CS2208b Lab No. 2
Introduction to Computer Organization and Architecture

Monday February 22, 2016 (section 3 @ *SSC-1032* from 2:30 pm to 3:30 pm)
Tuesday February 23, 2016 (section 4 @ *SSC-1032* from 1:30 pm to 2:30 pm)
Tuesday February 23, 2016 (section 8 @ *HSB-14* from 3:30 pm to 4:30 pm)
Thursday February 25, 2016 (section 5 @ *SSC-1032* from 2:30 pm to 3:30 pm)
Friday February 26, 2016 (section 6 @ *SSC-1032* from 1:30 pm to 2:30 pm)
Friday February 26, 2016 (section 7 @ *SSC-1032* from 3:30 pm to 4:30 pm)

The objective of this lab is:

- To practice ARM assembly ***pseudo*** instructions

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, he/she will give you the lab mark.

==========

REVIEW

ARM pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM instructions at assembly time. ARM pseudo-instructions include:

LDR ARM pseudo-instruction

The LDR pseudo-instruction loads a register with either:

- a 32-bit constant value
- an address

Syntax

The syntax of LDR is:

LDR{**condition**} **register**,=[**expression** | **label-expression**]

where:

condition is an optional condition code,
register is the register to be loaded,
expression evaluates to a numeric constant:

- If the value of **expression** is within range of a MOV or MVN instruction, the assembler generates the appropriate instruction.
- If the value of **expression** is **not** within range of a MOV or MVN instruction, the assembler places the constant in a ***literal pool*** and generates a program-relative LDR instruction that reads the constant from the literal pool.
  The offset from the PC to the constant must be less than 4KB. You are responsible for ensuring that there is a literal pool within range.

label-expression is a program-relative expression.
The assembler places the value of **label-expression** in a literal pool and generates a program-relative LDR instruction that loads the value from the literal pool.
The offset from the PC to the value in the literal pool must be less than 4KB. You are responsible for ensuring that there is a literal pool within range.

Usage

The LDR pseudo-instruction is used for two main purposes:

- to generate literal constants when an **immediate** value cannot be moved into a register because it is out of range of the MOV and MVN instructions.
- to load a program-relative address into a register.

Example

```
LDR r1,=0xfff        ; loads 0xfff into r1
LDR r2,=place        ; loads the address of place into r2
```

ADR ARM pseudo-instruction

The ADR pseudo-instruction loads a program-relative or register-relative address into a register.

Syntax

The syntax of ADR is:

ADR{**condition**} **register**,**expression**

where:

condition is an optional condition code,
register is the register to load,
expression is a program-relative or register-relative expression that evaluates to an address.
The address can be either before or after the address of the instruction or the base register.

Usage

ADR always assembles to one instruction. The assembler attempts to produce a single ADD or SUB instruction to load the address.

Example

```
start MOV r0,#10
      ADR r4,start          ; => SUB r4,pc,#0xc
```

PROBLEM SET

1. The ARM uses a pipeline to increase the speed of the flow of instructions to the processor. This allows several operations to take place simultaneously. Instructions are executed in three pipeline stages: fetch, decode, and execute. During normal operation, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory. Hence, when an ARM processor starts executing an instruction, the PC will not be pointing to that instruction any more.
   Consider the following instruction:
   ```
   MOV r0, pc
   ```
   Use the Keil simulator to write, assemble, and run the above program fragment.
   Note that you will need to use appropriate assembly directives to make this program fragment a program.
   What did you find the value of r0 after executing the instruction? Justify your finding..

2. Consider the following instructions:
   ```
   LDR r1, [r0]
   LDR r2, = 0x12345678
   LDR r3, = 0x12
   ```
   Use the Keil simulator to write, assemble, and run the above program fragment.
   Note that you will need to use appropriate assembly directives to make this program fragment a program.

   Study and relate the generated disassembled code to the code written above. Specifically, you need to understand the role of the PC register and the location of the literal pool, as well as the pipeline effect, in this situation.

   Why does each `LDR` instruction encode differently?

3. Consider the following instructions:
   ```
        LDR r3, X
        LDR r4, =X
        ADR r5, X
   loop B   loop
   X    DCD 0x70707070
   ```
   Use the Keil simulator to write, assemble, and run the above program fragment.
   Note that you will need to use appropriate assembly directives to make this program fragment a program.

   Study and relate the generated disassembled code to the code written above. Specifically, you need to understand the role of the PC register and the location of the literal pool, as well as the pipeline effect, in this situation.

4. Consider the following assembly programs:
   ```
        AREA prog1, CODE, READONLY
        ENTRY
        LDR r0, = 0x12345678
   loop b loop
        AREA prog1, READONLY
   X    DCD 0x70707070
        END

        AREA prog2, CODE, READONLY
        ENTRY
        LDR r0, = 0x12345678
   loop b loop
   X    DCD 0x70707070
        END
   ```
   Use the Keil simulator to run the above two programs.
   Study and compare the generated disassembled code for the `LDR` instruction in each program.
   Justify the difference between the two generated codes. Think of the location of the literal pool.