ECE 212: Microprocessors, Hardware, Software, and Interfacing

Lab 3: Getting Started with DE1-SoC board

Karysse Hay
College of Engineering
University of Miami
Coral Gables, United Sates
kdh85@miami.edu

***Abstract*— ARM assembly language is a low-level language that many professionals in the engineering field use. To better understand the specific components of ARM programming, such as data movement, arithmetic, code control, etc. This lab also aims to teach students how to manipulate strings and memory addressing in assembly**

***Keywords*— assembly language, data movement, arithmetic, code control, memory addressing**

I. CHALLENGE

This lab is important as it teaches us the different ARM instructions through programming assembly code. It will allow for a better understanding of how to navigate memory and data in the ARM processor. To do this, one begins by storing a message in memory and manipulating it in various ways using different ARM instructions. The changes to the message include storing memory by reference, changing the address of where it is stored, and aligning it. This was facilitated by the knowledge gained in the previous labs and in class. In the end, one should leave with a better understanding in ARM programming and string manipulation.

II. APPROACH

The provided lab manual the arm cortex a-9 manual [1] was looked at beforehand. For part one and two, the 'ldrb' instruction would be used to read in the characters from the source message and store it in a register [1]. To copy the string without spaces, a destination register would be needed and to store the message in the destination register, 'strb' instruction would be used [1]. The pointer to the character would need to be incremented each time one was read. A loop would be created using the branch instruction and the contents of the memory and the registers would be inspected after each part to ensure the lab is working correctly.

III. EXPERIMENT SETUP

A. Overview

The AMP was launched, and a new project was created. A text file named 'Lab3.s' was added to the project to facilitate altering of the code. Assembly code for each part of the lab will be inserted in the results section. When problems occurred, the program was debugged by checking the memory tab.

B. Part I

For Part I, the phrase "I am a Cane" had to be copied from one memory location to another but without the spaces. For this, a destination register had to be created with a different address. The load by byte instruction was utilized to load character by character. The character was then compared with the ASCII value of a space and if it was found to be a loop, it would branch back to the load instruction so that it was ignored. If it was not a space, it would store the character in the destination register and increment by one for the next letter. The code was downloaded to the board. The code can be seen below:

```
.global _start
_start:

.data

x: .asciz "I am a Cane"

ldr r1, =x
mov r2, #0x000000F0 @ destination address

loop: ldrb r3, [r1], #1


check: cmp r3, #0x20
          beq loop
          cmp r3, #0x00  @null
          beq end
          bne store

store: strb r3, [r2], #1
b loop

end: b end
       .end
```

Figure I: Assembly Code for Part I

C. Part II

For Part II, it was required that the initial code be optimized completely. To do this, a single compare instruction was removed as it was not necessary for the sequence of code. This was again downloaded to the board and results were recorded. The code can be seen below:

```
.global _start
_start:

.data

x: .asciz "I am a Cane"

ldr r1, =x
mov r2, #0x000000F0 @ destination address

loop: ldrb r3, [r1], #1


check: cmp r3, #0x20
beq loop
cmp r3, #0x00  @null
beq end

store: strb r3, [r2], #1
b loop

end: b end
.end
```

Figure II: Assembly Code for Part II

D. Part III

Part III required the program to print the string either forward or backward depending on its first character. If the first character was a lower-case letter, curly bracket or a tilde, the program should copy the string backwards, else it should be copied forward.

To do this, the first character had to be checked initially and compared to the characters listed above. If one of the listed was found to be a match, the code would branch to the 'backward' instruction. At this point, the pointer had already been moved to the end of the message. This was done by manually counting the number of characters. It would then copy the last character and decrement the pointer by one.

The 'forward' instruction remained the same as Part II. It should be noted that each character was still checked against the ASCII value for a space and also null, to eliminate the occurrence of error.

The code can be seen in Figure III.

```
.global _start
_start:
.data
x: .asciz "eI am a Cane"

ldr r1, =x
mov r2, #0x000000F0 @ destination address

ldr r5, =x
add r6, r5, #11

ldrb r10, [r1]

check1: cmp r10, #0x61 @a
            bge check2
            cmp r10, #0x7E @tilde
            beq backward
            cmp r10, #0x7B @{
            beq backward
            cmp r10, #0x7D @}
            beq backward

check2: cmp r10, #0x7A @z
            ble backward

forward: ldrb r3, [r1], #1

check3: cmp r3, #0x20
            beq forward
            cmp r3, #0x00  @null
            beq end

strb r3, [r2], #1
b forward

backward: ldrb r7, [r6], #-1
            b check4

check4: cmp r7, #0x20
            beq backward
            cmp r7, #0x00  @null
            beq end

strb r7, [r2], #1
b backward

end: b end
     .end
```

Figure III: Assembly Code for Part III

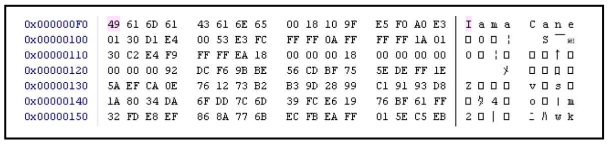
IV. Results and Analysis

A. *Overview*

Part III required the program to print the string either forward or backward depending on its first character. If the first character was a lower-case letter, curly bracket or a tilde, the program should copy the string backwards, else it should be copied forward.

B. *Part I Results and Analysis*

a) Results



```
0x000000F0  49 61 6D 61  43 61 6E 65  00 18 10 9F  E5 F0 A0 E3  I a m a  C a n e
0x00000100  01 30 D1 E4  00 53 E3 FC  FF FF 0A FF  FF FF 1A 01  □ 0 □ ¦    S ¯ 
0x00000110  30 C2 E4 F9  FF FF EA 18  00 00 00 18  00 00 00 00  0 □ ¦ □  □ □ ↑ □
0x00000120  00 00 00 92  DC F6 9B BE  56 CD BF 75  5E DE FF 1E        ﾒ  □ □ □ □
0x00000130  5A EF CA 0E  76 12 73 B2  B3 9D 28 99  C1 91 93 D8  Z □ □ □  v □ s □
0x00000140  1A 80 34 DA  6F DD 7C 6D  39 FC E6 19  76 BF 61 FF  □ ﾀ 4 □  o □ | m
0x00000150  32 FD E8 EF  86 8A 77 6B  EC FB EA FF  01 5E C5 EB  2 □ | □  ﾆ ﾊ w k
```

Figure IV: Memory Address for Part I

b) Comments

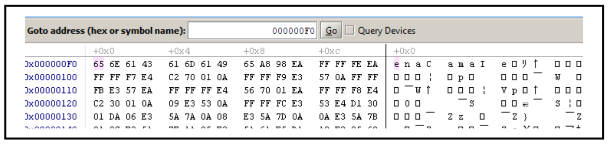
The destination address selected was 0x000000F0. It can be seen in Figure IV, the message 'I am a Cane' was successfully copied with no spaces in the correct destination address. The destination address was chosen based on its divisibility by four. This was done so that the message would sit on one line rather than two or more.

C. *Part II Results and Analyis*

A single line was removed from Part I's code as it was optimized already to a certain extent. The line 'bne store' was excluded as the program would already have gone to the next instruction if the first two compares were not matched. It yielded the same results and at the same address therefore the repetition of the photo above will be excluded.

D. *Part III Results and Ananlysis*

a) Results



```
Goto address (hex or symbol name): 000000F0  Go  Query Devices

            +0x0         +0x4         +0x8         +0xc         +0x0
0x000000F0  65 6E 61 43  61 6D 61 49  65 A8 98 EA  FF FF FE EA  e n a C  a m a I  e □ ﾘ ↑  □ □ □
0x00000100  FF FF F7 E4  C2 70 01 0A  FF FF F9 E3  57 0A FF FF  □ □ □ ¦  □ p □    □ □ □ ¯  W   □
0x00000110  FB E3 57 EA  FF FF FF E4  56 70 01 EA  FF FF F8 E4  □ ¯ W ↑  □ □ □ ¦  V p □ ↑  □ □ □
0x00000120  C2 30 01 0A  09 E3 53 0A  FF FF FC E3  53 E4 D1 30  □ 0 □      ¯ S    □ □   ¯  S ¦ □
0x00000130  01 DA 06 E3  5A 7A 0A 08  E3 5A 7D 0A  0A E3 5A 7B  □ □ □ ¯  Z z   □  ¯ Z }      ¯ Z
```

Figure V: Memory Address for Part III

b) Comments

The message was changed from "I am a Cane" to "eI am a Cane" so that the program could have been checked if it worked. Initially, the 'check' loops were nested within each other which was not working. To fix this, it was decided that the best way to do this was to check the first character outside of the 'check' loops so that it could identify right away if the message had to be printed forward or backward. A loop that contained the checking of all the special characters was kept separate from the 'backward' loop. For the lower-case letter, the character was checked to see if it was ASCII value 61 or greater (letter a) and if it was, it would branch to a separate loop that checked to see if it was less than ASCII value 7A (letter z) [3]. This allowed for the first character to be checked to see if it laid in the range of lower-case a to lower case z.

Figure V shows the message printed backwards at the same destination address. The code was successful in checking if the first character was in the lower-case range of 61-7A and stored the message backwards thus showing "enaCamaIe" at the destination address chosen [3].

V. Conclusion

This lab allowed for a better understanding of how to code a loop with conditional statements as parameters. As a result of the number of conditions and comparing that had to be done, it can be said that the students gained a better insight of the 'cmp' and 'b' instructions [1]. Finally, the manipulation of strings and memory addresses was better understood upon the completion of this lab.

References

[1] A. Holdings, "Cortex-a9 technical reference manual, revision r2p0," ARM Holdings. Report number: DDI0388E, 2009.

[2] ARM Ltd and ARM Germany GmbH, " Writing ARM Assembly Language," Assembler User Guide: Writing ARM Assembly Language. [Online]. Available: https://www.keil.com/support/man/docs/armasm/armasm_dom1359731144635.htm. [Accessed: 20-Mar-2021].

[3] "ASCII," Wikipedia, 19-Aug-2020. [Online]. Available: https://simple.wikipedia.org/wiki/ASCII#/media/File:ASCII-Tablewide.svg. [Accessed: 20-Mar-2021].