CS2208b Lab No. 3
Introduction to Computer Organization and Architecture

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

The objective of this lab is to practice:
- ARM shift operations
- ARM addressing modes
- ARM instructions encoding

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
ADDRESSING MODES

An addressing mode is simply a means of *expressing the location of an operand*. An address can be a register such as `r3` or `PC` (*Program Counter*). An address can be a location in memory such as address `0x12345678`. You can even express an address *indirectly* by saying, for example, "the address is loaded in register `r1`". The various ways of expressing the location of data are called collectively *addressing modes*.

Suppose someone said, "Here are ten dollars". They are giving you the actual item. This is called a **literal** or **immediate** value because it is what you actually get. Unlike all other addressing modes, you do not have to retrieve addresses from a register or memory location.

If someone says, "Go to this full address in the world and you will find the money on the table", they are actually telling you *where* the money is (i.e., its address or its actual location). This is called an **absolute address** because it expresses exactly where the money is in absolute terms. This addressing mode is also called **direct addressing**. ARM processors do ***not*** support this addressing mode.

Now here is where the fun starts. Suppose someone says, "Go to room 12 *in this building* and you will find something to your advantage on the table". You arrive in room 12 and see a message on the table saying, "The money is in *this full address in the world"*. In this case, we have an **indirect address** because room 12 does not have the money, but a pointer to where it is. We have to go to a second address to get the money. Indirect addressing is also called **pointer-based** addressing because you can think of the note in room 12 as pointing to the actual data.

In real life, we do not confuse a room number or an address with a sum of money. However, in a computer, all data is stored in binary form and the programmer has to remember whether a variable (or constant) is an address or a data value.

Immediate (literal) addressing is indicated by a '`#`' symbol in front of the operand. Thus, `#5` in an instruction means the actual value 5. A typical ARM instruction is `MOV r0,#5` which means move the value 5 into register `r0`.

Indirect addressing is indicated by ARM processors by placing the pointer in square parentheses; for example, `[r1]`. All ARM indirect addresses are of the basic form `LDR r0,[r1]` or `STR r3,[r6]`.
There are variations on this addressing mode; for example, `LDR r0,[r1,#4]` specifies an address that is four bytes from the location pointed at by the contents of register `r1`. It can also have a side effect, such as *autoindexing pre-indexed addressing mode*, e.g., `LDR r0,[r1,#4]!` or *autoindexing post-indexed addressing mode*, e.g., `LDR r0,[r1],#4`
In all these indirect addressing modes, the offset can be a constant, i.e., static (as indicated in the above examples), or *dynamic*, by putting the value of the offset in a register, e.g.,

```
LDR r0,[r1,r2]
LDR r0,[r1,r2]!
LDR r0,[r1],r2
```

All offsets associated with indirect addressing (regardless of constant or dynamic) can be positive or negative.

PROBLEM SET

Before you start practicing this lab, you need to review and fully understand tutorials 8 and 9 *(Tutorial_08_ARM_Shift_Instructions.pdf and Tutorial_09_ARM_Addressing_Modes.pdf).*

1. Consider the following assembly program:

```
      AREA prog1, code, READONLY
      ENTRY
0x00  MOV r3,#2    [r3] <- 2
0x04  LDR r1, =0xCCCCCCCC ;in binary 1100 1100 1100 1100 1100 1100 1100 1100
                           [r1] <- 1100 1100 1100 1100 1100 1100 1100 1100
0x08  LSL r1,r1,#5         [r1] <- 1001 1001 1001 1001 1001 1001 1000 0000   0x99999980
0x0C  LSL r1,r1,r3         [r1] <- 0110 0110 0110 0110 0110 0110 0000 0000   0x66666600
0x10  LSR r1,r1,#10        [r1] <- 0000 0000 0001 1001 1001 1001 1001 1001   0x00199999
0x14  LSR r1,r1,r3         [r1] <- 0000 0000 0000 0110 0110 0110 0110 0110   0x00066666
0x18  ASR r1,r1,#2         [r1] <- 0000 0000 0000 0001 1001 1001 1001 1001   0x00019999
0x1C  LSL r1,r1,#15        [r1] <- 1100 1100 1100 1100 1000 0000 0000 0000   0xCCCC8000
0x20  ASR r1,r1,#16        [r1] <- 1111 1111 1111 1111 1100 1100 1100 1100   0xFFFFCCCC

0x24  ASR r1,r1,r3         [r1] <- 1111 1111 1111 1111 1100 0011 0011 0011   0xFFFFF333

0x28  ROR r1,r1,#4         [r1] <- 0011 1111 1111 1111 1111 1111 0011 0011   0x3FFFFF33
0x2C  ROR r1,r1,r3         [r1] <- 1100 1111 1111 1111 1111 1111 1100 1100   Carry = 1   0xCFFFFFCC
0x30  RRX r1,r1            [r1] <- 0110 0111 1111 1111 1111 1111 1110 0110   Carry = 0   0x67FFFFE6
0x34  RRX r1,r1            [r1] <- 0011 0011 1111 1111 1111 1111 1111 0011   Carry = 0   0x33FFFFF3
0x38  RRX r1,r1            [r1] <- 0001 1001 1111 1111 1111 1111 1111 1001   Carry = 1   0x19FFFFF9
0x3C  RRX r1,r1            [r1] <- 0000 1100 1111 1111 1111 1111 1111 1100   Carry = 1   0x0CFFFFFC
0x40 loop B   loop
      END
```

Note: Since the last instruction sits at 0x40, it must be the case that 0xCCCCCCCC is stored at 0x44 (i.e. in the literal pool).

The answer has been verified and it is correct!

Manually calculate the value of `r1` after executing each instruction. **Verify your calculations** by assembling the above program and compare the program output with your calculations.

2. The following assembly program adds together a LIST of five numbers stored in memory.

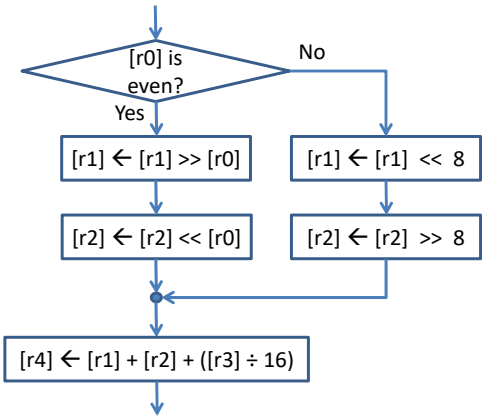
```
        AREA Pointers, CODE, READONLY
        ENTRY
Start   ADR  r0,List    ;register r0 points to List
        MOV  r1,#5      ;initialize loop counter in r1 to 5
        MOV  r2,#0      ;clear the sum in r2
Loop    LDR  r3,[r0]    ;copy the element pointed at by r0 to r3
        ADD  r0,r0,#4   ;point to the next element in the series
        ADD  r2,r2,r3   ;add the element to the running total
        SUBS r1,r1,#1   ;decrement to the loop counter
        BNE  Loop       ;repeat until all elements added
Endless B    Endless    ;infinite loop
List    DCD  3,4,3,6,7 ;the numbers to be added together
                        ;each one is 4 bytes (20 bytes in total)
        END
```

(a) Modify the original program to utilize:
- Autoindexing pre-indexed addressing mode See Assembly_Lab_3_Q2_Source-1V1.s
- Autoindexing post-indexed addressing mode See Assembly_Lab_3_Q2_Source-1V2.s

(b) Modify the original program by eliminating the instruction "`ADD r0,r0,#4`", while keeping the functionality of the program the same, without using any autoindexing mode. Think of utilizing the loop counter instead.

See Assembly_Lab_3_Q2_Source-1.s

3. Using only 7 ARM assembly instructions, execute the following flowchart.
Test your code by assigning various values to `r0`, `r1`, `r2`, and `r3`.




```
AREA prog3, code, READONLY
ENTRY
MOV r0, #123    ; Load a value to r0
MOV r1, #212    ; Load a value to r1
MOV r2, #99     ; Load a value to r2
MOV r3, #12     ; Load a value to r3

TST r0, #1     ; Test the parity of r0.
LSREQ r1, r1, r0   ; If r0 is even, logically shift r1 right by r0.
LSLEQ r2, r2, r0   ; If r0 is even, logically shift r2 left by r0.
LSLNE r1, r1, #8   ; If r0 is odd, logically shift r1 right by 8.
LSRNE r2, r2, #8   ; If r0 is odd, logically shift r2 left by 8.

ADD r4, r1, r2    ;[r4] <- [r1] + [r2]
ADD r4, r4, r3, ASR #3  ;[r4] <- [r4] + [r3]/16

loop  B loop
END
```

4. Without using any of the ARM multiplication instructions, write only one ARM instruction that executes [R4] ← 16385 × [R4].
Manually generate the machine language code for this instruction and verify it using the simulator.

```
ADD r4, r4, r4, LSL #14
```
E0844704 The answer has been verified and it is correct!

5. What is the machine language of `AND r2,r3,#0x1080`
Verify it using the simulator. 0xE3A0407B The answer has been verified and it is correct!

6. What is the reverse assembly of the following machine language instruction:
(a) `0x00000000`
(b) `0x50076808`
(c) `0xE3A01D22`
(d) `0xE3A01E88`
Verify it using the simulator. Hint: you may want to consult Table 3.2 and figure 3.26 in the textbook.

(a)

0000 0000 0000 0000 0000 0000 0000 0000
Bits 27-26 are 00 so it is a data processing instruction
Bits 31-28 are 0000, so condition is EQ
Bit 25 = 0, which means a register operand
Bits 24-21 are 0000, which means AND
Bit 20 (S) = 0, which means do NOT set the flags
Bits 19-16 are 0000, which means source1 is r0
Bits 15-12 are 0000, which means destination is r0
Bit 4 = 0, which means a static shift operation
Bits 11-7 are 00000, which means the shift length is 0
Bits 6-5 are 00, which means it is a LSL type of shift
Bits 3-0 are 0000, which means the shift is applied to r0

right

```
ANDEQ R0, R0, R0, LSL #0
ANDEQ R0, R0, R0
```
Equivalent

(b)

0101 0000 0000 0111 0110 1000 0000 1000
Bits 27-26 are 00 so it is a data processing instruction
Bits 31-28 are 0101, so condition is PL
Bit 25 = 0, which means a register operand
Bits 24-21 are 0000, which means AND
Bit 20 (S) = 0, which means do NOT set the flags
Bits 19-16 are 0111, which means source1 is r7
Bits 15-12 are 0110, which means destination is r6
Bit 4 = 0, which means a static shift operation
Bits 11-7 are 10000, which means the shift length is 16
Bits 6-5 are 00, which means it is a LSL type of shift
Bits 3-0 are 1000, which means the shift is applied to r8

right

```
ANDPL R6, R7, R8, LSL #16
ANDPL R6, R7, R8
```
Equivalently

(c)

1110 0011 1010 0000 0001 1101 0010 0010
Bits 27-26 are 00 so it is a data processing instruction
Bits 31-28 are 1110, so condition is AL
Bit 25 = 1, which means a literal operand
Bits 24-21 are 1101, which means MOV
Bit 20 (S) = 0, which means do NOT set the flags
Bits 19-16 are 0000 (IGNORE FOR MOV)
Bits 15-12 are 0001, which means destination is r1
Bits 11-8 are 1101, which means the alignment is 13
Bits 7-0 are 00100010, which means it is 34
Therefore, the literal is 2^(32 - 13*2)*34 = 2176

```
MOV r1, #2176
```
right

(d)

1110 0011 1010 0000 0001 1110 1000 1000
Note that the first (upper) 5 bytes are the same as the last question.
Therefore, we have the same form, i.e.: MOV r1, r0, #???

0xE88 = 1110 1000 1000
Bits 11-8 are 1110, which means the alignment is 14
Bits 7-0 are 10001000, which means it is 136
Therefore, the literal is 2^(32 - 14*2)*136 = 2176

```
MOV r1, #2176
```

right (just weird)