CG2028 COMPUTER ORGANIZATION

TUTORIAL 2: ARM V7-M ASSEMBLY LANGUAGE

WEEK 9, GU JING

Assembly Language

* Not frequently used, but for critical part of program, e.g. core part of operating system where requires <1ms running time.

* Program part written by assembly language allows fully control of microprocessor, the ASM instructions are executed by the processor are completely defined by the programmer, not by the compiler (which translate higher level language e.g. C program to different format of ASM depending on the compiler settings).

* **Types of ASM Instruction**:

  * **Memory Instructions**: LDR, STR (processor and memory may or may not physically separated)

  * **Data Processing Instructions**: ADD, CMP, AND, ROR etc

  * **Branch Instructions**: IT-block, B etc

Q1: Which of the following ARM instructions will cause the assembler to issue a syntax error message? Why?

(a) ADD R2, R2, R2

(b) SUB R0, R1, [R2,#4]

(c) MOV R0, NUM1

(d) LDR R0, R1

(e) ADDS R0, R1, R2, LSL #3

(f) BLXAL LOL

(g) MVNS R1, R1

(h) EOR R1, R1, R1

(i) RRX R1, R1

(j) TEQS R1, R2

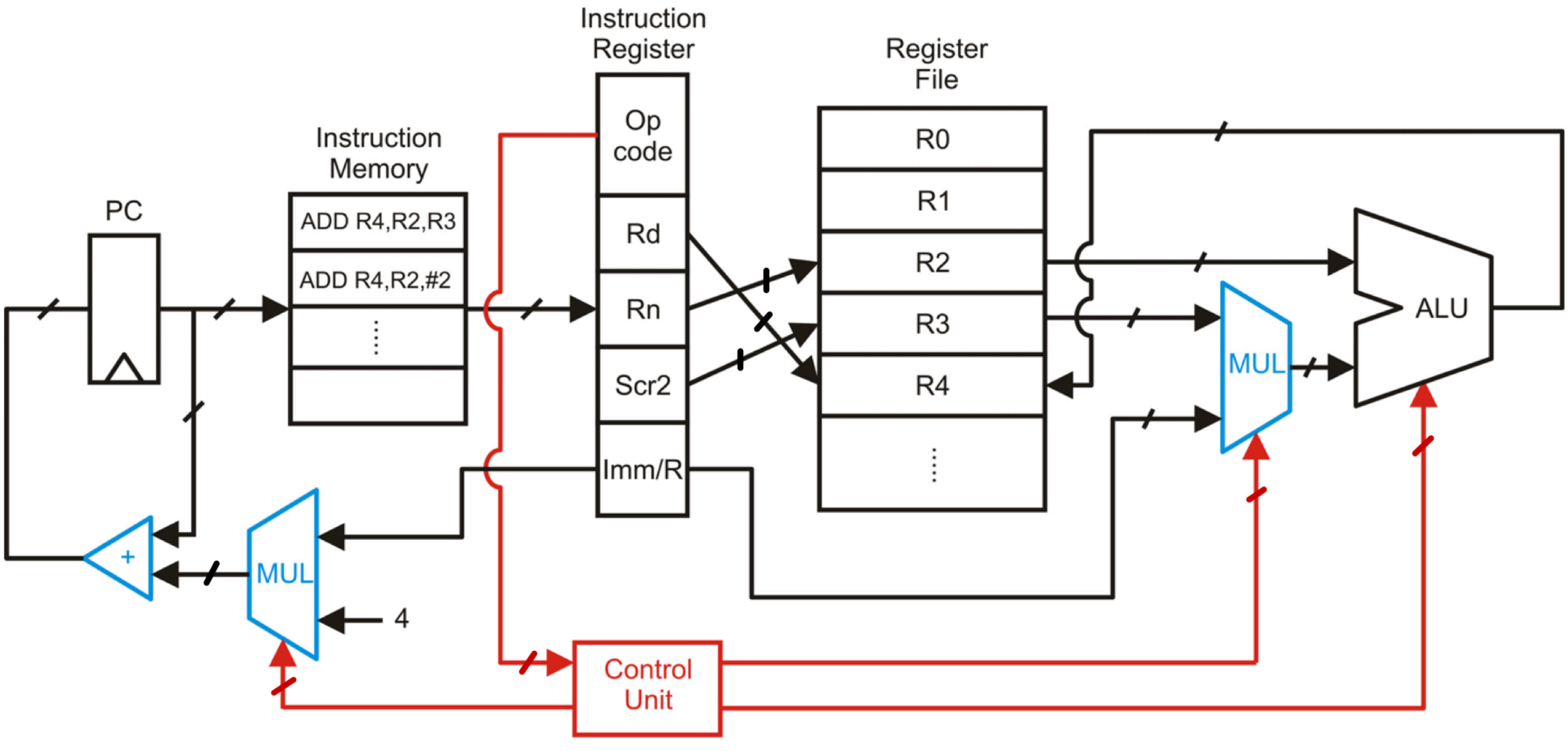
(k) PUSH {R0-R16}

(l) POPS {R0, R1, R2}

Q1: (a) ADD R2, R2, R2

Ans: Correct.

Rn and Scrs2 are taken out together for ALU calculation, then Rd is updated. So it is fine that Rd, Rn and Scrs2 are the same register.



Q1: (b) SUB R0, R1, [R2,#4]

Ans: Wrong.

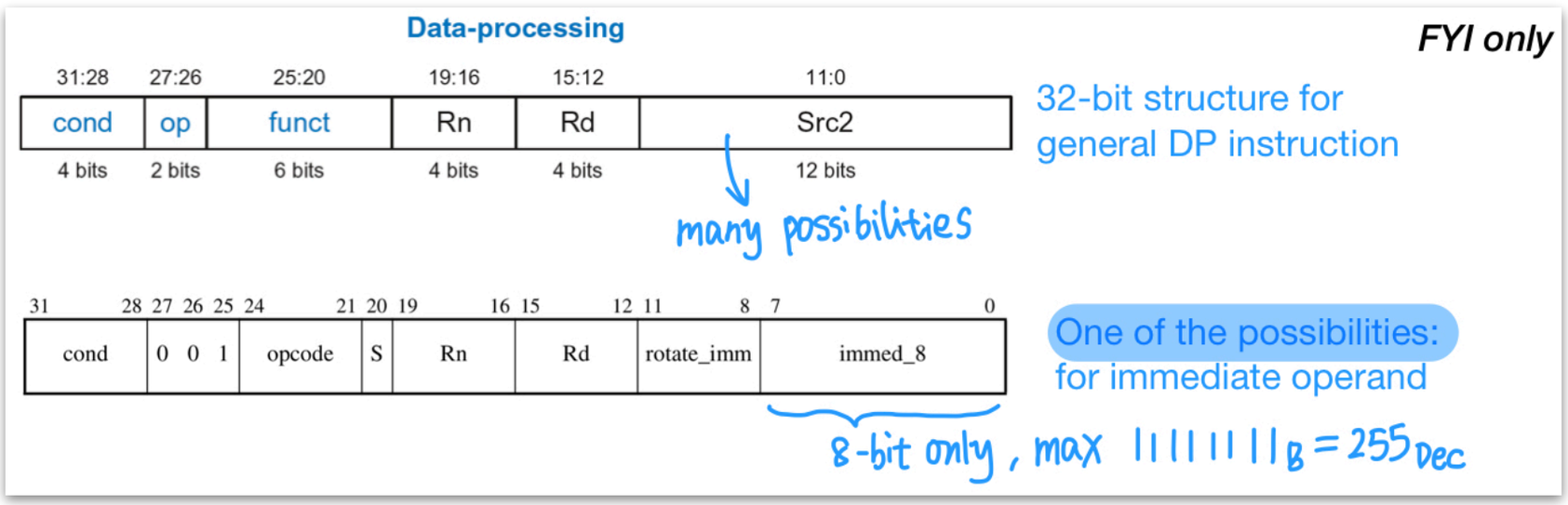
[R2, #4] is a memory operand, which cannot be referenced in a Subtraction instruction.

Q1: (c) MOV R0, NUM1

Ans: Wrong.

This instruction should follow format <MOV Rd, operand2>, where operand2 should be a register or Imm8 in this case.

The immediate value should within a range too, only 0-255 is allowed. E.g."MOV R0, #257" may cause an error. If immediate value is >255, "MOVW Rd, <imm16>" needs to be used.



Q1: (d) LDR R0, R1

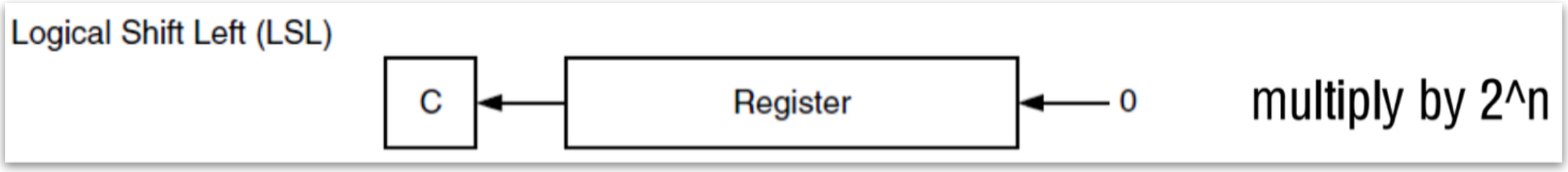
Ans: Wrong.

LDR is a memory instruction that used to access memory, instead of copying the value within the processor. MOV needs to be used in this case.

Q1: (e) ADDS R0, R1, R2, LSL #3

Ans: Correct.

Logical shift left R2 by 3 bits, then add with R1, the result is stored in R0.



Q1: (f) BLXAL LOL

Ans: Wrong.

AL is conditional suffix standing for "always". LOL here is a label referring to certain instruction line. But correct format should be BLX*{cond} Rm*

Branch Format	Operations	Application
B{cond} label	PC <= label	Jump without return, e.g. go back to main program, conditional loops
BX{cond} Rm	PC <= Rm	
BL{cond} label	LR<= PC, then PC<= label	Jump and remember the return location, e.g. subroutine / function
BLX{cond} Rm	LR<= PC, then PC<= Rm	

Q1: (g) MVNS R1, R1

Ans: Correct.

MVNS takes the value of Rm and performs a bitwise logical negate operation on the value, and places the result into Rd. Flags are modified accordingly.

Q1: (h) EOR R1, R1, R1

Ans: Correct.

Exclusive OR operation. R1 will be 0 when EOR-ed same pattern.

Q1: (i) RRX R1, R1

Ans: Correct. Rotate Right eXtended.

Q1: (j) TEQS R1, R2

Ans: Wrong. Test instruction updates conditional flags without the need to specify the S suffix. "S" is not a valid option in this instruction.

Q1: (k) PUSH {R0-R16}

Ans: Wrong.

In ARM V7-M, there are only 16 directly-accessible processor registers, R0 to R15.

Q1: (l) POPS {R0, R1, R2}

Ans: Wrong.

The S suffix is invalid in PUSH and POP instructions.

Q2: Assume the following register and memory contents in an ARM embedded system. What is the effect of executing each of the following instructions with the given initial values?

Main Memory	
0x00000100	**LDR R8, [R0]**
0x00000104	**LDR R9, [R0, #4]**
0x00000108	**ADD R10, R8, R9**
:	:
0x00001000	1
0x00001004	2
0x00001008	3
0x0000100C	4
0x00001010	5
0x00001014	6
:	:

Registers	
PC	
:	
R0	0x1000
R1	0x2000
R2	0x1010
R3	0x300
:	:
R8	0x400
R9	0x500
R10	0x600
:	:

Q2: **LDR R8, [R0]**

@ load data from memory location 0x1000 to R8

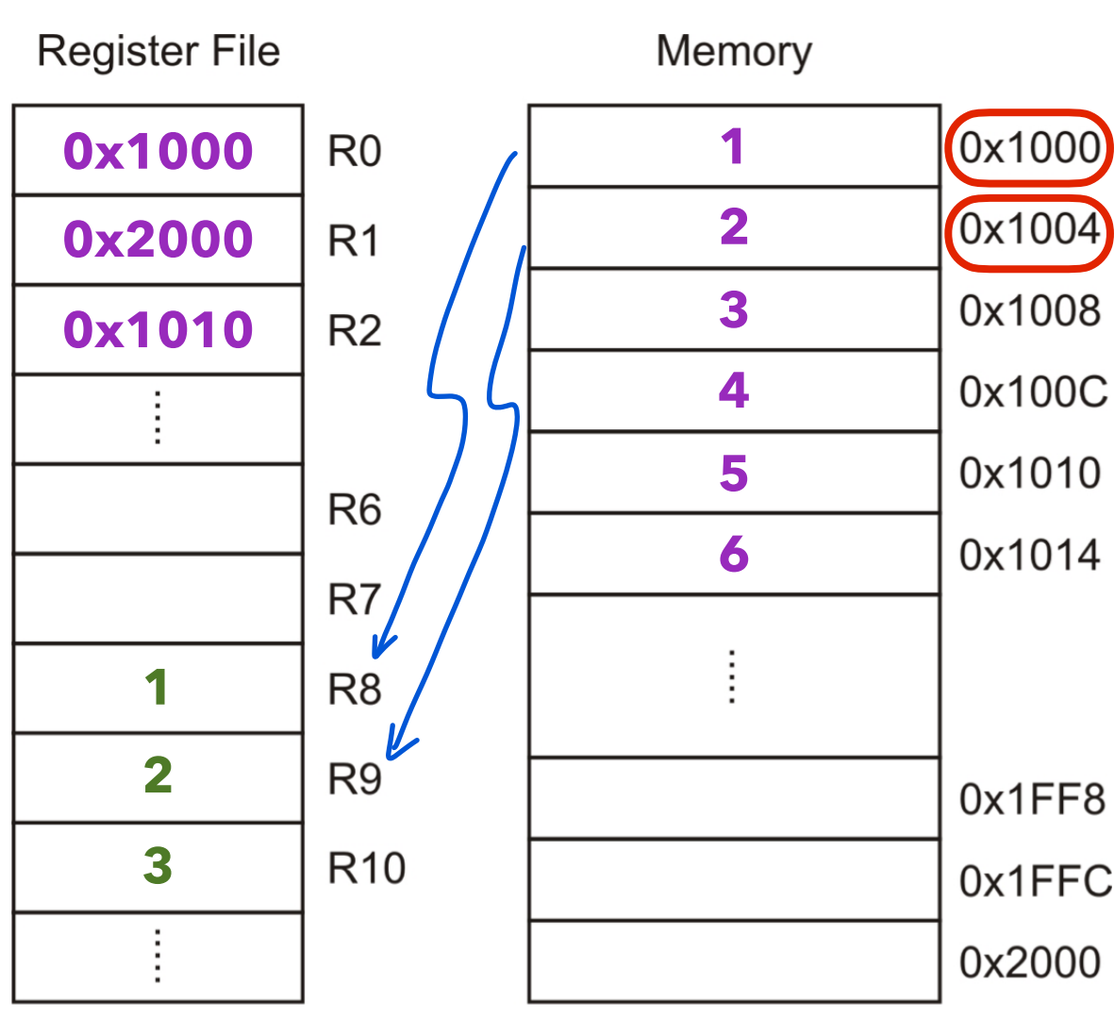
LDR R9, [R0, #4]

@ offset address loading

@ load data from memory location 0x1004 to R9

ADD R10, R8, R9

@ R8 + R9 => R10



Q3: Assume the following register and memory contents in an ARM embedded system. What is the effect of executing the following instructions with the given initial values?

Main Memory	
0x00000100	**STR R6, [R1, #-4]!**
0x00000104	**STR R7, [R1, #-4]!**
0x00000108	**LDR R8, [R1], #4**
0x0000010C	**LDR R9, [R1], #4**
0x00000110	**SUB R10, R8, R9**
:	:
0x00001000	0
0x00001004	1
:	:
0x00001FF8	1022
0x00001FFC	1023
0x00002000	1024

Registers	
R0	0x1000
R1	0x2000
R2	0x1010
R3	0x300
R4	0x200
R5	10
R6	20
R7	30
R8	0x400
R9	0x500
R10	0x600
:	:

Q3:

Main Memory	
0x00000100	**STR R6, [R1, #-4]!**
0x00000104	**STR R7, [R1, #-4]!**
0x00000108	**LDR R8, [R1], #4**
0x0000010C	**LDR R9, [R1], #4**
0x00000110	**SUB R10, R8, R9**
:	:
0x00001000	0
0x00001004	1
:	:
0x00001FF8	1022
0x00001FFC	1023
0x00002000	1024

Registers	
R0	0x1000
R1	0x2000
R2	0x1010
R3	0x300
R4	0x200
R5	10
R6	20
R7	30
R8	0x400
R9	0x500
R10	0x600
:	:

Q3:

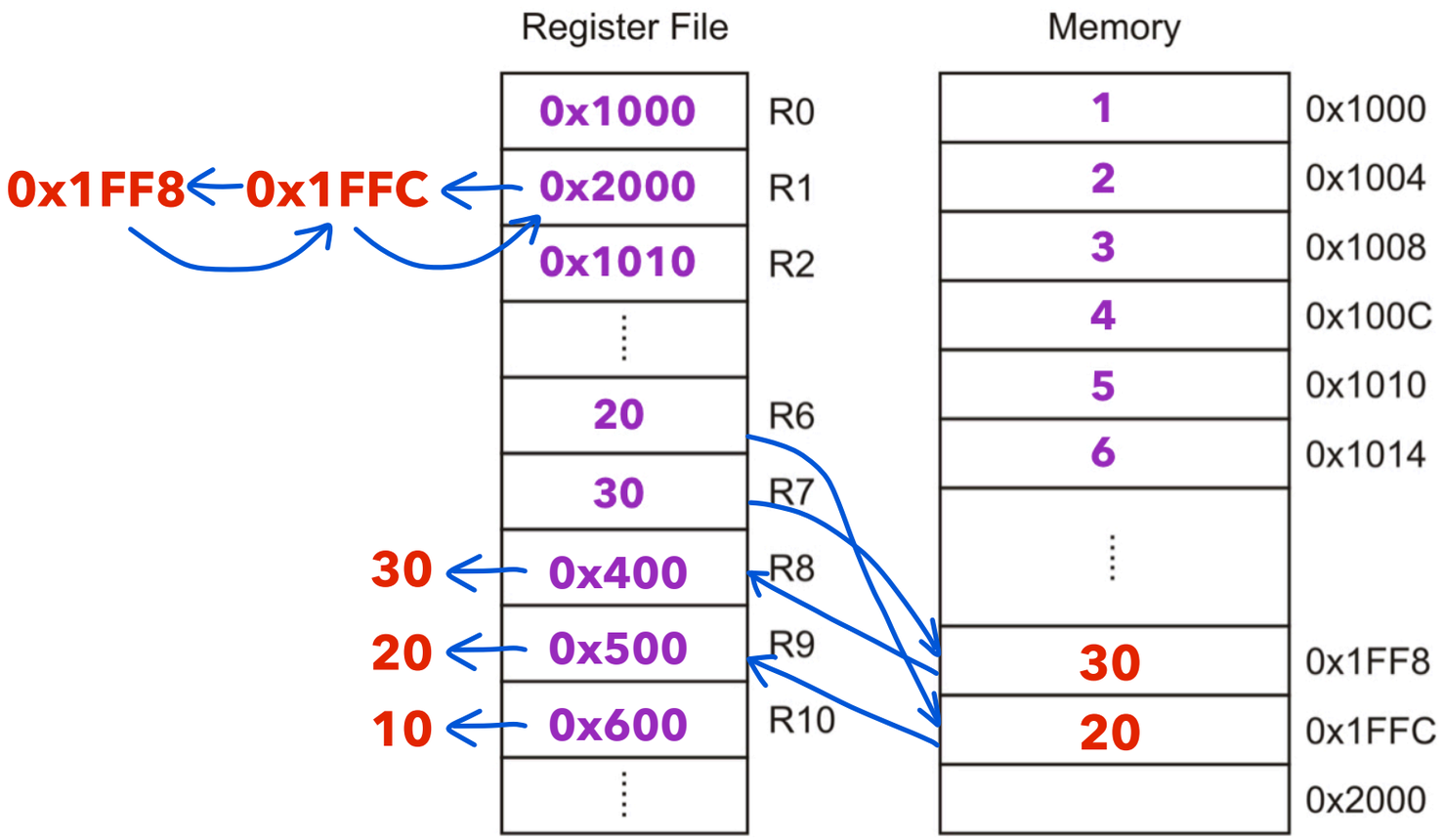
STR R6, [R1, #-4]! @R1-4=2000-4=1FFC, so store 20 to 1FFC location

STR R7, [R1, #-4]! @R1-4=1FFC-4=1FF8, so store 30 to 1FF8 location

LDR R8, [R1], #4 @R1 is 1FF8, so copy value from 1FF8 location to R8, and increase R1 by 4

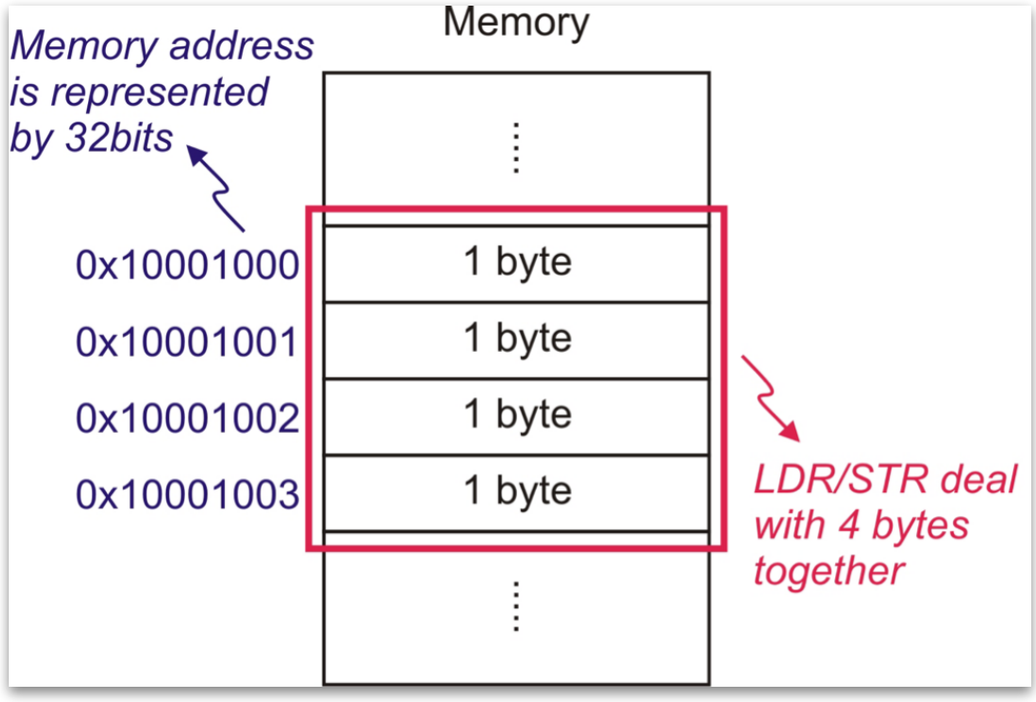
LDR R9, [R1], #4 @R1 is 1FFC, so copy value from 1FFC location to R9, and increase R1 by 4

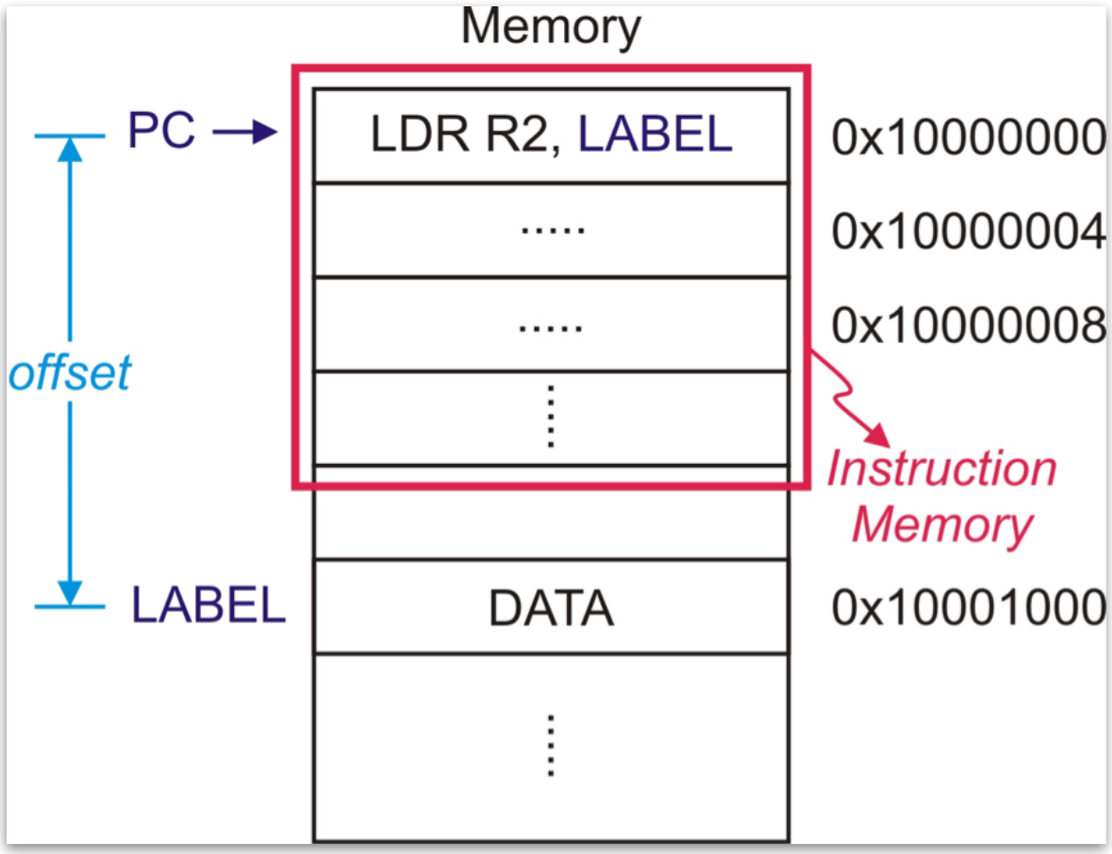
SUB R10, R8, R9 @Subtract R9 from R8 and store the result into R10



Addressing Modes:

* Offset mode of address:
  LDR R2, [R3, #offset]
  * Pre-indexing: LDR R2, [R3, #offset]!
  * Post-indexing: LDR R2, [R3], #offset
* PC-relative mode: LDR R2, LABEL
* Pseudo-instruction: LDR R2, =LABEL





PC-relative mode: equivalent to offset addressing mode, the effective address is implicitly Program Counter (PC, or R15), offset is calculated automatically.

LDR R2, LABLE

=> LDR R2, [R15, #offset]

Addressing Modes:

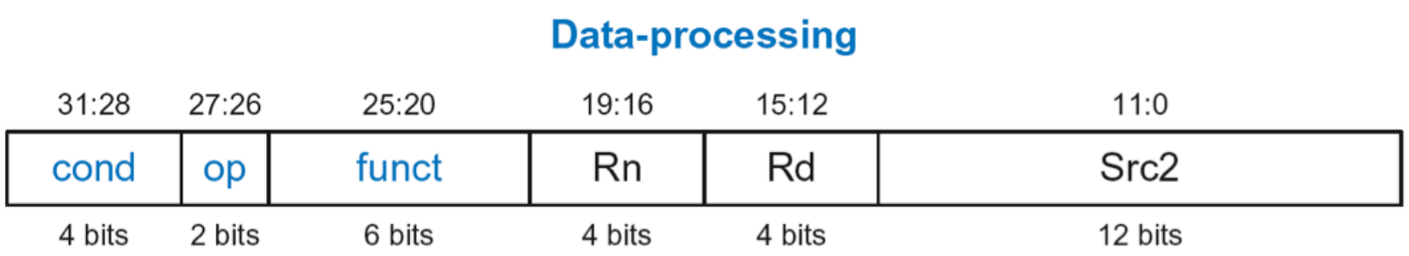
Discussion 1: how to load value 0xABCD1234 into R3?

Discussion 2: assume the data 0xABCD1234 is stored in the memory, how to get its address?

Addressing Modes:

Discussion 1: how to load value 0xABCD1234 into R3?

MOV R3, #0xABCD1234 is wrong. The entire instruction needs to be 32-bit, therefore immediate Src2 cannot be 0xABCD1234.



32-bit structure for DP instruction

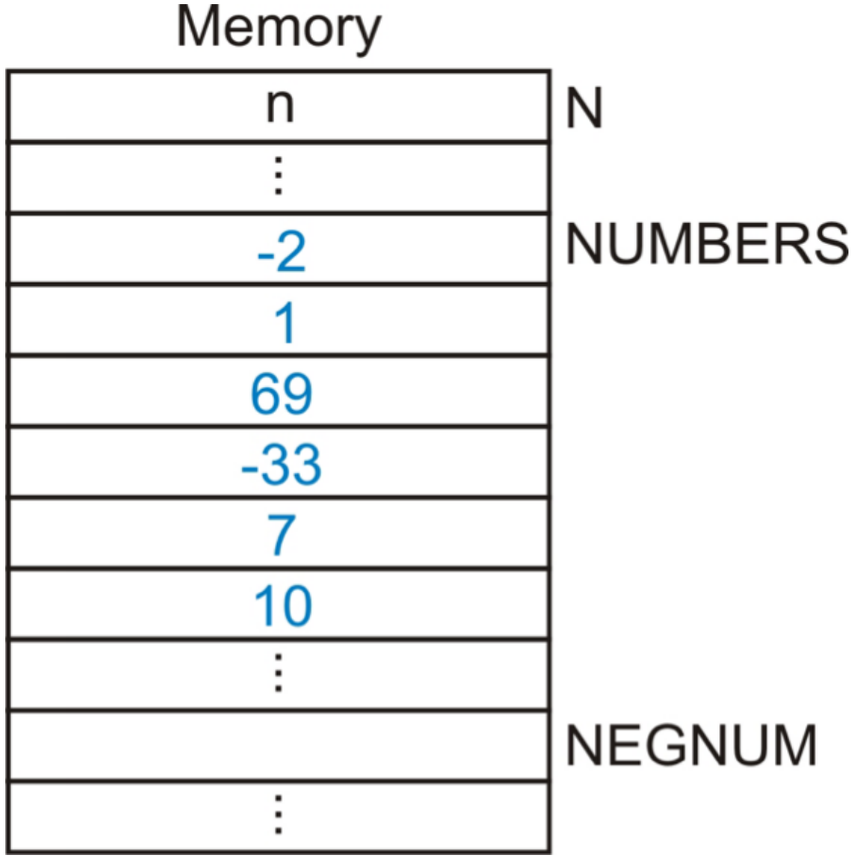
```
LABLE .word 0xABCD1234 @store the constant into memory
LDR R3, LABEL   @load the constant from memory by offset addressing
```

```
LDR R3, =0xABCD1234 @pseudo code to achieve R3=0xABCD1234
```

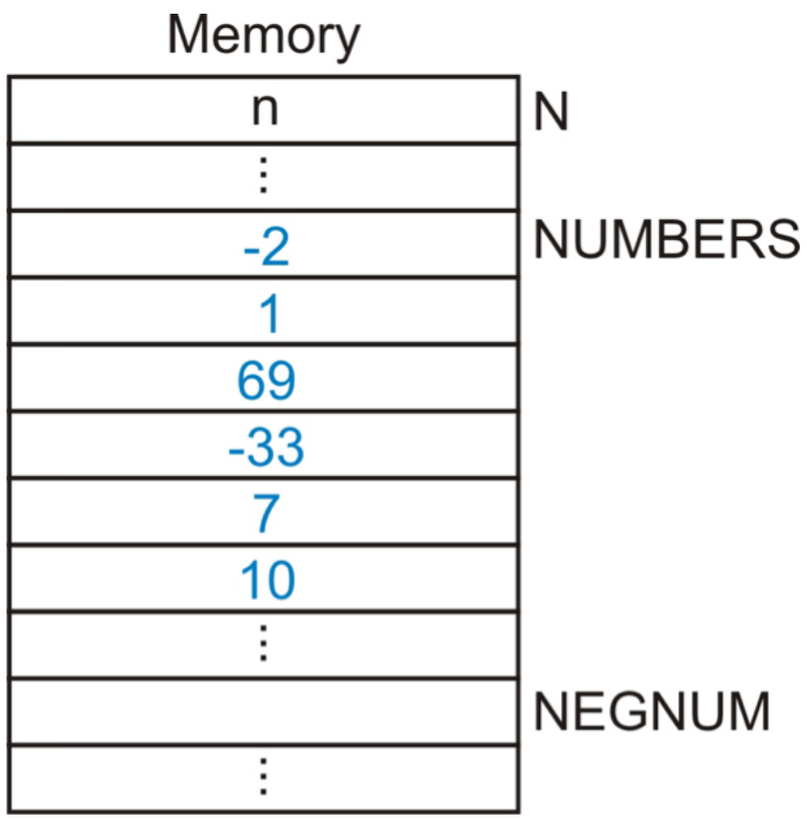
Discussion 2: assume the data 0xABCD1234 is stored in the memory, how to get its address?

```
LABLE .word 0xABCD1234 @store the constant into memory
LDR R3, =LABEL   @load the address of LABEL to R3
```

Q4: Write an ARM program that finds the number of negative integers in a list of n 32-bit integers and stores the count in location NEGNUM. The value n is stored in memory location N, and the first integer in the list is stored in location NUMBERS.
Hint: use the IF-THEN (IT) block.



Q4:



@ R0: count no. of negative values

@ R1: address of the data in the array / memory

@ R2: index number n

@ R3: value that is checking

@ R4: address of NEGNUM

Q4:

(1) Initiate the count of negative values (set R0 to 0)

(2) Get the memory address of 1st number into R1

(3) Get the counter value n

(4) LOOP:

(4.1) get the value from memory & update R3

(4.2) check the value (-ve or +ve)

(4.3) if +ve: add 1 to the count; else, do nothing

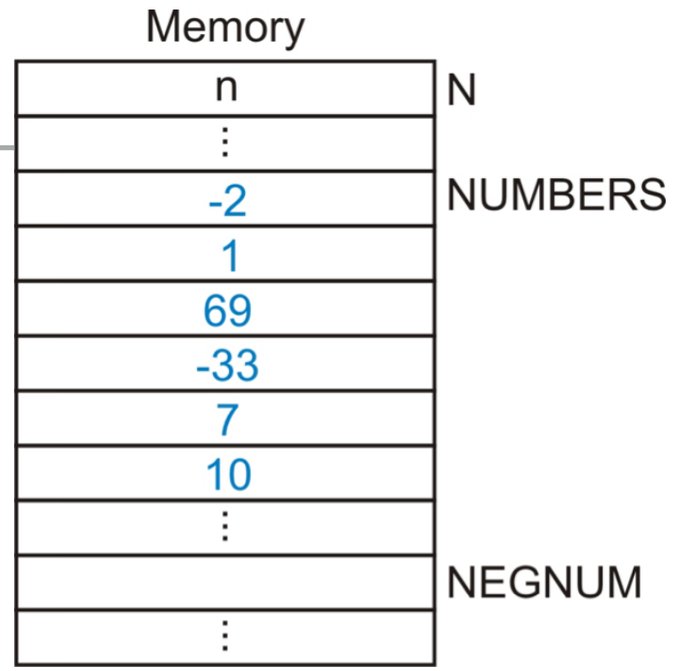
(4.4) reduce the counter value by 1

(4.5) go back to LOOP if the counter value is not zero yet.

(5) Store the final count into memory

(5.1) get the address of NEGNUM

(5.2) store final count (R0 value) to the memory



@ R0: count no. of negative values

@ R1: address of the data in the memory

@ R2: index number n

@ R3: value that is checking

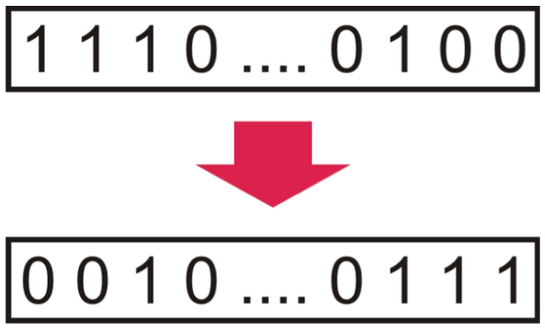
@ R4: address of NEGNUM

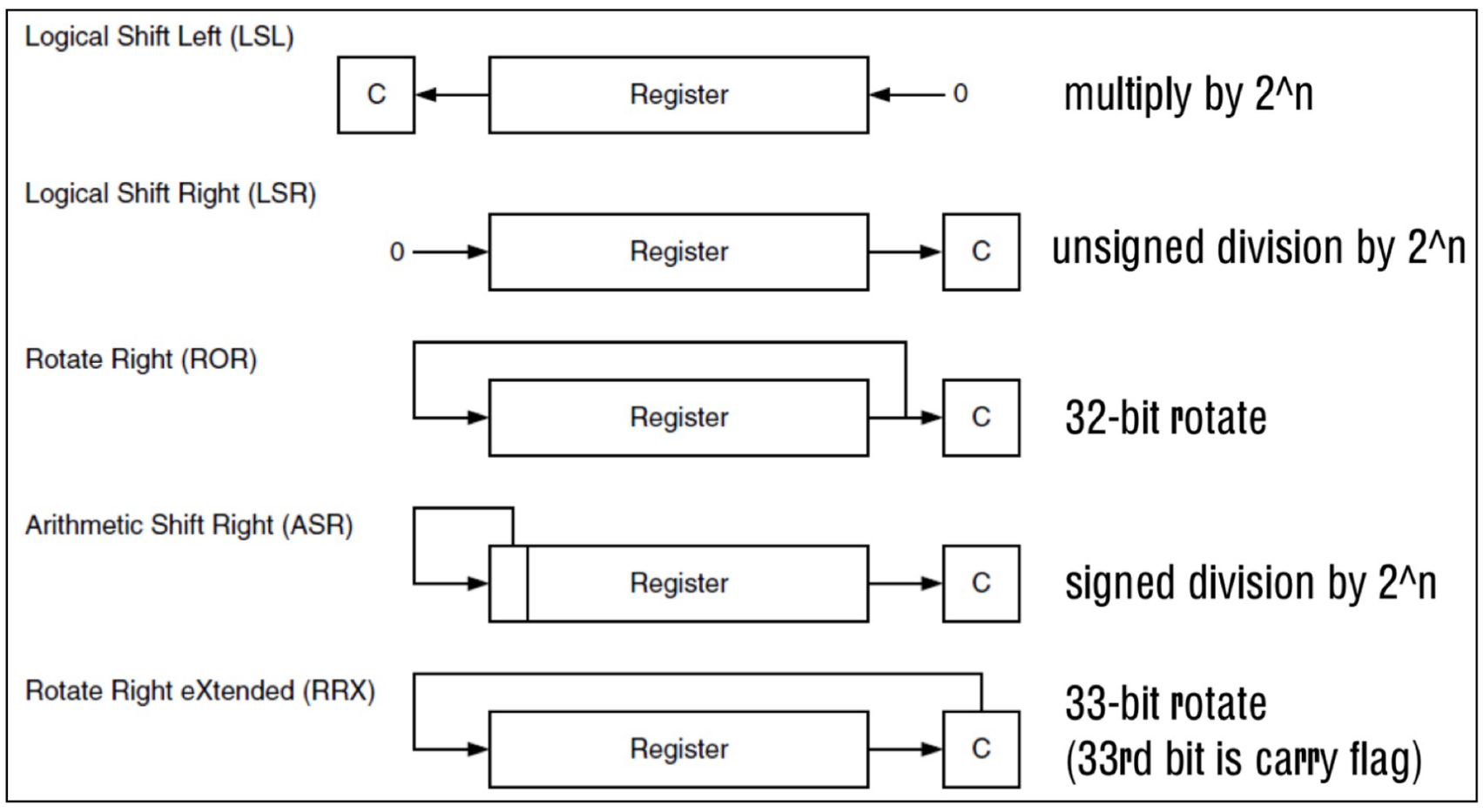
```
        LDR R1, =NUMBERS
        LDR R2, N
        MOV R0, #0

Loop:   LDR R3, [R1], #4
        CMP R3, #0
        IT MI
        ADDMI R0, #1
        SUBS R2, #1
        BNE Loop

        LDR R4, =NEGNUM
        STR R0, [R4]
```

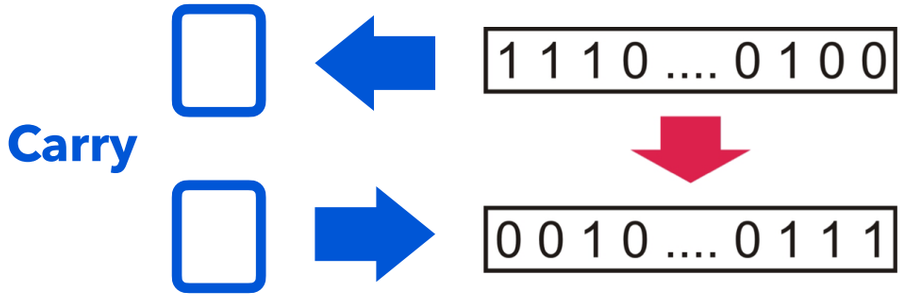
Q5: Write an ARM program to reverse the order of the bits in register R2. For example, if the starting pattern in R2 is 1110...0100, the final result in R2 should be 0010...0111. (*Hint*: use shift and rotate operations.)

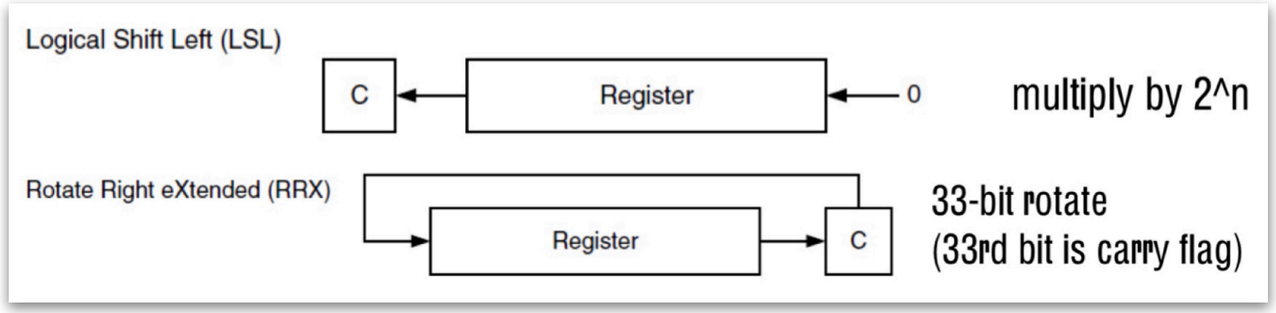




CG2028 Tutorial 2 (Week 9): ARM v7-M Assembly Language

Q5:





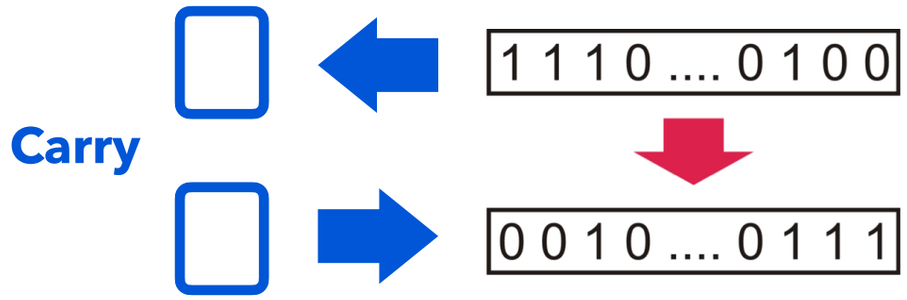
@ R0: count down for 32 times

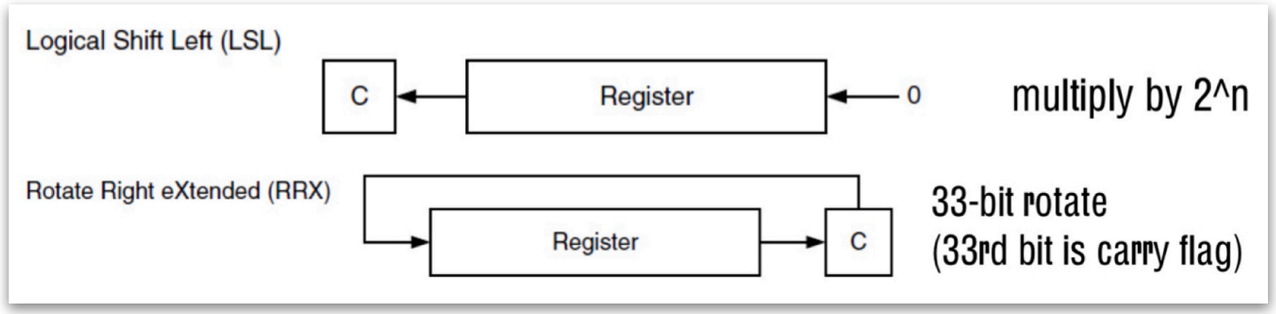
@ R1: resulted 32-bit data

@ R2: original 32-bit data

CG2028 Tutorial 2 (Week 9): ARM v7-M Assembly Language

Q5:





@ R0: count down for 32 times

@ R1: resulted 32-bit data

@ R2: original 32-bit data

(1) Set the counter R0 to 32

(2) LOOP:

(2.1) Shift R2 to left once, MSB of R2 will be put into the Carry

(2.2) RRX R1, to send the Carry bit into R1

(2.3) Reduce the counter R0 by 1

(2.4) Jump back to LOOP if the counter R0 is not zero

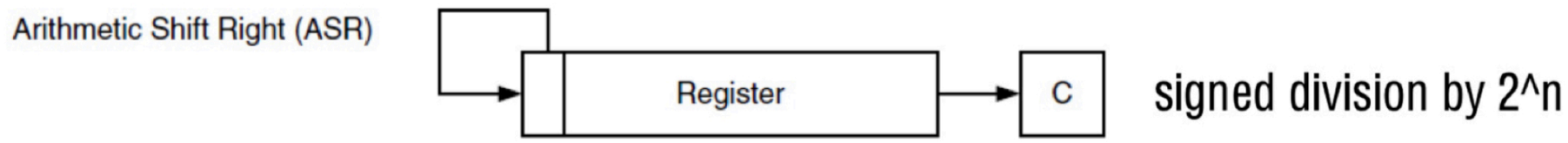
(3) Copy R1 (the result) to R2

```
        MOV R0, #32

Loop:   LSLS R2, #1
        RRX R1, R1
        SUBS R0, #1
        BNE Loop

        MOV R2, R1
```

Usage of ASR:

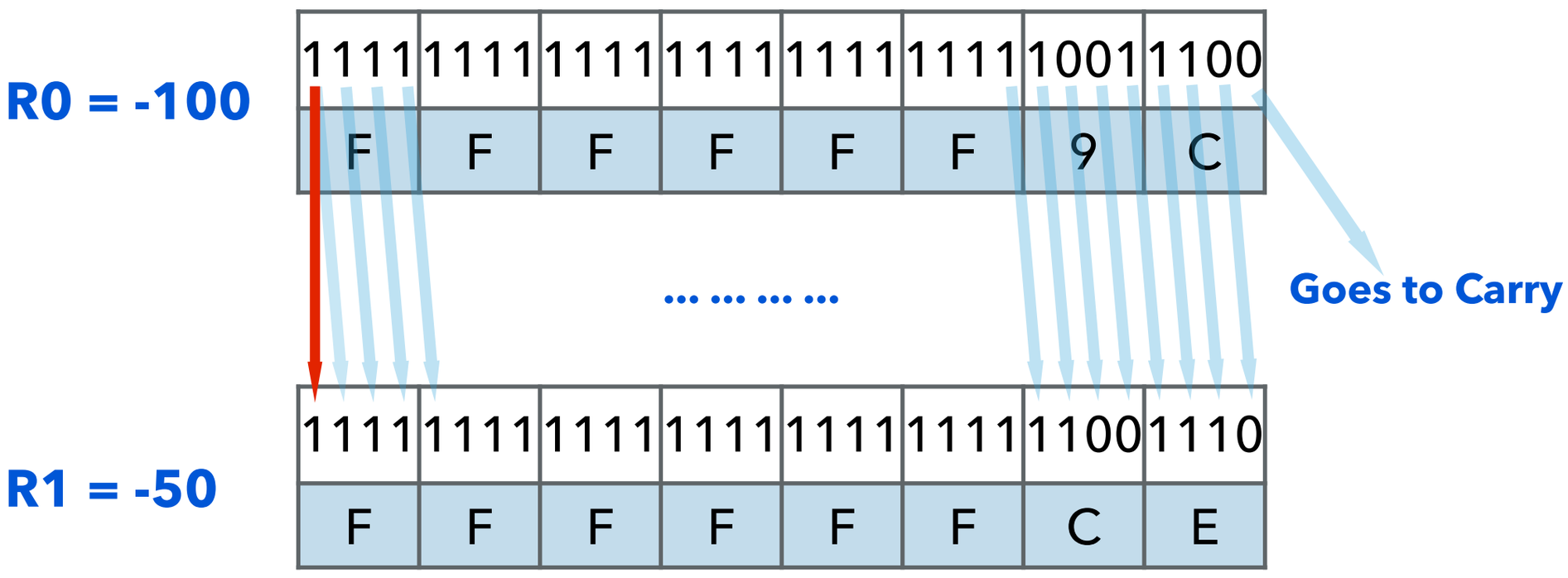


When we are dividing/multiplying a **signed** number using shift operation, we need to remain the MSB so that **sign** of the result will not be altered.

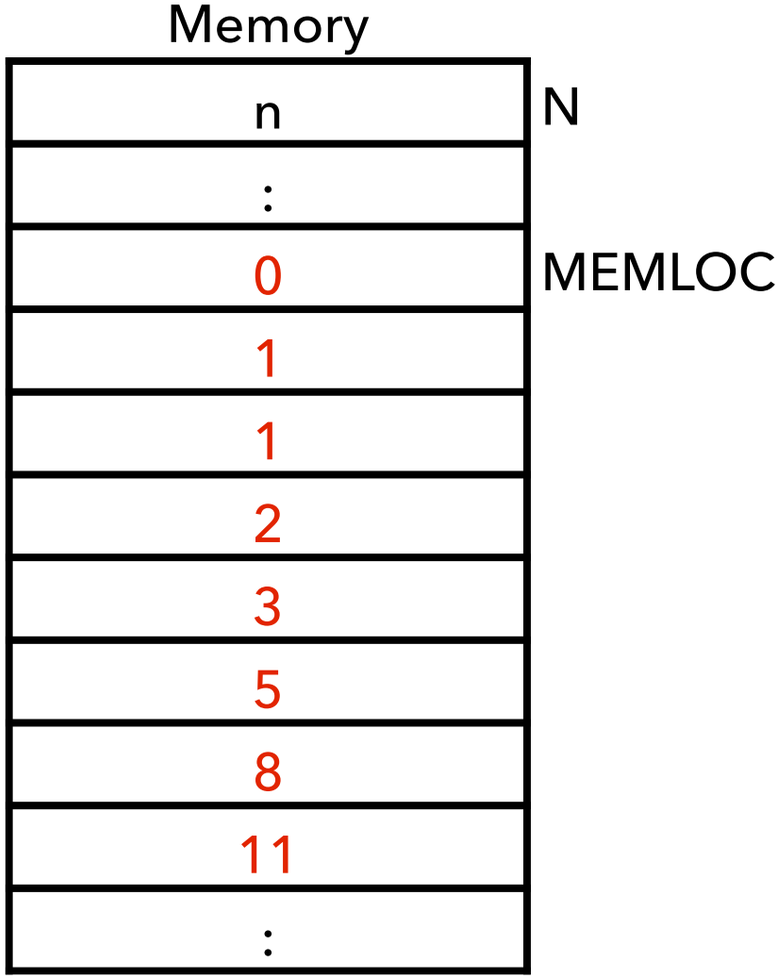
For example:

MOV R0, #-100 @ Let R0 = -100 in decimal

ASR R1, R0, #1 @ R1 = R0/(2^1) (because of 1-bit shifting to right)



Q6: Write an ARM program that generates the first n numbers of the Fibonacci series. In this series, the first two numbers are 0 and 1, and each subsequent number is generated by adding the preceding two numbers. For example, for n=8, the series is 0, 1, 1, 2, 3, 5, 8, 13. Your program should store the numbers in successive memory word locations starting at MEMLOC. Assume that the value n is stored in location N.



Q6:

Memory

n	N
:	
0	MEMLOC
1	
1	
2	
3	
5	
8	
11	
:	

Given		Calculate				
0	1	1	2	3	5	...

@ R0: count, from n to 0

@ R1: address of target memory

@ R2: 1st element

@ R3: 2nd element

Q6:

0	1	1	2	...
R2	~~R3~~ R2	R3		

(1) Get the count N (put into R0)

(2) Get the starting location (put into R1)

(3) Get 1st and 2nd number (R2: 1st, R3: 2nd), and store them into memory

(4) Reduce count by 2

(5) Loop:

(5.1) calculate the next number (put into R3), and store it into memory (pointer then point to the next memory slot;

(5.2) refresh R2

(5.3) reduce count by 1

(5.4) jump back to Loop if count is not zero

```
        @ R0: count, from n to 0
        @ R1: address of target memory
        @ R2: 1st element
        @ R3: 2nd element
        LDR R0, N           @ R0 for loop count for numbers
        SUB R0, #2          generated after 1

        LDR R1, =MEMLOC     @ R1 for memory address
        MOV R2, #0          @ Store the first 2 numbers, 0 and 1, from
        MOV R3, #1          R2 and R3, into memory
        STR R2, [R1], #4
        STR R3, [R1], #4

Loop:   ADD R3, R2, R3      @ Starting with number i-1 in R2 and i in
                            R3, compute and place i+1 in R3
        STR R3, [R1], #4    @ and store in memory
        SUB R2, R3, R2      @ Recover old i and place in R2
        SUBS R0, #1         @ Branch back if all numbers have not
        BGT Loop            been computed
```

THE END

For Consultation: eleguji@nus.edu.sg
Office: E4-03-10