet key, meaning it must contain every letter of the alphabet. A random ordering of the alphabet is difficult for anyone to remember, therefore instead we will use a phrase and add any missing alphabet characters.

Create the following function, `createKey`, to generate a mixed alphabet key based on a given phrase.

```
/**
 * Extracts all unique alphabet characters from the
 * input string 'phrase' in the order of their appearance
 * and places them into key.
 * Any remaining alphabet characters which did not appear
 * in phrase are added to 'key' in alphabetical order.
 *
 * Key will always be 26 characters in length and contain
 * only CAPITAL letters of the alphabet.
 *
 * @param phrase Starting address of the `phrase` to extract
 * characters from. `phrase` is a null-terminated string.
 * @param key Starting address of 26-bytes of memory to store
 * the output message.
 */
public abstract void createKey(char[] phrase, char [] key);
```

⚠ The `createKey` function must call the `toUpper` function.
ℹ Ignore symbols/punctuation, spaces, non-visible characters and numbers.
ℹ You may assume 26-bytes have been allocated to hold `key` values and it is followed by a Null terminator.

Examples:

phrase: Computer Science Is Fun.\0
key: COMPUTERSINFABDGHJKLQVWXYZ

phrase: I'LL be BACK!!!!!!!!!\0
key: ILBEACKDFGHJMNOPQRSTUVWXYZ

phrase: \0
key: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Helper function: Mapping Morse Code to Key Value

Using the created key and the morse code sequence, we can encrypt our message with the cipher. Each sequence of 3 morse code symbols, `.`, `-`, or `x`, is mapped to an index of the key (a single alphabet character).

Pattern	Key	Pattern	Key
`...`	Key[0]	`---`	Key[13]
`..-`	Key[1]	`--x`	Key[14]
`..x`	Key[2]	`-x.`	Key[15]
`.-.`	Key[3]	`-x-`	Key[16]
`.--`	Key[4]	`-xx`	Key[17]
`.-x`	Key[5]	`x..`	Key[18]
`.x.`	Key[6]	`x.-`	Key[19]
`.x-`	Key[7]	`x.x`	Key[20]
`.xx`	Key[8]	`x-.`	Key[21]
`-..`	Key[9]	`x--`	Key[22]
`-.-`	Key[10]	`x-x`	Key[23]
`-.x`	Key[11]	`xx.`	Key[24]
`--.`	Key[12]	`xx-`	Key[25]

These sequences are defined as one ascii string in the .data section of `hw2.asm` called `FMorseCipherArray`. Every 3 characters in the `FMorseCipherArray` correspond to the index of the key value.

Write a function `keyIndex` to search the `FMorseCipherArray`, find the matching sequence and return the index in the key array. Use your `strcmp` function from Part 1 to match only the first 3 characters of the morse code message to each of the 3 characters of

the `FMorseCipherArray`.

```
/**
 * Match the first 3 characters of the morse code message
 * and return the corresponding key index value.
 *
 * @param mcmg Starting address of string to match.
 * @return Return the key index of the 3 character match
 * or -1 if no match is found.
 */
public abstract int keyIndex(char[] mcmsg);
```

⚠ The `keyIndex` function must call the `strcmp` function.

Examples:

mcmsg: `--x..x.--.x...xx..xx..-.x-.-.--xx\0`
key: 14

mcmsg: `.x...xx..xx..-.x-.-.--xx\0`
key: 6

mcmsg: `...xx..xx..-.x-.-.--xx\0`
key: 0

mcmsg: `.A-xx\0`
key: -1

mcmsg: `\0`
key: -1

mcmsg: `-.\0`
key: -1

Bringing it all together

At this point, we have created all the functionality of the Fractionated Morse Cipher except producing the encrypted message. Encryption is performed by replacing every 3 characters of the morse code with it's corresponding letter in the key. The letters create the encrypted message.

For example, consider the following plaintext message, "GO SEAWOLVES!". We will encrypt this message using the key phrase, "Computer Science is cool!"

First "GO SEAWOLVES!" is translated to morse code using the `toMorse` function.
plaintext: GO SEAWOLVES!\0
size: 100
mcmsg: `--.x---xx...x.x.-x.--x---x.-..x...-x.x...x-.-.--xx\0`
return: 51, 1

Next, the phrase is translated to the key using `createKey` :
phrase: Computer Science is cool!\0
key: COMPUTERSINLABDFGHJKQVWXYZ

This means key[0] is 'C', key[1] = 'O', … , key[10] = 'N', …, key [25] = 'Z'.

To encrypt, we take each 3 character of the `mcmsg` and call funcion `keyIndex` to determine the key index. Any leftover characters at the end of the `mcmsg` will not be encrypted.

`--.` maps to key[12] which is 'A'
`x--` maps to key[22] which is 'W'
`-xx` maps to key[17] which is 'H'
`...` maps to key[0] which is 'C'
`x.x` maps to key[20] which is 'Q'
`.-x` maps to key[5] which is 'T'
`.--` maps to key[4] which is 'U'
`x--` maps to key[22] which is 'W'
`-x.` maps to key[15] which is 'F'
`-..` maps to key[9] which is 'I'
`x..` maps to key[18] which is 'J'
`.-x` maps to key[5] which is 'T'
`.x.` maps to key[6] which is 'E'
`..x` maps to key[2] which is 'M'
`-.-` maps to key[10] which is 'N'
`.--` maps to key[4] which is 'U'
`xx` maps to no key. These 2 characters will be dropped.

The resultant encoding of the message will be: `AWHCQTUWFIJTEMNU`

> Refer to RffNekk's Crypto Pages - Fractionated Morse Cipher for a tool which can verify your encryption is correct.

Create the function `FMCEncrypt` which will take the plaintext message, the key phrase and

encrypt the message. The encrypted message must fit in the given encrypt buffer with a maximum character length of argument size.

```
/**
 * Encrypt the plaintext message into the encryptBuffer
 * with the Fractionated Morse Cipher. Use the phrase to
 * create the key.
 * @param plaintext The start address of the plaintext message
 * @param phrase The start address of the phrase
 * @param encryptBuffer The start address of the buffer for the
 * encrypted message
 * @param size The size(ie number of bytes) in the buffer
 * @return char[] Returns address of the encryptBuffer
 * @return int Returns 1 if plaintext message was
 * completely and correctly encoded into the
 * encryptBuffer (with '\0'), 0 otherwise (not
 * enough space, error, etc).
 */
public abstract (char[], int) FMCEncrypt(char[] plaintext, char
[] phrase, char[] encryptBuffer, int size);
```

⚠ The `FMCEncrypt` function must call the `toMorse`, `createKey`, and the `keyIndex` function.

⚠ **Assume the maximum length of the plaintext message will be 100 characters, including the '\0'.** You may declare space in the .data section to hold the intermediate morse code for this function.

ℹ The return type **(char[], int)** is a "made-up" return type specifying that the function returns two values. Remember that in MIPS you have two registers (`$v0`, `$v1`) which are designated to store return values from a function.

Extra Credit - Fractionated Morse Cipher Decryption

To decrypt a message, the message and the phrase must be known. Using the same phrase as the message was encoded with, you must first create the key using `createKey`. Once the key is generated, each character in the ciphertext message can be mapped back to its 3 morse code symbols using the `length2Char` function on the key and the

`FMorseCipherArray`. With the morse code sequence, you will then need to translate back to plaintext. Create a `fromMorse` function to translate from morse code back to plaintext.

```
/**
 * Decrypt the ciphertext message into the plaintext message
 * with the Fractionated Morse Cipher. Use the phrase to
 * create the key.
 * @param ciphertext The start address of the ciphertext message
 * @param phrase The start address of the phrase
 * @param decryptBuffer The start address of the buffer for the
 * decrypted message
 * @param size The size(ie number of bytes) in the decryptBuffer
 * @return char[] Returns address of the decryptBuffer
 * @return int Returns 1 if ciphertext message was
 * completely and correctly decoded into the
 * encryptBuffer (with '\0'), 0 otherwise (not
 * enough space, error, etc).
 */
public abstract (char[], int) FMCDecrypt(char[] ciphertext,
char [] phrase, char[] decryptBuffer, int size);
```

⚠ The `FMCDecrypt` function must call the `fromMorse`, `createKey`, and the `length2Char` function.

⚠ **Assume the maximum length of the original plaintext message will be 100 characters, including the '\0'.** You may declare space in the .data section to hold the intermediate morse code for this function (or use the same space as you declared for encrypt).

ℹ The return type **(char[], int)** is a "made-up" return type specifying that the function returns two values. Remember that in MIPS you have two registers (`$v0`, `$v1`) which are designated to store return values from a function.

```
/**
 * Converts a morse code string to plaintext
 * message. The
 * character 'x' is used to denote a "stop" between
 * characters. The characters 'xx' are used to denote a
 * "stop" between words. At most (size-1) characters of
 * plaintext should be stored.
 *
 * The created plaintext string must be null-terminated.
```

```
 *
 * @param morsecode Starting address of null-terminated morse
code
 * string to convert to plaintext.
 * @param plaintextBuffer Starting address of allocated space to
store
 * plaintextBuffer.
 * @param size Total number of bytes allocated to store
 * plaintext and '\0' terminator.
 * @return int Returns length of plaintext message,
 * including '\0'.
 * @return int Returns 1 if morse code message was
 * completely and correctly decoded into plaintext,
 * 0 otherwise (not enough space, error, etc).
 */
public abstract (int,int) fromMorse(char[] morsecode, char []
plaintextBuffer, int size);
```

ℹ You may use the `MorseCode` and `FMorseCipherArray` labels defined in the .data section within your function.

ℹ The return type **(int, int)** is a "made-up" return type specifying that the function returns two integer values. Remember that in MIPS you have two registers (`$v0` , `$v1`) which are designated to store return values from a function.

Hand-in instructions

There is no tolerance for homework submission via email. They must be submitted through blackboard. Please do not wait until the last minute to submit your homework. If you are struggling, stop by tutoring service.

ℹ When writing your program try to comment as much as possible. Try to stay consistent with your formatting.