Michael Savo, Sre Krishna

# LAB 5.1

# PROCESSING TEXT IN ASSEMBLY LANGUAGE

# Contents

# 1 Introduction

## 1.1 Lab overview

In this exercise, you will execute assembly code on the Nucleo-L552ZE-Q board using the debugger to examine its execution at the processor level.

If you haven't already, you and/or your partner must have Keil setup using the provided instructions

# 2 Learning Outcomes

- Write a mixed C program and assembly language subroutines for the microcontroller.
- Call the subroutines written in assembly in a C function.
- Use Arm register calling conventions when writing subroutines in assembly language.
- Use a suitable debugging tool to view and analyse the processor state.

# 3 Requirements

In this lab, we will be using the following hardware and software:

- **Keil µVision5 MDK IDE**
- **STM32 Nucleo-L552ZE-Q**
    - For more information, click here.

# 4 Software

## 4.1 Mixing Assembly Language and C Code

We will program the board in C but add assembly language subroutines to perform the string copy and capitalization operations. Some embedded systems are coded purely in assembly language, but most are coded in C and resort to assembly language only for time-critical processing. This is because the code *development* process is much faster (and hence much less expensive) when writing in C when compared to assembly language. Writing an assembly language function which can be called as a C function results in a modular program which gives us the best of both worlds: the fast, modular development of C and the fast performance of assembly language. It is also possible to add *inline assembly code* to C code, but this requires much greater knowledge of how the compiler generates code.

## 4.2 Main

First, we will create the main C function. This function contains two variables (a and b) with character arrays.

```
int main(void)
{
   const char a[] = "Hello world!";
   char b[20];

   my_strcpy(a, b);
   my_capitalize(b);

   while (1);
}
```

## 4.3 Register Use Conventions

There are certain register use conventions which we need to follow if we would like our assembly code to coexist with C code. We will examine these in more detail later in the module “C as implemented in Assembly Language”.

### 4.3.1 Calling Functions and Passing Arguments

When a function calls a subroutine, it places the return address in the link register lr. The arguments (if any) are passed in registers r0 through r3, starting with r0. If there are more than four arguments, or they are too large to fit in 32-bit registers, they are passed on the stack.

### 4.3.2 Temporary Storage

Registers r0 through r3 can be used for temporary storage if they were not used for arguments, or if the argument value is no longer needed.

### 4.3.3 Preserved Registers

Registers r4 through r11 must be preserved by a subroutine. If any must be used, they must be saved first and restored before returning. This is typically done by pushing them to and popping them from the stack.

### 4.3.4 Returning from functions

Because the return address has been stored in the link register, the BX lr instruction will reload the PC with the return address value from the lr. If the function returns a value, it will be passed through register r0.

## 4.4 String Capitalization

Let’s look at a function to capitalize all the lowercase letters in the string. We need to load each character, check to see if it is a letter, and if so, capitalize it.

Each character in the string is represented with its ASCII code (if you haven’t encountered this term before, please do a small amount of Google searching, AI asking, etc. to learn a bit about it, though you don’t need to memorize the table or anything like that at this point). For example, ‘A’ is represented with a 65 (0x41), ‘B’ with 66 (0x42), and so on up to ‘Z’ which uses 90 (0x5a). The lowercase letters start at ‘a’ (97, or 0x61) and end with ‘z’ (122, or 0x7a). We can convert a lowercase letter to an uppercase letter by subtracting 32.

The following code ***should*** work in theory…

```
__asm void my_capitalize(char *str)
{
cap_loop
   LDRB  r1, [r0]    // Load byte into r1 from memory pointed to by r0 (str
                        pointer)
   CMP   r1, #'a'-1  // compare it with the character before 'a'
   BLS   cap_skip    // If byte is lower or same, then skip this byte

   CMP   r1, #'z'    // Compare it with the 'z' character
   BHI   cap_skip    // If it is higher, then skip this byte

   SUBS  r1,#32      // Else subtract out difference to capitalize it
   STRB  r1, [r0]    // Store the capitalized byte back in memory

cap_skip
   ADDS  r0, r0, #1  // Increment str pointer
   CMP   r1, #0      // Was the byte 0?
   BNE   cap_loop    // If not, repeat the loop
   BX    lr          // Else return from subroutine
}
```

But in Professor Yett’s experience at least, it does not quite work as is. We’ll use the following instead (and you should use the above with the already commented code to help you understand what is happening):

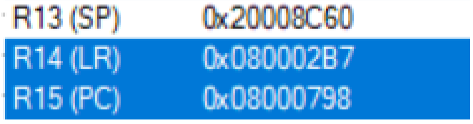
```
__attribute__((naked)) void my_strcpy(const char *src, char *dst)
{
    __asm(
        "loop: \n\t\
            LDRB  r2, [r0]  \n\t\
            ADDS  r0, #1    \n\t\
            STRB  r2, [r1]  \n\t\
            ADDS  r1, #1    \n\t\
            CMP   r2, #0    \n\t\
            BNE   loop      \n\t\
            BX    lr       \n\t\
    "
    );
}

__attribute__((naked)) void my_capitalize(char *str)
{
    __asm(
        "cap_loop: \n\t\
            LDRB  r1, [r0]   \n\t\
            CMP   r1, #'a'-1  \n\t\
            BLS   cap_skip   \n\t\
            CMP   r1, #'z'   \n\t\
            BHI   cap_skip   \n\t\
            SUBS  r1,#32     \n\t\
            STRB  r1, [r0]    \n\t\
        cap_skip:   \n\t\
            ADDS  r0, r0, #1  \n\t\
            CMP   r1, #0     \n\t\
            BNE   cap_loop   \n\t\
            BX    lr       \n\t\
        "
    );
}
```

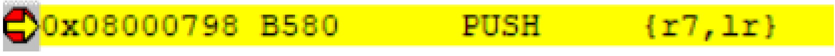
The code is shown above. It loads the byte into r1. If the byte is less than ‘a’ then the code skips the rest of the tests and proceeds to finish up the loop iteration.

This code has a quirk – the first compare instruction compares r1 against the character immediately before ‘a’ in the table. Why? What we would like is to compare r1 against ‘a’ and then branch if it is lower. However, there is no branch lower instruction, just branch lower or same (BLS). To use that instruction, we need to reduce by one the value we compare r1 against.

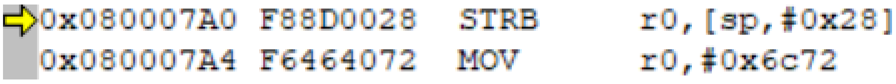
# 5 Lab Procedure

1. Compile the code.
2. Load it onto your board.
3. Run the program until the opening brace in the main function is highlighted. Open the Registers window (View->Registers Window) What are the values of the stack pointer (r13), link register (r14) and the Program Counter (r15)?
    a. 

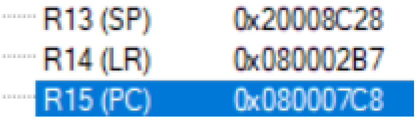
```
R13 (SP)     0x20008C60
R14 (LR)     0x080002B7
R15 (PC)     0x08000798
```

4. Open the Disassembly window (View->Disassembly Window). Which instruction does the yellow arrow point to, and what is its address? How does this address relate to the value of PC?
    a. 

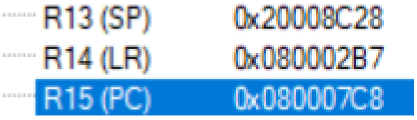
```
0x08000798 B580       PUSH      {r7,lr}
```

        i. It is the same value as PC
5. Step one machine instruction using the F11 (in older versions of Keil, this could be F10 instead) key while the Disassembly window is selected. Which two registers have changed (they should be highlighted in the Registers window), and how do they relate to the instruction just executed?
    a. R13 and R15 have changed. R13 changes because the code uses SUB on SP which is R13. There is an offnet of #0x30 and you can see the value at R13 changes according to that. R15 changes because the code executing makes the counter go up.
6. Look at the instructions in the Disassembly window. Do you see any instructions that are 4 bytes long? If so, what are the first two?
    a. I see STRB and MOV are both 4 bytes long.
    b. 

```
0x080007A0 F88D0028  STRB      r0,[sp,#0x28]
0x080007A4 F6464072  MOV       r0,#0x6c72
```

7. Continue execution (using F10) until reaching the BL.W my_strcpy instruction. What are the values of the SP, PC and lr?
    a. 

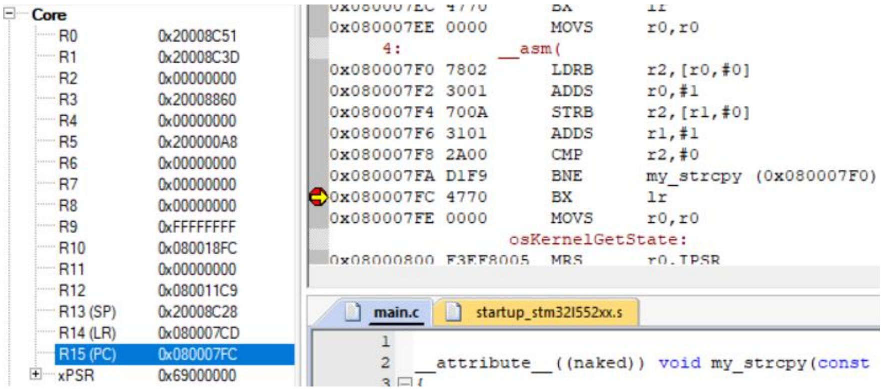
```
R13 (SP)     0x20008C28
R14 (LR)     0x080002B7
R15 (PC)     0x080007C8
```

8. Execute the BL.W instruction (continuing to go line by line with F11). What are the values of the SP, PC and lr? What has changed and why? Does the PC value agree with what is shown in the Disassembly window?
    a. 

```
R13 (SP)     0x20008C28
R14 (LR)     0x080002B7
R15 (PC)     0x080007C8
```

    b. The LR is the same as earlier and SP is different because it was used to store the values of r0 with an offset.
    c. The PC value does agree with the value shows in the Disassembly window.
9. Which registers hold the arguments to my_strcpy, and what are their contents?

   a. R0 is the source and R1 is the destination. R0 currently holds the contents of “Hello world!” while R1 should hold what makes up the phrase.

10. Open a Memory window (View->Memory Windows->Memory 1) for the address for src (as in the address that we are copying from!) determined above. Make sure you type the address in true hexadecimal (something like 0x20000000, but with the actual address of course)
11. Open a Memory window (View->Memory Windows->Memory 2) for the address for dst (as in the address that we are copying to!) determined above.
12. Right-click within each of these memory windows and select ASCII to display the contents as ASCII text.
13. What are the memory contents addressed by src?
    a. The memory contents addressed by src are the characters that make up “Hello world!”
14. What are the memory contents addressed by dst?
    a. The memory contents addressed by dst are the characters of the phrase after they have been processed individually by the code. First they are stored in src and end up in dst after it has been processed.
15. Single step through the assembly code watching memory window 2 to see the string being copied character by character from src to dst. Which register holds the character?
    a. R2 holds the character.
16. What are the values of the character, the src pointer, the dst pointer, the link register (r14) and the Program Counter (r15) when the code reaches the last instruction in the subroutine (BX lr)?
    a. 

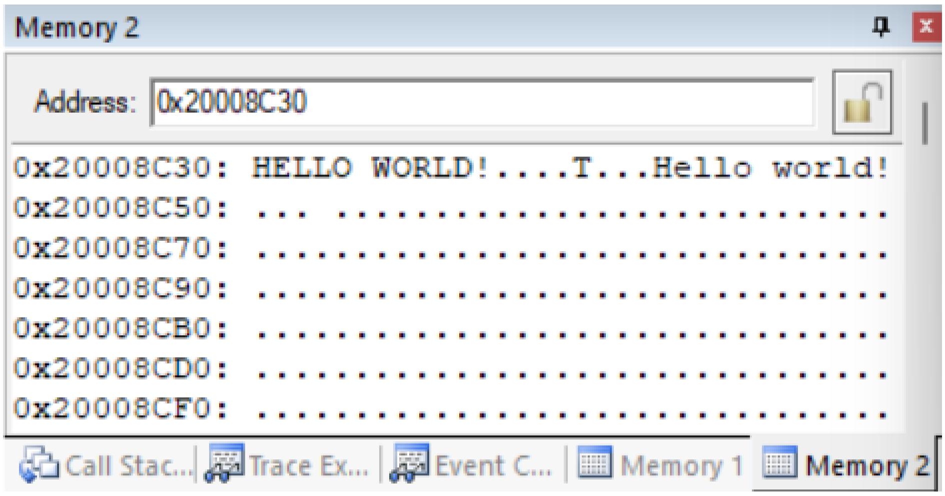
```
Core
  R0        0x20008C51
  R1        0x20008C3D
  R2        0x00000000
  R3        0x20008860
  R4        0x00000000
  R5        0x200000A8
  R6        0x00000000
  R7        0x00000000
  R8        0x00000000
  R9        0xFFFFFFFF
  R10       0x080018FC
  R11       0x00000000
  R12       0x080011C9
  R13 (SP)  0x20008C28
  R14 (LR)  0x080007CD
  R15 (PC)  0x080007FC
  xPSR      0x69000000
```

```
0x080007EC 4770      BX       lr
0x080007EE 0000      MOVS     r0,r0
     4:           __asm(
0x080007F0 7802      LDRB     r2,[r0,#0]
0x080007F2 3001      ADDS     r0,#1
0x080007F4 700A      STRB     r2,[r1,#0]
0x080007F6 3101      ADDS     r1,#1
0x080007F8 2A00      CMP      r2,#0
0x080007FA D1F9      BNE      my_strcpy (0x080007F0)
0x080007FC 4770      BX       lr
0x080007FE 0000      MOVS     r0,r0
               osKernelGetState:
0x08000800 F3EF8005  MRS      r0,IPSR
```

```
__attribute__((naked)) void my_strcpy(const
```

    b. The character returns to zero, src pointer is 0x20008C51, dst pointer is 0x20008C3D. LR is 0x080007CD and R15 is 0x080007FC.
17. Execute the BX lr instruction. Now what is the value of PC?
    a. Now PC is 0x080007CC.
18. What is the relationship between the PC value and the previous LR value? Explain.
    a. The PC value being 0x080007CC indicates that we have branched to LR. However, the value of PC is one less than LR. After some research this indicates that it is operating in ‘thumb mode’. This means the CPU is going to set LR as 1 + PC. Essentially, it is ensuring there is an instruction boundary. That is why they are one apart.
19. Now step through the my_capitalize subroutine and verify it works correctly, converting b from “Hello world!” to “HELLO WORLD!”.

a.



```
0x20008C30: HELLO WORLD!....T...Hello world!
0x20008C50: ... ............................
0x20008C70: ................................
0x20008C90: ................................
0x20008CB0: ................................
0x20008CD0: ................................
0x20008CF0: ................................
```