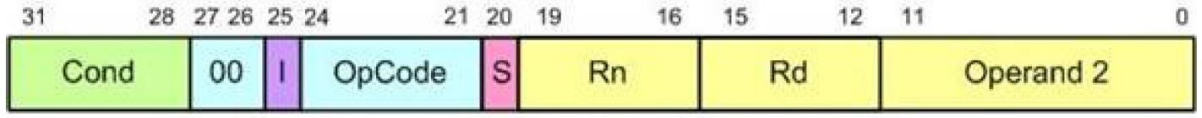
Arithmetic and Logic Instructions and Programs

Chapter -3-

Arithmetic Instructions



S=0, do not update flag (default). S=1 update flags

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	**MULS**	Multiply and set flags
UMULL	Multiply long	**UMULLS**	Multiply Long and set flags
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 Z, C, V and N flag bits of CPSR (current program status register) but the N and V flags are for signed data and are discussed in Chapter 5.

ADD Rd,Rn,Op2 ;Rd = Rn + Op2

```
LDR R2,=0xFFFFFFF5          ;R2=0xFFFFFFF5 (notice the = sign)
MOV R3,#0x0B
ADDS R1,R2,R3               ;R1=R2 + R3 and update the flags
```

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

No increment instruction in ARM

```
ADDS R4,R4,#1
```

ADC (add with carry)

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

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

Subtraction of unsigned numbers

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

Example:

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

SBC (subtract with borrow)

SBC Rd,Rn,Op2 ;Rd = Rn – Op2 – 1 + C

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

To invert the carry flag while running the subtract with borrow instruction it is implemented as “Rd = Rn – Op2 – 1 + C”

No decrement instruction in ARM

RSB (reverse subtract)

RSB Rd,Rn,Op2 ;Rd = Op2 – Rn

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

RSC (reverse subtract with carry)

RSC Rd,Rn,Op2 ;Rd = Op2 – Rn – 1 + C

Multiplication of unsigned numbers in ARM

Not all CPUs have instructions for multiplication and division. All the ARM processors have a multiplication instruction but not the division.

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 provides only the lower 32-bit result in Rd and the rest are dropped if the result is greater than 32-bit. If the result is greater than 0xFFFFFFFF, then we must use UMULL (unsigned Multiply Long) instruction.*

Note 2: *in word-by-word multiplication using MUL instruction, if the result is greater than 32-bit only the lower 32-bit is saved by ARM and the upper part is dropped without setting any flag. 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 (multiply)

The normal multiply instruction (MUL) is used when the result is less than 32-bit, while the long multiply (MULL) must be used when the result is greater than 32-bit.

```
MUL Rd,Rn,Op2     ;Rd = Rn × Op2

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

LDR R1,=100000  ;R1=100,000
LDR R2,=150000  ;R2=150,000
MUL R3,R2,R1      ;R3 is not 15,000,000,000 because it cannot fit in 32 bits.
```

* For this reason we must use **UMULL** (unsigned multiply long) instruction if the result is going to be greater than 0xFFFFFFFF.

UMULL (unsigned multiply long)

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

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 Instructions in ARM

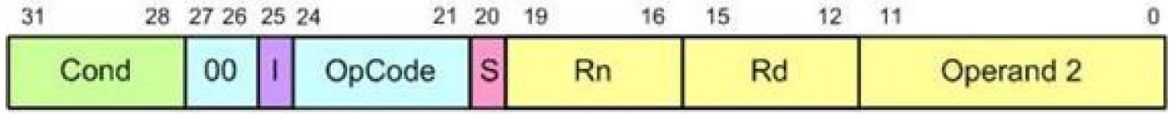
```
MLA Rd,Rm,Rs,Rn    ;Rd = Rm × Rs + Rn

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


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

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



S=0, do not update flag (default). S=1 update 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 (bit clear)

```
BIC     Rd,Rn,Op2       ;clear certain bits of Rn specified by
;the Op2 and place the result in Rd
;i.e. Rd = Rn AND NOT Op2
```

The BIC (bit clear) instruction is used to clear the selected bits of the Rn register. The selected bits are held by Op2. The bits that are HIGH in Op2 will be cleared and bits with LOW will be left unchanged.

Examples Page 140

MVN (move negative)

MVN Rd, Rn ;move negative of Rn to Rd generate one's complement of an operand

"MVN R2,#0" will make R2=0xFFFFFFFF Vs. LDR Rd,=0xFFFFFFFF

```
LDR R2,=0xAAAAAAAA          ;R2 = 0xAAAAAAAA
MVN R0,#0                   ;R0 = 0xFFFFFFFF
EOR R2,R2,R0                ;R2 = R2 ExORed with 0xFFFFFFFF ; = 0x55555555
```

MVN (move negative) instruction is used to generate one's complement of an operand.

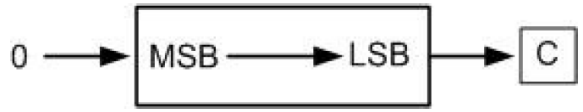
We can also use Ex-OR instruction to generate one's complement of an operand.

Rotate and Barrel Shifter

Barrel Shifter

There are two kinds of shifts: logical and arithmetic. The logical shift is for unsigned operands and the arithmetic shift is for signed operands.

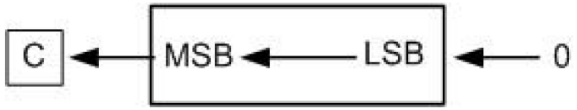
LSR




```
MOV R0,#0x9A              ;R0 = 0x9A
MOVS R1,R0,LSR #3         ;shift R0 to right 3 times ;and then move (copy) the result to R1

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

LSL



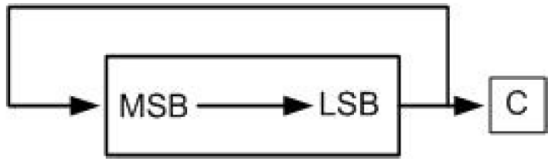

```
LDR R1,=0x0F000006
MOVS R2,R1,LSL #8

TIMES EQU 0x5
LDR R1,#0x7               ;R1=0x7
MOV R2,#TIMES             ;R2=0x05
MOV R1,R1,LSL R2          ;shift R1 left R2 number of times ;and place the result in R1
```

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.			

Arithmetic shift right ASR

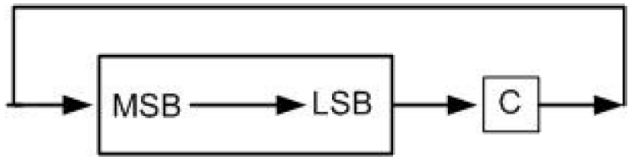
ROR (rotate right)



Rotate left

There is no rotate left option in ARM7 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

RRX rotate right through carry



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

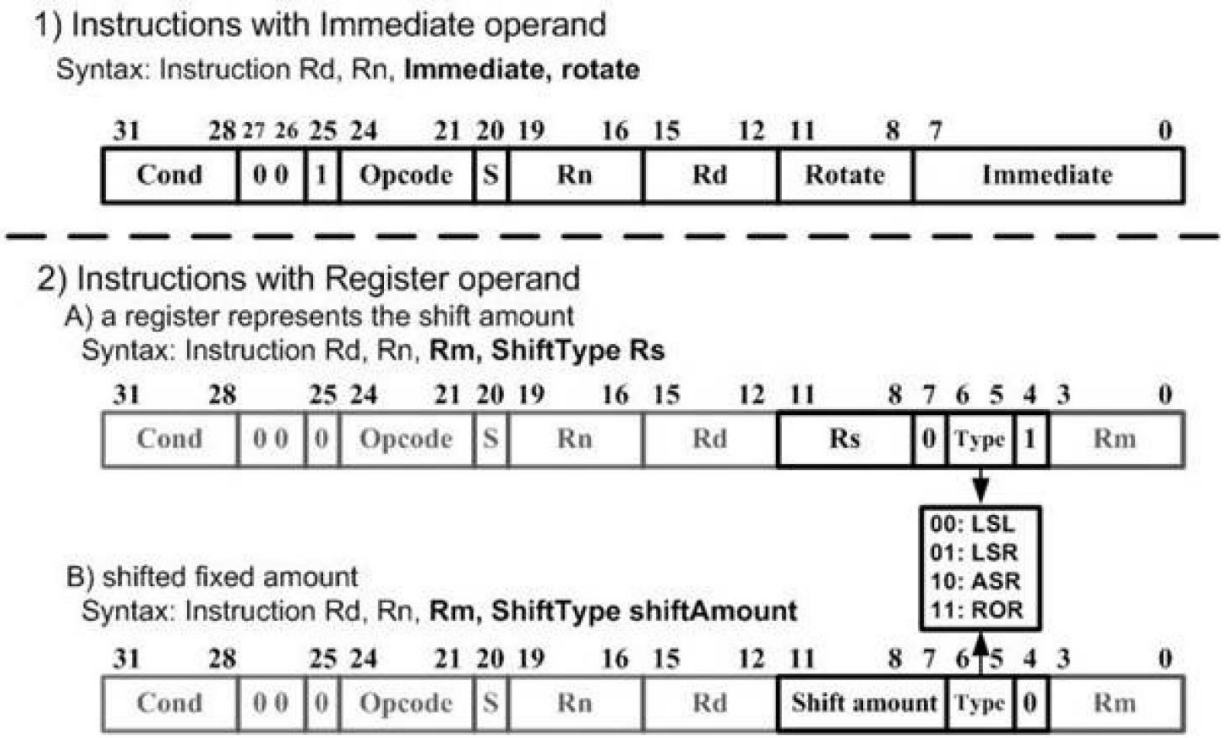


Figure 3- 5: Data Process Instructions

BCD and ASCII Conversion

Unpacked BCD “0000 1001” and “0000 0101”

Packed BCD “0101 1001” is packed BCD for 59

two terms for BCD numbers: (1) unpacked BCD, and (2) packed BCD.

In unpacked BCD, the lower 4 bits of the number represent the BCD number and the rest of the bits are 0.

In the case of packed BCD, a single byte has two BCD numbers in it, one in the lower 4 bits and one in the upper 4 bits.

ASCII numbers

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

In ASCII keyboards, when key “0” is pressed, “011 0000” (0x30) is provided to the computer.

ASCII to packed BCD conversion

```
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
```

Packed BCD to ASCII conversion

the ascii 3 and the number !

```
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
```

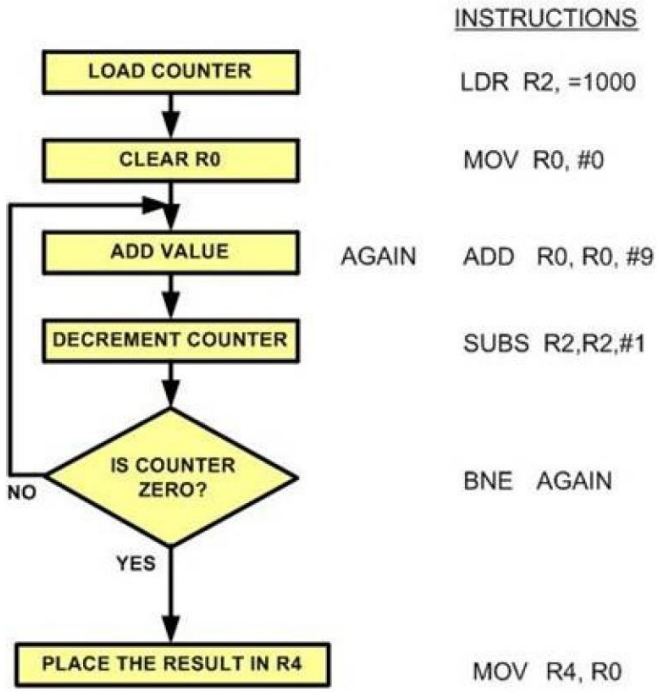
Chapter 4: Branch, Call, and Looping in ARM

Using instruction BNE for looping

The BNE (branch if not equal) instruction uses the zero flag in the status register

```
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
```

```
;– this program adds value 9 to the R0 a 1000 times –
AREA EXAMPLE4_1, CODE, READONLY
ENTRY
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 (when Z = 1)
MOV R4,R0             ;store the sum in R4
HERE B HERE           ;stay here
END
```



Example:
Write a program to place value 0x55 into 100 bytes of RAM locations

```
AREA EXAMPLE4_2, CODE, READONLY
ENTRY
RAM_ADDR EQU 0x40000000     ;change the address for your ARM
MOV R2,#25                  ;counter (25 times 4 = 100 byte block size)
LDR R1,=RAM_ADDR            ;R1 = RAM Address
LDR R0,=0x55555555           ;R0 = 0x55555555
OVER STR R0,[R1]             ;send it to RAM
ADD R1,R1,#4                 ;R1 = R1 + 4 to increment pointer
SUBS R2,R2,#1                ;R2 = R2 – 1 for dec. counter
BNE OVER                      ;keep doing it
HERE B HERE
END
```

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

```
MOV R1,#0                ;clear high word (R1 = 0)
MOV R0,#0                ;clear low word (R0 = 0)
LDR R2,=0x99999999       ;R2 = 0x99999999
ADDS R0,R0,R2            ;R0 = R0 + R2 and set the flags
BCC L1                   ;if C = 0, jump to L1 and add next number
ADDS R1,R1,#1            ;ELSE, increment ( R1 = R1 + 1)
L1 ADDS R0,R0,R2         ;R0 = R0 + R2 and set the flags
BCC L2                   ;if C = 0, add next number
ADDS R1,R1,#1            ;if C = 1, increment
L2 ADDS R0,R2            ;R0 = R0 + R2 and set the flags
BCC L3                   ;if C = 0, add next number
```

```
ADDS R1,R1,#1 ;C = 1, increment
L3 ADDS R0,R2 ;R0 = R0 + R2 and set the flags
BCC L4 ;if C = 0, add next number
ADDS R1,R1,#1 ;if C = 1, and set the flags
L4
```

Comparison of unsigned numbers

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

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

Division by repeated subtraction

```
AREA PROG_4_2, CODE, READONLY ;Division by subtractions
ENTRY
LDR R0,=2012 ;R0 = 2012 (numerator ) ;it will contain remainder
MOV R1,#10 ;R1 = 10 ( denominator )
MOV R2,#0 ;R2 = 0 ( quotient )
L1 CMP R0,R1 ;Compare R0 with R1 to see if less than 10
BLO FINISH ;if R0 < R1 jump to finish
SUB R0,R0,R1 ;R0 = R0 - R1 (division by subtraction)
ADD R2,R2,#1 ;R2 = R2 + 1 (quotient is incremented)
B L1 ;goto L1 (B is discussed in the next section)
FINISH B FINISH
```

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.

```
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
```

```
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

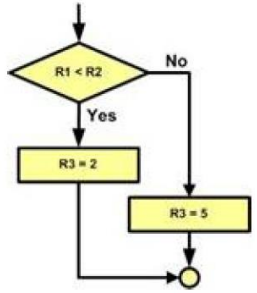
The TEQ instruction is used to test to see if the contents of two registers are equal

```
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
```

B (Branch)

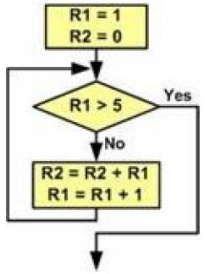
B (branch) is an unconditional jump that can go to any memory location

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




```
//in C
if(R1 < R2)
{
 R3 = 2;
}
else
{
 R3 = 5;
}
```

```
  MOV   R1,#1
  MOV   R2,#0
L1    CMP   R1,#5
  BHI   L2
  ADD   R2,R2,R1
  ADD   R1,R1,#1
  B     L1
L2    MOV   R3,#5
```



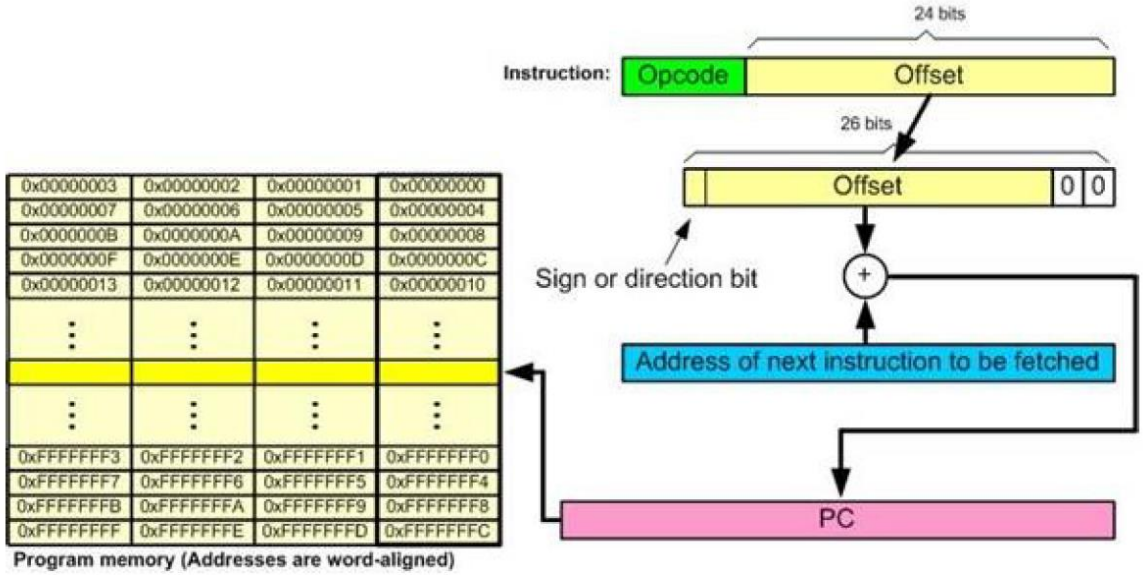

```
//in C
unsigned long R1 = 1;
unsigned long R2 = 0;
while (R1 <= 5)
{
 R2 = R2 + R1;
 R1 = R1 + 1;
}
```

In cases where there is no operating system or monitor program, we use the Branch to itself in order to keep the microcontroller busy. A simple way of doing that is shown below:

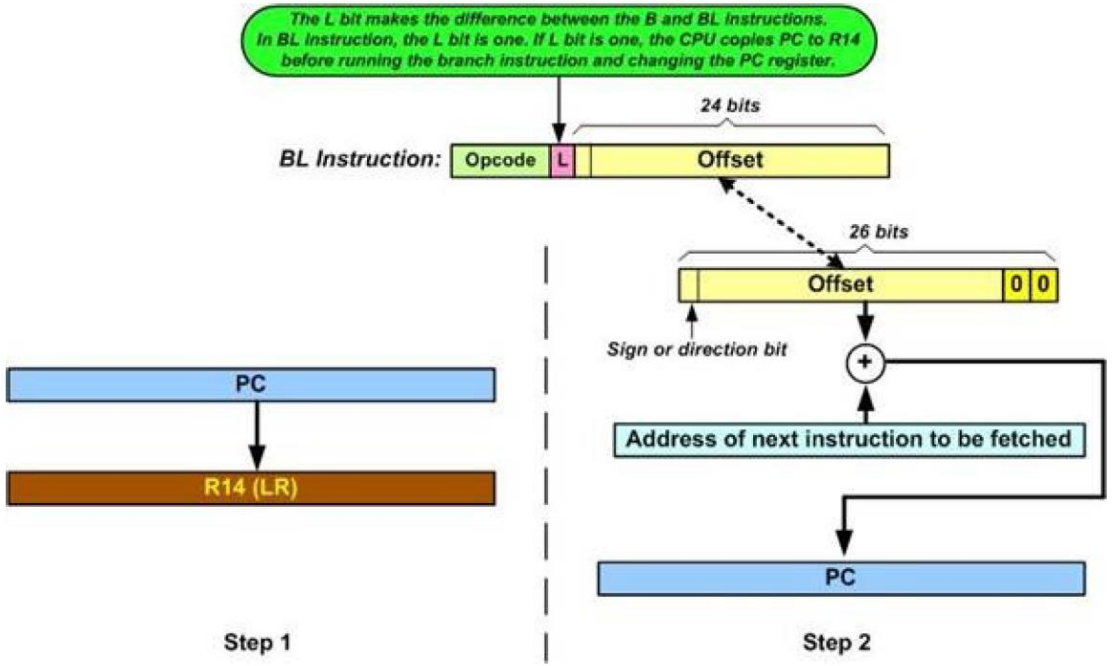
```
HERE B HERE ;stay here
```

Another syntax for the B instruction is BAL (branch always) as shown below:

```
HERE BAL HERE ;stay here
```



BL (Branch and Link) instruction and calling subroutine



When a subroutine is called using BL instruction, first the processor saves the address of the instruction just below the BL instruction on the R14 register (LR, linker register), and then control is transferred to that subroutine

```
AREA EXAMPLE4_8, CODE, READONLY
ENTRY
RAM_ADDR EQU 0x40000000 ;change the address for
your ARM
LDR R1,=RAM_ADDR ;R1 = RAM address
AGAIN MOV R0,#0x55 ;R0 = 0x55
STRB R0,[R1] ;send it to RAM
BL DELAY ;call delay (R14 = PC of next instruction)
MOV R0,#0xAA ;R0 = 0xAA
STRB R0,[R1] ;send it to RAM
BL DELAY ;call delay
B AGAIN ;keep doing it
;————————DELAY SUBROUTINE
DELAY LDR R3,=5 ;R3 =5, modify this value for different size
delay
L1 SUBS R3,R3,#1 ;R3 = R3 - 1
BNE L1
BX LR ;return to caller
;————————end of DELAY subroutine
END ;notice the place for END directive
```

```
;MAIN program calling subroutines
AREA PogramName, CODE, READONLY
ENTRY
MAIN BL SUBR_1 ;Call Subroutine 1
BL SUBR_2 ;Call Subroutine 1
BL SUBR_3 ;Call Subroutine 1
HERE BAL HERE ;stay here. BAL is the same as B
;—-end of MAIN
;————————SUBROUTINE 1
SUBR_1 ….
….
BX LR ;return to main
;— end of subroutine 1

;————————SUBROUTINE 2
SUBR_2 ….
….
BX LR ;return to main
;— end of subroutine 2
;————————SUBROUTINE 3
SUBR_3 ….
….
BX LR ;return to main
;— end of subroutine 3
END ;notice the END of file
```

Signed Multiplication

```
LDR R1,=-3500 ;R1 = -3500 (0xFFFFF254)
LDR R0,=-100 ;R0 = -100 (0xFFFFFF9C)
SMULL R2,R3,R0,R1
```

```
LDR R1,=-3500 ;R1 = -3500 (0xFFFFF254)
MOV R0,#-100 ;R0 = -100 (0xFFFFFF9C)
UMULL R2,R3,R0,R1
```

Signed number comparison

CMP Rn, Op2

Op2 > Rn V = N
Op2 = Rn Z = 1
Op2 < Rn N ≠ V

	Instruction	Action
BEQ	branch equal	Branch if Z = 1
BNE	Branch not equal	Branch if Z = 0
BMI	Branch minus (branch negative)	Branch if N = 1
BPL	Branch plus (branch positive)	Branch if N = 0
BVS	Branch if V set (branch overflow)	Branch if V = 1
BVC	Branch if V clear (branch if no overflow)	Branch if V = 0
BGE	Branch greater than or equal	Branch if N = V
BLT	Branch less than	Branch if N ≠ V
BGT	Branch greater than	Branch if Z = 0 and N = V
BLE	Branch less than or equal	Branch if Z = 1 or N ≠ V

Table 5- 3: ARM Conditional Branch (Jump) Instructions for Signed Data

Arithmetic shift

ASR (arithmetic shift right)

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

CONDITION		Flags	Note
0000	EQ	Z==1	Equal
0001	NE	Z==0	Not Equal
0010	HS/CS	C==1	>= (U) / C=1
0011	LO/CC	C==0	< (U) / C=1
0100	MI	N==1	minus(neg)
0101	PL	N==0	plus(pos)
0110	VS	V==1	V set(ovfl)
0111	VC	V==0	V clr
1000	HI	C==1&&Z==0	> (U)
1001	LS	C==0||Z==1	<= (U)
1010	GE	N==V	>=
1011	LT	N!=V	<
1100	GT	Z==0&&N==V	>
1101	LE	Z==1||N!=V	<=
1110	AL	always	
1111	NE	never	
			(U) = unsigned

All ARM instructions can be conditional instructions. Examples:

ADDEQ R0,R1, #23 ; R0 = R1 – 23 only if ZF==1

SUBGT R2,R3, R4 ; R2=R3-R4 only if VF=NF && ZF=0

And so on

Summary of ARM's Indexed Addessing Modes

Addressing Mode	Assembly Mnemonic	Effective address	FinalValue in R1
Pre-indexed, base unchanged	LDR R0, [R1, #d]	R1 + d	R1
Pre-indexed, base updated	LDR R0, [R1, #d]!	R1 + d	R1 + d
Post-indexed, base updated	LDR R0, [R1], #d	R1	R1 + d