3

Subroutines and the Stack

3.1 Objectives:

A subroutine is a reusable program module. A main program can call or jump to the subroutine one or more times. The stack is used in several ways when subroutines are called. In this lab you will learn:

- How to write subroutines and call them from the main program.
- Ways to pass parameters to and from subroutines.
- The function of the stack and the stack pointer.
- How to create and use software delays.

3.2 Related material to read:

- Text, Chapter 8, Subroutines, pp 151-188.
- Introduction to ARM Cortex-M Microcontrollers, Valvano, pp 120-121.
- Wikipedia entry, “Call_stack”
- The Stack Frame: Section 3.6 in the text, pages 153 – 163.
- Mathematical Subroutines: Section 3.7 in the text, pages 164 – 172.
3.3 Introduction:
A subroutine is a program module that is separate from the main or calling program. Frequently the subroutine is called by the main program many times, but the code for the subroutine only needs to be written once. When a subroutine is called, program control is transferred from the main program to the subroutine. When the subroutine finishes executing, control is returned to the main program. The stack provides the means of connecting the subroutines to the main program. Using subroutines saves memory, but a greater benefit is improved program organization. In future labs, you will find it beneficial to use subroutines for various functions of your programs.

3.4 The Stack:
The stack is actually just an area of memory whose highest address is in register R13. For this reason, register R13 is referred to as the stack pointer (SP). Make sure you understand the difference between the stack and the stack pointer. The stack is an area of memory; the stack pointer is the address of the last value pushed onto the stack. Usually, the stack is used for storing data when subroutines are called. The stack is a last-in-first-out, i.e., LIFO structure so the last thing stored in the stack is the first thing retrieved. A mechanical analogy will help you learn how the stack operates. One common mechanical stack is the plate rack used in some cafeterias (illustrated in Figure 3.1).

Figure 3.1: Plate and stack analogy to computer stack operation (taken from ‘Microcomputer engineering by Miller’).
Figure 3.1(a) illustrates a cross-section of a plate rack containing some plates. As hungry consumers go through the cafeteria line, each person takes a plate off the top of the rack. When a plate is removed, a mechanism moves the plate upward so that the next plate is now at the top of the rack. Figures 3.1(b) and 3.1(c) illustrate this process. The next consumer now can easily access the next plate that is at the top of the rack. When the dishwasher puts clean plates into the rack, the mechanism moves downward. The top plate is again at the top of the rack, as Figure 3.1(d) illustrates. If you consider the usage of the plates, you see that the last plate put into the rack is the first plate removed from the rack. Thus the rack is a last in first out or a LIFO device. The stack operates in exactly same way; hence it is also referred to as a LIFO data structure.

The *push* instructions first decrement the stack pointer by one or two, and then stores one or two bytes of data onto the stack. Consider what happens if register SP contains the address 0x2000.00FF and the register R0 contains the value 0x6B. Executing the instruction `PUSH {R0}` first decrements the stack pointer by one, so register SP is now 0x2000.00FE and pushes the value 0x6B onto the stack at memory location 0x2000.00FE. The `POP {R0}` instruction first puts the contents of the top of stack into the R0 register and then increments the stack pointer by one.

The main thing you will be using the stack for is saving data when a subroutine is called. For example, assume your main program uses registers R0-R4. You call a subroutine that is going to calculate some value and pass it back to the main program. After you call the subroutine, you can just push all the data onto the stack before executing any instructions in the subroutine. The subroutine can then use all the registers for its internal use and store the data that the main program needs in one of the memory locations. At the end of the subroutine, you can pop the stored data off of the stack and then return control to the main program.

Thus the important thing to know when using the stack is that pushes and pops have to be used in pairs because the stack is a LIFO data structure. If you push several registers, you have to pop them off the stack in the opposite order. The following illustrates this point. When you call a subroutine that you didn't write, you need to save your register data on the stack because it might be changed by the subroutine.

```
BL subrout ; Main program
.
subrout PUSH {R0} ; Subroutine
PUSH {R1}
PUSH {R2}
PUSH {R3}
.
; SUBROUTINE INSTRUCTIONS GO HERE
.
POP {R3}
POP {R2}
POP {R1}
POP {R0}
```
3.5 Calling a subroutine:

You will recall, the program counter, or register PC (R15), always contains the address of the next instruction to be executed in a program. Calling a subroutine is similar to an unconditional branch as the program jumps to a different address other than the next address. The difference is that when the subroutine finishes executing, control goes back to the instruction after the subroutine call in the main program. A BL instruction causes the address of the next memory instruction to be pushed onto R14 (Link Register or LR) and the argument of BL to be loaded into the program counter. The argument of BL is the starting address of the subroutine. But, when you write your program, you just give it a name and the assembler figures out the rest. At the end of a subroutine, the instruction BX LR causes what was last stored in LR to be loaded into the program counter. In this way, the instruction after the BL in the main program is the next one executed. Note that if the value of register LR is changed within your subroutine (e.g. by calling OutStr or InChar), register LR must be saved/restored on the stack in order to preserve its value. In general, it is a good habit to use instructions push {LR} and pop {LR} at the beginning and end of each subroutine.

3.6 Parameter passing:

A parameter is passed to the subroutine by leaving the data in a register, or memory, and allowing the subroutine to use it. A parameter is passed back to the main program by allowing the subroutine to change the data. This is the way parameters are passed in assembly language. When the parameter being passed to the subroutine is in a register, this is referred to as the call-by-value technique of passing parameters. If the data is in memory and the address is passed to the subroutine, this is called call-by-reference. It is important to document all parameter-passing details in subroutines.

3.7 Software delays:

One way of creating a specific delay is to write a subroutine that contains a loop. The time the loop takes to execute can be calculated by adding up the total number of clock cycles in the loop and multiplying this by the period, T, of one clock cycle. This is the inverse of the clock frequency. The number of times the loop executes can be adjusted to create different delays. This loop counter can be passed to the subroutine so it can be used for different length delays.

```
1 ;******************************************
2 ; Subroutine to create delay,
3 ; DELAY_CLOCKS is the counter which is
4 ; decremented to zero
5 ;******************************************
6 Delay LDR R2,=DELAY_CLOCKS ; set delay count
7 del SUBS R2, R2, #1  ; decrement count
8 BNE del  ; if not at zero, do again
9 BX LR   ; return when done
```

Figure 3.2: Subroutine to implement software delay loop containing 1 loop.

Figure 3.2 consists of one loop. The loop has two instructions, SUBS and BNE. SUBS takes 1 clock cycle and BNE takes 3. If N is the number of loops, we have 4N clock cycles, plus 1
for SUBS, and another 3 for the second BNE. This gives a delay for the inner loop as \((4N + 4) T\), where \(T\) is the clock period. If the loop only executes once, we have \(N = 1\) for a delay of \(8T\). Since Tiva has a clock rate of 16 MHz or a clock cycle time of \(T = 62.5\text{ns}\), 8 clock cycles gives us a delay of \(8T = 500\text{ns}\). This is the shortest delay this loop can create, so we say it has a resolution of 500ns. The max value for \(N\) is the largest binary number that can be represented with 32 bits, or 4,294,967,295 decimal, and gives a delay of

\[
(4 \times 4,294,967,295) \times 62.5\text{ns} = 1073.74\text{s} = 17.9\text{ minutes}.
\]

If a longer delay is needed, an outer loop is added.

Make sure that none of the counters are set to 0 otherwise the subroutine will go into infinite loop. Can you figure out why? How can you avoid it even if any of the counters is 0?

### 3.8 More utility subroutines:

In the last lab, we introduced 3 monitor utility subroutines: InChar, OurChar, and OutStr. In this lab, we introduce the utility subroutines **Out1BSP** and **Out2BSP**. Out1BSP takes the byte located in register R0 and outputs it as hex to the terminal emulator followed by space. Out2BSP does the same for a 2 byte number. Figure 3.3 shows a sample program written for this utility subroutine. **Note** the difference between calling functions OutStr and Out1BSP. The first subroutine expects, in index register R0, the address of the first character in the string to be printed and the second subroutine expects in index register R0 the value of number to be printed.

(Assume typical stack, reset and data areas)

1. EXTERN Out1BSP
2.
3. __main MOV R0,#0x08 BL Out1BSP
4. done B done

**Figure 3.3:** Utility subroutine example explaining Out1BSP subroutine.

The **EXTERN** directive in line 1 tells the assembler to include Out1BSP when it links the output files. Line 3 moves the byte you wish to output into R0. Line 4 calls the subroutine **Out1BSP** that outputs “08 “ (without the quotes) to the terminal emulator.

### 3.9 Procedure:

Before you come to the lab compute the value for **DELAY_CLOCKS** label in Fig 3.2 (*Hint: it is a symbol*) and verify that Procedure 1a (**DELAY** Subroutine) generates a 1 second delay (*Hint: call DELAY 10 times and make sure there is 10 s delay*) and you must create a flowchart for the pseudo code for the main program at the end of this section. At the start of the lab, show your TA this flowchart, your paper computation and DELAY subroutine.

Your uVision project should include the Startup.s file as well as OutStr.s, InChar.s, etc. files to be linked together.
1. Consider a 9-story building with a basement. The first floor is given the number 1, the second is given the number 2 and so on. The basement is given the number 0. You will write a program to simulate an elevator operated throughout the whole building. Write the following subroutines that will work with the main program,

a) **DELAY**: generates a software delay of 1 second.

b) **OPEN**: displays a ‘Door open’ message on the screen after a delay of 1 second. A working version of this code is supplied for your use, if you like on the “Software Downloads” page from the Lab page.

c) **CLOSE**: displays a ‘Door close’ message on the screen after a delay of 1 second.

d) **UP**: performs the following procedure:
   i. Displays a ‘Going UP’ message on the screen.
   ii. Goes up one floor at a time (1 second delay between each floor), until the elevator reaches the destination.
   iii. At each floor, prints that number (current floor number) on the screen.
   iv. After the elevator reaches the destination floor, stores this number as the current floor number in memory.

e) **DOWN**: performs the following procedure:
   i. Displays a ‘Going DOWN’ message on the screen.
   ii. Goes down one floor at a time (1 second delay between each floor), until the elevator reaches the destination.
   iii. At each floor, prints that number (current floor number) on the screen.
   iv. After the elevator reaches the destination floor, stores this number as the current floor number in memory.

f) **STAY**: displays a ‘Same floor’ message on the screen after a delay of 1 second.

Following is the pseudo code you can use for your main program,

1. Initialize the elevator at floor #1 (memory location containing the present location of the elevator).

2. Call the ‘OPEN’ subroutine.

3. Wait for the input from the user (0-9).

4. Call the ‘CLOSE’ subroutine.

5. Compare the number input with the present floor number.

6. If the number input is greater than the present floor number, call the ‘UP’ subroutine. Then jump to step 9.

7. If the number input is less than the present floor number, call the ‘DOWN’ subroutine. Then jump to step 9.

8. If the number input is equal to the present floor number, call the ‘STAY’ subroutine.

Show the working program to TA.

Optional procedure for more practice:

2. Write a subroutine to create a delay of up to 30 seconds and a resolution of 10 ms. Write a main program to call this subroutine and display the word 'START' on the terminal when the subroutine is called. Display the word 'STOP' when the program returns to the main program. Have the main program pass the length of the delay to the subroutine and test this for a 30 second delay. Show the TA when this works correctly.

3.10 Questions:

1. Consider the program written in Figure 3.4 and the memory map in Figure 3.5. Manually calculate the values of the memory map after the instructions on lines 8, 9, 10, 11, and 12 are executed. In other words, you should draw six separate memory maps and show the memory contents after each instruction.

```
1   Stack  EQU 0x00000400
2
3   AREA STACK, NOINIT, READWRITE, ALIGN=3
4   StackMem SPACE Stack
5
6   __main  MOV R0,#0x42 ; Initialize R0 and R1
7   MOV R1,#0x55
8   PUSH {R0} ; Save them on stack
9   PUSH {R1}
10  BL subnop ; Call subroutine
11  POP {R1} ; Pull off stack
12  POP {R0}
13  done  B done
14
15  subnop  NOP ; Does nothing
16     NOP
17     BX LR ; Return to main program
18  END
```

Figure 3.4: Program for Question 1.
**Figure 3.5:** Memory map for Question 1.

2. When you use push and pop instructions, why do they have to be paired?

3. What does the following bit of code do?

   1. **PUSH** {R0}
   2. **PUSH** {R1}
   3. **POP** {R0}
   4. **POP** {R1}

   Do you think this is a good way to do this, or is there a better way?

4. Regarding the previous question, what if you coded this as:

   1. **PUSH** {R0,R1}
   2. **POP** {R0,R1}

   Would you get the same results as the four instructions above? Why or why not?

**3.11 Lab report:**

For the lab write up, include

1. Flowcharts and programs that you have written for this lab.
3. A copy of your working .s files.
4. A brief discussion of the objectives of the lab and the procedures performed in the lab.
5. Answers to any questions in the discussion, procedure, or question sections of the lab.