INTRODUCTION

Lab 1 introduces us to the fundamentals of ARM assembly language programming, using the THUMB2 instruction set which is supported by the ARM microprocessor. The primary objective for this lab is to learn assembly language and gain hands-on experience with the NUCLEO-L432KC development board [1]. We were granted access to the STM32CubeIDE software for the programming and debugging of our assembly code.[1] The lab consisted of two parts, Part A and Part B. Part A focused on converting an ASCII character to its hexadecimal value, and Part B consisted of converting an ASCII character to its uppercase or lowercase representation. Both programs are required to reject invalid ASCII characters by returning either -1 for part A, or * for part B into the corresponding memory location. Both programs are required to exit when the ASCII “Enter” code is detected. With this lab will come a greater understanding of low-level programming, how data is processed in memory, assembly language instructions, and control of general program flow.

DESIGN

Part A of this lab focuses on writing a program that converts an ASCII character to its hexadecimal equivalent. The valid ASCII characters for this lab are 0-9, A-F, and a-f. Each memory location on the board contains one byte of information, and each ASCII character is one byte so each memory location would be able to hold one ASCII character. However, for this lab, we instead use four memory locations to store one ASCII character, while padding the unused storage with zeros.[1]
The program starts with reading ASCII characters from memory, starting at address 0x20001000. Then we process this data, and write the corresponding hexadecimal value to memory location starting at 0x20002000 and incrementing by 4 bytes. Invalid characters are assigned the error code -1.[1]

To design this program, we first make a flowchart as shown in Figure 1. First, we load the ASCII values from memory, then we check immediately if the value is ASCII “Enter”, if this is the case then we terminate the program. The next sequence is simply comparing the values to those of the ASCII table, and performing the corresponding operation. To convert from an ASCII decimal to hexadecimal, we just subtract 48 from the ASCII value. For example, the ASCII value for “1” is 49, so to get to the hexadecimal value, we simply do 49 - 48 = 1. We do a similar idea for capital letters; “A” is represented as 65, so we subtract 55 to get 10, which is the decimal representation of A in hexadecimal. Likewise, “a” is represented as 97, so we subtract 87 to get 10, which is the decimal representation of a in hexadecimal. It is worth noting that in order to include the upper bounds (9, F, f) we have to check if the ASCII value is less than the value next to the upper bound. The ASCII value directly after 9 is “:”, so we check if the ASCII value is less than this. The upper bound for “F” and “f” is simply “G” and “g” respectively. The code implementation of this flowchart can be found in Appendix [1].

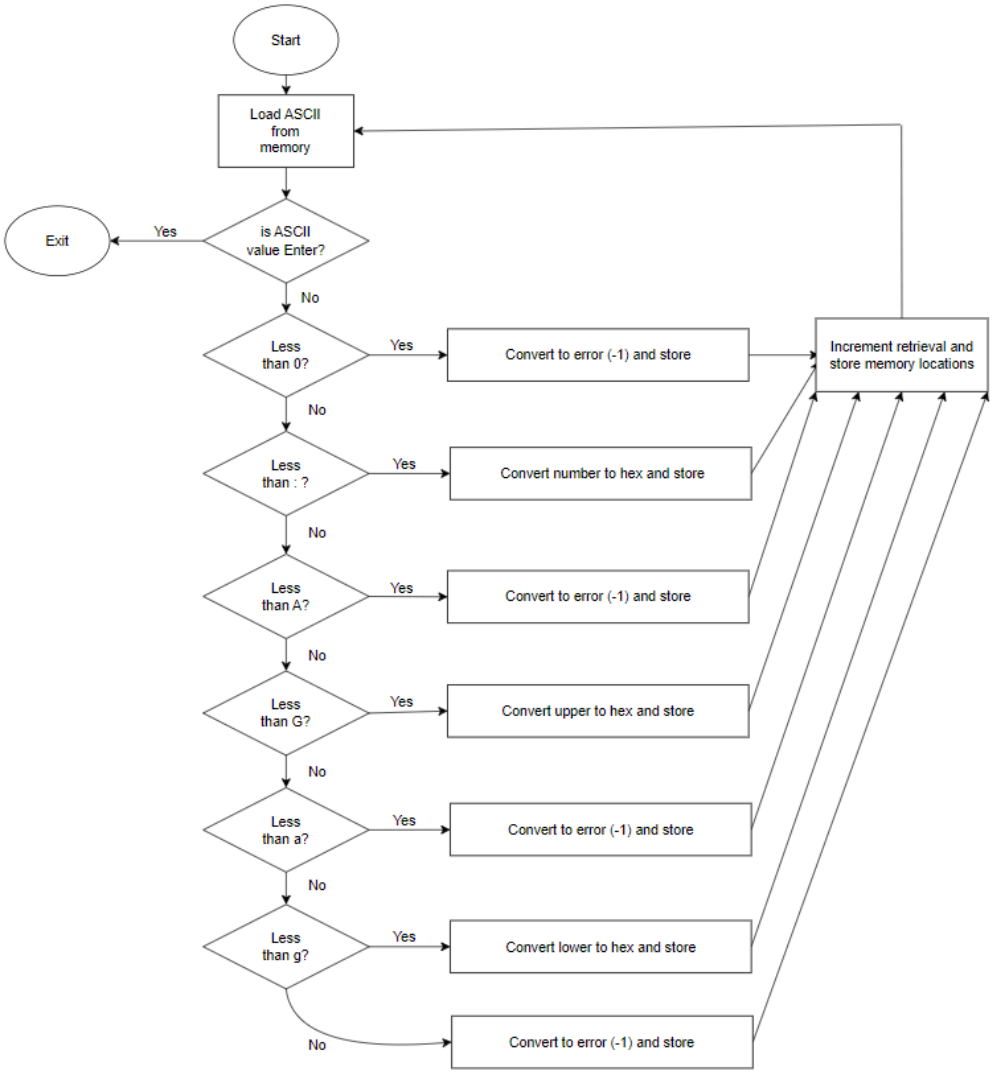

Figure 1: Flowchart for Part A

Part B of this lab focuses on writing a program that converts an ASCII character from its current case to its opposite case (lowercase to uppercase and vice versa). The valid ASCII characters for this lab are A-Z and a-z. Each memory location on the board contains one byte of information, and each ASCII character is one byte, so each memory location would be able to hold one ASCII character. However, just like in Part A, we instead use four memory locations to store one ASCII character, while padding the unused storage with zeros.

The program starts with reading ASCII characters from memory, starting at address 0x20001000. Then we process this data and write the corresponding uppercase or lowercase value to the memory location starting at 0x20003000, incrementing by 4 bytes. Invalid characters are assigned the error code '*'.[1]

To design this program, we first make a flowchart as shown in Figure 2. First, we load the ASCII values from memory, then we check immediately if the value is ASCII “Enter”, if this is the case then we terminate the program. The next sequence is simply comparing the values to those of the ASCII table, and performing the corresponding operation. To convert from uppercase to lowercase, we just add 32 to the uppercase ASCII value. For example, the ASCII value for “A” is 65, so to get the lowercase value, we simply do 65 + 32 = 97, which is the representation of “a” in the ASCII table. We do a similar thing for lowercase to uppercase, where we instead subtract 32 from the lowercase ASCII value. Although trivial, the calculation is again shown by the value “a” being 97, so we simply do 97 - 32 = 65, which is the representation of “A” in the ASCII table. It is worth noting that in order to include the upper bounds (Z, z) we have to check if the ASCII value is less than the value next to the upper bound. The ASCII value directly after Z is “[”, so we check if the ASCII value is less than this. The upper bound for “z” is “{”. The code implementation of this flowchart can be found in Appendix [1].

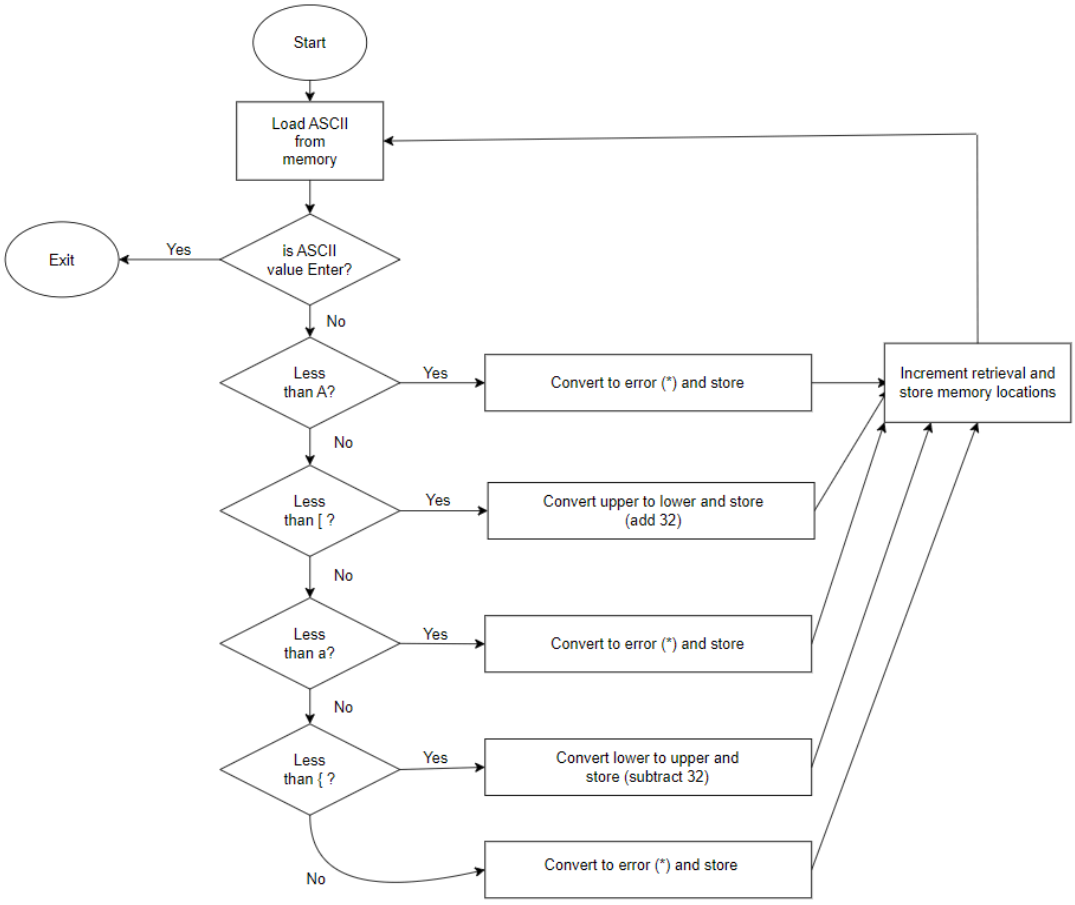

Figure 2: Flowchart for Part B

TESTING

To validate the functionality of the ARM assembly program, the testing strategy was to execute a predetermined set of test cases and verify the expected outcomes. The primary goal was to ensure the program converted ASCII characters into their hexadecimal or their decimal equivalent for Part A and for Part B, we needed to ensure that the program correctly performed the upper-case to lower-case and vice versa and outputs the correct error code for invalid characters. The testing was conducted through the STM32CubeIDE and the NUCLEO-L432KC board. The data sets we used to test the program were posted in the eClass [2] and contained a random array of characters. The inputs were manually stored in the memory location with the address 0x20001000 for both of the parts and the output was stored in 0x20002000 for Part A and 0x20003000 for Part B. We had 3 files containing 3 different random arrays of characters to test. All the test cases contained valid and invalid characters as well as the handling of the enter key (0x0D) to test the program stop. [2]

For Part A, we had valid ASCII characters (0-9, A-F and a-f) which were tested to confirm the program functionality. The testing included the program to correctly convert the valid ASCII characters to their equivalent hexadecimal or decimal values and then storing them in the memory location 0x20002000. According to the lab goal, all other ASCII characters such as F-Z, f-z and other symbols and spaces are invalid for this lab and so the testing also includes if the program correctly follows the logic of ruling out the invalid characters and stores an error code of -1 in case of invalid cases. [1]

For Part B, the program was tested to convert all the ASCII characters from A-Z and a-z to their upper-case or lower-case equivalent and for invalid characters such as 0-9 and other symbols and spaces, the error code of * was stored in the memory location of 0x20003000. [1]Additionally, the enter key was tested so that the program terminates correctly.

The observed results matched the logical outputs in all the test cases which confirms the correctness of the code. The program converted the valid range of ASCII characters to their equivalent hexadecimal values and successfully detected invalid ASCII characters and reported the error code -1 in their place for Part A. For Part B, the program correctly converted the valid ASCII characters to their upper-case or lower-case equivalent and also reported the correct error code of * for all invalid characters.

The structure of the testing procedure ensured that all the required functionality were implemented. The final program met all the lab requirements by correct conversion and correctly identifying the invalid inputs and reporting an error code.

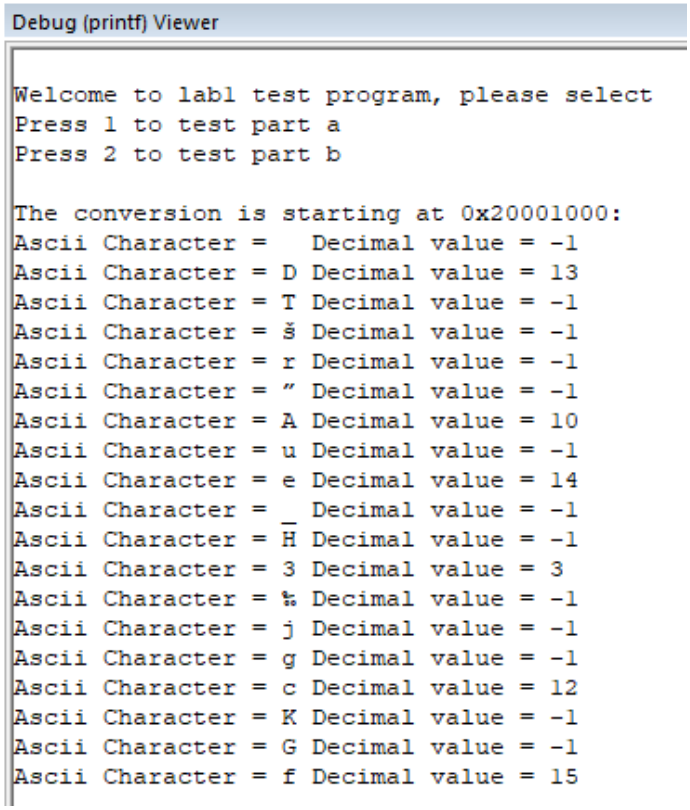

Figure 3: Example testing of Part A

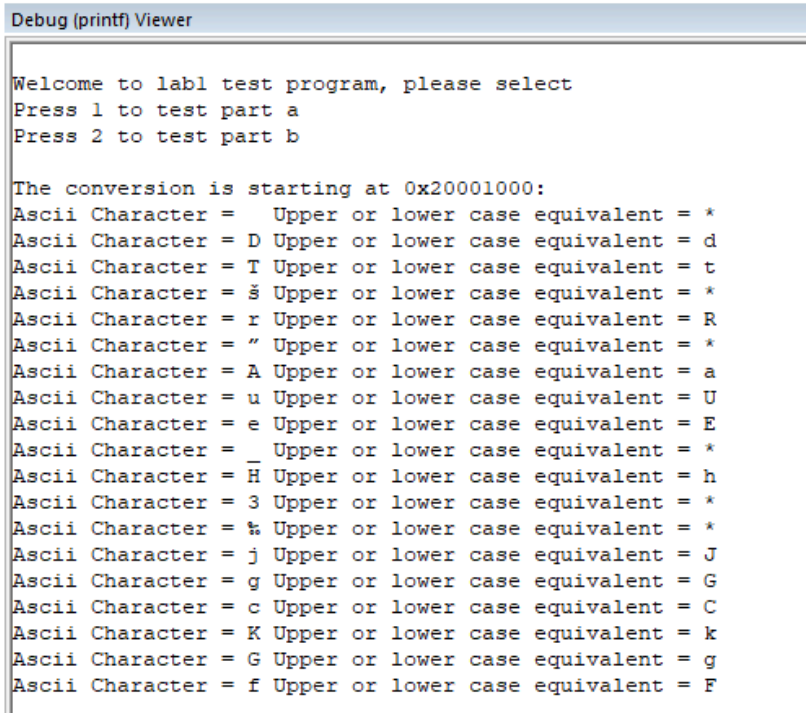

Figure 4: Example testing of Part B

QUESTIONS

1. What happens when there is no exit code ‘0x0D’ provided in the initialization process? Would it cause a problem? Why or Why not? How can we prevent this? Please give two options.

If there is no exit code in the initialization process, it will continue executing forever. This is because the loop structure is constantly being called on and it is designed to keep being called on until the exit code is detected. This could cause a problem as the program would keep reading memory further than the space we were looking at, leading to unpredictable values. One way to prevent this is by creating a search limit. We could set a value and if the program exceeds this value in iterations, then it terminates the program. This ensures that the program has a hard stopping point to prevent infinite loops. Another way to prevent this is by setting the last value to the exit code. We can initialize a memory location with the exit value, so when the program inevitably encounters this, it closes the loop safely.

2. Your program when compiled is stored as binary bits. How does the microprocessor separate where the instructions are stored and where the data is stored? Can you mix the two together in the same memory blocks?

The microprocessor has different memory locations that correspond to different tasks. For example, a certain part in memory is allocated for instruction, whereas a different part in memory is allocated for data. This allocated memory is known as memory segmentation. Technically you can mix the two together in the same memory blocks, but it usually isn't recommended. As for the Von Neumann machine, both instructions and data are in the same memory block, just a distance from each other. The downside is that there could be a potential unwanted execution of instructions if something goes wrong. The Harvard architecture machines however keep instructions and data in different locations to ensure that there is no potential unwanted execution.

CONCLUSION

This lab successfully implemented the ARM assembly program using the THUMB2 instruction set on the NUCLEO-L432KC board. The program correctly performs the required conversions and stores the results in the correct memory locations. The implemented assembly code systematically tests all the characters and implements the appropriate function of converting valid ASCII characters to their equivalent hexadecimal values, performing upper-case to lower-case transformations and vice versa with reporting appropriate error codes and closing the program when encountering the Enter Key.[1] The structured testing confirms the implementation of the code. Overall, the lab was a success in the correct execution of assembly instructions for data manipulation and memory operations.

APPENDIX

[1] Code for Part A

```
/* Code created by Ryan O'Handley (1883463) and Anurag Biswas Koushik
(1806274) */
/* This code converts ASCII characters into their hexadecimal
equivalent */

/* Preliminary setup*/
.global TestAsmCall
.global printf
.global cr
.syntax unified

.text
TestAsmCall:
PUSH {lr}
/*-------------------------------------*/

ldr r0, = 0x20001000 /* Loading the address we get values from */
ldr r2, = 0x20002000 /* Loading the address we store values to*/

/* Creating the main loop*/
loop:

/* Checking if the value at the address is "Enter"*/
ldr r1, [r0]
cmp r1, #13
beq done /* Exiting program if "Enter" is detected*/

/* Checking if the character is out of bounds, prompting invalid store
if so*/
cmp r1, #48
blo invalid

/* Checking if the character is a number between 0 and 9, prompting
number conversion if so*/
cmp r1, #58
blo number_conversion
```

```
/* Checking if the character is out of bounds, prompting invalid store
if so*/
cmp r1, #65
blo invalid

/* Checking if the character is uppercase, prompting uppercase
conversion if so*/
cmp r1, #71
blo upper_letter_conversion

/* Checking if the character is out of bounds, prompting invalid store
if so*/
cmp r1, #97
blo invalid

/* Checking if the character is lowercase, prompting lowercase
conversion if so*/
cmp r1, #103
blo lower_letter_conversion

/* Branch for storing invalid characters using the error code (-1) */
invalid:

mov r1, #-1
str r1, [r2] /* Setting the character to the error code*/

 /* Incrementing the memory addresses for the next value*/
add r0, #4
add r2, #4
b loop

/* Branch for converting ASCII number to hexadecimal number*/
number_conversion:
sub r1, #48
str r1, [r2] /* Storing the value after conversion*/

/* Incrementing the memory addresses for the next value*/
add r0, #4
add r2, #4
```

```
b loop

/* Branch for converting ASCII uppercase letter to hexadecimal
number*/
upper_letter_conversion:

sub r1, #55
str r1, [r2] /* Storing the value after conversion*/

/* Incrementing the memory addresses for the next value*/
add r0, #4
add r2, #4

b loop

/* Branch for converting ASCII lowercase letter to hexadecimal
number*/
lower_letter_conversion:

sub r1, #87
str r1, [r2] /* Storing the value after conversion*/

/* Incrementing the memory addresses for the next value*/
add r0, #4
add r2, #4

b loop

done:

/* Exiting the program*/
POP {PC}

.data
```

[2] Code for Part B:

```
/* Code created by Ryan O'Handley (1883463) and Anurag Biswas Koushik
(1806274) */
/* This code converts ASCII characters from their current case to
their opposite- */
/*case (upper to lower and vice versa) */

/* Preliminary setup*/
.global TestAsmCall
.global printf
.global cr
.syntax unified

.text
TestAsmCall:
PUSH {lr}
/*-------------------------------------*/

ldr r0, =0x20001000 /* Loading the address we get values from */
ldr r1, =0x20003000 /* Loading the address we store values to*/
ldr r3, =0x2A /* loading the error value '*' */

/* Creating the main loop*/
loop:

/* Checking if the value at the address is "Enter"*/
ldr r2, [r0]
cmp r2, #13
beq done /* Exiting program if "Enter" is detected*/

/* Checking if the character is out of bounds, prompting invalid store
if so*/
cmp r2, #65
blt invalid

/* Checking if the character is uppercase, prompting uppercase
conversion if so*/
cmp r2, #91
blo uppercase
```

```
/* Checking if the character is out of bounds, prompting invalid store
if so*/
cmp r2, #97
blt invalid

/* Checking if the character is lowercase, prompting lowercase
conversion if so*/
cmp r2, #123
blo lowercase

/* Branch for storing invalid characters using the error code*/
invalid:

str r3, [r1]

/* Incrementing the memory addresses for the next value*/
add r0, #4
add r1, #4
b loop

/* Branch for converting uppercase to lowercase*/
uppercase:

add r2, r2, #32
str r2, [r1] /* Storing the value after conversion*/

/* Incrementing the memory addresses for the next value*/
add r0, #4
add r1, #4
b loop

/* Branch for converting lowercase to uppercase*/
lowercase:

sub r2, r2, #32
str r2, [r1] /* Storing the value after conversion*/

/* Incrementing the memory addresses for the next value*/
add r0, #4
add r1, #4
```

```
b loop

done:

/* Exiting the program*/
POP {PC}

.data
```

REFERENCES

[1] "ECE 212 Lab 1: Nucleo," *ECE 212 Laboratory Manual*, University of Alberta. [Online]. Available: eClass Portal, University of Alberta. [ACCESSED: 28/02/2025]

[2] *Test Program Use*, University of Alberta. [Online]. Available: eClass Portal, University of Alberta. [ACCESSED: 28/02/2025].