Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
3/11/21

# Computer Architecture ECE-251: Project 1

**Requirements:** For project #1, you will ask the user to input 2 strings and concatenate the two. Both strings will have at most 12 ASCII characters. You will ask the user to enter the first string after which they will hit enter. If the number of characters exceeds the 12-character limit, you will print an error and exit the program with a 7-return code. If the number of characters is less than or equal to 12 characters, you will ask them to enter another string. If the second string exceeds the character limit, your program should print an error message and exit with an 8-return code.

If both strings meet the requirements, you will then output the concatenated string to stdout and return an error code that consists of the total number of characters in the concatenated string.

## Overall Architecture of Program

After declaring all the initial variable in the .data section, a loop is used to cycle through, count and store the characters of the first string using the getchar function. The loop will stop either when the input ends, or when the 13$^{th}$ character is reached. (if it hits the 13$^{th}$ character, it will jump to the end and exit with the proper return code. A similar process is used for the 2$^{nd}$ string. The strings are concatenated by copying the second string to the end of the first string and printing the result. Finally, the return code is calculated by adding the lengths of the two strings, which were determined the loops above.

### .data section

In the .data section, all the memory and variables that will be used throughout the program is initialized. The variables initialized are:

- message1: a null-terminated string with a message to be provided to the user to ask for the first input.
- message2: a null-terminated string with a message to be provided to the user to ask for the second input.
- message3: a null-terminated format string to be used for printing the final concatenated message with printf.
- messageTooLong: a null-terminated string with a message to be provided to the user if the input given by the user is too long.
- string1: a 24-character long string that will be used to store the full given string.
- string2: a 12-character long string that will be used to store the second given string.
- tmpStore: a byte in memory used to store temporary values throughout the software.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
3/11/21

- return: a byte in memory used to store the required return code.

**Main**

The address of the first message is loaded into r0 which is then used an input to the printf function. We chose printf because it is the most versatile function that can print a string of character to standard output. Furthermore, it does not append the string with a new line character, and thus allows the input of the string to be on the same line as the output.

Then, the program prepares to enter the loop. r0 is set to 0, the address of 'tmpStore' is loaded into r1, and then that value in r0 (i.e. 0) is placed in 'tmpStore.' This value is used for counting the number of characters in the inputted string.

**read1**

The read1 label is essentially a loop that counts the number of characters in the user's first string.

We leveraged the libc function 'getchar' to individually obtain each character from the user input. The function 'getchar' stores each character in r0. Using the 'cmp' instruction, the contents of r0 (i.e. each character) are first compared to the ASCII character 10 which indicates a new line feed and in the event that the character is in fact a new line character, the code branches to end1. It was necessary to place this instruction at the beginning of the loop, because a new line character terminates the string. So, if the string is less than the maximum of 12 characters, the newline character causes the code to branch to end1 before continuing through read1 (which would inadvertently account for the newline character as a character in the string). This is done using the 'beq' instruction following the 'cmp' instruction.

If the character stored in r0 is not 10 (i.e. not a new line character), the code continues linearly. The address of the user's first string input (string1) is loaded into r1 using the 'ldr' instruction. The address of 'tmpStore' is then loaded into r2 and the contents of 'tmpStore' is loaded into r2 as well. From the previous label, "main", the first time the code runs through this instruction, the value at the address of 'tmpStore' is 0 so the value in r0 becomes 0 as well. However, after running through read1 at least once, the value in r0 is incremented.

Before incrementing the counter, the value of r0 is stored into the address of memory pointed to by r1, which, as mentioned earlier, is the address of string1. The purpose of this instruction is to re-build the string from the individual characters.

The counter is then incremented. This is accomplished using the load instruction to load the value at the address of 'tmpStore' into r0 and then adding "1" to that value in r0. This value is then stored at the address of 'tmpStore'.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
3/11/21
At this point, the value in r0 represents the number of characters that have been counted thus far. Using the 'cmp' instruction, the value in r0 is compared to 13. If the value in r0 is less than 13 the code branches to the top of read1 because of the 'blt' instruction. This is what creates the "loop" functionality of the read1 label.

If the value in r0 is greater than or equal to 13, (i.e. the number of characters for the string exceeds the maximum of 12) the code continues linearly to print "ERROR: Your message is too long." and exit with a 7 return code. The address of "messageTooLong" is loaded into r0 using the 'ldr' instruction. We then used the libc function "printf" to print the message to stdout. To attain a 7-return code, the number 7 is moved into r0 using the 'mov' instruction. Afterwards, before branching to the end, the address of return is loaded into r1 and the value of r0, which is 7, is stored in the address of return. The code then branches to the end.

**end1**

Then the length of the string (from 'tmpStore') is placed in 'address_of_return.' This is done by loading 'address_of_return' into r1 and 'tmpStore' into r2, and then value of 'tmpStore' into r2, and then that value into r1.

Next, the length end of the first string is calculated. This is done by adding the length of the string (found in 'tmpStore') to the address of the string (found in 'address_of_string1').

Finally, a /0 character is added to the end of the string to terminate it.

Now, code prepares to deal with the second string. It asks the user to enter the second 2nd, using printf (as described above) and resets 'tmpStore' to zero, since it will be used to count the number characters in second string. This is done by moving 0 into r0, 'address_of_tmpStore' into r1, and then store r0 (the number zero) into the address listed in r1 ('tmpStore')

**read2**

The read2 label is essentially a loop that counts the number of characters in the user's second string.

We leveraged the libc function "getchar" to individually obtain each character from the user input. The function "getchar" stores each character in r0. Using the 'cmp' instruction, the contents of r0 (i.e. each character) are first compared to the ASCII character 10 which indicates a new line feed and in the event that the character is in fact a new line character, the code branches to end2. This is done using the 'beq' instruction following the 'cmp' instruction.

If the character stored in r0 is not 10 (i.e. not a new line character), the code continues linearly. The address of the user's second string input (string2) is loaded into r1 using the ldr instruction. The address of "tmpStore" is then loaded into r2 and the contents of "tmpStore" is loaded into

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
3/11/21
r2 as well. From the previous label, “main”, the first time the code runs through this instruction, the value at the address of “tmpStore” is 0 so the value in r0 becomes 0 as well. After the read2 loop runs through at least once, however, the value in r0 is incremented.

Before incrementing the counter, the value of r0 is stored into the address of memory pointed to by r1, which, as mentioned earlier, is the address of “string2”. The purpose of this instruction is to re-build the string from the individual characters.

The counter is then incremented. This is accomplished using the load instruction to load the value at the address of “tmpStore” into r0 and then adding “1” to that value in r0. This value is then stored at the address of “tmpStore”.

At this point, the value in r0 represents the number of characters that have been counted thus far. Using the ‘cmp’ instruction, the value in r0 is compared to 13. If the value in r0 is less than 13 the code branches to the top of read1 because of the ‘blt’ instruction. This is what creates the “loop” functionality of the read1 label.

If the value in r0 is greater than or equal to 13, (i.e. the number of characters for the string exceeds the maximum of 12) the code continues linearly to print “ERROR: Your message is too long.” and exit with an 8 return code. The address of “messageTooLong” is loaded into r0 using the ‘ldr’ instruction. We then used the libc function “printf” to print the message to stdout. To attain an 8-return code, #8 is moved into r0 using the ‘mov’ instruction. Afterwards, before the code continues to the end label, the address of return is loaded into r1 and the value of r0, which is 8, is stored in the address of return.

**end2** (strcpy, concatenation of strings)

The address of the second string and the length of it is added together to find the end of the second string. The address of the end of the second string is used to null terminate the second string. This is done by loading the ‘address_of_string2’ into r1, and the ‘address_of_tmpStore’ is loaded into r2 which currently stores the length of ‘string2’. The value of r2 is made to be the value of ‘tmpStore’ by then loading the value at the address of r2 into r2.

After this, the address of the end of the first string is found by adding the length of the string to the address of the first string. At this point, strcpy is called with the end of the first string and the address of the second string to copy the second string onto the end of the first string in memory. This is done by loading the ‘address_of_string1’ into r0 and the ‘address_of_return’ into r1 which currently stores the length of ‘string1’. The value at the address that is stored in r2 is then loaded into r2. r0 and r1 are then added together and put into r0. Then, the address of string2 is loaded into r1 and strcpy called.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
3/11/21
The length of the string is then calculated by adding the length of the second string onto the length of the first string. This is done by loading the 'address_of_return' into r1 which currently stores the length of string1 and loading the 'address_of_tmpStore' into r2 which currently stores the length of string2. The values stored at the address referenced in r1 and r2 are then loaded into r1 and r3 respectively. After this, the values in r2 and r3 are added then stored into r3. The value in r3 is then stored into the address referenced in r1.

At this point, the concatenated string is printed out by loading the address of the template known as 'address_of_string3' for printing it into r0 and 'address_of_string1' into r0 which now stores the concatenated string then calling printf.

**end**

At this point, the program exits with the value stored at the address of return. This is done by loading the 'address_of_return' into r0 then loading the value at the address contained in r0 into r0. After this, 1 is put into r7 then the exit system call is made with 'swi 0'.

**Challenges/Original Versions**

Originally, we tried to use scanf in order to take input from the user. We ran into several issues with this approach. At first, the scan format we tried to use was "%s". "%s" worked fine as long as the input contained no spaces. In an attempt to resolve this issue, we tried using "%[^/n]" instead. This would match the string until a new line which resolved the spaces issue. This created another issue though, if the user inputted more than 12 characters as we could end up writing more to memory then the space allotted for the string. To make sure that all cases would be handled without a crash, we switched to using 'getchar' instead to take in the user's input one character at a time. This allowed us to read until 12 characters and if the user tries to input more than 12, we can stop taking input and quit with the appropriate exit code and error message.

Instead of using "bx lr" at the end of our code to branch out, we used the "swi" instruction. Initially, we tried to end the program with the following:

**ldr lr**, **address_of_return**

**ldr lr**, [**lr**]

**bx lr**

However, when the program was compiled and executed, it would perform everything properly, but exit with a segmentation fault error. After some research, we discovered that it was possible the lr was overwritten. We found that 'swi' is a software interrupt instruction that performs a system call. So, we used the syscall number for exit to exit the program without a

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
3/11/21
segmentation fault. This was accomplished by moving #1, the syscall number for exit(), into r7 while moving the necessary return code into r0 and making the syscall with 0.

(https://jumpnowtek.com/shellcode/linux-arm-shellcode-part1.html)

(https://azeria-labs.com/writing-arm-shellcode/)