# ECE 251:
# Project 1: Concatenate two strings

Minh-Thai Nguyen, Sophie Jaro

March 2019

## 1 Program Specifications

This program concatenates two strings inputted by the user. Each string has at most 10 ASCII characters. First, the program asks the user for first input string. If the first string entered exceeds 10 characters, error code 21 is returned. If the second string entered exceeds 10 characters, error code 22 is returned. If both input strings are less than or equal to 10 characters each, the strings are concatenated. To concatenate, the program stores the first input (one byte at a time) in the result array then the second input in the result array with an offset of the size of the first input. The concatenated string is outputted to stdout. An error code of the length of the resulting concatenated string is returned on an error free run.

## 2 Architecture

To tackle this problem, the program first allocates memory to store the inputted strings and the resulting concatenated string. The result is given 21 bytes to store a maximum of 10 characters per input and one null character to terminate it. An excessive number of bytes is allotted to the inputs so that the program does not have a segmentation fault. However, if the length of the input is over 100 bytes, a seg fault error will occur. However, under reasonable conditions, this error should never occur. To concatenate the two strings, the program copies the characters from the first input into the result array byte by byte, or character by character. For the first word, the offset from the zeroth element for the desired character is the same for both the input and the output so a common iterator is used. For the second word, the offsets are not the same for the input and output, as the offset for the result is equal to the length of the first word plus the offset of the second word. In this case, the program uses two offsets: the result offset, which retains its value from the first loop (i.e. still points to the end of the char array), and the input offset. In both cases, however, the offsets are all incremented at the same time. To copy the character, the byte at the address of input_word plus the offset iterator is loaded into a register. For

example, in loop1 this looks like:

```
ldrb r2, [r0,r5]
```

where r2 is the register that stores the character, r0 is the address of the input_word, and r5 is the offset of the input.

The end of an input string is signified by a null character, so the loops run until either the character limit is reached or a null character is reached. If either input offset reaches 10 without encountering a null character, the loop will call its appropriate error function, which returns an error code and exits the program. If a null character is reached, the program progresses to the next function. For example, in loop2 this control sequence looks like this:

```
cmp r2, #0x00   /* Compares r2 with the null character */
beq end   /* If they are the same end successfully */
cmp r1, #10    /* Compares r1 with 10 */
beq error2    /* Branches to error2 if the second input is invalid */
```

where r2 is where the input char is stored and r1 is the input offset iterator.

The program consists of seven functions of four different types. The main and init_loop2 functions serve to handle user input and instantiate the variables necessary for the two loops. The loop1 and loop2 functions serve to place a copy of the characters from the first and second inputs, respectively, into the output_msg. The copying is done through a character by character iteration through each of the char arrays. In addition to copying, the two loop functions also check for invalid length strings, calling the correct error functions. error1 and error2 serve as the error functions, exiting the program with the correct error code if either input string is invalid. The end function handles the exit of an error free program. It adds a null character to the end of the result char array, prints the concatenated string, sets the exit code equal to the character count, and exits the program.

# 3 Design Choices and Difficulties

Although the final code is mostly written in assembly, the scanf and printf functions from the C std library were used to handle input reading and to write to stdout. As an alternative to scanf, the two input strings for the program could be passed through the argv arguments, however, this method was not used mainly because it goes against design specifications and is also imperfect (doesn't handle '\' very well). Two loop functions were written instead of one common loop function because the loops were very short and two different loops provide more clarity to a reader. In addition, the branch handling that would accompany a common loop would also be longer than the second loop function. The length errors were checked during the copying process because it saved

computational time and shortened the amount of instructions in a successful case.

Throughout the project there were various hurdles. The first difficulty arose from a lack of information. While trying to work on the project a couple weeks early, we were unable to figure out why some registers would lose their expected values when the printf function was called. We learned soon after that the registers 0-3 were volatile when calling a function. When debugging with gdb, we were unable to figure out why characters were not being stored in the result array, only to find out that they were being stored right and that it was not printing because we were misusing the print function in gdb. Other than these errors, there were not many other noteworthy hardships.