15CS44: MICROPROCESSORS AND MICROCONTROLLERS

QUESTION BANK with SOLUTIONS

<u>MODULE-5</u>

1) Which are the different data processing instructions of ARM processor?

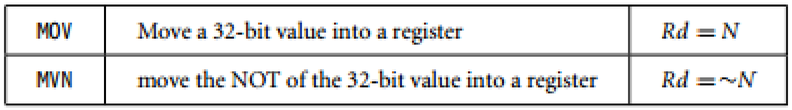
Data Processing Instructions

The data processing instructions manipulate data within registers. They are move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions. Most data processing instructions can process one of their operands using the barrel shifter.

MOV Instructions

Move is the simplest ARM instruction. It copies N into a destination register Rd, where N is a register or immediate value. This instruction is useful for setting initial values and transferring 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	$Rd = \sim N$

Example: This example shows a simple move instruction. The MOV instruction takes the contents of register r5 and copies them into register r7, in this case, taking the value 5, and overwriting the value 8 in register r7.

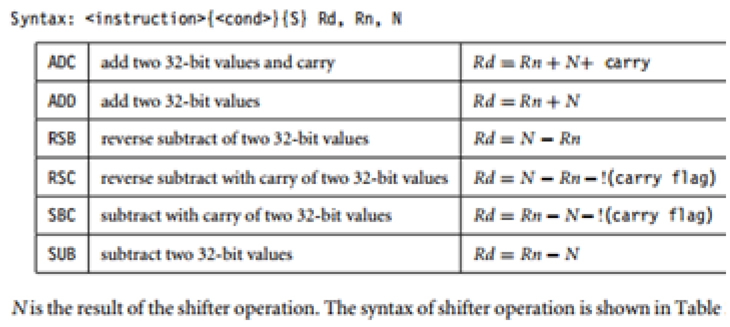
```
PRE    r5 = 5  r7 = 8
       MOV r7, r5
POST   r5 = 5  r7 = 5
```

Barrel Shifter

In the above example, we showed a MOV instruction where N is a simple register. But 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. Data processing instructions are processed within the arithmetic logic unit (ALU). 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. This shift increases the power and flexibility of many data processing operations.

Arithmetic Instructions

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



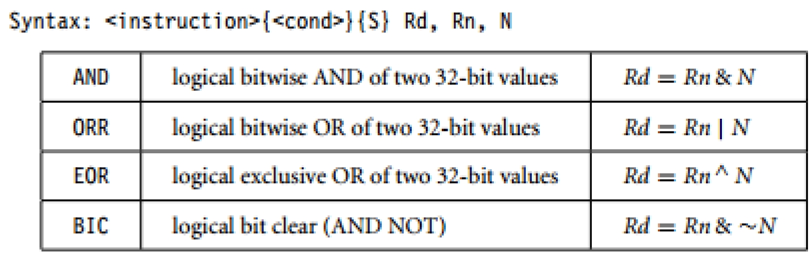
Syntax: <instruction>{<cond>}{S} Rd, Rn, N

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	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation. The syntax of shifter operation is shown in Table

Logical Instructions

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

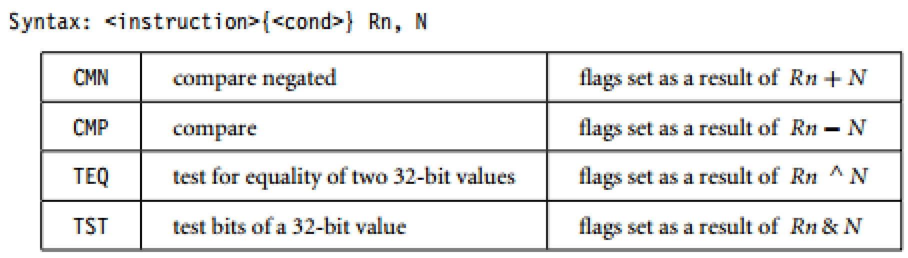


Syntax: <instruction>{<cond>}{S} Rd, Rn, N

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)	$Rd = Rn \& \sim N$

Comparison Instructions

The comparison instructions are used to compare or test a register with 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 by using conditional execution. For more information on conditional execution take a look at Section 3.8. You do not need to apply the S suffix for comparison instructions to update the flags.



Syntax: <instruction>{<cond>} Rn, N

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

Multiply Instructions

The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register. The long multiplies accumulate onto a pair of registers representing a 64-bit value. The final result is placed in a destination register or a pair of registers.

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

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

```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

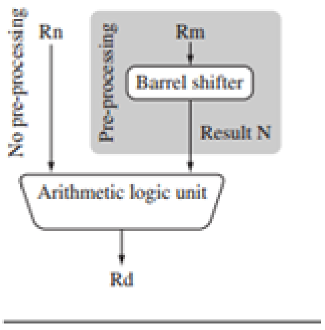
2) What is a barrel shifter? Which are the different barrel shifter operations?

Barrel Shifter

In MOV instruction, the second operand N can be more than just a register or immediate value; it can also be a register Rm that has been preprocessed by the barrel shifter prior to being used by a data processing instruction. Data processing instructions are processed within the arithmetic logic unit (ALU). 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. This shift increases the power and flexibility of many data processing operations.

There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.

Pre-processing or shift occurs within the cycle time of the instruction. This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.



Barrel shifter and ALU.

Example

We apply a logical shift left (LSL) to register Rm before moving it to the destination register. This is the same as applying the standard C language shift operator to the register. The MOV instruction copies the shift operator result N into register Rd. N represents the result of the LSL operation described in Table.

```
PRE   r5 = 5
      r7 = 8
      MOV r7, r5, LSL #2              ; r7 = r5*4 = (r5 << 2)
POST  r5 = 5
      r7 = 20
```

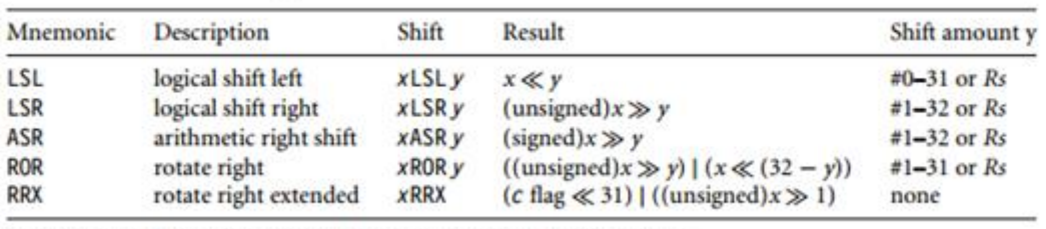
The example multiplies register r5 by four and then places the result into register r7.

3) Tabulate barrel shift operation syntax for data processing instructions.

The five different shift operations that you can use within the barrel shifter are summarized in Table.

It illustrates a logical shift left by one. For example, the contents of bit 0 are shifted to bit 1. Bit 0 is cleared. The C flag is updated with the last bit shifted out of the register. This is bit (32 − y) of the original value, where y is the shift amount. When y is greater than one, then a shift by y positions is the same as a shift by one position executed y times.

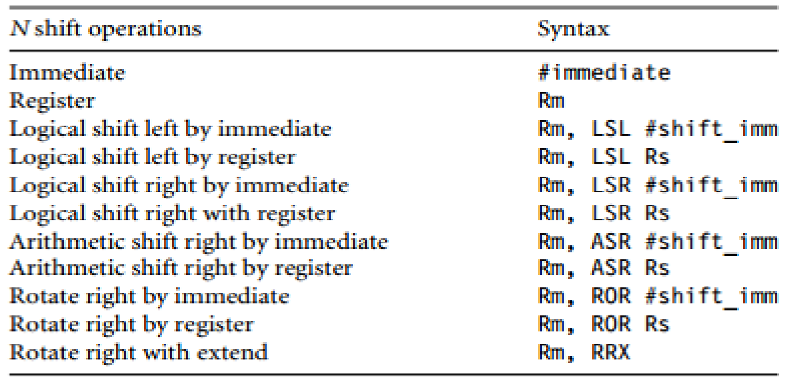
Barrel shifter operations.



Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	(unsigned)$x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	(signed)$x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	(*c* flag $\ll$ 31) | ((unsigned)$x \gg 1$)	none

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

Barrel shift operation syntax for data processing instructions.



N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

Example

This example of a MOVS instruction shifts register r1 left by one bit. This multiplies register r1 by a value 21. As you can see, the C flag is updated in the cpsr because the S suffix is present in the instruction mnemonic.

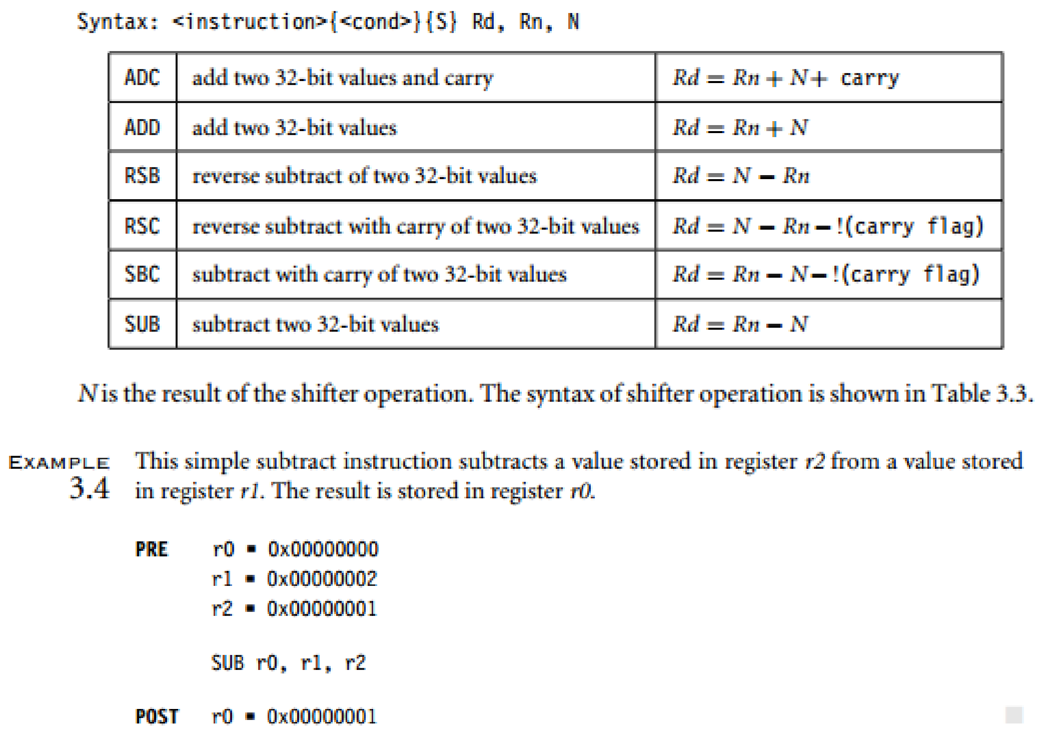
```
PRE cpsr = nzcvqiFt_USER
r0 = 0x00000000
r1 = 0x80000004

MOVS r0, r1, LSL #1

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

Table lists the syntax for the different barrel shift operations available on data processing instructions. The second operand N can be an immediate constant preceded by#, a register value Rm, or the value of Rm processed by a shift.

4) Explain in detail Arithmetic instructions. How Barrel shifter is used with Arithmetic instructions?

Arithmetic Instructions

The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned 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	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

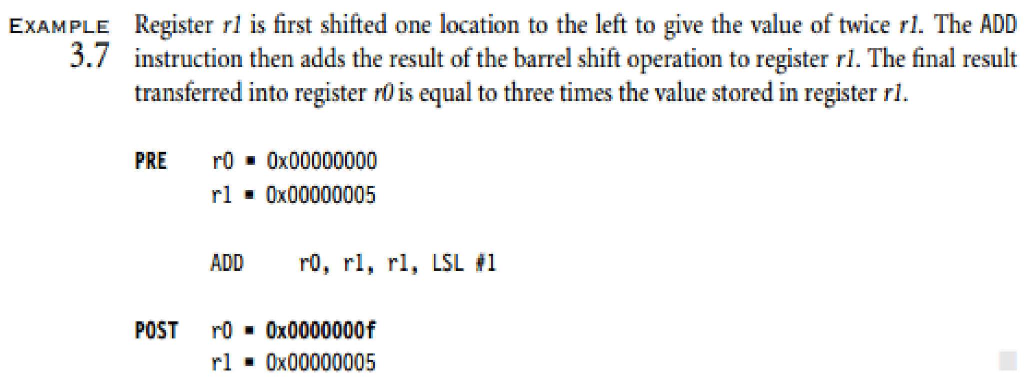
N is the result of the shifter operation. The syntax of shifter operation is shown in Table 3.3.

EXAMPLE 3.4 This simple subtract instruction subtracts a value stored in register *r2* from a value stored in register *r1*. The result is stored in register *r0*.

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

        SUB r0, r1, r2

POST    r0 = 0x00000001
```

Using the Barrel Shifter with Arithmetic Instructions

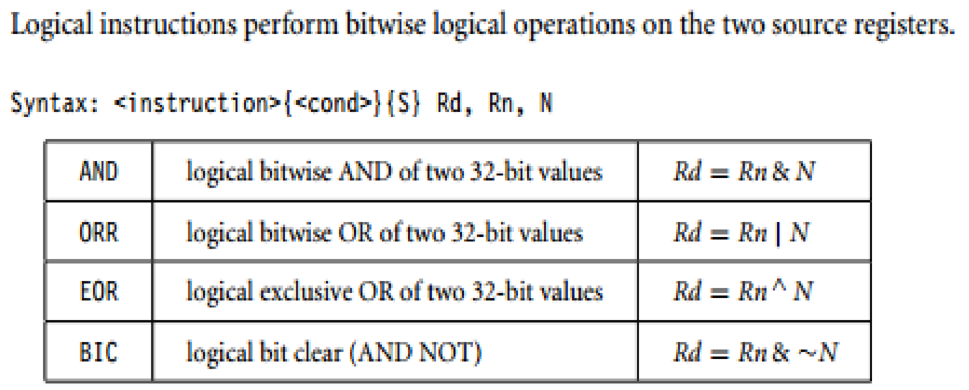
The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set. Example 3.7 illustrates the use of the inline barrel shifter with an arithmetic instruction. The instruction multiplies the value stored in register r1 by three.



EXAMPLE 3.7 Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

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

        ADD    r0, r1, r1, LSL #1

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

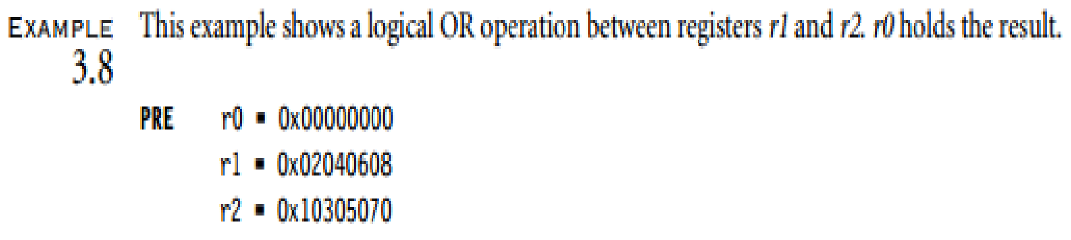
5) Along with suitable examples describe various logical (AND, ORR, EOR, BIC) and comparison instructions (CMN, CMP, TEQ, TST).



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

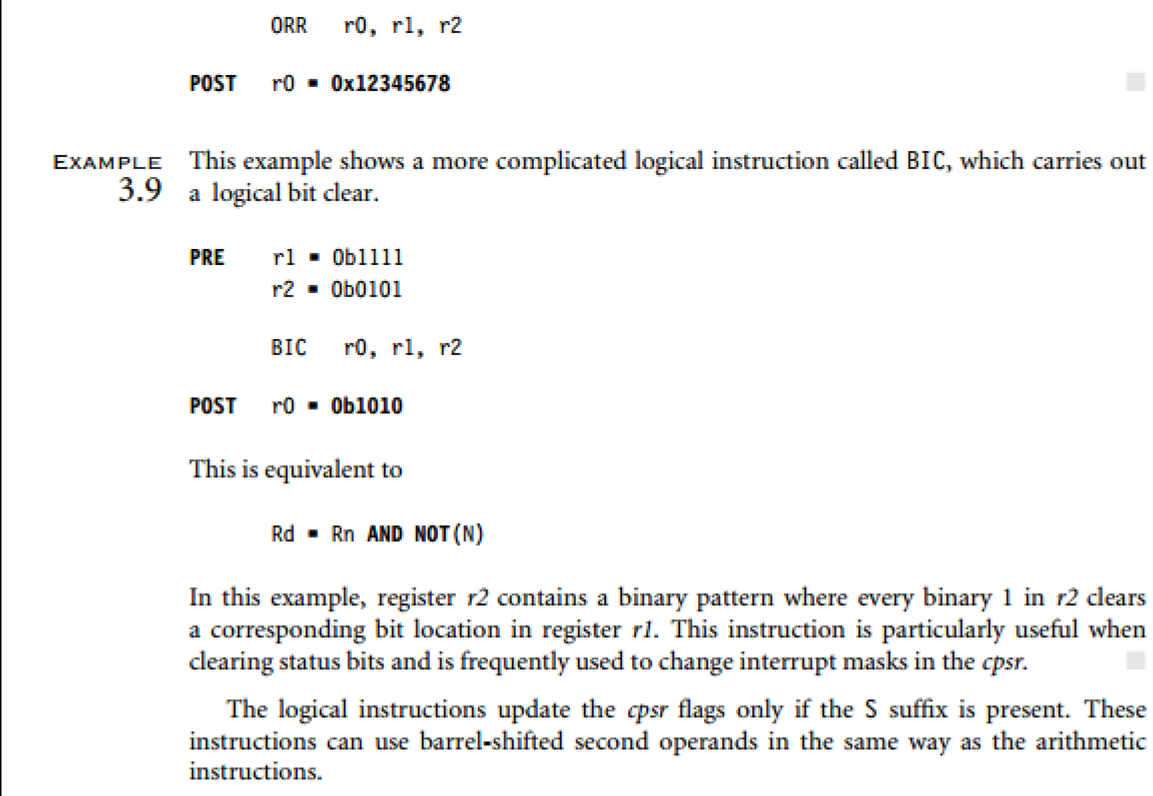
Syntax: `<instruction>{<cond>}{S} Rd, Rn, N`

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)	$Rd = Rn \& \sim N$



EXAMPLE 3.8 This example shows a logical OR operation between registers *r1* and *r2*. *r0* holds the result.

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



```
        ORR   r0, r1, r2

POST    r0 = 0x12345678
```

Example 3.9 This example shows a more complicated logical instruction called BIC, which carries out a logical bit clear.

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

        BIC   r0, r1, r2

POST    r0 = 0b1010
```

This is equivalent to

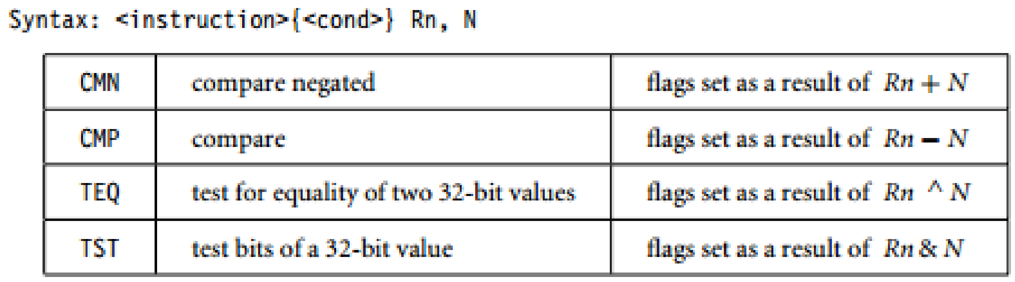
```
        Rd = Rn AND NOT(N)
```

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*.

The logical instructions update the *cpsr* flags only if the S suffix is present. These instructions can use barrel-shifted second operands in the same way as the arithmetic instructions.

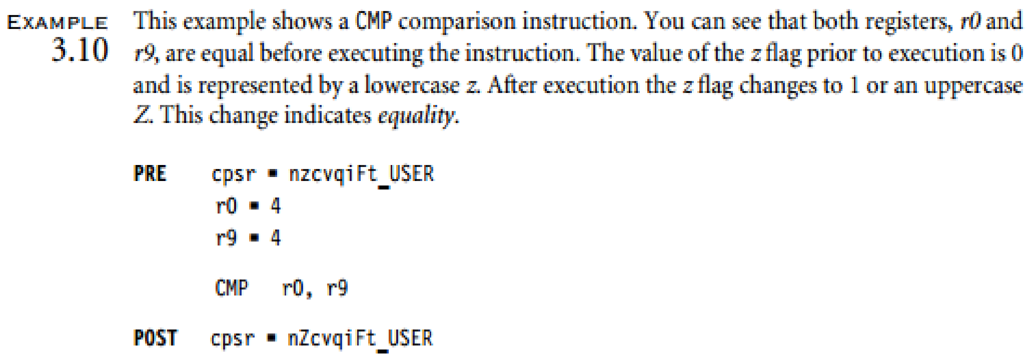
Comparison Instructions

The comparison instructions are used to compare or test a register with a 32-bit value.



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

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of Rn ^ N
TST	test bits of a 32-bit value	flags set as a result of Rn & N



Example 3.10 This example shows a CMP comparison instruction. You can see that both registers, *r0* and *r9*, are equal before executing the instruction. The value of the *z* flag prior to execution is 0 and is represented by a lowercase *z*. After execution the *z* flag changes to 1 or an uppercase *Z*. This change indicates *equality*.

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

        CMP   r0, r9

POST    cpsr = nZcvqiFt_USER
```

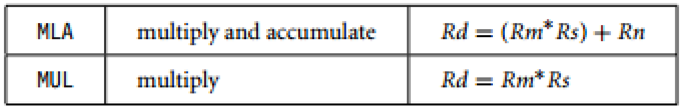
6) With example illustrate how following instructions work

i) MLA
ii) MUL
iii) SMLAL
iv) SMULL
v) UMLAL
vi) UMULL

Multiply Instructions

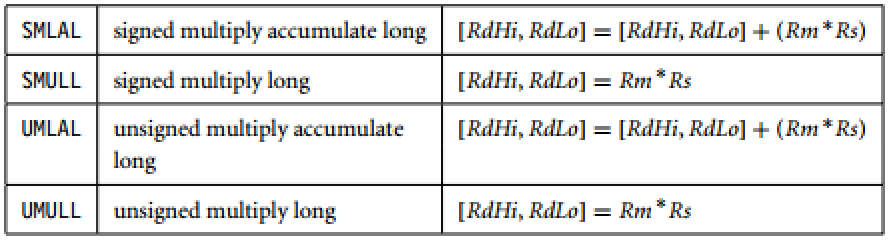
The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register. The long multiplies accumulate onto a pair of registers representing a 64-bit value. The final result is placed in a destination register or a pair of registers.

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

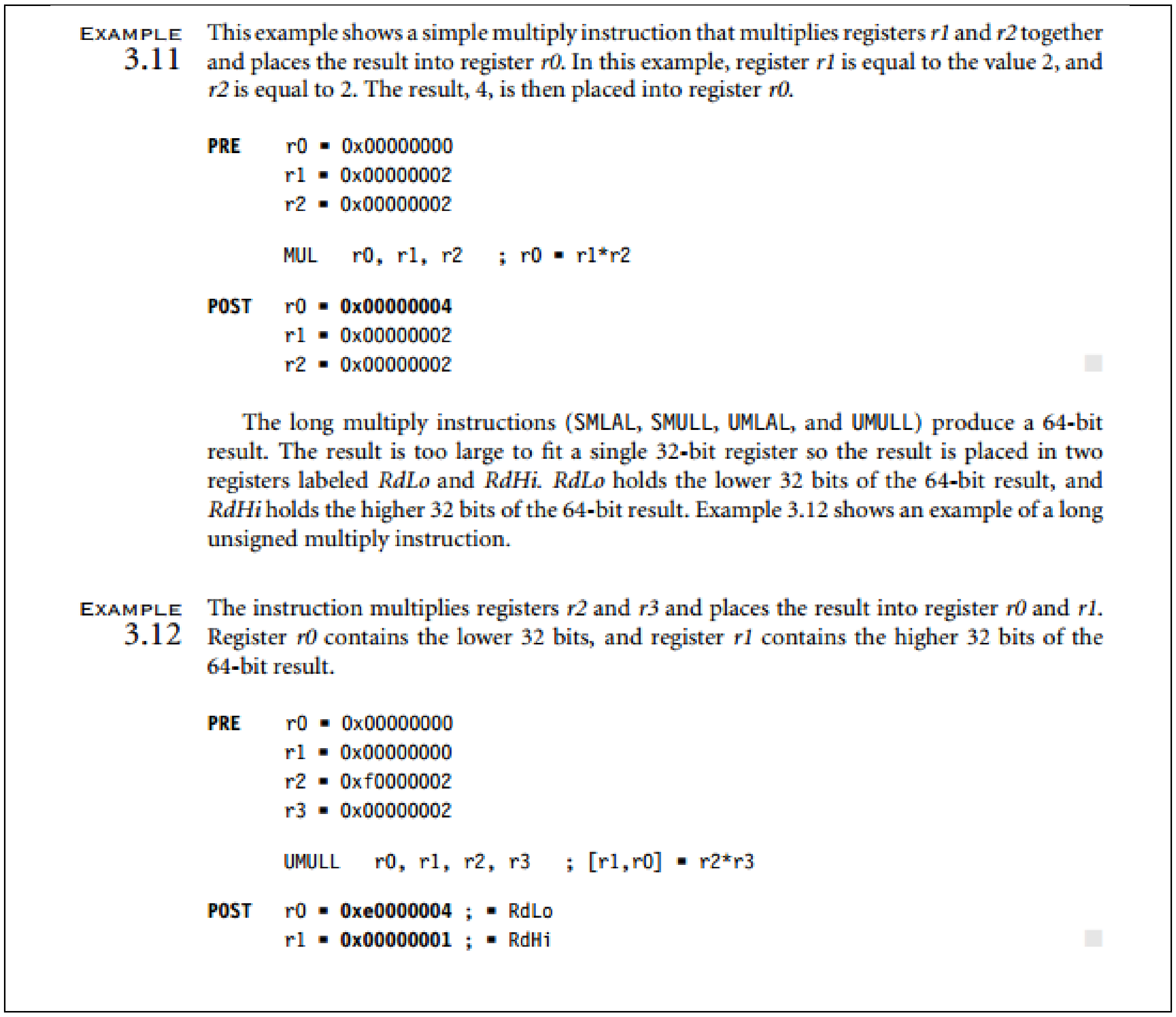


MLA	multiply and accumulate	$Rd = (Rm^* Rs) + Rn$
MUL	multiply	$Rd = Rm^* Rs$

```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```



SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm^* Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm^* Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm^* Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm^* Rs$



EXAMPLE 3.11 This example shows a simple multiply instruction that multiplies registers *r1* and *r2* together and places the result into register *r0*. In this example, register *r1* is equal to the value 2, and *r2* is equal to 2. The result, 4, is then placed into register *r0*.

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

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

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

The long multiply instructions (SMLAL, SMULL, UMLAL, and UMULL) produce a 64-bit result. The result is too large to fit a single 32-bit register so the result is placed in two registers labeled *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. Example 3.12 shows an example of a long unsigned multiply instruction.

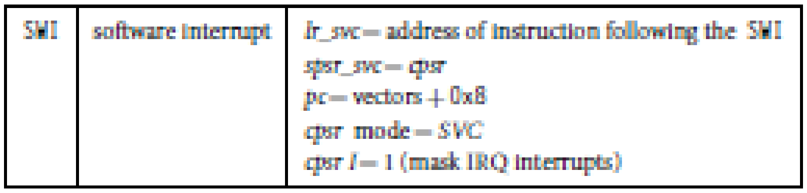
EXAMPLE 3.12 The instruction multiplies registers *r2* and *r3* and places the result into register *r0* and *r1*. Register *r0* contains the lower 32 bits, and register *r1* contains the higher 32 bits of the 64-bit result.

```
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
```

7) Explain software interrupt instructions (SWI)

A software interrupt instruction (SWI) causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.

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)

When the processor executes an SWI instruction, it sets the program counter *pc* to the offset 0x8 in the vector table. The instruction also forces the processor mode to *SVC*, which allows an operating system routine to be called in a privileged mode.

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

Example:
Here we have a simple example of an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

```
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
```

Since SWI instructions are used to call operating system routines, you need some form of parameter passing. This is achieved using registers. In this example, register *r0* is used to pass the parameter 0x12. The return values are also passed back via registers.

Code called the *SWI handler* is required to process the SWI call. The handler obtains the SWI number using the address of the executed instruction, which is calculated from the link register *lr*.

The SWI number is determined by SWI_Number = <SWI instruction> **AND NOT**(0xff000000). Here the *SWI instruction* is the actual 32-bit SWI instruction executed by the processor.

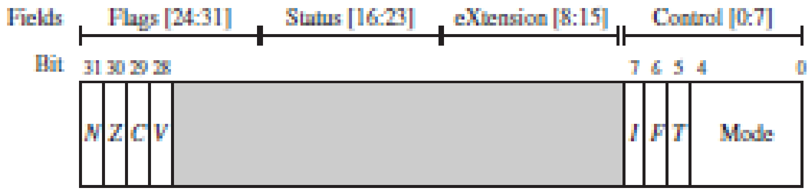
8) Explain program status register byte fields and explain – MRS & MSR.

The ARM instruction set provides two instructions to directly control a program status register (psr). The **MRS** instruction transfers the contents of either the cpsr or spsr into a register; in the reverse direction, the **MSR** instruction transfers the contents of a register into the cpsr or spsr. Together these instructions are used to read and write the cpsr and spsr.

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*

In the syntax you can see a label called *fields*. This can be any combination of control (*c*), extension (*x*), status (*s*), and flags (*f*). These fields relate to particular byte regions in a *psr*, as shown in Figure.

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



The *c* field controls the interrupt masks, Thumb state, and processor mode.

The following **Example** shows how to enable IRQ interrupts by clearing the *I* mask. This operation involves using both the MRS and MSR instructions to read from and then write to the *cpsr*.

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. You can see from this example that this code preserves all the other settings in the *cpsr* and only modifies the *I* bit in the control field.

```
PRE cpsr = nzcvqIFt_SVC
        MRS r1, cpsr
        BIC r1, r1, #0x80 ; 0b01000000

MSR cpsr_c, r1

POST cpsr = nzcvqiFt_SVC
```

This example is in *SVC* mode. In *user* mode you can read all *cpsr* bits, but you can only update the condition flag field *f*.

9) Explain Branch Instructions with examples.

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

The ARMv5E instruction set includes four different branch instructions.

Syntax: B{<cond>} label
BL{<cond>} label
BX{<cond>} Rm
BLX{<cond>} label | 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* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

The address *label* is stored in the instruction as a signed *pc*-relative offset and must be within approximately 32 MB of the branch instruction. *T* refers to the Thumb bit in the *cpsr*. When instructions set *T*, the ARM switches to Thumb state.

Example:
This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. 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
```

Branches are used to change execution flow. Most assemblers hide the details of a branch instruction encoding by using labels. In this example, *forward* and *backward* are the labels. The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset.

10) With example, explain the operation of four stack instructions.

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; in contrast, descending stacks grow towards lower memory addresses.

When you use a ***full stack (F),*** the stack pointer *sp* points to an address that is the last used or full location (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 empty location (i.e., it points after the last item on the stack).

There are a number of load-store multiple addressing mode aliases available to support stack operations (see Table). Next to the ***pop*** column is the actual load multiple instruction equivalents. For example, a full ascending stack would have the notation ***FA*** appended to the load multiple instruction—***LDMFA***. This would be translated into an ***LDMDA*** instruction.

ARM has specified an ARM-Thumb Procedure Call Standard (***ATPCS***) that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the ***LDMFD*** and ***STMFD*** instructions provide the pop and push functions, respectively.

<u>Addressing Modes for Stack Operations</u>

Addressing methods for stack operations.

Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

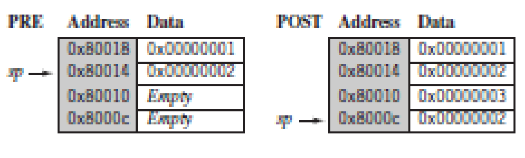
Example:

The STMFD instruction pushes registers onto the stack, updating the *sp*. Figure shows a push onto a full descending stack. You can see that when the stack grows the stack pointer points to the last full entry in the stack.

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

STMFD sp!, {r1,r4}

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



STMFD instruction—full stack push operation.

11) Explain Load-Store Instructions.

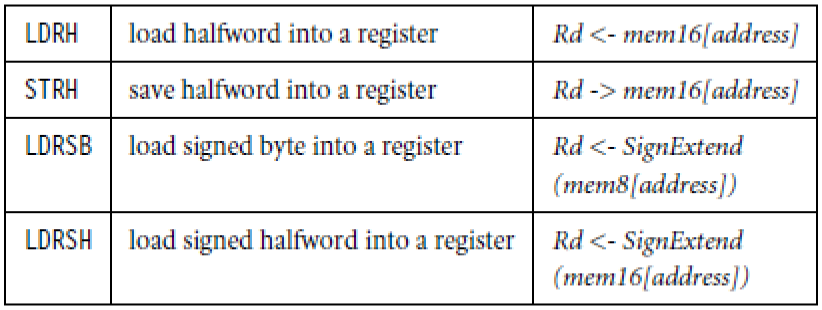
Load-Store Instructions:
Load-store instructions transfer data between memory and processor registers. ***There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.***

1. Single - Register Transfer

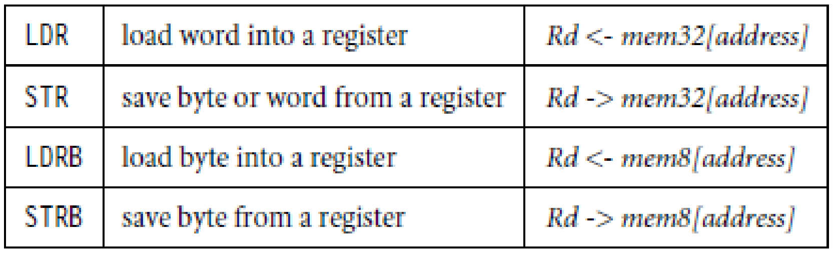
These instructions are used for moving a single data item in and out of a register. The data types supported are signed and unsigned words (32-bit), halfwords (16-bit), and bytes.

Here are the various load-store single-register transfer instructions

Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
LDR{<cond>}SB|H|SH Rd, addressing2
STR{<cond>}H Rd, addressing2



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])*



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]*

LDR and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored. For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on. This example shows a load from a memory address contained in register *r2*, followed by a store back to the same address in memory.

load register r0 with the contents of ; the memory address pointed to by register r2.

LDR r0, [r2] ; = LDR r0, [r2, #0]

store the contents of register r0 to the memory address pointed to by register r2.

STR r0, [r2] ; = STR r0, [r2, #0]

The first instruction loads a word from the address stored in register *r2* and places it into register *r0*. The second instruction goes the other way by storing the contents of register *r0* to the address contained in register *r2*. The offset from register *r2* is zero. Register *r2* is called the *base address register*.

2. Multiple - Register Transfer

Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction. The transfer occurs from a base address register *Rn* pointing in to memory. Multiple-register transfer instructions are more efficient from single-register transfers for moving blocks of data around memory and saving and restoring context and stacks.

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

3. SWAP instructions.

The swap instruction is a special case of a load-store instruction. **It swaps the contents of memory with the contents of a register**. This instruction is an *atomic operation*—it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]

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:
The swap instruction loads a word from memory into register *r0* and overwrites the memory with register *r1*.

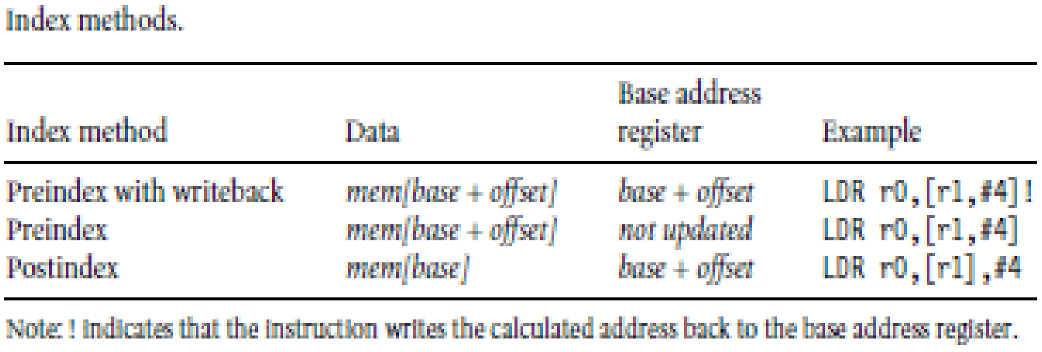
PRE mem32[0x9000] = 0x12345678
r0 = 0x00000000
r1 = 0x11112222
r2 = 0x00009000

SWP r0, r1, [r2]

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

12) Explain Single - Register Load-Store Addressing Modes.

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



Index methods.

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 and then updates that address base register with the new address.
  Example: **LDR r0, [r1, #4]!**

- In contrast, the **preindex** offset is the same as the preindex with writeback but does not update the address base register.
  Example: **LDR r0, [r1, #4]**

- **Postindex** only updates the address base register after the address is used.
  Example: **LDR r0, [r1], #4**

The preindex mode is useful for accessing an element in a data structure. The postindex and preindex with writeback modes are useful for traversing an array.

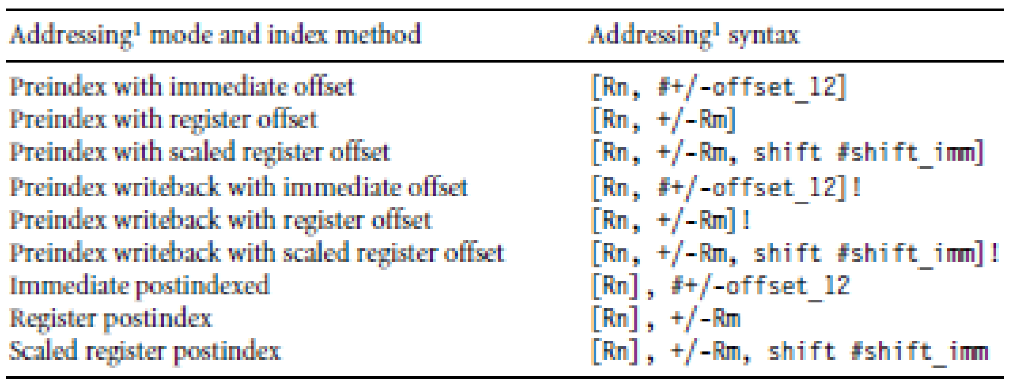
Detailed Explanation:

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

Preindexing with writeback	**Preindexing**	**Postindexing**
LDR r0, [r1, #4]**!**	LDR r0, [r1, #4]	LDR r0, [r1], #4
POST r0 = 0x02020202 r1 = 0x00009004 LDR r0, [r1, #4]	**POST** r0 = 0x02020202 r1 = 0x00009000	**POST** r0 = 0x01010101 r1 = 0x00009004

The following table shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.

Single-register load-store addressing, word or unsigned byte.



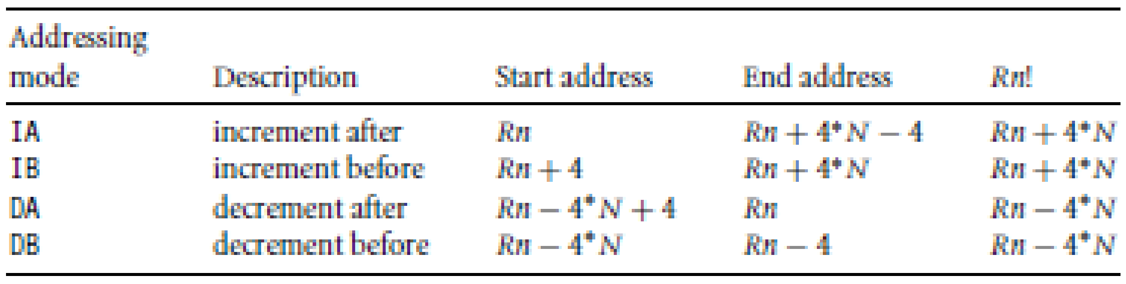
Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	`[Rn, #+/-offset_12]`
Preindex with register offset	`[Rn, +/-Rm]`
Preindex with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]`
Preindex writeback with immediate offset	`[Rn, #+/-offset_12]!`
Preindex writeback with register offset	`[Rn, +/-Rm]!`
Preindex writeback with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]!`
Immediate postindexed	`[Rn], #+/-offset_12`
Register postindex	`[Rn], +/-Rm`
Scaled register postindex	`[Rn], +/-Rm, shift #shift_imm`

13) Explain Multiple - Register Load-Store Addressing Modes.

Table shows the different addressing modes for the load-store multiple instructions. Here *N* is the number of registers in the list of registers.

Any subset of the current bank of registers can be transferred to memory or fetched from memory. The base register *Rn* determines the source or destination address for a load store multiple instruction. This register can be optionally updated following the transfer. This occurs when register *Rn* is followed by the ! character, similiar to the single-register load-store using preindex with writeback.

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	$Rn - 4*N$	$Rn - 4$	$Rn - 4*N$

Example:

In this example, 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.

```
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
```

14) **Explain the operation of this instruction: LDR r0, [r1], r2, LSR#04 indicate the pre and post conditions of memory and register.**

Basics: (refer previous question)

Now, let's understand - **LDR r0, [r1], r2, LSR#04**

This uses a postindex addressing method.

Steps:
1. go to r1's location, take whatever value of that location and put to r0.
2. update r1 as r1 = r1 + (r2 >> 04) → **LSR means Logical Shift Right**

Example:

PRE r0 = 0x00000000
r1 = 0x00009000
mem32[0x00009000] = 0x12121212
r2 = 0x00000064

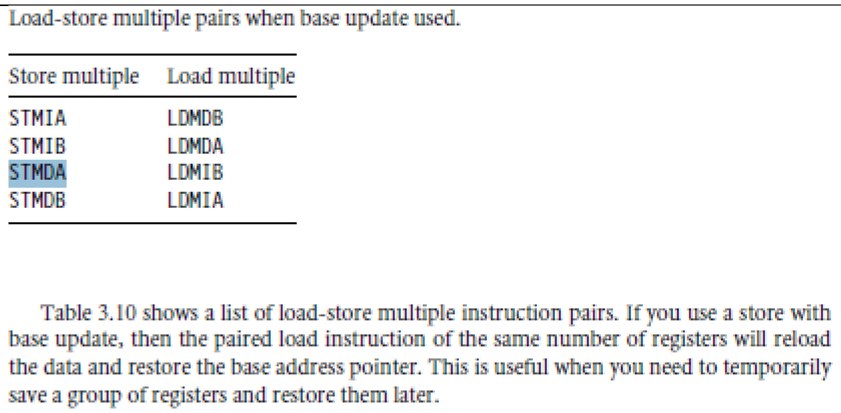
LDR r0, [r1], r2, LSR#04

POST r0 = 0x12121212
r1 = 0x00009004 (r1 = r1 + (64>>4))
=>r1 = 0x00009000 + 4
=>r1 = 0x00009004

15) With example illustrate how following instructions work

i) LDRSB
ii) LDRSH
iii) STRB
iv) STRH
v) LDMIA
vi) LDMDA
vii) STMIA
viii) STMDA

LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*
STRB	save byte from a register	*Rd -> mem8[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDMIA	load multiple and Increment After	
LDMDA	load multiple and Decrement After	
STMIA	Save after Increment	
STMDA	Save after Decrement	


Load-store multiple pairs when base update used.

Store multiple	Load multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

Table 3.10 shows a list of load-store multiple instruction pairs. If you use a store with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer. This is useful when you need to temporarily save a group of registers and restore them later.

Examples:

This example shows an STM *increment before* instruction followed by an LDM *decrement after* instruction.

PRE r0 = 0x00009000
r1 = 0x00000009
r2 = 0x00000008
r3 = 0x00000007

STMIB r0!, {r1-r3}

MOV r1, #1
MOV r2, #2
MOV r3, #3

PRE(2) r0 = 0x0000900c
r1 = 0x00000001
r2 = 0x00000002
r3 = 0x00000003

LDMDA r0!, {r1-r3}

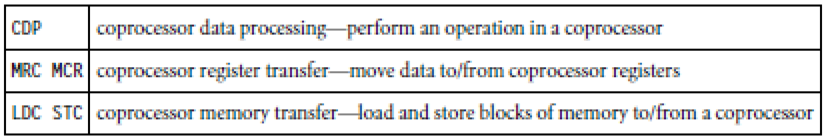
POST r0 = 0x00009000
r1 = 0x00000009
r2 = 0x00000008
r3 = 0x00000007
The STMIB instruction stores the values 7, 8, 9 to memory. We then corrupt register *r1* to *r3*.
The LDMDA reloads the original values and restores the base pointer *r0*.

16) Which are the different coprocessor instructions?

Coprocessor instructions are used to extend the instruction set. A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management. The coprocessor instructions include data processing, register transfer, and memory transfer instructions. Wewill provide only a short overview since these instructions are coprocessor specific. Note that these instructions are only used by cores with a coprocessor.

Syntax:

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



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

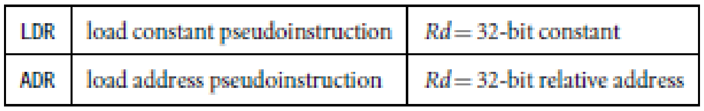
In the syntax of the coprocessor instructions, the *cp* field represents the coprocessor number between *p0* and *p15*. The *opcode* fields describe the operation to take place on the coprocessor. The *Cn*, *Cm*, and *Cd* fields describe registers within the coprocessor.

The coprocessor operations and registers depend on the specific coprocessor you are using. Coprocessor 15 (CP15) is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

17) Explain how a 32-bit constant can be loaded in a register.

We might have noticed that there is no ARM instruction to move a 32-bit constant into a register. SinceARMinstructions are 32 bits in size, they obviously cannot specify a general 32-bit constant.

To aid programming there are two pseudo instructions to move a 32-bit value into a register.
Syntax:
LDR Rd, =constant
ADR Rd, label

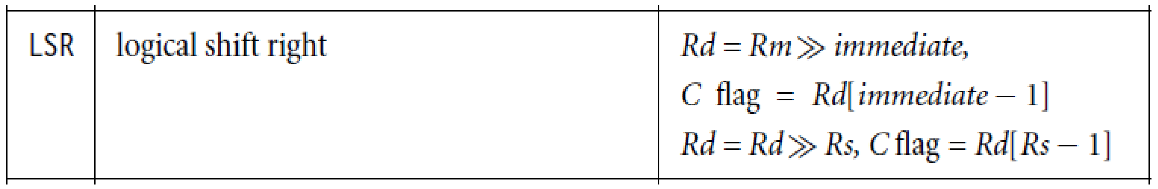


LDR	load constant pseudoinstruction	*Rd*= 32-bit constant
ADR	load address pseudoinstruction	*Rd*= 32-bit relative address

The first pseudo instruction writes a 32-bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions.

The second pseudo instruction writes a relative address into a register, which will be encoded using a *pc*-relative expression.

18) Illustrate with a net diagram Logical shift left operation.



LSR	logical shift right	$Rd = Rm \gg immediate,$ C flag $= Rd[immediate - 1]$ $Rd = Rd \gg Rs$, C flag $= Rd[Rs - 1]$

NOTE: REFER PREVIOUS EXAMPLES FOR MORE INFORMATION ON LSR.