Lab 1 – ARM Assembly Program - Simulation

In this exercise you will execute assembly code on the Keil simulator (based on Nucleo-F103RB board) in order to examine its execution at the processor level. You will also try to add a function to the code.

SOFTWARE

Mixing Assembly Language and C Code

We will use MDK with a C program, 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.

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 qmul!";
   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. We will examine these in more detail later in the module “C as implemented in Assembly Language”.

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, save it into the destination pointer and increment both pointers until the end of the string.

```
__asm void my_strcpy(const char *src, char *dst)
{
loop
    LDRB  r2, [r0]  // Load byte into r2 from memory 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 function 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.

Task 1: Trace the simulation - procedure

Follow the steps below and answer the questions according. Enter your answer into the online lab sheet (on QMPlus).

1. Compile the code.
2. Start a debugger session.
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**)?
4. Check 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?
5. 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?
6. 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?
7. Continue execution (using F10) until reaching the **BL.W** my_strcpy instruction. What are the values of the **sp**, **pc** and **lr**?
8. Step (using F11) and execute the **BL.W** instruction. What are the values of the **sp**, **pc** and **lr**? Does the pc value agree with what is shown in the Disassembly window?
9. What registers hold the arguments to my_strcpy, and what are their contents?
10. Open a Memory window (View->Memory Windows->Memory 1) for with the address for src determined above. Open a Memory window (View->Memory Windows->Memory 2) for with the address for dst determined above. Right-click on each of these memory windows and select ASCII to display the contents as ASCII text.

11. What are the memory contents addressed by src?
12. What are the memory contents addressed by dst?
13. Single step through the assembly code watching memory window 2 to see the string being copied character by character from src to dest. What register holds the character?
14. 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**)?
15. Execute the **BX lr** instruction. Now what is the value of PC?
16. What is the relationship between the PC value and the previous LR value?
17. Now step through the my_capitalize subroutine and verify it works correctly, converting b from "**Hello qmul!**" to "**HELLO QMUL!**".

Task 2: Adding a my_strcat() - procedure

1. Add a new ASM function in the main.c, called **__asm void my_strcat(char *str)**.
2. Add a call to the function in the main function:

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

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

    while (1);
}
```

3. Based on the example in my_capitalize(), write your function my_strcat to append your first name in Pinyin to the end of the given string (str). For example, append "John" to "HELLO QMUL!" and make it "HELLO QMUL!John". The end of the string should still be terminated by a NULL character (0 in byte value) and all code and constant data **must** be written in assembly (inside my_strcat).
4. Simulate and test your function.
5. Submit the C source code main.c to QMPlus when you have completed.

- End of lab 1 -