ECE 1181: Microcontrollers and Embedded Systems Lab
Spring 2024 – Professor Joe Camp

Lab 7: Functions and the Stack (4/4 or 4/5)
(Due before the start of the next lab – 4/11 or 4/12)

Reminder: Each lab report should have the following.
1. Name and SMU ID
2. Lab Session number & objective of the lab assignment
3. Description of the program and if writing new code, algorithm in pseudocode (25%)
4. The actual code used comments (25%)
5. Answers to any questions on the lab description, results, screenshots, and verification (50%)

- *Reports must be submitted before the start of the next lab.*

Description:

In lecture, we vicariously went through an example of using a *subroutine* or *function* (terms that we use interchangeably below) in the code examples from the lecture on loops and branches with a "print" subroutine/function to avoid having to have that chunk of code in every loop type. Subroutines (i.e., functions) are reusable code components that can operate independently to allow repeated functionality with the use of inputs (also known as arguments or parameters) and outputs (also known as return values).

Key Point: Subroutines or Functions. Small independent units that allow reusable components that can be called from anywhere that could have values that are passed in as inputs or arguments, operates on these values, and then potentially outputs returned value(s).

With this "print" function, we passed in a pointer to the start of the string to print and the size of the string to print through two registers (R1 and R2, respectively). This is known as *pass by register*, where the main program and the subroutine use a register as a portal/wormhole/oracle by which a value is seen from both the piece of code calling the subroutine and the subroutine itself. However, there are two other mechanisms we discussed in class to pass values in and out of functions. An alternative way is *pass by reference*, where the register contains a pointer to values in memory that are being passed in or out of the subroutine. A third way is *pass by stack* in which the piece of code that is calling the subroutine *pushes* values onto the stack for values that are then read by the subroutines and *popped* off the stack for values that are then returned to piece of code that called the subroutine. In the figure below, we see the calling piece of code on the left and the function or subroutine on the right. The control flow is changed by the branch and link (BL) from the calling piece of code to the subroutine by setting the program counter (PC) to the first address of the first ARM instruction of the function/subroutine func1. As the control flow is moved from left to right, there could be values passed in via one of the three aforementioned methods. At the end of func1, the control flow is then returned back to the calling piece of code via the

MOV PC, LR function (where the program counter gets the value of the return address of the next instruction after the BL func1 line of code). As the control flow is moved from right to left, there could be values possed out via one of the three aforementioned methods. However, the same method should be used to pass in values as return values.

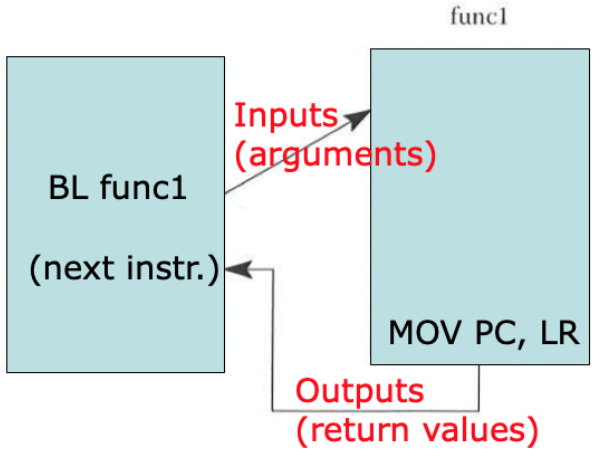

Figure 1: One piece of code pictured on the left calls another piece of code (a subroutine or function called func1). When func1 completes, the control flow returns to the next instruction after the call (BL func1).

Stacks are useful for another reason related to subroutines related to having the ability to have nested subroutine calls. For example, consider the following control flow pictured below:

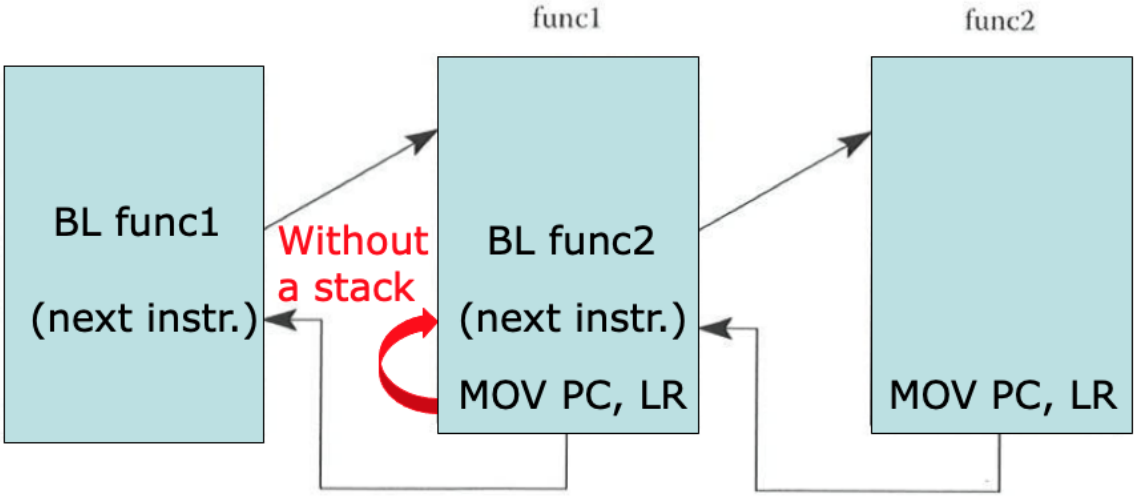

Figure 2: If func1 were to call func2, the return address contained in the link register (LR) would be overwritten, and func1 then has an infinite loop when attempting to return to the leftmost piece of code that made the original function call.

Figure 2 shows an example of a nested function call, where one function attempts to call a second function. When this occurs, the return address held in the

link register will be overwritten, and the return address of the original calling code on the far left is lost. This results in an infinite loop. However, stacks can be used to allow the return address to be stored at each level of the nested function calls and allowing reentrancy. Reentrancy not only preserves the return address for each nested function call levels but also allows an ability to keep track of inputs and outputs per level. Hence, the use of *pass by stack* is the strongly preferred methodology for professional code development.

Key Point: Stack. Stacks allow subroutine/function reentrancy. This reentrancy has two primary effects: (i.) there could be an unbounded level of nested calls to functions (including calls from within a function to itself). They also allow passing of inputs and outputs, and (ii.) passing into and out of functions are encapsulated on the *stack frame* that is associated with the current level of reentrancy. Here, a stack frame is defined as the memory allocation for the variables and data structures used in a particular function for each time it is called.

Lastly, this lab is the first that uses multiple source files in an integrated manner, something also common in professional code development.

For this lab, we will now thoroughly read Chapter 6 of Smith textbook. The book implements an uppercase example from Chapter 5 with a couple of interesting twists: stacks and multiple source files. In particular, the lab consists of the following steps:

1. Read through pages 109-116. This should mostly be a review of the material we covered in lecture with some additional specifics to Raspbian.
2. Follow the instructions from the bottom of page 116 through page 121 (Uppercase Revisited section). This includes insuring that your program is operating as expected.
3. Now, in class we learned about various stack conventions: full, empty, ascending, and descending. However, the Smith book only uses the ARM instructions of push and pop to manage the stack. Choose a stack convention that makes the most sense for you. In order to do this, you will need to change the push and pop statements according to your chosen convention by subsituting the appropriate LDMxx or STMyy statements, where xx and yy represent Increment After (IA), Increment Before (IB), Decrement After (DA), and Decrement Before (DB).
4. Test your code in a similar manner as you did in #2 above to ensure that the uppercase functionality of the code remains. If not, address the issues before moving on.
5. Now, read through the Stack Frames section on pages 121-124.
6. Take the code from Listings 6-3 and 6-4 that has been modified according to your stack convention using the LDMxx and STMyy statements. This code uses *pass by register* to get values from the main program to the toupper function. In particular, R0 and R1 are inputs to the toupper function and point to the start of the input and output string, respectively.

R0 is then returned to the main program as the size length of the string. Change the code to use *pass by stack* by using LDMxx or STMyy commands to push the R0 and R1 inputs in the main program before the call to the toupper function. Then, use LDMxx or STMyy commands to pop the R0 output returned from the toupper function to the main program. The use of xx and yy should agree with your stack convention. *Note that when you are in the toupper function, you will need to create offsets that look beyond the R4 and R5 registers in the function's stack frame to access the input R0 and R1 values on the stack. The offset could be positive or negative depending on your stack convention.*

7. Lastly, use LDMxx or STMyy commands to push and pop the value of the link register to allow the code to be reentrant. The use of xx and yy should agree with your stack convention.

Code Given: Listings 6-3 and 6-4.

You have completed this lab when:

Demonstrate to your lab instructor that you have completed the steps above.