CS2208b Lab No. 1
Introduction to Computer Organization and Architecture

Tuesday February 11, 2020 (section 8 @ *HSB-13* from 3:30 pm to 4:30 pm)
Wednesday February 12, 2020 (section 6 @ *HSB-16* from 1:30 pm to 2:30 pm)
Wednesday February 12, 2020 (section 10 @ *HSB-16* from 3:30 pm to 4:30 pm)
Thursday February 13, 2020 (section 11 @ *HSB-14* from 11:30 am to 12:30 pm)
Thursday February 13, 2020 (section 4 @ *HSB-14* from 1:30 pm to 2:30 pm)
Thursday February 13, 2020 (section 5 @ *HSB-14* from 3:30 pm to 4:30 pm)
Thursday February 13, 2020 (section 7 @ *HSB-13* from 4:30 pm to 5:30 pm)

The objective of this lab is:

- To get familiar with the Keil ARM simulator and to practice using it.

If you would like to leave, and at least 30 minutes have passed, raise your hand and wait for the TA.
Show the TA what you did. If, and only if, you did a reasonable effort during the lab, the TA will give you the lab mark.

==========

REVIEW

COMPUTER INSTRUCTION SET ARCHITECTURE

An *instruction set architecture*, or ISA, is an abstract model of a computer that describes *what* it does, rather than *how* it does it. You could say that a computer's instruction set architecture is its *functional definition*. Essentially, the ISA is concerned with a computer's *internal storage (e.g., its registers)*, the *operations (i.e., the instruction set)* that the computer can perform on data, and the *addressing modes* used to access data.

REGISTERS

A *register* is a storage device that holds a single data word exactly like a memory location. Registers are physically located in the CPU chip and can be accessed far more rapidly than memory. In principle, there's no fundamental difference between a location in memory and a register. There are just a few registers in a computer, but millions of storage locations in memory. Consequently, you need far fewer bits to specify a register than a memory location.

IMPORTANT POINT

Never confuse the following two concepts: value and address (or location). A memory location holds a value, which is the information stored in that location. The address of an item is where it is in memory, and its value is what it is.

For example, suppose memory location 1234 contains the value 55. If we add 1 to 55, we get 55 + 1, which is 56. That is, we've changed the value of a variable. Now, if we add 1 to the address 1234, we get 1235. That's a different location in memory, which holds a different variable.

The reason for making this point is that it is all too easy to confuse these two concepts because of the way we learn algebra in high school. We use equations like $x = 4$. When we write programs that use variables, the variables usually refer to the locations of data, not to the values. So, when we say $x = 4$, we actually mean that the memory location called x contains the value 4.

QUICK OVERVIEW OF THE ARM

The ARM processor is classified as a 32-bit RISC (*reduced instruction set computer*) with a three-operand register-to-register instruction set. This is just a fancy way of saying that computer operations involve *three* operands in registers such as `ADD r1,r2,r3`. There are a few instructions that have *two* operands and some that have *four*, but that doesn't change the overall classification.

In order to get data into and out of a register (transfers data between memory and register), there are two special instructions called *load* and *store*. *Load* (`LDR`) transfers data from memory to a register, while *store* (`STR`) transfers data from a register to memory. These instructions have the forms `LDR r0,[r1]` and `STR r0,[r1]`, respectively. These instructions use register indirect (i.e., pointer-based) addressing, where the location of the memory element to be accessed is held in a register. The register indirect addressing mode is indicated here by `[r1]`.

The ARM uses another special instruction called ADR (*load register with an address*) that sets up a pointer to a place in the memory. We will practice this instruction in lab number 2.

An ARM instruction like `SUB r3,r2,#4` subtracts the actual value 4 (*remember that the literal is indicated by the # symbol*) from the contents of register `r2` and puts the result in `r3`. Data operations implemented by ARM processors write the destination (result) operand first on the left. We write the destination operand in **bold** font to remind *ourselves* where the result goes.

BYTE OPERATIONS

If you wish to access individual bytes in a 32-bit processor, you need special instructions. RISC processors do not (generally) support 8-bit *data-processing operations* on 32-bit registers, but they do support 8-bit *memory accesses*.
Consider the following ARM examples.

```
LDR  r0,[r1] ;load r0 with the 32-bit contents of memory pointed @ by register r1
LDRB r0,[r1] ;load r0 with the  8-bit contents of memory pointed @ by register r1
             ;the rest of the 3 bytes will be filled by zeros.
```

There is also a similar set of *store* mnemonics with the forms STR, and STRB.

```
STR  r0,[r1] ;store the content      of r0 in memory pointed @ by register r1
STRB r0,[r1] ;store the LSB     byte of r0 in memory pointed @ by register r1
```

USING THE KEIL SIMULATOR

The processor in the PC is not a member of the ARM family. It's usually a member of Intel's IA32 family, IA64 family, or an AMD processor. However, you can run ARM code on your PC using a program from Keil. The Keil package, called µVision4, is very sophisticated and is intended for engineers designing embedded ARM-based systems.
Consequently, it includes far more facilities than we need. The demonstration version that we will use for a PC this term is limited to assembly-level programs smaller than 32K bytes. This restriction is not a problem for students.

If you wish, you can download this Keil package for your PC from the CS2208B course site on OWL.

The Keil package is also installed in all ***Genlabs***.

PROBLEM SET

Before you start practicing this lab, you need to review and fully understand how to use *the Keil ARM simulator*, which is covered in tutorial 4 *(Tutorial_04_Introduction_to_ARM_Simulator.pdf)*.

1. Let's create a very simple example.

```
MOV r1,#2     ;Put  2 in register r1
MOV r2,#-3    ;Put -3 in register r2
ADD r0,r1,r2 ;Add r1 to r2 and put the result in r0
```

   Note how simple the instructions are. Basically, each instruction performs one primitive operation.
   Note also, the semicolon indicates the start of a comment field that is not part of the executable code.

   Use the Keil simulator to write, assemble, and run the above program fragment.
   Do you get the expected answer, especially the negative numbers?
   Note that you will need to use appropriate ***assembly directives*** to make this program fragment a program.

   Modify your program by initializing the values of `r3` and `r4` by literals `-254` and `-65281`, respectively.
   In addition, add together the four values in registers `r1`, `r2`, `r3`, and `r4`. Store the result in `r0`.
   Do not forget to comment on each line of your code. Assemble and run the program. ***Do you get the expected answer?***

2. Write a program that includes *deliberate* syntax errors.
   Enter it in the Keil simulator, assemble (build) it and attempt to fix these errors.

3. Write a simple program to perform: Z = A + B – (C × D). The instructions you may use are `ADD`, `SUB`, and `MUL`. Assume that the data are in registers `r1` to `r4` (representing A to D), and the result will be put in `r0`. You can use `MOV` instruction to initialize `r1` to `r4` by `4, 12, -4,` and `05`, respectively.
   Enter your program into the Keil simulator and run it. Try various values for A, B, C, and D
   Do you get the expected answer?

4. Now assume that A, B, C, D, and Z are 32-bit values in memory. You can initialize them by using a `DCD` directive. Remember, you can use a label to define the *first* location of a memory block. Note that you can put multiple values after the directive `DCD` or `DCB`, but you need to separate the values with commas. However, since each data item needs its own name, you are going to have to use one directive per element; e.g.,

```
A    DCD  4
B    DCD 12
C    DCD -4
D    DCD -5
Z    DCD  0
```

   Modify your program in Q3 to load (`LDR`) the values of `r1` to `r4` using these A, B, C, and D values, respectively. You also need to store (`STR`) the result in location Z. ***Do you get the expected answer?***

   To see the content of a memory segment, you need to open a memory window. To do this, *while you are in the debugging mode*, click on **View,** then select **Memory Window** and then **Memory 1**. Enter the starting address in the **Address** box in the memory window (which is 0) and hit return. In the memory window, we can see both the *code* and *data*. Note that you may need to resize the memory window, by dragging the edge of the memory window to display as many or as few bytes per line as you want.

   In Keil simulator, memory locations are *read-only*, by default. To change some of these locations to *read-write*, we have to perform an additional step. *Click* the **Debug** and then the **Memory Map** tabs. *Enter* a memory map range in the **Map Range box** and *check the box* **Read,** and **Write,** *as needed*, and finally *click* on the **Map Range button.** The format of the range is defined as `0x000, 0x200`. This range is from memory location `0x000` to memory location `0x200`.
   This step must be performed each time you enter the debugging mode.
   Without this step, you will ***NOT*** be able to successful *store* data in memory.

5. Now assume that A, B, C, D, and Z are 8-bit values in memory. You can initialize them by using a `DCB` directive.
   Modify your code accordingly in Q4 to deal with these byte variables.
   Do you get the expected answer? Think of the sign extend situation.