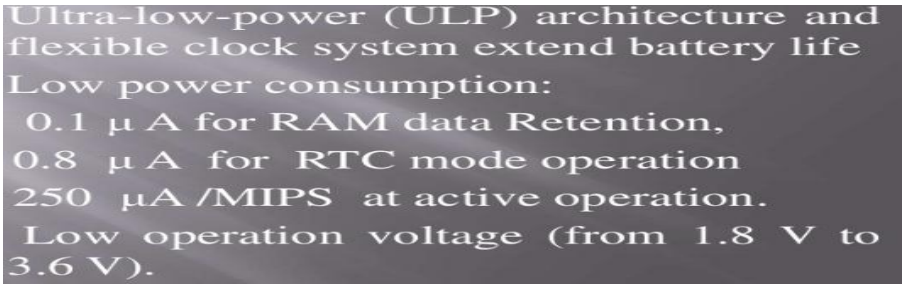
PAPER SOLUTION FOR Q.P.CODE : 22565

Q1) A)

-	RISC	CISC
1	RISC stands for Reduced Instruction Set Computer	CISC stands for Complex Instruction Set Computer
2	RISC processors have simple instructions taking about one clock cycle. The average Clock cycles Per Instruction(CPI) of a RISC processor is 1.5	CISC processors have complex instructions that take up multiple clock cycles for execution. The average Clock cycles Per Instruction of a CISC processor is between 2 and 15
3	There are hardly any instructions that refer memory.	Most of the instructions refer memory
4	RISC processors have a fixed instruction format	CISC processors have variable instruction format.
5	The instruction set is reduced i.e. it has only few instructions in the instruction set. Many of these instructions are very primitive.	The instruction set has a variety of different instructions that can be used for complex operations.
6	RISC has fewer addressing modes and most of the instructions in the instruction set have register to register addressing mode.	CISC has many different addressing modes and can thus be used to represent higher level programming language statements more efficiently.
7	Complex addressing modes are synthesized using software.	CISC already supports complex addressing modes

Q1) B)



Q1) C)

Macros. A **macro** is an extension to the basic **ASSEMBLER language**. They provide a means for generating a commonly used sequence of **assembler**instructions/statements. The sequence of instructions/statements will be coded ONE time within the **macro** definition.

SYNTAX: **MACRO**{

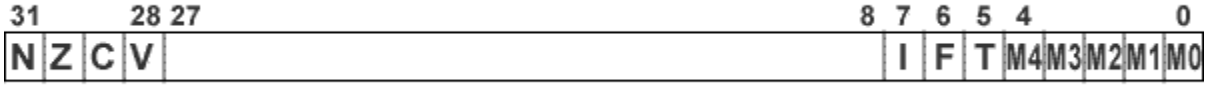
}

ENDM

Q1) D)

The Current Program Status Register is a 32-bit wide register used in the ARM architecture to record various pieces of information regarding the state of the program being executed by the processor and the state of the processor. This information is recorded by setting or clearing specific bits in the register.

ARM CPSR format



The top four bits (bits 31, 30, 29, and 28) are the condition code (**cc**) bits and are of most interest to us. Condition code bits are sometimes referred to as "flags". The lowest 8 bits (bit 7 through to bit 0) store information about the processor's own state. The remaining bits (i.e. bit 27 to bit 8) are currently unused in most ARM processors.

The N bit is the "negative flag" and indicates that a value is negative.
The Z bit is the "zero flag" and is set when an appropriate instruction produces a zero result.
The C bit is the "carry flag" but it can also be used to indicate "borrows" (from subtraction operations) and "extends" (from shift instructions (LINK)).
The V bit is the "overflow flag" which is set if an instruction produces a result that overflows and hence may go beyond the range of numbers that can be represented in 2's complement signed format.

For completeness, the other state bits are:

The I and F bits which determine whether interrupts (such as requests for input/output) are enabled or disabled.
The T bit which indicates whether the processor is in "Thumb" mode, where the processor can execute a subset of the assembly language as 16-bit compact instructions. As Thumb code packs more instructions into the same amount of memory, it is an effective solution to applications where physical memory is at a premium.
The M4 to M0 bits are the mode bits. Application programs normally run in user mode (where the mode bits are 10000). Whenever an interrupt or similar event occurs, the processor switches into one of the alternative modes allowing the software handler greater privileges with regard to memory manipulation.

Q1)E) EA

The 8051 family members, such as the 8751/52, 89C51/52, or DS89C4xO, all come with on-chip ROM to store programs. In such cases, the EA pin is connected to V_{cc}. For family members such as the 8031 and 8032 in which there is no on-chip ROM. code is stored on an external ROM and is fetched by the 8031/32. Therefore, for the 8031 the EA pin must be connected to GND to indicate that the code is stored externally. EA. which stands for “external access,” is pin number

31 in the DIP packages. It is an input pin and must be connected to either V_{cc} or GND. In other words, it cannot be left unconnected.

the 8031 uses this pin along with PSEN to access programs stored in ROM memory located outside the 8031. In 8051 chips with on-chip ROM, such as the 8751/52, 89C51/52, or DS89C4xO, EA is connected to V_{cc},

PSEN

This is an output pin. PSEN stands for "program store enable." In an 8031-based system in which an external ROM holds the program code, this pin is connected to the OE pin of the ROM.

ALE

ALE (address latch enable) is an output pin and is active high. When connecting an 8031 to external memory, port 0 provides both address and data. In other words, the 8031 multiplexes address and data through port 0 to save pins. The ALE pin is used for demultiplexing the address and data by connecting to the G pin of the 74LS373 chip.

Q2) A)

SCON : Serial Port Control Register (Bit Addressable)

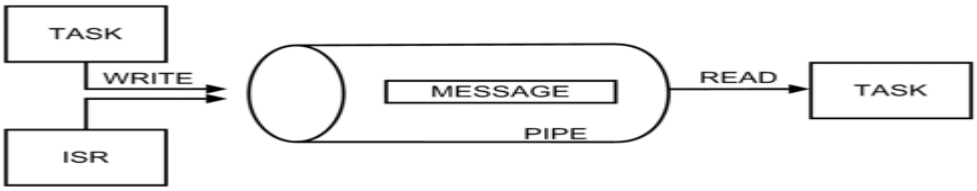
SM0	SM1	SM2	REN	TN8	RB8	TI	RI

SM0	SCON.7	Serial Port mode specifier (NOTE 1).
SM1	SCON.6	Serial Port mode specifier (NOTE 1).
SM2	SCON.5	Enables the multiprocessor communication feature in mode 2 & 3. In mode 2 or 3, if SM2 is set to 1 then RI will not be activated if the received 9th data bit (RB8) is 0. In mode 1, if SM2 = 1 then RI will not be activated if a valid stop bit was not received. In mode 0, SM2 should be 0 (See table 9).
REN	SCON.4	Set/Cleared by software to Enable/Disable reception.
TB8	SCON.3	The 9th bit that will be transmitted in modes 2 & 3. Set/Cleared by software.
RB8	SCON.2	In modes 2 & 3, is the 9th data bit that was received. In mode 1, if SM2 = 0, RB8 is the stop bit that was received. In mode 0, RB8 is not used.
TI	SCON.1	Transmit interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or at the beginning of the stop bit in the other modes. Must be cleared by software.
RI	SCON.0	Receive interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or half way through the stop bit time in the other modes (except see SM2). Must be cleared by software.

Note 1 :

SM0	SM1	MODE	DESCRIPTION	BAUD RATE
0	0	0	SHIFT REGISTER	Fosc./12
0	1	1	8 bit UART	Variable
1	0	2	8 bit UART	Fosc./64 OR Fosc./32
1	1	3	8 bit UART	Variable

Q2) B) ***Pipe:***



1. A message pipe is an inter-process task communication tool used for inserting, deleting messages between two given interconnected task or two sets of tasks.
2. In a pipe there are no fixed numbers of bytes per message but there is an end-point. A pipe can therefore be inserted limited number of bytes and have a variable number of bytes per message between initial and final pointers.
3. Pipes are unidirectional i.e. one thread or task inserts into it and the other one reads and deletes from it.
4. The read method of a pipe has no knowledge of where the write started, the interacting tasks must therefore share start and end locations as the messages are inserted into the pipe.

Mailbox:

1. A message mailbox, also called a message exchange, is typically a pointer-size variable. Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into this mailbox.
2. One or more tasks can receive messages through a service provided by the kernel.
3. Both the sender and receiving task agree on what the pointer is actually pointing to.
4. A waiting list is associated with each mailbox in case more than one task wants to receive messages through the mailbox. A task desiring a message from an empty mailbox is suspended and placed on the waiting list until a message is received
5. Typically, the kernel allows the task waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting task is made ready to run, and an error code (indicating that a timeout has occurred) is returned to it
6. When a message is deposited into the mailbox, either the highest priority task waiting for the message is given the message (priority-based), or the first task to request a message is given the message (First-In First-Out, or FIFO).

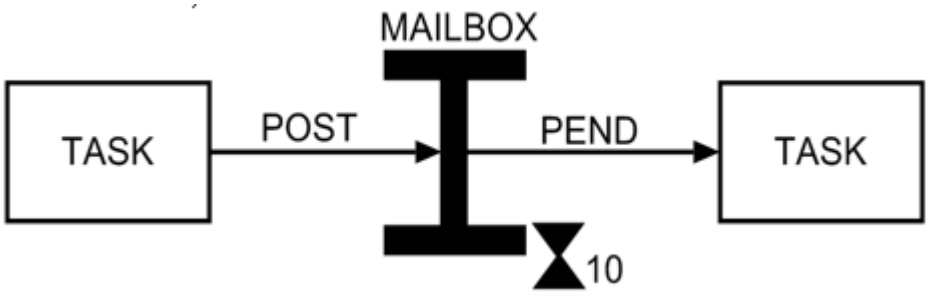


Figure shows a task depositing a message into a mailbox. Note that the mailbox is represented by an I-beam and the timeout is represented by an hourglass. The number next to the hourglass represents the number of clock ticks the task will wait for a message to arrive.

Message queue:

1. A message queue is used to send one or more messages to a task. A message queue is basically an array of mailboxes. Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into a message queue.
2. One or more tasks can receive messages through a service provided by the kernel. Both the sender and receiving task or tasks have to agree as to what the pointer is actually pointing to.
3. The first message inserted in the queue is the first message extracted from the queue (FIFO). In addition, to extract messages in a FIFO fashion, RTOSes allows a task to get messages Last-In-First-Out (LIFO).
4. A waiting list is associated with each message queue, in case more than one task is to receive messages through the queue.
5. The kernel allows the task waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting task is made ready to run, and an error code (indicating a timeout has occurred) is returned to it.
6. A task desiring a message from an empty queue is suspended and placed on the waiting list until a message is received.
7. When a message is deposited into the queue, either the highest priority task, or the first task to wait for the message is given the message.

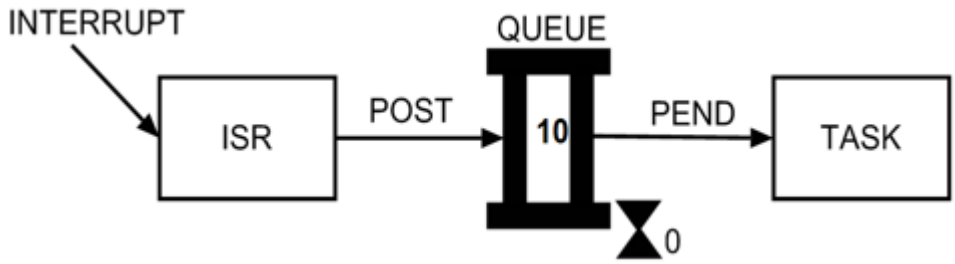


Figure shows an ISR depositing a message into a queue. Note that the queue is represented graphically by a double I-beam. The “10” indicates the number of messages that can accumulate in the queue. A “0” next to the hourglass indicates that the task will wait forever for a message to arrive.

Q3) A)

```
data segment
        num db -3,1,-5,6,-7,9,'#'
        p_cnt db 0h
        n_cnt db 0h

data ends


code segment
        assume ds:data,cs:code

start:
        mov ax,data
        mov ds,ax
```

```
        lea si,num
main:
        cmp num[si],0h
        jg pos
        inc si
        add n_cnt,01h
        cmp num[si],'#'
        je exit
        jmp main

pos:
        add p_cnt,01h
        inc si
        cmp num[si],'#'
        je exit
        jmp main

exit:
        mov bl,p_cnt
        mov cl,n_cnt

        mov ax,4c00h
        int 21h

code ends
        end start
```

Q3) B) Processor modes

There are seven processor modes:

Mode	Bits	Description	Family
User	usr %10000	Normal program execution, no privileges	All
FIQ	fiq %10001	Fast interrupt handling	All
IRQ	irq %10010	Normal interrupt handling	All
Supervisor	svc %10011	Privileged mode for the operating system	All
Abort	abt %10111	For virtual memory and memory protection	ARMv3+
Undefined	und %11011	Facilitates emulation of co-processors in hardware	ARMv3+
System	sys %11111	Runs programs with some privileges	ARMv4+

User mode

This is the mode in which user application tasks should run. It has access to the base register set, and no privileges.

FIQ mode

The ARM processor supports two types of interrupt handling. There is the *regular* type of interrupt, and there is this, the *fast interrupt*. The difference is that fast interrupts can interrupt regular ones.

FIQ mode provides a large number of shadow registers (R8 to R14, CPSR) and is useful for things that must complete extremely quickly or else data loss is a possibility. The original (8MHz) ARM used FIQ for networking and floppy disc which *had* to be serviced as soon as data was available. Modern ARMs would probably use FIQ for high speed DMA-style transfers.

IRQ mode

This is the other, regular, interrupt mode. Only R13, R14, and CPSR are shadowed. All interrupts that don't require extreme speed (clock ticks, screen VSync, keyboard, etc...) will use IRQ mode.

SVC mode

This is the privileged mode reserved for the operating system. R13, R14, and CPSR are shadowed.

OS calls (SWI) set the processor to SVC mode, and then the processor jumps to &8 (or &FFFF0008).

After system reset, the ARM begins processing at address &0 (or &FFFF0000 if high vectors configured), with interrupts disabled and in SVC mode. This address is the location of the Reset Vector, which should be a branch to the reset code.

Abort mode

An abort is signalled by the memory system as a result of a failure to load either an instruction (Prefetch Abort) or data (Data abort).

A **Prefetch Abort** occurs if the processor attempts to *execute* a failed instruction load (note - no abort happens if the processor fails to load an instruction, but said instruction is ***not*** executed due to a branch or suchlike).
In ARMv5 a Prefetch Abort can be generated programatically by the use of the breakpoint instruction.

A **Data Abort** occurs if the processor attempts to fetch data but the memory system says it is unable to. The abort occurs before the failed instruction alters the processor state.

Undefined mode

When an undefined instruction is encountered, the ARM will wait for a coprocessor to acknowledge that it can deal with the instruction (if in co-processor instruction space). If no coprocessor responds, or the instruction is one that is not defined, then the undefined instruction vector is taken. This will branch to &4 (or &FFFF0004) to allow such things as software emulation of coprocessors, or other extensions to the instruction set.

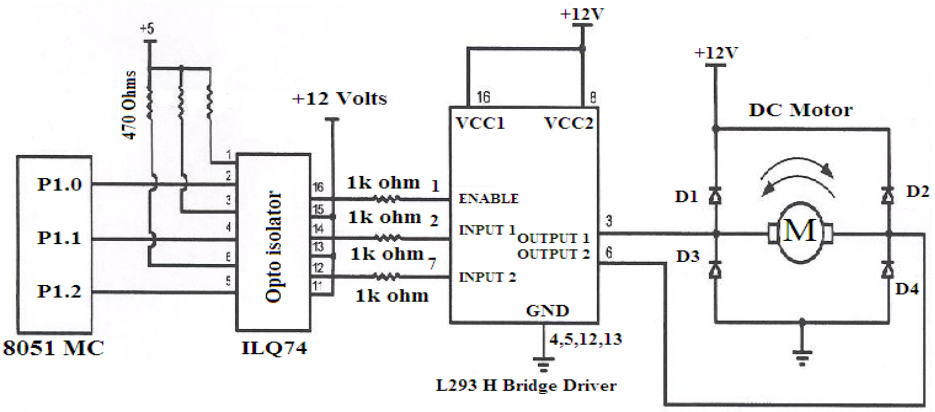
System mode

A problem with the original design of the ARM is that as processor vectors modify R14 with the return address, an exception handler (for example, *IRQ*) that calls subroutines *cannot* act in a re-entrant way; for if the IRQ handler is taken while *in* the IRQ handler and having called a subroutine to handle the (first) IRQ, the return address in R14 will be trashed by the second IRQ exception.

Q4) A)

INTERFACING DC MOTOR- 8051

A DC motor runs with the help of Direct Current. It produces torque by using both electricity and magnetic fields. The DC motor has rotor, stator, field magnet, brushes, shaft, commutator. The DC motor requires more current to produce initial torque than in running state.Interfacing the DC motor directly to 8051 microcontroller is not possible. Because the DC motor uses large current (200-300mA in small DC motors) to run. When this current flow into the 8051 microcontroller, the IC will get damaged. Therefore we use a driving circuit with an opto isolator and a L298 Dual H-Bridge driver. The opto-isolator provides additional protection to the microcontroller.



Continuous, sustained operation of the motor will cause the L293 Dual H-Bridge driver to overheat. So, a suitable heat sink must be used.

Assembly Language program

ORG	**0000H**		**Remarks**
MAIN	**CLR**	**P1.0**	
	CLR	**P1.1**	
	CLR	**P1.2**	
	SETB	**P2.7**	
MONITOR			
	SETB	**P1.0**	**Enable the H-bridge driver**
	JNB	**P2.7 CLOCKWISE**	
	CLR	**P1.1**	**01 is for Counter clockwise**
	SETB	**P1.2**	
	SJMP	**MONITOR**	
CLOCKWISE	**SETB**	**P1.1**	**10 is for clockwise**
	CLR P1.2		
	SJMP	**MONITOR**	

Q4)B) priority inversion is a problematic scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively "inverting" the relative priorities of the two tasks.

This violates the priority model that high priority tasks can only be prevented from running by higher priority tasks and briefly by low priority tasks which will quickly complete their use of a resource shared by the high and low priority tasks.

A priority ceiling

With priority ceilings, the shared mutex process (that runs the operating system code) has a characteristic (high) priority of its own, which is assigned to the task locking the mutex. This

works well, provided the other high priority task(s) that tries to access the mutex does not have a priority higher than the ceiling priority.

Priority inheritance

Under the policy of priority inheritance, whenever a high priority task has to wait for some resource shared with an executing low priority task, the low priority task is temporarily assigned the priority of the highest waiting priority task for the duration of its own use of the shared resource, thus keeping medium priority tasks from pre-empting the (originally) low priority task, and thereby affecting the waiting high priority task as well. Once the resource is released, the low priority task continues at its original priority level.

Random boosting

Ready tasks holding locks are randomly boosted in priority until they exit the critical section. This solution is used in Microsoft Windows.

Q4)C)

MUL and MLA

Multiply and multiply-accumulate (32-bit by 32-bit, bottom 32-bit result).

Syntax

MUL{*cond*}{S} *Rd, Rm, Rs*

MLA{*cond*}{S} *Rd, Rm, Rs, Rn*

where:

cond is an optional condition code (see *Conditional execution*).

S is an optional suffix. If *S* is specified, the condition code flags are updated on the result of the operation

Rd is the destination register.

Rm, Rs, Rn are registers holding the operands.

Do not use r15 for *Rd*, *Rm*, *Rs*, or *Rn*.

Rd cannot be the same as *Rm*.

Usage

The MUL instruction multiplies the values from *Rm* and *Rs*, and places the least significant 32 bits of the result in *Rd*.

The MLA instruction multiplies the values from *Rm* and *Rs*, adds the value from *Rn*, and places the least significant 32 bits of the result in *Rd*.

Examples

```
MUL     r10,r2,r5
```

UMULL, UMLAL, SMULL and SMLAL

Unsigned and signed long multiply and multiply accumulate (32-bit by 32-bit, 64-bit accumulate or result).

Syntax

Op{cond}{S} RdLo, RdHi, Rm, Rs

where:

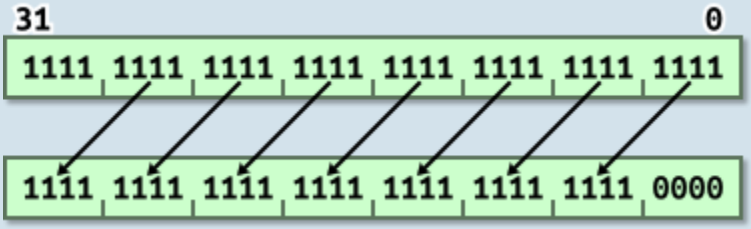
Op is one of UMULL, UMLAL, SMULL, or SMLAL.

Q5)A)

The *barrel shifter* is a functional unit which can be used in a number of different circumstances. It provides five types of shifts and rotates which can be applied to Operand2.

LSL – Logical Shift Left

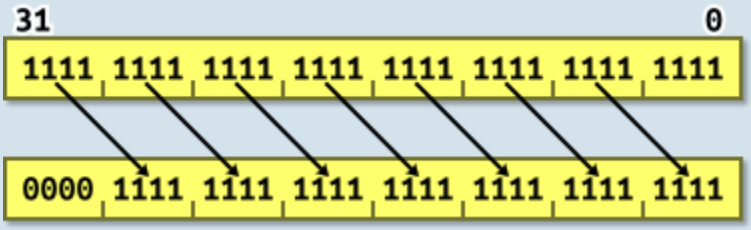
Example: Logical Shift Left by 4.



Equivalent to << in C.

LSR – Logical Shift Right

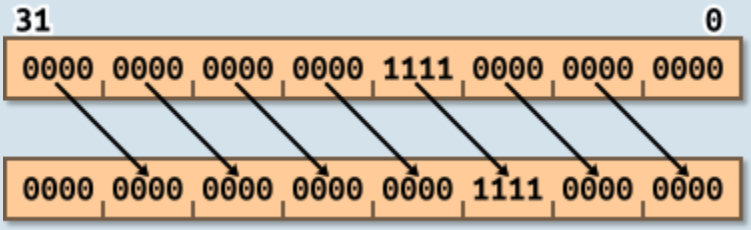
Example: Logical Shift Right by 4.



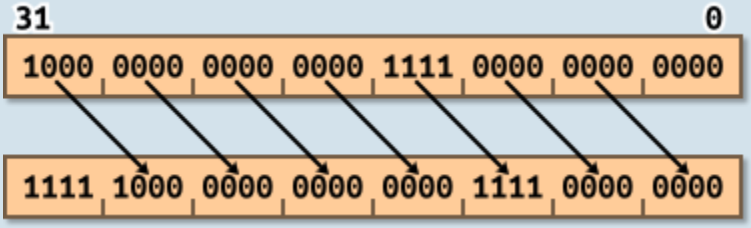
Equivalent to >> in C. i.e. unsigned division by a power of 2.

ASR – Arithmetic Shift Right

Example: Arithmetic Shift Right by 4, positive value.



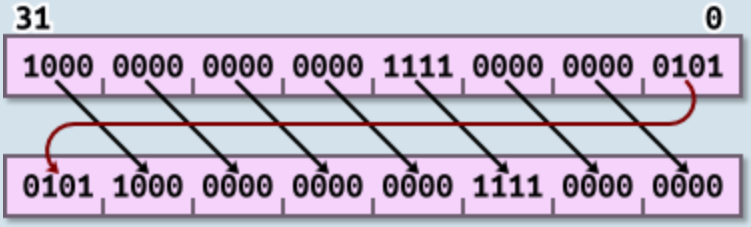
Example: Arithmetic Shift Right by 4, negative value.



Equivalent to >> in C. i.e. signed division by a power of 2.

ROR – Rotate Right

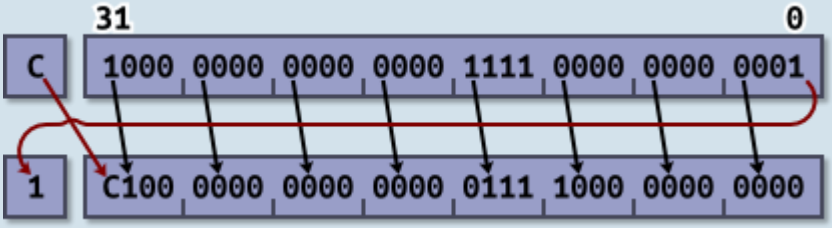
Example: Rotate Right by 4.



Bit rotate with wrap-around.

RRX – Rotate Right Extended

Example: Rotate Right Extended.



33-bit rotate with wrap-around through carry bit.

Q5) B)

1. Immediate addressing mode:

In this type, the operand is specified in the instruction along with the opcode. In simple way, it means data is provided in instruction itself.

Ex: MOV A,#05H -> Where MOV stands for move, # represents immediate data. 05h is the data. It means the immediate date 05h provided in instruction is moved into A register.

2.Register addressing mode:

Here the operand in contained in the specific register of microcontroller. The user must provide the name of register from where the operand/data need to be fetched. The permitted registers are A, R7-R0 of each register bank. Ex: MOV A,R0-> content of R0 register is copied into Accumulator.

3. Direct addressing mode:

In this mode the direct address of memory location is provided in instruction to fetch the operand. Only internal RAM and SFR's address can be used in this type of instruction.
Ex: MOV A, 30H => Content of RAM address 30H is copied into Accumulator.

4. Register Indirect addressing mode:

Here the address of memory location is indirectly provided by a register. The '@' sign indicates that the register holds the address of memory location i.e. fetch the content of memory location whose address is provided in register.
Ex: MOV A,@R0 => Copy the content of memory location whose address is given in R0 register.

5. Indexed Addressing mode:

This addressing mode is basically used for accessing data from look up table. Here the address of memory is indexed i.e. added to form the actual address of memory.
Ex: MOVC A,@A+DPTR => here 'C' means Code. Here the content of A register is added with content of DPTR and the resultant is the address of memory location from where the data is copied to A register.

Q5) C)

8051 ASSEMBLY LEVEL CODE TO FIND THE FACTORIAL OF GIVEN NUMBER
//GIVEN NUMBER STORED IN R1 REGISTER
//RESULT IS STORED IN R7 REGISTER

```
ORG 0000
MOV R1,#04
MOV R7,#01
LCALL FACT
MOV R7,A
FACT:
MOV A,R7
CJNE R1,#00,UP
SJMP UP1
UP:
MOV B,R1
MUL AB
DJNZ R1,UP
UP1:
RET
END
```

Q6)A)

1. **Compiler**

- A compiler is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language).
- Typically, from high level source code to low level machine code or object code.

2. **Assembler**

- An assembler translates assembly language programs into machine code. The output of a assembler is called an object file, which contains a combination of machine instruction as well as the data required to place these intstructions in memory.

3. **Linker**

- Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assembler.
- The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded making the program instruction to have absolute reference.

4. **Loader**

- Loader is a part of operating system and is responsible for loading executable files into memory and execute them.
- It calculates the size of a program (instructions and data) and create memory space for it. It initializes various registers to initiate execution.

Q6) B)

Mode	Encoding	Syntax	Description
Register	00/0	Rn	Register contents.
Indexed	01/1	X(Rn)	Value at address X+Rx; *X is stored in the word following the instruction.*
Symbolic	01/1	ADDR	Indexed mode x(PC) is used where x is the offset from the PC to the label ADDR. So the value is stored at address (x + PC)
Absolute	01/1	&ADDR	Copy the contents of one address into the contents of another address
Indirect Register	10/-	@Rn	Rn contains address of operand.
Indirect Autoincrement	11/-	@Rn+	Rn contains address of operand; afterwards, the contents of Rn are incremented
Immediate	11/-	#N	N is a constant; it is stored in the word following the instruction.

Q6) C)

scheduler is the part of the kernel responsible for deciding which task should be executing at any particular time. The kernel can suspend and later resume a task many times during the task lifetime.

The **scheduling policy** is the algorithm used by the scheduler to decide which task to execute at any point in time. The policy of a (non real time) multi user system will most likely allow each task a "fair" proportion of processor time.

Some commonly used RTOS scheduling algorithms are:

- Cooperative scheduling
- Preemptive scheduling
  - Rate-monotonic scheduling
  - Round-robin scheduling
  - Fixed priority pre-emptive scheduling, an implementation of preemptive time slicing
  - Fixed-Priority Scheduling with Deferred Preemption
  - Fixed-Priority Non-preemptive Scheduling
  - Critical section preemptive scheduling
  - Static time scheduling