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

Lab 3: Moving and Adding (2/8 or 2/9)
(Due before the start of the next lab – 2/15 or 2/16)

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:

Now that we can see the values of the registers in the ARM7TDMI data path via the debugger, we can return to Chapter 2 in the Smith textbook.

Key Point #1: Interpretation of Values. One key point in this lab is how bits and bytes can be interpreted in different ways by both the programmer and the ARM processor, both in terms of positive/negative values (this was a focus of HW1 and a concept that is revisited on pages 27-30 of the Smith text) and endianness (a concept that is discussed on pages 30-32). A *programmer* (you) is responsible for knowing the interpreted meaning of the bits, which could be positive or negative number or another symbol altogether (*e.g.*, a representation of the ASCII character for "@"). The programmer must be consistent in the way in which a particular set of bits are interpreted or operated on throughout the program to not end up with non-sensical values. A *processor* (ARM) will interpret a number in two's complement form and simultaneously report multiple forms overflow for signed/unsigned arithmetic. The programmer then checks for overflow based on their interpretation.

Key Point #2: Initializing Registers. Another key point in this lab assignment is to understand how register values can be initiated without accessing memory. This is referred to as immediate values or those that are *immediately available* in the instruction. Note that while the author calls this chapter "Loading and Adding", the term "load" is a misnomer because the word "load" in the context of ARM processors is reserved for the process by which a value is *loaded* from memory to registers. In this chapter and lab assignment, you are not accessing data portions of memory to read values for the initialization of registers but rather getting the values from what is held inside the 32-bit ARM instructions. Note also that the reference point is always from the perspective of the processor, so loading means pulling values from memory into the ARM registers, whereas stores will be putting values in memory from ARM registers. Loading and storing will be covered in the next lab assignment.

We will now follow Chapter 2 very closely. In particular, the lab consists of the following steps:

1. If you have not already due to the referenced pages with the two key points above, read through pages 27-30 on negative numbers and 30-32 on endianness.
2. Continue to work through the text on pages 33-41 on Shifting and Rotating values, the Carry flag, the Barrel Shifter, the MOV and MVN instructions, and the second operand. These topics should be familiar to you from lecture but offer another way in which to learn about them. Now, you are ready to interact with these concepts.
3. With the code given in Listing 2-1, follow the instructions to compile (with DEBUG = 1) and disassemble the program according to page 43.
4. Now, take your code from Listing 2-1 and modify all the MOV instructions to be MOVS instructions. When the "S" is added, bit 20 of the MOV instruction is set to 1 (see Figure 1-3 and the bullet describing "Set condition code:"), which means that the Current Program Status Register (CPSR) will be changed by the instruction. Here, we are just interested in the Carry (C) flag, which is bit 29 of the CPSR. We can observe the status of the C bit of the CPSR through the debugger.
    a. Based on your understanding of the debugger from the last lab, create break points in the first 7 MOVS instructions of the code to see what the value of the C flag is after each line is stepped over. . You do not need to worry about the line with MOVT. To do so, I would recommend that you put labels, such as "mv1:", "mv2:", etc. on each line you are interested in and then set breakpoints to those labels. Conversely, you can just use line numbers for setting breakpoints, but you will have to retrieve the line number from your source file.
    b. When you have captured the value for C of each of the first 7 MOVS instructions, take a screen shot of your terminal and include it in your report.
    c. Describe why each line produced the resulting C value in the CPSR. Hint: the diagram included below from Figure 7.5 of the Hohl textbook will help you in your description.
5. Continue to work through the text on pages 45-49 on Adds and Adding with a Carry. These two instructions should be familiar to you based on digital logic and HW1.
6. With the code given in Listing 2-3, create a source file, which you might call, for example, "mvnaddexamp.s". Follow the previous procedure to compile that new source code. Note that the Smith text is using the return code to give a value back to the terminal. However, the return code can only span 0 to 255. So, the amount of information that can be passed back using this approach is very limited compared to the debugger. Take a screen shot and include this portion in your lab report.
7. With the code given in Listing 2-4, create a source file, which you might call, for example, "addsadcexamp.s". Follow the previous procedure to compile that new source code.

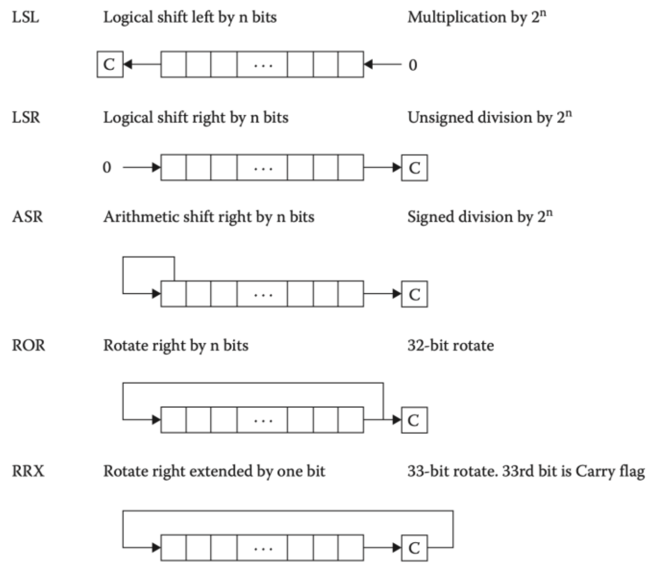
LSL
Logical shift left by n bits
Multiplication by 2^n
LSR
Logical shift right by n bits
Unsigned division by 2^n
ASR
Arithmetic shift right by n bits
Signed division by 2^n
ROR
Rotate right by n bits
32-bit rotate
RRX
Rotate right extended by one bit
33-bit rotate. 33rd bit is Carry flag
C
0

FIGURE 7.5 Shifts and rotates.

8. We will now expand to examining other condition codes in the CPSR of N (bit 31), Z (bit 30), C (bit 29), and V (bit 28). With a starting point of your code from Listing 2-4, we are going to add two 32-bit numbers. The first 32-bit number will be the hexadecimal version of your student ID. For example, if your student ID is "12345678", then your hexadecimal version of your student ID would be "0x12345678". This would go into R3. You will now iterate on different values of R5, to produce the following results of the condition codes in the CPSR as a result of stepping over the ADDS instruction:
    a. N=0
    b. N=1
    c. Z=0
    d. Z=1
    e. C=0
    f. C=1
    g. V=0
    h. V=1

   Show with a screen shot of the debugger that you have successfully chosen a value for R5 that achieves a CPSR condition code for each one of these cases. The values of R5 that satisfy one condition may also satisfy others. Hence, you do not necessarily need 8 unique R5 values.

Code Given: Listing 2-1, Listing 2-3, and Listing 2-4 from Chapter 2 of Smith text.

You have completed this lab when:

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