CS2208b Lab No. 2
Introduction to Computer Organization and Architecture

Tuesday February 25, 2020 (section 8 @ *HSB-13* from 3:30 pm to 4:30 pm)
Wednesday February 26, 2020 (section 6 @ *HSB-16* from 1:30 pm to 2:30 pm)
Wednesday February 26, 2020 (section 10 @ *HSB-16* from 3:30 pm to 4:30 pm)
Thursday February 27, 2020 (section 11 @ *HSB-14* from 11:30 am to 12:30 pm)
Thursday February 27, 2020 (section 4 @ *HSB-14* from 1:30 pm to 2:30 pm)
Thursday February 27, 2020 (section 5 @ *HSB-14* from 3:30 pm to 4:30 pm)
Thursday February 27, 2020 (section 7 @ *HSB-13* from 4:30 pm to 5:30 pm)

The objective of this lab is to practice:
- o ARM data definition directives
- o 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, the TA 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

Note that, the LDR instruction can also be used as non-pseudo-instruction, e.g., LDR r1,[r2]

Syntax

The syntax of LDR *when it is used as pseudo-instruction* 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 the range of a MOV or MVN instruction, the assembler generates the appropriate MOV or MVN instruction to perform the task.
- If the value of the **expression** is **not** within the range of a MOV or MVN instruction, the assembler places the constant in a ***literal pool*** and generates a program-counter-relative LDR instruction that reads the constant from the literal pool.
  The offset from the PC to the constant ***must be less than 4KB***.

label-expression is a program-counter-relative expression.
The assembler places the value of **label-expression** (i.e., an address) in a literal pool and generates a program-counter-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***.

Usage

The LDR pseudo-instruction is used for two main purposes:
- to load a literal constant to a register when an **immediate** value cannot be moved into the register because it is out of range of the MOV and MVN instructions, which must be represented by a value from 0 to 255 and a rotation.
- to load a program-counter-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 uses to load a lable address into a register.

Syntax
The syntax of ADR is:
ADR{**condition**} **register**, **lable expression**
where:

condition is an optional condition code,
register is the register to load,
lable expression is to evaluate to an address.
The address can be either before or after the address of the instruction.

Usage
ADR always assembles to one instruction. The assembler attempts to produce a single ADD or SUB instruction to load the address. The distance between the address in the lable expression and the ADR instruction MUST be represented as a value from 0 to 255 and a rotation.

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

PROBLEM SET

Before you start practicing this lab, you need to review and fully understand tutorials 6 and 7 *(Tutorial_06_ARM_Data_Definition_Directives.pdf and Tutorial_07_ARM_Pseudo Instructions).*

1. Consider the following assembly programs:

```
            AREA More_data_definitions, CODE, READONLY
            ENTRY
loop        B    loop

data_1 SPACE 3
data_2 SPACE 3
       ALIGN
data_3 SPACE 3
       DCD   0x12345678
       DCD   +2_1111000011110000
       DCD   -2_1111000011110000

       DCW   255
       DCW  -255

       DCB   &0A
       ALIGN

       DCD  1,2,3,4
       DCB  5
       DCD  6

       END
```

Addresses	1st byte	2nd byte	3rd byte	4th byte	Comments
0x00000000	0xEA	0xFF	0xFF	0xFE	B instruction encoding
0x00000004	0x00	0x00	0x00	0x00	3-0'd Bs (data_1), 1-0'd B (data_2)
0x00000008	0x00	0x00	0x00	0x00	2-0'd Bs (data_2), 1-0'd B (ALIGN)
0x0000000C	0x00	0x00	0x00	0x00	3 0'-d Bs (data_3), 1-0'd B (for DCD)
0x00000010	0x12	0x34	0x56	0x78	DCD 0x12345678
0x00000014	0x00	0x00	0xF0	0xF0	sign extend arg of DCD
0x00000018	0xFF	0xFF	0x0F	0x10	take 2's complement and sign extend
0x0000001C	0x00	0xFF	0xFF	0x01	255 sign extended; -255 sign extended
0x00000020	0x0A	0x00	0x00	0x00	1 B for DCB arg, and 3-0's B (ALIGN)
0x00000024	0x00	0x00	0x00	0x01	DCD first arg (1)
0x00000028	0x00	0x00	0x00	0x02	DCD second arg (2)
0x0000002C	0x00	0x00	0x00	0x03	DCD third arg (3)
0x00000030	0x00	0x00	0x00	0x04	DCD forth arg (4)
0x00000034	0x05	0x00	0x00	0x00	1 B for DCB arg, and 3-0's B (for DCD)
0x00000038	0x00	0x00	0x00	0x06	4 B for DCD
0x0000003C					

The above program consists of one instruction (the machine code of this branch instruction is "0xEAFFFFFE") and a bunch of data definition directives.

<u>Manually calculate</u> the memory map for the *<u>entire program</u>*, i.e., mention the address location and the content of each memory location of this program. Use this information to fill the table above. **<u>Verify your calculations</u>** by assembling the above program and compare the generated machine language code with your calculations.

2. 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 anymore.
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 findings.

[r0] = 0x00000008 This is because when the instruction at 0x00 is executed, memory location 0x04 is being decoded, and memory location 0x08 is fetched.

3. Consider the following instructions:

```
    LDR r1, [r0]
    LDR r2, = 0x12345678
    LDR r3, = 0x12
```

```
0x00000000  E5901000  LDR     R1,[R0]
0x00000004  E59F2000  LDR     R2,[PC]
0x00000008  E3A03012  MOV     R3,#0x00000012
0x0000000C  12345678  EORNES  R5,R4,#0x07800000
```

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.

1. Real Instruction   2. Pseudo-Instruction: 0x12345678 is outside MOV range, so it is placed in literal pool

3. Pseudo-Instruction: In MOV range, generated using shift.

Literal Pool is always placed after END directive.

Why is each LDR instruction encoded differently?

4. Consider the following instructions:

```
      LDR r3, X
      LDR r4, =X
      ADR r5, X
loop  B   loop
X     DCD 0x70707070
```

```
0x00000000  E59F3008  LDR      R3,[PC,#0x0008]
0x00000004  E59F4008  LDR      R4,[PC,#0x0008]
0x00000008  E28F5000  ADD      R5,PC,#0x00000000
0x0000000C  EAFFFFFE  B        0x0000000C
0x00000010  70707070  RSBVCS   R7,R0,R0,ROR R0
0x00000014  00000010  ANDEQ    R0,R0,R0,LSL R0
```

1. X in literal pool at 0x10 is accessed by [PC] + 0x08, so [r3] <- [X], then PC is updated to [PC] + 8, which is 0x0C.

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.

2. address(X), i.e. 0x10, in literal pool at 0x14 is accessed by [PC] + 0x08, so [r4] <- 0x10

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.

3. R5 takes the sum of the PC pointer (currently at 0x10) and 0, so it stores 0x10, so [R5] <- 0x10.

5. Consider the following assembly programs: The program was run and the result matches the explanation.

```
      AREA prog1, CODE, READONLY
      ENTRY
      LDR r0, = 0x12345678
loop B loop

      AREA prog1, DATA, 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.

6. Consider the following assembly program:

```
      area program, code, readonly
      ENTRY
N     EQU 4
      LDR r0,=X1
loop B loop
      SPACE N
X1    DCD   0x12345678
      END
```

```
0x00000000  E59F0008  LDR      R0,[PC,#0x0008]
0x00000004  EAFFFFFE  B        0x00000004
0x00000008  00000000  ANDEQ    R0,R0,R0
0x0000000C  12345678  EORNES   R5,R4,#0x07800000
0x00000010  0000000C  ANDEQ    R0,R0,R12
```

result: [r0] <- 0x0C

explanation: The pseudo instruction LDR r0, =X1 is implemented as an instruction. It does [R0] <- [PC + 0x08], i.e. [0x10], which is 0x0C, as expected.

Use the Keil simulator to write, assemble, and run the above program fragment.
You need to understand how the "LDR r0,=X1" instruction is implemented.

What is the largest possible N that you can have in this program without having any errors? ***Why?***
Verify your answer by editing, assembling, and running the program.

maximum possible N is 4088 because the literal pool becomes out of reach after 4KB.
In this case, we have 4KB = 2^12 = 4096
Now, adding that 4088 bytes of zero means LDR will jump exactly 4088+8 = 4096 = 4KB, the maximum possible.

Q5

```
0x00000000  E59F0000  LDR      R0,[PC]
0x00000004  EAFFFFFE  B        0x00000004
0x00000008  12345678  EORNES   R5,R4,#0x07800000
0x0000000C  70707070  RSBVCS   R7,R0,R0,ROR R0
```

result: [r0] <- 0x12345678

explanation: in the first instruction, R0 <- [PC], i.e. R0 <- [0x08], and [0x08] = 0x12345678, as expected.

```
0x00000000  E59F0004  LDR      R0,[PC,#0x0004]
0x00000004  EAFFFFFE  B        0x00000004
0x00000008  70707070  RSBVCS   R7,R0,R0,ROR R0
0x0000000C  12345678  EORNES   R5,R4,#0x07800000
```

result: [r0] <- 0x12345678
explanation: in the first instruction, R0 <- [PC + 0x04], i.e. R0 <- [0x0C], and [0x0C] = 0x12345678, as expected.

Note: In the second program, 0x70707070 is placed first in the literal pool. I hypothesize it is due to the placement of DCD in the program. Since it is in the same chunk as the code, it is placed in the pool first because it invokes DCD explicitly.