(To be done over 2 week)

2.1 ARM Addressing Modes, their Characteristics and Applications

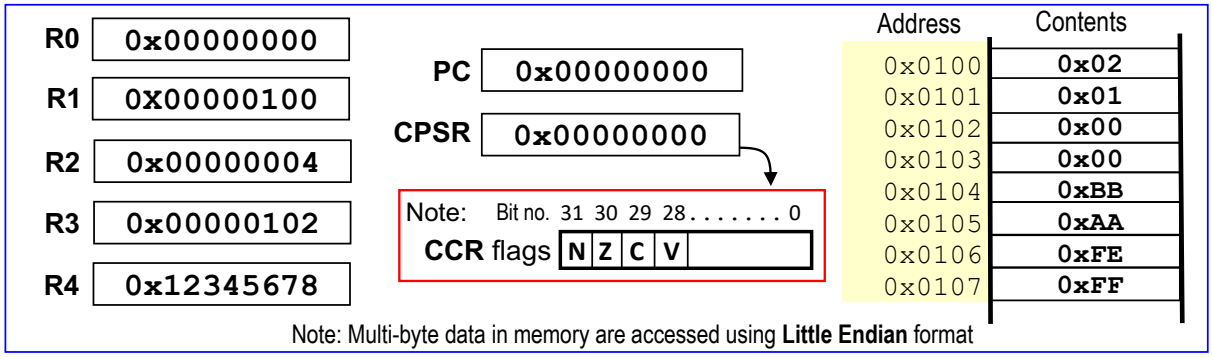

Figure 1 – Initial state of some ARM processor registers and memory contents

(1) Give the name of the source operand addressing mode for each mnemonic in Table 1.

(2) Give the 32-bit content in register **R0** immediate after the execution of each instructions in Table 1. (**Note:** Each execution is independent of the others and uses the same initial values shown in Figure 1 and Little Endian byte-ordering is assumed).

No.	Mnemonics	Source addressing mode	Content in R0
1.	`MOV R0,R2`	Register Direct	R2 → 0x00000004
2.	`MOV R0,#0x0100`	Immediate Addressing	0x0100 ✗ 0x00000100
3.	`LDR R0,[R1]`	Register Indirect	0x00000102
4.	`LDR R0,[R1,#4]`	Base plus ~~Register Indirect with~~ Offset pre index	0xFFFEAABB
5.	`LDR R0,[R1,R2]`	Base plus ~~Register Indirect with~~ Index Register pre index	0xFFFEAABB
6.	`LDR R0,[R1,#4]!`	Offset with AutoIndexing, pre index	0xFFFEAABB
7.	`LDR R0,[R1],#4`	Offset with Autoindexing, post index	0x00000102
8.	`LDR R0,[R1,R2]!`	Index Register with Autoindex,pre index	0xFFFEAABB
9.	`LDR R0,[R1],R2`	Index Register with Autoindexing, post index	0x00000102

Table 1 – The different ARM addressing modes and its effects after execution

(3) Using the initial values shown in Figure 1, give the ARM instructions that will move the 32-bit value of **`0x00000102`** into register **R0** using the following addressing modes:

(a) Register direct (b) Immediate (c) Register indirect

Note: If necessary, you may use more than one instruction to achieve this requirement.

(4) With reference to the three addressing modes (a) to (c) in part (3), describe which addressing mode you would use for the following operations and why:

(a) Fetch the next element of an integer array into a register for subsequent comparison.

(b) Copy a frequently used 32-bit constant value into another temporary register.

(c) Initialise a loop counter register with a small value that does not exceed 255.

(5) Suggest how you can move the 32-bit content in address location **`0x0100`** to **`0x0104`**. You may assume the initial values in the registers are shown in Figure 1.

(6) How can the memory copy operation specified in part (5) be done efficiently for 7 memory locations withing a loop structure, as shown in Figure 2? Assume the initial values in the registers are shown in Figure 1. Do not concern yourself with the code constructs for the loop structure, only the body of the loop that does the memory copy for contents in **0x0100** to **0x0104** , **0x0104** to **0x0108** and so on.

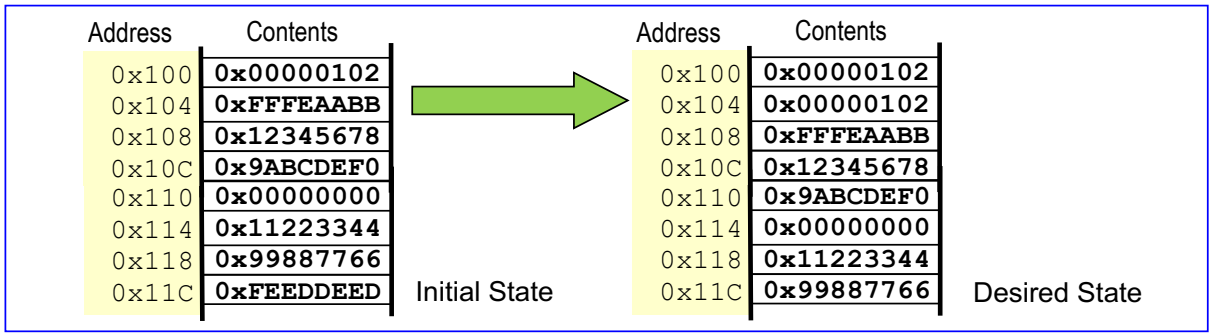

Figure 2 – The initial and desired end result of the memory transfer operation.

(7) Give the ARM assembly code to swap the 32-bit values in registers **R2** and **R3**. Do the same for the 32-bit values in address locations **0x0100** to **0x0104**?

To be covered only if time permits.

(8) Describe the characteristics of the **Full Descending** (FD) stack. Give the instruction to push register **R0** to a FD stack which is maintained by the stack pointer (**SP**) and pop a word from a FD stack into register **R1**.
(**Note**: operations on the FD stack are relevant to the chapter on Modular Programming)

2.2 Arithmetic and Logical Instructions

(1) Give **four** different ways to **clear** register **R0** to **0x00000000** using only a single ARM mnemonic.

(2) You are required to **set** bits 0, 4, 8, 12, 24 and 28 in register **R2** while leaving all other bits unchanged. For example, if **R2**=**0x03468ACF**, then after execution, the new value after execution should be **R2**=**0x13469BDF**.

Give the sequence of ARM mnemonics to achieve this. See if you can do this using only one additional register besides the target register **R2**.

(3) With reference to your solution in question 2.1 part (7), did you have to use an additional register to swap the registers **R2** and **R3**? Consider the situation where there are no spare registers left for you to use. How can one of the logical instructions be used to swap the registers **R2** and **R3** without the use of an additional register.

(4) With the aid of the Shift operation, give the ARM instruction(s) that will multiply the **signed** content in register **R0** by the constant value **9**.

(i) Give the largest positive signed 32-bit value in **R0** that will not result in an overflow after this operation.

(ii) Suggest how this idea could be extended to multiply the 32-bit content in register **R0** with the constant value **7**.

2.3 ARM Assembly Program Analysis – Comparison and Counting

Fig. 3 shows an ARM assembly language program that scans through a sample of average daily temperature readings stored consecutively as unsigned 8-bit numbers in data memory starting at address **0x102** onwards. Two byte-sized memory variables **HotDays** and **DaySum** are located at addresses **0x100** and **0x101** respectively. The end of valid temperature readings in the array is detected when the decimal value of -128 is encountered.

Note: Replicate the program in Fig. 3 using the partial completed VisUAL ARM assembly program template ***Tutorial 2_3-Template***. Analyse the operation of the program by executing the code. All memory values shown in Fig. 3 will be pre-loaded into their respective memory locations by the given DCD directives.

(1) Based on the hexadecimal memory contents shown in Fig. 3, compute the hexadecimal contents in registers **R0**, **R1**, **R2**, **R3**, **R4** and byte-sized memory variables **HotDays** and **DaySum** at the end of program execution. Assume execution begins at instruction label **Start** and ends at the program termination directive **END**.

(2) Describe the operation of the assembly language program in Fig. 3 by filling in appropriate comments for each line of instruction? In brief, what does this program do?

(3) Modify the program to speed up its execution. You can evaluate how well you have optimized your code by the reduction in clock cycle count whilst still getting the same results in all the registers and memory variables **HotDays** and **DaySum**. The given program takes 127 clock cycles to execute but it is possible to reduce it to 93 cycles!

(4) It was observed that if temperature samples had negative values (e.g. -1 or **0xFF**), the results obtained were incorrect. Could you suggest a reason for this error and how it can be rectified in the program shown in Fig. 3?

```
Mem_100     DCD   0x3224FFFF    ;
Mem_104     DCD   0x22282317
Mem_108     DCD   0x17208013
Mem_10C     DCD   0x2D142580
Start       MOV   R1,#0x100     ;
            MOV   R4,#0x100     ;
            MOV   R0,#0         ;
            STRB  R0,[R4]       ;
            ADD   R1,R1,#2      ;
Loop        LDRB  R3,[R1]       ;
            CMP   R3,#0x80      ;
            BEQ   Done          ;
            ADD   R0,R0,#1      ;
            LDRB  R3,[R1]
            CMP   R3,#36        ;
            BHS   HotFound      ;
            ADD   R1,R1,#1      ;
            B     Loop          ;
HotFound    LDRB  R2,[R4]       ;
            ADD   R2,R2,#1      ;
            STRB  R2,[R4]       ;
            ADD   R1,R1,#1      ;
            B     Loop          ;
Done        ADD   R4,R4,#1
            STRB  R0,[R4]       ;
            END
```

Address	Contents
0x100	0xFF
0x101	0xFF
0x102	0x24
0x103	0x32
0x104	0x17
0x105	0x23
0x106	0x28
0x107	0x22
0x108	0x13
0x109	0x80
0x10A	0x20
0x10B	0x17
0x10C	0x80
0x10D	0x25
0x10E	0x14
0x10F	0x2D

Memory map of data area

Figure 3 – An ARM assembly language program and some contents in memory

B in LDRB used to copy one byte only, 8bit

Not necessary to be covered during tutorial classes.

The following questions are for your self-practice and may not be covered during tutorial classes. Suggested solutions to these problems will be uploaded to the NTULearn site about a week after the tutorial.

2.4 Assembly Programming Exercise – Computing Average

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.

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

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

2.5 Program Counter related Addressing Modes

Figure 4 shows an ARM assembly program and the starting address of various incomplete instructions in code memory. Complete the mnemonics **M1** to **M5** based on their respective comments and ensure that all your solutions support **position-independent code**. You may use the partially completed VisUAL ARM assembly program template ***Tutorial 2_5-Template*** to test out your answers.

Note: The **PC** points 8 addresses ahead during the execution of each instruction as this is a consequence of how the ARM processes instructions using "pipeline" techniques.

Address	Mnemonics or Hexa Data	Comments
0x000	MOV R0,#0	; Clear R0
0x004	? PC,?,?	; **M1** - Relative Jump to instruction at address 0x00C (label Addr_0C)
0x008	MOV R1,R0	; Dummy instruction
0x00C	ADD R1,?,?	; **M2** - Get start address Var_100 into R1 in a PC-relative manner
0x010	LDR R0,?	; **M3** - Move the content at memory variable Var_104 into R0.
0x014	ADD R2,?,?	; **M4** - Get start address of next instruction into R2.
0x018	MOV ?,?	; **M5** - Create an infinite loop with this instruction.
:	:	
0x100	0x01234567	; Var_100 - Constant stored in data memory at address 0x100
0x104	0x89ABCDEF	; Var_104 - Constant stored in data memory at address 0x104

Figure 4 – Various ARM mnemonics and their respective start addresses in code memory