SHIFT INSTRUCTIONS: PRACTICE PROBLEM

```
LDR R4, =0x12345678
LSL R6, R4, #6
```

Find R6 ?

R4 = 0001 0010 0011 0100 0101 0110 0111 1000

0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	R4
0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	0	1st Shift
0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	2nd Shift
1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	3rd Shift
0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	4th Shift
0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	5th Shift
1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	6th Shift

R6 = 0x8D159E00

SHIFT INSTRUCTIONS: PRACTICE PROBLEM

```
LDR R4, =0x12345678
LSR R6, R4, #6
```

Find R6 ?

R4 = 0001 0010 0011 0100 0101 0110 0111 1000

0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	R4
0	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	1st Shift
0	**0**	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	2nd Shift
0	**0**	**0**	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	3rd Shift
0	**0**	**0**	**0**	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	4th Shift
0	**0**	**0**	**0**	**0**	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	5th Shift
0	**0**	**0**	**0**	**0**	**0**	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	6th Shift

R6 = 0x0048D159

SHIFT INSTRUCTIONS: PRACTICE PROBLEM

```
LDR R4, =0x12345678
ASR R6, R4, #6
```

Find R6 ?

R4 = 0001 0010 0011 0100 0101 0110 0111 1000

0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	0	R4
0	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	0	1st Shift
0	**0**	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	0	2nd Shift
0	**0**	**0**	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	1	3rd Shift
0	**0**	**0**	**0**	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	1	4th Shift
0	**0**	**0**	**0**	**0**	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	1	5th Shift
0	**0**	**0**	**0**	**0**	**0**	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	0	1	6th Shift

R6 = 0x0048D159

SHIFT INSTRUCTIONS: PRACTICE PROBLEM

```
LDR R4, =0xF123FABC
LSL R5, R4, #3
LSR R6, R4, #6
ASR R7, R4, #8
```

Find R5, R6, R7?

R4= 0xF123FABC

R4= 1111 0001 0010 0111 1111 1010 1011 1100

R5= ~~111~~ 1000 1001 0011 1111 1101 0101 1110 0000

R6= 0000 0011 1100 0100 1001 1111 1110 1010 ~~11 1100~~

R7 = 1111 1111 1111 0001 0010 0111 1111 1010 ~~1011 1100~~

SHIFT INSTRUCTIONS: PRACTICE PROBLEM

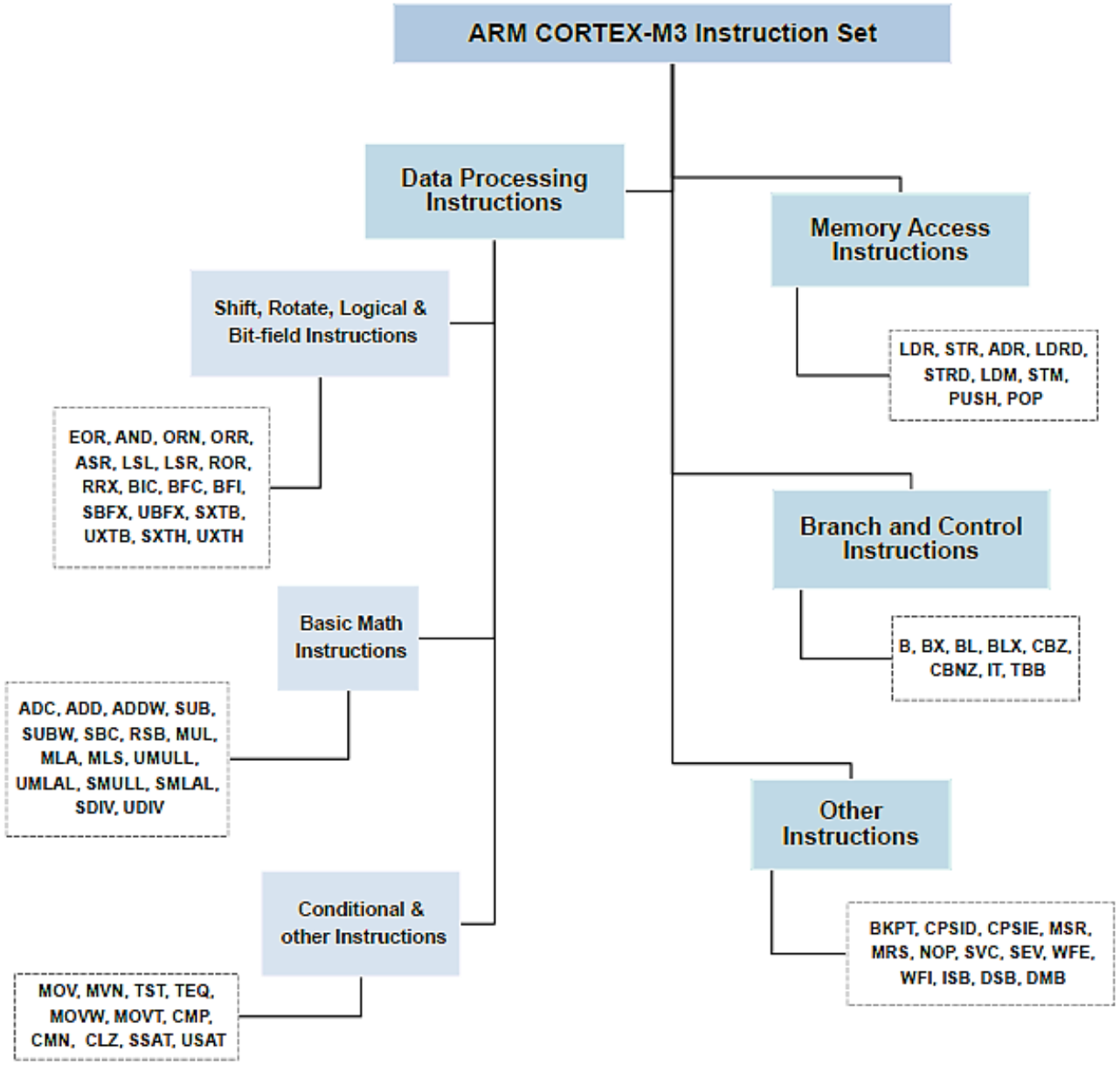
Write an ALP to multiply R2 content with 35 using shift instructions and store result in R4 reg?

It can obtained as follows

$$R4 = R2 \times 35 = R2 \times (32 + 2 + 1) = R2 \times (2^5 + 2^1 + 2^0) = R2 \times 2^5 + R2 \times 2^1 + R2 \times 2^0$$

```
LSL R1, R2, #5
LSL R3, R2, #1
ADD R5, R1, R3
ADD R4, R5, R2
```

ARM CORTEX M3 INSTRUCTION SET



Data processing instructions

- ✓ The data processing instructions are further divided into multiple subgroups based on either their functionality or if they follow similar generic instruction syntax.
  - Data movement instructions- performing an operation (from a set of operations) on the data before moving it to the register
  - Basic math instructions- addition, subtraction, multiplication, and division
  - **Bitwise operations-** Shift, rotate, and logical instructions
  - Bitfield instructions- operate on a group of adjacent bits
  - **Test and compare instructions-** are used to affect the application program status register to control the flow of program execution
  - Saturating instructions- that allow us to reduce distortion caused by the saturation phenomenon.

LOGICAL INSTRUCTIONS

Mnemonic	Brief description	Encoding	Flags
AND	Logical AND operation	16 or 32 bit	N,Z,C
ORR	Logical OR	16 or 32 bit	N,Z,C
EOR	Exclusive OR	16 or 32 bit	N,Z,C
ORN	Logical OR NOT	32 bit	N,Z,C
BIC	Bit clear	16 or 32 bit	N,Z,C

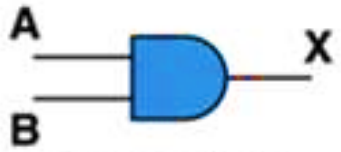
LOGICAL INSTRUCTIONS

AND Rd, Rm, #n

ANDS Rd, Rm, #n

E.g. :

```
AND  R2, R1, #0xFF00  ; R2 = R1 & 0 xFF00
ANDS R2, R1, #0xFF00  ; R2 = R1 & 0 xFF00 (Flags affected)
```



A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

AND Rd, Rm, Rs

ANDS Rd, Rm, Rs

E.g. :

```
AND  R2, R1, R0   ; R2 = R1 & R0
ANDS R2, R1, R0   ; R2 = R1 & R0
                  ; Flags affected
```

	D7	D6	D5	D4	D3	D2	D1	D0
AND	1	1	1	1	1	0	1	1
	D7	D6	D5	D4	D3	0	D1	D0

Clearing of a bit

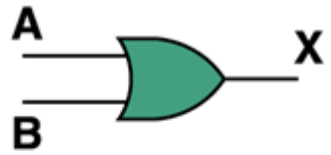
LOGICAL INSTRUCTIONS

ORR Rd, Rm, #n

ORRS Rd, Rm, #n

E.g. :

```
ORR  R2, R1, #0xFF00  ; R2 = R1 | 0xFF00
ORRS R2, R1, #0xFF00  ; R2 = R1 | 0xFF00 (Flags affected)
```



A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

ORR Rd, Rm, Rs

ORRS Rd, Rm, Rs

E.g. :

```
ORR  R2, R1, R0   ; R2 = R1 | R0
ORRS R2, R1, R0   ; R2 = R1 | R0
                  ; Flags affected
```

	D7	D6	D5	D4	D3	D2	D1	D0
ORR	0	0	0	0	0	1	0	0
	D7	D6	D5	D4	D3	1	D1	D0

Setting of a bit

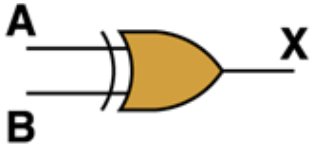
LOGICAL INSTRUCTIONS

EOR Rd, Rm, #n

EORS Rd, Rm, #n

E.g. :

```
EOR  R2, R1, #0xFF00  ; R2 = R1 ^ 0xFF00
EORS R2, R1, #0xFF00  ; R2 = R1 ^ 0xFF00 (Flags affected)
```



A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

EOR Rd, Rm, Rs

EORS Rd, Rm, Rs

E.g. :

```
EOR  R2, R1, R0   ; R2 = R1 ^ R0
EORS R2, R1, R0   ; R2 = R1 ^ R0
                  ; Flags affected
```

	D7	D6	D5	D4	D3	D2	D1	D0
EOR	1	0	0	0	1	1	0	0
	D7'	D6	D5	D4	D3'	D2'	D1	D0

LOGICAL INSTRUCTIONS

ORN Rd, Rm, #n

ORNS Rd, Rm, #n

E.g. :

```
ORN  R2, R1, #0xFF00  ; R2 = R1 | ~(0xFF00)
ORNS R2, R1, #0xFF00  ; R2 = R1 | ~(0xFF00)  (Flags affected)
```

ORN Rd, Rm, Rs

ORNS Rd, Rm, Rs

E.g. :

```
ORN  R2, R1, R0   ; R2 = R1 | ~R0
ORNS R2, R1, R0   ; R2 = R1 | ~R0, Flags affected
```

LOGICAL INSTRUCTIONS

BIC Rd, Rm, #n

BICS Rd, Rm, #n

E.g. :

```
BIC  R2, R1, #0xFF00  ; R2 = R1 & ~(0xFF00)
BICS R2, R1, #0xFF00  ; R2 = R1 & ~(0xFF00) (Flags affected)
```

BIC Rd, Rm, Rs

BICS Rd, Rm, Rs

E.g. :

```
BIC  R2, R1, R0   ; R2 = R1 & ~R0
BICS R2, R1, R0   ; R2 = R1 & ~R0, Flags affected
```

```
BIC  R2, R1  ; R2 = R2 & ~R1
```

TEST AND COMPARE INSTRUCTIONS

Mnemonic	Brief description	Encoding	Flags
CMN	Compare negative	16 or 32 bit	N,Z,C,V
CMP	Compare	16 or 32 bit	N,Z,C,V
TEQ	Test equivalence	32 bit	N,Z
TST	Test	16 or 32 bit	N,Z

TEST AND COMPARE INSTRUCTIONS

CMP Rn, #n

CMP Rn, Rm

E.g. :

```
CMP R2, #0xFF00  ; R2 - 0xFF00  (N, Z, C & V Flags affected)
CMP R2, R1       ; R2 - R1  (N, Z, C & V Flags affected)
```

- ✓ The CMP instruction subtracts the value of n or Rm from the value in Rn. This is the same as a SUBS instruction, except that the result is discarded.

	Negative	Zero
Rn = Rm	0	1
Rn > Rm	0	0
Rn < Rm	1	0

TEST AND COMPARE INSTRUCTIONS

CMN Rn, #n

CMN Rn, Rm

E.g. :

```
CMN R2, #0xFF00  ; R2 + 0xFF00  (N, Z, C & V Flags affected)
CMN R2, R1       ; R2 + R1  (N, Z, C & V Flags affected)
```

- ✓ The CMP instruction adds the value of n or Rm from the value in Rn. This is the same as a ADDS instruction, except that the result is discarded.

TEST AND COMPARE INSTRUCTIONS

TST Rn, #n

TST Rn, Rm

E.g. :

```
TST R2, #0xFF00  ; R2 & 0xFF00 (N, Z Flags affected)
TST R2, R1       ; R2 & R1 (N, Z Flags affected)
```

- ✓ The TST instruction performs the bitwise AND operation with the value of n or Rm. This is the same as a ANDS instruction, except that the result is discarded.

TEST AND COMPARE INSTRUCTIONS

TEQ Rn, #n

TEQ Rn, Rm

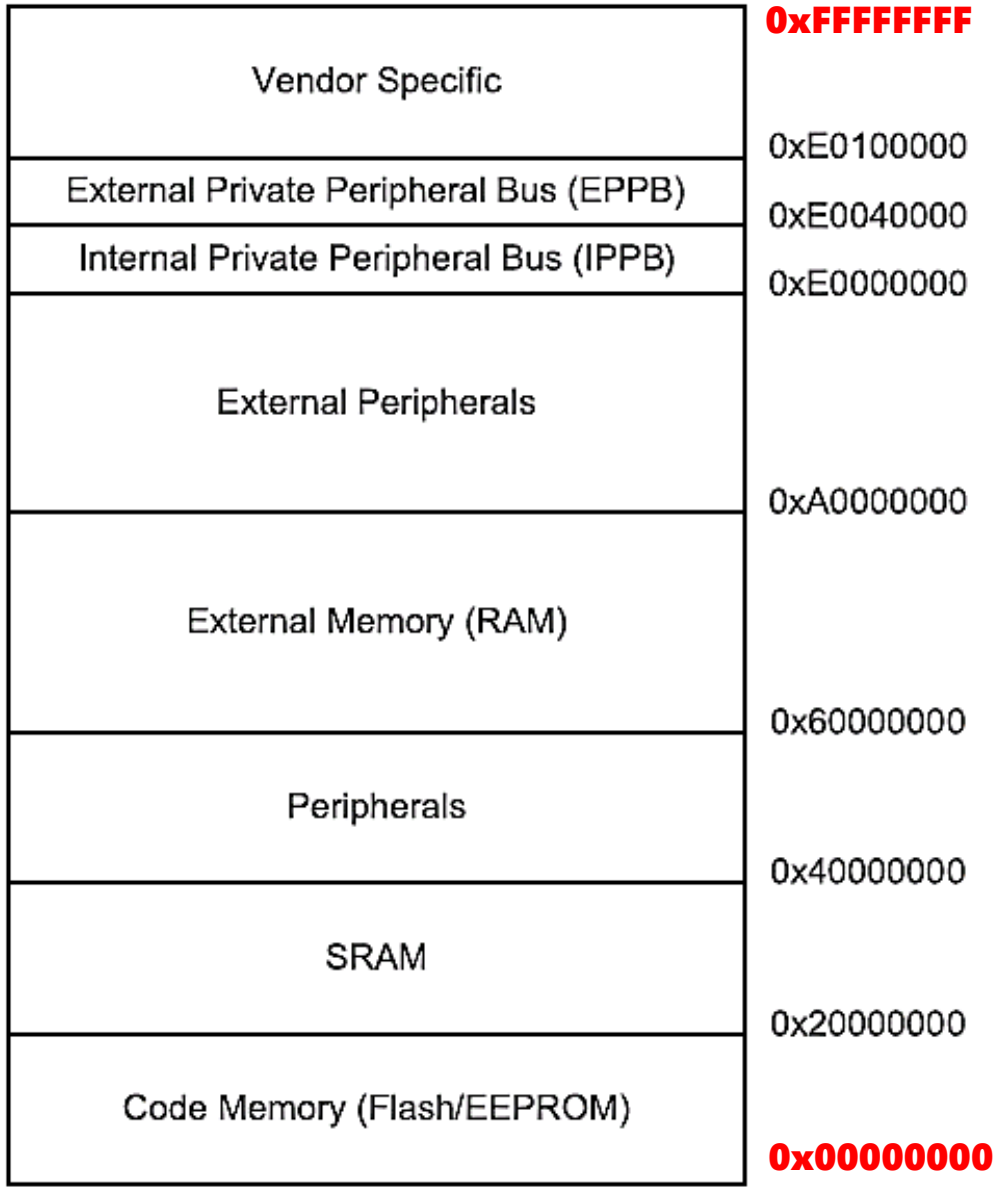
E.g. :

```
TEQ R2, #0xFF00  ; R2 ^ 0xFF00 (N, Z Flags affected)
TEQ R2, R1       ; R2 ^ R1 (N, Z Flags affected)
```

- ✓ The TEQ instruction performs the bitwise Ex-OR operation with the value of n or Rm. This is the same as a EORS instruction, except that the result is discarded.

MEMORY ACCESS INSTRUCTIONS

- ✓ Memory holds the data besides the code of a program. While carrying out operations given in a program, a processor may need to exchange data with the memory as well as peripherals.
- ✓ ARM supports Load/Store architecture means Load and Store instructions can access memory whereas other instructions based on register access.
- ✓ Data transfer instructions transfer data between registers and memory:
  - Memory to register or **LOAD** from memory to register
  - Register to memory or **STORE** from register to memory



MEMORY ACCESS INSTRUCTIONS

- ✓ The ARM is a Load/Store Architecture:
  - Does not support memory to memory data processing operations.
  - Must move data values into registers before using them.

Mnemonic	Brief description	Encoding	Flags
LDR	Load register using immediate offset	16 or 32 bit	No change
STR	Store register using immediate offset	16 or 32 bit	No change
LDM	Load multiple registers	16 or 32 bit	No change
STM	Store multiple registers	16 or 32 bit	No change
ADR	Generate address relative to PC	16 or 32 bit	No change
POP	Pop registers from stack	16 or 32 bit	No change
PUSH	Push registers onto stack	16 or 32 bit	No change

MEMORY ACCESS INSTRUCTIONS

LDR Rd, = data(32-bit) ; Load 32-bit data in Register R2

E.g. :

```
LDR R2, =0xFF00FF11     ; R2 = 0xFF00FF11
```

MOV Rd, #data(16-bit)

E.g.:

```
MOV R2, #0xFF
```

EXAMPLES

Example: Write an ALP to add and subtract of FF008800H data with 99FFABCDH data?

```
LDR R0, =0xFF008800
LDR R1, =0x99FFABCD
ADD R10, R0, R1
SUB R11, R0, R1
```

EXAMPLES

Example: Write an ALP statement to reset D7 bit of R2 register?

	D31	D30	D29	- - - - - - - - - -	D7	D6	D5	D4	D3	D2	D1	D0
AND	1	1	1	- - - - - - - - - -	0	1	1	1	1	1	1	1
	D31	D30	D29	- - - - - - - - - -	0	D6	D5	D4	D3	D2	D1	D0

```
LDR R1, =0xFFFFFF7F
AND R2, R2, R1
```

EXAMPLES

Example: Write an ALP statement to set D5 bit of R2 register?

	D31	D30	D29	- - - - - - - - - -	D7	D6	D5	D4	D3	D2	D1	D0
ORR	0	0	0	- - - - - - - - - -	0	0	1	0	0	0	0	0
	D31	D30	D29	- - - - - - - - - -	D7	D6	1	D4	D3	D2	D1	D0

```
LDR R1, =0x00000020
ORR R2, R2, R1
```

EXAMPLES

Example: Write an ALP to set D29, D3 bit and reset D7, D1 bit of R2 register?

	D31	D30	D29	- - - - - - - - - -	D7	D6	D5	D4	D3	D2	D1	D0
AND	0	0	1	- - - - - - - - - -	0	0	0	0	1	0	0	0
	D31	D30	1	- - - - - - - - - -	D7	D6	D5	D4	1	D2	D1	D0
ORR	1	1	1	- - - - - - - - - -	0	1	1	1	1	1	0	1
	D31	D30	1	- - - - - - - - - -	0	D6	D5	D4	1	D2	0	D0

EXAMPLES

Example: Write an ALP to set D29, D3 bit and reset D7, D1 bit of R2 register?

```
LDR R1, =0x20000008
ORR R2, R2, R1
LDR R1, =0xFFFFFF7D
AND R2, R2, R1
```

LOAD/STORE INSTRUCTIONS

- ✓ If the processor is required to access a large memory segment sequentially, then one possible implementation is to maintain a pointer to the starting address (base address) of that memory segment and an offset from this address can be used to generate a relative address.
- ✓ In Cortex- M processors this offset based addressing can be implemented in the following two possible ways
  - Immediate offset addressing
  - Register offset addressing

IMMEDIATE OFFSET ADDRESSING

- ✓ When the offset is an immediate value from the base address stored in a register, the resulting addressing mode is called immediate offset addressing.

```
LDR{type}  Rt, [Rn, #offset]
STR{type}  Rt, [Rn, #offset]
```

- ✓ The optional offset field is used to specify an offset from the base address register Rn. Its value is either added to or subtracted from the value of Rn to evaluate the memory address.
- ✓ The immediate offset can take any value between -255 and +255 for 16-bit instruction encoding, while in the case of 32-bit instruction encoding, the offset can take values between -4095 and 4095.

IMMEDIATE OFFSET ADDRESSING

{*type*}	Memory Access Type
	When this field is omitted then either signed or unsigned 32-bit word access is performed.
B	Unsigned 8-bit byte. Zero extended to 32 bits for load instructions.
SB	Signed 8-bit byte. Sign extended to 32 bits for load instructions.
H	Unsigned 16-bit halfword. Zero extended to 32 bits for load instructions.
SH	Signed 16-bit halfword. Sign extended to 32 bits for load instructions.

- ✓ For instance, when an 8-bit number, say -6 (or 0xFA) is loaded to a 32-bit register, the number needs to be sign extended. The sign extension will yield 0xFFFFFFFA in the register, which is still -6 in decimal.
- ✓ If we load an unsigned number, say 6 (or 0x06) to a 32-bit register, then zero extended 32-bit value will result in 0x00000006.

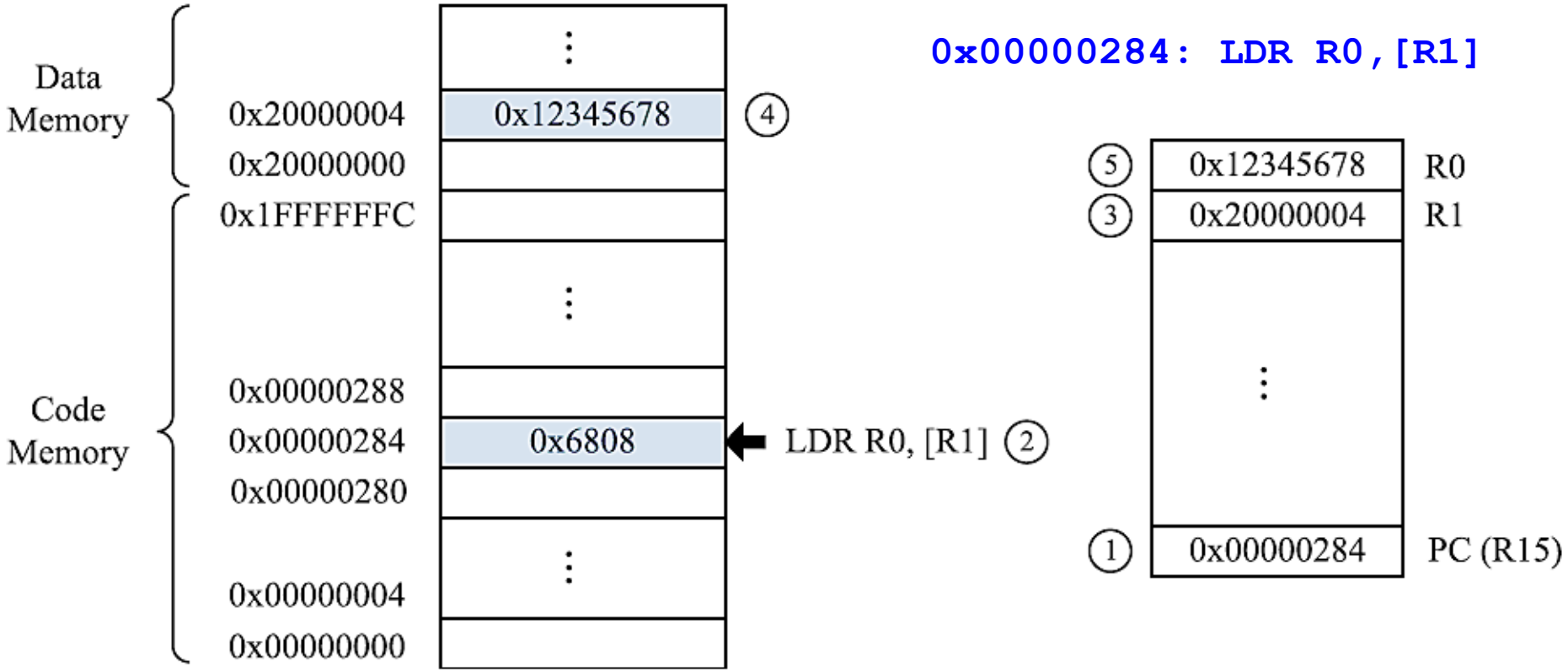
IMMEDIATE OFFSET ADDRESSING

LDR	Load word (32 Bit)	STR	Store word (32 Bit)
LDRH	Load half word (16 Bit)	STRH	Store half word (16 Bit)
LDRSH	Load signed half word (16 Bit)	STRSH	Store signed half word (16 Bit)
LDRB	Load byte (8 Bit)	STRB	Store byte (8 Bit)
LDRSB	Load signed byte (8 Bit)	STRSB	Store signed byte (8 Bit)

IMMEDIATE OFFSET ADDRESSING

✓ Example:

```
LDR R0, [R1]          ; Loads R0 from address in R1
STR R2, [R9 ,#0x7]    ; Stores R2 to a memory location with address R9 +7
```



IMMEDIATE OFFSET ADDRESSING

- ✓ Immediate offset addressing is of two types
  - Pre-indexed immediate offset addressing
  - Post-indexed immediate offset addressing
- ✓ **Pre indexed addressing:** The register containing the base address is updated first

```
LDR{type}  Rt, [Rn, #offset]!
STR{type}  Rt, [Rn, #offset]!
```

- ✓ Example:

```
LDR R8, [R10, #4]!      ; R10 = R10 + 4 is performed first and
                        ; then load operation is performed
STR R2, [R9, #0xA]!     ; First , R9 = R9 + 0xA and then store
                        ; R2 to memory with address R9
```

IMMEDIATE OFFSET ADDRESSING

✓ **Post indexed addressing:** The register containing the base address is updated first

```
LDR{type}  Rt, [Rn], #offset
STR{type}  Rt, [Rn], #offset
```

✓ Example:

```
STR R1, [R3], #1     ; Store R1 data into the address stored in R3
                     ; and then R3 is updated as R3 = R3 +1.
LDR R0, [R6], #4     ; Load the data from the address stored into R6 to R0
                     ; and then update R6 as R6 = R6 +4.
```

REGISTER OFFSET ADDRESSING

- ✓ In this addressing mode, the base address is contained in a register Rn while the offset value is also in a register Rm.

```
LDR{type} Rt, [Rn, Rm, {LSL #n}]
STR{type} Rt, [Rn, Rm, {LSL #n}]
```

- ✓ The offset can be shifted by up to 3 bits by left shift logical operation. The { LSL #n} field is optional where the parameter n can take values in the range 0 to 3.
- ✓ Example:

```
STR R7, [R4, R2, LSL #2]      ; R7 is stored to memory
                              ; location [R4 + R2 *4]
LDR R3, [R2, R0, LSL #1]      ; R3 is loaded from [R2 + R0 *2]
```

EXAMPLES

Example 1: Write a ALP for transferring the data present in memory location 0xF1ABACDF to R8.

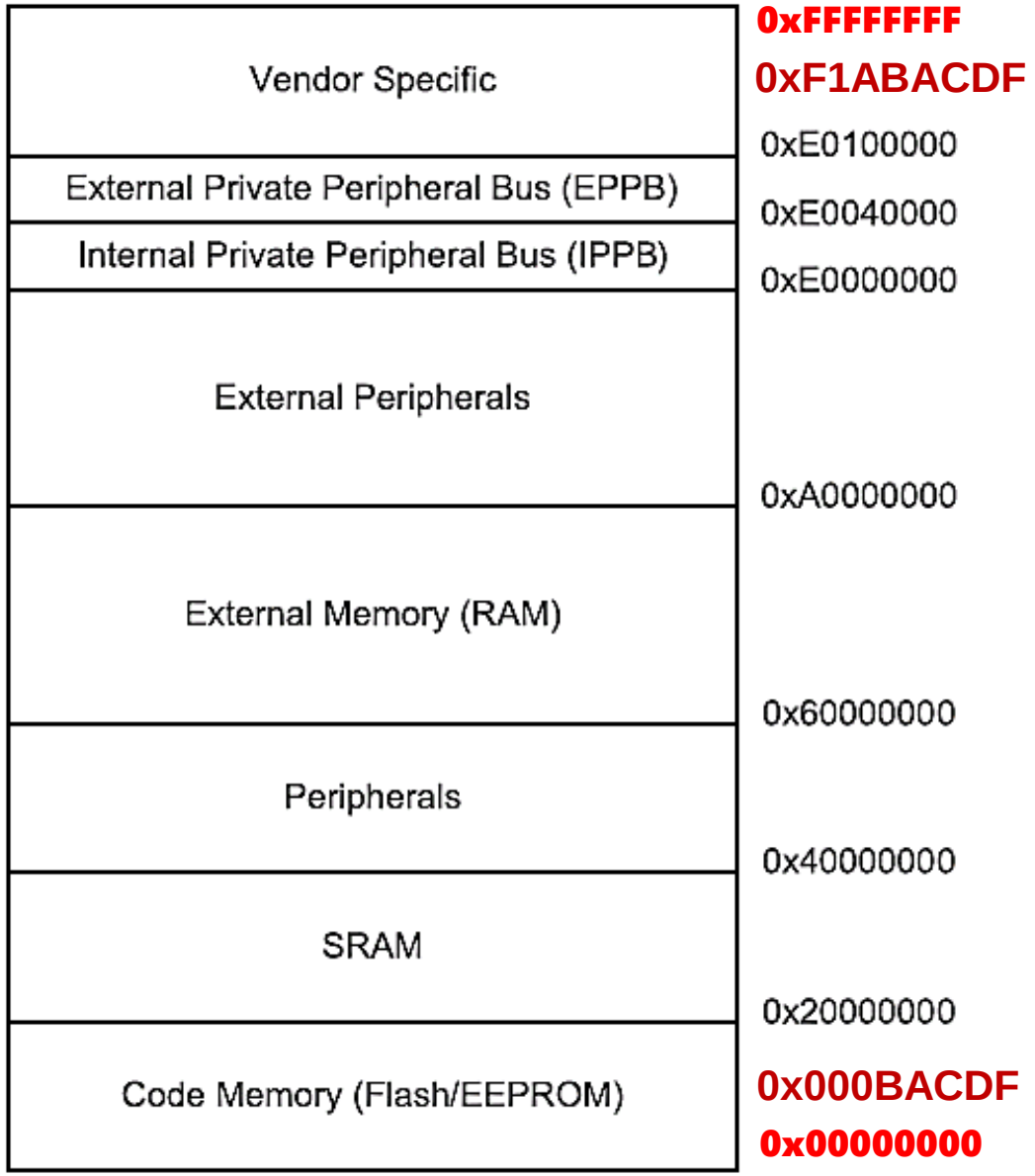
```
LDR R0, =0xF1ABACDF

LDR R8, [R0], #1
```

Example 2: Write a ALP for transferring the data present in memory location 0x000BACDF to R6.

```
LDR R1, =0x000BACDF

LDR R6, [R1], #1
```



EXAMPLES

Example 3: Write a ALP for transferring the data present in R7 to memory location 0x71ABACDF.

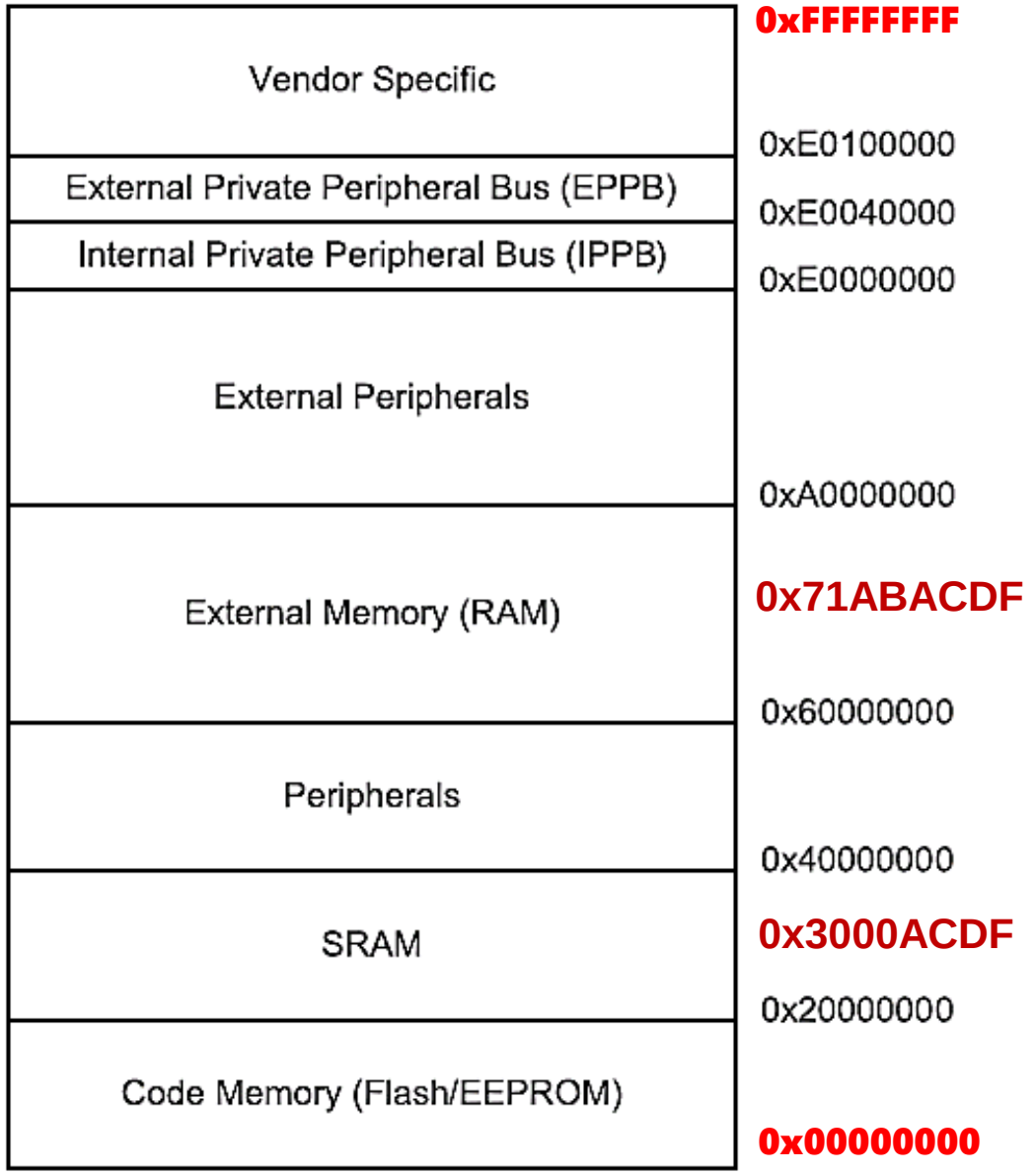
```
LDR R0, =0x71ABACDF

STR R7, [R0], #1
```

Example 4: Write a ALP for transferring the data present in R5 to memory location 0x3000ACDF.

```
LDR R1, =0x3000ACDF

STR R5, [R1], #1
```



PROGRAMS

Program 1: Write an ALP to add and subtract two data present F00F1122H and EF00FF22H memory location and store result in R10 and R11 registers respectively?

```
        LDR R0, =0xF00F1122
        LDR R1, [R0], #1
        LDR R2, =0xEF00FF22
        LDR R3, [R2], #1
        ADD R10, R1, R3
        SUB R11, R1, R3
STOP    B STOP
        END
```

PROGRAMS

Program 1: Write an ALP to add two numbers present in memory location of F100F200H and F200F100H and store result in FF00F210H memory location?

```
        LDR R0, =0xF100F200
        LDR R1, [R0], #1
        LDR R2, =0xF200F100
        LDR R3, [R2], #1
        ADD R4, R1, R3
        LDR R5, =0xFF00F210
        STR R4, [R5], #1
STOP    B STOP
        END
```

PROGRAMS

Program 2: Write an ALP to set 19th bit and 9th bit of R2 without disturbing the remaining bits and store the value in FF00FF11H memory location?

D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16
												1			

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
						1									

```
        LDR R1, =0x00080200
        ORR R2, R2, R1
        LDR R5, =0xFF00FF11
        STR R2, [R5], #1
STOP    B STOP
        END
```

PROGRAMS

Program 3: Write an ALP to reset 22th bit and 7th bit of R2 and without disturbing the remaining bits and store the value in FF00FF11H memory location?

D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16
									0						

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
								0							

```
        LDR R1, =0xFFBFFF7F
        AND R2, R2, R1
        LDR R5, =0xFF00FF11
        STR R2, [R5], #1
STOP    B STOP
        END
```

PROGRAMS

Program 4: Write an ALP to add two 64-bit data present in memory location FF00FF10H and FA00FA10H memory location and store result in FFBBFFBBH memory location?

```
LDR R0, =0xFF00FF10
LDR R1, [R0], #1
```

R1(7-0 bits) ← [R0] i.e. 8-bit data present in **FF00FF10**H memory location

R1(15-8 bits) ← [R0+1] i.e. 8-bit data present in **FF00FF11**H memory location

R1(23-16 bits) ← [R0+2] i.e. 8-bit data present in **FF00FF12**H memory location

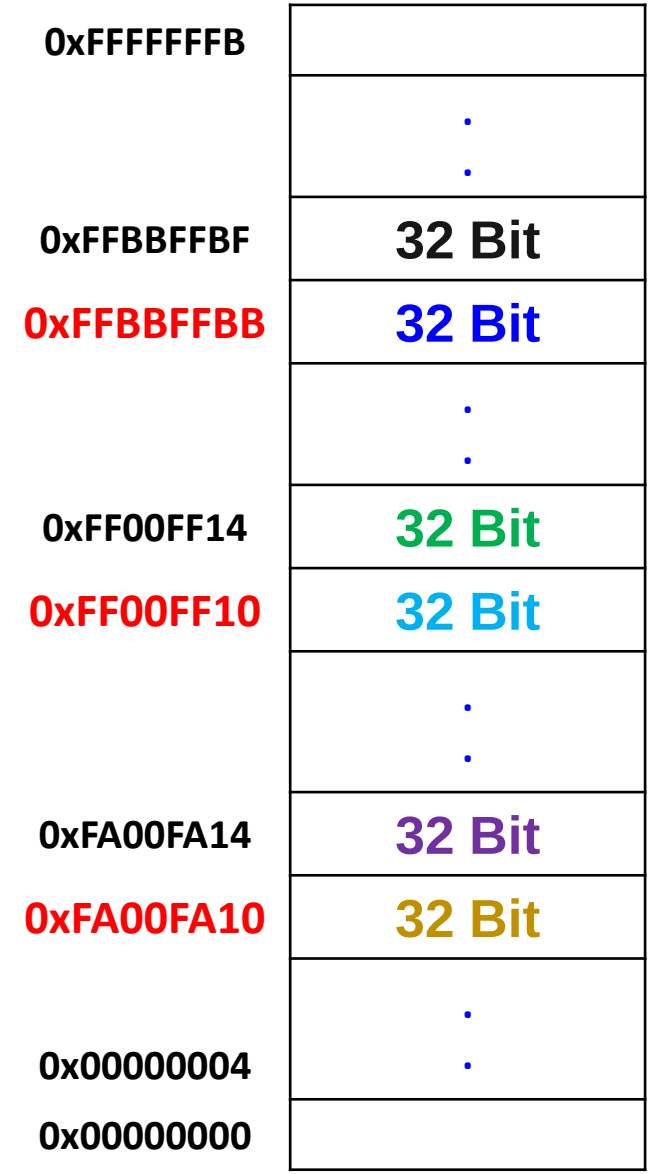
R1(31-24 bits) ← [R0+3] i.e. 8-bit data present in **FF00FF13**H memory location

PROGRAMS

Program 4: Write an ALP to add two 64-bit data present in memory location FF00FF10H and FA00FA10H memory location and store result in FFBBFFBBH memory location?

	C	
	32 Bit	32 Bit
+	32 Bit	32 Bit
	32 Bit	32 Bit

64 Bit

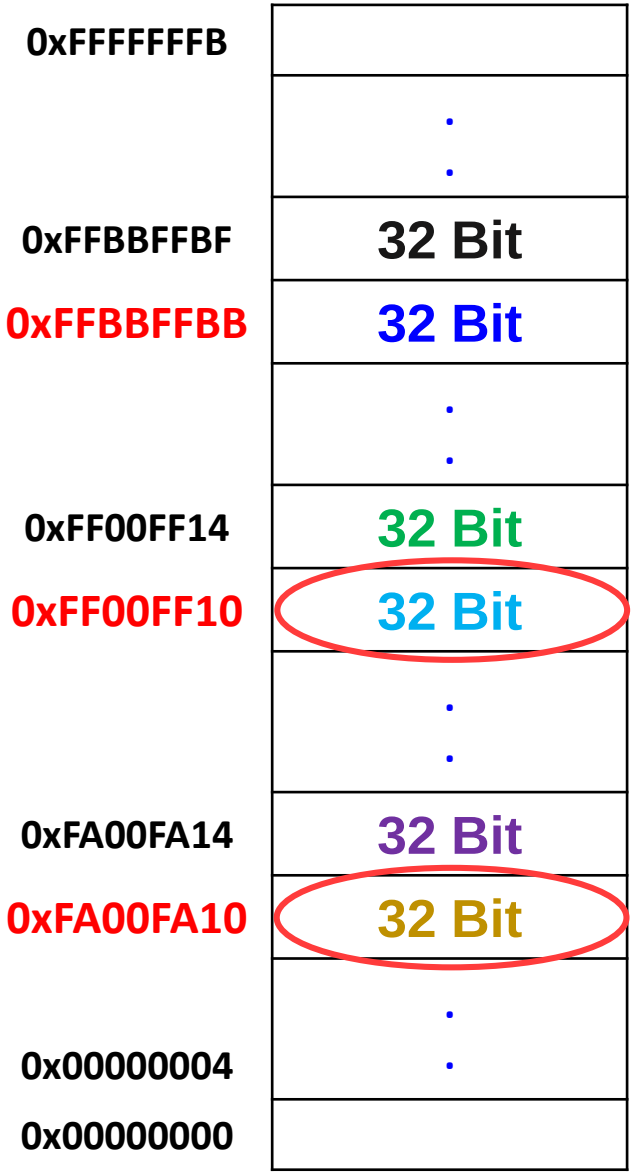


PROGRAMS

Program 4:

	C	
	32 Bit	32 Bit
+	32 Bit	32 Bit
	32 Bit	32 Bit

```
LDR   R0, =0xFF00FF10
LDR   R1, [R0], #1
LDR   R2, =0xFA00FA10
LDR   R3, [R2], #1
ADDS  R4, R1, R3
LDR   R5, =0xFFBBFFBB
STR   R4, [R5], #1
```

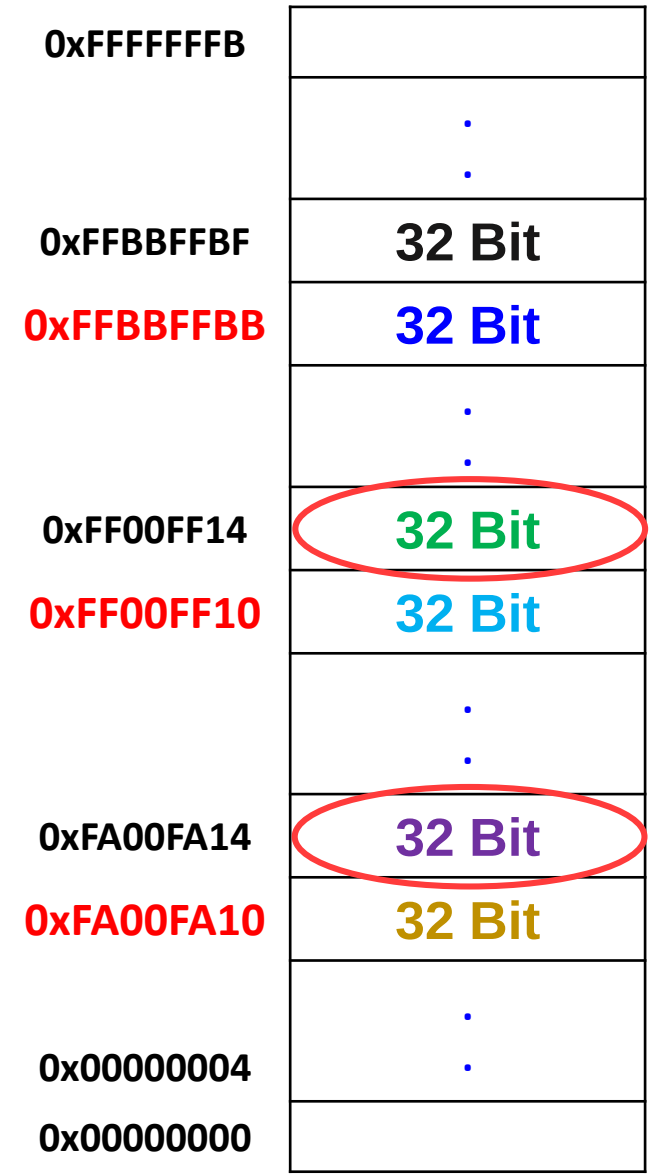


PROGRAMS

Program 4:

	C		
	32 Bit	32 Bit	
+	32 Bit	32 Bit	
	32 Bit	32 Bit	

```
        LDR R0, =0xFF00FF14
        LDR R1, [R0], #1
        LDR R2, =0xFA00FA14
        LDR R3, [R2], #1
        ADC R4, R1, R3
        LDR R5, =0xFFBBFFBF
        STR R4, [R5], #1
STOP    B STOP
        END
```



PROGRAMS

Practice Problem: Write an ALP to add two 128-bit data present in memory location FF00FF10H and FA00FA10H memory location and store result in FFBBFFBBH memory location?

		C	C	C	
	32 Bit	32 Bit	32 Bit	32 Bit
+	32 Bit	32 Bit	32 Bit	32 Bit
	32 Bit	32 Bit	32 Bit	32 Bit

128 Bit

0xFFFFFFFB	
	⋮
0xFFBBFFC7	32 Bit
0xFFBBFFC3	32 Bit
0xFFBBFFBF	32 Bit
0xFFBBFFBB	32 Bit
	⋮
0xFF00FF1C	32 Bit
0xFF00FF18	32 Bit
0xFF00FF14	32 Bit
0xFF00FF10	32 Bit
	⋮
0xFA00FA1C	32 Bit
0xFA00FA18	32 Bit
0xFA00FA14	32 Bit
0xFA00FA10	32 Bit
	⋮
0x00000000	

PROGRAMS

Practice Problem:

	C	C	C	
	32 Bit	32 Bit	32 Bit	32 Bit
+	32 Bit	32 Bit	32 Bit	32 Bit
	32 Bit	32 Bit	32 Bit	32 Bit

```
LDR   R0, =0xFF00FF10
LDR   R1, [R0], #1
LDR   R2, =0xFA00FA10
LDR   R3, [R2], #1
ADDS  R4, R1, R3
LDR   R5, =0xFFBBFFBB
STR   R4, [R5], #1
```

0xFFFFFFFB	
	⋮
0xFFBBFFC7	32 Bit
0xFFBBFFC3	32 Bit
0xFFBBFFBF	32 Bit
0xFFBBFFBB	32 Bit
	⋮
0xFF00FF1C	32 Bit
0xFF00FF18	32 Bit
0xFF00FF14	32 Bit
0xFF00FF10	32 Bit
	⋮
0xFA00FA1C	32 Bit
0xFA00FA18	32 Bit
0xFA00FA14	32 Bit
0xFA00FA10	32 Bit
	⋮
0x00000000	

PROGRAMS

Practice Problem:

		C	C	C	
	32 Bit	32 Bit	32 Bit	32 Bit	
+	32 Bit	32 Bit	32 Bit	32 Bit	
	32 Bit	32 Bit	32 Bit	32 Bit	

```
LDR   R0, =0xFF00FF14
LDR   R1, [R0], #1
LDR   R2, =0xFA00FA14
LDR   R3, [R2], #1
ADC   R4, R1, R3
LDR   R5, =0xFFBBFFBF
STR   R4, [R5], #1
```

0xFFFFFFFB	
	⋮
0xFFBBFFC7	32 Bit
0xFFBBFFC3	32 Bit
0xFFBBFFBF	32 Bit
0xFFBBFFBB	32 Bit
	⋮
0xFF00FF1C	32 Bit
0xFF00FF18	32 Bit
0xFF00FF14	32 Bit
0xFF00FF10	32 Bit
	⋮
0xFA00FA1C	32 Bit
0xFA00FA18	32 Bit
0xFA00FA14	32 Bit
0xFA00FA10	32 Bit
	⋮
0x00000000	

PROGRAMS

Practice Problem:

	C	C	C		
	32 Bit	32 Bit	32 Bit	32 Bit	
+	32 Bit	32 Bit	32 Bit	32 Bit	
	32 Bit	32 Bit	32 Bit	32 Bit	

```
LDR   R0, =0xFF00FF18
LDR   R1, [R0], #1
LDR   R2, =0xFA00FA18
LDR   R3, [R2], #1
ADC   R4, R1, R3
LDR   R5, =0xFFBBFFC3
STR   R4, [R5], #1
```

0xFFFFFFFB	
	⋮
0xFFBBFFC7	32 Bit
0xFFBBFFC3	32 Bit
0xFFBBFFBF	32 Bit
0xFFBBFFBB	32 Bit
	⋮
0xFF00FF1C	32 Bit
0xFF00FF18	32 Bit
0xFF00FF14	32 Bit
0xFF00FF10	32 Bit
	⋮
0xFA00FA1C	32 Bit
0xFA00FA18	32 Bit
0xFA00FA14	32 Bit
0xFA00FA10	32 Bit
	⋮
0x00000000	

PROGRAMS

Practice Problem:

		C	C	C	
	32 Bit	32 Bit	32 Bit	32 Bit	
+	32 Bit	32 Bit	32 Bit	32 Bit	
	32 Bit	32 Bit	32 Bit	32 Bit	

```
        LDR R0, =0xFF00FF1C
        LDR R1, [R0], #1
        LDR R2, =0xFA00FA1C
        LDR R3, [R2], #1
        ADC R4, R1, R3
        LDR R5, =0xFFBBFFC7
        STR R4, [R5], #1
STOP    B STOP
        END
```

0xFFFFFFFB	
	⋮
0xFFBBFFC7	32 Bit
0xFFBBFFC3	32 Bit
0xFFBBFFBF	32 Bit
0xFFBBFFBB	32 Bit
	⋮
0xFF00FF1C	32 Bit
0xFF00FF18	32 Bit
0xFF00FF14	32 Bit
0xFF00FF10	32 Bit
	⋮
0xFA00FA1C	32 Bit
0xFA00FA18	32 Bit
0xFA00FA14	32 Bit
0xFA00FA10	32 Bit
	⋮
0x00000000	

MCQs

✓ Find the value of R0 after executing the following instructions.

```
MOV R0, #0xFF
```

Before execution R0=XX

```
MOV R0, #0xFF
```

After execution R0=0xFF

MCQs

✓ Find the value of R0 after executing the following instructions.

```
LDR R0, #0x1FF45BFF
```

Before execution R0=XXXXXXXX

```
LDR R0, #0x1FF45BFF
```

After execution R0 = 0x1FF45BFF

MCQs

✓ Find the value of R0, R1, R2 after executing the following instructions.

```
MOV R0, #0xFF
MOV R1, R0
MOV R2, R1
```

Before execution R0 = R1 = R2 = XX

```
MOV R0, #0xFF
MOV R1, R0
MOV R2, R1
```

After execution R0 = R1 = R2 = 0xFF

MCQs

✓ Find the value of R0, R1 after executing the following instructions. Assume [F000F111] = 11H, [F000F112] = 22H, [F000F113] = AAH and [F000F114] = FFH

```
LDR R0, =0xF000F111
LDR R1,[R0], #1
```

Before execution R0 = XX

```
LDR R0, =0xF000F111
```

After execution R0 = 0xF000F111

Before execution R1 = XX

R1[7:0] ← [F000F111] i.e. R1[7:0] =11H

R1[15:8] ← [F000F112] i.e. R1[15:8] =22H

R1[23:16] ← [F000F113] i.e. R1[23:16] =AAH

R1[31:24] ← [F000F114] i.e. R1[31:24] =FFH

After execution R1 = 0xFFAA2211

MCQs

✓ Find the value of R0, R1 after executing the following instructions. Assume [F000F111] = 11H, [F000F112] = 22H, [F000F113] = AAH and [F000F114] = FFH

```
LDR R0, =0xF000F111
STR R1,[R0], #1
```

Before execution R0 = XX

```
LDR R0, =0xF000F111
```

After execution R0 = 0xF000F111

Before execution R1 = XX

R1[7:0] → [F000F111]

R1[15:8] → [F000F112]

R1[23:16] → [F000F113]

R1[31:24] → [F000F114]

After execution R1 = XX

MCQs

✓ Find the value of R2 after executing the following instructions. Assume [F000F111]=11H, [F000F112]=22H, [F000F113]=AAH and [F000F114]=FFH.

```
LDR R0, =0xF000F111

LDR R1, [R0], #1

ADD R2, R1, R0
```

BITFIED INSTRUCTIONS

Mnemonic	Brief description	Encoding	Flags
BFC	Bit Field Clear	32 bit	
BFI	Bit Field Insert	32 bit	
CLZ	Count leading zeros	32 bit	
RBIT	Reverse bits	32 bit	
REV	Reverse byte order in a word	16 or 32 bit	
REV16	Reverse byte order in each halfword	16 or 32 bit	
REVSH	Reverse byte order in bottom halfword & sign extend	16 or 32 bit	
SBFX	Signed Bit Field Extract	32 bit	
UBFX	Unsigned Bit Field Extract	32 bit	
SXT	Sign extend	16 or 32 bit	
UXT	Zero extend	16 or 32 bit	

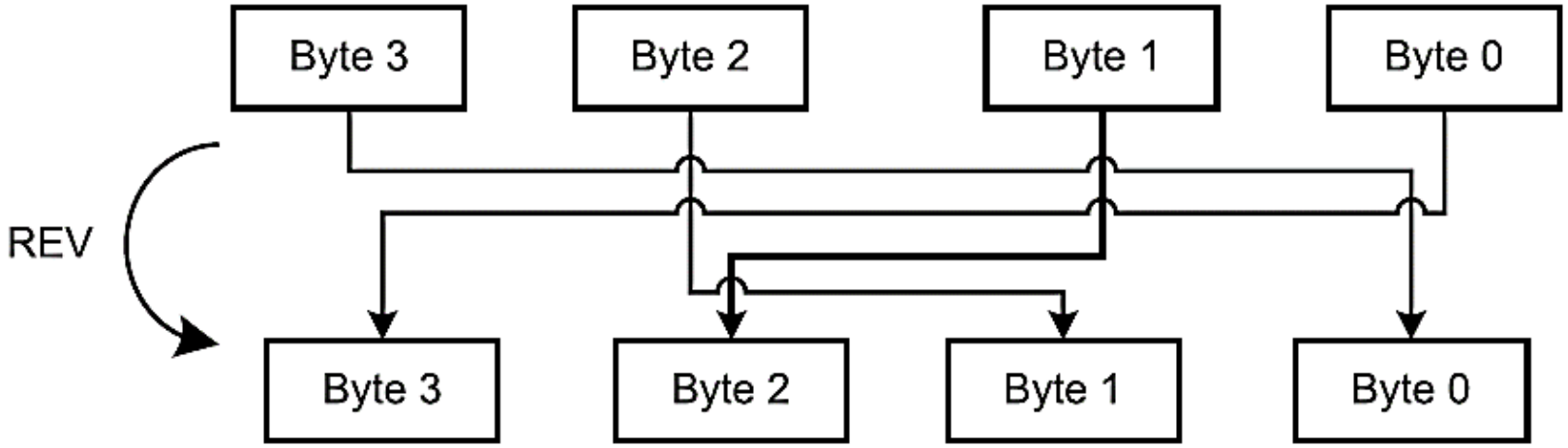
BITFIED INSTRUCTIONS

- ✓ The bitfield instructions operate on the adjacent group of bits in the operand registers.
- ✓ It can be categorized further into the following subgroups
  - Bit and byte reversal instructions
  - Bitfield clear and insert instructions
  - Bitfield extract instructions
  - Miscellaneous bitfield instructions

BITFIED INSTRUCTIONS

1. Bit and byte reversal instructions

REV Rd, Rn



E.g. :

```
MOVW R1, 0x5678
MOVT R1, 0x1234
```

; create 32- bit word data, **R1 = 0x12345678**

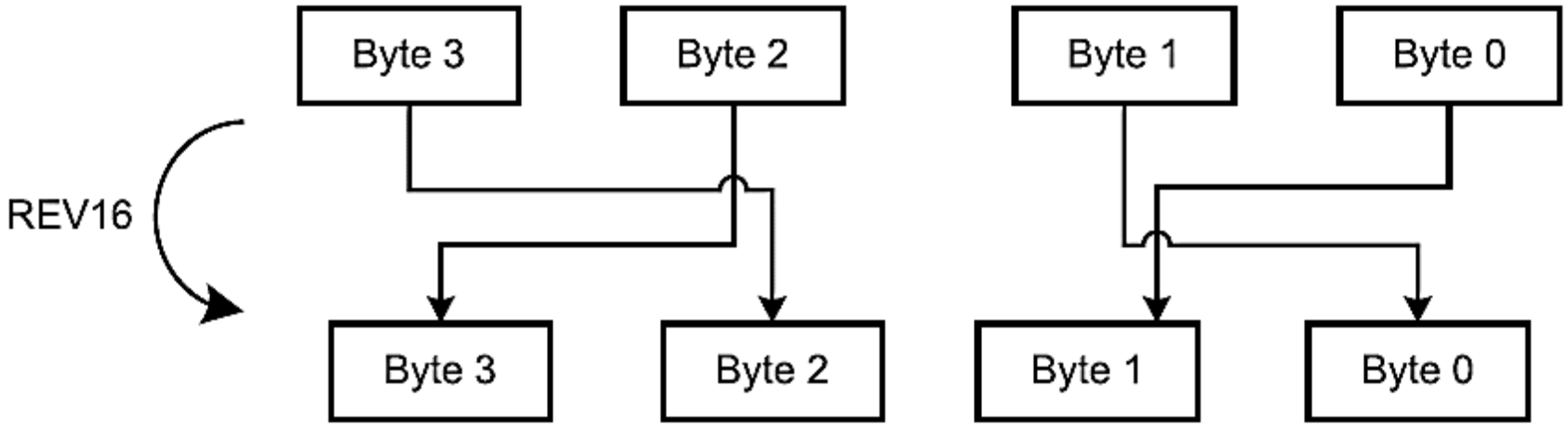
```
REV  R4, R1
```

; byte reversal of data, **R4 = 0x78563412**

BITFIED INSTRUCTIONS

1. Bit and byte reversal instructions

REV16 Rd, Rn



E.g. :

```
MOVW    R1, 0x5678
MOVT    R1, 0x1234
```

; create 32- bit word data, **R1 = 0x12345678**

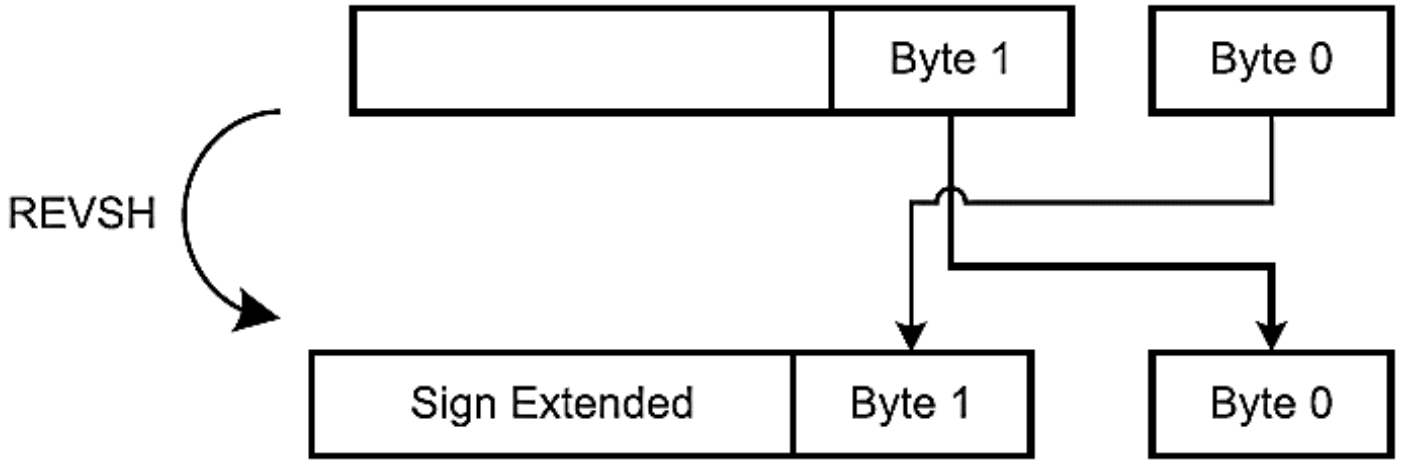
```
REV16   R4, R1
```

; byte reversal in each halfword, **R4 = 0x34127856**

BITFIED INSTRUCTIONS

1. Bit and byte reversal instructions

REVSH Rd, Rn



E.g. :

```
MOVW    R1, 0x12E3      ; lower Halfword data,
REVSH   R4, R1          ; sign extended byte reversal,
```

R1 = 0x12E3

R4 = 0xFFFFE312

1110

This sign bit will be extended

BITFIED INSTRUCTIONS

1. Bit and byte reversal instructions

RBIT Rd, Rn

E.g. :

```
MOVW R1, 0x5678
MOVT R1, 0x1234
```

; create 32- bit word data, **R1 = 0x12345678**

```
RBIT R3, R1
```

; bit reversal of data, **R3 = 0x1E6A2C48**

R1 = 00010010001101000101011001111000 (0x12345678)

R3 = 00011110011010100010110001001000 (0x1E6A2C48)

BITFIED INSTRUCTIONS

2. Bitfield Clear and Insert Instructions

BFC Rd, ***#lsb***, ***#width***

BFI Rd, Rn, ***#lsb***, ***#width***

where

Rd Specifies the destination register.

#lsb Specifies the position of the least significant bit of the bitfield to be modified. *lsb* must be in the range 0 to 31.

#width Specifies the width of the bitfield and must be in the range 1 to 32-*lsb*.

Rn Specifies the source register.

BITFIED INSTRUCTIONS

2. Bitfield Clear and Insert Instructions

E.g. : `LDR R1, =0x11223344`

`BFC R1, #4, #8` ; Clear bit 4 to bit 11 (8 bits) of R1 to 0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	1	0	0	1	1	0	1	0	0	0	1	0	0

R1

After execution of

`BFC R1, #4, #8`

0	0	0	1	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0

R1

R1 =0x11223004

BITFIED INSTRUCTIONS

2. Bitfield Clear and Insert Instructions

E.g. :
```
LDR R2, =0x5577AACC
MOV R3, #0x123
BFI R2, R3, #4, #12   ; Replace bit 4 to bit 15 (12 bits ) of R2 with bit 0 to bit 11 of R3
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	1	0	1	1	1	0	1	1	1	1	0	1	0	1	0	1	0	1	1	0	0	1	1	0	0

R2

After execution of

```
BFI R2, R3, #4, #12
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	1	0	1	1	1	0	1	1	1	0	0	0	1	0	0	1	0	0	0	1	1	1	1	0	0

R2

R1 =0x5577123C

BITFIED INSTRUCTIONS

3. Bitfield Extract Instructions

SBFX Rd, Rn, *#lsb*, *#width*

UBFX Rd, Rn, *#lsb*, *#width*

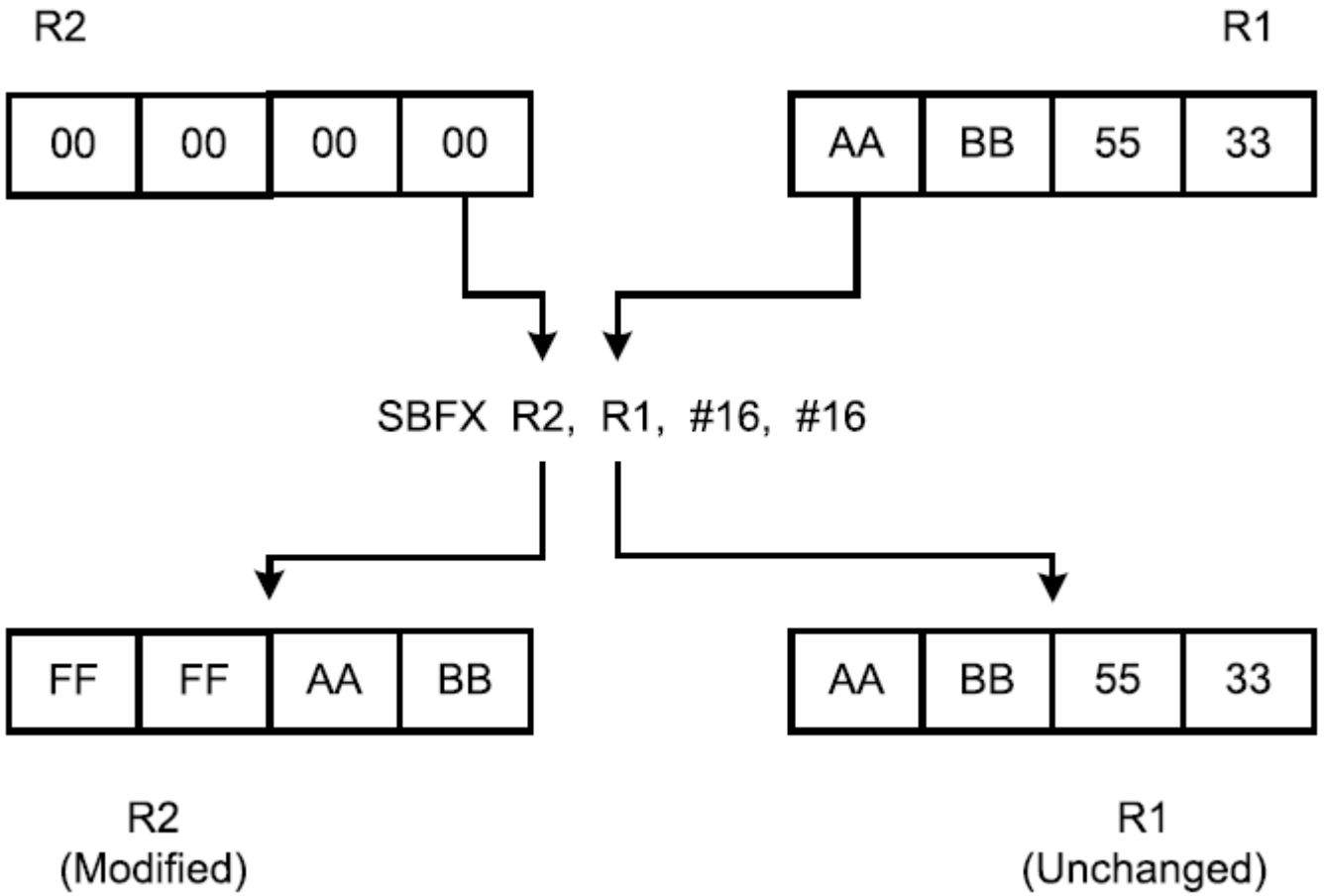
E.g. :

```
SBFX R2, R1, #16, #16      ; Extract bit 16 to bit 31 (16 bits) from R1
                           ; sign extend it to 32 bits
                           ; and finally write the result to R2.
UBFX R3, R1, #6, #10       ; Extract bit 6 to bit 15 (10 bits) from R1 and zero
                           ; extend to 32 bits and then write the result to R3.
```

BITFIED INSTRUCTIONS

3. Bitfield Extract Instructions

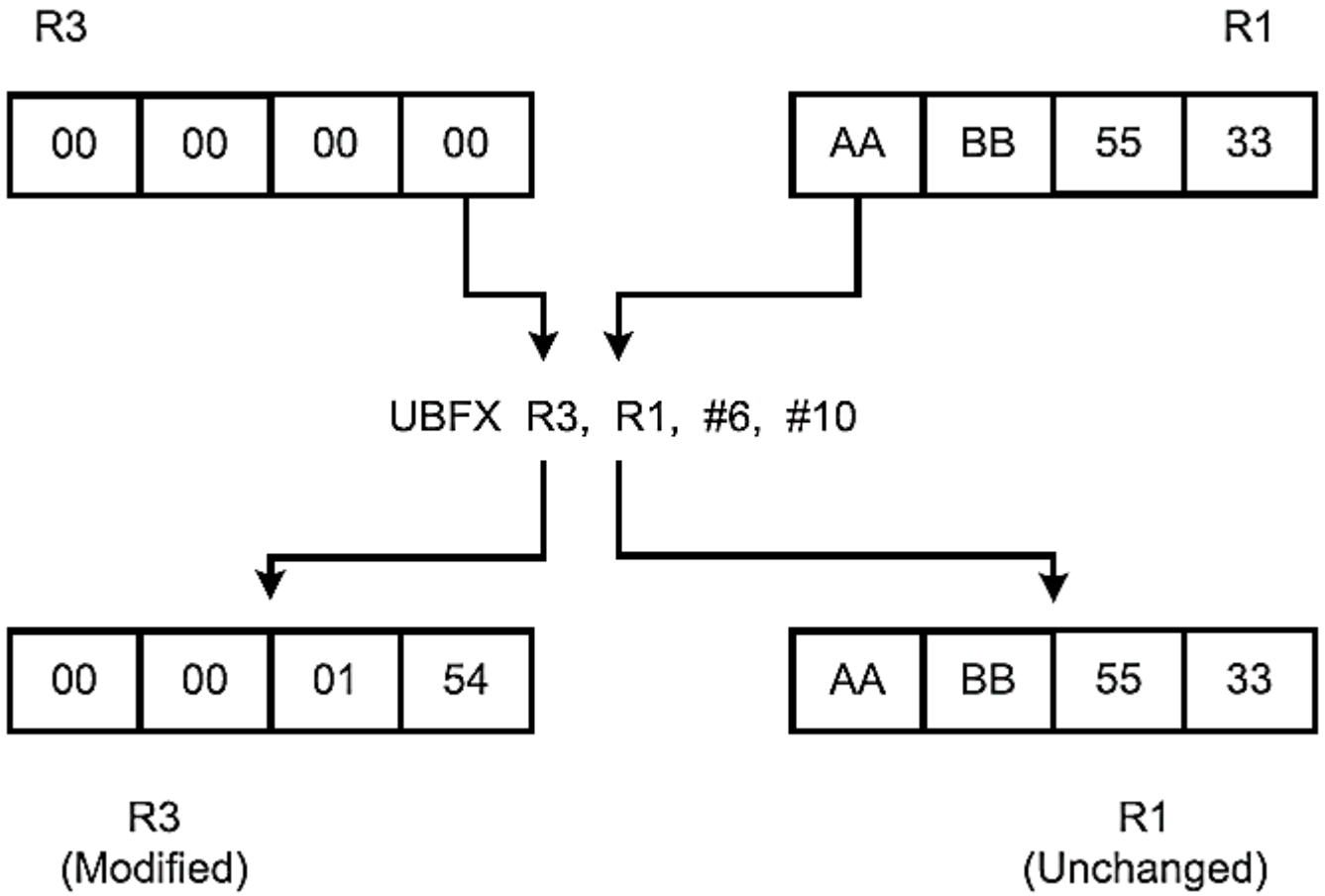
```
SBFX R2, R1, #16, #16
```



BITFIED INSTRUCTIONS

3. Bitfield Extract Instructions

```
UBFX R3, R1, #6, #10
```



BITFIED INSTRUCTIONS

Mnemonic	Brief description	Encoding	Flags
BFC	Bit Field Clear	32 bit	
BFI	Bit Field Insert	32 bit	
CLZ	Count leading zeros	32 bit	
RBIT	Reverse bits	32 bit	
REV	Reverse byte order in a word	16 or 32 bit	
REV16	Reverse byte order in each halfword	16 or 32 bit	
REVSH	Reverse byte order in bottom halfword & sign extend	16 or 32 bit	
SBFX	Signed Bit Field Extract	32 bit	
UBFX	Unsigned Bit Field Extract	32 bit	
SXT	Sign extend	16 or 32 bit	
UXT	Zero extend	16 or 32 bit	

BITFIED INSTRUCTIONS

- ✓ The bitfield instructions operate on the adjacent group of bits in the operand registers.
- ✓ It can be categorized further into the following subgroups
  - Bit and byte reversal instructions
  - Bitfield clear and insert instructions
  - Bitfield extract instructions
  - Miscellaneous bitfield instructions

BITFIED INSTRUCTIONS

4. Miscellaneous bitfield instructions

```
SXTB  Rd, Rm,  {ROR  #n}
SXTH  Rd, Rm,  {ROR  #n}
UXTB  Rd, Rm,  {ROR  #n}
UXTH  Rd, Rm,  {ROR  #n}
CLZ   Rd, Rm
```

- ✓ The SXT and UXT instructions optionally rotate right the value from the register Rm by 0, 8, 16, or 24 bits and then extract bits from the resulting value.
- ✓ SXTB instruction extracts 8 bits, i.e., bits [7:0] and sign extends it to 32 bits. SXTH instruction extracts 16 bits, bits [15:0] and sign extends to 32 bits. Similar for UXTB, UXTH with zero extended.

BITFIED INSTRUCTIONS

4. Miscellaneous bitfield instructions

E.g. :

```
SXTH R1, R2, ROR #16        ; Rotate R2 right by 16 bits, obtain
                            ; the lower halfword of the result
                            ; and then sign extend to 32 bits.
                            ; Write the result to R1.
UXTB R2, R1                 ; Extract lowest byte from source
                            ; R1 zero extend it and write the
                            ; result to register R2.
```

BITFIED INSTRUCTIONS

4. Miscellaneous bitfield instructions

- ✓ The CLZ instruction counts the number of leading zeros in the value provided by the register Rm and returns the result in destination register Rd.
- ✓ The result of CLZ instruction is 32 if all the bits are set to zero in the register Rm while it is zero if bit 31 of source register is equal to 1.
- ✓ It is possible to use CLZ instruction for data normalization.

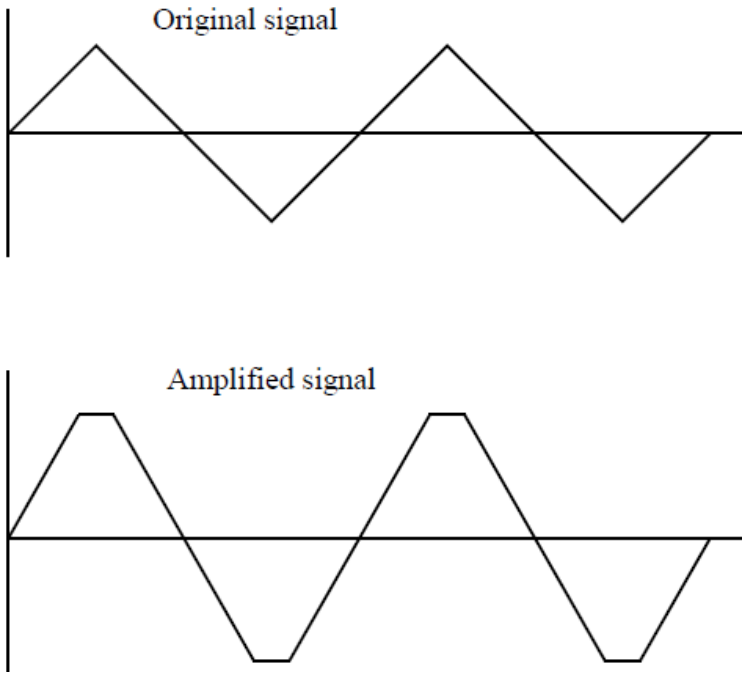
E.g. :

```
CLZ R5, R9
MOVS R9, R9, LSL R5
```

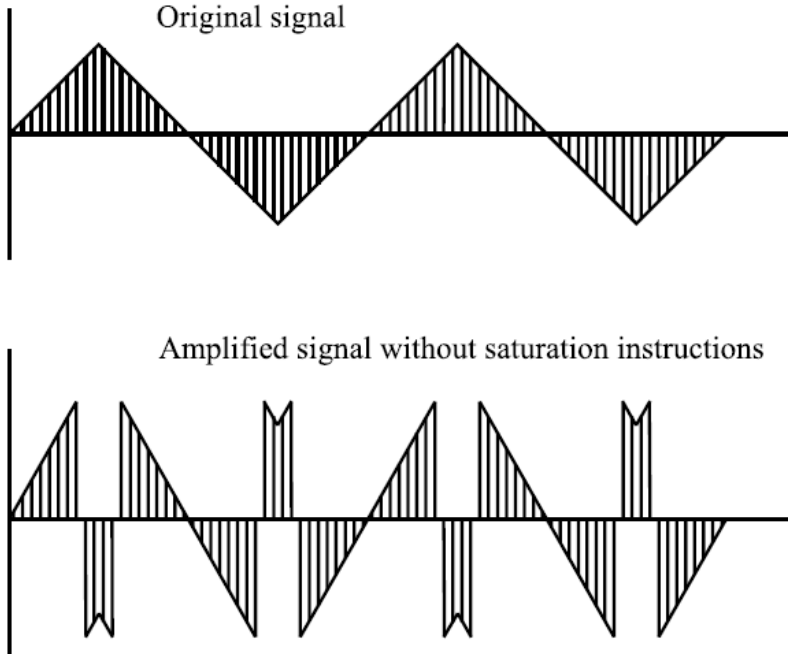
SATURATION INSTRUCTIONS

- ✓ One of the fundamental operations that is performed in the process of signal conditioning is the signal amplification and performed in either
  - Hardware - operation/instrumentation amplifiers are widely used for this purpose---signal clipping is experienced
  - Software - is effectively equivalent to multiplication of the sampled signal data with an integer
- ✓ When the signal amplification factor (also termed amplification gain) is too large, a saturation phenomenon is observed.
- ✓ For example consider an 8- bit resolution sampled data (bipolar data),the maximum values permitted for the signal range between -128 and +127. If the result of multiplication of an arbitrary sample value exceeds either the lower or upper limit, saturation will happen.
- ✓ This effectively is equivalent to the overflow condition and highly distorts the data.

SATURATION INSTRUCTIONS



Signal saturation due to hardware amplification

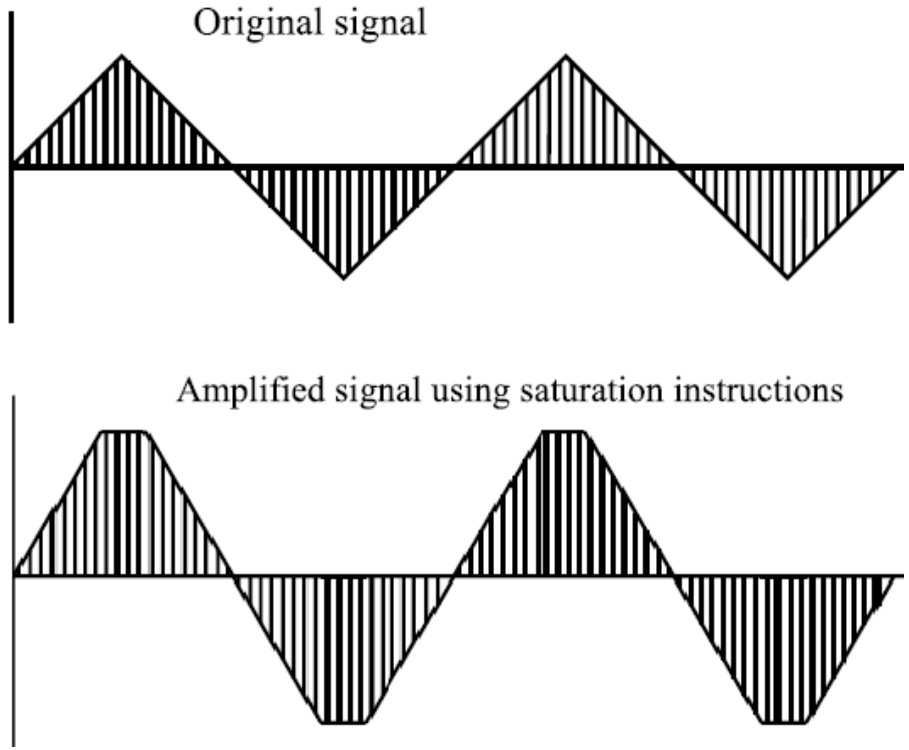


Signal saturation due to software amplification

- ✓ Let suppose to multiply 87 with an integer 2 for 8 bit resolution, the range is only -128 to +127 for signed and 0 to 255 for unsigned number. So the value which we get is 174, which exceeded the maximum limit, the corresponding hexa value is 0XAE (10101110) and it can be read as negative value of -52.

SATURATION INSTRUCTIONS

Mnemonic	Brief description	Encoding	Flags
USAT	Unsigned saturation	32-bit	Q
SSAT	Signed saturation	32-bit	Q



SATURATION INSTRUCTIONS

- ✓ The saturation instructions effectively avoid overflow condition by saturating the data similar to the saturation due to hardware amplification.
- ✓ The saturation instructions are used to saturate a given value, respectively, to a signed or unsigned n-bit final result.

```
SSAT Rd, #n, Rn, {shift #s}
USAT Rd, #n, Rn, {shift #s}
```

Where

#n	Species the most significant bit position to which the saturation action is performed to. The value of n ranges from 1 to 32 for SSAT and it ranges from 0 to 31 for USAT.
Rn	Species the register that contains the value to be saturated.
shift #s	This is an optional shift applied to register Rm before saturating. The shift operation must be one of the following: ASR #s with a range of s from 1 to 31. LSL #s with a range of s from 0 to 31.

SATURATION INSTRUCTIONS

SSAT Rd, #n, Rn, {shift #s}

Rn	Rd	Q
$-2^{(n-1)} \le Rn \le 2^{(n-1)} - 1$	Rd = Rn	Q = 0
$Rn > 2^{(n-1)} - 1$	$Rd = 2^{(n-1)} - 1$	Q = 1
$Rn < -2^{(n-1)}$	$Rd = -2^{(n-1)}$	Q = 1

E.g.

```
MOV   R1, #0x05
SSAT  R2, #4, R1
```

$-2^{(4-1)} \le R1 \le 2^{(4-1)} - 1$ i.e. $-8 \le R1 \le 7$

SATURATION INSTRUCTIONS

USAT Rd, #n, Rn, {shift #s}

Rn	Rd	Q
$0 \le Rn \le 2^n - 1$	Rd = Rn	Q = 0
$Rn > 2^n - 1$	$Rd = 2^n - 1$	Q = 1
Rn < 0	Rd = 0	Q = 1

E.g.

```
MOV   R1, #0x05
USAT  R2, #4, R1
```

$0 \le R1 \le 2^4 - 1$ i.e. $0 \le R1 \le 15$

SATURATION INSTRUCTIONS

- ✓ If saturation occurs, the instruction sets the 'Q' flag to 1 in the APSR. Otherwise, it leaves the Q flag unchanged.
- ✓ These instructions do not affect the condition code flags.
- ✓ To clear the Q flag to 0, you must use the MSR instruction.
- ✓ To read the state of the Q flag, the MRS instruction can be used.

SATURATION INSTRUCTIONS

Signed saturation instruction execution with 16-bit saturated result

Input value	Output value	Q Flag update
0x00012000	0x00007FFF	Flag set
0xFF3F0000	0xFFFF8000	Flag set
0x00008000	0x00007FFF	Flag set
0xFFFF8000	0xFFFF8000	Unaffected
0x00007FFF	0x00007FFF	Unaffected
0xFFFF7FFF	0xFFFF8000	Flag set

Unsigned saturation instruction execution with 12-bit saturated value

Input value	Output value	Q Flag update
0x00012000	0x00000FFF	Flag set
0xFF3F0000	0x00000000	Flag set
0x00000824	0x00000824	Unaffected
0x80000180	0x00000000	Flag set
0x70000180	0x00000FFF	Flag set

BRANCH AND CONTROL INSTRUCTIONS

- ✓ The Cortex-M processor supports different types of branch and control instructions with varying complexity.
  - Branch instructions (conditional and unconditional)
  - Function calls (conditional and unconditional)
  - Combined compare and conditional branch
  - Conditional execution of instructions (IF-THEN instruction)
  - Table branch

BRANCH AND CONTROL INSTRUCTIONS

- ✓ Following are some possible scenarios, where branch and control instructions can be used to implement required program flow control.
  - The conditional instructions in a high level language make use of branch and control instructions.
  - There can be an unconditional branch causing the user program to execute a specific code segment.
  - A conditional branch will call a function or execute a specific set of instructions only when a certain condition is fulfilled.
  - A function call requiring the processor to execute a specific code implemented by the function.
  - A return from a function call will return to the next instruction following the instruction that called the function.
  - An interrupt suspends the normal program execution and starts executing an interrupt service routine to make use of branch and control instructions.
  - A return from an interrupt service routine will return to the instruction where the program execution was before the interrupt has occurred.
  - A hardware or software error or bug will usually cause a bus fault and stops the normal execution.
  - The system exceptions as well as software interrupts use branch and control instructions.

BRANCH INSTRUCTIONS

Mnemonic	Brief description	Encoding	Flags
B	Branch	16 or 32 bit	-
BX	Branch indirect	16 bit	-
BL	Branch with link	32 bit	-
BLX	Branch indirect with link	16 bit	-

✓ The general syntax for branch instructions is given below

B {*cond*} label

BX {*cond*} Rm

Bl {*cond*} label

BLX {*cond*} Rm

BRANCH INSTRUCTIONS

{*cond*}

Suffix	Condition	Flags
EQ	EQual	Z = 1
NE	Not Equal	Z = 0
CS HS	Carry Set Higher or Same	C = 1
CC LO	Clear Carry LOwer	C = 0
MI	MInus	N = 1
PL	PLus	N = 0
VS	oVerflow Set	V = 1
VC	oVerflow Clear	V = 0
HI	unsigned HIgher	(C = 1 AND Z = 0)
LS	unsigned Lower or Same	(C = 0 OR Z = 1)
GE	signed Greater than or Equal	N = V
LT	signed Less Than	N ≠ V
GT	signed Greater Than	Z = 0 AND N = V
LE	signed Less than or Equal	Z = 1 OR N ≠ V

BRANCH INSTRUCTIONS

B	16/32	**Simple Branch**
B<c> <label>		PC ← label
Equivalent to a simple assignment of the instruction pointer.		

BL	32	**Branch with Link**
BL<c> <label>		LR ← return @ PC ← label
The return @ is the current value of the PC+4 with bit[0]=1.		

BRANCH INSTRUCTIONS

B	16/32	**Simple Branch**
B<c> <label>		PC ← label
Equivalent to a simple assignment of the instruction pointer.		

BL	32	**Branch with Link**
BL<c> <label>		LR ← return @ PC ← label
The return @ is the current value of the PC+4 with bit[0]=1.		

BRANCH INSTRUCTIONS

BLX	16	**Branch and eXchange with register link**
BLX<c> <Rm>		LR ← return @ PC ← Rm
This uses the same principle as BL. *The calling address is contained in the Rm register.* *It allows the management of "jump tables" or function pointers. The X of the mnemonic means eXchange,* *but in no case does Rm receive the previous value of PC.*		

BX	16	**Branch and eXchange by register**
BX<c> <Rm>		PC ← Rm
This instruction allows the return of procedure when the register is LR. X means the same as it does in BLX.		

CONDITIONAL BRANCH EXECUTION

- ✓ Conditional branch instructions can be broadly categorized in three groups:
  - Single-flag branch instructions
  - Signed branch instructions
    - ✓ that are used when operands are treated as signed numbers,
  - Unsigned branch instructions
    - ✓ which are used when the operands are interpreted as unsigned numbers.

CONDITIONAL BRANCH EXECUTION

Single-flag branch instructions	Unsigned branch instructions	Signed branch instructions
BEQ label	BLO label	BLT label
BNE label	BHS label	BGE label
BCS label	BLS label	BGT label
BCC label	BHI label	BLE label
BMI label		
BPL label		
BVS label		
BVC label		

COMBINED COMPARE AND CONDITIONAL BRANCH

Mnemonic	Brief description	Encoding	Flags
CBZ	Compare and branch if zero	16 bit	-
CBNZ	Compare and branch if not zero	16 bit	-

- ✓ The general syntax for these two instructions is give by

```
CBZ   Rn, label
CBNZ  Rn, label
```

COMBINED COMPARE AND CONDITIONAL BRANCH

CBZ, CBNZ	16	**Conditional Branch on the Nullity of a register**
CBZ<c> <Rm> <label>		PC ← label if (Rm = 0)
CBNZ<c> <Rm> <label>		PC ← label if (Rm≠0)
This instruction does not modify the flags and allows us to carry out a conditional branch without carrying out the prior test. The jump must be "forward" and limited to 126 bytes (containing a maximum of 62 instructions).		

COMBINED COMPARE AND CONDITIONAL BRANCH

Usage of CBZ

Example: Iterative function calling

```
i=6;
while(i!=0)
{
      funcA();
      i--;
}
```

```
      MOV R0, #6              ; Set loop counter
loop1
      CBZ R0, loop1exit       ; If loop counter = 0
                              ; then exit loop
      BL funcA                ; Call a function
      SUB R0, #1        ; Loop counter decrement
      B loop1           ; Next loop iteration
loop1exit
```

COMBINED COMPARE AND CONDITIONAL BRANCH

Usage of CBNZ

Example:

```
status = strchr(emailid, '@');
// if emailid does not contain @, then status is 0.
if ( status == 0){
     generate_error_message ();
}
```

```
     ...
     CBNZ R0, email_id_ok        ; R0 contains a character from the emailid string
     BNE generate_error_message
email_id_ok
     ...
```

PASSING PARAMETERS TO FUNCTIONS

- ✓ In programming languages, there are two well-known mechanisms for parameter passing to functions,
  - call by value
  - call by reference
- ✓ In the first strategy, the actual data values are passed to the called function, while in the latter method the address of the data is passed to the function.
- ✓ The latter method is particularly useful when dealing with data arrays.

CALL BY VALUE

- ✓ For R4 = M + N - R3, where M and N are 32-bit data. We implement addition as a function and pass the operands through the call by value method.

```
int sum(int M, int N)
{
      int c=M+N;
      return c;
}
void main()
{
      int a = 5;
      int b = 3;
      int R3 = 3;
      int c = sum(a, b);
      R4 = c-R3;
}
```

CALL BY VALUE

- ✓ For R4 = M + N - R3, where M and N are 32-bit data. We implement addition as a function and pass the operands through the call by value method.

```
main
        LDR R6, =M          ; M=0xF0000100
        LDR R7, =N          ; N=0xF0000200
        LDR R0, [R6]
        LDR R1, [R7]
        BL  addition        ; Branch to function 'addition ' and
                            ; pass R0 and R1 as parameters
        MOV R3, #0x3
        SUB R4, R1, R3
        B   stop
addition
        ADD R1, R1, R0      ; R1 = R1 + R0
        BX  LR
stop    B   stop
        END
```

CALL BY REFERENCE

- ✓ For R4 = M + N - R3, where M and N are 32-bit data. We implement addition as a function and pass the operands through the call by value method.

```
int sum(int *M, int *N)
{
        int c = *M + *N;
        return c;
}
void main()
{
        int a = 5;
        int b = 3;
        int R3 = 3;
        int c = sum(&a, &b);
        R4 = c - R3;
}
```

CALL BY REFERENCE

- ✓ For R4 = M + N - R3, where M and N are 32-bit data. We implement addition as a function and pass it the operands through the call by reference method.

```
main
        LDR R6, =M              ; M=0xF0000100
        LDR R7, =N              ; N=0xF0000200
        BL  addition            ; Branch to function 'addition '
        MOV R3, #0x3
        SUB R4, R1, R3
        B   stop
addition
        LDR R0, [R6]
        LDR R1, [R7]
        ADD R1, R1, R0          ; R1 = R1 + R0
        BX  LR
stop    B   stop
        END
```

EXAMPLE-1

Automated Gate

EXAMPLE-1

✓ Write an ALP for Automated Gate. When any car in front of Gate then Gate will OPEN. When NO car in front of Gate then Gate will CLOSED.

Assume

- IR value is available in 0xFF00FF11 memory address.
- When any car in front of gate then AAH will be provide by IR sensor.
- To OPEN gate, Store FFH value in 0xFF11FF00 memory address.
- To CLOSE gate, Store 00H value in 0xFF11FF00 memory address.

EXAMPLE-1

Read the IR sensor

```
If(IR = 0xAA)
{
   Gate OPEN;
}
else
{
   Gate CLOSE;
}
```

```
            MOV R0, #0xAA
            LDR R1, =0xFF00FF11
LOOP:       LDR R2, [R1], #1
            CMP R2, R0
            BNE SKIP
            MOV R3, #0xFF          ; Open gate
            B Forward
SKIP:       MOV R3, #0x00          ; Close gate
Forward:    LDR R4, =0xFF11FF00
            STR R3, [R4], #1
            B LOOP
STOP        B STOP
            END
```

EXAMPLE-2

Temperature Monitoring Unit of Food Industry

EXAMPLE-2

- ✓ Write an ALP for Temperature Monitoring Unit of Food Industry.
  - If Temp>20^0 C then Cooler must be ON
  - If Temp<20^0 C then Cooler must be OFF

Assume

- ❖ Temp value is available in 0xFF00FF11 memory address.
- ❖ 20^0 C in Hexadecimal value is 14H.
- ❖ For ON of Cooler, Store FFH value in 0xFF11FF00 memory address.
- ❖ For OFF of Cooler, Store 00H value in 0xFF11FF00 memory address.

EXAMPLE-2

Read the Temperature

```
If(Temp>20⁰C)
{
   Cooler ON;
}
else
{
   Cooler OFF;
}
```

```
            MOV R0, #0x14
            LDR R1, =0xFF00FF11
LOOP:       LDR R2, [R1], #1
            CMP R2, R0
            BLE SKIP
            MOV R3, #0xFF        ; Cooler ON
            B Forward
SKIP:       MOV R3, #0x00        ; Cooler OFF
Forward:    LDR R4, =0xFF11FF00
            STR R3, [R4], #1
            B LOOP
STOP        B STOP
            END
```

EXAMPLE-3

Automated Light

EXAMPLE-3

- ✓ Write an ALP for Automated Light system.
  - When person is present in room then LED light must be ON.
  - When no person is present in room then LED light must be OFF.

Assume

- ❖ PIR sensor value is available in 0xFF00FF11 memory address.
- ❖ When any person is present then DDH will be provide by PIR sensor.
- ❖ For ON LED, Store ADH value in 0xFF11FF00 memory address.
- ❖ For OFF LED, Store BCH value in 0xFF11FF00 memory address.

EXAMPLE-3

Read the Temperature

```
If(PIR= 0xDD)
{
   LED ON;
}
else
{
   LED OFF;
}
```

```
            MOV R0, #0xDD
            LDR R1, =0xFF00FF11
LOOP:       LDR R2, [R1], #1
            CMP R2, R0
            BNE SKIP
            MOV R3, #0xAD       ; LED ON
            B Forward
SKIP:       MOV R3, #0xBC       ; LED OFF
Forward:    LDR R4, =0xFF11FF00
            STR R3, [R4], #1
            B LOOP
STOP        B STOP
            END
```

EXAMPLES

- ✓ Finding the maximum of a set of numbers

R0 – Pointer for the data
R1 – Stores the largest number
R2 – Stores the total number of numbers

```
    LDR R0, =0xF0000100
    EOR R1, R1, R1          ;clear R1 to store the largest
    CMP R2, #0
    BEQ Over                ;if block is empty, done
Loop
    LDR R3, [R0]            ;get the data
    CMP R3, R1              ;do comparison
    BCC Looptest            ;skip if R1 is bigger
    MOV R1, R3              ;else get the larger in R1
Looptest
    ADD R0, R0, #4          ;increment pointer R0
    SUBS R2, R2, #1         ;decrement number of elements left
    BNE Loop                ;if not done, loop
Over
                            ;R1 holds the largest
```

EXAMPLES

- ✓ Comparing two null terminated strings

```
Loop
        LDRB R3, [R0]     ;get next character of string 1
        LDRB R4, [R1]     ;get next character of string 2
        CMP R3, R4        ;compare
        BNE Notsame       ;if not same, strings do not match
        CMP R3, #0        ;check if end of string reached
        BEQ Same          ;if equal, same
        ADD R0, R0, #1    ;increment pointer to string 1
        ADD R1, R1, #1     ;increment pointer to string 2
        B   Loop          ;branch always to check next character
Notsame
        MOV R2, #-1       ;mark not matched
        B   Over
Same
        MOV R2, #0        ;mark matched
Over
                          ;R2 holds the match
```

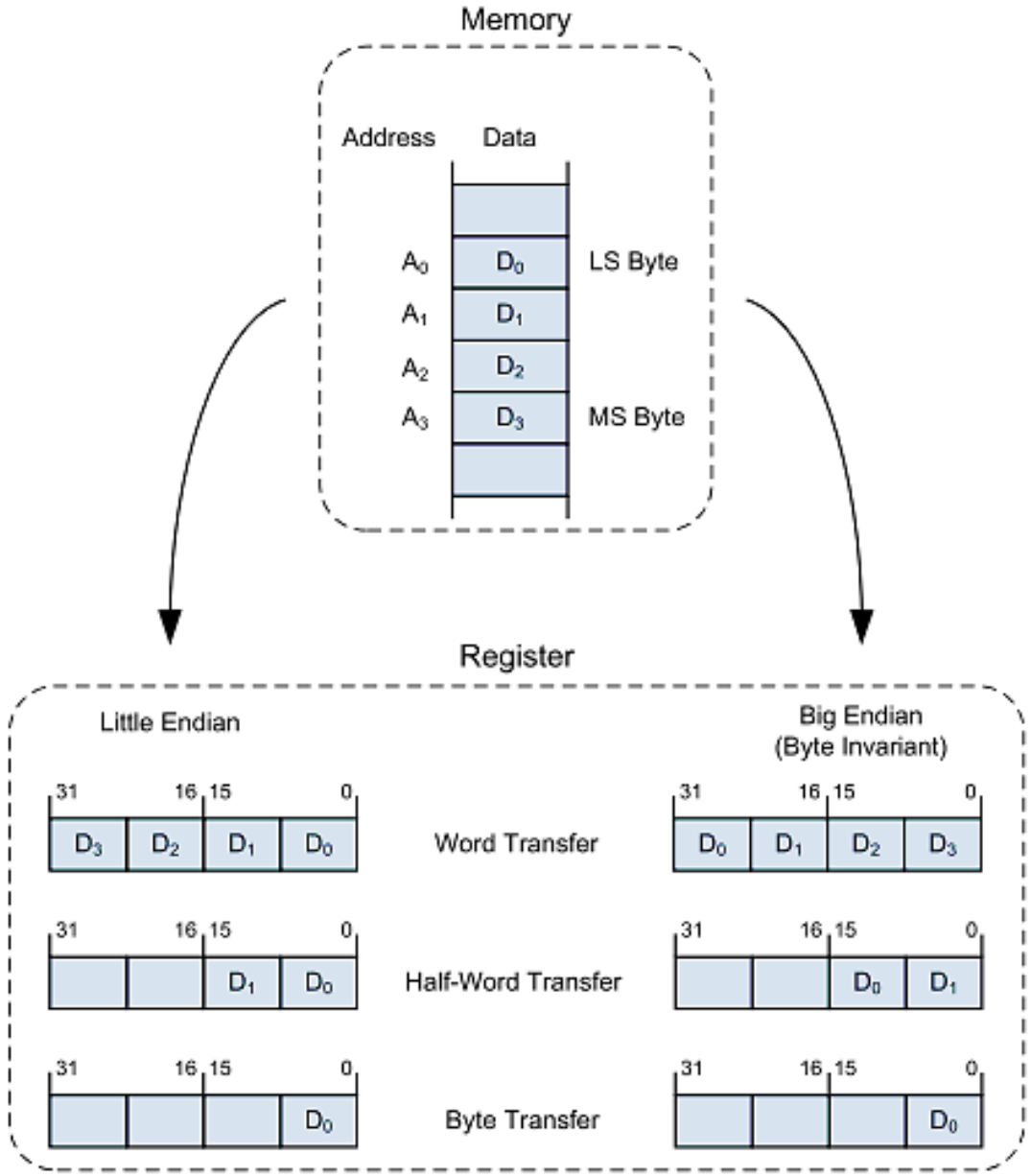
MEMORY ENDIANNESS

- ✓ The processor views memory as a linear collection of bytes numbered in ascending order from zero.
- ✓ For example, bytes 0-3 hold the first stored word, and bytes 4-7 hold the second stored word. Similarly, bytes 0-1 and 2-3 hold first and second halfwords, respectively.
- ✓ This byte order in the construction of halfwords and byte or halfword order for the construction of words is called the memory endianness.
- ✓ The are two possible types of memory endianness, namely little endian and big endian.

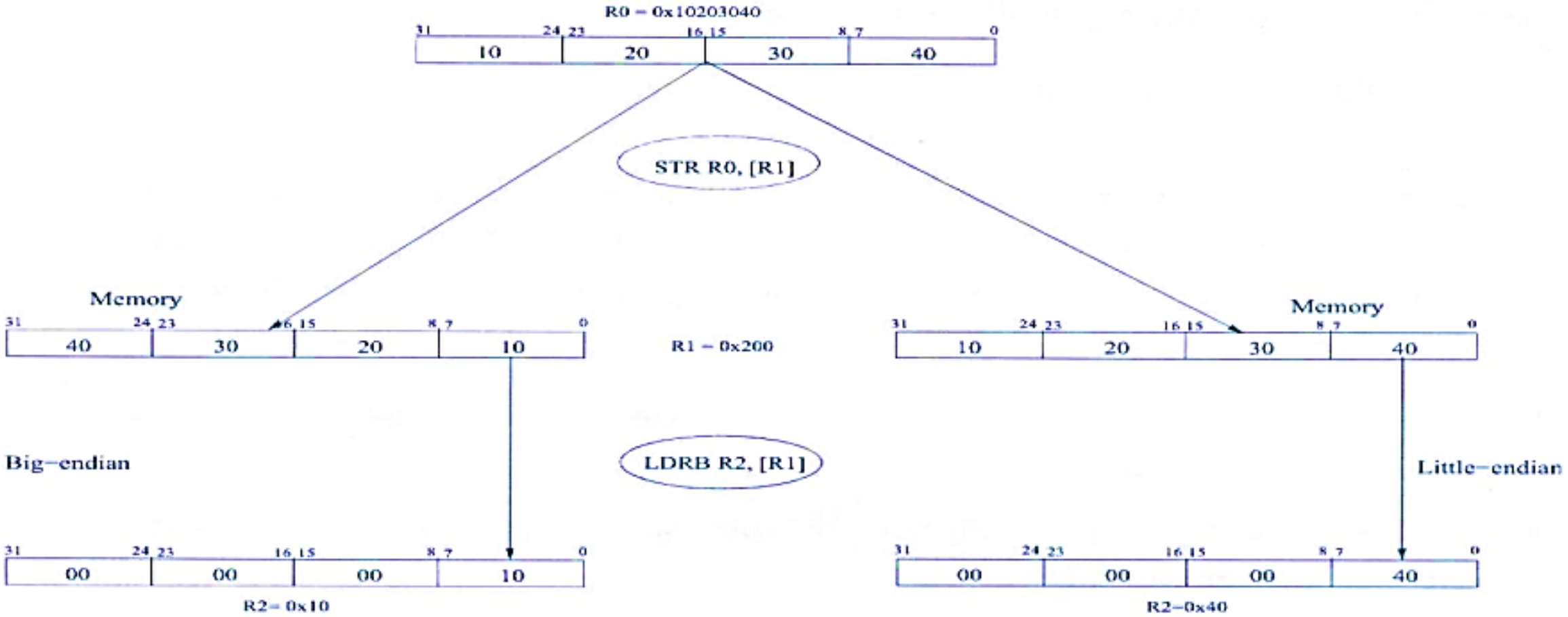
MEMORY ENDIANNESS

- ✓ Little Endian: In little-endian format, the processor stores the least significant byte (halfword) of a word at the lowest-numbered byte (halfword) address, and the most significant byte (halfword) at the highest-numbered byte (halfword) address. Similarly, the least significant byte in a halfword occupies the lower address.
- ✓ Big Endian: In big-endian format, the processor stores the most significant byte of a word at the lowest byte address, and the least significant byte at the highest byte address.

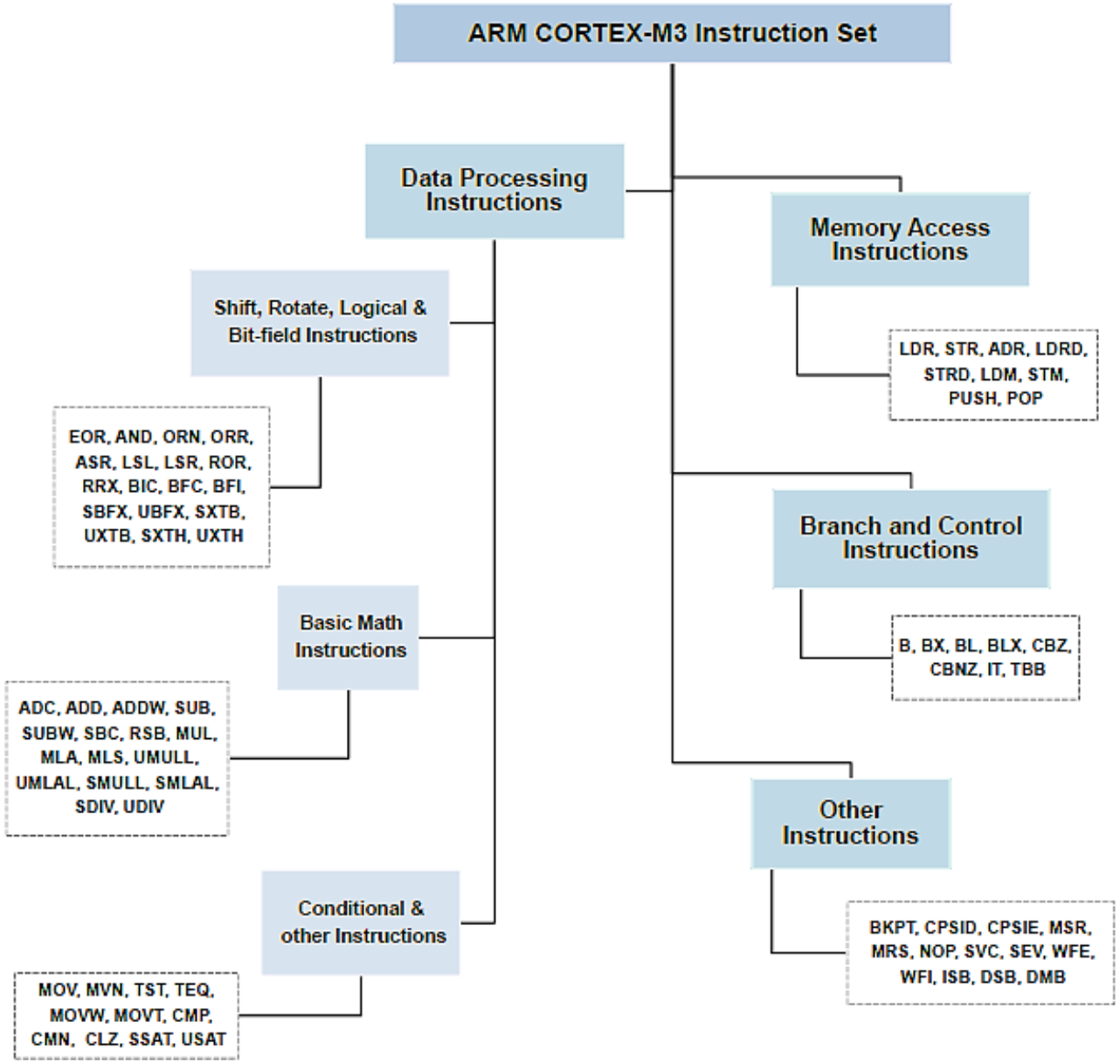
LITTLE ENDIAN VS BIG ENDIAN



LITTLE ENDIAN VS BIG ENDIAN



ARM CORTEX M3 INSTRUCTION SET



MEMORY ACCESS INSTRUCTIONS

Mnemonic	Brief description	Encoding	Flags
LDR	Load register using immediate offset	16 or 32 bit	No change
STR	Store register using immediate offset	16 or 32 bit	No change
LDM	Load multiple registers	16 or 32 bit	No change
STM	Store multiple registers	16 or 32 bit	No change
ADR	Generate address relative to PC	16 or 32 bit	No change
POP	Pop registers from stack	16 or 32 bit	No change
PUSH	Push registers onto stack	16 or 32 bit	No change

MEMORY ACCESS INSTRUCTIONS

Double Word Memory Accesses

✓ These instructions allow memory accesses for two data words.

```
LDRD {cond}   Rt1,   Rt2,   [ Rn,  #offset ]
LDRD {cond}   Rt1,   Rt2,   [ Rn,  #offset ]!
LDRD {cond}   Rt1,   Rt2,   [ Rn ], #offset
LDRD {cond}   Rt1,   Rt2,   label

STRD {cond}   Rt1,   Rt2,   [ Rn,  #offset ]
STRD {cond}   Rt1,   Rt2,   [ Rn,  #offset ]!
STRD {cond}   Rt1,   Rt2,   [ Rn ], #offset
STRD {cond}   Rt1,   Rt2,   label
```

MEMORY ACCESS INSTRUCTIONS

Double Word Memory Accesses

E.g.

```
LDRD   R1, R0, [R2, #0x20]   ; immediate offset addressing
                             ; mode

STRD   R3, R4, [R7], #-16    ; post indexed offset
                             ; addressing mode
```

- ✓ The first instruction loads register R1 with a word at an address obtained by adding an offset of 32 bytes to the value in register R2, and loads a word to R0 from an offset of 36 bytes.
- ✓ The second instruction stores R3 to an address in R7, and the register R4 to an address equal to R7+4 and then decrements R7 by 16.

MEMORY ACCESS INSTRUCTIONS

Multiple Word Memory Accesses

✓ These instructions allow memory accesses for multiple data words.

LDM {*addrmode*} {*cond*} Rd{!}, reglist

STM {*addrmode*} {*cond*} Rd{!}, reglist

✓ The optional suffix {*addrmode*} can be either increment address (IA) after each access or decrement address before (DB) each access.

MEMORY ACCESS INSTRUCTIONS

Multiple Word Memory Accesses

✓ Instructions

LDMIA/STMIA	; Increment After
LDMIB/STMIB	; Increment Before
LDMDA/STMDA	; Decrement After
LDMDB/STMDB	; Decrement Before

MEMORY ACCESS INSTRUCTIONS

Multiple Word Memory Accesses

Example –

```
; R12 points to start of source data
; R14 points to the end of source data
; R13 points to the start of the destination data
Loop    LDMIA R12!, {R0-R11}; load 48 bytes
        STMIA R13!, {R0-R11}; and store them
        CMP R12, R14;         check for the end
        BNE Loop;             and loop until done
```

MEMORY ACCESS INSTRUCTIONS

Stack Memory Accesses

✓ The general syntax for these two instructions is given below.

PUSH{*cond*} reglist

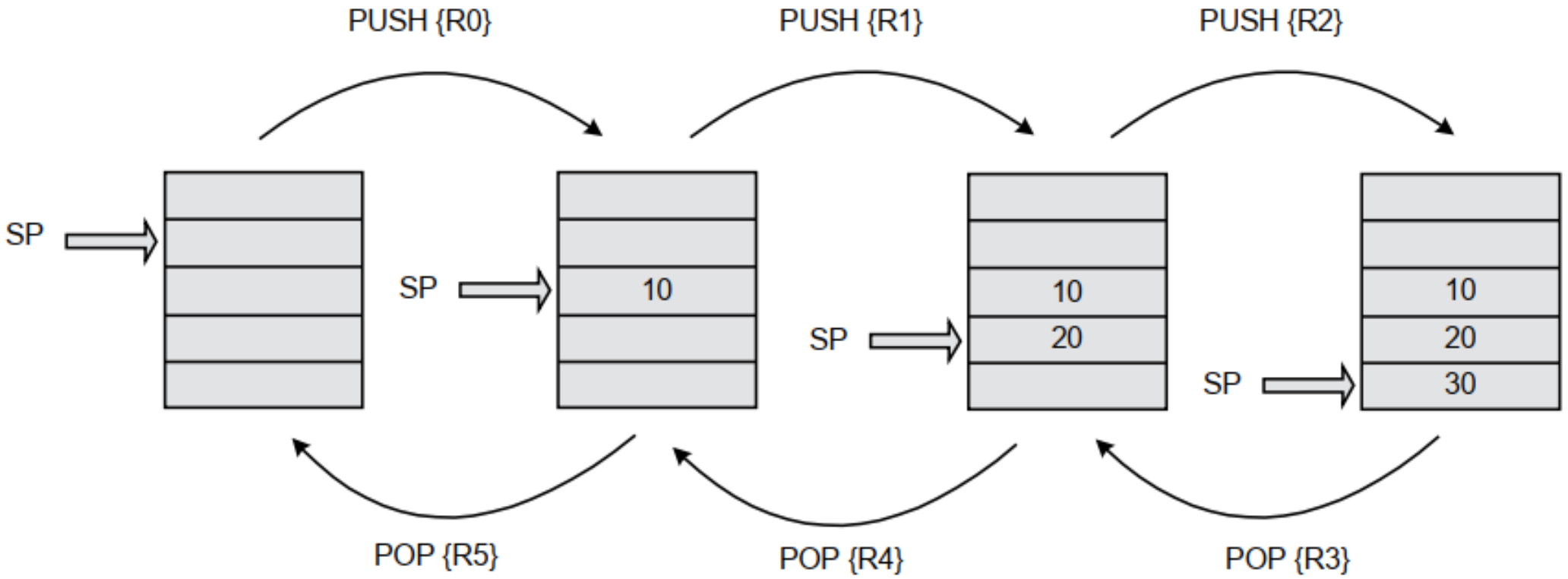
POP{*cond*} reglist

E.g.

```
PUSH {R0}      ; push the 32- bit value of R0 onto the stack
PUSH {R1}      ; push the 32- bit value of R1 onto the stack
PUSH {R2}      ; push the 32- bit value of R2 onto the stack
POP {R3}       ; retrieve a 32- bit value from the stack and store in R3
POP {R4}       ; retrieve a 32- bit value from the stack and store in R4
POP {R5}       ; retrieve a 32- bit value from the stack and store in R5
```

MEMORY ACCESS INSTRUCTIONS

Stack Memory Accesses



BRANCH AND CONTROL INSTRUCTIONS

- ✓ The Cortex-M processor supports different types of branch and control instructions with varying complexity.
  - Branch instructions (conditional and unconditional)
  - Function calls (conditional and unconditional)
  - Combined compare and conditional branch
  - **Conditional execution of instructions (IF-THEN instruction)**
  - Table branch

BRANCH AND CONTROL INSTRUCTIONS

If-Then Conditional Instruction Block

- ✓ Conditional execution of an instruction is implemented by Thumb2 ISA using If-Then (IT) block.
- ✓ The IT block instruction is very useful for handling small conditional codes.
- ✓ It avoids branch penalties because there is no change to program flow.
- ✓ The IT instructions allow up to four succeeding instructions to be conditionally executed and they collectively form an IT block.
- ✓ Conditional instructions, except for conditional branches, must be inside an IT instruction block.

IT{*x*{*y*{*z*}}} *cond*

BRANCH AND CONTROL INSTRUCTIONS

{*cond*}

Suffix	Condition	Flags
EQ	EQual	Z = 1
NE	Not Equal	Z = 0
CS HS	Carry Set Higher or Same	C = 1
CC LO	Clear Carry LOwer	C = 0
MI	MInus	N = 1
PL	PLus	N = 0
VS	oVerflow Set	V = 1
VC	oVerflow Clear	V = 0
HI	unsigned HIgher	(C = 1 AND Z = 0)
LS	unsigned Lower or Same	(C = 0 OR Z = 1)
GE	signed Greater than or Equal	N = V
LT	signed Less Than	N ≠ V
GT	signed Greater Than	Z = 0 AND N = V
LE	signed Less than or Equal	Z = 1 OR N ≠ V

BRANCH AND CONTROL INSTRUCTIONS

If-Then Conditional Instruction Block

IT{*x*{*y*{*z*}}} *cond*

- ✓ In the above instruction syntax
  - *x*, *y*, and *z* are the optional conditional execution switches for second, third, and fourth instructions in the IT block.
  - *cond* is the base condition for the IT instruction block. The first instruction following IT instruction is executed if the *cond* is true.
- ✓ Each of the optional condition switches x, y, and z can be either T (THEN) or E (ELSE).
- ✓ When condition switch T is used, then *cond* is applied to the corresponding instruction, whereas use of condition switch E applies the inverse *cond* to the corresponding instruction.

BRANCH AND CONTROL INSTRUCTIONS

If-Then Conditional Instruction Block

- ✓ When condition switch T is used, then *cond* is applied to the corresponding instruction, whereas use of condition switch E applies the inverse *cond* to the corresponding instruction.
- ✓ The *cond* operand in IT instruction uses the same condition codes as conditional branch.
- ✓ The condition code suffix appended to an instruction inside an IT block requires the processor to test the condition code based on the current status of the flags.

BRANCH AND CONTROL INSTRUCTIONS

If-Then Conditional Instruction Block

Example: if-then condition

```
if (x<y)
{
      z = x + y;
      p = z*x;
      p = p/2;
}
```

```
CMP     R0 , R1          ; Compare parameters
                         ; loaded to R0 with R1
ITTT    LT               ; If R0 < R1
ADDLT   R2, R0, R1       ; add the two operands
MULLT   R3, R2, R0       ; multiply the sum with first
                         ; operand in R0
ASRLT   R3, R3, #1       ; divide the result by 2
```

BRANCH AND CONTROL INSTRUCTIONS

If-Then Conditional Instruction Block

Example: If-then-else condition

```
if (x<y)
{
      z = x + y;
      p = z*x;
}
else
{
      z = x - y;
      z = z/2;
}
```

```
CMP     R0 , R1          ; Compare parameters
                         ; loaded to R0 with R1

ITTEE   LT               ; If R0 < R1
ADDLT   R2, R0, R1       ; part of if
MULLT   R3, R2, R0       ; part of if
SUBGE   R2, R0, R1       ; part of else
ASRGE   R2, R2, #1       ; part of else
```

BRANCH AND CONTROL INSTRUCTIONS

If-Then Conditional Instruction Block

Example: Nest-If condition

```
if(R0 > R1)
{
    if(R1 > R2)
    {
        if(R2 > R3)
        {
            R4 = 0x123;
        }
    }
}
```

```
CMP     R0, R1          ; Compare R0 with R1
ITTT    GT              ; Condition to check if R0>R1
SUBSGT  R5, R1, R2      ; check if R1 > R2
SUBSGT  R5, R2 , R3     ; check if R2 > R3
MOVGT   R4, #0x123
```

SPECIAL INSTRUCTIONS

MSR and MRS Instructions

- ✓ The access to special resisters is not permitted using either MOV or LDR/STR instructions.
- ✓ The following two instructions provide access to the special registers in the Cortex-M3.

MRS{*cond*} Rd *spec_reg*

MSR{*cond*} *spec_reg* Rn

- ✓ The first instruction, MRS, is responsible for moving the contents from a special register to one of the processor general purpose registers. The second instruction, MSR, performs a move from the general purpose register to a special register.

SPECIAL INSTRUCTIONS

MSR and MRS Instructions

- ✓ The MRS and MSR instructions can be used in privileged mode only with the exception of APSR special register which can be read in unprivileged mode as well. The registers used by MSR and MRS are

Register	Description
APSR	Application program status register. The status registers are read using MRS instruction.
IPSR	Interrupt status register.
MSP	Main stack pointer. However, stack pointer(s) is(are) also accessible using other assembly instructions as well.
PSP	Process stack pointer.
PRIMASK	Priority mask register.
BASEPRI	Base priority register.
FAULTMASK	Fault mask register.
CONTROL	Control register.