1 PA3: String Operations in Assembly

In this assignment, you will write several ARM assembly functions to compute operations on strings. The programs you will be writing will mimic the behavior of functions found in the C standard library.

2 Background

Reading section 6.3.7 in the Harris textbook will be useful before starting this assignment.

In C, strings are represented as an array of characters terminated by the null character '\0' (ASCII value `0x0`). Each character in the string is allocated a byte, in a contiguous chunk of memory, to represent the ASCII value for that character. Now suppose we have an array of characters starting at memory address `0x2000`. The following piece of C code shows how the string "cse30" would be laid out in memory:

```
char str[6] = "cse30";
```

Array:	c	s	e	3	0	\0
Value in memory:	0x63	0x73	0x65	0x33	0x30	0x00
Address:	0x2000	0x2001	0x2002	0x2003	0x2004	0x2005

Note that enough memory has been allocated to accommodate for the null character in denoting the end of the string. Also, note that no memory was allocated for the name of the array. Rather, `str` is a pointer to the first character of the string at memory address `0x2000`. The following piece of C code demonstrates an alternative syntax which equivalently has `str` point to the first character in the string "cse30". The function headers you will be implementing in this assignment will follow the syntax below where variables of type `char *` will point to the starting address of the C string.

```
char * str = "cse30";
```

3 Assembly Functions

You will be implementing the following functions in ARM assembly. Do not modify the given function label. The C function header will be provided, and you will be expected to translate the function to assembly with the same input and output. Inputs for the function will be given to you in R0 - R3 in the same order as they appear in the function header. Remember you must preserve caller registers (R4 - R11) if you use them in your assembly function. In addition, please only use R0 - R10 as general-purpose registers. You can test your implementations by running your code with the provided tester program (see Section 4 for details).

3.1 `int strlen(char * input);`

Input: A string whose length is to be found
Output: Return an int representing the length of the string

Sample input in `testCases.txt`:

```
TEST CASES FOR STRLEN:
cse30
```

Relevant output displayed:

```
$ make pa3.run
$ python runTests.py
...
TESTING STRLEN:
Length of "cse30": returned 5
...
```

3.2 `int strcmp(char * input1, char * input2);`

Input: Two strings to be compared
Output: Return -1 if `input1` is less than `input2`, 0 if the strings are equal, or 1 if `input1` is greater than `input2`

Note that we are comparing the ASCII representations of the strings. `input1` is greater than `input2` if the first differing character has a larger ASCII value. If inputs are differing lengths, the longer string is greater unless the previous condition is already met.

Sample input in `testCases.txt`:

```
TEST CASES FOR STRCMP:
cse30, cse12
CSE30, cse30
abc, abcde
```

Relevant output displayed:

```
$ make pa3.run
$ python runTests.py
...
TESTING STRCMP:
Comparing "cse30" and "cse12": returned 1
Comparing "CSE30" and "cse30": returned -1
Comparing "abc" and "abcde": returned -1
...
```

3.3 `int strtrunc(char * input, int val);`

Input: A string and an int to truncate the string by
Output: Return -1 if `input` is unable to be truncated by `val`. Otherwise, return the new length of the string. The string must also be modified to reflect its new length.

Sample input in `testCases.txt`:

```
TEST CASES FOR STRTRUNC:
cse30, 2
cse30, 6
```

Relevant output displayed:

```
$ make pa3.run
$ python runTests.py
...
TESTING STRTRUNC:
Truncating "cse30" by 2 characters: returned 3, string is now "cse"
Truncating "cse30" by 6 characters: returned -1, string is now "cse30"
...
```

3.4 `int strrev(char * input, int start, int end);`

Input: A string and start/end indices representing the portion of the string to reverse
Output: Return -1 if indices result in an invalid operation (i.e. start $>$ end, start $<$ 0, end $\geq$ strlen(`input`)). Otherwise, return the number of characters that were reversed. The string must also be modified to reflect characters that were

reversed.

Sample input in `testCases.txt`:

```
TEST CASES FOR STRREV:
cse30, 1, 3
cse30, -1, 4
```

Relevant output displayed:

```
$ make pa3.run
$ python runTests.py
...
TESTING STRREV:
Reversing "cse30" from 1 to 3: returned 2, string is now "c3es0"
Reversing "cse30" from -1 to 4: returned -1, string is now "cse30"
...
```

3.5 `int palindrome(char * input);`

Input: A string to check if it's a palindrome
Output: Return 1 if `input` is a palindrome. Otherwise, return 0. Your implementation should be case and space sensitive (i.e. "Racecar" should return false).

Sample input in `testCases.txt`:

```
TEST CASES FOR PALINDROME:
cse30
racecar
```

Relevant output displayed:

```
$ make pa3.run
$ python runTests.py
...
TESTING PALINDROME:
Checking if palindrome "cse30": returned 0
Checking if palindrome "racecar": returned 1
...
```

3.6 `int strfind(char * input, char * toFind);`

Input: Two input strings
Output: Return the index of the first occurrence of `toFind` in `input`. If `toFind` is not found in `input`, return -1. Note if `toFind` is the empty string, the function should return 0.

Sample input in `testCases.txt`:

```
TEST CASES FOR STRFIND:
cse30, a
cse30, 30
```

Relevant output displayed:

```
$ make pa3.run
$ python runTests.py
...
TESTING STRFIND:
Find "a" in "cse30": returned -1
Find "30" in "cse30": returned 3
...
```

4 Testing Your Functions

We have provided a program that will allow you to run your assembly files with multiple test cases and view its output. To add a test case, enter the required arguments for the function on a new line separated by commas. Any test cases should be added beneath the corresponding section in `testCases.txt`. You may add as many test cases as you like; however, do not modify any of the given text.

1. Add test cases in `testCases.txt` as follows:

```
TEST CASES FOR STRLEN:
cse30

TEST CASES FOR STRCMP:
cse30, cse12
CSE30, cse30
abc, abcde

TEST CASES FOR STRTRUNC:
cse30, 2
cse30, 6

TEST CASES FOR STRREV:
cse30, 1, 3
cse30, -1, 4

TEST CASES FOR PALINDROME:
cse30
racecar

TEST CASES FOR STRFIND:
cse30, a
cse30, 30
```

2. Compile your assembly files using `make pa3.run`

```
$ make pa3.run
as -g -o strlen.o strlen.s
as -g -o strcmp.o strcmp.s
as -g -o strtrunc.o strtrunc.s
as -g -o strrev.o strrev.s
as -g -o palindrome.o palindrome.s
as -g -o strfind.o strfind.s
gcc -g -o pa3-runner strlen.o strfind.o strcmp.o strtrunc.o strrev.o
palindrome.o pa3.o
```

3. Then, run the following command to perform the test cases you specified:

```
$ python runTests.py
<-------------------------------------------------------------->
TESTING STRLEN:
Length of "cse30": returned 5
<-------------------------------------------------------------->
TESTING STRCMP:
Comparing "cse30" and "cse12": returned 1
Comparing "CSE30" and "cse30": returned -1
Comparing "abc" and "abcde": returned -1
<-------------------------------------------------------------->
TESTING STRTRUNC:
```

```
Truncating "cse30" by 2 characters: returned 3, string is now "cse"
Truncating "cse30" by 6 characters: returned -1, string is now "cse30"
<------------------------------------------------------------------>
TESTING STRREV:
Reversing "cse30" from 1 to 3: returned 2, string is now "c3es0"
Reversing "cse30" from -1 to 4: returned -1, string is now "cse30"
<------------------------------------------------------------------>
TESTING PALINDROME:
Checking if palindrome "cse30": returned 0
Checking if palindrome "racecar": returned 1
<------------------------------------------------------------------>
TESTING STRFIND:
Find "a" in "cse30": returned -1
Find "30" in "cse30": returned 3
<------------------------------------------------------------------>
```

5 Debugging your programs

The runTests.py is a script that simply calls `pa3-runner` functions at once. It's possible to test each of your functions separately using `pa3-runner`. The `pa3-runner` can be run individually for each of the six functions as follows.
From your local repository that holds the cloned files, run:

- strlen:

  `./pa3-runner strlen <input>`

- strcmp:

  `./pa3-runner strcmp <input1> <input2>`

- strtrunc:

  `./pa3-runner strtrunc <input> <val>`

- strrev:

  `./pa3-runner strrev <input> <start> <end>`

- palindrome:

  `./pa3-runner palindrome <input>`

- strfind:

  `./pa3-runner strfind <input> <toFind>`

Optional: To use `pa3-runner` instead of `./pa3-runner`, create an alias from the folder you're working out of:

```
$ echo "alias pa3-runner=$(pwd)/pa3-runner" >> ~/.bash_profile
$ source ~/.bash_profile
```

You can then use the runner as:
`pa3-runner strlen <input>` **Using gdb:** To use gdb, you'll have to use the `--args` argument to pass parameters to the string function being called. Following that, you can set a breakpoint at the *label* used in the assembly file. For example, to debug `strlen`:

```
$ gdb --args pa3-runner strlen cse30
```

(gdb) `b mystrlen` [set a breakpoint]

(gdb) `start` [start the program]

(gdb) `layout reg` [show the layout with sourc e code and registers]

(gdb) `continue` [jump to the breakpoint]

From here on, you can debug the program.

6 Reminders

Please keep the following points in mind when completing this assignment:

- Parameters of type `char *` will point to the starting address of the string.
- Do not modify the given function labels in the assembly files.
- Do not modify the given text in `testCases.txt`.
- You must preserve caller registers if you use them in your assembly function.
- Please only use R0 - R10 as general-purpose registers for this assignment.

7 README

In addition to your code, please include a README file which contains your answers to the following questions:

1. Given the following piece of C code: `char * str = ‘‘abcd’’;`
   Suppose `str` points to memory address `0x50`. Show what values are laid out in memory from memory address 0x50 to 0x54 after the code runs. Additionally, explain why a character needs a byte of memory.
2. Explain how you would concatenate a C-string `input2` to the end of another C-string `input1` in assembly. You do not need to provide code. Assume enough memory has been allocated to accommodate the length of the new string, and the starting address of the new string is the same as the starting address of `input1`.
3. Explain what information a function needs to preserve as the callee.

8 Commenting and Style Guide

Every section of code, loop, or branch command needs to be commented, although you are free to include other comments. Lines of code should not exceed 80 characters. On including your name in files: To detect instances of academic integrity violations in programming assignments we may use 3rd party software. We recommend you only include your class lab account ID (not your name or PID) in your submissions. Including your name and/or PID will disclose that information to the 3rd party.

9 Handin

Commit and push the six files to the Github repository that was created for you by 11:59PM on Tuesday, October 31. You can push up to one day late for a 20% penalty. After you push, make sure to check on Github that the files are actually there; we will mark all of the repositories for grading a few minutes after midnight and grade precisely what is there. Your handin should include:

- Six assembly files (`strlen.s`, `strcmp.s`, `strtrunc.s`, `strrev.s`, `palindrome.s`, `strfind.s`)
- A single README.txt file that has answers to the open-ended questions above, as specified.

Note that you do not need to hand in the `.o` files.