Features of ARM instruction set:

- ARM defines two instruction
  - ARM state instruction set (32-bit instruction)
  - Thumb state instruction set (16-bit instruction)
- All instruction are 32bit long (i.e. fixed length)
- Most of the instructions are executed in single clock cycle
- ARM supports conditional execution of every instruction
- ARM instruction set combines shift and ALU operation in a single instruction.
- The ARM Processor has load store architecture. In ARM load and store instruction access the memory
- Instruction set extension via coprocessor.

The ARM processor operations illustrated using pre- and post-conditions.

- Pre- and post-conditions describes registers and memory before and after the instruction or instructionsareexecuted.
- The examples follow this format:

  PRE <pre-conditions>
  < Instructions >
  POST <post-conditions>
- We will represent hexadecimal numbers with the prefix **0x** and binary numbers with the prefix **0b**.
- ARM has data processing instructions, branch instructions, load-store instructions, software interrupt instruction, and program status registerinstructions.

Data Processing Instructions

- The data processing instructions manipulate data within registers.
- They are move instructions,arithmetic instructions, logical instructions, comparison instructions, andmultiplyinstructions.
- Most data processing instructions can process one of their operands using thebarrel shifter.
- If you use the **S suffix** on a data processing instruction, then it **updates the flags in thecpsr**.
- The carry flag is set from the result of the barrel shift as the last bit shifted out.
- The N flag is set a bit 31 of the result.
- The Z flag is set if the result is zero.

Move Instructions

- Move is the simplest ARM instruction.
- It copies N into a destination register Rd, whereN is a register or immediate value.
- This instruction is useful for setting initial values andtransferring data between registers.

Syntax: <instruction>{<cond>}{S} Rd, N

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

- The below example shows a simple move instruction.
- The MOV instruction takes the contents ofregister r5 and copies them into register r7, in this case,taking the value 5, and overwritingthe value 8 in register r7.

PRE r5 = 5
r7 = 8
MOV r7, r5; let r7 = r5
POST r5 = 5
r7 = 5

- The below example of a **MOVS** instruction shifts register r1 left by one bit.
- As you can see, the C flag is updated in the cpsr because the S suffix ispresent in the instruction mnemonic.

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 0x00000000
        r1 = 0x80000004

        MOVS    r0, r1, LSL #1

POST    cpsr = nzCvqiFt_USER
        r0 = 0x00000008
        r1 = 0x80000004
```

More examples:

MOVCS R0, R1 ; if carry is setthen R0:=R1
MOVS R0, #0 ; R0 = 0 (Z=1, N=0, C &V unaffected)

Barrel Shifter

- In a MOV instruction where *N* can be more than just a register or immediate value; it can also be a register *Rm*that has been Pre-processed by the barrel shifter prior to being used by a data processing instruction.
- A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.

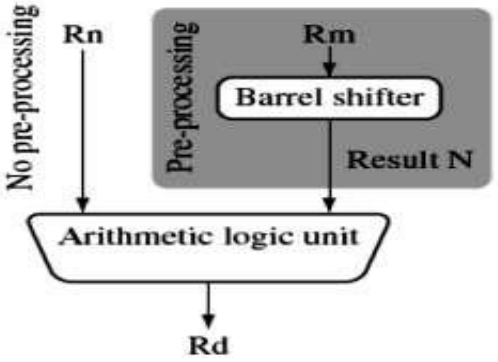
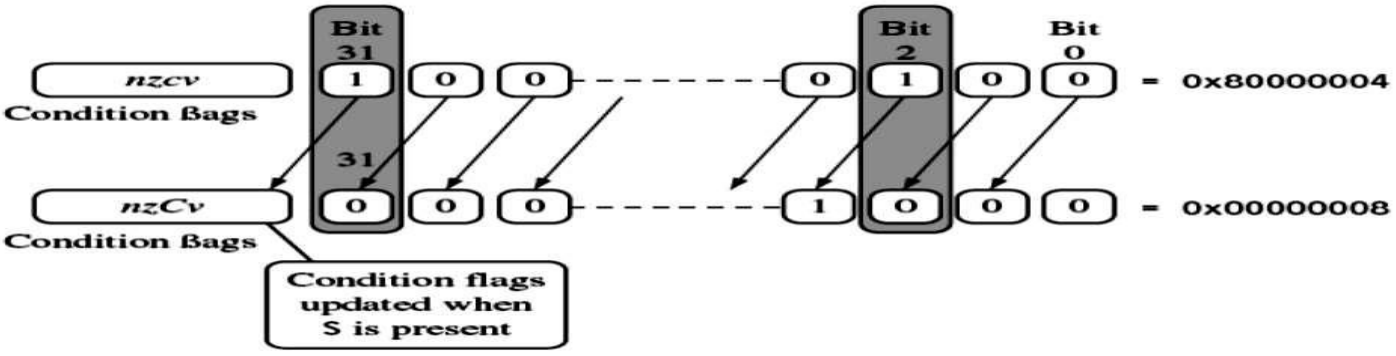


Table below lists the syntax for the different barrel shift operations available on data processinginstructions.

Mnemonic	Description	Shift
LSL	logical shift left	*x*LSL *y*
LSR	logical shift right	*x*LSR *y*
ASR	arithmetic right shift	*x*ASR *y*
ROR	rotate right	*x*ROR *y*

Note: *x* represents the register being shifted and *y* represents the shift amount.



Note:

SHIFT Operation:

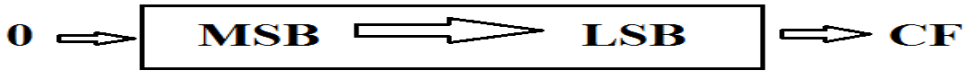
- Shifts the contents of a register or memory location right or left.
- There are two kinds of shifts:

  Logical - for unsigned operands.

  Arithmetic - for signed operands.
- The number of times (or bits) the operand is shifted can be specified directly if it is once only.
- if it is more than once specified through the CL register.

LSR - logical shift right:

- Operand is shifted right bit by bit.
- For every shift the LSB (least significant bit) will go to the carry flag. (CF)
- The MSB (most significant bit) is filled with 0.



Show the result of **LSR** in the following:

MOV R0, # 0x9A

MOV R1, R0 , LSR #3

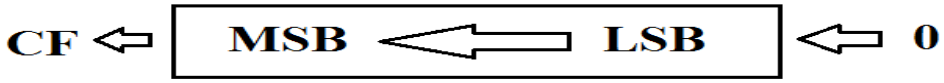
Solution:

9A =	**0000 .....**	10011010	
	0000	01001101	CF = 0 (shifted once)
	0000	00100110	CF = 1 (shifted twice)
	0000	00010011	CF = 0 (shifted three times)

After three time shift R1 = 0x00000013

LSL - Logical shift left

- After every shift, the LSB is filled with 0.
- MSB goes to CF.



Show the effects of **LSL** in the following:

MOV R0, # 0x6

MOV R1, R0, LSL # 4

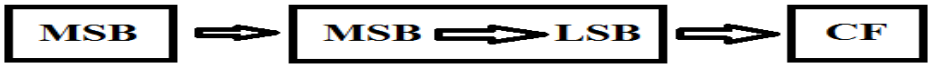
Solution:

	06 = 0000	00000110	
CF=0	**0000 .....**	00001100	(shifted left once)
CF=0	**0000 .....**	00011000	
CF=0	**0000 .....**	00110000	
CF=0	**0000 .....**	01100000	(shifted four times)

After the four shifts left, the **R1** register has = **0X00000060**

ASR(shift arithmetic right):

- As the bits of the destination are shifted to the rightinto CF, the empty bits are filled with the sign bit.



Example:

```
PRE   CPSR = nzcvqift_user
      MOV r0 , # 1
      MOVS  r1, r0 , ASR #1
```

r0 = 0x00000001 = 0000 0000 0000 0000 0000 0000 0000 0001

MSB LSB

0000 0000 0000 0000 0000 0000 0000 0000 carry

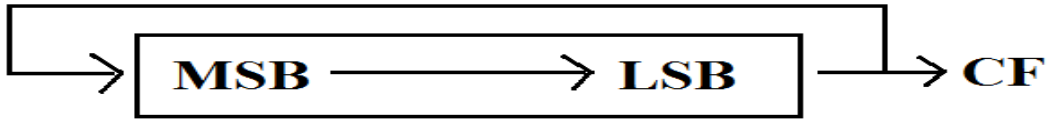
POST CPSR = nZCvqift_user

ROTATE INSTRUCTIONS

- ROR, ROL are designed specifically to perform a bitwise rotation of an operand.
- They allow a program to rotate an operand right or left.

ROR (Rotate Right):

- In ROR the LSB is moved to the MSB, & is also copied to CF.

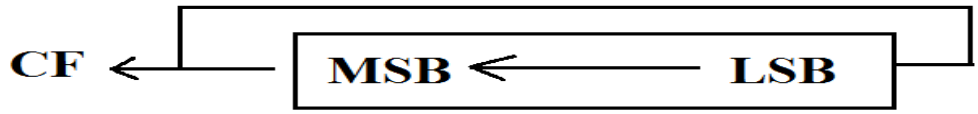


EXAMPLE:

```
MOV R0, # 0X36          ; R0 =  0000 .....   0011 0110
MOV R1, R0, ROR #1      ; R1 =  0000 .....   0001 1011   CF=0 ,  R1 =0X0000001B
```

ROL (Rotate Left):

- In ROL the MSB is moved to the LSB and is also copied to CF




```
MOV R0, #0X72
MOV R1, R0, ROL #1      ; R0 =  0000 ..... 0111 0010
                        ; R1= 0000 .....   1110 0100   CF=0
                          R1 = 0X000000E4
```

Arithmetic Instructions

The arithmetic instructions implement addition and subtraction of 32-bit signed andunsigned values.

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
SUB	subtract two 32-bit values	$Rd = Rn - N$

*** N is the result of the shifter operation.

Addition:

- For example an add instruction takes the form:

ADD r0, r1, r2 ; r0 = r1 + r2

- To execute an instruction conditionally; simply postfix it with theappropriate condition:

ADD**EQ** r0, r1, r2 ; If zero flag set then… r0 = r1 + r2

- By default, data processing operations do not affect the condition flags. To cause the condition flags tobe updated, the S bit of the instruction needs to be setby post-fixing the instruction with an "S".
- For example to add two numbers and set the condition flags:

ADDS r0, r1, r2 ; r0 = r1 + r2 ... and set flags

- In below example Register r1 is first shifted one location to the left. TheADD instruction then adds the result of the barrel shift operation to register r1. The final resulttransferredinto register r0.

```
PRE     r0 = 0x00000000
        r1 = 0x00000005

        ADD     r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

Subtraction:

- This simple subtract instruction subtracts a value stored in register r2 from a value storedin register r1.
- The result is stored in register r0.

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000001

        SUB r0, r1, r2

POST    r0 = 0x00000001
```

How to perform SUBTRACTION:

- Take the 2's complement of the r2
- Add it to the r1
- Invert the carry.
- Store result in r0

Note: After the execution, if CF = 0, the result is positive. If CF = 1, the result is negative and the destination has the 2's complement of the result.

```
r2 = 0x00000001 = 0000 0000 0000 0000 0000 0000 0000 0001
     1's complement = 1111 1111 1111 1111 1111 1111 1111 1110
                                                        +1
r2 =  2's complement = 1111 1111 1111 1111 1111 1111 1111 1111
r1 = 0x00000002 =  0000 0000 0000 0000 0000 0000 0000 0010
   add r1+r2 =   1  0000 0000 0000 0000 0000 0000 0000 0001
              carry

      ro = 0000 0000 0000 0000 0000 0000 0000 0001 =0x00000001
```

Reverse subtract

- The reverse subtract instruction (RSB) subtracts r1 from the constant value #0, writing theresult to r0.
- You can use this instruction to negate numbers.

```
PRE     r0 = 0x00000000
        r1 = 0x00000077

        RSB r0, r1, #0      ; Rd = 0x0 - r1

POST    r0 = -r1 = 0xffffff89
```

Subtract with cpsr is updated:

- The SUBS instruction is useful for decrementing loop counters.
- In below example we subtractthe immediate value one from the value one stored in register r1.
- The result value zero iswritten to register r1.
- The cpsr is updated with the ZC flags being set.

```
PRE     cpsr = nzcvqiFt_USER
        r1 = 0x00000001

        SUBS r1, r1, #1

POST    cpsr = nZCvqiFt_USER
        r1 = 0x00000000
```

Logical Instructions:

- Logical instructions perform bitwise logical operations on the two source registers

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

Logical OR:

- This example shows a logical OR operation between registers r1 and r2and r0holds the result.

```
PRE     r0 = 0x00000000
        r1 = 0x02040608
        r2 = 0x10305070

        ORR   r0, r1, r2

POST    r0 = 0x12345678
```

Bit clear:

- This example shows a more complicated logical instruction called BIC, which carries outa logical bitclear.

```
PRE     r1 = 0b1111
        r2 = 0b0101

        BIC   r0, r1, r2

POST    r0 = 0b1010

This is equivalent to

        Rd = Rn AND NOT(N)
```

- In this example, register *r2* contains a binary pattern where every binary 1 in *r2* clears a corresponding bit location in register *r1*.
- This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the *cpsr*.

Comparison Instructions:

- The comparison instructions are used to compare a 32-bit value.
- They update the cpsr flag bits according to the result, but do not affect other registers.
- After the bits have been set, the information can then be used to change program flow byusingconditional execution.
- We do not need to apply the S suffix for comparison instructions to updatethe flags.

Example:

```
PRE     cpsr = nzcvqiFt_USER
        r0 = 4
        r9 = 4

        CMP   r0, r9

POST    cpsr = nZcvqiFt_USER
```

This example shows a CMP comparison instruction.

- We can see that both registers, r0 andr9, are equal before executing the instruction.
- The value of the z flag prior to execution is 0and is represented by a lowercase z.
- After execution the z flag changes to 1 or an uppercaseZ. This change indicates equality.
- The CMP is effectively a subtract instruction with the result discarded

TST and TEQ:

- The TSTinstruction is a logical AND operation, and TEQ is a logical exclusive OR operation.
- Foreach, the results are discarded but the condition bits are updated in the cpsr.

Multiply Instructions

- The multiply instructions multiply the contents of a pair of registers andaccumulate the results in with destination register.

```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm * Rs) + Rn$
MUL	multiply	$Rd = Rm * Rs$

Example:

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002

        MUL   r0, r1, r2    ; r0 = r1*r2

POST    r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002
```

- ❖ Above example shows a simple multiply instruction that multiplies registers r1 and r2 togetherandplaces the result into register r0.
- ❖ In this example, register r1 is equal to the value 2, andr2is equal to 2. The result, 4, is then placed intoregister r0.

- ➢ The long multiply instructions (SMLAL, SMULL, UMLAL, and UMULL) produce a 64-bitresult.
- ➢ The result is too large to fit a single 32-bit register so the result is placed in tworegisters labelled*RdLo*and *RdHi*.
- ➢ *RdLo*holds the lower 32 bits of the 64-bit result, and*RdHi*holds the higher 32 bits of the 64-bit result.
- ➢ In below example instruction multiplies registers *r2* and *r3* and places the result into register *r0* and *r1*.
- ➢ Register *r0* contains the lower 32 bits, and register *r1* contains the higher 32 bits of the64-bit result.

```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002

        UMULL   r0, r1, r2, r3   ; [r1,r0] = r2*r3

POST    r0 = 0xe0000004 ; = RdLo
        r1 = 0x00000001 ; = RdHi
```

Branch Instructions

- ❖ A branch instruction changes the flow of execution or is used to call a routine.
- ❖ The change of execution flow forces the program counter [pc]to point to a new address.
- ❖ This typeof instruction allows programs to have subroutines, if-then-else structures, and loops.
- ❖ The commonly used branch instructions are.

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
```

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm*

This example shows a forward and backward branch.

- ❖ The forward branch skips threeinstructions.
- ❖ The backward branch creates an infinite loop.

```
        B       forward
        ADD     r1, r2, #4
        ADD     r0, r6, #2
        ADD     r3, r7, #4
forward
        SUB     r1, r2, #4
```

```
backward
        ADD     r1, r2, #4
        SUB     r1, r2, #4
        ADD     r4, r6, r7
        B       backward
```

- ✓ The branch with link, or BL, instruction is similar to the B instruction but overwrites thelink register *lr*with a return address.
- ✓ It performs a subroutine call.
- ✓ To returnfrom a subroutine, we copy the link register to the *pc*.

```
        BL      subroutine      ; branch to subroutine
        CMP     r1, #5          ; compare r1 with 5
        MOVEQ   r1, #0          ; if (r1==5) then r1 = 0
        :
subroutine
        <subroutine code>
        MOV     pc, lr          ; return by moving pc = lr
```

- ✓ The branch exchange (BX) is the third type ofbranch instruction.
- ✓ The BX instruction uses an absolute address stored in register *Rm*.
- ✓ Itis primarily used to branch to and from Thumb code

Load-Store Instructions

- Load-store instructions transfer data between memory and registers.
- There arethree types of load-store instructions: single-register transfer, multiple-register transfer,andswap.

Single-Register Transfer

- These instructions are used for moving a single data item in and out of a register.
- Thedatatypes supported are signed and unsigned words (32-bit), half-words (16-bit), and bytes(8bit).

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*

LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

Example:

```
LDR     r0, [r1]            ; = LDR r0, [r1, #0]
```

Above instruction load register ro with the content of the memory address pointed by register r1

```
STR     r0, [r1]            ; = STR r0, [r1, #0]
```

Above instruction store the content of register r0 to the memory address pointed by register r1

Single-Register Load-Store Addressing Modes

- The ARM instruction set provides different modes for addressing memory.
- These modesincorporate one of the indexing methods: preindex with writeback, preindex, and postindex

Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

Note: ! indicates that the instruction writes the calculated address back to the base address register.

- Preindex with writeback calculates an address from a base register plus address offset andthen updatesthat address base register with the new address.

```
PRE         r0 = 0x00000000
            r1 = 0x00090000
            mem32[0x00009000] = 0x01010101
            mem32[0x00009004] = 0x02020202

            LDR     r0, [r1, #4]!

Preindexing with writeback:

POST(1)     r0 = 0x02020202
            r1 = 0x00009004
```

- Preindexcalculates an address from a base register plus address offset but does not update theaddress base register.
- The preindexmode is useful for accessing an element in a data structure.

```
PRE         r0 = 0x00000000
            r1 = 0x00090000
            mem32[0x00009000] = 0x01010101
            mem32[0x00009004] = 0x02020202

            LDR     r0, [r1, #4]

Preindexing:

POST(2)     r0 = 0x02020202
            r1 = 0x00009000
```

- Postindex only updates the address base register after the address is used.
- The postindex and preindex with writeback modes are useful for traversing an array.

```
PRE        r0 = 0x00000000
           r1 = 0x00090000
           mem32[0x00009000] = 0x01010101
           mem32[0x00009004] = 0x02020202

           LDR      r0, [r1], #4

Postindexing:

POST(3)    r0 = 0x01010101
           r1 = 0x00009004
```

Multiple-Register Transfer

Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction.

Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}

LDM	load multiple registers	{Rd}^{*N} <- mem32[start address + 4*N] optional Rn updated
STM	save multiple registers	{Rd}^{*N} -> mem32[start address + 4*N] optional Rn updated

Addressing mode for load-store multiple instructions.

Addressing mode	Description	Start address	End address	*Rn*!
IA	increment after	*Rn*	*Rn* + 4*N* − 4	*Rn* + 4*N*
IB	increment before	*Rn* + 4	*Rn* + 4*N*	*Rn* + 4*N*
DA	decrement after	*Rn* − 4*N* + 4	*Rn*	*Rn* − 4*N*
DB	decrement before	*Rn* − 4*N*	*Rn* − 4	*Rn* − 4*N*

Note- N is the number of registers in the list of registers.

Example:1

```
PRE     mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02
        mem32[0x80010] = 0x01
        r0 = 0x00080010
        r1 = 0x00000000
        r2 = 0x00000000
        r3 = 0x00000000

        LDMIA    r0!, {r1-r3}

POST    r0 = 0x0008001c
        r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003
```

Address pointer	Memory address	Data	
	0x80020	0x00000005	
	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000000
	0x80014	0x00000002	*r2* = 0x00000000
***r0* = 0x80010** →	0x80010	0x00000001	*r1* = 0x00000000
	0x8000c	0x00000000	

Pre-condition for LDMIA instruction.

Address pointer	Memory address	Data	
	0x80020	0x00000005	
***r0* = 0x8001c** →	0x8001c	0x00000004	
	0x80018	0x00000003	*r3* = 0x00000003
	0x80014	0x00000002	*r2* = 0x00000002
	0x80010	0x00000001	*r1* = 0x00000001
	0x8000c	0x00000000	

Post-condition for LDMIA instruction.

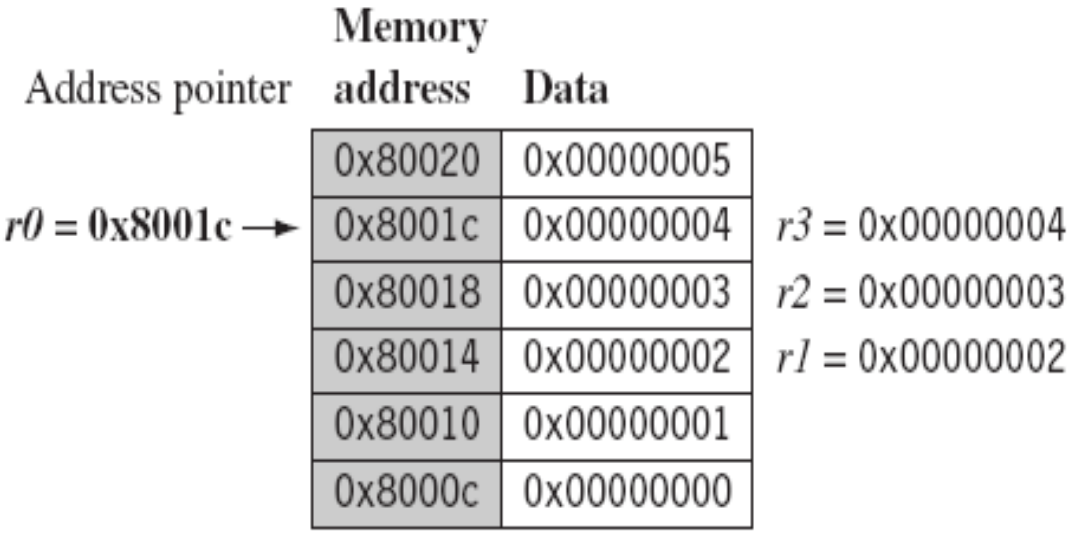
- Register *r0*is the base register *Rn*and is followed by !, indicating that the register is updated after the instruction is executed.
- You will notice within the load multiple instruction that the registers are not individually listed.
- Instead the "-" character is used to identify a range of registers. In this case the range is from register *r1* to *r3* inclusive.
- Each register can also be listed, using a comma to separate each register within "{" and "}" brackets.

Example-2

Memory address	Data
0x80020	0x00000005
0x8001c	0x00000004
0x80018	0x00000003
0x80014	0x00000002
0x80010	0x00000001
0x8000c	0x00000000

```
r0  =  0x00080010
r1  =  0x00000000
r2  =  0x00000000
r3  =  0x00000000

LDMIB       r0!, {r1-r3}
```



Example-3

```
PRE         r1  =  0x00000001
            r2  =  0x00000002

            r3  =  0x00000003
            r4  =  0x9000

            STMIA    r4!,{r1,r2,r3}
```

```
POST    mem32[0x9000] = 0x00000001
        mem32[0x9004] = 0x00000002
        mem32[0x9008] = 0x00000003
        r4 = 0x900c
```

Example:4

```
PRE  r0 = 0x00080010
     r1 = 0x00000001
     r2 = 0x00000002
     r3 = 0x00000003
```

Address pointer	Memory address	Data
	0x80020	0x00000005
r0 = 0x8001c →	0x8001c	0x00000009
	0x80018	0x00000008
	0x80014	0x00000007
	0x80010	0x00000001
	0x8000c	0x00000000

```
LDMDA r0!, {r1-r3}
   POST      r0 = 0x00080010
             r1 = 0x00000009
             r2 = 0x00000008
             r3 = 0x00000007
```

Swap Instruction:

- The swap instruction swaps the contents ofmemory with the contents of a register.
- The swap instruction loads a word from memory into register r0 and overwrites the memorywith registerr1.

SWP	swap a word between memory and a register	*tmp = mem32[Rn]* *mem32[Rn] = Rm* *Rd = tmp*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] = Rm* *Rd = tmp*

Example:

```
PRE     mem32[0x9000] = 0x12345678
        r0 = 0x00000000
        r1 = 0x11112222
        r2 = 0x00009000

        SWP     r0, r1, [r2]

POST    mem32[0x9000] = 0x11112222
        r0 = 0x12345678
        r1 = 0x11112222
        r2 = 0x00009000
```

Program Status Register Instructions

- The ARM instruction set provides two instructions to directly control a program statusregister (psr).
- The MRS instruction transfers the contents of either the cpsrorspsrintoa register
- The MSR instruction transfers the contents of a registerinto the cpsrorspsr.
- Together these instructions are used to read and write the cpsrandspsr.

Syntax:

MRS{<cond>} Rd,<cpsr|spsr>
MSR{<cond>} <cpsr|spsr>_<fields>,Rm
MSR{<cond>} <cpsr|spsr>_<fields>,#immediate

MRS	copy program status register to a general-purpose register	*Rd = psr*
MSR	move a general-purpose register to a program status register	*psr[field] = Rm*
MSR	move an immediate value to a program status register	*psr[field] = immediate*

Example:

```
PRE     cpsr = nzcvqIFt_SVC

        MRS     r1, cpsr
        BIC     r1, r1, #0x80 ; 0b01000000
        MSR     cpsr_c, r1

POST    cpsr = nzcvqiFt_SVC
```

- The MSR first copies the *cpsr*into register *r1*.
- The BIC instruction clears bit 7 of *r1*.
- Register*r1* is then copied back into the *cpsr*, which enables IRQ interrupts.

Loading Constants

- There is no ARM instruction to move a 32-bit constant intoa register.
- SinceARMinstructions are 32 bits in size, they obviously cannot specify a general32-bit constant.
- To aid programming there are two pseudoinstructions to move a 32-bit value intoa register.

```
Syntax: LDR Rd, =constant
        ADR Rd, label
```

LDR	load constant pseudoinstruction	*Rd* = 32-bit constant
ADR	load address pseudoinstruction	*Rd* = 32-bit relative address

- The first pseudoinstruction writes a 32-bit constant to a register using whatever instructions are available.
- The second pseudoinstruction writes a relative address into a register.

Stack Operations:

- The ARM architecture uses the load-store multiple instructions to carry out stack operations.
- The pop operation (removing data from a stack) uses a load multiple instruction
- Similarly, the push operation (placing data onto the stack) uses a store multiple instruction.
- When using a stack you have to decide whether the stack will grow up or down in memory.
- A stack is either ascending (A) or descending (D).
- Ascending stacks grow towards higher memory addresses; descending stacks grow towards lowermemoryaddresses.
- When you use a full stack (F), the stack pointer **sp** points to an address that is the last used or fulllocation (i.e., sp points to the last item on the stack).
- In contrast, if you use an empty stack (E) the sp points to an address that is the first unused or emptylocation (i.e., it points after the last item on the stack).
- The LDMFD and STMFD instructions provide the pop and push functions, respectively.
- The STMFD instruction pushes registers onto the stack, updating the sp. Figure below shows a pushonto a full descending stack.
- We can see that when the stack grows the stack pointerpoints to the last full entry in the stack.

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080014

        STMFD   sp!, {r1,r4}
```

PRE

	Address	Data
	0x80018	0x00000001
sp →	0x80014	0x00000002
	0x80010	*Empty*
	0x8000c	*Empty*

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
sp →	0x8000c	0x00000002

```
POST    r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x0008000c
```

Addressing methods for stack operations.

Addressing mode	Description
FA	full ascending
FD	full descending
EA	empty ascending
ED	empty descending

Example:2

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080010

        STMED   sp!, {r1,r4}
```

```
POST    r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080008
```

PRE

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
sp →	0x80010	*Empty*
	0x8000c	*Empty*
	0x80008	*Empty*

POST

	Address	Data
	0x80018	0x00000001
	0x80014	0x00000002
	0x80010	0x00000003
	0x8000c	0x00000002
sp →	0x80008	*Empty*

Software Interrupt Instruction

- A software interrupt instruction (SWI) causes a software interrupt exception, which providesa mechanism for applications to call operating system routines.
- When the processor executes an SWI instruction, it sets the program counter *pc* to theoffset 0x8 in the vector table.
- The instruction also forces the processor mode to *SVC*.

- Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

```
Syntax: SWI{<cond>} SWI_number
```

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts)

Example:

```
PRE     cpsr = nzcVqift_USER
        pc = 0x00008000
        lr = 0x003fffff; lr = r14
        r0 = 0x12

      0x00008000   SWI     0x123456

POST    cpsr = nzcVqIft_SVC
        spsr = nzcVqift_USER
        pc = 0x00000008
        lr = 0x00008004
        r0 = 0x12
```

Coprocessor Instructions

- Coprocessor instructions are used to extend the instruction set.
- Coprocessor instructionsprovide additional computation.
- The coprocessor instructions include dataprocessing, register transfer, and memory transfer instructions.

```
Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
        <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
        <LDC|STC>{<cond>} cp, Cd, addressing
```

- In the syntax of the coprocessor instructions, the *cp*field represents the coprocessor number.
- The *opcode*fields describe the operation to take place on the coprocessor.
- The *Cn*, *Cm*, and *Cd* fields describe registers within the coprocessor.

CDP	coprocessor data processing—perform an operation in a coprocessor
MRC MCR	coprocessor register transfer—move data to/from coprocessor registers
LDC STC	coprocessor memory transfer—load and store blocks of memory to/from a coprocessor

Assignment Questions:

1) Explain MOV instruction with examples.**5M**
2) Explain with example different multiply instruction with example?**8M**
3) Explain barrel shifter in ARM instructions with an example.**4M**
4) Explain the arithmetic instructions in ARM with examples.**6M**
5) Explain the logical instructions with examples.**6M**
6) Explain the Branch instruction with an example. **5M**
7) Explain the stack operation instructions in ARM?**5M**
8) Write a short note on SWAP instructions with examples with respect to ARM.**4M**
9) For the following instruction find the value of result register and CPSR
   [Given CPSR=nzcvqiFt_USER, r0=0x00000000, r1=0x80000004, r2= 0x00000077, r3=0x00000001, r4=0x00000005, r5=0x00000005, r6=0b1111, r7=0b0101,]
   i) MOVS ro,r1,LSL #1 find value of r0 ,r1 and CPSR
   ii) SUBS r3,r3, #1 find value of r3 and CPSR
   iii) RSB r0,r2,#0 find value of r0
   iv) ADD r0,r4,r4,LSL #1 find value of r0
   v) BIC r0,r6,r7 find value of r0
10) Assume r5=5, r7 = 8, Show the contents of registers r5 & r7 after the execution of MOV r7, r5, LSL #2.
11) What are the salient features of ARM instruction set? **5M**
12) Explain software interrupt instruction of ARM processor? **5M**
13) Explain briefly co-processor instruction of ARM processor?**12M**
14) Explain Profiling and Cycle Counting? **5M**
15) Write an alp to display the message Hello students. **5M**
16) Explain the syntax with example the following instruction of ARM processor?**10M**
    i) MVN ii) RSB iii) ORR iv) MLA v) SMULL vi) LDR vii) SWPB viii) LSL ix) QADD
17) Discuss the load store instruction with example with respect to **7M**
    i) single register transferii) multiple register transfer

Note:

ARMv5 EXTENSIONS

CLZ {<cond>} Rd, Rm	; count leading zeros
QADD {<cond>} Rd, Rm, Rn	;signed saturated 32-bit add
QSUB{<cond>} Rd, Rm, Rn	; signed saturated 32-bit subtract

Example-1:

(i) CLZ

```
PRE     r1 = 0b00000000000000000000000000010000

        CLZ r0, r1

POST    r0 = 27
```

Example-2:
(ii) QADD

```
PRE     cpsr = nzcvqiFt_SVC
        r0 = 0x00000000
        r1 = 0x70000000 (positive)
        r2 = 0x7fffffff (positive)

        QADD    r0, r1, r2

POST    cpsr = nzcvQiFt_SVC
        r0 = 0x7fffffff
```

Note that the saturated number is returned in register *r0*. Also the *Q* bit (bit 27 of the *cpsr*) has been set, indicating saturation has occurred.

More examples:

(i) LDR r0 , [r1,#5]! ; r0 = mem [r1 +5] and r1 =r1 +5 i.e. copy the content of memory location pointed by base register r1 plus offset to the register r0 and then update base register with new address.

(ii) LDR r0 , [r1,#5] ; r0 = mem [r1 +5]i.e. copy the content of memory location pointed by base register r1 plus offset to the register r0.

(ii) LDR r0 , [r1] ,#5 ; r0 = mem [r1] and r1 =r1 +5.i.e. copy the content of memory location pointed by base register r1 and then update base register with new address.