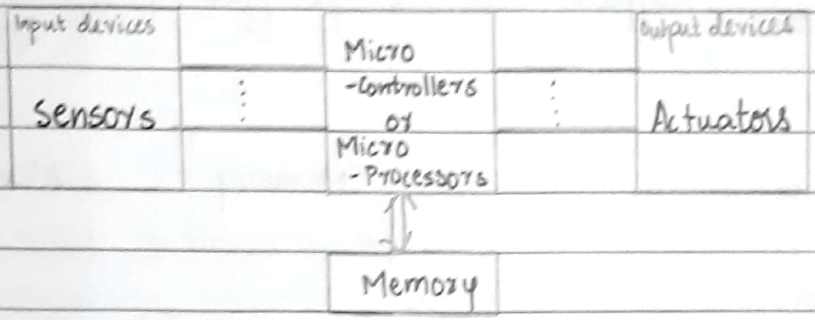
Introduction

★ 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.

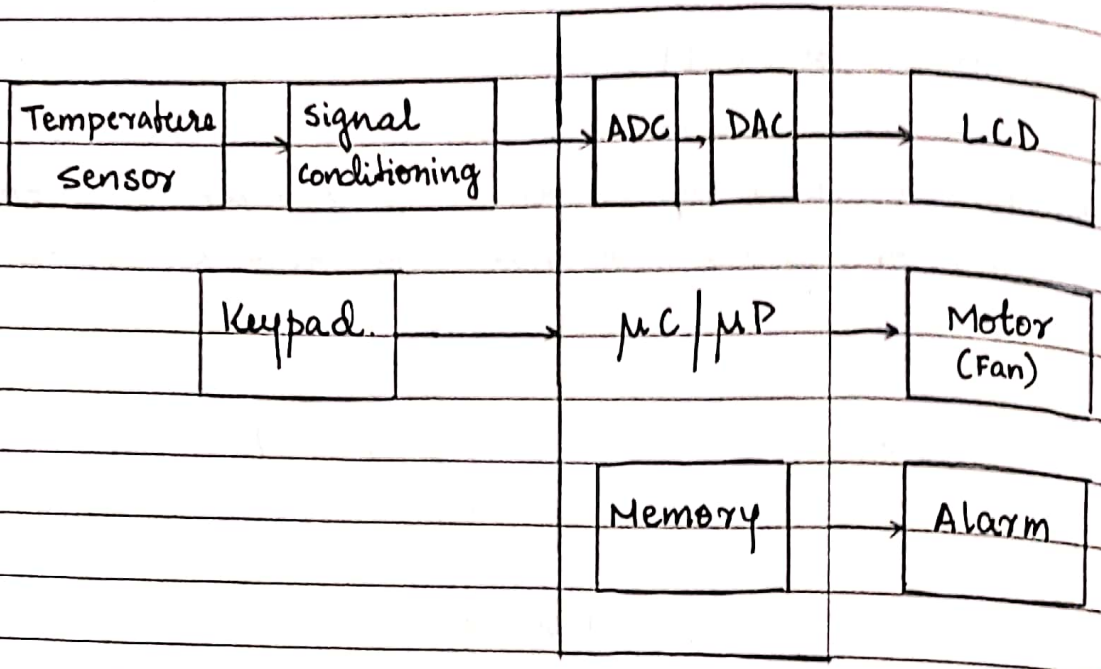
Input and output devices are interfaced to microcontroller or microprocessor through input and output interfacing devices as the speed of input or output device and also the operation of input and output devices donot match with that of microcontroller/microprocessor. Only one device is attended at a 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):

C

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.

Keil: 8051 microcontrollers

Eclipse: ARM microcontrollers

MPLab: PIC microcontrollers

AVR studio: AVR microcontrollers.

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

Important characteristics of embedded systems

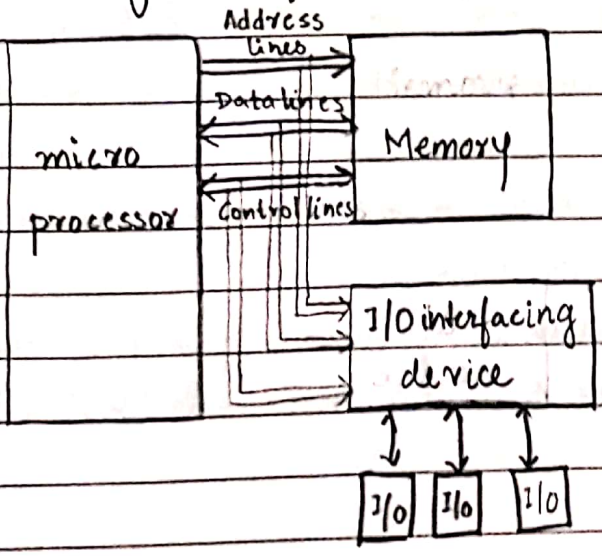
- Affordability
- Portability
- Battery driven
- Response time (high speed)

Unit-1

MICROPROCESSORS AND MICROCONTROLLERS

*	Microprocessors	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.

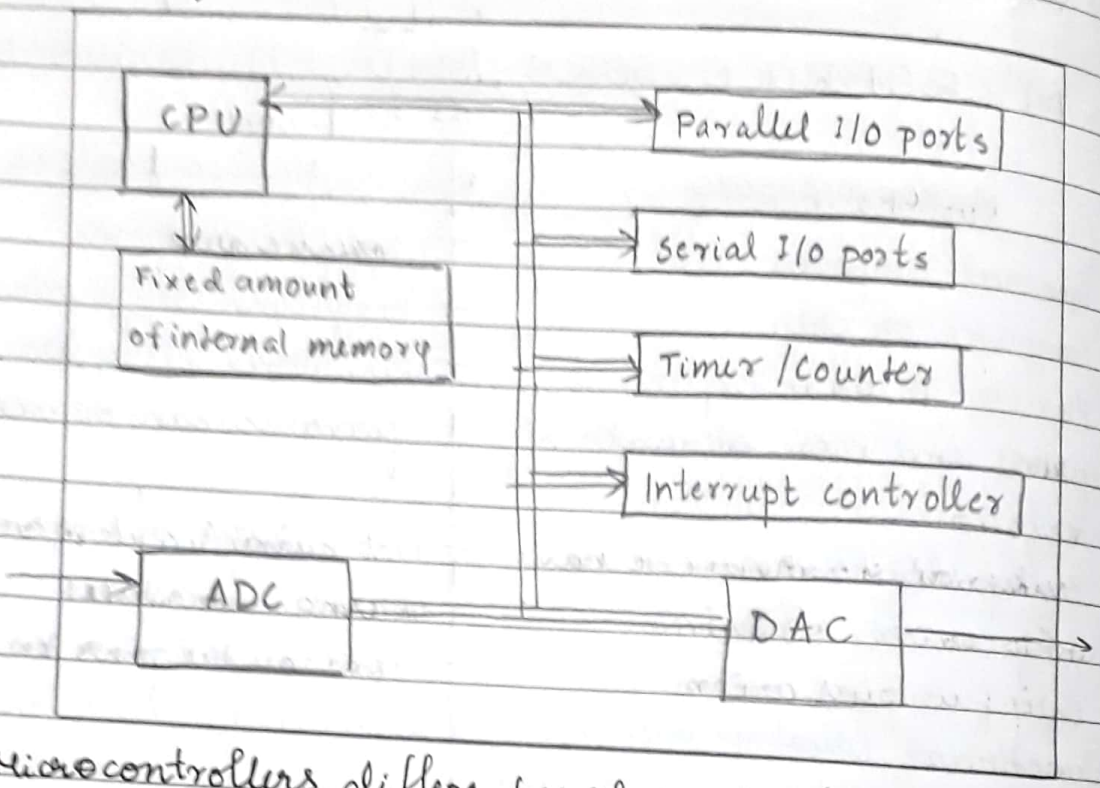
Address lines determines the memory locations.

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 differs based on, on chip resources and internal memory.

Based on the organisation of the memory.

Van Neumar

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

Hence there is a chance of overlapping of data with code we should make sure we dont let them overlap.

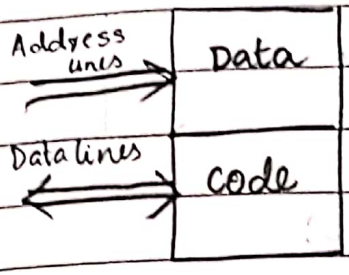
It is not possible to acess data and code simultaneously

Hardvard

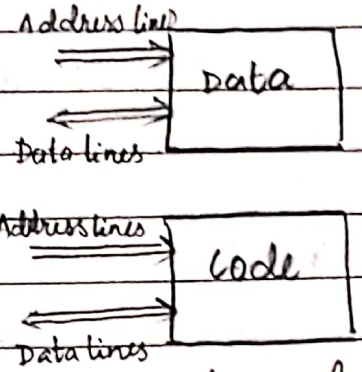
Data and code have their own respective set of address bus and data bus.

Hence there is no chance of overlapping of data with code

It is possible to access data and code simultaneously



Van Neumar



Hardvard

CISC	RISC
- More number of instructions	- Less number of instructions
- Hardware for Software Dedicated hardware is required for a complex operation. Ex: multiplication	- Software for hardware Software is developed for the basic operation. Ex: multiplication by repeative addition
- Expensive	- Cheaper
- complex circuitary	- Simple circuitary
- Hardware pipelining	- Software pipelining
- Less internal registers.	- More internal registers
- Instruction format is variable and each instruction is executed in different cycles.	- Instruction format is usually fixed and instructions are executed in single clock cycle.

Microcontrollers belong to certain families. They are not compatiable with each other as they have different set of instructions of different languages. There are many members with in 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 8051µc.
With in the family all members are code compatible.
Families: 8051, Motorola, PIC, AVR, ARM

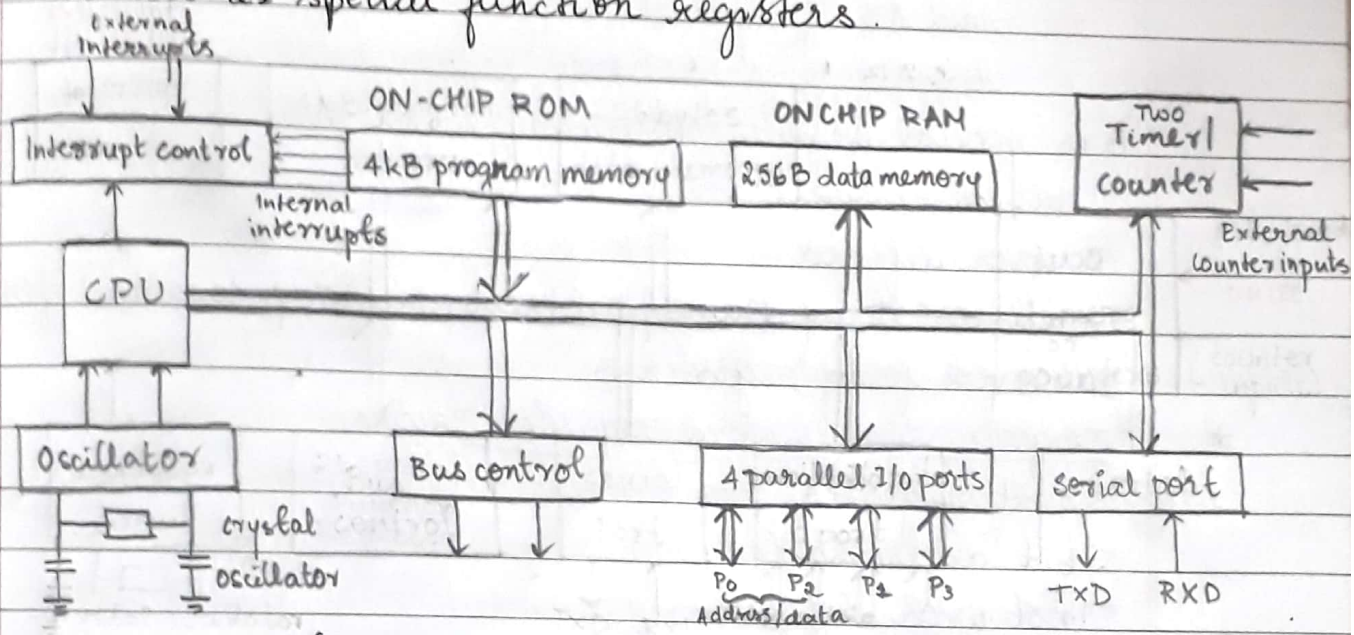
8051 Microcontroller

★ 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} = 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 6, 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 interfa
   2. I/O ports can be used as address lines as well.
8. Hardward 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.

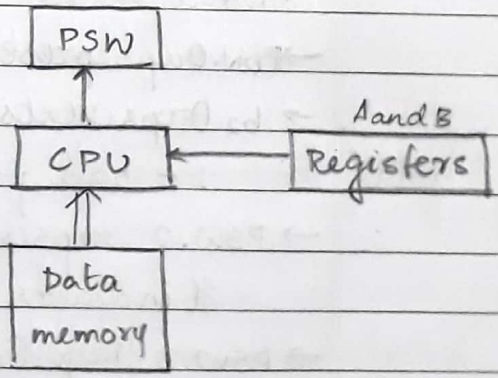
16. It has 21 special function registers.



The ALU performs operation on 8 bit of data. It recieves 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.
ie. MOV A, 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 E0H 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 F0H 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
CY	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, r_2 here by default r_2 belongs to bank 0.

RS1	RS0	
0	0	: Bank0
0	1	: Bank1
1	0	: Bank2
1	1	: Bank3.

For overflow:

8 bit range ⇒ 00h to ffh

It is divided into half

00h to 7Fh and 80h to FFh

in binary MSB = 0 in binary MSB = 1

∴ OV is 0 ∴ OV is 1

if in this range. if in this range.

→ PSW.6 : Auxillary carry flag

Carry from lower nibble to higher nibble: auxiliary carry

This flag is used by instructions that perform BCD arithmetic.

→ PSW.7: Carry flag : indicates out of width of the register.

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

→ 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 power)

* 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 (82h).

* MEMORY ORGANISATION OF 8051 WITH MEMORY MAPPING

Memory Organisation

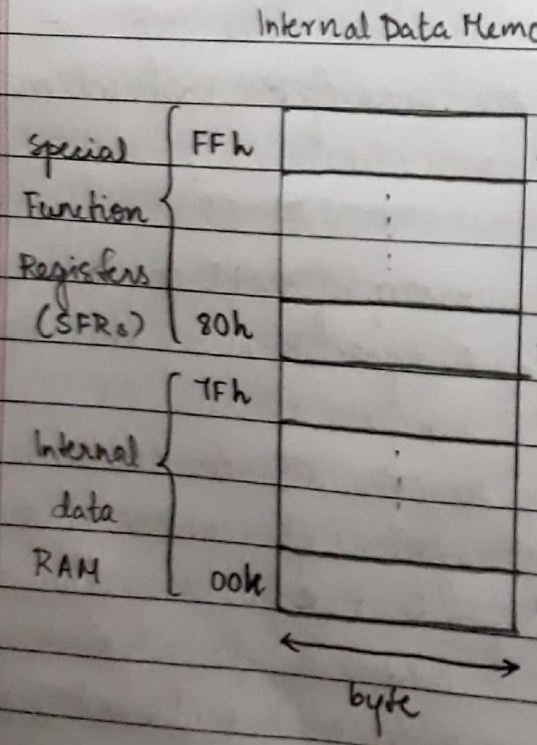
- Internal Memory
  - Internal data Memory (256 bytes)
  - Internal Program Memory 4k bytes
- External Memory
  - External Data Memory 64k bytes
  - External Program Memory (60kB / 64kB)

$\overline{PA}$: Program access (active low)

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

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

* Internal Data Memory

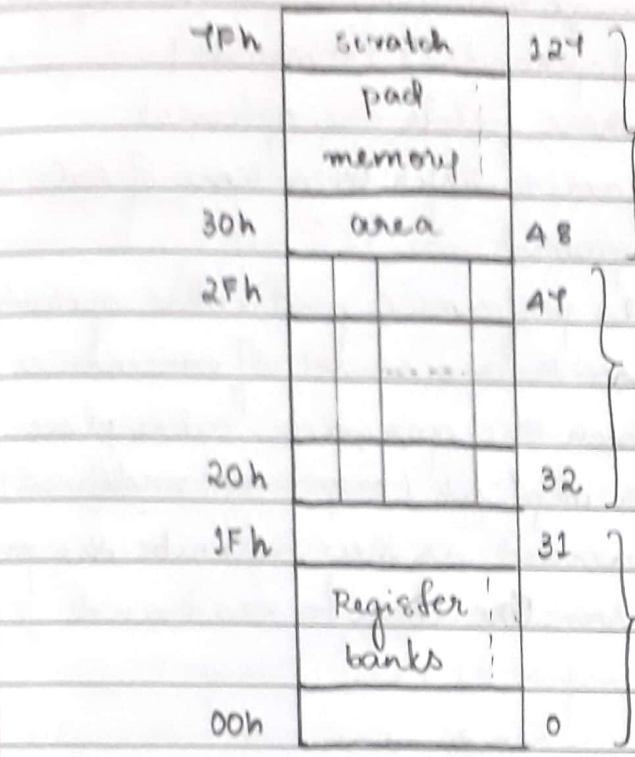


The internal data memory is segmented into two equal parts.

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

00h - 7Fh : Internal Data RAM.

Internal Data RAM

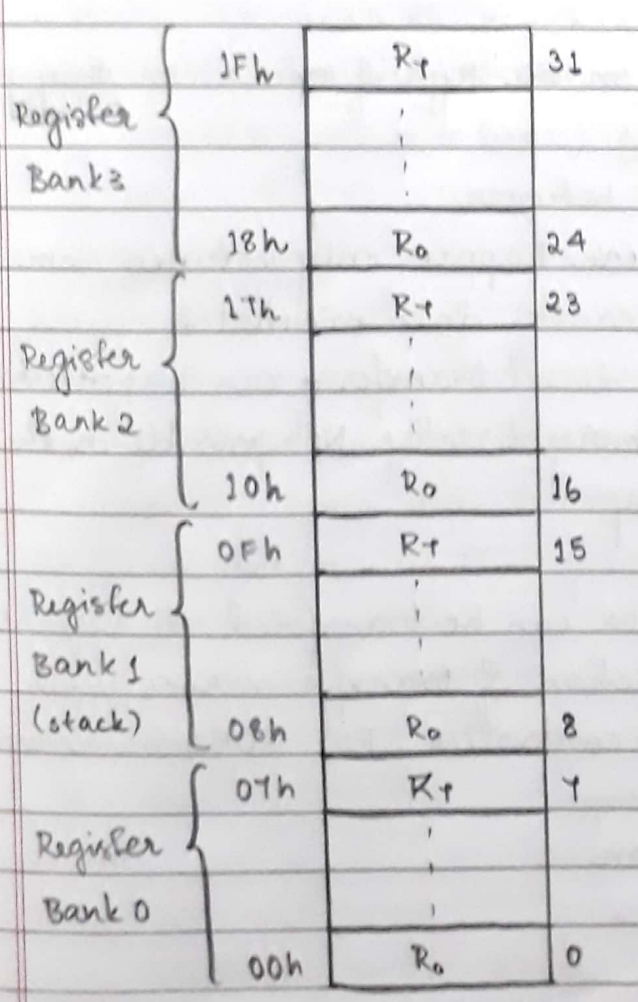


The internal data RAM is further segmented into three parts:

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 banks

The register banks are further divided into 4 banks of registers with 8 registers each (R_0 - R_7).

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

08h - 0Fh - 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 - 17h - 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.

* Structure of Assembly language:

An assembly level language instruction consists of four fields

[label:] mnemonics [operands]; [comments]

bracket indicates that these 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/opcodes

operands: data on which the operation takes place. There 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 with in the internal memory directly. Not possible in the case of external memory.
- register to memory:

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

MOV op1, op2

where op1: destination

and op2: source.

Syntax: MOV destination, source

destination cannever be data. It can be address or register where as 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 ⇒ here the controller is alive and keeps on performing this function. It is used to display any data for a long time.

address width

→ Org 0000h

→ MOV a, #12h appropriate data to appropriate register (i.e., same size): load operation

8bit 8bit

→ MOV b, #12 12 = Ch (stores as C as mentioned in decimal)

→ MOV r2, # 0001 0010b

→ MOV r_3, #'3' hexadecimal value of ASCII value is stored. Here 3 in ASCII is 33 in here.
(ASCII value)
(because for 0 to 9 : 30h is added and for A to F : 37h is added)
Here 33h is stored in r_3.

→ MOV a, #fdh : Syntax error
Any MSB greater than 9 should be preceded with zero.
∴ MOV a, #0fdh.

→ MOV dptr, #1234h
(16 bit) (16 bit)
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 subprogram is saved in stack.
It saves the content of the progeam counter when CALL and RETURN instructions are executed.
Default value of stack pointer SP = 07h 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:
→ initialising stack pointer : 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).

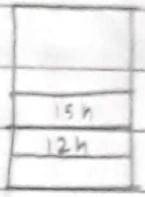
sp = 41h is incremented

sp = 42h : write 12h

sp = 42h is incremented

sp = 43h : write 15h

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



To read the data

push oeoh } write data

push ofoh }

pop ofoh } retrive the data

pop oeoh }

Syntax

push address

pop address

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 oeoh

push ofoh → top of stack before pop

pop ofoh

pop oeoh → top of stack after pop

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

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 into a r_2 and r_5 of bank 0, bank 1, bank 2 and bank 3 registers by i. names or
ii. address

by names =

```
org 0000h          (on reset content of PC = 0000h)
MOV r2, #12h   }
MOV r5, #15h   }  Bank 0
setb PSW.3     }
MOV r2, #12h   }  Bank 1
MOV r5, #15h   }
clr PSW.3      }
setb PSW.4     }  Bank 2
MOV r2, #12h   }
MOV r5, #15h   }
setb PSW.3     }
MOV r2, #12h   }  Bank 3
MOV r5, #15h   }
    end
```

more memory
less time to execute

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
```

less memory
more time to execute

In any embedded system without any OS only one program is written and run in the microcontroller.
If the bank is not mention it is Bank 0 by default.
The bank is selected by modifying the PSW register.

PSW.4	PSW.3	
RS1	RS0	
0	0	- bank 0
0	1	- bank 1
1	0	- bank 2
1	1	- bank 3

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.

UNIT - 2

8051 Addressing Modes, Instruction set and Programs

* **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 (8bit / 16 bit)
- a 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.
                                     16bit  8bit
```

Both the registers have to be of same size.

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 — 30h is address where the required data is present is part of the instruction. Address may be source or destination operand.
Operation (a) ← (30h)

MOV b, 40h

write MOV 40h, b
operation

MOV 30h, a ; (30h) ← (a)

MOV 30h, 50h — Address may be both source and destination operand
(30h) ← (50h)

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) ← ((r0)) — Here content of r_0 and r_1 is address whose content is moved to a and b respectively (read)
MOV b, @r_1 ; (b) ← ((r1))

write MOV @r_0, b ; ((r0)) ← (b)
MOV @r_1, a ; ((r1)) ← (a)

Invalid Syntax : MOV @r1, @r0

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) ← ((dptr))
write MOVX @dptr, a ; ((dptr)) ← (a)

A data can be read or written only through a register

Invalid Syntax : MOVX b, @dptr
MOVX r_2, @dptr

4. Program Memory Addressing Mode:

Indirect internal program memory addressing mode

read: MOVC a, @a+dptr. ; (a) ← ((a)+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 continous memory location starting from address 30h.

```
Org 0000h
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
Repeat: MOV @r0, a ;((r0)) ← (a)
inc r0
djnz r2, repeat
end.
```

(Initialisation, loop)

(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

a. zero : the control is transfered to next line

b. not zero : the control is transfered 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_0, #30h
MOV r_2, #5
MOV a, #0
Repeat: MOV @r_0, a
inc r_0
djnz r_2, 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_0, #30h
MOV r_2, #5
repeat: inc @r_0 ; ((r_0)) ← ((r_0))+1
inc r_0
djnz r_2, repeat
end

* To perform addition

Syntax: add a, source ; (a) ← (a) + (source)

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

Add with carry

Syntax: adc a, source ; (a) ← (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 000h
  MOV a, #12h
  MOV r3, #30h
  MOV r2, #100        r2 - outer counter
up: MOV r3, #5        r3 - inner counter
Repeat: MOV @r0, a
        inc r0
        djnz r3, repeat
        djnz r2, up
```

Here it counts for 5×100 = 500 times
Number of counter = Number of loops

* **Transfer Operation**

- Transfer of data between registers.

* Transfer of data from r2 and r5 of bank1 to r2 and r5 of bank 2 by:

i. name ii. address iii. stack

```
= By name
  Org 0000h
  setb PSW.3
  MOV r2, #12h
  MOV r5, #15h
  MOV a, r2
  MOV b, r5
  setb PSW.4
  clr PSW.3
  MOV r2, a
  MOV r5, b
  end
```

```
= By address
  Org 0000h
  MOV 0ah, #12h
  MOV 0dh, #15h
  MOV a, 0ah
  MOV b, 0dh
  MOV 12h, a
  MOV 15h, b
  end
```

= By stack

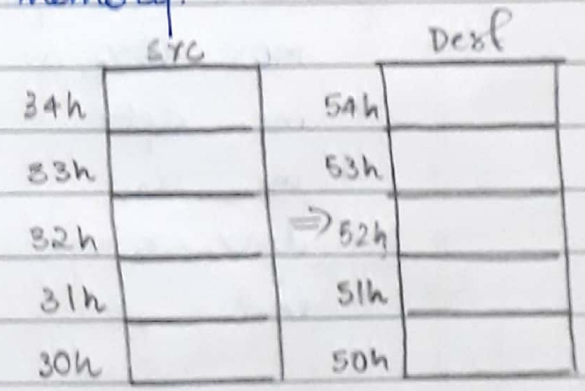
```
Org 0000h
MOV 0ah, #12h
MOV 0dh, #15h
MOV sp, #3fh          sp is initialised one address less
push 0ah ; 40h ← (0ah)  that the address we have to use
push 0dh ; 41h ← (0dh)  (top of the stack)
pop 15h
pop 12h               now top of stack is 3fh
end.
```

- 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.

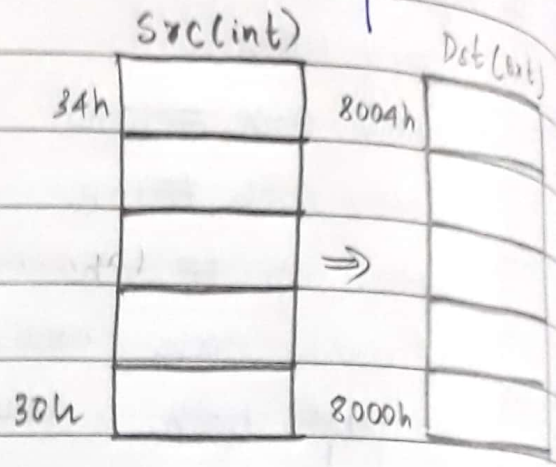
=

```
Org 0000h
MOV r0, #30h // source pointer
MOV r1, #50h // destination pointer
MOV r2, #5
up: MOV a, @r0
MOV @r1, a
inc r0
inc r1
djnz r2, up
end.
```



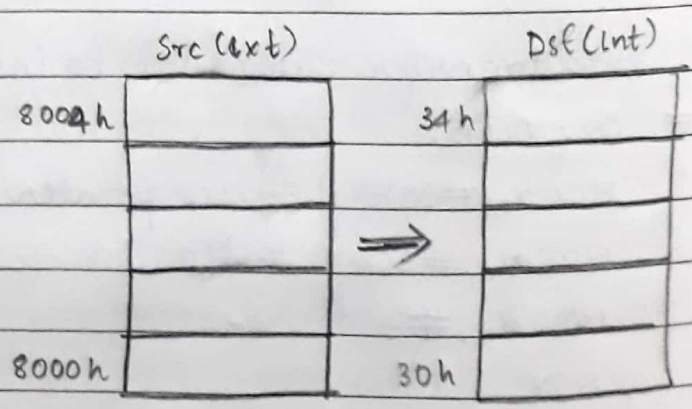
* data transfer : internal memory to external memory

```
Org 0000h
MOV r0, #30h // source pointer
MOV dptr, #8000h // dest pointer
MOV r2, #5
up: MOV a, @r0
MOVX @dptr, a
inc r0
inc dptr
djnz r2, up
end.
```



* data transfer: external memory to internal memory

```
Org 0000h
MOV r0, #30h // Dest pointer
MOV dptr, #8000h // source pointer
MOV r2, #5
up: MOVX a, @dptr
MOV @r0, a
inc dptr
inc r0
djnz r2, up
end
```



* data transfer: external memory to external memory.

= Org 0000h

src — MOV dptr, #8000h
MOV r_2, #5 // counter
MOV r_3, dph (dph only because only the higher byte address varies)

dst — MOV dptr, #9000h
MOV r_4, dph

up: MOV dph, r_3
MOVX a, @dptr
MOV dph, r_4
MOVX @dptr, a
inc dpl increments 8000h
~~inc dph~~ increments 9000h
djnz r_2, up
end.

	(Ext) src		(Ext) dst
8004h		9004h	
		⇒	
8000h		9000h	

We move the address (only the higher bit) into dph as only one pointer dptr is present for both source and destination.

case1: Here higher byte address is different whereas lower byte address is same.

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

Org 0000h
MOV r_2, #10
MOV dptr, #8000h // source pointer
MOV r_3, dph
MOV r_5, dpl
MOV dptr, #9006 // destination pointer
MOV r_4, dph
MOV r_6, dpl
repeat: MOV dph, r_3
MOV dpl, r_5
MOVX a, @dptr
MOV dph, r_4
MOV dpl, r_6
MOVX @dptr, a
inc r_5

	(Ext) src		(Ext) dst
8009h		9015h	
		⇒	
8000h		9006h	

```
inc r6
djnz 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, 0fdh, 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 1: 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, @A+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 transfering large strings, initialisation of counter is difficult.

= CASE2: 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
(or) MOV dptr, #my_data
MOV r0, #30h // destination pointer
repeat: clr a
MOV a, @a+dptr
jz stop
MOV @r0, a
inc dptr
inc r0
sjmp repeat
stop: nop
end
```

nop: no operation.

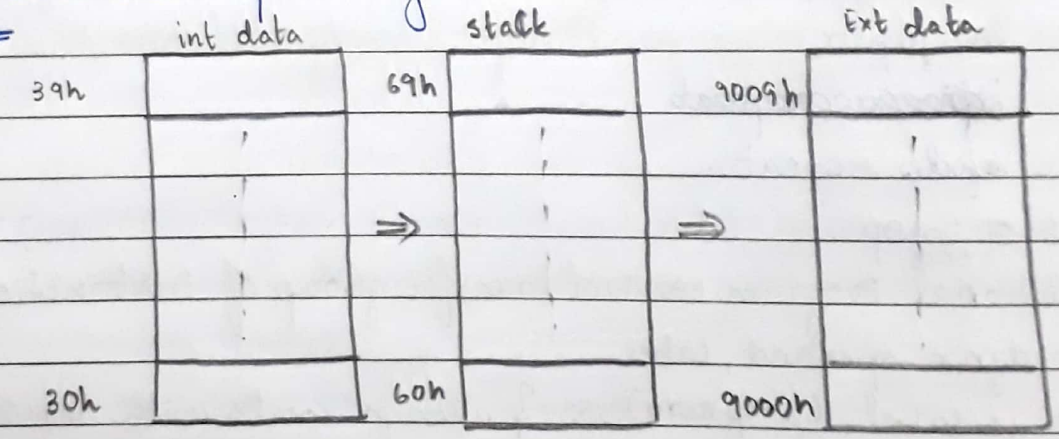
* Conditional Program control transfer group of instructions.
- djnz operand, label
- jz label (jump zero) } always works with register a.
- jnz label (jump non zero) }
- jc label (jump carry)
- jnc label (jump no carry)
- jb bit, label (if bit 1 jump to label)
- jnb bit, label (if bit 0 jump to label)
- cjne op1, op2, label (compare jump if not equal)

* 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 subprograms
  - l call label }
  - return (ret) }

2 byte inst sjmp : up - 128 bytes to down 128 bytes.
2½ byte inst ajmp : with in one page. (2k bytes)
3 byte inst ljmp : anywhere with in the available memory. (64k bytes)

a call : main memory to subprogram memory (with in one page memory)
l call : subprogram : anywhere in the memory.

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




```
Org 0000h
MOV r2, #10
MOV r0, #30h
MOV sp, #5fh
up: MOV a, @r0
    push oeoh
```

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

writing data to a

```
inc r0
djnz r2, up.
MOV r2, #10
MOV dptr, #9000h
up1: pop oeah
MOVX @dptr, a
inc dptr
djnz r2, up1
end
```

Internal data memory to stack

Initialising the counter (∵ it would be zero after first loop)

reading data from a stack to external data memory

* **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) ⟷ (b)

XCH a, @r0 ; (a) ⟷ ((r0))

XCH a, 30h ; (a) ⟷ (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 mod.
- Direct addressing mode
- using concept of stack
- Using exchange instruction

* To exchange data between r_2 and r_5 register.

= Org 0000h
MOV r_2, #12h
MOV r_5, #15h
MOV a, r_2
MOV b, r_5
MOV r_2, b
MOV r_5, a
end
(Register Addressing mode)

= Org 0000h
MOV 02h, #12h
MOV 05h, #15h
MOV sp, #2fh
same order → push 02h
push 05h
so exchange takes place → pop 02h
pop 05h
end
(using stack)

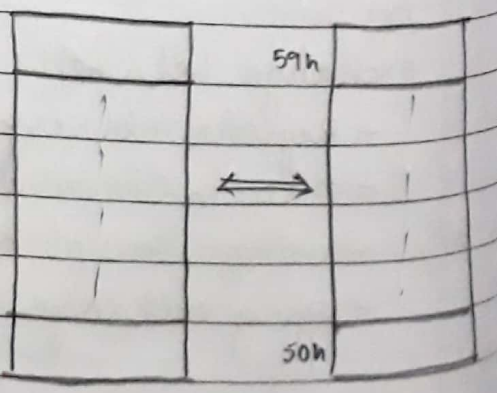
= Org 0000h
MOV 02h, #12h
MOV 05h, #15h
MOV 03h, 02h
MOV 02h, 05h
~~MOV 02h, 04h~~
MOV 05h, 03h
end
(Direct Addressing mode)

≠ Org 0000h
MOV r_2, #12h
MOV r_5, #15h
XCH a, r_2 ; a = 12h r_2 = 15h
XCH a, r_5 ; a = 15h r_5 = 12h
XCH a, r_2 ; a = 00h r_2 = 15h
end
(using exchange operation)

Memory block exchange

* To exchange a block of data in internal memory.

without exchange =
Org 0000h
MOV r_2, #10h
MOV r_0, #30h
MOV r_1, #50h
MOV a, @r_0
MOV b, @r_1
MOV @r_0, b
MOV @r_1, a



```
inc r0
inc r1
djnz r2, repeat
end
```

= Using exchange operation

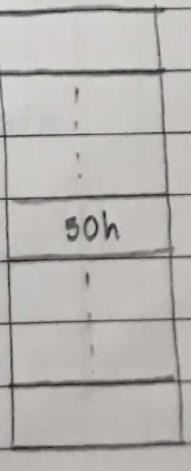
```
Org 0000h
MOV r2, #10h
MOV r0, #30h
MOV r3, #50h
repeat: MOV a, @r0      ; @r0 = 30h  a = 30h
        XCH a, @r1      ; @r1 = 30h  a = 50h
        XCH @r0, a      ; @r0 = 50h  a = 30h
        inc r0
        inc r1
        djnz r2, repeat
        end.
```

★ **Search Operation:**

Data to be searched can be a number or character.

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

★ searching a data in an array of size 10.



=

```
Org 0000h
MOV r2, #10 // size of array
MOV r0, #30h // starting address of array
repeat: MOV a, @r0
        cjne a, #50h, nteq
        MOV 40h, #0ffh // if 50h is present: true
nteq: inc r0
      djnz r2, repeat.
```

deleting by saving 0ffh in 40h

duplicating

MOV 40h, #00h // 50h is not present : false
end.

by saving 00h in 40h

* To track position

```
ORG 0000h
MOV r2, #10 // size of array
MOV r0, #30h // starting address
repeat: MOV a, r0
        cjne a, #50h, nteq
        MOV 40h, #0ffh // data is present
        MOV b, r0 // address where data is present.
nteq : inc r0
       djnz r2, repeat
       MOV 40h, #00h // data is not present.
       end.
```

* To find number of occurance.

```
ORG 0000h
MOV r2, #10 // size of array
MOV r0, #30h // starting address
MOV r3, #00h // number of occurance.
repeat: mov a, @r0
        cjne a, #50h, nteq
        inc r3
nteq : inc r0
       djnz r2, repeat
       end.
```

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

```
org 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 nxt // if a=b, to go to next comparision
nteq: jc nxt // largest number
      MOV b, a
nxt: inc r0
     djnz r2, repeat
```

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.

```
* nteq: jnc nxt // smallest number
       MOV b, a
nxt: inc r0
     djnz r2, repeat
end
```

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

There are two counter for bubble sort
- Pass counter (outer counter)
- Comparision counter (inner counter)

n - number of element in an array

number of pass → comparision

1st pass
n-1 → n-1

2nd pass
n-2 → n-2

3rd pass
n-3 → n-3

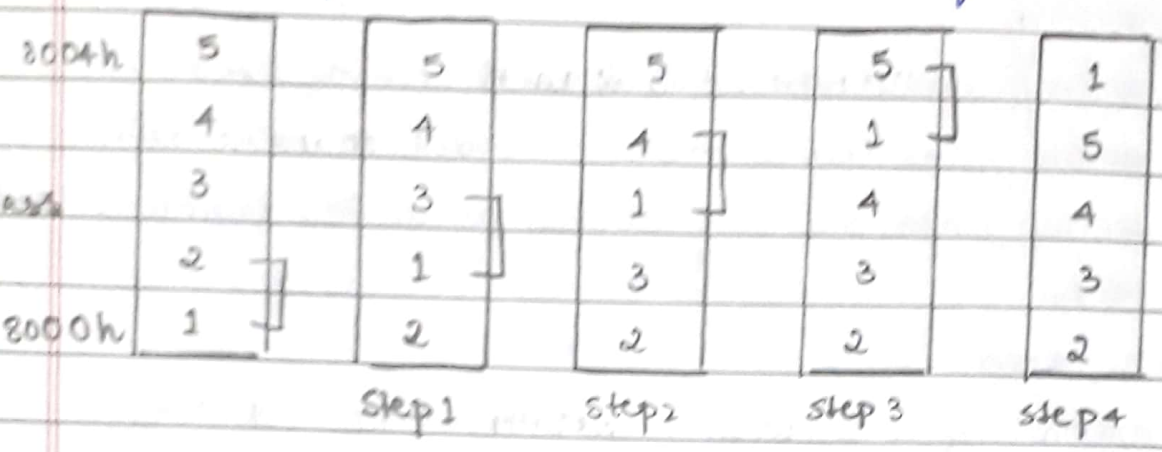
:

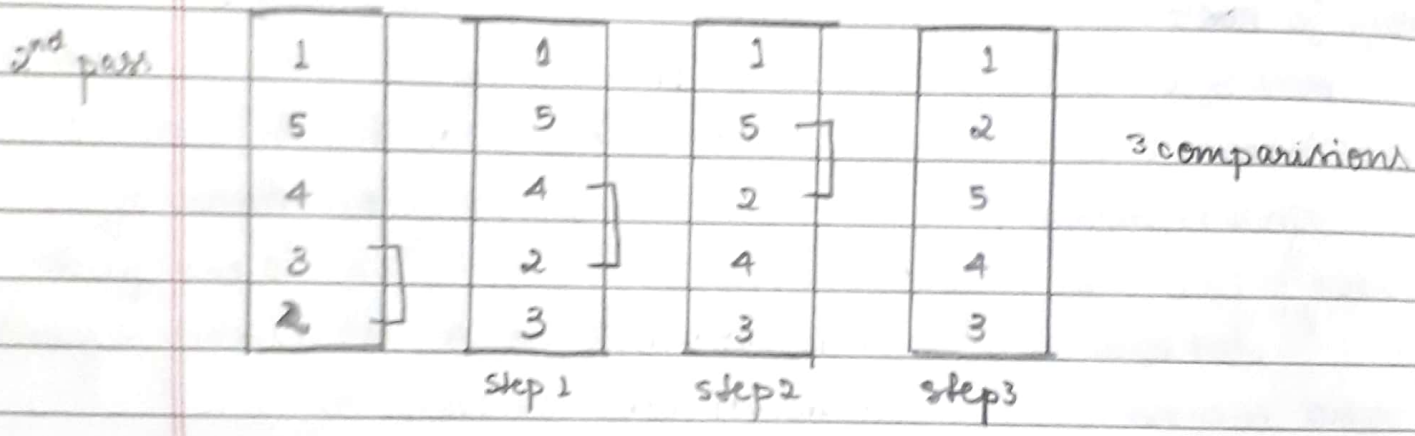
:

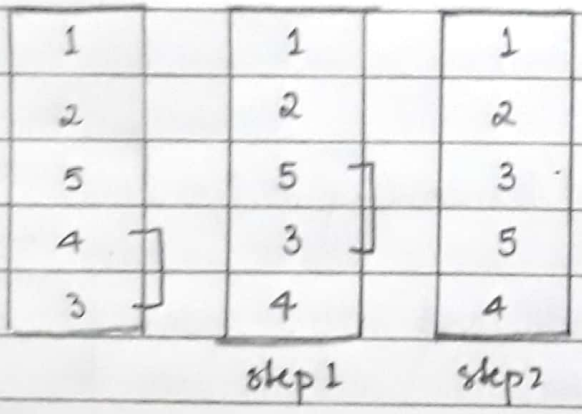
	Initially
8004h	5
8003h	4
8002h	3
8001h	2
8000h	1

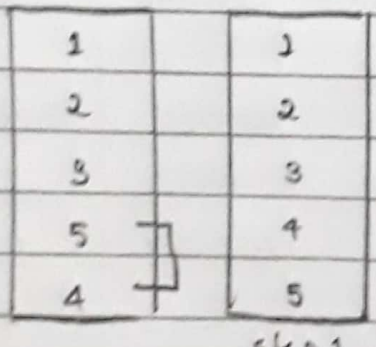
in ascending order

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









Bubble Sort

```
    org 0000h
    MOV r2, #4  // (n-1) pass counter; n - number of elements of array
up1: MOV a, r2      Here number of passes is equal to number of
    MOV r3, a       comparision. Hence both counters are given same.
    MOV dptr, #8000h // same pointer needs to incremented and value decremented
up: MOV r4, dph // lower byte address changes.
    ~~cjne 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
    cjne a, b, nteq // if a > b: no change ; a < b : exchange
    sjmp nc
nteq: jc nc         for no change
    MOV dpl, r4     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: djnz r3, up
    djnz r2, up1
    end
```

* **Arithmetic Operations:**

1. **Addition**

Syntax: ADD a, src ; (a) ← (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) ← (a) + 12h
add a, 30h ; (a) ← (a) + (30h)
add a, @r0 ; (a) ← (a) + ((r0))

Disadvantage
only two 8 bit numbers can be added at a time.

Add with carry
Syntax: addc a, src ; (a) ← (a) + (src) + (cy)
Ex: addc a, #12h ; (a) ← (a) + 12h + (cy)
addc a, 30h ; (a) ← (a) + (30h) + (cy)
addc a, @r0 ; (a) ← (a) + ((r0)) + (cy)

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 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 ajust.
2 - checks the higher nibble and adjusts the valid bcd numbers
Higher nibble of accumular
if A-F adds a correction factor of 60. and then adjusts
3 - Auxillary carry flag
It also checks the auxillary 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
```

```
  0abh
+ 6fh
------
```

* Addition of two 16 bit numbers.

```
Org 0000h
MOV a, #0abh
add a, #0f4h
MOV r2, a
MOV a, #24h
addc a, #38h
end
```

```
  24ab
+ 38f4
------
```

Result is stored as

cy	a	r2
0	5D	9F

* Sum of abh, fdh, e8h.

```
Org 0000h
MOV r2, #0 //carry tracker
MOV a, #0abh
add a, #0fdh
jnc nxt
inc r2
nxt: add a, #0e8h
jnc nxt1
inc r2
nxt1: nop
end.
```

Result

r2	a
2	90

addc should not be used while adding n numbers.

```
  ab
  cd
+ ef
------
```

addc should not be used. Hence one of the register is used as a carry tracker.

```
  abcdef
+ 123456
--------
```

addc should be used.

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 nxt
        inc r3
next:   inc r0
        djnz r2, repeat
end.
```

Result
r3 a

★ 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 nxt1
        inc r4
nxt1:   inc r0
        inc r0
        djnz r2
end
Result: r4 a.
```

30h	34
31h	12
32h	cd
33h	ab
34h	78
35h	56
36h	8Y
37h	65
38h	23
39h	64
3Ah	ef
3Bh	ab
3Ch	aY
3Dh	23

1234
abcd
5078
658Y
6423
abef
+ 23aY

* Addition of two arrays.

```
Org 0000h
MOV r2, #5 || 5 bit number
MOV r3, #0 || carry tracker
MOV r0, #30h
MOV r1, #50h
MOV dptr, #3000h
Repeat: MOV a, @r0
addc a, @r1
MOVX @dptr, a
inc r0
inc r1
inc dptr.
djnz r2, repeat
jnc stop
MOV a, #1
MOVx @dptr, a.
stop: nop
end.
```

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	87	33h	34
52h	65	32h	56
51h	43	+ 31h	78
50h	21	30h	90

3006h	
3000h	

2. Subtraction:

Syntax: subb a, src ; (a) ← (a) − (src) − (cy)

destination operand has to be accumulator.

source can by any - 8 bit data
- 8 bit register
- memory location
- pointer

Ex: Subb a, #12h ; (a) ← (a) − 12h − (cy)

subb a, b ; (a) ← (a) − (b) − (cy)

subb a, r2 ; (a) ← (a) − (r2) − (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 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
  daa
  end.
```

daa: to get bcd result

```
  84
+ 18
----
  9C  = 92 (BCD)
```

* BCD subtraction.

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

```
  4   - 1st number
- 2   - 2nd number
---
  2
```

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

```
  28
- 62
----
```

* 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 X 56
---------
   b  a
   11 78   - 34 X 56
b  a
06 0C      - 12 X 56
---------
06 1D 78
```

Result: r4 r3 r2

```
MUL ab
add a, r5
MOV r3, a // 2nd byte final result (1D)
MOV a, b
addc a, #00h // (06 + 00h (CARRY))
MOV r4, a // 3rd 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.

$\frac{344}{88}$ Q = 3, R = 80

344 - 88 =
2|144 - 88 = 56 -
2|56 - 88 =
1|156 - 88 = 76 -
1|76 - 88 =
0|176 - 88 = 86 -
86 - 88 =

```
Org 0000h
MOV dptr, # 0344h // dividend
MOV b, # 88h // divisor.
MOV r2, # 00h // number of possible subtraction / quotient.
cnt: MOV a, dpl // (a) = 44h
     subb a, b
     MOV dpl, a
     jnc nxt
     dec dph
nxt: inc r2
     MOV a, dph
     cjne a, #00h, cnt
     MOV a, dpl
     cjne a, b, nteq
     sjmp cnt
nteq: jnc cnt
     end
```

* Squareroot of a number:

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

```
Org 0000h
MOV a, #25h // number whose sqrt is to be found.
MOV b, #1 // to subtract with odd number
MOV r2, #0 // to track the number of possible subtraction and is the square root
cnt: subb a, b
     jn stop // 0-anything cy-1
     inc b
     inc b
     inc r2
     sjmp cnt
stop: nop.
     end
```

$\sqrt{25} = 5$

25 - 1 = 24
24 - 3 = 21
21 - 5 = 16
16 - 7 = 9
9 - 9 = 0

} 5

* LCM of two number:

```
Org 0000h
MOV r2, #3h
MOV r3, #12h
MOV a, r2 // to retain the original
MOV b, r3      number.
cnt: cjne a, b, nteq
     sjmp stop // if equal to end.
nteq: jnc more // b > a then
     add a, r2      add a with 3
     sjmp cnt
more: XCH a, b // a > b
     add a, r3
```

3, 15
3+3, 12
6, 12 - compare
6+3, 12
9, 12 - compare
9+3, 12
12, 12 - compare

and add until they are equal.
Always add to the smallest number.

```
xch a,b
sjmp cnt
stop: nop
end.
```

★ GCD of two numbers:

```
org 0800h
MOV a, #30h // dividend
MOV b, #12h // divisor.
MOV r2, b // to save divisor as
cnt: div ab      b becomes remainder
MOV a, r2 ↘      after division. (r2) = 12h
MOV r3, b ↗ // if r≠0, remainder
                becomes divisor.
cjne r3, #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
```

∴ 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.
```

Limitations: suitable only for 8 bit numbers.

* Generation of series

= First 10 natural numbers.

```
Org 0000h
MOV r2, #10 // number of elements in the series
MOV r0, #30h
MOV a, #1 // starting point
up: MOV @r0, a
    inc a
    inc r0
    djnz r2, up
    end
```

= First 10 odd number

```
Org 0000h
MOV r2, #10
MOV r0, #30h
MOV a, #1
up: MOV @r0, a
    inc r0
    add a, #2
    daa
    djnz r2, up
    end.
```

1 → 0 - first 10 even numbers.

*5

We dont double increment as on incrementing it leads to a hexadecimal numbers.

*5 → multiples of 5.

= Fibbonacci series: 0, 1, 1, 2, 3, 5, 8 .....

```
Org 0000h
MOV r2, #8 // First 10 number of series (since first 2 numbes are stored)
MOV dptr, #3000h
MOV a, #0
MOVx @dptr, a
inc a
inc dptr
MOVx @dptr, a
```

```
        MOV r3, #1
repeat: add a, r3
        XCH a, R0
        inc dptr
        MOVX @dptr, a
        djnz r2, repeat
        end
```

3009h	34
	3
	2
	1
	1
3000h	0

★ To check an odd number / even number

LSB : 0 – Even number

LSB : 1 – Odd number

```
= Org 0000h
MOV r0, #30h // To check the data present in some address
MOV a, @r0 // 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 positive / negative number

MSB : 0 – positive

MSB : 1 – negative.

```
= Org 0000h
MOV r0, #30h
MOV a, @r0
jb acc.7, negative_no //
MOV 40h, # 00h
sjmp stop
```

negative_no: MOV 40h, #0ffh
stop: nop
end

* check the status of particular bit.

= org 0000h
MOV R0, #80h || data to be checked
MOV a, @R0
jb acc.4, make_it_zero
setb acc.4
sjmp stop
make_it_zero: clr acc.4
stop: nop
end.

b7 b6 b5 b4 b3 b2 b1 b0
1 0 0 (1) 0 0 1 1
check this bit
make it 0 if 1 and
1 if 0.

To change more than one bit we use logical instructions.
To change with 1
To keep as it is xor with 0.

* Logical group of instruction

orl a, src : or operation (setting)
anl a, src : and operation (Masking)
xrl a, scr : xor operation (complementing)

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

* To change more than one bit.

(1)010(1)100 10011000
ACh 98h

MOV a, #0ACh
xrl a, #98h

* Masking

to clear we can use anl operator.
more than 1 bit without disturbing other bits

25h

0010 0101

0000 1111

0000 0101

20h

masking: clearing lower nibble

⇒ Org 0000h
MOV a, #25h
and a, #0fh
end

* setting
setting more than one bit without disturbing other bits we use orl operation.

⇒ Org 0000h
MOV a, #0e0h
orl a, #0efh
end

e0h

1110 0000

1110 1111

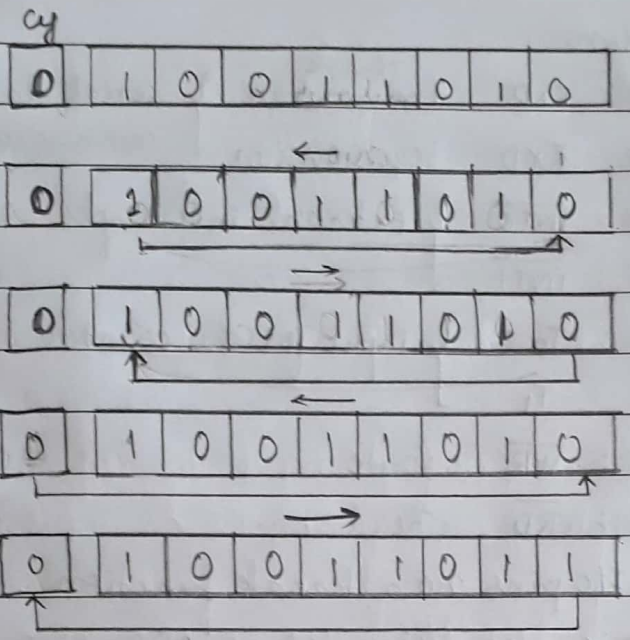
1110 1111

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:



Note: swap a = 4 rotate instruction.

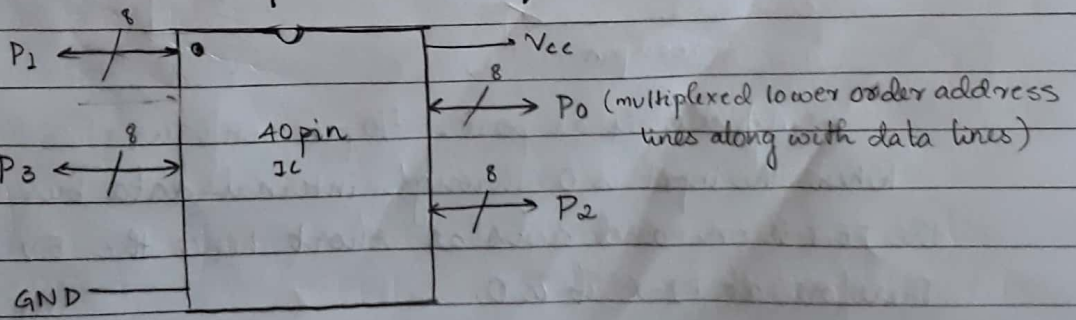
* I/O ports:

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.

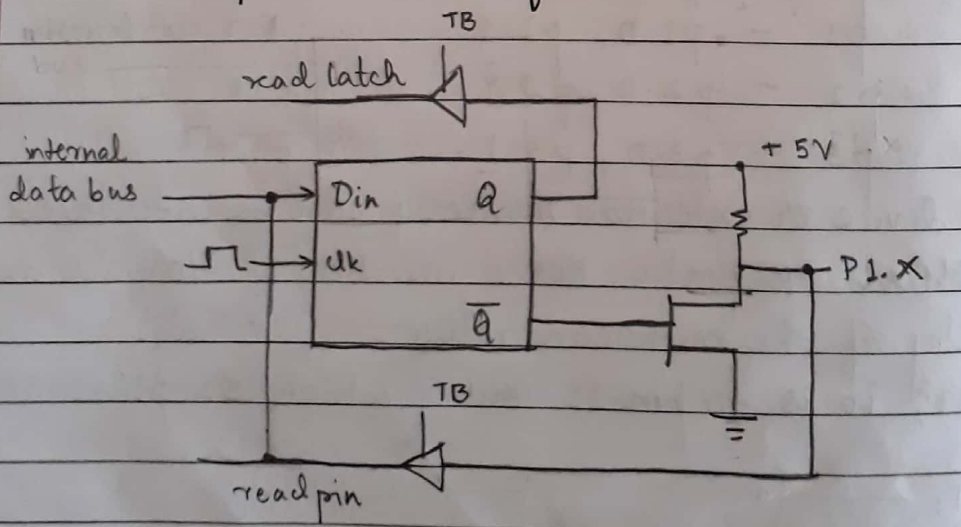


- Alternate function

Port 3: P3.0 : RXD (recieve) } serial data communication.
P3.1 : TXD (transmission)
P3.2 : $\overline{INT0}$ } External interrupts
P3.3 : $\overline{INT1}$
P3.4 : T0 } Input for the counter
P3.5 : T1
P3.6 : $\overline{WR}$ Write
P3.7 : $\overline{RD}$ 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 8 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 ⇒ $\overline{Q}$ = 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 ⇒ $\overline{Q} = 0$ 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). ⇒ Din = 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:

```
again: sjmp again
(infinite loop)
```

* To send and observe 16 bit data.

```
= org 0000h
  MOV a, #18h
  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 configured.

* Port 0 as input and port 1 as output (complementing input)

Ex:

```
= org 0000h
  MOV p0, #0ffh // configure as input port
```

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

```
loop: MOV a, p0
      cpl a
```

P0.7 P0.0
0 0 0 0 1 0 1 0
input pin

```
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 zeroes to port 1 and send the number of one's to port 2.

=
```
Org 0000h
MOV P0, #0ffh // configure port 0 as input port
MOV r3, #00h // number of 0's
MOV r4, #00h // number of 1's
MOV r2, #8 // counter: number of bits to be checked
MOV a, P0
repeat: RLC a
        jc ones
        inc r3
        sjmp nxt
ones: inc r4
nxt: djnz r2, repeat
MOV p1, r3
```

```
MOV p2, r4
again: sjmp again
end.
```

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

```
org 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 zeros
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
```

★ Number of odd and even numbers.

```
= Org 0300h
array: dB 01h, 64h, 0abh, 74h, 80h,
          33h, 59h, 43h, 0Bh, 16h
Org 0000h
MOV r2, #10
MOV dptr, # array
MOV r3, #0 // number of odd numbers
MOV r4, #0 // number of even numbers
repeat: clr a
MOVC a, @a+dptr
jb acc.0, odd
inc r4
sjmp nxt
odd: inc r3
nxt: inc dptr
   djnz r2, repeat
end
```

negative
positive
acc.4

To store data sequence of data

```
Org 0000h
MOV r2, #10
MOV r0, #30h
MOV a, #1
up: mov @r0, a
   inc r0
   inc a
   djnz r2, up
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...
movc a, @a+dptr (to...
```

★ To separate odd and even numbers

```
= Org 0300h
array: db 01h, 64h, 0abh, 74h, 80h, 33h, 59h, 43h, 0eh, 16h
Org 0000h
MOV r2, #10
MOV dptr, #array
MOV r0, #30h // to store odd numbers
MOV r1, # 50h // pointer to store even numbers,
Repeat: clr a
```

negative
positive

```
MOVC a, @a+dptr
jb acc.0, odd        (acc.4)
MOV @r1, a
inc r1
sjmp nxt
odd: MOV @r0, a
     inc r0
nxt: inc dptr.
     djnz r2, 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 dptr, # array
MOV r2, #10
MOV r3, #0 // sum of odd numbers
MOV r4, #0 // sum of even numbers
repeat: clr a
MOVC a, @a+dptr
jb acc.0, odd
add a, r4
MOV r4, a
sjmp nxt
odd: add a, r3
     MOV r3, a
nxt: inc dptr
     djnz r2, 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
anl 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 ×
00010011
✓ ×
11100000

12 ✓
00010010
✓ ✓

To check this we perform and operation with 11100000 (e0h). 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, #0ffh // P0 as input port
MOV a, p0
MOV b, a
anl a, #0fh // 02
MOV p1, a
MOV a, b
anl a, #0f0h // 30
swap a // 03
MOV p2, a
end.
```

– Unpacked BCD to packed BCD

```
Org 0000h
MOV p0, #0ffh // P0 as input port
MOV p1, #0ffh // 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, #0ffh // input port
MOV a, P0
cjne a, #0ah, nteq
```

```
sjmp nxt
nteq : jc nxt1
nxt : add a, #1
nxt1 : adda, #30h
       MOV P1, a
       end.
```

ASCII to Hex:

```
Org0000h
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 // inputport.
MOV a, P0 //45
MOV b, a //45
anl a, #0fh //05
add a, #30h //35
MOV P1, a
MOV a, b
anl a, #0f0h //40
swap a //04
add a, #30h //34
MOV P2, a
end
```

ASCII to BCD:

```
Org 0000h
MOV p0, # 0ffh || input port
MOV a, p0 || '4'
anl 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 p0, # 0ffh
MOV a, p0
MOV b, a
LCALL conversion
MOV p1, a
```

Subroutine - conversion

```
Org 0400h
conversion: anl 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 p0, #0ffh
MOV a, p0  // 32 BCD
MOV b, a
anl a, #0fh // 02
MOV r2, a
MOV a, b
anl a, #0f0h // 30
swap a // 03
MOV b, #0ah
MUL ab
add a, r2
MOV p1, a
end.
```

32 → 20h

03 × 0ah + 02 × 1

= 1E + 2

= 20h

Hexa to BCD :

5. Binary to Gray

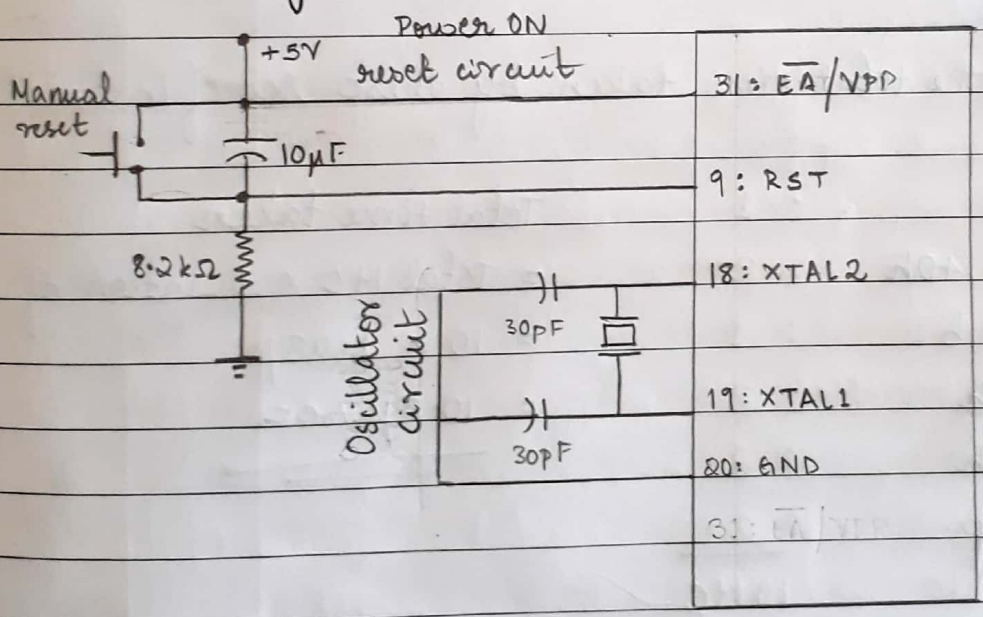
```
Org 0000h
MOV P0, #0ffh
MOV R2, P0
MOV A, R2
RRC A
xrl A, R2
end.
```

* **INTERFACING:** Pin configuration of 8051: textbook 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.

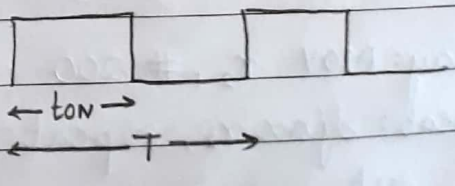


$$f = \frac{1}{2\pi\sqrt{LC_{eq}}}$$

where $C_{eq} = \frac{C_1 C_2}{C_1 + C_2}$

- **Duty cycle:**

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



where t_{ON}: time it is ON

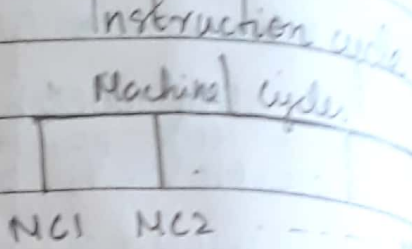
T: total time period.

- **Instruction cycle:**

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

Each machine cycle consists of 12 clock cycles.

Operating frequency
$f = 11.059\ MHz$



Time period: $T = 0.09\mu sec$

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

As reset takes 2 machine cycles

∴ It takes $1.08\mu \times 2 = 2.16\mu sec$ 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, jmp
- Only two instructions take 4MC: MUL and DIV.

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

1.

MOV a,b	: 1
MOV 30h,40h	: 2
MOV r_2, b	: 1
PUSH 30h	: 2
POP 40h	: 2
jmp up.	: 2
	10 MC

Total time taken
= Total MC × Duration of 1 MC
= $10 \times 1.08\mu$
= $10.8\mu sec$

2.

delay: MOV r_2, #200	: 1×1	= 1
repeat: djnz r_2, repeat	: 2×200	= 400
ret	: 2×1	= 2
		403 NC

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

3. delay: MOV r_2, #200 : 1×1 = 1
up1: MOV r_3, #200 : 1×200 = 200
up: djnz r_3, up : 2×200×200 = 80000
djnz r_2, up1 : 2×200 = 400
ret. : 2×1 = 2
80603MC

∴ Total time taken
= 80603 × 1.08μ = 87.051msec //

4. delay: MOV r_2, #200 : 1×1 = 1
up1: MOV r_3, #200 : 1×200 = 200
up: nop : 1×200×200 = 40000
djnz r_3, up : 2×200×200 = 80000
djnz r_2, up1 : 2×200 = 400
ret : 2×1 = 2
120603 MC

∴ total time taken
= 120603 × 1.08μ = 130.251msec //

5. delay: MOV r_2, # 0ffh : 1×1 = 1 because 0ffh = 255
up1: MOV r_3, # 0ffh : 1×255 = 255
up: djnz r_3, up : 2×255×255 = 130050
djnz r_2, up1 : 2×255 = 510
ret : 2×1 = 2
130818 MC

∴ total time taken
= 130818 × 1.08μ = 141.283mS //

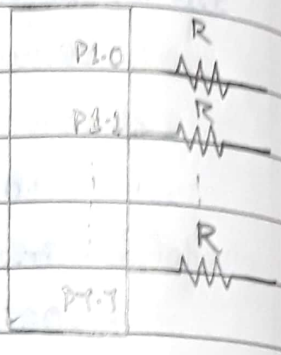
Q: Calculate the time taken by 8051 microcontroller to PUSH content of dptr, b, a, PSW and PC.

For dptr → push 82h : 2 } 4MC
push 83h : 2 }
For b → push f0h : 2MC
For a → push e0h : 2MC

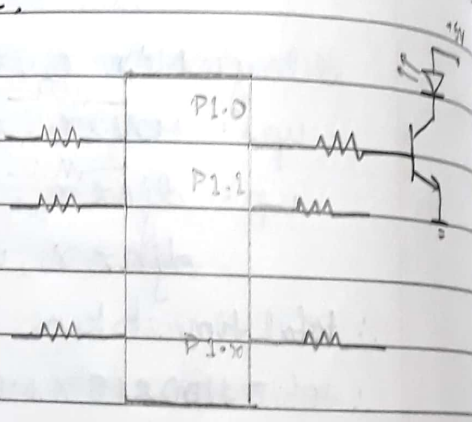
For PSW → push PSW : 2MC
For PC → push PCL : 2] 4MC
push PCH : 2

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

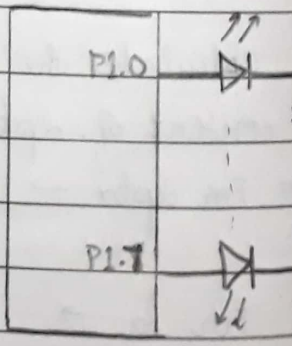
— R : current limiting resistor
Due to interfacing of input/output devices there may be excessive flow of current due to which controller might get damaged. Hence the current limiting resistor limits the current flowing from or into the µC.



— For all output devices a driver is essential as the current recieved 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 up counter, down counter or up-down counter.



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

```
Org 0000h
MOV a, #00h        (ffh for down counter)
cnt: MOV p0, a
     ACALL delay
     inc a         (dec, ffh)
     cjne a, #00, cnt
back: sjmp back
     end
```

Delay subroutine:

```
delay: MOV r2, #10
up2:   MOV r3, #255
up1:   MOV r4, #255
up:    djnz r4, up
       djnz r3, up1
       djnz r2, up2
       ret
```

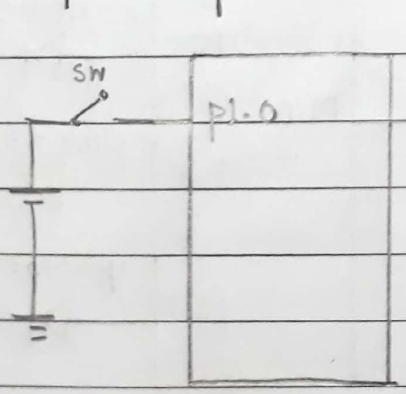
* BCD up counter

```
Org 0000h
repeat: MOV a, #00      (99h)
cnt : MOV p0, a
      ACALL delay
      add a, #1         (99h)
      daa               (99h)
      cjne a, #00, cnt
      end
```

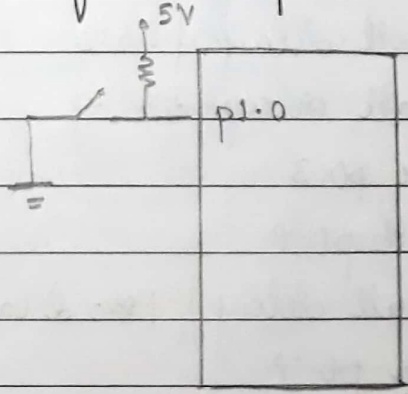
(for down counter)

```
delay: MOV r2, #10
up2:   MOV r3, #255
up1:   MOV r4, #255
up:    djnz r4 up
       djnz r3, up1
       djnz r2, up2
       ret
```

* simplest input device to be interfaced: Dip switch



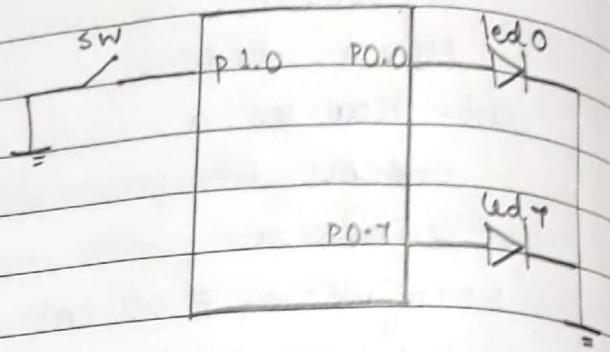
pressed: 1
not pressed: 0



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
sjmp nxt.
led Y: setb p0.Y
nxt : sjmp repeat
end
```



* TRAFFIC LIGHT

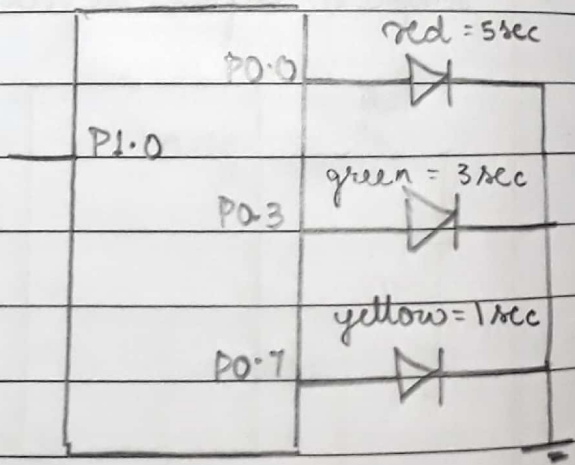
```
= org 0000h
repeat: setb p0.0
acall delay
acall delay
acall delay
acall delay
acall delay
clr p0.0
setb p0.3
acall delay
acall delay
acall delay
clr p0.3
setb p0.Y
acall delay // 1 sec delay
clr p0.Y
sjmp repeat
end
```

(5 sec delay)

(3 sec delay)

delay routine: introduces delay of 1 sec

```
delay: MOV r2, #10
up2: MOV r3, #255
up1: MOV r4, #255
up: djnz r4, up
    djnz r3, up1
    djnz r2, up2
    ret
```



* BLINKING OF LED:

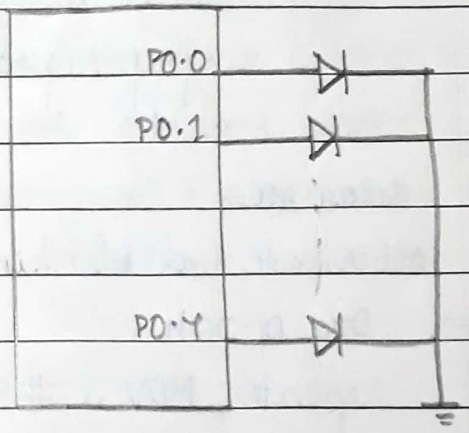
All the leds at the same time

```
= org 0000h
repeat: MOV a, #0ffh
MOV p0, a
ACALL delay
MOV a, #00h
MOV p0, a
ACALL delay
sjmp repeat
end.
```

```
delay: MOV r2, #10
up2: MOV r3, #255
up1: MOV r4, #255
up: djnz r4, up
    djnz r3, up1
    djnz r2, up2
    ret
    ~~ret~~.
```

or:

```
org 0000h
repeat: cpl a
MOV p0, a
ACALL delay
sjmp repeat
end.
```



Blinking of particular led

```
= org 0000h
repeat: cpl a
MOV p0, a
ACALL delay
sjmp repeat
end.
```

```
delay: MOV r2, #10
up2: MOV r3, #255
up1: MOV r4, #255
up: djnz r4, up
    djnz r3, up1
    djnz r2, up2
    ret
    ~~end~~.
```

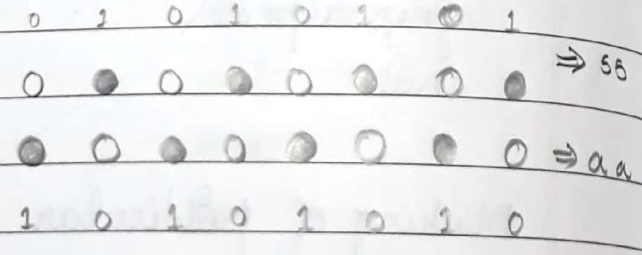
Led with switch

```
= 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
```

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.
```

```
= Org 0000h
repeat: setb p0.0
  MOV 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.
```



UNIT - 3

8051 Programming in C

- C language is machine independent wheres as 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 (8bit) : -128 to 127
ii. unsigned char a : 0 to 255 : $2^8 - 1$
iii. signed short char a (16 bit) : -32768 to +32767
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 (32bit) : -2147483648 to +2147483647
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
or: bit — as a variable

Ex: sbit led0 = p1^0; sbit led7 = p1^7;
or bit led0 = p1^0; sbit s10 = p2^0;

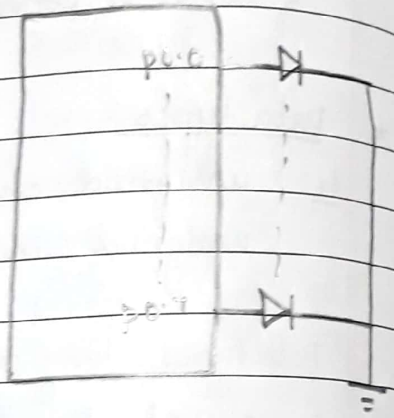
ii. sfr : special function register : with respect to byte.

Ex: sfr input = 0x80; port 0
sfr output = 0x90; port 1
sfr leds = 0xa0; port 2
sfr sws = 0xb0; port 3
sfr acc = 0xe0; accumulator

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

* To send data to represent in binary form.

```
#include<reg51.h>
sfr output port = 0x80;
void main (void)
{
  while (1)
  { output port
    P0 = 0x20;
  }
}
```



* To switch on only led 0 and led 7.

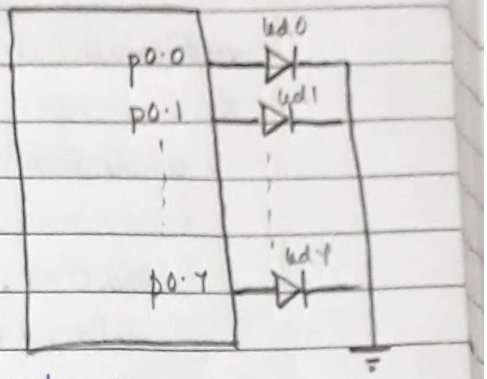
```
#include <reg51.h>
sbit led0 = p0^0;
sbit led7 = p0^7;
void main()
{
  while(1)
  {
    led0 = 1;
    led7 = 1;
  }
}
```

* To blink leds of port 0.

```
#include <reg51.h>          to declare as
void delay ();              → global function
void delay ()               → then it can be written after main
{
  unsigned int x, y;
  for (x=0; x<=60000; x++)
  {
    for (y=0; y<=50000; y++);
  }
}
void main ()
{
  while (1)
  {
    P0 = 0x00;
    delay ();
    P0 = 0xff;
    delay ();
  }
}
```

software delay time taken not known accurately. Just to have a delay.

1275 : for delay of 1m sec

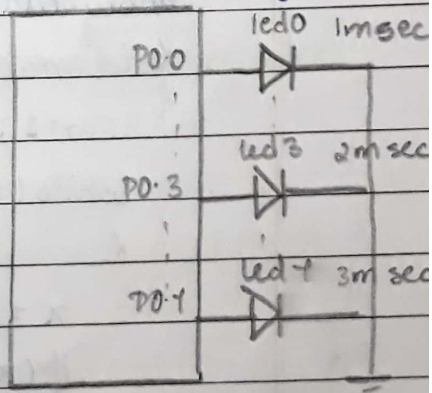


delay function
- if local function: before main
- if global function: anywhere.

* To blink certain leds for certain time

```
#include <reg51.h>
sbit led0 = P0^0;
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++);
}
```

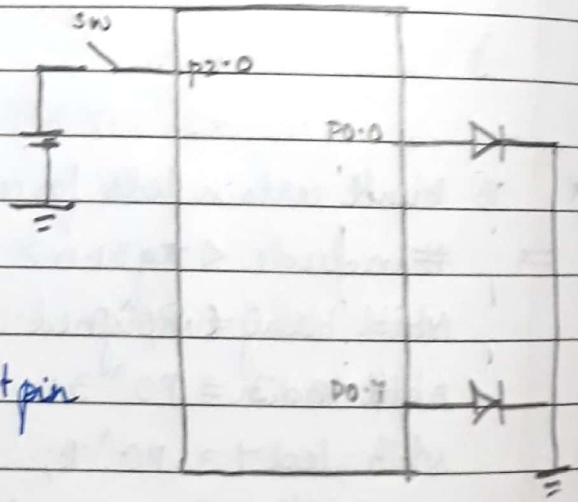
Assuming 1275 gives 1ms delay.



```
void main()
{
  while(1)
  {
    led0=1;
    delay(1270);
    led0=0;
    led3=1;
    delay(2550);
    led3=0;
    led7=1;
    delay(3825);
    led7=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)
    {
      led0 = 1;
```



```
        delay(500);
        led 0 = 0;
        delay(500);
    }
    else
    {
        led 7 = 1;
        delay(500);
        led 7 = 0;
        delay(500);
    }
  }
}
void delay(unsigned int a)
{
    unsigned int i;
    for(i=0; i<=a; i++);
}
```

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 >> (3) ← no of bits
2. << shift left
   Ex: a = a << 2

* Timers and counters:

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

Timers:

8051 has two 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 (higher byte) TL0 (lower byte) : timer 0
  TH1 TL1 : timer 1

Each register can be addressed individually as TL0, TH0, 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/$\overline{T}$	m_1	m_0	gate	C/$\overline{T}$	m_1	m_0
Timer 1 / counter 1				Timer 0 / counter 0			

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

m_1 and m_0 : used to configure to different modes

m_1	m_0	
0	0	: mode 0
0	1	: mode 1
1	0	: mode 2
1	1	: mode 3

C/$\overline{T}$: 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.

IE1 : External interrupt 1 edge flag. set when hardware interrupt.

IT1 : Interrupt 1 type control bit. set / cleared : falling edge / level triggered external interrupt.

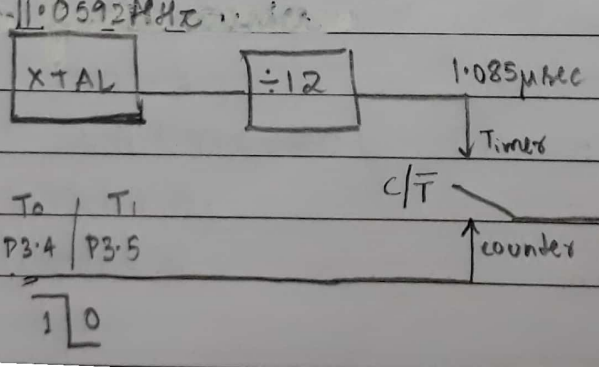
IE0 : Ext Int 0 edge flag. Set when hadware interrupt. cleared once served

IT0 : Int 1 type control bit. set / cleared by software : falling edge / level triggered external interrupt.

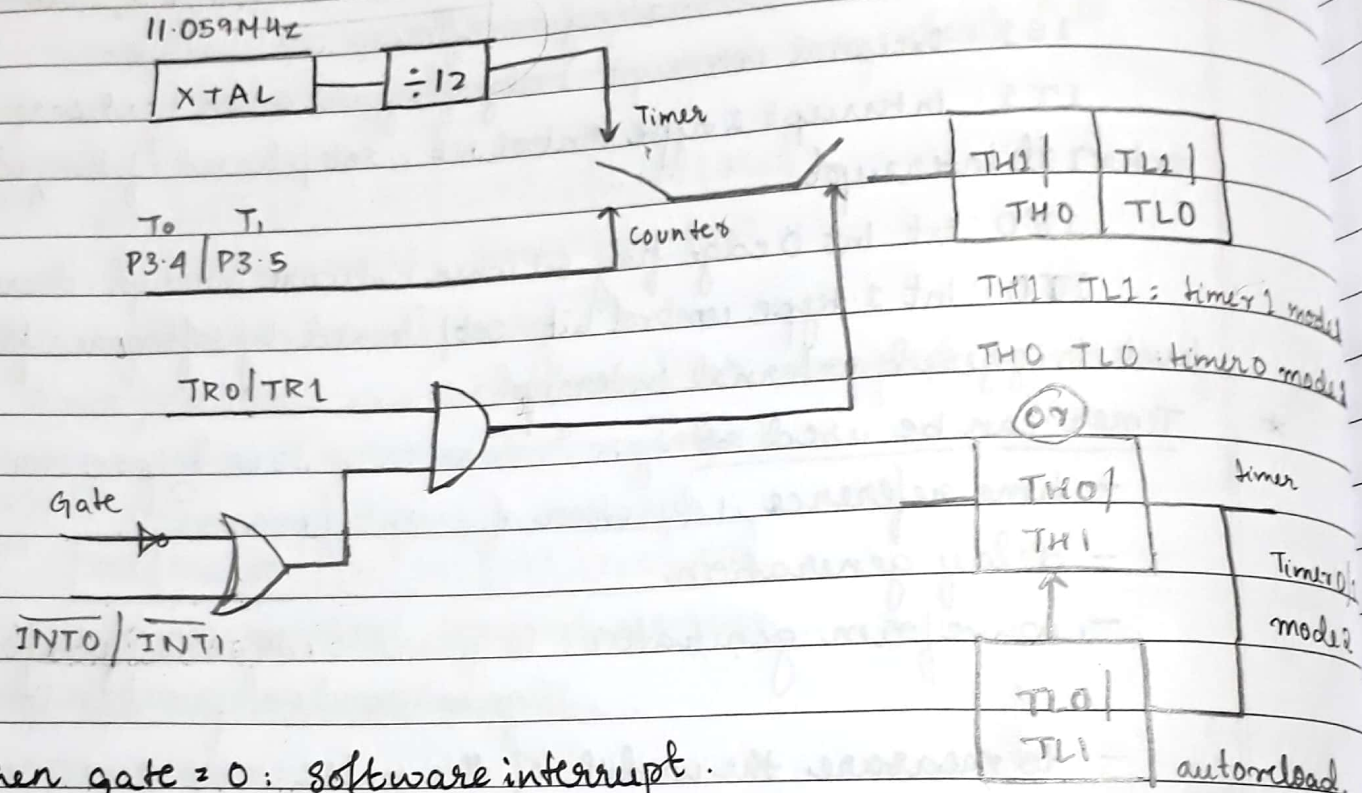
* Timer can be used as
  - time reference : duration of event / process.
  - delay generation
  - waveform generator : to generate square waveform only (symmetric and unsymmetric (pulse) square wave)
  - to measure the width of the pulse : measure on time.
  - to measure frequency of the pulse :
  - 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 = 2MC ; The transition has to be 1 to 0.



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 $\overline{INT0}$ and $\overline{INT1}$ (i.e., pins P3.2 and P3.3) for timers 0 and 1 respectively.

* Modes of Timer / counter

Mode 1 : 16 bit mode → 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

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

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

Max value of 16 bit − count = 65535 − 230 = 65305 = FF19h

Now FF19h is loaded initially to the timer.

TMOD — Timer 0

0	0	0	0	0	0	0	1
Gate	C/T	m_1	m_0				

TCON

			1				
TF1	TR1	TF0	TR0				

In ALP:

```
delay: MOV tmod, #01h
       MOV tlo, #19h
       MOV tho, #offh
       setb tro
check: jnb tf0, check  // in loop when TF0 = 0.
       clr tro  //stops timer
       clr tf0
       ret
```

In C:

```
void delay()
{
  TMOD = 0X01;
  TLO = 0X19;
  THO = 0XFF;
  TRO = 1;
  while (TF0 == 0);
  TRO = 0;
  TF0 = 0;
}
```

* 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 = $\frac{\text{delay required}}{\text{timer rate}} = \frac{50m}{1.085\mu} = 46083$

65535 − 46083 = 19452 = 4BFC h

```
void delay()
{
  TMOD = 0X10 ; // Timer1: mode1
  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_1	m_0	0	0	0	0
0	0	0	1	Timer 0 not used			

TMOD = 10 h

TF1	TR1	TF0	TR0				
1							

4BFC → FFFF
then it resets 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 71.6m sec. (i.e., 65535 x 1.085 μ)

* To generate a delay of 1 sec 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 → 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 0.27 m sec. (255 × 1.085μ) Hence not preferred.

Hence we use timers in mode 2 to generate pulse.

Wave form generation:

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

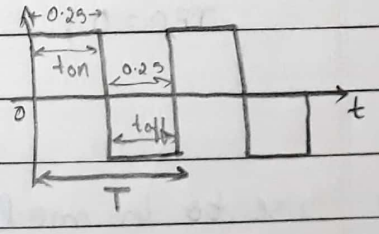
* Develop an 8051 C program to generate a square wave of frequency 2 kHz using timer 0 in mode 2 with 50% duty cycle at pin P2.0.

$f = 2kHz \Rightarrow T = 0.5 m sec$

$T = t_{ON} + t_{OFF}$

50% duty cycle $\therefore t_{ON} = t_{OFF}$

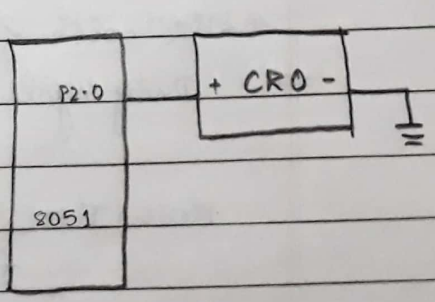
$\therefore t_{ON} = t_{OFF} = 0.25 m sec$



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

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

$255 - 230 = 25 = 19h$ //



```
#include <reg51.h>
sbit squa = P2^0;
void delay();
void main()
{
    while(1)
    {
        squa = 1;
        delay();
        squa = 0;
        delay();
    }
}
void delay()
{
    TMOD = 0x02; //timer 0; mode 2
    TH0 = 0x19;
    TR0 = 1;
    while(TF0==0);
    TR0 = 0;
    TF0 = 0;
}
```

* Use t0 in mode 1 to produce a square wave of frequency 4 kHz with 20% duty cycle at pin P2.0.

= Duty cycle : $\delta = \frac{t_{ON}}{T}$

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

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

$T = t_{ON} + t_{OFF}$

$\Rightarrow t_{off} = T - t_{ON} = 0.25\mu - 0.05m$

$t_{off} = 0.2m$ sec

$\therefore t_{off} = 4\, t_{ON}$

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

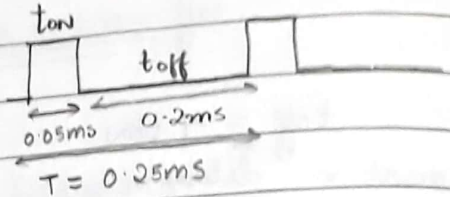
$$\text{Timer rate} = \left[\frac{\text{crystal frequency}}{12}\right]^{-1} = \left[\frac{11.0592M}{12}\right]^{-1} = 1.085\mu sec$$

$65535 - 46 = 65489 = FFD1H$

```
#include <reg51.h>
sbit sqw = P2^0;
void delay (); unsigned char
void main ()
{
   while (1)
   {
      sqw = 1;
      delay (1);
      sqw = 0;
      delay (4);
      delay ();
      delay ();
      delay ();
   }
}
void delay ()
{ unsigned char i;
  for (i=0; i<=2; i++)
  {TMOD = 0X01;
   TLO = 0XD1;
```

* Passing parameter technique

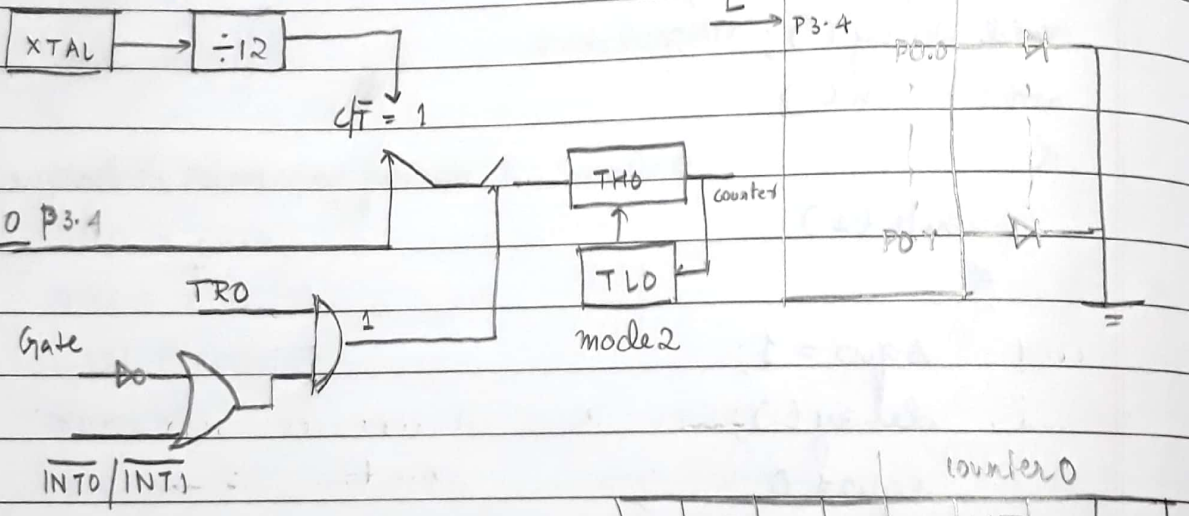
```
TH0 = 0xFF;
TR0 = 1;
while (TR0 == 0);
TR0 = 0;
TF0 = 0;
}
```



★ **Counters:**

★ use counter 0 in mode 2 to count the number of external pulses applied and display the count value on set of LED's interfaced through port 0.

Source:



counter 0

Gate	C/T	m_1	m_0	Gate	C/T	m_1	m_0
0	0	0	0	0	1	1	0

TMOD = 06h

= **In ALP:**

```
Org 0000h
setb P3.4 // configuring as input pin.
MOV tmod, #06h //counter 0 in mode2
repeat: setb tr0 //start counter.
```

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

To increment it waits for the pulse to be applied after it is started.

```
up: MOV a, th0
    MOV p0, a
    jnb tf0, up
    clr tr0
```

dr tfo
simp repeat
end.

In C:

```
#include <reg51.h>
sbit exp = p3^4;
void main()
{ exp = 1;
  while(1)
  {TMOD = 0x06;
   TR0 = 1;
   do
   {
     P0 = TL0;
   } while (TF0 == 0);
   TR0 = 0;
   TF0 = 0;
  }
}
```

* Timers:

- Mode 0: 13 bit mode → 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.

$$y = \left[\left(\text{number}/0x0a \right) << 4 \right) + \left(\text{number} \,\%\, 0x0a \right) \Big]$$

10h = 16

10/a = 01 quotient

10 % a = 06 remainder

01 swap 10

10 + 06 = 16

* 7 segment display:

1. Common anode

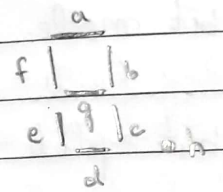
   Input 0: ON

   Input 1: OFF

2. Common cathode

   Input 0: OFF

   Input 1: ON



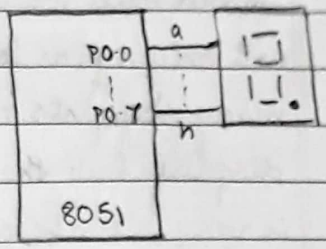
To interface a 7 segment display to any port of a microcontroller there are three ways:

1. Static mode } parallel mode
2. Multiplexed mode } parallel mode
3. Serial mode
4. Direct 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 : h g f e d c b a

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

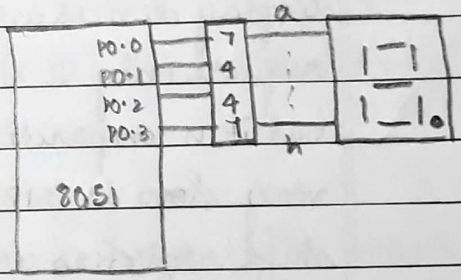
Disadvantage:

- 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 drives.

* **Parallel Mode:**

- **Static Mode**

In this mode a display driver 7447 IC which is 4:8 decoder it converts BCD input to 7 segment code.



Advantages

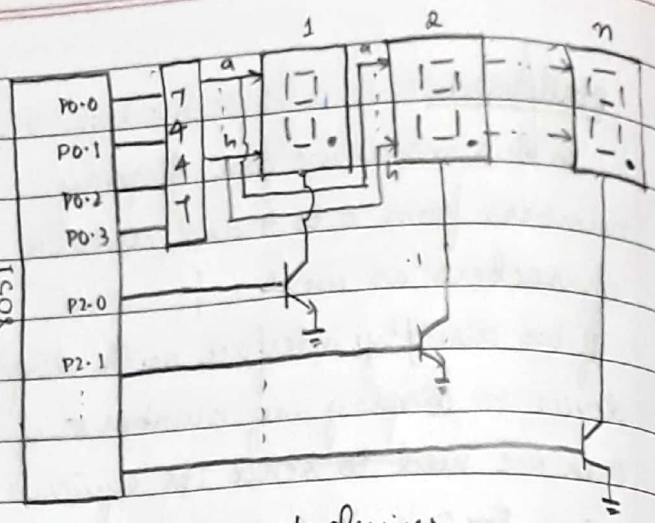
- Conversion to equivalent code is done by 7447.
- Number of I/O pins required is reduced by 4. Hence maximum of 8 display segments can be interfaced.
- Parallel transmission.

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

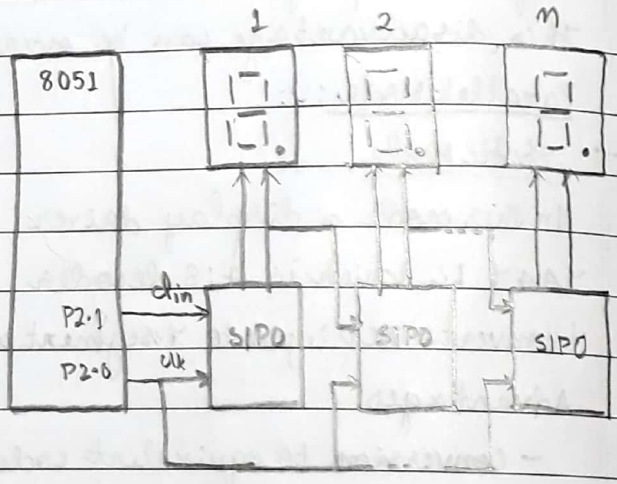
- Each 7447 can be used for a maximum of 28 displays.

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 comparitively very slow as SIPO shift registers are used.



Each display device requires a separate SIPO shift register.

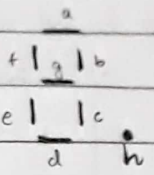
For SIPO shift register we require 5 clock pulses.

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

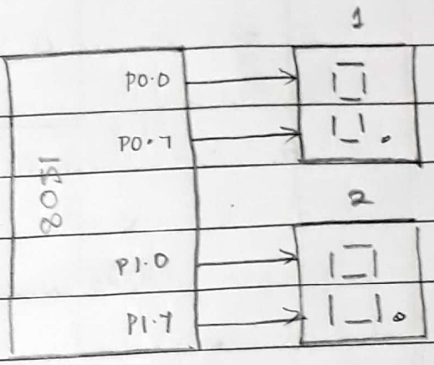
Since no display driver 7447 is used we will have to send the equivalent code for each display segment.

* Display 12 on 4-segment display in different modes of interfacing.

i. Direct mode
ii. Parallel mode
    - static mode
    - Multiplexed mode
iii. Serial mode



i. Direct mode 16 I/O pins for two 7 segment




```
#include <reg51.h>
void main()
{
  while(1)
  {
    P0=0xf9;
    P1=0xA4;
  }
}
```

For common anode

hgfedcba

1: 11111001 - f9h

2: 10100100 - A4h

For common cathode

hgfedcba

1: 00000110 - 06h

2: 01011011 - 5Bh

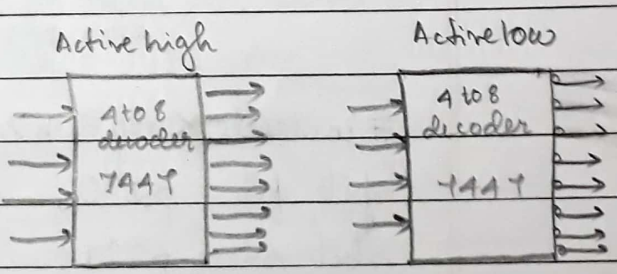
ii. Parallel mode

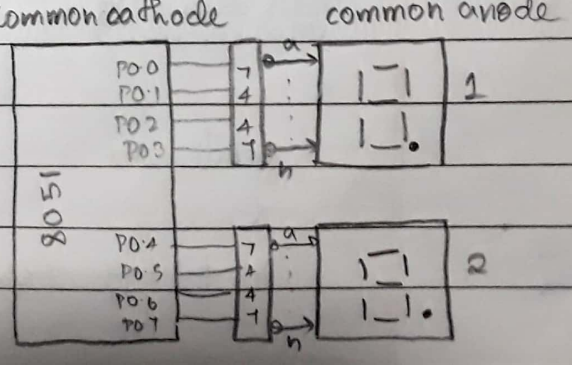
static mode : 8 I/O pins for two segments

```
#include <reg51.h>
void main()
{
  unsigned char a,b,c,d;
  while(1)
  {
    P0 = 0x12;
  }
}
```

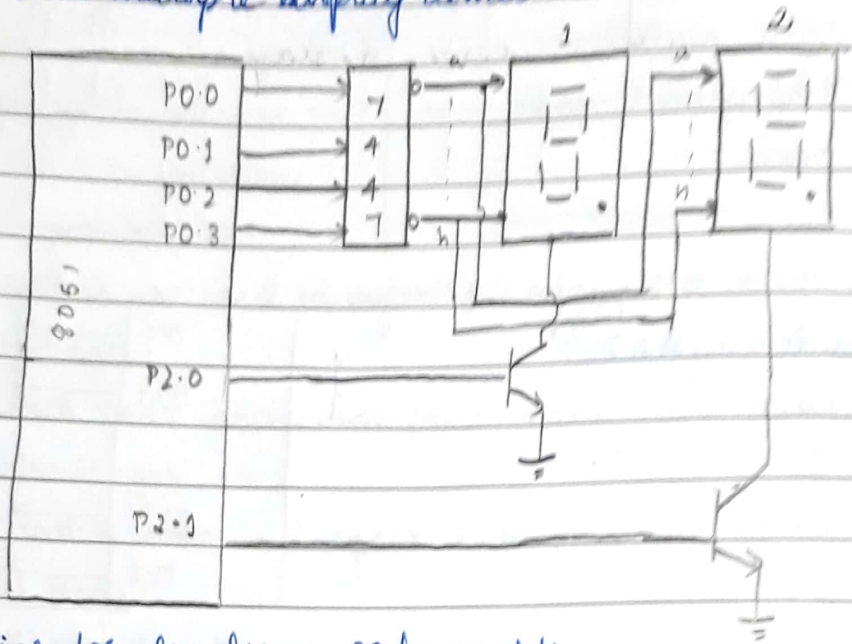
```
a=8;
b=4;
c=a+b;
d=[((c/10)<<4)|(c%10)];
P0=d;
```

Here no conversion has to be done as 7447 decoder is used.

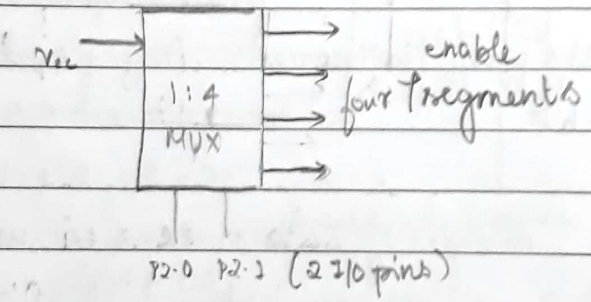




Multiplexed Parallel mode : 6 I/O pins for two 7 segments.
Single display driver is used
to drive multiple display devices



4 pins for decoder + 28 for enable
Hence 28, 7 segments can be interfaced by using transistor
4 pins for decoder + 28 for multiplexer (1:4).
Hence 28x2 = 56, 7 segments can be interfaced

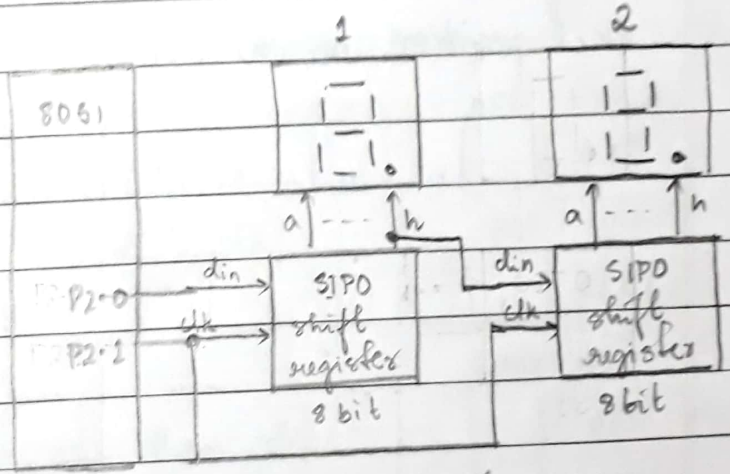



```
#include <reg51.h>
sbit fd = p2^0;  // enable first display
sbit sd = p2^1;  // enable second display
void delay();
void main()
{
  while(1)
  {
    fd = 1;  // select first display
```

```
P0 = 0x01;
delay (500); // latch delay
fd = 0;
sd = 1; // select second display
P0 = 0x02;
delay (500);
}
}
void delay (unsigned int)
{
unsigned int i;
for (i=0; i<=a; i++);
}
```

iii. = Serial Mode

```
org 0000h
MOV r2, #8
MOV a, #a4h
repeat: rrc a
jc one
clr P2.0
sjmp nxt
one: setb P2.0
nxt: setb P2.1
clr P2.1
djnz r2, repeat
```



Data is sent bit wise

1: F9h = 1111 1001

2: A4h = 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 on after the other.

1 line module

1 × 8 : 8 characters

1 × 16 : 16 characters

2 line module

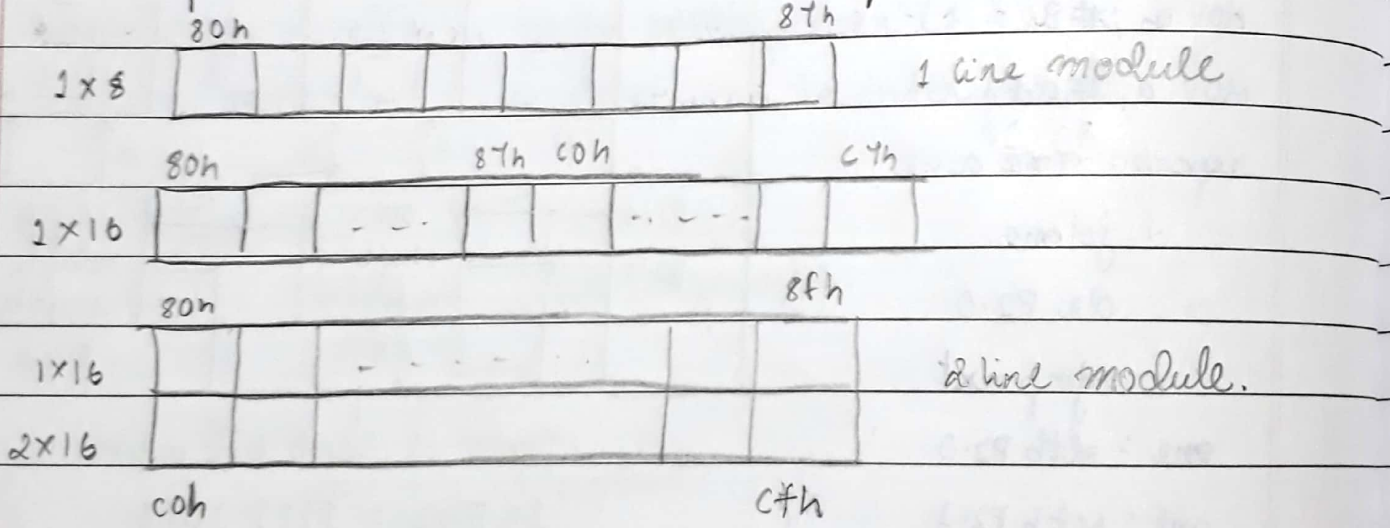
2 × 8 : 8 characters / line

2 × 16 : 16 characters / line

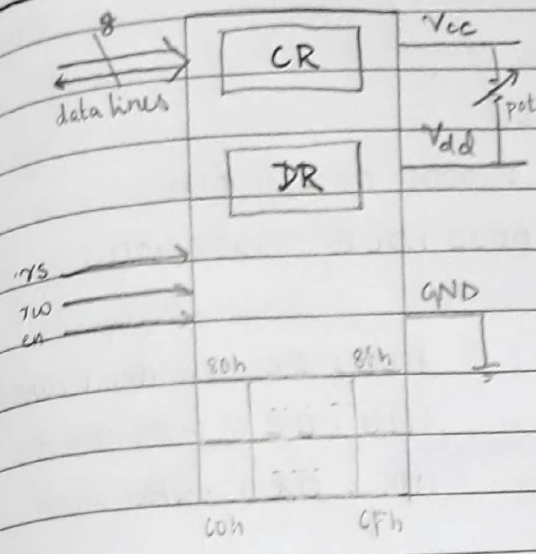
Each position on the LCD acts as a memory location.

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 data line, hence each character can be sent at a time
D_0 to D_7, where only D_7 is bidirectional and others are unidirectional. → acts as busy bit
→ It consist 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
delay (200 μsec) } to wait until we send the next data
EN = 0

Every module has certain latching time, ie, 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 f 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 (5×7)
f = 1 : 5×10
→ x x → dont cares.

2. LCD clear command : 0000 0001 : 01h

3. Display command : 0000 1DCB : 0000 1100 (optional)
→ D : display
If D = 0 : off
D = 1 : on
→ C : cursor on
If C = 0 : off
C = 1 : on
→ B : cursor on with blink
If B = 0 : without blink
B = 1 : with blink

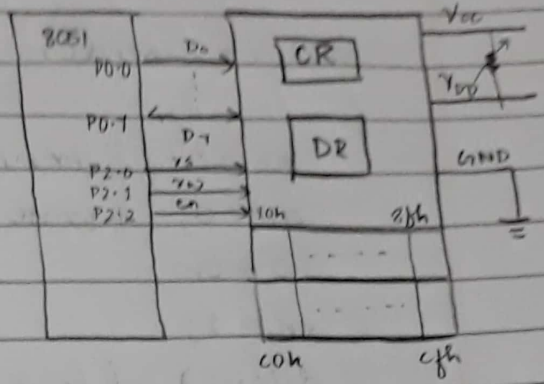
1100 : 0C : without cursor
1110 : 0E : with cursor
1101 : 0d : with blink

4. Entry mode command : 0000 01 I/D S : 0000 01 1 0
→ I/D : Increment / Decrement
If I/D = 0 : decrement
I/D = 1 : increment
→ S : Shift
If S = 0 : no shift
S = 1 : with shift

06 : without shift
07 : with shift

NOTE : To display a still 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(); ⟶ unsigned char cw, dt;
void main()
{
  lcd_init();
  while(1)
  {
    cw = 0x80; // First line 2x16   cw: command word
    cmd_send();
    dt = 'A'; // 'A' is to be displayed   dt: data
    data_send();
  }
}
void lcd_init()
{
  cw = 0x38; // function command
  cmd_send();
  cw = 0x01; // LCD clear command
  cmd_send();
  cw = 0x0C; // Display command
  cmd_send();
  cw = 0x06; // Entry mode command
  cmd_send();
}
```

```
void cmd_send()
{
    rs = 0; // command register
    rw = 0; // write
    P0 = cw;
    en = 1;
    delay(); // latching delay
    en = 0;
}
void data_send()
{
    rs = 1; // data register
    rw = 0; // write
    P0 = dt;
    en = 1;
    delay(); // latching delay
    en = 0;
}
void delay()
{
    unsigned int i;
    for(i=0; i<=500; i++);
}
```

* To display a message.

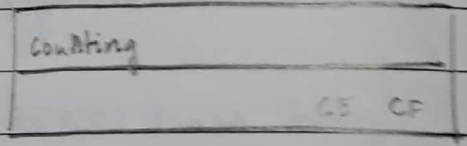
```
#include <reg51.h>
sbit rs = P2^0;
sbit rw = P2^1;
sbit en = P2^2;
unsigned char msg1[] = {" I am very happy"};
unsigned char msg2[] = {" Microcontrollers"};
void lcd_init();
```

```
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)   To display two messages.
    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);
}                          4: scroll due to decrement
void cmd_send(unsigned char a)
{
    rs=0;
    rw=0;
    P0=a;
    en=1;
    delay(500);
    en=0;
}
```

```
void data_send(unsigned char b)
{
    rs = 1;
    rw = 0;
    P0 = 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;
unsigned char msg[] = {"counting 10"};
void lcd_init();
void cmd_send(unsigned char);
void data_send(unsigned char);
void delay(unsigned int);
void main()
{
  lcd_init()
  {
    while(1)
    {
```

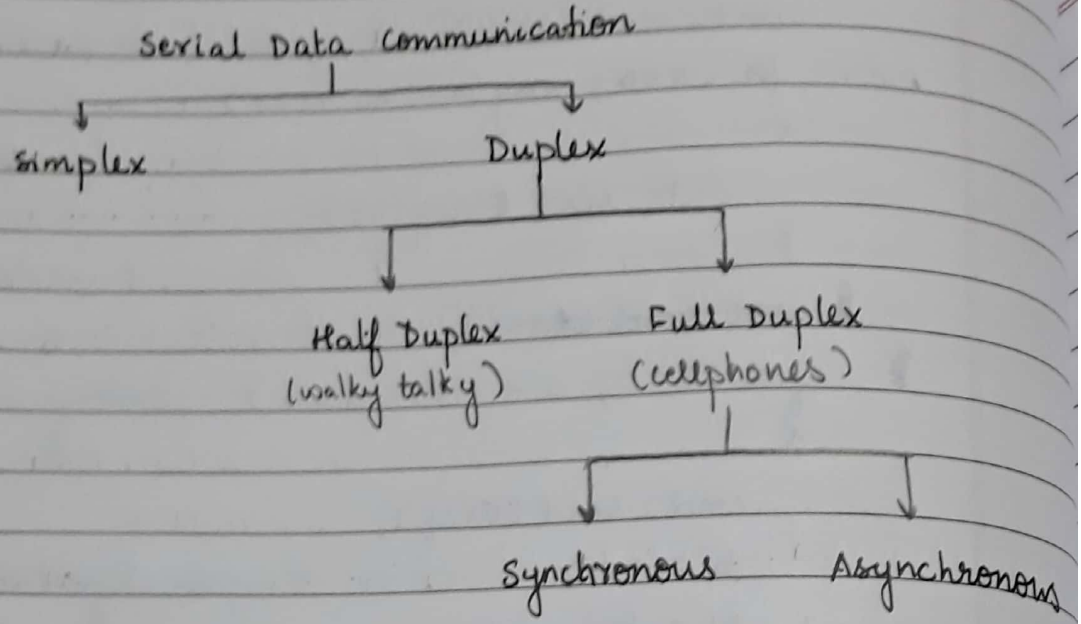


2 digit up counter in last two locations.

```
cmd_send (0x80);
for (x=0; msg[x]!=0; x++)
{
    data_send (msg[x]);
}
cmd_send (0xCF);
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	Serial Data communication
– All the bits are transfered at a time.	– Each bit is transferred at a particular time.
– Fast data communication	– slow data communication
– Hardware is more	– Hardware is less
– cost increases as it is bulky	– cost is less as it is not bulky.
– Preferred for short distance communication.	– Preferred for long distance communication.
– No modulation is required	– Modulation is required.

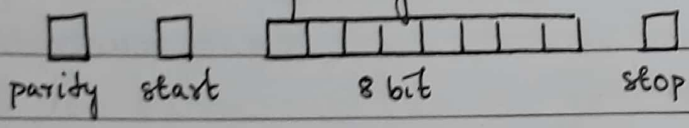


* Protocol:

procedure or algorithm or rules agreed by both reciever 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 and those who are not authorized cannot.

Framing | Deframing
Encoder | Decoder
Baudrate | bps
} Taken care by inbuilt UART and UASRT
Universal Asynchronous Synchronous Reciever and Transmitter.

UART	USART
- Data write is slow	- Data write is fast
- stop | 8 bit | start (10 bit)	- single switching bit | 8 bit (9 bit)
- clock independent	- Clock dependent.
- Data rate is variable	- Data rate is constant

Baud rate generation is taken care by UART and timer.

XTAL → ÷12 (921000Hz) → ÷30 UART (28800Hz) → Serial clk frequency

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
4800 : FAh
9600 : FDh

$256 - 24 = 232 = E8h$

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

Ex: $\frac{1}{9600} = 0.104ms$

- Other registers for serial data communication

SBUF : Serial Buffer Register
(should be sent in ASCII only)
- 8 bit register
- Special function register
- user defined

Ex: SBUF = A.

SBUF → framed → pfu serial → TXD P3.1

SBUF ← pfu serial ← Deframing UART ← RXD P3.0

2. SCON: Serial control Register

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

SM0	SM1	SM2	REN	TBS	RBS	TI	RI
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

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 8 bit UART with variable Baud rate

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

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

REN: Reciever enable : 1.

SM2 : Not useful for 8051 microcontroller : Hence 0.

In Serial mode 1

1200
2400
4800
9600
} Variable Baud Rate

8 bit + 1 start bit + 1 stop bit.

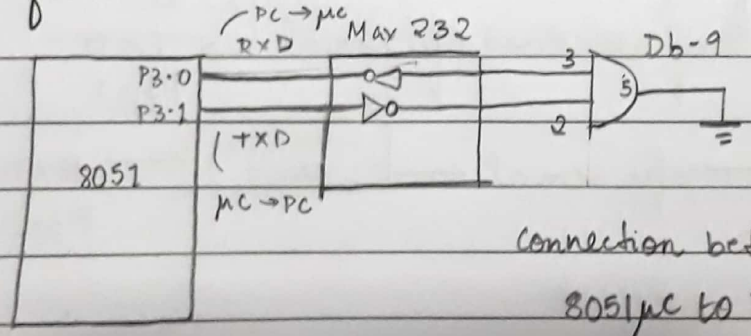
TBS and RBS : 0

TI : Transmitter flag } serial interrupt Flags.

RI : Receiver flag

Initially both are zero

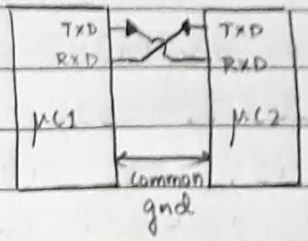
- if the SBUF data has been transmitted then TI = 1.
- if the SBUF receives data then RI = 1.

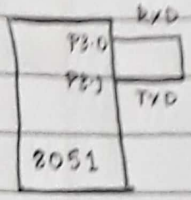


Connection between 8051 µc to PC

Transmission of data can be done from:

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





µC1 → µC1

All the connections can be realised using RS232 cables. µC1 ↔ µC2

* To transmit data from microcontroller to PC.

```
#include <reg51.h>
void main()
{
  TMOD = 0x20; // Timer 1 in mode 2 (Baud rate generation)
  TH1 = 0xfd;
  TR1 = 1; // start timer
  SCON = 0x50; // serial mode 1 with 1 start bit and 1 stop bit
  SBUF = 'A';
  while (TI == 0);
  TI = 0;
  while (1);
}
```

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

$= \frac{28800}{9600} = 3$

256 - 3 = 253 = FDh.

* To send a message from microcontroller to PC

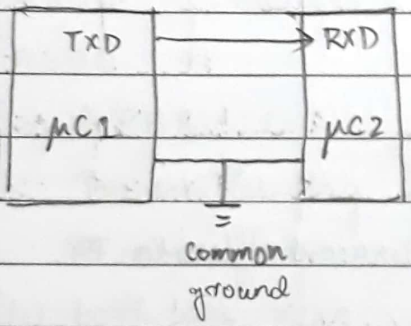
```
#include <reg51.h>
unsigned msg[] = {"My vote my choice\0"};
void main()
{
  TMOD = 0x20; // timer 1 in mode 2: for baud rate generation
  TH1 = 0xfd; // 9600 baud rate
  TR1 = 1; // start the timer for baud rate generation.
  SCON = 0x50; // 1 stop bit and 1 start bit (serial mode 1)
```

```
for (x=0; msg[x] != '\0'; x++)
{
    SBUF = msg[x];
    while (TI == 0);
    TI = 0;
    delay (50000);   for observing characterwise transmission
}
while (1);
}
```

* To transmit data from pc to microcontroller.

```
#include <reg51.h>
while (RI != 0);
x = SBUF;
P0 = x;
RI = 0;
```

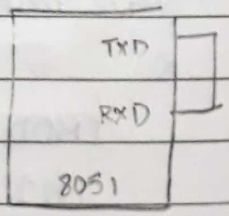
* To transmit data from one microcontroller to another



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()
{
    TMOD = 0x20;
    TH1 = 0xfd;
```



```
TR1 = 1;
SCON = 0x50;
SBUF = 'A';
while (TI==0);
TI = 0;
while (RI==0);
x = SBUF;
P0 = x;
RI = 0;
while (1);
}
```

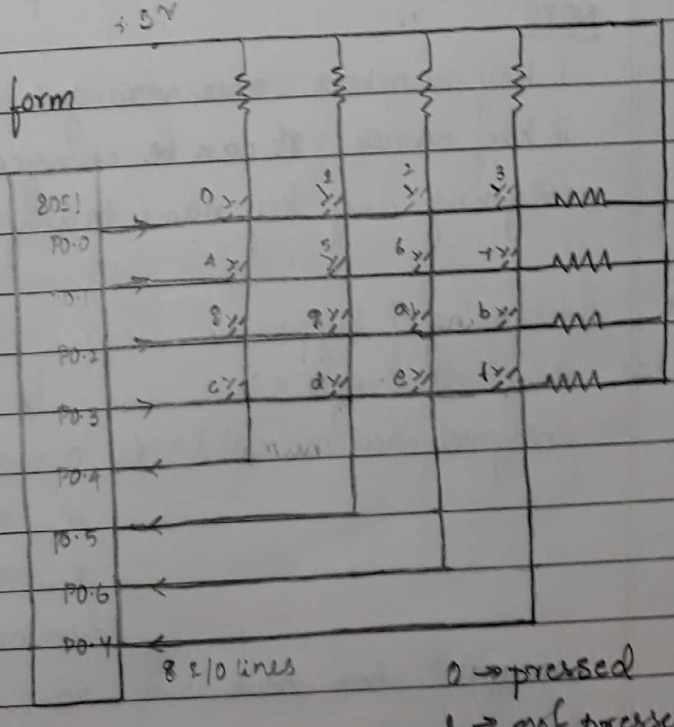
* Interfacing:

1. Keypad interfacing

keys can be arranged
- linearly / non-matrix form
- matrix form (n×n)

Applications
- key pressed number can be found out and can be displayed
- realise as calculator (+, -, ÷, ×)

It can be done by
1. 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 output 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 realised 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, 0x0a, 0x0b,
                         0x0c, 0x0d, 0x0e, 0x0f };
unsigned char check_key, row, key_no;
void main()
{
  P2 = 0xf0; // to enable all four rows at a time
```

```
do
{
  check_key = P2; // to read the switch status.
  check_key = check_key & 0xf0; // to check which key is pressed.
}while(check_key == 0xf0);
delay(500); // to eliminate key-bouncing
key_no = 0;
for(row = 0; row < 4; row++)
{
  P2 = ~(0x01 << row);
  check_key = P2 & 0xf0;
  if(check_key != 0xf0)
  {
    switch(check_key);
    {
      case 0xe0:
        P0 = nums[key_no + 0];
        break;
      case 0xd0:
        P0 = nums[key_no + 1];
        break;
      case 0xb0:
        P0 = nums[key_no + 2];
        break;
      case 0x70:
        P0 = nums[key_no + 3];
        break;
    }
  }
  key_no = key_no + 4;
}
}
```

0000 0001

1111 1110 - row 1 is enabled

0000 1000

1111 0111 - row 4 is enabled

* Calculator
1. read the 1st number and store
2. read the 2nd number and store
3. read the operator and then check
   - if (+): perform addition
   - if (-): perform substraction
   - if (x): perform multiplication
   - if (÷): perform division.

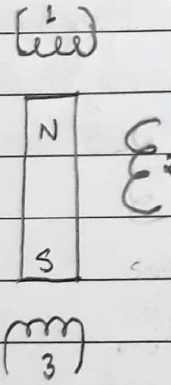
2. Motor Interfacing:

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.

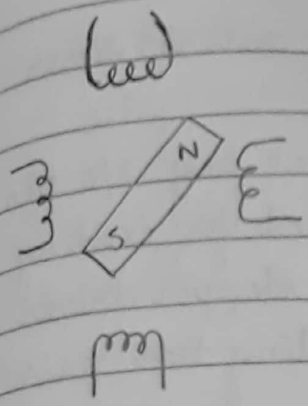
1. Stepper motor:



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°. Excitation should be done in sequence only.



b. Double coil Excitation / Half step

Two coils are excited at a time

Here the step size is 0.9°.

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

For single coil excitation

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

For double coil excitation

$= \frac{360^\circ}{0.9^\circ} = 400$ 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 8 bit register is used, the code is 8 bit.

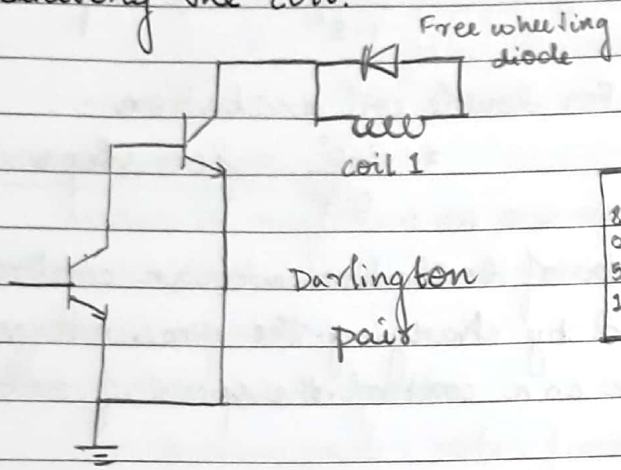
virtual physical

88	1000	1000	Single coil Excitation code
44	0100	0100	
22	0010	0010	
11	0001	0001	
CC	1100	1100	Double coil Excitation code
66	0110	0110	
33	0011	0011	
99	1001	1001	

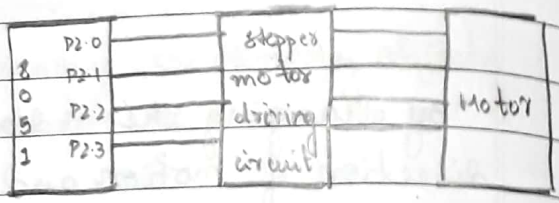
For 100 steps we need to excite 4 coils 25 times

$4 \times 25 = 100$

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.



Interfacing motor



* To rotate in clockwise direction continuously

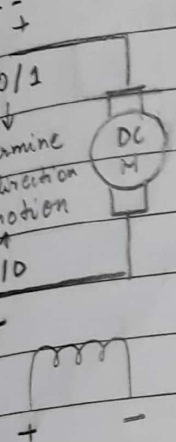
```
#include <reg51.h>
void delay (unsigned int j)
{
  unsigned int i;
  for (i=0; i<=j; i++);
}
void main()
{
  while (1)
  {
    P0 = 0x88;
    delay (500);
    P0 = 0x44;
    delay (600);
```

```
#include <reg51.h>
void delay(unsigned int j)
{
  unsigned int i;
  for (i=0; i<=j; i++)
}
void main()
{
  x = 0x01;
  while (1)
  {
    for (y=0; y<=25; y++)
    {
      P0 = x
```

```
P0 = 0x22;
delay (500);
P0 = 0x11;
delay (500);
}
}
```

```
delay (500);
P0 = x << 1;
}
}
}
```

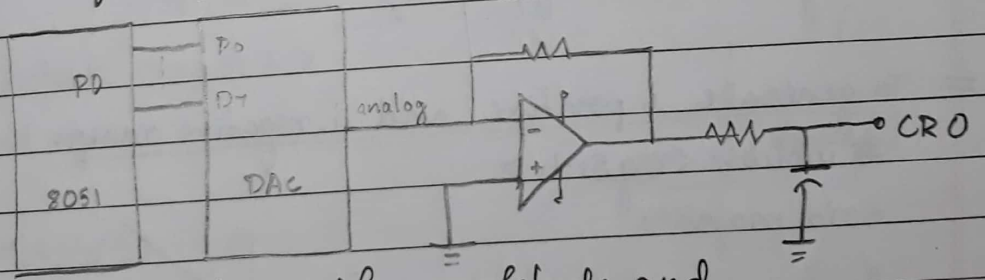
2. DC Motor:



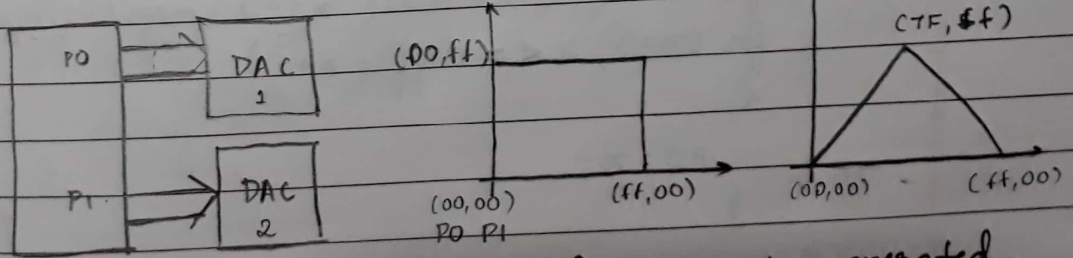
It has only one coil. The speed of a DC motor is controlled by PWM (Pulse Width Modulation).

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 → ffh = 255

1V → $\frac{255}{5}$ = 51 = 33h

= To generate square wave : A = 5V

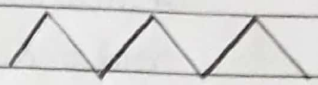
```
#include <reg51.h>
void main()
{ while(1)
  { P0 = 0xFF; // On : high
    delay(500);
    P0 = 0x00; // Off : low
    delay(500);
  }
}
void delay(unsigned int j)
{
  unsigned int i;
  for(i=0; i<=j; i++);
}
```

0x99; 3V amplitude

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

```
#include <reg51.h>
void main()
{
  while(1)
  { for(x=0xff; x>=0x00; x--) : negative ramp
    for(x=0; x<=0xff; x++)
    {
      P0 = x;
    }
  }
}
```

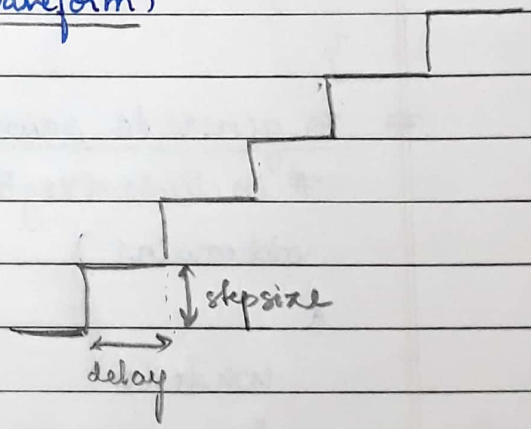
= To generate triangular waveform:



```
#include <reg51.h>
void main()
{ unsigned int x, y;
  while(1)
  {
    for (x=0x00; x<=0xff; x++)
    {
      P0 = x;
    }
    for (y=0xff; y>=0x00; y--)
    {
      P0 = y;
    }
  }
}
```

= To generate step (positive staircase waveform) (negative)

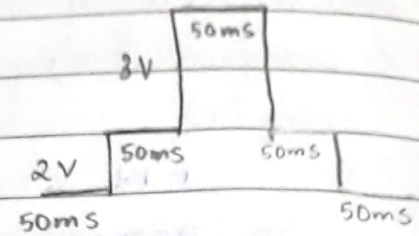
- $\text{step size} = \frac{255}{\text{no of step}} = \frac{255}{5} = 33h$




```
#include <reg51.h>
void main()
{
  while(1)
  {
    for (x=0; x<=0xff; x=x+0x33)
    {
      P0 = x;
      delay();
    }
  }
}
```

= To generate combination of steps:

```
#include <reg51.h>
void main()
{
  while(1)
  {
    P0 = 0x00;
    delay(50ms);
    P0 = 0x66;
    delay(50ms);
    P0 = 0x99;
    delay(50ms);
    P0 = 0x66;
    delay(50ms);
    P0 = 0x00;
    delay(50ms);
  }
}
```

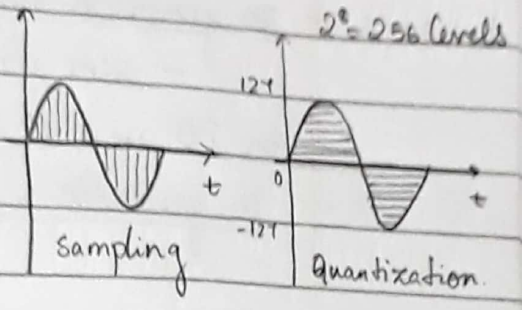


= To generate sawtooth waveform:

```
#include <reg51.h>
void main()
{
  while(1)
  {
    for(x=0; x<=0xff; x++)
    {
      P0 = x;
    } P0 = 0x00;
    delay();
  }
}
```

To generate sine wave:

Sine wave can be generated by
i. Sampling and
ii. Quantization.



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

θ varies from 0 to 360°.

varying θ by an interval of 5°.

$\therefore \frac{360^\circ}{5^\circ}$ = 72 values

```
#include <reg51.h>
unsigned char nums[72] = {"....."};
void main()
{
  while(1)
  {
    for(x=0; x<=72; x++)
    {
      P0 = nums[x];
    }
  }
}
```

When the values are calculated manually

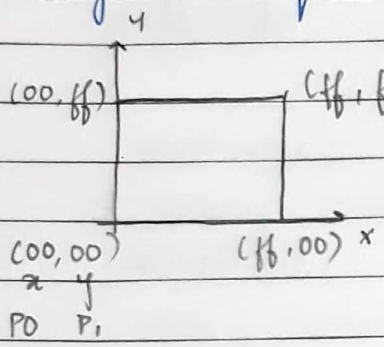
```
#include <reg51.h>
#include <math.h>
void main()
{
  while(1)
  {
    for(x=0; x<=72; x++)
    {
      y = 127 + 127 sin( x*22 / 7*180 );
      P0 = y; }}}
```

When the values are not calculated manually

— To generate circle

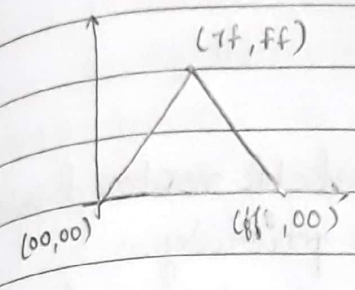
- sine wave : send the positive values (0 to 180°) in P0 and the negative values (180° to 360°) in P1.

— To generate square

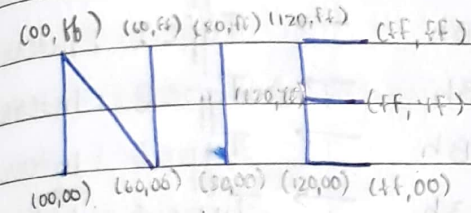



```
        org 0000h
        MOV a, #00
        MOV b, #00
cnt:    MOV P0, a
        MOV P1, b
        inc b
        cjne b, #00, cnt
cnt1:   MOV a, #00
        MOV b, #0ffh
        MOV P0, a
        MOV P1, b
        inc a
        cjne a, #00, cnt 1
cnt2:   MOV a, #0ffh
        MOV b, #0ffh
        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



Q5 * SEE (15 marks)

INTERRUPTS:

ISR : Interrupt Service Routine.

Interrupt can be by

i. Hardware : External interrupts.

ii. Software : Internal interrupts.

Maskable Interrupts : These interrupts can be ignored

Non-maskable 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 interrupts is written.

* 8051 Interrupts:

5 interrupts + reset

(2 external interrupts + 3 internal interrupts)

- P3.2: $\overline{INT0}$: ext 0 — By default : level trigger (upon reset)
- P3.3: $\overline{INT1}$: 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 interrupts. Vector address is based on the priority.
When there are multiple interrupts, the interrupts are served based on the default priority.

Interrupt structures

1. $\overline{INT0}$ → 0003h → Type 0 : Interrupt 0
2. T0 → 000Bh → Type 1 : Interrupt 1
3. $\overline{INT1}$ → 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	TR1	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	ET0	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

ET0 : 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	PX1	PT0	PX0

PT2 : timer 2 : 0
Here 0 - not assign
1 - assign
when assigned 1, then that interrupt becomes first and comes on top.

PS : Priority for serial interrupts.
PT1 : Priority for timer-1 interrupt
PX1 : 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
ljmp ext0_isr
~~ljmp ext0_isr~~
org 0000h
MOV ie, #81h
wait : sjmp wait // wait without doing any work
org 0300h
ext0_isr : setb P1.3
           acall delay
           clr P1.3
           acall delay
           reti   // return from interrupt
```

IE

EA	-	ET2	ES	ET1	EX1	ET0	EX0
1	0	0	0	0	0	0	1

81h

or: wait by doing other work

```
org 0000h
MOV ie, #81h
MOV a, #0
cnt : MOV P0, a
inc a
cjne a, #0ah, cnt
```

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, #~~0ffh~~ 10
up2 : MOV r3, #0ffh
```

```
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
{
  for (x=0; x<50; x++)
  {
     led = 1;
     delay();
     led = 0;
     delay();
  }
}
void main()
{ while(1)
  { IE = 0x81; // enabling EX0.
    while(1); // 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++)
    {
      P0=y;
      delay();
    }
  }
}
```