Lab 3: String Manipulation and Memory Addressing

Joseph Salazar
College of Engineering
University of Miami
Miami, United States
jes409@miami.edu

***Abstract*— The ARM assembly language is widely used by professionals in the computer engineering field. However, there is very little material that introduces entry-level persons into assembly language. To help individuals better understand assembly for their futures in engineering, we propose following a quick and efficient program provided by the University of Miami's College of Engineering. This program allows individuals to practice assembly by manipulating strings and observing memory addresses.**

Keywords—ARM, assembly,

I. Introduction

The purpose of this lab is to interact and familiarize oneself with the ARM processor and Altera Monitor Program. By using prior knowledge about the components of the DE1 board, one can now begin to write and compile code using the processor. To achieve a better understanding of how to navigate memory and data in the ARM processor, one is going to manipulate a string by removing the spaces. Along with this, one will also utilize subroutines to make their code as efficient as possible. Finally, one will be asked to reverse the string given a condition. By making these changes, one can start to make certain connections to information discussed preceding this lab. By the end of this exercise, one should leave with a better understanding of ARM commands to help them excel in their future as a computer engineer.

II. Method/Experient

A. Overview

To perform this lab, one should have Altera Monitor Program installed before continuing. Then, power on the De1 board and connect it to the computer. Once connected, one should proceed to set-up their coding environment/document so that the code remains organized. After one's ".s" file is created, the programmer should include the data, global, text, start, and end directives to ensure compilation goes smoothly after code has been added. If one is not familiar with this routine, referencing the DE1 manual would be most beneficial [2].

To begin this lab, one is asked to store the string "I am a cane" in memory. At this point one should be able to observe where there string is stored. After observing this, the programmer is asked to write code to remove the spaces in the string. This can be achieved by referencing an ASCII to hexadecimal chart and utilizing a compare directive. This can be seen in Figure 1.

Next, the programmer is asked to make their code more efficient. If the original code did not include subroutines, it is now time to implement them. This step ensures that the programmer knows how to utilize loops correctly and efficiently in ARM assembly.

Finally, the programmer is asked to copy the string backwards if the first character is lower case, a curly bracket, or a tilde. To achieve this, the programmer should be confident in their previous loop and understand how to update pointers. The final code should interpret the first character, then decide whether to copy the string forward or backward. This can be observed in Figure 2 and Figure 3.

B. Results

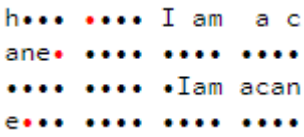

Figure 1. The following figure shows a forward copy without spaces of the string.

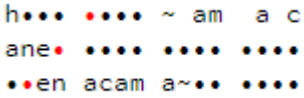

Figure 2. The following figure shows how the program should reverse the string without spaces given the first character is a tilde.

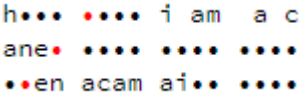

Figure 3. The following figure shows how the program should reverse the string without spaces given the first character is a lowercase.

C. Discussion

After completing the previous tasks, one could reflect and take many lessons away. These exercises emphasize the importance of knowing how to navigate through memory, manipulate strings, and manipulate pointers. If the programmer does not have a full grasp on this knowledge, these exercises will be an indicator of how much one really knows.

While this is a thorough and well-thought exercise, it does not come without its limitations. For example, it is very difficult to optimize you code from part one to part two if one had already utilized subroutines. It is at this point that the practice really shows a programmer just how efficient one could make their code. While this may be difficult to perform, it is

University of Miami

important to understand how to optimize your code to make future endeavors easier to follow.

III. Conclusion

In sum, this exercise introduces instructions in ARM assembly that are essential for more difficult programs. After following the instructions, the programmer will have walked away with a deeper understanding of subroutine instructions, as well as how to manipulate and navigate memory. While the instructions are not completely full, the information that is left out ensures that the programmer does his/her research before attempting the exercise.

References

[1] ARM Limited. ARM Architecture Reference Manual. 1996.

[2] *Documentation–ArmDeveloper*, developer.arm.com/documentation/ddi0388/latest/.

[3] "GNU Manuals Online - GNU Project - Free Software Foundation." *[A GNU Head]* , www.gnu.org/manual/manual.en.html.

[4] Pyeatt, Larry D. *Modern Assembly Language Programming with the ARM Processor*. Newnes, 2016.

Questions from the Lab

How many instructions were saved if any from part one to part 2?

-About 2 extra instructions as well as making the code easier to follow.

Code Used

```
@**********************Part One*****************************
.data
msg:    .asciz          "I am a cane\n"
cop:    .space          60

.text
.global _start
_start:
        ldr r0,=msg             @address of msg
        ldr r1,=cop             @address of double
first:
        ldrb r2,[r0],#1 @copy char of msg
        cmp     r2,#0x20        @compare character to space
        bne     nul                     @if not equal, check null, then copy or end
        beq     first           @if equal, check next

second:
        strb r2,[r1],#1     @store character to duplicate
        b       first

nul:
        cmp r2,#0x00    @compare char to null
        beq     done            @if null, end
        bne     second          @if not equal, str the char

done:
        b       done
.end
@**********************Part 2*****************************
.data
msg:    .asciz          "I am a cane\n"
cop:    .space          60

.text
.global _start
_start:
        ldr r0,=msg //address of msg
        ldr r1,=cop //address of double
first:
        ldrb r2,[r0],#1 //copy char of msg
        cmp r2,#0x20 //compare to a space
        beq first //if equal check next
        strb r2,[r1], #1 //store char to duplicate
        cmp r2, #0x00 //compare to null
        beq done
        bne first
```

```
done:
        b done
.end
@**********************Part 3*****************
.data
msg:    .asciz          "~ am a cane\n"
                .skip           20
cop:    .space          60

.text
.global _start
_start:
        ldr r0,=msg             @address of msg
        ldr r1,=cop             @address of double

reqs:
        ldrb r2,[r0]
        cmp r2,#0x60    @hex vals greater than this include lowercase and symbols
        bge rev                 @if first char is hex equiv of 60 or more, reverse it
        ble first       @if less, copy like normal

first:
        ldrb r2,[r0],#1 @copy char of msg
        cmp     r2,#0x20        @compare character to space
        bne     nul                     @if not equal, check null, then copy or end
        beq     first           @if equal, check next

second:
        strb r2,[r1],#1 @store character to duplicate
        b       first

nul:
        cmp r2,#0x00    @compare char to null
        beq     done            @if null, end
        bne     second          @if not equal, str the char

rev:
        ldrb r2,[r0],#1 @give r2 a char
        cmp r2,#0x20    @compare it to a space
        beq rev                 @if its a space, restart process
        bne     nulrev          @if not a space, check null

nulrev:
        strb r2,[r1],#-1        @store char, but move pointer backwards
        cmp r2,#0x00            @compare char to null
        bne rev                         @if not null, keep checking
        beq done                        @if it was null, end

done:
        b       done
.end
```