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

Lab 4: Thanks for the Memories (2/15 or 2/16) – Smith Ch. 5
(Due before the start of the next lab – 2/22 or 2/23)

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%)

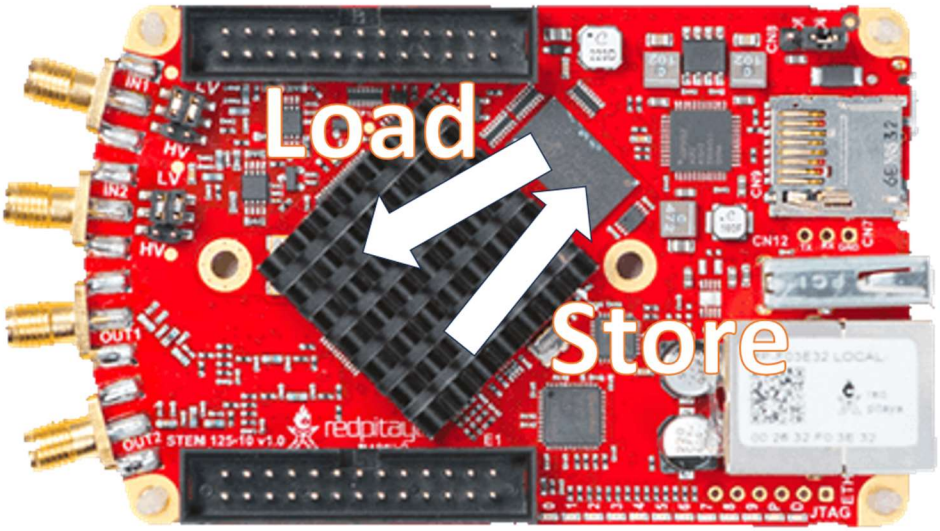
- *Reports must be submitted before the start of the next lab.*

Description:

Last lab, we initialized registers with immediate values and worked with the condition codes within the CPSR. While the Smith textbook progresses to changing the program flow in Chapter 4 where the program counter goes beyond just incrementing (PC = PC + 4) to potential movement anywhere in memory, this is not necessary to learning about accessing memory that would directly follow from the immediate-value concepts learned in Lab 3. Hence, we now move to Chapter 5 (Thanks for the Memories) to learn about loading data values from memory to registers and storing data values from registers into memory. Since we did not yet go into Chapter 4 of the Smith text. There are select parts before which use loops but the Code Given and code to complete can be done easily without loops.

Key Point #1: ARM Processor is Point of Reference. Remember that the point of reference is always the ARM processor.

a. Loading means pulling values from memory into the ARM registers.
b. Storing means putting values in memory from ARM registers.



Key Point #2: Importance of Load-Store Architecture. As discussed in class, the instructions that you learn in this lab: load and store, are critical for a Reduced

Instruction Set Computer (RISC). The idea is that you only touch data portions of memory with loads and stores and all other instructions just operate on registers. Hence, if you restrict all the other instructions to only operate on registers, this reduces the number of different versions of those instructions, thereby dramatically reducing the overall size of the instruction set. The reduced instruction set then has the added benefits discussed in class (reduced design time for the chip, similar execution times across instructions, thereby allowing pipelining, etc.).

We will now follow Chapter 5 very closely (at least at the beginning). In particular, the lab consists of the following steps:

1. Read through pages 87-91 on defining memory contents to reinforce your understanding of assembler directive statements from the discussion in lecture.
2. Mostly, skip over the section on PC Relative Addressing on pages 92-95. We will cover this when we talk about literal pools next week. However, you should observe Table 5-3 that shows that the load instruction (LDR) can have the added letters of B, SB, H, SH that symbolize unsigned byte, signed byte, unsigned halfword (16 bits), and signed halfword (16 bits). If there is no such added letters, the LDR instruction defaults to a word size (32 bits).
3. Go through the Loading from Memory section on pages 95-96. Take the code listed given in Listing 5-2 and "wrap" it inside the starting code and ending code that we have mostly been using so far in the labs so that it can compile.
    a. *Beginning:* In particular, put the beginning lines of code that we have been using that use the .global directive for _start and defining the label _start on the first line of the code.
    b. *Ending:* End the program by setting R0 to 0 (for a 0 return code), R7 to 1 for a service command to terminate the program, and service call of 0 to call Linux to terminate the program. This program ending should come before the .data directive and the corresponding mynumber: label with the .word that is defined in the data section.
4. Then, make sure this newly-formed code compiles. If you have done this successfully, congratulations! *This will be the normal process that you need to use to get ARM assembly code up and running on your Red Pitaya.*
5. You will need to enable the DEBUG flag (DEBUG=1) with the corresponding DEBUG condition in your makefile, as you did in the last two labs to enable the debugger. Once DEBUG is enabled and the assembly program is made with that makefile, run the debugger "gdb <root name of program (no file extension)>".
6. Now, we will observe all of the relevant elements of a load instruction. To do so, set a breakpoint ("b _start"), run ("r"), and step over ("s") the first LDR statements, observing the contents of R1 ("i r" at the command line). This is the effective address of the data to be loaded.

7. Run the following command "x /4ubfx <contents of R1>" where the contents of R1 is the value you saw held in R1 after the "i r" command. The "x" is telling the debugger to display memory contents. The "/4ubfx" means "show the next 4 unsigned bytes in hex form of the following memory address". This means that the effective address currently contains the contents of those 4 bytes that are listed. This should be the same bytes of the mynumber. However, their order is reversed because the Red Pitaya uses Little Endian by default.
8. Next, step over ("s") the second LDR statement, which reads the contents of the effective address held in R1 and places the value in R2 and display the contents of the registers ("i r") to ensure that R2 now contains that value that is still held in memory at the place that will be read from to perform the load. *In other words, R1 holds the effective address.* Note, however, that the order of the bytes in the register is the same order specified at label mynumber: of the source code as opposed to the order seen in memory at effective address R1 ("x /4ubfx <contents of R1>"). You will need this in step 15 to see what is changing in memory with the store (STR) instructions.
9. Change the word in the mynumber line to be your student ID in hexadecimal form. Now, remake the program and disassemble it to show the memory contents. Take a screen capture and include it on your report, discussing which memory addresses now hold your student ID.
10. We will now iterate on this source code (with your student ID) in different ways to understand pre- and post-indexed addressing and sign and unsigned loads of various sizes. To do so, modify the line that contains: LDR R2, [R1] to do the following. With each of these, if the base register (R1) is updated with a new value due to the load (LDR), reset it to the original location of the label mynumber before comparing it to the subsequent instruction.
    a. Load a byte into R2 and replace "[R1]" with "[R1, #1]" and compare that to when you replace "[R1]" with "[R1], #1".
    b. Load a signed byte into R2 and replace "[R1]" with "[R1, #1]!" and compare that to when you replace "[R1]" with "[R1], #1".
    c. Load a halfword into R2 and replace "[R1]" with "[R1, #2]" and compare that to when you replace "[R1]" with "[R1], #2".
    d. Load a signed halfword into R2 and replace "[R1]" with "[R1, #2]!" and compare that to when you replace "[R1]" with "[R1], #2".
11. For each of these, in your lab report, discuss the differences in addressing (pre-, post-, or auto-indexed addressing) and why the contents of R2 results in what it does.
12. All of this has been pulling values into the ARM processor (specifically to registers) from memory (i.e., **loading** values). We now switch to doing the reverse process of pushing values into memory from the ARM processor (specifically coming from registers). Here, the effective address will still be the term that we use to say where the action is taking place in

the data portion of memory. However, with stores, the contents in memory at the effective address will be changed (as opposed to the contents of the register.)

13. To directly compare what was happening before with a load (LDR) to what will happen with a store (STR), change the line that was previously "LDR R2, [R1]" in the original Code Given to "STR R2, [R1]". What does this do to the contents of memory at the effective address? Please answer this in your lab report.
14. Now, add back the "LDR R2, [R1]" above the store and add 0x1 to each of the bytes of your student ID with the following two instructions:
    LDR R3, =0x01010101
    ADD R2, R2, R3
    Next, have the line of code "STR R2, [r1]". What does this do? Answer this in your lab report.
15. We will now iterate on this source code (with an incremented version of all the bytes in your student ID) in different ways to understand pre- and post-indexed addressing and stores of various sizes. Before doing this, do "SUB R2, R2, R3" to use your original mynumber value to see what is changing in memory with the following statements that make changes from the original line of "STR R2, [R1]" in the following manner. With each of these, if the base register (R1) is updated with a new value due to the store (STR), reset it to the original location of the label mynumber before comparing it to the subsequent instruction.
    a. Store a byte from R2 and replace "[R1]" with "[R1, #1]" and compare that to when you replace "[R1]" with "[R1], #1".
    b. Store a byte into R2 and replace "[R1]" with "[R1, #1]!" and compare that to when you replace "[R1]" with "[R1], #1".
    c. Store a halfword into R2 and replace "[R1]" with "[R1, #2]" and compare that to when you replace "[R1]" with "[R1], #2".
    d. Store a halfword into R2 and replace "[R1]" with "[R1, #2]!" and compare that to when you replace "[R1]" with "[R1], #2".
16. For each of these, in your lab report, discuss the differences in addressing (pre-, post-, or auto-indexed addressing) and why the contents of memory at the effective address held in R1 results in what it does.
17. Lastly, why is there no signed store instruction? Please answer this in your lab report.

Code Given: Listing 5-2 from Chapter 5 of Smith text.

You have completed this lab when:

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