MOV Instruction
MOV RD, op2 ; op2 can be a register or 8 bit constant

MOVT Move Top

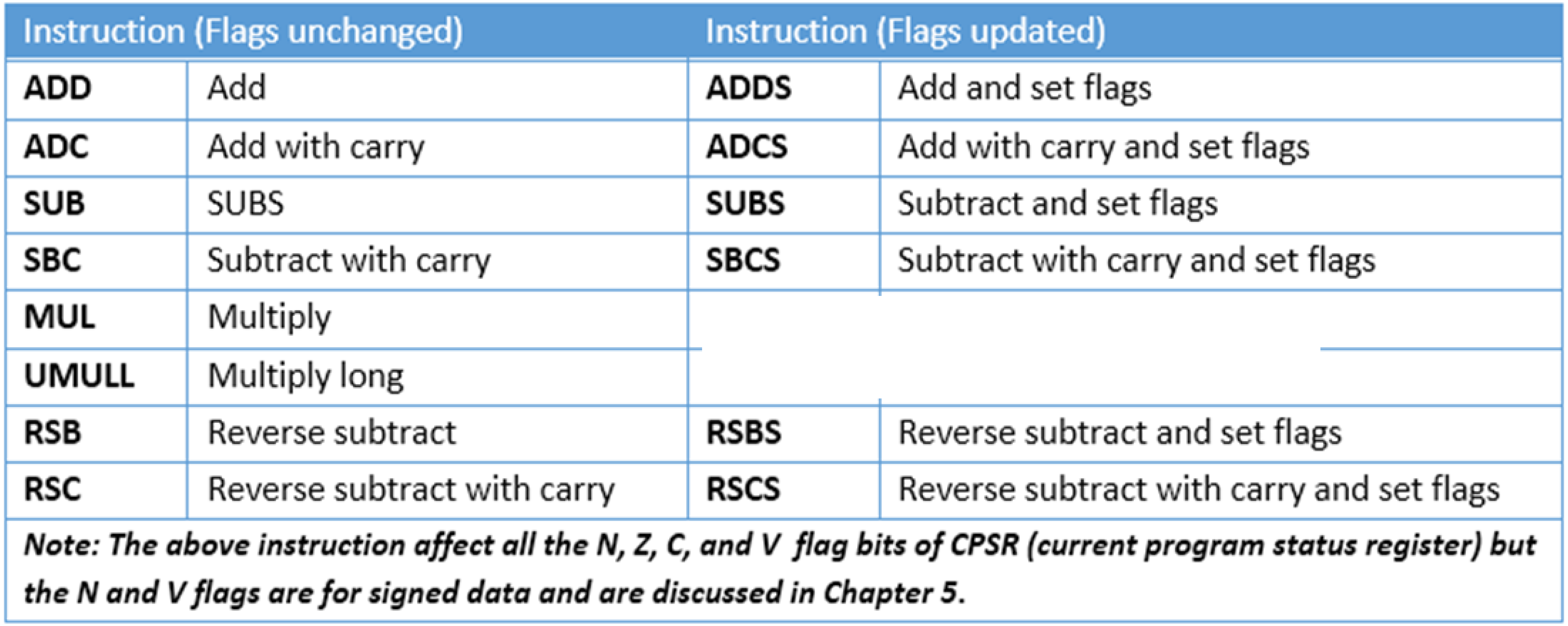
Format: MOVT Rd,#imm_value ;imm_value < 0x10000
Loads the upper 16-bit of Rd register with an immediate value. The
immediate value cannot be larger than 0xFFFF (0–65535). The lower 16-bit of the Rd register remains unchanged.

MOVS Move (and update flags)

Flags: Affected: C, N, Z
Format: MOV Rd,#immediate_value
Function: Load the Rd register with an immediate value and update the flags.

Example:
MOVS R0,#0x25 ;R0=0x25, N=0,Z=0, and C=0
MOVS R0,#0x0 ;R0=0x0, N=0,Z=1, and C=0

Arithmetic and Logic Instructions



Instruction (Flags unchanged)		Instruction (Flags updated)	
ADD	Add	**ADDS**	Add and set flags
ADC	Add with carry	**ADCS**	Add with carry and set flags
SUB	SUBS	**SUBS**	Subtract and set flags
SBC	Subtract with carry	**SBCS**	Subtract with carry and set flags
MUL	Multiply		
UMULL	Multiply long		
RSB	Reverse subtract	**RSBS**	Reverse subtract and set flags
RSC	Reverse subtract with carry	**RSCS**	Reverse subtract with carry and set flags
Note: The above instruction affect all the N, Z, C, and V flag bits of CPSR (current program status register) but the N and V flags are for signed data and are discussed in Chapter 5.			

Arithmetic Instructions and Flag Bits for Unsigned Data

If suffix S is used after the opcode, CPSR register will be effected by the result. Instructions without S executes without having any effect on the flags

```
ADD Rd,Rn,Op2        ;Rd = Rn + Op2
ADC Rd,Rn,Op2        ;Rd = Rn + Op2 + C
```

- The destination operand must be a register. The Op2 operand can be a register or immediate.
- Remember that memory-to-register or memory-to-memory arithmetic and logic operations are never allowed in ARM Assembly language since it is a RISC processor.

The instruction could change any of the Z, C, N, or V bits of the status flag register, as long as we use the ADDS or ADCS instead of ADD or ADC. ADC is used in the addition of multiword data.

Show the flag bits of status register for the following cases:

a) LDR R2,=0xFFFF FFF5

MOV R3,#0x0B

ADDS R1,R2,R3

a)

```
0xFFFFFFF5:    1111  1111  1111  1111  1111  1111   1111  0101
+
0x0000000B:     0000  0000   0000  0000  0000  0000  0000 1011
0x100000000   1 0000 0000 0000 0000 0000 0000 0000 0000
```

C = 1, since there is a carry out from D31
Z = 1, the result of the action is zero (for the 32 bits)

b) LDR R2,=0xFFFF FFFF

ADDS R1,R2,#0x95

b)

```
0xFFFFFFFF:      1111 1111 1111 1111 1111 1111 1111 1111
+
0x00000095:       0000 0000 0000 0000 0000 0000 1001 0101
 0x100000094:  1 0000 0000 0000 0000 0000 0000 1001 0100
```

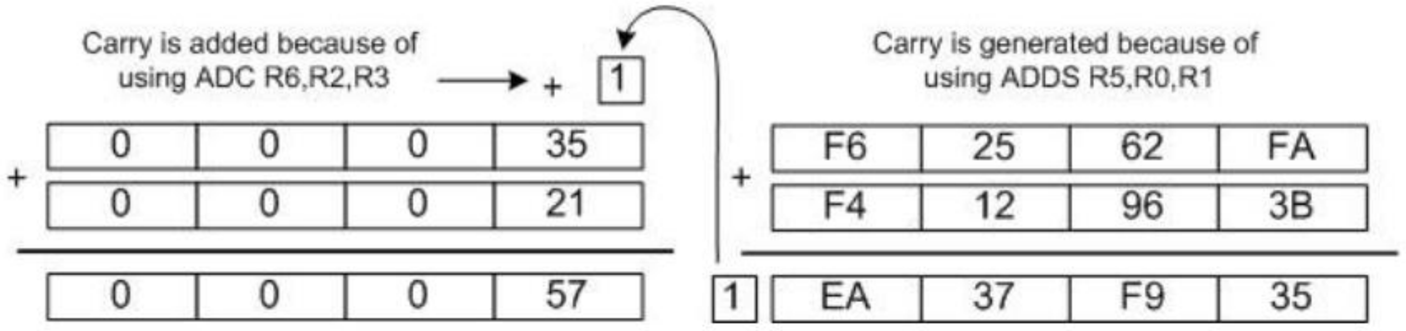
C = 1, since there is a carry out from D31

Z = 0, the result of the action is not zero (for the 32 bits)

Analyze the following program which adds 0x35F62562FA to 0x21F412963B:

```
LDR     R0,=0xF62562FA          ;R0 = 0xF62562FA
LDR     R1,=0xF412963B          ;R1 = 0xF412963B
MOV    R2,#0x35          ;R2 = 0x35
MOV    R3,#0x21          ;R3 = 0x21
ADDS    R5,R1,R0          ;R5 = 0xF62562FA + 0xF412963B
;now C = 1
ADC     R6,R2,R3          ;R6 = R2 + R3 + C
;   = 0x35 + 21 + 1 = 0x57
```

After the R5 = R0 + R1 the carry flag is one. Since C = 1, when ADC is executed, R6 = R2 + R3 + C = 0x35 + 0x21 + 1 = 0x57.


Carry is added because of using ADC R6,R2,R3
+ 1
0 0 0 35
+ 0 0 0 21
0 0 0 57
Carry is generated because of using ADDS R5,R0,R1
F6 25 62 FA
+ F4 12 96 3B
1 EA 37 F9 35

SUB Rd,Rn,Op2 ;Rd = Rn - Op2

In ARM SUB instruction is executed as follows:

1. Take the 2's complement of the subtrahend (Op2 operand).
2. Add it to the minuend (Rn operand).
3. Place the result in destination Rd.
4. Set the carry flag if there is a carry.

These four steps are performed for every SUBS instruction by the internal hardware of the ARM CPU. It is after these four steps that the result is obtained and the flags are set.

We must look at the carry flag (not the sign flag) to determine if the result is positive or negative. After the execution of SUBS, if C=1, the result is positive; if C = 0, the result is negative and the destination has the 2's complement of the result.

Show the steps involved

```
MOV     R2,#0x4F          ;R2 = 0x4F
MOV     R3,#0x39          ;R3 = 0x39
SUBS    R4,R2,R3          ;R4 = R2 – R3
```

```
  0x4F        0000004F
– 0x39      + FFFFFFC7   2's complement of 0x39
    16      1 00000016   (C = 1 step 4)
```

2's complement of Hex No

```
    FFFF  FFFF
-   0000 0039
---------------------
    FFFF  FFC6
+            1
---------------------
    FFFF  FFC7
```

The flags would be set as follows: C = 1, and Z = 0. The programmer must look at the carry flag (not the sign flag) to determine if the result is positive or negative.

Note: C=0 when there is borrow & result is negative and C=1 when there is no borrow & result is positive. In negative result, the destination has the 2's complement of the result.

Analyze the following instructions:

```
MOV R1,#0x4C ;R1 = 0x4C
MOV R2,#0x6E ;R2 = 0x6E
SUBS R0,R1,R2 ;R0 = R1 – R2
```

Solution:

Following are the steps for “SUB R0,R1,R2”:

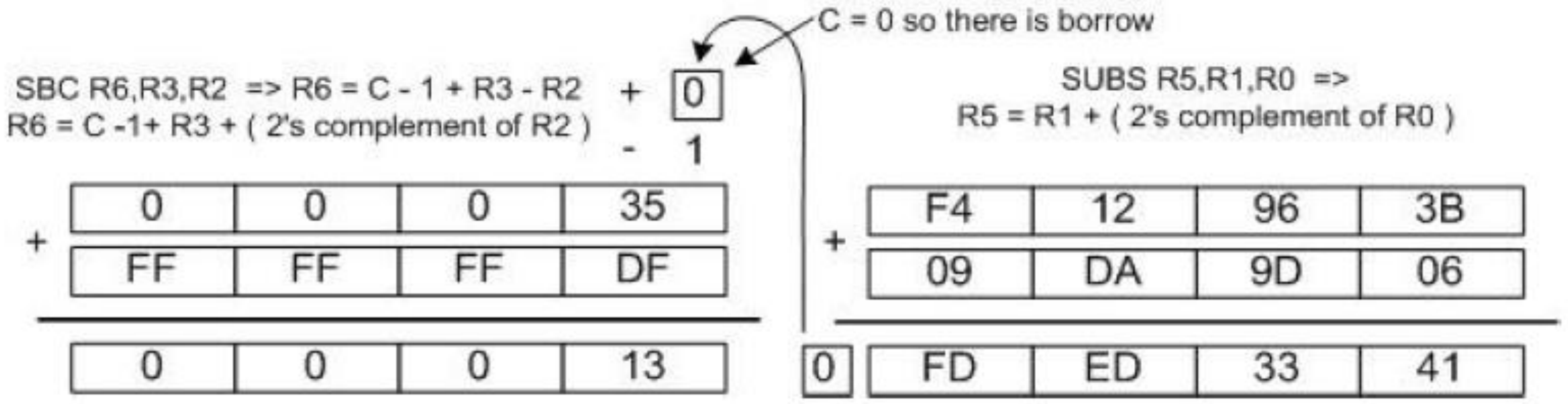
```
  4C             0000004C
–6E          + FFFFFF92 (2’s complement of 0x6E)
– 22         0 FFFFFFDE (C = 0) result is negative
```

SBC Rd,Rn,Op2 ;Rd = Rn – Op2 – 1 + C / Rn – Op2 – !C

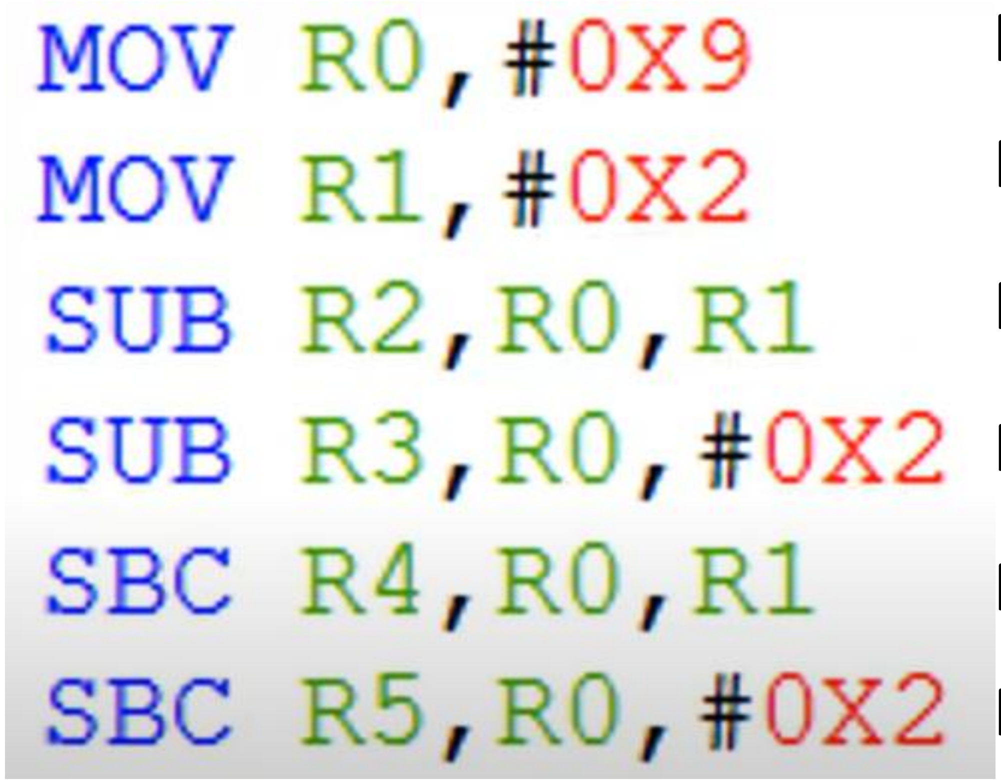
This instruction is used for subtraction of multiword (data larger than 32-bit) numbers.

In ARM the carry flag is not inverted after subtraction and carry flag is invert of borrow. To invert the carry flag while running the subtract with borrow instruction it is implementedas "Rd = Rn – Op2 – 1 + C"

```
LDR R0,=0xF6 2562FA          ;R0 = 0xF62562FA,
LDR R1,=0xF4 12963B          ;R1 = 0xF412963B
MOV R2,#0x21                 ;R2 = 0x21
MOV R3,#0x35                 ;R3 = 0x35
SUBS R5,R1,R0                ;R5 = R1 – R0
                             ; =0xF412963B – 0xF62562FA, and C = 0
SBC R6,R3,R2                 ;R6 = R3 – R2 – 1 + C
```


C = 0 so there is borrow
SBC R6,R3,R2 => R6 = C - 1 + R3 - R2
R6 = C -1+ R3 + (2's complement of R2)
+ 0
- 1
+
0 0 0 35
FF FF FF DF
0 0 0 13
SUBS R5,R1,R0 =>
R5 = R1 + (2's complement of R0)
+
F4 12 96 3B
09 DA 9D 06
0
FD ED 33 41

Show Register contents



```
MOV R0,#0X9
MOV R1,#0X2
SUB R2,R0,R1
SUB R3,R0,#0X2
SBC R4,R0,R1
SBC R5,R0,#0X2
```

R0=9

R1=2

R2= 9-2=7

R3= 9-2=7

R4= 9-2-1+0(CARRY)=6

R5= 9-2-1+0(CARRY)=6

RSB (reverse subtract)

RSB Rd,Rn,Op2   ;Rd = Op2 – Rn

This instruction can be used to get 2’s complement of a 32-bit operand

```
MOV R1,#0x6E  ;R1=0x6E
RSB R0,R1,#0     ;R0= 0 – R1
```

```
   0       0000000
–6E    + FFFFFF92 (2’s complement)
----------------------
–6E      FFFFFF92
```

```
MOV R1,#0x1  ;R1=1
RSB R0,R1,#0   ;R0= 0 – R1 = 0 – 1
```

This is one way to get a fixed value of 0xFFFFFFFF in a register.   R0=0xFFFFFFFF.

RSC (reverse subtract with carry)

RSC Rd,Rn,Op2 ;Rd = Op2 – Rn – 1 + C

This instruction can be used to get the 2's complement of the 64-bit operand.

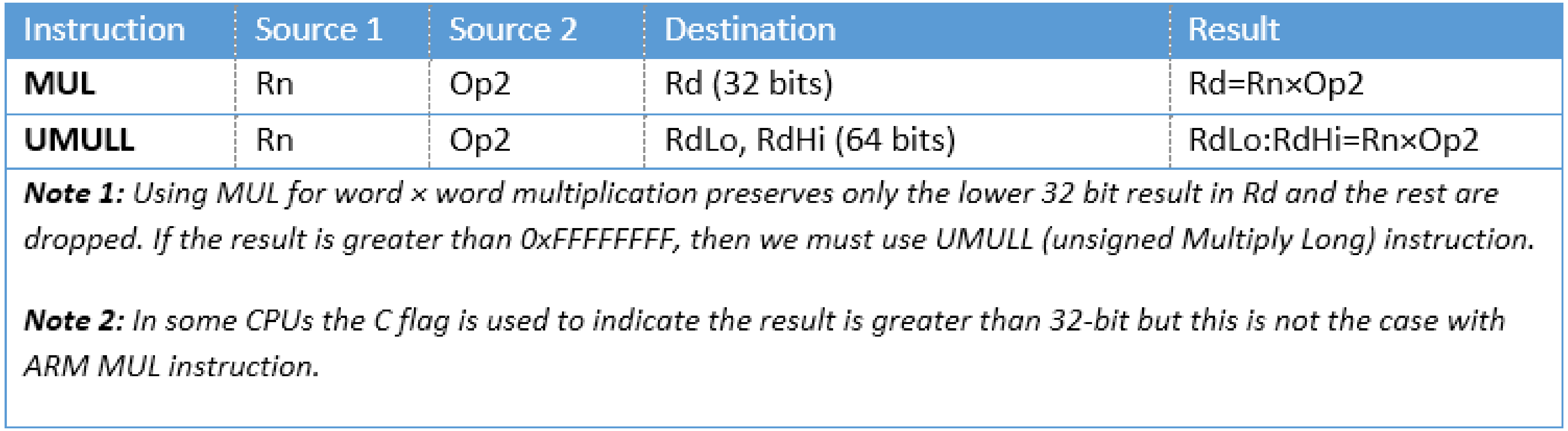
Show how to create 2's complement of a 64-bit data in R0 and R1 register. The R0 hold the lower 32-bit.

Solution:

```
LDR R0,=0xF62562FA  ;R0 = 0xF62562FA
LDR R1,=0xF812963B  ;R1 = 0xF812963B
RSBS R5,R0,#0       ;R5 = 0 – R0
                    ; = 0 – 0xF62562FA = 9DA9D06 and C = 0
RSC R6,R1,#0        ;R6 = 0 – R1 – 1 + C
                    ; = 0 – 0xF812963B – 1 + 0 = 7ED69C4
```

Multiplication of unsigned numbers in ARM

Two choices for unsigned multiplication: normal multiply and long multiply.



Instruction	Source 1	Source 2	Destination	Result
MUL	Rn	Op2	Rd (32 bits)	Rd=Rn×Op2
UMULL	Rn	Op2	RdLo, RdHi (64 bits)	RdLo:RdHi=Rn×Op2

Note 1: *Using MUL for word × word multiplication preserves only the lower 32 bit result in Rd and the rest are dropped. If the result is greater than 0xFFFFFFFF, then we must use UMULL (unsigned Multiply Long) instruction.*

Note 2: *In some CPUs the C flag is used to indicate the result is greater than 32-bit but this is not the case with ARM MUL instruction.*

```
MUL Rd,Rn,Rm                    ;Rd = Rn × Rm
```

All the operands must be in registers. Immediate value is not allowed as an operand.

Example for MULL

```
MOV R1,#0x25 ;R1=0x25
MOV R2,#0x65 ;R2=0x65
MUL R3,R1,R2 ;R3 = R1 × R2 = 0x65 × 0x25
```

- In the case of 16bit X16bit→ 32bit product→ No problem

- In the case of 32bit X32bit→ 64bit product→ Register cant hold 64 bit. Here destination register will hold the lower word (32-bit) and the portion beyond 32-bit is dropped.
- So, it is not safe to use MUL for multiplication of numbers greater than 65,536, i.e., numbers above 32 bits.

UMULL (unsigned multiply long)

```
UMULL RdLo,RdHi,Rn,Op2      ;RdHi:RdLo = Rn × Op2
```

In unsigned long multiplication, the operands must be in registers. After the multiplication, the destination registers will contain the result.

Notice that the left most register
RdLo, will hold the lower word
RdHi will hold upper word.

```
LDR R1,=0x54000000     ;R1 = 0x54000000
LDR R2,=0x10000002     ;R2 = 0x10000002
UMULL R3,R4,R2,R1      ;0x54000000 × 0x10000002
                       ; = 0x054000000A8000000
                       ;R3 = 0xA8000000, the lower 32 bits
                       ;R4 = 0x05400000, the higher 32 bits
```

Multiply and Accumulate Instruction in ARM

```
MLA Rd,Rm,Rs,Rn                ;Rd = Rm × Rs + Rn
```

In multiplication and add, the operands must be in registers. After the multiplication and add, the destination register will contain the result.

```
MOV R1,#100          ;R1 = 100
MOV R2,#5            ;R2 = 5
MOV R3,#40           ;R3 = 40
MLA R4,R1,R2,R3      ;R4 = R1 × R2 + R3 = 100 × 5 + 40 = 540
```

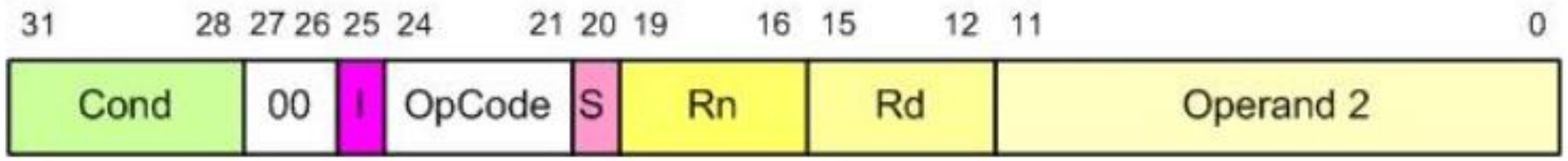
Notice that multiply and add can produce a result greater than 32-bit, if the MLA instruction is used, the destination register will hold the lower word (32-bit) and the portion beyond 32-bit is dropped.

UMLAL RdLo,RdHi,Rn,Op2 ;RdHi:RdLo = Rn × Op2 + RdHi:RdLo

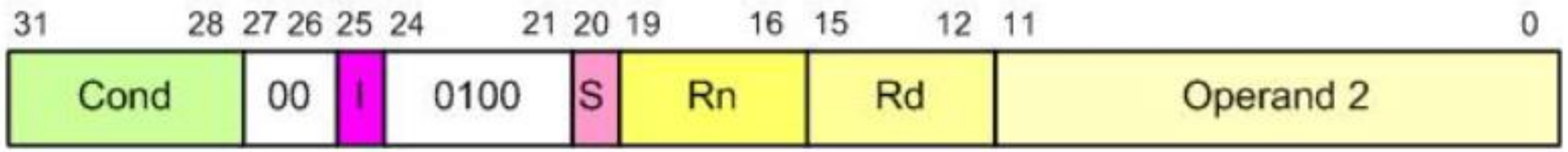
Multiplies unsigned words in Rn and Rm register, adds the 64-bit result to RdHi:RdLo registers, and saves the final result in RdHi:RdLo. The RdLo (low) and RdHi(high) are the unsigned lower word and higher word of the 64-bit value.

In multiplication and add, the operands must be in registers. Notice that the addend and the high word of the destination use the same registers, the two left most registers in the instruction. It means that the contents of the registers which have the addend will be changed after execution of UMLAL instruction.

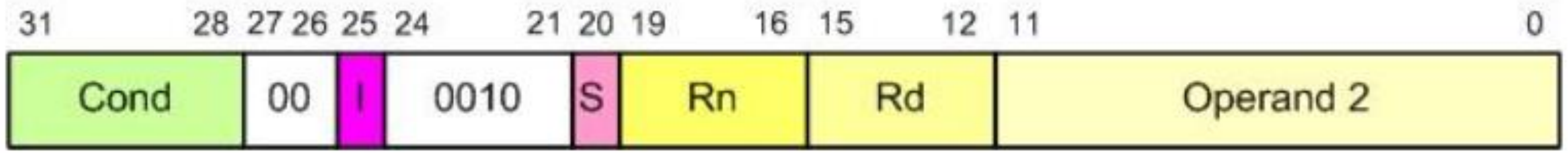
Division of unsigned numbers in ARM
It is done using repeated subtraction



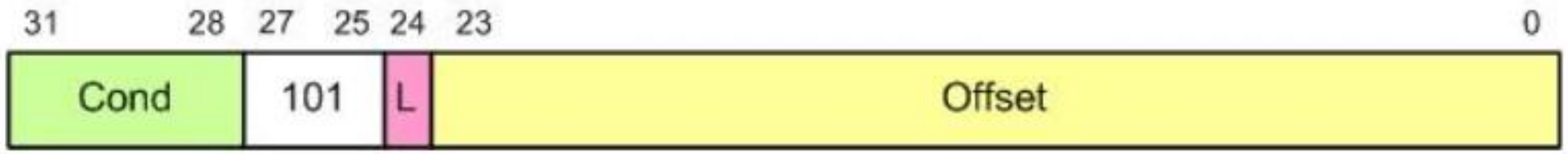
General Formation of Data Processing Instructions



ADD Instruction Formation



SUB Instruction Formation



Branch Instruction Formation

Logic Instructions

Instruction (Flags Unchanged)	Action	Instruction (Flags Changed)	Hexadecimal
AND	ANDing	**ANDS**	Anding and set flags
ORR	ORRing	**ORS**	Oring and set flags
EOR	Exclusive-ORing	**EORS**	Exclusive Oring and set flags
BIC	Bit Clearing	**BICS**	Bit clearing and set flags

```
AND Rd, Rn, Op2          ;Rd = Rn ANDed Op2
ORR Rd, Rn, Op2          ;Rd = Rn ORed Op2
EOR Rd,Rn,Op2             ;Rd = Rn Ex-ORed with Op2
BIC Rd,Rn,Op2            ;clear certain bits of Rn specified by the Op2 and place the result in Rd
```

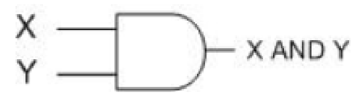
In all these instructions Op2 can be register or immediate value

AND

AND Rd, Rn, Op2 ;Rd = Rn ANDed Op2

- This instruction will perform a bitwise logical AND on the operands and place the result in the destination.
- ANDS will change the C and Z flags according to the result

```
MOV R0,#0x97
MOV R1,#0xF0
AND R2,R0,R1 ;R2= R0 ANDed with R1
```

	Inputs		Output	Symbol
	X	Y	X AND Y	
	0	0	0	
	0	1	0	
	1	0	0	
	1	1	1	

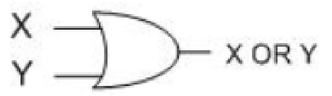
```
        0x97 1 0 0 1 0 1 1 1
AND     0xF0 1 1 1 1 0 0 0 0
        0x90 1 0 0 1 0 0 0 0
```

ORR

ORR Rd, Rn, Op2 ;Rd = Rn ORed Op2

- ORR can be used to set certain bits of an operand to one.
- ORRS change the C and Z flags according to the result

```
 MOV R1,#0x04              ;R1 = 0x04
 ORRS R2,R1,#0x68        ;R2= R1 ORed 0x68

          0x04 0000 0100
OR        0x68 0110 1000
          0x6C 0110 1100   Flag will be: Z = 0
```

Inputs		Output	Symbol
X	Y	X OR Y	
0	0	0	
0	1	1	
1	0	1	
1	1	1	

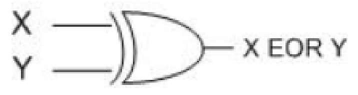
Note: The ORR instruction can also be used to test for a zero operand.
For example, “ORRS R2,R2,#0” will OR the register R2 with zero and make Z = 1 if R2 is zero.

EOR

EOR Rd,Rn,Op2 ;Rd = Rn Ex-ORed with Op2

```
MOV R1,#0x54
EOR R2,R1,#0x78   ;R2 = R1 ExOred with 0x78
```

```
        0x54 0 1 0 1 0 1 0 0
XOR     0x78 0 1 1 1 1 0 0 0
        0x2C 0 0 1 0 1 1 0 0
```

Inputs		Output	Symbol
X	Y	X EOR Y	
0	0	0	
0	1	1	
1	0	1	
1	1	0	

- The EOR instruction can be used to clear the contents of a register by Ex-ORing it with itself.

```
    0x45       0 1 0 0 0 1 0 1
XOR 0x45       0 1 0 0 0 1 0 1
    0x00       0 0 0 0 0 0 0 0
```

- EOR can also be used to see if two registers have the same value.
    - “EORS R2,R3,R4” will make Z = 1 if both registers R4 and R3 have the same value, and if they do, the result (00000000) is saved in R2, the destination.

- EOR is widely used to toggle bits of an operand.

```
EOR R2,R2,#0x04 ;EOR R2 with 0000 0100
```

Here the bit 2 of R2 to change to the opposite value whereas other bits remain same.

BIC (bit clear)

BIC Rd,Rn,Op2   ;clear certain bits of Rn specified by Op2 & place the result in Rd

- The bits that are HIGH in Op2 will be cleared and bits with LOW will be left unchanged.
- In reality, the BIC instruction performs AND operation on Rn register with the complement of Op2 and places the result in destination register

```
MOV R1,#0x0F
                               complement
MOV R2,#0xAA
BIC R3,R2,R1            ;now R3 = 0xAA ANDed with 0xF0 = 0xA0
```

Inputs		Output
X	Y	X AND (NOT Y)
0	0	0
0	1	0
1	0	1
1	1	0

MVN (move negative)

MVN Rd, Rn ;move negative of Rn to Rd

- Used to generate one’s complement of an operand.
  - “MVN R2,#0” will make R2=0xFFFFFFFF.
  - LDR R2,=0xAAAAAAAA ;R2 = 0xAAAAAAAA
    MVN R2,R2 ;R2 = 0x55555555

It must be noted that the instruction “MVN Rd,#0” is widely used to load the fixed value of 0xFFFFFFFF into destination register.

Comparison of unsigned numbers

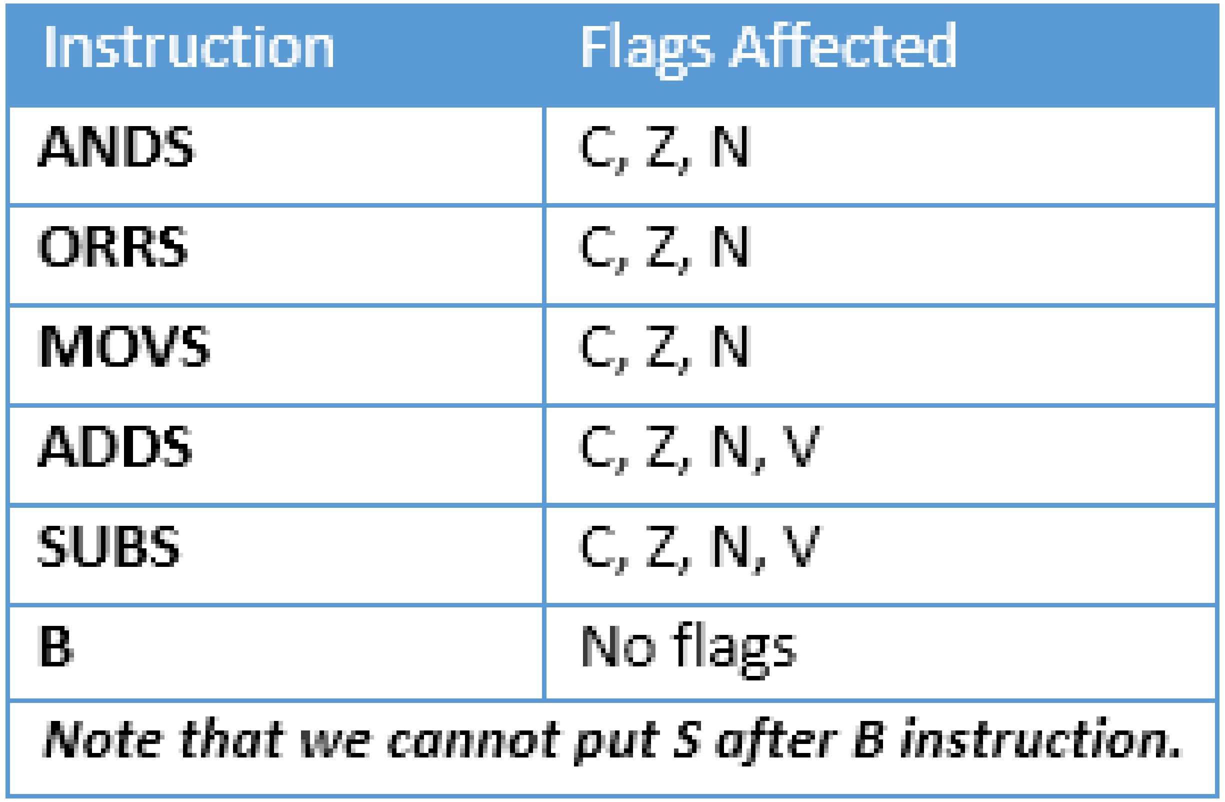
CMP Rn,Op2 ;compare Rn with Op2 and set the flags

- The CMP instruction compares two operands and changes the flags according to the result of the comparison. The operands themselves remain unchanged. The second source operands can be a register or an immediate value not larger than 0xFF.
- It must be emphasized that "CMP Rn,Op2" instruction is really a subtract operation. Op2 is subtracted from Rn (Rn – Op2) and the result is discarded and flags are set accordingly.
- Although all the C, S, Z, and V flags reflect the result of the comparison, only C and Z are used for unsigned numbers, as shown below:

Instruction	C	Z
Rn > Op2	1	0
Rn = Op2	1	1
Rn < Op2	0	0

```
        LDR R1,=0x35F          ;R1 = 0x35F
        LDR R2,=0xCCC          ;R2 = 0xCCC
        CMP R1,R2              ;compare 0x35F with 0xCCC
        BCC OVER               ;branch if C = 0
        MOV R1,#0              ;if C = 1, then clear R1
OVER    ADD R2,R2,#1           ;R2 = R2 + 1 = 0xCCC + 1 = 0xCCD
```

In the above program, R1 is less than the R2 (0x35F < 0xCCC); therefore, C = 0 and BCC (branch if carry clear) will go to target OVER.



Instruction	Flags Affected
ANDS	C, Z, N
ORRS	C, Z, N
MOVS	C, Z, N
ADDS	C, Z, N, V
SUBS	C, Z, N, V
B	No flags
Note that we cannot put S after B instruction.	

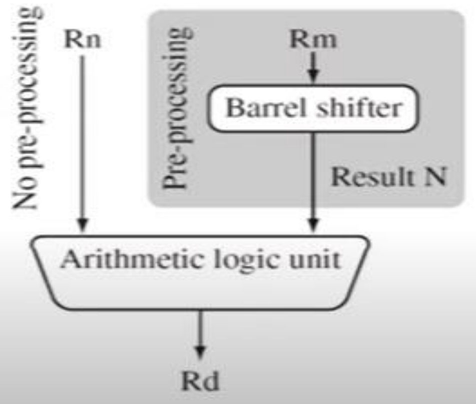
Flag Bits Affected by Different Instructions

Rotate and Barrel Shifter

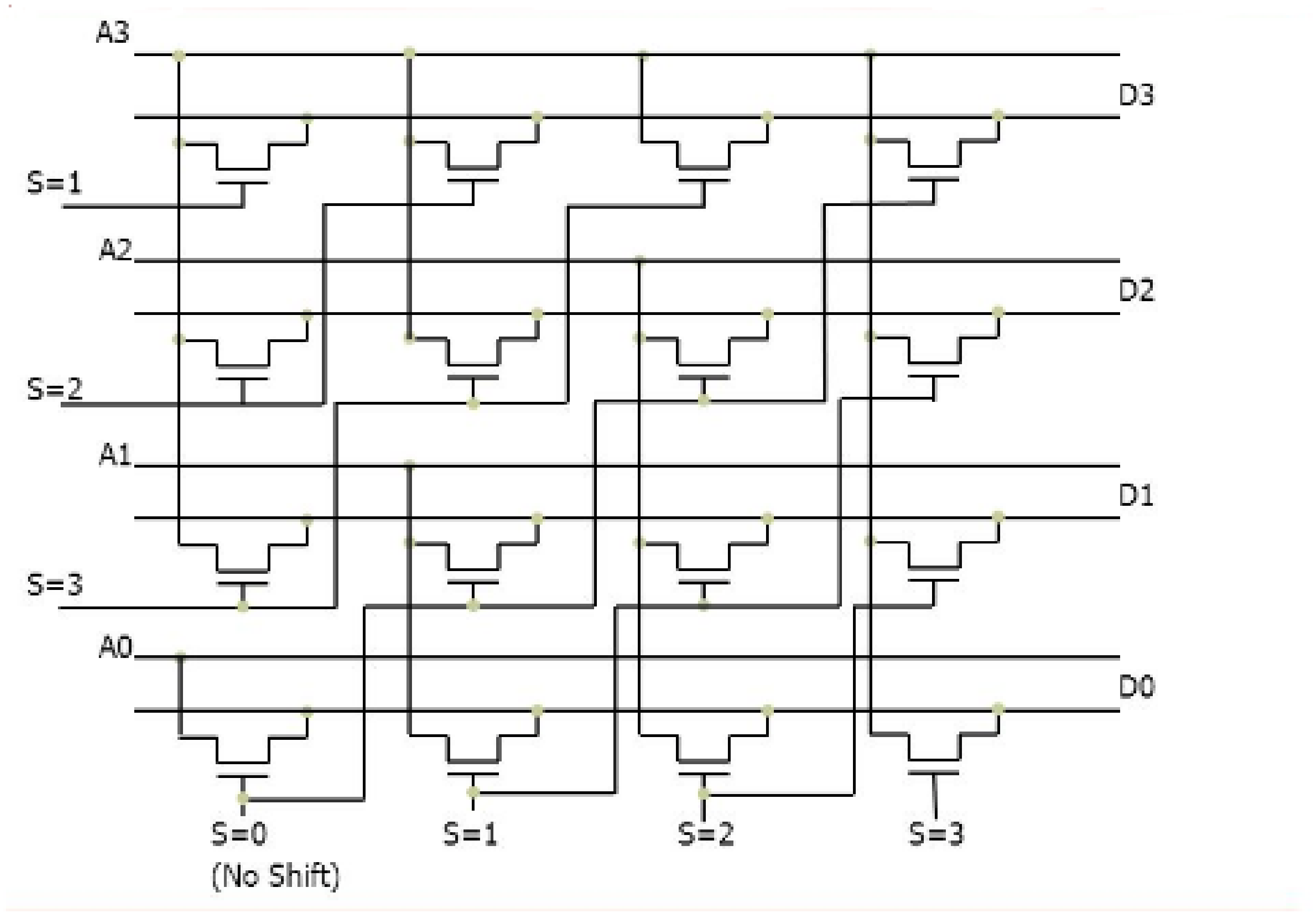
we can perform the shift and rotate operations as part of other instructions such as MOV

The process instructions can be used in one of the following forms:
1. opcode Rd, Rn, Rs (e.g. ADD R1,R2,R3)
2. opcode Rd, Rn, immediate Value (e.g. ADD R2,R3,#5)
ARM is able to shift or rotate the second argument before using it as the argument.

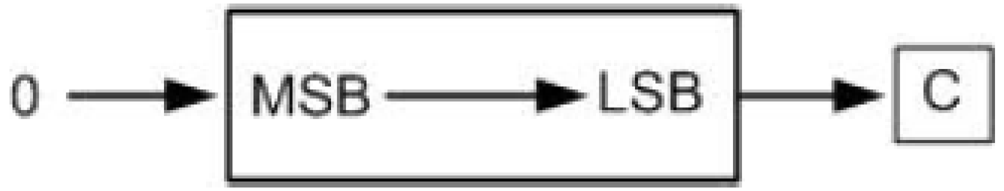
- The ARM arithmetic logic unit has a 32-bit barrel shifter that is capable of shift and rotate operations.
- <operation>{cond}{S} Rd, Rn, N

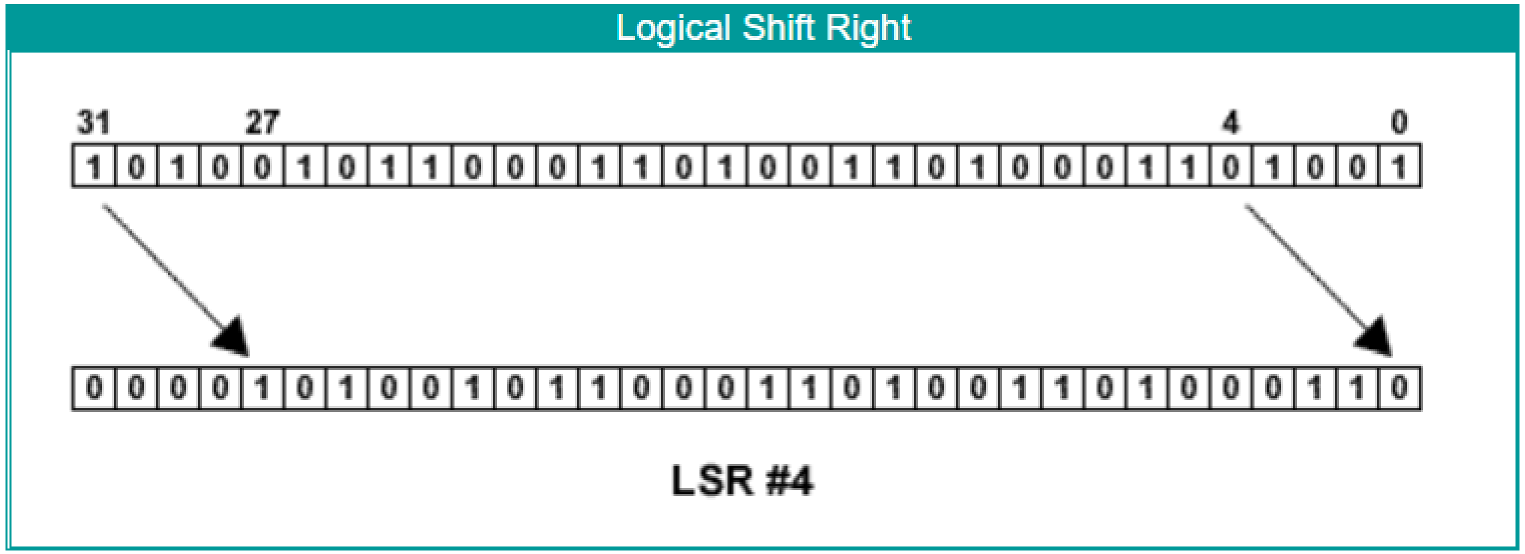


MOS Barrel Shifter



LSR Logical Shift Right



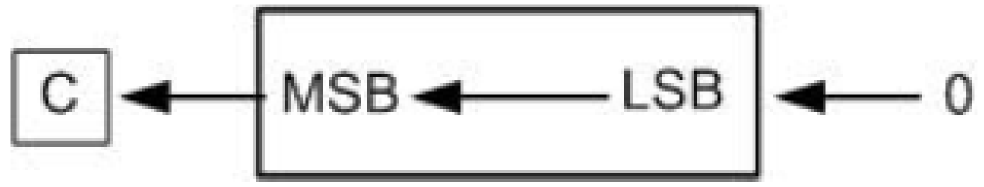


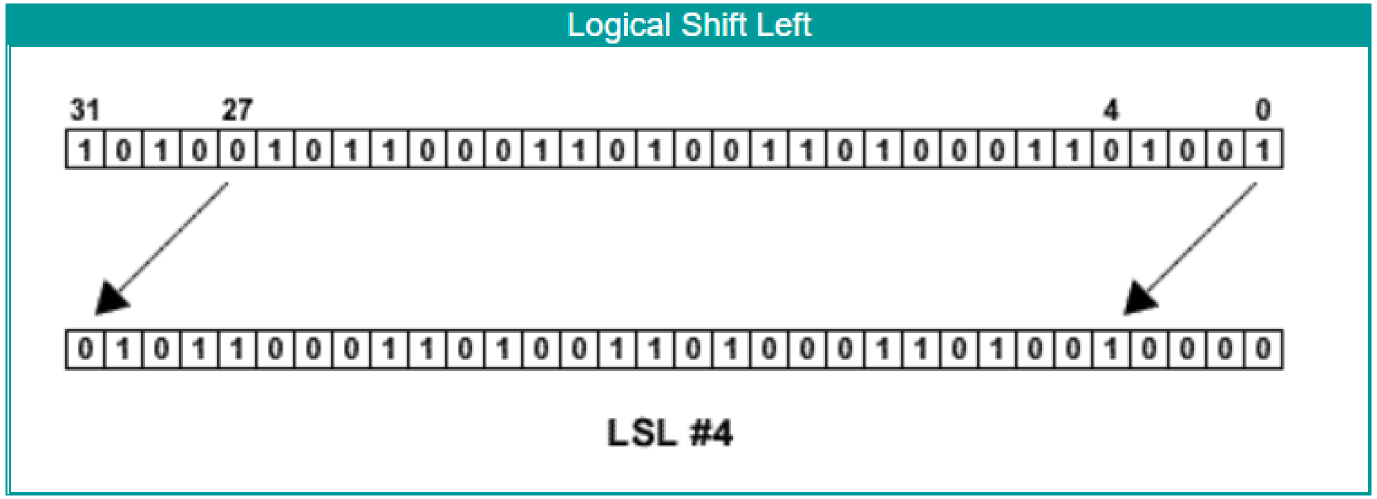
```
MOV R0,#0x9A
MOV R2,#0x03
MOV R1,R0,LSR R2      ;shift R0 to right R2 times and move the result to R1
```

```
0x9A =  00000000 00000000 0000000 00000000 10011010
first shift:        00000000 00000000 0000000 00000000 01001101  C = 0
second shift:     00000000 00000000 0000000 00000000 00100110  C = 1
third shift:       00000000 00000000 0000000 00000000 00010011  C = 0
```

After shifting right three times, R1 = 0x00000013 and C = 0.

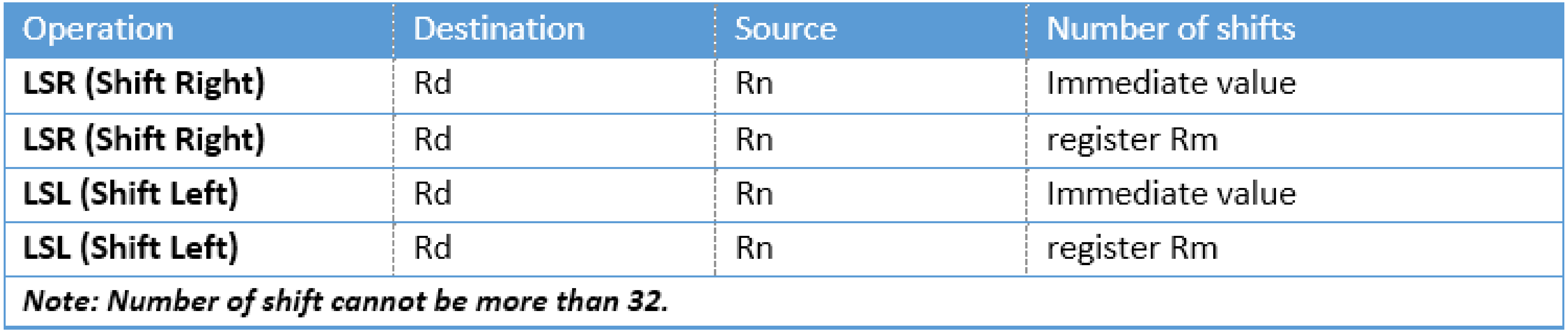
LSL Logical Shift Left





```
LDR R1,=0x0F000006
MOV R0,#0x08
MOV R2,R1,LSL R0
```

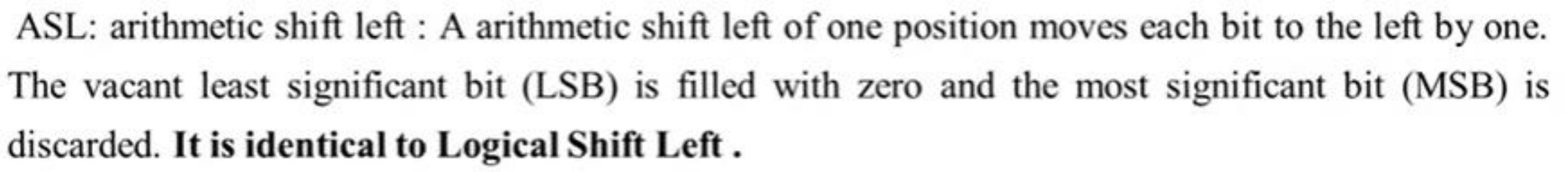
```
         00001111 00000000 00000000 00000110
C=0      00011110 00000000 00000000 00001100 (shifted left once)
C=0      00111100 00000000 00000000 00011000
C=0      01111000 00000000 00000000 00110000
C=0      11110000 00000000 00000000 01100000
C=1      11100000 00000000 00000000 11000000
C=1      11000000 00000000 00000001 10000000
C=1      10000000 00000000 00000011 00000000
C=1      00000000 00000000 00000110 00000000 (shifted eight times)
```

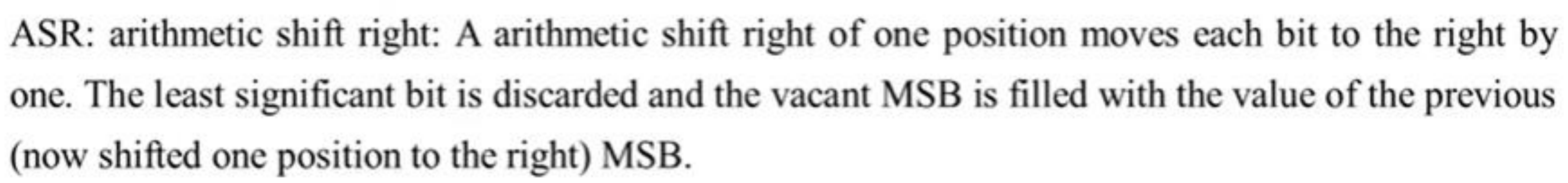


Operation	Destination	Source	Number of shifts
LSR (Shift Right)	Rd	Rn	Immediate value
LSR (Shift Right)	Rd	Rn	register Rm
LSL (Shift Left)	Rd	Rn	Immediate value
LSL (Shift Left)	Rd	Rn	register Rm
Note: Number of shift cannot be more than 32.			

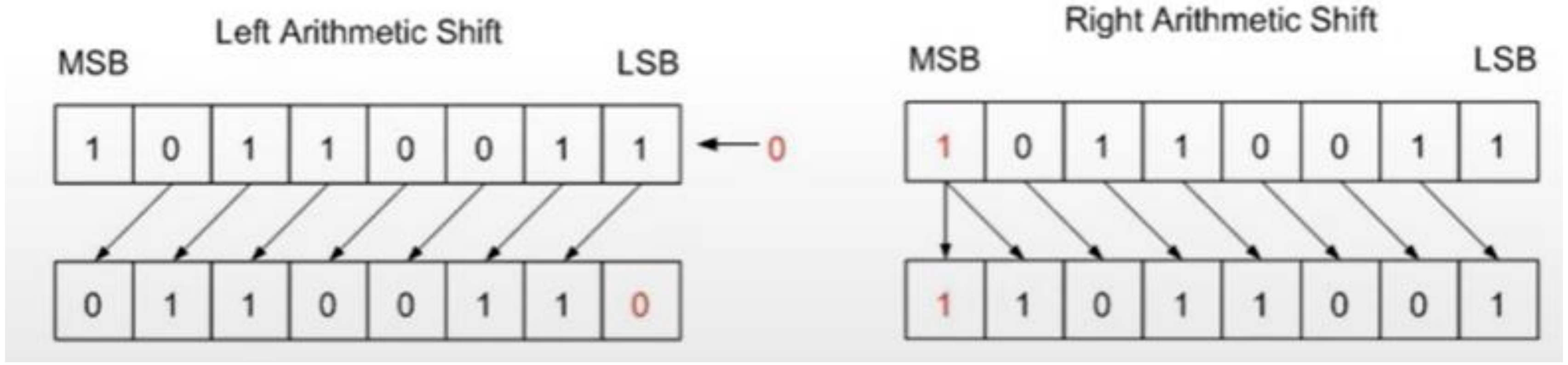
Logic Shift operations for unsigned numbers in ARM

ASR (arithmetic shift right) and ASL (arithmetic shift left)

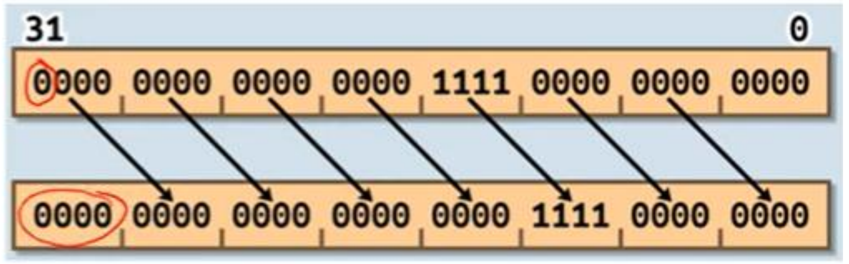

ASL: arithmetic shift left : A arithmetic shift left of one position moves each bit to the left by one. The vacant least significant bit (LSB) is filled with zero and the most significant bit (MSB) is discarded. **It is identical to Logical Shift Left .**


ASR: arithmetic shift right: A arithmetic shift right of one position moves each bit to the right by one. The least significant bit is discarded and the vacant MSB is filled with the value of the previous (now shifted one position to the right) MSB.

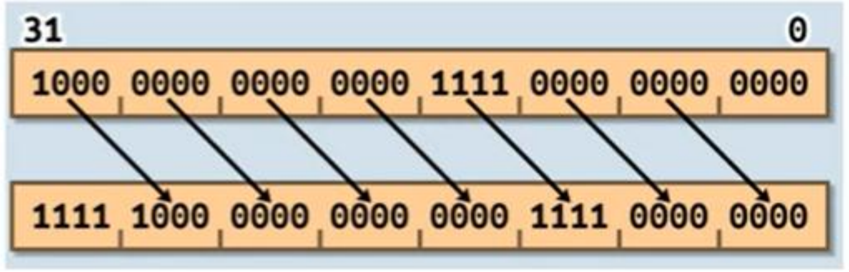
```
 MOV Rn,Op2, ASR count

MOV R0,#-10          ;R0 = -10 = 0xFFFFFFF6
MOV R3,R0,ASR #1     ;R0 is arithmetic shifted right once
                              ;R3 = 0xFFFFFFFB = -5
```



- Arithmetic Shift Right by 4, positive value.



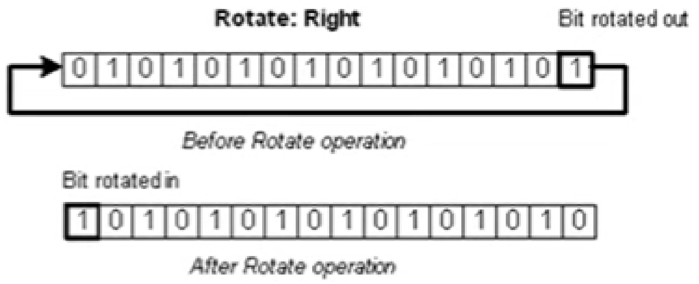
- Arithmetic Shift Right by 4, negative value.

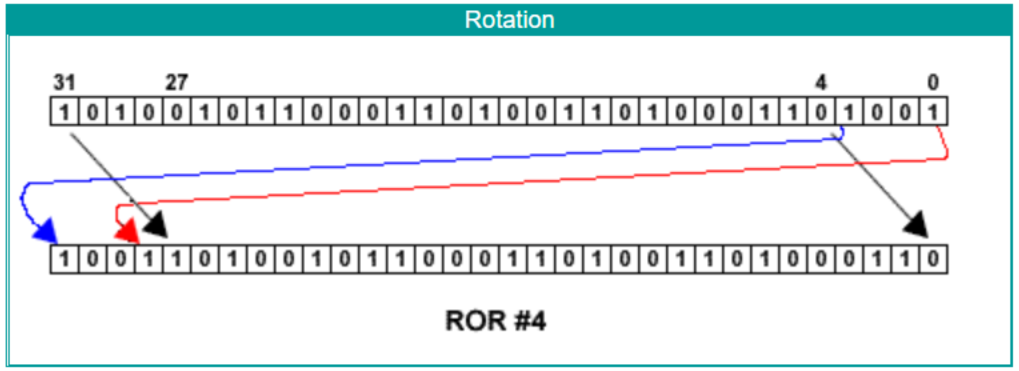


ROR (rotate right)

The Rotate function moves the bits in an integer to the left or to the right.

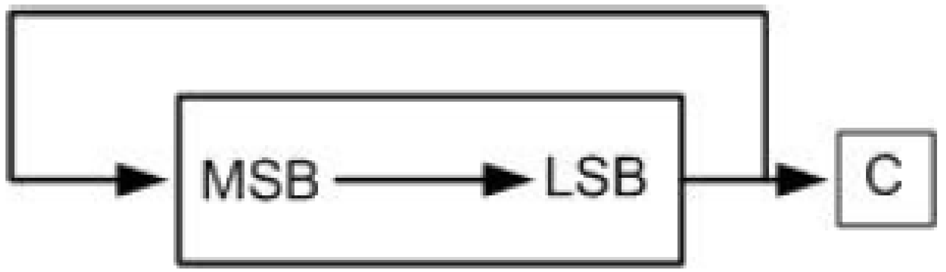
ROR: rotate right





```
MOV R1,#0x36
MOVS R1,R1,ROR #1
```

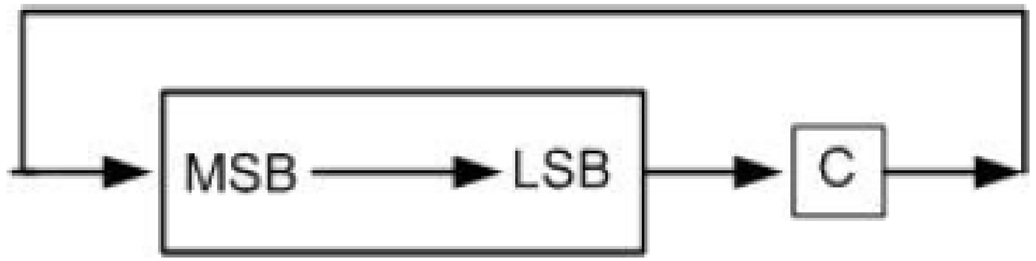
```
MOV R1,#0x36
MOV R0,#3
MOVS R1,R1,ROR R0
```



Rotate left

There is no rotate left option in ARM since one can use the rotate right (ROR) to do the job. That means instead of rotating left n bits we can use rotate right 32–n bits to do the job of rotate left. Using this method does not give us the proper carry if actual instruction of ROL was available

RRX rotate right through carry



In RRX the LSB is moved to C and C is moved to the MSB. In reality, C flag acts as if it is part of the operand. That means the RRX is like 33-bit register since the C flag is 33rd bit. The RRX takes no arguments and the number of times an operand to be rotated is fixed at one.

```
;assume C=0
MOV R2,#0x26
;R2 = 0000 0000 0000 0000 0000 0000 0010 0110
MOVS R2,R2,RRX
;R2 = 0000 0000 0000 0000 0000 0000 0001 0011 C=0
```

```
RRX Rd,Rm                 ;Rd=rotate Rm right 1 bit position
```

Function: Each bit of Rm register is shifted from left to right one bit. The RRX does not update the flags.

```
LDR R2,=0x00000002
RRX R0,R2                        ;R0=R2 is shifted right one bit
                                 ;now, R0=0x00000001
```

RRXS Rotate Right with extend (update the flags)

```
RRXS Rd,Rm                ;Rd=rotate Rm right 1 bit position
```

Function: Each bit of Rm register is shifted from left to right one bit. The RRXS updates the flags.

```
LDR R2,=0x00000002
RRXS R0,R2                ;R0=R2 is shifted right one bit
                                 ;now, R0=0x00000001
```

Operation	Destination	Source	Number of Rotates
ROR (Rotate Right)	Rd	Rn	Immediate value
ROR (Rotate Right)	Rd	Rn	register Rm
RRX (Rotate Right Through Carry)	Rd	Rn	1 bit

Rotate operations for unsigned numbers in ARM

Looping and Branch Instructions

Repeating a sequence of instructions or an operation a certain number of times is called a *loop*.

Unconditional Branching Instructions

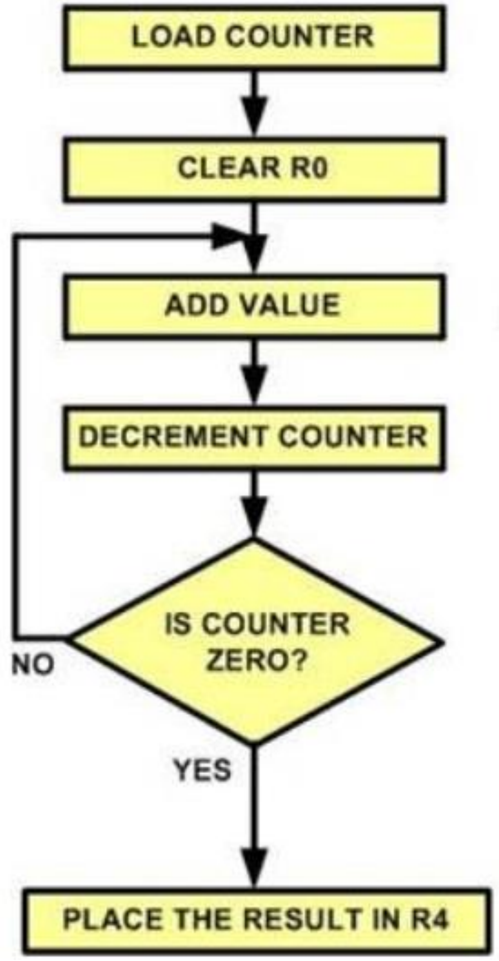
The conditional branch is a jump in which control is transferred to the target location based upon certain conditions

BNE (branch if not equal)

```
BACK    .........             ;start of the loop
        .........             ;body of the loop
        .........             ;body of the loop
        SUBS Rn,Rn,#1 ;Rn = Rn - 1, set the flag Z = 1 if Rn = 0
        BNE BACK              ;branch if Z = 0
```

Write a program to
(a) clear R0, (b) add 9 to R0 a thousand times, then (c) place the sum in R4.
Use the zero flag and BNE instruction.

```
       LDR R2,=1000          ;R2 = 1000 (decimal) for counter
       MOV R0,#0             ;R0 = 0 (sum)
AGAIN  ADD R0,R0,#9          ;R0 = R0 + 9 (add 09 to R1, R1 = sum)
       SUBS R2,R2,#1         ;R2 = R2 - 1 and set the flags. Decrement counter
       BNE AGAIN             ;repeat until COUNT = 0
       MOV R4,R0             ;store the sum in R4
```



BEQ (branch if equal); (Branch if Z = 1)

```
          LDR R2,=1000              ;R2 = 1000 (decimal) for counter
          MOV R0,#0                 ;R0 = 0 (sum)
AGAIN    CMP R2,#0                  ;R2-0 and update Z flag.
          BEQ STOP                  ; If Z=0 then jump to STOP
          ADD R0,R0,#9              ;R0 = R0 + 9 (add 09 to R1, R1 = sum)
          SUB R2,R2,#1              ;R2 = R2 - 1 and set the flags. Decrement counter
          B AGAIN                   ;Branch to AGAIN
STOP      MOV R4,R0                 ;store the sum in R4
```

ARM Conditional Branch Instructions for Unsigned Data

Instruction		Action
BCS/BHS	branch if carry set/branch if higher or same	Branch if C = 1
BCC/BLO	branch if carry clear/branch lower	Branch if C = 0
BEQ	branch if equal	Branch if Z = 1
BNE	branch if not equal	Branch if Z = 0
BLS	branch if less or same	Branch if Z = 1 or C = 0
BHI	branch if higher	Branch if Z = 0 and C = 1

Note: Generally Branch instruction follows either CMP or SUBS instruction

Comparison of unsigned numbers

CMP Rn,Op2 ;compare Rn with Op2 and set the flags

- The CMP instruction compares two operands and changes the flags according to the result of the comparison. The operands themselves remain unchanged. The second source operands can be a register or an immediate value not larger than 0xFF.
- It must be emphasized that "CMP Rn,Op2" instruction is really a subtract operation. Op2 is subtracted from Rn (Rn – Op2) and the result is discarded and flags are set accordingly.
- Although all the C, S, Z, and V flags reflect the result of the comparison, only C and Z are used for unsigned numbers, as shown below:

Instruction	C	Z
Rn > Op2	1	0
Rn = Op2	1	1
Rn < Op2	0	0

CMP Rn,Op2 ;compare Rn with Op2 and set the flags

Instruction		Action
BCS/BHS	branch if carry set/branch if higher or same	Branch if Rn ≥ Op2
BCC/BLO	branch if carry clear/branch lower	Branch if Rn < Op2
BEQ	branch if equal	Branch if Rn = Op2
BNE	branch if not equal	Branch if Rn ≠ Op2
BLS	branch if less or same	Branch if Rn ≤ Op2
BHI	branch if higher	Branch if Rn > Op2

ARM Conditional Branch Instructions for Unsigned Data

Although BCS (branch carry set) and BCC (branch carry clear) check the carry flag and can be used after a compare instruction, it is recommended that BHS (branch higher or same) and BLO (branch below) be used as it is easier to understand.

Assume that there is a class of five people with the following grades: 69, 87, 96, 45, and 75. Find the highest grade.

```
            COUNT RN R0                 ;COUNT is the new name of R0
            MAX RN R1                   ;MAX is the new name of R1
            POINTER RN R2               ;POINTER is the new name of R2
            NEXT RN R3                  ;NEXT is the new name of R3
MYDATA DCD 69,87,96,45,75
ENTRY
            MOV COUNT,#5                ;COUNT = 5
            MOV MAX,#0                  ;MAX = 0
            LDR POINTER,=MYDATA         ;POINTER = MYDATA (first data )

AGAIN LDR NEXT,[POINTER]                ;load contents of POINTER to NEXT
            CMP MAX,NEXT                ;compare MAX and NEXT
            BHS CTNU                    ;if MAX > NEXT branch to CTNU
            MOV MAX,NEXT                ;MAX = NEXT
CTNU        ADD POINTER,POINTER,#4      ;Increment the address of the POINTER
            SUBS COUNT,COUNT,#1          ;decrement counter
            BNE AGAIN                   ;branch AGAIN if counter is not zero
            END
```

TST (Test)

```
TST Rn,Op2              ;Rn AND with Op2 and flag bits are updated
```

The TST instruction is used to test the contents of register to see if any bit is set to HIGH.
After the operands are ANDed together the flags are updated.
After the TST instruction if result is zero, then Z flag is raised and one can use BEQ (branch equal) to make decision.
example:

```
          MOV R0,#0x04        ;R0=00000100 in binary
          LDR R1,=myport      ;port address
OVER      LDRB R2,[R1]        ;load R2 from myport
          TST R2,R0           ;is bit 2 HIGH?
          BEQ OVER            ;keep checking
```

In TST, the Op2 can be an immediate value of less than 0xFF.
example:

```
        LDR R1,=myport     ;port address
OVER    LDRB R2,[R1]       ;load R2 from myport
        TST R2,#0x04       ;is bit 2 HIGH?
        BEQ OVER           ;keep checking
```

TEQ (test equal)

- TEQ Rn,Op2   ;Rn EX-ORed with Op2 and flag bits are set
- The TEQ instruction is used to test to see if the contents of two registers are equal.
- After the source operands are Ex-ORed together the flag bits are set according to the result.
- After the TEQ instruction if result is 0, then Z flag is raised and one can use BEQ (branch zero) to make decision.

```
        TEMP EQU 100
        MOV R0,#TEMP            ;R0 = Temp
        LDR R1,=myport          ;port address
OVER    LDRB R2,[R1]            ;load R2 from myport
        TEQ R2,R0               ;is it 100?
        BNE OVER                ;keep checking
```

Branch instructions

B label ; branch to label Always

BEQ label ; branch if Z == 1 Equal

BNE label ; branch if Z == 0 Not equal

BCS label ; branch if C == 1 Higher or same, unsigned ≥

BHS label ; branch if C == 1 Higher or same, unsigned ≥

BCC label ; branch if C == 0 Lower, unsigned <

BLO label ; branch if C == 0 Lower, unsigned <

BMI label ; branch if N == 1 Negative

BPL label ; branch if N == 0 Positive or zero

BVS label ; branch if V == 1 Overflow

BVC label ; branch if V == 0 No overflow

BHI label ; branch if C==1 and Z==0 Higher, unsigned >

BLS label ; branch if C==0 or Z==1 Lower or same, unsigned ≤

BGE label ; branch if N == V Greater than or equal, signed ≥

BLT label ; branch if N != V Less than, signed <

BGT label ; branch if Z==0 and N==V Greater than, signed >

BLE label ; branch if Z==1 or N!=V Less than or equal, signed ≤

BX Rm ; branch indirect to location specified by Rm

BL label ; branch to subroutine at label

BLX Rm ; branch to subroutine indirect specified by Rm

Unconditional branch (jump) instruction

The unconditional branch is a jump in which control is transferred unconditionally to the target location.

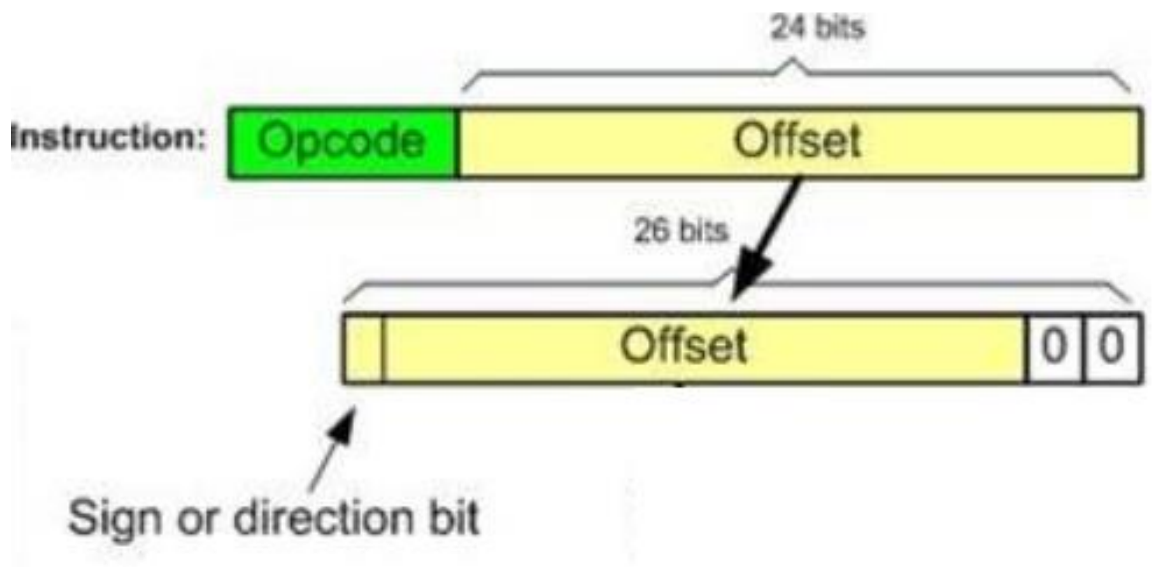
B (Branch)

B (branch) is an unconditional jump that can go to any memory location in the 32M byte address space of the ARM. Another syntax for B instruction is BAL (branch always).

```
      CMP R1,R2
      BHS L1
      MOV R3,#2
      B OVER
L1    MOV R3,#5
      OVER
```

All branches (conditional and unconditional) are short branches (jumps)

- Target must be within 32M bytes of the program counter (PC).
- That means the short jumps cannot cover the entire address space of 4G bytes (0x00000000 to 0xFFFFFFFF).

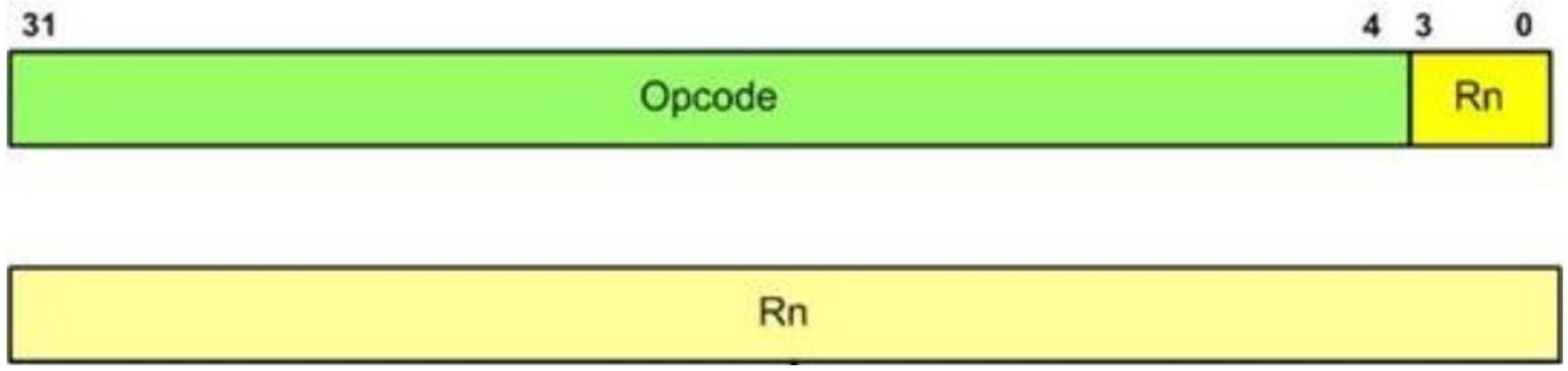


The reason for 32MB is the fact that 24 bits are shifted left twice by the ARM CPU automatically. That gives us 26 bits. Now, since one bit is used for +ve or -ve sign, we have only 25 bits. So 2^{25} = 32M in each direction. That is –32MB if it is backward and +32MB if it is forward jump.

Branching beyond 32M byte limit

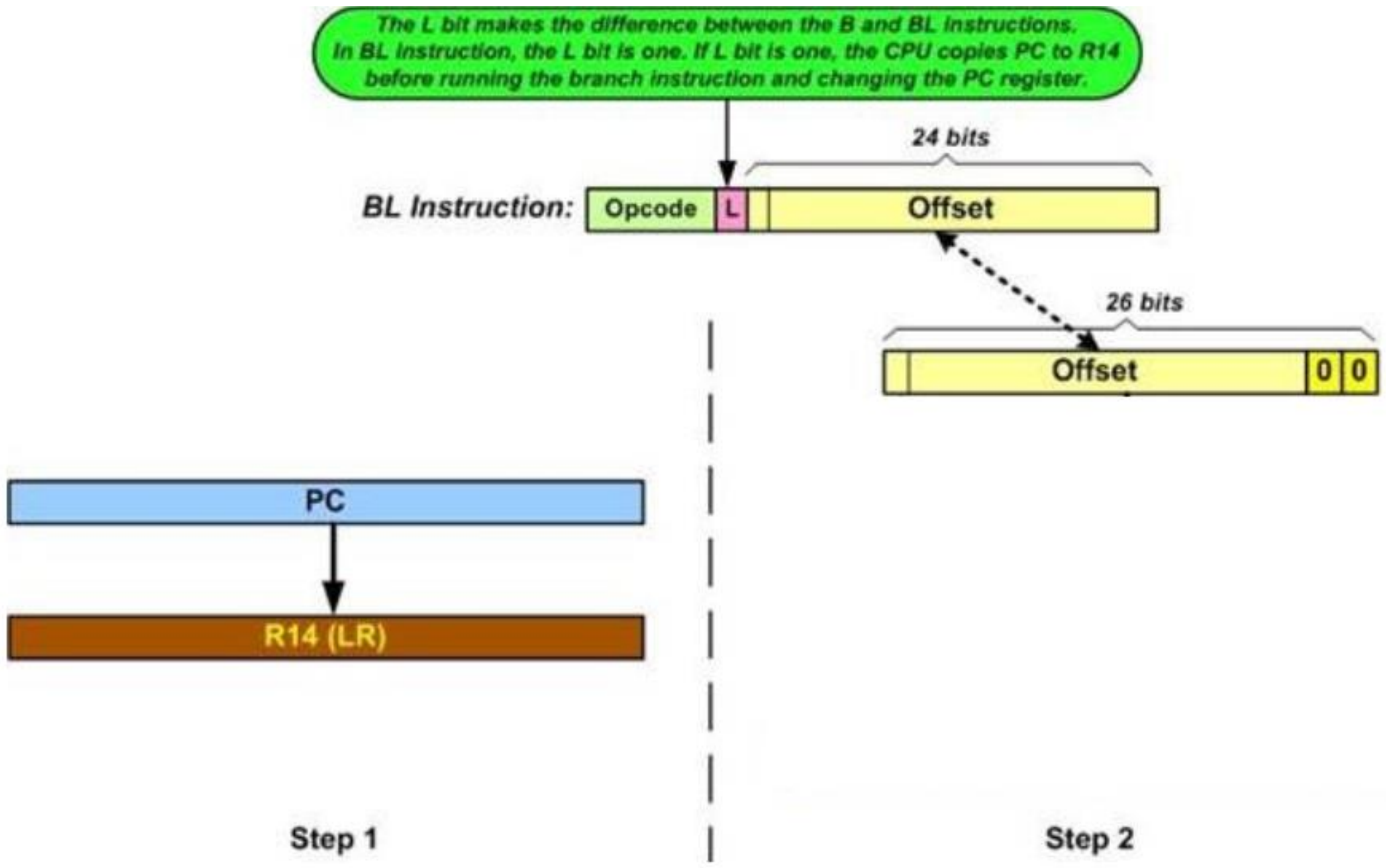
BX (branch and exchange)

The “BX Rn” instruction uses register Rn to hold target address.
Since Rn can be any of the R0–R14 registers and they are 32-bit registers, the “BX Rn” instruction can land anywhere in the 4G bytes address space of the ARM.



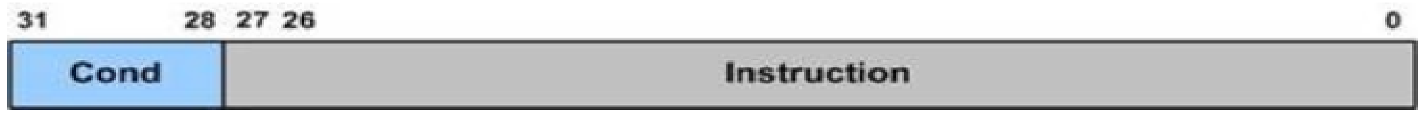
Calling Subroutine with BL   BL (branch and link)

- Subroutines are often used to perform tasks that need to be performed frequently.
- In the ARM there is only one instruction for call and that is BL (branch and link).
- BL is 32-bit instruction, where 1st 8-bits are opcode and next 24 bits are used for the address of the target subroutine.
- Therefore, BL can be used to call subroutines located anywhere within the 32M address space of the ARM
- When a subroutine is called by the BL, control is transferred to that subroutine, and the processor saves the PC in the R14 register and begins to fetch instructions from the new location.
- After finishing execution of the subroutine, we must use "BX LR" instruction to transfer control back to the caller. Every subroutine needs "BX LR" as the last instruction for return address.


The L bit makes the difference between the B and BL instructions.
In BL instruction, the L bit is one. If L bit is one, the CPU copies PC to R14
before running the branch instruction and changing the PC register.
24 bits
BL Instruction:
Opcode
L
Offset
26 bits
Offset
0
0
PC
R14 (LR)
Step 1
Step 2

Conditional Execution

- In ARM, not only branch instructions, but all instruction can be to run or ignored depending on the status of flag bits.
- In other words, not only the branch instruction but all of the ARM instructions can be conditional.
- WKT ADD, SUB, and other arithmetic instruction do not affect the flag bits in CPSR by default. But if we add S then flag bits in CPSR are updated
- The same thing is true about conditional field of each instruction.
- If we do not add a condition after an instruction, it will be executed unconditionally because the default is not to check the flags and execute unconditionally.
- If we want an instruction to be executed only when a condition is met, we put the condition syntax right after the instruction.
- To do that, the ARM instructions have set aside the most significant 4 bits of the instruction field for the conditions.



Bits	Mnemonic Extension	Meaning	Flag
0000	EQ	Equal	Z = 1
0001	NE	Not equal	Z = 0
0010	CS/HS	Carry Set/Higher or Same	C = 1
0011	CC/LO	Carry Clear/Lower	C = 0
0100	MI	Minus/Negative	N = 1
0101	PL	Plus	N = 0
0110	VS	V Set (Overflow)	V = 1
0111	VC	V Clear (No Overflow)	V = 0
1000	HI	Higher	C = 1 and Z = 0
1001	HS	Lower or Same	C = 1 and Z = 1
1010	GE	Greater than or Equal	N = V
1011	LT	Less than	N ≠ V
1100	GT	Greater than	Z = 0 and N = V
1101	LE	Less than or Equal	Z = 0 or N ≠ V
1110	AL	Always (unconditional)	
1111	–	Not Valid	

EQ Z set equal
NE Z clear not equal
CS/HS C set unsigned higher or same
CC/LO C clear unsigned lower
MI N set negative
PL N clear positive or zero
VS V set overflow
VC V clear no overflow
HI C set and Z clear unsigned higher
LS C clear or Z set unsigned lower or same
GE N equals V signed greater or equal
LT N not equal to V signed less than
GT Z clear AND (N equals V) signed greater than
LE Z set OR (N not equal to V) signed less than or equal
AL (ignored) always (usually omitted)

```
MOV R1,#10      ;R1 = 10
MOV R2,#12      ;R2 = 12
CMP R2,R1       ;compare 12 with 10, Z = 0 because they are not equal
MOVEQ R4,#20 ;this line is not executed because the condition EQ is not met
```

The following code adds 10 to R1 if it is not zero:

```
CMP R1,#0                  ;compare R1 with 0
ADDNE R1,R1,#10           ;this line is executed if Z = 0
                          ;(if in the last CMP operands were not equal)
```

Note that we can add both S and condition to syntax of an instruction. It is common to put S after the condition. See the following examples:

```
ADDNES R1,R1,#10          ;this line is executed and set the flags if Z = 0
```

Without conditional execution

```
MOV R1,#0              ;clear high word (R1 = 0)
MOV R0,#0              ;clear low word (R0 = 0)
LDR R2,=0x99999999     ;R2 = 0x99999999
MOV R3,#10             ;counter
L1 ADDS R0,R0,R2       ;R0 = R0 + R2 and update the flags
BCC NEXT               ;if C = 0, go to next number
ADD R1,R1,#1           ;if C = 1, increment the upper word
NEXT SUBS R3,R3,#1     ;R3 = R3 - 1 and update the flags
BNE L1                 ;next round if z = 0
```

With conditional execution

```
MOV R1,#0              ;clear high word (R1 = 0)
MOV R0,#0              ;clear low word (R0 = 0)
LDR R2,=0x99999999     ;R2 = 0x99999999
MOV R3,#10             ;counter
L1 ADDS R0,R0,R2       ;R0 = R0 + R2 and update the flags
ADDCS R1,R1,#1         ;if C set (C = 1),increment the upper word
NEXT SUBS R3,R3,#1     ;R3 = R3 - 1 and update the flags
BNE L1                 ;next round if z = 0
```

```
 i = 10;
  if(i>5)
    i = 7;
  else
    i = 0;
```

```
MOV R0, #10
CMP R0, #5
MOVGT R0, #7
MOVLE R0, #0
```

If bit0 & bit 31 are both 1, set R1 to 1, else set R1 to 0

```
MOV32   R0, 0xDEADBEEF
MOV     R1, #1
TST     R0, #0x80000001
MOVEQ   R1, #0
```

```
 int a = 2;
 int b = 3;
  if(a < b)
 {
   max = b;
  }
  else
{
   max = a;
  }
```

```
MAX RN r0
A RN r1
B RN r2
mov     A, #2     ;setting up initial variable a
mov     B, #3     ; setting up initial variable b
cmp     A, B     ; comparing variables to determine which is bigger
movlo     MAX, B   ; B is bigger so store it to into MAX
mov     MAX, A   ; A is bigger so store it to into MAX
```

Write ALP program for ARM7 to find number of zeros and number of ones in a 32-bit number.

```
          MOV R3, #0            ; Count for ZEROS
          MOV R4, #0            ; Count for ONES
          MOV R1, 0X20026       ; Some No. 0X20026
          MOV R2, #32            ; Count for 32 bits
AGAIN RORS R1, #1               ; Rotate bits and put LSB to C bit
          BCS ONES              ; Branch if C=1
          ADD R3, R3, #1        ;Else ZERO++
          B NEXT                ;Branch to NEXT
ONES ADD R4, R4, #1             ; ONES++
NEXT SUB R2, R2, #1             ; Subtract 32--
          CMP R2, #0            ;Check 32-COUNT= 0?
BNE AGAIN
```

BCD and ASCII Conversion

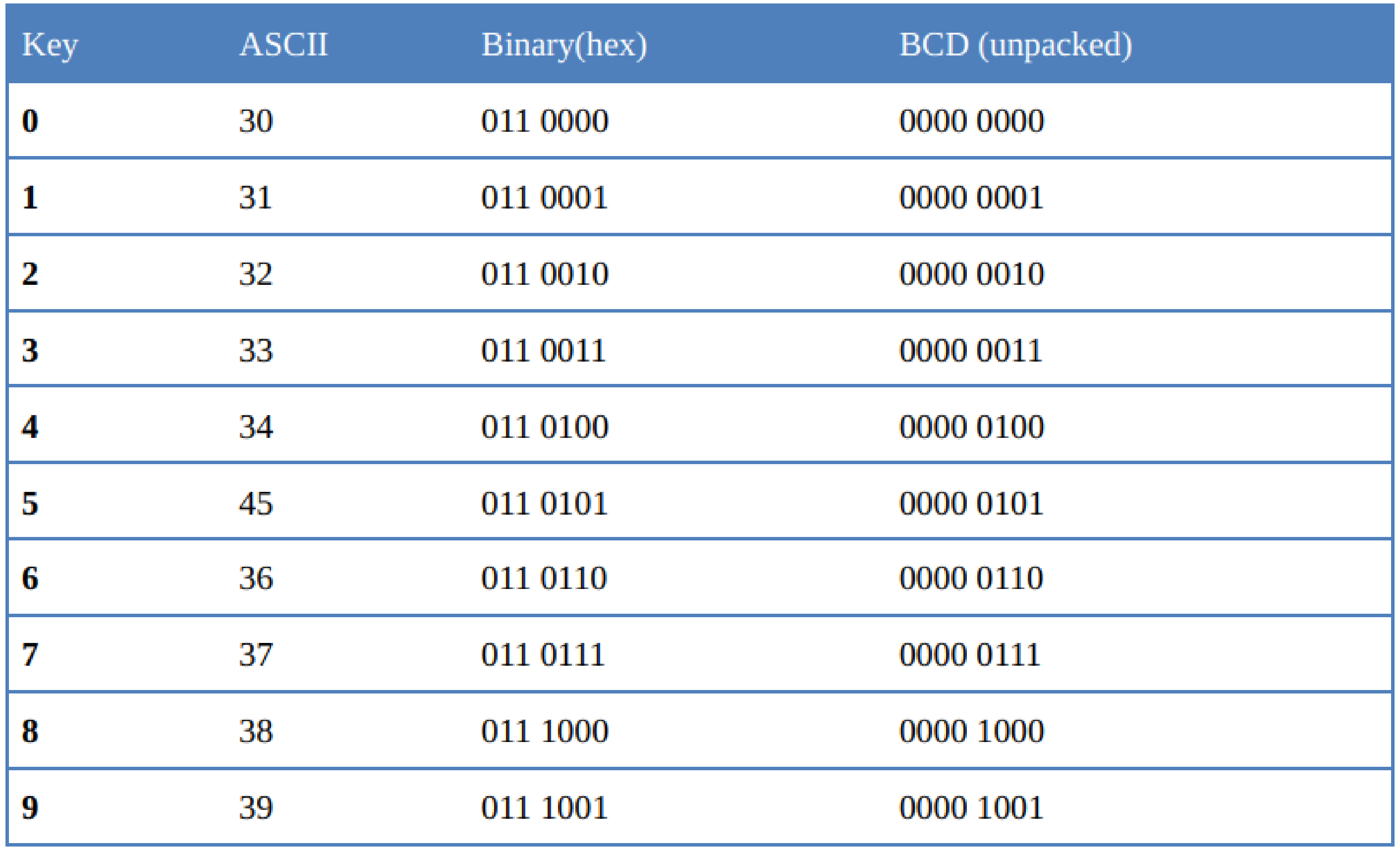
In computer literature one encounters two terms for BCD numbers:

- Unpacked BCD,
- Packed BCD

- **Unpacked BCD** it takes 1 byte of memory location or a register of 8 bits to hold the number
  - Lower 4 bits of the number represent the BCD number and the rest of the bits are 0
    - “0000 1001” ➔ 9
    - “0000 0101” ➔ 5
- **Packed BCD**, a single byte has two BCD numbers in it, one in the lower 4 bits and one in the upper 4 bits.
  - 0101 1001 ➔ 59
- Packed BCD is **twice as efficient** as unpacked BCD while storing the data.

ASCII numbers

In ASCII keyboards, when key “0” is pressed, “011 0000” (0x30) is provided to the computer. In the same way, 0x31 (011 0001) is provided for key “1”, and so on,



Key	ASCII	Binary(hex)	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	45	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

ASCII to unpacked BCD conversion

- To convert ASCII data to unpacked BCD, the programmer must get rid of the tagged “011” in the upper 4 bits of the ASCII.
- To do that, each ASCII number is ANDed with “0000 1111” (0x0F).

ASCII to packed BCD conversion

1. Convert to unpacked BCD (to get rid of the 3)
2. Combined to make packed BCD.

Example

- Lets imagine 27 is pressed in the key board to produce 0x27

Key	ASCII	Unpacked BCD	Packed BCD
2	32	00000010	
7	37	00000111	00100111 (0x27)

```
MOV R1,#0x37        ;R1 = 0x37
MOV R2,#0x32        ;R2 = 0x32
AND R1,R1,#0x0F     ;mask 3 to get unpacked BCD
AND R2,R2,#0x0F     ;mask 3 to get unpacked BCD
MOV R3,R2,LSL #4    ;shift R2 4 bits to left to get R3 = 0x20
ORR R4,R3,R1        ;OR them to get packed BCD, R4 = 0x27
```

1. Convert to unpacked BCD (to get rid of the 3)
2. Combined to make packed BCD.

Packed BCD to ASCII conversion

For data to be printed with printer, data needs to be in the ASCII.
So convert from packed BCD to ASCII.

1. First ,convert the Packed BCD to unpacked BCD
2. Then unpacked BCD is tagged with 011 0000 (0x30)

Packed BCD	Unpacked BCD	ASCII
0x29	0x02 & 0x09	0x32 & 0x39
0010 1001	0000 0010 & 0000 1001	011 0010 & 011 1001

```
MOV R0,#0x29
AND R1,R0,#0x0F         ;mask upper four bits
ORR R1,R1,#0x30         ;combine with 30 to get ASCII
MOV R2,R0,LSR #04       ;shift right 4 bits to get unpacked BCD
ORR R2,R2,#0x30         ;combine with 30 to get ASCII
```

- When a subroutine is called by the BL instruction, control is transferred to that subroutine, and the processor saves the PC (program counter) in the R14 register and begins to fetch instructions from the new location.

- After finishing execution of the subroutine, we must use “BX LR“ instruction to transfer control back to the caller. Every subroutine needs “BX LR” as the last instruction for return address.

Home Work:

- Convert 8 digit ASCII into unpacked BCD

Input: 0x31323334, Output: 0x01020304

- Unpack 8 digit packed BCD

Input: 0x12345678 Output: 0102030405060708

- Pack 8 digit unpacked BCD

Input: 0x01020304, Output: 0x1234

- GCD and LCM