Salient Features of 8051:

1) A Microcontroller is a **complete computer system** built on a **single chip**.

2) It contains all components like Processor (CPU), RAM, ROM, Serial port, Parallel port, Interrupt logic, Timers etc.. on chip.

3) A Microcontroller **saves cost**, saves **power** consumption and makes the circuit **compact**.

4) Microcontrollers are ideally suited for **appliances** like **remote controllers**, **refrigerators**, microwave ovens, modems etc.

5) **8051** is an **8-bit Microcontroller,** it has an 8 bit ALU. This means all arithmetic and logic operations are of 8 bits.

6) **8051** has an **8-bit data bus,** so all external Data Transfers will be of 8-bits in one cycle.

7) It has **internal ROM of 4KB** used for storing **programs**.

8) It has **internal RAM of 128 bytes** used for storing **data**.

9) Since **program memory (ROM) and data memory (RAM) are separate**, 8051 follows **Harvard Model**. In contrast, Processors based on Von Neumann Model store programs and data in a common memory space.

10) There are **4, 8bit, bidirectional I/O ports** for interfacing external devices like **keyboards, displays** etc. These ports can also be used for their **alternate functions** like multiplexed **address data buses** and **control signals**.

11) It has a **serial port for long distance communication**.
The serial port can perform synchronous and asynchronous transfers.

12) 8051 has **two, 16bit Timers**, which act as 'up' counters.
They are used to produce hardware delays and for counting external events.

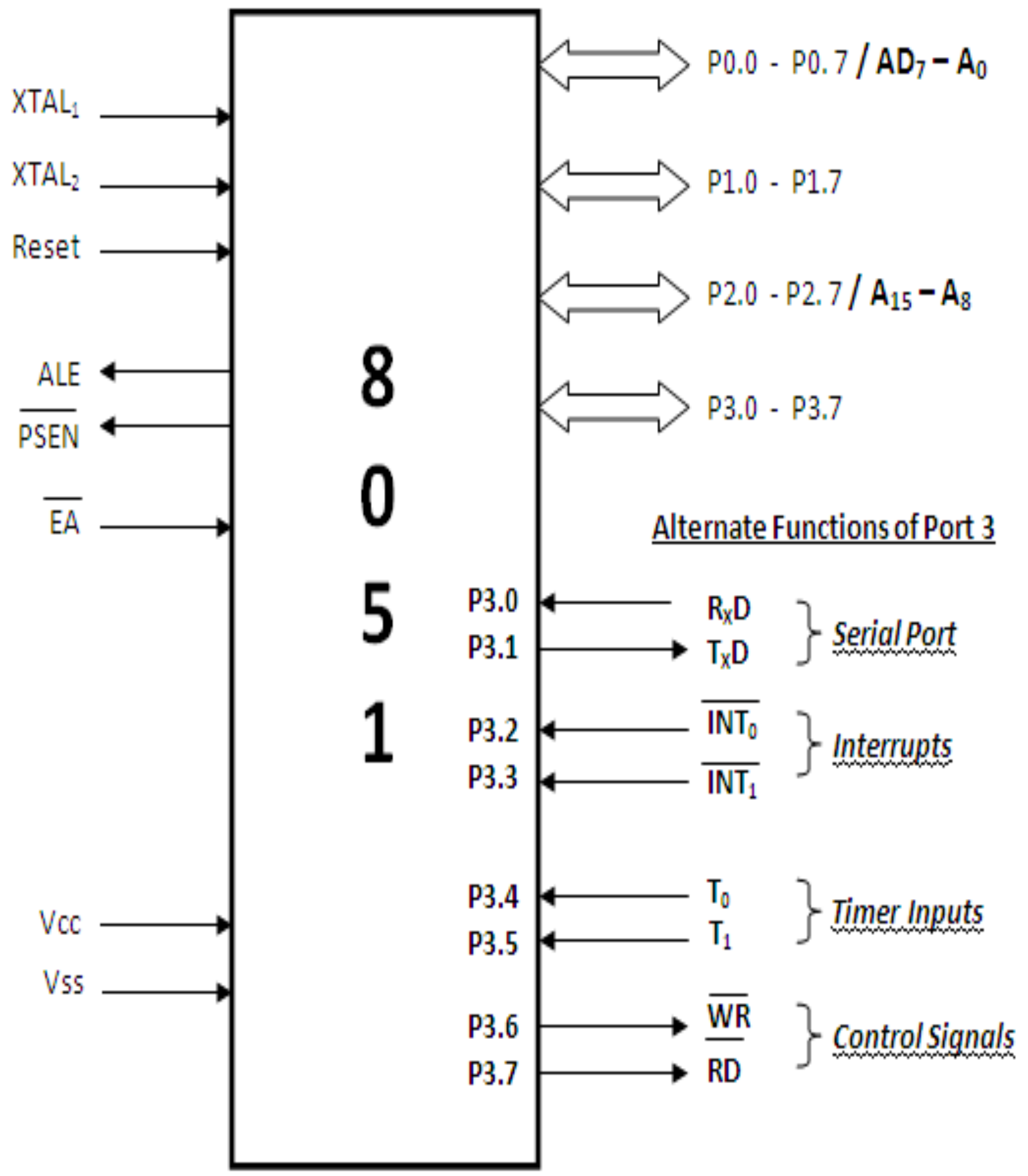
13) There are **5 interrupts**, operating at two priority levels.

14) 8051 has **two power saving modes** called "Idle mode" and "Power Down mode".

15) In addition to internal memory, **up to 64 KB** of **external RAM and External ROM** can be connected, as per user requirement. The figure 64 KB is due to the **16-bit address bus**.

16) 8051 is a **40-pin IC** and typically **operates at 12 MHz frequency.**

PIN DIAGRAM OF 8051



8051 has 40 pins.
The function of these pins is briefly explained as follows.

XTAL1 & XTAL2

These are connected to the **crystal oscillator**.
The **typical operating frequency is 12 MHz**.
In **Serial communication** based applications, the operating frequency is chosen to be **11.0592 MHz**, in order to derive the standard universal baud rates. This will be discussed in detail in the further chapters.

Reset

It is used to **reset** the 8051 microcontroller.
On reset **PC becomes 0000H**.
This address is called the **reset vector address**.
From here, 8051 executes the **BIOS program** also called the Booting program or the monitor program. It is used to **set-up the system** and make it **ready**, to be used by the **end-user**.

ALE
Address Latch Enable

It is used to **enable the latching of the address**.
The **address and data buses are multiplexed**.
This is done to **reduce the number of pins** on the 8051 IC.
Once out of the chip, address and data **have to be separated** that is called **de-multiplexing**.
This is done by **a latch**, with the **help of ALE signal**.
ALE is **"1"** when the bus carries **address** and **"0"** when the bus carries **data**.
This informs the latch, when the bus is carrying address so that the latch captures only address and not the data.

$\overline{\text{EA}}$
Enable External Access

It decides whether the first 4 KB of program memory space (0000H… 0FFFH) will be assigned to internal ROM or External ROM.

If $\overline{\text{EA}}$ = 0, the External ROM begins from 0000H.

In this case the Internal ROM is discarded.
8051 now uses only External ROM.

If $\overline{\text{EA}}$ = 1, the External ROM begins from 1000H.

In this case the Internal ROM is used. It occupies the space 0000H… 0FFFH.
In modern **FLASH ROM versions**, this pin also acts as **VPP** (12 Volt programming voltage) to write into the FLASH ROM.

$\overline{\text{PSEN}}$
Program Status Enable

8051 has a **16-bit address bus** ($A_{15} - A_0$).
This should allow 8051 to access **64 KB of external Memory** as 2^{16} = 64 KB.
Interestingly though, 8051 can access **64 KB of External ROM and 64 KB of External RAM**, making a total of 128 KB.
Both have the same address range **0000H to FFFFH**.
This does not lead to any confusion because there are separate control signals for External RAM and External ROM.

$\overline{\text{RD}}$ and $\overline{\text{WR}}$ are control signals for External RAM.

$\overline{\text{PSEN}}$ is the READ signal for External ROM.

It is called Program Status Enable as it allows reading from ROM also known as Program Memory.
Having separate control signals for External RAM and External ROM actually **allows us to double the size of the external memory** to a total of 128 KB from the original 64 KB.

Vcc
&
GND

These are **power supply** pins.
8051 works at **+5V / 0V** power supply.

P0.0… P0.7

These are **8 pins** of Port 0.
We can perform a **byte operation** (8-bit) on the whole port 0.
We can also **access every bit of port 0 individually** by performing **bit operations like set, clear, complement** etc.
The bits are called **P0.0… P0.7**.
Additionally, Port 0 also has an **alternate function**.
It carries the **multiplexed address data lines**.
A0-A7 (the lower 8 bits of address) and **D0-D7** (8 bits of data) are **multiplexed into AD0-AD7**.
In any operation address and data are not issued simultaneously.
First, address is given, then data is transferred.
Using a common bus for both, **reduces the number of pins**.
To identify if the bus is carrying address or data, we look at the ALE signal.
If **ALE = 1**, the bus carries **address**,
If **ALE = 0**, the bus carries **data**.

P1.0… P1.7

These are **8 pins** of Port 1.
We can perform a **byte operation** (8-bit) on the whole port 1.
We can also **access every bit of port 1 individually** by performing bit operations like set, clear, complement etc. on **P1.0… P1.7**.
Port 1 also has NO alternate function.

P2.0… P2.7

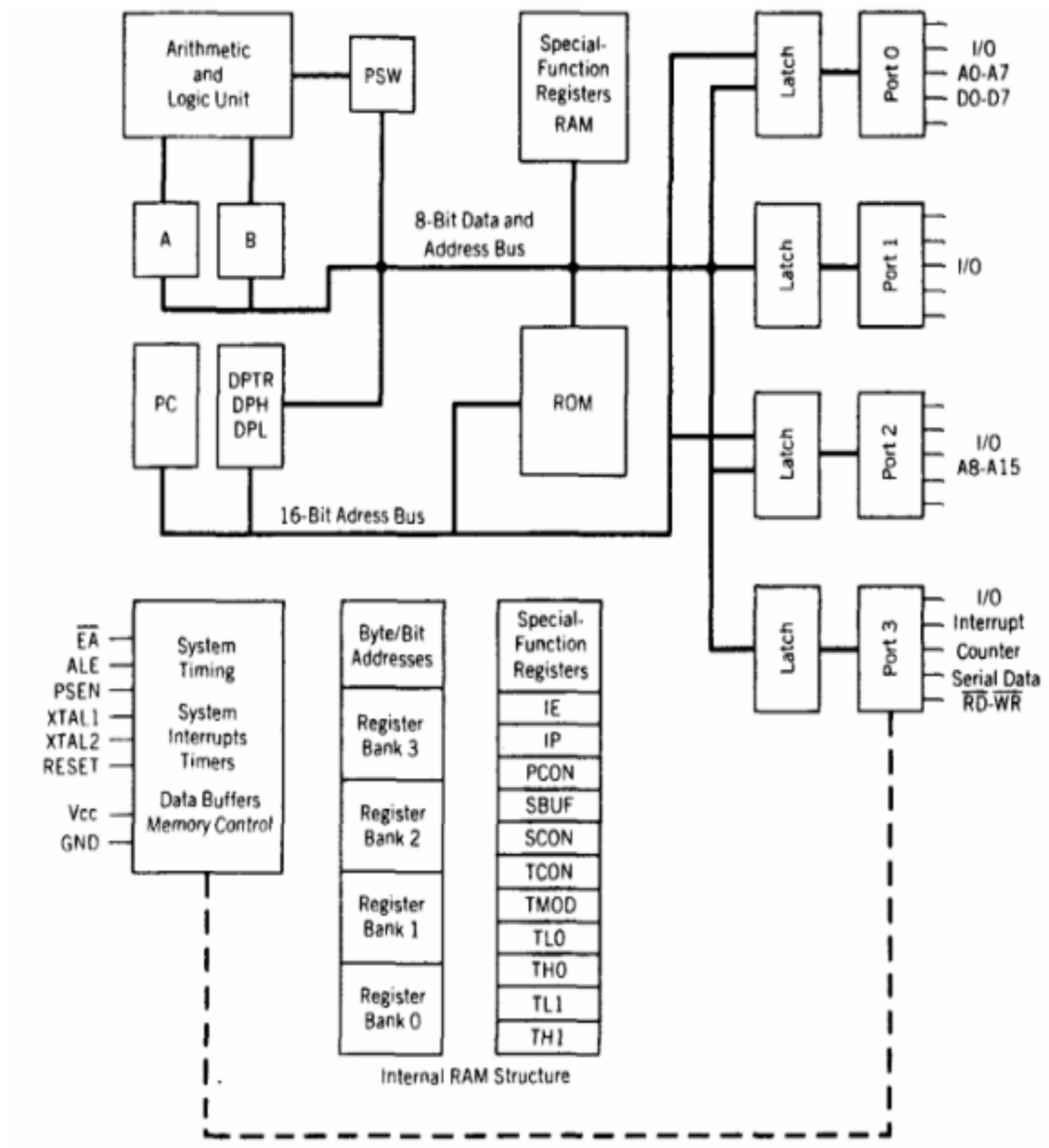
These are **8 pins** of Port 2.
We can perform a **byte operation** (8-bit) on the whole port 2.
We can also **access every bit of port 2 individually** by performing bit operations like set, clear, complement etc. on **P2.0… P2.7**.
Additionally, Port 2 also has an **alternate function**.
It carries the **higher order address lines A8-A15**.

P3.0… P3.7

These are **8 pins** of Port 3.
We can perform a **byte operation** (8-bit) on the whole port 3.
We can also **access every bit of port 3 individually.**
The bits are called **P0.0… P0.7**.
The various pins of Port 3 have a lot of alternate functions.

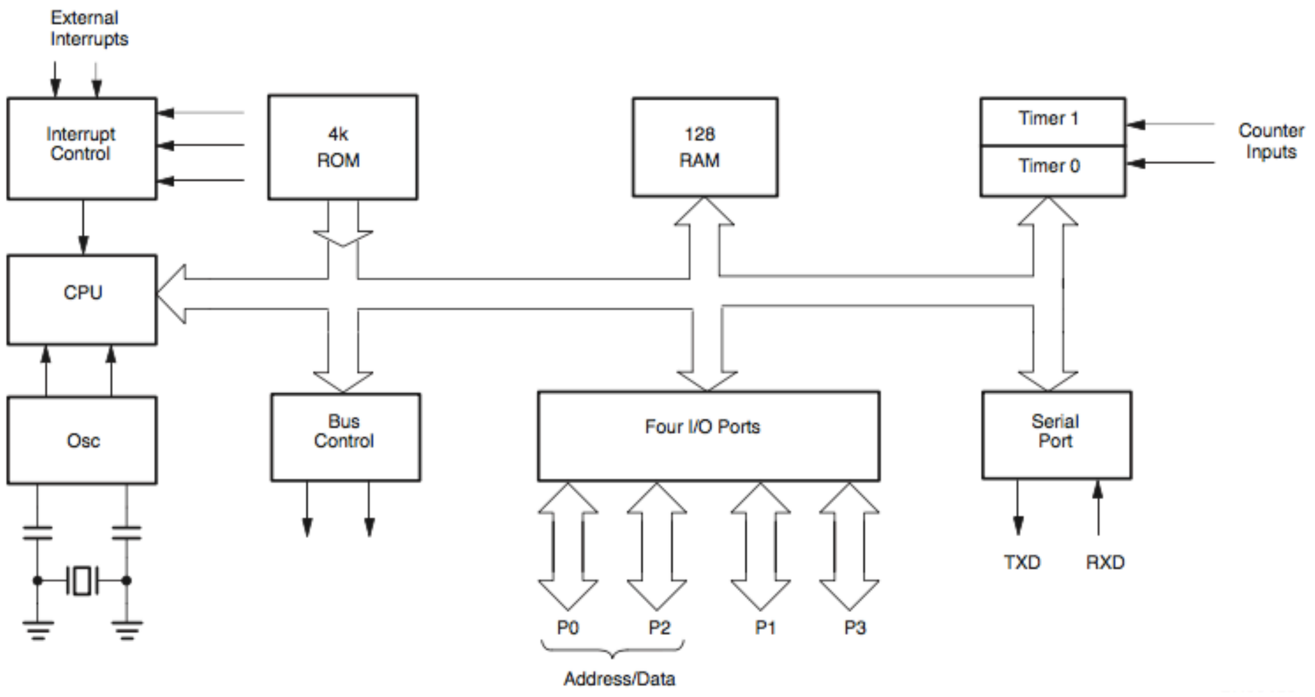
P3.0 (**Rxd**) and P3.1 (**Txd**):	They are used to **receive and transmit serial data**. This forms the **serial port of 8051**.
P3.2 ($\overline{\textbf{INT0}}$) and P3.3 ($\overline{\textbf{INT1}}$):	They are external **hardware interrupts of 8051**. If they occur simultaneously, INTO is by default higher priority.
P3.4 (**T0**) and P3.5 (**T1**):	They are used **timer clock inputs**. They provide external clock inputs to Timer 0 and Timer 1.
P3.6 ($\overline{\textbf{WR}}$) and P3.7 ($\overline{\textbf{RD}}$):	They are used as **control signals for External RAM**. 8051 can access 64 KB External RAM from 0000H to FFFFH..

8051 BLOCK DIAGRAM



Alternate diagram …

ALTERNATE DIAGRAM FOR 8051 ARCHITECTURE / BLOCK DIAGRAM



8051 is a microcontroller. This means it has an internal processor, internal memory and an I/O section. The architecture of 8051 is thus divided into three main sections:

- The CPU
- Internal Memory
- I/O components.

CPU

8051 has an 8 bit CPU.
This is where all 8-bot arithmetic and logic operations are performed.
It has the following components.

ALU – ARITHMETIC LOGIC UNIT

It performs **8-bit arithmetic and logic operations**.
It can also perform some bit operations.

Example:

ADD A, R0	; *Adds contents of A register and R0 register and stores the result in A register.*
ANL A, R0	; *Logically ANDs contents of A register and R0 register and stores the result in A register.*
CPL P0.0	; *Complements the value of P0.0 pin.*

A – REGISTER (ACCUMULATOR)

It is an **8-bit register**.
In most arithmetic and logic operations, **A register hold the first operand and also gets the result of the operation**.
Moreover, it is the only register to be used for **data transfers** to and from **external memory**.

Example:

ADD A, R1	; *Adds contents of A register and R1 register and stores the result in A register.*
MOVX A, @DPTR	; *A gets the data from External RAM location pointed by DPTR*

B – REGISTER

It is an **8-bit register**.
It is dedicated for **Multiplication and Division**.
It can also be used in other operations.

Example:

MUL AB	; *Multiplies contents of A and B registers. Stores 16-bit result in B and A registers.*
DIV AB	; *Divides contents of A by those of B. Stores quotient in A and remainder in B.*

PC – PROGRAM COUNTER

It is an **16-bit register**.
It holds **address of the next instruction** in program memory (ROM).
PC gets automatically **incremented** as soon as any **instruction is fetched**.
That's what makes the program move ahead in a **sequential manner**.
In the case of a **branch**, a **new address is loaded into PC**.

DPTR – DATA POINTER

It is an **16-bit register**.
It holds **address data** in data memory (RAM).
DPTR is divided into two registers DPH (higher byte) and DPL (lower byte).
It is typically used by the programmer **to transfer data from External RAM**.
It can also be used as a pointer to a **look up table** in ROM, using **Indexed addressing mode**.

Example:
```
MOVX A, @DPTR      ; A gets the data from External RAM location pointed by DPTR
MOVC A, @A+DPTR    ; A gets the data from ROM location pointed by A + DPTR
```

SP – STACK POINTER

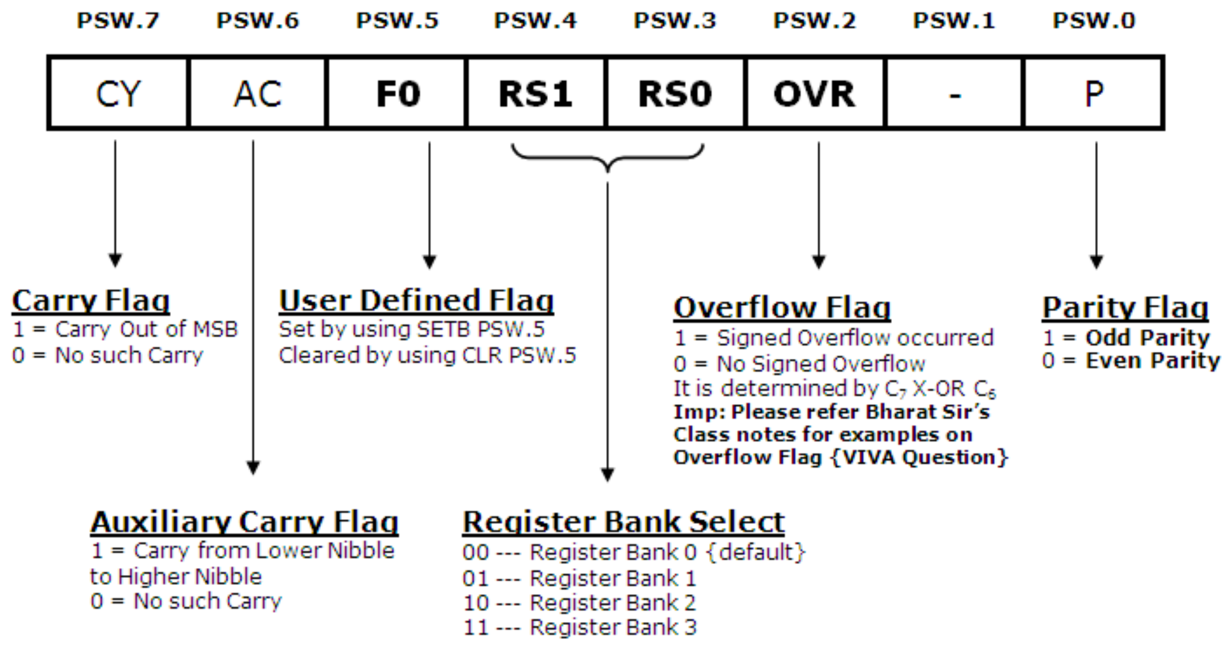
It is an **8-bit register**.
It contains **address of the top of stack**.
The Stack is present in the Internal RAM.
Internal RAM has 8-bit addresses from 00H… 7FH.
Hence SP is an 8-bit register.
It is affected during Push and Pop operations.
During a **Push**, SP gets **incremented**.
During a **Pop**, SP gets **decremented**.

PSW – PROGRAM STATUS WORD

It is an **8-bit register**.
It is also called the **"Flag register"**, as it mainly contains the status flags.
These flags indicate **status of the current result**.
They are **changed by the ALU after every arithmetic or logic operation**.
The flags can also be **changed by the programmer**.
PSW is a **bit addressable** register.
Each bit can be individually set or reset by the programmer.
The bits can be referred to by their bit numbers (**PSW.4**) or by their name (**RS1**).

Example:
```
SETB PSW.3         ; Makes PSW.3 ← 1
CLR PSW.4          ; Makes PSW.4 ← 0
```

FLAG REGISTER (PSW) OF 8051



RS1 RS0	REGISTER BANK	RAM ADDRESS	SELECTED BY INSTRUCTIONS
0 0	Bank 0	00H … 07H	CLR PSW.4, CLR PSW.3
0 1	Bank 1	08H … 0FH	CLR PSW.4, SETB PSW.3
1 0	Bank 2	10H … 17H	SETB PSW.4, CLR PSW.3
1 1	Bank 3	18H … 1FH	SETB PSW.4, SETB PSW.3

Internal Memory

8051 has two forms of internal memories.
It has 128 bytes of Internal RAM and 4 KB of Internal ROM.

Internal RAM

8051 has **128 bytes of Internal RAM**.
RAM is used to store data, hence is also called **Data Memory**.
The are **128 locations** each containing one byte information.
The **address range** is **00H… 7FH**.
It contains **Register banks**, a **Bit addressable area** and a **General purpose area**.

Internal ROM

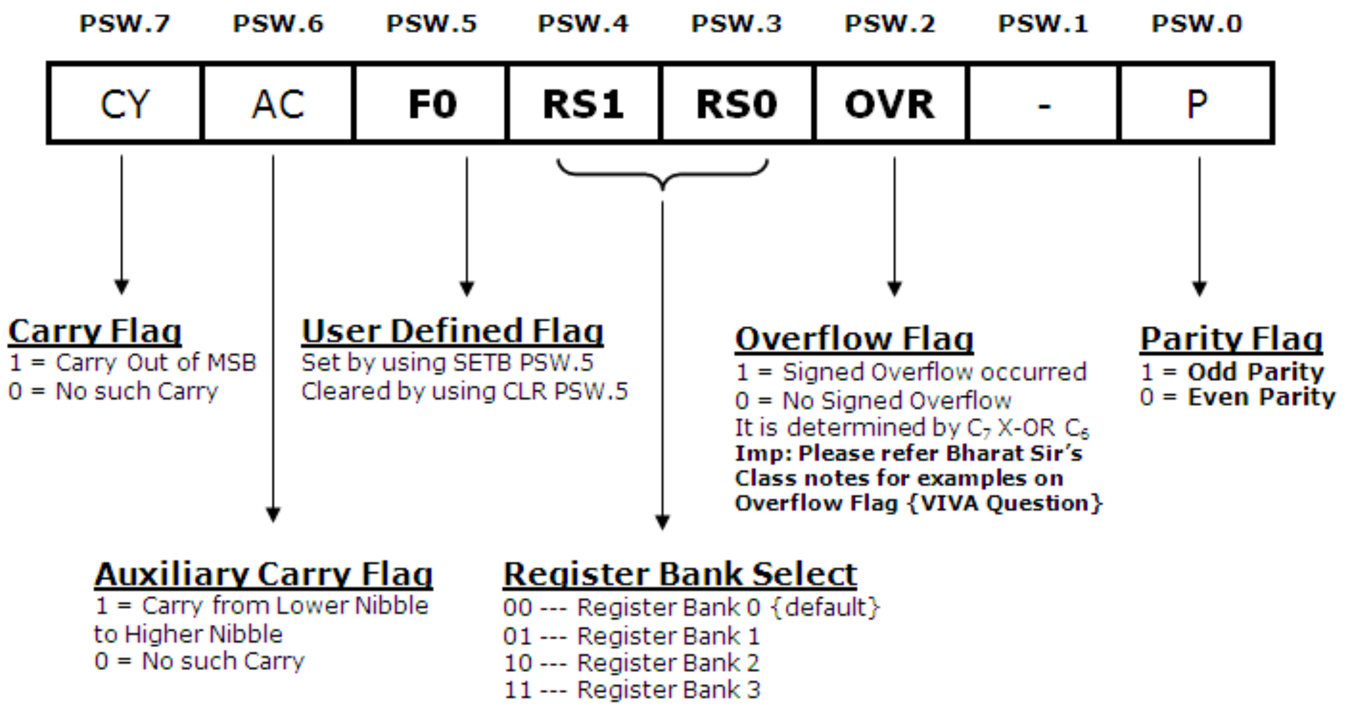
8051 has **4 KB of Internal ROM**.
ROM is used to store programs, hence is also called **Program Memory or Code Memory**.
The are **4 K locations** each containing one byte information.
The **address range** is **0000H… 0FFFH**.
It mainly contains **programs.**
It may also contains some **permanent data** stored in the form of **look up tables**.
To access **programs**, the address is given by **PC** – Program Counter.
To access **data**, the address is given by **DPTR** – Data Pointer.

I/O Components

Like any other typical microcontroller, 8051 has several I/O components.
They include I/O ports, Timers, Serial port etc.
8051 has **4, 8-bit I/O ports: P0, P1, P2 and P3.**
They support **bit and byte operations.**
They also have several **alternate functions**.
There are **two 16-bit timers**, which operate as down counters.
There is a **serial port** having pins **Rxd and Txd** to receive and transmit data serially.
There are **two external interrupt pins**.
Additionally there are **address, data and control signals** for transfers with **External RAM and External ROM**.

Finally, 8051 has **21, 8-bit internal SFRs** (Special Function Registers).
These are used to **control operations of the various I/O components** mentioned above.

FLAG REGISTER (PSW) OF 8051



PSW – PROGRAM STATUS WORD

It is an **8-bit register**.
It is also called the **"Flag register"**, as it mainly contains the status flags.
These flags indicate **status of the current result**.
They are **changed by the ALU after every arithmetic or logic operation**.
The flags can also be **changed by the programmer**.
PSW is a **bit addressable** register.
Each bit can be individually set or reset by the programmer.
The bits can be referred to by their bit numbers (**PSW.4**) or by their name (**RS1**).

CY - CARRY FLAG

It indicates the carry out of the MSB, after any arithmetic operation.

If CY = 1 : There was a carry out of the MSB
If CY = 0 : There was no carry out of the MSB

AC – AUXILIARY CARRY FLAG

It indicates the carry from lower nibble (4-bits) to higher nibble.
If the 8bits are numbered Bit 7 --- Bit 0, this is the carry from Bit 3 to Bit 4.

If AC = 1 : There was an auxiliary carry
If AC = 0 : There was no auxiliary carry

Note: It is particularly useful in an operation called DA A (Decimal Adjust after Addition).

OVR - OVERFLOW FLAG

It indicates if there was an overflow during a signed operation.
An 8-bit signed number has the range -80H… 00H… +7FH.
Any result, out of this range causes an overflow.

If OVR = 1 : There was an overflow in the result
If OVR = 0 : There was no overflow in the result

Overflow is determined by doing an Ex-Or between the 2nd last carry (C_6) and the last carry (C_7)

Note: After an overflow, the Sign (MSB) of the result becomes wrong.

P - PARITY FLAG

It indicates the Parity of the result.
Parity is determined by the number of 1's in the result.

If PF = 1 : The result has ODD parity
If PF = 0 : The result has EVEN parity

F0 – USER DEFINED FLAG

This flag is **available to the programmer**.
It can be used by us to store any **user defined information**.
For example: In an Air Conditioning unit, programmer can use this flag indicate whether the compressor is ON or OFF (1 or 0).
This flag can be changed by simple instructions like SETB and CLR.

SETB PSW.5; This makes F0 bit ← 1
CLR PSW.5; This makes F0 bit ← 0

RS1, RS0 – REGISTER BANK SELECT

The initial 32 locations (bytes) of the Internal RAM are available to the programmer as registers.
Having so many registers makes programming easier and faster.
Naming R0… R31, would tremendously increase the number of opcodes.
Hence the registers are divided into 4 banks: Bank0… Bank3.
Each bank has 8 registers named R0… R7.
At a time, only of the four banks is the "active bank".
RS1 and RS0 are used by the programmer to select the active bank.

RS1 RS0	REGISTER BANK	SELECTED BY INSTRUCTIONS
0 0	Bank 0	CLR PSW.4 CLR PSW.3
0 1	Bank 1	CLR PSW.4 SETB PSW.3
1 0	Bank 2	SETB PSW.4 CLR PSW.3
1 1	Bank 3	SETB PSW.4 SETB PSW.3

NUMERICAL EXAMPLES FOR FLAG REGISTER

		1				1	1	
31 H	0	0	1	1	0	0	0	1
+ 23 H	0	0	1	0	0	0	1	1
= 54 H	0	1	0	1	0	1	0	0
Flags Affected:	CY			AC	OVR			P
	0			0	0			1

		1	1	1	1	1	1	
39 H	0	0	1	1	1	0	0	1
+ 27 H	0	0	1	0	0	1	1	1
= 60 H	0	1	1	0	0	0	0	0
Flags Affected:	CY			AC	OVR			P
	0			1	0			0

	1							
42 H	0	1	0	0	0	0	1	0
+ 44 H	0	1	0	0	0	1	0	0
= 86 H	1	0	0	0	0	1	1	0
Flags Affected:	CY			AC	OVR			P
	0			0	1			1

The result 86H is out of range for a "Signed" Number as it has become greater than +7FH.
Such an event is called a "Signed Overflow".
In such a case the MSB of the result gives a wrong sign.
Though the result is +ve (+86H) the MSB is "1" indicating that the result is –ve.
Overflow is determined by doing an Ex-Or between the 2nd last Carry and the last Carry.
Here the 2nd last Carry (the one coming into the MSB) is "1".
The final carry (The one going out of the MSB) is "0".
As "1" Ex-Or "0" = "1", the Overflow flag is "1".

For similar examples, but on Negative numbers…
Please refer to Bharat Sir's Classroom Notes.

MEMORY ORGANIZATION OF 8051

8051 operates with 4 different memories:

Internal ROM
External ROM

Internal RAM
External RAM

Being based on **Harvard Model**, 8051 stores **programs and data in separate memory spaces**.
Programs are stored in ROM, whereas data is stored in RAM.

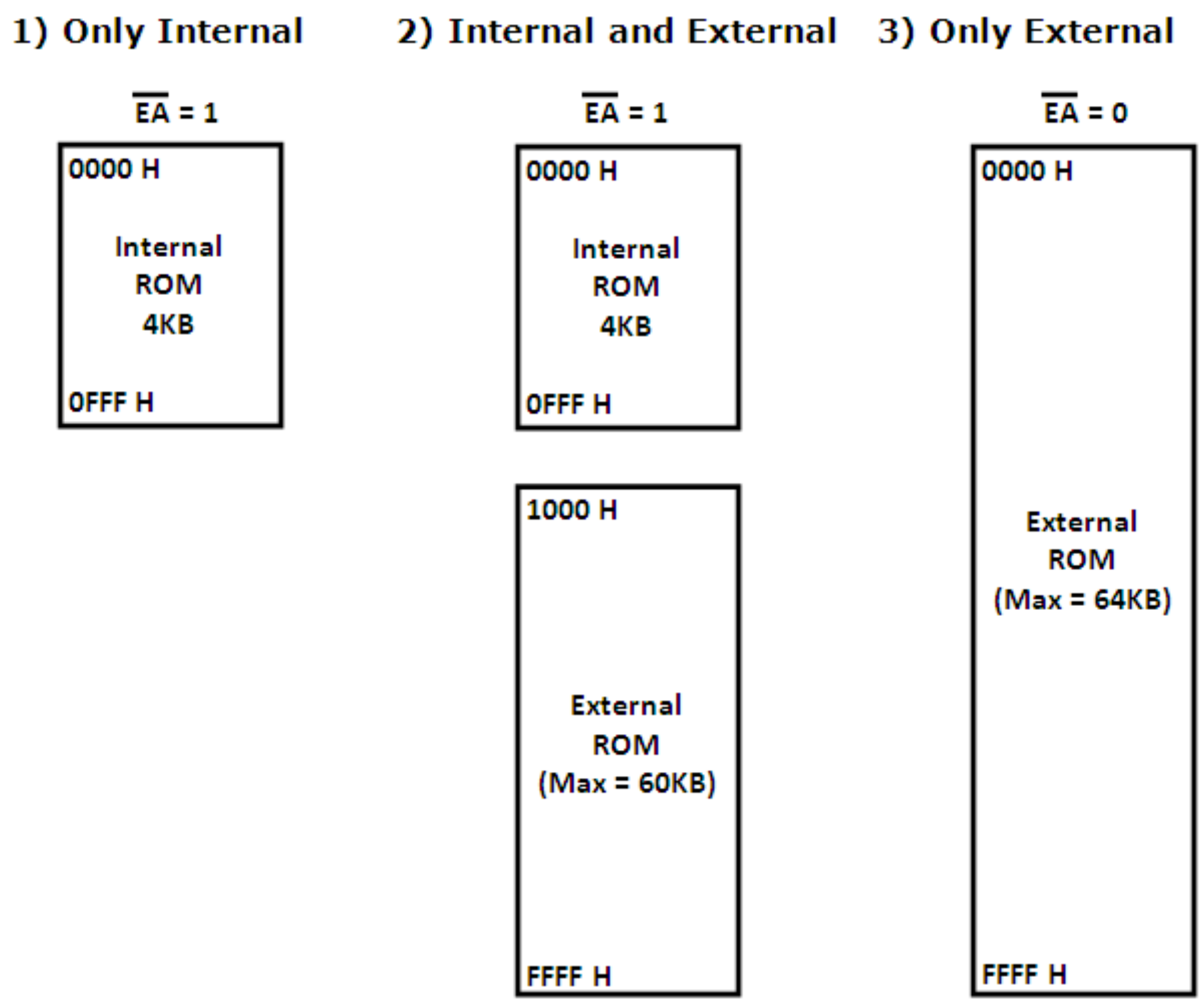
Microcontrollers are used in **appliances**.
Washing machines, remote controllers, microwave ovens are some of the examples.
Here **programs are generally permanent** in nature and very rarely need to be modified.
Moreover, the programs must be **retained** even after the device is completely **switched off**.
Hence programs are stored in permanent (non-volatile) memory like ROM.

Data on the other hand is continuously **changed at runtime**.
For example current temperature, cooking time etc. in an oven.
Such data is not permanent in nature and will certainly be modified in every usage of the device.
Hence Data is stored in writeable memory like RAM.

However, sometimes there is **permanent data**, such as ASCII codes or 7-segment display codes.
Such data is stored in **ROM**, in the form of **Look up tables** and is accessed using a dedicated addressing mode called **Indexed** Addressing mode. We will discover this in more depth in further topics.

We are now going to take a closer look at all four memories.

ROM ORGANIZATION / CODE MEMORY / PROGRAM MEMORY



We can implement ROM in three different ways in 8051.

ONLY INTERNAL ROM

8051 has 4 KB internal ROM.
In many cases this size is sufficient and there is no need for connecting External ROM.
Such systems use only Internal ROM of 8051.
All addresses from 0000H… 0FFFH will be accessed from Internal ROM.
Any address beyond that will be invalid.
In such systems $\overline{\textbf{EA}}$ will be "1" as Internal ROM is being used.

(PS: Read the whole answer to understand $\overline{\text{EA}}$ clearly)

INTERNAL AND EXTERNAL ROM

8051 has 4 KB internal ROM.
In many cases this size is may be insufficient and we may need to add some External ROM.
Such systems use a combination of Internal ROM and External ROM.
The **"total"** ROM that can be accessed is 64 KB.
Since we are using the Internal ROM of 4 KB, the maximum amount of External ROM that can be connected is 60 KB.
All addresses from 0000H… 0FFFH will be accessed from Internal ROM.
Addresses 1000H… FFFH will be accessed from External ROM.
In such systems $\overline{\textbf{EA}}$ will be "1" as Internal ROM is being used.

(PS: Read the whole answer to understand $\overline{\text{EA}}$ clearly)

ONLY EXTERNAL ROM

This is the most interesting case.
Though 8051 has 4 KB of Internal ROM, the user may choose the discard it and connect only External ROM.
This may happen due to several reasons.
The program stored in the Internal ROM may have become invalid or outdated, or the system may need to be upgraded etc.
Such systems use only External ROM, and the Internal ROM is discarded.
Here we can connect up to 64 KB of External ROM.
All addresses from 0000H… FFFFH will be accessed from External ROM.
But do keep in mind, that the Internal ROM is still present in 8051.
We need to clearly indicate to 8051 that the Internal ROM must be ignored and every address from 0000H… FFFFH must be accessed externally. This is indicated by us to 8051 using $\overline{\textbf{EA}}$ **.**

By making $\overline{\textbf{EA}}$ **= 0**, we inform 8051 that the Internal ROM must be discarded and all ROM must be accessed externally.

Note: Use of $\overline{\textbf{EA}}$ pin of 8051.

The $\overline{\textbf{EA}}$ pin of 8051 decides whether the Internal ROM will be used or not.

If the Internal ROM has to be used we must make $\overline{\textbf{EA}}$ = 1.

Now 8051 will Access the internal ROM for all addresses from 0000H to 0FFFH and will only access external ROM for addresses 1000H and beyond.

But if $\overline{\textbf{EA}}$ = 0, then the Internal ROM is completely discarded.

Now 8051 will access the External ROM for all addresses from 0000H to FFFFH, hence discarding the internal ROM.

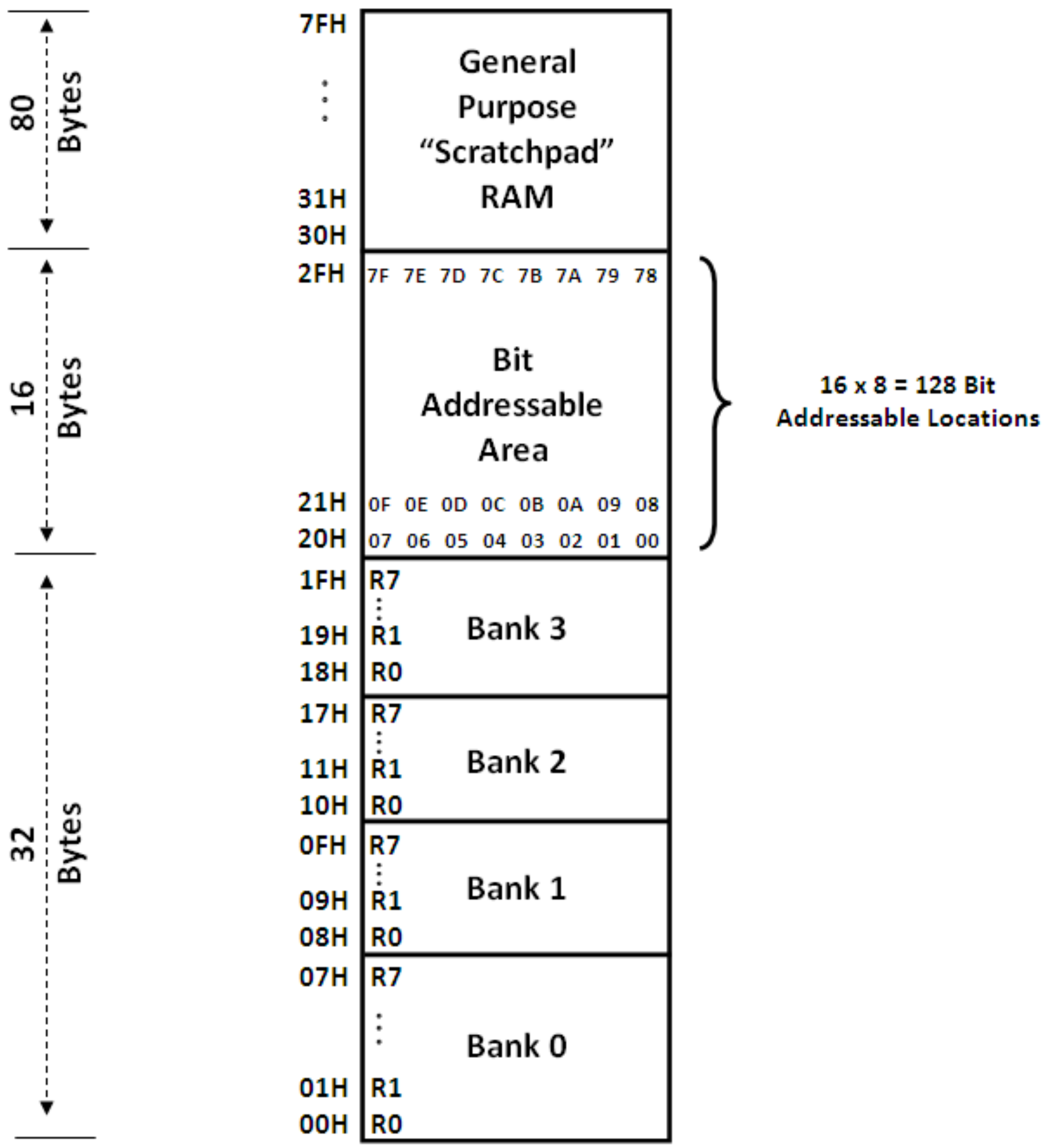
8051 checks $\overline{\textbf{EA}}$ pin during every ROM operation where the address is 0000H… 0FFFH.

If $\overline{\textbf{EA}}$ **= 1,** this location is accessed from internal ROM.

If $\overline{\textbf{EA}}$ **= 0,** this location is accessed from external ROM.

If the address is 1000H or more, 8051 does not check $\overline{\textbf{EA}}$ as this location can only be present in External ROM.

STRUCTURE OF INTERNAL RAM OF 8051



8051 has a 128 Bytes of internal RAM.
These are 128 locations of 1 Byte each.
The address range is 00H… 7FH.
This RAM is used for storing data.
It is divided into three main parts: Register Banks, Bit addressable area and a general purpose area.

REGISTER BANKS

1) The **first 32 locations (Bytes)** of the Internal RAM from **00H… 1FH**, are used by the programmer as general purpose registers.
2) Having so many general purpose registers makes programming easier and faster.
3) But as a downside, this also vastly increases the number of opcodes (refer my class lectures for detailed understanding of this).
4) **Hence the 32 registers are divided into 4 banks, each having 8 Registers R0… R7.**
5) The first 8 locations 00H… 07H are registers R0… R7 of bank 0.
6) Similarly locations 08H… 0FH are registers R0… R7 of bank 1 and so on.
A register can be addressed using its name, or by its address.
Eg: Location 00H can be accessed as R0, if Bank 0 is the active bank.
MOV A, R0; "A" register gets data from register R0.
It can also be accessed as Location 00H, irrespective of which bank is the active bank.
MOV A, 00H; "A" register gets data from Location 00H.
7) The appropriate bank is selected by the **RS1, RS0 bits of PSW**.
Since PSW is available to the programmer, any Bank can be selected at run-time.
8) **Bank 0** is selected by **default**, on reset.

BIT ADDRESSABLE AREA

1) The **next 16-bytes** of RAM, from **20H… 2FH**, is available as **Bit Addressable** Area.
2) We can perform ordinary byte operations on these locations, as well as bit operations.
#Please refer Bharat Sir's Lecture Notes for detailed explanation on this ...
3) As each location has 8-bits, we have a total of ➔ 16 × 8 = **128 Addressable Bits.**
4) These bits can be addressed using their individual address **00H … 7FH**.
SETB 00H; Will store a "1" on the LSB of location 20H
CLR 07H; Will store a "0" on the MSB of location 20H
5) Normal **"BYTE"** operations can also be performed at the addresses: **20H … 2FH**.
MOV 20H, #00H; Will store a "0" on all 8-bits of location 20H.
6) Here is something very interesting to know and will also help you understand further topics.
The entire internal RAM is of 128 bytes so the address range is 00H… 7FH.
The bit addressable area has 128 bits so its bit addresses are also 00h… 7FH.
7) This means every address 00H… 7FH can have two meanings, it could be a byte address or a bit address.
8) This does not lead to any confusion, because the instruction in which we use the address, will clearly indicate whether it is a bit operation or a byte operations.
SETB, CLR etc. are bit ops whereas ADD, SUB etc. are byte operations.
9) **SETB 00H;** This is a bit operation.
It will make Bit location 00H contain a value "1".
10) **MOV A, 00H;** This is a byte operation.
"A" register will get 8-bit data from byte location 00H.

GENERAL PURPOSE AREA

1) The general-purpose area ranges from location 30H … 7FH.
2) This is an 80-byte area which can be used for general data storage.

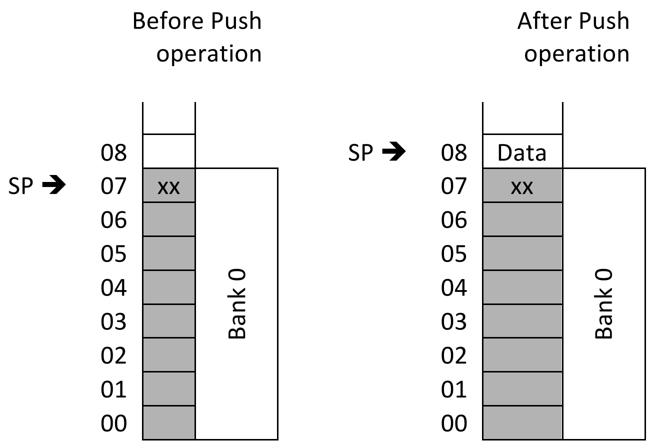
STACK OF 8051

1) Another important element of the Internal RAM is the **Stack**.
2) Stack is a set of memory locations operating in Last In First Out (LIFO) manner.
3) It is used to store return addresses during ISRs and also used by the programmer to store data during programs.
4) In 8051, the **Stack** can only be present in the **Internal RAM**.
5) This is because, **SP** which is an **8-bit register**, can only contain an 8-bit address and External RAM has 16-bit address. (#Viva)
6) On **reset SP** gets the value **07H**.
7) Thereafter SP is changed by every PUSH or POP operation in the following manner:

PUSH:
SP ← SP + 1
[SP] ← New data

POP:
Data ← [SP]
SP ← SP – 1

8) The reset value of SP is 07H because, on the first PUSH, SP gets incremented and then data is pushed on to the stack. This means the very first data will be stored at location 08H.
9) This does not affect the default bank (0) and still gives the stack, the maximum space to grow. (#Viva)



10) The programmer can relocate the stack to any desired location by simply putting a new value into SP register.

SPECIAL FUNCTION REGISTERS (SFRS) OF 8051

8051 has 21, 8-bit Special Function registers.

	NAME	FUNCTION	BYTE ADDRESS	BIT ADDRESS
Used for holding data and status during Programming	A*	Accumulator	0E0H	0E7H…0E0H
	B*	Arithmetic	0F0H	0F7H…0F0H
	PSW*	Program Status Word	0D0H	0D7H…0D0H
Used in instructions to point to memory	SP	Stack Pointer	81H	NA
	DPL	Address External Memory	82H	NA
	DPH	Address External Memory	83H	NA
Used by the respective I/O Ports	P0*	I/O Port latch	80H	87H…80H
	P1*	I/O Port latch	90H	97H…90H
	P2*	I/O Port latch	0A0H	0A7H…0A0H
	P3*	I/O Port latch	0B0H	0B7H…0B0H
Used by the Serial Port	SCON*	Serial Port Control	98H	9FH…98H
	SBUF	Serial Port Data Buffer	99H	NA
	TCON*	Timer/Counter Control	88H	8FH…88H
Used for Timer Control	TMOD	Timer/Counter Mode Control	89H	NA
	TL0	Timer 0 Low Byte	8AH	NA
	TL1	Timer 1 Low Byte	8BH	NA
	TH0	Timer 0 High Byte	8CH	NA
	TH1	Timer 1 High Byte	8DH	NA
Used for Interrupt Control	IE*	Interrupt Enable	0A8H	0AFH…0A8H
	IP*	Interrupt Priority	0B8H	0BFH…0B8H
Used for Power Control	PCON	Power Control	87H	NA

*Means the SFR is Bit Addressable

1) SFRs are **8-bit** registers.
Each SFR has its own **special function**.
2) **They are** placed **inside** the **Microcontroller.** #Please refer Bharat Sir's Lecture Notes for this ...
3) They are used by the programmer to perform special functions like controlling the timers, the serial port, the I/O ports etc.
4) As SFRs are available to the programmer, we will use them in instructions.
This causes another problem.
SFRs are registers after all, and hence using them would tremendously increase the number of opcodes. ***(Refer to Bharat Sir's Lecture notes for more on this)***
5) To reduce the number of opcodes, **SFRs are allotted addresses.**
These addresses must not clash with any other addresses of the existing memories.
6) Incidentally, the internal RAM is of 128 bytes and uses addresses only from 00H… 7FH.
This gives an entire range of addresses from 80H… FFH completely unused and can be freely allotted to the SFRs.
7) **Hence SFRs are allotted addresses from 80H… FFH.**
It is not a co-incidence that these addresses are free. It is how 8051 design was planned. The Internal RAM was restricted to 128 bytes instead of 256 bytes so that these addresses are free for SFRs.
8) Moreover, some SFRs are bit addressable, like Port 0.
All 8-bits can be individually accessed from P0.0… P0.7, by instructions like SETB, CLR etc.
But again, this will again tremendously increase the number of opcodes.
9) To avoid this problem, **even the bits of the SFRs are allotted addresses.**
These are bit addresses, which are different from byte addresses.
These bit addresses must not clash with those of the bit addressable area of the Internal RAM.
Amazingly, even the bit addresses in the Internal RAM are 00H… 7FH (again 128 bits), keeping bit addresses 80H… FFH free to be used by the SFR bits.
10) **So bit addresses 80H… FFH are allotted to the bits of various SFRs.**
(Watch Bharat Acharya Education, videos on YouTube for more on this)
11) Port 0 has a byte address of 80H and its bit addresses are from 80H… 87H.
A byte operation at address 80H will affect entire Port0.
E.g.:: MOV A, P0; this refers to Byte address 80H that's whole Port 0.
12) **A bit operation at 80H will affect only P0.0.**
E.g.:: SETB P0.0; this refers to bit address 80H that's Port0.0

ADDRESSING MODES OF 8051

Addressing Modes is the manner in which operands are given in the instruction.
8051 supports the following 5 addressing modes:

1) IMMEDIATE ADDRESSING MODE

In this addressing mode, the **Data** is given **in** the **Instruction** itself.
We put a "**#**" symbol, before the data, to identify it as a data value and not as an address.

Eg: MOV A, **#**35H *; A ← 35H*

MOV DPTR, **#**3000H *; DPTR ← 3000H*

2) REGISTER ADDRESSING MODE

In this addressing mode, **Data** is given by **a Register** in the instruction.
The permitted registers are **A**, **R7 … R0** of each memory bank.
Note: Data transfer between two RAM registers is not allowed.

Eg: MOV A, R0 *; A ← R0 … If R0 = 25H, then A gets the Value 25H.*

MOV R5, A *; R5 ← A*

MOV Rx, Ry *; NOT ALLOWED. That's because this would allow 64 combinations of registers*
; As registers invite opcodes, this would need 64 opcodes!

3) DIRECT ADDRESSING MODE

Here, the **address** of the operand is given **in** the **instruction**.
Only Internal RAM addresses (**00H…7FH**) and **SFR** addresses (from **80H to FFH**) allowed.

Eg: MOV A, 35H *; A ← Contents of RAM location 35H*

MOV A, 80H *; A ← Contents of Port 0 (SFR at address 80H)*

MOV 20H, 30H *; [20H] ← [30H]*
; i.e. Location 20H gets the contents of Location 30H

4) INDIRECT ADDRESSING MODE

Here, the **address** of the operand is given **in** a **register.**
Internal RAM and External RAM can be accessed using this mode.
The advantage of giving an address using a register is that we can increment the address in a loop, by simply incrementing the register, and hence access a series of locations.

INTERNAL RAM: (8-BIT ADDRESS GIVEN BY R0 OR R1)

ONLY R1 or R0, called as *Data Pointers*, can be used to specify **address** (00H … 7FH).
An "**@**" sign is present before the register to indicate that the register is giving an address.

Eg: MOV A, @R0 *; A ← [R0]*
; i.e. A ← Contents of Internal RAM Location whose address is given by R0.
; if R0 = 25H, then A gets the contents of Location 25H from Internal RAM

MOV @R1, A *; [R1] ← A i.e. Internal RAM Location pointed by R1 gets value of A.*

EXTERNAL RAM: (16 BIT ADDRESS GIVEN BY DPTR)

For the **External RAM, address** is provided by **R1 or R0,** or by **DPTR**.
If DPTR is used to give an address, then the full 64KB range of External RAM from 0000H… FFFFH is available. This is because DPTR is 16-bit and 2^{16} = 65536.
An "**X**" is present in the instruction, to indicate External RAM.

Eg: MOV**X** A, @DPTR *; A ← [DPTR]^*
; A gets the contents of External RAM Location whose address is given by DPTR
; If DPTR = 2000H, then A gets contents of Location 2000H from External RAM

MOV**X** @DPTR, A *; [DPTR]^ ← A*
; i.e. A is stored at the External RAM Location whose address is given by DPTR

EXTERNAL RAM: (8 BIT ADDRESS GIVEN BY R0 OR R1)

If R0 or R1 is used to give an address, then only the first 256 locations of External RAM is available from 0000 H to 00FF H. This is because R0 or R1 are 8-bit and 2^8 = only 256.

Eg: MOV**X** A, @R0 *; A ← [R0]^*
; i.e. A gets the contents of External RAM Location whose address is given by R0
; If R0 = 25H, then A gets contents of Location 0025H from the External RAM

MOV**X** @R1, A *; [R1]^ ← A*
; i.e. A is stored at the External RAM Location whose address is given by R1

5) INDEXED ADDRESSING MODE

This mode is used to access data from the Code memory (**Internal ROM or External ROM**).
In this addressing mode, address is indirectly specified as a "SUM" of (A and DPTR) or (A and PC)**.**
This is very useful because ROM contains permanent data which is stored in the form of Look Up tables. To access a Look Up table, address is given as a SUM or two registers, where one acts as the base and the other acts as the index within the table.
A "**C**" is present in such instructions, to indicate Code Memory.

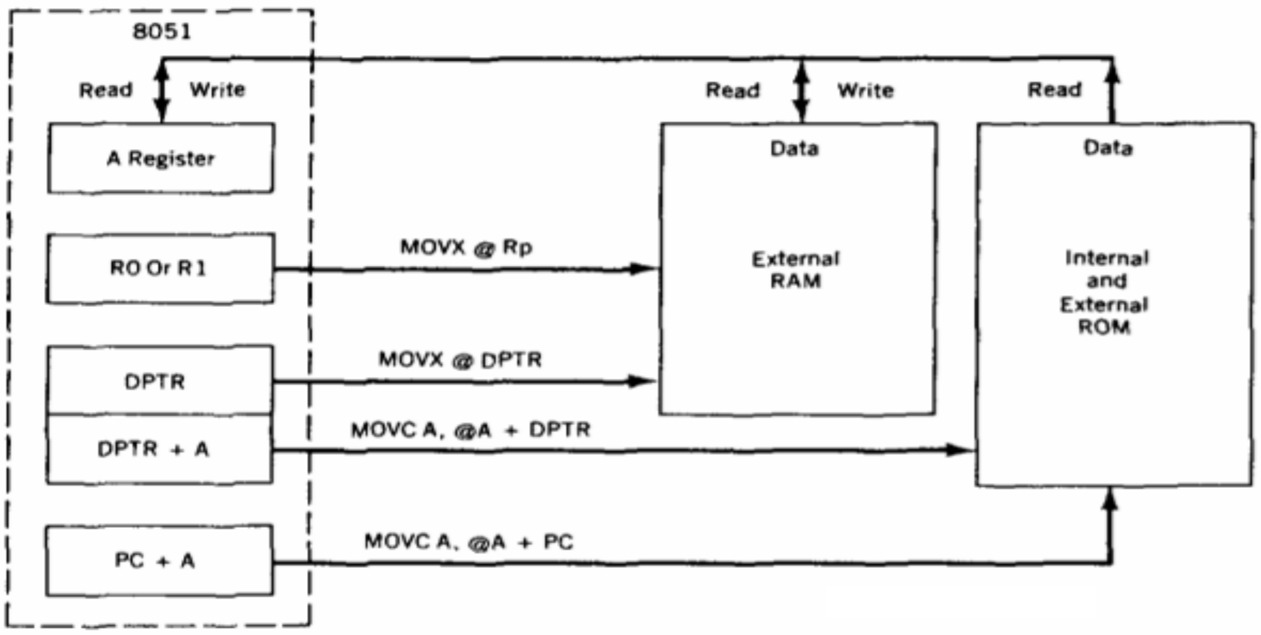
Eg: MOV**C** A, @A+DPTR *; A ← Contents of a ROM Location pointed by A+DPTR.*
; If DPTR = 0400H and A = 05H,
; then A gets the contents of ROM Location whose address is 0405 H.

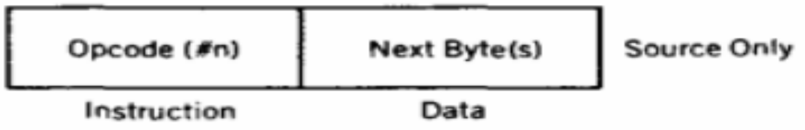
MOV**C** A, @A+PC *; A ← Contents of a ROM Location pointed by A+PC.*

The same instruction may operate on Internal or External ROM, depending upon the address and on the value of $\overline{\textbf{EA}}$ pin of 8051.

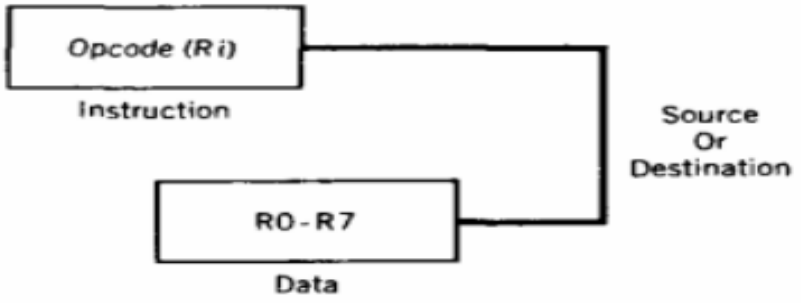
If the address is in the range of 0000... 0FFFH, then $\overline{\textbf{EA}}$ pin will decide if it operates on Internal ROM or External ROM. IF $\overline{\textbf{EA}}$ = 0, External ROM else Internal ROM.
If Address is 1000H and more, it will certainly be External ROM.

External Addressing using MOVX and MOVC

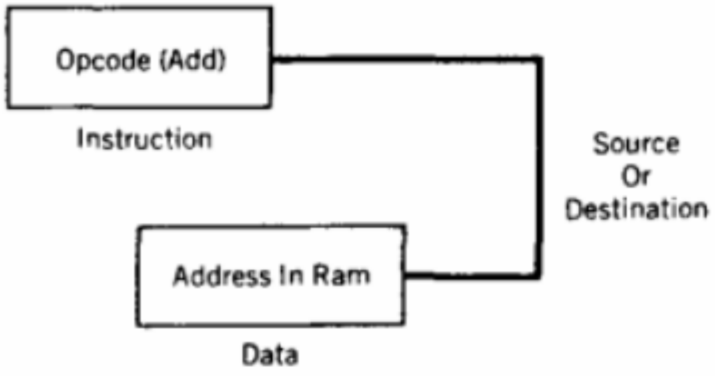




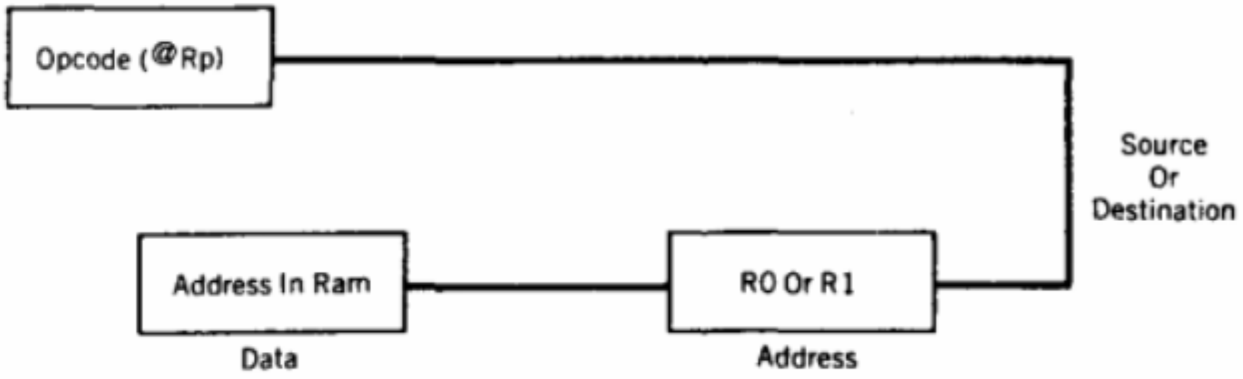
Immediate Addressing Mode



Register Addressing Mode



Direct Addressing Mode



Indirect Addressing Mode

ARITHMETIC INSTRUCTIONS

Here you will see operations like:

- Addition
- Addition with Carry
- Subtraction with Borrow
- Increment
- Decrement
- Multiply
- Divide
- Decimal Adjustment after addition.

1) ADD A, #n | "Add"

Example:

ADD A, #25H; A ← A + 25H

Operation:

Adds A Register with Immediate data.
Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA
Please do remember to check for carry after performing addition.
Carry flag is checked by instructions like JC and JNC.

2) ADD A, Rr | "Add"

Example:

ADD A, R0; A ← A + R0

Operation:

Adds A Register with the value of a RAM register.
Stores the result in A Register.

No of cycles required: **1**

3) ADD A, addr | "Add"

Example:

ADD A, 25H; A ← A + [25H]

Operation:

Adds A Register with the contents of the address.
Stores the result in A Register.

No of cycles required: **1**

4) ADD A, @Rp | "Add"

Example:

ADD A, @R0; A ← A + [R0]

Operation:

Adds A Register with the contents of the location pointed by the register.
Result is stored in A Register.

If R0 = 20H and Location 20H contains value 35H, then 35H will be added to A register.

No of cycles required: **1**

5) ADDC A, #n

| "Add with carry"

Example:

ADDC A, #25H; A ← A + 25H + Carry Flag

Operation:

Adds A Register with Immediate data along with the Carry of the previous addition which is present in the Carry Flag. Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA

ADDC is used when we want to ADD two large numbers like 16 bit numbers.
First we add the lower bytes using ADD instruction.
Then we add the higher bytes using ADDC instruction.
If the Lower byte has produced a Carry, then CF will be 1.
This Carry will be added into the higher bytes.
Please refer to our classroom example of adding 12FFH + 0001H = 1300H.

---

6) ADDC A, Rr

| "Add with carry"

Example:

ADDC A, R0; A ← A + R0 + Carry Flag

Operation:

Adds A Register with the value of a RAM register along with the Carry of the previous addition. Stores the result in A Register.

No of cycles required: **1**

---

7) ADDC A, addr

| "Add with carry"

Example:

ADDC A, 25H; A ← A + [25H] + Carry Flag

Operation:

Adds A Register with the contents of the address along with the Carry of the previous addition. Stores the result in A Register.

No of cycles required: **1**

---

8) ADDC A, @Rp

| "Add with carry"

Example:

ADDC A, @R0; A ← A + [R0] + Carry Flag

Operation:

Adds A Register with the contents of the location pointed by the register along with the Carry of the previous addition. Result is stored in A Register.

No of cycles required: **1**

9) SUBB A, #n | "Subtract with borrow"

Example:

SUBB A, #25H; A ← A - 25H - Carry Flag

Operation:

Performs A Register – Immediate data – Carry Flag (Carry Flag holds the borrow of the previous subtraction).
Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA
SUBB is used when we want to Subtract two large numbers like 16 bit numbers.
First we Subtract the lower bytes.
If the Lower byte subtraction needs a borrow, then CF will be 1.
This Carry will be subtracted from the higher bytes.
It is important to realize that there is no ordinary SUB instruction.
Hence if we want to perform simple 8-bit subtraction, we still have to use SUBB instruction.
If we don't want Carry flag to interfere with the operation,
We must first clear the carry flag using CLR C instruction before using SUBB.

---

10) SUBB A, Rr | "Subtract with borrow"

Example:

SUBB A, R0; A ← A - R0 - Carry Flag

Operation:

Performs A Register – value of RAM register – Carry Flag (Carry Flag holds the borrow of the previous subtraction).
Stores the result in A Register.

No of cycles required: **1**

---

11) SUBB A, addr | "Subtract with borrow"

Example:

SUBB A, 25H; A ← A - [25H] - Carry Flag

Operation:

Performs A Register – contents of memory address – Carry Flag (Carry Flag holds the borrow of the previous subtraction).
Stores the result in A Register.

No of cycles required: **1**

---

12) SUBB A, @Rp | "Subtract with borrow"

Example:

SUBB A, @R0; A ← A - [R0] - Carry Flag

Operation:

Performs A Register – Contents of the memory location pointed by the register – Carry Flag.
Result is stored in A Register.

No of cycles required: **1**

13) INC A | "Increment"

Example:

INC A; A ← A + 1

Operation:

Increments the value of A register. Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA
During INC A, if A was FFH, it will roll over to 00H.

---

14) INC Rr | "Increment"

Example:

INC R0; R0 ← R0 + 1

Operation:

Increments the value of a RAM register. Stores the result in the same RAM Register.

No of cycles required: **1**

---

15) INC addr | "Increment"

Example:

INC 25H; [25H] ← [25H] + 1

Operation:

Increments the contents of a memory address. Stores the result back in the same location.

No of cycles required: **1**

---

16) INC @Rp | "Increment"

Example:

INC @R0; [@R0] ← [@R0] + 1

Operation:

Increments the contents of a memory location pointed by R0. Will store the result back at the same location.

No of cycles required: **1**

---

17) INC DPTR | "Increment"

Example:

INC DPTR; DPTR ← DPTR + 1

Operation:

Increments the 16-bit value of DPTR register. Stores the result in DPTR Register.

No of cycles required: **2**

18) DEC A | "Decrement"

Example:

DEC A; A ← A – 1

Operation:

Decrements the value of A register.
Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA
During DEC A, if A was 00H, it will roll back to FFH.
That's because 00H – 1 = – (01H) which is FFH in 2's complement form.
Please refer to classroom explanation for more clarity on this.
Also, do remember, **DEC DPTR does not exists.**
If you need to decrement DPTR, you can do it by individually decrementing DPL and DPH.
First decrement DPL. If it rolls back from 00H to FFH, then also decrement DPH.
E.g.:: 1300H – 1 = 12FFH.

---

19) DEC Rr | "Decrement"

Example:

DEC R0; R0 ← R0 – 1

Operation:

Decrements the value of a RAM register.
Stores the result in the same RAM Register.

No of cycles required: **1**

---

20) DEC addr | "Decrement"

Example:

DEC 25H; [25H] ← [25H] – 1

Operation:

Decrements the contents of a memory address.
Stores the result back in the same location.

No of cycles required: **1**

---

21) DEC @Rp | "Decrement"

Example:

DEC @R0; [@R0] ← [@R0] – 1

Operation:

Decrements the contents of a memory location pointed by R0.
Will store the result back at the same location.

No of cycles required: **1**

22) MUL AB

| "Multiply A and B"

Example:

MUL AB; B $_{(Higher)}$. A $_{(Lower)}$ ← A x B

Operation:

Multiples the 8-bit values of A register and B register.
Stores the 16 bit result in B and A Registers.
B register gets the Higher Byte, A register gets the Lower Byte.

No of cycles required: **4**

---

23) DIV AB

| "Divide A by B"

Example:

DIV AB; B $_{(Remainder)}$. A $_{(Quotient)}$ ← A ÷ B

Operation:

Divides the 8-bit value of A register by the 8-bit value of B register.
Stores the result in B and A Registers.
B register gets the Remainder, A register gets the Quotient.

No of cycles required: **4**

IMPORTANT TIP FROM BHARAT ACHARYA
During DIV AB, if B register is 00H, then the instruction will be aborted.
A and B registers will contain garbage values after the operation.
This is indicated to the programmer by the **OVERFLOW FLAG**.
If Overflow Flag becomes "1" after the operation, it simply means division by 0 was attempted.

---

24) DA A | "Decimal Adjust after Addition"

Example:

DA A;

Operation:

It is used when we want to add two decimal numbers (BCD numbers).
We first enter the decimal numbers, as if they are Hexadecimal.
We add them by normal ADD instruction.
The answer is then adjusted using DA A instruction.

DA A always works on A Register only.

It first checks the Lower nibble of A Register.
If Lower nibble is > 9 or Aux Carry is 1, then ADD 06H

It then checks the Higher nibble of A Register.
If Higher nibble is > 9 or Carry Flag is 1, then ADD 60H

The final answer will be stored in A and Carry flag.

Please refer numerous examples discussed in the class. For doubts call #BharatSir @9820408217

Assume we want to add 25d + 25d, the result must obviously be 50d.

We enter the numbers as if they are hexadecimal, and add them by normal ADD instruction.

```
MOV A, #25H    ; A ← 25
ADD A, #25H    ; A ← 4A
```

Now we perform the adjustment using DAA instruction.

```
DA A           ; A ← 50
```

	24H		26H		28H		50H		80H		99H
	+ 25H		+ 26H		+ 28H		+ 50H		+ 80H		+ 99H
After ADD	= 49H	After ADD	= 4CH	After ADD	= 50H	After ADD	= A0H	After ADD	= 100H	After ADD	= 132H
After DA A	→ 49	After DA A	→ 52	After DA A	→ 56	After DA A	→ 100	After DA A	→ 160	After DA A	→ 198

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA

Please get this clear, "DA A does not convert any number from Hexadecimal to Decimal".
It simply makes the addition work like decimal addition.

DA A can only be used "After performing an Addition" operation.

VIVA question: Put 25H in A register and show the working of DA A.
Reply: Invalid question! We must first perform addition. Simply putting 25H in A and doing DA A is absurd.

VIVA question: 29H and 3CH and show the working of DA A.
Reply: Invalid question! DA A is used to Add decimal numbers. 3C is not decimal!

Also, do remember, DA A is an adjustment for Addition. So there is always a chance of a carry.
If the answer exceeds 99, the lower two digits will be in A and the highest digit will be Carry Flag.
If the answer is 160, A will contain 60 and Carry Flag will get 1.

LOGIC INSTRUCTIONS

Here you will see operations like:

- AND
- OR
- XOR
- Clear
- Complement
- Rotate
- Rotate with Carry
- Swap
- No Operation

25) ANL A, #n

| "AND logically"

Example:

ANL A, #25H; A ← A AND 25H

Operation:

Logically ANDs the value of A register with the immediate data.
Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA

AND is used to CLEAR any bit of a register.
If we want to clear any bit, we must AND that particular bit with "0" and the remaining bits with "1".
This is because, if we AND anything (0 or 1) with 0, it becomes 0.
But if we AND anything (0 or 1) with 1, it remains the same.

Suppose we want to CLEAR the lower nibble of A register.
Assume: A register is 95H → 1001 0101
AND this register with the number F0H → 1111 0000
As a result A will become 90H → 1001 0000

Please refer examples form Bharat Academy lecture notes for more clarity on this.

---

26) ANL A, Rr

| "AND logically"

Example:

ANL A, R0; A ← A AND R0

Operation:

Logically ANDs the value of A register with the value of RAM register.
Stores the result in A Register.

No of cycles required: **1**

---

27) ANL A, addr

| "AND logically"

Example:

ANL A, 25H; A ← A AND [25H]

Operation:

Logically ANDs the value of A register with contents of the address.
Stores the result in A Register.

No of cycles required: **1**

28) ANL A, @Rp | "AND logically"

Example:

ANL A, @R0; A ← A AND [R0]

Operation:

Logically ANDs the value of A reg. with contents of the location pointed by the reg.
Stores the result in A Register.

No of cycles required: **1**

---

29) ANL addr, A | "AND logically"

Example:

ANL 25H, A; [25H] ← [25H] AND A

Operation:

Logically ANDs contents of the address with the value of A register.
Stores the result at the address.

No of cycles required: **1**

---

30) ANL addr, #n | "AND logically"

Example:

ANL 25H, #30H; [25H] ← [25H] AND 30H

Operation:

Logically ANDs contents of the address with the immediate data.
Stores the result at the address.

No of cycles required: **2**

31) ORL A, #n | "OR logically"

Example:

ORL A, #25H; A ← A OR 25H

Operation:

Logically ORs the value of A register with the immediate data.
Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA
OR is used to SET any bit of a register.
If we want to set any bit, we must OR that particular bit with "1" and the remaining bits with "0".
This is because, if we OR anything (0 or 1) with 1, it becomes 1.
But if we OR anything (0 or 1) with 0, it remains the same.

Suppose we want to SET the lower nibble of A register.
Assume: A register is 95H → 1001 0101
OR this register with the number 0FH → 0000 1111
As a result A will become 9FH → 1001 1111

Please refer examples form Bharat Academy lecture notes for more clarity on this.

---

32) ORL A, Rr | "OR logically"

Example:

ORL A, R0; A ← A OR R0

Operation:

Logically ORs the value of A register with the value of RAM register.
Stores the result in A Register.

No of cycles required: **1**

---

33) ORL A, addr | "OR logically"

Example:

ORL A, 25H; A ← A OR [25H]

Operation:

Logically ORs the value of A register with contents of the address.
Stores the result in A Register.

No of cycles required: **1**

34) ORL A, @Rp | "OR logically"

Example:

ORL A, @R0; A ← A OR [R0]

Operation:

Logically ORs the value of A reg. with contents of the location pointed by the reg.
Stores the result in A Register.

No of cycles required: **1**

---

35) ORL addr, A | "OR logically"

Example:

ORL 25H, A; [25H] ← [25H] OR A

Operation:

Logically ORs contents of the address with the value of A register.
Stores the result at the address.

No of cycles required: **1**

---

36) ORL addr, #n | "OR logically"

Example:

ORL 25H, #30H; [25H] ← [25H] OR 30H

Operation:

Logically ORs contents of the address with the immediate data.
Stores the result at the address.

No of cycles required: **2**

37) XRL A, #n

| "Ex-OR logically"

Example:

XRL A, #25H; A ← A XOR 25H

Operation:

Logically XORs the value of A register with the immediate data.
Stores the result in A Register.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA

XOR is used to COMPLEMENT any bit of a register.
If we want to complement any bit, we must XOR that bit with "1" and the remaining bits with "0".
This is because, if we XOR anything (0 or 1) with 1, it gets complemented.
But if we XOR anything (0 or 1) with 0, it remains the same.

Suppose we want to COMPLEMENT the lower nibble of A register.
Assume: A register is 95H → 1001 0101
XOR this register with the number 0FH → 0000 1111
As a result A will become 9AH → 1001 1010

Please refer examples form Bharat Academy lecture notes for more clarity on this.

---

38) XRL A, Rr

| "Ex-OR logically"

Example:

XRL A, R0; A ← A XOR R0

Operation:

Logically XORs the value of A register with the value of RAM register.
Stores the result in A Register.

No of cycles required: **1**

---

39) XRL A, addr

| "Ex-OR logically"

Example:

XRL A, 25H; A ← A XOR [25H]

Operation:

Logically XORs the value of A register with contents of the address.
Stores the result in A Register.

No of cycles required: **1**

40) XRL A, @Rp | "Ex-OR logically"

Example:

XRL A, @R0; A ← A XOR [R0]

Operation:

Logically XORs the value of A reg. with contents of the location pointed by the reg.
Stores the result in A Register.

No of cycles required: **1**

---

41) XRL addr, A | "Ex-OR logically"

Example:

XRL 25H, A; [25H] ← [25H] XOR A

Operation:

Logically XORs contents of the address with the value of A register.
Stores the result at the address.

No of cycles required: **1**

---

42) XRL addr, #n | "Ex-OR logically"

Example:

XRL 25H, #30H; [25H] ← [25H] XOR 30H

Operation:

Logically XORs contents of the address with the immediate data.
Stores the result at the address.

No of cycles required: **2**

43) RL A | "Rotate Left"

Example:

RL A; A ← A register rotated left by one position

Operation:

Rotates the bits of A register in the left direction by one position.
Each bit goes one position to the left.
MSB goes to the Carry flag as well as to the LSB.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA

Rotates are used to determine the value of any bit, in A register.
To know the value of a bit, Rotate the register as many times, so that the bit comes into Carry Flag.
Now check the carry flag to know if your desired bit was 0 or 1.

---

44) RR A | "Rotate Right"

Example:

RR A; A ← A register rotated right by one position

Operation:

Rotates the bits of A register in the right direction by one position.
Each bit goes one position to the right.
LSB goes to the Carry flag as well as to the MSB.

No of cycles required: **1**

---

45) RLC A | "Rotate Left, with Carry"

Example:

RLC A; A ← A register rotated left by one position along with the carry flag

Operation:

Rotates the bits of A register in the left direction by one position along with the carry flag.
Each bit goes one position to the left.
MSB goes to the Carry flag and Carry Flag goes to the LSB.

No of cycles required: **1**

---

46) RRC A | "Rotate Right, with Carry"

Example:

RRC A; A ← A register rotated right by one position along with the carry flag

Operation:

Rotates the bits of A register in the right direction by one position along with the carry flag.
Each bit goes one position to the right.
LSB goes to the Carry flag and Carry Flag goes to the MSB.

No of cycles required: **1**

47) CPL A | "Complement A"

Example:

CPL A; A ← One's complement of A.

Operation:

Complements the value of A register.
Works just like a Not gate.

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA
CPL is used as a NOT operation.
If we do CPL after AND, it works like **NAND**.
CPL after OR, it works like **NOR**.
CPL after XOR, it works like **XNOR**.

---

48) CLR A | "Clear A"

Example:

CLR A; A ← 00H.

Operation:

Clears the entire A register and makes it 00H.

No of cycles required: **1**

---

49) SWAP A | "SWÃP A"

Example:

SWAP A; A $_{\text{Lower Nibble}}$ **←→ A** $_{\text{Higher Nibble}}$

Operation:

Interchanges the Nibbles of A register.
If A register was 35H it will become 53H.
It is as good as rotating A register four times.

No of cycles required: **1**

---

50) NOP | "No operation"

Example:

NOP; No operation. PC simply becomes PC + 1.

Operation:

This instruction performs no operation.
It is typically used to produce a delay.

No of cycles required: **1**

BOOLEAN INSTRUCTIONS

Here you will see bit wise operations like:

- Set
- Clear
- Complement
- Movement between a bit and Carry flag
- Logical AND
- Logical OR

IMPORTANT TIP FROM BHARAT ACHARYA
These are bit wise operations.
"b" represents any bit from the bit addressable area of the internal RAM having bit addresses 00H… 7FH.
"b" can also be a bit of any bit addressable SFR like PSW.3 or P0.7 etc.

---

100) SETB C | "SET the carry Flag"

Example:

SETB C; Carry Flag ← 1

Operation:

This instruction is used to SET the Carry Flag.
This makes Carry Flag ← 1

No of cycles required: **1**

---

101) CLR C | "CLEAR the carry Flag"

Example:

CLR C; Carry Flag ← 1

Operation:

This instruction is used to CLEAR the Carry Flag.
This makes Carry Flag ← 0

No of cycles required: **1**

---

102) CPL C | "COMPLEMENT the carry Flag"

Example:

CPL C; Carry Flag ← complement of Carry Flag

Operation:

This instruction is used to COMPLEMENT the Carry Flag.
This makes Carry Flag ← Complement of Carry Flag

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA
These are bit wise operations.
"b" represents any bit from the bit addressable area of the internal RAM having bit addresses 00H… 7FH.
"b" can also be a bit of any bit addressable SFR like PSW.3 or P0.7 etc.

---

103) SETB b | "SET a bit"

Example:

SETB P0.2; P0.2 ← 1

SETB 00H; bit location 00H of Internal RAM gets the value ← 1

Operation:

This instruction is used to SET the any bit "b".
It can be a bit from the bit addressable area of the Internal RAM having bit address 00H… 7FH
It can also be a bit from a bit addressable SFR like TCON.4

No of cycles required: **1**

---

104) CLR b | "CLEAR a bit"

Example:

CLR P0.2; P0.2 ← 0

CLR 00H; bit location 00H of Internal RAM gets the value ← 0

Operation:

This instruction is used to CLEAR the any bit "b".
It can be a bit from the bit addressable area of the Internal RAM having bit address 00H… 7FH
It can also be a bit from a bit addressable SFR like TCON.4

No of cycles required: **1**

---

105) CPL b | "COMPLEMENT a bit"

Example:

CPL P0.2; P0.2 gets complemented

CPL 00H; bit location 00H of Internal RAM gets the value gets complemented

Operation:

This instruction is used to COMPLEMENT the any bit "b".
It can be a bit from the bit addressable area of the Internal RAM having bit address 00H… 7FH
It can also be a bit from a bit addressable SFR like TCON.4

No of cycles required: **1**

106) MOV C, b

"Move a bit into the Carry Flag"

Example:

MOV C, P0.2; Carry Flag ← P0.2

MOV C, 25H; Carry Flag ← [25H] bit address

Operation:

This instruction is used to copy the value of some bit into the Carry Flag.
It can be a bit from the bit addressable area of the Internal RAM having bit address 00H… 7FH
It can also be a bit from a bit addressable SFR like TCON.4

No of cycles required: **1**

IMPORTANT TIP FROM BHARAT ACHARYA
There is a clear difference between the following two instructions:
MOV A, 25H; This refers to Byte Location 25H.
We can make this conclusion because A is an 8-bit register.
MOV C, 25H; This refers to Bit Location 25H.
We can make this conclusion because C is the Carry Flag.

---

107) MOV b, C

"Move a from Carry flag into a bit"

Example:

MOV P0.2, C; P0.2 ← Carry Flag

MOV 25H, C; [25H] bit address **← Carry Flag**

Operation:

This instruction is used to copy the value of the Carry Flag into some bit.
It can be a bit from the bit addressable area of the Internal RAM having bit address 00H… 7FH
It can also be a bit from a bit addressable SFR like TCON.4

No of cycles required: **2**

IMPORTANT TIP FROM BHARAT ACHARYA
There is a clear difference between the following two instructions:
MOV 25H, A; This refers to Byte Location 25H.
We can make this conclusion because A is an 8-bit register.
MOV 25H, C; This refers to Bit Location 25H.
We can make this conclusion because C is the Carry Flag.

108) ANL C, b

| "AND Carry Flag with a bit"

Example:

ANL C, P0.2; Carry Flag ← Carry Flag AND P0.2

ANL C, 25H; Carry Flag ← Carry Flag AND [25H] bit address

Operation:

This instruction is used to AND the value of some bit with the Carry Flag.
It can be a bit from the bit addressable area of the Internal RAM having bit address 00H… 7FH
It can also be a bit from a bit addressable SFR like TCON.4
The result will be stored in the Carry Flag
It is interesting to note that, in bit wise operations, Carry Flag acts as the "Accumulator".

No of cycles required: **2**

---

109) ANL C, /b

| "AND Carry Flag with the COMPLEMENT of a bit"

Example:

ANL C, P0.2; Carry Flag ← Carry Flag AND (NOT) P0.2

ANL C, 25H; Carry Flag ← Carry Flag AND (NOT) [25H] bit address

Operation:

This instruction is used to AND the value of Carry Flag with the COMPLEMENT of a bit.

No of cycles required: **2**

---

108) ORL C, b

| "OR Carry Flag with a bit"

Example:

ORL C, P0.2; Carry Flag ← Carry Flag OR P0.2

ORL C, 25H; Carry Flag ← Carry Flag OR [25H] bit address

Operation:

This instruction is used to OR the value of some bit with the Carry Flag.

No of cycles required: **2**

---

109) ORL C, /b

| "OR Carry Flag with the COMPLEMENT of a bit"

Example:

ORL C, P0.2; Carry Flag ← Carry Flag OR (NOT) P0.2

ORL C, 25H; Carry Flag ← Carry Flag OR (NOT) [25H] bit address

Operation:

This instruction is used to OR the value of Carry Flag with the COMPLEMENT of a bit.

No of cycles required: **2**

DATA TRANSFER INSTRUCTIONS

Here you will see operations like:

- MOV
- MOVX
- MOVC
- PUSH
- POP
- Exchange
- Exchange Digit

51) MOV A, #n | "Move"

Example:

MOV A, #25H; A ← 25H

Operation:

A Register gets the value of the Immediate data.

No of cycles required: **1**

---

52) MOV A, Rr | "Move"

Example:

MOV A, R0; A ← R0

Operation:

A Register gets the value of a RAM register.
The value remains in the RAM register and is also copied into A register.

No of cycles required: **1**

---

53) MOV A, addr | "Move"

Example:

MOV A, 25H; A ← [25H]

Operation:

A Register gets the contents of the address.

No of cycles required: **1**

---

54) MOV A, @Rp | "Move"

Example:

MOV A, @R0; A ← [R0]

Operation:

A Register gets the contents of the location pointed by the register.

If R0 = 20H and Location 20H contains value 35H, then 35H will be copied into to A register.

No of cycles required: **1**

55) MOV Rr, A | "Move"

Example:

MOV R0, A; R0 ← A

Operation:

RAM Register gets the value of A Register.

No of cycles required: **1**

---

56) MOV Rr, #n | "Move"

Example:

MOV R0, #25H; R0 ← 25H

Operation:

RAM Register gets the value of the Immediate Data.

No of cycles required: **1**

---

57) MOV Rr, addr | "Move"

Example:

MOV R0, 25H; R0 ← [25H]

Operation:

RAM Register gets the contents of the address.

No of cycles required: **2**

58) MOV addr, A | "Move"

Example:

MOV 25H, A; [25H] ← A

Operation:

The RAM location gets the value of A Register.

No of cycles required: **1**

---

59) MOV addr, #n | "Move"

Example:

MOV 25H, #25H; [25H] ← 25H

Operation:

The RAM location gets the value of the Immediate Data.

No of cycles required: **2**

---

60) MOV addr, Rr | "Move"

Example:

MOV 25H, R0; [25H] ← R0

Operation:

The RAM location gets the contents of the RAM register.

No of cycles required: **2**

---

61) MOV addr1, addr2 | "Move"

Example:

MOV 30H, 25H; [30H] ← [25H]

Operation:

The RAM location 1 gets the contents of RAM location 2.

No of cycles required: **2**

---

62) MOV addr, @Rp | "Move"

Example:

MOV 25H, @R0; [25H] ← [R0]

Operation:

The RAM location gets the contents of the address pointed by the register.

No of cycles required: **2**

63) MOV @Rp, A | "Move"

Example:

MOV @R0, A; [R0] ← A

Operation:

The RAM location pointed by the register gets the value of A Register.

No of cycles required: **1**

---

64) MOV @Rp, #n | "Move"

Example:

MOV @R0, #25H; [R0] ← 25H

Operation:

The RAM location pointed by the register gets the value of the Immediate Data.

No of cycles required: **1**

---

65) MOV @Rp, addr | "Move"

Example:

MOV @R0, 25H; [R0] ← [25H]

Operation:

The RAM location pointed by the register gets the contents of the address.

No of cycles required: **2**

---

66) MOV DPTR, #nn | "Move"

Example:

MOV DPTR, #2000H; DPTR ← 2000H

Operation:

DPTR register gets the 16 bit Immediate Data given in the instruction.

No of cycles required: **2**

67) MOVX A, @Rp | "Move Ex"

Example:

MOVX A, @R0; A ← [R0]^

Operation:

A Register gets the data from the location pointed by R0 in the External RAM.
If R0 = 20H then A Register gets data from External RAM location 0020H.

No of cycles required: **2**

---

68) MOVX A, @DPTR | "Move Ex"

Example:

MOVX A, @DPTR; A ← [DPTR]^

Operation:

A Register gets the data from the location pointed by DPTR in the External RAM.
If DPTR = 4000H then A Register gets data from External RAM location 4000H.

No of cycles required: **2**

---

69) MOVX @Rp, A | "Move Ex"

Example:

MOVX @R0, A; [R0]^ ← A

Operation:

Location pointed by R0 in the External RAM gets the data from A Register.
If R0 = 20H then value of A Register gets stored into External RAM location 0020H.

No of cycles required: **2**

---

70) MOVX @DPTR, A | "Move Ex"

Example:

MOVX @DPTR, A; [DPTR]^ ← A

Operation:

Location pointed by R0 in the External RAM gets the data from A Register.
If DPTR = 4000H then value of A Register gets stored into External RAM location 4000H.

No of cycles required: **2**

IMPORTANT TIP FROM BHARAT ACHARYA
When working with External RAM, remember the following rules:
1) We cannot use direct addressing mode.
We can ONLY use Indirect addressing using R0/ R1 or DPTR.
2) All data can only be transferred to or from "A" register.
MOVX operates only on A Register.
3) We can only perform data transfers with these memories.
We can NOT directly perform ADD or SUB etc.
Such operations are only allowed with Internal RAM.

71) MOVC A, @A + DPTR | "Move C"

Example:

MOVC A, @A + DPTR; A 🡐 [A + DPTR]$_{ROM}$

Operation:

A Register gets the data from the location pointed by A + DPTR in ROM.
If A = 05H and DPTR = 0400H, then A Register gets data from ROM location 0405H.

This operation can happen either on Internal ROM or External ROM.
If the address formed after adding A + DPTR is 1000H or more, the operation will happen on External ROM.

If the address is less than 1000H, then it depends upon $\overline{EA}$ pin.

If $\overline{EA}$ pin = 0, operation will happen on External ROM.

If $\overline{EA}$ pin = 1, operation will happen on Internal ROM.

No of cycles required: **2**

IMPORTANT TIP FROM BHARAT ACHARYA
This operation can happen either on Internal ROM or External ROM.
If the address formed after adding A + DPTR is 1000H or more,
the operation will happen on External ROM.

If the address is less than 1000H,
then it depends upon $\overline{EA}$ pin.

If $\overline{EA}$ pin = 0,
operation will happen on External ROM.
If $\overline{EA}$ pin = 1,
operation will happen on Internal ROM.

This instruction is extremely useful in accessing Look Up Tables.
DPTR is initialized with the starting address of the table. E.g.:: 0400H.
The required index in the table is initialized in A register. E.g.:: 05H.
By doing MOVC A, @ A+DPTR, A register gets the value from element 5, of table staring at 0400H.
Please recollect the 7-Segment code translation example from the classroom lecture.

---

71) MOVC A, @A + PC | "Move C"

Example:

MOVC A, @A + PC; A 🡐 [A + PC]$_{ROM}$

Operation:

A Register gets the data from the location pointed by A + PC in ROM.
If A = 25H and PC = 0400H, then A Register gets data from ROM location 0425H.

No of cycles required: **2**

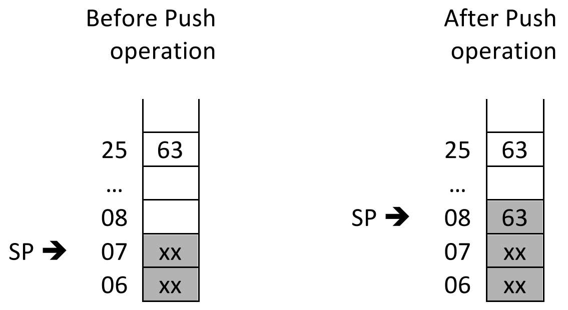
73) PUSH addr | "Push"

Example:

PUSH 25H; First SP 🡐 SP + 1, Then [SP] 🡐 [25H]

Operation:

This instruction is used to push new data into the top of stack.
First SP will become SP + 1
Then at the new location pointed by SP, data from the specified address will be pushed.
This newly pushed data will now become the top of stack.



No of cycles required: **2**

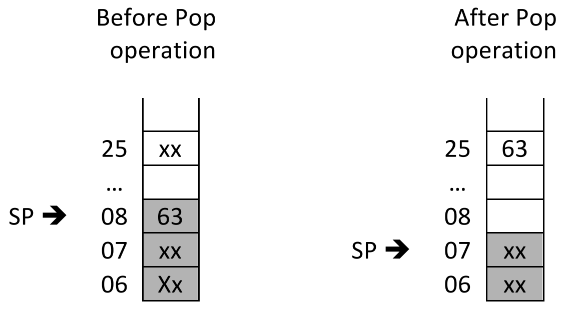
74) POP addr | "Pop"

Example:

POP 25H; First [25H] 🡐 [SP], Then SP 🡐 SP - 1

Operation:

This instruction is used to pop data from the top of Stack and store it into the desired RAM location.
First data from the top of Stack will be popped and stored into the desired RAM location.
Then SP will become SP - 1
The data now pointed by SP becomes the new top of stack.



No of cycles required: **2**

75) XCH A, Rr | "Exchange"

Example:

XCH A, R0; A ←→ R0

Operation:

It will interchange the values of A Register and the specified RAM Register.
Suppose A Register contains 34H and R0 contains 67H, then after the operation,
A will get 67H and R0 will get 34H.

No of cycles required: **1**

---

76) XCH A, addr | "Exchange"

Example:

XCH A, 25H; A ←→ [25H]

Operation:

It will interchange the values of A Register and the contents of the specified RAM location.
Suppose A Register contains 34H and location 25H contains 67H, then after the operation,
A will get 67H and location 25H will get 34H.

No of cycles required: **1**

---

77) XCH A, @Rp | "Exchange"

Example:

XCH A, @R0; A ←→ [R0]

Operation:

It will interchange the values of A Register and the contents of the location pointed by the Register.
Suppose A Register contains 34H and location pointed by R0 contains 67H, then after the operation,
A will get 67H and location pointed by R0 will get 34H.

No of cycles required: **1**

---

78) XCHD A, @Rp | "Exchange Digit"

Example:

XCHD A, @R0; A $_{\text{Lower Nibble}}$ **←→ [R0]** $_{\text{Lower Nibble}}$

Operation:

It will interchange only the lower nibbles of A Register and the contents of the location pointed by the Register.
Since only one digit is exchanged it is called Exchange Digit.
Suppose A Register contains 34H and location pointed by R0 contains 67H, then after the operation,
A will get 37H and location pointed by R0 will get 64H.

No of cycles required: **1**

BRANCH CONTROL INSTRUCTIONS

<u>Here you will see operations like:</u>

- SJMP, AJMP & LJMP
- ACALL & LCALL
- RET & RETI
- CJNE
- DJNZ
- JC & JNC
- JZ & JNZ
- JB, JNB & JBC

BRANCH OPERATIONS OF 8051 (SJMP, AJMP AND L JMP)

SHORT JUMP

Syntax: **SJMP radd**; // Short Jump using the relative address

Range: **(-128 … +127) locations** because "radd" is an 8-bit signed number

Size of instruction: **2 Bytes** (Opcode of SJMP= 1Byte, radd = 1Byte)

New address calculation: **PC ← PC (address of next instruction) + radd**

Usage: **SJMP** (unconditional) and ALL Conditional jumps like JC, JNC, CJNE, DJNZ etc. ← Important for VIVA

Description: Here the branch address (radd) is calculated as a relative distance from the next instruction to the branch location. In simple terms, instead of telling where we want to jump, we are telling how far we want to jump. This "radd" is then added to PC, which normally contains address of the next instruction. ***For examples of SJMP, please refer #BharatSir Lecture Notes***

ABSOLUTE JUMP

Syntax: **AJMP sadd**;// Absolute Jump using the short address

Range: **max 2KB** as long as the Jump is within the **Same Page**

Size of instruction: **2 Bytes** (Opcode of AJMP= 1Byte, sadd = 1Byte)

New address calculation:

PC ←	**PC**	**Opcode of AJMP**	**Sadd**
(16)	5 bits Remains the same as branch is in the same page	3 bits Hence AJMP has 8 opcodes	8 bits Lower 8 bits of the jump location

Usage: **AJMP and ACALL.**

Description: *Here the entire program memory (64 KB), is divided into 32 pages, each page being of 2KB. We can jump to any location of the same page, giving a max range of 2 KB. As the jump is in the same page, only the lower 11 bits of the address will change. Out of them, lower 8 bits are given by "sadd" and the higher 3 bits are given by the opcode of AJMP. 3 bits have 8 combinations, hence AJMP has 8 opcodes.* ***For examples of AJMP, please refer #BharatSir Lecture Notes***

LONG JUMP

Syntax: **LJMP ladd**; // Long Jump using the long (full) address

Range: **64 KB** because "ladd" is a 16-bit address so can be any value from 0000H… FFFFH.

Size of instruction: **3 Bytes** (Opcode of LJMP= 1Byte, ladd = 2Bytes)

New address calculation: **PC ← ladd**

Usage: **LJMP, LCALL.**

Description: *This is the simplest type of Jump. Here we simply give the address where we wish to jump using "ladd". This "ladd" is then simply put into PC.* ***For examples of LJMP, please refer #BharatSir Lecture Notes***

	Jump operation	**Call operation**
1	In a Jump, we simply branch to the new location and continue from there on.	In a Call, we branch to the new location, which is basically a subroutine, we execute the subroutine, and at the end, we return to the main program right at the NEXT instruction after the Call instruction.
2	There is no intention to return to the previous location..	To invoke the subroutine we use Call instruction. To return to the main program we use RET instruction.
3	There is no need for storing any return address into the stack	During Call, the return address (PC), is pushed into the stack. During RET, the return address is popped from the stack and put back into PC.
4	Jumps can be of three types: Short, Absolute or Long Jump.	Call can be of two types: Absolute or Long Call.
5	Furthermore, Jumps can be unconditional or conditional.	Calls are always unconditional in 8051.

	RET	**RETI**
1	RET is used at the end of an ordinary Subroutine	RETI is used at the end of an ISR
2	RET will simply return back to the next instruction of the main program.	RETI will not only return back to the next instruction of the main program, but will also re-enable the interrupts, by making EA bit 🡐 1.
3	Operation: **POP PC;** PCH🡐 [SP] PCL🡐 [SP-1] SP🡐 SP-2	Operation: **POP PC;** PCH🡐 [SP] PCL🡐 [SP-1] SP🡐 SP-2 **EA 🡐1;** Enables interrupts by making EA bit "1" in the IE-SFR.

79) SJMP radd
| "Short jump to relative address"

Example:

SJMP label;

Operation:

Program takes a short jump to the location "label".
Please refer to the earlier discussion for a detailed explanation on SJMP, AJMP and LJMP.
They are VERY IMPORTANT for Theory, Practicals and VIVA exams.

No of cycles required: **2**

---

80) AJMP sadd
| "Absolute jump to short address"

Example:

AJMP label;

Operation:

Program takes an absolute jump to the location "label".
Please refer to the earlier discussion for a detailed explanation on SJMP, AJMP and LJMP.
They are VERY IMPORTANT for Theory, Practicals and VIVA exams.

No of cycles required: **2**

---

81) LJMP ladd
| "Long jump to long address"

Example:

LJMP label;

Operation:

Program takes a long jump to the location "label".
Please refer to the earlier discussion for a detailed explanation on SJMP, AJMP and LJMP.
They are VERY IMPORTANT for Theory, Practicals and VIVA exams.

No of cycles required: **2**

---

82) JMP @A+DPTR
| "Jump"

Example:

JMP @A+DPTR;

Operation:

Program takes a jump to the location whose address is calculated by A + DPTR.
If A = 25H and DPTR is 1000H then the program will jump to location 1025H by putting 1025H into PC.

No of cycles required: **2**

CONDITIONAL JUMPS

A conditional jump instruction can have two possible executions, depending upon the condition.
If condition is true, program will jump to the branch location, specified by the label.
If condition is false, program will simply proceed to the next instruction.
Please do remember, **All conditional jumps are SHORT type of jumps.**

---

87) DJNZ Rr, radd | "Decrement and Jump if Not Zero"

Example:

DJNZ R7, Back; Decrement R7 and if not equal to "0", go to "Back"

Operation:

This instruction is extremely useful.
It is used by the programmer to create loops.
We initialize the loop count in some register like R7.
At the end of the loop we write this instruction: "DJNZ R7, Back"
Back is the Label where we wish to begin the loop from.
This instruction will first decrement R7.
It will then check if R7 has become 0 or not.
If R7 is NOT ZERO, program will jump to the location Back.
This will go on happening till R7 finally becomes Zero.
That's when the loop will break and program will simply proceed to the next instruction.
The count could have been in any register from R0… R7.
Get this clear, whatever is the count, we get the exact same number of iterations.
Neither one more, nor one less.
A count of 5 will produce exactly 5 iterations, neither 4 and nor 6.
Hope you remember various cases we discussed at BHARAT ACADEMY, for minimum and maximum number of iterations.

No of cycles required: **2**

---

88) DJNZ addr, radd | "Decrement and Jump if Not Zero"

Example:

DJNZ 25H, Back; Decrement the contents of location 25H, and if not equal to "0", go to "Back"

Operation:

This instruction is again used to make loops just like the previous one.
Except that, now the count is in a memory address, instead of a register.
Say, we initialize the loop count in some RAM location like 25H.
At the end of the loop we write this instruction: "DJNZ 25H, Back"
This instruction will first decrement contents of location 25H.
It will then check if the value has become 0 or not.
Based on that, it will do the iterations the same way as explained in the above instruction.

No of cycles required: **2**

---

89) CJNE A, #n, radd | "Compare and Jump if Not Equal"

Example:

CJNE A, #25H, Down; Compare A with number 25H. If not equal jump to location "Down"

Operation:

This instruction will compare A register with the number 25H.
If they are NOT EQUAL, program will jump to location "Down", else will simply proceed to the next instruction.

No of cycles required: **2**

---

90) CJNE A, addr, radd | "Compare and Jump if Not Equal"

Example:

CJNE A, 25H, Down; Compare A with [25H]. If not equal jump to location "Down"

Operation:

This instruction will compare A register with the contents of location 25H.
If they are NOT EQUAL, program will jump to location "Down", else will simply proceed to the next instruction.

No of cycles required: **2**

---

91) CJNE Rr, #n, radd | "Compare and Jump if Not Equal"

Example:

CJNE R0, #25H, Down; Compare R0 with number 25H. If not equal jump to location "Down"

Operation:

This instruction will compare R0 register with the number 25H.
If they are NOT EQUAL, program will jump to location "Down", else will simply proceed to the next instruction.

No of cycles required: **2**

---

92) CJNE @Rp, #n, radd | "Compare and Jump if Not Equal"

Example:

CJNE @R0, #25H, Down; Compare [R0] with number 25H. If not equal jump to location "Down"

Operation:

This instruction will compare the contents of the location pointed by R0, with the number 25H.
If they are NOT EQUAL, program will jump to location "Down", else will simply proceed to the next instruction.

No of cycles required: **2**

93) JC radd | "Jump if Carry"

Example:

JC Down; If Carry flag is "1", jump to location "Down"

Operation:

This instruction is used to check the Carry flag.
If Carry flag = "1", program will jump to the location "Down"
Else, program will simply proceed to the next location.
JC and JNC are used very frequently in programs where carry is involved.

No of cycles required: **2**

---

94) JNC radd | "Jump if No Carry"

Example:

JNC Down; If Carry flag is "0", jump to location "Down"

Operation:

This instruction is used to check the Carry flag.
If Carry flag = "0", program will jump to the location "Down"
Else, program will simply proceed to the next location.

No of cycles required: **2**

---

95) JZ radd | "Jump if Zero"

Example:

JZ Down; If A register is "Zero", jump to location "Down"

Operation:

This instruction is used to check the value of A Register.
If A Register = "0", program will jump to the location "Down"
Else, program will simply proceed to the next location.
In other processors, JZ and JNZ operate on Zero flag.
Since 8051 does not have a Zero Flag, these instructions simply check the value of A Register.

No of cycles required: **2**

---

96) JNZ radd | "Jump if Not Zero"

Example:

JNZ Down; If A register is "Not Zero", jump to location "Down"

Operation:

This instruction is used to check the value of A Register.
If A Register ≠ "0", program will jump to the location "Down"
Else, program will simply proceed to the next location.

No of cycles required: **2**

BOOLEAN CONDITIONAL INSTRUCTIONS

These instructions check the value if individual bits to decide whether to Jump or Not.
They are very useful in Embedded system Application programming of Timers, Serial Port etc.

Often this sub-group is asked as an independent 5 mark question in the exam.

---

97) JB bit, radd

| "Jump if Bit"

Example:
JB P0.0, Down; If P0.0 = "1", jump to location "Down"

Operation:
This instruction is used to check the value of a bit.
Suppose the bit in question is P0.0
If P0.0 = "1", then jump to location "Down"
Else, simply proceed top next location.

No of cycles required: **2**

---

98) JNB bit, radd

| "Jump if Not Bit"

Example:
JNB P0.0, Down; If P0.0 = "0", jump to location "Down"

Operation:
This instruction is used to check the value of a bit.
Suppose the bit in question is P0.0
If P0.0 = "0", then jump to location "Down"
Else, simply proceed top next location.

No of cycles required: **2**

---

99) JBC bit, radd

| "Jump if Bit and complement/ clear"

Example:
JBC P0.0, Down; If P0.0 = "1", jump to location "Down" and make P0.0 ← "0"

Operation:
This instruction is used to check the value of a bit.
Suppose the bit in question is P0.0
If P0.0 = "1", then jump to location "Down", and while jumping, also make P0.0 ← 0
Else, simply proceed top next location.

No of cycles required: **2**

BRANCH OPERATIONS OF 8051 (SJMP, AJMP AND L JMP)

SHORT JUMP

Syntax: **SJMP radd**; // Short Jump using the relative address

Range: **(-128 … +127) locations** because "radd" is an 8-bit signed number

Size of instruction: **2 Bytes** (Opcode of SJMP= 1Byte, radd = 1Byte)

New address calculation: **PC ← PC (address of next instruction) + radd**

Usage: **SJMP** (unconditional) and ALL Conditional jumps like JC, JNC, CJNE, DJNZ etc. ← Important for VIVA

Description: Here the branch address (radd) is calculated as a relative distance from the next instruction to the branch location. In simple terms, instead of telling where we want to jump, we are telling how far we want to jump. This "radd" is then added to PC, which normally contains address of the next instruction. ***For examples of SJMP, please refer #BharatSir Lecture Notes***

ABSOLUTE JUMP

Syntax: **AJMP sadd**;// Absolute Jump using the short address

Range: **max 2KB** as long as the Jump is within the **Same Page**

Size of instruction: **2 Bytes** (Opcode of AJMP= 1Byte, sadd = 1Byte)

New address calculation:

PC ←	**PC**	**Opcode of AJMP**	**Sadd**
(16)	5 bits	3 bits	8 bits
	Remains the same as branch is in the same page	Hence AJMP has 8 opcodes	Lower 8 bits of the jump location

Usage: **AJMP and ACALL.**

Description: Here the entire program memory (64 KB), is divided into 32 pages, each page being of 2KB. We can jump to any location of the same page, giving a max range of 2 KB. As the jump is in the same page, only the lower 11 bits of the address will change. Out of them, lower 8 bits are given by "sadd" and the higher 3 bits are given by the opcode of AJMP. 3 bits have 8 combinations, hence AJMP has 8 opcodes. ***For examples of AJMP, please refer #BharatSir Lecture Notes***

LONG JUMP

Syntax: **LJMP ladd**; // Long Jump using the long (full) address

Range: **64 KB** because "ladd" is a 16-bit address so can be any value from 0000H… FFFFH.

Size of instruction: **3 Bytes** (Opcode of LJMP= 1Byte, ladd = 2Bytes)

New address calculation: **PC ← ladd**

Usage: **LJMP, LCALL.**

Description: This is the simplest type of Jump. Here we simply give the address where we wish to jump using "ladd". This "ladd" is then simply put into PC. ***For examples of LJMP, please refer #BharatSir Lecture Notes***

INTERRUPTS OF 8051

8051 supports **5** interrupts.
2 External Interrupts are on the following pins

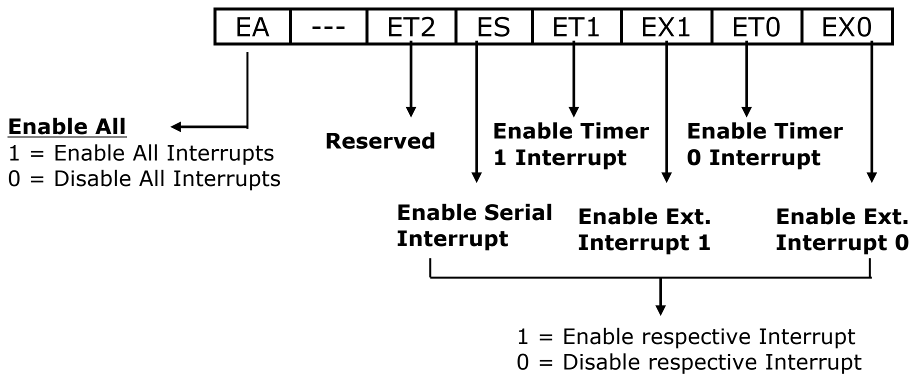
- **$\overline{\text{INT1}}$**
- **$\overline{\text{INT0}}$**

2 Internal Timer interrupts are:

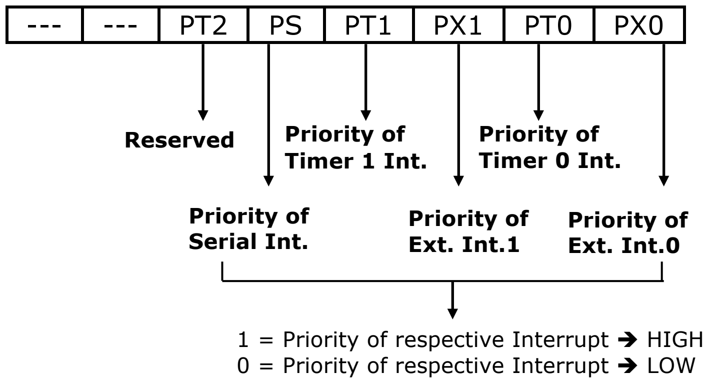
- Timer 1 Overflow Interrupt
- Timer 0 Overflow Interrupt

1 Serial Port Interrupt (Common for RI or TI)
All interrupts are **vectored** i.e. they cause the program to execute an ISR from a pre-determined address in the Program Memory. #Please refer Bharat Sir's Lecture Notes for this ...
Interrupts are controlled mainly by **IE** and **IP** SFR's and also by some bits of **TCON** SFR.

IE - Interrupt Enable (SFR) [Bit-Addressable As IE.7 to IE.0]



IP - Interrupt Priority (SFR) [Bit-Addressable As IP.7 to IP.0]



Timer Overflow Interrupts (TF1 and TF0)

When any of the 2 **Timers overflow**, their respective bit **TFX** (TF1 or TF0) is **set** in **TCON** SFR.
If Timer Interrupts are enabled then the **timer interrupt occurs**.
The **TFX** bits are **cleared** when their respective **ISR** is executed.

Serial Port Interrupt (RI or TI)

While receiving serial data, when a **complete byte** is **received** the **RI** (receive interrupt) bit is set in the **SCON.**
During transmission, when a **complete byte** is **transmitted** the **TI** (transmit interrupt) bit is set in the **SCON**.
ANY of these events can cause the **Serial Interrupt** (provided Serial Interrupt is enabled).
The **RI/TI** bit is **not cleared** automatically on **executing** the **ISR**. The program should **explicitly clear** this bit to allow further Serial Interrupts.

External Interrupts ($\overline{\text{INT1}}$ and $\overline{\text{INT0}}$)

Pins $\overline{\textbf{INT1}}$ and $\overline{\textbf{INT0}}$ are inputs for external interrupts.

These interrupts can be -ve **edge** or low-**level triggered** depending upon the **IT0 and IT1** bit in **TCON** SFR. (ITX = 1 ➔ -ve edge triggered)
Whew any of these interrupts occur the respective bits **TE1** or **IE0** are **set** in the **TCON** SFR.
If External Interrupts are enabled then the **ISR** is **executed** from the respective address.

Interrupt Sequence

The following sequence is executed to service an interrupt:
Address of next instruction of the main program i.e. **PC** is **Pushed** into the **Stack**.
All **interrupts** are **disabled**, by making EA bit in IE SFR 🡐 0.
Program Control is shifted to the **Vector Address** (location) of the **ISR**.
The **ISR begins**.

Returning Sequence

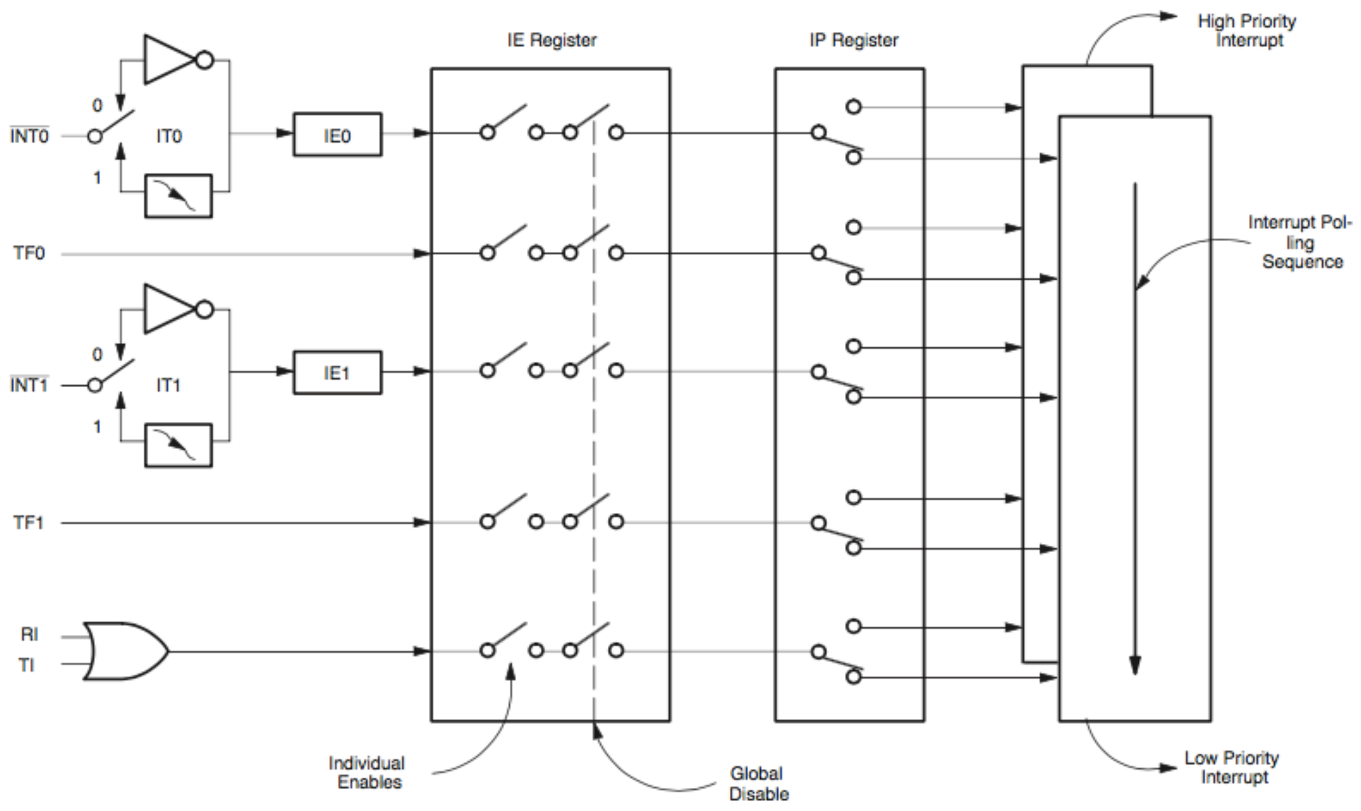
RETI instruction denotes the **end** of the **ISR**.
It causes the processor to **POP** the contents of the Stack Top into the **PC.**
It also re-enables interrupts by making EA bit in IE SFR 🡐 1.
The **main program resumes**.

Interrupt Priorities

8051 has only **two priority levels** for the interrupts: **Low** and **High**.
Interrupt priorities are set using the **IP** SFR.
As the name suggests, a high priority interrupt can interrupt a low priority interrupt.
It two or more interrupts at the same level occur simultaneously then priorities are decided as follows:

INTERRUPT	**PRIORITY**	**VECTOR ADDRESS**
$\overline{\text{INT0}}$	1	00**03**H
TF0	2	00**0B**H
$\overline{\text{INT1}}$	3	00**13**H
TF1	4	00**1B**H
Serial (RI or TI)	5	00**23**H

DIAGRAM FOR INTERRUPTS... OPTIONAL
DRAW ONLY IF ASKED


IE Register
IP Register
High Priority Interrupt
INT0
0
1
IT0
IE0
TF0
INT1
0
1
IT1
IE1
TF1
RI
TI
Interrupt Polling Sequence
Individual Enables
Global Disable
Low Priority Interrupt