2.4 Assembly Programming Exercise – Computing Average

Note to students: *Some important concepts to bring across are:*

1. *Understand how consecutive elements in an array can be accessed.*
2. *Understand how predetermine looping cycles can be implemented using a countdown counter.*
3. *Understand how unsigned and signed division by 2^n can be implemented using the logical and arithmetic shift right operation.*
4. *Understand how the use of appropriate logical operators can mask out unwanted high-order bits.*
5. *Understand the efficient use of conditional execution instructions to avoid branching.*

Write an ARM assembly language program to compute the average value of **eight** numbers in an array.

(1) Your program should be written based on the following specifications:

a) These numbers are word-sized unsigned integers (i.e. **32-bit unsigned** numbers).

b) The eight numbers are in consecutive memory locations starting at address **`0x0100`**.

c) The average (quotient) and remainder are stored in registers **`R0`** and **`R1`** respectively.

Note: You are free to use the partial completed VisUAL ARM assembly program template ***Tutorial 2_4-Template*** to test out your answer.

Suggested solution:

```
Start    MOV   R1,#0x100       ; initialize array pointer to start of array
         MOV   R0,#0           ; initialize cumulating register R0 to zero
         MOV   R2,#8           ; initialize loop counter for 8 loops
SumLoop  LDR   R3,[R1],#4      ; get current element and increment array pointer
         ADD   R0,R0,R3        ; add array element to current sum in R0
         SUBS  R2,R2,#1        ; decrement loop counter R2 (check Z flag)
         BNE   SumLoop         ; branch back until count is 0

GetQnR   MOV   R1,R0           ; make copy of cumulated sum in R1
         AND   R1,R1,#0x07     ; clear all bits except lowest 3 bits to get remainder in R1
         MOV   R0,R0,LSR #3    ; logical shift right by 3 bits to divide by 8 to get quotient
```

Note the following:

1. The register **`R1`** is used as an address pointer to the array of numbers. It is first initialized to **`0x100`**, the start address of the number array. The **`LDR`** instruction uses **`R1`** as an indirect register with index (and post increment) to retrieve each 32-bit number each time around the loop. 4 is added to the content in **`R1`** because each number (4 bytes) is stored in 4 consecutive memory locations.
2. The register **`R2`** is the decrementing loop counter which is initialized to 8 at the start to do 8 loops. **`BNE`** will continue to loop back to the **`SumLoop`** label until the **`SUBS`** instruction sets the Z flag.
3. Once the sum of all 8 numbers have been computed in register **`R0`**, a copy of this is made in **`R1`** in order to compute the remainder in **`R1`**. The remainder of any division by 8 (i.e. 2^3) is the least significant 3 binary bits of original number before division. This is obtained by using the **`AND`** operation to mask all high order bits to 0, except bits 0 to 2.
4. The positive quotient is obtained in **`R0`** by doing a logical right shift (**`LSR`**) by 3 bits.

(2) How would your program be changed if these numbers are **32-bit signed** numbers instead?

Suggested solution:

For signed numbers, Arithmetic Shift Right (**ASR**) can be used. Arithmetic right shifts are equivalent to logical right shifts (**LSR**) for positive signed numbers. Arithmetic right shifts for negative numbers (2's complement) is similar in concept to division by a power of 2^n but the negative quotients with non-zero remainder will round towards negative infinity instead of towards 0 as is usually expected (e.g. –28 / 8 gives a quotient of –4 instead of –3, because –3.5 is rounded toward –4 and not –3. As such, additional steps need to be taken to handle this rounding behaviour when using **ASR** when a negative quotient is detected.

Method #1 – Convert to positive to compute quotient and then negate quotient again

```
GetQnR  MOVS   R1,R0          ; make copy of cumulated sum in R1 (and check N flag)
        RSBMI  R0,R0,#0       ; if negative, negate to positive
        AND    R1,R1,#0x07    ; clear all bits except lowest 3 bits to get remainder in R1
        MOV    R0,R0,LSR #3   ; logical shift right by 3 bits to divide by 8
        RSBMI  R0,R0,#0       ; if quotient was earlier negative, negate to negative again
```

Note: **MOVS** is used to check if the sum is negative. If the sum is negative, then the conditional execution version of **RSBMI**, which will execute only if the N flag is set (i.e. condition is MInus). Under this situation, **RSB** will negate the sum to its positive equivalent so that the quotient can be computed as in the unsigned example earlier. However, the positive quotient must be reverted to if negative equivalent with the second **RSBMI**. Remember **RSB** unlike **SUB** implements reverse subtraction, which is (0 – R0).

Method #2 – Use ASR but add #1 to negative quotient if there is a non-zero remainder

```
GetQnR  MOV    R1,R0          ; make copy of cumulated sum in R1
        MOVS   R0,R0,ASR #3   ; arithmetic shift right by 3 bits to divide by 8
        ADDMI  R0,R0,#1       ; add 1 if quotient is -ve to round quotient towards zero
        ANDS   R1,R1,#0x07    ; clear all bits except lowest 3 bits to get remainder in R1
        SUBEQ  R0,R0,#1       ; if remainder is zero, undo earlier add 1 operation quotient
```

Note: **ASR** is used to implement an arithmetic shift right, which will handle the sign of the quotient. However, if the quotient is negative, **ASR** will only give us a negative quotient that is rounded towards negative infinity. In this case, 1 is added to the negative quotient to get a rounding towards zero. This is done by the **ADDMI** instruction, which will only execute if the computed quotient is negative. However, this should only be done if there is a non-zero remainder. If there is no remainder, the addition of 1 will give us an incorrect answer. The **SUBEQ** instruction will subtract the negative quotient by 1 if the computed remainder done using the **ANDS** instruction yields a zero (i.e. Z flag set).