# CSE 220: Systems Fundamentals I

# Homework #2

# Fall 2016

**Assignment Due: October 7, 2016 by 11:59 pm**

## Assignment Overview

In this assignment you will be creating functions. The goal is to understand passing arguments, returning values, and the role of register conventions. The theme of the assignment is basic data compression and will give you good practice manipulating arrays and strings in MIPS.

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

⚠ **You MUST follow the MIPS calling and register conventions. If you do not, you WILL lose points.**

⚠ 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 pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS.

ℹ When writing your program, try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly.

## 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 SBU ID number.

```
# 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, or create your own files, to test your functions with more test cases.

⚠ Your assignment will not be graded using these tests.

Any modifications to `main.asm` will not be graded. 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 (new individual files) to test each of your functions thoroughly.

⚠ Make sure to initialize all of your values within your functions! Never assume registers or memory will hold any particular values!

# Run-length Encoding

Run-length encoding is a simple compression scheme best used when a data-set consists primarily of numerous, long runs of repeated characters. For example, `AAAAAAAAAA` is a run of 10 A's. We could encode this run using a notation like `*A10`, where the `*` is a special **flag character** that indicates a run, `A` is the symbol in the run, and `10` is the length of the run. As another example, the string

`bbbbbbbbbbbbBccddddddddddddddddKKKKKMNUUUGGGGG`

would be encoded

`$b13Bcc$d15$K5MNUUU$G5`

assuming in this case that `$` is the flag character. For the sake of this assignment, we will make the following assumptions:

- the input strings to be encoded contain only lowercase and uppercase letters from the Latin alphabet
- single letters, runs of two letters, and runs of three letters are not encoded (Doing so does not save memory or actually compress the data. Why??).

⚠ You may assume that no run is any longer than 99 characters in length.

⚠ You may assume that no string to be encoded is any longer than 1,000 characters in length.

Strive to implement the functions in the order given below, as some of the later functions depend on the earlier ones to work properly. Remember that you must implement the functions as specified below.

# Part I: Helper Functions

a. `int atoui(char[] input)`

This function converts a string of ASCII decimal digits representing a positive integer into a positive integer.

- `input`: the starting address of a string representing the positive integer to convert, which will be terminated by a non-digit character or NULL.
- returns: the converted, positive integer

⚠ When a null terminator or a non-digit character is encountered while processing the input, the function returns the number computed up to that point. If no digit is encountered at all, then 0 is returned.

⚠ The space character is considered a non-digit character.

Examples:

| Code | Return Value |
|---|---|
| `atoui("723go1")` | 723 |
| `atoui("abc31")` | 0 |
| `atoui("-24")` | 0 |
| `atoui("12#34")` | 12 |
| `atoui("15\0")` | 15 |
| `atoui(" 22\0")` | 0 |
| `atoui("\0")` | 0 |

b. `(char[], int) uitoa(int value, char[] output, int outputSize)`

This function converts the positive integer `value` into a string of ASCII decimal digits. The function does NOT write a null-terminator. If `value` is negative or zero, the function returns the pair (address_of_`output`, 0) without making any changes to the `output` parameter. Note that the function returns two values: `$v0` holds the first return value and `$v1` holds the second return value.

- `value`: the positive integer to convert
- `output`: the address of string where the converted number is to be stored

- `outputSize`: the number of bytes for the `output` string.
- returns: (i) the address of the byte immediately following the last character written by the function, or the address of `output` and (ii) 1 if the conversion was successful and 0, otherwise.

⚠ `value` is a positive 2's complement value in a register. Therefore, the largest value which can be converted by `uitoa` is $2^{31} - 1$.

⚠ If the `output` string is not of sufficient length to store the converted value, the function must return 0 without making changes to the `output` array.

Examples:

| Code | Return Values | Stored in Output |
|---|---|---|
| `uitoa(123, 0x400, 10)` | (0x403,1) | "123" |
| `uitoa(987654321, 0x800, 10)` | (0x809,1) | "987654321" |
| `uitoa(123, 0x400, 2)` | (0x400,0) | Unmodified |
| `uitoa(0, 0x408, 100)` | (0x408,0) | Unmodified |
| `uitoa(-24, 0x800, 10)` | (0x800,0) | Unmodified |

# Part II: Decoding Data

In this section, you will be writing functions to decode a run-length encoding.

c. `int decodedLength(char[] input, char runFlag)`

This function determines how many bytes (including a null-terminator) that would be required to store the decoded `input` string.

- `input`: a pointer to a null-terminated string containing a run-length encoded string to be decoded. Your function may assume that `input` is a properly-formatted, run-length-encoded string.
- `runFlag`: a symbol flag character in the set !#$%^&* that indicates the start of a run. The function returns 0 if `runFlag` is an alphanumeric character.
- returns: the number of bytes that would be needed to store the decoded string passed as input. If `input` is the empty string, the function returns 0.

❗ Under no circumstances may the function alter the contents of `input`.

Examples:

| Code | Return Value |
|---|---|
| `decodedLength("sss!j4q!F5\0", '!')` | 14 |
| `decodedLength("sx*j24qyyy*g6\0", '*')` | 37 |
| `decodedLength("sss!j4q!F5\0", 'g')` | 0 |
| `decodedLength("\0", '$')` | 0 |

d. `(char[], int) decodeRun(char letter, int runLength, char[] output)`

This function writes `runLength` copies of the `letter` parameter into the string `output`. No other characters of the array may be modified. The function does not insert any null terminators. If `runLength` is less than 1 or `letter` is a non-alphabetical character, then no encoding takes place and `output` is not modified. The function returns the pair (address_of_`output`, 0) in such cases. The function may safely assume that enough memory has been set aside in `output` to store the decoded run.

- `letter`: the letter of the alphabet in the run.
- `runLength`: the length of the run.
- `output`: the address where the run is written to
- returns: the address of the character immediately following the final letter in the run that was written into the `output` string, or the address of `output` upon error. The second return value is 1 if the decoding was successful and 0, otherwise.

Examples:

| Code | Return Values | Stored in Output |
|---|---|---|
| `decodeRun("G", 6, 0x400)` | (0x406, 1) | "GGGGGG" |
| `decodeRun("q", 12, 0x800)` | (0x80C, 1) | "qqqqqqqqqqqq" |
| `decodeRun("h", -2, 0x404)` | (0x404, 0) | Unmodified |
| `decodeRun("*", 9, 0x80C)` | (0x80C, 0) | Unmodified |

e. `int runLengthDecode(char[] input, char[] output, int outputSize, char runFlag)`

This function performs a run-length decoding of the `input` string and saves it in the `output` parameter.

- `input`: a pointer to a null-terminated string containing a run-length-encoded string to be decoded. Your function may assume that `input` is a properly-formatted, run-length-encoded string. Your function may assume that no other characters (including whitespace) are present in the string.
- `output`: a pointer to a region in memory where the function stores the null-terminated, run-length decoded string.

- `outputSize`: the (positive) number of bytes in the output string, including memory for the null terminator. For instance, if `outputSize` were 10, this means at most 9 characters could be stored in output. The function may assume that `outputSize` is positive.
- `runFlag`: a symbol flag character in the set !@#$%^&* that indicates the start of a run. The function returns 0 if `runFlag` is an alphanumeric character.
- returns: 1 if the encoding was successful and 0, otherwise.

❗ If the `output` string is not of sufficient length to store the decoded data, the function must return 0 without making changes to the `output` array.

❗ Under no circumstances may the function alter the contents of `input`.

❗ `runLengthDecode` MUST call `decodeRun`, `decodedLength` and `atoui`.

Example #1:

| **Code** | **Return** |
|---|---|
| `runLengthDecode(0x600, 0x704, 18, '!')` | 1 |
| **`input[]`** | **`output[]`** |
| "sss!j4q!F5\0" | "sssjjjjqFFFFF\0" |

Example #2:

| **Code** | **Return** |
|---|---|
| `runLengthDecode(0x800, 0x400, 8, '*')` | 0 |
| **`input[]`** | **`output[]`** |
| "*A5hhh*U11V\0" | Unmodified |

Example #3:

| **Code** | **Return** |
|---|---|
| `runLengthDecode(0x500, 0x208, 29, 'z')` | 0 |
| **`input[]`** | **`output[]`** |
| "jjj*p15ZzZz*h22\0" | Unmodified |

# Part III: Encoding Data

In this section, you will be writing functions to encode a string with run-length encoding.

f. `int encodedLength(char[] input)`

This function determines how many bytes (including a null-terminator) that would be required to store the `input` string if it were run-length encoded.

- `input`: a pointer to a null-terminated string containing a string to be run-length encoded. Your function may assume that `input` contains only letters.

- returns: the number of bytes that would be needed to store the string passed as input after being run-length encoded. If `input` is the empty string, the function returns 0.

❗ Under no circumstances may the function alter the contents of `input`.

Examples:

| Code | Return Value |
|---|---|
| `encodedLength("AAAAAAAAAAAAAAAAAAAAA\0")` | 5 |
| `encodedLength("xxhhhhhhhhhhhhhhhuuunnnnnnnrere\0")` | 17 |
| `encodeLength("\0")` | 0 |

g. `(char[], int) encodeRun(char letter, int runLength, char[] output, char runFlag)`

This function encodes a run of `runLength` copies of the `letter` parameter inside the `output` string. No other characters of the array may be modified. The function MUST NOT insert any null terminators. If (i) `letter` is a non-alphabetical character, or (ii) `runFlag` is an alphanumeric character, or (iii) `runLength` $\leq 0$, then no encoding takes place and `output` is not modified; the function returns (address_of_`output`, 0) in such cases. If $1 \leq$ `runLength` $\leq 3$, then `runLength` copies of `letter` are written into `output`.

ℹ The function may safely assume that enough memory has been set aside in `output` to store the encoded run.

- `letter`: the letter of the alphabet in the run
- `runLength`: the length of the run
- `output`: the address where the run will be encoded
- `runFlag`: a symbol flag character in the set !#$%^&* that indicates the start of a run.
- returns: (i) the address of the byte immediately following the final letter or digit of the run that was written to the `output` string; and (ii) 1 if the encoding was successful and 0, otherwise.

Examples:

| Code | Return Values | Stored in Output |
|---|---|---|
| `encodeRun("G", 17, 0x400, "!")` | (0x404, 1) | "!G17" |
| `encodeRun("q", 2, 0x800, "%")` | (0x802, 1) | "qq" |
| `encodeRun("h", -2, 0x404, "$")` | (0x404, 0) | Unmodified |
| `encodeRun("x", 12, 0x408, "h")` | (0x408, 0) | Unmodified |
| `encodeRun("*", 9, 0x80C, "!")` | (0x80C, 0) | Unmodified |

h. `int runLengthEncode(char[] input, char[] output, int outputSize,`

`char runFlag)`

This function performs a run-length encoding of the `input` string and saves it in the `output` parameter. Single letters, double letters and triple letters are not encoded, as doing so does not save memory or actually compress the data.

- `input`: a pointer to a null-terminated string containing only uppercase and lowercase letters. The function may assume that no other characters (including whitespace) are present in the string.
- `output`: a pointer to a region in memory where the function stores the null-terminated, run-length encoded string.
- `outputSize`: the (positive) number of bytes in the `output` string, including memory for the null terminator. For instance, if `outputSize` were 10, this means at most 9 characters could be stored in `output`.
- `runFlag`: a symbol flag character in the set !#$%^&* that indicates the start of a run. The function returns 0 if `runFlag` is an alphanumeric character.
- returns: 1 if the encoding was fully successful and 0, otherwise.

❗ If the `output` string is not of sufficient length to store the encoded data, the function must return 0 without making changes to the `output` array.

❗ Under no circumstances may the function alter the contents of `input`.

❗ `runLengthEncode` MUST call `encodeRun`, `encodedLength` and `uitoa`.

Example #1:

| **Code** | **Return** |
|---|---|
| `runLengthEncode(0x800, 0x400, 10, "*")` | 1 |
| **`input[]`** | **`output[]`** |
| "AAAhhhhhhhabc\0" | "AAA*h7abc\0" |

Example #2:

| **Code** | **Return** |
|---|---|
| `runLengthEncode(0x600, 0x704, 14, "!")` | 1 |
| **`input[]`** | **`output[]`** |
| "sssjjjjqFFFFF\0" | "sss!j4q!F5\0" |

Example #3:

| Code | Return |
|---|---|
| `runLengthEncode(0x500, 0x208, 29, "z")` | 0 |
| **`input[]`** | **`output[]`** |
| "UUuUUhhhhhhaaaaaaBBk\0" | Unmodified |

# Hand-in Instructions

See Sparky Submission Instructions on piazza for hand-in instructions.

❗ There is no tolerance for homework submission via email. Work must be submitted through Sparky. Please do not wait until the last minute to submit your homework. If you are struggling, stop by office hours for additional help.