Laboratory 3
Subroutines and the Stack

Introduction

The objective of this laboratory is to familiarize you with the use of a stack in a microprocessing system. The stack is an important data structure that facilitates the implementation of function/procedure calls, particularly in cases where the calls are nested.

An explanation of subroutine calls and the use of the stack with the ARM processor are covered in sections 3.8 and 3.10 of the textbook.

Procedure

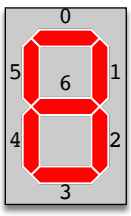
To begin this laboratory, you will once again be staring with an example program and exploring its operation before writing your own programs.

As you did in Laboratory 1, create a new ARM assembly language project using the sample program Simple Program. Modify the **`simple_program.s`** file so it matches the following (changes highlighted in blue).

```
.include    "address_map_arm.s"

.text
.global     _start
_start:
        LDR     R1, =LED_BASE  /* Address of red LEDs. */
        LDR     R2, =SW_BASE    /* Address of switches. */
        LDR     R3, =HEX3_HEX0_BASE /* Address of the low four 7segment digits */
        LDR     R4, =HEX5_HEX4_BASE /* Address of the high two 7segment digits */
LOOP:
        LDR     R5, [R2]        /* Read the state of switches. */
        STR     R5, [R3]        /* Display the state on 7segment. */
        STR     R5, [R4]        /* Display the state on 7segment. */
        B       LOOP
.end
```

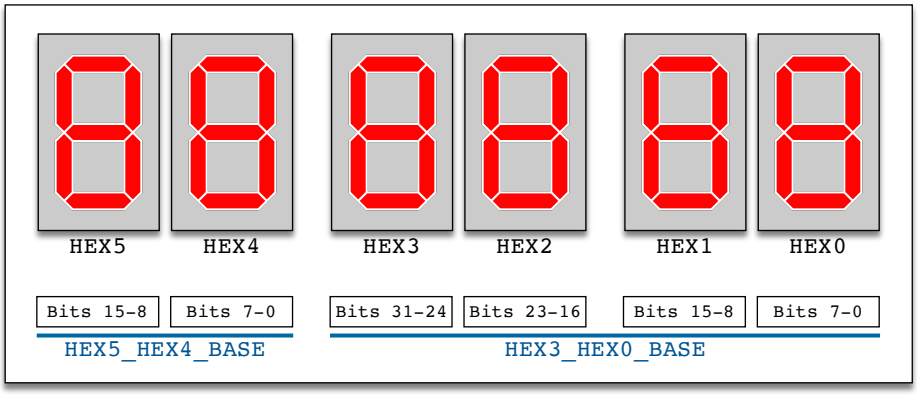
This program will read the state of the switches into `R5` and then store it to two memory locations, each of which directly control the individual segments of a bank of six 7-segment displays. The memory location `HEX3_HEX0_BASE` controls the lower four 7-segment digits (labelled `HEX3`, `HEX2`, `HEX1`, and `HEX0`), with each byte controlling a separate digit. The memory location `HEX5_HEX4_BASE` controls the upper two 7-segment digits (labelled `HEX5`, and `HEX4`). The figure at the top of the following page illustrates the relationship between the individual bytes and their corresponding digit.



Each bit within each byte directly controls the state of an individual segment within each digit. The segment associated with each bit is identified in the illustration of the 7-segment digit to the right.

For the program above, digits `HEX4` and `HEX0` will display the same pattern, since both are associated with the least significant byte of their corresponding memory location and the same value is being stored to both of these locations. The same is true of digits `HEX5` and `HEX1`.

Test the operation of the program by downloading it to your board and observing how the switches affect the 7-segments.



Modifying the Program

1. The supplied program does not directly display a number, but rather the raw state of the individual bits within each byte associated with a digit. Your first task is to create a subroutine that will function as a binary to seven segment decoder. The subroutine will take the value of the least significant byte read from the switches (i.e. the value in `R5`) and interpret that value as a binary encoded number (between 0 and 9). Based on that value, set the contents of register `R6` so that it hold the correct pattern of segments to create that digit on a 7-segment display. The mainline code should then store the contents of `R6` to the 7-segment displays to show the value.
   Call the subroutine from the main program using a branch-with-link (BL) instruction and return from the subroutine using the return address stored in the link register (`R14`).

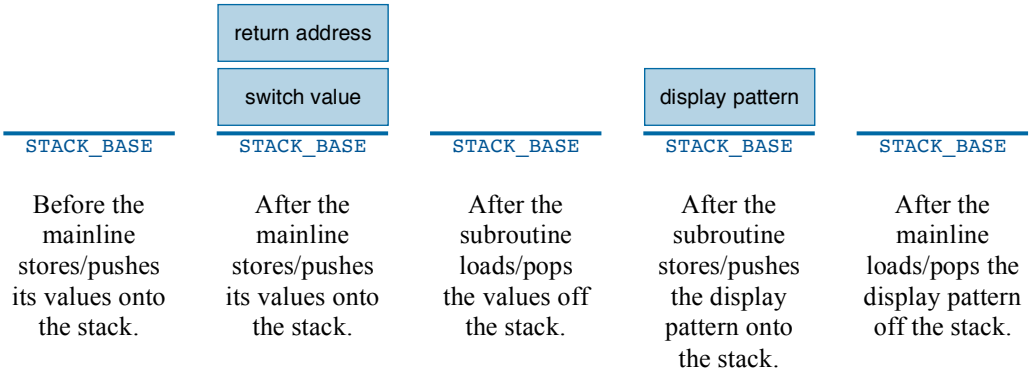
2. To illustrate use of the stack, modify your program further by including the assembler directive…

   ```
   .equ    STACK_BASE,    0x30000000
   ```

   …at the top of your program. This will to declare a meaningful label defining the location for the base address of a stack (i.e. the starting address from which the stack will grow). Then, at the beginning of your program (before `LOOP`), load this value into the stack pointer register (`R13`). Now in place of the branch-with-link instruction used in part 1, use a store multiple instruction (`STMEA`) to push both the desired return address and the switch value (from `R5`) onto the stack, then call the subroutine using an unconditional branch (`B`).

   The subroutine should then pop the switch value and return value off the stack using the load multiple instruction (`LDMEA`). Use this popped switch value to determine the 7-segment pattern to display (as you did before). Before your subroutine returns, push the 7-segment display pattern onto the stack, then return from the subroutine by setting the program counter to the return address. The mainline program can then pop the 7-segment display pattern off the stack and output it to the 7-segment displays.

   The figure on the following page illustrates what is happening to the stack at each stage of this operation.



Step through the program while using the debugger to observing the contents of the stack in memory (at location `0x30000000`) and the processor registers. Observe how the stack functions and confirm that the subroutine is returning to the correct location in the mainline program (i.e. the instruction after the unconditional branch).

3. Test the other three variations of the stack specific store multiple, load multiple instruction pairs (`STMED`/`LDMED`, `STMFA`/`LDMFA`, `STMFD`/`LDMFD`) and observe how the data is stored on the stack.

4. Finally, test at least one version of the general store multiple, load multiple instruction pairs. i.e. `STMIA`/`LDMDB`. This should help you appreciate the advantage of using the stack specific version over the general version.