LAB1: PROCESSING TEXT IN ASSEMBLY LANGUAGE EXAMPLE

OVERVIEW

In this exercise you will write and compile assembly code and debug the program image on an mbed board (namely the Nucleo F401RE board) using the Keil MDK-ARM tool.

Before you embark on this lab, you need to go through the "Getting Started" document which includes instruction on how you can setup the necessary firmware.

LAB EXERCISE

MIXING ASSEMBLY LANGUAGE AND C CODE

We will use Keil MDK with a C program, but add assembly subroutines to perform string copy and capitalization operations.

Note that some embedded systems are coded purely in assembly language, but most are coded in C with assembly language used only for time-critical processing if at all. This is because the code *development* process is much faster (and hence less expensive) when writing in C when compared to assembly language.

Writing assembly code as functions, which can be called from C code as C functions, result in modular programs, which gives us the best of both worlds: the fast, modular development of C and the high performance of assembly code. It is also possible to add *inline assembly code* to C code, but this requires much greater knowledge of how compilers generate code.

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)
	    ;
}
```

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.

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.

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.

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.

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.

STRING COPY

The function my_strcpy has two arguments (src, dst). Each is a 32-bit long pointer to a character. In this case, a pointer fits into a register, so argument src is passed through register r0 and dst is passed through r1.

Our function will load a character from memory

```
__asm void my_strcpy(const char *src, char *dst)
{
loop
      LDRB  r2, [r0]     ; Load byte into r2 from mem. pointed to by r0 (src pointer)
       ADDS   r0, #1     ; Increment src pointer
      STRB  r2, [r1]     ; Store byte in r2 into memory pointed to by (dst pointer)
       ADDS   r1, #1     ; Increment dst pointer
      CMP   r2, #0       ; Was the byte 0?
      BNE   loop         ; If not, repeat the loop
      BX    lr           ; Else return from subroutine
}
```

STRING CAPITALIZATION

Let's look at a subroutine to capitalize all the lower-case 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. For example, 'A' is represented with a 65 (0x41), 'B' with 66 (0x42), and so on up to 'Z' which uses 90 (0x5a). The lower case letters start at 'a' (97, or 0x61) and end with 'z' (122, or 0x7a). We can convert a lower case letter to an upper case letter by subtracting 32.

```
__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
}
```

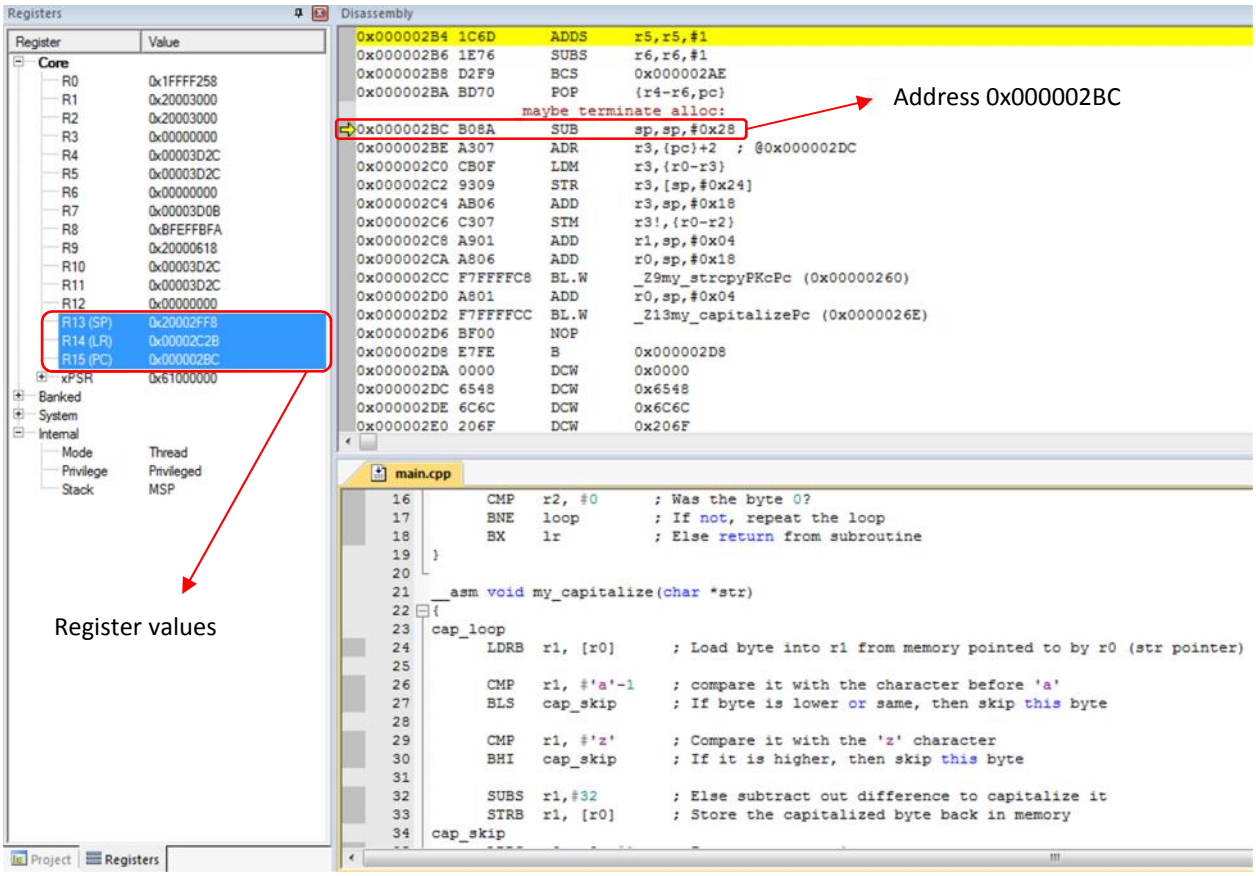
The code is shown above. It loads the byte into r1. If the byte is less than 'a' then the code skips the rest of the tests and proceeds to finish up the loop iteration.

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.

LAB PROCEDURE

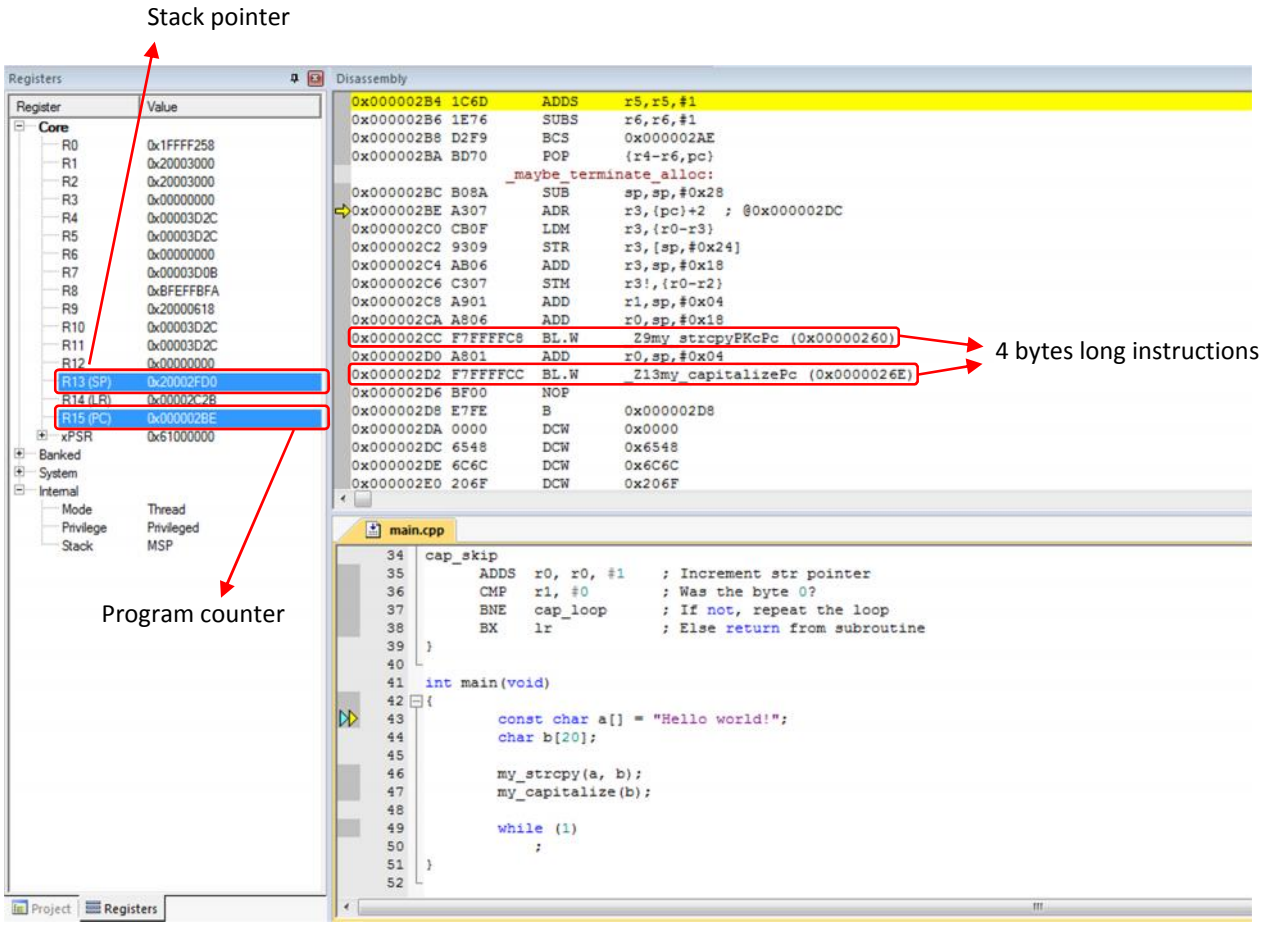
1. Open the asm project.
2. Compile the code.
3. Load it onto your mbed board.
4. Start the Debug Session (Ctrl + F5) and run the program until the opening brace in the main function is highlighted. Open the Registers window (View -> Registers Window). What are the values of the stack pointer (r13), link register (r14) and the program counter (r15)? (see picture below).

(Please be aware, that register values and addresses shown in pictures might be different from what you get)



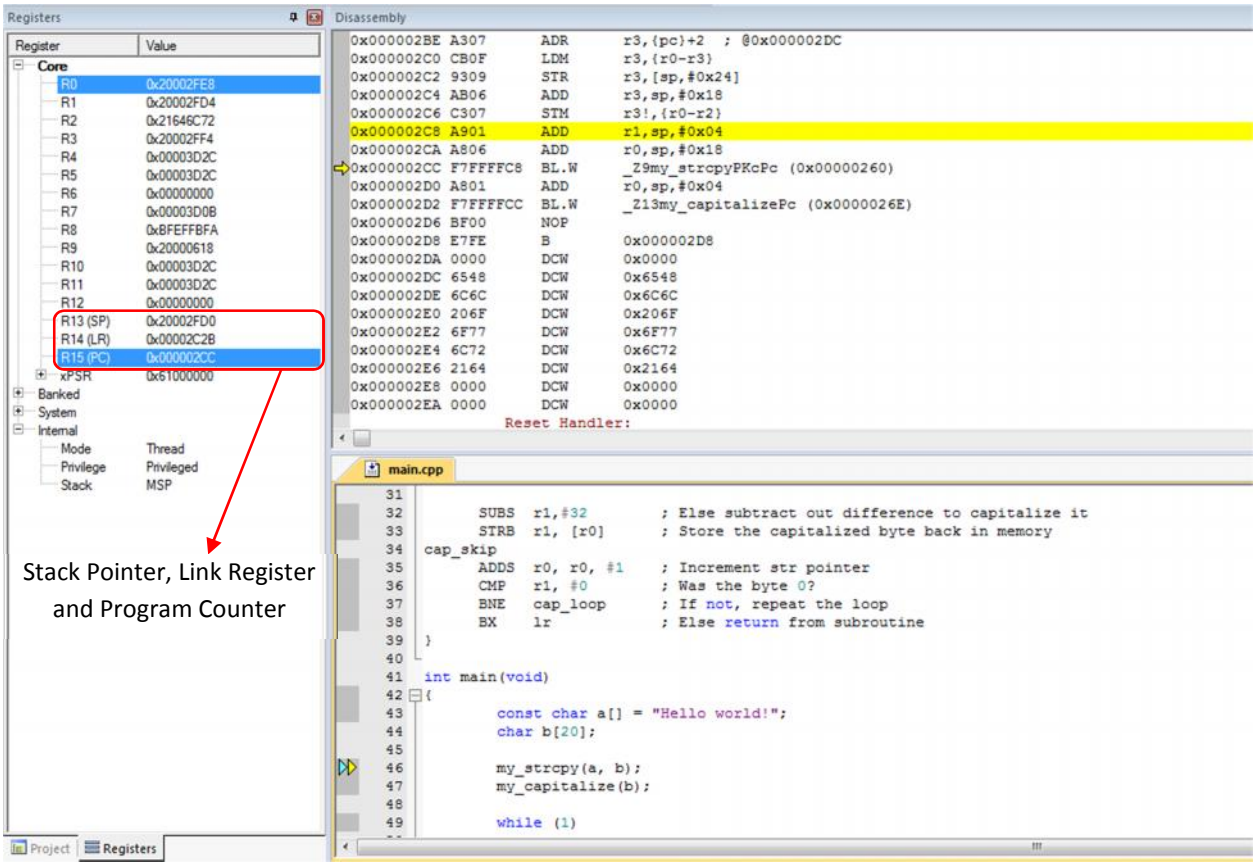
5. Open the Disassembly window (View->Disassembly Window). Which instruction does the yellow arrow point to, and what is its address? How does this address relate to the value of pc? (see picture above)

6. Step one machine instruction using the F10 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? (see picture on the next page)



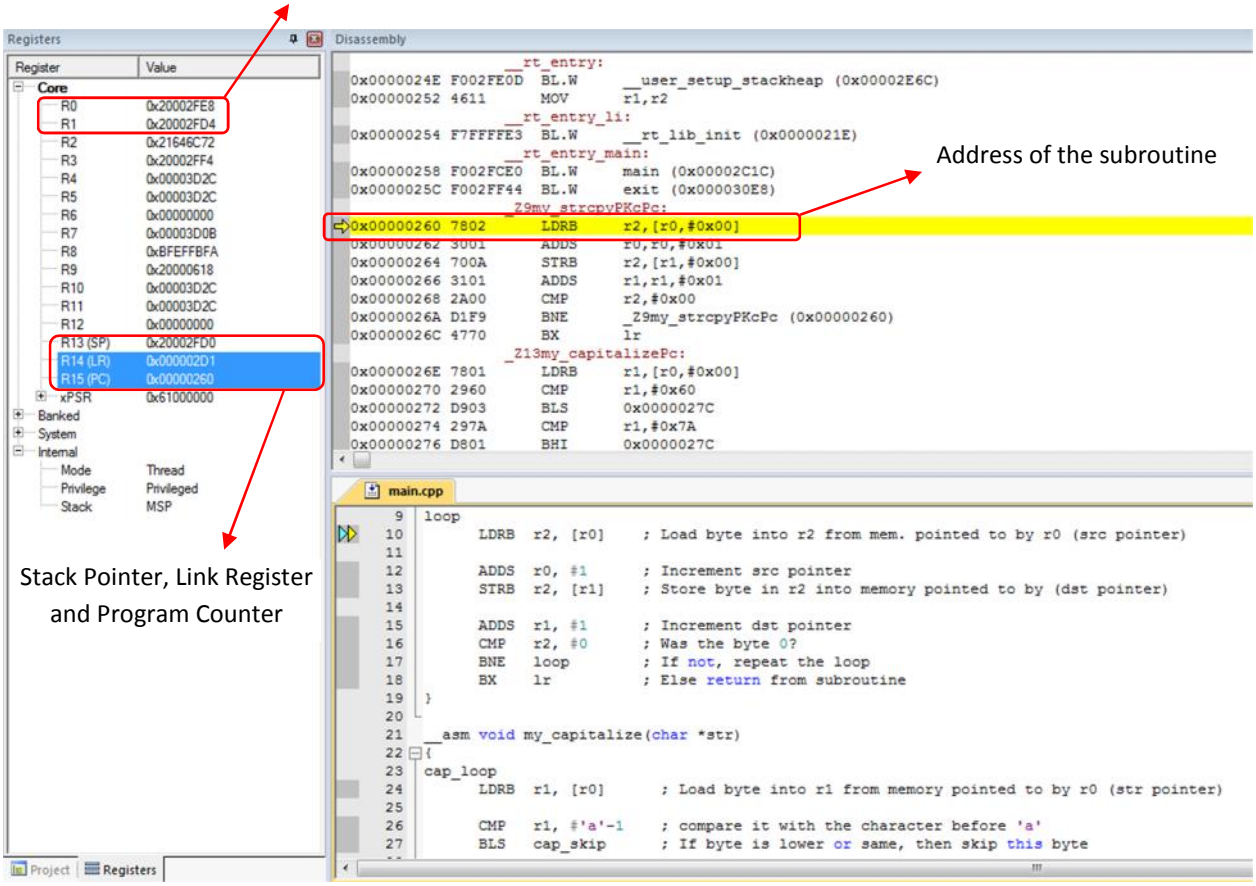
7. Look at the instructions in the Disassembly window. Do you see any instructions which are four bytes long? If so, what are the first two? (see picture above)

8. Continue execution (using F10) until reaching the *BL.W my_strcpy* instruction. What are the values of the SP, PC and LR? (see picture on the next page)

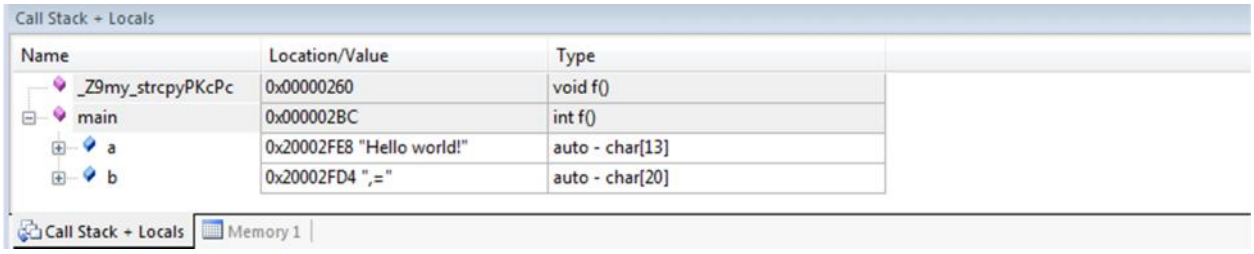


9. Execute the *BL.W* instruction. What are the values of the SP, PS and LR? What has changed and why? Does the PC value agree with what is shown in the Disassembly window? (see picture on the next page).

10. What registers hold the arguments to *my_strcpy*, and what are their contents? (see picture on the next page)



11. Watch the “Call Stack + Locals” window, to analyze the variable “a” and “b” (see picture below)

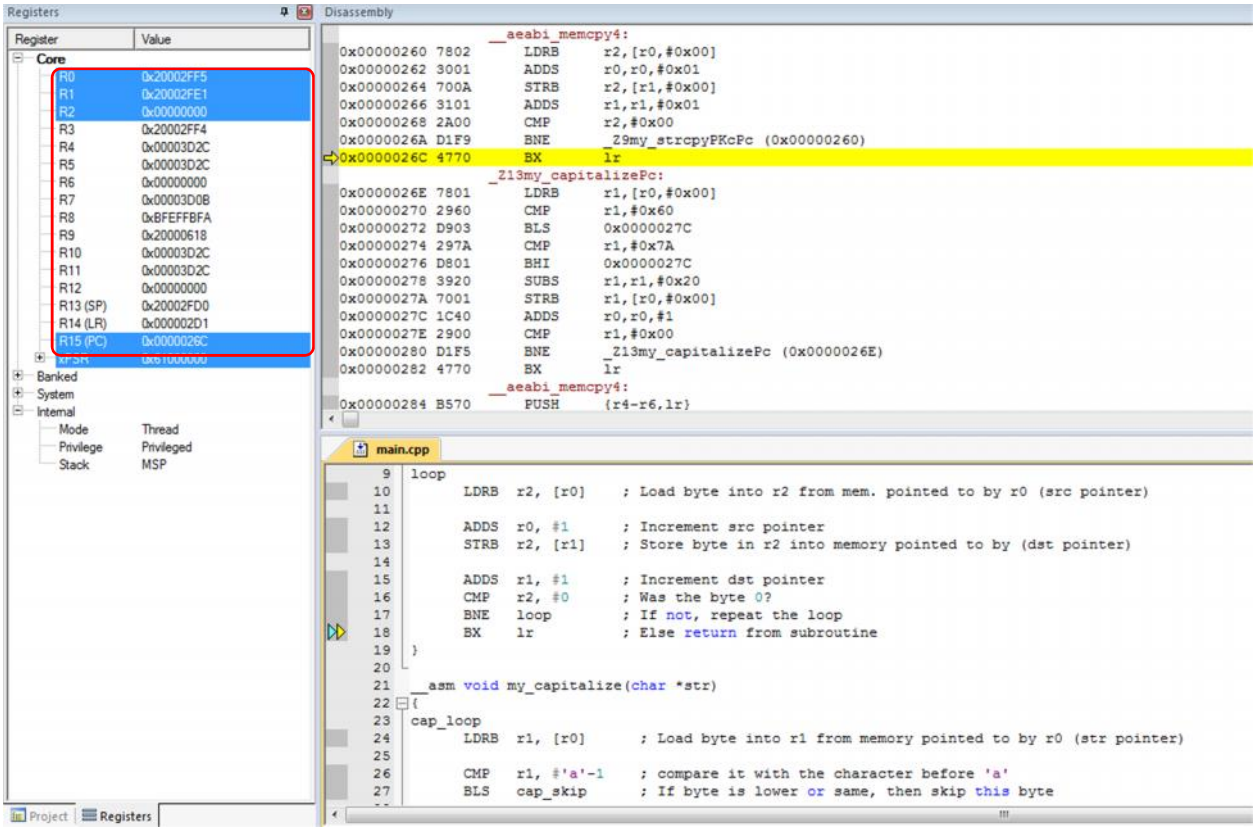


12. What is the value of “a”?

13. What is the value of “b”?

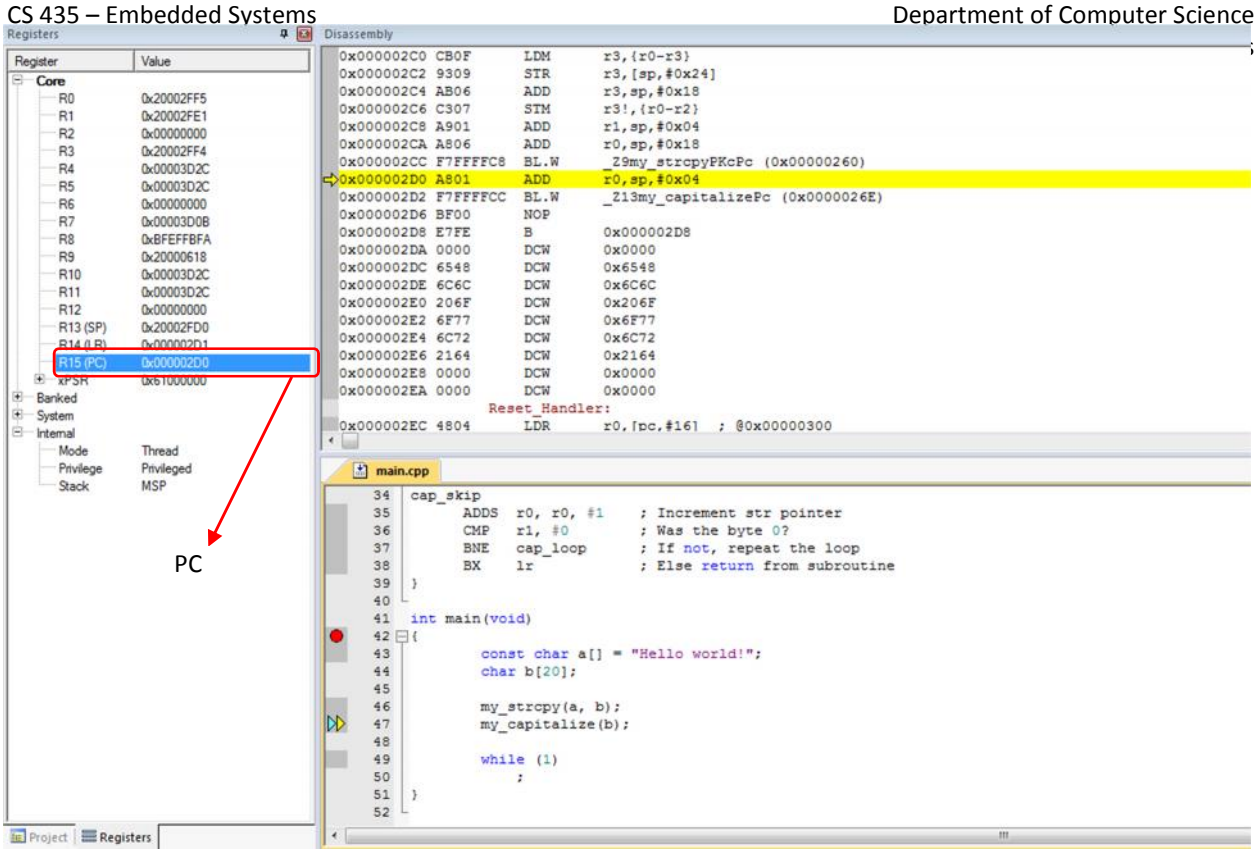
14. Single step through the assembly code watching “Call Stack + Locals” window to see the string being copied character by character from a to b. What register holds the character?

15. 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*)?



16. Execute the *BX lr* instruction. Now what is the value of PC? (see picture on the next page).

17. What is the relationship between the PC value and the previous LR value? Explain.



18. Now step through the *my_capitalize* subroutine and verify it works correctly, converting b from “Hello world!” to “HELLO WORLD!”.