Course Coordinator: Prof. Mahesh P.Yanagimath

Module 1: 8051 Microcontroller Basics

Inside the computer

Computer: A computer is a multipurpose programmable machine that reads binary instructions from its memory, accepts binary data as input, processes the data according to those instructions and provides results as output. It is a programmable device made up of both hardware and software. The various components of the computer are called hardware. A set of instructions written for the computer to solve a specific task is called program and collection of programs is called software.

One of the most important feature of a computer is its memory. Common terms used to describe amount of memory used are bit, byte, Nibble etc.

Bit: Its a binary digit that can have only 0 or 1.

Nibble: Half byte: 4 bits ----0000

Byte: 8 bits--- 0000 0000

word: 2 byte---16 bits---0000 0000 0000 0000

above all are composed of combination of 0 and 1. Terms used to describe amount of memory used in IBM PC's and compatibles is given below.

Kilobyte(K)---2^{10} bytes ---1024 bytes

Megabyte(M)--- 2^{20} bytes

Gigabytes(G)---2^{30} bytes

Terabytes(T)----2^{40} bytes

For example if computer has 16 megabytes of memory i.e 16x 2^{20} =2^{24} i.e it is having 24 addressable lines with 16 megabytes of memory. Each location can have a maximum of 1 byte of data.

Internal Organization of Computers

A computer that is designed using a microprocessor as its CPU, is known as a microcomputer. The computer hardware consists of four main components. The central processing unit which acts as computer's brain. Input unit through which program and data can be entered to computer, output unit on which the results of the computations can be displayed. Memory in which data and program are stored.

It is having two types of memory. 1) RAM 2) ROM

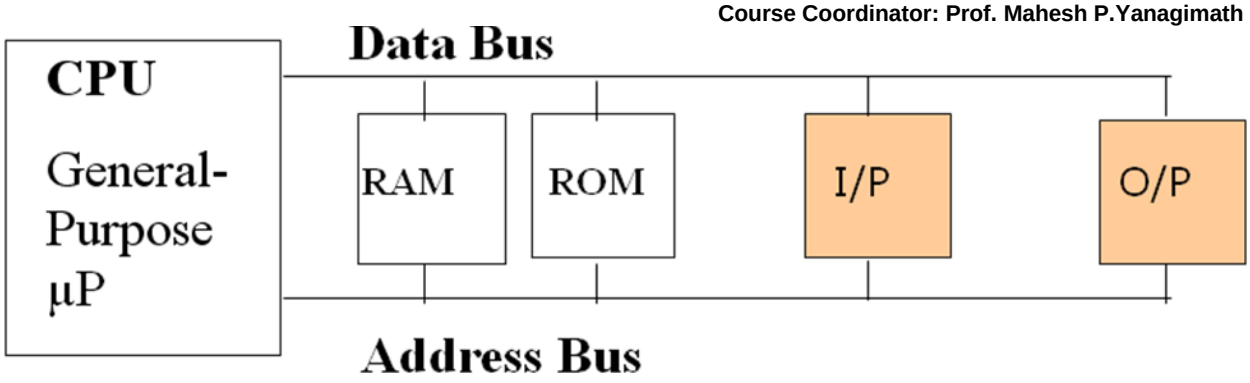


Fig 1. Block diagram of a microcomputer

CPU and its relation to RAM and ROM

CPU to process information data must be stored in RAM or ROM. RAM and ROM are called primary memory and disks are called secondary memory.

Inside the CPU

Program stored in memory provides instructions to CPU to perform. The function is to fetch instructions from memory and execute them. CPU registers stores information temporarily. Registers inside the CPU may be 8 bit, 16 bit,32 bit, 64 bit etc. depending on CPU. If bigger registers are used then it is good for CPU but cost will increase. CPU has ALU which is responsible for performing arithmetic operations like add, subtract and logical functions like AND, OR and NOT. Program counter always points to address of next instruction to be executed. After the instruction execution program counter will be automatically incremented to point next instruction. Instruction decoder interprets the instruction fetched into CPU.

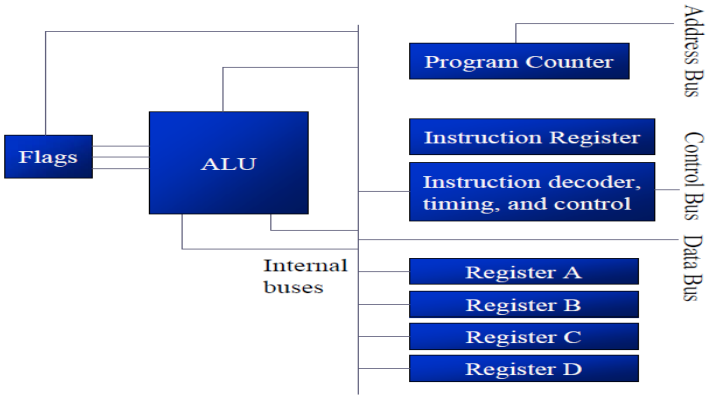


Fig 2: Internal block diagram of CPU

Microprocessors

A microprocessor is a general purpose digital computer central processing unit (CPU). Also known as a 'Computer on Chip'. Block diagram of a Microprocessor CPU which contains ALU, Program counter (PC), a stack pointer (SP), some working registers, a clock timing circuit and interrupt circuit s is shown in the following figure.

To make a computer, microcontroller one must add memory usually RAM and ROM, memory decoders, an oscillator and a number of Input, Output devices such as serial and parallel ports. In addition special purpose devices such as interrupt handler and counters may be added to relieve the CPU from time consuming counting or timing cores.

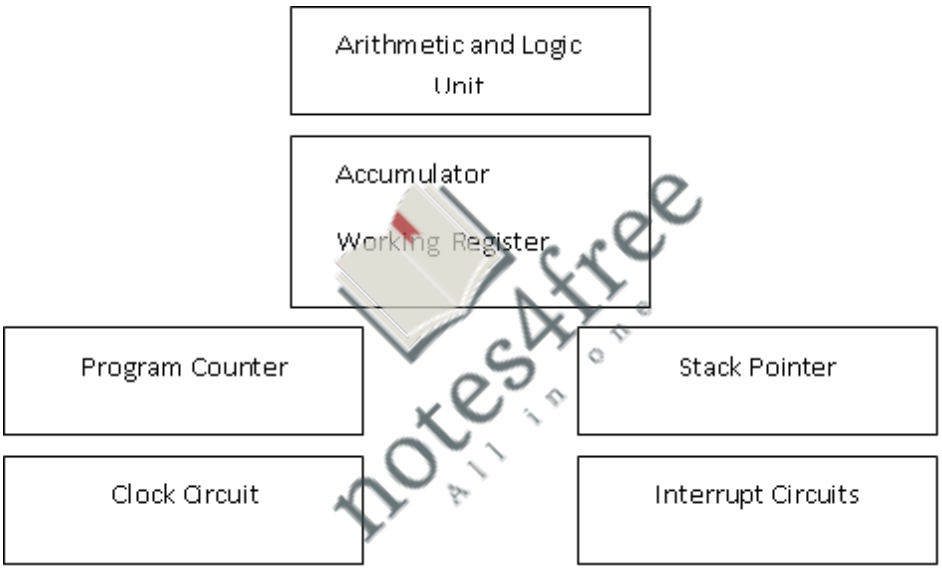


Fig.3.Block Diagram of a Microprocessor

The hardware design of a microprocessor is arranged such that a very small or very large system can be configured around the CPU as the application demands. The prime use of the Microprocessor is to read data, perform extensive calculations on that data, and store those calculations in a mass storage device or display the results for human use. The programs used by microprocessor are stored in the mass storage device and loaded into RAM as user directs. A few microprocessor programs are stored in ROM. The ROM based programs are primarily small fixed programs that operate peripherals and other fixed devices that are connected to the system.

Microcontrollers and Embedded Processors

Microcontroller: A Microcontroller is a programmable digital processor with necessary peripherals. Both microcontrollers and microprocessors are complex sequential digital circuits meant to carry out job according to the program / instructions. The design incorporates all the features found in microprocessor CPU, ALU, PC, SP and registers. It also has other features needed to make a complete computer. ROM, RAM, Parallel I/O, serial I/O, Counters and clock circuits. Like the microprocessor, a microcontroller is a general purpose device, but one that is meant to read data, perform limited calculations on that data.

The prime use of microcontroller is to control the operation of a machine using a fixed program that is stored in ROM and that does not change over the lifetime of the system.

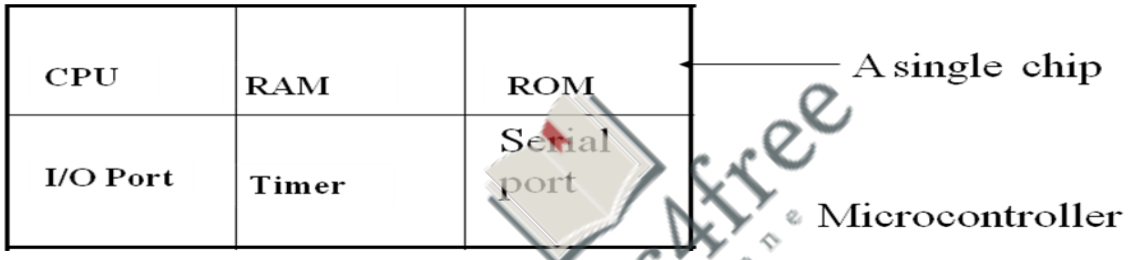


Fig4. Block diagram of a single chip computer

Comparison between Microprocessor and Microcontroller

Microprocessor	Microcontroller
1) It contains ALU, General purpose registers, stack pointer, program counter, interrupt circuits etc.	1) It contains microprocessors and in addition it has built in ROM, RAM, I/O devices, timers and counters.
2) Requires more hardware, increases in PCB	2) Requires less hardware, reduced PCB size.
3) Access time for memory and I/O devices are more	3) Access time is less due to in built memory and I/O port
4) Many instructions to move data between memory and CPU	4) One or more instructions to move data between memory and CPU
5) More flexible from design point of view	5) Less flexible
6) Single memory map for data and code	6) Separate memory map for data and code.
7) Few pins are multifunctional	7) More pins are multifunctional
8) One or more bit handling instructions are	8) More number of bit handling instructions are

available	available.

Criteria for choosing a Microcontroller

There are wide varieties of Microcontrollers available in the market. Program written for one Microcontroller will not run others. Choice of the microcontroller is based on three parameters.

1. It must perform the required task efficiently and effectively.
   a) Speed, Amount of RAM and ROM on chip, Power consumption
   b) Number of I/O pins and timer on chip
   c) Cost per unit, Ease of upgrading
   d) Packaging-No of Pins and Packaging formats
2. Availability of software development tools such as compilers and assemblers, debuggers.
3. Availability of reliable source for the microcontroller.

Other members of 8051 family

There are two other members in 8051 family of Microcontroller. They are 8052 and 8031. 8052 has all standard features of 8051 as well as an extra 128 bytes of RAM and an extra timer. 8052 has 256 bytes of RAM and 3 timers. It also has 8K bytes of on chip program instead of 4K bytes. 8031 Microcontroller is often referred as a ROM less 8051 since it has 0K bytes of on chip ROM. To use this chip we must add external ROM to it. Two ports will be lost in the process of adding external ROM.

Comparison of 8051 family members

Feature	**8051**	**8052**	**8031**
ROM [on chip program space in bytes]	4K	8K	0K
RAM [bytes]	128	256	128
Timers	2	3	2
I/O pins	32	32	32
Serial port	1	1	1
Interrupt sources	6	8	6

Block diagram of 8051

8051 is the original member of 8051 family. Features of 8051 microcontroller are as shown below.

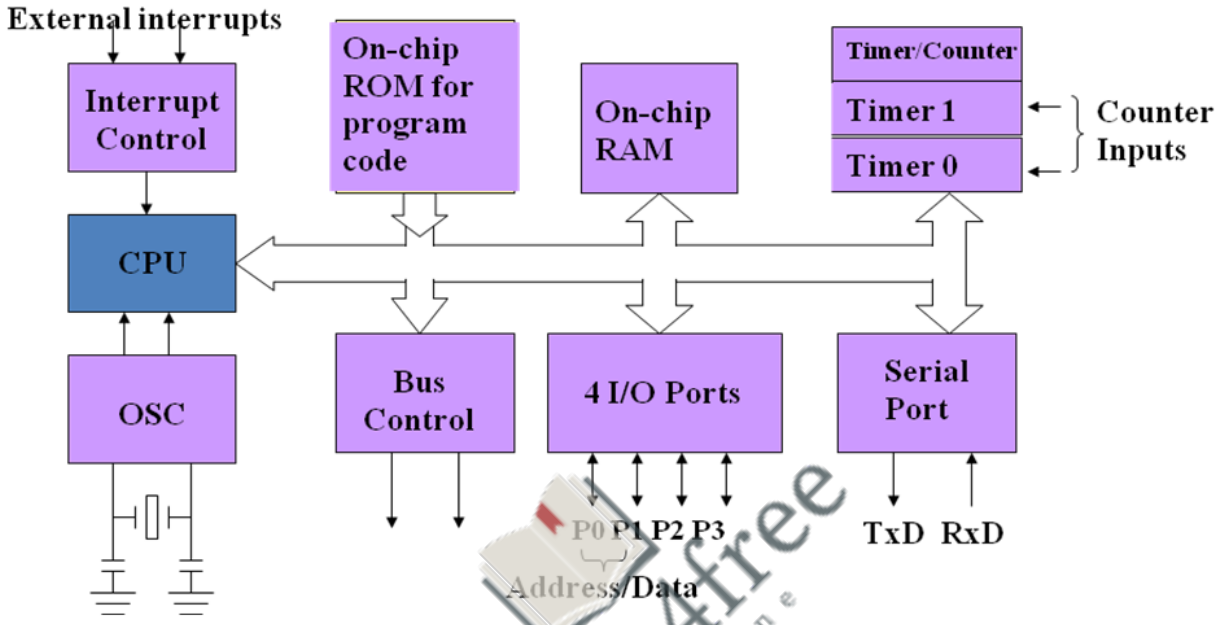


Fig 5. Architecture of 8051 Microcontroller

Salient Features

- •Eight bit CPU with registers A (Accumulator) and B
- •Sixteen bit Program counter (PC) and a data pointer (DPTR)
- •8 Bit Program Status Word (PSW), 8 Bit Stack Pointer, 4K Code Memory
- •Internal Memory of 128 Bytes, 32 I/O Pins arranged as 4 , 8 Bit ports
- •Two 16 Bit Timer/Counter :T0, T1
- •Full Duplex serial data receiver/transmitter
- •Control Registers : TCON,TMOD,SCON,PCON,IP and IE
- •Two External and Internal Interrupt sources
- •Oscillator and clock circuits.

The programming model of 8051 shows the 8051 as the collection of 8 and 16 bit registers and 8 bit memory locations. These registers and memory locations can be made to operate using software instructions that are incorporated as part of the program instructions.

Program Counter: It addresses program instruction bytes which are to be fetched from memory locations.

Data pointer: It contains two registers DPG and DPL. It is used to access internal or external data. It is under the control of program instructions and can be specified by 16 bit name.

8051 Clock and Instruction Cycle:

The heart of 8051 is the circuitry that generates the clock pulses by which all internal operations are synchronized. Pins XTAL1 and XTAL2 are provided for connecting resonator to form an oscillator. The crystal frequency is the basic internal frequency of the microcontroller. 8051 is designed to operate between 1MHz to 16MHz and generally operates with a crystal frequency 11.04962 MHz.

The oscillator formed by the crystal, capacitor and an on-chip inverter generates a pulse train at the frequency of the crystal. The clock frequency establishes the smallest interval to accomplish any simple instruction. The time taken to complete any instruction is called as machine cycle or instruction cycle. In 8051 one instruction cycle consists of 6 states or 12 clock cycles, instruction cycle is also referred as Machine cycle.

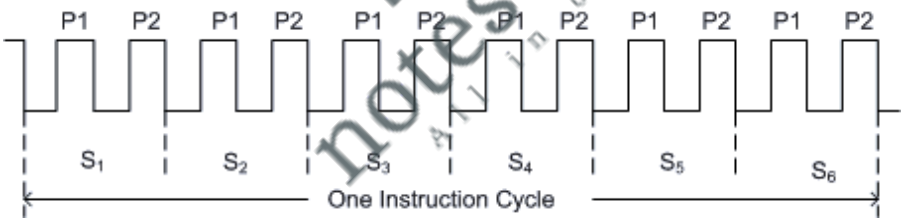


Fig. 6 Instruction cycle of 8051(Instruction cycle has six states (S_1 - S_6). Each state has two pulses (P1 and P2))

Course Coordinator: Prof. Mahesh P.Yanagimath

PSW and flag bits

Program Status word

It is an 8 bit register. Only 6 bits of it are used by 8051. Bits of PSW are shown below.

PSW : Program Status Word (Bit Addressable)

CY	AC	F0	RS1	RS0	OV	F1	P
psw.7	psw.6	psw.5	psw.4	psw.3	psw.2	psw.1	psw.0

CY	PSW.7	Carry Flag.
AC	PSW.6	Auxiliary Carry Flag.
F0	PSW.5	Flag 0 available to the user for general purpose.
RS1	PSW.4	Register Bank selector bit 1 (SEE NOTE).
RS0	PSW.3	Register Bank selector bit 0 (SEE NOTE).
OV	PSW.2	Overflow Flag.
F1	PSW.1	Flag F1 available to the user for general purpose.
P	PSW.0	Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/even number of "1" bits in the accumulator.

Note :

The value presented by RS0 and RS1 selects the corresponding register bank.

RS1	RS0	REGISTER BANK	ADDRESS
0	0	0	00H–07H
0	1	1	08H–0FH
1	0	2	10H–17H
1	1	3	18H–1FH

Fig 7: Program Status word

Carry Flag(C): This flag sets when there is a carry out from D7 bit. This flag bit is affected after 8 bit addition or subtraction. It can also be set to 0 or 1 directly by instruction such as SETB C and CLR C.

Auxiliary Carry Flag(AC): If there is a carry out from D3 to D4 during an ADD or SUB operation then this bit sets. Otherwise it is

8051 Register banks and Stacks

The collection of general purpose registers (R0-R7) is called as register banks, which accept one byte of data. The register bank is a part of the RAM memory in the microcontrollers, and it is used to store the program instructions.

A total of 32 bytes of RAM are set aside for the register banks and stack. These 32 bytes are divided into 4 banks of registers in which each bank has 8 registers, RO – R7. RAM locations from 0 to 7 are set aside for bank 0 of RO – R7 where RO is RAM location 0, Rl is RAM location 1, R2 is location 2, and so on, until memory location 7, which belongs to R7 of bank 0. The second bank of registers RO – R7 starts at RAM location 08H and goes upto location OFH. The third bank of RO – R7 starts at memory location 10H and goes upto location 17H. Finally, RAM locations 18H to 1FH are set aside for the fourth bank of RO – R7. The following figure shows how the 32 bytes are allocated into 4 banks:

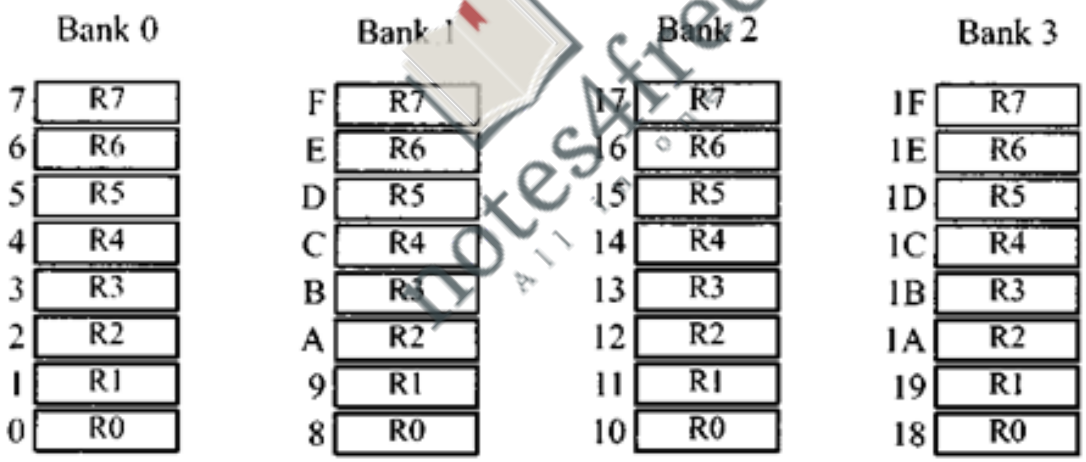


Fig 8. 8051 Register Banks and their RAM Addresses

As shown in above Figure, bank 1 uses the same RAM space as the stack. This is a major problem in programming the 8051. We must either not use register bank 1, or allocate another area of RAM for the stack.

Ex: MOV R0, #99H; /* Load R0 with 99H, RAM location 0H has the value 99H*/

Default register bank 0 is accessed when programming 8051. We can switch to other banks by the use of PSW (program status word) register. Bits D4 and D3 of the PSW are used to select the desired register bank.

The Stack and Stack pointer:

The stack refers to an area of internal RAM that is used in conjunction with certain opcodes to store and retrieve data quickly. The 8 bit Stack Pointer (SP) register is used by the 8051 to hold internal RAM address that is called the top of the stack.

When data is to be placed on the stack , the SP increments before storing data on the stack so that the stack grows up as data is stored. Whenever data is retrieved from the stack, the byte is read from the stack and then the SP decrements to point to the next available byte of stored data.

Operation of the Stack and Stack Pointer:

Operation of the stack is as shown below. The SP is set to 07 when the 8051 is reset and can be changed to any internal RAM address by the programmer. The stack is limited in height to the size of internal RAM. The stack can overwrite valuable data in register banks, bit addressable RAM and scratched pad RAM areas. It is programmer's responsibility to make it sure that the stack does not grow beyond predefined bounds. The stack is normally placed high in the internal RAM by an appropriate choice of the number placed in SP register, to avoid conflict with registers or RAM.

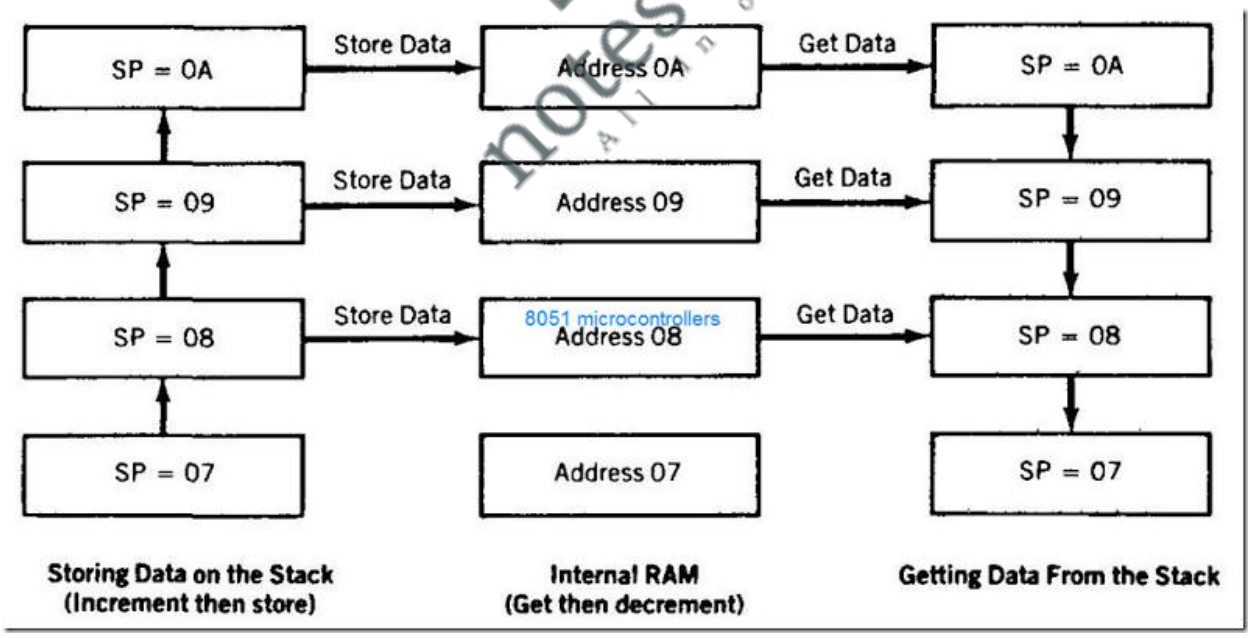


Fig 9: Stack operation

Course Coordinator: Prof. Mahesh P.Yanagimath

Internal Memory organization of 8051

A functioning computer memory for program code bytes, commonly in ROM, and RAM memory for variable data that can be altered as the program runs. Additional memory can be added externally using suitable circuits.

The 8051 has Harvard architecture which uses the same address in different memories for code and data The internal circuitry accesses the current memory based on the nature of operation in the program.

Internal RAM: The 128 bytes internal RAM is organized into 3 distinct areas.

1. 32 bytes from address 00h to 1fh that make up 32 working registers organized as 4 memory banks of 8 registers each. The 4 register banks are numbered 0 to 3 and are made up of 8 registers named R0 to R7. Each register can be addressed by name or by its RAM addresses. Thus R0 of bank3 is R0 (if bank3 is selected) or address 18h (where bank3 is selected) . Bits RS0 and RS1 in the PSW determine which bank of registers is currently in use at any time when program is running. Register banks not selected can be used as general purpose RAM. Bank0 is selected by default on reset..
2. A bit addressable area of 16 bytes occupies RAM byte addresses 20h to 2fh, forming total of 128 bits. An addressable bit may be specified by its bit address of 00h to 7fh or 8 bits may form any byte address from 20h to 2fh.For example bit address 4fh is also bit 7 of byte address 29h. Addressable bits are useful when the program need only remember a binary event.
3. A general purpose RAM area above the bit area from 30h to 7f h, addressable as byte.

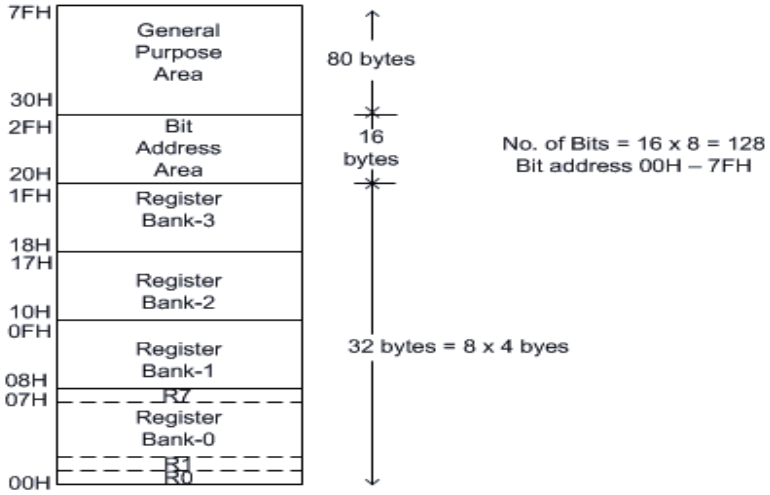


Fig.10. Internal RAM structure

Internal ROM

8051 is organized so that data memory and program code memory can be two entirely different physical memory entities. Each has the same address ranges. The internal program ROM occupies code address space 000h to 0fffh. The PC is normally used to address program code bytes from address 0000h to ffffh. Program addresses higher than offfh which exceed the internal ROM capacity will cause the 8051 to automatically fetch code bytes from external memory, addresses 00h to ffffh by connecting the external access pin (EA) to ground. 8051 microcontroller supports 4K bytes internal ROM. Program addresses higher than 0FFFH which excedes internal ROM capacity will cause 8051 to automatically fetch code bytes from external program memory.

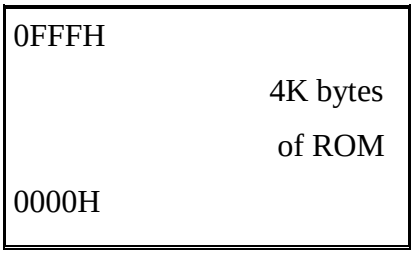


Fig 11: Internal ROM

External Memory

8051 supports external memory of 64K bytes of data memory and 64K bytes of program memory. External memory is interfaced and accessed by using 16 address lines available on port 0 and port 2. Address and data lines are multiplexed on port 0.

Types of Special Function Registers (SFRs)

The 8051 operations that do not use the internal RAM addresses from 00h to 7fh are done by a group of specific internal registers each called a specific function register (SFR) which may be addressed much like internal RAM using addresses from 80h to ffh. Some SFRs are also bit addressable as is the case for the bit area of RAM.

SFR Map: The set of Special Function Registers (SFRs) contain important registers such as Accumulator, Register B, I/O Port latch registers, Stack pointer, Data Pointer, Processor Status Word (PSW) and various control registers. Some of these registers are bit addressable. The detailed map of various registers is shown in the following figure.

Symbol	Name	Address (Hex)	Remarks
ACC	Accumulator	0E0	Bit addressable
B	B Register	0F0	Bit addressable
PSW	Program Status Word	0D0	Bit addressable
SP	Stack Pointer	81	
DPTR	Data Pointer 2 Bytes		
DPL	Low Byte	82	
DPH	High Byte	83	
P0	Port 0	80	Bit addressable
P1	Port 1	90	Bit addressable
P2	Port 2	0A0	Bit addressable
P3	Port 3	0B0	Bit addressable
IP	Interrupt Priority Control	0B8	Bit addressable
IE	Interrupt Enable Control	0A8	Bit addressable
TMOD	Timer/Counter Mode Control	89	
TCON	Timer/ Counter Control	88	Bit addressable
T2CON	Timer/ Counter 2 Control	0C8	Bit addressable, 8052 only
TH0	Timer/ Counter 0 High Byte	8C	
TL0	Timer/ Counter 0 Low Byte	8A	
TH1	Timer/ Counter 1 High Byte	8D	
TL1	Timer/ Counter 1 Low Byte	8B	
TH2	Timer/ Counter 2 High Byte	0CD	8052 only
TL2	Timer/Counter 2 Low Byte	0CC	8052 only
RCAP2H	T/C 2 Capture Reg. High Byte	0CB	8052 only
RCAP2L	T/C 2 Capture Reg. Low Byte	0CA	8052 only
SCON	Serial Control	98	Bit addressable
SBUF	Serial Data Buffer	99	
PCON	Power Control	87	

Fig 12: SFR Map

Pin configuration of 8051

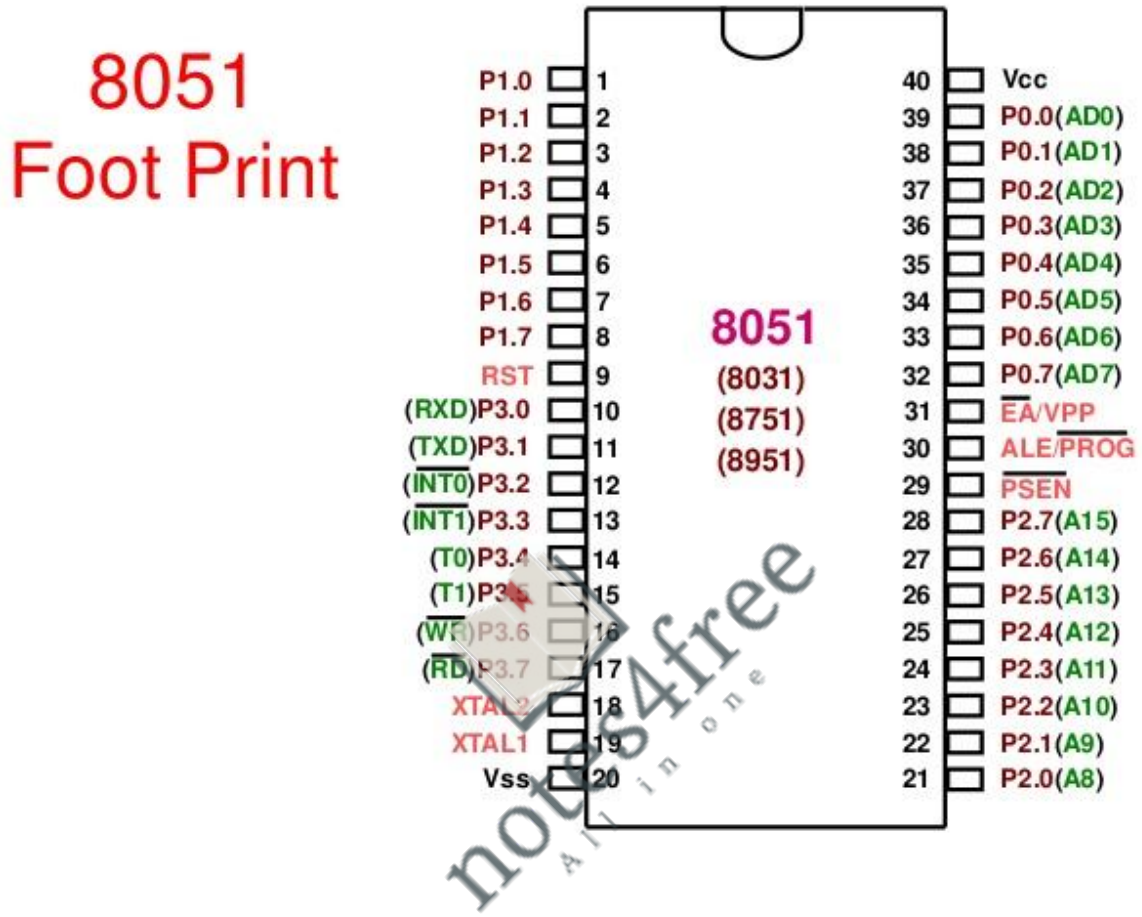


Fig 13. Pin configuration of 8051

Pin Description

- **Pins 1 to 8** − These pins are known as Port 1. It is a general purpose port. This port doesn't serve any other functions. It is internally pulled up, bi-directional I/O port.
- **Pin 9** − It is a RESET(RST) pin, which is used to reset the microcontroller to its initial values.
- **Pins 10 to 17** − These pins are known as Port 3. This port serves some functions like interrupts, timer input, control signals, serial communication signals RxD and TxD, etc.
- **Pins 18 & 19** − These pins are used for interfacing an external crystal to get the system clock.
- **Pin 20** − This pin provides the power supply to the circuit.

- **Pins 21 to 28** − These pins are known as Port 2. It serves as I/O port. Higher order address bus signals are also multiplexed using this port.
- **Pin 29** − This is PSEN pin which stands for Program Store Enable. It is used to read a signal from the external program memory.
- **Pin 30** − This is EA pin which stands for External Access input. It is used to enable/disable the external memory interfacing.
- **Pin 31** − This is ALE pin which stands for Address Latch Enable. It is used to demultiplex the address-data signal of port.
- **Pins 32 to 39** − These pins are known as Port 0. It serves as I/O port. Lower order address and data bus signals are multiplexed using this port.
- **Pin 40** − This pin is used to provide power supply to the circuit.

IO Port Usage in 8051

The four 8-bit I/O ports P0, P1, P2 and P3 each uses 8 pins. All the ports upon RESET are configured as input, ready to be used as input ports. When the first 0 is written to a port, it becomes an output port. To reconfigure it as an input, 1 must be sent to the port. To use any of these ports as an input port, it must be programmed. It can be used for input or output, each pin must be connected externally to a 10K ohm pull-up resistor. This is due to the fact that P0 is an open drain, unlike P1, P2, and P3. Open drain is a term used for MOS chips in the same way that open collector is used for TTL chips.

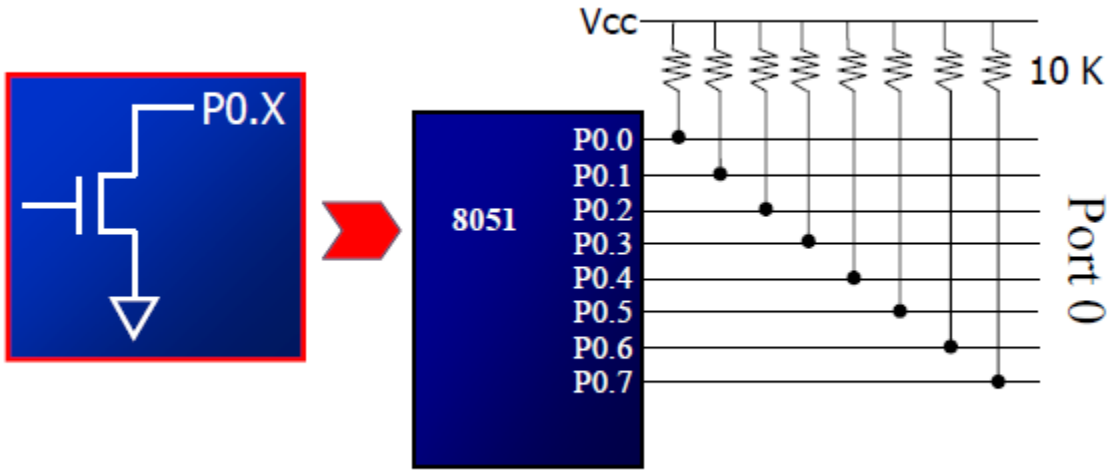


Fig 14: Port 0 with Pull up registers

The following code will continuously send out to port 0 the alternating value 55H and AAH

```
BACK: MOV A,#55H
      MOV P0,A
      ACALL DELAY
      MOV A,#0AAH
      MOV P0,A
      ACALL DELAY
      SJMP BACK
```

Port 0 as input

In order to make port 0 an input, the port must be programmed by writing 1 to all the bits. Port 0 is configured first as an input port by writing 1s to it, and then data is received from that port and sent to P1

```
MOV A,#0FFH      ;A=FF hex
MOV P0,A         ;make P0 an i/p port ;by writing it all 1s
BACK: MOV A,P0   ;get data from P0
MOV P1,A         ;send it to port 1
SJMP BACK
```

Dual role of Port 0

Port 0 is also designated as AD0-AD7, allowing it to be used for both address and data. When connecting an 8051/31 to an external memory, port 0 provides both address and data.

Port 1 can be used as input or output

In contrast to port 0, this port does not need any pull-up resistors since it already has pull-up resistors internally. Upon reset, port 1 is configured as an input port.

The following code will continuously send out to port 0 the alternating value 55H and AAH

```
        MOV A,#55H
BACK: MOV P1,A
        ACALL DELAY
        CPL A
        SJMP BACK
```

To make port 1 an input port, it must be programmed as such by writing 1to all its bits.

Port 1 is configured first as an input port by writing 1s to it, then data is received from that port and saved in R7 and R5

```
MOV A,#0FFH      ;A=FF hex
MOV P1,A         ;make P1 an input port ,by writing it all 1s
MOV A,P1         ;get data from P1
MOV R7,A         ;save it to in reg R7
ACALL DELAY      ;wait
MOV A,P1         ;another data from P1
MOV R5,A         ;save it to in reg R5
```

Port 2 can be used as input or output

Just like P1, port 2 does not need any pullup resistors since it already has pull-up resistors internally. Upon reset, port 2 is configured as an input port. To make port 2 an input port, it must be programmed as such by writing 1 to all its bits. In many 8051-based systems, P2 is used as simple I/O. In 8031-based systems, port 2 must be used along with P0 to provide the 16-bit address for the external memory. Port 2 is also designated as A8 – A15, indicating its dual function. Port 0 provides the lower 8 bits via A0 – A7.

Port 3 can be used as input or output

Port 3 does not need any pull-up resistors. Port 3 is configured as an input port upon reset.Port 3 has the additional function of providing some extremely important signals.

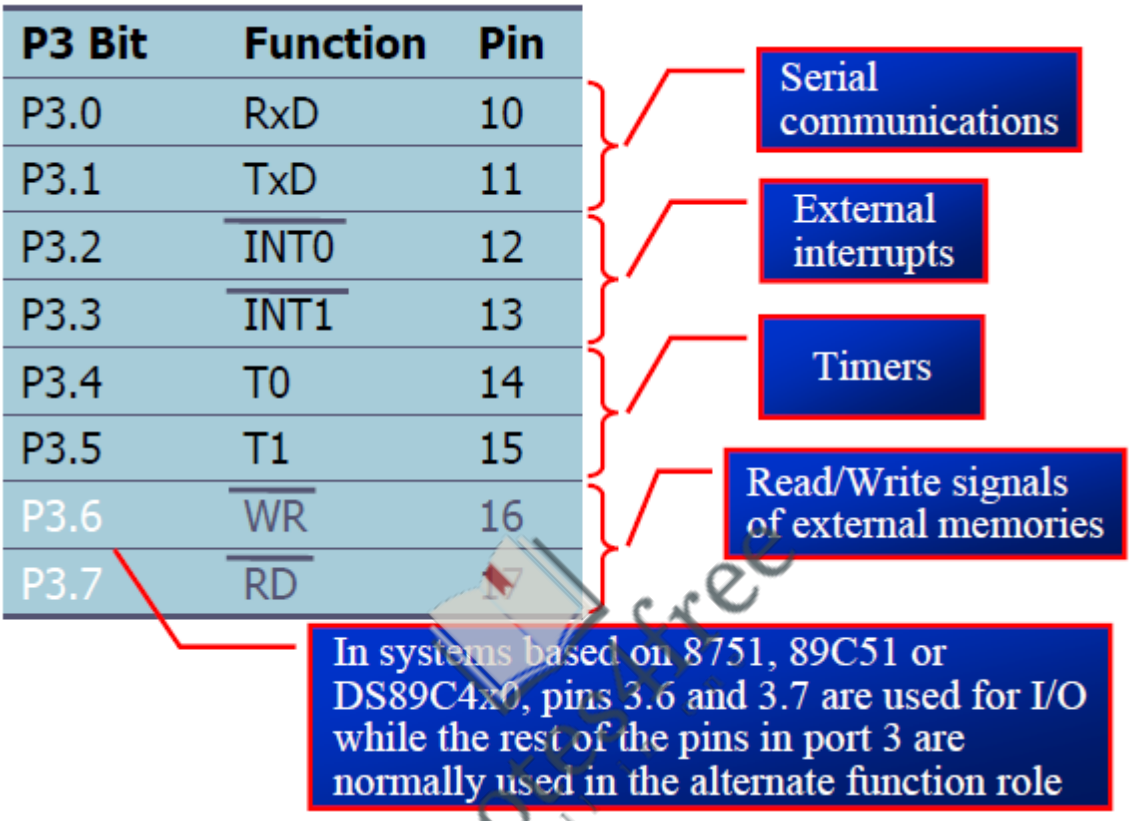


Fig 15: Port 3 alternate functions

Memory Address Decoding

The CPU provides the address of the data desired, but it is the job of the decoding circuitry to locate the selected memory block . Memory chips have one or more pins called CS (chip select), which must be activated for the memory contents to be accessed. Sometimes the chip select is also referred as chip enable (CE).

In connecting a memory chip to the CPU, note the following points

1. The data bus of the CPU is connected directly to the data pins of the memory chip

2. Control signals RD (read) and WR (memory write) from the CPU are connected to the OE (output enable) and WE (write enable) pins of the memory chip.

3. In the case of the address buses, while the lower bits of the address from the CPU go directly to the memory chip address pins, the upper ones are used to activate the CS pin of the memory chip.

Normally memories are divided into blocks and the output of the decoder selects a given memory block

1. Using simple logic gates
2. Using the 74LS138
3. Using programmable logics.

1. Simple Logic gate address decoder

The simplest way of decoding circuitry is the use of NAND or other gates. The fact that the output of a NAND gate is active low, and that the CS pin is also active low makes them a perfect match.

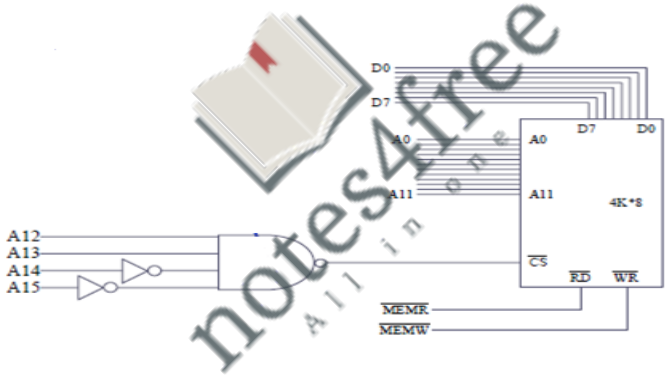
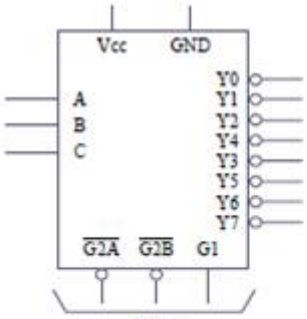


Fig 16: Logic gate as decoder

2. Using the 74LS138 3-8 decoder

This is one of the most widely used address decoder. The 3 inputs A, B, and C generate 8 active low outputs Y0 – Y7. Each Y output is connected to CS of a memory chip, allowing control of 8 memory blocks by a single 74LS138. In the 74LS138, where A, B, and C select which output is activated, there are three additional inputs, G2A, G2B, and G1. G2A and G2B are both active low, and G1 is active high. If any one of the inputs G1, G2A, or G2B is not connected to an address signal, they must be activated permanently either by Vcc or ground, depending on the activation level.



Function Table

Inputs				Outputs							
Enable		Select									
G1	G2	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

Figure: 74LS138

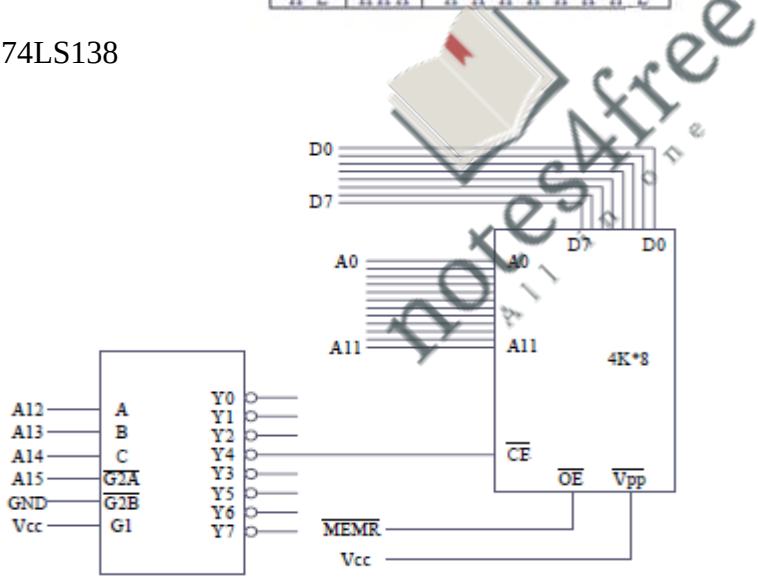


Fig 17: 74LS138 as decoder

Find the address range for the Following. (a) Y4 (b) Y2 and (c) Y7.

(a) The address range for Y4 is calculated as follows.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1

The above shows that the range for Y4 is 4000H to 4FFFH. We notice that A15 must be 0 for the decoder to be activated. Y4 will be selected when A14 A13 A12 = 100 (4 in binary). The remaining A11-A0 will be 0 for the lowest address and 1 for the highest address.

(b) The address range for Y2 is 2000H to 2FFFH.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1

(c) The address range for Y7 is 7000H to 7FFFH.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

4. Using programmable logic as an address decoder

Other widely used decoders are programmable logic chips such as PAL and GAL chips. One disadvantage of these chips is that one must have access to a PAL/GAL software and burner, whereas the 74LS138 needs neither of these. The advantage of these chips is that they are much more versatile since they can be programmed for any combination of address ranges.

8031/51 interfacing with external ROM and RAM

8031/51 interfacing with external ROM

The 8031 chip is a ROM less version of the 8051. It is exactly like any member of the 8051 family as far as executing the instructions and features are concerned, but it has no on-chip ROM. To make the 8031 execute 8051 code, it must be connected to external ROM memory containing the program code. 8031 is ideal for many systems where the on-chip ROM of 8051 is not sufficient, since it allows the program size to be as large as 64K bytes.

For 8751/89C51/DS5000-based system, we connected the EA pin to Vcc to indicate that the program code is stored in the microcontroller's on-chip ROM. To indicate that the program code is stored in external ROM, this pin must be connected to GND.

Since the PC (program counter) of the 8031/51 is 16-bit, it is capable of accessing up to 64K bytes of program code. In the 8031/51, port 0 and port 2 provide the 16-bit address to access

external memory. P0 provides the lower 8 bit address A0 – A7, and P2 provides the upper 8 bit address A8 – A15. P0 is also used to provide the 8-bit data bus D0 – D7. P0.0 – P0.7 are used for both the address and data paths address/data multiplexing.

ALE (address latch enable) pin is an output pin for 8031/51. ALE = 0, P0 is used for data path, ALE = 1, P0 is used for address path

To extract the address from the P0 pins we connect P0 to a 74LS373 and use the ALE pin to latch the address.

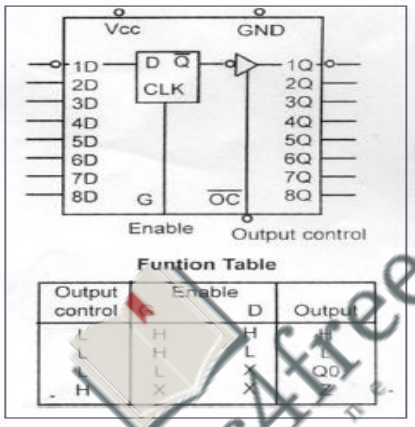


Fig 18: 74LS373 D Latch

Normally ALE = 0, and P0 is used as a data bus, sending data out or bringing data in. Whenever the 8031/51 wants to use P0 as an address bus, it puts the addresses A0 – A7 on the P0 pins and activates ALE = 1.

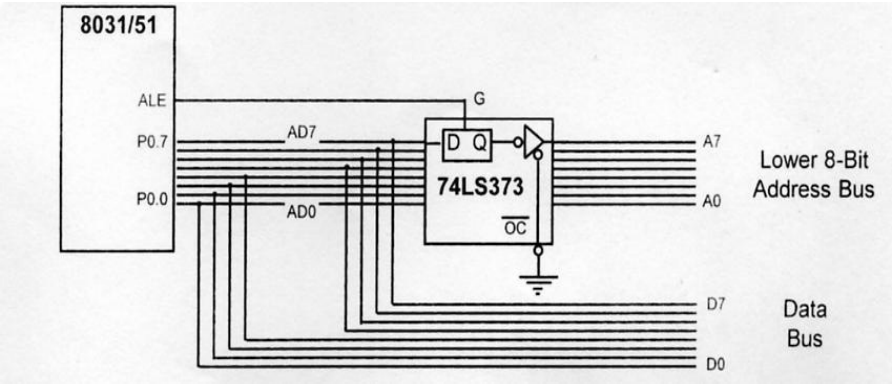


Fig 19: Address/Data multiplexing

PSEN (program store enable) signal is an output signal for the 8031/51microcontroller and must be connected to the OE pin of a ROM containing the program code. It is important to emphasize

the role of EA and PSEN when connecting the 8031/51 to external ROM. When the EA pin is connected to GND, the 8031/51 fetches opcode from external ROM by using PSEN.

The connection of the PSEN pin to the OE pin of ROM. In systems based on the 8751/89C51/ DS5000 where EA is connected to Vcc, these chips do not activate the PSEN pin. This indicates that the on-chip ROM contains program code.

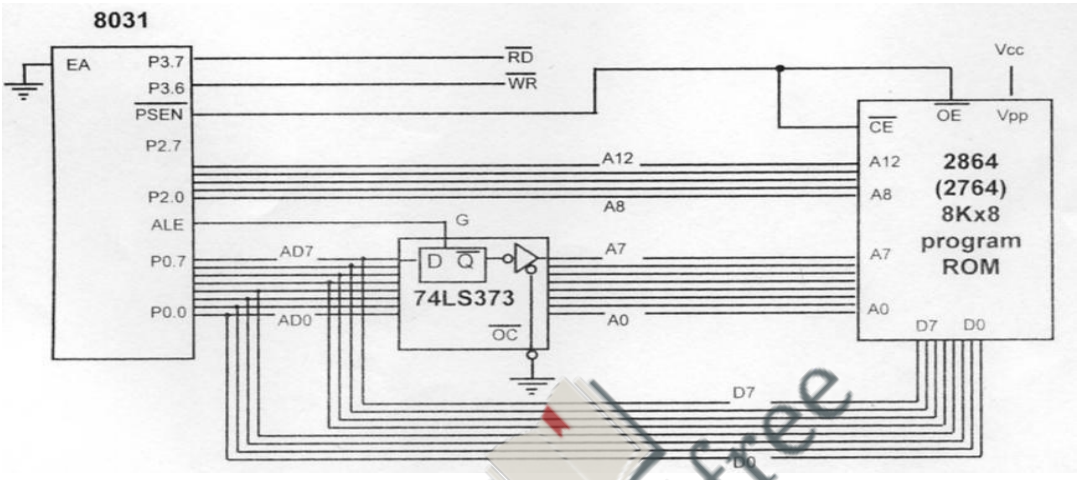


Fig 20: Data , address and control buses for the 8031/51

In an 8751 system we could use onchip ROM for boot code and an external ROM will contain the user's program. We still have EA = Vcc, Upon reset 8051 executes the on-chip program first, then when it reaches the end of the on-chip ROM, it switches to external ROM for rest of program.

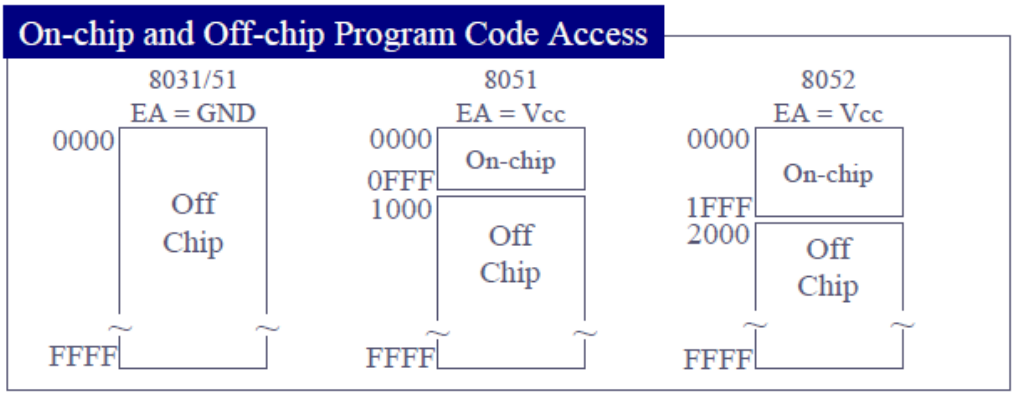


Fig 21: On chip and off chip program code access

8051 data memory space

The 8051 has 128K bytes of address space. 64K bytes are set aside for program code. Program space is accessed using the program counter (PC) to locate and fetch instructions. In some example we placed data in the code space and used the instruction MOVC A,@A+DPTR to get data, where C stands for code. The other 64K bytes are set aside for data. The data memory space is accessed using the DPTR register and an instruction called MOVX, where X stands for external. The data memory space must be implemented externally. We use RD to connect the 8031/51 to external ROM containing data. For the ROM containing the program code,
PSEN is used to fetch the code.

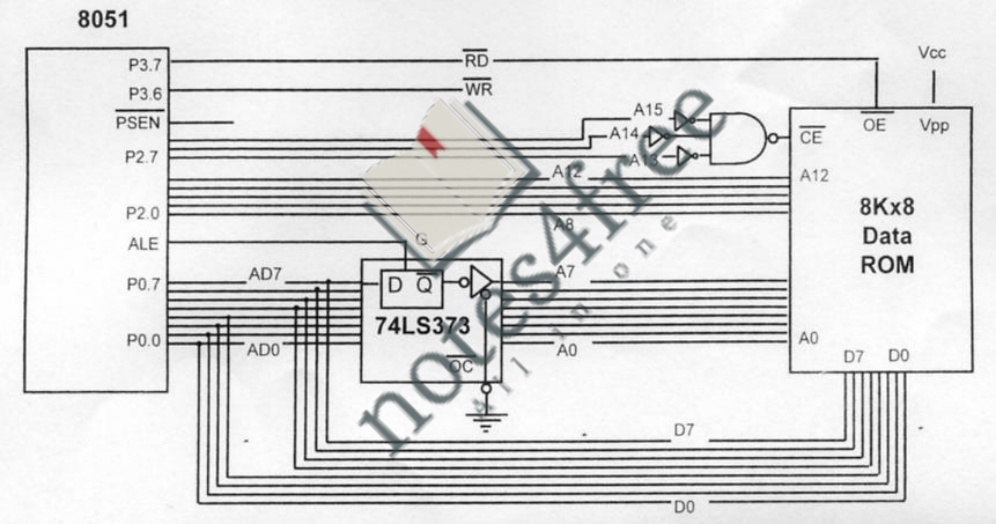


Fig 22: 8051 connection to external data ROM

MOVX is a widely used instruction allowing access to external data memory space. MOVX A,@DPTR instruction is used to bring externally stored data into the CPU.

8031 Connection to External Data ROM and External Program ROM

Design of an 8031-based system with 8K bytes of program ROM and 8K bytes of data ROM is as shown below. For program ROM, PSEN is used to activate both OE and CE. For data ROM, we use RD to active OE, while CE is activated by a Simple decoder.

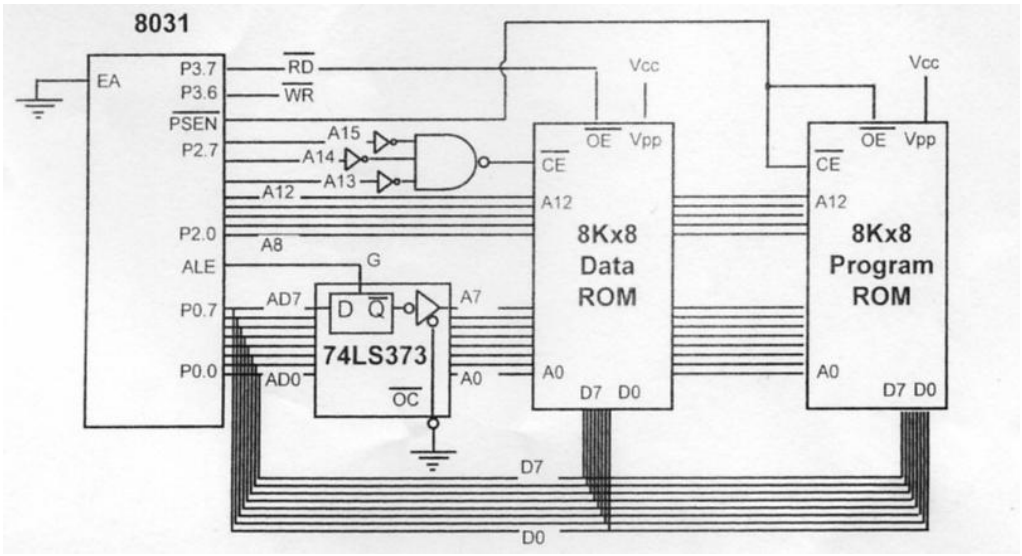


Fig 23: 8031 Connection to External Data ROM and External Program ROM

8051 Data memory space

To connect the 8051 to an external SRAM, we must use both RD (P3.7) and WR (P3.6) pins. In writing data to external data RAM, we use the instruction MOVX @DPTR, A. In some applications we need a large amount of memory to store data. The 8051 can support only 64K bytes of external data memory since DPTR is 16-bit.

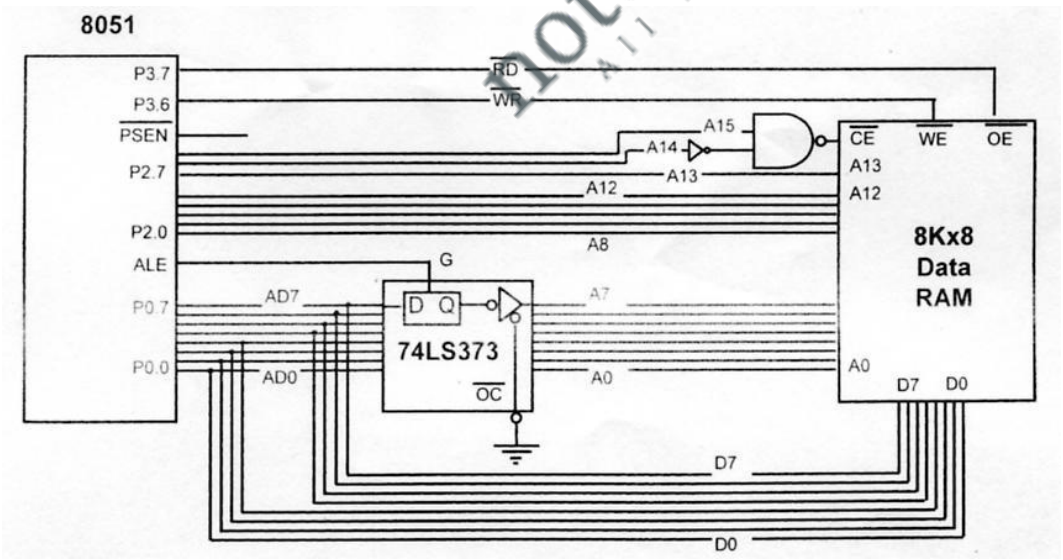


Fig 24: 8051 Connection to External Data RAM

External program memory is fetched if either of the following two conditions are satisfied.

1. EA (Enable Address) is low. The microcontroller by default starts searching for program from external program memory.
2. PC is higher than FFFH for 8051 or 1FFFH for 8052.
3. PSEN tells the outside world whether the external memory fetched is program memory or data memory. EA is user configurable. PSEN is processor controlled.

Some typical use of code/program memory access: External program memory can be not only used to store the code, but also for lookup table of various functions required for a particular application. Mathematical functions such as Sine, Square root, Exponential, etc. can be stored in the program memory (Internal or external) and these functions can be accessed using MOVC instruction.

Addressing Modes

8051 Addressing modes

Addressing mode is a way to address an operand. Operand means the data we are operating upon. It can be a direct address of memory, it can be register names, it can be any numerical data etc.

MOV A,#6AH

Here the data 6A is the operand, often known as source data. When this instruction is executed, the data 6AH is moved to accumulator A.

There are 5 different ways to execute this instruction and hence we say, we have got 5 addressing modes for 8051. They are **1.** Immediate addressing mode

2. Direct addressing mode
3. Register direct addressing mode
4. Register indirect addressing mode
5. indexed addressing mode.

1. Immediate Addressing Mode

In this mode data can be specified as a part of instruction to get data easily to a destination. Here source operand is constant. Here mnemonic for immediate data is pound sign(#).

This can copy immediate numbers form the opcode into registers(R0 to R7), A and DPTR.

Ex: MOV R0,#08H

MOV DPTR,#1234H

Let's begin with an example. **MOV A, #6AH**

In general we can write MOV A, #data

This addressing mode is named as "**immediate"** because it transfers an 8-bit data immediately to the accumulator (destination operand). The opcode for MOV A, # data is 74H. The opcode is saved in program memory at 0202 address. The data 6AH is saved in program memory 0203. This instruction is of two bytes and is executed in one cycle. So after the execution of this instruction, program counter will add 2 and move to 0204 of program memory.

Note: The '**#'** symbol before 6AH indicates that operand is a data (8 bit). If '**#'** is not present then the hexadecimal number would be taken as address.

2.Direct Addressing Mode

This is another way of addressing an operand. Here the address of the data (source data) is given as operand. Complete 128 bytes of internal RAM and SFR's may be addressed directly using single byte address assigned to each RAM location.

Ex: MOV A, 50H Load data stored in 50H into Accumulator

ADD A, 65H Adds the content of accumulator with content of 65H memory location.

3. Register Addressing Mode

In this addressing mode we use the register name directly (as source operand). **MOV A, R4.** At a time registers can take value from R0,R1…to R7. You may already know there are 32 such registers. There are 4 register banks named 0,1,2 and 3. Each bank has 8 registers named from R0 to R7. At a time only one register bank can be selected. Selection of register bank is made possible through a Special Function Register (SFR) named Processor Status Word (PSW). A data move does not alter the contents of data source address. Contents of destination address are replaced by source address contents.

Ex: MOV A,R4: Move the content of register R4 into accumulator

MOV R5,A: Move the content of accumulator into register R5

4.Indirect Addressing Mode

In this addressing mode, address of the data (source data to transfer) is given in the register operand. It uses a register to hold the actual address that will finally be used in data move. The register itself is not an address. R0 and R1 registers are used as pointer registers An example is shown below. To fetch of RAM location indirectly.

The instruction with indirect addresses uses @ sign.

Ex: MOV A, @R0: Load the content pointed by RAM location into Accumulator. Here the value inside

R0 is considered as an address, which holds the data to be transferred to accumulator. If R0 holds the value 20H, and we have a data 2F H stored at the address 20H, then the value 2FH will get transferred to accumulator after executing this instruction.

ADD A,@R1: Adds the content of accumulator with content pointed by RAM location.

Note: Only R0 and R1 are allowed to form a register indirect addressing instruction. In other words programmer can must make any instruction either using @R0 or @R1.

5.Indexed Addressing Mode

This mode uses a base register either as program counter or data pointer and an offset as accumulator in forming effective address for JMP or MOVC instruction. Jump table and lookup tables are easily created using indexed addressing.

MOVC A, @A+DPTR and MOVC A, @A+PC

where DPTR is data pointer and PC is program counter (both are 16 bit registers).

Lets take the first example.

MOVC A, @A+DPTR

The source operand is @A+DPTR and we know we will get the source data (to transfer) from this location. It is nothing but adding contents of DPTR with present content of accumulator. This addition will result a new data which is taken as the address of source data (to transfer). The data at this address is then transferred to accumulator.

The opcode for the instruction is 93H. DPTR holds the value 01FE, where 01 is located in DPH (higher 8 bits) and FE is located in DPL (lower 8 bits). Accumulator now has the value 02H. A 16 bit addition is performed and now 01FE H+02 H results in 0200 H. What ever data is in 0200 H will get transferred to accumulator. The previous value inside accumulator (02H) will get replaced with new data from 0200H.

This is a 1 byte instruction with 2 cycles needed for execution. .

The other example **MOVC A, @A+PC** works the same way as above example. The only difference is, instead of adding DPTR with accumulator, here data inside program counter (PC) is added with accumulator to obtain the target address.

Course Coordinator: Prof. Mahesh P.Yanagimath

Course Coordinator: Prof. Mahesh P.Yanagimath

Module-2

Assembly programming and instruction of 8051

In the early days of the computer, programmers coded in machine language, consisting of 0s and 1s is tedious, slow and prone to error. Assembly languages, which provides mnemonics for the machine code instructions, plus other features, were developed. An Assembly language program consists of a series of lines of Assembly language instructions. Assembly language is referred to as a low level language. It deals directly with the internal structure of the CPU.

Assembly language instruction includes a mnemonic (abbreviation easy to remember) the commands to the CPU, telling it what those to do with those items optionally followed by one or two operands the data items being manipulated. A given Assembly language program is a series of statements, or lines. Assembly language instructions tell the CPU what to do. Directives (or pseudo-instructions) Give directions to the assembler.

Assembling and running an 8051 program

The step of Assembly language program are outlines as follows:

1) First we use an editor to type a program, many excellent editors or word processors are available that can be used to create and/or edit the program. Notice that the editor must be able to produce an ASCII file. For many assemblers, the file names follow the usual DOS conventions, but the source file has the extension "asm" or "src", depending on which assembly you are using

2) The "asm" source file containing the program code created in step 1 is fed to an 8051assembler. The assembler converts the instructions into machine code.The assembler will produce an object file and a list file. The extension for the object file is "obj" while the extension for the list file is "lst".

3) Assembler require a third step called linking. The linker program takes one or more object code files and produce an absolute object file with the extension "abs". This abs file is used by 8051 trainers that have a monitor program.

4) Next the "abs" file is fed into a program called "OH" (object to hex converter) which creates a file with extension "hex" that is ready to burn into ROM. This program comes with all 8051 assemblers. Recent Windows-based assemblers combine step 2 through 4 into one step.

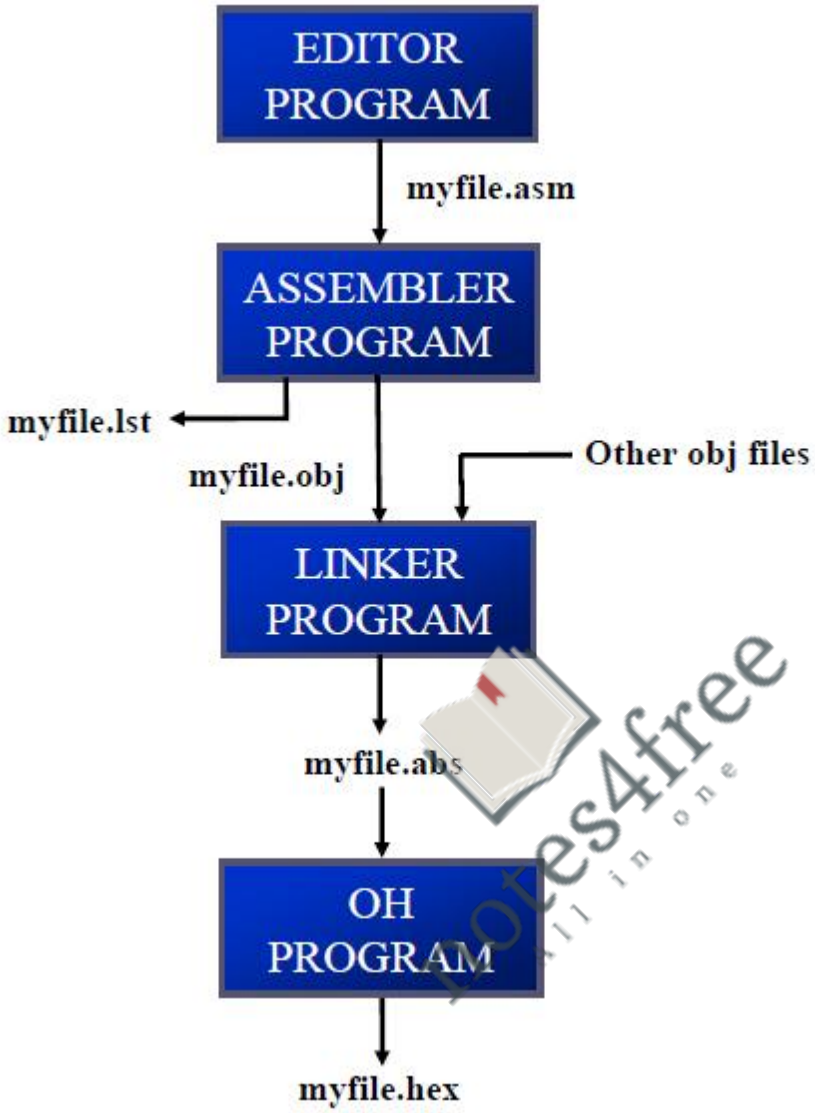


Fig: Flowchart showing Assembly programming.

8051 Data types and directives

The 8051 microcontroller has only one data type. It is 8 bits, and the size of each register is also 8 bits. It is the job of the programmer to break down data larger than 8 bits (00 to FFH, or 0 to 255 in decimal) to be processed by the CPU.

DB (define byte)

The DB directive is the most widely used data directive in the assembler. It is used to define the 8-bit data. When DB is used to define data, the numbers can be in decimal, binary, hex, or ASCII

formats. For decimal, the "D" after the decimal number is optional, but using "B" (binary) and "H" (hexadecimal) for the others is required. Regardless of which is used, the assembler will convert the numbers into hex. To indicate ASCII, simply place the characters in quotation marks ('like this'). The assembler will assign the ASCII code for the numbers or characters automatically. The DB directive is the only directive that can be used to define ASCII strings larger than two characters; therefore, it should be used for all ASCII data definitions.

Assembler directives

The following are widely used directives of the 8051.

ORG (origin)

The ORG directive is used to indicate the beginning of the address. The number that comes after ORG can be either in hex or in decimal. If the number is not followed by H, it is decimal and the assembler will convert it to hex. Some assemblers use ". ORG" (notice the dot) instead of "ORG" for the origin directive.

EQU (equate)

This is used to define a constant without occupying a memory location. The EQU directive does not set aside storage for a data item but 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. The following uses EQU for the counter constant and then the constant is used to load the R3 register.

When executing the instruction "MOV R3, #COUNT", the register R3 will be loaded with the value 25 (notice the # sign).

What is the advantage of using EQU?

Assume that there is a constant (a fixed value) used in many different places in the program, and the programmer wants to change its value throughout. By the use of EQU, the programmer can change it once and the assembler will change all of its occurrences, rather than search the entire program trying to find every occurrence.

END directive

Another important pseudocode is the END directive. This indicates to the assembler the end of the source (asm) file. The END directive is the last line of an 8051 program, meaning that in the source code anything after the END directive is ignored by the assembler. Some assemblers use "END" (notice the dot) instead of "END".

Instruction Syntax

8051 instruction set is coded set that have been defined by manufacturer of 8051.

An 8051 instruction syntax is shown below.

Mnemonic	Destination operand	Source operand

Operand is data address it specifies the destination for the data i.e being copied form the source and also it gives us source address. Destination and source addresses are separated by (,) comma. Assembler will come to know when one operand name ends and other begins.

Data transfer instructions

MOV <dest-byte>,<src-byte>-

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

1. mov direct , A
2. mov A, @R_i
3. mov A, R_n
4. mov direct, direct
5. mov A, #data

EX: MOV 30h, A

MOV A,@R0 ; moves the content of memory pointed to by Ro into A

MOV A, R_1; moves the content of Register R_1 to Accumulator A

MOV 20h,30h; moves the content of memory location 30h to 20h

MOV A,#45h; moves 45h to Accumulator A

MOV <dest-bit>,<src-bit>

Function: Move bit data

Description: MOV <dest-bit>,<src-bit> copies the Boolean variable indicated by the second operand into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Example: MOV P1.3,C; moves the carry bit to 3rd bit of port1

MOV DPTR,#data16

Function: Load Data Pointer with a 16-bit constant

Description: MOV DPTR,#data16 loads the Data Pointer with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte(DPL) holds the lower-order byte. No flags are affected.

This is the only instruction which moves 16 bits of data at once.

Example: The instruction,MOV DPTR, # 4567H loads the value 4567H into the Data Pointer. DPH holds 45H, and DPL holds 67H.

MOVC A,@A+ <base-reg>

Function: Move Code byte

Description: The MOVC instructions load the Accumulator with a code byte or constant from program memory. The address of the byte fetched is the sum of the original unsigned 8-bit Accumulator contents and the contents of a 16-bit base register, which may be either the Data Pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added with the Accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

MOVC A,@A+PC

(PC) (PC) + 1

(A) ((A) + (PC))

MOVX <dest-byte>,<src-byte>

Function: Move External

Description: The MOVX instructions transfer data between the Accumulator and a byte of external data memory, which is why "X" is appended to MOV. There are two types of instructions, differing in whether they provide an 8-bit or 16-bit indirect address to the external data RAM.

This form of MOVX is faster and more efficient when accessing very large data arrays (up to 64K bytes), since no additional instructions are needed to set up the output ports.

Example:

MOVX A,@DPTR

MOVX @DPTR, A

(A) ((DPTR))

PUSH direct

Function: Push onto stack

Description: The Stack Pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the Stack Pointer. No flags are affected.

Example: On entering an interrupt routine, the Stack Pointer contains 09H. The Data Pointer holds the value 0123H. The following instruction sequence,

PUSH DPL

PUSH DPH

leaves the Stack Pointer set to 0BH and stores 23H and 01H in internal RAM locations 0AH and 0BH respectively.

POP direct

Function: Pop from stack.

Description: The contents of the internal RAM location addressed by the Stack Pointer is read, and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

Example: The Stack Pointer originally contains the value 32H, and internal RAM locations 30H through 32H contain the values 20H, 23H, and 01H, respectively. The following instruction sequence,

POP DPH

POP DPL leaves the Stack Pointer equal to the value 30H and sets the Data Pointer to 0123H.

Arithmetic Group of Instructions

ADD A,<src-byte>

Function: Add

Description: ADD adds the byte variable indicated to the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred. OV is set if a carry-out of bit 7 but not bit 6, otherwise, OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands. Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H (11000011B), and register 0 holds 0AAH (10101010B). The following instruction,

ADD A,R0

leaves 6DH (01101101B) in the Accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADD A, direct

(A) (A) + (direct)

ADD A, @Ri

(A) (A) + data

ADDC A, <src-byte>

Function: Add with Carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary-carry flags

are set respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands. Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The following instruction,

ADDC A,R0

leaves 6EH (01101110B) in the Accumulator with AC cleared and both the Carry flag and OV set to 1.

ADDC A,Rn

Operation: ADDC

- (A) + (C) + (Rn)

ADDC A, direct

Operation: ADDC

(A) (A) + (C) + (direct)

ADDC A,@Ri

Operation: ADDC

(A) (A) + (C) + ((Ri))

ADDC A, #data

Operation: ADDC

(A) (A) + (C) + #data

SUBB A,<src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the Accumulator, leaving the result in the Accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7 and clears C otherwise. (If C was set *before* executing a SUBB

instruction, this indicates that a borrow was needed for the previous step in a multiple-precision subtraction, so the carry is subtracted from the Accumulator along with the source operand.) AC is set if a borrow is needed for bit 3 and cleared otherwise. The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry flag is set. The instruction, SUBB A,R2 will leave the value 74H (01110100B) in the accumulator, with the carry flag and AC cleared but OV set.

Instructions	**OpCode**	**Bytes**	**Flags**
SUBB A,*#data*	0x94	2	C, AC, OV
SUBB A,*iram addr*	0x95	2	C, AC, OV
SUBB A,@R0	0x96	1	C, AC, OV
SUBB A,@R1	0x97	1	C, AC, OV
SUBB A,R0	0x98	1	C, AC, OV
SUBB A,R1	0x99	1	C, AC, OV
SUBB A,R2	0x9A	1	C, AC, OV
SUBB A,R3	0x9B	1	C, AC, OV
SUBB A,R4	0x9C	1	C, AC, OV
SUBB A,R5	0x9D	1	C, AC, OV
SUBB A,R6	0x9E	1	C, AC, OV
SUBB A,R7	0x9F	1	C, AC, OV

SUBB A,Rn

Operation: SUBB

(A) (A) - (C) - (Rn)

SUBB A, direct

Operation: SUBB

(A) (A) - (C) - (direct)

SUBB A,@Ri

Operation: SUBB

(A) (A) - (C) - ((Ri))

MUL AB

Function: Multiply

Description: MUL AB multiplies the unsigned 8-bit integers in the Accumulator and register B. The low-order byte of the 16-bit product is left in the Accumulator, and the high-order byte in B. If the product is greater than 255 (0FFH), the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.

Example: Originally the Accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction,

MUL AB

will give the product 12,800 (3200H), so B is changed to 32H (00110010B) and the Accumulator is cleared. The overflow flag is set, carry is cleared.

DIV AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B. The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags are cleared. *Exception*: if B had originally contained 00H, the values returned in the Accumulator and B-register are undefined and the overflow flag are set. The carry flag is cleared in any case.

Example: The Accumulator contains 251 (0FBH or 11111011B) and B contains 18 (12H or 00010010B). The following instruction, DIV AB leaves 13 in the Accumulator (0DH or 00001101B) and the value 17 (11H or 00010001B) in B, since 251 = (13 x 18) + 17. Carry and OV are both cleared.

DA A

Function: Decimal-adjust Accumulator for Addition

Description: DA A adjusts the eight-bit value in the Accumulator resulting from the earlier addition of two variables (each in packed-BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition. If Accumulator bits 3 through 0 are greater than nine or if the AC flag is one, six is added to the Accumulator producing the proper BCD digit in the low-order nibble. This internal addition sets the carry flag if a carry-out of the low-order four-bit field propagates through all high-order bits, but it does not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed nine, these high-order bits are added with six, producing the proper BCD digit in the high-order nibble. Again, this sets the carry flag if there is a carry-out of the high-order bits, but does not clear the carry.

SWAP A

Function: Swap nibbles within the Accumulator

Description: SWAP A interchanges the low- and high-order nibbles (four-bit fields) of the Accumulator (bits 3 through 0 and bits 7 through 4). The operation can also be thought of as a 4-bit rotate instruction. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The instruction, SWAP A leaves the Accumulator holding the value 5CH (01011100B

Operation: SWAP

(A3-0) D (A7-4)

XCH A,<byte>

Function: Exchange Accumulator with byte variable

Description: XCH loads the Accumulator with the contents of the indicated variable, at the same time writing the original Accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

Example: R0 contains the address 20H. The Accumulator holds the value 3FH (0011111lB). Internal RAM location 20H holds the value 75H (01110101B). The following instruction, XCH A,@R0 leaves RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the accumulator.

XCHD A,@Ri

Function: Exchange Digit

Description: XCHD exchanges the low-order nibble of the Accumulator (bits 3 through 0), generally representing a hexadecimal or BCD digit, with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Example: R0 contains the address 20H. The Accumulator holds the value 36H (00110110B). Internal RAM location 20H holds the value 75H (01110101B). The following instruction, XCHD A,@R0 leaves RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the Accumulator.

CPL A

Function: Complement Accumulator

Description: CPLA logically complements each bit of the Accumulator (one's complement). Bits which previously contained a 1 are changed to a 0 and vice-versa. No flags are affected.

Example: The Accumulator contains 5CH (01011100B).

The following instruction, CPL A leaves the Accumulator set to 0A3H (10100011B).

CPL bit

Function: Complement bit

Description: CPL bit complements the bit variable specified. A bit that had been a 1 is changed to 0 and vice-versa. No other flags are affected. CLR can operate on the carry or any directly addressable bit.

Example: Port 1 has previously been written with 5BH (01011101B). The following instruction sequence, CPL P1.1CPL

P1.2 leaves the port set to 5BH (01011011B).

DEC byte

Function: Decrement

Description: DEC byte decrements the variable indicated by 1. An original value of 00H underflows to 0FFH. No flags are affected.

Example: Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The following instruction sequence,

DEC R0

DEC @R0

leaves register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH.

DEC A

DEC Rn

DEC direct

DEC @Ri

INC <byte>

Function: Increment

Description: INC increments the indicated variable by 1. An original value of 0FFH overflows to 00H. No flags are affected.

Example: Register 0 contains 7EH (01111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The following instruction sequence,

INC R0

INC @R0 leaves register 0 set to 7FH and internal RAM locations 7EH and 7FH holding 00H and 41H, respectively.

INC A Operation: INC (A) (A) + 1

INC DPTR

Function: Increment Data Pointer

Description: INC DPTR increments the 16-bit data pointer by 1. A 16-bit increment (modulo 216) is performed, and an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H increments the high-order byte (DPH). No flags are affected. This is the only 16-bit register which can be incremented.

Example: Registers DPH and DPL contain 12H and 0FEH, respectively. The following instruction sequence,

INC DPTR

INC DPTR

INC DPTR changes DPH and DPL to 13H and 01H.

NOP

Function: No Operation

Description: Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

Logical instructions

ANL <dest-byte>,<src-byte>

Function: Logical-AND for byte variables

Description: ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected. The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Example: If the Accumulator holds 0C3H (1100001lB), and register 0 holds 55H (01010101B), then the following instruction, ANL A,R0 leaves 41H (01000001B) in the Accumulator.

When the destination is a directly addressed byte, this instruction clears combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the Accumulator at run-time. The following instruction, ANL P1,#01110011B clears bits 7, 3, and 2 of output port 1.

Instructions	OpCode	Bytes	Flags
ANL *iram addr*,A	0x52	2	None
ANL *iram addr*,*#data*	0x53	3	None
ANL A,*#data*	0x54	2	None
ANL A,*iram addr*	0x55	2	None
ANL A,@R0	0x56	1	None
ANL A,@R1	0x57	1	None
ANL A,R0	0x58	1	None
ANL A,R1	0x59	1	None
ANL A,R2	0x5A	1	None
ANL A,R3	0x5B	1	None
ANL A,R4	0x5C	1	None
ANL A,R5	0x5D	1	None

ANL A,R6	0x5E	1	None
ANL A,R7	0x5F	1	None
ANL C,*bit addr*	0x82	2	C
ANL C,/*bit addr*	0xB0	2	C

ANL A,Rn

Operation: AND the content of accumulator with the content of Register specified.

ANL A,@Ri

Operation: ANL

ANL direct,#data

Operation: ANL

(direct) #data ^ (direct)

ORL <dest-byte> <src-byte>

Function: Logical-OR for byte variables

Description: ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected.

Example: If the Accumulator holds 0C3H (11000011B) and R0 holds 55H (01010101B) then the following instruction, ORL A,R0 leaves the Accumulator holding the value 0D7H (1101011lB).

The instruction,ORL P1,#00110010B sets bits 5, 4, and 1 of output Port 1.

ORL A, Rn ; or the content of Accumulator and Register Rn and store the result in Accumulator

ORL A, direct ; or the content of Accumulator and the memory and store the result in Accumulator

ORL A, @Ri ; or the content of accumulator and the memory location whose address is specified in Ri

ORL C,<src-bit>

Function: Logical-OR for bit variables

Description: Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise. A slash (/) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Example:

ORL C, ACC.7 ;OR CARRY WITH THE ACC. BIT 7

ORL C, /OV ;OR CARRY WITH THE INVERSE OF OV.

SETB

Operation: **SETB**
Function: Set Bit
Syntax: SETB *bitaddr*

Description: Sets the specified bit.

XRL <dest-byte>,<src-byte>

Function: Logical Exclusive-OR for byte variables

Description: XRL performs the bitwise logical Exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected. The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

Example: If the Accumulator holds 0C3H (1100001lB) and register 0 holds 0AAH (10101010B) then the instruction, XRL A,R0 leaves the Accumulator holding the value 69H (01101001B).

Instructions	OpCode	Bytes	Flags
XRL *iram addr*,A	0x62	2	None
XRL *iram addr*,*#data*	0x63	3	None
XRL A,*#data*	0x64	2	None
XRL A,*iram addr*	0x65	2	None

XRL A,@R0	0x66	1	None
XRL A,@R1	0x67	1	None
XRL A,R0	0x68	1	None
XRL A,R1	0x69	1	None
XRL A,R2	0x6A	1	None
XRL A,R3	0x6B	1	None
XRL A,R4	0x6C	1	None
XRL A,R5	0x6D	1	None
XRL A,R6	0x6E	1	None
XRL A,R7	0x6F	1	None

Rotate Instructions

RL A

Function: Rotate Accumulator Left

Description: The eight bits in the Accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B). The following instruction,

RL A leaves the Accumulator holding the value 8BH (10001011B) with the carry unaffected.

RLC A

Function: Rotate Accumulator Left through the Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.

Example: The Accumulator holds the value 0C5H(11000101B), and the carry is zero. The following instruction, RLC A leaves the Accumulator holding the value 8BH (10001010B) with the carry set.

RRC A

Function: Rotate Accumulator Right through Carry flag

Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.

Example: The Accumulator holds the value 0C5H (11000101B), the carry is zero. The following instruction, RRC A leaves the Accumulator holding the value 62 (01100010B) with the carry set.

Jump Instructions

Unconditional Branch Instructions: It is a jump in which control is transferred unconditionally to target instruction. Basically there are three unconditional jump instructions 1) AJMP 2) LJMP 3)SJMP

Operation:	**AJMP**
Function:	Absolute Jump Within 2K Block
Syntax:	AJMP *code address*

Instructions	**OpCode**	**Bytes**	**Flags**
AJMP *page0*	0x01	2	None
AJMP *page1*	0x21	2	None
AJMP *page2*	0x41	2	None
AJMP *page3*	0x61	2	None
AJMP *page4*	0x81	2	None
AJMP *page5*	0xA1	2	None
AJMP *page6*	0xC1	2	None
AJMP *page7*	0xE1	2	None

Description: AJMP unconditionally jumps to the indicated *code address*. The new value for the Program Counter is calculated by replacing the least-significant-byte of the Program Counter with the second byte of the AJMP instruction, and replacing bits 0-2 of the most-significant-byte of the Program Counter with 3 bits that indicate the page *of the byte following the AJMP instruction*. Bits 3-7 of the most-significant-byte of the Program Counter remain unchanged. Since only 11 bits of the Program Counter are affected by AJMP, jumps may only be made to code located within the same 2k block as the first byte that follows AJMP.

LJMP(Long jump)

Operation: **LJMP**
Function: Long Jump
Syntax: LJMP *code address.*

Description: LJMP jumps unconditionally to the specified *code address*. It can jump in any memory from 0000H to FFFFH.

SJMP

SJMP

Operation:
Function: Short Jump
Syntax: SJMP *reladdr*

Description: SJMP jumps unconditionally to the address specified *reladdr*. *Reladdr* must be within -128 or +127 bytes of the instruction that follows the SJMP instruction

Conditional Jump Instructions

These instructions jumps to the specified location after checking a condition.

Operation: **JNC**
Function: Jump if Carry Not Set
Syntax: JNC *reladdr*

Description: JNC branches to the address indicated by *reladdr* if the carry bit is not set. If the carry bit is set program execution continues with the instruction following the JNB instruction.

Operation: **JC**
Function: Jump if Carry Set
Syntax: JC *reladdr*

Description: JC will branch to the address indicated by *reladdr* if the Carry Bit is set. If the Carry Bit is not set program execution continues with the instruction following the JC instruction.

Operation: **JNB**
Function: Jump if Bit Not Set
Syntax: JNB *bit addr,reladdr*

Description: JNB will branch to the address indicated by *reladdress* if the indicated bit is not set. If the bit is set program execution continues with the instruction following the JNB instruction.

Operation: **JB**
Function: Jump if Bit Set
Syntax: JB *bit addr, reladdr*

Description: JB branches to the address indicated by *reladdr* if the bit indicated by *bit addr* is set. If the bit is not set program execution continues with the instruction following the JB instruction.

Operation: **JNZ**
Function: Jump if Accumulator Not Zero
Syntax: JNZ *reladdr*

Description: JNZ will branch to the address indicated by *reladdr* if the Accumulator contains any value except 0. If the value of the Accumulator is zero program execution continues with the instruction following the JNZ instruction.

Operation: **JZ**
Function: Jump if Accumulator Zero
Syntax: JNZ *reladdr*

Description: JZ branches to the address indicated by *reladdr* if the Accumulator contains the value 0. If the value of the Accumulator is non-zero program execution continues with the instruction following the JNZ instruction.

Operation:	**DJNZ**
Function:	Decrement and Jump if Not Zero
Syntax:	DJNZ *register, reladdr*

Instructions	**OpCode**	**Bytes**	**Flags**
DJNZ *iram addr,reladdr*	0xD5	3	None
DJNZ R0,*reladdr*	0xD8	2	None
DJNZ R1,*reladdr*	0xD9	2	None
DJNZ R2,*reladdr*	0xDA	2	None
DJNZ R3,*reladdr*	0xDB	2	None
DJNZ R4,*reladdr*	0xDC	2	None
DJNZ R5,*reladdr*	0xDD	2	None
DJNZ R6,*reladdr*	0xDE	2	None
DJNZ R7,*reladdr*	0xDF	2	None

Description: DJNZ decrements the value of *register* by 1. If the initial value of *register* is 0, decrementing the value will cause it to reset to 255 (0xFF Hex). If the new value of *register* is not 0 the program will branch to the address indicated by *relative addr*. If the new value of *register* is 0 program flow continues with the instruction following the DJNZ instruction.

Operation:	**CJNE**
Function:	Compare and Jump If Not Equal
Syntax:	CJNE *operand1,operand2,reladdr*

Instructions	**OpCode**	**Bytes**	**Flags**
CJNE A,*#data, reladdr*	0xB4	3	C
CJNE A,*iram addr,reladdr*	0xB5	3	C
CJNE @R0,*#data,reladdr*	0xB6	3	C
CJNE @R1,*#data,reladdr*	0xB7	3	C
CJNE R0,*#data,reladdr*	0xB8	3	C
CJNE R1,*#data,reladdr*	0xB9	3	C
CJNE R2,*#data,reladdr*	0xBA	3	C
CJNE R3,*#data,reladdr*	0xBB	3	C
CJNE R4,*#data,reladdr*	0xBC	3	C
CJNE R5,*#data,reladdr*	0xBD	3	C
CJNE R6,*#data,reladdr*	0xBE	3	C
CJNE R7,*#data,reladdr*	0xBF	3	C

Description: CJNE compares the value of *operand1* and *operand2* and branches to the indicated relative address if *operand1* and *operand2* are not equal. If the two operands are equal program flow continues with the instruction following the CJNE instruction. The **Carry bit (C)** is set if *operand1* is less than *operand2*, otherwise it is cleared.

CALL INSTRUCTIONS

Another control transfer instruction is the CALL instruction, which is used to call a subroutine. Subroutines are often used to perform tasks that need to be performed frequently. This makes a program more structured in addition to saving memory space.

In the 8051 there are two instructions for call: LCALL (long call) and ACALL (absolute call). Deciding which one to use depends on the target address. Each instruction is explained next.

LCALL (long call)

In this 3-byte instruction, the first byte is the opcode and the second and third bytes are used for the address of the target subroutine. Therefore, LCALL can be used to call subroutines located anywhere within the 64K-byte address space of the 8051. To make sure that after execution of the called subroutine the 8051 knows where to come back to, the processor automatically saves

on the stack the address of the instruction immediately below the LCALL. When a subroutine is called, control is transferred to that subroutine, and the processor saves the PC (program counter) on the stack and begins to fetch instructions from the new location. After finishing execution of the subroutine, the instruction RET (return) transfers control back to the caller. Every subroutine needs RET as the last instruction.

ACALL (absolute call)

ACALL is a 2-byte instruction in contrast to LCALL. Since ACALL is a 2-byte instruction, the target address of the subroutine must be within 2K bytes because only 11 bits of the 2 bytes are used for the address. There is no difference between ACALL and LCALL in terms of saving the program counter on the stack or the function of the RET instruction. The only difference is that the target address for LCALL can be anywhere within the 64K-byte address space of the 8051 while the target address of ACALL must be within a 2K-byte range.

RET

Function: Return From Subroutine

Syntax: RET

Description: RET is used to return from a subroutine previously called by LCALL or ACALL. Program execution continues at the address that is calculated by popping the topmost 2 bytes off the stack. The most-significant-byte is popped off the stack first, followed by the least-significant-byte.

RETI

Operation: **RETI**

Function: Return From Interrupt

Syntax: RETI

Description: RETI is used to return from an interrupt service routine. RETI first enables interrupts of equal and lower priorities to the interrupt that is terminating. Program execution continues at the address that is calculated by popping the topmost 2 bytes off the stack. The most-significant-byte is popped off the stack first, followed by the least-significant-byte.

Module No.3

8051 Programming using C

Compilers produce hex files that is downloaded to ROM of microcontroller. The size of hex file is the main concern. Microcontrollers have limited on-chip ROM. Code space for 8051 is limited to 64K bytes. C programming is less time consuming, but has larger hex file size. The reasons for writing programs in C. It is easier and less time consuming to write in C than Assembly. C is easier to modify and update. We can use code available in function libraries. C code is portable to other microcontroller with little no of modifications.

Data types in C

A good understanding of C data types for 8051 can help programmers to create smaller hex files.
Unsigned char
Signed char
Unsigned int
Signed int
Sbit (single bit)
Bit and sfr

The character data type is the most natural choice. Unsigned char is an 8-bit data type in the range of 0 – 255 (00 – FFH). C compilers use the signed char as the default if we do not put the keyword unsigned.

Data Type	Size in Bits	Data Range/Usage
unsigned char	8-bit	0 to 255
(signed) char	8-bit	-128 to +127
unsigned int	16-bit	0 to 65535
(signed) int	16-bit	-32768 to +32767
sbit	1-bit	SFR bit-addressable only
bit	1-bit	RAM bit-addressable only
sfr	8-bit	RAM addresses 80 – FFH only

Write an 8051 C program to send values 00 – FF to port P1.

```
#include <reg51.h>
void main(void)
{
unsigned char z;
for (z=0;z<=255;z++)
P1=z;
}
```

Write an 8051 C program to send hex values for ASCII characters of 0, 1, 2, 3, 4, 5, A, B, C, and D to port P1.

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char mynum[]="012345ABCD";
unsigned char z;
for (z=0;z<=10;z++)
P1=mynum[z];
}
```

Write an 8051 C program to toggle all the bits of P1 continuously.

Solution:

```
//Toggle P1 forever
#include <reg51.h>
void main(void)
{
for (;;)
{
p1=0x55;
p1=0xAA;
}
}
```


The signed char is an 8-bit data type. Use the MSB D7 to represent – or +. Give values from –128 to +127. We should stick with the unsigned char unless the data needs to be represented as signed numbers.

Write an 8051 C program to send values of –4 to +4 to port P1.

Solution:

```
//Singed numbers
#include <reg51.h>
void main(void)
{
char mynum[]={+1,-1,+2,-2,+3,-3,+4,-4};
unsigned char z;
for (z=0;z<=8;z++)
P1=mynum[z];
}
```

The unsigned int is a 16-bit data type. Takes a value in the range of 0 to 65535 (0000 – FFFFH). Define 16-bit variables such as memory addresses. Set counter values of more than 256. Since registers and memory accesses are in 8-bit chunks, the misuse of int variables will result in a larger hex file. Signed int is a 16-bit data type. Use the MSB D15 to represent – or +.We have 15 bits for the magnitude of the number from –32768 to +32767. The bit data type allows to access single bits of bit-addressable memory spaces 20 – 2FH. To access the byte-size SFR registers, we use the sfr data types.

Time Delay using C

There are two way s to create a time delay in 8051 C.

1) Using the 8051 timer
2) Using a simple for loop

Three factors that can affect the accuracy of the delay are.
The 8051 design

a) The number of machine cycle
b) The number of clock periods per machine cycle

The crystal frequency connected to the X1 – X2 input pins. C compiler converts the C statements and functions to Assembly language instructions. Different compilers produce different code.

Write an 8051 C program to toggle bits of P1 continuously forever with some delay.
Solution:
Toggle P1 forever with some delay in between "on" and "off"

```
#include <reg51.h>
void main(void)
{
unsigned int x;
for (;;) //repeat forever
{
P1=0x55;
for (x=0;x<40000;x++); //delay size
//unknown
P1=0xAA;
for (x=0;x<40000;x++);
}
}
```

Write an 8051 C program to toggle bits of P1 ports continuously with a 250 ms.
Solution:

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
{
```

```
while (1) //repeat forever
{
P1=0x55;
MSDelay(250);
P1=0xAA;
MSDelay(250);
}
}
void MSDelay(unsigned int itime)
{
unsigned int i,j;
for (i=0;i<itime;i++)
for (j=0;j<1275;j++);
}
```

I/O Programming in C

LEDs are connected to bits P1 and P2. Write an 8051 C program that shows the count from 0 to FFH (0000 0000 to 1111 1111 in binary) on the LEDs.

Solution:

```
#include <reg51.h>
#define  LED P2;
void main(void)
{
P1=00; //clear P1
LED=0; //clear P2
for (;;) //repeat forever
{
P1++; //increment P1
LED++; //increment P2
}
}
```


Write an 8051 C program to get a byte of data form P1, wait ½ second, and then send it to P2.

Solution:

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
{
unsigned char mybyte;
P1=0xFF; //make P1 input port
while (1)
{
mybyte=P1; //get a byte from P1
MSDelay(500);
```

```
P2=mybyte; //send it to P2
}
}
```

Write an 8051 C program to get a byte of data form P0. If it is less than 100, send it to P1; otherwise, send it to P2.

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char mybyte;
P0=0xFF; //make P0 input port
while (1)
{
mybyte=P0; //get a byte from P0
if (mybyte<100)
P1=mybyte; //send it to P1
else
P2=mybyte; //send it to P2
}
}
```


Write an 8051 C program to monitor bit P1.5. If it is high, send 55H to P0; otherwise, send AAH to P2.

Solution:

```
#include <reg51.h>
sbit mybit=P1^5;
void main(void)
{
mybit=1; //make mybit an input
while (1)
{
if (mybit==1)
P0=0x55;
else
P2=0xAA;
}
}
```

Write an 8051 C program to turn bit P1.5 on and off 50,000 times.

Solution:

```
sbit MYBIT=0x95;
void main(void)
{
```

```
unsigned int z;
for (z=0;z<50000;z++)
{
MYBIT=1;
MYBIT=0;
}
}
```

Write an 8051 C program to get the status of bit P1.0, save it, and send it to P2.7 continuously.

Solution:

```
#include <reg51.h>
sbit inbit=P1^0;
sbit outbit=P2^7;
bit membit; //use bit to declare
//bit- addressable memory
void main(void)
{
while (1)
{
membit=inbit; //get a bit from P1.0
outbit=membit; //send it to P2.7
}
}
```


Logic Operations in C

Logical operators
AND (&&), OR (||), and NOT (!)
Bit-wise operators
AND (&), OR (|), EX-OR (^), Inverter (~),
Shift Right (>>), and Shift Left (<<)
These operators are widely used in software engineering for embedded systems and control

Run the following program on your simulator and examine the results.

Solution:

```
#include <reg51.h>
void main(void)
{
P0=0x35 & 0x0F; //ANDing
P1=0x04 | 0x68; //ORing
P2=0x54 ^ 0x78; //XORing
```

```
P0=~0x55; //inversing
P1=0x9A >> 3; //shifting right 3
P2=0x77 >> 4; //shifting right 4
P0=0x6 << 4; //shifting left 4
}
```

Write an 8051 C program to toggle all the bits of P0 and P2 continuously with a 250 ms delay. Using the inverting and Ex-OR operators, respectively.

Solution:

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
{
P0=0x55;
P2=0x55;
while (1)
{
P0=~P0;
P2=P2^0xFF;
MSDelay(250);
}
}
```


Write an 8051 C program to get bit P1.0 and send it to P2.7 after inverting it.

Solution:

```
#include <reg51.h>
sbit inbit=P1^0;
sbit outbit=P2^7;
bit membit;
void main(void)
{
while (1)
{
membit=inbit; //get a bit from P1.0
outbit=~membit; //invert it and send
//it to P2.7
}
}
```

Data Conversion using C

Write an 8051 C program to convert packed BCD 0x29 to ASCII and display the bytes on P1 and P2.

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char x,y,z;
unsigned char mybyte=0x29;
x=mybyte&0x0F;
P1=x|0x30;
y=mybyte&0xF0;
y=y>>4;
P2=y|0x30;
}
```

Write an 8051 C program to convert ASCII digits of '4' and '7' to packed BCD and display them on P1.

Solution:


```
#include <reg51.h>
void main(void)
{
unsigned char bcdbyte;
unsigned char w='4';
unsigned char z='7';
w=w&0x0F;
w=w<<4;
z=z&0x0F;
bcdbyte=w|z;
P1=bcdbyte;
}
```

Write an 8051 C program to calculate the checksum byte for the data 25H, 62H, 3FH, and 52H.

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char mydata[]={0x25,0x62,0x3F,0x52};
unsigned char sum=0;
unsigned char x;
unsigned char chksumbyte;
for (x=0;x<4;x++)
{
```

```
P2=mydata[x];
sum=sum+mydata[x];
P1=sum;
}
chksumbyte=~sum+1;
P1=chksumbyte;
}
```

Write an 8051 C program to perform the checksum operation to ensure data integrity. If data is good, send ASCII character 'G' to P0.Otherwise send 'B' to P0.

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char mydata[]
={0x25,0x62,0x3F,0x52,0xE8};
unsigned char shksum=0;
unsigned char x;
for (x=0;x<5;x++)
chksum=chksum+mydata[x];
if (chksum==0)
P0='G';
else
P0='B';
}
```


Write an 8051 C program to convert 11111101 (FD hex) to decimal and display the digits on P0, P1 and P2.

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char x,binbyte,d1,d2,d3;
binbyte=0xFD;
x=binbyte/10;
d1=binbyte%10;
d2=x%10;
d3=x/10;
P0=d1;
P1=d2;
P2=d3;
}
```

Accessing Code ROM Space

The 8051 C compiler allocates RAM locations. Bank 0 – addresses 0 – 7. Individual variables addresses 08 and beyond. Array elements – addresses right after variables. Array elements need contiguous RAM locations and that limits the size of the array due to the fact that we have only 128 bytes of RAM for everything. Stack – addresses right after array elements.

Compile and single-step the following program on your 8051 simulator. Examine the contents of the 128-byte RAM space to locate the ASCII values.

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char mynum[]="ABCDEF"; //RAM space
unsigned char z;
for (z=0;z<=6;z++)
P1=mynum[z];
}
```

Write, compile and single-step the following program on your 8051 simulator. Examine the contents of the code space to locate the values.

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char mydata[100]; //RAM space
unsigned char x,z=0;
for (x=0;x<100;x++)
{
z--;
mydata[x]=z;
P1=z;
}
}
```


Data Serialization

Serializing data is a way of sending a byte of data one bit at a time through a single pin of microcontroller using the serial port. Transfer data one bit a time and control the sequence of data and spaces in between them is known as serialization. In many new generations of devices such as LCD, ADC, and ROM the serial versions are becoming popular since they take less space on a PCB.

Write a C program to send out the value 44H serially one bit at a time via P1.0. The LSB should go out first.

Solution:

```
#include <reg51.h>
sbit P1b0=P1^0;
sbit regALSB=ACC^0;
void main(void)
{
unsigned char conbyte=0x44;
unsigned char x;
ACC=conbyte;
for (x=0;x<8;x++)
{
P1b0=regALSB;
ACC=ACC>>1;
}
}
```

Write a C program to send out the value 44H serially one bit at a time via P1.0. The MSB should go out first.

Solution:

```
#include <reg51.h>
sbit P1b0=P1^0;
sbit regAMSB=ACC^7;
void main(void)
{
unsigned char conbyte=0x44;
unsigned char x;
ACC=conbyte;
for (x=0;x<8;x++)
{
P1b0=regAMSB;
ACC=ACC<<1;
}
}
```

Write a C program to bring in a byte of data serially one bit at a time via P1.0. The LSB should come in first.

Solution:

```
#include <reg51.h>
sbit P1b0=P1^0;
sbit ACCMSB=ACC^7;
bit membit;
void main(void)
```

```
{
unsigned char x;
for (x=0;x<8;x++)
{
membit=P1b0;
ACC=ACC>>1;
ACCMSB=membit;
}
P2=ACC;
}
```

Write a C program to bring in a byte of data serially one bit at a time via P1.0. The MSB should come in first.

Solution:

```
#include <reg51.h>
sbit P1b0=P1^0;
sbit regALSB=ACC^0;
bit membit;
void main(void)
{
unsigned char x;
for (x=0;x<8;x++)
{
membit=P1b0;
ACC=ACC<<1;
regALSB=membit;
}
P2=ACC;
}
```

Module No: 4

8051 SERIAL PORT PROGRAMMING IN ASSEMBLY AND C

Computers transfer data in two ways: parallel and serial. In parallel data transfers, often 8 or more lines (wire conductors) are used to transfer data to a device that is only a few feet away. Examples of parallel transfers are printers and hard disks; each uses cables with many wire strips. Although in such cases a lot of data can be transferred in a short amount of time by using many wires in parallel, the distance cannot be great. To transfer to a device located many meters away, the serial method is used. In serial communication, the data is sent one bit at a time, in contrast to parallel communication, in which the data is sent a byte or more at a time. The 8051 has serial communication capability built into it, thereby making possible fast data transfer using only a few wires.

BASICS OF SERIAL COMMUNICATION

When a microcontroller communicates with the outside world, it provides the data in byte-sized chunks. In some cases, such as printers, the information is simply grabbed from the 8-bit data bus and presented to the 8-bit data bus of the printer. This can work only if the cable is not too long, since long cables diminish and even distort signals. Furthermore, an 8-bit data path is expensive. For these reasons, serial communication is used for transferring data between two systems located at distances of hundreds of feet to millions of miles apart. Figure 10-1 diagrams serial versus parallel data transfers.

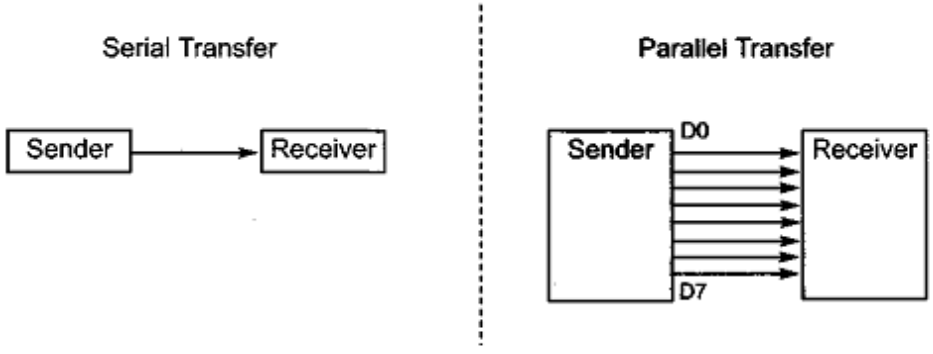


Fig: Serial vs Parallel transfer

The fact that serial communication uses a single data line instead of the 8-bit data line of parallel communication not only makes it much cheaper but also enables two computers located in two

different cities to communicate over the telephone. For serial data communication to work, the byte of data must be converted to serial bits using a parallel-in-serial-out shift register; then it can be transmitted over a single data line. This also means that at the receiving end there must be a serial-in-parallel-out shift register to receive the serial data and pack them into a byte. Of course, if data is to be transferred on the telephone line, it must be converted from Os and 1s to audio tones, which are sinusoidal-shaped signals. This conversion is performed by a peripheral device called a *modem*, which stands for "modulator/demodulator."

When the distance is short, the digital signal can be transferred as it is on a simple wire and requires no modulation. This is how IBM PC keyboards transfer data to the motherboard. However, for long-distance data transfers using communication lines such as a telephone, serial data communication requires a modem to *modulate* (convert from Os and 1 s to audio tones) and *demodulate* (converting from audio tones to Os and 1 s).

Serial data communication uses two methods, Asynchronous and Synchronous. The *synchronous* method transfers a block of data (characters) at a time, while the *asynchronous* method transfers a single byte at a time. It is possible to write software to use either of these methods, but the programs can be tedious and long. For this reason, there are special IC chips made by many manufacturers for serial data communications. These chips are commonly referred to as UART (universal asynchronous receiver-transmitter) and USART (universal synchronous-asynchronous receiver-transmitter). The 8051 chip has a built-in UART.

Half- and full-duplex transmission

In data transmission if the data can be transmitted and received, it is a *duplex* transmission. This is in contrast to *simplex* transmissions such as with printers, in which the computer only sends data. Duplex transmissions can be half or full duplex, depending on whether or not the data transfer can be simultaneous. If data is transmitted one way at a time, it is referred to as *half duplex.* If the data can go both ways at the same time, it *is full duplex.* Of course, full duplex requires two wire conductors for the data lines (in addition to the signal ground), one for transmission and one for reception, in order to transfer and receive data simultaneously.

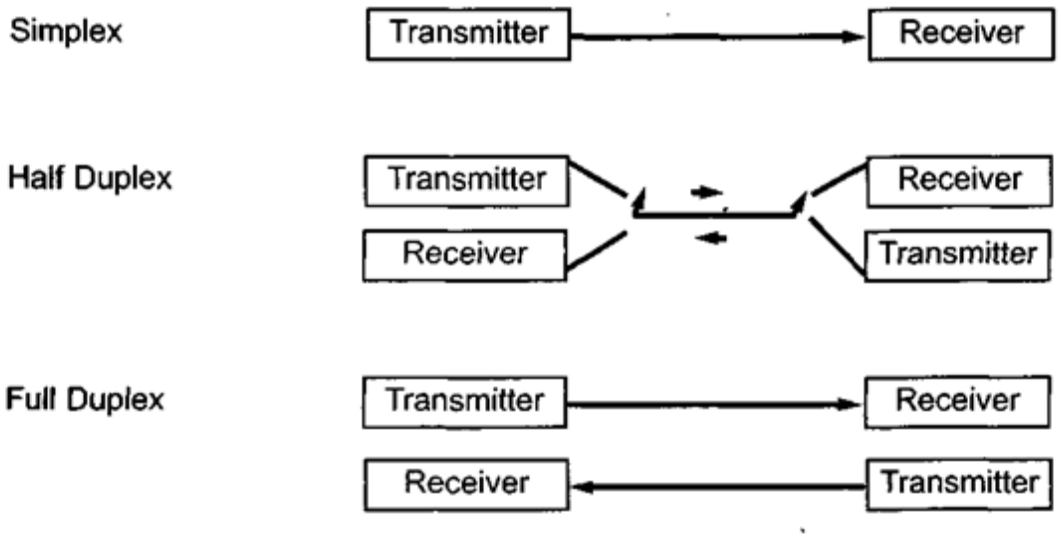


Fig: Simplex, duplex transfer

Asynchronous serial communication and data framing

Asynchronous serial data communication is widely used for character-oriented transmissions, while block-oriented data transfers use the synchronous method. In the asynchronous method, each character is placed between start and stop bits. This is called *framing.* In data framing for asynchronous communications, the data, such as ASCII characters, are packed between a start bit and a stop bit. The start bit is always one bit, but the stop bit can be one or two bits. The start bit is always a 0 (low) and the stop bit(s) is 1 (high). For example, look at Figure in which the ASCII character "A" (8-bit binary 0100 0001) is framed between the start bit and a single stop bit. Notice that the LSB is sent out first.

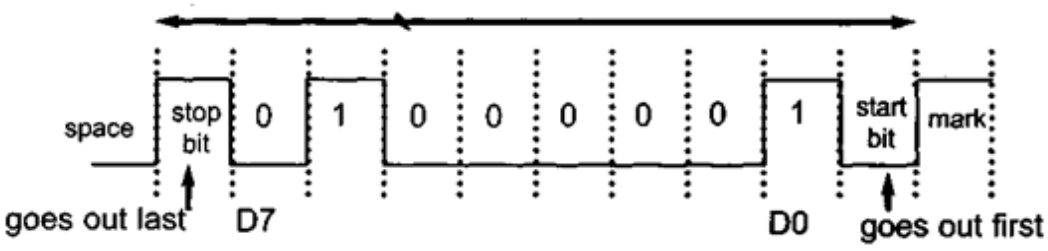


Fig: Data framing

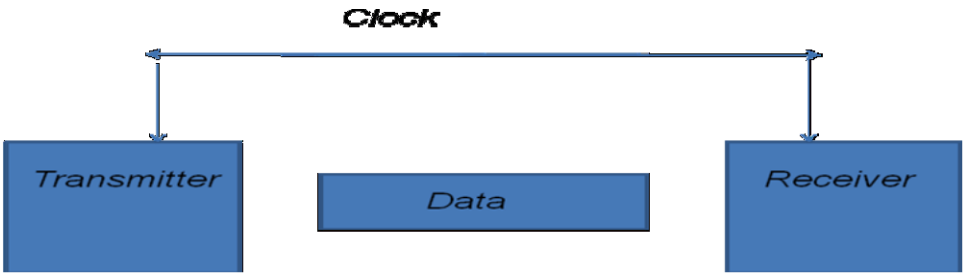
when there is no transfer, the signal is 1 (high), which is referred to as *mark.* The 0 (low) is referred to as *space.* Notice that the transmission begins with a start bit followed by DO, which

is the LSB, then the rest of the bits until the MSB (D7), and finally, the one stop bit indicating the end of the character "A".

In asynchronous serial communications, peripheral chips and modems can be programmed for data that is 7 or 8 bits wide. This is in addition to the number of stop bits, 1 or 2. While in older systems ASCII characters were 7-bit, in recent years, due to the extended ASCII characters, 8-bit data has become common. In some older systems, due to the slowness of the receiving mechanical device, two stop bits were used to give the device sufficient time to organize itself before transmission of the next byte. In modern PCs however, the use of one stop bit is standard. Assuming that we are transferring a text file of ASCII characters using 1 stop bit, we have a total of 10 bits for each character: 8 bits for the ASCII code, and 1 bit each for the start and stop bits. Therefore, for each 8-bit character there are an extra 2 bits, which gives 20% overhead.

In some systems, the parity bit of the character byte is included in the data frame in order to maintain data integrity. This means that for each character (7- or 8-bit, depending on the system) we have a single parity bit in addition to start and stop bits. The parity bit is odd or even. In the case of an odd-parity bit the number of data bits, including the parity bit, has an odd number of 1s. Similarly, in an even-parity bit system the total number of bits, including the parity bit, is even. For example, the ASCII character "A", binary 0100 0001, has 0 for the even-parity bit. UART chips allow programming of the parity bit for odd-, even-, and no-parity options.

In Asynchronous Serial Data Communication Pins TxD (P3.1) and RxD (P3.0) are used for transmitting and receiving the data serially. Figure below shows synchronous serial data communication which uses a common clock for synchronization of transmitter and receiver.



Data transfer rate

The rate of data transfer in serial data communication is stated in *bps* (bits per second). Another widely used terminology for bps is *baud rate*. However, the baud and bps rates are not

necessarily equal. This is due to the fact that baud rate is the modem terminology and is defined as the number of signal changes per second. In modems a single change of signal, sometimes transfers several bits of data. As far as the conductor wire is concerned, the baud rate and bps are the same, and for this reason in this book we use the terms bps and baud interchangeably.

The data transfer rate of a given computer system depends on communication ports incorporated into that system. For example, the early IBM PC/XT could transfer data at the rate of 100 to 9600 bps. In recent years, however, Pentium-based PCs transfer data at rates as high as 56K bps. It must be noted that in asynchronous serial data communication, the baud rate is generally limited to 100,000 bps.

RS232 standards

To allow compatibility among data communication equipment made by various manufacturers, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. In 1963 it was modified and called RS232A. RS232B and RS232C were issued in 1965 and 1969, respectively. Today, RS232 is the most widely used serial I/O interfacing standard. This standard is used in PCs and numerous types of equipment. However, since the standard was set long before the advent of the TTL logic family, its input and output voltage levels are not TTL compatible. In RS232, a 1 is represented by -3 to -25 V, while a 0 bit is +3 to +25 V, making -3 to +3 undefined. For this reason, to connect any RS232 to a micro controller system we must use voltage converters such as MAX232 to convert the TTL logic levels to the RS232 voltage levels, and vice versa. MAX232 IC chips are commonly referred to as line drivers.

RS232 pins

Figure shows DB 25 connector. But IBM introduced the DB-9 version of the serial I/O standard, which uses 9 pins only,

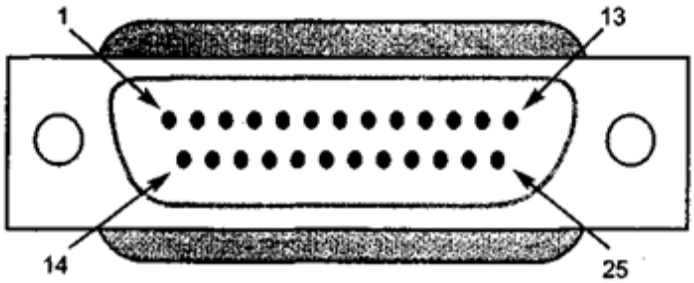


Fig: DB 25 connector

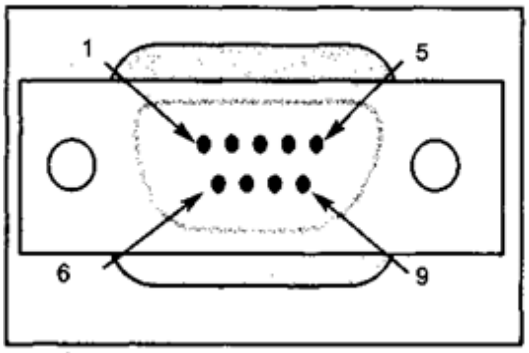


Fig: DB 9 connector

RS232 Pins (DB-25)

Pin	Description
1	Protective ground
2	Transmitted data (TxD)
3	Received data (RxD)
4	Request to send ($\overline{\text{RTS}}$)
5	Clear to send ($\overline{\text{CTS}}$)
6	Data set ready ($\overline{\text{DSR}}$)
7	Signal ground (GND)
8	Data carrier detect ($\overline{\text{DCD}}$)
9/10	Reserved for data testing
11	Unassigned
12	Secondary data carrier detect
13	Secondary clear to send
14	Secondary transmitted data
15	Transmit signal element timing
16	Secondary received data
17	Receive signal element timing
18	Unassigned
19	Secondary request to send
20	Data terminal ready ($\overline{\text{DTR}}$)
21	Signal quality detector
22	Ring indicator
23	Data signal rate select
24	Transmit signal element timing
25	Unassigned

fig: Pin description of DB-25

Pin	Description
1	Data carrier detect (DCD)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready (DSR)
7	Request to send (RTS)
8	Clear to send (CTS)
9	Ring indicator (RI)

fig: Pin description of DB-9

Data Communication

Current terminology classifies data communication equipment as DTE (data terminal equipment) or DCE (data communication equipment). DTE refers to terminals and computers that send and receivedata, while DCE refers to communication equipment, such as modems, that are responsible for transferring the data.

The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD, RxD, and ground, Notice in that figure that the RxD and TxD pins are interchanged.

Examining RS232 handshaking signals

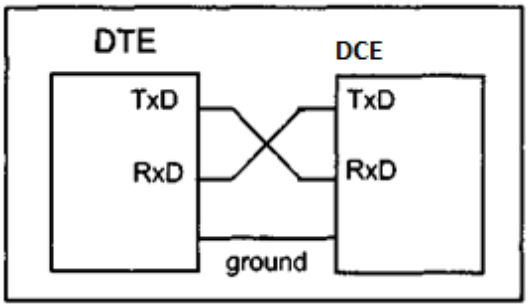


Fig:Null modem connection

The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD,RxD, and ground, Notice in that figure that the RxD and TxD pins are interchanged.

Examining RS232 handshaking signals

To ensure fast and reliable data transmission between two devices, the data transfer must be coordinated. Just as in the case of the printer, because the receiving device in serial data communication may have no room for the data, there must be a way to inform the sender to stop sending data. Many of the pins of the RS-232 connector are used for handshaking signals.

1.DTR(data terminal ready). When a terminal (or a PC COM port) is turned on, after going through a self-test, it sends out signal DTR to indicate that it is ready for communication. If there is something wrong with the COM port, this signal will not be activated. This is an active-low signal and can be used to inform the modem that the computer is alive and kicking. This is an output pin from DTE (PC COM port) and an input to the modem.DSR (data set ready).

2. When DCE (modem) is turned on and has gone through the self-test, it asserts DSR to indicate that it is ready to communicate. Thus, it is an output from the modem (DCE) and input to the PC (DTE). This is an active- low signal. If for any reason the modem cannot make a connection to the telephone, this signal remains inactive, indicating to the PC (or terminal) that it cannot accept or send data.

3. RTS (request to send). When the DTE device (such as a PC) has a byte to transmit, it asserts RTS to signal the modem that it has a byte of data to transmit. RTS is an active-low output from the DTE and an input to the modem.

4.CTS (clear to send). In response to RTS, when the modem has room for storing the data it is to receive, it sends out signal CTS to the DTE (PC) to indicate that it can receive the data now. This input signal to the DTE is used by the DTE to start transmission.

5. DCD (carrier detect, or DCD, data carrier detect). The modem asserts signal DCD to inform the DTE (PC) that a valid carrier has been detected and that contact between it and the other modem is established. Therefore, DCD is an output from the modern and an input to the PC (DTE).

6.RI (ring indicator). An output from the modem (DCE) and an input to a PC (DTE) indicates that the telephone is ringing. It goes on and off in synchronization with the ringing sound. Of the six handshake signals, this is the least often used, due to the fact that modems take care of

answering the phone.

However, if the PC is in charge of answering the phone, this signal can be used.

While signals DTE and DSR are used by the PC and modem, respectively, to indicate that they are alive and well, it is RTS and CTS that actually control the flow of data. When the PC wants to send data it asserts RTS, and in response, if the modem is ready (has room) to accept the data, it sends back CTS. If, for lack of room, the modem does not activate CTS, the PC will deassert DTR and try again. RTS and CTS are also referred to as hardware control flow signals.

Serial Interface

The serial port of 8051 is full duplex, i.e., it can transmit and receive simultaneously. The register SBUF is used to hold the data. The special function register SBUF is physically two registers. One is, write-only and is used to hold data to be transmitted out of the 8051 via TXD. The other is, read-only and holds the received data from external sources via RXD. Both mutually exclusive registers have the same address 099H.

Serial Port Control Register (SCON)

SCON Register

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Serial control register: SCON

SM0, SM1 : Serial port mode specifier

Register SCON controls serial data communication. Address: 098H (Bit addressable)

Mode select bits

SM0	SM1	MODE
0	0	Mode0
0	1	Mode1
1	0	Mode2
1	1	M ode3

SM2:used for multiprocessor communication.

REN: set or cleared by software to enable/disable reception.

TB8: Transmitted bit 8,not widely used.

RB8:Received bit 8.

TI:Transmit interrupt flag –set by the hardware at the beginning of the stop bit in mode 1, must be cleared by software.

RI: Receive interrupt flag –set by the hardware halfway through the stop bit time in mode1, must be cleared by software.

Programming the 8051 to transfer data serially

In programming the 8051 to transfer character bytes serially, the following steps must be taken.

1. The TMOD register is loaded with the value 20H, indicating the use of Timer 1 in mode 2 (8-bit auto-reload) to set the baud rate.
2. The TH1 is loaded with one of the values in Table 10-4 to set the baud rate for serial data transfer (assuming XTAL = 11.0592 MHz).
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8-bit data is framed with start and stop bits.
4. TR1 is set to 1 to start Timer 1.
5. TI is cleared by the "CLR TI" instruction.
6. The character byte to be transferred serially is written into the SBUF register.
7. The TI flag bit is monitored with the use of the instruction " JNB TI, xx" to see if the character has been transferred completely.
8. To transfer the next character, go to Step 5.

Importance of the Tl flag

To understand the importance of the role of TI, look at the following sequence of steps that the 8051 goes through in transmitting a character via TxD.

1. The byte character to be transmitted is written into the SBUF register.
2. The start bit is transferred.
3. The 8-bit character is transferred one bit at a time.

4. The stop bit is transferred. It is during the transfer of the stop bit that the 8051 raises the TI flag (TI =1), indicating that the last character was transmitted and it is ready to transfer the next character.
5. By monitoring the TI flag, we make sure that we are not overloading the SBUF register. If we write another byte into the SBUF register before TI is raised, the untransmitted portion of the previous byte will be lost.

In other words, when the 8051 finishes transferring a byte, it raises the TI flag to indicate it is ready for the next character. 6. After SBUF is loaded with a new byte, the TI flag bit must be forced to 0 by the "CLR TI" instruction in order for this new byte to be transferred.

From the above discussion we conclude that by checking the TI flag bit, we know whether or not the 8051 is ready to transfer another byte. More importantly, it must be noted that the TI flag bit is raised by the 8051 itself when it finishes the transfer of data, whereas it must be cleared by the programmer with an instruction such as "CLR TI". It also must be noted that if we write a byte into SBUF before the TI flag bit is raised, we risk the loss of a portion of the byte being transferred. The TI flag bit can be checked by the instruction "JNB TI, . . ." .

Programming the 8051 to receive data serially

In the programming of the 8051 to receive character bytes serially, the following steps must be taken.

1. The TMOD register is loaded with the value 20H, indicating the use of Timer 1 in mode 2 (8-bit auto-reload) to set the baud rate.
2. TH1 is loaded with one of the values in Table 10-4 to set the baud rate (assum ing XTAL = 11.0592MHz).
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where 8-bit data is framed with start and stop bits and receive enable is turned on.
4. TR1 is set to 1 to start Timer 1.RI is cleared with the "CLR RI" instruction.
5. The RI flag bit is monitored with the use of the instruction "JNB RI, xx" to see if an entire character has been received yet. When RI is raised, SBUF has the byte. Its contents are moved into a safe place.

6. To receive the next character, go to Step 5.

Importance of the Rl flag bit

In receiving bits via its RxD pin, the 8051 goes through the following steps.

1. It receives the start bit indicating that the next bit is the first bit of the charac ter byte it is about to receive.
2. The 8-bit character is received one bit at time. When the last bit is received, a byte is formed and placed in SBUF.
3. The stop bit is received. When receiving the stop bit the 8051 makes RI = 1, indicating that an entire character byte has been received and must be picked up before it gets overwritten by an incoming character.
4. By checking the RI flag bit when it is raised, we know that a character has been received and is sitting in the SBUF register. We copy the SBUF contents to a safe place in some other register or memory before it is lost.
5. After the SBUF contents are copied into a safe place, the RI flag bit must be forced to 0 by the "CLR RI" instruction in order to allow the next received character byte to be placed in SBUF. Failure to do this causes loss of the received character.

Data Transmission

Transmission of serial data begins at any time when data is written to SBUF. Pin P3.1 (Alternate function bit TXD) is used to transmit data to the serial data network. TI is set to 1 when data has been transmitted. This signifies that SBUF is empty so that another byte can be sent.

Data Reception

Reception of serial data begins if the receive enable bit is set to 1 for all modes. Pin P3.0 (Alternate function bit RXD) is used to receive data from the serial data network. Receive interrupt flag, RI, is set after the data has been received in all modes. The data gets stored in SBUF register from where it can be read.

Serial Data Transmission Modes:

Mode-0**:** In this mode, the serial port works like a shift register and the data transmission works synchronously v to the external circuitry for synchronization. The shift frequency or baud rate is always 1/12 of the oscillator freq

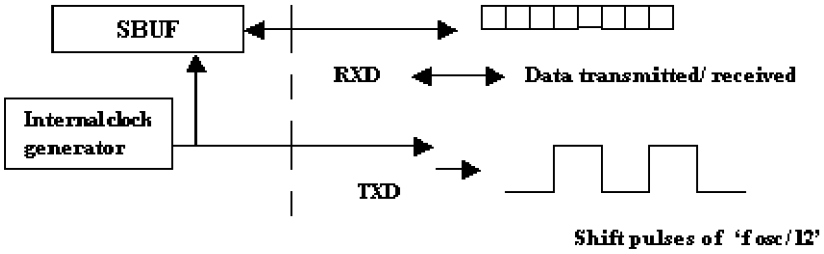
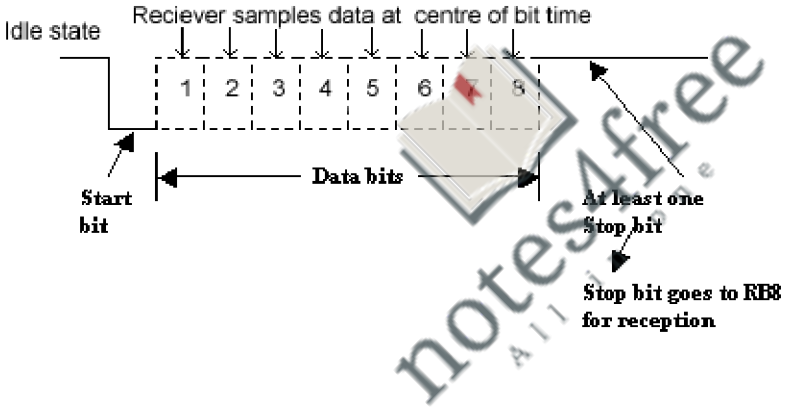


Fig : Data transmission/reception in Mode-0

In mode-1, the serial port functions as a standard Universal Asynchronous Receiver Transmitter (UART) mode received, the stop bit goes into RB8 in the special function register SCON. The baud rate is variable.

The following figure shows the way the bits are transmitted/ received.



Bit time= $1/f_{baud}$

In receiving mode, data bits are shifted into the receiver at the programmed baud rate.

The data word (8-bits) will be loaded to SBUF if the following conditions are true.

- RI must be zero. (i.e., the previously received byte has been cleared from SBUF)
- Mode bit SM2 = 0 or stop bit = 1.

After the data is received and the data byte has been loaded into SBUF, RI becomes one.

Mode-1 baud rate generation:

Timer-1 is used to generate baud rate for mode-1 serial communication by using overflow flag of the timer to determine the baud frequency. Timer-1 is used in timer mode-2 as an auto-reload 8-bit timer. The data rate is generated by timer-1 using the following formula.

Where,

SMOD is the

f_{osc} is the crystal oscillator frequency of the microcontroller

It can be noted that f_{osc}/ (12 X [256- (TH1)]) is the timer overflow frequency in timer mode-2, which is the auto-reload mode.

If timer-1 is not run in mode-2, then the baud rate is,

Timer-1 can be run using the internal clock, fosc/12 (timer mode) or from any external source via pin T1 (P3.5) (Counter mode).

Serial Data Mode-2 - Multiprocessor Mode :

In this mode 11 bits are transmitted through TXD or received through RXD. The various bits are as follows: a start bit (usually '0'), 8 data bits (LSB first), a programmable 9 th (TB8 or RB8)bit and a stop bit (usually '1').

While transmitting, the 9 th data bit (TB8 in SCON) can be assigned the value '0' or '1'. For example, if the information of parity is to be transmitted, the parity bit (P) in PSW could be moved into TB8. On reception of the data, the 9 th bit goes into RB8 in 'SCON', while the stop bit is ignored. The baud rate is programmable to either 1/32 or 1/64 of the oscillator frequency.

Mode-3 - Multi processor mode with variable baud rate :

In this mode 11 bits are transmitted through TXD or received through RXD. The various bits are: a start bit (usually '0'), 8 data bits (LSB first), a programmable 9 th bit and a stop bit (usually '1').

Mode-3 is same as mode-2, except the fact that the baud rate in mode-3 is variable (i.e., just as in mode-1).

$f_{baud} = (2^{SMOD}/32) * (f_{osc} / 12 (256-TH1))$.

This baudrate holds when Timer-1 is programmed in Mode-2.

Programming the 8051 to transfer data serially

Write a program for the 8051 to transfer letter "A" serially at 4800baud, continuously.

```
MOV TMOD,#20H ;timer 1, mode 2
MOV TH1,#-6 ;4800 baud rate
MOV SCON,#50H ;8-bit,1 stop,REN enabled
SETB TR1 ;start timer 1
AGAIN: MOV SBUF,#"A" ;letter "A" to be transferred
HERE: JNB TI,HERE ;wait for the last bit
CLR TI ;clear TI for next char
SJMP AGAIN ;keep sending A
```

Write a program to transfer the message "YES" serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously.

```
MOV TMOD,#20H ;timer 1, mode 2
MOV TH1,#-3 ;9600 baud
MOV SCON,#50H
SETB TR1
AGAIN: MOV A,#"Y" ;transfer "Y"
ACALL TRANS
MOV A,#"E" ;transfer "E"
ACALL TRANS
MOV A,#"S" ;transfer "S"
ACALL TRANS
SJMP AGAIN ;keep doing it
```

;serial data transfer subroutine

```
TRANS:                          MOV
HERE:           JNB             TI,HERE
CLR                 TI                      ;ge
RET
```

Baud Rates in the 8051

- Timer 1, mode 2 (8-bit, auto-reload)

- Define TH1 to set the baud rate.

XTAL = 11.0592 MHz

The system frequency = 11.0592 MHz / 12 = 921.6 kHz

Timer 1 has 921.6 kHz/ 32 = 28,800 Hz as source.

TH1=FDH means that UART sends a bit every 3 timer source.

Baud rate = 28,800/3= 9,600 Hz

Example

With XTAL = 11.0592 MHz, find the TH1 value needed to have the following baud rates. (a) 9600 (b) 2400 (c) 1200

Solution:

With XTAL = 11.0592 MHz, we have:

The frequency of system clock = 11.0592 MHz / 12 = 921.6 kHz

The frequency sent to timer 1 = 921.6 kHz/ 32 = 28,800 Hz

(a) 28,800 / 3 = 9600 where -3 = FD (hex) is loaded into TH1

(b) 28,800 / 12 = 2400 where -12 = F4 (hex) is loaded into TH1

(c) 28,800 / 24 = 1200 where -24 = E8 (hex) is loaded into TH1

Registers Used in Serial Trans1fer Circuit

SUBF (Serial data buffer)

SCON (Serial control register)

PCON (Power control register)

SBUF Register

Serial data register: **SBUF**

MOV SBUF,#'A' ;put char 'A' to transmit

MOV SBUF,A ;send data from A

MOV A, SUBF ;receive and copy to A

An 8-bit register

Set the usage mode for two timers

For a byte of data to be transferred via the TxD line, it must be placed in the SBUF.

SBUF holds the byte of data when it is received by the 8051's RxD line.

Write a C program for 8051 to transfer the letter "A" serially at 4800 baud continuously. Use 8-bit data and 1 stop bit.

Solution:

```
#include <reg51.h>
void main(void){
TMOD=0x20; //use Timer 1, mode 2
TH1=0xFA; //4800 baud rate
SCON=0x50;
TR1=1;
while (1) {
SBUF='A'; //place value in buffer
while (TI==0);
TI=0;
}
}
```

Write an 8051 C program to transfer the message "YES" serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously.

Solution:

```
#include <reg51.h>
void SerTx(unsigned char);
void main(void){
TMOD=0x20; //use Timer 1, mode 2
TH1=0xFD; //9600 baud rate
SCON=0x50;
TR1=1; //start timer
while (1) {
SerTx('Y');
SerTx('E');
SerTx('S');
}
```

```
}
void SerTx(unsigned char x){
SBUF=x; //place value in buffer
while (TI==0); //wait until transmitted
TI=0;
}
```

Program the 8051 in C to receive bytes of data serially and put them in P1. Set the baud rate at 4800, 8-bit data, and 1 stop bit.

Solution:

```
#include <reg51.h>
void main(void){
unsigned char mybyte;
TMOD=0x20; //use Timer 1, mode 2
TH1=0xFA; //4800 baud rate
SCON=0x50;
TR1=1; //start timer
while (1) { //repeat forever
while (RI==0); //wait to receive
mybyte=SBUF; //save value
P1=mybyte; //write value to port
RI=0;
}
}
```


Write an 8051 C Program to send the two messages "Normal Speed" and "High Speed" to the serial port. Assuming that SW is connected to pin P2.0, monitor its status and set the baud rate as follows:

SW = 0, 28,800 baud rate, SW = 1, 56K baud rate. Assume that XTAL = 11.0592 MHz for both cases.

Solution:

```
#include <reg51.h>
sbit MYSW=P2^0; //input switch
void main(void){
unsigned char z;
unsigned char Mess1[]="Normal Speed";
unsigned char Mess2[]="High Speed";
TMOD=0x20; //use Timer 1, mode 2
TH1=0xFF; //28800 for normal
SCON=0x50;
TR1=1; //start timer
if(MYSW==0) {
for (z=0;z<12;z++) {
SBUF=Mess1[z]; //place value in buffer
while(TI==0); //wait for transmit
TI=0;
}
}
else {
PCON=PCON|0x80; //for high speed of 56K
for (z=0;z<10;z++) {
SBUF=Mess2[z]; //place value in buffer
while(TI==0); //wait for transmit
TI=0;
}
}
}
```


8051 Interrupt

A computer has only two ways to determine the conditions that exist in internal and external circuits. One method uses software instructions that jump to subroutines on the status of flags and port pins. The second method responds to hardware signals, called interrupts that force the

program to call a subroutine. Most applications of microcontroller involve responding to events quickly enough to control the environment that generates the events termed real-time programming.

An interrupt is an external or internal event that interrupts the microcontroller to inform it that a device needs its service. A single microcontroller can serve several devices by two ways.

1) Interrupts

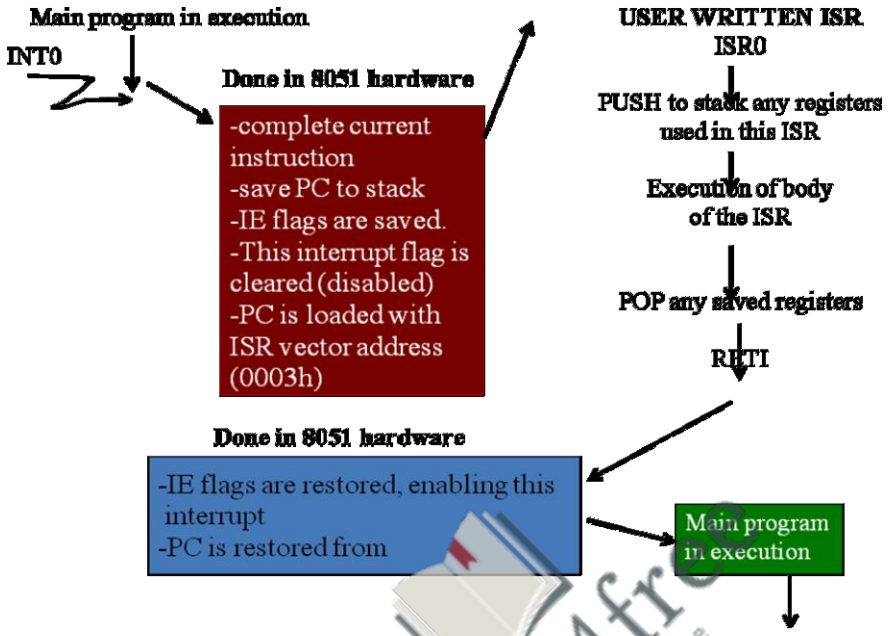
Whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal. Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device. The program which is associated with the interrupt is called the interrupt service routine(ISR) or interrupt handle

2) Polling

The microcontroller continuously monitors the status of a given device. When the conditions met, it performs the service. After that, it moves on to monitor the next device until everyone is serviced. Polling can monitor the status of several devices and serve each of them as certain conditions are met. The polling method is not efficient, since it wastes much of the microcontroller's time by polling devices that do not need service. ex. JNB TF,target.

The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time). Each devices can get the attention of the microcontroller based on the assigned Priority. For the polling method, it is not possible to assign priority since it checks all devices in a round-robin fashion. The microcontroller can also ignore (mask) a device request for service. This is not possible for the polling method. For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler. When an interrupt is invoked, the microcontroller runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called interrupt vector table.

Upon activation of an interrupt, the microcontroller goes through the following steps



1. It finishes the instruction it is executing and saves the address of the next instruction (PC) on the stack .
2. It also saves the current status of all the interrupts internally (i.e: not on the stack)
3. It jumps to a fixed location in memory, called the interrupt vector table, that holds the address of the ISR
4. The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine which is RETI (return from interrupt).
5. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top two bytes of the stack into the PC. Then it starts to execute from that address.

Basically there are Six interrupts allocated as follows

1) Reset – power-up reset

2) Two interrupts are set aside for the timers: one for timer 0 and one for timer 1
3) Two interrupts are set aside for hardware. external interrupts P3.2 and P3.3 are for the external hardware interrupts INT0 (or EX1), and INT1 (or EX2).
4) Serial communication has a single interrupt that belongs to both receive and transfer.

Interrupt vector table

Interrupt	ROM Location (hex)	Pin
Reset	0000	9
External HW (INT0)	0003	P3.2 (12)
Timer 0 (TF0)	000B	
External HW (INT1)	0013	P3.3 (13)
Timer 1 (TF1)	001B	
Serial COM (RI and TI)	0023	

Upon reset, all interrupts are disabled (masked), meaning that none will be responded by the microcontroller if they are activated. The interrupts must be enabled by software in order for the microcontroller to respond to them.

What is difference between RET and RETI Instructions.

The instruction RET is a return from a function or a subroutine while RETI is return from an interrupt. The RETI instruction is executed at the end of interrupt subroutine. After the execution of the RETI instruction the PC address will be restored from the stack.

Comparison between Call instruction and the Interrupt action can be as:

Call Instruction	Interrupt Action
Completely under the control of programmer	Occurs at any time in the program
Placed in the program by the programmer	Enabled by the programmer
Execution is determined by where it is placed in the program	Calls the subroutine at any time and any place the program is executed

Table 1: Comparison of Call Instruction and Interrupt Action

Interrupt Enable (IE) SFR:

This is a bit addressable SFR with byte address A8H. The bits and addresses are shown in table 2. The bits are explained below.

IE.7	IE.6	IE.5	IE.4	IE.3	IE.2	IE.1	IE.0
EA	-	ET2	ES	ET1	EX1	ET0	EX0
AFH	AEH	ADH	ACH	ABH	AAH	A9H	A8H

Table 2: Interrupt Enable SFR

EA: This bit is a global interrupt enable/disable bit. When set to 1, it permits individual interrupts to be enable by their respective enable bits. When 0, it disables all interrupts.

IE.6: Not implemented

ET2: Reserved for future use.

ES: Enable serial port interrupt. Set to 1 by program to enable serial port interrupt. Cleared to 0 to disable serial port interrupt.

ET1: Enable (=1)/disable (=0) timer 1 interrupt

EX1: Enable (=1)/disable (=0) external interrupt 1

ET0: Enable (=1)/disable (=0) timer 0 interrupt

ET1: Enable (=1)/disable (=0) external interrupt 0

Interrupt Priority (IP):

This a bit addressable register, with byte address B8H. The addresses are shown in table 3. The priority of the interrupts is determined by the bits of IP SFR. The bits which are set to 1, have a high priority and bits with 0 have low priority. Interrupts with high priority can interrupt another interrupt with low priority. The lower priority interrupt is serviced after higher priority interrupt is finished.

IP.7	IP.6	IP.5	IP.4	IP.3	IP.2	IP.1	IP.0
-	-	PT2	PS	PT1	PX1	PT0	PX0
-	-	-	BC	BB	BA	B9	B8

Table 3: Interrupt Priority (IP)

IP.7 & IP.6: Not implemented

IP.5: Reserved for future use

PS: Serial port priority interrupt

PT1: Priority of timer 1 interrupt

PX1: Priority of external interrupt 1

PT0: Priority of timer 0 interrupt

PX0: Priority of external interrupt 0

When two or more interrupts have the same priority the microcontroller has its own ranking of providing service which is as below:

- External Interrupt 0 (P3.2)
- Timer 0 Overflow
- External Interrupt 0 (P3.3)
- Timer 1 Overflow
- Serial port

To enable an interrupt, we take the following steps:

1. Bit D7 of the IE register (EA) must be set to high to allow the rest of register to take effect

2. The value of EA. If EA = 1, interrupts are enabled and will be responded to if their corresponding bits in IE are high. If EA = 0, no interrupt will be responded to, even if the associated bit in the IE register is high.

Example

Show the instructions to (a) enable the serial interrupt, timer 0 interrupt, and external hardware interrupt 1 (EX1),and (b) disable (mask) the timer 0 interrupt, then (c) show how to disable all the interrupts with a single instruction.

Solution:

(a) MOV IE,#10010110B ;enable serial,;timer 0, EX1 Another way to perform the same manipulation is SETB IE.7 ;EA=1, global enable

SETB IE.4 ;enable serial interrupt

SETB IE.1 ;enable Timer 0 interrupt

SETB IE.2 ;enable EX1

(b) CLR IE.1 ;mask (disable) timer 0

;interrupt only

(c) CLR IE.7 ;disable all interrupts

The timer flag (TF) is raised when the timer rolls over .In polling TF, we have to wait until the TF is raised. The problem with this method is that the microcontroller is tied down while waiting for TF to be raised, and cannot do anything else.

Using interrupts solves this problem and, avoids tying down the controller. If the timer interrupt in the IE register is enabled, whenever the timer rolls over, TF is raised, and the microcontroller is interrupted in whatever it is doing, and jumps to the interrupt vector table to service the ISR. In this way, the microcontroller can do other until it is notified that the timer has rolled over.

Ex**ample**

Write a program that continuously get 8-bit data from P0 and sends it to P1 while simultaneously creating a square wave of 200 µs period on pin P2.1. Use timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

Solution:

We will use timer 0 in mode 2 (auto reload). TH0 = 100/1.085 us = 92;--upon wake-up go to main, avoid using ;memory allocated to Interrupt Vector Table

ORG 0000H

LJMP MAIN ;by-pass interrupt vector table;--ISR for timer 0 to generate square wave

ORG 000BH ;Timer 0 interrupt vector table

```
CPL P2.1 ;toggle P2.1 pin
RETI ;
The main program for initialization
ORG 0030H ;after vector table space
MAIN: MOV TMOD,#02H ;Timer 0, mode 2
MOV P0,#0FFH ;make P0 an input port
MOV TH0,#-92 ;TH0=A4H for -92
MOV IE,#82H ;IE=10000010 (bin) enable;Timer 0
SETB TR0 ;Start Timer 0
BACK: MOV A,P0 ;get data from P0
MOV P1,A ;issue it to P1
SJMP BACK ;keep doing it loop;unless interrupted by TF0
END
```

Example

Rewrite above Example to create a square wave that has a high portion of 1085 us and a low portion of 15 us. Assume XTAL=11.0592MHz. Use timer 1.

Solution:

Since 1085 us is 1000 × 1.085 we need to use mode 1 of timer 1.

```
;--upon wake-up go to main, avoid using
;memory allocated to Interrupt Vector Table
ORG 0000H
LJMP MAIN ;by-pass int. vector table
;--ISR for timer 1 to generate square wave
ORG 001BH ;Timer 1 int. vector table
LJMP ISR_T1 ;jump to ISR
The main program for initialization
ORG 0030H ;after vector table space
MAIN: MOV TMOD,#10H ;Timer 1, mode 1
MOV P0,#0FFH ;make P0 an input port
MOV TL1,#018H ; TL1=18 low byte of -1000
```

```
MOV TH1,#0FCH          ;TH1=FC high byte of -1000
MOV IE,#88H                              ;10001000 enable Timer 1 int
SETB TR1               ;Start Timer 1
BACK: MOV A,P0         ;get data from P0
MOV P1,A               ;issue it to P1
SJMP BACK              ;keep doing it
;Timer 1 ISR. Must be reloaded, not auto-reload
ISR_T1: CLR TR1            ;stop Timer 1
MOV R2,#4                  ; 2MC
CLR P2.1               ;P2.1=0, start of low portion
HERE: DJNZ R2,HERE         ;4x2 machine cycle 8MC
MOV TL1,#18H           ;load T1 low byte value 2MC
MOV TH1,#0FCH              ;load T1 high byte value 2MC
SETB TR1                   ;starts timer1 1MC
SETB P2.1                  ;P2.1=1,back to high 1MC
RETI                   ;return to main
END
```


Example

Write a program to generate a square wave if 50Hz frequency on pin P1.2. This is similar to Example 9-12 except that it uses an interrupt for timer 0. Assume that XTAL=11.0592 MHz

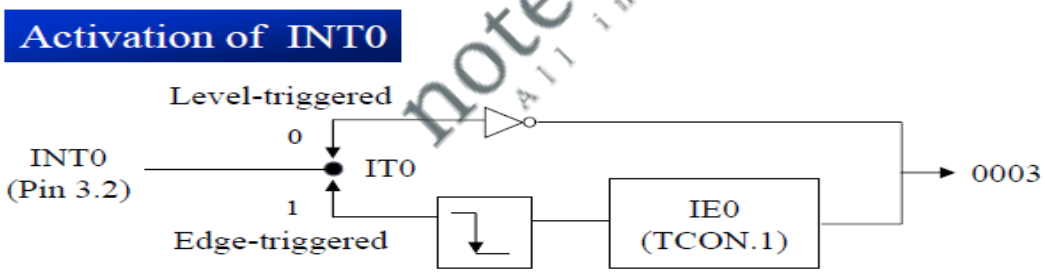
Solution:

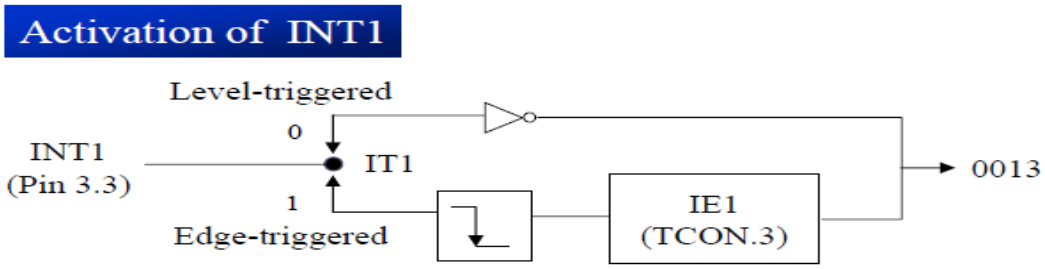
```
ORG 0
LJMP MAIN
ORG 000BH          ;ISR for Timer 0
CPL P1.2
MOV TL0,#00
MOV TH0,#0DCH
RETI
```

```
ORG 30H
;--------main program for initialization
MAIN:MOV TM0D,#00000001B ;Timer 0, Mode 1
MOV TL0,#00
MOV TH0,#0DCH
MOV IE,#82H ;enable Timer 0 interrupt
SETB TR0
HERE:SJMP HERE
END
```

The 8051 has two external hardware interrupts. Pin 12 (P3.2) and pin 13 (P3.3) of the 8051, designated as INT0 and INT1, are used as external hardware interrupts. The interrupt vector table locations 0003H and 0013H are set aside for INT0 and INT1. There are two activation levels for the external hardware interrupts.

1) Level trigged
2) Edge trigged





In the level-triggered mode, INT0 and INT1 pins are normally high.If a low-level signal is

applied to them, it triggers the interrupt. Then the microcontroller stops whatever it is doing and jumps to the interrupt vector table to service that interrupt. The low-level signal at the INT pin must be removed before the execution of the last instruction of the ISR, RETI; otherwise, another interrupt will be generated. This is called a level-triggered or level activated interrupt and is the default mode upon reset of the 8051.

TCON (Timer/Counter) Register (Bit-addressable)

D7							D0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON.7	Timer 1 overflow flag. Set by hardware when timer/counter 1 overflows. Cleared by hardware as the processor vectors to the interrupt service routine
TR1	TCON.6	Timer 1 run control bit. Set/cleared by software to turn timer/counter 1 on/off
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when timer/counter 0 overflows. Cleared by hardware as the processor vectors to the interrupt service routine
TR0	TCON.4	Timer 0 run control bit. Set/cleared by software to turn timer/counter 0 on/off

IE1	TCON.3	External interrupt 1 edge flag. Set by CPU when the external interrupt edge (H-to-L transition) is detected. Cleared by CPU when the interrupt is processed
IT1	TCON.2	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt
IE0	TCON.1	External interrupt 0 edge flag. Set by CPU when the external interrupt edge (H-to-L transition) is detected. Cleared by CPU when the interrupt is processed
IT0	TCON.0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt

Module No: 05

Interfacing 8051 to LCD

LCD is finding widespread use replacing LEDs for the following reasons:

- The declining prices of LCD
- The ability to display numbers, characters, and graphics
- Incorporation of a refreshing controller into the LCD, thereby relieving the CPU of the task of refreshing the LCD
- Ease of programming for characters and graphics

Interfacing LCD with 8051

The LCD requires 3 control lines (RS, R/W & EN) & 8 (or 4) data lines. The number on data lines depends on the mode of operation. If operated in 8-bit mode then 8 data lines + 3 control lines i.e. total 11 lines are required. And if operated in 4-bit mode then 4 data lines + 3 control lines i.e. 7 lines are required.

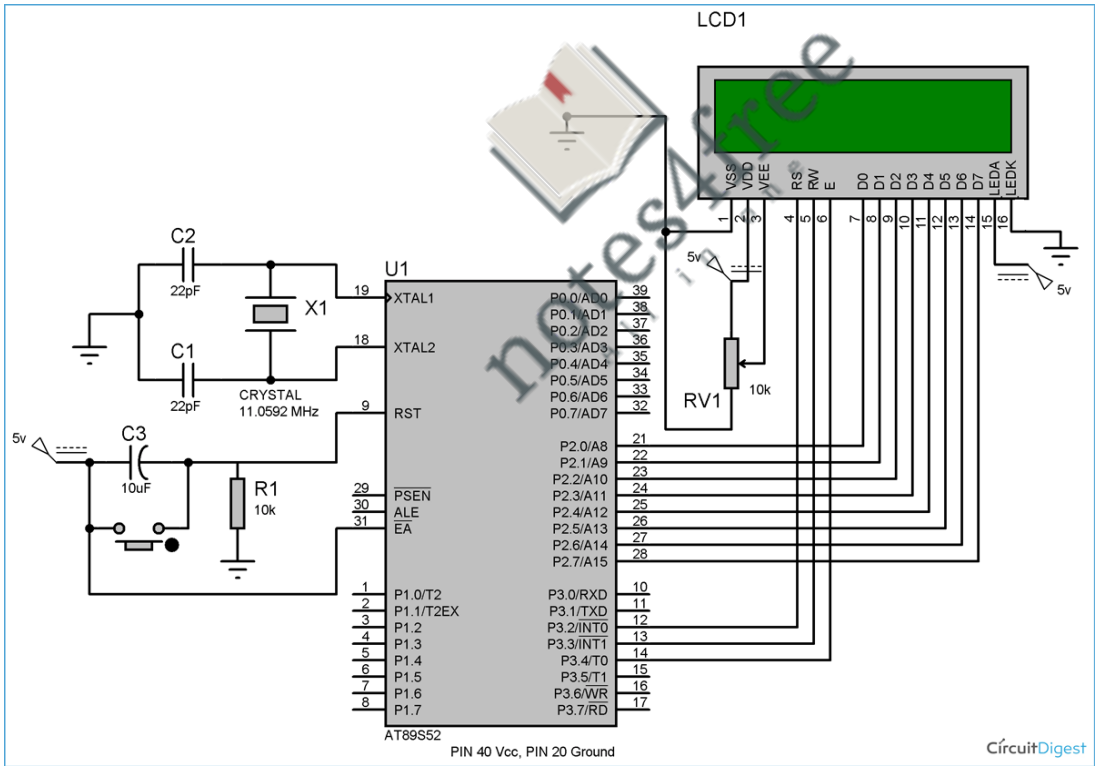


Figure: Interfacing 8051 with LCD

Pin Description:

Pin	Symbol	I/O	Description
1	V_{SS}	--	Ground
2	V_{CC}	--	+5V power supply
3	V_{EE}	--	Power supply to control contrast
4	RS	I	RS=0 to select command register, RS=1 to select data register
5	R/W	I	R/W=0 for write, R/W=1 for read
6	E	I/O	Enable
7	DB0	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 8-bit data bus
12	DB5	I/O	The 8-bit data bus
13	DB6	I/O	The 8-bit data bus
14	DB7	I/O	The 8-bit data bus

- Send displayed information or instruction command codes to the LCD
- Read the contents of the LCD's internal registers

LCD timing diagram for reading and writing is as shown in figure

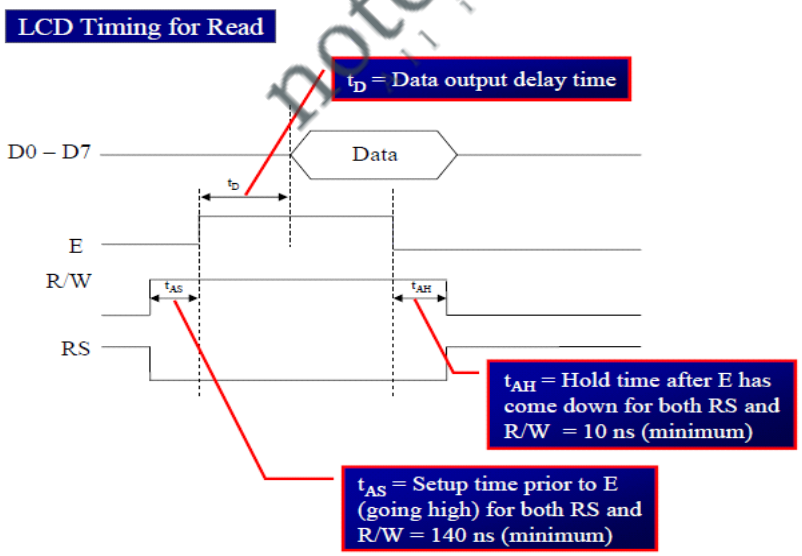


Figure : LCD timing for read

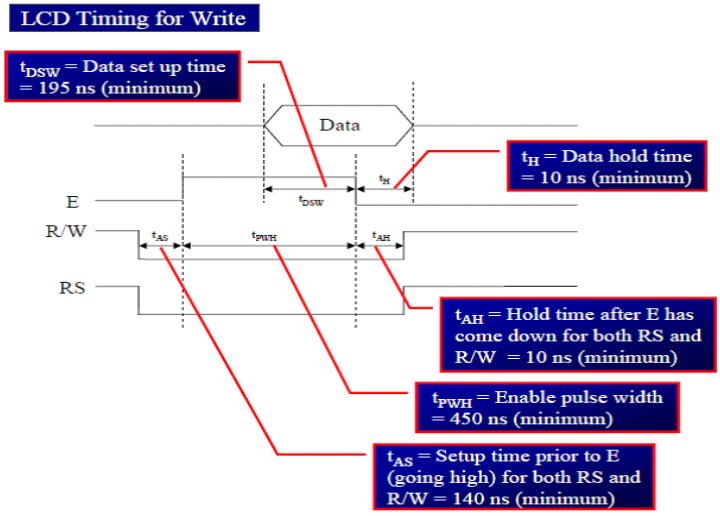


Figure : LCD timing for write

Sending Data/ Commands to LCDs with Time Delay:

To send any of the commands to the LCD, make pin RS=0. For data, make RS=1. Then send a high-to-low pulse to the E pin to enable the internal latch of the LCD. This is shown in the code below. The interfacing diagram of LCD to 8051 is as shown in the figure .

Example 11: Write an ALP to initialize the LCD and display message "YES". Say the command to be given is :38H (2 lines ,5x7 matrix), 0EH (LCD on, cursor on), 01H (clear LCD), 06H (shift cursor right), 86H (cursor: line 1, pos. 6)
Program: Calls a time delay before sending next data/command ;P1.0-P1.7 are connected to LCD data pins D0-D7 ;P2.0 is connected to RS pin of LCD ;P2.1 is connected to R/W pin of LCD ;P2.2 is connected to E pin of LCD

```
        ORG 0H
        MOV A,#38H          ;INIT. LCD 2 LINES, 5X7 MATRIX
        ACALL COMNWRT       ;call command subroutine
        ACALL DELAY         ;give LCD some time
        MOV A,#0EH          ;display on, cursor on
        ACALL COMNWRT       ;call command subroutine
        ACALL DELAY          ;give LCD some time
        MOV A,#01           ;clear LCD
        ACALL COMNWRT       ;call command subroutine
        ACALL DELAY         ;give LCD some time
        MOV A,#06H          ;shift cursor right
        ACALL COMNWRT        ;call command subroutine
        ACALL DELAY         ;give LCD some time
        MOV A,#86H          ;cursor at line 1, pos. 6
        ACALL COMNWRT       ;call command subroutine
        ACALL DELAY         ;give LCD some time
        MOV A,#'Y'          ;display letter Y
```

```
        ACALL DATAWRT       ;call display subroutine
        ACALL DELAY         ;give LCD some time
        MOV A,#'E'          ;display letter E
        ACALL DATAWRT       ;call display subroutine
        ACALL DELAY         ;give LCD some time
        MOV A,#'S'          ;display letter S
        ACALL DATAWRT       ;call display subroutine
AGAIN: SJMP AGAIN           ;stay here
COMNWRT:                    ;send command to LCD
        MOV P1,A            ;copy reg A to port 1
        CLR P2.0            ;RS=0 for command
        CLR P2.1            ;R/W=0 for write
        SETB P2.2           ;E=1 for high pulse
        ACALL DELAY         ;give LCD some time
        CLR P2.2            ;E=0 for H-to-L pulse
RET
DATAWRT:                    ;write data to LCD
        MOV P1,A            ;copy reg A to port 1
        SETB P2.0           ;RS=1 for data
        CLR P2.1            ;R/W=0 for write
        SETB P2.2           ;E=1 for high pulse
        ACALL DELAY         ;give LCD some time
        CLR P2.2            ;E=0 for H-to-L pulse
RET

DELAY:
        MOV R3,#50          ;50 or higher for fast CPUs
HERE2: MOV R4,#255          ;R4 = 255
HERE: DJNZ R4,HERE          ;stay until R4 becomes 0
        DJNZ R3,HERE2
        RET

        END
```

Example 12: Modify example 11, to check for the busy flag (D7=>P1.7), then send the command and hence display message "NO".

```
;Check busy flag before sending data, command to LCD;p1=data pin ;P2.0 connected to RS pin
;P2.1 connected to R/W pin ;P2.2 connected to E pin
ORG 0H
        MOV A,#38H          ;init. LCD 2 lines ,5x7 matrix
        ACALL COMMAND       ;issue command
        MOV A,#0EH          ;LCD on, cursor on
        ACALL COMMAND       ;issue command
        MOV A,#01H          ;clear LCD command
        ACALL COMMAND       ;issue command
        MOV A,#06H          ;shift cursor right
```

```
        ACALL COMMAND          issue command
        MOV A,#86H             ;cursor: line 1, pos. 6
        ACALL COMMAND          ;command subroutine
        MOV A,#'N'             ;display letter N
        ACALL DATA_DISPLAY
        MOV A,#'O'             ;display letter O
        ACALL DATA_DISPLAY
HERE:SJMP HERE          ;STAY HERE
COMMAND:
        ACALL READY            ;is LCD ready?
        MOV P1,A               ;issue command code
        CLR P2.0               ;RS=0 for command
        CLR P2.1               ;R/W=0 to write to LCD
        SETB P2.2              ;E=1 for H-to-L pulse
        CLR P2.2               ;E=0,latch in
RET
DATA_DISPLAY:
        ACALL READY            ;is LCD ready?
        MOV P1,A               ;issue data
        SETB P2.0              ;RS=1 for data
        CLR P2.1               ;R/W =0 to write to LCD
        SETB P2.2              ;E=1 for H-to-L pulse
        CLR P2.2               ;E=0,latch in
RET
READY:
        SETB P1.7              ;make P1.7 input port
        CLR P2.0               ;RS=0 access command reg
        SETB P2.1              ;R/W=1 read command reg ;
 BACK:SETB P2.2                ;E=1 for H-to-L pulse
        CLR P2.2               ;E=0 H-to-L pulse
        JB P1.7,BACK           ;stay until busy flag=0
        RET

        END
```

Sending Data/ Commands to LCDs checking the Busy Flag

Example 13: Write an 8051 C program to send letters 'P', 'I', and 'C' to the LCD using the busy flag method.

Solution:

```
#include <reg51.h>
sfr ldata = 0x90;                    //P1=LCD data pins
sbit rs = P2^0;
sbit rw = P2^1;
sbit en = P2^2;
sbit busy = P1^7;
```

```
void main()
{
        lcdcmd(0x38);
        lcdcmd(0x0E);
        lcdcmd(0x01);
        lcdcmd(0x06);
        lcdcmd(0x86);                   //line 1, position 6
        lcddata('P');
        lcddata('I');
        lcddata('C');
}
void lcdcmd(unsigned char value){
        lcdready();                     //check the LCD busy flag
        ldata = value;                  //put the value on the pins
        rs = 0;
        rw = 0;
        en = 1;                          //strobe the enable pin
        MSDelay(1);
        en = 0;
        return;
}
void lcddata(unsigned char value){
        lcdready();                     //check the LCD busy flag
        ldata = value;                  //put the value on the pins
        rs = 1;
        rw = 0;
        en = 1;                         //strobe the enable pin
        MSDelay(1);
        en = 0;
        return;

}
void lcdready(){
        busy = 1;                   //make the busy pin at input
        rs = 0;
        rw = 1;
        while(busy==1){             //wait here for busy flag
        en = 0;                     //strobe the enable pin
        MSDelay(1);
        en = 1;
}}
void Msdelay(unsigned int itime){
        unsigned int i, j;
        for(i=0;i<itime;i++)
        for(j=0;j<1275;j++);}
```

Keyboard Interfacing:

Keyboards are organized in a matrix of rows and columns. The CPU accesses both rows and columns through ports. Therefore, with two 8-bit ports, an 8 x 8 matrix of keys can be connected to a microprocessor. When a key is pressed, a row and a column make a contact. Otherwise, there is no connection between rows and columns. A 4x4 matrix connected to two ports. The rows are connected to an output port and the columns are connected to an input port.

Scanning and Identifying the Key:

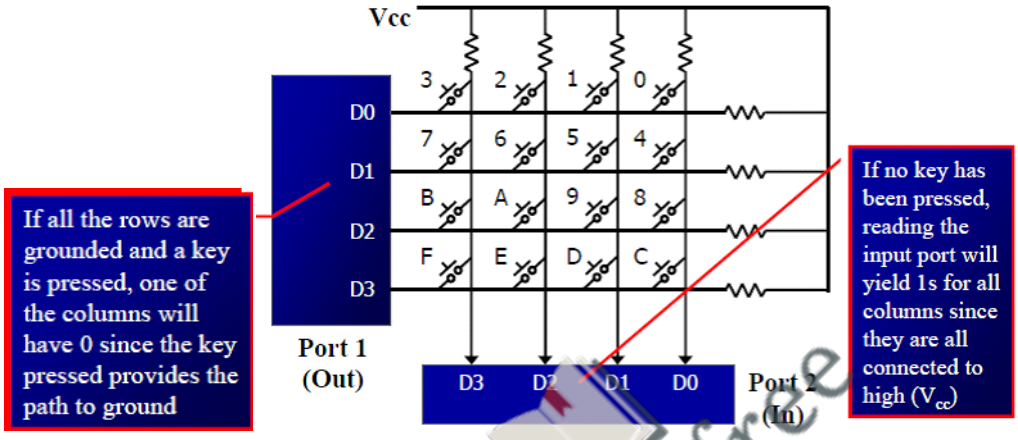


Figure : A 4X4 matrix keyboard

It is the function of the microcontroller to scan the keyboard continuously to detect and identify the key pressed

- To detect a pressed key, the microcontroller grounds all rows by providing 0 to the output latch, then it reads the columns
- If the data read from columns is D3 – D0 =1111, no key has been pressed and the process continues till key press is detected
- If one of the column bits has a zero, this means that a key press has occurred For example, if D3 – D0 = 1101, this means that a key in the D1 column has been pressed After detecting a key press, microcontroller will go through the process of identifying the key
- Starting with the top row, the microcontroller grounds it by providing a low to row D0 only. It reads the columns, if the data read is all 1s, no key in that row is activated and the process is moved to the next row
- It grounds the next row, reads the columns, and checks for any zero. This process continues until the row is identified.
- After identification of the row in which the key has been pressed. Find out which column the pressed key belongs to.

Algorithm for detection and identification of key activation goes through the following stages:

1. To make sure that the preceding key has been released, 0s are output to all rows at once, and the columns are read and checked repeatedly until all the columns are high

- When all columns are found to be high, the program waits for a short amount of time before it goes to the next stage of waiting for a key to be pressed

2. To see if any key is pressed, the columns are scanned over and over in an infinite loop until one of them has a 0 on it

- Remember that the output latches connected to rows still have their initial zeros (provided in stage 1), making them grounded
- After the key press detection, it waits 20 ms for the bounce and then scans the columns again
  (a) It ensures that the first key press detection was not an erroneous one due a spike noise
  (b) The key press. If after the 20-ms delay the key is still pressed, it goes back into the loop to detect a real key press

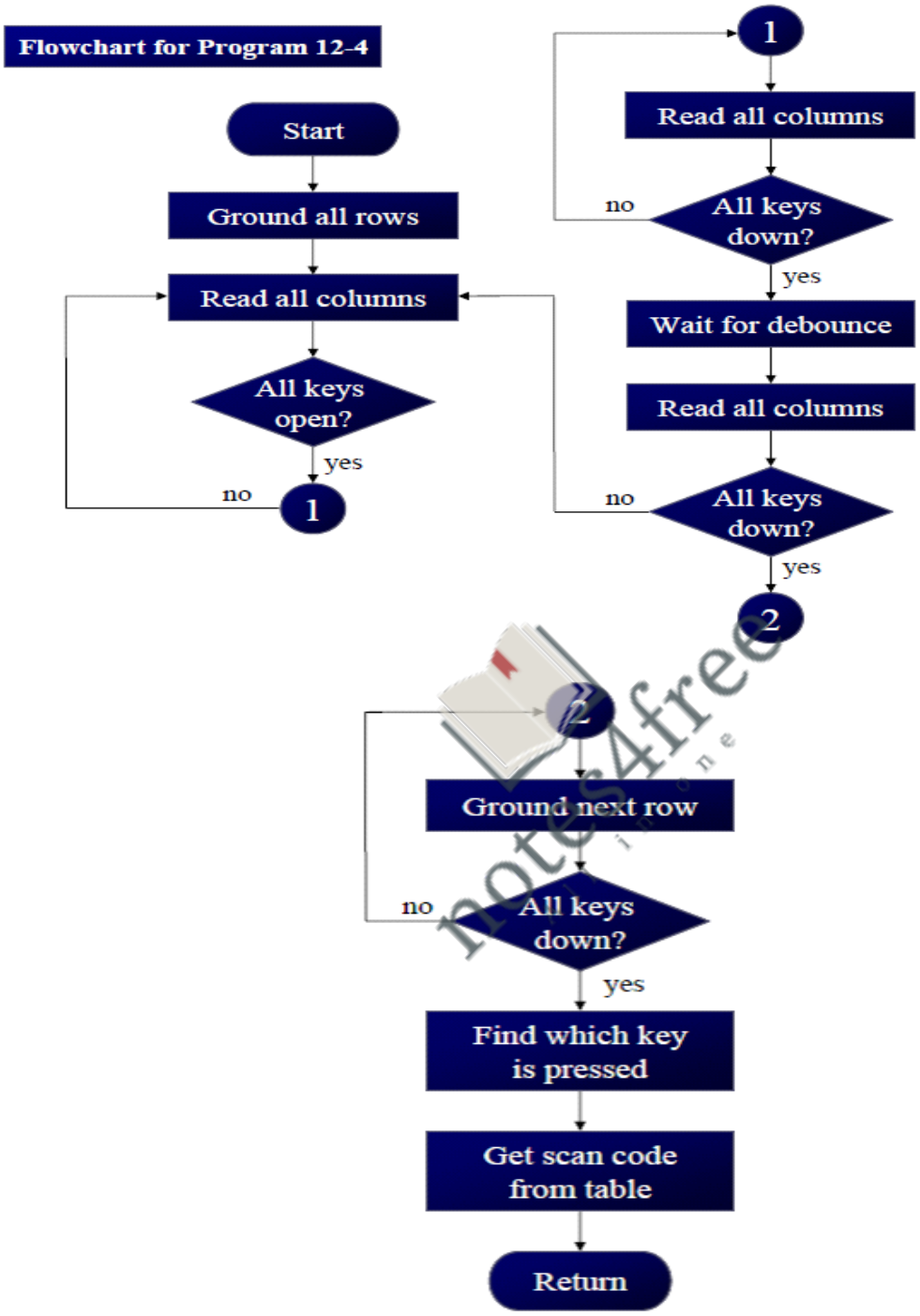
3. To detect which row key press belongs to, it grounds one row at a time, reading the columns each time

- If it finds that all columns are high, this means that the key press cannot belong to that row. Therefore, it grounds the next row and continues until it finds the row the key press belongs to
- Upon finding the row that the key press belongs to, it sets up the starting address for the look-up table holding the scan codes (or ASCII) for that row

4. To identify the key press, it rotates the column bits, one bit at a time, into the carry flag and checks to see if it is low

- Upon finding the zero, it pulls out the ASCII code for that key from the look-up table
- otherwise, it increments the pointer to point to the next element of the look-up table

The flowchart for the above algorithm is as shown below:



Note: The assembly as well as the C program can be written in accordance to the algorithm of the flowchart shown.

Interfacing 8051 to parallel and Serial ADC

Analog-to-digital converter (ADC) interfacing

ADCs (analog-to-digital converters) are among the most widely used devices for data acquisition. A physical quantity, like temperature, pressure, humidity, and velocity, etc., is converted to electrical (voltage, current) signals using a device called a transducer, or sensor
We need an analog-to-digital converter to translate the analog signals to digital numbers, so microcontroller can read them.

ADC804 chip:

ADC804 IC is an analog-to-digital converter. It works with +5 volts and has a resolution of 8 bits. Conversion time is another major factor in judging an ADC. Conversion time is defined as the time it takes the ADC to convert the analog input to a digital (binary) number. In ADC804 conversion time varies depending on the clocking signals applied to CLK R and CLK IN pins, but it cannot be faster than 110μs.

Pin Description of ADC804:

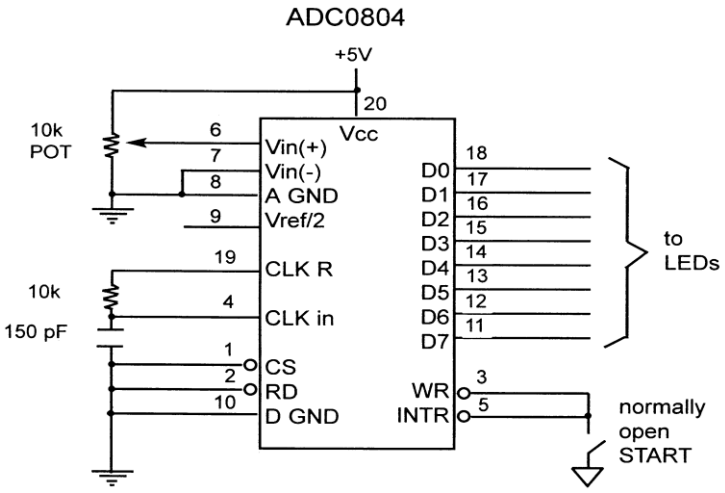


Figure: Pin out of ADC0804

- **CLK IN and CLK R:** CLK IN is an input pin connected to an external clock source. To use the internal clock generator (also called self-clocking), CLK IN and CLK R pins are connected to a capacitor and a resistor and the clock frequency is determined by:

$$f = \frac{1}{1.1RC}$$

Typical values are R = 10K ohms and C =150pF. We get f = 606 kHz and the conversion time is 110μs.

- **Vref/2 :** It is used for the reference voltage. If this pin is open (not connected), the analog input voltage is in the range of 0 to 5 volts (the same as the Vcc pin). If the analog input range needs to be 0 to 4 volts, Vref/2 is connected to 2 volts. Step size is the smallest change can be discerned by an ADC Vref/2 Relation to Vin Range

$V_{ref}/2$(v)	V_{in}(V)	Step Size (mV)
Not connected*	0 to 5	5/256=19.53
2.0	0 to 4	4/255=15.62
1.5	0 to 3	3/256=11.71
1.28	0 to 2.56	2.56/256=10
1.0	0 to 2	2/256=7.81
0.5	0 to 1	1/256=3.90

D0-D7: The digital data output pins. These are tri-state buffered. The converted data is accessed only when CS =0 and RD is forced low. To calculate the output voltage, use the following formula

$$D_{out} = \frac{V_{in}}{step\ size}$$

Dout = digital data output (in decimal),
Vin = analog voltage, and
step size (resolution) is the smallest change

Analog ground and digital ground: Analog ground is connected to the ground of the analog Vin and digital ground is connected to the ground of the Vcc pin. To isolate the analog Vin signal from transient voltages caused by digital switching of the output D0 – D7. This contributes to the accuracy of the digital data output.

Vin(+) & Vin(-): Differential analog inputs where Vin = Vin (+) – Vin (-). Vin (-) is connected to ground and Vin (+) is used as the analog input to be converted.

RD: Is "output enable" a high-to-low RD pulse is used to get the 8-bit converted data out of ADC804.

INTR: It is "end of conversion" When the conversion is finished, it goes low to signal the CPU that the converted data is ready to be picked up.

WR: It is "start conversion" When WR makes a low-to-high transition, ADC804 starts converting the analog input value of Vin to an 8- bit digital number.

CS: It is an active low input used to activate ADC804.

The following steps must be followed for data conversion by the ADC804 chip:

- Make CS= 0 and send a L-to-H pulse to pin WR to start conversion.
- Monitor the INTR pin, if high keep polling but if low, conversion is complete, go to next step.
- Make CS= 0 and send a H-to-L pulse to pin RD to get the data out

Figure shows the read and write timing for ADC804. Figure 9 and 10 shows the self clocking with the RC component for frequency and the external frequency connected to XTAL2 of 8051.

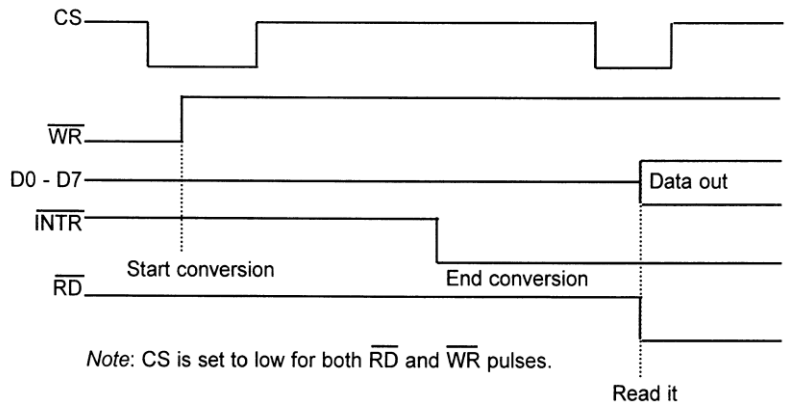


Figure : Read and Write timing for ADC0804

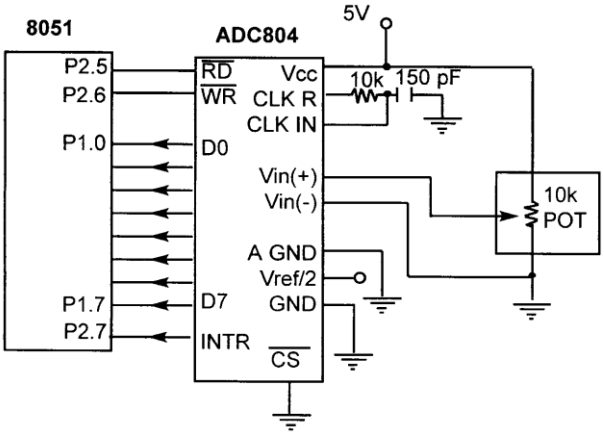


Figure : 8051 Connection to ADC0804 with Self-clocking

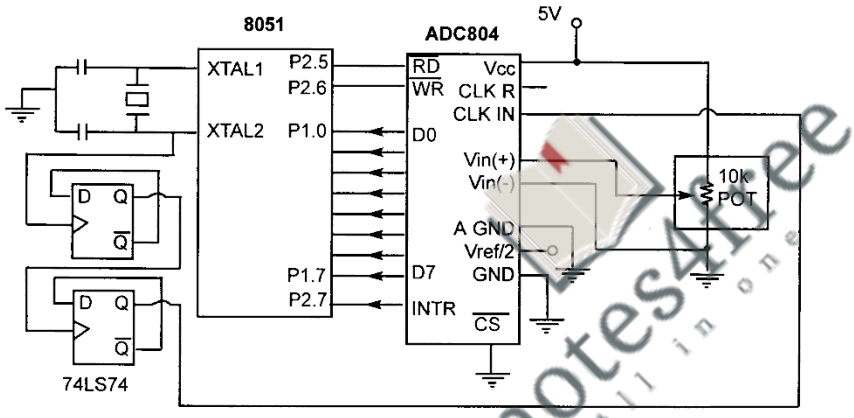


Figure : 8051 Connection to ADC0804 with Clock from XTAL2 of 8051

Now let us see how we write assembly as well as C program for the interfacing diagram shown in figure .

Programming ADC0804 in assembly

```
        MYDATA EQU P1
        MOV P1, #0FFH
        SETB P2.7
BACK:   CLR P2.6
        SETB P2.6
HERE:   JB P2.7, HERE
        CLR P2.5
        MOV A, MYDATA
        SETB P2.5
        SJMP BACK
```

Programming ADC0804 in C

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

```
Sbit RD=P2^5;
Sbit WR=P2^6;
Sbit INTR=P2^7;
Sfr Mydata=P1;
Void main ( )
{
        Unsigned char value;
        Mydata =0xFF;
        INTR=1;
        RD=1;
        WR=1;
        While (1)
        {
          WR=0;
           WR=1;
           While (INTR == 1);
          RD=0;
          Value =Mydata;
          RD=1;
        }
}
```


ADC0808/0809 chip:

ADC808 has 8 analog inputs. It allows us to monitor up to 8 different transducers using only single chip. The chip has 8-bit data output just like the ADC804. The 8 analog input channels are multiplexed and selected according to the values given to the three address pins, A, B, and C. that is; if CBA=000, CH0 is selected; CBA=011, CH3 is selected and so on. The pin details of ADC0808 are as shown in the figure below. (Explanation can be done as is with ADC0804).

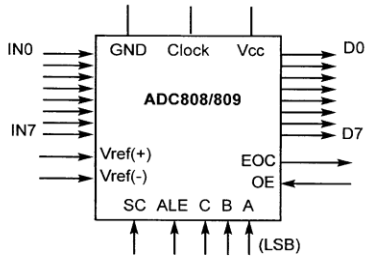


Figure : Pin out of ADC0808

Steps to Program ADC0808/0809

1. Select an analog channel by providing bits to A, B, and C addresses.
2. Activate the ALE pin. It needs an L-to-H pulse to latch in the address.
3. Activate SC (start conversion) by an H-to-L pulse to initiate conversion.
4. Monitor EOC (end of conversion) to see whether conversion is finished.

5. Activate OE (output enable) to read data out of the ADC chip. An H-to-L pulse to the OE pin will bring digital data out of the chip.

Let us write an assembly and C program for the interfacing of 8051 to ADC0808

Programming ADC0808/0809 in assembly

```
        MYDATA EQU P1
        ORG 0000H
        MOV MYDATA, #0FFH
        SETB P2.7
        CLR P2.4
        CLR P2.6
        CLR P2.5
BACK:   CLR P2.0
        CLR P2.1
        SETB P2.2
        ACALL DELAY
        SETB P2.4
        ACALL DELAY
        SETB P2.6
        ACALL DELAY
        CLR P2.4
        CLR P2.6
HERE:   JB P2.7, HERE
HERE1:  JNB P2.7, HERE1
        SETB P2.5
        ACALL DELAY
        MOV A, MYDATA
        CLR P2.5
        SJMP BACK
```


Interfacing 8051 to DAC

Digital-to-Analog (DAC) converter:

The DAC is a device widely used to convert digital pulses to analog signals. In this section we will discuss the basics of interfacing a DAC to 8051.

The two method of creating a DAC is binary weighted and R/2R ladder.
The Binary Weighted DAC, which contains one resistor or current source for each bit of the DAC connected to a summing point. These precise voltages or currents sum to the correct output value. This is one of the fastest conversion methods but suffers from poor accuracy because of the high precision required for each individual voltage or current. Such high-precision resistors and current-sources are expensive, so this type of converter is usually limited to 8-bit resolution or less.

The R-2R ladder DAC, which is a binary weighted DAC that uses a repeating cascaded structure of resistor values R and 2R. This improves the precision due to the relative ease of producing

equal valued matched resistors (or current sources). However, wide converters perform slowly due to increasingly large RC-constants for each added R-2R link.

The first criterion for judging a DAC is its resolution, which is a function of the number of binary inputs. The common ones are 8, 10, and 12 bits. The number of data bit inputs decides the resolution of the DAC since the number of analog output levels is equal to 2^n, where n is the number of data bit inputs.

DAC0808:

The digital inputs are converter to current $I_{out,}$ and by connecting a resistor to the I_{out} pin, we can convert the result to voltage. The total current I_{out} is a function of the binary numbers at the D0-D7 inputs of the DAC0808 and the reference current I_{ref}, and is as follows:

$$I_{out} = I_{ref}\left(\frac{D7}{2}+\frac{D6}{4}+\frac{D5}{8}+\frac{D4}{16}+\frac{D3}{32}+\frac{D2}{64}+\frac{D1}{128}+\frac{D0}{256}\right)$$

Usually reference current is 2mA. Ideally we connect the output pin to a resistor, convert this current to voltage, and monitor the output on the scope. But this can cause inaccuracy; hence an opamp is used to convert the output current to voltage. The 8051 connection to DAC0808 is as shown in the figure below.

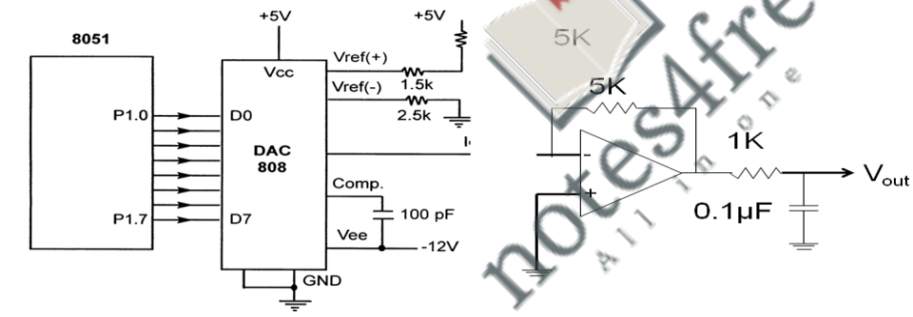


Figure : 8051 connection to DAC0808

Example 9: Write an ALP to generate a triangular waveform.
Program:

```
        MOV A, #00H
INCR:   MOV P1, A
        INC A
        CJNE A, #255, INCR
DECR:   MOV P1, A
        DEC A
        CJNE A, #00, DECR
        SJMP INCR
        END
```

Example 10: Write an ALP to generate a sine waveform.

$V_{out} = 5V(1+\sin\theta)$

Solution: Calculate the decimal values for every 10 degree of the sine wave. These values can be maintained in a table and simply the values can be sent to port P1. The sinewave can be observed on the CRO.

Program:

```
        ORG 0000H
AGAIN:MOV DPTR, #SINETABLE
        MOV R3, #COUNT
  UP:CLR A
        MOVC A, @A+DPTR
        MOV P1, A
        INC DPTR
        DJNZ R3, UP
        SJMP AGAIN
        ORG 0300H
SINETABLE DB 128, 192, 238, 255, 238, 192, 128, 64, 17, 0, 17, 64, 128
        END
```

Note: to get a better wave regenerate the values of the table per 2 degree.

Sensor interfacing and signal conditioning

The sensors of the LM34/LM35 series are precision integrated-circuit temperature sensors whose output voltage is linearly proportional to the Fahrenheit/Celsius temperature. The LM34/LM35 requires no external calibration since it is inherently calibrated. It outputs 10 mV for each degree of Fahrenheit/Celsius temperature.

Temperature sensors

Transducers convert physical data such as temperature, light intensity, flow, and speed to electrical signals. Depending on the transducer, the output produced is in the form of voltage, current, resistance, or capacitance. For example, temperature is converted to electrical signals using a transducer called a *thermistor*. A thermistor responds to temperature change by changing resistance, but its response is not linear.

LM34 and LM35 temperature sensors

The sensors of the LM34 series are precision integrated-circuit temperature sensors whose output voltage is linearly proportional to the Fahrenheit temperature. See Table 13-9. The LM34 requires no external calibration since it is internally calibrated. It outputs 10 mV for each degree of Fahrenheit temperature. Table 13-9 is a selection guide for the LM34.

Table 13-9: LM34 Temperature Sensor Series Selection Guide

Part	Temperature Range	Accuracy	Output Scale
LM34A	−50 F to +300 F	+2.0 F	10 mV/F
LM34	−50 F to +300 F	+3.0 F	10 mV/F
LM34CA	−40 F to +230 F	+2.0 F	10 mV/F
LM34C	−40 F to +230 F	+3.0 F	10 mV/F
LM34D	−32 F to +212 F	+4.0 F	10 mV/F

Note: Temperature range is in degrees Fahrenheit.

The LM35 series sensors are precision integrated-circuit temperature sensors whose output voltage is linearly proportional to the Celsius (centigrade) temperature. The LM35 requires no external calibration since it is internally calibrated. It outputs 10 mV for each degree of centigrade temperature.

Signal Conditioning

Signal conditioning is widely used in the world of data acquisition. The most common transducers produce an output in the form of voltage, current, charge, capacitance, and resistance. However, we need to convert these signals to voltage in order to send input to an A-to-D converter. This conversion (modification) is commonly called *signal conditioning.* Signal conditioning can be a current-to-voltage conversion or a signal amplification. For example, the thermistor changes resistance with temperature. The change of resistance must be translated into voltages in order to be of any use to an ADC. Look at the case of connecting an LM35 to an ADC0848. Since the ADC0848 has 8-bit resolution with a maximum of 256 (2^8) steps and the LM35 (or LM34) produces 10 mV for every degree of temperature change, we can condition V_{in} of the ADC0848 to produce a V_{out} of 2560 mV (2.56 V) for full-scale output. Therefore, in order to produce the full-scale V_{out} of 2.56 V for the ADC0848, we need to set V_{ref} = 2.56. This makes V_{out} of the ADC0848 correspond directly to the temperature as monitored by the LM35.

Figure 13-21 shows the connection of a temperature sensor to the ADC0848.

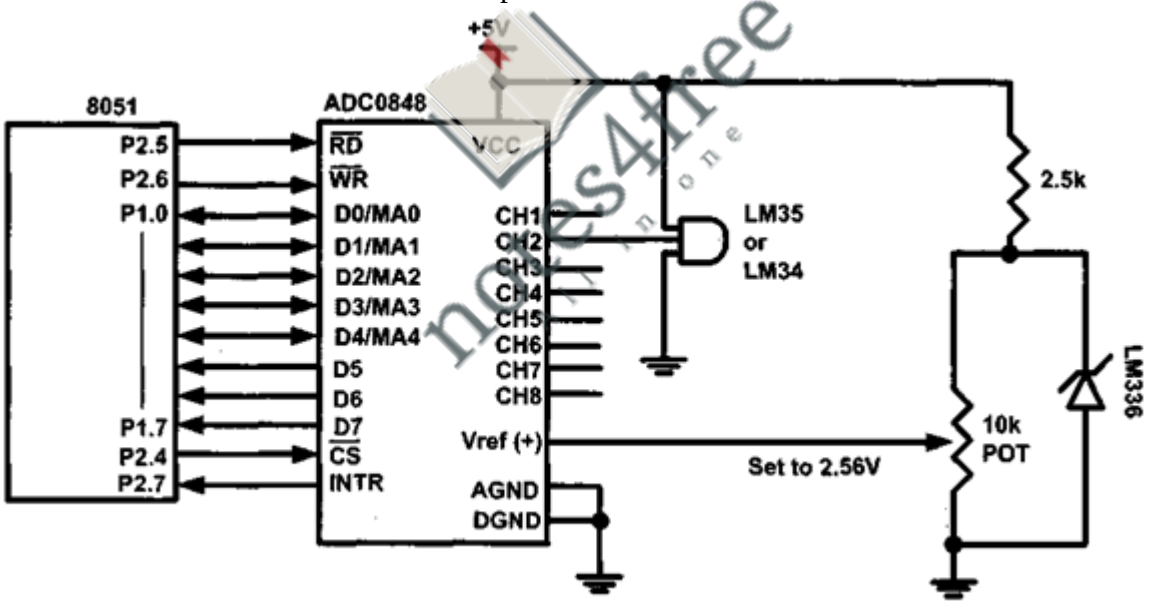


Figure. 8051 Connection to ADC0848 and Temperature Sensor

Stepper Motor Interfacing

Stepper motor is a widely used device that translates electrical pulses into mechanical movement. Stepper motor is used in applications such as; disk drives, dot matrix printer, robotics etc,. The construction of the motor is as shown in figure 1 below.

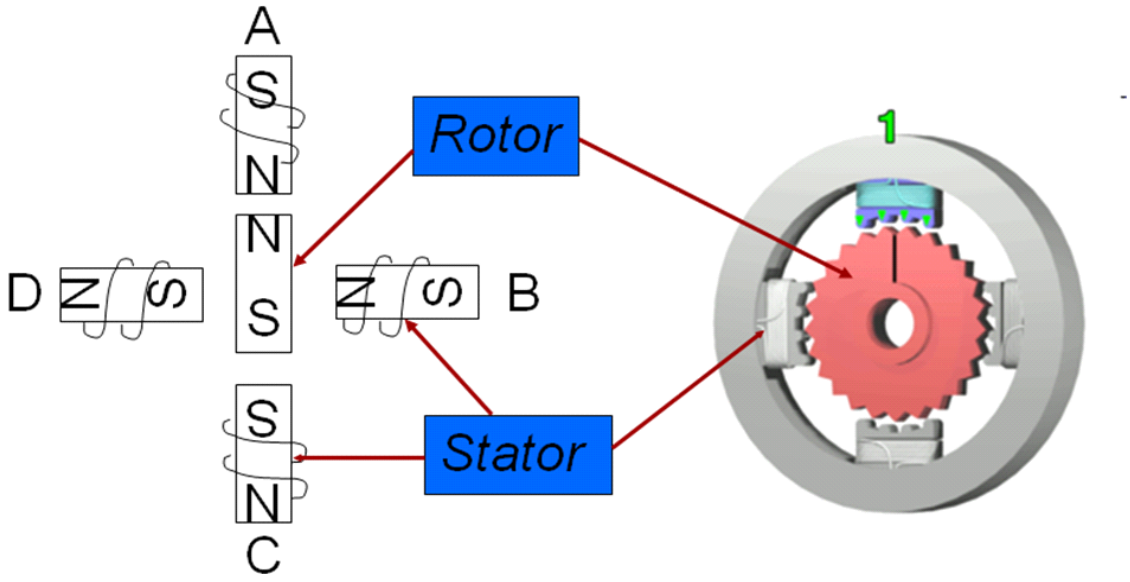


Figure: Structure of stepper motor

It has a permanent magnet rotor called the shaft which is surrounded by a stator. Commonly used stepper motors have four stator windings that are paired with a center – tapped common. Such motors are called as four-phase or unipolar stepper motor.

The stator is a magnet over which the electric coil is wound. One end of the coil are connected commonly either to ground or +5V. The other end is provided with a fixed sequence such that the motor rotates in a particular direction. Stepper motor shaft moves in a fixed repeatable increment, which allows one to move it to a precise position. Direction of the rotation is dictated by the stator poles. Stator poles are determined by the current sent through the wire coils.

Step angle:

Step angle is defined as the minimum degree of rotation with a single step.

No of steps per revolution = 360° / step angle

Steps per second = (rpm x steps per revolution) / 60

Example: step angle = 2°

No of steps per revolution = 180

Switching Sequence of Motor:

As discussed earlier the coils need to be energized for the rotation. This can be done by sending a bits sequence to one end of the coil while the other end is commonly connected. The bit sequence sent can make either one phase ON or two phase ON for a full step sequence or it can be a combination of one and two phase ON for half step sequence. Both are tabulated below.

Full Step:

Two Phase ON

Clockwise ↓

Step #	A	B	C	D
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1

↑ Counter-clockwise

One Phase ON

Clockwise ↓

Step #	A	B	C	D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

↑ Counter-clockwise

Half Step (8 – sequence):

The sequence is tabulated as below:

Clk ↓

Step #	A	B	C	D
1	1	0	0	1
2	1	0	0	0
3	1	1	0	0
4	0	1	0	0
5	0	1	1	0
6	0	0	1	0
7	0	0	1	1
8	0	0	0	1

↑ Anti-Clk

8051 Connection to Stepper Motor: (explanation of the diagram can be done)

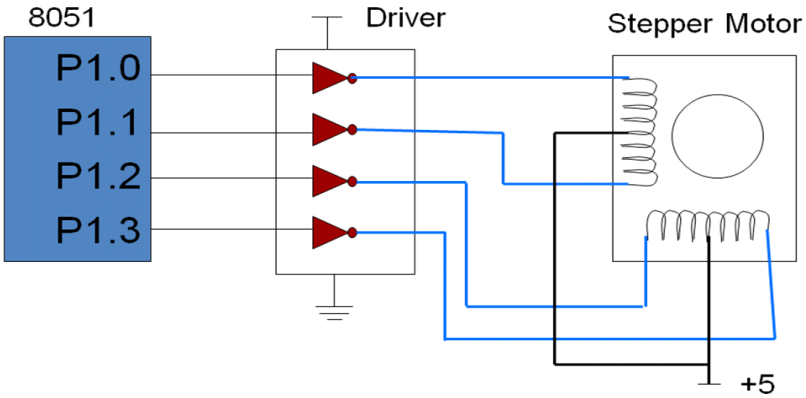


Figure : 8051 interface to stepper motor

Example 1: Write an ALP to rotate the stepper motor clockwise / anticlockwise continuously with full step sequence.

Program:

```
        MOV A,#66H
BACK: MOV P1,A
        RR A
        ACALL DELAY
        SJMP BACK
DELAY: MOV R1,#100
 UP1:    MOV R2,#50
 UP:     DJNZ R2,UP
         DJNZ R1,UP1
         RET
```

Note: motor to rotate in anticlockwise use instruction RL A instead of RR A

Example 2: A switch is connected to pin P2.7. Write an ALP to monitor the status of the SW. If SW = 0, motor moves clockwise and if SW = 1, motor moves anticlockwise.

Program:

```
        ORG 0000H
        SETB P2.7
        MOV A, #66H
        MOV P1,A
TURN: JNB P2.7, CW
        RL A
        ACALL DELAY
        MOV P1,A
        SJMP TURN
CW:    RR A
        ACALL DELAY
        MOV P1,A
        SJMP TURN
DELAY: as previous example
```


Example 4: Rotate the stepper motor continuously clockwise using half-step 8-step sequence. Say the sequence is in ROM locations.

Program:

```
        ORG 0000H
START: MOV R0, #08
         MOV DPTR, #HALFSTEP
RPT:    CLR A
         MOVC A, @A+DPTR
         MOV P1, A
         ACALL DELAY
         INC DPTR
         DJNZ R0, RPT
         SJMP START
```

```
        ORG 0200H
HALFSTEP DB 09, 08, 0CH, 04, 06, 02, 03, 01
        END
```

Programming Stepper Motor with 8051 C

The following examples 5 and 6 will show the programming of stepper motor using 8051 C.

Example 5: Problem definition is same as example 1.

Program:

```
#include <reg51.h>
void main ()
{
        while (1)
         {
            P1=0x66;
            MSDELAY (200);
            P1=0x33;
            MSDELAY (200);
                    P1=0x99;
            MSDELAY (200);
                 P1=0xCC;
            MSDELAY (200);
         }
}
void MSDELAY (unsigned char value)
{
         unsigned int x,y;
         for(x=0;x<1275;x++)
         for(y=0;y<value;y++);
}
```


Example 6: Problem definition is same as example 2.

Program:

```
#include <reg51.h>
sbit SW=P2^7;
void main ()
{
        SW=1;
         while (1)
         {
           if(SW==0){
            P1=0x66;
            MSDELAY (100);
            P1=0x33;
            MSDELAY (100);
```

```
            P1=0x99;
            MSDELAY (100);
            P1=0xCC;
            MSDELAY (100);
        }
        else {
            P1=0x66;
            MSDELAY (100);
            P1=0xCC;
            MSDELAY (100);
            P1=0x99;
            MSDELAY (100);
            P1=0x33;
            MSDELAY (100);
}
void MSDELAY (unsigned char value)
{
        unsigned int x,y;
        for(x=0;x<1275;x++)
        for(y=0;y<value;y++);
}
```

DC Motor Interfacing with 8051:

The DC motor is another widely used device that translates electrical pulses into mechanical movement. Motor has 2 leads +ve and – ve , connecting them to a DC voltage supply moves the motor in one direction. On reversing the polarity rotates the motor in the reverse direction. Basic difference between Stepper and DC motor is stepper motor moves in steps while DC motor moves continuously. Another difference is with stepper motor the number of steps can be counted while it is not possible in DC motor. Maximum speed of a DC motor is indicated in rpm. The rpm is either with no load it is few thousands to tens of thousands or with load rpm decreases with increase in load.

Voltage and current rating : Nominal voltage is the voltage for a motor under normal condition. It ranges from 1V to 150V. As voltage increases, rpm goes up. Current rating is the current consumption when the nominal voltage is applied with no load that is 25mA to a few amperes. As load increases, rpm increases, unless voltage or current increases implies torque increases. With fixed voltage, as load increases, power consumption of a DC motor is increased.

Unidirectional Control:

Figure 3 shows the rotation of the DC motor in clockwise and anticlockwise direction.

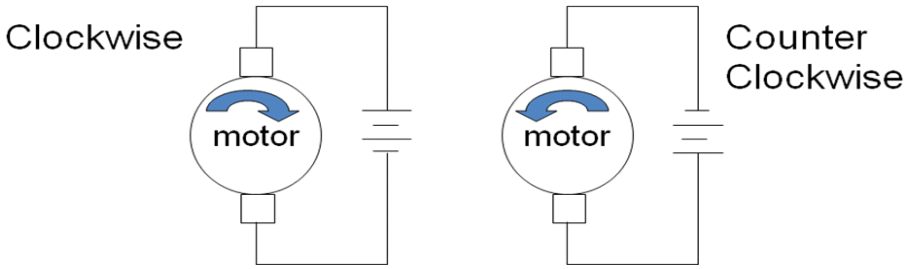
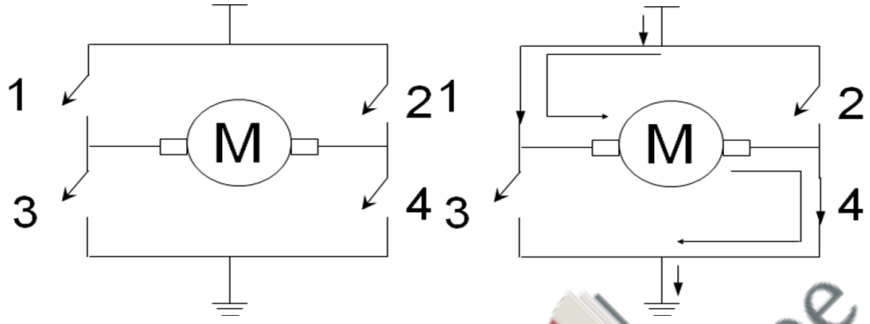
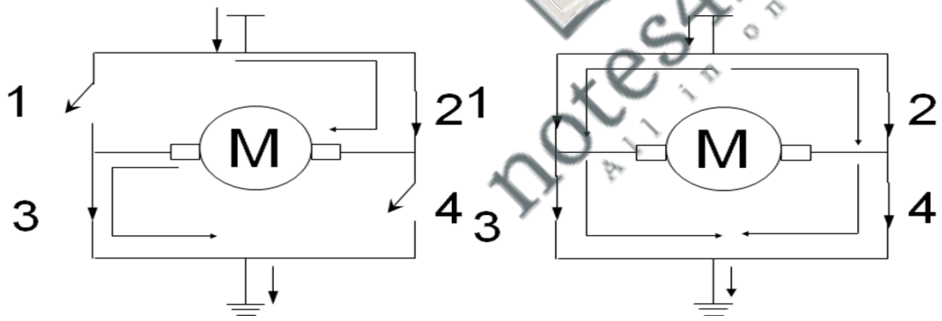


Figure 3: DC motor rotation

Bidirectional Control:



- Motor not running (b) Clockwise direction



(c) Counter clockwise direction (d) Invalid state (short circuit)

Figure : H-Bridge Motor Configuration

Figure shows the H-Bridge motor configuration. It consists of four switches and based on the closing and opening of these switches the motor either rotates in clockwise or anti-clockwise direction.

As seen in figure 4a, all the switches are open hence the motor is not running. In b, turning of the motor is in one direction when the switches 1 and 4 are closed that is clockwise direction.

Similarly, in c the switches 2 and 3 are closed so the motor rotates in anticlockwise direction, while in figure 4d all the switches are closed which indicates a invalid state or a short circuit.

The interfacing diagram of 8051 to bidirectional motor control can be referred to fig 17-18 from text prescribed.

Example 6: A switch is connected to pin P2.7. Write an ALP to monitor the status of the SW. If SW = 0, DC motor moves clockwise and if SW = 1, DC motor moves anticlockwise.

Program:

```
        ORG 0000H
        CLR P1.0
        CLR P1.1
        CLR P1.2
        CLR P1.3
        SETB P2.7
MONITOR: JNB P2.7, CLOCK
        SETB P1.0
        CLR P1.1
        CLR P1.2
        SETB P1.3
        SJMP MONITOR
CLOCK:  CLR P1.0
        SETB P1.1
        SETB P1.2
        CLR P1.3
        SJMP MONITOR
        END
```

Pulse Width Modulation (PWM)

The speed of the motor depends on 3 parameters: load, voltage and current. For a given load, we can maintain a steady speed by using PWM. By changing the width of the pulse applied to DC motor, power provided can either be increased or decreased. Though voltage has fixed amplitude, has a variable duty cycle. The wider the pulse, higher the speed obtained. One of the reasons as to why dc motor are referred over ac is, the ability to control the speed of the DC motor using PWM. The speed of the ac motor is dictated by the ac frequency of voltage applied to the motor and is generally fixed. Hence, speed of the AC motors cannot be controlled when load is increased.

Figure below shows the pulse width modulation comparison.

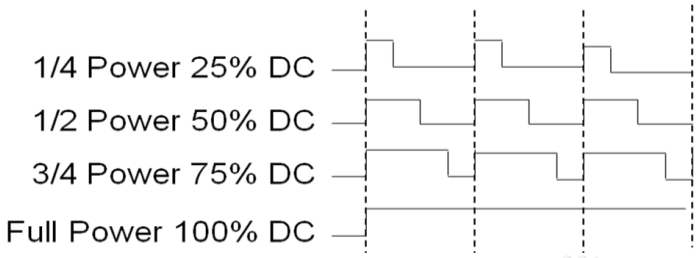


Figure : PWM comparison

Example 7 and 8 are the 8051 C version of the programs written earlier.

Example 7: A switch is connected to pin P2.7. Write a C to monitor the status of the SW. If SW = 0, DC motor moves clockwise and if SW = 1, DC motor moves anticlockwise.

Program:

```
# include <reg51.h>
sbit SW =P2^7;
sbit Enable = P1^0;
sbit MTR_1 = P1^1;
sbit MTR_2 = P1^2;
void main ( )
{
        SW=1;
        Enable = 0;
        MTR_1=0;
        MTR_2=0;
while( )
{
        Enable =1;
        if( SW==1)
        {       MTR_1=1;
                MTR_2=0;
        }
        else
        {       MTR_1=0;
```

```
            MTR_2=1;
        }
        }
        }
```

Example 8: A switch is connected to pin P2.7. Write an C to monitor the status of the SW. If SW = 0, DC motor moves 50% duty cycle pulse and if SW = 1, DC motor moves with 25% duty cycle pulse.

Program:

```
# include <reg51.h>
sbit SW =P2^7;
sbit MTR = P1^0;
void main ( )
{
        SW=1;
        MTR=0;
while( )
{       if( SW==1)
        {       MTR=1;
                Msdelay(25);
                MTR=0;
                Msdelay(75);
        }
        else
        {       MTR=1;
                Msdelay(50);
                MTR=0;
                Msdelay(50);
        }
        }
        }
```


8051 interfacing with the 8255

The Intel 8255 Programmable Peripheral Interface (PPI) chip, developed and manufactured by Intel in the first half of the 1970s for the Intel 8080 microprocessor. The 8255 provides 24 parallel input/output lines with a variety of programmable operating modes. The 8255 gives a CPU or digital system access to programmable parallel I/O. The 8255 has 24 input/output pins. These are divided into three 8-bit ports (A, B, C). Port A and port B can be used as 8-bit input/output ports. Port C can be used as an 8-bit input/output port or as two 4-bit input/output ports or to produce handshake signals for ports A and B.

The three ports are further grouped as follows:

1. Group A consisting of port A and upper part of port C.
2. Group B consisting of port B and lower part of port C.

Eight data lines (D0–D7) are available (with an 8-bit data buffer) to read/write data into the ports or control register under the status of the **RD** (pin 5) and **WR** (pin 36), which are active-low

signals for read and write operations respectively. Address lines A_1 and A_0 allow to access a data register for each port or a control register.

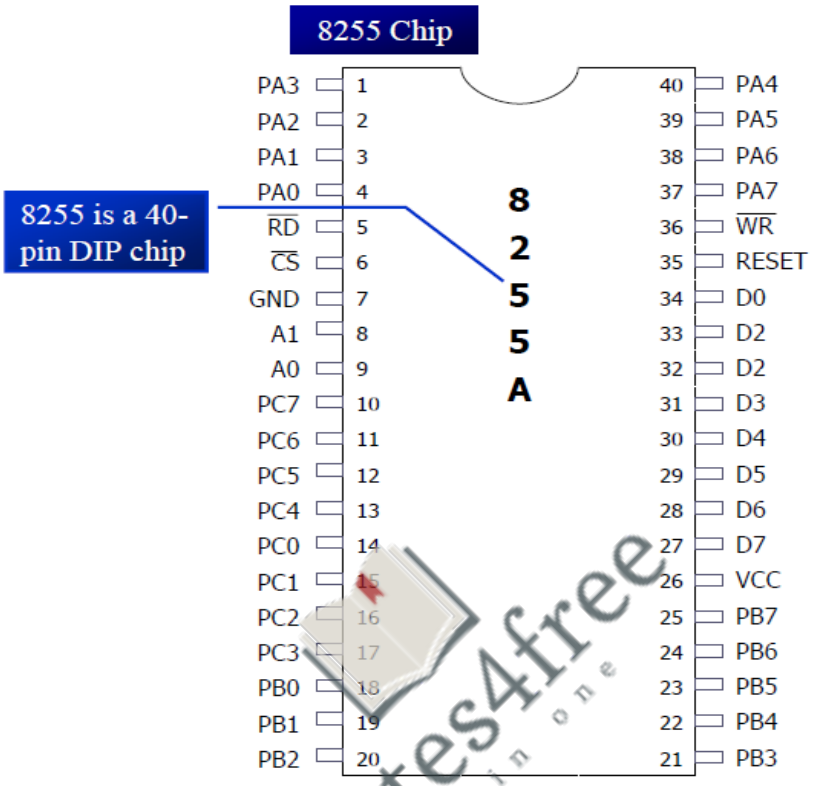


Figure: Pin diagram of 8255

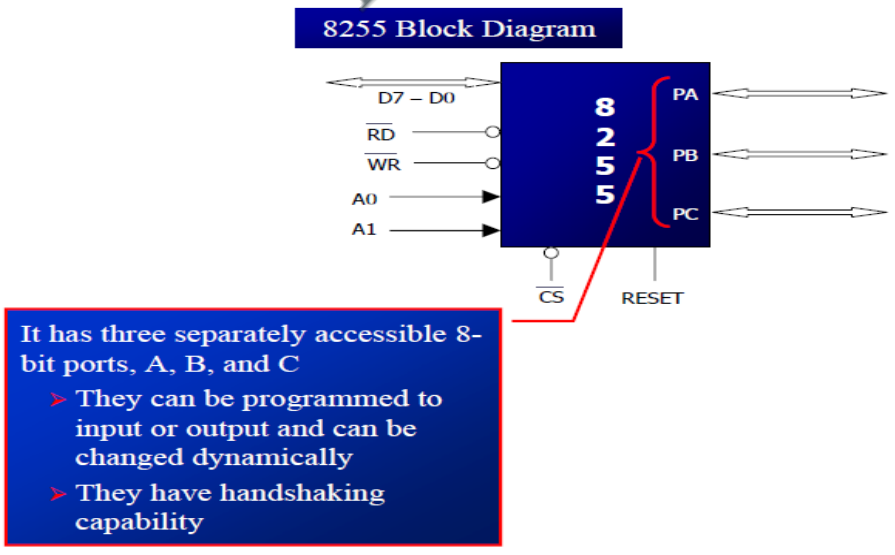


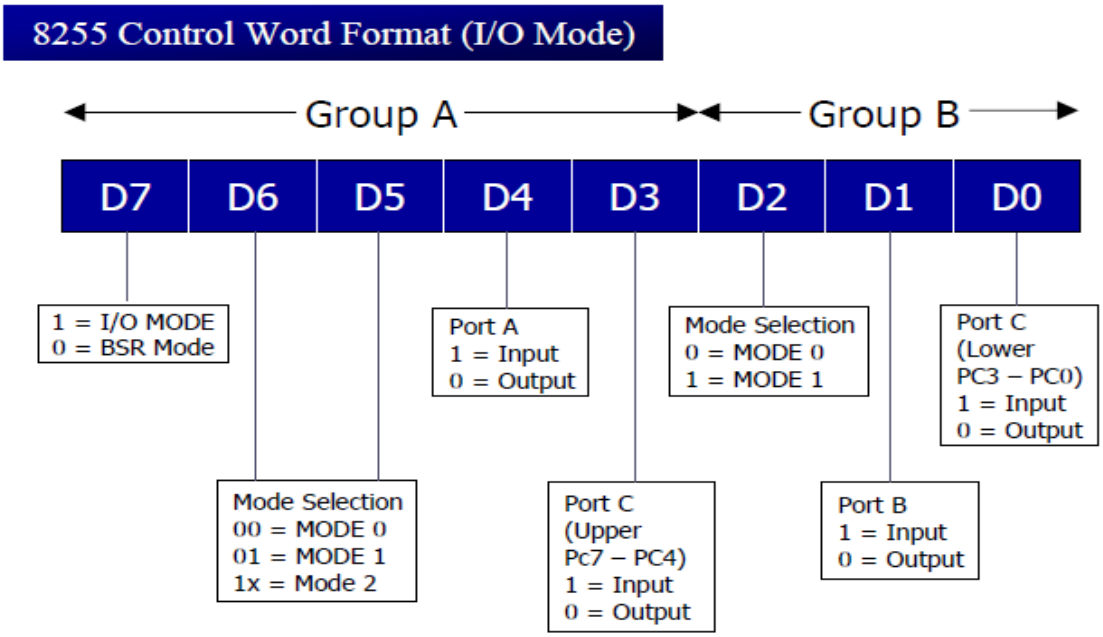
Figure: Block diagram of 8255

PA0 - PA7 (8-bit port A): Can be programmed as all input or output, or all bits as bidirectional input/output.
PB0 - PB7 (8-bit port B): Can be programmed as all input or output, but cannot be used as a bidirectional port.
PC0 – PC7 (8-bit port C): Can be all input or output, can also be split into two parts.
CU (upper bits PC4 - PC7), CL (lower bits PC0 – PC3) each can be used for input or output
Any of bits PC0 to PC7 can be programmed individually.
RD and WR: These two active-low control signals are inputs to the 8255. The RD and WR signals from the 8031/51 are connected to these inputs.
D0 – D7: are connected to the data pins of the microcontroller allowing it to send data back and forth between the controller and the 8255 chip.
RESET: An active-high signal input. Used to clear the control register. When RESET is activated, all ports are initialized as input ports.
A0, A1, and CS (chip select): CS is active-low. While CS selects the entire chip, it is A0 and A1 that select specific ports. These 3 pins are used to access port A, B, C or the control register.

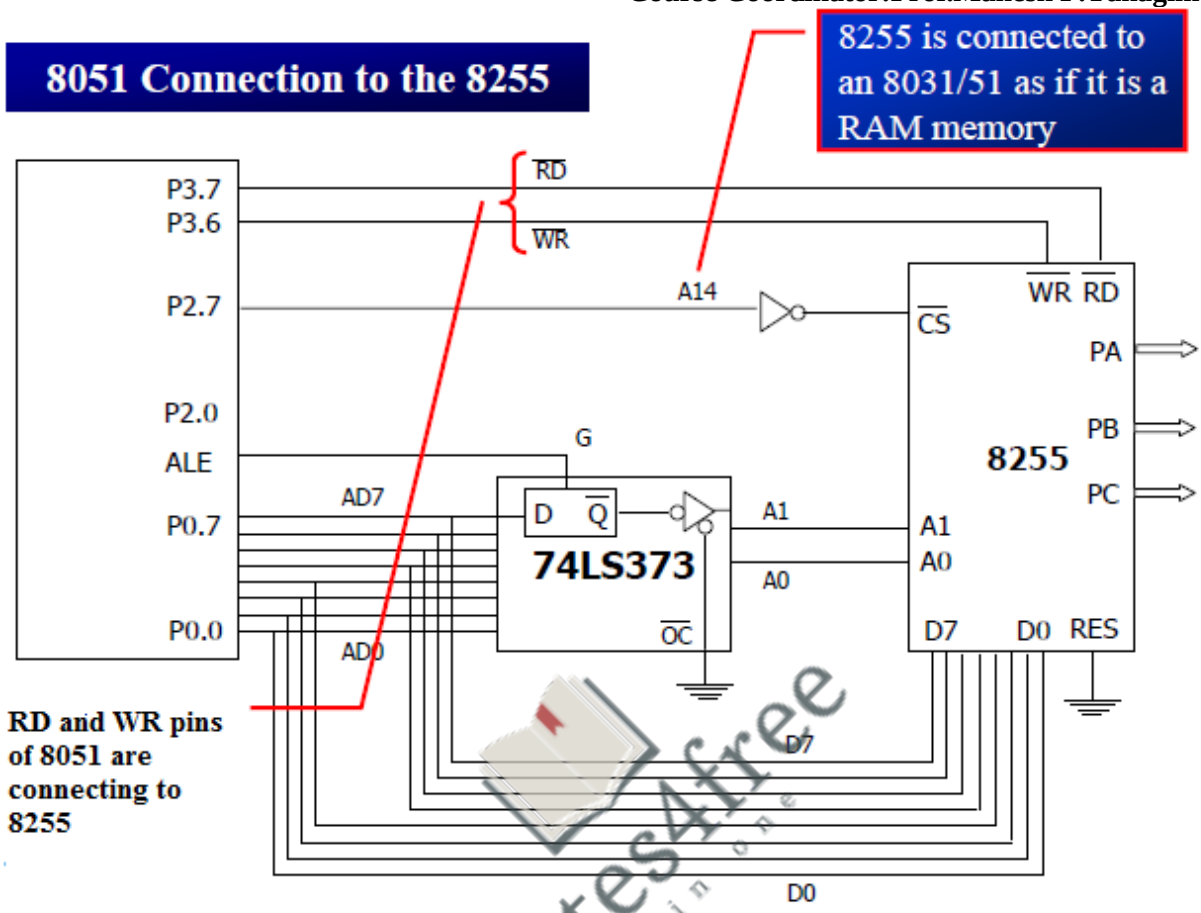
8255 Port Selection

CS	A1	A0	Selection
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control register
1	X	X	8255 is not selected

While ports A, B and C are used to input or output data, the control register must be programmed to select operation mode of three ports. The ports of the 8255 can be programmed in any of the following modes:
1. Mode 0, simple I/O: Any of the ports A, B, CL, and CU can be programmed as input out output. All bits are out or all are in. There is no signal-bit control as in P0-P3 of 8051
2. Mode 1: Port A and B can be used as input or output ports with handshaking capabilities. Handshaking signals are provided by the bits of port C.
3. Mode 2: Port A can be used as a bidirectional I/O port with handshaking capabilities provided by port C. Port B can be used either in mode 0 or mode 1.
4. BSR (bit set/reset) mode
Only the individual bits of port C can be Programmed.



The more commonly used term is I/O.
Mode 0: Intel calls it the basic input/output mode. In this mode, any ports of A, B, or C can be programmed as input or output . A given port cannot be both input and output at the same time
The 8255 chip is programmed in any of the 4 modes mentioned earlier by sending a byte (Intel calls it a control word) to the control register of 8255. We must first find the port address assigned to each of ports A, B ,C and the control register called mapping the I/O port.



Example

Find the control word of the 8255 for the following configurations:

(a) All the ports of A, B and C are output ports (mode 0)

(b) PA = in, PB = out, PCL = out, and PCH = out

Solution:

From Figure 15-3 we have:

(a) 1000 0000 = 80H (b)1001 0000 = 90H.

Example

For Figure shown below

(a) Find the I/O port addresses assigned to ports A, B, C, and the control register.

(b) Program the 8255 for ports A, B, and C to be output ports.

(c) Write a program to send 55H and AAH to all ports continuously.

Solution

(a) The base address for the 8255 is as follows:

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	
X	1	X	x	X	x	x	x	x	x	x	x	x	x	0	0	= 4000H PA
X	1	X	X	x	x	x	x	x	x	x	x	x	X	0	1	= 4001H PB
X	1	X	X	x	x	x	x	x	x	x	x	X	X	1	0	= 4002H PC
x	1	x	X	x	x	x	x	x	x	x	x	x	x	1	1	= 4003H CR

(b) The control byte (word) for all ports as output is 80H.

(c)

```
        MOV A,#80H ;control word;(ports output)
        MOV DPTR,#4003H ;load control reg; port address
        MOVX @DPTR, A ;issue control word
        MOV A,#55H        ;A = 55H
AGAIN: MOV DPTR,#4000H ;PA address
        MOVX @DPTR,A ;toggle PA bits
        INC DPTR ;PB address
        MOVX @DPTR,A ;toggle PB bits
        INC DPTR ;PC address
        MOVX @DPTR,A ;toggle PC bits
        CPL A ;toggle bit in reg A
ACALL DELAY ;wait
         SJMP AGAIN ;continue.
```