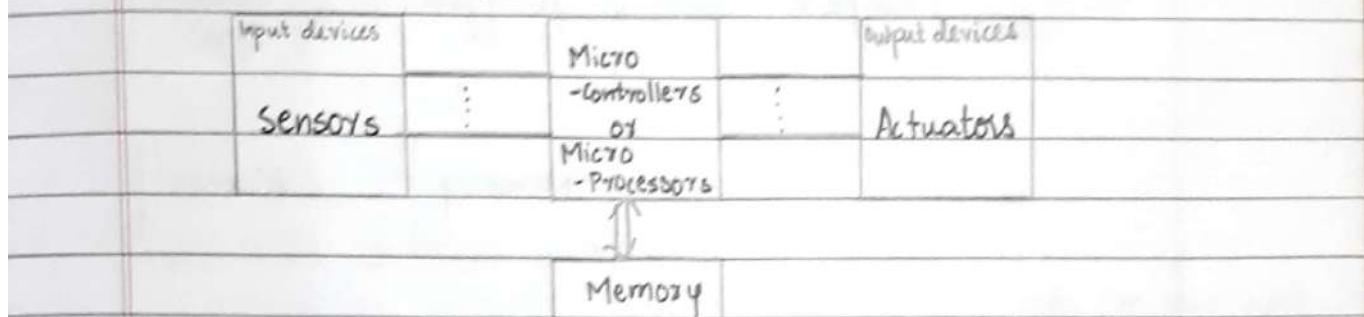


## Introduction

### \* EMBEDDED SYSTEM:

Embedded system is a programmable, digital electronic system designed to perform a particular function or sequence of functions using both hardware and software.



General block diagram of embedded systems.

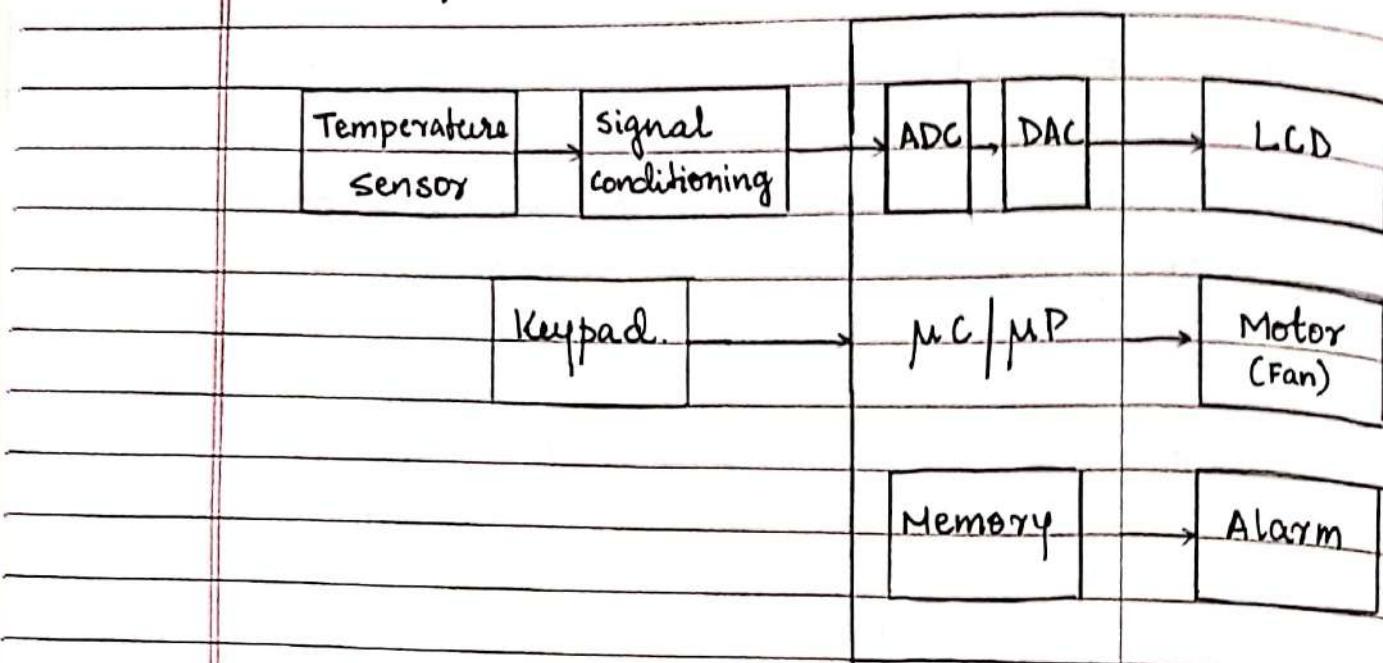
Embedded systems can be processor based or controller based. All the hardwares except the microcontroller or the microprocessor are called as peripherals which may be internal or external hardware.

Input and output devices are interfaced to microcontroller or microprocessor through input and output interfacing devices as the speed of input or output device and also the operation of input and output devices do not match with that of microcontroller/microprocessor. Only one device is attended at a time as they are interfaced by common system bus.

The two basic operations of an embedded system are

- monitoring : based on time
- controlling : based on sequence.

- Example for an embedded system: Temperature Controller



Signal conditioning circuits are used to amplify the sensed physical quantity.

DAC and ADC can be inbuilt (on chip) or may not be inbuilt, in such a case they are interfaced externally.

Motor is used to run the fan incase the temperature rises more than the preset value.

Alarm indicates the rise in temperature.

Keypad can be used to change or set the reference value of the temperature.

Memory holds the application program that is the software on which it operates.

NOTE:

Software is developed using software development tools. IDE (Integrated development environment) is a package of many software development tools.

Software is which is developed and tested / not tested whereas Firmware is developed, tested and available in internal memory.

Burning, dumping, flashing, downloading is to put the

program into the internal memory of the microcontroller or the microprocessor.

IDE is used to develop the code.

Simulator is used to test the developed code.

Downloader is used to dump the code in the memory.

Emulator is used to test the hardware functionality of the microcontroller / microprocessor.

### IDE:

- Code Editor: To edit the code

- Assembler / cross Assembler

Assembler: Assembly language to machine code for the host.

Cross assembler: Assembly language to machine code for the target system.

- Compiler / cross-compiler

Compiler: Machine code to higher level language for the host system.

Cross compiler: Machine code to higher level language for the target system.

- Linker: Source code is linked with library functions.

- Loader: It stores the data in user defined locations.

- Simulators: Debuggers.

- Graphical User Interface (GUI):

{

One IDE is specific to only one microcontroller, they might have same name but are different versions. But one microcontroller can have multiple IDE's.

Kiel: 8051 microcontrollers

Eclipse: ARM microcontrollers

MPLab: PIC microcontrollers

AVR studio: AVR microcontrollers.

classmate  
Date \_\_\_\_\_  
Page \_\_\_\_\_

Lower end microcontrollers need not require any operating system (OS) whereas higher end microcontrollers require operating system (OS).

### Important characteristics of embedded systems

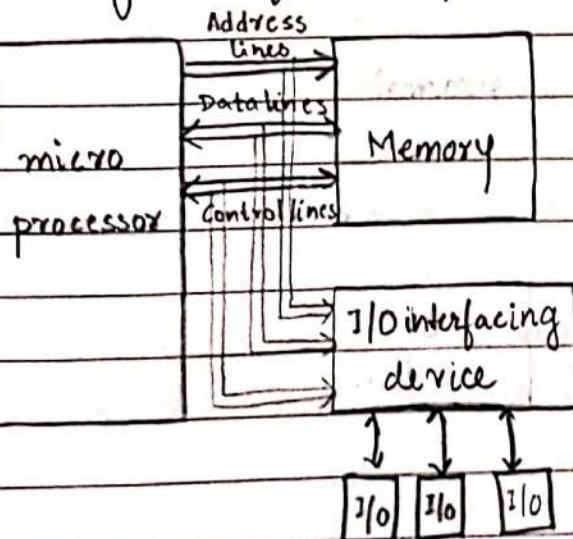
- Affordability
- Portability
- Battery driven
- Response time (high speed)

## Unit - 1

## MICROPROCESSORS AND MICROCONTROLLERS

*	Micropocessors	Microcontrollers
Purpose	General Purposes	Specific purposes
Hardware	Computer on chip	Programmable system on chip
Design	Flexible : It can be expanded by more and more external interfaces.	Primitive : very less external interfaces can be made.
Size	Bulky : due to more external interfaces.	Not bulky : due to very less external interfaces.
Functionality	Only few pins are multi-functional (dual operation but only one type of operation at a time)	Most of the pins in microcontrollers are multi-functional.
Cost	Expensive	Cheaper
Software	Byte Addressable	Bit / Byte addressable.
Addressing Modes	More addressing modes (memory related)	Less addressing modes.

## Block diagram of microprocessor



Data lines determines the processing capability of a processor.

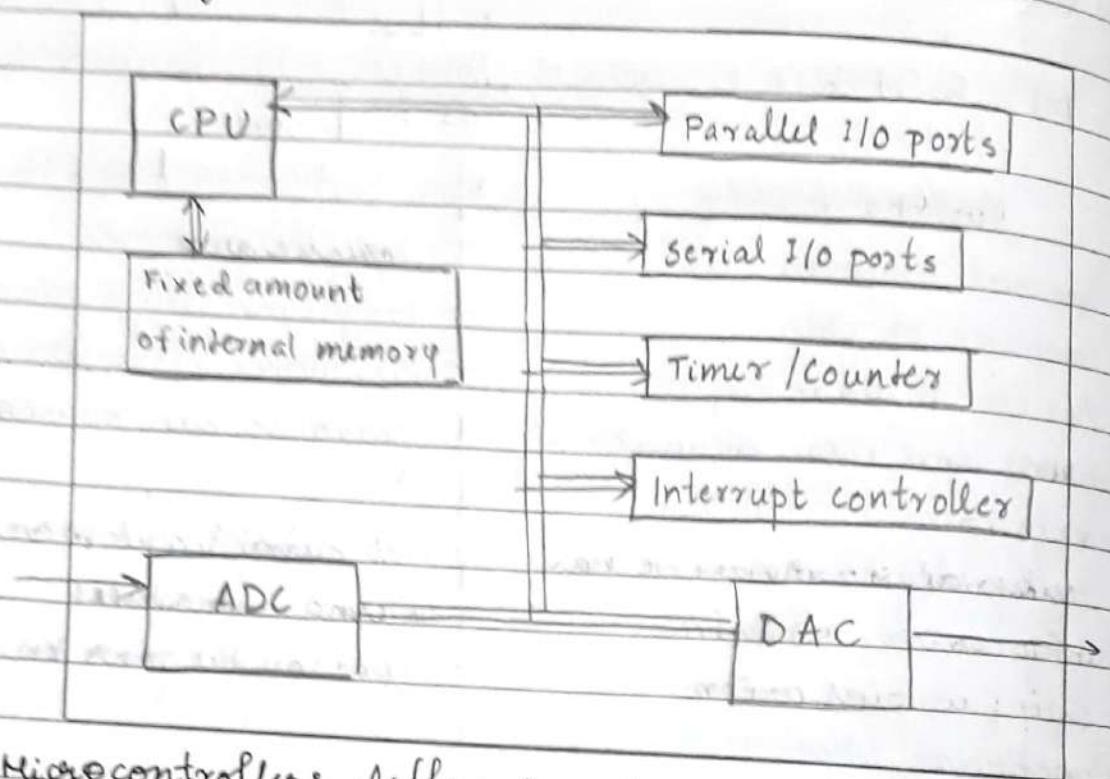
Address lines determines the memory locations.

$$\text{Memory} = 2^{\text{number of address lines}}$$

Data lines also determines the number of internal registers width.

control lines determine the activity of data lines.

## Block Diagram of Microcontrollers.



Microcontrollers differ based on, on chip resources and internal memory.

Based on the organisation of the memory.

Van Neuman

Common set of address bus and data bus is used as data and code are not physically separated.

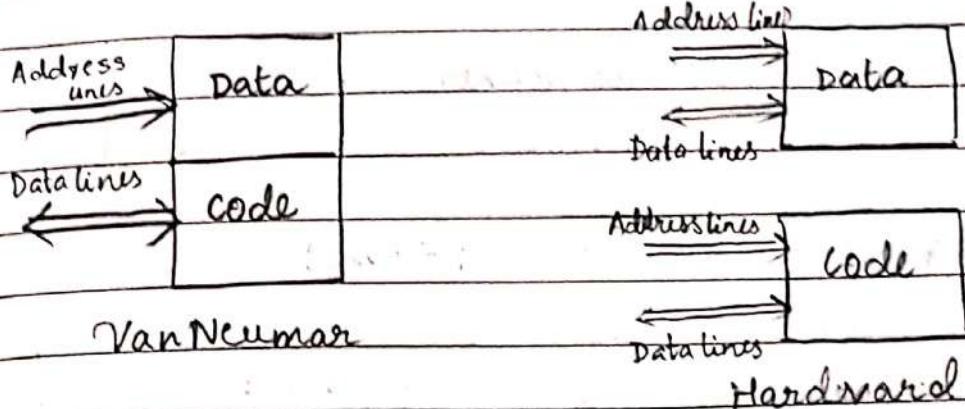
Hence there is a chance of overlapping of data with code. We should make sure we don't let them overlap.

It is not possible to access data and code simultaneously.

Harvard.

Data and code have their own respective set of address bus and data bus.

Hence there is no chance of overlapping of data with code. It is possible to access data and code simultaneously.



CISC	RISC
- More number of instructions	- less number of instructions
- Hardware for software	- software for hardware
Dedicated hardware is required for a complex operation. <u>Ex: multiplication</u>	software is developed for the basic operation.
- Expensive	<u>Ex: multiplication by repetitive addition</u>
- Complex circuitry	- Cheaper
- Hardware pipelining	- Simple circuitry
- less internal registers.	- software pipelining
- Instruction format is variable. and each instruction is executed in different cycles.	- More internal registers
	- Instruction format is usually fixed and instructions are executed in single clock cycle.

Microcontrollers belong to certain families. They are not compatible with each other as they have different set of instructions of different languages. There are many members within the family based on:

- fixed number inbuilt memory
- on chip resources

There are many vendors for these microcontrollers.  
Ex: Intel, Siemens, Atmel, etc all manufacture 8051 uc.  
Within the family all members are code compatible.  
Families: 8051, Motorola, PIC, AVR, ARM

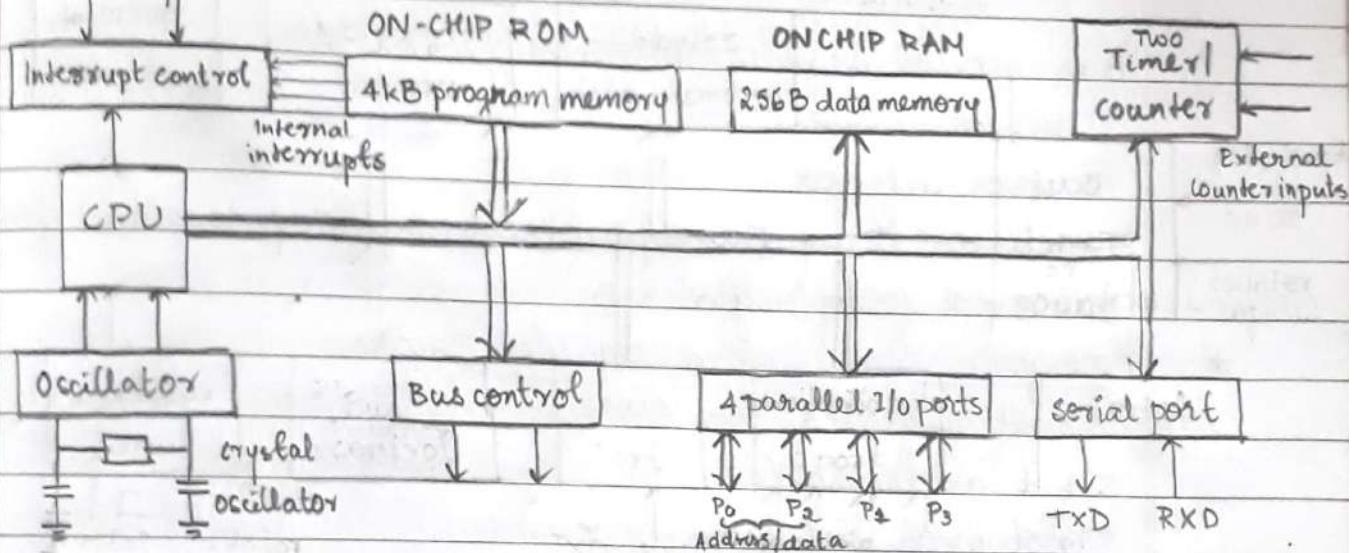
## 8051 Microcontroller

### \* Features of 8051 Microcontroller:

1. It is an 8 bit controller  
(8 bit operation can take place at a time)
2. It has 8 bit data lines.
3. It has 16 bit address lines
4. Along with internal memory, it also supports external interfacing of external memory if required.  
It supports  $2^{16}$  ~~data~~ = 64 k bytes of data memory  
= 64 k bytes of programmable memory
5. 4 kB of internal program memory. It ~~also~~ has 256 bytes of data memory internally.
6. It has 5 interrupt sources internally.  
Among 5, 2 are external interrupts and 3 are internal interrupts. Along with this there is another interrupt: RESET
7. It has 4 parallel I/O ports where all the ports are bidirectional. Each port is 8 bit and programmable.  
Number of I/O ports determines the number of I/O devices can be interfaced or number of memory that can be interfaced.  
2. I/O ports can be used as address lines as well.
8. Hardware Architecture.
9. It has a dedicated serial I/O port. It is used to transfer data through single line (full duplex-serial port).
10. It has 2 inbuilt 16bit timer/counter. Counter has an input which is given as external pulse
11. It has an inbuilt oscillator circuit for synchronisation.  
It is activated by externally interfacing crystal oscillator
12. It supports idle mode - no activity takes place and power down mode/sleep mode - low power.
13. It operates at a frequency of 11.0592 MHz.
14. It has a bus control unit internally.

15. It is developed by HCMOS (Hybrid VLSI) technique.

16. It has 82 special function registers.



The 8051 performs operation on 8 bit of data. It receives data from data memory or registers. After the operation it gives the status of the result in the register Program Status Word (PSW).

~~Registers~~

\* For operation one of the data will be present in A (Accumulator).

Even after operation the result is usually stored in accumulator.

\* It is a 8-bit register. Register can be general purpose or special purpose but accumulator can be both general purpose or special purpose register.

\* Accumulator is both bit/byte addressable register.

i.e.  $\text{NOT A}, \text{B}$  here it is byte addressable

syntax acc.7, acc.6, acc.5 - acc.1, acc.0 bit addressable.

\* Accumulator is a user defined register.

\* It is addressed either by name or address. Memory location of address EOF is reserved for the accumulator.

\* Register B

- \* can be used as general purpose or special purpose register.
- \* mul AB } special purpose
- \* div AB } A: quotient B: remainder

B is special register only along with A.

\* bit addressable

\* user defined.

\* addressed by name / address. Memory location for is reserved for register B.

\* Program Status Word Register (PSW)

\* 8 bit register

\* bit addressable

\* not user defined

\* Memory location D0H is reserved for PSW.

PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
C4	AC	X	RS1	RS0	OV	X	P

based on the number of 1's in the result

→ PSW.0 represents parity

0 - even parity

1 - odd parity

→ PSW.2 represents overflow flag

It indicates if the result is out of range

→ PSW.3: Register bank selector bit 0.

PSW.4: Register bank selector bit 1.

All register contents is zero by default on reset.

MOV A, R2 here by default R2 belongs to bank 0.

RS1      RS0

0      0 : Bank 0

0      1 : Bank 1

1      0 : Bank 2

1      1 : Bank 3.

For overflow:

8 bit range  $\Rightarrow$  00h to ffh

It is divided into half

00h to 7Fh and 80h to FFh

in binary MSB = 0

$\therefore$  OV is 0

in binary MSB = 1

$\therefore$  OV is 1

if in this range.

if in this range.

$\rightarrow$  PSW.6 : Auxiliary carry flag

Carry from lower nibble to higher nibble: auxiliary carry

This flag is used by instructions that perform BCD arithmetic.

$\rightarrow$  PSW.7: Carry flag : indicates out of width of the register.

i.e., carry out from the 8th bit (of 8bit register).

$\rightarrow$  PSW.5 and PSW.1 : user defined.

#### \* Program counter

\* 16 bit register : since address lines is 16 lines the address width is 16 bit hence it is a 16 bit register.

\* It holds the address of the program memory in which the next instruction to be fetched is present.

\* It is not user defined.

\* It is incremented based on the size of the instruction being executed.

#### \* Data pointer: dptr (external data memory pointer)

\* 16 bit register : Since external memory of 64kB can be interfaced.

\* It can be used as both General Purpose Register and special purpose register.

If used as general purpose register, its content is 16 bit data.

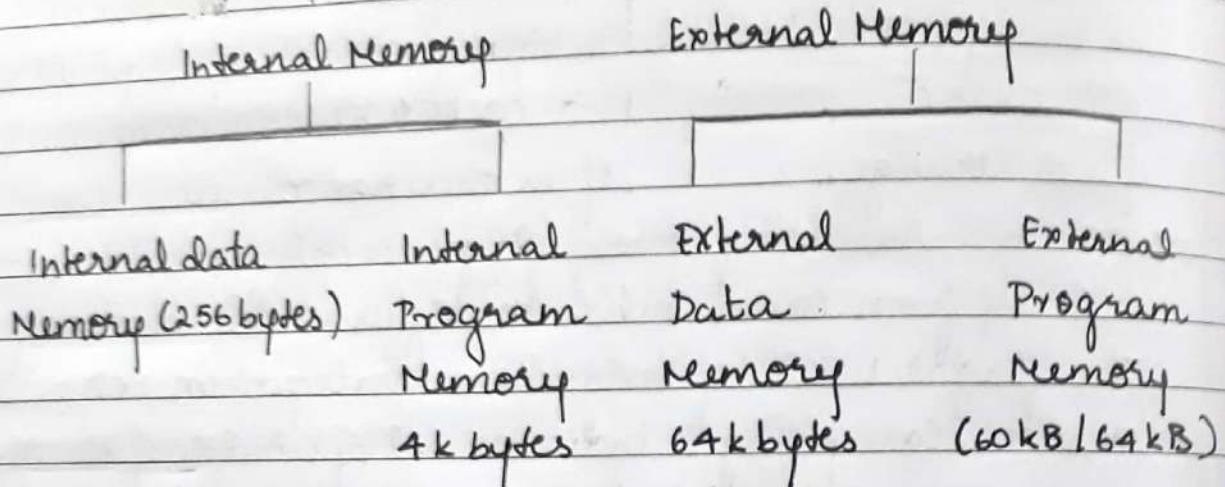
If used as special purpose register, its content is the address.

\* It can be treated as two independent 8 bit registers that is dpl and dph but they cannot be used as data pointers.

\* dptr alone does not have any address but its internal registers have address: dph (83h) and dpl (2ah).

## \* MEMORY ORGANISATION OF 8051 WITH MEMORY MAPPING

### Memory Organisation



PA : Program access (active low)

If 0 it allows to interface a program memory of 60 kB.  
 If 1 it allows to interface a program memory of 64 kB.

### \* Internal Data Memory

Internal Data Memory			
Special Function Registers (SFRs)	FFh	:	255
		:	
	80h		128
Internal data RAM	7Fh	:	127
		:	
	00h		0

← byte →

The internal data memory is segmented into two equal parts.

80h - FFh : Special function register, they cannot be used for general purposes by the user.

00h - 7Fh : Internal Data RAM.

## Internal Data RAM

-1Fh	scratch pad memory area	32
-30h		48
-2Fh		44
-20h		32
-1Fh	Register banks	31
00h		0

The internal data RAM is further segmented into three parts:

Scratch pad memory area is where we can erase and write data.

It is also called general purpose memory area. (80 bytes)

Bit addressable memory area.

Total of 16 bytes are bit addressable read / write memory.

Register Banks : A total of 32 bytes are set aside for register banks and the stack.

Register Bank 3	-1Fh	R <sub>t</sub>	31
	-18h	R <sub>o</sub>	24
	-17h	R <sub>t</sub>	23
	-10h	R <sub>o</sub>	16
Register Banks (stack)	-0Fh	R <sub>t</sub>	15
	-08h	R <sub>o</sub>	8
	-07h	R <sub>t</sub>	7
	00h	R <sub>o</sub>	0

The register banks are further divided into 4 banks of registers with 8 registers each (R<sub>0</sub> - R<sub>7</sub>).

00h - 07h - Register Bank 0 is the default register bank.

08h - DFh - Register Bank 1 is used as stack. Hence we must either

not use register bank 1 or we must allocate another area of RAM for stack.

10h - 14h - Register Bank 2

18h - 1Fh - Register Bank 3.

Each bank is 8 bit. All the bank registers are identified by same name but have their own set of address.

Register banks

## \* Structure of Assembly language:

An assembly level language instruction consists of four fields  
[label:] mnemonics [operands]; [comments]  
bracket indicates that those fields are optional.

label: allows the program to refer to a line of code by name.  
Its name cannot be a keyword.

mnemonics: every opcode performs a particular operation.  
Each microcontroller have their own set of mnemonics.

operands: data on which the operation takes place. They  
can be one, two...etc or implicit (no operands: predefined)

comments: It is recommended as they describe the program  
and make it easy for someone else to read and  
understand.

## UNIT 2

### \* Instruction sets:

It is classified based on the type of operation they perform

#### 1. Data Transfer Instructions: (copied or exchanged)

Data transfer may be between:

- register to register : can happen only between same size registers if not it causes data mismatch.
- memory to memory: direct transfer - can happen only within the internal memory directly. Not possible in the case of external memory.
- register to memory:

MOV instruction: Any data can be transferred. It requires two operands. The direction of transfer varies from microcontroller to micro controller. For 8051 microcontroller

MOV op<sub>1</sub>, op<sub>2</sub>

where op<sub>1</sub>: destination

and op<sub>2</sub>: source.

Syntax: MOV destination, source

destination can never be data. It can be address or register whereas source may be data, address or register.

Program consists instruction, assembler directives and operating system commands.

Assembler directives: gives instruction to the assembler which can be data, memory or control related. The following are some widely used directives of 8051.

ORG (origin): memory related. It is used to indicate the beginning of the address. The address can be in hex or in decimal if decimal then the assembler will convert it into hex.

EQU (equate): used to define a constant without occupying a memory location. It associates a constant value with a data label so that when the label appears in the program, its constant value will be substituted for the label.

COUNT EQU 25

MOV R3, #COUNT : R3 will be loaded with value 25

END directive: It indicates to the assembler the end of the source file. Anything after the END directive in the source code is ignored by the assembler.

A program can also be ended by

repeat: sjmp repeat  $\Rightarrow$  here the controller is alive and keeps on performing this function. It is used to display any data for a long time.

address width

$\rightarrow$  Org 0000h

$\rightarrow$  MOV a, #12h

appropriate data to appropriate register (i.e., same size) : load operation

$\rightarrow$  MOV b, #12

12 = Ch (stores as c as mentioned in decimal)

$\rightarrow$  MOV r2, #0001 0010b

→  $\text{MOV } Y_3, \# '3'$       ASCII value      hexadecimal value of ASCII value is stored. Here 3 in ASCII is 33 in hexa  
(because for 0 to 9 : 30h is added and for A to F : 3fh is added)  
Here 33h is stored in  $Y_3$ .

→  $\text{MOV } a, \# f dh$  : Syntax error

Any MSB greater than 9 should be preceded with zero.  
∴  $\text{MOV } a, \# 0 f dh$ ,

→  $\text{MOV } dptr, \# 1234 h$   
16bit                  16bit  
                        no data mismatch

\* Stack: It is a part of internal data RAM. It is used by both controller (to perform internal operation) and user.

To transfer control from main function to subroutines we use program control transfer group of instructions. The instruction that has to be performed after executing the subprograms is saved in stack.

It saves the content of the program counter when CALL and RETURN instructions are executed.

~~Consider~~ Default value of stack pointer  $SP = 04h$  hence the stack starts from 08h. (default stack)

This has a conflict between default stack and bank 1. When there is a CALL or RETURN functions is used in a program Bank 1 should not be stored with any data. If not there will be overlapping of data.

User stack: scratch pad memory area can be used as user stack (30h to 7Fh). User stack can be defined only by three instructions:

- initializing stack pointer :  $\text{MOV } SP, \# 2FH$  (stack from 30h)
- PUSH : write data from memory
- POP : read data from memory

Top of the stack: current position of the stack pointer.

Ex: store contents of a and b

mov a, #12h

mov b, #15h

in stack to address 42h and 43h

sp is initialised to 41h. (top of the stack).

sp = 41h is incremented

sp = 42h : write 12h (top of stack)

sp = 42h is incremented

Sp → 41h	15h
42h	
43h	12h
44h	

sp = 43h : write 15h

now sp = 43h is the top of the stack.

To read the data

Syntax

push oeah } write data

push address

push ofoh }

pop address

pop ofoh } retrieve the data

pop oeah

Data written last will be read first.

If push and pop is in same order it leads to exchange of data. Hence always works in opposite direction.

push: increment and write

pop: read and decrement.

works on last in first out principle.

push oeah

push ofoh → top of stack before pop

pop ofoh

pop oeah → top of stack after pop

If no registers represent to store data then stack can be used to store data.

Notes:

load: moving data in register

store: storing data into data memory

Program: putting data into program memory.

- \* Develop an 8051 ALP to load / store any 8 bit data in  
a  $r_2$  and  $r_5$  of bank0, bank1, bank2 and bank3 registers  
by i. ~~names~~ names or  
ii. address

by names = org 0000h (on reset content of PC = 0000h)

MOV  $r_2, \#12h$  }  
MOV  $r_5, \#15h$  }

Bank 0

in any embedded system  
without any OS only one program  
is written and run in the  
microcontroller

setb PSW.3 }  
MOV  $r_2, \#12h$  }

Bank 1

MOV  $r_5, \#15h$  }

clr PSW.3 }

setb PSW.4 }

Bank 2

MOV  $r_2, \#12h$  }

MOV  $r_5, \#15h$  }

setb PSW.3 }

MOV  $r_2, \#12h$  }

Bank 3

MOV  $r_5, \#15h$  }

end

If the bank is not mentioned it is  
Bank 0 by default  
The bank is selected by modifying  
the PSW register

PSW.4      PSW.3

RSI      PSO

0      0 - bank 0

0      1 - bank 1

1      0 - bank 2

1      1 - bank 3

by address

= org 0000h

MOV 02h, #12h }

Bank 0

MOV 05h, #15h }

MOV 0ah, #12h }

Bank 1

MOV 0dh, #15h }

MOV 12h, #12h }

Bank 2

MOV 15h, #15h }

MOV 1ah, #12h }

Bank 3

MOV 1dh, #15h }

end

Any bit of any bit addressable  
register can be modified by  
the instructions

- clr bit (to reset)

- setb bit (to set)

Read data from register is faster  
than read data from memory

\* Store Operation:

It can be done by different addressing modes which depends on the type of data. A data may be

- a single constant number (8 bit / 16 bit)

- an array of numbers (8 bit / 16 bit)

- array of numbers may be a sequence of numbers or random numbers.

- They may be stored in a sequence or stored in random memory locations.

Based on these there are four combinations

- a. sequence of data in sequence of memory location

- b. sequence of data in random memory location

- c. Random data in sequence of memory location

- d. Random data in random memory location.

In order to realise this we have the following addressing modes: (also called as data addressing modes)

1. Immediate Addressing Mode:

In this mode data is a single constant number and data is a part of the instruction.

MOV a, #12h

Data can never be the destination

MOV dptr, #1234h

Invalid syntax: MOV #12h, a.

Data is always the source operand.

2. Register Addressing Mode:

In this mode data is the content of the register. Here both the operands are registers of same size.

MOV b, a

Due to data mismatch

MOV dpl, a

Invalid syntax: MOV dptr, a.

Both the registers have to be of same size. 16 bit 8 bit

3. Data Memory Addressing Mode:

In this mode data is in the memory. It may be in internal memory or in external memory. It can be classified as:

a. Internal data memory addressing mode.

- direct data memory addressing mode.

- indirect data memory addressing mode.

b. External data memory addressing mode.

- indirect data memory addressing mode.

Internal Direct data memory addressing mode

read MOV a, 30h

operation  $(a) \leftarrow (30h)$

30h is address where the required data is present is part of the instruction.

MOV b, 40h

write MOV 40h, b

operation

MOV 30h, a ;  $(30h) \leftarrow (a)$

Address may be source or destination operand.

MOV 30h, 50h

$(30h) \leftarrow (50h)$

Address may be both source and destination operand.

Internal Indirect data memory addressing mode

In this mode we use pointers.  $r_0$  and  $r_1$  of any bank registers as internal data memory pointers.

read MOV a, @ $r_0$  ;  $(a) \leftarrow ((r_0))$  Here content of  $r_0$  and

MOV b, @ $r_1$  ;  $(b) \leftarrow ((r_1))$   $r_1$  is address where

write MOV @ $r_0$ , b ;  $((r_0)) \leftarrow (b)$  content is moved to a.

MOV @ $r_1$ , a ;  $((r_1)) \leftarrow (a)$  and b respectively (read)

Invalid Syntax : MOV @ $r_1$ , @ $r_0$

External Indirect data memory addressing mode

One of the register is used as a pointer, that is dptr is used as external data memory pointer.

read MOVX a, @dptr ;  $(a) \leftarrow ((dptr))$

write MOVX @dptr, a ;  $((dptr)) \leftarrow (a)$

A data can be read or written only through a register.

Invalid Syntax : MOVX b, @dptr

MOVX  $r_2$ , @dptr

#### 1. Program Memory Addressing Mode:

Indirect internal program memory addressing mode

read:  $\text{MOV} a, @a + \text{dptr}; (a) \leftarrow ((a + \text{dptr}))$

since program memory is ROM hence only read operation is possible. There is no write instruction.

a - base dptr-index

: Base + Index Indirect program memory addressing mode

- \* Demonstrate different addressing mode to store any 8 bit constant number into a 5 continuous memory location starting from address 30h.

```
= Org 000Ah
    MOV 30h, #12h
    MOV 31h, #12h
    MOV 32h, #12h
    MOV 33h, #12h
    MOV 34h, #12h
    end
```

(Immediate Addressing mode)

```
= Org 0000h
    MOV a, #12h
    MOV 30h, a
    MOV 31h, a
    MOV 32h, a
    MOV 33h, a
    MOV 34h, a
    end
```

(Register Addressing Mode)

```
internal
memory =
Org 0000h
MOV 30h, #12h
MOV 31h, 30h
MOV 32h, 30h
MOV 33h, 30h
MOV 34h, 30h
end
```

(Direct Addressing Mode)

```
= Org 0000h
    MOV r0, #30h // r0 int. dataptr.
    MOV r2, #5 // r2 used as counter
    MOV a, #12h // data to be stored
    loop { repeat: MOV @r0, a ; ((r0)) <- (a)
            inc r0
            djnz r2, repeat
            end. }
```

(Indirect Addressing Mode)

NOTE: Arithmetic group of instructions:  
inc, dec : single operand instruction  
operand may be content of register  
or content of memory location.

NOTE: To decrement a counter:

Program control Transfer group of instructions

djnz operand, label : conditional transfer group of instructions

First it decrements the content of the register and then checks the content of operand if it is zero or not.

The operand has to be a register.

If the content of register is

- zero: the control is transferred to next line
- not zero: the control is transferred to label.

NOTE:

The direct addressing mode is preferred for random address and random data.

The indirect addressing mode is preferred for sequence of data in sequence of memory location.

External memory

=

```
org 0000h
MOV dptr, #8000h
MOV r2, #5
MOV a, #12h
```

Repeat: MOVX dptr@a

inc dptr

djnz r2, repeat

end

(Indirect data addressing mode)

\* clear the data:

Clear operation: It is used to clear the data stored in some memory locations.



\* To clear data from continuous 5 memory locations.

= org 0000h

MOV r<sub>0</sub>, #30h

MOV r<sub>2</sub>, #5

MOV a, #0

Repeat: MOV @r<sub>0</sub>, a

inc r<sub>0</sub>

djnz r<sub>2</sub>, repeat  
end.

As it is continuous memory location indirect data addressing mode is used

- \* To increment the content present in the memory location.

= org 0000h

MOV r<sub>0</sub>, #30h

MOV r<sub>2</sub>, #5

repeat: inc @r<sub>0</sub>; (r<sub>0</sub>)  $\leftarrow$  (r<sub>0</sub>) + 1

inc r<sub>0</sub>

djnz r<sub>2</sub>, repeat  
end

- \* To perform addition

Syntax: add a, source; (a)  $\leftarrow$  (a) + (source)

Source may be a register, immediate data, address of the memo (other than a)

Add with carry

Syntax: adc a, source; (a)  $\leftarrow$  (a) + (source) + (cy)

(used in binary adder)

NOTE: A register as a counter can count a maximum of 255 as it is a 8 bit register. For counting more than that we can use more than one counter.

\* To store data in 500 memory location (using 2 counters)

= Org 0000h

MOV a, #12h

MOV r<sub>2</sub>, #30h

MOV r<sub>2</sub>, #100

r<sub>2</sub> - outer counter

Up: MOV r<sub>3</sub>, #5

r<sub>3</sub> - inner counter

Repeat: MOV @r<sub>2</sub>, a

inc r<sub>2</sub>

djnz r<sub>3</sub>, repeat

djnz r<sub>2</sub>, up

Here it counts for  $5 \times 100 = 500$  times

Number of counter = Number of loops

### \* Transfer Operation

- Transfer of data between registers.

\* Transfer of data from r<sub>2</sub> and r<sub>5</sub> of bank 1 to r<sub>2</sub> and r<sub>5</sub> of bank 2 by:

i. name      ii. address      iii. stack

= By name

Org 0000h

setb PSW.3

MOV r<sub>2</sub>, #12h

MOV r<sub>5</sub>, #15h

MOV a, r<sub>2</sub>

MOV b, r<sub>5</sub>

setb PSW.4

clr PSW.3

MOV r<sub>2</sub>, a

MOV r<sub>5</sub>, b

end

= By address

Org 0000h

MOV 0ah, #12h

MOV 0dh, #15h

MOV a, 0ah

MOV b, 0dh

MOV 12h, a~~0ah~~

MOV 15h, b~~0dh~~

end.

= By stack

org 0000h

MOV 0ah, #12h

MOV 0dh, #15h

MOV sp, #3fh

sp is initialised one address less  
push 0ah ; 10h ← (0ah) that the address we have to use  
push 0dh ; 11h ← (0dh) (top of the stack)

pop 15h

pop 12h

now top of stack is 3fh

end.

- Transfer of data between memory: block move.

- i. data transfer between internal memory to internal memory
- ii. data transfer between internal memory to external memory
- iii. data transfer between external memory to internal memory
- iv. data transfer between external memory to external memory

\* data transfer : internal to internal memory.

= org 0000h

MOV r0, #30h // source pointer

6CYC

Dest

34h

54h

MOV r1, #50h // destination pointer

33h

53h

MOV r2, #5

32h

⇒ 52h

up: MOV a, @r0

31h

51h

MOV @r1, a

30h

50h

inc @r0

inc r1

djnz r2, up

end.

\* data transfer: internal memory to external memory

= Org 0000h

MOV Y<sub>0</sub>, #30h // source pointer

MOV dptr, #8000h // dest pointer

MOV Y<sub>2</sub>, #5

up: MOV a, @Y<sub>0</sub>

MOVX @dptr, a

inc Y<sub>0</sub>

inc dptr

djnz Y<sub>2</sub>, up

end.

Src(int)	Dst(ext)
34h	8004h
	⇒
30h	8000h

\* data transfer: external memory to internal memory

= Org 0000h

MOV Y<sub>0</sub>, #30h // Dest pointer

MOV dptr, #8000h // source pointer

MOV Y<sub>2</sub>, #5

up: MOVX a, @dptr

MOV @Y<sub>0</sub>, a

inc dptr

inc Y<sub>0</sub>

djnz Y<sub>2</sub>, up

end

Src(ext)	Dst(int)
8004h	34h
	⇒
8000h	30h

\* data transfer: external memory to external memory.

= org 0000h

(Ext)src

(Ext)dst

- MOV dptr, # 8000h

8000h

9000h

src - MOV r<sub>2</sub>, # 5 // counter

- MOV r<sub>3</sub>, dph (dph only because only

⇒

dst - MOV dptr, # 9000h (the higher byte)

- MOV r<sub>4</sub>, dph (address varies) 8000h

9000h

up: MOV dph, r<sub>3</sub>

we move the address (only the

MOVX a, @dptr

higher bit) into dph as only one

MOV dph, r<sub>4</sub>

pointer dptr is present for both

MOVX @dptr, a

source and destination.

inc dpl increments 8000h. case 1: Here higher byte address is

~~increments 9000h~~ increments 9000h different whereas lower byte

dinx r<sub>2</sub>, up

address is same.

end.

= Case 2: Both higher byte and lower byte address are different.

org 0000h

MOV r<sub>2</sub>, # 10

MOV dptr, # 8000h // source pointer

MOV r<sub>3</sub>, dph

MOV r<sub>5</sub>, dpl

MOV dptr, # 9006 // destination pointer

MOV r<sub>4</sub>, dph

(Ext)src

(Ext)dst

MOV r<sub>6</sub>, dpl

9015h

repeat: MOV dph, r<sub>3</sub>

MOV dpl, r<sub>5</sub>

⇒

MOVX a, @dptr

MOV dph, r<sub>4</sub>

8000h

9006h

MOV dpl, r<sub>6</sub>

MOVX @dptr, a

inc r<sub>3</sub>

inc r0

djnx r2, repeat  
end.

NOTE: There is no instruction to write into code memory but there are instructions only to read from code memory. You can write into code memory only by assembler directives.

\* To store data into code memory:

= Org 0300h

my-data db 09h, 0fh, 12h . . . .

my-data1: db '2', '5', 'a', 'f' . . . .

my-data2: db "1234567890"

my-data3: db "ABCDEFGHIJ"

my-data4: db "GOD IS GREAT"

my-data5: db "INDIAN TEAM \0"

Ways to store  
data into program  
memory.

If there is main program after this do not end this with an end. If there is nothing after this end with an end

\*

= CASE I: string of data without null character

In this case we will have to be using a counter.

Org 0300h

my-data: db "GOD IS GREAT"

org 0000h

Mov r2, #12 // counter

Mov dptr, #0300h or Mov dptr, # my-data // source pointer  
Mov r0, #30h // destination pointer.

repeat: clr a

Movc a, @ r0+dptr

```

MOV @R0, A
INC DPTR
INC R0
DJNZ R2, repeat
END.

```

clear

syntax:

- i. CLR A
- ii. CLR bit
- iii. CLR C

In this case for transferring large strings, initialisation of counter is difficult.

= CASE 2: string of data with null character.

In this case no counter is required.

ORG 0300H

my-data: DB "GOD IS GREAT" //

ORG 0000H

MOV DPTR, #0300H

(0) MOV DPTR, # my-data

MOV R0, # 30H // destination pointer

repeat : CLR A

MOV A, @ R0 + DPTR

JZ Stop

MOV @R0, A

INC DPTR

nop : no operation

INC R0

JNP repeat

Stop: nop

\* Conditional Program control transfer group of instructions.

- DJNZ operand, label

- JZ label (jump zero)

- JNZ label (jump non zero)

- JC label (jump carry)

- JNC label (jump no carry)

- JB bit, label (if bit 1 jump to label)

- JNB bit, label (if bit 0 jump to label)

- CJNE op1, op2, label (compare jump if not equal)

} always works with register A.

- \* Unconditional program control Transfer group of instructions
  - sjmp label (short jump) } transfer within the main program
  - ajmp label (absolute jump)
  - ljmp label (long jump)
  - a call label } transfer between main program
  - l call label } subprograms
  - return (ret)

1 byte inst sjmp : up to down 128 bytes.

2 1/2 byte inst ajmp : within one page. (2k bytes)

3 byte inst ljmp : anywhere within the available memory. (64kbytes)

a call : main memory to subprogram memory  
(within one page memory)

l call : subprogram : anywhere in the memory.

- \* To transfer data from internal data memory to external data memory through stack.

=	int data	stack	Ext data
	39h	69h	9009h
	,	,	,
	,	⇒	,
	,	,	,
	30h	60h	9000h

Org 0000h

MOV R2, #10

MOV R0, #30h

MOV sp, #5fh

up: MOV a, @R0

push 0e0h

sp need not be incremented  
as it increments on its own after  
executing the line

writing data to a

inc r<sub>0</sub>      ] Internal data memory to stack  
 djnz r<sub>2</sub>, up.      ]  
 MOV r<sub>2</sub>, #10      initializing the counter ( $\because$  it would be zero after first loop)  
 MOV dptr, #9000h  
 up: pop oeah      ] reading data from a  
 MOVX @dptr, a      ]  
 inc dptr      ] stack to external data memory  
 djnz r<sub>2</sub>, up  
 end

### \* Exchange Operation

Syntax : XCH a, ~~source~~

source may be - a 8 bit register

- memory

- pointer but not data.

Destination has to be accumulator.

Ex: XCH a, b ; (a)  $\leftrightarrow$  (b)

XCH a, @r<sub>0</sub> ; (a)  $\leftrightarrow$  ((r<sub>0</sub>))

XCH a, 30h ; (a)  $\leftrightarrow$  (30h)

a = 25 b = 34

XCHD a, b

Nibble wise exchange takes place. for XCHD

After execution : Step 1: a = 24 b = 35

Step 2: a = 34 b = 25

Ex:

Exchange between ~~Registers~~ can be by

- Register addressing mode.
- Direct addressing mode
- using concept of stack
- Using exchange instruction

\* To exchange data between  $r_2$  and  $r_5$  register.

=

```

    Org 0000h
    MOV r2, #12h
    MOV r5, #15h
    MOV a, r2
    MOV b, r5
    MOV r2, b
    MOV r5, a
    end
  
```

(Register Addressing mode)

=

```

    Org 0000h
    MOV 02h, #12h
  
```

MOV 05h, #15h

MOV 03h, 02h

MOV 02h, 05h

~~MOV 02h, 02h~~

MOV 05h, 03h

end

(Direct Addressing mode)

=

```

    Org 0000h
  
```

MOV r<sub>2</sub>, #12h

MOV r<sub>5</sub>, #15h

XCH a, r<sub>2</sub>; a=12h r<sub>2</sub>:

XCH a, r<sub>5</sub>; a=15h r<sub>5</sub>:

XCH a, r<sub>2</sub>; a=00h r<sub>2</sub>:

end

(using exchange operation)

same order (push 02h  
no exchange) push 05h  
takes place | pop 02h  
| pop 05h  
| end

(using stack)

### - Memory block exchange

\* To exchange a block of data in internal memory.

without  
exchange =

Org 0000h

MOV r<sub>2</sub>, #10h

MOV r<sub>0</sub>, #30h

MOV r<sub>1</sub>, #50h

MOV a, @r<sub>0</sub>

MOV b, @r<sub>1</sub>

MOV @r<sub>0</sub>, b

MOV @r<sub>1</sub>, a

30h		50h
	↔	
30h		50h

inc  $r_0$   
 inc  $r_1$   
 djnx  $r_2$ , repeat  
 end

= String exchange operation

Org 0000h

MOV  $r_2, \#10h$

MOV  $r_0, \#30h$

MOV  $r_1, \#50h$

repeat: MOV a,  $\@r_0$ ;  $\@r_0 = 30h$  a =  $30h$

XCH a,  $\@r_1$ ;  $\@r_1 = 30h$  a =  $50h$

XCH  $\@r_0, a$ ;  $\@r_0 = 50h$  a =  $30h$

inc  $r_0$

inc  $r_1$

djnx  $r_2$ , repeat

end.

#### \* Search Operation:

Data to be searched can be a number or character.

- Its occurrence can be found (present or not)
- its position can be found (address)
- number of occurrence.

#### \* searching a data in an array of size 10.

= Org 0000h

MOV  $r_2, \#10$  // size of array

MOV  $r_0, \#30h$  // starting address of array

repeat: MOV a,  $\@r_0$

CJNE a,  $\#50h$ , nteq

MOV 40h,  $\#0ffh$  // if 50h is present: true

nteq: inc  $r_0$

djnx  $r_2$ , repeat.

depicting

by saving

offh in 40h

30h

Page

depicting

MOV 40h, #00h // 50h is not present: false  
by saving  
00h in 40h

end.

\* To track position

= Org 0000h

MOV r<sub>2</sub>, #10 // size of array

MOV r<sub>0</sub>, #30h // starting address

repeat: MOV a, r<sub>0</sub>

cjne a, #50h, nteq

MOV 40h, #0ffh // data is present

MOV b, r<sub>0</sub> // address where data is present.

nteq : inc r<sub>0</sub>

djnz r<sub>2</sub>, repeat

MOV 40h, #00h // data is not present.

end.

\* To find number of occurrence.

= Org 0000h

MOV r<sub>2</sub>, #10 // size of array

MOV r<sub>0</sub>, #30h // starting address

MOV r<sub>3</sub>, #00h // number of occurrence.

repeat: mov a, @r<sub>0</sub>

cjne a, #50h, nteq

inc r<sub>3</sub>

nteq : inc r<sub>0</sub>

djnz r<sub>2</sub>, repeat

end.

\* To find the smallest and largest number in an array of data.

MOV R0, #0000h

MOV R2, #10h // array consists 10 elements each of 8 bit.

MOV R0, #50h // starting address

MOV B, #00h // b to hold 00h initially assuming 00h to be highest.

MOV A, @R0

repeat: CJNE A, B, NTEQ

SJMP R2, repeat // if a=b, to go to next comparison

NTEQ: JC NEXT // largest number

MOV B, A

NEXT: INC R0

DJNZ R2, repeat

NTEQ: JNC R0, NEXT // smallest number

MOV B, A

NEXT: INC R0

DJNZ R2, repeat

end

cases:  
 1. a = b  
 2. a ≠ b  
 i. a > b : CY = 0  
 ii. a < b : CY = 1

Checking the status of carry flag we can find whether the number is greater or not.

\* To arrange in ascending / descending order.  
 (using Bubble sort)

Initially

There are two counter for bubble sort

8004h

5

- Pass counter (outer counter)

8003h

4

- Comparison counter (inner counter)

8002h

3

n - number of element in an array

number of pass      comparison

8001h

2

1st pass

$n-1 \rightarrow n-1$

2nd pass

$n-2 \rightarrow n-2$

3rd pass

$n-3 \rightarrow n-3$

8000h

1

in ascending order

Two elements in a row are compared at a time.  
If in required order don't disturb if not then exchange

8004h    5    5    5    5    1  
           4    4    1    1    5  
           3    3    1    4    4  
           2    1    3    3    3  
 1st pass    8000h    1    2    2    2    2    1 comparison

8004h	5	5	5	5	1	
	4	4	1	1	5	
1st pass	3	3	1	4	4	
	2	1	3	3	3	
8000h	1	2	2	2	2	1 comparison

Step 1      Step 2      Step 3      Step 4

2nd pass    1    1    1    1  
           5    5    5    2    3 comparisons  
           4    4    2    5  
           3    2    4    4  
           2    3    3    3

2nd pass	1	1	1	1	
	5	5	5	2	3 comparisons
	4	4	2	5	
	3	2	4	4	
	2	3	3	3	

Step 1      Step 2      Step 3

3rd pass    1    1    1  
           2    2    2    2 comparisons  
           5    5    3    3  
           4    3    5  
           3    4    4

3rd pass	1	1	1	
	2	2	2	2 comparisons
	5	5	3	
	4	3	5	
	3	4	4	

Step 1      Step 2

4th pass    1    1  
           2    2    1 comparison  
           3    3  
           5    4  
           4    5

4th pass	1	1	
	2	2	1 comparison
	3	3	
	5	4	
	4	5	

Step 1

Bubble Sort

```

= org 0000h
MOV r2, #4 // (n-1) pass counter; n - number of elements of array
up: MOV a, r2 Here number of passes is equal to number of
    MOV r3, a comparison. Hence both counters are given same.
    MOV dptr, #8000h // same pointer needs to incremented and decremented
up: MOV r1, dph // lower byte address changes.
    cine a, b, nteq
    MOVX a, @dptr Initially ((dptr)) = (8000h) = 1h → (a)
    MOV b, a (b) = 1h
    inc dptr (dptr) = 8001h
    MOVX a, @dptr ((dptr)) → (a) = 2h
    cine a, b, nteq // if a > b: no change; a < b : exchange
    jmp nc for no change
    nteq: jc nc
    MOV dpl, r2 dptr is at 8001h so to make it 8000h
    MOVX @dptr, a exchange when a < b
    MOV a, b
    inc dptr
    MOVX @dptr, a
nc: dynx r3, up
    dynx r2, up1
end

```

★ Arithmetic operations:1. Addition

Syntax: ADD a, src ; (a)  $\leftarrow$  (a) + (src)

destination operand has to be accumulator.

source can be any 8 bit data

- 8 bit register (other than a) (can be a)

- memory location

- pointers

Ex: add a, #12h ;  $(a) \leftarrow (a) + 12h$   
 add a, 30h ;  $(a) \leftarrow (a) + (30h)$   
 add a, @x0 ;  $(a) \leftarrow (a) + ((x_0))$

Disadvantage

only two 8 bit numbers can be added at a time

Add with carry

Syntax: addc a, src ;  $(a) \leftarrow (a) + (src) + (cy)$

Ex: addc a, #12h ;  $(a) \leftarrow (a) + 12h + (cy)$

addc a, 30h ;  $(a) \leftarrow (a) + (30h) + (cy)$

addc a, @x0 ;  $(a) \leftarrow (a) + ((x_0)) + (cy)$

~~addc~~

Addition operation on 8 bit binary data

da a : decimal adjust : adjust content of accumulator after addition (implicit addressing mode).

1 - always checks the lower nibble and adjusts ~~the~~ the valid bcd numbers. Ex: adjust only 0 to 9 not A-F.  
 lower nibble of accumulator

If A-F adds a correction factor of 6. and then adjust

2 - checks the higher nibble and adjusts the valid bcd numbers  
 Higher nibble of accumulator

If A-F adds a correction factor of 60. and then adjust

3 - Auxiliary carry flag

It also checks the auxiliary carry flag, if high adds a correction factor of 6.

14h	41h	49h
+ 18h	+ 81h	+ 29h
2Ch	C2h	72h
- 06	+ 60	- 06
32d	182d	78d
①	②	③

\* Addition of two 8 bit numbers

```
= org 0000h
MOV a, #0abh
ADD a, #6fh
MOV r2, a // result in any register
MOV 30h, a // result in any memory location
MOV dptr, #8000h
MOVX @dptr, a // result in external memory.
end //
```

$$\begin{array}{r} 0abh \\ + 6fh \\ \hline \end{array}$$

\* Addition of two 16 bit numbers.

```
= org 0000h
MOV a, #0abh
ADD a, #0f4h
MOV r2, a
MOV a, #24h
ADDC a, #38h
end
```

$$\begin{array}{r} 24ab \\ + 38f4 \\ \hline \end{array}$$

Result is stored as

$$\begin{array}{r} cy \ a \ r_2 \\ 0 \ 5D \ 9F \end{array}$$

\* Sum of abh, f dh, e8h.

```
= org 0000h
MOV r2, #0 // carry tracker
MOV a, #0abh
ADD a, #0f4h
```

jnc next

inc r2

nxt : ADD a, #0e8h

jnc nxt1

inc r2

nxt1 : NOP

end.

addc should not be used while adding n numbers.

ab abcdef

cd + 123456

+ ef addc should

addc should be used.  
not be used

Result Hence one of the registers is  
 $\begin{array}{r} r_2 \ a \\ \hline 2 \ 90 \end{array}$  used as a carry tracker.

For number of elements equal to number of addition  
Initially we load accumulator with 0.

Looping can be used only when data of internal/external memory has to be added and not in immediate mode.

\* Addition of elements of an array in internal memory.

= Org 0000h

MOV r2, #10 // no of elements / no of addition

MOV r3, #0 // carry tracker

MOV r0, #30h // pointer.

Repeat: add a, @r0.

jnc next

inc r3

Result

r3 a

next: inc r0

djn r2, repeat

end.

\* Addition

= Org 0000h

MOV r2, #7 // no of elements

MOV r4, #0 // carry pointer.

MOV r0, #30h

MOV a, @r0

Repeat: add a, @r0.

jnc next1

inc r4

next1: inc r0

inc r0

djn r2,

end r2000

Result: r4 a.

3Dh	23	
3Ch	0A9	
3Bh	ab	
3Ah	ef	1234
39h	64	abcd
38h	23	5678
37h	65	6589
36h	87	6473
35h	56	abc7
34h	78	+ 23a4
33h	ab	
32h	cd	
31h	12	
30h	34	

\* Addition of two arrays.

Org 0000h

MOV r<sub>2</sub>, #5 // 5 bit number

MOV r<sub>3</sub>, #0 // carry tracker

MOV r<sub>0</sub>, #30h

MOV a, #50h

MOV dptr, #3000h

repeat: MOV a, @r<sub>0</sub>

addc a, @r<sub>1</sub>

MOVX @dptr, a

inc r<sub>0</sub>

inc r<sub>1</sub>

inc dptr.

djnx r<sub>2</sub>, repeat

jnc A stop

MOV a, #1

MOVX @dptr, a.

1	2	3	4	5	6	7	8	9	0
+ 0	9	8	7	6	5	4	3	2	1

54h	09	34h	12
53h	84	33h	34
52h	65	32h	56
51h	43	+ 31h	78
50h	21	30h	90

↓

3006h

3000h

2. Subtraction:

Syntax : subb a, src ; (a)  $\leftarrow$  (a) - (src) - (cy)

destination operand has to be accumulator.

source can be any -8 bit data

- 8 bit register

- memory location

- pointer

Ex : Subb a, #12h ; (a)  $\leftarrow$  (a) - 12h - (cy)

subb a, b; (a)  $\leftarrow$  (a) - (b) - (cy)

subb a, r<sub>2</sub>; (a)  $\leftarrow$  (a) - (r<sub>2</sub>) - (cy)

subtract with borrow.

Negative result is always present in its 2's complement  
Subtraction can also be performed by 2's complement  
method without using the subb instruction directly  
complement command

cpl a // 1's complement

inc a // 2's complement.

The operand has to be accumulator.

### 3. Multiplication:

Syntax: MUL ab

Numbers to be multiplied has to be in registers a and b only  
On multiplying two 8 bit numbers the result is 16 bit,  
lower byte result is available in a and higher byte  
result is available in b.

mul ab ; result : (b)(a)

It is realised internally by successive addition.  
This instruction takes 4 clock cycles since it has its own  
internal code.

### 4. Division:

Syntax DIV ab // a/b

Numbers to be divided has to be in register a and b only  
The dividend is in a and the divisor is in b. and the  
dividend has to be greater than the divisor. After division  
the quotient is available in a and the remainder is  
available in b.

\* Addition using daa instruction.

```
= org 0000h
MOV a, #84h
add a, #18h
da a
end.
```

d.a.a: to get bcd result

y4

$$\begin{array}{r} 18 \\ - 90 \\ \hline = 92 \text{ (BCD)} \end{array}$$

\* BCD Subtraction.

```
= org 0000h
MOV a, #2
cpl a
inc a
add a, #4
end
```

$$\begin{array}{r} 4 & \text{- 1st number} \\ - 2 \\ \hline 2 & \text{- 2nd number} \end{array}$$

```
= org 0000h
MOV a, #99h // 9's complement
SUBB a, #62h
inc a // 10's complement
add a, #28h
da a
end.
```

\* Multiplication : 16 bit x 8 bit

```
= org 0000h
MOV a, #56h
MOV b, #34h
MUL ab
MOV r2, a // 1st byte result (78)
MOV r3, b // save content of b. (11)
MOV a, #56h
MOV b, #12h
```

16 bit    8 bit

$$1234 \times 56$$

b a

$$1178 - 34 \times 56$$

b a

$$060C - 12 \times 56$$

$$\hline 061D78$$

Result: r4 r3 r2

MUL ab

add a,  $\bar{r}_5$

MOV  $r_3, a$  // 2<sup>nd</sup> byte final result (1D)

MOV a, b

addc a, #00h //  $(06 + 00h)^{CARRY}$

MOV  $r_1, a$  // 3<sup>rd</sup> byte final result

end.

\* Division: 16bit / 8bit

= Number of possible subtraction  
is equal to quotient in successive  
subtraction method for division.

Subtraction is continued till

- Higher byte is 0

- lower byte is less than divisor

Org 0000h

MOV dph, # 0344h // dividend

MOV b, # 88h // divisor.

MOV  $r_2, \# 00h$  // number of possible subtraction / quotient.

cnt: MOV a, dph //  $(a) = 44h$

subb a, b

MOV dph, a

jnc nxt

dec dph

nxt: inc  $r_2$

MOV a, dph

gne a, #00h, cnt

MOV a, dph

gne a, b, nteq

sjmp cnt

nteq: jnc cnt

end

344

88

Q = 3

R = 80

344 - 88 =

2/144 - 88 = 56 -

2/56 - 88 =

1/156 - 88 = 76 -

1/76 - 88 =

0/196 - 88 = 86 -

86 - 88 =

\* Square root of a number:

Number of possible subtraction in this logic gives the square root of the number.

$$\sqrt{25} = 5$$

$$25 - 1 = 24$$

$$24 - 3 = 21$$

$$21 - 5 = 16$$

5

Org 0000h

MOV a, #25h // number whose sqrt is to be found.  $9 - 9 = 0$

MOV b, #1 // to subtract with odd number

MOV r<sub>2</sub>, #0 // to track the number of possible subtraction and is the square root

cnt: subb a, b

jne stop // 0 - anything cy - 1

inc b

inc b

inc r<sub>2</sub>

sjmp cnt

stop: hlt

end

\* LCM of two numbers:

3, 15

= Org 0000h

3+3, 12

MOV r<sub>2</sub>, #3h

6, 12 - compare

MOV r<sub>3</sub>, #12h

6+3, 12

MOV a, r<sub>2</sub> // to retain the original

9, 12 - compare

MOV b, r<sub>3</sub> number.

9+3, 12

cnt: jne a, b, nteq

12, 12 - compare

sjmp stop // if equal to end.

and add until they are equal.

nteq: jnc more //  $b > a$  then

Always add to the smallest number

add a, r<sub>2</sub> add a with 3

sjmp cnt

more: xch a, b //  $a > b$

add a, r<sub>3</sub>

hlt,

xch a, b  
sjmp cnt  
stop: nop  
end.

\* GCD of two numbers:

= org 0000h

MOV a, #30h // dividend

MOV b, #12h // divisor

MOV r<sub>2</sub>, b // to save divisor as

cnt: div ab      b becomes remainder

MOV a, r<sub>2</sub>      after division. (r<sub>2</sub>) = 12h

MOV r<sub>3</sub>, b      // if r ≠ 0, remainder becomes divisor.

cjne r<sub>3</sub>, #00h, cnt

end

12,30

keep performing  
division until remainder  
is zero. If r ≠ 0 exchange  
the divisor and remainder.

12 | 30 | 2

24

6

6 | 12 | 2

12

0

i.e. 6 is the GCD

\* Factorial of a number

= org 0000h

MOV a, #1

MOV b, #6 // number whose factorial is to be found

repeat: MUL ab

dec b

cjne b, #0, repeat

end.

Limitation: suitable only for 8 bit numbers.

\* Generation of series

= First 10 natural numbers.

Org 0000h

MOV  $\gamma_2$ , #10 // number of elements in the series

MOV  $\gamma_0$ , #30h

MOV a, #1 // starting point

up: MOV @ $\gamma_0$ , a

inc a

inc  $\gamma_0$

djnz  $\gamma_2$ , up

end

= First 10 odd numbers

Org 0000h

MOV  $\gamma_2$ , #10

MOV  $\gamma_0$ , #30h

MOV a, #1 - first 10 even numbers.

up: MOV @ $\gamma_0$ , a

inc  $\gamma_0$

add a, #2

$\star^5$

daa

We dont double increment as on

incrementing it leads to a hexa decimal  
numbers.

djnz  $\gamma_2$ , up

multiples of 5.

end.

= Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, ...

Org 0000h

MOV  $\gamma_2$ , #8 // First 10 number of series (since first 2 numbers  
are stored)

MOV dptr, #3000h

MOV a, #0

MOVX @dptr, a

inc a

inc dptr

MOVX @dptr, a

is even/odd

MOV R<sub>3</sub>, #1

repeat: add a, R<sub>3</sub>

XCH a, R<sub>3</sub>

inc dptr

MOVX @ dptr, a

djnz R<sub>2</sub>, repeat

end

3009h

34

3

2

1

1

0

3000h

\* To check an odd number / even number

LSB : 0 - Even number

LSB : 1 - Odd number

= Org 0000h

MOV R<sub>0</sub>, #30h // To check the data present in some address

MOV a, @R<sub>0</sub> // To move to a bit address register.

jb acc.0, odd\_no // jb : if bit is 1.

mov 40h, #00h

sjmp stop

odd\_no : mov 40h, #0ffh

stop : nop

end

\* To check a positive / negative number

MSB : 0 - positive

MSB : 1 - negative.

= Org 0000h

MOV R<sub>0</sub>, #30h

MOV a, @R<sub>0</sub>

jb acc.7, negative\_no // ~~jb~~

MOV 40h, #00h

sjmp stop

negative\_no: MOV 90h, #0ffh  
Stop: nop  
end

- \* check the status of particular bit.

= org 0000h

by b<sub>6</sub> b<sub>5</sub> b<sub>4</sub> b<sub>3</sub> b<sub>2</sub> b<sub>1</sub> b<sub>0</sub>

MOV A, #30h // data to be checked

1 0 0 ① 0 0 1 1

MOV A, @R0

check this bit

jb acc. 4, make\_it\_zero

make it 0 if 1 and  
1 if 0.

Setb acc. 4

To change more than one bit  
we use logical instructions.

jmp stop

To change ~~acc~~ with 1

make\_it\_zero : clr acc. 4

To keep as it is xor with 0.

stop: nop

end.

- \* Logical group of instruction

orl a, src : or operation (setting)

anl a, src : and operation (masking)

xrl a, src : xor operation (complementing)

The ~~destination~~ has to accumulator and source can be memory  
register or pointer.

- \* To change more than one bit.

1010 1100 10011000  
ACh 98h

MOV A, #0ACh

XRL A, #98h

- \* Masking

to clear higher / lower bits we can use anl operator.

more than 1 bit without disturbing other bits

\* Org 0000h  
MOV a, #20h  
and a, #0fh  
end

- 25h  
0010 0101  
0000 1111  
0000 0101  
20h  
masking: clearing lower nibble

- \* setting  
setting more than one bit without disturbing other bits we use ORL operation.

- Org 0000h  
MOV a, #0eh  
ORL a, #0efh  
end

0eh  
1110 0000  
1110 1111  
1110 1111  
efh

- \* swap instruction (Logical grp of instruction)

syntax: swap a  
It swaps the lower nibble of a with the higher nibble of a.

Ex: Org 0000h  
MOV a, #25h  
swap a ; (a) ← 52h  
end

- \* Rotate Instructions

Rotate: no data is lost

Shift: data shifted out are lost and filled with 0.

In 8051 there is only rotate operation.

- Rotate left

Syntax: RL a  
RLC a (through carry)

- Rotate right

Syntax: RR a  
RRC a (through carry)

Ex:

$\text{cy}$   
 0 1 0 0 1 1 0 1 0

RL a 0 1 0 0 1 1 0 1 0  
 ← →

RR a 0 1 0 0 1 1 0 1 0  
 ↑ ← →

RLC a 0 1 0 0 1 1 0 1 0  
 ← →

RCR a 0 1 0 0 1 1 0 1 1  
 ← →

Note: swap a = 4 rotate instruction.

#### \* I/O ports:

There are four ports each of 8 bit. Each port can be addressed as 8 individual I/O pins. All the ports are bidirectional. Each port is addressed as one of 8 bit register.

80h : P0 - P0.0, P0.1 ... P0.7

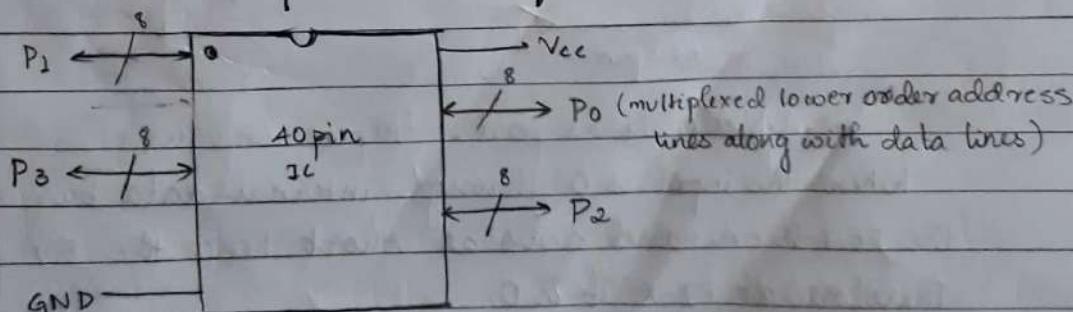
90h : P1 - P1.0, P1.1 ... P1.7

A0h : P2 - P2.0, P2.1 ... P2.7

B0h : P3 - P3.0, P3.1 ... P3.7

Any I/O device or external memory can be interfaced only through these I/O ports. Hence number of I/O pins determines the capacity of the microcontroller.

In 8051, it is 40 pin IC. out of which 32 pins are I/O pins.



- Alternate function

Port 3: P3.0 : RXD receive  
 P3.1 : TXD transmission } serial data  
 P3.2 : INT0 transmission } communication.  
 P3.3 : INT1 } external interrupts  
 P3.4 : T0 } input for the counter  
 P3.5 : T1  
 P3.6 : WR write  
 P3.7 : RD Read

Port 1: only I/O pins no alternate function.

Port 0: port 0 is also designated as AD0 - AD7, allowing it to be used for both address and data.

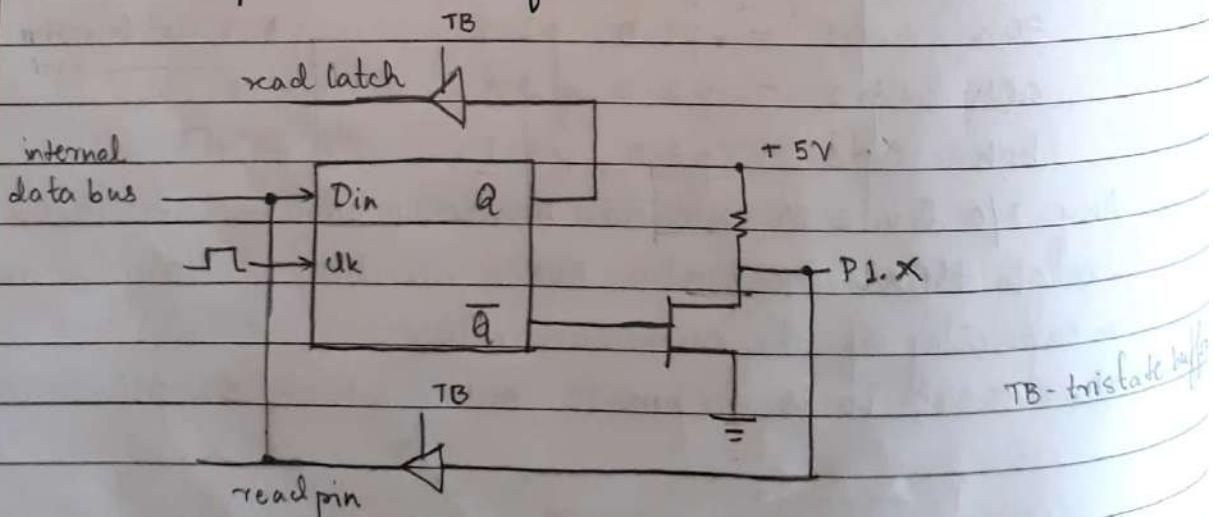
Port 2: port 2 is designated as AD8 - AD15, it is used for the upper of the 16 bit address.

By default (on reset) all the pins acts as output pins.

= Internal hardware configuration of port 1

(simple I/O operation only)

Each I/O pin consists of a latch



Individual port 1 I/O pin configuration

When the input is 0 through internal data bus  $\Rightarrow \bar{Q} = 1$  hence the FET is on and acts as short hence the 5V flows to ground. Therefore at P1.X it is 0.

When the input is 1 through internal data bus  $\Rightarrow \overline{Q} = 0$  hence the FET is off now there is 5V at P1.x.

To use as input pin/ port it has to be configured.  
that is FET should be off (i.e, open)  $\Rightarrow D_{in} = 1$

- \* To send and observe data

= org 0000h

MOV a, #12h

add a, #15h

loop: MOV P1,a : to send data

sjmp loop : to observe data for a while

end

(or)

to observe the result

we can make the controller alive

again: sjmp again

(infinite loop)

- \* To send and observe 16 bit data.

= org 0000h

MOV a, #f8h

MOV b, #0fdh

MUL ab

loop: MOV P0,a

MOV P1,b

sjmp loop

end

To use any port/pin for input it has to be configured.

- \* Port 0 as input and Port 1 as output (complementing input)

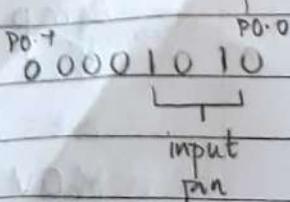
= org 0000h

MOV P0, #0fh // configure as input port

immediate addressing mode with any port/pin  
is not data but configuration.

loop: MOV a, P0

cpl a



MOV P1, a  
Sjmp loop  
end.

- \* Read data from more than one port

= Org 0000h

MOV P0, #0ffh // configuring P0 as input port

MOV P1, #0ffh // configuring P1 as input port

MOV a, P0 ]- read input  
MOV b, P1

MUL ab

MOV P2, a

MOV P3, b

end

- \* Read data from port 0, find the number of 1's and 0's. Then send the number of zeros to port 1 and send the number of ones to port 2.

= Org 0000h

MOV P0, #0ffh // configure port0 as input port

MOV Y1, #00h // number of 0's

MOV Y2, #00h // number of 1's

MOV Y3, #8 // counter: number of bits to be checked

MOV a, P0

repeat: RLC a

jc ones

inc Y3

Sjmp nxt

ones: inc Y1

nxt: djnz Y3, repeat

MOV P1, Y3

MOV P2, R4

again: sjmp again  
end.

\* To find number of zeroes and ones in a 16 bit data.

= 00g 0000h

MOV P0, #0ffh // Configuring port0 and port1 to  
MOV P1, #0ffh read 16 bit data

MOV A, P0

MOV B, P1

MOV R2, #08 // counter

MOV R3, #0 // number of zeroes

MOV R4, #0 // number of ones

repeat: RLC A

jc ones

inc R3

sjmp nxt

ones: inc R4

nxt: djnz R2, repeat

MOV A, B. // to check higher byte

MOV R2, #8

repeat1: RRC A

jc ones1

inc R3

sjmp nxt1

ones1: inc R4

nxt1: djnz R2, repeat1

end

Page

\* Number of odd and even numbers.

= Org 0300h

array: db 01h, 64h, 0abh, 74h, 80h,  
 33h, 59h, 43h, 08h, 16h

To store data sequence of data

Org 0000h

MOV r<sub>2</sub>, #10

MOV r<sub>0</sub>, #30h

MOV a, #1

up: mov r<sub>0</sub>, a  
 inc r<sub>0</sub>  
 inc a  
 djnz r<sub>2</sub>, up  
 end

negative

MOV r<sub>3</sub>, #0 // number of odd numbers

positive

MOV r<sub>4</sub>, #0 // number of even numbers

repeat: clr a  
 MOV a, @a+dptr

jb acc.0, odd  
 inc r<sub>0</sub> acc.1

sjmp next

odd: inc r<sub>3</sub>

next: inc dptra  
 djnz r<sub>2</sub>, repeat  
 end

Random data

Org 0000h

MOV 30h, #78h

MOV 31h, #0abh

MOV 32h, #01h

store array of data.

Org 0300h

array: db 64h, 0abh, 74h  
 move a, @a+dptra

\* To separate odd and even numbers

= Org 0300h

array: db 01h, 64h, 0abh, 74h, 80h, 33h, 59h, 43h, 0eh, 16h

Org 0000h

MOV r<sub>2</sub>, #10

MOV dptra, #array negative

MOV r<sub>0</sub>, #30h // to store odd numbers

MOV r<sub>1</sub>, #50h // pointer to store even numbers, positive

Report: clra

MOV a, @a+dptr

jb acc.0, odd

MOV @r<sub>1</sub>, a

inc r<sub>1</sub>

sjmp nxt

odd: MOV @r<sub>0</sub>, a

inc r<sub>0</sub>

nxt: inc dptra

~~inc r<sub>1</sub>~~

djnz r<sub>2</sub>, repeat

end.

\* In array of elements sum of only odd/even numbers.

= org 0000h

array: db 01h, 64h, 0abh, 74h, 80h, 33h, 59h, 43h, 0eh, 16h

org 0000h

MOV dptra, # array

MOV r<sub>2</sub>, #10

MOV r<sub>3</sub>, #0 // sum of odd numbers

MOV r<sub>4</sub>, #0 // sum of even numbers

repeat: clra

MOV a, @a+dptra

jb acc.0, odd

add a, r<sub>3</sub>

MOV r<sub>4</sub>, a

sjmp nxt

odd: add a, r<sub>3</sub>

MOV r<sub>3</sub>, a

nxt: inc dptra

djnz r<sub>2</sub>, repeat

end.

\* 2 out of 5 code: (with respect to 8 bit data)  
It has to satisfy two conditions to check whether it belongs to the group or not.

1. First three MSB bits must be zero

2. In the remaining five bits, the number of one's must be two

= Assume data to be read from port 0.

org 0000h

MOV P0, #0ffh // configuring port 0.

MOV a, P0

MOV b, a // to retain original data  
and a, #0e0h

jnz invalid

MOV r2, #5 // lower 5 bits.

MOV r3, #0 // to hold number of ones

MOV a, b

repeat: RRC a // since lower 5 bits is checked

jnc nxt

inc r3

nxt: djnz r2, repeat

cjne r3, #2, invalid

MOV 40h, #0ffh // if condition is satisfied.  
sjmp stop

invalid: MOV 40h, #00h // if condition is not satisfied  
stop: nop  
end.

13 X      12 /  
0001 0011    0001 0010  
✓ X           ✓  
1110 0000    ✓ ✓

To check this  
we perform and  
operation with  
11100000 (eoh).

If the result is 0  
then condition 1 is  
satisfied.

## \* Code Converters:

### 1. BCD

#### - Packed BCD to unpacked BCD [32 = 0302]

org 0000h

MOV P0, #0fh // P0 as input port

MOV a, P0

MOV b, a

and a, #0fh // 02

MOV P1, a

MOV a, b

and a, #0fh // 30

swap a // 03

MOV P2, a

end.

#### - Unpacked BCD to packed BCD

org 0000h

MOV P0, #0fh // P0 as input port

MOV P1, #0fh // P1 as input port

MOV a, P0 // higher byte unpacked BCD

swap a

MOV b, P1 // lower byte unpacked BCD

add a, b

MOV P2, a

end.

### 2. Hexadecimal to ASCII and ASCII to hexadecimal.

Hex → ASCII

Org 0000h

MOV P0, #0fh // input port

MOV a, P0

cine a, #0ah, nteq

rjmp nxt  
 nteq : jneq nxt1  
 nxt : add a, #1  
 nxt1 : add a, #30h  
 MOV P1, a  
 end.

ASCII to hex : org 0000h  
 MOV P0, #0ffh // input port  
 MOV a, P0

### 3. BCD to ASCII and vice versa (Packed BCD → ASCII)

BCD to ASCII : org 0000h  
 MOV P0, #0ffh // input port.  
 MOV a, P0 // 45  
 MOV b, a // 45  
 anl a, #0fh // 05  
 add a, #30h // 35  
 MOV P1, a  
 MOV a, b  
 anl a, #0fh // 40  
 swap a // 04  
 add a, #30h // 34  
 MOV P2, a  
 end

ASCII to BCD: Org 0000h

MOV po, #0ffh // input port

MOV a, po // '4'

and a, #0fh

swap a

MOV b, a

H

\* Subprogram: in assembly language: subroutine.

With respect to processor subroutine can be realised using procedure or macro.

Procedure : a. near procedure

b. far procedure.

Subroutine can be written anywhere in the program memory.

To call the subroutine:

1. ACALL : Absolute call : [0 - 2k]

2. LCALL: long call : [more than 2k]

\* BCD → ASCII

Org 0000h

MOV po, #0ffh

MOV a, po

MOV b, a

LCALL conversion

MOV pl, a

Subroutine - conversion

Org 0400h

conversion: and a, #0fh

add, #30h

ret

end

MOV a, b

swap a

LCALL conversion

MOV p2, a

end.

#### 4. BCD to Hexadecimal and vice versa

BCD to hexa: Org 0000h

MOV po, #0fh

MOV a, po // 32 BCD

MOV b, a

and a, #0fh // 02

MOV r2, a

MOV a, b

and a, #0f0h // 30

swap a // 03

MOV b, #0ah

MUL ab

add a, r2

MOV p1, a

end.

Hexa to BCD:

32 → 20h

$$03 \times 0ah + 02 \times 1$$

$$= 1E + 2$$

$$= 20h$$

0C0 → 20h

0X16 + 1

### 5. Binary to Gray

Org 0000h  
 MOV P0, #0ffh  
 MOV Y2, P0

MOV A, Y2

RRC A

XRL A, Y2

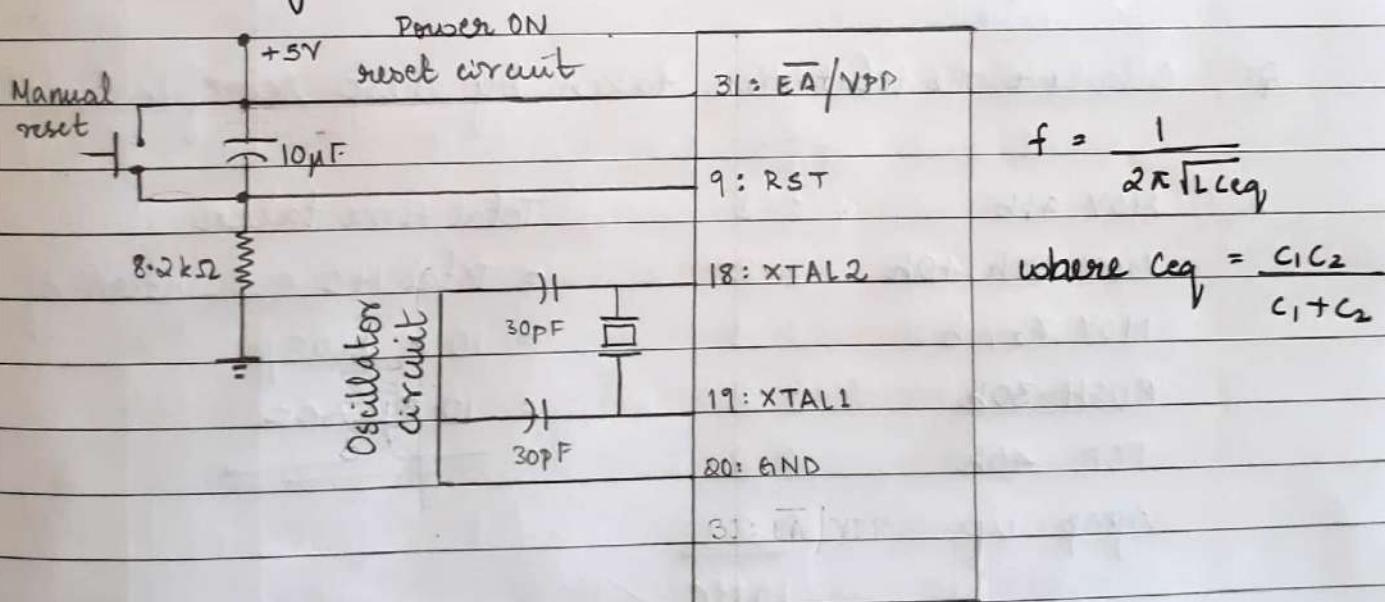
end.

\* INTERFACING: Pin configuration of 8051: text book pg 184.

Operating frequency of 8051 = 11.0592 MHz

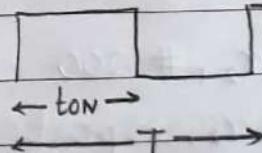
This frequency is suitable for baud rate of 9600.

Reset is an active high instruction - A controller takes 2 machine cycles to reset.



- Duty cycle:

$$\delta = \frac{t_{on}}{T}$$



where  $t_{on}$ : time it is ON

$T$ : total time period.

- Instruction cycle:

Time taken to complete one instruction operation. It consists of machine cycles.

Each machine cycle consists of  
12 clock cycles.

Operating frequency

$$f = 11.059 \text{ MHz}$$

Time period :  $T = 0.09 \mu \text{sec}$

∴ Duration of 1 machine cycle =  $0.09 \mu \text{sec} \times 12 = 1.08 \mu \text{sec}$

As reset takes 2 machine cycles

∴ It takes  $1.08 \mu \text{sec} \times 2 = 2.16 \mu \text{sec}$  to reset.

NOTE :

- Most of the instructions take 1MC. : All instructions take 1 MC other than memory related instructions.
- All memory related instructions take 2MC : push, pop, etc.
- Only two instructions take 4MC : MUL and DIV.

Q: Calculate the total time taken by these set of instructions.

		Total time taken
1.	MOV a,b : 1	
	MOV 30h, 40h : 2	= Total MC × Duration of 1MC
	MOV r2, b : 1	= $10 \times 1.08 \mu$
	PUSH 30h : 2	= $10.8 \mu \text{sec}$
	POP 40h : 2	<hr/>
	jmp up. : 2	<hr/>
		<hr/>
		<u>10 MC</u>

2. delay: MOV r2, # 200 :  $1 \times 1 = 1$

repeat: djnz r2, repeat :  $2 \times 200 = 400$

ret

:  $2 \times 1 = 2$

403 NC

∴ Total time taken =  $403 \times 1.08 \mu = 135.24 \mu \text{sec}$

Instruction cycle  
Machine cycle

NC1 NC2

3. delay: MOV  $r_2, \#200$  :  $1 \times 1$  = 1  
 up1: MOV  $r_3, \#200$  :  $1 \times 200$  = 200  
 up: djnz  $r_3, up$  :  $2 \times 200 \times 200$  = 80000  
 djnz  $r_2, up1$  :  $2 \times 200$  = 400  
 ret. :  $2 \times 1$  = 2  
 Total time taken = 80603MC  
 $= 80603 \times 1.08\mu = 87.051\text{msec} //$

4. delay: MOV  $r_2, \#200$  :  $1 \times 1$  = 1  
 up1: MOV  $r_3, \#200$  :  $1 \times 200$  = 200  
 up: nop :  $1 \times 200 \times 200$  = 40000  
 djnz  $r_3, up$  :  $2 \times 200 \times 200$  = 80000  
 djnz  $r_2, up1$  :  $2 \times 200$  = 400  
 ret :  $2 \times 1$  = 2  
 Total time taken = 120603 MC  
 $= 120603 \times 1.08\mu = 130.251\text{msec} //$

5. delay: MOV  $r_2, \#0ffh$  :  $1 \times 1$  = 1 because ffh =  
 up1: MOV  $r_3, \#0ffh$  :  $1 \times 255$  = 255  
 up: djnz  $r_3, up$  :  $2 \times 255 \times 255$  = 130050  
 djnz  $r_2, up1$  :  $2 \times 255$  = 510  
 ret :  $2 \times 1$  = 2  
 Total time taken = 130818 MC  
 $= 130818 \times 1.08\mu = 141.283\text{mS} //$

Q: calculate the time taken by 8051 microcontroller to PUSH content of dptr, b, a, PSW and PC.

= For dptr  $\rightarrow$  push 82h : 2 ] 4MC  
 push B3h : 2

For b  $\rightarrow$  push 0fh : 2MC

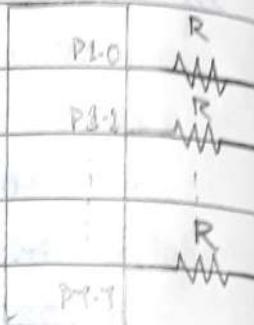
For a  $\rightarrow$  push 0eh : 2MC

For PSW  $\rightarrow$  push PSW : 2 MC  
 For PC  $\rightarrow$  push PC : 2 ] 1 MC  
 push PLU : 2

- = If executed 10 times an approximate delay of one second can be introduced. To introduce delay we can use two methods
  - $\rightarrow$  software delay : executing certain instructions certain number of times gives delay but inaccurate.
  - $\rightarrow$  hardware delay : also called as time delay. It is one of the on-chip resources

- R : current limiting resistor

Due to interfacing of input/output devices there may be excessive flow of current due to which controller might get damaged. Hence the current limiting resistor limits the current flowing from or into the PC.



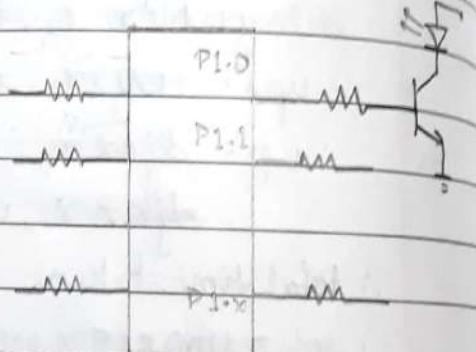
- For all output devices a driver

is essential as the current

received by that pin may not be sufficient to drive the device.

It drives the input current or voltage to required level.

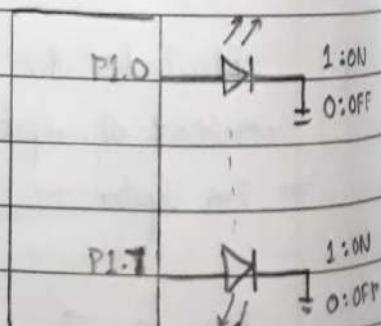
Simplest driver is a transistor.



- For each port we interface 8 LED's

At the port if 1 then LED will be ON or if 0 then LED will be OFF.

On setting of these 8 LED's we can implement 8 bit upcounter, down counter or up-down counter.



\* Implementing 8 bit up counter with delay of 1 sec

= Org 0000h off

MOV a, #00h

cnt: MOV p0, a for down  
ACALL delay counter

dec inc a off

cjne a, #00, cnt

back: sjmp back

end

delay subroutine:

delay: MOV r2, #10

up2: MOV r3, #255

up1: MOV r4, #255

up: djnx r4, up

djnx r3, up1

djnx r2, up2

ret

\* BCD up counter

= Org 0000h

repeat: MOV a, #00<sup>99h</sup>

cnt : MOV p0, a

ACALL delay

add a, #1<sup>99h</sup>

daa

<sup>99h</sup>

cjne a, #00, cnt

end

delay: MOV r2, #10

up2: MOV r3, #255

up1: MOV r4, #255

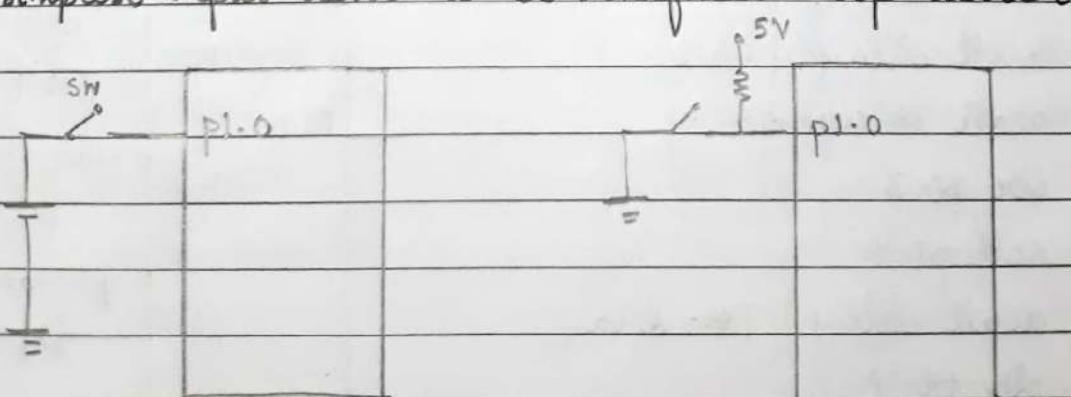
up: djnx r4, up

djnx r3, up1

djnx r2, up2

ret

\* simplest input device to be interfaced: Dip switch



pressed: 1

not pressed: 0

pressed: 0

not pressed: 1

\* One switch for one input pin: LINEAR INTERFACE

= Org 0000h

repeat: setb P1.0 // ip pin

jb P1.0, led Y

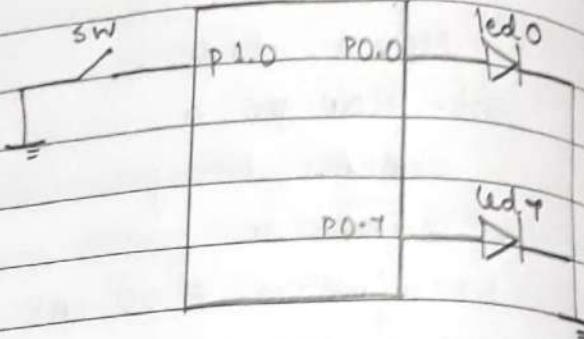
setb P0.0

jmp next.

led Y: setb P0.Y

nxt: jmp repeat

end



\* TRAFFIC LIGHT

= Org 0000h

repeat: setb P0.0

acall delay

acall delay 5 sec

acall delay delay

acall delay

acall delay

clr P0.0

setb P0.3

acall delay

acall delay 3 sec

acall delay delay

clr P0.3

setb P0.Y

acall delay // 1 sec delay

clr P0.Y

jmp repeat

end

delay routine: introduces delay  
delay: MOV r<sub>2</sub>, #10 of 1 sec

up2: MOV r<sub>3</sub>, #255

up1: MOV r<sub>4</sub>, #255

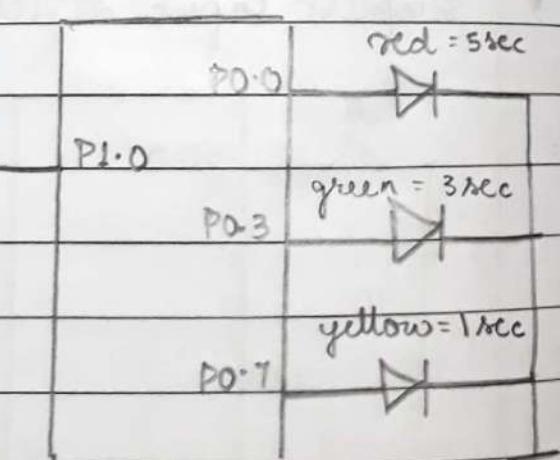
up: djnz r<sub>4</sub>, up\*

djnz r<sub>3</sub>, up1

djnz r<sub>2</sub>, up2

ret

loop.



### \* BLINKING OF LED:

All the leds at the same time

= org 0000h

delay : MOV r<sub>2</sub>, #10

repeat: MOV a, #0ffh

up2: MOV r<sub>3</sub>, #255

MOV p0, a

up1: MOV r<sub>4</sub>, #255

ACALL delay

up: djnx r<sub>4</sub>, up

MOV a, #00h

djnx r<sub>3</sub>, up1

MOV p0, a

djnx r<sub>2</sub>, up2

ACALL delay

ret

sjmp repeat

zhet.

end.

or:

Org 0000h

p0.0



repeat: cpl a

p0.1



MOV p0, a

p0.4



ACALL delay

sjmp repeat

end.

### Blinking of particular led

= org 0000h

delay : MOV r<sub>2</sub>, #10

repeat: cpl a

up2: MOV r<sub>3</sub>, #255

MOV p0, a

up1: MOV r<sub>4</sub>, #255

ACALL delay

up: djnx r<sub>4</sub>, up

sjmp repeat

djnx r<sub>3</sub>, up1

end.

djnx r<sub>2</sub>, up2

ret

exit.

led with switch

```
= org 0000h
repeat: setb p0.0
        jb p0.0, led7
        setb p0.0
        acall delay
        clr p0.0
        acall delay
        led7: setb p0.7
        acall delay
        clr p0.7
        acall delay
        sjmp repeat
end.
```

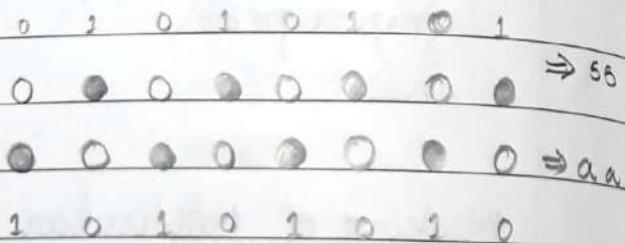
= org 0000h

```
repeat: setb p0.0
        move c, p0.0
        jc, led7
        setb p0.0
        acall delay
        clr p0.0
        acall delay
        led7: setb p0.7
        acall delay
        clr p0.7
        acall delay
        sjmp repeat
end.
```

diagrams

## Alternate led blinking

```
= org 0000h
repeat: mov a, #55h
        mov p0, a
        acall delay
        mov a, #aah
        mov p0, a
        acall delay
        sjmp repeat
end.
```



## UNIT - 3

### 8051 Programming in C

- c language is machine independent whereas assembly level language is machine dependent.
- c language requires more memory than assembly language.
- Embedded C:
  - All the registers should be used in capital letters only if not it is considered as a variable.  
(b: variable B: register)

#### \* Data types:

Ex: unsigned char a  
      unsigned int a

Appropriate data types should be used to define appropriate variable to save memory.

unsigned char a : 0 to 255  
unsigned int a : 0 to 5536

#### Data types along with range

- i. signed char a : -128 to 127 (8 bit)
- ii. unsigned char a : 0 to 255 :  $2^8 - 1$
- iii. signed short char a : -32768 to +32767 (16 bit)
- iv. unsigned short char a : 0 to 65535 :  $2^{16} - 1$
- v. signed int : -32768 to +32767
- vi. unsigned int : 0 to 65535
- vii. signed long : -2147483648 to +2147483647 (32 bit)
- viii. unsigned long : 0 to 4294967295 :  $2^{32} - 1$

These data types are used in C language, along with this there are particular data types for particular microcontrollers.

- i. sbit - to define any bit of any bit addressable as a variable
- Ex: sbit led0 = p1^0 ;      sbit led4 = p1^4 ;  
      sbit led0 = p1^0 ;      sbit s10 = p2^0 ;

iii. sfr : special function register : with respect to byte.

Ex: sfr input = 0x80; port 0

sfr output = 0x90; port 1

sfr led0 = 0xa0; port 2

sfr sios = 0xb0; port 3

sfr acc = 0xe0; accumulator

sfr is used to address any register as a byte through variable.

\* To send data to supervisor  
in binary form.

= #include <reg51.h>

sfr output port = 0x80;

void main (void)

{

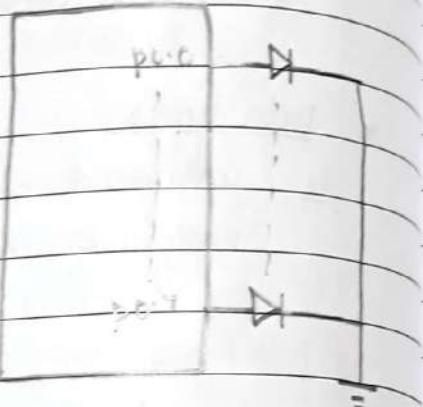
    while (1)

        output port

        PO = 0X20;

}

}



\* To switch on only led 0 and led 7.

= #include <reg51.h>

sbit led0 = PO^0;

sbit led7 = PO^7;

void main()

{

    while (1)

{

        led0 = 1;

        led7 = 1;

}

}

To blink LEDs of port 0.

= #include <reg51.h> to declare as  
void delay(); → global function  
{ → then it can be written  
after main

unsigned int x, y;

for (x=0; x <= 60000; x++)

1275 : for delay of  
1msec

delay function

- if local function: before main
- if global function: anywhere.

software  
delay time  
when not  
known accurately.

Just to have }  
delay }  
} delay

void main()

{

while (1)

{

P0 = 0x00;

delay();

P0 = 0xff;

delay();

}

}

\* To blink certain LEDs for certain time

= #include <reg51.h>

sbit led0 = P0^0; //int)

sbit led3 = P0^3;

sbit led7 = P0^7;

void delay(unsigned int);

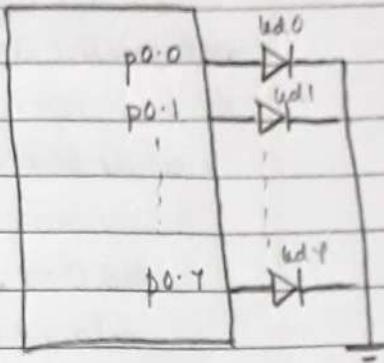
void delay(unsigned int x)

{

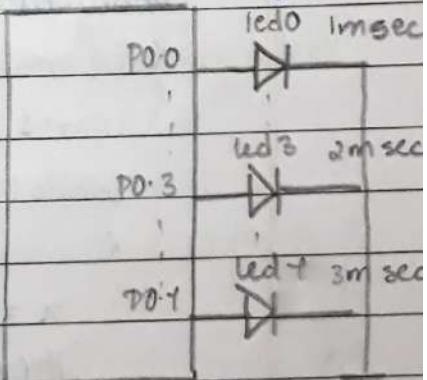
unsigned int i;

for (i=0; i <= x; i++)

}



Assuming 1275 gives  
1ms delay.



```

void main()
{
    while(1)
    {
        led 0 = 1;
        delay (1275);
        led 0 = 0;
        led 3 = 1;
        delay (2550);
        led 3 = 0;
        led 7 = 1;
        delay (3825);
        led 7 = 0;
    }
}

```

\* To blink led based on the status of the switch.

= #include <reg51.h>

sbit sw = P2^0;

sbit led0 = P0^0;

sbit led7 = P0^7;

void delay(unsigned int);

unsigned char x;

void main()

{ sw = 1; //configure as input pin

while(1)

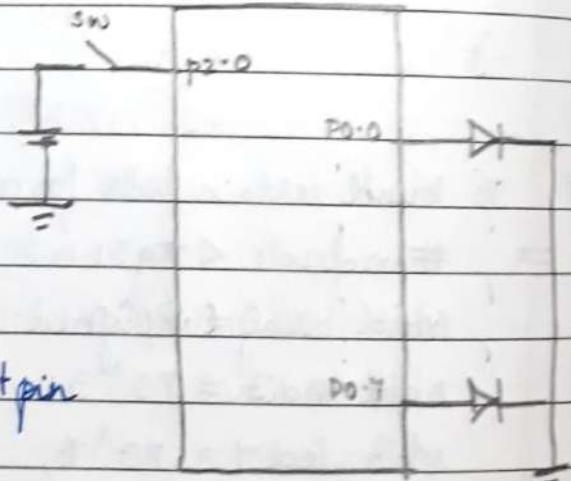
{

x = sw;

if (x == 1)

{

led 0 = 1;



logical operators used

in embedded c

1. cpl a : ~
2. anl a = &
3. orl a : |
4. xrl a : ^

shift operators:

1. >> shift right  
Ex:  $a = a \gg 3$  <sup>no of bits</sup>
2. << shift left  
Ex:  $a = a \ll 2$

led Y = 1;

delay (500);

led Y = 0;

delay (500);

}

}

void delay (unsigned int);

}

unsigned int i;

for (i=0; i &lt;= a; i++);

}

\* Timers and counters:

- Greater the number of timers / counters, powerfull the micro-controller.

Timers:

8051 has five timers which can also be used as counters.

Hence it can be used as:

- 2 timers : timer 0 and timer 1
- 2 counters : counter 0 and counter 1
- 1 timer and 1 counter.

Each timer / counter are 16 bit.

- Each timer consists of two registers : TH0 TL0 : timer 0  
TH1 TL1 : timer 1

higher byte  
lower byte

Each register can be addressed individually as TLO, THO, TL1 and TH1. The time is determined by the value of these registers.

- Timer/counter can be configured to work in different modes by using TMOD register (timer mode register).  
(89h)

TMOD	gate	C/T	M1	M0	gate	C/T	M1	M0
	Timer1/counter1					Timer0/counter0		

It is byte addressable only.

- Timer/counter can be started or stopped by using TCON register (timer control register).  
(88h)

TCON	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
	Timer/counter operation					Interrupt operation		

It is bit addressable register.

#### \* TMOD register

m1 and m0 : used to configure to different modes

m1      m0

0      0 : mode 0

0      1 : mode 1

1      0 : mode 2

1      1 : mode 3

C/T : If 1 : acts as counter

If 0 : acts as timer

Gate : If 0 : software startup

If 1 : hardware startup : interrupt startup.

#### \* TCON register

lower 4 bit : Interrupt operation

Upper 4 bit : Timer/counter operation.

TR0 : 1 : Start timer 0 ; 0 : stop timer 0

TF0 : Timer Flag : increments at the rate of timer and reaches maximum value and rolls back to initial value and sets to 1.

TR1 : 1 : Start timer 1 ; 0 : stop timer 1

TF1 : Timer flag : increments at the rate of timer and once reaches maximum value it rolls back to initial / zero value and sets to 1 indicating overflow.

I E1 : External interrupt 1 edge flag. set when hardware interrupt.

I T1 : Interrupt 1 type control bit. set / cleared : falling edge / level triggered external interrupt.

I EO : Ext Int 0 edge flag. Set when hardware interrupt. cleared once served

I TO : Int 1 type control bit. set / cleared by software : falling edge / level triggered external interrupt.

\* Timer can be used as

- time reference : duration of event / process
- delay generation
- waveform generator : to generate square wave form only  
(symmetric and unsymmetric (pulse) squarewave)
- to measure the width of the pulse : measure on time.
- to measure frequency of the pulse :
- to measure the speed of a moving system : rpm
- schedule interrupt and also periodic generation of an interrupt
- baud rate generator.

\* Source for the timer : crystal oscillator.

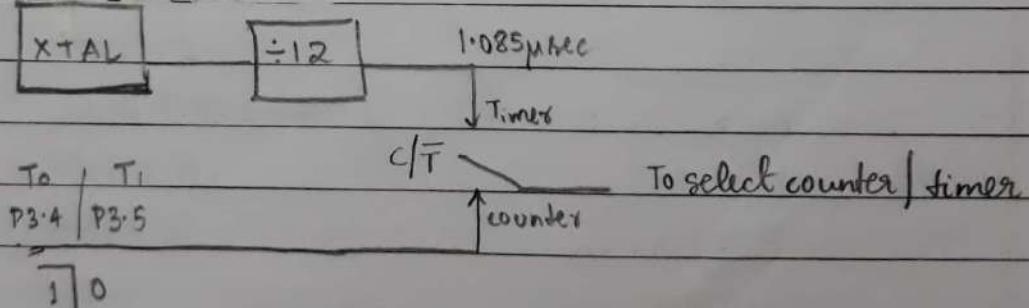
Timer rate depends on the frequency of the crystal.

For 8051 : 11.0592 MHz

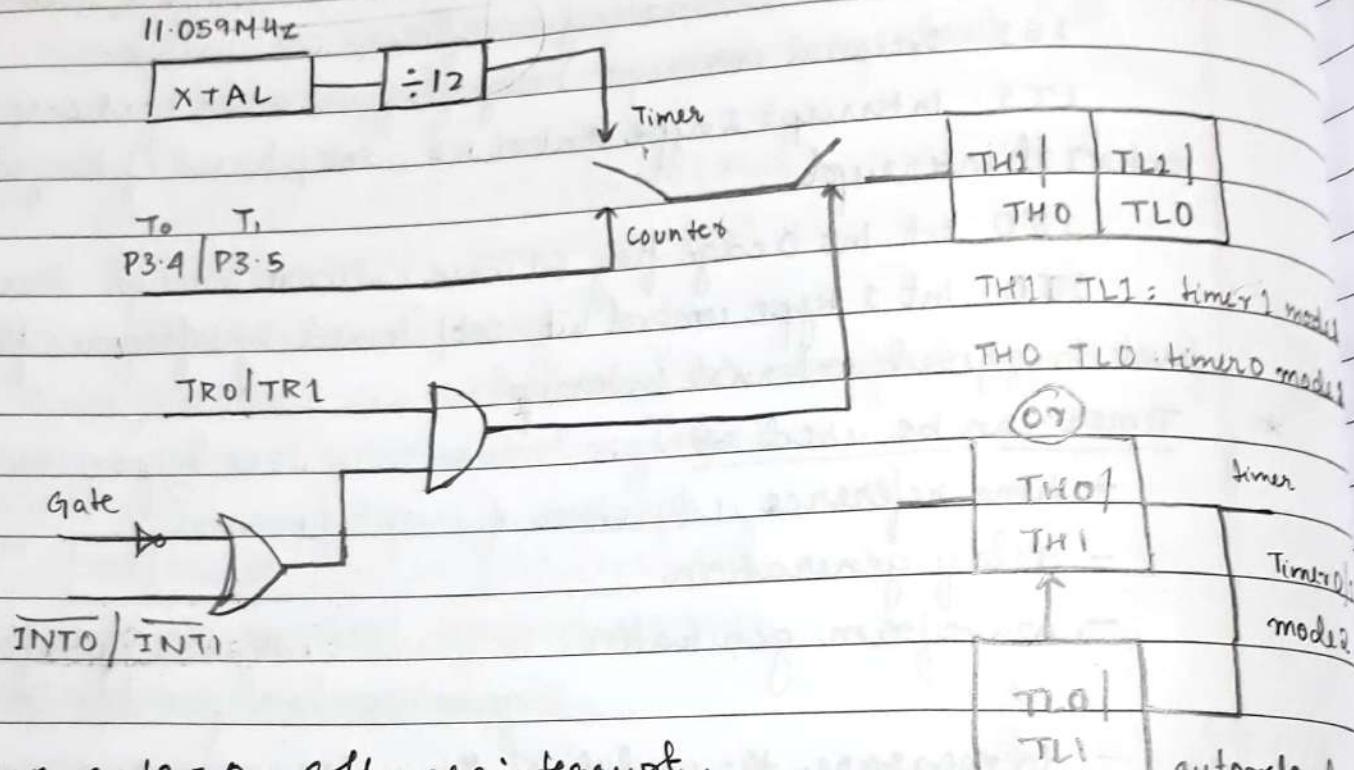
\* Source for the counter : pulse applied externally at the ports P3.4 for counter 0 and P3.5 for counter 1.

Duration of pulse = 2 MC ; The transition has to be 1 to 0.

11.0592 MHz



To start timer / counter through hardware startup or software startup.



When gate = 0: software interrupt.

Here the timer is started with instructions

TR0 and TR1 for timer 0 and 1 respectively.

When Gate = 1: Hardware interrupt.

The start and stop of the timer are done externally through INT0 and INT1 (i.e., pins P3.2 and P3.3) for timers 0 and 1 respectively.

## Modes of Timer / counter

\* Mode 1 : 16 bit mode  $\rightarrow$  FFFFh = 65535

generally preferred to generate delay and time reference  
Develop an 8051 C program to generate a delay of 0.25 msec.  
use timer 0 in mode 1

Initial value to be loaded to the timer

$$\text{count} = \frac{\text{required delay}}{\text{time scale}}$$

$$\text{count} = \frac{0.25 \text{ m}}{1.085 \mu} \approx 230$$

$$\text{Max value of 16 bit - count} = 65535 - 230 = 65305 = \underline{\underline{FF19h}}$$

Now FF19h is loaded initially to the timer. TMOD

In ALP:

delay: MOV timer, #01h

MOV tLo, #19h

MOV tho, # off

setb tro

check: jnb tf0, check // in loop when

dr t<sub>SO</sub> // stops timer. TFO = 0.

dr tfo

get

Inc:

void delay()

1

TMOD = 0XD1 ;

TLO = 0x19;

THO = 0xFF;

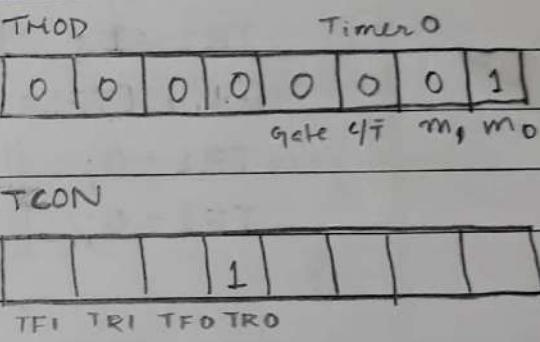
$$TRO = 1;$$

while ( TFO == 0 );

$$TRO = 0;$$

$$TFD = 0;$$

3



- \* To generate a delay of 50 msec using timer 1 in mode 1.  
 Assume crystal frequency = 11.0592 MHz.
- = Initial value to be stored  
 count :  $\frac{\text{delay required}}{\text{timer rate}} = \frac{50\text{m}}{1.085\mu} = \underline{\underline{46083}}$
- $65535 - 46083 = 19452 = \underline{\underline{4BFC\text{ h}}}$

void delay()

{

TMOD = 0x10; // Timer 1: mode 1

TL1 = 0xFC; // amount of delay

TH1 = 0x4B; to be generated.

TR1 = 1; // start timer

while (TF1 == 0); (statement)

TR1 = 0; // stop the timer

TF1 = 0; // reset the flag

}

Timer 1							
Gate	C/T	m <sub>1</sub>	m <sub>0</sub>	0	0	0	0
0	0	0	1	0	0	0	0

Timero / not used

TF1	TR1	TF0	TR0				
1							

4BFC → FFFF

then it reverts to zero

and timer flag TF1 sets to 1.  
 Hence its reset and stopped.

NOTE: Maximum delay we can obtain by timer 1 and timer 0 in mode 1 is 71.07m sec. (i.e.,  $65535 \times 1.085\mu$ )

- \* To generate a delay of 1sec using timer 1 in mode 1.  
 Assume crystal frequency = 11.0592 MHz.

= void delay

{ unsigned char x;

for (x=1; x <= 20; x++)

{

TMOD = 0x10;

TL1 = 0xFC;

TH1 = 0x4B;

```
TR1 = 1;
```

```
while (TF1 == 0);
```

```
TR1 = 0;
```

```
TF1 = 0;
```

```
}
```

```
}
```

Mode 2: 8 bit mode  $\rightarrow$  FFh = 255

- Autoreload mode

generally preferred for baud rate generation.

It can also be used for delay generation but it can generate a delay only upto. 0.24 m sec. hence not preferred.  
 $(255 \times 1.085\mu)$

Hence we use timer 1 in mode 2 to generate pulse.

Waveform generation:

We can generate pulse with variable duty cycle using timer 1 or timer 0.

\* Develop an 8051 C program to generate a square wave of frequency 2 kHz using timer 0 in mode 2 with 50% duty cycle at pin P2.0.

$$f = 2 \text{ kHz} \Rightarrow T = 0.5 \text{ m sec}$$

$$T = t_{on} + t_{off}$$

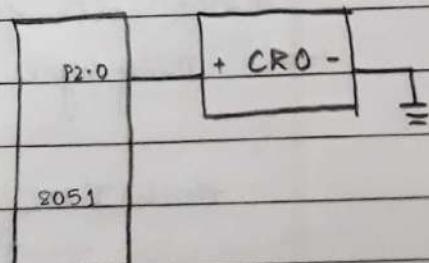
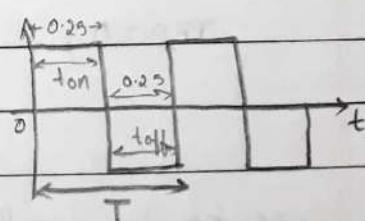
$$50\% \text{ duty cycle} \therefore t_{on} = t_{off}$$

$$\therefore t_{on} = t_{off} = 0.25 \text{ m sec}$$

$$\text{count} = \frac{\text{delay required}}{\text{timer rate}}$$

$$\text{count} = \frac{0.25 \text{ m}}{1.085\mu} = 230$$

$$255 - 230 = 25 = 19 \text{ h} //$$



```
#include <reg51.h>
sbit sqa = P2^0;
void delay();
void main()
{
```

```
    while(1)
```

```
{
```

```
    sqa = 1;
    delay();
    sqa = 0;
    delay();
}
```

```
}
```

```
void delay()
```

```
{
```

```
    TMOD = 0x02; // timer 0; mode 2
```

```
    TH0 = 0x19;
```

```
    TR0 = 1;
```

```
    while(TFO == 0);
```

```
    TR0 = 0;
```

```
    TFO = 0;
```

```
}
```

- \* Use T0 in mode 1 to produce a square wave of frequency 4 kHz with 20% duty cycle at pin P2.0.

$$= \text{Duty cycle} : S = \frac{t_{on}}{T}$$

$$\text{Here } T = \frac{1}{f} = \frac{1}{4k} = 0.25 \text{ m sec}$$

$$\therefore \frac{t_{on}}{0.25 \text{ m}} = \frac{20}{100} \Rightarrow t_{on} = 50 \mu \text{sec} = \underline{\underline{0.05 \text{ m sec}}}$$

$$T = t_{on} + t_{off}$$

$$\Rightarrow t_{off} = T - t_{on} = 0.25 \mu s - 0.05 \mu s$$

$$t_{off} = \underline{\underline{0.2 \mu s}}$$

$$\therefore t_{off} = 4 t_{on}$$

$$\text{count} = \frac{\text{delay required}}{\text{timer rate}} = \frac{0.05 \mu s}{1.085 \mu s} = 46$$

$$\text{Timer rate} = \left[ \frac{\text{crystal frequency}}{12} \right] = \left[ \frac{1.0592 \text{ MHz}}{12} \right] = \underline{\underline{1.085 \mu s}}$$

$$65535 - 46 = 65489 = \underline{\underline{FFD1H}}$$

```
#include <reg51.h>
```

```
abit sqw = P2^0;
```

```
void delay();
```

```
void main()
```

```
{
```

\* Passing parameter technique

```
    while(1)
```

```
{
```

```
    sqw = 1;
```

```
    delay();
```

```
    sqw = 0;
```

```
    delay();
```

```
    delay();
```

```
    delay();
```

```
}
```

```
}
```

```
void delay()
```

```
{ unsigned char i;
```

```
for(i=0; i<=x; i++)
```

```
{ TMOD = 0X01;
```

```
TLO = 0XD1;
```



dr tfo  
sjmp repeat  
end.

in C:

#include <reg51.h>

sbit exp = P3^4;

void main()

{ exp = L;  
while (1)

{ TMOD = 0x06;

TR0 = 1;

do

    PO = TLO;

    } while (TFO == 0);

    TR0 = 0;

    TFO = 0;

    }

}

#### \* Timers:

- Mode 0: 13 bit mode  $\rightarrow$  0001 1111 1111 1111 = 1FFFh = 8191

Ex: For delay (8191 - count = value to be stored).

- Mode 3: 8 bit split mode

#### \* To convert hexadecimal to decimal.

$$y = \left[ \left( (\text{number}/0x0a}) \ll 4 \right) \right] \mid (\text{number} \% 0x0a)$$

$$10h = 16$$

$$10/10 = 01 \text{ quotient}$$

$$10 \% 10 = 06 \text{ remainder}$$

$$01 \text{ swap } 10$$

$$10 + 06 = 16$$

\* 7 segment display:

1. Common anode

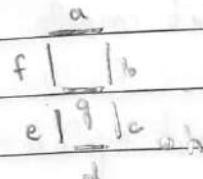
Input 0: ON

Input 1: OFF

2. Common cathode

Input 0: OFF

Input 1: ON



To interface a 7 segment display to any port of a microcontroller there are three ways:

- 1. Static mode
  - 2. Multiplexed mode
  - 3. Serial mode
  - 4. Direct mode
- parallel mode

Direct Mode:

\* In this mode we can display numbers from 0 to 9 and certain characters as well.

If we directly interface with display device to display any number or character, then we need to send its equivalent led code.

Ex: For 0 : n g f c d c b a

0 0 1 1 1 1 1 : For common cathode.

P0.0	a	1
P0.1	b	1
P0.2	c	1

8051

Disadvantage:

- It is required to send the equivalent code.

- Each display segment requires 8 pins i.e., 1 port. Hence only one display device can be interfaced at each port (i.e., 4 segments). This disadvantage can be overcome using display drivers.

Parallel Mode:Static Mode

In this mode a display driver 7447 IC which is 4:8 decoder

it converts BCD input to 4 segment code.

P0.0	7	a	1
P0.1	4	1	1
P0.2	4	1	1
P0.3	1	1	1

8051

Advantages

- Conversion to equivalent code is done by 7447.

- Number of I/O pins required is reduced by 4. Hence maximum of 8 display segments can be interfaced.

- Parallel transmission.

Disadvantage

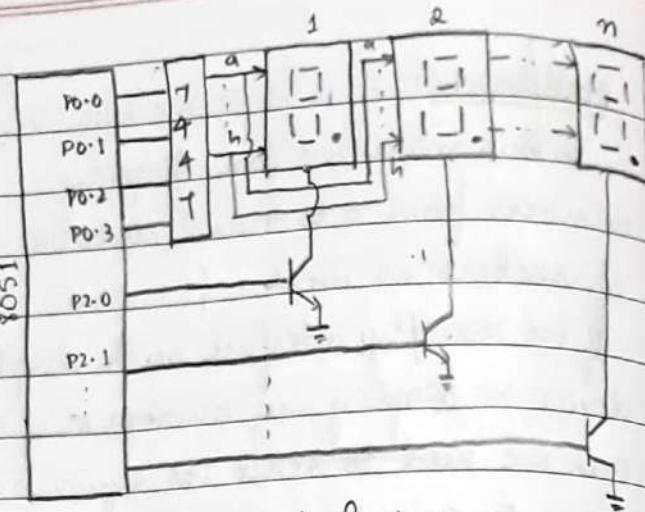
- Only numbers from 0 to 9 can be displayed.

- Hardware is increased hence costlier.

Multiplexed Mode

Using single display driver, we can drive multiple display devices. But in this case we cannot send all the data simultaneously. There are only two factors to be compromised: speed and number of I/O pins.

In this mode we can use only 4 pins for many segments of display but the speed is very slow because parallel transmission is not possible. The transistors are used to select the segment devices.



#### Advantages

- Each 7447 can be used for a maximum of 8 displays.

#### Disadvantages

- Cost is more and speed is low.

#### \* Serial Mode:

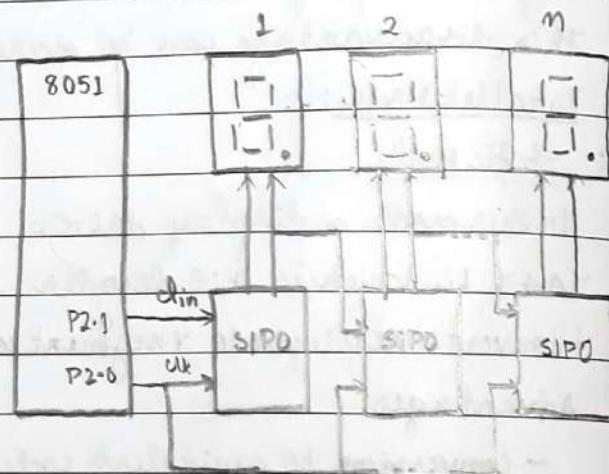
In this mode to interface n number of display devices we require only 2 I/O pins but it is comparatively very slow as SIPO shift registers are used.

Each display device requires a separate SIPO shift register.

For SIPO shift register we require 5 clock pulses.

In this mode we can view all the display only after  $5 \times n$  clock pulses where n is the number of display devices.

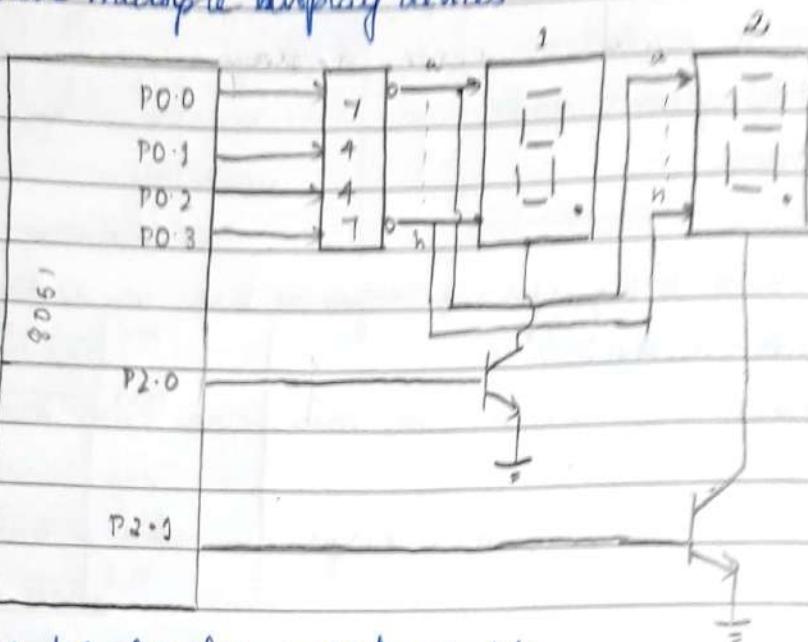
Since no display driver 7447 is used we will have to send the equivalent code for each display segment.





Multiplexed Parallel mode : 6 I/O pins for two segments.

single display driver is used  
to drive multiple display devices

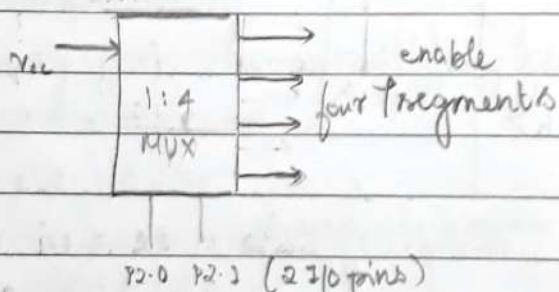


4 pins for decoder + 28 for enable

Hence 28, 7 segments can be interfaced by using transistors

4 pins for decoder + 28 for multiplexer (1:4).

Hence  $28 \times 2 = 56$ , 7 segments can be interfaced



```
#include <reg51.h>
```

```
bit fd = p2^0; // enable first display
bit sd = p2^1; // enable second display
void delay();
```

```
d
```

```
while(1)
```

```
d
```

```
fd=1; // select first display
```

```

PO = 0X01;
delay (500); // latch delay
fd = 0;
sd = 1; // select second display
PO = 0X02;
delay (500);
}
}

void delay (unsigned int)
{
    unsigned int i;
    for (i=0; i<=a; i++);
}

```

### iii. = Serial Mode

org 0000h

MOV P2, #8

MOV a, #A4h

repeat: RRC a

jc one

clr P2.0

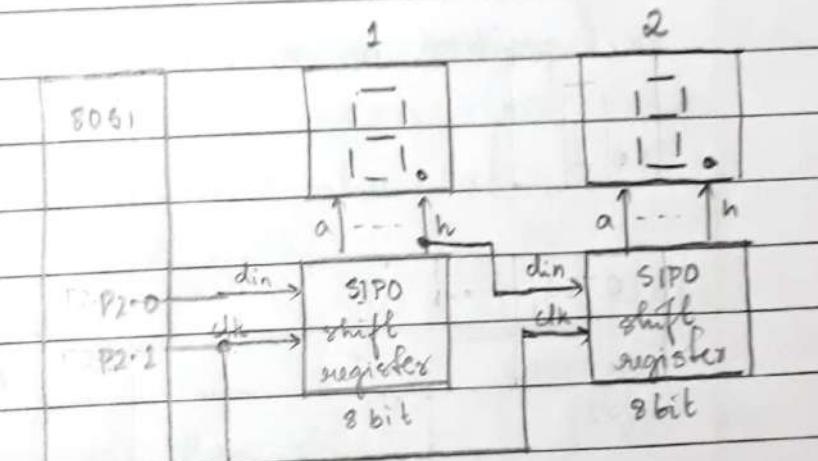
jmp nxt

one: srlb P2.0

nxt: srlb P2.1

clr P2.1

djnx r2, repeat



Data is sent bit wise

1: F9h = 1111 1001

2: A9h = 1010 0100

\* LCD: Alpha-numeric display device.

Advantages:

1. It requires less number of I/O lines
2. It can display any number and any character.
3. It consumes less power.
4. Brightness of the display can be controlled.
5. Left entry or right entry is possible.
6. Any number of characters can be displayed one after the other.

1 line module

1x8 : 8 characters

1x16: 16 characters

2 line module

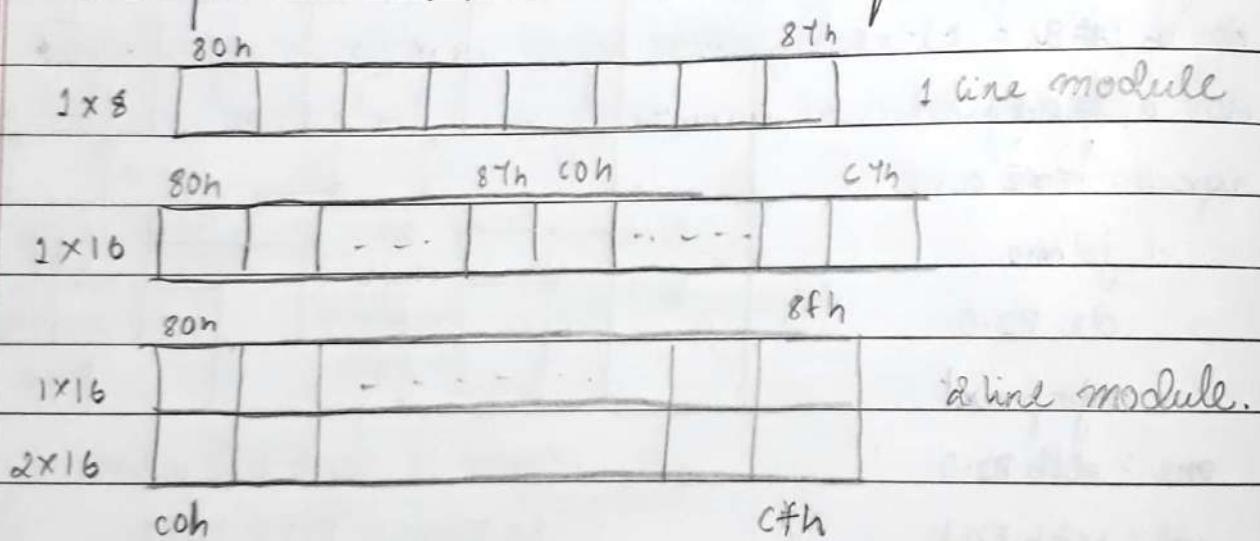
2x8: 8 characters / line

2x16: 16 characters / line

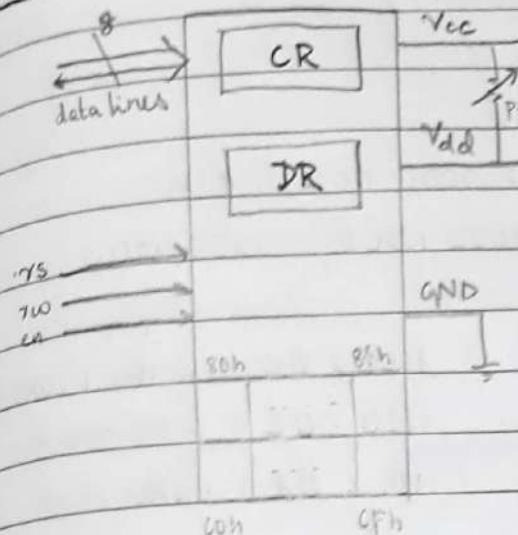
Each position on the LCD acts as a memory location.

Disadvantage: Any number or character to be displayed its equivalent ASCII is required.

Each position on the LCD has a unique address.



### - Architecture



→ It consists two registers

CR - command register } 8 bit

DR - Data/Display register } 8 bit

→ 8-bit data line, hence each character can be sent at a time

Do to Dr, where only Dr is bidirectional and others are unidirectional.  
acts as busy bit

→ It consists 3 control line

rs = 0 : CR } rs used to select

rs = 1 : DR } the register

rw = 0 : write } usually write

rw = 1 : read

en = 1 } to wait until  
delay (200 μsec) } we send the next  
data

en = 0

every module has certain  
batching time, i.e., time to display  
entire data, at that time busy bit is checked.  
only while checking the busy  
line / busy bit rw=1 : read.

By varying the potentiometer  
we can vary the brightness of  
the display.

\* For programming, there are three steps

→ LCD module initialisation

→ commands sending (8 bit only)

→ Data sending.

For initialisation of LCD module.

1. Function command : 001 dl n } X X : 001 1 1 0 00 : 38h

→ dl → data lines

If dl = 0 : 4 bit data lines

dl = 1 : 8 bit data lines

→ n → number of memory lines

If n = 0 : 1 line module

n = 1 : 2 line module

→ f → font size.

If  $f = 0$ : default ( $5 \times 7$ )

$f = 1$ :  $5 \times 10$

→ x x → dont care.

2. LCD clear command : 0000 0001 : 01h

3. Display command : 0000 1DCB : 0000 1100

→ D: display

If  $D = 0$ : off

1100: 0c : without cursor

$D = 1$ : on

1110: 0c : with cursor

→ C: cursor on

1101: 0d : with blink

If  $C = 0$ : off

$C = 1$ : on

→ B: cursor on with blink

If  $B = 0$ : without blink

$B = 1$ : with blink

4. Entry mode command: 0000 01 1/D S : 0000 01 1 0

→ I/D: Increment /Decrement

If  $I/D = 0$ : decrement

06: without shift

$I/D = 1$ : increment

07: with shift

→ S: Shift

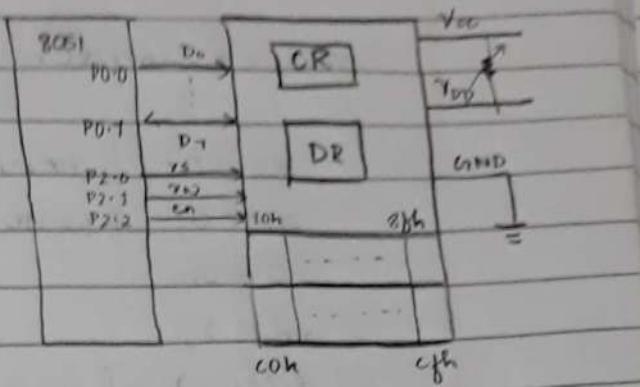
If  $S = 0$ : no shift

$S = 1$ : with shift

NOTE: To display a full message only 32 characters can be displayed at a time (16 characters /line).

To display n character message , it has to be scrolled.

code to display 'A':  
#include <reg51.h>  
sbit rs = P2^0;  
sbit rw = P2^1;  
sbit en = P2^2;  
void lcd\_init();  
void cmd\_send();  
void data\_send();  
void delay();  
void Main() {



(cd - init();

while (1)

1

`cw = 0x80; // First line  
2 x 16`

cw: command word

cmd - send();

~~dt = 'A'; // 'A' is to be displayed at data~~

~~data-send();~~

3

3

void Lcd\_init()

d

`CW = 0x38; // function command`

cmd\_send();

cmd\_send(),  
cw = 0x01; // LCD clear command

cmd->send();

`cw=0x0c;` // Display command

cmd\_send();

`cw = 0x06; // Entry mode command`

cmd - send ();

3

```
void cmd_send()
```

{

```
    rs = 0; // command register
```

```
    rw = 0; // write
```

```
    PO = CW;
```

```
    en = 1;
```

```
    delay(); // latching delay
```

```
    en = 0;
```

}

```
void data_send()
```

{

```
    rs = 1; // data register
```

```
    rw = 0; // write
```

```
    PO = DT;
```

```
    en = 1;
```

```
    delay(); // latching delay
```

```
    en = 0;
```

}

```
void delay()
```

{

```
    unsigned int i;
```

```
    for (i = 0; i <= 500; i++);
```

}

\* To display a message.

= #include <reg51.h>

```
sbit rs = P2^0;
```

```
sbit rw = P2^1;
```

```
sbit en = P2^2;
```

```
unsigned char msg[ ] = { "I am very happy" };
```

```
unsigned char msg2[ ] = { "Microcontrollers" };
```

```
void lcd_init();
```

```
void cmd_send(unsigned char);  
void data_send(unsigned char);  
void delay(unsigned int);  
unsigned char  
void main()  
{
```

```
    lcd_init();
```

```
    while(1)
```

```
{
```

```
    cmd_send(0x80); // starting address.
```

```
    for(x=0; x<=16; x++) // 16 characters.
```

```
{
```

```
        data_send(msg[x]);
```

```
}
```

```
} cmd_send(0xc0); // (microcontrollers)
```

```
for(p=0; p<=16; p++)
```

```
{ data_send(msg2[p]); }
```

```
void lcd_init()
```

```
{
```

```
    cmd_send(0x38);
```

```
    cmd_send(0x01);
```

```
    cmd_send(0x0c);
```

```
    cmd_send(0x06);
```

```
}
```

: scroll due to decrement

```
void cmd_send(unsigned char a)
```

```
{
```

```
    sw = 0;
```

```
    sw = 0;
```

```
    P0 = a;
```

```
    en = 1;
```

```
    delay(500);
```

```
    en = 0;
```

```
}
```

```
void data_send(unsigned char b)
```

```
{
```

```
    rs = 1;
```

```
    rw = 0;
```

```
    PO = b;
```

```
    en = 1;
```

```
    delay(500);
```

```
    en = 0;
```

```
}
```

```
void delay(unsigned int i)
```

```
<
```

```
unsigned int y;
```

```
for (y=0; y <= i; y++);
```

```
}
```

\* To implement counter

= #include <reg51.h>

```
sbit rs = P2^0;
```

```
sbit rw = P2^1;
```

```
sbit en = P2^2;
```

counting

CE CF

unsigned char msg[] = { "counting10" }; last two locations.

```
void lcd_init();
```

```
void cmd_send(unsigned char);
```

```
void data_send(unsigned char);
```

```
void delay(unsigned int);
```

```
void main()
```

```
{
```

```
    lcd_init();
```

```
&
```

```
    while (1)
```

```
{
```

```

cmd_send(0x80);
for (n=0; msg[n]!=0; n++)
{
    data_send(msg[n]);
}
cmd_send(0x6F);
for (a=0; a<=99; a++)
{
    cmd_send(0xcf);
    b = a/10;
    b = b | 0x30;
    data_send(b);
    cmd_send(0xce);
    b = a%10;
    b = b | 0x30;
    data_send(b);
}
}
}

```

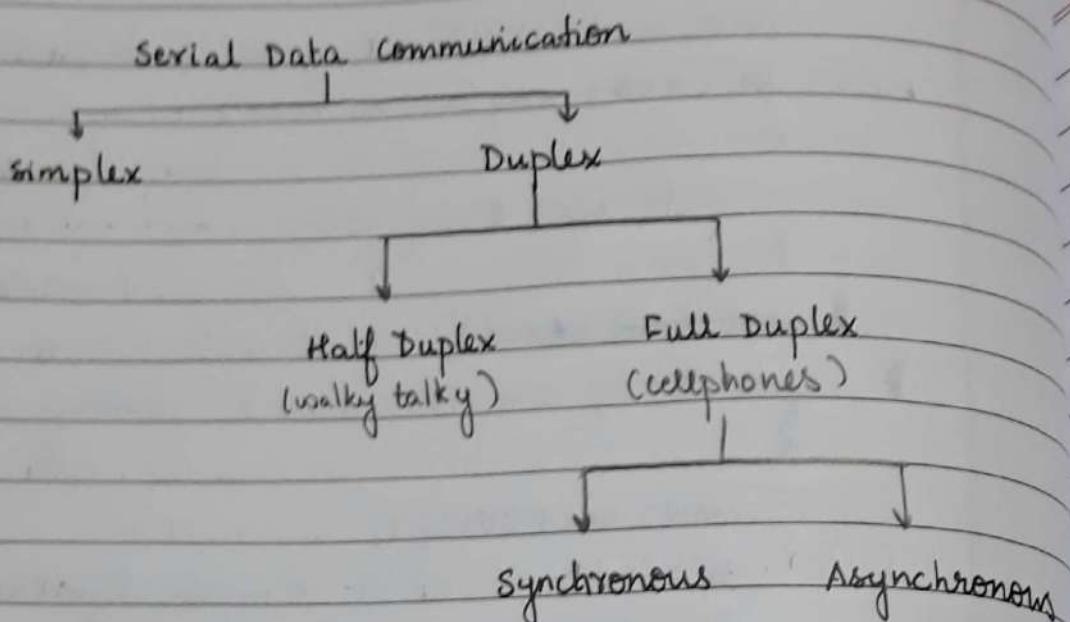
#### \* Data Communication:

##### Parallel Data communication

- All the bits are transferred at a time.
- Fast data communication
- Hardware is more
- cost increases as it is bulky
- Preferred for short distance communication.
- No modulation is required

##### Serial Data communication

- Each bit is transferred at a particular time.
- slow data communication
- Hardware is less
- cost is less as it is not bulky.
- Preferred for long distance communication.
- Modulation is required.

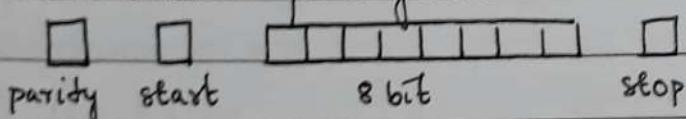


\* Protocol:

procedure or algorithm or rules agreed by both receiver and transmitter in data format.

## \* Framing | Deframing:

Packing of actual data along with start and stop bit with and without parity bit.



## \* Encryption | De-encryption:

Encryption: Encoding a message or information in such a way that only authorized parties can access it and those who are not authorized cannot.

Framing / Deframing } Taken care by inbuilt  
Encoder / Decoder } UART and USART  
Baudrate | bps      Universal Asynchronous Synchronous  
                        Receiver and Transmitter.

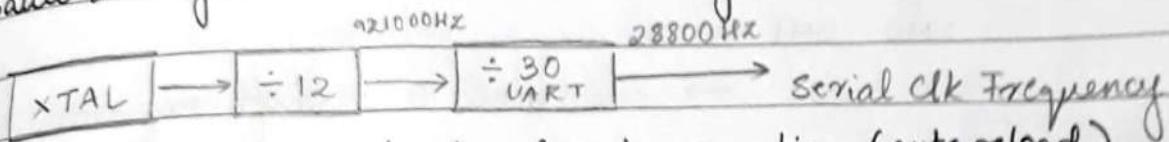
**UART**

- Data write is slow
- Stop | 8 bit | Start |  
10 bit →
- clock independent
- Data rate is variable

**USART**

- Data write is fast
- Single switching bit | 8 bit |  
9 bit ← →
- clock dependent.
- Data rate is constant

Baud rate generation is taken care by USART and timer.



Timer 1 and Timer 2 for baud rate generation (auto reload).

$$\text{Value of TH1} = \frac{\text{Serial clk frequency}}{\text{required baud rate}}$$

Ex:  $= \frac{28800}{1200} = 24$

2400 : F4h

1800 : FAh

9600 : EDh

$$256 - 24 = 232 = E8h$$

$$\text{Bit rate} = \frac{1}{\text{Baud Rate}} \quad (\text{time taken to transmit a high bit}).$$

Ex:  $\frac{1}{9600} = 0.904 \text{ ms}$

- other registers for serial data communication

SBUF : serial Buffer Register

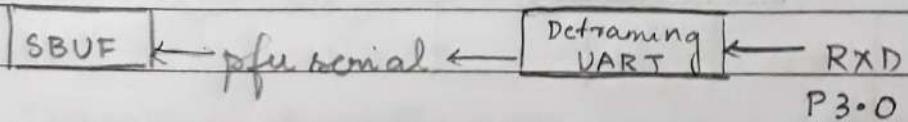
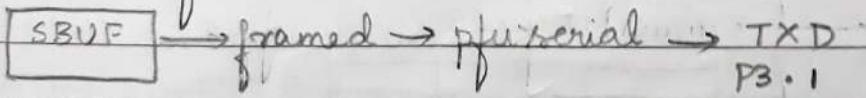
(should be sent in ASCII only)

- 8 bit register

Ex: SBUF = A.

- special function register

- user defined



## 2. SCON: Serial control Register

It is both bit and byte addressable and cannot be used as general purpose register.

SM0	SM1	SM2	RBN	TBS	RBS	TI	RI
b <sub>4</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	b <sub>0</sub>

These two modes are required to use serial port in different modes.

SM0 SM1

0 0 → serial mode 0 : shift register

0 1 → serial mode 1 : Inbuilt 8bit UART with variable Baud rate

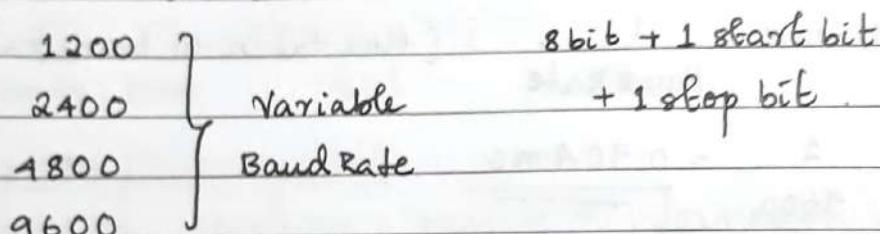
1 0 → serial mode 2 : 8bit UART with constant Baud rate

1 1 → serial mode 3 : Inbuilt 9bit UART with variable Baud Rate.

REN: Receiver enable : 1.

SM2 : Not useful for 8051 microcontroller : Hence 0.

In serial mode 1



TBS and RBS : 0

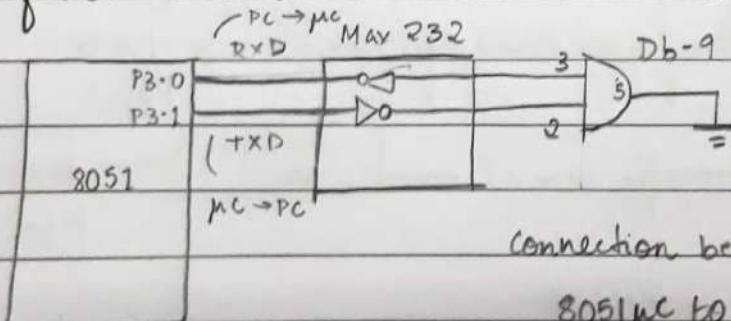
TI : Transmitter flag of serial interrupt Flags.

RI : Receiver flag

Initially both are zero

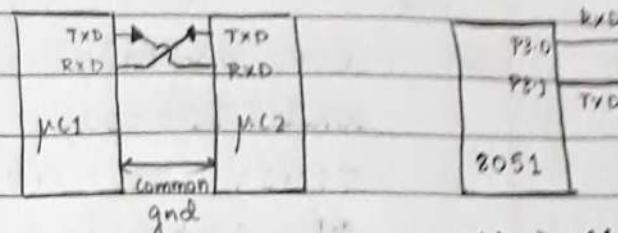
- if the SBUF data has been transmitted then TI = 1.

- if the SBUF receives data then RI = 1.



Transmission of data can be done from:

- μC to PC
- PC to μC
- μC1 to μC2
- μC1 to μC1



All the connections can be realised using RS232 cables.  $\mu C1 \leftrightarrow \mu C2$

To transmit data from microcontroller to PC.

\* = #include <reg51.h>  
void main()

value of TH2  
= serial clk frequency  
required baud rate

L  
TMOD = 0x20; // Timer 1 mode 2  
(Baud rate generation) =  $\frac{28800}{9600} = 3$   
TH1 = 0xfd;

TR1 = 1; // Start timer  
256 - 3 = 253 = FDh.

SCON = 0x50; // serial mode 1 with  
1 start bit and 1 stop bit

SBUF = 'A';

while (TI == 0);

T1 = 0;

while (1);

}

\* To send a message from microcontroller to PC

= #include <reg51.h>

unsigned msg[] = { "My note my choice \0" };  
void main()

a

TMOD = 0x20; // timer 1 in mode 2: for baud rate generation

TH1 = 0xfd; // 9600 baud rate

TR1 = 1; // start the timer for baud rate generation.

SCON = 0x50; // 1 stop bit and 1 start bit (serial mode 1)

```

for (n=0; msg[n] != '\0'; n++)
{
    SBUF = msg[n];
    while (TI == 0);
    TI = 0;
    delay(50000); // for observing character wise transmission
}
while (1);

```

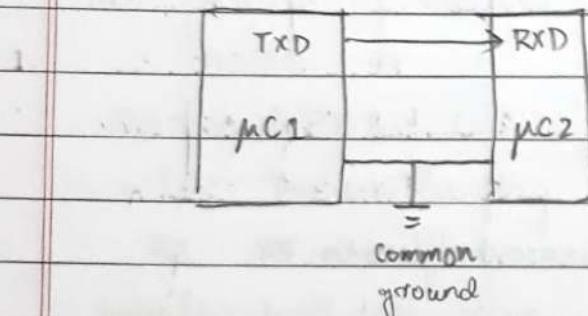
\* To transmit data from pc to microcontroller.

```

= #include <reg51.h>
while (RI == 0);
RI = 0;

```

\* To transmit data from one microcontroller to another



Two different codes for the both the microcontrollers is required.

\* To transmit data from a microcontroller to itself.

```

= #include <reg51.h>
unsigned msg[] = "....\0";
void main()
{

```

```

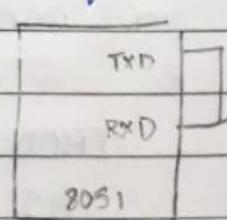
    TH1 = 0x20;

```

```

    TMOD = 0x20;

```



TRI = 1;

SCON = 0x50;

SBUF = 'A';

while (RI == 0);

TI = 0;

while (RI == 0);

x = SBUF;

PO = x;

RI = 0;

while (1);

}

### \* Interfacing:

#### 1. key pad interfacing

keys can be arranged

- linearly / non-matrix form

- matrix form ( $n \times n$ )

#### Applications

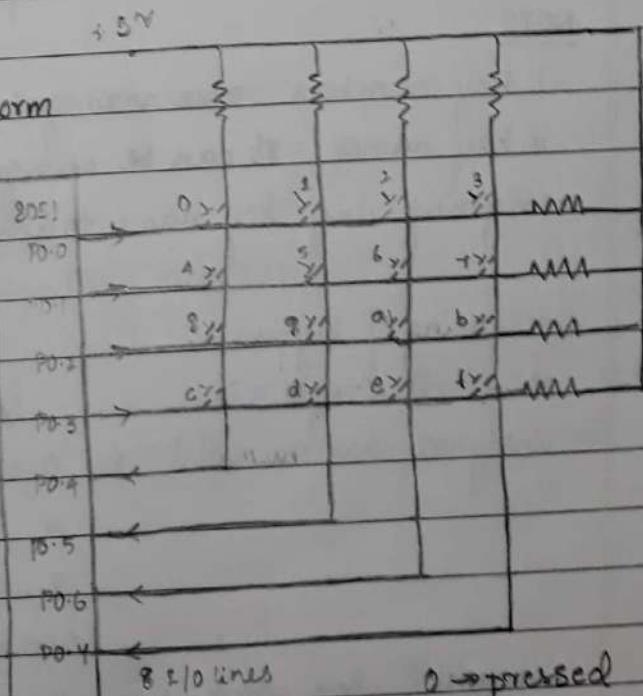
- key presscol  
number can be  
found out and can  
be displayed

- realise as calculator  
 $(+, -, \div, \times)$

It can be done by

2. Row scanning

2. Line reversal technique.



0 → pressed

1 → not pressed

0: enable

1: disable

### \* Row scanning method

1. Configure input and output pins connected to rows as DATA pins and input /output pins connected to columns as input pins.
2. Enable all the rows (0) to check if key is pressed.
3. Wait for any key to be pressed.
4. Eliminate key-bouncing (key debouncing) by waiting for 20m sec. It can be realized by
  - Hardware technique : use RS latch (for each key but this makes it bulky)
  - Software technique : Each key usually takes 20m sec for a solid contact. hence we wait or give a delay for some time. This technique is usually preferred.
5. Find particular row by enabling each row at a time.
6. Find the column and then display.

#### NOTE:

- i. key numbers : once arranged cannot be changed
- ii. key names : It can be changed based on user by assigning any key name to any key number

### \* Interfacing of keypad.

= #include <reg51.h>

```
unsigned char nums[] = { 0, 1, 2, 3,
                        4, 5, 6, 7,
                        8, 9, oxaa, oxab,
                        oxac, oxad, oxae, oxaf };
```

unsigned char check\_key, row, key\_no;

void main()

<

P2 = 0XF0; // to enable all four rows at a time

{ do

{ 1

check\_key = P2; // to read the switch status.

check\_key = check\_key &amp; 0xFO; // to check which key is pressed

3 while (check\_key == 0xFO);

delay (500); // to eliminate key-bouncing

key\_no = 0;

for (row = 0; row &lt; 4; row++)

{ 2

P2 = ~ (0x01 &lt;&lt; row);

0000 0001

check\_key = P2 &amp; 0xFO;

1111 1110 - row 0 is enabled

if (check\_key != 0xFO)

0000 1000

{ 3

1111 0111 - row 4 is enabled

switch (check\_key);

{ 4

case 0x00:

PO = nums [key\_no + 0];

break;

case 0x10:

PO = nums [key\_no + 1];

break;

case 0x20:

PO = nums [key\_no + 2];

break;

case 0x30:

PO = nums [key\_no + 3];

break;

}

{ 5

key\_no = key\_no + 4;

{ 6

\* Calculator

1. read the 1<sup>st</sup> number and store
2. read the 2<sup>nd</sup> number and store
3. read the operator and then check
  - if (+): perform addition
  - if (-): perform subtraction
  - if (x): perform multiplication
  - if (÷): perform division.

2. Motor Interfacing:

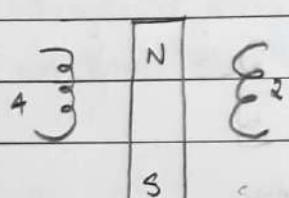
Motor is required to move any stationary objects either in clockwise or anti-clockwise direction in a required speed.

Motor has two parts

- moving part: rotor / conducting material
- stationary part: stator / winding / coil

Size of the motor determines the power handling capacity and amount of power required for moving. Size depends on winding.

1. Stepper motor:



Any stepper motor consists of four coils and a permanent magnet. Hence it is also called permanent magnet motor.

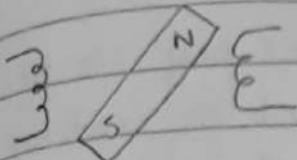
In order to initiate motion we need to excite the coil.



a. single coil Excitation | Full step

Exciting each coil at a time. When a coil is excited the magnet will align with the coil facing north. Step size is  $1.8^\circ$ . Excitation should be done in sequence only.

(b)(ii)



## b. Double coil excitation / Half step

Two coils are excited at a time

Here the step size is  $0.9^\circ$ .

$$\text{Number of steps} = \frac{360^\circ}{\text{step size}}$$

(iii)

For single coil excitation

$$= \frac{360^\circ}{1.8^\circ} = 200 \text{ steps}$$

For double coil excitation

$$= \frac{360^\circ}{0.9^\circ} = 400 \text{ steps.}$$

By changing the order of excitation we can control the direction of motion and by changing the time interval in exciting two coils we can control the speed of rotation.

Stepper motor

1. normal winding: the coils are connected in a sequence hence has a general excitation codes.

2. random winding: the coils are connected randomly hence they have their own excitation codes.

For ALP:

The code usually requires 4 bits since ~~it is~~ a 8 bit register is used, the code is 8 bit.

		virtual	physical
88	1000 1000		
44	0100 0100		
22	0010 0010		
11	0001 0001		

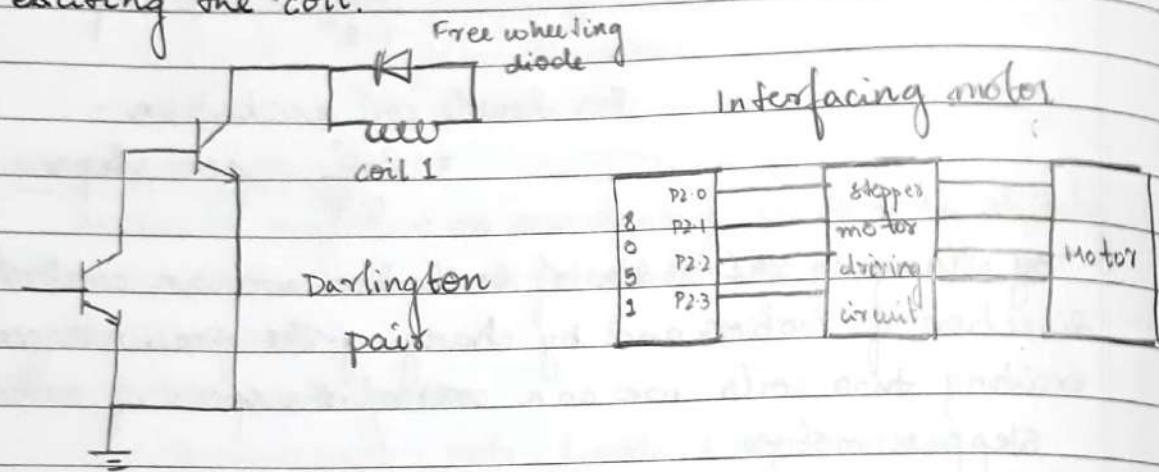
} single coil Excitation code

cc	1100 1100	} Double coil Excitation code
66	0110 0110	
33	0011 0011	
99	1001 1001	

For 100 steps we need to excite 4 coils 25 times

$$4 \times 25 = 100$$

The current / voltage available at any I/O pin of any microcontroller is not sufficient for exciting any coil of the motor so we require a stepper driver circuit so for each coil a separate darlington pair for exciting the coil.



- \* To rotate in clockwise direction continuously
- = #include <reg51.h> #include <reg51.h>
- void stepay (unsigned int j) void delay(unsigned int j)
- < <
- unsigned int i;      unsigned int i;
- for (i=0; i <=j; i++);      for (i=0; i <=j; i++)
- } };
- void main () void main ()
- < <
- while (1)      n = 0x01;
- {      while (1)
- PO = 0x88;      }
- delay (500);
- PO = 0x44;
- delay (500);
- }
- PO = n

$PO = 0x22;$

$\text{delay}(500);$

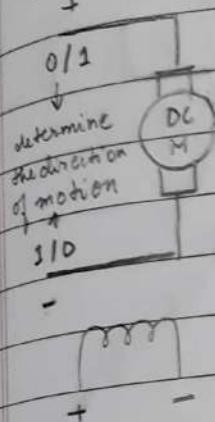
$PO = 0x11;$

$\text{delay}(500);$

$\text{delay}(500);$

$PO = n \ll 1;$

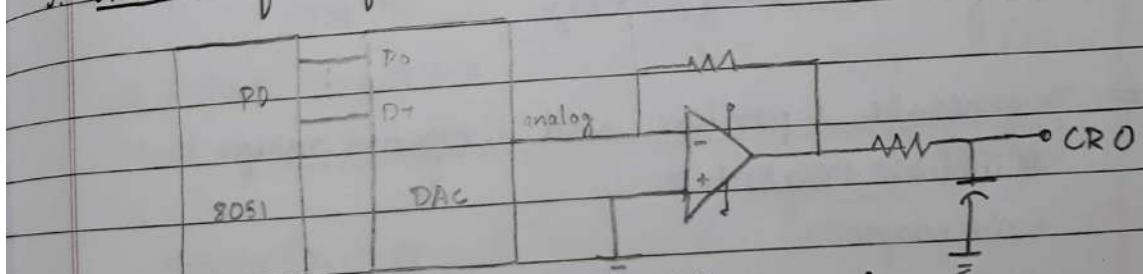
### 1. DC Motor:



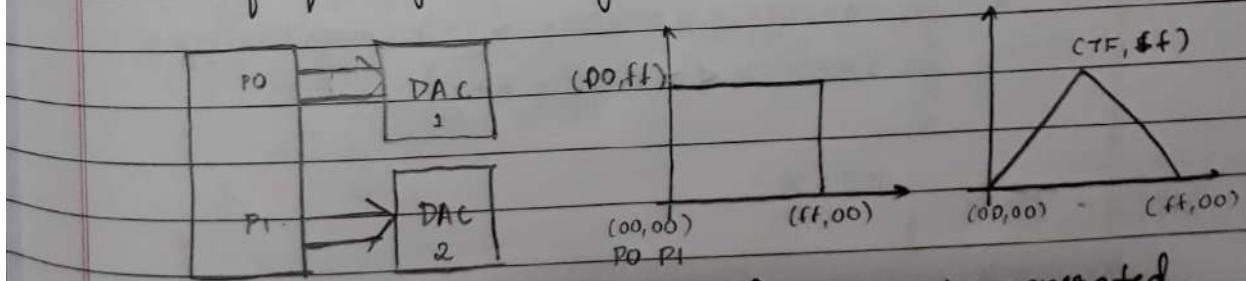
It has only one coil. The speed of a DC motor is controlled by PWM (Pulse Width Modulation).

By varying the average power provided to the power terminal we can control the speed of motor by PWM.

### 3. DAC Interfacing:



Any waveform of variable amplitude and variable frequency can be generated.



With each port as one axis certain patterns can be generated.

- \* Develop the embedded - C code to generate the following waveforms by interfacing DAC to a controller with variable amplitude and variable frequency.

$$5V \rightarrow f_{fh} = 255$$

$$1V \rightarrow \frac{255}{5} = 51 = 33hw$$

= To generate square wave :  $A = 5V$

```
#include <reg51.h>
```

```
void main()
```

```
{ while(1)
```

```
    PO = 0x00; // On : high
```

```
    delay(500);
```

```
    PO = 0xFF; // Off : low
```

```
    delay(500); // 0x99; 3V amplitude
```

```
}
```

```
void delay(unsigned int j)
```

```
{
```

```
    unsigned int i;
```

```
    for (i=0; i<=j; i++)
```

```
}
```

= To generate i. positive and ii. negative ramp:

```
#include <reg51.h>
```

```
void main()
```

```
{
```

```
    while(1)
```

```
        for (n=0x00; n>=0xFF; n--) : negative ramp
```

```
        for (n=0; n<=0xFF; n++)
```

```
{
```

```
    PO = n;
```

```
}
```

```
}
```

= To generate triangular waveform:

```
#include <reg51.h>
```

```
void main()
```

```
{ unsigned int x, y;
```

```
while(1)
```

```
{
```

```
for (x=0x00; x<=0xff; x++)
```

```
{
```

```
PO = x;
```

```
}
```

```
* for (y=0xff; y>=0x00; y--)
```

```
{
```

```
PO = y;
```

```
}
```

```
y
```

```
y
```

= To generate step (positive staircase waveform)

$$\text{step size} = \frac{255}{\text{no of step}} = \frac{255}{5} = 33h$$

```
#include <reg51.h>
```

```
void main()
```

```
{
```

```
while(1)
```

```
{
```

```
for (x=0xff; x>=0x00; n=x-0x33)
```

```
for (x=0; x<=0xff; x=x+0x33)
```

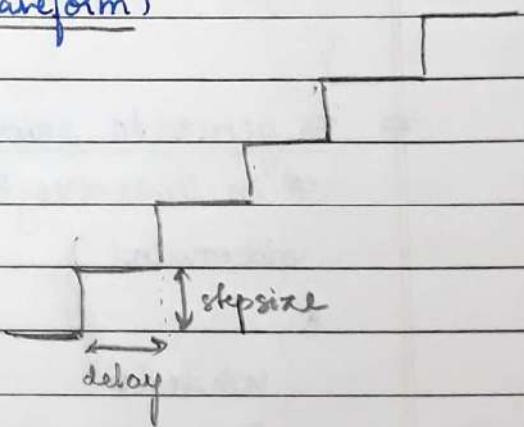
```
{
```

```
PO = x;
```

```
delay();
```

```
y
```

```
y
```



= To generate combination of steps:

```
#include <reg51.h>
```

```
void main()
```

```
{
```

```
    while(1)
```

```
{
```

```
    PO = 0x00;
```

```
    delay(50ms);
```

```
    PO = 0x66;
```

```
    delay(50ms);
```

```
    PO = 0x99;
```

```
    delay(50ms);
```

```
    PO = 0x66;
```

```
    delay(50ms);
```

```
    PO = 0x00;
```

```
    delay(50ms);
```

```
}
```

```
}
```

= To generate sawtooth waveform:

```
#include <reg51.h>
```

```
void main()
```

```
{
```

```
    while(1)
```

```
{
```

```
    for(x=0; x<=0xff; x++)
```

```
}
```

```
    PO = x;
```

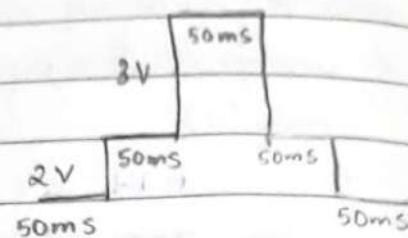
```
}
```

```
    PD = 0x00;
```

```
    delay();
```

```
}
```

```
}
```



= To generate sine wave:

Sine wave can be generated by

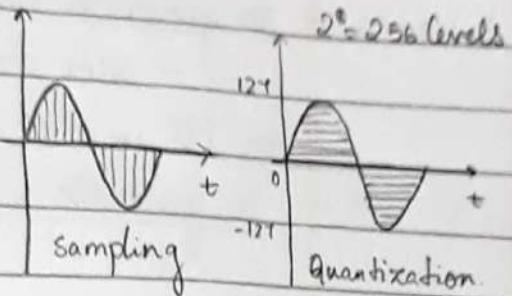
- i. Sampling and
- ii. Quantization.

$$V_{out} = 127 + 127 \sin \theta$$

$\theta$  varies from 0 to  $360^\circ$ .

varying  $\theta$  by an interval of  $5^\circ$ .

$$\frac{360^\circ}{5^\circ} = 72 \text{ values}$$



For 8 bits.

$2^8 = 256$  levels

```
#include <reg51.h>
```

```
unsigned char nums[72] = { "....." };
```

```
void main()
```

```
{
```

```
    while(1)
```

```
{
```

```
    for (n=0; n<=72; n++)
```

```
{
```

```
    PO = nums[n];
```

```
}
```

```
}
```

```
#include <reg51.h>
```

```
#include <math.h>
```

```
void main()
```

```
{
```

```
    while(1)
```

```
{
```

```
    for (n=0; n<=72; n++)
```

```
{
```

$$y = 127 + 127 \sin \left( \frac{n * 22}{180} \right);$$

```
    PO = y; }
```

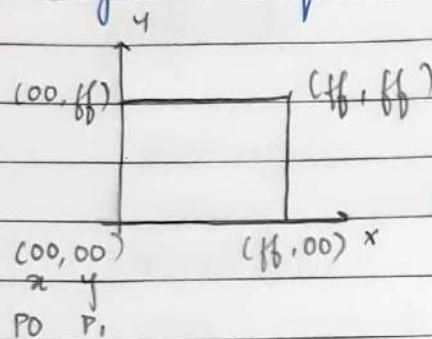
When the values are  
calculated manually

When the values are not  
calculated manually

- To generate circle

- sine wave : send the positive values ( $0$  to  $180^\circ$ ) in P0 and the negative values ( $180^\circ$  to  $360^\circ$ ) in P1.

- To generate square



org 0000h

MOV a, #00

MOV b, #00

cnt: MOV P0, a

MOV P1, b

inc b

cjne b, #00, cnt

cnt1: MOV a, #00

MOV b, #0ffh

MOV P0, a

MOV P1, b

inc a

cjne a, #00, cnt 1

cnt2: MOV a, #0ffh

MOV b, #0ffh

MOV P0, a

MOV P1, b

dec b

cjne b, #0ffh, cnt 2

cnt3: MOV a, #0ffh

MOV b, #00

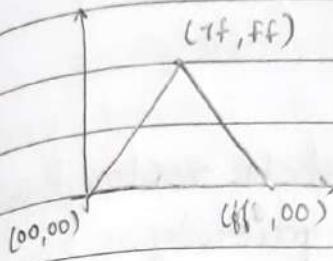
MOV P0, a

MOV P1, b

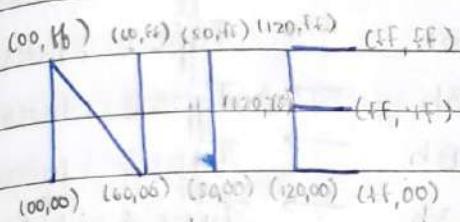
dec a

cjne a, #0ffh, cnt 3

- To generate triangle



- To generate NIE



### \* INTERRUPTS:

ISR: Interrupt Service Routine.

Interrupt can be by

i. Hardware : External interrupts.

ii. software : Internal interrupts.

Markable Interrupts : These interrupts can be ignored

Non-markable Interrupts : These interrupts has to be served.

Vectored Interrupts : Predefined address.

Nonvectored Interrupts : User defined address. ISR address can be defined by the user where the interrupt is written.

### \* 8051 Interrupts:

5 interrupts + reset

(2 external interrupts + 3 internal interrupts)

- P3.2: INT0 : ext 0 By default : level trigger (upon reset)

- P3.3: INT1 : ext 1 Through program it can be made edge triggered.

## Internal interrupts

- Timer 0 (T0)
- Timer 1 (T1)
- Serial interrupt (TI and RI)

All the 5 interrupts are non maskable vectored interrupt.  
 Vector address is based on the priority.  
 When there are multiple interrupts, the interrupts are served based on the default priority.

Interrupt structures

1. INT0 → 0003h → Type 0 : Interrupt 0
2. T0 → 000Bh → Type 1 : Interrupt 1
3. INT1 → 0013h → Type 2 : Interrupt 2
4. T1 → 001Bh → Type 3 : Interrupt 3
5. TI and RI → 0023h → Type 4 : Interrupt 4

If the ISR is greater than 8 bytes, it will overlap with ISR of the next interrupt. This can be overcome by starting ISR from the corresponding address and then jump to other memory location and write the interrupt routine.

## Programming of Interrupts.

### TCON

tcon.7 tcon.6 tcon.5 tcon.4 tcon.3 tcon.2 tcon.1 tcon.0

TF1	TRI	TF0	TR0	IE1	IT1	IE0	IT0
-----	-----	-----	-----	-----	-----	-----	-----

By default external interrupts are level triggered, it can be made edge triggered by making IT1 and IT0 as 1 to make interrupt 0 and interrupt 1 respectively as edge triggered.

IT1 and IT0 : 0: level triggered

1: edge triggered

IE: Interrupt enable register

EA	-	ET2	ES	ET1	EX1	ETO	EX0
----	---	-----	----	-----	-----	-----	-----

EA : enable

1 : enable (to wait for interrupts)

0 : disable (to avoid interrupts)

ET2 : timer-2 : 0

ES : Enable serial Interrupt

1 : enable

0 : disable

ET1 : timer-1 interrupt

1 : enable

0 : disable

EX1 : External interrupt 1

1 : enable

0 : disable

ETO : timer-0 interrupt

1 : enable

0 : disable

EX0 : external interrupt 0

1 : enable

0 : disable

- IP : Interrupt Priority Register

using this register we can change the default order of priority of interrupts.

-	-	PT2	PS	PT1	PXI	PT0	PX0
---	---	-----	----	-----	-----	-----	-----

PT2 : timer 2 : 0

Here 0 - not assign      when assigned 1 , then that interrupt  
1 - assign      becomes first and comes on top .

PS : Priority for serial interrupts .

PT1 : Priority for timer-1 interrupt

PXI : Priority for external interrupt 1 .

PT0 : Priority for timer-0 interrupt

PX0 : Priority for external interrupt 0 .

When more than one interrupt is assigned 1, then within them the default priority is considered which is then followed by remaining interrupts.

- \* Write an 8051 ALP to blink an LED at P1.3 whenever there is an interrupt through P3.2.

= P3.2 : External interrupt 0

vector address = 0003h

Org 0003h

~~jmp ext0-isr~~

~~jmp ext0-isr~~

Org 0000h

MOV IE, #81h

wait : sjmp wait // wait without doing any work

07: wait by doing other work

Org 0300h ext0-isr : setb P1.3

acall delay

clr P1.3

acall delay

reti // return from interrupt

Or: to blink led n times.

Org 0300h

ext0-isr : MOV R2, #5

up: setb P1.3

acall delay

clr P1.3

acall delay

djnz R2, up

reti

delay : MOV R2, # ~~0ff~~ 10

up2: MOV R3, # 0ffh

IE							
EA	-	ET2	ES	ET1	EX1	ET0	EX0
		1	0	0	0	0	0

81h

Org 0000h

MOV IE, #81h

MOV A, #0

cnt : MOV PD, A

inc a

gje a, #0ah, cnt

up1: MOV R4, #0ffh

up: djnz R4, up

djnz R3, up1

djnz R2, up2

ret.

= #include <reg51.h>

sbit led = P1^3;

sbit ex0 = P3^2;

void ext-int (void) interrupt 0 { type - short }

}

for (x=0; x<50; x++)

}

led = 1;

delay();

led = 0;

delay();

}

}

void main()

{ while(1)

  IE = 0x81; // enabling EX0.

  white(); // wait without work

}

}

NOTE:

It can be converted to edge trigger.

TCON = 0x02.

wait with work

void main()

{

  while(1)

  {

    IE = 0x81;

    for (y=0; y<10; y++)

  }

}

  PO = y;

  delay();

  }

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}