8051 MICROCONTROLLER

For V semester EC/TC of VTU, Belagavi

As Per CBCS Scheme

Dr. ARUN KUMAR G
Associate Professor & HoD,
Department of ECE, JSS Academy of Technical Education, Noida, Uttar Pradesh.

Dr. NAGARAJA B G
Professor & HoD,
Department of ECE, Jain Institute of Technology, Davangere, Karnataka.

Dr. SPOORTI J JAINAR
Assistant Professor,
Department of ECE, JSS Academy of Technical Education, Noida, Uttar Pradesh.

Dr. PUROHIT SHRINIVASACHARYA
Associate Professor,
Department of ISE, Siddaganga Institute of Technology, Tumakuru, Karnataka.

Gouthami Publications

No. 101(3), Yashashwini Nilaya
2nd Cross, Basaveshwara Nagara,
Shimoga, Karnataka

8051 Microcontroller for V Sem. BE Students of VTU

written by, Dr. Arun Kumar G, Dr. Nagaraja B G, Dr. Spoorti J Jainar and Dr. Purohit Shrinivasacharya

Published by

Gouthami Publications

No. 101(3), Yashashwini Nilaya

2nd Cross, Basaveshwara Nagara,

Shimoga, Karnataka

Mobile: 9686799389

E-mail: *ngajjanna@gmail.com*

First Edition - 2020

Price: ` 200.00

Dedicated

To Almighty,

All Our Family Members,

Teachers

&

Friends

Preface

The importance of microcontroller in this modern computing world is unbelievable as it plays a very important role in each & every day-to-day application as majority of the products in the markets have an inbuilt microcontroller for its operation. It is therefore necessary for any technical person of any branch to know the basics of microprocessor & microcontroller as this is the era of computing. Thus, it is a matter of great pleasure for us to write a text-book covering the entire syllabi for the V semester engineering students of microcontrollers at the fundamental level.

The authors are immensely pleased to release the first edition of their book, **'8051 Microcontroller'**. The fundamental concepts of this book are presented in an easily understandable, clear & systematic manner so that any student can understand the programming concepts and get through his/her fundas. The book prepares very carefully as several background topics with essential illustrations and practical examples and then further gives the complex programming concepts step by step and also their explanations with the list of instruction sets and gives totally an exam oriented approach, which will be very useful from the scoring point of view. This book covers the entire syllabus of **'8051 Microcontroller'** for the V semester CBCS Scheme (Common to EC/TC) of VTU, Belagavi, Karnataka. A large number of programming exercises along with their algorithms are also given, so that the students can develope their own programs for any type of applications.

Five modules are present in this text-book & the topics that are covered are

Module -1: 8051 Microcontroller

Module -2: 8051 Instruction Set

Module -3: 8051 Stack, I/O Port Interfacing and Programming

Module -4: 8051 Timers and Serial Port

Module -5: 8051 Interrupts and Interfacing Applications

All the five modules are well addressed theoretically as well as problematically. A large number of examination problems have been solved to substantiate the theoretical concepts. At the same time, solutions to the examination question papers are also has been put up & thus the book seems to be giving totally an examination oriented approach.

This book will be very much useful not only to the students of various engineering and polytechnic colleges, but also to the teachers. The book also serves as a ready reckoner for some of the competitive exams. Suggestions for the improvement of this book are highly appreciated in this regard & are welcomed.

Special Naman is due to His Holiness Sri Sri Shivarathrishwara Deshikendra Mahaswamiji, President JSS Mahavidyapeetha Mysore and my sincere thanks to the Management of JSS Mahavidyapeetha Mysore.

We also acknowledge all the help rendered by the personnels of JSS Academy of Technical Education, Noida, Uttar Pradesh, Jain Institute of Technology, Davangere and Siddaganga Institute of Technology, Tumakuru, for their constant support in bringing out this master piece for the V semester CBCS Scheme (Common to E&CE/TC) of VTU, Belagavi, Karnataka.

Not but the least, we would like to thank all our family members for their infinite patience and understanding they have demonstrated in allowing us to allocate family time for the writing of this book. Finally, we are indebted to all the persons who have helped us in preparing this manuscript directly or indirectly by giving us valuable suggestions & moral support, in the sense we wish to express our profound thanks to all those people who have helped in making this book a reality.

October 2020

Authors

arunkumargowdru.1981@gmail.com

Cell: +91-8095382275

Syllabus

Module 1: **8051 Microcontroller:** Microprocessor Vs Microcontroller, Embedded Systems, Embedded Microcontrollers, 8051 Architecture- Registers, Pin diagram, I/O ports functions, Internal Memory organization. External Memory (ROM & RAM) interfacing.

Module 2: **8051 Instruction Set:** Addressing Modes, Data Transfer instructions, Arithmetic instructions, Logical instructions, Branch instructions, Bit manipulation instructions. Simple Assembly language program examples (without loops) to use these instructions.

Module 3: **8051 Stack, I/O Port Interfacing and Programming:** 8051 Stack, Stack and Subroutine instructions. Assembly language program examples on subroutine and involving loops - Delay subroutine, Factorial of an 8 bit number (result maximum 8 bit), Block move without overlap, Addition of N 8 bit numbers, Picking smallest/largest of N 8 bit numbers, Interfacing simple switch and LED to I/O ports to switch on/off LED with respect to switch status.

Module 4: **8051 Timers and Serial Port:** 8051 Timers and Counters – Operation and Assembly language programming to generate a pulse using Mode-1 and a square wave using Mode-2 on a port pin.

8051 Serial Communication- Basics of Serial Data Communication, RS-232 standard, 9 pin RS232 signals, Simple Serial Port programming in Assembly and C to transmit a message and to receive data serially.

Module 5: **8051 Interrupts and Interfacing Applications:** 8051 Interrupts. 8051 Assembly language programming to generate an external interrupt using a switch, 8051 C programming to generate a square waveform on a port pin using a Timer interrupt.

Interfacing 8051 to ADC-0804, LCD and Stepper motor and their 8051 Assembly language interfacing programming.

*****

Contents

Module 1 8051 Microcontroller

Module 2 8051 Instruction Set

Module 3 8051 Stack, I/O Port Interfacing and Programming

Module 4 8051 Timers and Serial Port

1

8051 Microcontroller

1.1 Introduction to Microprocessor & Microcontrollers

1.1.1 Microprocessor

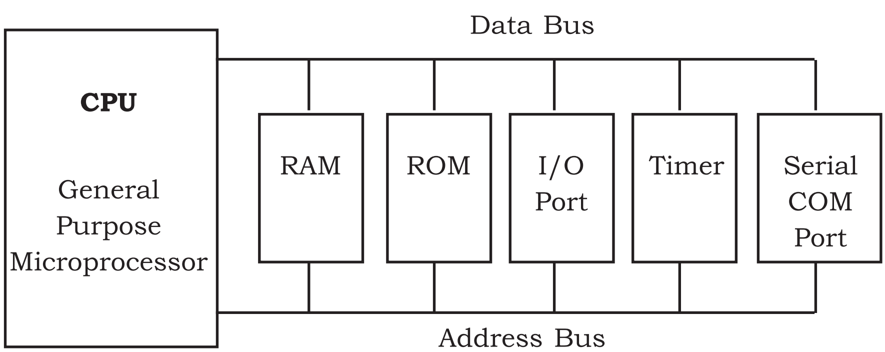


Fig. 1.1.1: Block diagram of Microprocessor.

- The microprocessor mainly contains CPU and general purpose registers. It **does not have built-in RAM, ROM, I/O ports** etc., on the **chip.**
- The microprocessors are commonly referred to as ***general-purpose microprocessor***.

  Examples:

  Intel: 8086, 80286, 80386, 80486, Pentium etc.

  Motorola: 68000, 68010, 68020, 68030 etc.

Note: The microprocessor is the heart of microcomputer.

1.1.2 Evolution of Microprocessors

- In 1971, Intel produced 4004, 4-bit microprocessor. The 4004 was a 4-bit device intended to be used with some other devices in making a calculator.
- In 1972, Intel came out with the 8008 which was capable of doing a function with 8-bit words.
- In 1974, Intel announced the 8080, which had a much larger instruction set than the 8008. The 8080 is referred to as a second generation microprocessor.
- After Intel produce the 8080, Motorola came out with the MC6800- 8-bit general purpose CPU and it required only a +5V supply rather than the -5V.
- The Intel produced the 8085 microprocessor with general purpose CPU which gives most or all of the computing power of earlier minicomputers.
- In 1978 Intel came out with the 8086 which is a full 16-bit processor. Intel 8086 was certainly the highest performance single-chip 16-bit microprocessor when it was first introduced.
- Soon after 8086, Motorola introduced the 16-bit MC68000. The 8086 and the MC68000 work directly with 16-bit words instead of 8-bit words.
- In 1979, Intel 8088.the first microprocessor to make a real splash in the market was introduced, and the evolution has continued, the PC market moved from 8088 to 80286, 80386 and 80486, Pentium, Pentium II, Pentium III, and Pentium 4.

  Pentium 4 is about **5,000 times faster** than **8088**.

Note: Intel makes all these microprocessors and all of them are improvements of design base of the 8088.

1.1.3 Microcontrollers

CPU	RAM	ROM
I/O	Timer	Serial COM Port

Fig. 1.1.2: Block diagram of Microcontroller.

- A microcontroller is a single integrated circuit which is dedicated to perform one task and execute one specific application.

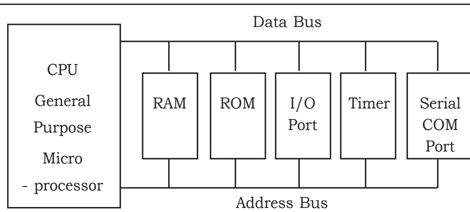
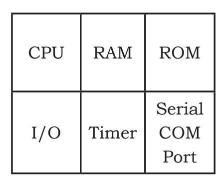
- Microcontroller has a CPU (a microprocessor) and in addition it has built-in RAM, ROM, Input/output devices, Timers/Counters on a single chip.

  Examples: 8051, 8052, ARM processor etc.

Note:

Fig. 1.1.3: Shows applications of microcontroller

1.2 Microprocessor V/s Microcontroller

Sl.No	Microprocessor	Microcontroller
1	 ***Fig.1.2.1 Block diagram of microprocessor***	 ***Fig.1.2.2 Block diagram of microcontroller***
2	It **contains only CPU.** The RAM, ROM, Input/output devices, timers & counters are separately interfaced.	Microcontroller has a CPU (a microprocessor) and in addition it has built-in RAM, ROM, Input/output devices, Timers/Counters on a single chip.
3	Designers decide the amount of ROM, RAM and Input /output ports etc.	Fixed amount of on-chip ROM, RAM and Input /output ports etc.
4	It has many instructions to move data between memory & CPU.	It has one or two instructions to move data between memory & CPU.

5	It has one or two bit handling instructions.	It has many bit handling instructions. **Ex:** CLR C, SETB P1.0 etc.
6	It has single memory for data & program.	It has separate memory for data & program.
7	Access time for memory & I/o devices are more.	Less access time for built – in memory & I/o devices.
8	Large number of instruction set.	Limited number of instruction set.
9	Few pins are multifunctioned.	More number of pins are multifunctioned.
10	Very few bit handling instructions	Many bit handling instructions
11	Design is very flexible	Design is less flexible
12	Versatile.	Not versatile.
13	High cost	Low cost
14	General-purpose applications.	Single-purpose applications in which cost, space & power are critical.
15	Examples: Intel: 8086, 80286, 80386, 80486, Pentium etc. Motorola: 68000, 68010, 68020, 68030 etc.	Examples: 8051, 8052, ARM processor, PIC controllers etc.

1.2.1 RISC and CISC

Reduced Instruction Set Computer (RISC)

- *The* ***Reduced Instruction Set Computer*** is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.
- A simplified instruction set provides higher performance when combined with microprocessor architecture capable of executing those instructions using fewer microprocessor cycles per instruction.

Complex Instruction Set Computer (CISC)

- The ***Complex Instruction Set Computer*** architecture is a type of microprocessor design containing a large set of computer instructions that range from very simple to very complex and specialized.

- In CISC architecture, single instruction can execute several low-level operations such as a load from memory, an arithmetic operation, and a memory store or capable of multi-step operations or addressing modes within single instruction.

Difference between RISC and CISC processors

❖ ***Compare the features of RISC and CISC***

Sl. No.	RISC	CISC
1	Only few instructions.	Many instructions.
2	Highly pipelined.	Not pipelined or Less pipelined.
3	Instruction executed by the hardware.	Instruction interpreted by the micro program.
4	Simple instructions taking one cycle.	Complex instructions taking multiple cycles.
5	Multiple register set.	Single register set.
6	Very few instructions refer memory.	Most of instructions may refer memory.
7	Fixed length instructions.	Variable length instructions.
8	Complexity is in the compiler.	Complexity is in the micro-program.
9	Few addressing modes.	Many addressing modes.
10	Only Load/Store instructions access memory	Many instructions can access memory.
11	Coding in RISC processor requires more number of lines. i.e. program size is large	Coding in CISC processor is simple i.e. program size is small
12	It has multi-clock.	It has single-clock.
13	Examples: PIC Microcontroller series etc.	Examples: INTEL 80286, 80386 etc.

1.2.2 Harvard and Von Neumann architectures

Von Neumann architectures

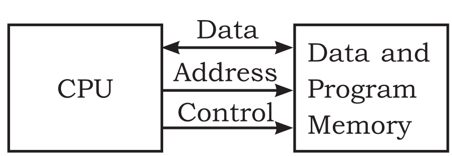


Fig. 1.2.3: Block diagram of Von Neumann architectures

- It is also referred to as Princeton architecture.
- Von Neumann architecture has single memory storage to hold both program instructions and data.
- The CPU can either read an instruction or data from the memory one at a time or write data to memory because instructions and data are accessed using same bus system.
- The advantage of Von Neumann architecture is simple design of microcontroller chip because only one memory is to be implemented which in turn reduces required hardware.
- The disadvantage is slower execution of program.

Harvard

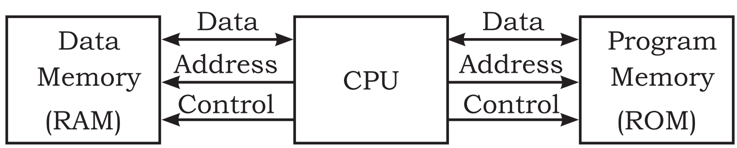


Fig. 1.2.4: Block diagram of Harvard architecture

- Harvard architecture has physically separate memory storage to hold program instructions and data i.e. separate program and data space.
- Since it has separate buses to access program and data memory, it is possible to access program memory and data memory simultaneously.
- The advantage of a Harvard architecture microcontroller is that it is faster for a given circuit complexity because it offers greater amount of parallelism.
- The disadvantage is that it requires more hardware, because two sets of buses and memory blocks are required.

1.2.3 Selection of Microcontrollers

❖ ***List the points to be considered during the selection of a microcontroller for an application.***

The three criteria in choosing microcontrollers are as follows:

1. Microcontroller must perform the required task efficiently & cost effectively i.e.
   - **Speed**
   - **Amount of RAM & ROM on chip**
   - **Power consumption**
   - **The number of input pins & the timer on the chip**
   - **Cost per unit**
   - **Easy to upgrade**

- **Packaging** *(The number of pins & the packaging format. This determines the required space & assembly layout.)*

2. Availability of assembler, debuggers, complier, emulator, technical support and expertise both in-house and outside.
3. Microcontroller availability in needed quantities both now and in the future.

1.3 Embedded Systems

An embedded system uses microprocessors or microcontrollers with software embedded in it to do one task only.

Example: A printer performs one task only.

1.4 Embedded Microcontrollers

A microcontroller can be considered as a system with a processor, memory and peripheralsand can be used as an embedded system.

The majority of microcontrollers in use today are embedded in other machinery, such as automobiles, telephones, home appliances, and peripheralsfor computersystems.

1.5 8051 Architecture - Registers

Central processing unit (CPU):

- The 8051 Central processing unit consists of 8-bit arithmetic & Logic unit (ALU), Registers: A, B, PSW, SP, 16 bit program counter & "Data pointer registers" (DPTR).
- The ALU can perform arithmetic functions on 8-bit data i.e. addition, subtraction, multiplication & division.
- Similarly, the logic unit perform logical operations such as AND, OR, NOT etc.

Register

Registers are used to store information temporarily, while the information could be

- a byte of data to be processed, or
- an address pointing to the data to be fetched

Majority of 8051 register are 8-bit registers. The most widely used registers are

- **Accumulator** (A), for all arithmetic and logic instructions.
- **B, R0, R1, R2, R3, R4, R5, R6, R7.**
- **DPTR** (data pointer) and PC (Program Counter).

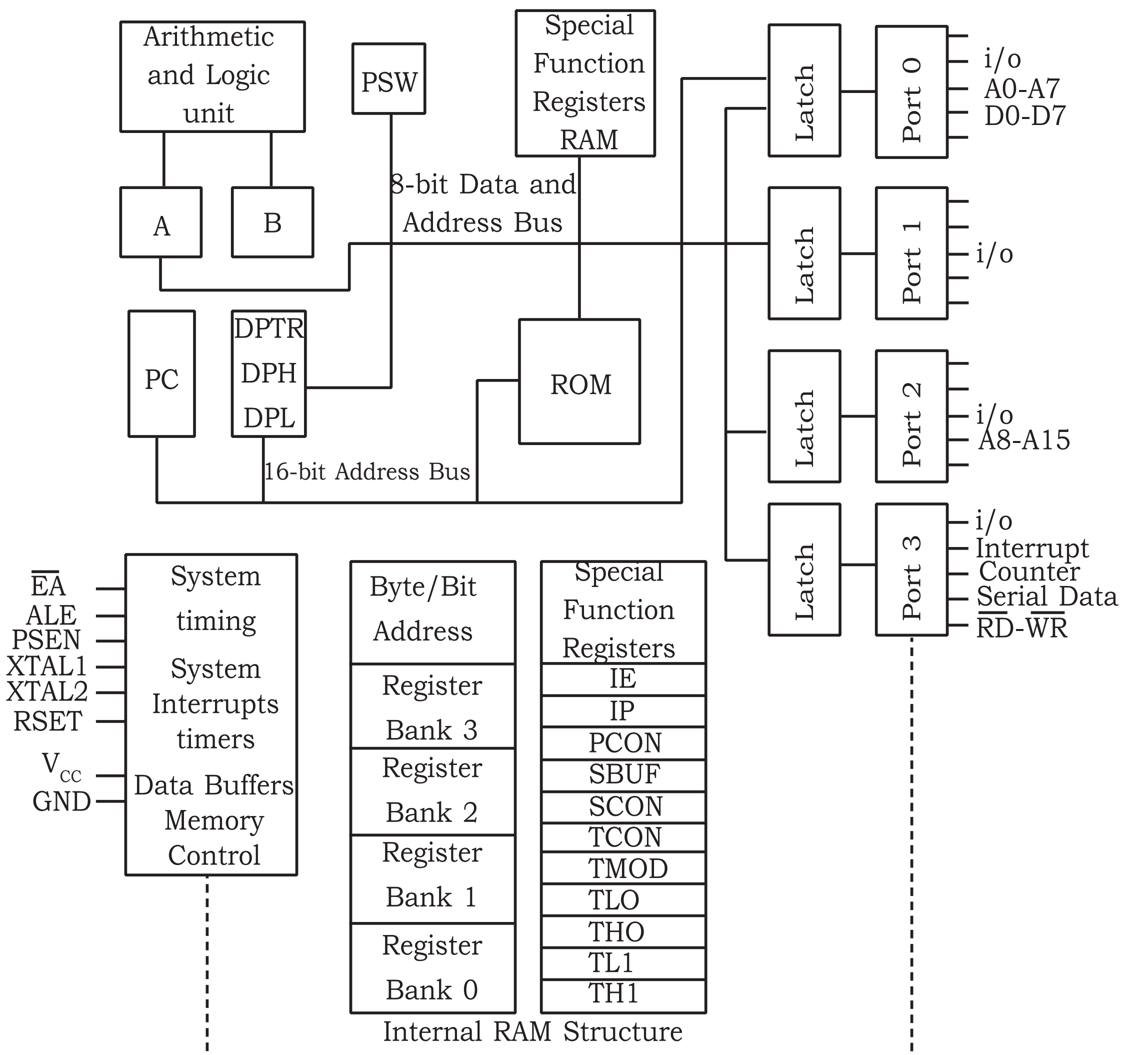


Fig.1.5.1: Detailed block diagram of 8051 Microcontroller

A register (Accumulator):

- Accumulator is a 8 bit register & is widely used for arithmetic and data transfer operations.
- The arithmetic operations are addition, subtraction, multiplication, division & Boolean bit manipulating etc.
- The data transfer operation between the 8051 microcontroller and any external memory.

Note: It can be accessed through its SFR address of 0E0H.

B-register:

- B-register to store 8-bit result of multiplication & division operations
- It is used as temporary register where data may be stored.

Note: It can be accessed through its SFR address 0F0H.

BUS:

Bus is a collection of wires which work as a communication channel or medium for transfer of data. The 8051 has two types of buses:

1. Address Bus (16-bits) and **2. Data Bus** (8-bits)

Program Counter (PC):

- PC is a16-bit register which points to the address of the next instruction to be executed.
- The PC is automatically incremented after every instruction byte is fetched.
- PC is the only register that does not have an **internal address.**
- When 8051 is RESET, the default value of PC is **0000 H.**

Input-Output ports (I/O Ports):

- The 8051 has 4 Input/output ports i.e. PORT 0, PORT 1, PORT 2 and PORT 3 and each port has 8 Input/output pins.
- It has total 32 Input/output pins and each pin can be configured as input or output pin.

Times & Counters

Fig.1.5.2: Timer 0 & Timer 1

The 8051 has two **16-bit** timers/counters, they can be used either as

- **Timers** to generate a time delay or as
- **Event counters** to count events happening outside the microcontroller.

The two timers are

i) Timer/Counter T0 and

ii) Timer/Counter T1

- Each register can be used either as Timer or counter and can be divided into two 8-bit registers called Timer Low (TL) and Timer High (TH).

STACK (8-bit)

- The stack is a **section of RAM** used by the CPU to **store information temporarily**. This information could be **data** or an **address.**
- The register used to access the stack is called the SP (stack pointer) register. The stack pointer in 8051 is only 8 bits wide
- The storing of a CPU register in the stack is called a **PUSH**, and loading the contents of the stack back into a CPU register is called a **POP**.

Data pointer (DPTR):

DPH	DPL
83 h	82 h
8 bit	8 bit

Fig.1.5.3: DPTR

- DPTR is a 16-bit register, which holds a 16-bit address.
- DPTR can be split into 2 parts:
  - **DPL:** Data pointer Low byte having internal address 82h.
  - **DPH:** Data pointer high byte having internal address 83h.
- DPTR is very useful for **string operations** and **look up table** operations.

Memory Organization:

The 8051 microcontroller's memory is divided into

1. **Internal RAM – 128 bytes:** Used for temporarily storing and keeping intermediate results and variables.
2. **ROM – 4K bytes:** Used for permanent saving program being executed

Special Function Registers (SFR):

The operations of 8051 are done by a group of specific internal registers, each called a special function Register (SFR).

Interrupts:

The 8051 Microcontroller has 6 interrupts:

RESET, $\overline{\text{INT0}}$, $\overline{\text{INT1}}$, Timer 0,Timer 1 Serial Port (TI/RI)

Program status Word (PSW) or Flag register:

CY	AC	F0	RS1	RS0	OV	-	P
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

Fig.1.5.4: Program Status Word.

- The program status word (PSW) register is an 8-bit register. It is also referred to as the flag register. Only 6 bits are used by the 8051. The two unused bits are user-definable flags.
- The Carry (CY), Auxiliary carry (AC), Overflow (OV) and Parity (P) are called **Conditional flags** because these flags indicate some conditions after the instruction is executed.

Carry Flag (CY):

After performing arithmetic & logic operation if there is a carryout from the MSB (D_7 i.e. 7^{th}- bit) then CY = 1, otherwise CY = 0

Auxiliary carry Flag (AC):

After performing arithmetic & logic operation if a carry from D_3 to D_4 bit then, AC = 1, otherwise AC = 0.

FO: Available for user for general purpose

RS1 & RS0: These two bits are used to select Register Bank and are shown in table 1.5.1.

Table 1.5.1: Register Bank Selector

RS1	RS0	Register Bank	Address
0	0	**Bank 0**	**00H-07H**
0	1	**Bank 1**	**08H-0FH**
1	0	**Bank 2**	**10H-17H**
1	1	**Bank 3**	**18H-1FH**

Overflow Flag (OV):

OV flag is set to 1 if either of the following two conditions occurs:

i. There is a carry from D_6 to D_7, but no carry out of D7 (CY = 0).

ii. There is a carry out from D_7 bit (CY = 1) but no carry from D_6 to D_7 bit.

Parity Flag (P):

Parity flag indicates the number of 1's present in the accumulator.

i. If the number of 1's in the accumulator is odd then P = 1.

ii. If the number of 1's in the accumulator is even then P = 0.

1.5.1 Variants of MCS-51 family and their features

The variants of MCS-51 family are

1. 8051 2. 8052 3. 8751 4. 8752 5. 8031 and 6. 8032

MCS-51 family are

Feature	8051	8052	8751	8752	8031	8032
			EPROM			
On-Chip ROM	4K	8K	4k	8K	0K	0K
RAM (Bytes)	128	256	128	256	128	256
Timer	2	3	2	3	2	3
I/O Ports	4	4	4	4	2	2
Seral Port	1	1	1	1	1	1
Interrupt Sources	6	8	6	8	6	8

1.5.2 Applications of Microcontrollers

The applications of microcontroller are:

Home Appliances	Office	Automobiles
TVs, VCR, Camcorders, Remote Controller, Video games, Cellular phones, Telephones, Paging, Camera, Answering machines, Musical Instruments etc.	Telephones, Computers, Security Systems, Fax Machines, Copiers, Laser Printer, Color Printer etc.	Engine control, ABS, Air bags, Transmission control, Climate Control, Keyless entry, Trip computer etc.

Example 1-1

Show the status of the CY, AC and P flag after the addition of 38H and 2FH in the following instructions.

MOV A, #38H

ADD A, #2FH ; after the addition A=67H, CY=0

Solution:

				1	1	1			
38		0	0	1	1	1	0	0	0
+ **2F**	+	0	0	1	0	1	1	1	1
67		0	1	1	0	0	1	1	1

CY = 0 since there is no carry beyond the D7 bit.

AC = 1 since there is a carry from the D3 to D4 bit.

P = 1 since the accumulator has an odd number of 1s (it has five 1s).

Example 1-2

Show the status of the CY, AC and P flag after the addition of 9CH and 64H in the following instructions.

MOV A, #9CH

ADD A, #64H ; after the addition A=00H, CY=1

Solution:

		1	1	1	1	1	1			
	9C		1	0	0	1	1	1	0	0
+	**64**	+	0	1	1	0	0	1	0	0
1	**00**	**1**	0	0	0	0	0	0	0	0

CY = 1 since there is a carry beyond the D7 bit

AC = 1 since there is a carry from the D3 to the D4 bit

P = 0 since the accumulator has an even number of 1s (it has zero 1s)

Example 1-3

Show the status of the CY, AC and P flag after the addition of 88H and 93H in the following instructions.

MOV A, #88H

ADD A, #93H ; after the addition A=1BH, CY=1

Solution:

		1								
	88		1	0	0	0	1	0	0	0
+	**93**	+	1	0	0	1	0	0	1	1
1	**1B**	**1**	0	0	0	1	1	0	1	1

CY = 1 since there is a carry beyond the D7 bit.

AC = 0 since there is no carry from the D3 to D4 bit.

P = 0 since the accumulator has an even number of 1s (it has four 1s).

1.5.3 Special Function Register (SFR)

The functions of any 5 SFR can be explained i.e. Accumulator, PSW, B register, Ports, Timers, DPTR etc.

F8h									FFh
F0h	B								F7h
E8h									EFh
E0h	ACC								E7h
D8h									DFh
D0h	PSW								D7h
C8h									CFh
C0h									C7h
B8h	IP								BFh
B0h	P3								B7h
A8h	IE								AFh
A0h	P2								A7h
98h	SCON	SBUF							9Fh
90h	P1								97h
88h	TCON	TMOD	TL0	TL1	TH0	TH1			8Fh
80h	PO	SP	DPL	DPH				PCON	87h

Fig.1.4.7: Special Function Register.

- The operations of 8051 are done by a group of specific internal registers; each called a Special Function Register (SFR) and its address ranges from **80H** to **FFH** (80 bytes).
- There are 21 Special function registers (SFR) in 8051 micro controller and these are Register A, Register B, PSW, PCON etc. and each of these registers are of 1 byte size. Some of these special function registers are bit addressable, while some are byte addressable.

1.5.4 FEATURES OF 8051 MICROCONTROLLER

❖ ***List the features of 8051 microcontroller*** **5-Marks**

The Feature of 8051 are as follows

1. *8-bit CPU.*
2. *4 Kbytes Internal ROM (Program Memory).*
3. *128 bytes Internal RAM (Data Memory).*
4. *It has four 8-bit ports (Port 0, 1, 2, & 3), total 32 input/output lines.*
5. *Two 16-bit timers (T0 & T1).*
6. *One Full duplex serial communication port (data Transmitter/ Receiver).*
7. *Six Interrupt sources.*

8. *16-bit program counter (PC) & data pointer (DPTR).*
9. *8-bit Stack pointer.*

1.6 8051 PIN DETAILS

The8051 microcontroller is a dual in-line pin packages has **40 pins**, out of which **32 pins** are assigned for **Ports P0, P1, P2** and **P4**, where each ports takes **8 pins**.

The rest of the pins are **V_{cc}, GND, XTAL1, XTAL2, RST, $\overline{\text{EA}}$, ALE/ $\overline{\text{PROG}}$ and $\overline{\text{PSEN}}$**

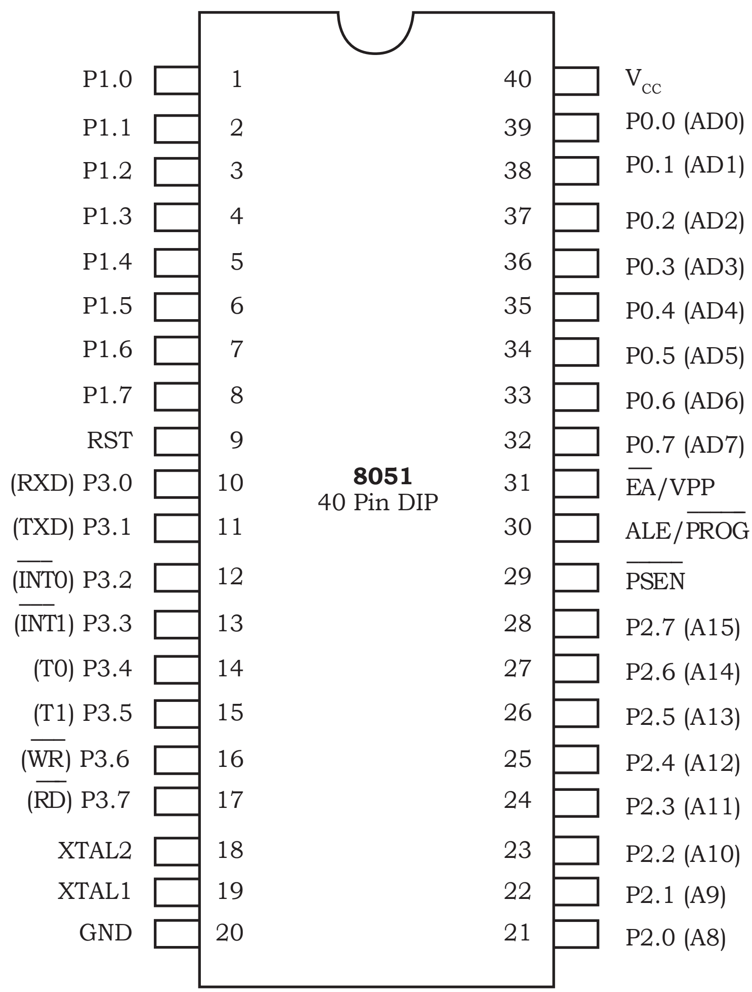


Fig. 1.6.1: Pin diagram of 8051 microcontroller

Pin No	Pin Name	Description
1-8	**Port1 (P1.0-P1.7)**	Port 1 is an 8-pin bi-directional Port. Each of these pins can be configured as either input or output pins.
9	**RST (Reset)**	• When a pulse (square wave) is applied to this pin, microcontroller will terminate all its activities & reset. • **Program counter** is loaded with **0000**.
10-17	**Port 3 (P3.0-P3.7)**	Port 3 is an 8-pin bi-directional Port with **dual function**. Each of these pins can be configured as either input or output pins.
10-11	**RXD & TXD**	• In 8051, the data is received from or transmitted to **RXD** & **TXD** pins. • The **data** is **transmitted** out of 8051 through the **TXD** line. • The **data** is **received** by 8051 through the **RXD** line.
12-13	$\overline{\textbf{INT0}}$ **&** $\overline{\textbf{INT1}}$	• The 8051 has two external hardware interrupts i.e., Interrupt 0 & Interrupt 1. These two pins are used in Timers/Counter operation. • These Pins are triggered by external circuits.
14-15	**T0 & T1**	• The 8051 has two 16-bit Timers/ Counters. T0-Timer0 register (16-bit) T1-Timer1 register (16-bit). • These can be used either as Timers to generate a time delay or as counters to count events happening outside the microcontroller. • Each 16-bit registers can be accessed as two separate 8-bit registers.
16-17	$\overline{\textbf{RD}}$ **&** $\overline{\textbf{WR}}$	These are active low pins. • When $\overline{RD}$ =0, microcontroller **reads** the **data** from **external RAM**. • When $\overline{WR}$ =0, microcontroller **writes** the **data** into **external RAM**.
18-19	**XTAL2 & XTAL1**	• The 8051 has an on-chip oscillator but requires an external clock to run it. • A Quartz crystal oscillator is connected to inputs XTAL1 & XTAL2 with two capacitors having values 30Pf.

18-19 **(continued)**	**XTAL2 & XTAL1** **(continued)**	• If an **external frequency** (from AFO) has to be applied, then it must be applied between **XTAL1** & **ground**. **XTAL2** must be **left open**. • Oscillator frequency may vary from 4MHz to 40MHz.
20	**VSS**	It is a ground pin i.e. $\mathbf{V_{SS}=0V}$
21-28	**Port 2 (P2.0-P2.7)**	• Port 2 is an 8-pin bi-directional Port. • If external memory is not used, these pins can be used as either input or output pins. • If **external memory** is used then the **higher address** i.e. $\mathbf{A_8\text{-}A_{15}}$ will appear on this port.
29	$\overline{\textbf{PSEN}}$ **(program store Enable)**	• $\overline{\textbf{PSEN}}$ is an active low input to 8051. • $\overline{\textbf{PSEN}}$ is an **output** pin used to **access** the **external program memory** (ROM).This pin is connected to the **OE pin** of the ROM.
30	**ALE/** $\overline{\textbf{PROG}}$ **(Address Latch Enable)**	• ALE is an output pin. It is used for demultiplexing the address and the data bus. • When **ALE=1**, **Port 0** is providing **lower order address** ($\mathbf{A_0\text{-}A_7}$). • When **ALE=0**, **Port 0** is used as **data lines** ($\mathbf{D_0\text{-}D_7}$). • This pin also has program pulse input $\overline{\textbf{PROG}}$ during EEPROM programming.
31	$\overline{\textbf{EA}}$**/ VPP(External Access Enable/ Programming supply voltage)**	• $\overline{\text{EA}}$ is an active low input to 8051. • When $\overline{\text{EA}}$ is connected to VCC i.e. $\overline{\text{EA}}$ =1, the 8051 can access 4 K bytes of internal ROM i.e. 0000 H to 0FFF H and external ROM of 60 K bytes i.e. **1000 H to FFFF H**. • When $\overline{\text{EA}}$ is connected to GND i.e. $\overline{\text{EA}}$ =0, then all program fetches are directed to external ROM i.e. **0000 H** to **FFFF H**.
32-39	**Port0 (P0.0-P0.7)**	• Port 0 is an 8-pin bi-directional Port. • Port 0 is also multiplexed low order address and data bus i.e. $AD_0\text{-}AD_7$. • If external memory is not used, then these pins can be used as either input or output pins.

32-39 (continued)	**Port0 (P0.0-P0.7)** (continued)	• If external memory is used then the lower address and data linesAD_0-AD_7 will appear on this port. • When **ALE=1**, **Port 0** is providing **lower order address** (**A_0-A_7**). • When **ALE=0**, **Port 0** is used as **data lines** (**D_0-D_7**).
40	**V_{cc}**	DC power supply +5V is connected to this pin.

1.7 I/O PORTS FUNCTIONS

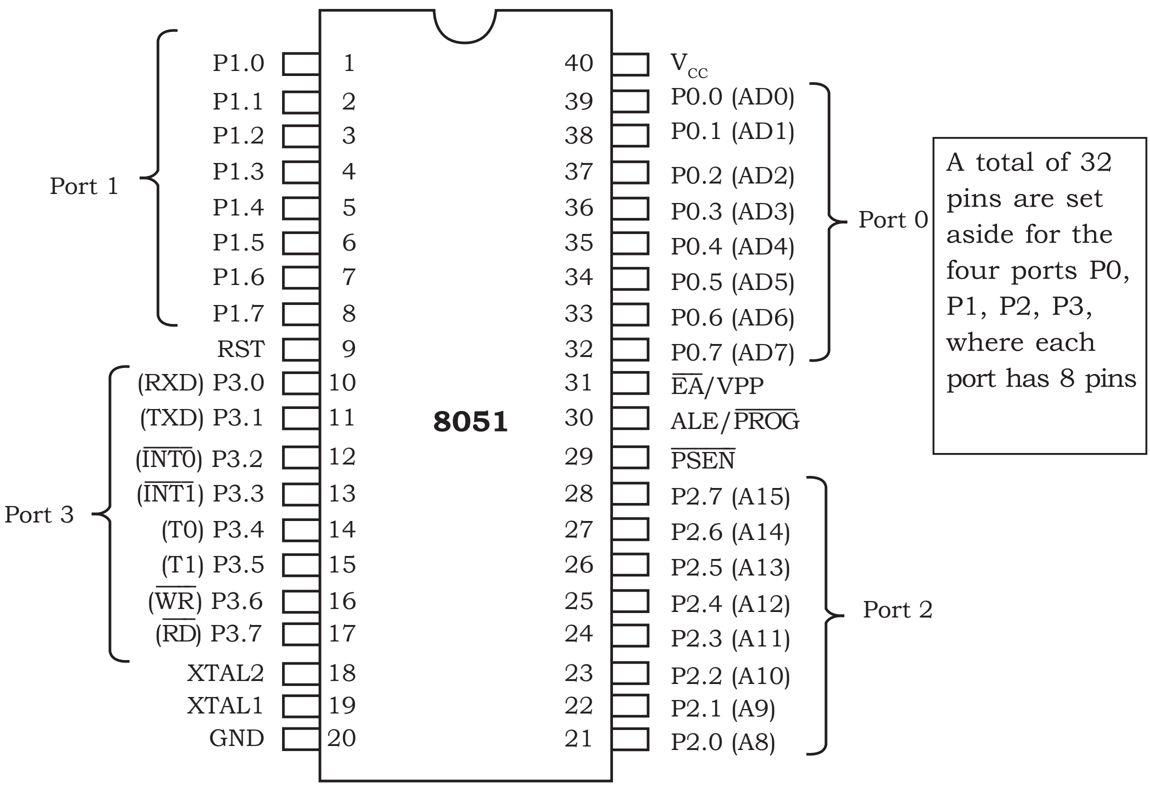


Fig.1.7.1: Pin diagram of 8051 showing I/O ports

Features of I/O ports

- An 8051 microcontroller has four I/O ports P0, P1, P2 & P3, where each port has 8 bits i.e. total 32 I/O pins which can be configures as input or output ports.
- All the ports upon reset are configured as input, ready to use it as input port.

- To use any of these ports as an input port, it must be programmed by writing 1 to all the bits.
- To use any of these ports as an output port, it must be programmed by writing 0 to all the bits.

Examples: (Refer After reading instruction set)

i) Port 0 configured as output port

```
MOV A, #00H   ; A=00H
MOV P0, A     ; Make Port 0 as output port.
```

ii) Port 1 configured as input port

```
MOV A, #0FFH ; A=FFH
MOV P1, A    ; Make Port 1 as input port.
```

iii) Port 2: P2.0 to P0.3 configured as input & P2.4 to P2.7 configured as output port.

```
MOV A, #0F0H ; A=F0H
MOV P2, A    ; Make P2.0 to P2.3 as output & P2.4 to P2.7 as input pins.
```

1.7.1 PORT 0 (Pins 32-39)

- Port 0 occupies a total of 8 pins.
- To use the pins of PORT 0 as both input and output port, each pin must be **connected** externally to **10 KΩ pull-up** resistors because **P0** is an **open drain**.
- Upon **reset**, Port 0 is configured as **input port**.

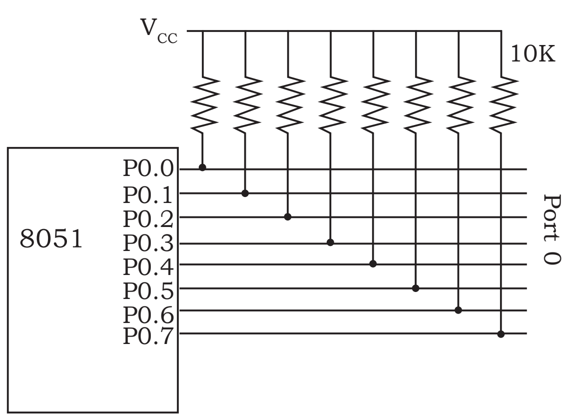


Fig.1.7.2: Port 0 with 10 KΩ Pull-Up Resistors

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

- When connecting an 8051 to an external memory, port 0 provides both address and data.
- Port 0 provides the lower 8 bits address via A0 – A7

Example 1.4:(Refer After reading instruction set)

Write an Assembly Language Program (ALP) to toggle Port 0 continuously.

Solution:

The P0 is 1st loaded with 55H = 01010101 and its complement i.e. 10101010 = AAH is given to P0 continuously.

```
BACK: MOV A, #55H
      MOV P0, A
      ACALL DELAY         ; Delay routine not shown
      MOV A, #0AAH
      MOV P0, A
      ACALL DELAY         ; Delay routine not shown
      SJMP BACK
```

Example 1.5: (Refer After reading instruction set)

Write an ALP to configure Port 0 first as an input port and then data is received from P0 and sent to P1.

Solution:

```
      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       ; keep doing it
```

1.7.2 PORT 1(Pins 1-8)

- Port 1 occupies a total of 8 pins.
- Port 1 does not need any pull-up resistors since it already has pull-up resistors internally.
- Upon reset, Port 1 is configured as input port.

Example 1.6: (Refer After reading instruction set)

Write an ALP to continuously send out to port 0 the alternating value 55H and AAH.

Solution:

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

Example 1.7: (Refer After reading instruction set)

Port 1 is configured first as an input port by writing 1s to it and then data is received from that port and saved in R3 and R1.

Solution:

```
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 R3, A           ; save it to in register R3
ACALL DELAY         ; wait
MOV A, P1           ; another data from P1
MOV R1, A           ; save it to in register R1
```

1.7.3 PORT 2(Pins 21-28)

- Port 2 occupies a total of 8 pins.
- Port 2 does **not need** any **pull-up resistors** since it already has pull-up resistors internally.
- Upon reset, Port 2 is configured as **inputport**.
- In many 8051-based system, P2 is used as **simple I/O**. Port 2 is also designated as **A8 - A15,** indicating its **dual function**.

1.7.4 PORT 3(Pins 10-17)

- Port 3 occupies a total of 8 pins.
- Port 3 does **not need** any **pull-up resistors** since it already has pull-up resistors internally.
- Upon **reset**, Port 3 is configured as **input port**.
- Port 3 has the additional functions of proving some extremely important signals such as interrupts.

Table 1.7.1 PORT 3 Alternative Functions

P3 bit	Pin No.	Function	Description
P3.0	10	**RxD**	Receive data for serial port
P3.1	11	TxD	Transmit data for serial port
P3.2	12	$\overline{\text{INT0}}$	External interrupt 0
P3.3	13	$\overline{\text{INT1}}$	External interrupt 1
P3.4	14	**T0**	Timer/counter 0
P3.5	15	**T1**	Timer/counter 1
P3.6	16	$\overline{\text{WR}}$	External data memory write strobe
P3.7	17	$\overline{\text{RD}}$	External data memory read strobe

Note: Table 1.7.2 RESET Values Of 8051 ports

PORTS	RESET Values	
	Binary	Hex
P0	11111111	FF
P1	11111111	FF
P2	11111111	FF
P3	11111111	FF

1.8 Memory Organization

The 8051 microcontroller's memory is divided into

1. Data Memory (RAM) and
2. Program Memory (ROM).

- The Data Memory is used for temporarily storing data, keeping intermediate results and variablesused during the operation of the microcontroller.
- The Program Memory is used for permanent saving program being executed.

Data Memory (RAM)

The Data memory is of two types:

1. Internal RAM and 2. External RAM.

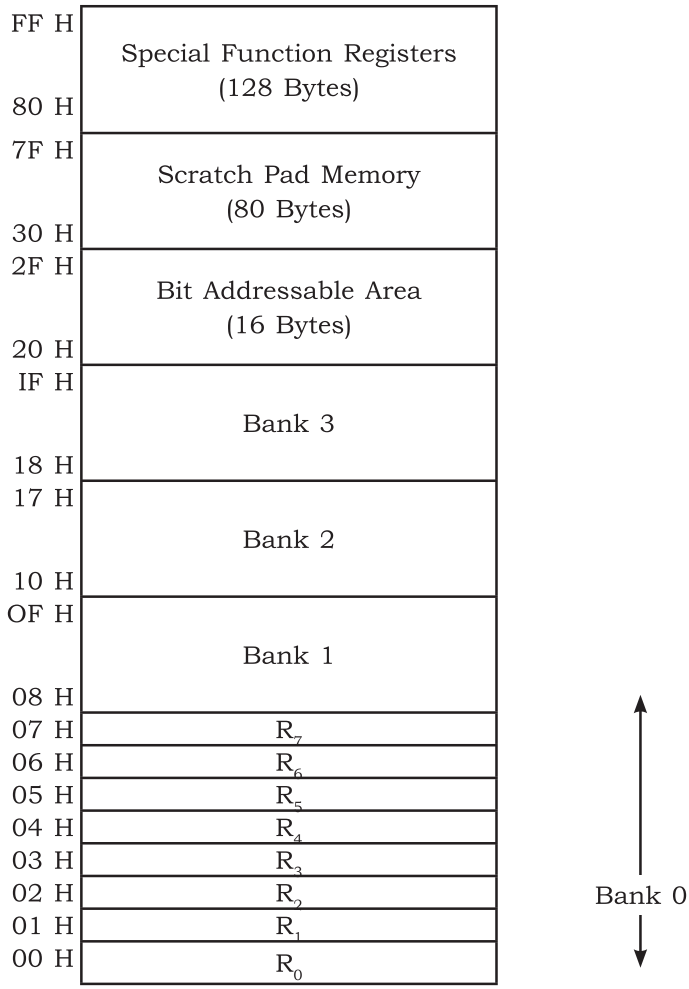
1. Internal RAM

The internal data memory consists of 256 bytes; these are divided into two parts:

i) **Internal data RAM**- 00H-FFH (128 bytes)

ii) **Special function registers**- 80H-FFH (128 bytes)

Internal data RAM



The Internal data RAM is divided into 3 parts:

1) Register banks or General purpose RAM, 2) Bit addressable area
3) Scratch pad area

Register banks or General purpose RAM

The section of the Register Banks and their addresses are given below.

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

- The 8051 microcontroller consists of **four** register banks: **Bank 0, Bank 1, Bank 2, Bank 3.** Each register bank contains **8 registers of 1 byte**. So total **32 register** in register bank.
- To change the register bank we have to set values of **PSW** register bits **RS0** and RS1.
- Only one register bank is in use at a time.
- When 8051 is RESET, by default register bank 0 is selected.

Note:

Hex	**Bank 0**	Hex	**Bank 1**	Hex	**Bank 2**	Hex	**Bank 3**
07	**R7**	0F	**R7**	17	**R7**	1F	**R7**
06	**R6**	0E	**R6**	16	**R6**	1E	**R6**
05	**R5**	0D	**R5**	15	**R5**	1D	**R5**
04	**R4**	0C	**R4**	14	**R4**	1C	**R4**
03	**R3**	0B	**R3**	13	**R3**	1B	**R3**
02	**R2**	0A	**R2**	12	**R2**	1A	**R2**
01	**R1**	09	**R1**	11	**R1**	19	**R1**
00	**R0**	0S	**R0**	10	**R0**	1S	**R0**

Fig. 1.8.2: Register Banks

The detailed diagrams of Register Banks are shown in Fig. 1.8.2.

Bit addressable RAM

Byte Address	Bit Address							
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	'48
28	47	46	45	44	43	42	41	46
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00

Fig. 1.8.3: Bit addressable RAM

- The area of bit addressable RAM is usually used to store bit variables.
- The address range from 20h to 2Fh (16 bytes) is bit-addressable RAM. Each bit can be accessed from 00H to 7FH.
- The total bit addressable location are 16 bytes x 8 bits = 128 bits.
- Each bit can be accessed from 00H to 7FH.
- The programming using bit addressable area saves wastage of memory.

Note: For example, Bit 0 of byte 20h has the bit address 0, and bit 7 of byte 2Fh has the bit address 7Fh).

Scratch pad area

- The upper 80 bytes are scratch pad area which is used for general purpose storing of data.
- The Scratch pad area is in the address range 30H to 7FH.
- The Scratch pad area can be used for stack memory.

Special function registers (SFR)

- The operations of 8051 are done by a group of specific internal registers; each called a Special Function Register (SFR) and its address ranges from **80H** to **FFH** (80 bytes).

F8h									FFh
F0h	B								F7h
E8h									EFh
E0h	ACC								E7h
D8h									DFh
D0h	PSW								D7h
C8h									CFh
C0h									C7h
B8h	IP								BFh
B0h	P3								B7h
A8h	IE								AFh
A0h	P2								A7h
98h	SCON	SBUF							9Fh
90h	P1								97h
88h	TCON	TMOD	TL0	TL1	TH0	TH1			8Fh
80h	PO	SP	DPL	DPH				PCON	87h

Fig. 1.8.4: Special Function Register

- There are 21 Special function registers (SFR) in 8051 micro controller and these are Register A, Register B, PSW, PCON etc. and each of these registers are of 1 byte size. Some of these special function registers are bit addressable, while some are byte addressable.
- SFRs are used to **control** timers, counters, serial ports, I/O ports and peripherals.

External RAM

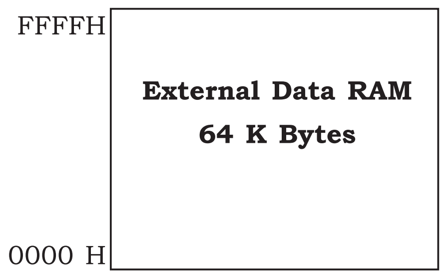


Fig. 1.8.5: External RAM

- External data memory is 64 K Bytes read/write memory.
- The external data memory is indirectly accessed through a **Data Pointer Register**, it is slower than access to internal data memory.

Program Memory

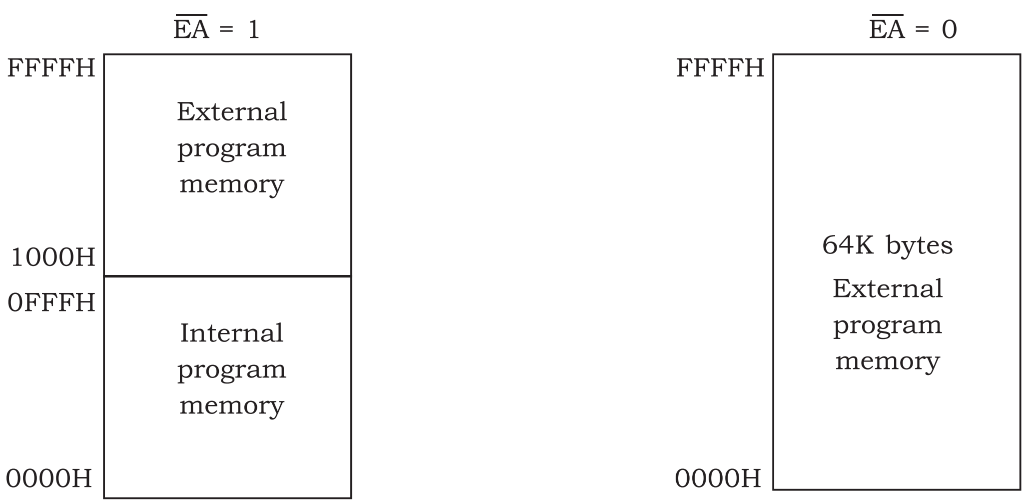


Fig. 1.8.6: Program memory (ROM)

- The 8051 microcontroller has an on chip internal program ROM of 4K size and if needed can add an external memory of size 60K maximum by interfacing i.e. total 64K size memory.

- The Program memory accessed through $\overline{EA}$ pin. The $\overline{EA}$ is an active low input.
- When $\overline{EA}$ is connected to VCC i.e. $\overline{EA}$ =1, the 8051 can access 4 K bytes of internal ROM i.e. 0000H to 0FFFH and external ROM of 60 K bytes i.e. **1000H to FFFFH**.
- When $\overline{EA}$ is connected to GND i.e., $\overline{EA}$ =0, then all program fetches are directed to external ROM i.e. **0000H** to **FFFFH**.

Pins of 8051 used for external memory interfacing and list their functions.
The pins which are used for external memory interfacing are:

$\overline{EA}$, $\overline{RD}$, $\overline{WR}$, $\overline{OE}$, $\overline{PSEN}$, & $\overline{ALE}$

Refer pin details of 8051 to explain the functions of each pin.

1.9 External Memory (ROM & RAM) interfacing

1.9.1 Interfacing External Data

- To address up to 64 K Bytes of external data memory then the hardware should be configured as shown in figure 1.6.1.
- The **MOVX** instruction is used to access the external data memory.
- The **Port 0** outputs the **low address** (**A_0 to A_7**) while **Port 2** outputs the **high address** (**A_8 to A_{15}**).

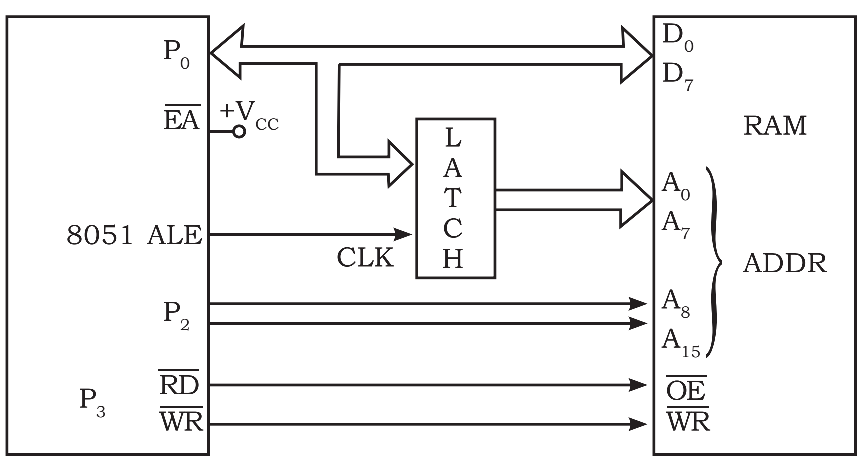


Fig.1.9.1: Interfacing 8051 with External Data Memory

- The Port 0 is a multiplexed address/data bus. The LATCH is used to **demultiplex address** and data bus. The LATCH will be enabled when ALE =1, so output of LATCH has lower order address **A_0-A_7** as shown in Fig 1.9.1.

- The $\overline{RD}$ **&** $\overline{WR}$ pins are used when a RAM has to be accessed.
- When $\overline{RD}$ = 0, a data byte can be read from a RAM location.
- When $\overline{WR}$= 0, a data byte can be written into a RAM location.

1.9.2 Interfacing External ROM

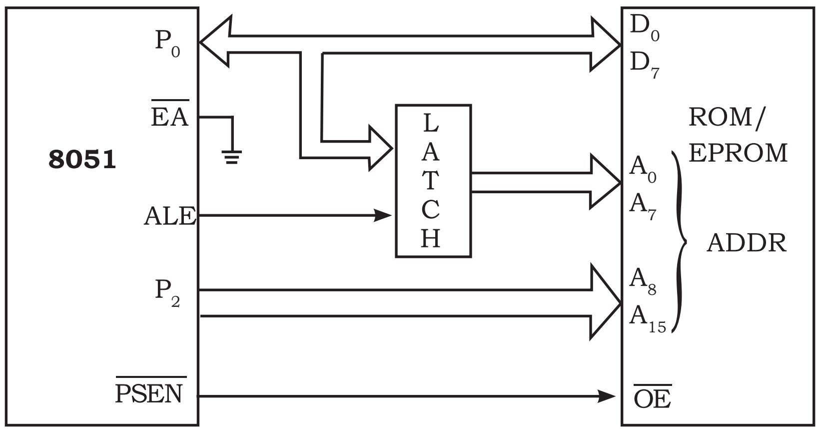


Fig.1.9.2: Interfacing 8051 with External Program Memory.

- To address up to 64 K Bytes of external Program memory then the hardware should be configured as shown in figure 1.9.2.
- The **Port 0** outputs the **low address** (**A_0 to A_7**) while **Port 2** outputs the **high address** (**A_8 to A_{15}**).
- The Port 0 is a multiplexed address/data bus. The LATCH is used to **demultiplex address** and data bus. The LATCH will be enabled when ALE=1, so output of LATCH has lower order address **A_0-A_7** as shown in Fig 1.6.2.
- The **MOVC** instruction is used to get data from code space.
- The $\overline{EA}$ is an active low input pin. When $\overline{EA}$ is connected to GND i.e. $\overline{EA}$ =0, then all program fetches are directed to external ROM i.e. **0000 H** to **FFFF H**.
- The $\overline{PSEN}$ is an active low pin used to **access** the **external program memory** (ROM), $\overline{PSEN}$ pin is connected to the $\overline{OE}$ pin of the ROM chip.
- To access the program code, $\overline{EA}$ must be grounded then $\overline{PSEN}$ will go low to enable the external ROM to place a byte of program code on the data bus.

Example 1.8:

Interface 4K RAM to 8051 Microcontroller

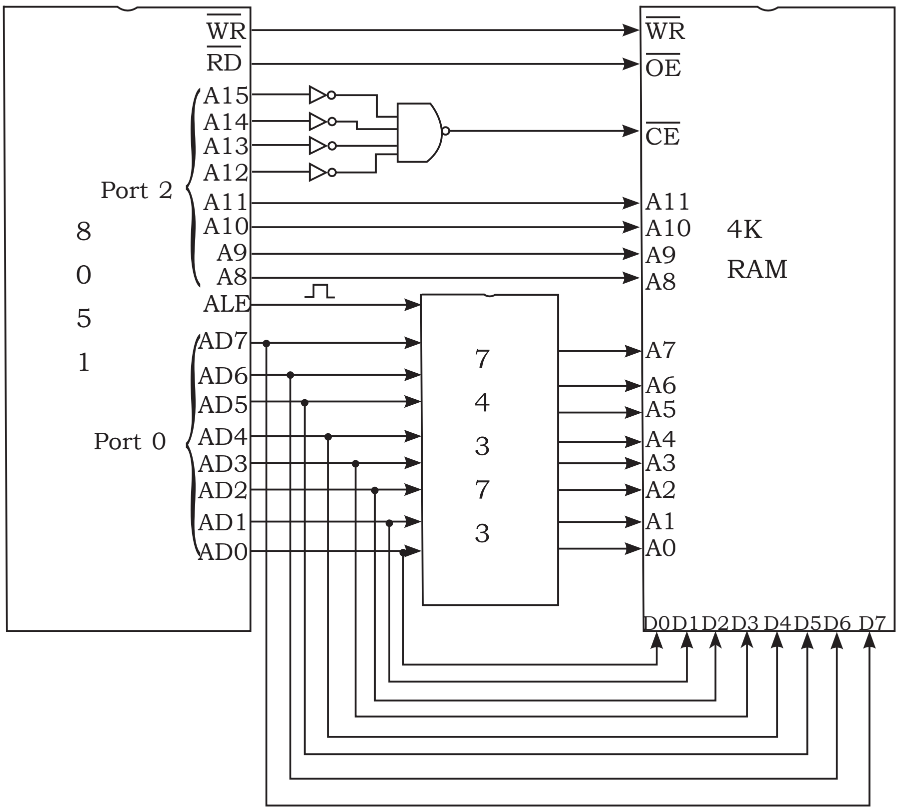


Fig.1.9.3: Interface 4K RAM to 8051 Microcontroller

Example 1.9:

Interface 4K ROM to 8051 Microcontroller

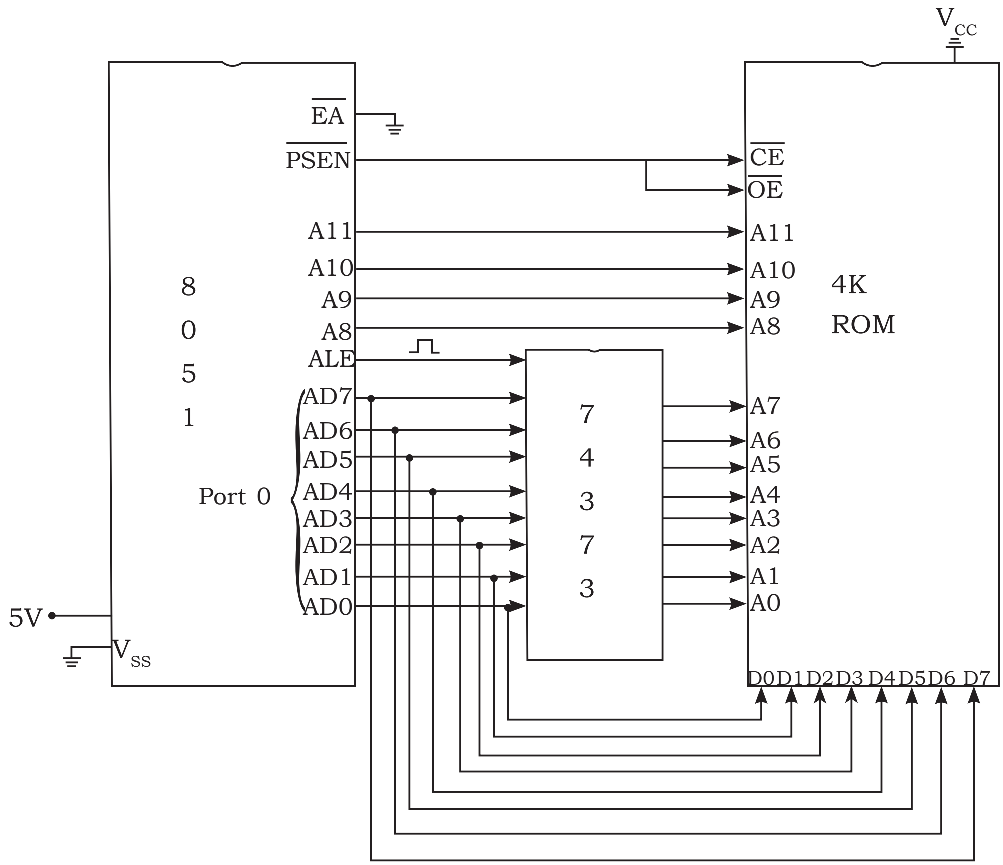


Fig.1.9.4:Interface 4K ROM to 8051 Microcontroller

Example 1.10:

Interface 8K RAM to 8051 Microcontroller

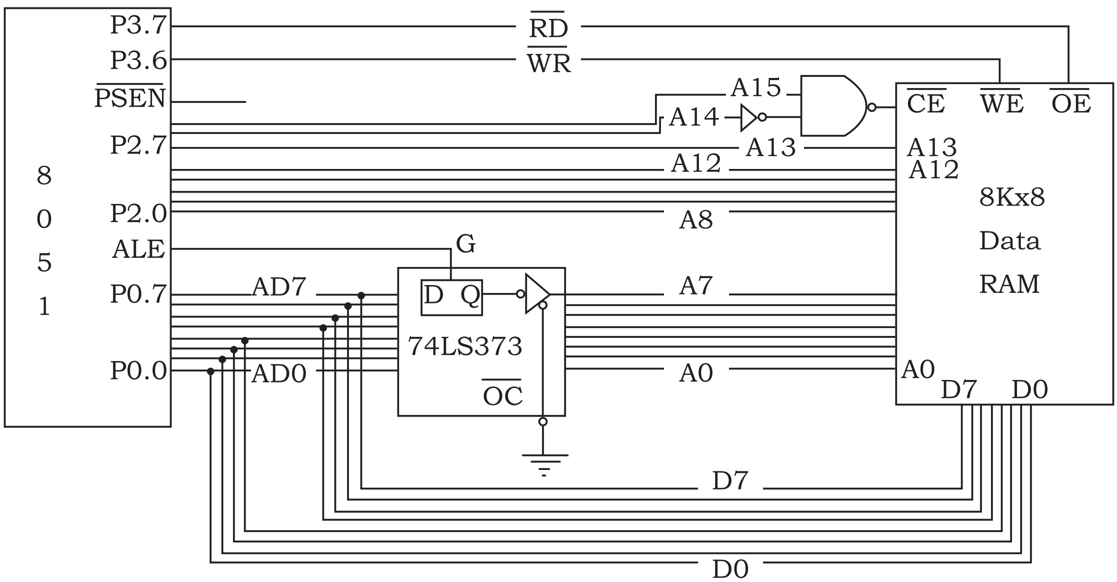


Fig.1.9.6: Interface 8K RAM to 8051 Microcontroller

Example 1.11:

Interface 8K ROM to 8051 Microcontroller

❖ ***Describe the method of interfacing 8K PROM to 8051 microcontroller 10-Marks***

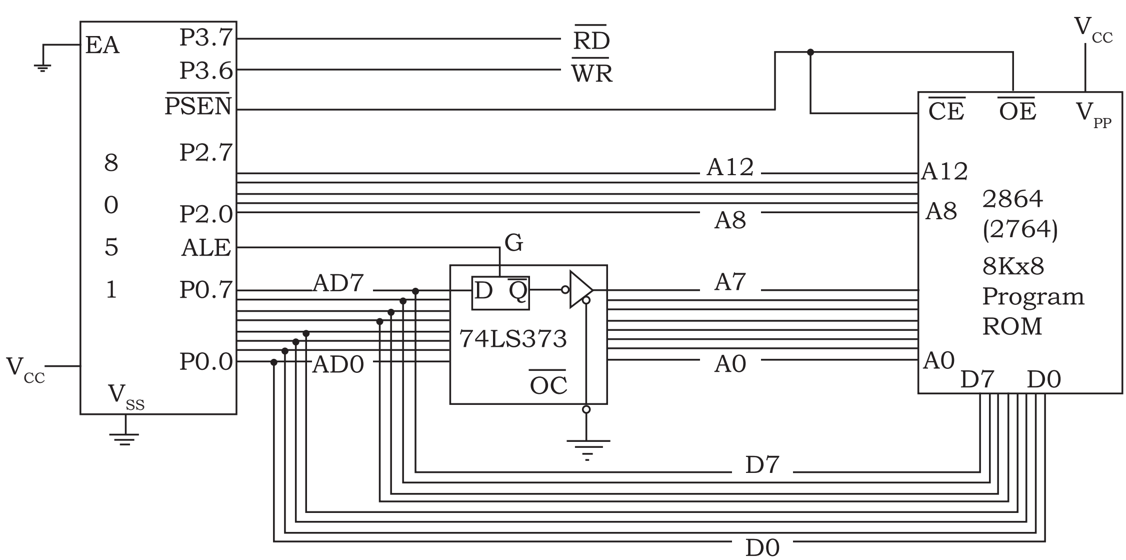


Fig.1.9.7a: Interface 8K ROM to 8051 Microcontroller

Interface 8K DATA ROM to 8051 Microcontroller

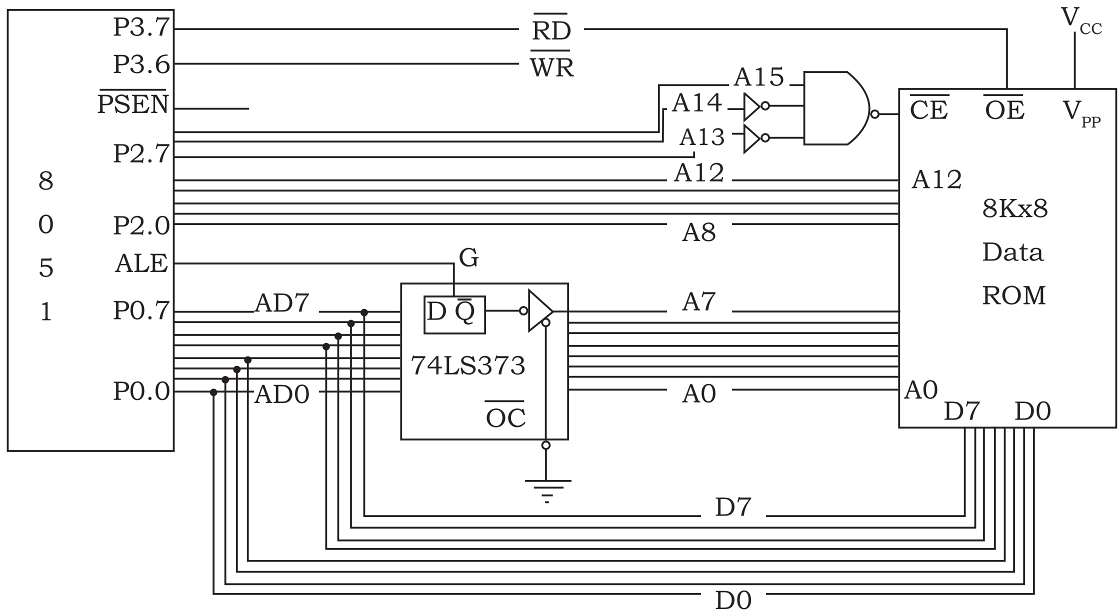


Fig.1.9.7 b: Interface 8K DATA ROM to 8051 Microcontroller

Interface 8K of single external ROM for both CODE and DATA.

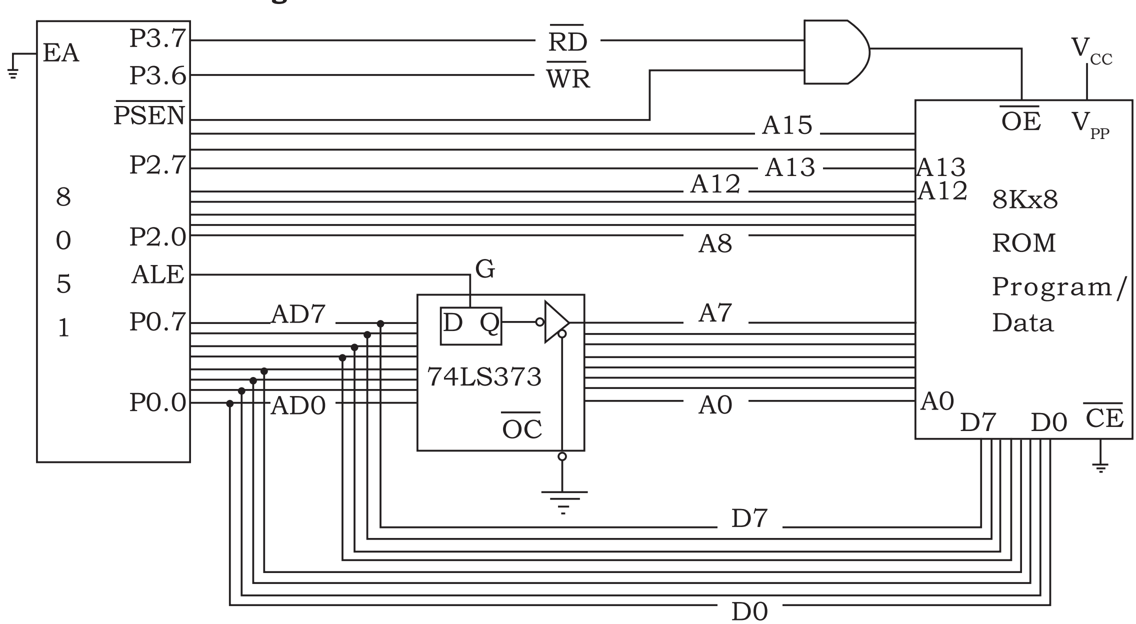


Fig.1.9.7 c: Interface 8K of single external ROM for both CODE and DATA

Example 1.12:

Interface 8K EPROM & 4K RAM to 8051 Microcontroller

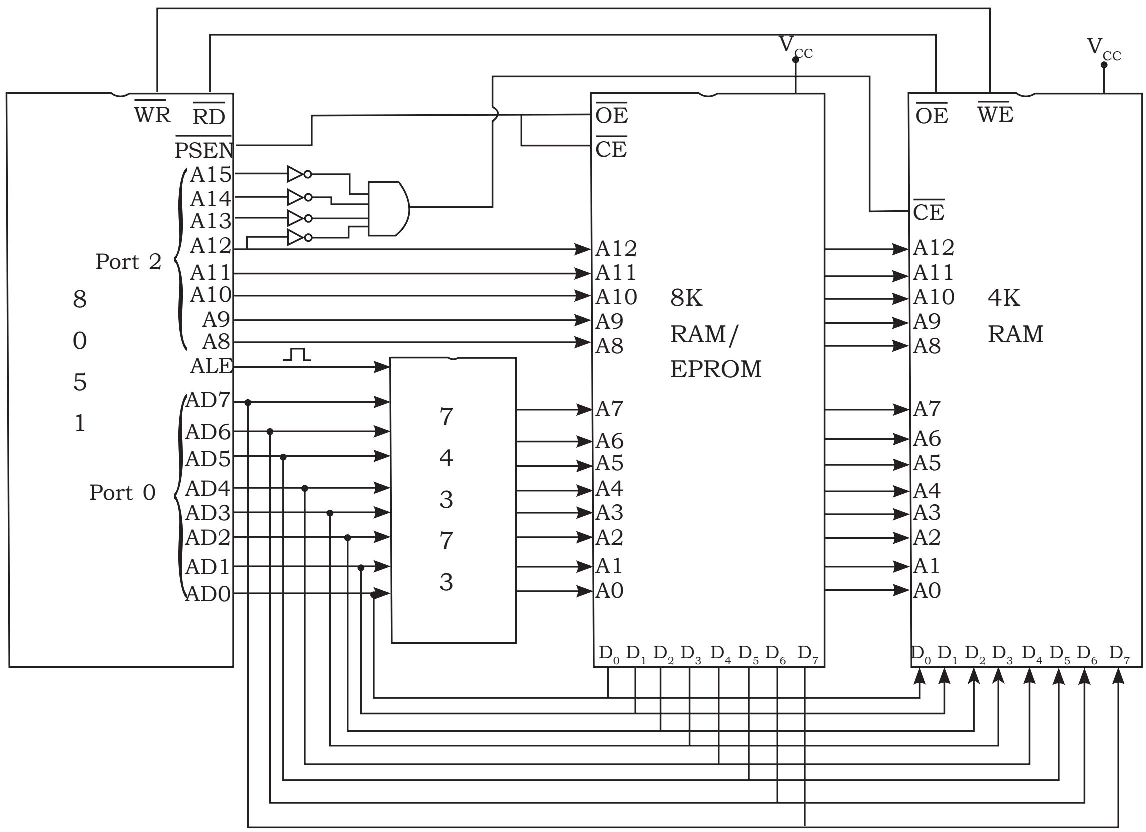


Fig.1.9.8: Interface 8K EPROM & 4K RAM to 8051 Microcontroller

Example 1.13:

Interface 16K EPROM & 8K RAM to 8051 Microcontroller.

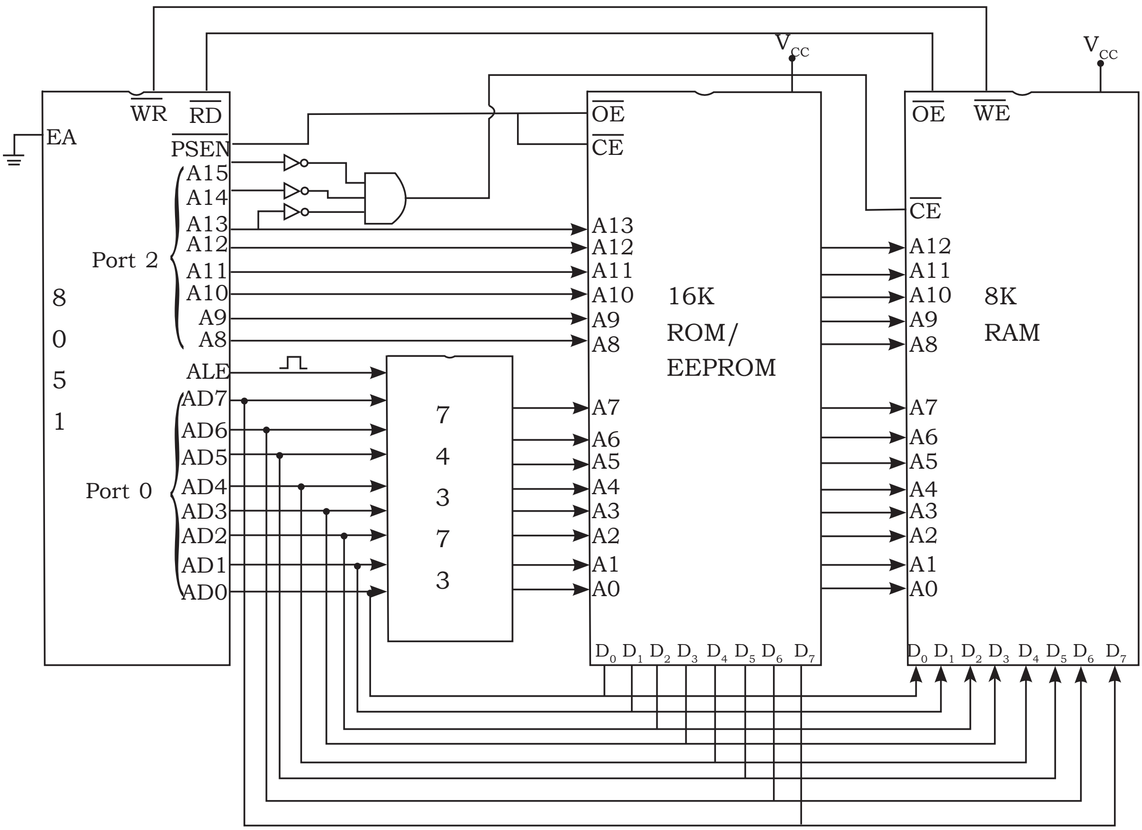


Fig.1.9.9: Interface 16K EPROM & 8K RAM to 8051 Microcontroller.

Example 1.14:

Interfacing 8 Kbyte RAM and 8 Kbyte ROM to 8051 microcontroller.

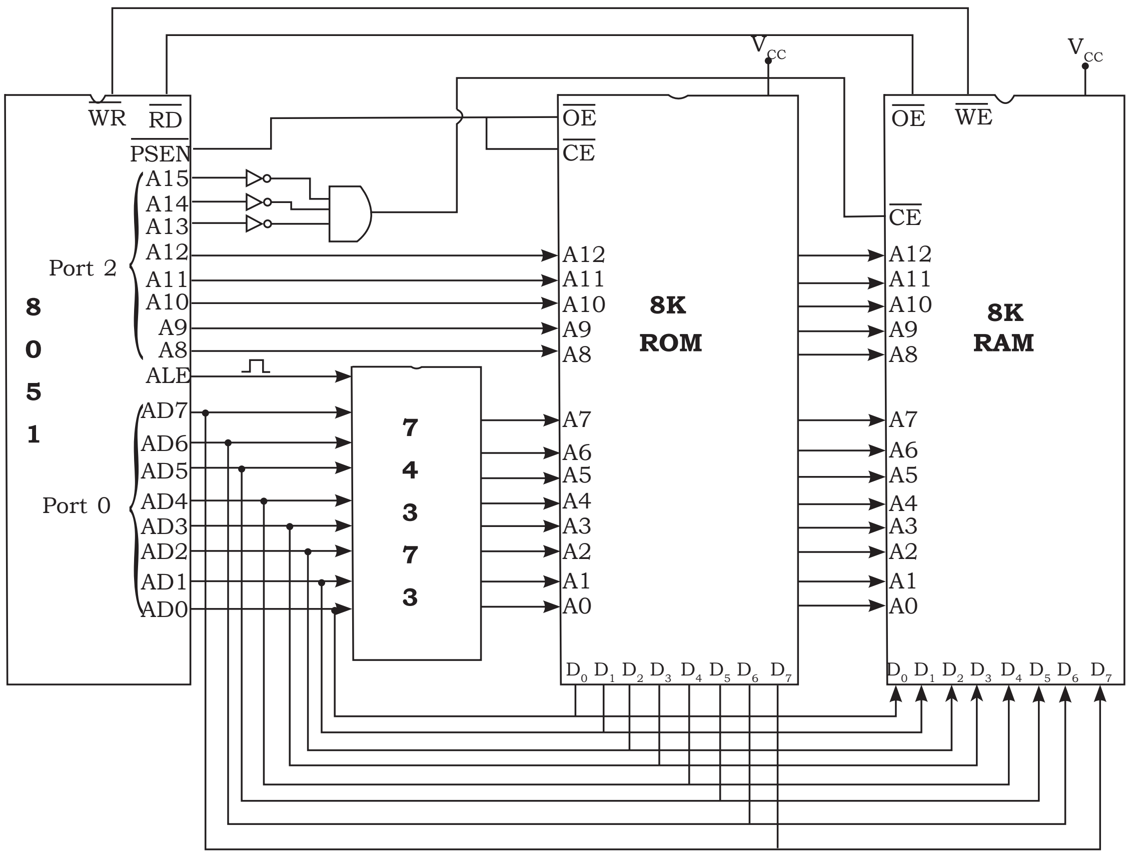


Fig.1.9.10: Interfacing 8 Kbyte RAM and ROM with 8051.

Example 1.15:

Interfacing 8 Kbyte RAM and 8 Kbyte ROM to 8051 microcontroller.

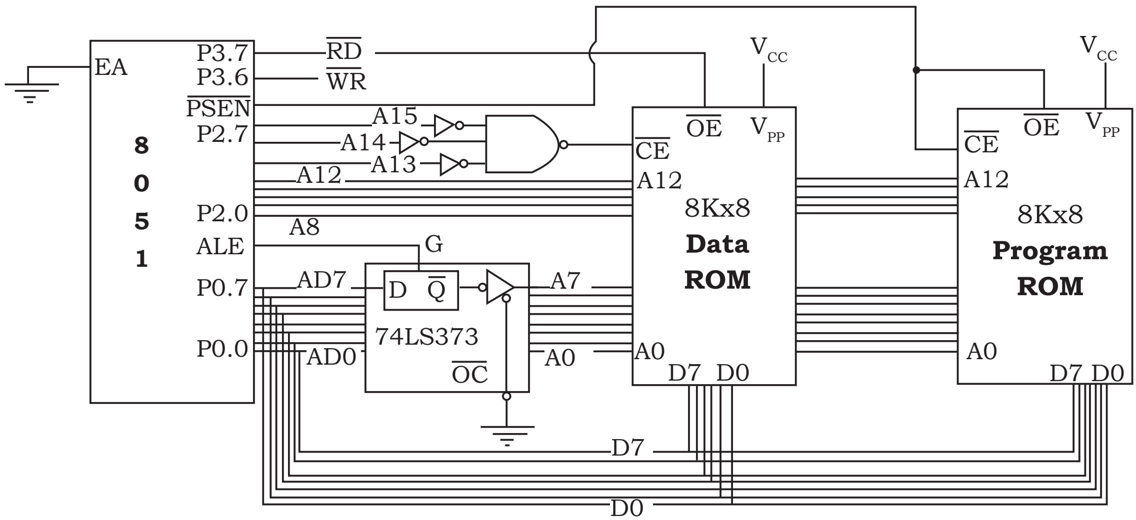


Fig.1.9.11: Interfacing 8 Kbyte RAM and ROM with 8051.

Example 1.16:

Interfacing 16 Kbyte DATA RAM, DATA ROM and PROM to 8051 microcontroller.

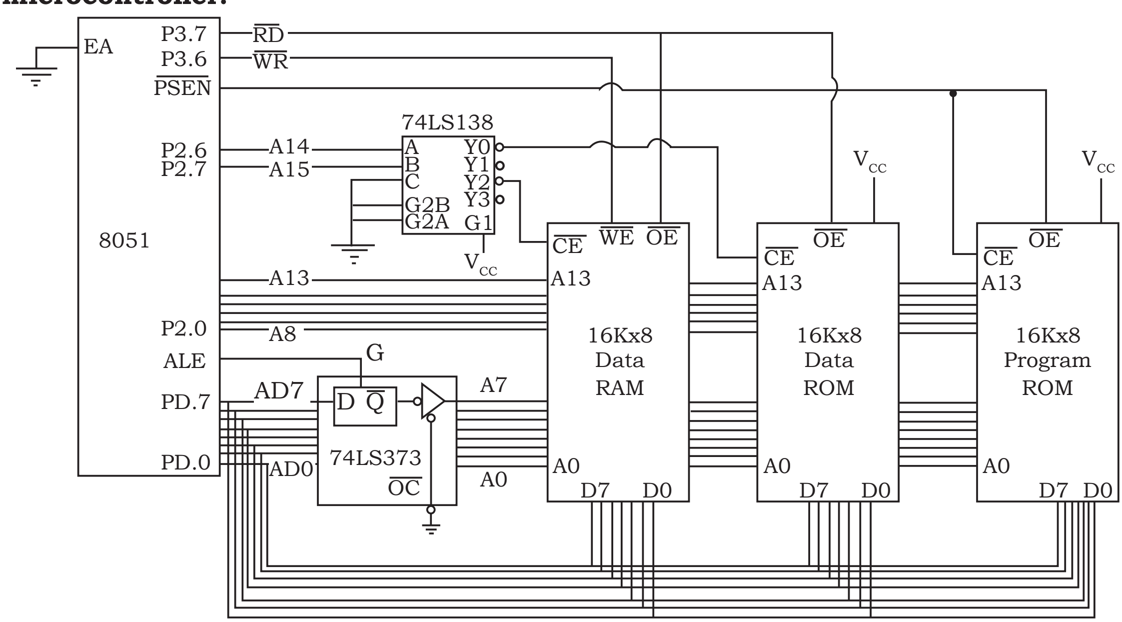


Fig.1.9.12: Interfacing 16 Kbyte DATA RAM, DATA ROM and PROM to 8051 microcontroller.

Example 1.17:

Identify to which Memory Location (ML) the data is moved after the execution of the following program segment

SETB RS1

CLR RS0

MOV R1,#25h

MOV R3,#65h ***5-Marks***

Solution.

- The figure below gives the details of the banks and their memory location (ML).

Hex	Bank 0	Hex	Bank 1	Hex	Bank 2	Hex	Bank 3
07	R7	0F	R7	17	R7	1F	R7
06	R6	0E	R6	16	R6	1E	R6
05	R5	0D	R5	15	R5	1D	R5
04	R4	0C	R4	14	R4	1C	R4
03	R3	0B	R3	13	R3	1B	R3
02	R2	0A	R2	12	R2	1A	R2
01	R1	09	R1	11	R1	19	R1
00	R0	08	R0	10	R0	18	R0

- After execution of above code, Bank 2 is selected and is shown in below table.

RS1	RS0	Bank Selected	Address
1	0	Bank 2	10H – 17H

- From above diagram it is clear that in Bank 2, R1 memory location is 11H and R2 memory location is 12H.

Bank 2 Selected	
Register	Memory location
R1	11H
R2	12H

2

8051 Instruction Set

2.1 INTRODUCTION

Computer languages are mainly classified into three types:

1. Low Level Languages 2. Middle Level Language 3. High Level Language

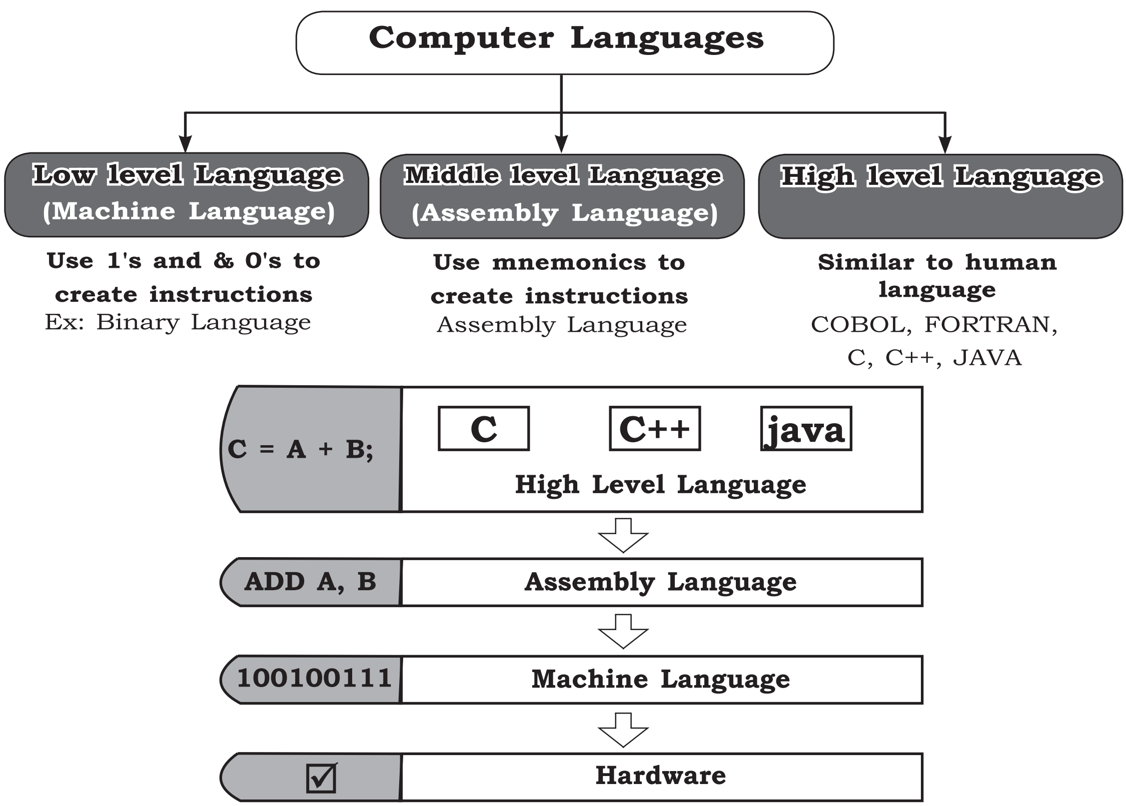


Fig. 2.1.1: Computer Languages

2.1.1 Low-Level Language (Machine language or Machine code)

Features of Low-Level Language

- A machine language is sometimes referred to as machine code or object code.
- It is a system of instructions and data executed directly by a computer CPU.
- It is a collection of binary digits or bits that the computer reads and interprets.
- It is the only language a computer is capable of understanding.
- It consists 0s and 1s.
- It is very difficult for human beings to understand this language.

Advantages:

1. It is fast and makes efficient use of computer
2. No translator required to translate the code i.e. directly understood by computer.

Disadvantages:

1. All operation code (OPCODE) has to be remembered
2. All memory addresses have to be remembered.
3. Programming is time consuming.
4. It is difficult to find errors in a program written in the machine language.

2.1.2 Middle-Level Languages (Assembly language)

- Medium-level language serves as the bridge between machine understandable machine level language and High Level language.
- Medium-level language is also known as intermediate programming language and pseudo language.
- Medium-level language is mainly an output of the programming source code written in a higher-level language.
- The source code of the medium-level language is not directly executable by the CPU as it's an intermediate step before being converted into machine code.
- In assembly language, one statement maps to one machine language instruction.
- **Applications:** To do system programming for writing operation as well as application programming.

Features of Assembly language (Middle-Level Languages)

- Assembly language programs need to be "assembled" for execution by the computer.

- Each assembly language instruction is translated into one machine language instruction
- Very efficient code
- Easy programming as compared to machine language
- Requires language translator (Assembler)
- Program execution is slow as compared to machine
- Detailed instruction set knowledge is required
- More error prone
- Difficulty in debugging program.

Advantages:

1. More standardize and easier than machine languages.
2. Operate efficiently.
3. Easy to debug programs.

Disadvantages:

1. Assembly language is defined for a specific machine and specific processor. Therefore, programs are not portable to other computers.
2. Source programs tend to be large and difficult to follow.
3. Many instructions are required to achieve small tasks.
4. Easy for the computer to understand but are more difficult for the programmer to write (Complex).
5. The program has to be translated into machine code in order to work.

2.1.3 High-Level Languages

- High level languages are computer languages that are closer to human language (ENGLISH) and are designed to be used by the human operator or the programmer.
- High-level language is a programming language in which a program is written in much simpler programming context and is generally independent of the computer's hardware architecture i.e. the programmer does not have to be concerned with all the registers of the CPU and the size of each registers etc.
- The source code is the code written by the programmer in the high level language. It must use interpreter, compiler or translator to convert human understandable program to computer readable code called machine code (objet code).
- In High Level language, one statement maps to one or more assembly language statement.
- **Examples:** BASIC, PASCAL, C, C++, Java etc.

Advantages:

1. Easier to read, write and maintain.
2. High-Level languages make complex programming simpler.
3. Error ratio is less in high level language and debugging is easier.
4. Length of the program is also small compared with low-level language.
5. Many real time problems can be easily solved with high level language.
6. Portable (can work across different CPU families and support a wide range of data types)

Disadvantages:

1. Usually slower than lower-level languages (for example assembler is fastest than C).

Differentiate between high level and low level language.

Sl. No.	High level language	Low level language
1	Easily understood by humans	Understood by computers
2	Uses English like words	Difficult for humans to read and understand
3	Easy to locate and identify errors	It's difficult to spot errors in the code
4	Must translate before the computer can understand it	No need of translator.

2.1.4 8051 Data Type

- The 8051 microcontroller has **only one data** type. It is 8-bits, and the size of each register is also 8 bits.
- The **programmer** has to **break down data** larger than **8 bits** (1 Byte) to be processed by the CPU.
- The data types used by the 8051 can be **positive** or **negative**.

2.1.5 Assembler Directives

- The assembler directives are the statements that direct the assembler what to do during assembling.
- They reserve memory space for data, define constants, and tell assembler where to assemble program in a memory.
- They are also referred as pseudo instructions statements as they are effective only during the assembly of the program but they do not generate any machine code.

The 8051 has 4 assembler directives:

1. ORG (origin) **2. DB** (Define data) **3. END** **4. EQU** (Equate)

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 hexadecimal.
- Some assemblers use "**.** ORG" (**dot** ORG) instead of "ORG" for the origin directive.

  Example: **ORG 1000H**

DB (Define data)

- It is used to **define** the **8-bit data** and most widely used data directive in the assembler.
- 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, "**B"**for binary and **"H"** for hexadecimal.
- To indicate **ASCII**, simply place the characters in **single quotes** or **double quotes**. 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.

  Examples:

```
ORG 1000H
DATA1:     DB    40          ;DECIMAL NUMBER
DATA2:     DB    00100111B   ; BINARY NUMBER
DATA3:     DB    1Fh         ;HEXADECIMAL NUMBER
DATA4:     DB    "DIPLOMA"   ;ASCII CHARACTER
DATA5:     DB    "2016"      ;ASCII NUMBER
END
```

EQU

- EQU 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.
- When the label appears in the program, its constant value will be substituted for the label.
- Assume that there is a constant used in many different places in the program, and the programmer wants to change its value throughout. By the use of EQU, one can change it once and the assembler will change all of its occurrences.

Example:

```
ORG 1000H
COUNT    EQU    40H
MOV A,#COUNT              ; COPY 40 INTO A
END
```

- EQU 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" ("dot END") instead of "END".

Example:

```
ORG 1000H
COUNT    EQU    40H
MOV A,#COUNT              ; COPY 40 INTO A
END
```

2.2 8051 ADDRESSING MODES

The **CPU** can **access data** in **various ways**. The data could be in a memory or in register or it may be an immediate value (CONSTANT). The **various ways** of **accessing** these **data** are called **addressing mode**.
There are 5 addressing modes in 8051

1. Immediate addressing mode
2. Register addressing mode
3. Direct addressing mode
4. Register indirect addressing mode
5. Indexed addressing mode.

Immediate addressing mode

- In Immediate addressing mode, the **source operand** is a **constant**. The immediate data must be preceded by the hash sign, **"#"**.
- This addressing mode can load information into any **registers**, including **16-bit DPTR** register and 8051 **ports**.

Examples:

```
MOV R1,#50          ;load 50 into R1
MOV B,#50H          ;load 50H into B
MOV A,#35H          ;load 35H into A
MOV DPL,#66H
```

```
MOV DPH,#55H          ;DPTR=5566H
MOV DPTR,#5566H       ;DPTR=5566H
                      ;This is the same as above
MOV DPTR,#69925       ;illegal Value because value is> 65535
                      ;(FFFFH)
MOV P1,#35H           ;load 35H into Port 1
```

Register addressing mode

- Register addressing mode uses registers to hold the data to be manipulated.
- The source and destination registers must match in size.
- The movement of data between R_n registers is not allowed.

  Examples:

```
MOV A,R1          ;copy contents of R1 into A
MOV R3,A          ;copy contents of A into R3
ADD A,R2          ;add contents of R2 to A
ADD A, R0         ;add contents of R0 to A
MOV R5,A          ;save accumulator in R5
MOV DPTR,#F5ABH   ;16-bit data F5ABH is moved to 16-bit register
MOV R7,DPL        ;Lower byte of DPTR copied to R7
MOV R6,DPH        ;Higher byte of DPTR copied to R7
MOV DPTR,A        ;will give an error because A=8 bit and DPTR=
                   16-bit
MOV R4,R7         ;is invalid
```

Direct addressing mode

- The entire 128 bytes of RAM can be accessed using direct addressing mode. The RAM locations 30-7FH are most often used.
- The register bank locations are accessed by its address or by its register names.
- In this instruction, address is given as a part of the instructions.

  Examples:

```
MOV R1, 30H       ;Save content of RAM location 30H in R1
MOV 50H, A        ;Save content of A in RAM location 50H
MOV A,4           ;is same as copying R4 into A
MOV A,R4          ;copy R4 into A
```

NOTE:

- The "#" sign distinguishes between the immediate and direct addressing mode. The **absence** of the **"#" sign** is the direct addressing mode.

Register indirect addressing mode

- In Register indirect addressing mode, a **register** is **used** to **hold** the **address** of the **data** (as a pointer to the data).
- Only register **R0** and **R1** are used for this purpose. The R2 – R7 cannot be used to hold the address of an operand located in RAM.
- When R0 and R1 hold the addresses of RAM locations, they must be preceded by the **"@" sign**.

  Examples:

MOV A, @R1	;move contents of RAM whose address is held by R1 into A
MOV @R0, B	;move contents of B into RAM whose address is held by R0
MOV @R1, 04H	;move contents of 04H into RAM whose address is held by R1
MOV 30H, @R1	;move contents of RAM whose address is held by R1 into RAM 30H
MOVC A,@A+DPTR	;the contents of A are added to the 16-bit register DPTR to form the 16-bit address of the needed data.

Advantages

- The advantage is that it makes accessing data dynamic rather than static as in direct addressing mode. Looping is not possible in direct addressing mode.

Limitations

- R0 and R1 are the only registers that can be used for pointers in register indirect addressing mode Since R0 and R1 are 8 bits wide, their use is limited to access any information in the internal RAM.
- The accessing of externally connected RAM or on-chip ROM need 16-bit pointer. In such case, the DPTR register is used.

NOTE:

- Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM.

Indexed addressing mode

- Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM space.
- Only program memory can be accessed in the index addressing. Either the DPTR or PC can be used as an index register.

  Examples:

```
MOVC A,@A+DPTR
MOVC A,@A+PC
```

2.3 STRUCTURE OF ASSEMBLY LANGUAGE

An assembly language instruction consists of a mnemonic, followed by one or two operands. An assembly language instruction consists of four fields:

```
[label:]    mnemonic    [operands]    [;comment]
```

Example:

```
Again: MOV A, R0 ; Copy the content of R0 into A register
```

NOTE:

- Brackets indicate that a field is optional, and not all lines have them. Brackets should not be typed in the instructions.

2.4 INSTRUCTION SET

- Based on the operations performed, the instruction set of 8051 are classified as
  1. Arithmetic instructions
  2. Logical instructions
  3. Data transfer instructions
  4 Boolean instructions
  5. Program branching instructions.
- Each instruction has two parts: **operation code** and **operands**. The operands may be one or more i.e. **operation code operand 1, operand 2 & operand 3.**

  Example:

  ```
  MOV A, B
  ```
- The following nomenclatures for register, data, address and variables are used while write instructions.

Table 2.1.1: Nomenclatures for register, data, address and variables are used while write instructions.

A	Accumulator
B	"**B**" register
C	**Carry bit**
Rn	Register **R0 - R7** of the currently selected register bank
Direct	**8-bit** internal direct address for data. The data could be in lower **128 bytes** of **RAM** (**00 - 7FH**) or it could be in the **special function register** (**80 - FFH**).
@Ri	**8-bit** external or internal **RAM address** available in register **R0** or **R1**. This is used for **indirect addressing** mode.

#data8	Immediate **8-bit data** available in the instruction.
#data16	Immediate **16-bit data** available in the instruction.
Addr11	**11-bit** destination address for short absolute jump. Used by instructions **AJMP** & **ACALL**. Jump range is **2 kbyte** (one page).
Addr16	**16-bit** destination address for **long call** or **long jump**.
Rel	2's complement 8-bit offset (one - byte) used for **short jump** (**SJMP**) and all conditional jumps.
bit	Directly addressed **bit** in internal **RAM** or **SFR**

Arithmetic Instructions

Syntax	Flags affected	Bytes	Cycles
ADD A, #data	**CY, OV & AC**	**2**	**1**
Operation	**(A) ⟵ (A)+ 8-bit data**		
Description	Adds the **8-bit immediate** data with **accumulator** contents and result is stored in accumulator.		
Example	**ADD A, #04H**		
Before Execution	**After Execution**		
A = 05H	A = 09H		

Syntax	Flags affected	Bytes	Cycles
ADD A, Rn	**CY, OV & AC**	**1**	**1**
Operation	**(A) ⟵ (A) + (Rn)** where **n** = 0,1,2,3,4,5,6,7 i.e., **R0 to R7**		
Description	Adds the contents of **register Rn** with **accumulator** contents and result is stored in accumulator.		
Example	**ADD A, R1**		
Before Execution	**After Execution**		
A = 05H & R1 = 04H	A = 09H		

Syntax	Flags affected	Bytes	Cycles
ADD A, direct	**CY, OV & AC**	**2**	**1**
Operation	**(A) ⟵ (A) + (direct address)** where **direct** is the **address.**		
Description	Adds the content of **direct address** with **accumulator** contents and result is stored in accumulator.		
Example	**ADD A, 50H**		
Before Execution	**After Execution**		
A = 05H & 50H = 04H	A = 09H		

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">ADD A, @Ri</td><td>CY, OV & AC</td><td>1</td><td>1</td></tr>
<tr><td>Operation</td><td colspan="4">(A) ⟵ (A) + ((Ri)) where i = 0 & 1 i.e. R0 & R1.</td></tr>
<tr><td>Description</td><td colspan="4">Adds the contents of indirect RAM address with accumulator contents and result is stored in accumulator.</td></tr>
<tr><td>Example</td><td colspan="4">ADD A, @R0</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">A = 05H, R0 = 50H & 50H = 04H</td><td colspan="3">A = 09H i.e. A = (A) + (50H)</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">ADDC A, #data</td><td>CY, OV & AC</td><td>2</td><td>1</td></tr>
<tr><td>Operation</td><td colspan="4">(A) ⟵ (A) + (C) + 8-bit data</td></tr>
<tr><td>Description</td><td colspan="4">Adds the 8-bit immediate data, the carry flag and the accumulator contents, result is stored in accumulator.</td></tr>
<tr><td>Example</td><td colspan="4">ADDC A, #04H</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">A = 05H, CY=0
A = 05H, CY=1</td><td colspan="3">A = 09H
A = 0AH</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">ADDC A, Rn</td><td>CY, OV & AC</td><td>1</td><td>1</td></tr>
<tr><td>Operation</td><td colspan="4">(A) ⟵ (A) + (C) + (Rn) where n = 0,1,2,3,4,5,6,7 i.e., R0 to R7</td></tr>
<tr><td>Description</td><td colspan="4">Adds the contents of register Rn, the carry flag and the accumulator contents, result is stored in accumulator.</td></tr>
<tr><td>Example</td><td colspan="4">ADDC A, R0</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">A = 05H, CY=0 & R0 = 04H
A = 05H, CY=1 & R0 = 04H</td><td colspan="3">A = 09H
A = 0AH</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">ADDC A, direct</td><td>CY, OV & AC</td><td>2</td><td>1</td></tr>
<tr><td>Operation</td><td colspan="4">(A) ⟵ (A) + (C) + (direct address)</td></tr>
<tr><td>Description</td><td colspan="4">Adds the contents of direct address, the carry flag and the accumulator contents, result is stored in accumulator.</td></tr>
<tr><td>Example</td><td colspan="4">ADDC A, 50H</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">A = 05H, CY=0 & 50H = 04H
A = 05H, CY=1 & 50H = 04H</td><td colspan="3">A = 09H
A = 0AH</td></tr>
</table>

Syntax		Flags affected	Bytes	Cycles
ADDC A, @Ri		**CY, OV & AC**	**1**	**1**
Operation	**(A) ⟵ (A) + (C) + ((Ri))** where **i = 0 & 1 i.e. R0 & R1.**			
Description	Adds the contents of **direct address, the carry flag** and the **accumulator** contents, result is stored in accumulator.			
Example	**ADDC A, @R1**			
Before Execution		**After Execution**		
A = 05H, CY=0, R1=50H & 50H = 04H		A = 09H		
A = 05H, CY=1, R1=50H & 50H = 04H		A = 0AH		

Syntax		Flags affected	Bytes	Cycles
SUBB A,#data		**CY, OV & AC**	**2**	**1**
Operation	**(A) ⟵ (A) - #-bit data**			
Description	Subtract the **8-bit immediate** data with **accumulator** contents and result is stored in accumulator.			
Example	**SUBB A, #04H**			
Before Execution		**After Execution**		
A = 05H		A = 01H		

Syntax		Flags affected	Bytes	Cycles
SUBB A, Rn		**CY, OV & AC**	**1**	**1**
Operation	**(A) ⟵ (A) - (Rn)** where **n** = 0,1,2,3,4,5,6,7 i.e., **R0 to R7**			
Description	Subtract the contents of **register Rn** with **accumulator** contents and result is stored in accumulator.			
Example	**SUBB A, R0**			
Before Execution		**After Execution**		
A = 05H, R0 = 04H		A = 01H		

Syntax		Flags affected	Bytes	Cycles
SUBB A, direct		**CY, OV & AC**	**2**	**1**
Operation	**(A) ⟵ (A) - (direct address)**			
Description	Subtract the contents of **direct address** with **accumulator** contents and result is stored in accumulator.			
Example	**SUBB A, 50H**			
Before Execution		**After Execution**		
A = 05H, 50H = 04H		A = 01H		

Syntax		Flags affected	Bytes	Cycles
SUBB A, @Ri		**CY, OV & AC**	**1**	**1**
Operation	**(A) ⟵ (A) - ((Ri))**			
Description	Subtract the contents of **indirect RAM** address with **accumulator** contents and result is stored in accumulator.			
Example	**SUBB A, @R1**			
Before Execution		**After Execution**		
A = 05H, R1 = 50H & 50H = 04H		A = 01H		

Syntax		Flags affected	Bytes	Cycles
INC A		**NONE**	**1**	**1**
Operation	**(A) ⟵ (A) + 1**			
Description	Increment the content of accumulator by 1.			
Example	**INC A**			
Before Execution		**After Execution**		
A = 05H		A = 06H		

Syntax		Flags affected	Bytes	Cycles
INC Rn		**NONE**	**1**	**1**
Operation	**(Rn) ⟵ (Rn) + 1**			
Description	Increment the content of register Rn by 1.			
Example	**INC R0**			
Before Execution		**After Execution**		
R0 = 05H		R0 = 06H		

Syntax		Flags affected	Bytes	Cycles
INC direct		**NONE**	**2**	**1**
Operation	**(direct) ⟵ (Direct) + 1**			
Description	Increment the content of direct address by 1.			
Example	**INC 50H**			
Before Execution		**After Execution**		
50H = 05H		50H = 06H		

Syntax		Flags affected	Bytes	Cycles
INC @Ri		**NONE**	**1**	**1**
Operation	**((Ri))⟵((Ri)) + 1**			
Description	Increment the content of indirect RAM address by 1.			
Example	**INC @R0**			
Before Execution		**After Execution**		
R0=50H, 50H = 05H		50H = 06H		

Syntax		Flags affected	Bytes	Cycles
DEC A		**NONE**	**1**	**1**
Operation	**(A) ⟵ (A) - 1**			
Description	Decrement the content of Accumulator by 1			
Example	**DEC A**			
Before Execution		**After Execution**		
A = 05H		A = 04H		

Syntax		Flags affected	Bytes	Cycles
DEC Rn		**NONE**	**1**	**1**
Operation	**(Rn) ⟵ (Rn) - 1**			
Description	Decrement the content of register Rn by 1			
Example	**DEC R0**			
Before Execution		**After Execution**		
R0 = 05H		**R0 = 04H**		

Syntax		Flags affected	Bytes	Cycles
DEC direct		**NONE**	**2**	**1**
Operation	**(direct) ⟵ (Direct) - 1**			
Description	Decrement the content of direct address by 1			
Example	**DEC 50H**			
Before Execution		**After Execution**		
50H = 05H		50H = 04H		

Syntax		Flags affected	Bytes	Cycles
DEC @Ri		**NONE**	**1**	**1**
Operation	**((Ri)) ⟵ (((Ri)) - 1**			
Description	Decrement the content of indirect address by 1			
Example	**DEC @R0**			
Before Execution		**After Execution**		
R0=50H & 50H = 05H		50H = 04H		

Syntax		Flags affected	Bytes	Cycles
INC DPTR		**NONE**	**1**	**2**
Operation	**(DPTR) ⟵ (DPTR) + 1**			
Description	Increment the content of 16-bit register DPTR address by 1			
Example	**INC DPTR**			
Before Execution		**After Execution**		
DPTR = 1255H i.e. **DH =12H, DL = 55H**		**DPTR = 1256H** i.e. **DH =12H, DL = 56H**		

Syntax	Flags affected	Bytes	Cycles
MUL AB	**CY & OV**	**1**	**4**
Operation	**$(B)_{15-8}$ $(A)_{7-0} \longleftarrow$ A X B**		
Description	Multiply the Contents of Accumulator with the contents of register B. **Lower byte** of the result in **Accumulator** and **higher byte** of the result in **Register B.**		
Example	MUL AB		
Before Execution	**After Execution**		
A = 05H & B = 03H **A = FFH & B = 02H**	**0015H i.e. A = 15H & B = 00H** **01FEH i.e. A = FEH & B = 01H**		

Syntax	Flags affected	Bytes	Cycles
DIV AB	**CY**	**1**	**4**
Operation	**Quotient in A & Remainder in B $\longleftarrow$ A/B**		
Description	Divide the Contents of Accumulator with the contents of register B. **Quotient** of the result is stored in **Accumulator** & **remainder** of theresult is stored in **Register B.**		
Example	DIV AB		
Before Execution	**After Execution**		
A = FBH (251d) & B = 12H (18d)	A = 0DH & B = 11H Quotient = 0DH & Remainder = 11H		

Syntax	Flags affected	Bytes	Cycles
DAA	**CY & AC**	**1**	**1**
Operation	**Decimal Adjustment Accumulator.** If $(A)_{3-0} > 9$ or (AC) = 1 Then Add +6 to $(A)_{3-0}$ If $(A)_{7-4} > 9$ or (CY) = 1 Then Add +6 to $(A)_{7-4}$		
Description	Decimal adjustment of the accumulator according to BCD code. The data is adjusted in the following two possible cases It adds 6 to the lower 4-bits of A if it is greater than 9 or if AC=1. It adds 6 to the upper 4-bits of A if it is greater than 9 or if CY=1.		

Example	MOV A, #45H
	ADD A, #15H
	DAA
	
	Result:
	4 5 H
	+ 1 5 H
	———
	5 A H **INVALID BCD**
	+ **6** After **DAA**
	6 0 H **Valid BCD**
		MOV A, #50H
		ADD A, #45H
		DAA
		
		Result:
		5 0 H
		+ 5 5 H
		———
		A 5 H **INVALID BCD**
		+ **6** After **DAA**
		1 0 5 H Valid BCD

Mnemonics	Byte	Cycle	Operation	Flags Affected
ADD A, #data	2	1	(A) ⟵ (A) + 8-bit data	CY, OV & AC
ADD A, Rn	1	1	(A) ⟵ (A) + (Rn)	CY, OV & AC
ADD A, direct	2	1	(A) ⟵ (A) + (direct address)	CY, OV & AC
ADD A, @Ri	1	1	(A) ⟵ (A) + ((Ri))	CY, OV & AC
ADDC A, #data	2	1	(B) ⟵ (A) + (C) + 8-bit data	CY, OV & AC
ADDC A, Rn	1	1	(B) ⟵ (A) + (C) + (Rn)	CY, OV & AC
ADDC A, direct	2	1	(A) ⟵ (A) + (C) + (direct address)	CY, OV & AC

ARITHMETIC INSTRUCTIONS

ADDC A, @Ri	1	1	(A) ⟵ (A) + (C) + ((Ri))	CY, OV & AC
SUBB A,#data	2	1	(A) ⟵ (A) - #-bit data	CY, OV & AC
SUBB A, Rn	1	1	(A) ⟵ (A) - (Rn)	CY, OV & AC
SUBB A, direct	2	1	(A) ⟵ (A) - (direct address)	CY, OV & AC
SUBB A, @Ri	1	1	(A) ⟵ (A) - ((Ri))	NONE
INC A	1	1	(A) ⟵ (A) + 1	NONE
INC Rn	1	1	(Rn) ⟵ (Rn) + 1	NONE
INC direct	2	1	(direct) ⟵ (direct) + 1	NONE
INC @Ri	1	1	((Ri)) ⟵ ((Ri)) + 1	NONE
DEC A	1	1	(A) ⟵ (A) - 1	NONE

DEC Rn	**1**	**1**	**(Rn) ← (Rn) - 1**	**NONE**
DEC direct	**2**	**1**	**(direct) ← (direct) - 1**	**NONE**
DEC @Ri	**1**	**1**	**((Ri)) ← ((R1)) - 1**	**NONE**
INC DPTR	**1**	**2**	**(DPTR) ← (DPRT) + 1**	**NONE**
MUL AB	**1**	**4**	**$(B)_{15-8}$ $(A)_{7-8}$ ← AxB**	**CY & OV**
DIV AB	**1**	**4**	**Quotient in A & Remainder in B ← (A/B)**	**CY**
DA A	**1**	**1**	**Decimal Adjustment Accumulator.** **If $(A)_{3-0}$ > 9 or (AC) = 1** **Then** **Add +6 to $(A)_{3-0}$** **If $(A)_{7-4}$ > 9 or (CY) = 1** **Then** **Add +6 to $(A)_{7-4}$**	**CY & AC**

Data Transfer Instructions

Syntax		Flags affected	Bytes	Cycles
MOV A, #data		**NONE**	**2**	**1**
Operation	**(A) ← 8-bit data**			
Description	Moves the **8-bit immediate** data to the Accumulator.			
Example	**MOV A, #04H**			
Before Execution		**After Execution**		
A = XX		A = 04H		

Syntax		Flags affected	Bytes	Cycles
MOV A, Rn		**NONE**	**1**	**1**
Operation	**(A) ← (Rn)** where **n** = 0,1,2,3,4,5,6,7 i.e., **R0 to R7**			
Description	Moves the contents of register Rn to the Accumulator.			
Example	**MOV A, R1**			
Before Execution		**After Execution**		
A = XX & R1 = 05H		A = 05H		

Syntax	Flags affected	Bytes	Cycles
MOV A, direct	**NONE**	**2**	**1**
Operation	**(A) ⟵ (direct)** where **direct** is the **address**		
Description	Moves the contents of **direct address** to the **Accumulator**.		
Example	**MOV A, 50H**		
Before Execution		**After Execution**	
A = XX & 50H = 04H		A = 04H	

Syntax	Flags affected	Bytes	Cycles
MOV A, @Ri	**NONE**	**1**	**1**
Operation	**(A) ⟵ ((Ri)) where i = 0 & 1 i.e. R0 & R1.**		
Description	Moves the contents of **indirect RAM** address to the Accumulator.		
Example	**MOV A, @R0**		
Before Execution		**After Execution**	
A = XX, R0 = 50H & 50H = 04H		A = 04H	

Syntax	Flags affected	Bytes	Cycles
MOV Rn, A	**NONE**	**1**	**1**
Operation	**(Rn) ⟵ (A)** where **n** = 0,1,2,3,4,5,6,7 i.e., **R0 to R7**		
Description	Moves the contents of Accumulator to the register Rn.		
Example	**MOV R1, A**		
Before Execution		**After Execution**	
R1 = XX & A = 04H		**R1 = 04H**	

Syntax	Flags affected	Bytes	Cycles
MOV Rn, #data	**NONE**	**2**	**1**
Operation	**(Rn) ⟵ 8-bit data**		
Description	Moves the **8-bit immediate** data to the register **Rn**.		
Example	**MOV R1, #04H**		
Before Execution		**After Execution**	
R1 = XX		R1 = 04H	

Syntax	Flags affected	Bytes	Cycles
MOV Rn, direct	**NONE**	**2**	**2**
Operation	**(Rn) ⟵ (Direct)** where **direct** is the **address**		
Description	Moves the contents of **direct address** to the register **Rn**.		
Example	**MOV R1, 50H**		
Before Execution		**After Execution**	
R1 = XX & 50H = 04H		A = 04H	

Syntax		Flags affected	Bytes	Cycles
MOV direct, A		**NONE**	**2**	**1**
Operation	**(direct)** $\longleftarrow$ **(A)** where **direct** is the **address**			
Description	Moves the contents of **Accumulator** to **direct** address.			
Example	**MOV 50H, A**			
Before Execution		**After Execution**		
50H = XX & A = 04H		50H = 04H		

Syntax		Flags affected	Bytes	Cycles
MOV direct, Rn		**NONE**	**2**	**2**
Operation	**(direct)** $\longleftarrow$ **(Rn)** where **n** = 0,1,2,3,4,5,6,7i.e. **R0 to R7**			
Description	Moves the contents of register **Rn** to **Accumulator**.			
Example	**MOV 50H, R1**			
Before Execution		**After Execution**		
R1 = 04H & 50H = XX		50H = 04H		

Syntax		Flags affected	Bytes	Cycles
MOV direct, direct		**NONE**	**3**	**2**
Operation	**(direct)** $\longleftarrow$**(direct)** where **direct** is the **address**			
Description	Moves the contents of direct address to direct address.			
Example	**MOV 50H, 30H**			
Before Execution		**After Execution**		
50H = XX & 30H = 04H		50H = 04H		

Syntax		Flags affected	Bytes	Cycles
MOV @Ri, A		**NONE**	**1**	**1**
Operation	**((Ri))** $\longleftarrow$ **(A) where i = 0 & 1 i.e. R0 & R1.**			
Description	Moves the contents of **Accumulator** to **indirect RAM** address.			
Example	**MOV @R1, 50H**			
Before Execution		**After Execution**		
R1 =30H, 30H = XX & 50H = 04H		30H = 04H		

Syntax	Flags affected	Bytes	Cycles
MOV @Ri, #data	**NONE**	**2**	**1**
Operation	**((Ri)) ← 8 bit data where i = 0 & 1 i.e. R0 & R1.**		
Description	Moves the 8-bit data to **indirect RAM** address.		
Example	**MOV @R1, #04H**		
Before Execution	**After Execution**		
R1 =30H & 30H = XX	30H = 04H		

Syntax	Flags affected	Bytes	Cycles
MOV @Ri, direct	**NONE**	**2**	**2**
Operation	**((Ri)) ← (direct) where i = 0 & 1 i.e. R0 & R1.**		
Description	Moves the **contents of direct address t**o the register **indirect RAM address.**		
Example	**MOV @R1, 50H**		
Before Execution	**After Execution**		
R1 =30H, 30H = XX & 50H = 04H	**30H = 04H**		

Syntax	Flags affected	Bytes	Cycles
MOV DPTR, #data	**NONE**	**3**	**2**
Operation	**(DPTR) ←16 bit data** where **DPTR = DPH + DPL**		
Description	Moves the 16-bit data tothe **data pointer register**.		
Example	**MOV DPTR, #1234H**		
Before Execution	After Execution		
DPTR = XX	DPTR = 1234H i.e. DPH = 12H & DPL = 34H		

Syntax	Flags affected	Bytes	Cycles
MOVC A,@A+DPTR	**NONE**	**1**	**2**
Operation	**(A) ← (A + DPTR)**		
Description	The **MOVC** instruction moves a byte from the code or program memory to the accumulator. The Code Memory address from which the byte will be moved is calculated by summing the value of the Accumulator with either DPTR.		
Example	**MOVC A,@A+DPTR**		

Syntax		Flags affected	Bytes	Cycles
MOVC A,@A+PC		**NONE**	**1**	**2**
Operation	**(PC) ⟵ (PC + 1)** (A) ⟵ (A+PC)			
Description	The **MOVC** instruction moves a byte from the code or program memory to the accumulator. The Code Memory address from which the byte will be moved is calculated by summing the value of the Accumulator with the Program Counter (PC). The Program Counter (PC) is first incremented by 1 before being summed with the Accumulator.			
Example	**MOVC A,@A+PC**			

Syntax		Flags affected	Bytes	Cycles
MOVX A,@Ri		**NONE**	**1**	**2**
Operation	**(A) ⟵ ((Ri)) where i = 0 & 1 i.e. R0 & R1**			
Description	Moves the **indirect** external RAM (8-bit address) to the accumulator			
Example	**MOVX A,@R1**			
Before Execution		After Execution		
A = XX , R1 =30H & 30H = FFH		A = FFH		

Syntax		Flags affected	Bytes	Cycles
MOVX A,@DPTR		**NONE**	**1**	**2**
Operation	**(A) ⟵ ((DPTR))**			
Description	Moves the external RAM (16-bit address) to the accumulator			
Example	**MOVX A,@DPTR**			
Before Execution		**After Execution**		
A = XX, DPTR = 1234H & 1234H = **FFH**		**A = FFH**		

Syntax		Flags affected	Bytes	Cycles
MOVX @Ri,A		**NONE**	**1**	**2**
Operation	**((Ri)) ⟵ (A) where i = 0 & 1 i.e. R0 & R1**			
Description	Moves the contents of accumulator to the **indirect** external RAM (8-bit address)			
Example	**MOVX @R0,A**			
Before Execution		After Execution		
R0 =30H, 30H = XX & A = FFH		30H = FFH		

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">MOVX @DPTR,A</td><td>NONE</td><td>1</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4"></td></tr>
<tr><td>Description</td><td colspan="4">Moves the contents of Accumulator to the indirect external RAM (16-bit address)</td></tr>
<tr><td>Example</td><td colspan="4">MOVX @DPTR,A</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">DPTR = 1234H, 1234H = XX & A = FFH</td><td colspan="3">1234H = FFH</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">PUSH direct</td><td>NONE</td><td>2</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(SP) ←—— (SP) + 1
(SP) ←—— (direct)</td></tr>
<tr><td>Description</td><td colspan="4">Push onto Stack
The stack pointer is incremented by one. The content of the indicated variable is then copied into the internal RAM location address by the stack pointer.
The PUSH instruction supports only direct addressing mode. Therefore, PUSH A, PUSH R0 etc. are invalid instructions.</td></tr>
<tr><td>Example</td><td colspan="4">1. PUSH 0E0H ; 0E0H is the RAM address of Accumulator.
2. PUSH 00H ; 00H is the RAM address of R0 of Bank 0.</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">POP direct</td><td>NONE</td><td>2</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(direct) ←—— (SP)
(SP) ←—— (SP) - 1</td></tr>
<tr><td>Description</td><td colspan="4">POP from the stack.
The POP instruction copies the byte pointed to by Stack pointer (SP) to the location where direct address is indicated and decremented SP by 1.
The POP instruction supports only direct addressing mode. Therefore, Therefore, POP A, POP R0 etc. are invalid instructions.</td></tr>
<tr><td>Example</td><td colspan="4">POP 0E0H ; 0E0H is the RAM address of Accumulator.
POP 00H ; 00H is the RAM address of R0 of Bank 0.</td></tr>
</table>

Syntax		Flags affected	Bytes	Cycles
XCH A, Rn		**NONE**	**1**	**1**
Operation	**(A)** $\longleftrightarrow$ **(Rn)** where **n** = 0,1,2,3,4,5,6,7 i.e., **R0 to R7**			
Description	Exchanges the contents of register Rn with the contents of Accumulator.			
Example	**XCH A, R1**			
Before Execution		**After Execution**		
A = FFH & R1 = BBH		**A = BBH & R1 = FFH**		

Syntax		Flags affected	Bytes	Cycles
XCH A, direct		**NONE**	**2**	**1**
Operation	**(A)** $\longleftrightarrow$ **(direct)**			
Description	Exchanges the contents of **direct address** with **Accumulator** contents.			
Example	XCH A, 50H			
Before Execution		**After Execution**		
A = FFH & 50H = BBH		**A = BBH & 50H = FFH**		

Syntax		Flags affected	Bytes	Cycles
XCH A, @Ri		**NONE**	**1**	**1**
Operation	(A) $\longleftrightarrow$ ((Ri)) where i = 0 & 1 i.e. R0 & R1			
Description	Exchanges the contents of **indirect RAM** address with **accumulator** contents.			
Example	**XCH A, @R0**			
Before Execution		**After Execution**		
A = FFH, R0 = 50H & 50H = BBH		**A = BBH, R0 = 50H & 50H = FFH**		

Syntax		Flags affected	Bytes	Cycles
XCHD A,@Ri		**NONE**	**1**	**1**
Operation	$(A)_{3-0} \longleftrightarrow ((Ri))_{3-0}$ where i = 0 & 1 i.e. R0 & R1			
Description	Exchanges the low-order nibble indirect RAM with the accumulator contents.			
Example	**XCHD A,@R0**			
Before Execution		**After Execution**		
A = FFH, R0 = 50H & 50H = BBH		**A = FBH, R0 = 50H & 50H = FBH**		

Table: 2.4.2: Data Transfer Instructions

Mnemonics	Byte	Cycle	Operation	Flags Affected
MOV A, #data	2	1	(A) ⟵ 8 bit data	NONE
MOV A, Rn	1	1	(A) ⟵ (Rn)	NONE
MOV A, direct	2	1	(A) ⟵ (Direct)	NONE
MOV A, @Ri	1	1	(A) ⟵ ((Ri))	NONE
MOV Rn, A	1	1	(Rn) ⟵ (A)	NONE
MOV Rn, #data	2	1	(Rn) ⟵ 8 bit data	NONE
MOV Rn, direct	2	2	(Rn) ⟵ (Direct)	NONE
MOV direct, A	2	1	(direct) ⟵ (A)	NONE
MOV direct, Rn	2	2	(direct) ⟵ (Rn)	NONE
MOV direct, direct	3	2	(direct) ⟵ (Direct)	NONE
MOV @Ri, A	1	1	((Ri)) ⟵ (A)	NONE
MOV @Ri, #data	2	1	((Ri)) ⟵ 8 bit data	NONE
MOV @Ri, direct	2	2	((Ri)) ⟵ (Direct)	NONE
MOV DPTR, #data	3	2	(DPTR) ⟵ 16 bit data	NONE
MOVC A,@A+DPTR	1	2	(A) ⟵ (A + DPTR)	NONE
MOVC A,@A+PC	1	2	(PC) ⟵ (PC + 1) (B) ⟵ (A+PC)	NONE
MOVX A,@Ri	1	2	(A) ⟵ ((Ri))	NONE
MOVX A,@DPTR	1	2	(A) ⟵ ((DPTR))	NONE
MOVX @Ri,A	1	2	((Ri))⟵ (A)	NONE
MOVX @DPTR,A	1	2	((DPTR)) ⟵ (A)	NONE
PUSH direct	2	2	(SP) ⟵ (SP) + 1 (SP) ⟵ (direct)	NONE
POP direct	2	2	(direct) ⟵ (SP) (SP) ⟵ (SP) - 1	NONE
XCH A, Rn	1	1	(B) ⟷ (Rn)	NONE
XCH A, direct	2	1	(B) ⟷ (direct)	NONE
XCH A, @Ri	1	1	(A) ⟷ ((Ri))	NONE
XCHD A,@Ri	1	1	$(A)_{3-0} \longleftrightarrow ((Ri))_{3-0}$	NONE

Data Transfer Instructions

Logial instructions

Syntax	Flags affected	Bytes	Cycles
ANL A,#data	**NONE**	**2**	**1**
Operation	**(A) ⟵ (A) AND 8-bit data**		
Description	AND the content of **8-bit immediate data** with the content of accumulator and result is stored in accumulator.		
Example	**ANL A,#0FH**		
Before Execution	**After Execution**		
A = FFH	**A = 0FH**		

Syntax	Flags affected	Bytes	Cycles
ANL A,Rn	**NONE**	**1**	**1**
Operation	**(A) ⟵ (A) AND (Rn)** where **n** = 0,1,2,3,4,5,6,7 i.e., **R0 to R7**		
Description	AND the content of register Rn with the content of accumulator and result is stored in accumulator.		
Example	**ANL A,R1**		
Before Execution	**After Execution**		
A = FFH & R1 = 0FH	**A = 0FH**		

Syntax	Flags affected	Bytes	Cycles
ANL A,direct	**NONE**	**2**	**1**
Operation	**(A) ⟵ (A) AND (direct address)** where **direct** is the **address**		
Description	AND the content of **direct address** with the content of accumulator and result is stored in accumulator.		
Example	**ANL A,50H**		
Before Execution	**After Execution**		
A = FFH & 50H = 0FH	**A = 0FH**		

Syntax	Flags affected	Bytes	Cycles
ANL A,@Ri	**NONE**	**1**	**1**
Operation	**(A) ⟵ (A) AND (Ri))** where **n** = 0,1,2,3,4,5,6,7i.e. **R0 to R7**		
Description	AND the content of **indirect address** with the content of accumulator and result is stored in accumulator.		
Example	**ANL A,@R1**		
Before Execution	**After Execution**		
A = FFH, R1 = 50H & 50H = F0H	**A = F0H**		

Syntax		Flags affected	Bytes	Cycles
ANL direct,A		**NONE**	**2**	**1**
Operation	**(direct) ←— (direct) AND (A)**			
Description	AND the content of accumulator with the content of direct address and result is stored in direct address.			
Example	**ANL 50H,A**			
Before Execution		**After Execution**		
50H = FFH & A = 0FH		**50H = 0FH**		

Syntax		Flags affected	Bytes	Cycles
ANL direct,#data		**NONE**	**3**	**2**
Operation	**(direct) ←— (direct) AND 8-bit data**			
Description	AND the content of 8-bit immediate data with the content of direct address and result is stored in direct address.			
Example	**ANL 50H,#0FH**			
Before Execution		**After Execution**		
50H= AAH		**50H= 0AH**		

Syntax		Flags affected	Bytes	Cycles
ORL A,#data		**NONE**	**2**	**1**
Operation	**(A) ←— (A) OR 8-bit data**			
Description	OR the content of Accumulator with the 8-bit immediate data and result is stored in accumulator.			
Example	**ORL A,#00H**			
Before Execution		**After Execution**		
A = 75H & R0 = 00H		**A = 75H**		

Syntax		Flags affected	Bytes	Cycles
ORL A,Rn		NONE	1	1
Operation	**(A) ←— (A) OR (Rn)** where **n** = 0,1,2,3,4,5,6,7 i.e., **R0 to R7**			
Description	OR the content of register Rn with the content of accumulator and result is stored in accumulator.			
Example	ORL A,R0			
Before Execution		**After Execution**		
A = 75H & R0 = 00H		**A = 75H**		

Syntax		Flags affected	Bytes	Cycles
ORL A,direct		**NONE**	**2**	**1**
Operation	**(A) ←— (A) OR (direct)** where **direct** is the **address**			
Description	OR the content of **direct address** with the content of accumulator and result is stored in accumulator.			
Example	**ORL A,50H**			
Before Execution		**After Execution**		
A = 75H & 50H = FFH		**A=FFH**		

Syntax		Flags affected	Bytes	Cycles
ORL A,@Ri		**NONE**	**1**	**1**
Operation	**(A) ←— (A) OR ((Ri))** where **n** = 0,1,2,3,4,5,6,7 i.e., **R0 to R7**			
Description	OR the content of **indirect address** with the content of accumulator and result is stored in accumulator.			
Example	**ORL A,@R0**			
Before Execution		**After Execution**		
A = FFH, R1 = 50H & 50H = 00H		**A = FFH**		

Syntax		Flags affected	Bytes	Cycles
ORL direct,A		**NONE**	**2**	**1**
Operation	**(A) ←— (direct) OR (A)** where **direct** is the **address**			
Description	OR the content of accumulator with the content of direct address and result is stored in direct address.			
Example	**ORL 50H,A**			
Before Execution		**After Execution**		
A = FFH & 50H = FFH		**A = FFH**		

Syntax		Flags affected	Bytes	Cycles
ORL direct,#data		**NONE**	**3**	**2**
Operation	**(direct) ←— (direct) OR 8 - bit data**			
Description	OR the content of 8-bit immediate data with the content of direct address and result is stored in direct address.			
Example	ORL 20H,#50H			
Before Execution		**After Execution**		
20H = 32H & 8-bit data (50H)		**20H = 72H**		

Syntax		Flags affected	Bytes	Cycles
XRL A,#data		**NONE**	**2**	**1**
Operation	**(A) ⟵ (A) EX-OR 8-bit data**			
Description	**Exclusive OR** the **8-bit immediate** data with **accumulator** content and result is stored in accumulator.			
Example	**XRL A,#09H**			
Before Execution		**After Execution**		
A = 39H & 8-bit data = 09H		**A = 30H**		

Syntax		Flags affected	Bytes	Cycles
XRL A,Rn		**NONE**	**1**	**1**
Operation	**(A) ⟵ (A)EX-OR (Rn)** where **n** = 0,1,2,3,4,5,6,7 i.e., **R0 to R7**			
Description	**Exclusive OR** the content of **register Rn** with **accumulator** content and result is stored in accumulator.			
Example	**XRL A,R1**			
Before Execution		**After Execution**		
A = 39H & R1 = 09H		**A = 30H**		

Syntax		Flags affected	Bytes	Cycles
XRL A,direct		**NONE**	**2**	**1**
Operation	**(A) ⟵ (A) EX-OR (direct address)** where **direct** is the **address**			
Description	**Exclusive OR** the content of **direct address** with **accumulator** contents and result is stored in accumulator.			
Example	**XRL A,50H**			
Before Execution		**After Execution**		
A = 39H & 50H = 09H		**A = 30H**		

Syntax		Flags affected	Bytes	Cycles
XRL A,@Ri		**NONE**	**1**	**1**
Operation	(A) ⟵ (A) EX-OR ((Ri)) where i = 0 & 1 i.e. R0 & R1			
Description	**Exclusive OR** the content of **indirect RAM** address with **accumulator** content and result is stored in accumulator.			
Example	**XRL A,@R0**			
Before Execution		**After Execution**		
A = 39H, R0 = 50H & 50H = 09H		**A = 30H**		

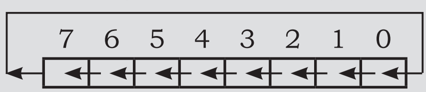
Syntax	Flags affected	Bytes	Cycles
XRL direct,A	**NONE**	**2**	**1**
Operation	**(A) $\longleftarrow$ (direct) EX-OR (A)**		
Description	**Exclusive OR** the content of **direct address** with **accumulator** contents and result is stored in direct address.		
Example	**XRL 50H,A**		
Before Execution	**After Execution**		
50H = 39H & A = 09H	**50H = 30H**		

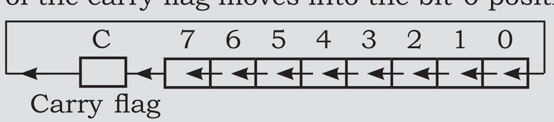
Syntax	Flags affected	Bytes	Cycles
XRL direct,#data	**NONE**	**3**	**2**
Operation	(A) $\longleftarrow$ (direct) EX-OR 8-bit data		
Description	**Exclusive OR** the content of **direct address** with **8-bit data** and result is stored in direct address.		
Example	**XRL 50H,#09H**		
Before Execution	**After Execution**		
50H = 39H & A = 09H	**50H = 30H**		

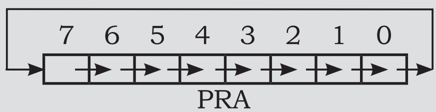
Syntax	Flags affected	Bytes	Cycles
CLR A	**NONE**	**1**	**1**
Operation	**Clear Accumulator** **(A) $\longleftarrow$ 0**		
Description	The content of Accumulator is cleared (A=00H). All the bits of the accumulator are set to 0.		
Example	CLR A		
Before Execution	**After Execution**		
A = FFH	**A = 00H**		

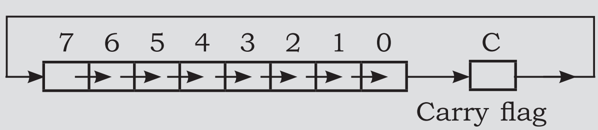
Syntax	Flags affected	Bytes	Cycles
CPL A	**NONE**	**1**	**1**
Operation	**Complement Accumulator** **(A) $\longleftarrow$ ($\overline{A}$)**		
Description	The content of Accumulator is Complemented. The result is 1s complement of the accumulator i.e. 0s become 1s & 1s become 0s.		
Example	**CPL A**		
Before Execution	**After Execution**		
A = 00H	**A = FFH**		

Syntax		Flags affected	Bytes	Cycles
SWAP A		**NONE**	**1**	**1**
Operation	**SWAP** nibbles within the Accumulator $\mathbf{(A) \longleftarrow [(A)_{7-4} \longleftrightarrow (A)_{3-0}]}$			
Description	The SWAP instruction interchanges the lower nibble $\mathbf{(A)_{3\text{-}0}}$ with the upper nibble $\mathbf{(A)_{7\text{-}4}}$ within the Accumulator.			
Example	SWAP A			
Before Execution		**After Execution**		
A = AFH		**A = FAH**		

Syntax		Flags affected	Bytes	Cycles
RL A		**NONE**	**1**	**1**
Operation	Rotate Accumulator Left $\mathbf{A_{n+1} \longleftarrow A_n}$ where n = 0 to 6 $\mathbf{A_0 \longleftarrow A_7}$			
Description	The **RL A** instruction rotates the eight bits in the accumulator left one bit position. Thebit 7 of the accumulator is rotated into bit 0, bit 0 into bit 1, bit 1 into bit 2, and so on. 			
Example	**RL A**			
Before Execution		**After Execution**		
A = C2 H		**A = 85H**		

Syntax		Flags affected	Bytes	Cycles
RLC A		**CY**	**1**	**1**
Operation	Rotate Accumulator Left through the Carry Flag. $\mathbf{A_{n+1} \longleftarrow A_n}$ where n = 0 to 6 $\mathbf{A_0 \longleftarrow C}$ $\mathbf{C \longleftarrow A_7}$			
Description	The 8-bits in the accumulator & the carry flag are together rotated 1-bit to the left. The bit-7 moves into the Carry flag. The original state of the carry flag moves into the bit-0 position. 			
Example	**RLC A**			
Before Execution		**After Execution**		
A = C2 H & CY = 0		**A = 84H & CY =1**		

Syntax	Flags affected	Bytes	Cycles
RR A	**NONE**	**1**	**1**
Operation	Rotate Accumulator Right through the carry flag. $\mathbf{A_n \longleftarrow A_{n+1}}$ where n = 0 to 6 $\mathbf{A_7 \longleftarrow A_0}$		
Description	The 8-bits in the accumulator are rotated 1-bit to the right i.e. bit-0 is rotated into the bit-7 position. 		
Example	**RR A**		
Before Execution	**After Execution**		
A = C2 H	**A = 61H**		

Syntax	Flags affected	Bytes	Cycles
RRC A	**CY**	**1**	**1**
Operation	Rotate Accumulator Right through the Carry Flag. $\mathbf{A_n \longleftarrow A_{n+1}}$ where n = 0 to 6 $\mathbf{A_7 \longleftarrow C}$ $\mathbf{C \longleftarrow A_0}$		
Description	The 8-bits in the accumulator & the carry flag are together rotated 1-bit to the right. Bit-0 moves into the Carry flag. The original state of the carry flag moves into the bit-7 position. 		
Example	**RRC A**		
Before Execution	**After Execution**		
A = C2 H & CY = 0	**A = 61H & CY = 0**		

Logical Instructions

Mnemonics	**Byte**	**Cycle**	**Operation**	**Flags Affected**

ANL A,#data	**2**	**1**	**(A) ← (A) AND 8-bit data**	**NONE**
ANL A,Rn	**1**	**1**	**(A) ← (A) AND (Rn)**	**NONE**
ANL A,direct	**2**	**1**	**(A) ← (A) AND (direct address)**	**NONE**
ANL A,@Ri	**1**	**1**	**(A) ← (A) AND (Ri))**	**NONE**
ANL direct,A	**2**	**1**	**(direct) ← (direct) AND (A)**	**NONE**
ANL direct,#data	**3**	**2**	**(direct) ← (direct) AND 8-bit data**	**NONE**
ORL A,#data	**2**	**1**	**(A) ← (A) OR 8-bit data**	**NONE**
ORL A,Rn	**1**	**1**	**(A) ← (A) OR (Rn)**	**NONE**
ORL A,direct	**2**	**1**	**(A) ← (A) OR (direct)**	**NONE**
ORL A,@Ri	**1**	**1**	**(A) ← (A) OR ((Ri))**	**NONE**
ORL direct,A	**2**	**1**	**(A) ← (direct) OR (A)**	**NONE**
ORL direct,#data	**3**	**2**	**(direct) ← (direct) OR 8 - bit data**	**NONE**
XRL A,#data	**2**	**1**	**(A) ← (A) EX-OR 8-bit data**	**NONE**
XRL A,Rn	**1**	**1**	**(A) EX-OR (Rn)**	**NONE**
XRL A,direct	**2**	**1**	**(A) ← (A) EX-OR (direct address)**	**NONE**
XRL A,@Ri	**1**	**1**	**(A) ← (A) EX-OR ((Ri))**	**NONE**
XRL direct,A	**2**	**1**	**(A) ← (direct) EX-OR (A)**	**NONE**
CLR A	**1**	**1**	**(A) ← 0**	**NONE**
CPL A	**1**	**1**	**(A) ←** $(\bar{A})$	**NONE**
SWAP A	**1**	**1**	$(B) \leftarrow [(A)_{7-4} \longleftrightarrow (A)_{3-0}]$	**NONE**
RL A	**1**	**1**	**Rotate Accumulator Left** $A_{n+1} \longleftarrow A_n$ where n = 0 to 6 $A_0 \longleftarrow A_7$	**NONE**
RLC A	**1**	**1**	**Rotate Accumulator Left through the Carry Flag.** $A_{n+1} \longleftarrow A_n$ where n = 0 to 6 $A_0 \longleftarrow C$ $C \longleftarrow A_7$	**CY**
RR A	**1**	**1**	**Rotate Accumulator Right through the carry flag.** $A_n \longleftarrow A_{n+1}$ where n = 0 to 6 $A_7 \longleftarrow A_0$	**NONE**
RRC A	**1**	**1**	**Rotate Accumulator Right through the Carry Flag.** $A_n \longleftarrow A_{n+1}$ where n = 0 to 6 $A_7 \longleftarrow C$ $C \longleftarrow A_0$	**CY**

Boolean Instructions

Syntax		Flags affected	Bytes	Cycles
CLR C		**CY**	**1**	**1**
Operation	**(CY) ←— 0**			
Description	**Clear** the **carry flag bit.**			
Example	**CLR C**			
Before Execution		**After Execution**		
CY = 1		**CY = 0**		

Syntax		Flags affected	Bytes	Cycles
CLR bit		**NONE**	**2**	**1**
Operation	**(bit) ←— 0**			
Description	**Clear** the **specified bit.**			
Example	CLR P0.0			
Before Execution		**After Execution**		
P0.0 = X		**P0.0 = 0**		

Syntax		Flags affected	Bytes	Cycles
SETB C		**CY**	**1**	**1**
Operation	(CY) ←— 1			
Description	**SET** the **carry flag bit.**			
Example	**SETB C**			
Before Execution		**After Execution**		
CY = X		**CY = 1**		

Syntax		Flags affected	Bytes	Cycles
SETB bit		NONE	2	1
Operation	(bit) ←— 1			
Description	**SET** the **specified bit.**			
Example	**SETB ACC.1**			
Before Execution		**After Execution**		
ACC.1 = X		**ACC.1 = 1**		

Syntax		Flags affected	Bytes	Cycles
CPL C		**CY**	**1**	**1**
Operation	**(CY) ⟵ NOT(CY)**			
Description	**Complement** the **carry flag bit.**			
Example	**CPL C**			
Before Execution		**After Execution**		
CY = 0		**CY = 1**		

Syntax		Flags affected	Bytes	Cycles
CPL bit		**NONE**	**2**	**1**
Operation	**(bit) ⟵ NOT(bit)**			
Description	**Complement** the **specified bit.**			
Example	**CPL ACC.5**			
Before Execution		**After Execution**		
ACC.5 = 1		**ACC.5 = 0**		

Syntax		Flags affected	Bytes	Cycles
ANL C,bit		**CY**	**2**	**2**
Operation	**(C) ⟵ (C) AND (bit)**			
Description	**AND** the content of **carry flag bit** with a **source bit** and the **result** is placed in **carry flag**.			
Example	**ANL C, P1.7**			
Before Execution		**After Execution**		
CY = 1 & P1.7 = 1		**CY = 1**		

Syntax		Flags affected	Bytes	Cycles
ANL C,/bit		**CY**	**2**	**2**
Operation	**(C) ⟵ (C) AND [NOT(bit)]**			
Description	**AND** the content of **carry flag bit** with a **complement** of **source bit** and the **result** is placed in **carry flag**.			
Example	**ANL C,/ACC.7**			
Before Execution		**After Execution**		
CY = 1 & ACC.7 = 0		**CY = 1**		

Syntax		Flags affected	Bytes	Cycles
ORL C,bit		**CY**	**2**	**2**
Operation	(C) ⟵ (C) OR (bit)			
Description	**OR** the content of **carry flag bit** with a **source bit** and the **result** is placed in **carry flag**.			
Example	ORL C,P0.0			
Before Execution		**After Execution**		
CY = 0 & P0.0 = 1		**CY = 1**		

Syntax		Flags affected	Bytes	Cycles
ORL C,/bit		**CY**	**2**	**2**
Operation	**(C) ⟵ (C) OR [NOT(bit)]**			
Description	**OR** the content of **carry flag bit** with a **complement** of **source bit** and the **result** is placed in **carry flag**.			
Example	**ORL C,/ACC.0**			
Before Execution		**After Execution**		
CY = 0 & ACC.0 = 0		**CY = 1**		

Syntax		Flags affected	Bytes	Cycles
MOV C,bit		**CY**	**2**	**1**
Operation	**(C) ⟵ (bit)**			
Description	**Move** the content of **source bit** to **carry flag**.			
Example	**MOV C,P1.7**			
Before Execution		**After Execution**		
CY = X & P1.7 = 1		**CY = 1**		

Syntax		Flags affected	Bytes	Cycles
MOV bit,C		NONE	2	2
Operation	**(bit) ⟵ (C)**			
Description	**Move** the content of **carry flag** to **destination bit**.			
Example	**MOV ACC.0,C**			
Before Execution		**After Execution**		
ACC.0 = X & CY = 1		**ACC.0 = 1**		

Boolean Instructions

Mnemonics	Byte	Cycle	Operation	Flags Affected
CLR C	1	1	(CY) $\longleftarrow$ 0	CY
CLR bit	1	1	(bit) $\longleftarrow$ 0	NONE
SETB C	1	1	(CY) $\longleftarrow$ 1	CY
SETB bit	2	1	(bit) $\longleftarrow$ 1	NONE
CPL C	1	1	(CY) $\longleftarrow$ NOT(CY)	CY
CPL bit	2	1	(bit) $\longleftarrow$ NOT(bit)	NONE
ANL C,bit	2	2	(C) $\longleftarrow$ (C) AND (bit)	CY
ANL C,/bit	2	2	(C) $\longleftarrow$ (C) AND [NOT(bit)]	CY
ORL C,bit	2	2	(C) $\longleftarrow$ (C) OR (bit)	CY
ORL C,/bit	2	2	(C) $\longleftarrow$ (C) OR [NOT(bit)]	CY
MOV C,bit	2	1	(C) $\longleftarrow$ (bit)	CY
MOV bit,C	2	2	(bit) $\longleftarrow$ (C)	NONE

Program Branching Instructions

Syntax	Flags affected	Bytes	Cycles
ACALL addr11	**NONE**	**2**	**2**

Operation	(PC) $\longleftarrow$ (PC) + 2 (SP) $\longleftarrow$ (SP) + 1 (SP) $\longleftarrow$ $(PC)_{7\text{-}0}$ (SP) $\longleftarrow$ (SP) + 1 (SP) $\longleftarrow$ $(PC)_{15\text{-}8}$ $(PC)_{10\text{-}0}$ $\longleftarrow$ Page address
Description	**Absolute subroutine call. Transfer control to a subroutine.** • ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes 16-bit result onto the stack i.e., low order byte 1st & increment the stack pointer to store higher-order byte. • ACALL is a 2-byte instruction, in which 5-bits are used for the opcode and the remaining 11-bits are used for the target subroutine address. • A 11-bit address limits the range to 2 Kbytes.

Syntax		Flags affected	Bytes	Cycles
LCALL addr16		**NONE**	**3**	**2**
Operation	**(PC) ⟵ (PC) + 3** **(SP) ⟵ (SP) + 1** **(SP) ⟵ $(PC)_{7\text{-}0}$** **(SP) ⟵ (SP) + 1** **(SP) ⟵ $(PC)_{15\text{-}8}$** **(PC) ⟵ Page address**			
Description	• **Long call. Transfer control to a subroutine.** LCALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC thrice to obtain the address of the next instruction & then pushes 16-bit result onto the stack i.e. low order byte 1st & increment the stack pointer to store higher-order byte. • LCALL is a 2-byte instruction, in which 16-bits are used for the opcode and for the target subroutine address. • A 16-bit address may be anywhere within the 64 Kbytes of program memory.			

Syntax		Flags affected	Bytes	Cycles
RET		**NONE**	**1**	**2**
Operation	**$(PC)_{15\text{-}8}$ ⟵ (SP)** **(SP) ⟵ (SP) -1** **$(PC)_{7\text{-}0}$ ⟵ (SP)** **(SP) ⟵ (SP) -1**			
Description	**Returns from subroutine.** • RET instruction is used to return from a subroutine previously entered by instruction LCALL or ACALL. The top two bytes of the stack are popped in the program counter (PC) & program execution continues at this new address. • After popping the top two bytes of the stack into the program counter, the stack pointer (SP) is decremented by 2.			

Syntax	Flags affected	Bytes	Cycles

RETI		**NONE**	**1**	**2**
Operation	**(PC)15-8 ← (SP)** **(SP) ← (SP)-1** **$(PC)_{7\text{-}0}$ ← (SP)** **(SP) ← (SP) -1**			
Description	**Absolute subroutine call.** • RET instruction is used at the end of an Interrupt Service Routine (ISR). The top two bytes of the stack are popped in the program counter (PC), the stack pointer (SP) is decremented by 2.			

Syntax		Flags affected	Bytes	Cycles
AJMP addr11		**NONE**	**2**	**2**
Operation	**(PC) ← (PC) + 2** **$(PC)_{10\text{-}0}$ ← $(A)_{10\text{-}0}$**			
Description	Absolute subroutine call • The **AJMP** instruction transfers program execution to the specified address. • The address is formed by combining the 5 high-order bits of the address of the following instruction (for A_{15}-A_{11}), the 3 high-order bits of the opcode (for A_{10}-A_8), and the second byte of the instruction (for A_7-A_0). • The destination address must be located in the same 2 Kbyte block of program memory as the opcode following the **AJMP** instruction.			
Example	**AJMP LABEL**			

Syntax		Flags affected	Bytes	Cycles
LJMP addr16		**NONE**	**3**	**2**
Operation	**(PC) ← (PC) + 2** **(PC) ← (PC) + rel**			
Description	**Long Jump.** The **LJMP** instruction transfers program execution to the specified 16-bit address. • Jump unconditionally to the specified address (i.e. LABEL) by loading high order and low order bytes of the PC respectively. • Its range is -32768 bytes to +32767 bytes. • The destination may be anywhere within the 64 Kbytes of program memory.			
Example	**LJMP LABEL**			

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">SJMP rel</td><td>NONE</td><td>2</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(PC) ⟵ (PC) + 2
(PC) ⟵ (PC) + rel</td></tr>
<tr><td>Description</td><td colspan="4">Short Jump.
Jump unconditionally to the specified address (i.e. LABEL).
Its range is -128 bytes to +127 bytes.</td></tr>
<tr><td>Example</td><td colspan="4">0091 MOV R0,#05H
0092 SJMP, NEXT
0093 MOV P1,A
0094 NEXT: INC R1</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">PC = 0092</td><td colspan="3">PC = 0094</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">JC rel</td><td>NONE</td><td>2</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(PC) ⟵ (PC) + 2
If CY=1
Then,
(PC) ⟵ (PC) + rel</td></tr>
<tr><td>Description</td><td colspan="4">Jump if carry is set.
If CY = 1, jump to the address indicated otherwise proceeds (Execute) with the next instruction.</td></tr>
<tr><td>Example</td><td colspan="4">0090 SETB C
0091 JC, NEXT
0092 MOV P1,A
0093 NEXT: INC R1</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">CY = 1 & PC = 0091</td><td colspan="3">PC = 0093</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">JNC rel</td><td>NONE</td><td>2</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(PC) ⟵ (PC) + 2
If CY=0
Then,
(PC) ⟵ (PC) + rel</td></tr>
<tr><td>Description</td><td colspan="4">Jump if not carry.
If CY = 0, jump to the address indicated otherwise proceeds (Execute) with the next instruction.</td></tr>
<tr><td>Example</td><td colspan="4">0090 SETB C
0091 JNC, NEXT
0092 MOV P1,A
0093 NEXT: INC R1</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">CY = 1 & PC = 0091</td><td colspan="3">PC = 0092</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">JB bit,rel</td><td>NONE</td><td>3</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(PC) ⟵ (PC) + 3
If (bit) = 1
Then,
(PC) ⟵ (PC) + rel</td></tr>
<tr><td>Description</td><td colspan="4">Jump if bit set.
If the indicated bit is SET (1), jump to the address indicated otherwise proceeds (Execute)with the next instruction.</td></tr>
<tr><td>Example</td><td colspan="4">0090 SETB ACC.0
0091 JB ACC.0, NEXT
0092 MOV P1,A
0093 NEXT: INC R1</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">ACC.0 = 1 & PC = 0091</td><td colspan="3">ACC.0 = 1 & PC = 0093</td></tr>
</table>

<table>
<tr><td>Syntax</td><td></td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">JNB bit,rel</td><td>NONE</td><td>3</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(PC) ⟵ (PC) + 3
If (bit) = 0
Then,
(PC) ⟵ (PC) + rel</td></tr>
<tr><td>Description</td><td colspan="4">Jump if bit not set
If the indicated bit is RESET (0), jump to the address indicated otherwise proceeds (Execute) with the next instruction.</td></tr>
<tr><td>Example</td><td colspan="4">0090 MOV C, 0
0091 MOV ACC.0, C
0092 JNB ACC.0, NEXT
0093 MOV P1, A
0094 NEXT: INC R1</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">ACC.0 = 0 & PC = 0092</td><td colspan="3">ACC.0 = 0 & PC = 0094</td></tr>
</table>

<table>
<tr><td>Syntax</td><td></td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">JBC bit,rel</td><td>NONE</td><td>3</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(PC) ⟵ (PC) + 3
If (bit) = 1
Then,
(PC) ⟵ (PC) + rel</td></tr>
<tr><td>Description</td><td colspan="4">Jump if bit set & clear it.
If the bit = 1, then processor jump to the specified address (i.e. LABEL), at the same time the bit is cleared to zero i.e. bit = 0.
If the bit = 0, then processor proceeds (Execute)with the next instruction.</td></tr>
<tr><td>Example</td><td colspan="4">0090 SETB ACC.0
0091 JBC ACC.0, NEXT
0092 MOV P1,A
0093 NEXT: INC R1</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">ACC.0 = 1 & PC = 0091</td><td colspan="3">ACC.0 = 0 & PC = 0093</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">JMP @A+DPTR</td><td>NONE</td><td>1</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(PC) ←— (A) + (DPTR)</td></tr>
<tr><td>Description</td><td colspan="4">Jump indirect relative to the DPTR.
Jump unconditionally to the specified address (i.e. LABEL). The target address is provided by the total sum of Accumulator & the content of DPTR register.
This instruction is not widely used.</td></tr>
<tr><td>Example</td><td colspan="4">0090 MOV A,#24H
0091 MOV DPTR,#0070H
0092 JMP @A+DPTR
0093 MOV P1,A
0094 INC R1</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">A = 24H, DPTR = 0070H
& PC = 0092</td><td colspan="3">PC = 0094</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">JNZ rel</td><td>NONE</td><td>2</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(PC) ←— (PC) + 2
If A ≠ 0
Then,
(PC) ←— (PC) + rel</td></tr>
<tr><td>Description</td><td colspan="4">Jump if Accumulator is NOT zero.
If ACC ≠ 0, then processor jump to the specified address (i.e. LABEL),
If ACC = 0, then processor proceeds (Execute) with the next instruction.</td></tr>
<tr><td>Example</td><td colspan="4">0090 MOV A,#05H
0091 ADD A,#03H
0092 JNZ, NEXT
0093 MOV P1,A
0094 NEXT: INC R1</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">A = 08H & PC = 0092</td><td colspan="3">PC = 0094</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">JZ rel</td><td>NONE</td><td>2</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(PC) ⟵ (PC) + 2
If A = 0
Then,
(PC) ⟵ (PC) + rel</td></tr>
<tr><td>Description</td><td colspan="4">Jump if Accumulator is zero.
If ACC = 0, then processor jump to the specified address (i.e. LABEL),
If ACC ≠ 0, then processor proceeds (Execute) with the next instruction.</td></tr>
<tr><td>Example</td><td colspan="4">0090 MOV A,#05H
0091 ADD A,#03H
0092 JZ, NEXT
0093 MOV P1,A
0094 NEXT: INC R1</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="3">After Execution</td></tr>
<tr><td colspan="2">A = 08H & PC = 0092</td><td colspan="3">PC = 0093</td></tr>
</table>

<table>
<tr><td colspan="2">Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td colspan="2">CJNE A,direct,rel</td><td>CY</td><td>3</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="4">(PC) ⟵ (PC) + 3
If (A) ≠ (direct)
then
(PC) ⟵ (PC) + relative address

If (A) < (direct)
then
(C) ⟵ 1
else
(C) ⟵ 0</td></tr>
<tr><td>Description</td><td colspan="4">Compare and Jump if not equal.
• The magnitudes of the source byte and destination byte are compared. If they are not equal, it jumps to the target.</td></tr>
</table>

Syntax	Flags affected	Bytes	Cycles
CJNE A,#data,rel	**CY**	**3**	**2**
Operation	**(PC) ⟵ (PC) + 3** **If (A) ≠ data** **then** **(PC) ⟵ (PC) + relative address** **If (A) < (data)** **then** **(C) ⟵ 1** **else** **(C) ⟵ 0**		
Description	**Compare and Jump if not equal.** The magnitudes of the source byte and destination byte are compared. If they are not equal, it jumps to the target.		

Syntax	Flags affected	Bytes	Cycles
CJNE Rn,#data,rel	**CY**	**3**	**2**
Operation	**(PC) ⟵ (PC) + 3** **If (Rn) ≠ data** **then** **(PC) ⟵ (PC) + relative address** **If (Rn) < (data)** **then** **(C) ⟵ 1** **else** **(C) ⟵ 0**		
Description	**Compare and Jump if not equal.** The magnitudes of the source byte and destination byte are compared. If they are not equal, it jumps to the target.		

Syntax	Flags affected	Bytes	Cycles
CJNE @Ri,#data,rel	**CY**	**3**	**2**

Operation	**(PC) ←—— (PC) + 3**
	If ((Ri)) ≠ data
	then
	(PC) ←—— (PC) + relative address
	
	If ((Ri)) < (data)
	then
	    (C) ←—— 1
	else
	    (C) ←—— 0		

Description	Absolute subroutine call		
Example	**Compare and Jump if not equal.** The magnitudes of the source byte and destination byte are compared. If they are not equal, it jumps to the target.		

Syntax	Flags affected	Bytes	Cycles
DJNZ Rn,rel	**NONE**	**2**	**2**
Operation	**(PC) ←—— (PC) + 2** **(Rn) ←—— (Rn) -1** **If (Rn) ≠ 0 i.e. (Rn) > 0& (Rn) < 0** Then, **(PC) ←—— (PC) + rel**		
Description	**Decrement and Jump if not zero.** Decrement the content of Rn register & then checks the condition i.e. Rn≠ **0**. If Rn≠ **0,** jump to the specified address (i.e. LABEL), If Rn= **0**, then processor proceeds (Execute) with the next instruction.		
Example	0091 MOV R0,#05H 0092 **DJNZ R0**, **NEXT** 0093 MOV P1,A 0094 **NEXT:** INC R1		
Before Execution	**After Execution**		
Rn = 05H & PC = 0092	**PC = 0094**		

<table>
<tr><td>Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td>DJNZ direct,rel</td><td></td><td>3</td><td>2</td></tr>
<tr><td>Operation</td><td colspan="3">(PC) ← (PC) + 2
(direct) ← (direct) -1
If (direct) ≠ 0 i.e. (direct) > 0& (direct) < 0
Then,
(PC) ← (PC) + rel</td></tr>
<tr><td>Description</td><td colspan="3">Decrement and Jump if not zero.
Decrement the content of direct address & then checks the condition i.e. (direct) ≠ 0.
If (direct) ≠ 0, jump to the specified address (i.e. LABEL),
If (direct) = 0, then processor proceeds (Execute) with the next instruction.</td></tr>
<tr><td>Example</td><td colspan="3">0091 MOV R0,#05H
0092 DJNZ 50H, NEXT
0093 MOV P1,A
0094 NEXT: INC R1</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="2">After Execution</td></tr>
<tr><td colspan="2">direct = 50H, 50H = 01 & PC = 0092</td><td colspan="2">PC = 0094</td></tr>
</table>

<table>
<tr><td>Syntax</td><td>Flags affected</td><td>Bytes</td><td>Cycles</td></tr>
<tr><td>NOP</td><td>NONE</td><td>1</td><td>1</td></tr>
<tr><td>Operation</td><td colspan="3">(PC) ← (PC) + 1</td></tr>
<tr><td>Description</td><td colspan="3">No operation
• NOP instruction performs NO OPERATION& execution continues with the next instruction.
• It is sometimes used for timing delays to waste clock cycles.
• This instruction only updates the PC to point to the next instruction following up.</td></tr>
<tr><td>Example</td><td colspan="3">0091 MOV A,#25H
0092 ADD A,#20H
0093 NOP
0094 INC A
END</td></tr>
<tr><td colspan="2">Before Execution</td><td colspan="2">After Execution</td></tr>
<tr><td colspan="2">PC = 0093H</td><td colspan="2">PC = 0094H</td></tr>
</table>

Program Branching Instructions

Mnemonics	Byte	Cycle	Operation	Flags Affected
ACALL addr11	2	2	(PC) ⟵ (PC) + 2 (SP) ⟵ (SP) + 1 (SP) ⟵ $(PC)_{7-0}$ (SP) ⟵ (SP) + 1 (SP) ⟵ $(PC)_{15-8}$ $(PC)_{10-0}$ ⟵ Page address	NONE
LCALL addr16	3	2	(PC) ⟵ (PC) + 3 (SP) ⟵ (SP) + 1 (SP) ⟵ $(PC)_{7-0}$ (SP) ⟵ (SP) + 1 (SP) ⟵ $(PC)_{15-8}$ (PC) ⟵ Page address	NONE
RET	1	2	(PC)15-8 (SP) (SP) ⟵ (SP) –1 $(PC)_{7-0}$(SP) ⟵ (SP) ⟵ (SP) –1	NONE
RETI	1	2	(PC)15-8 ⟵ (SP) (SP) ⟵ (SP) -1 $(PC)_{7-0}$(SP) ⟵ (SP) ⟵ (SP) -1	NONE
AJMP addr11	2	2	(PC) ⟵(PC) + 2 $(PC)_{10-0}$ ⟵ $(A)_{10-0}$	NONE
LJMP addr16	3	2	(PC) ⟵ (PC) + 2 (PC) ⟵ (PC) + rel	NONE
SJMP rel	2	2	(PC) ⟵ (PC) + 2 (PC) ⟵ (PC) + rel	NONE
JC rel	2	2	(PC) ⟵ (PC) + 2 If CY=1 Then, (PC) ⟵ (PC) + rel	NONE
JNC rel	2	2	(PC) ⟵ (PC) + 2 If CY=0 Then, (PC) ⟵ (PC) + rel	NONE

JB bit,rel	**3**	**2**	**(PC) ← (PC) + 3** **If (bit) = 1** **Then,** **(PC) ← (PC) + rel**	**NONE**
JBCbit,rel	**3**	**2**	**(PC) ← (PC) + 3** **If (bit) = 1** **Then,** **(PC) (PC) + rel**	**NONE**
JMP @A+DPTR	**1**	**2**	**(PC) ← (A) + (DPTR)**	**NONE**
JNZ rel	**2**	**2**	**(PC) ← (PC) + 2** **If A ≠ 0** **Then,** **(PC) ← (PC) + rel**	**NONE**
JZ rel	**2**	**2**	**(PC) ← (PC) + 2** **If A = 0** **Then,** **(PC) (PC) + rel**	**NONE**
CJNE A,direct,rel	**3**	**2**	**(PC) ← (PC) + 3** **If (A) ≠ (direct)** **then** **(PC) ← (PC) + relative address** **If (A) < (direct)** **then** **(C) ← 1** **else** **(C) ← 0**	**CY**
CJNE A,#data,rel	**3**	**2**	**(PC) ← (PC) + 3** **If (A) ≠ data** **then** **(PC) ← (PC) + relative address** **If (A) < (data)** **then** **(C) ←1** **else** **(C) ← 0**	**CY**

CJNE Rn,#data,rel	**3**	**2**	**(PC) $\leftarrow$ (PC) + 3** **If (Rn) ≠ data** **then** **(PC) $\leftarrow$ (PC) + relative address** **If (Rn) < (data)** **then** **(C) $\leftarrow$ 1** **else** **(C) $\leftarrow$ 0**	**CY**
CJNE @ Ri,#data,rel	**3**	**2**	**(PC) $\leftarrow$ (PC) + 3** **If ((Ri)) ≠ data** **then** **(PC) $\leftarrow$ (PC) + relative address** **If ((Ri)) < (data)** **then** **(C) $\leftarrow$ 1** **else** **(C) $\leftarrow$ 0**	**CY**
DJNZ Rn,rel	**2**	**2**	**(PC) $\leftarrow$ (PC) + 2** **(Rn) $\leftarrow$ (Rn) -1** **If (Rn) ≠ 0 i.e. (Rn) > 0& (Rn) < 0** **Then,** **(PC) $\leftarrow$ (PC) + rel**	**NONE**
DJNZ direct,rel	**3**	**2**	**(PC) $\leftarrow$ (PC) + 2** **(direct) $\leftarrow$ (direct) -1** **If (direct) ≠ 0 i.e. (direct) > 0& (direct) < 0** **Then,** **(PC) $\leftarrow$ (PC) + rel**	**NONE**
NOP	**1**	**1**	**(PC) $\leftarrow$ (PC) + 1**	**NONE**

Machine Cycle

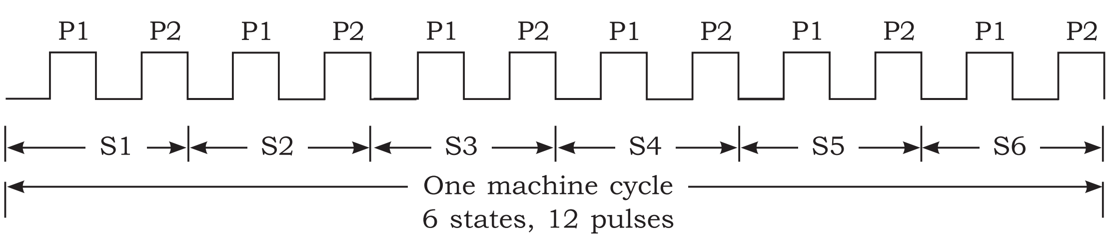


Fig. 2.1.1: Machine cycle

In 8051, two pulses constitute a state and machine cycle is made up of six states. Some instructions may require more than one machine cycle.

The time required to execute an instruction is given by

$$T = \frac{C \times 12d}{f}$$

Where, **T** is the time for instruction to be executed

f is the crystal frequency and

C is the number of machine cycles.

Example2.1

For 8051 microcontroller, find the time taken for an instructions which takes
i) 1 Machine cycle ii) 2 Machine cycles iii) 4 Machine cycles

Solution:

i) $$T = \frac{C \times 12d}{f} = \frac{1 \times 12}{11.0592 \times 10^6} = 1.085\ \mu\text{Sec}$$

ii) $$T = \frac{C \times 12d}{f} = \frac{2 \times 12}{11.0592 \times 10^6} = 2.170\ \mu\text{Sec}$$

iii) $$T = \frac{C \times 12d}{f} = \frac{4 \times 12}{11.0592 \times 10^6} = 4.340\ \mu\text{Sec}$$

Assembly Language Programs

Write an ALP to find the square of a number stored at 30H.

```
Program              Comments
ORG 00H              ; Start
MOV A,30H            ; Move content of 30H to A register
MOV 0F0H, 30H        ; Move content of 30H to B register
MUL AB               ; Square of 30H content
MOV 31H,A            ; Move LSB of the result to 31H
MOV 32H,0F0H         ; Move MSB of the result to 32H
END                  ; End
```

3

8051 Stack, I/O Port Interfacing and Programming

3.1 STACK

- Stack is a section of internal RAM used by the CPU to store information temporarily. This information could be data or an address.
- The register used to access the stack is called the **stack pointer** (SP) register.
- The stack pointer is a 8-bit register used by the 8051 to hold an 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 i.e. **SP = SP + 1**, so that the stack grows up as data is stored.
- When 8051 is **RESET**, the SP is set to **07H**.
- As the data is retrieved from the stack, the byte is read from stack & then SP decrements i.e. SP = SP-1 to point to the next available byte of stored data.
- Storing the data onto stack is called a PUSH.
- Retrieving the contents of the stack is called a POP.
- RAM location 08H is the 1st location used by the stack to store the data.

3.1.1Pushing into stack:

Example 3.1

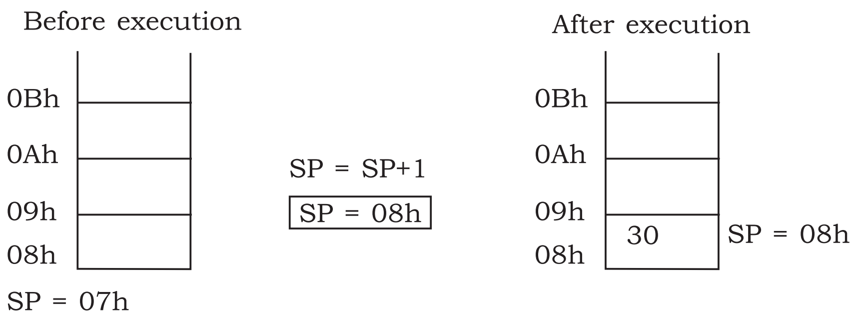
MOV R2, #30H

PUSH 2

Assume that initially Bank 0 is selected & SP=07H

Solution:

SP is 1st incremented by one i.e. SP = SP+1, then the contents of R2 is stored in top of stack i.e. 08H address.

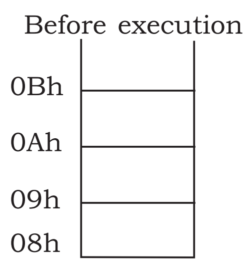


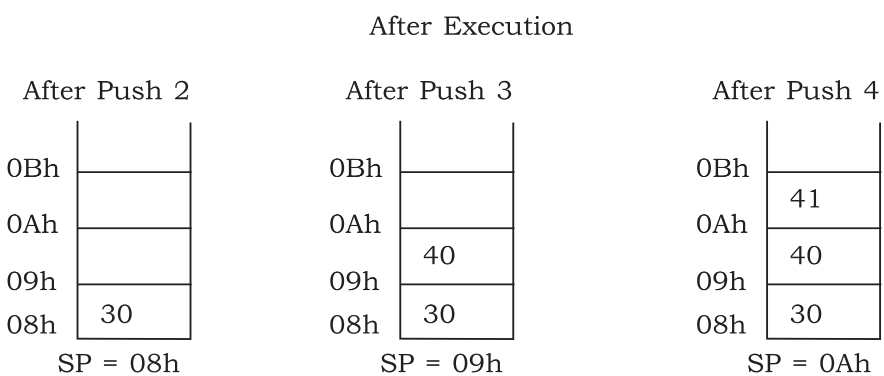
Example 3.2

ORG 00H
MOV R2, #30H
MOV R3, #40H
MOV R4, #41H
PUSH 2
PUSH 3
PUSH 4
END

Solution:

Assume Bank0 is selected and SP has initially 07H





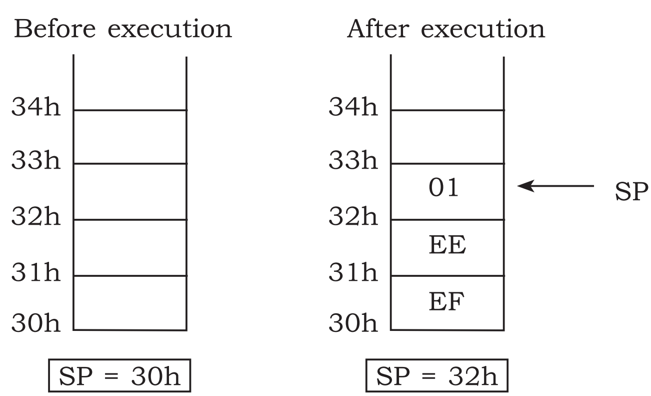
Upper limit of stack:

- Locations 08H to 1FH in 8051 RAM can be used for stack because location 20H-2Fh of RAM are reserved for bit addressable memory and must not be used by stack.
- If in program, we need more than 24 bytes (08H to 1FH = 24 bytes) of stack then we can change SP to point to RAM location 30H to 7FH. This is done by the instruction 'MOV SP,#XXH"

Example 3.3

```
ORG 00H
MOV SP, #30H
SETB PSW.3
MOV R0, #0FFH
MOV R1, #0EEH
MOV A, #01H
PUSH 8
PUSH 9
PUSH 0E0H
END
```

Solution:



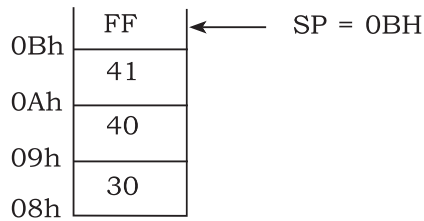
3.1.2 Popping from stack:

Example 3.4:

```
POP    4
POP    3
POP    2
```

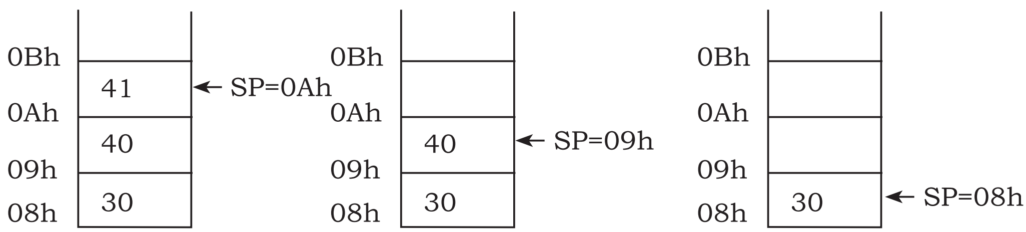
Solution:

Before execution



After execution

After Pop 4 → SP = SP – 1 After Pop 3 → SP = SP – 1 After Pop 2 → SP = SP – 1



(R4) = FFh (R3) = 41h (R2) = 40h

3.2 Jump and Call Instructions:

Jump and call instructions replaces the contents of program counter (PC) with new address and program execution to start from that new address. The difference of this new address from address in program where jump or call instruction is called range of jump or call.

Jump or call instructions may have one of the three ranges:

i) **Relative range**: +127d to -128d

ii) **Absolute range**: within a page (2K bytes)

iii) **Long range**: 0000H to FFFFH

i) Relative range:

- The Jump can be within –128 bytes. (for backward Jump) or +127 bytes (for forward Jump) of memory relative to the address of current program counter (PC).
- Jump or call instruction with relative range will be of 2-byte instructions. The 1st byte is opcode and second byte is relative address of target location.

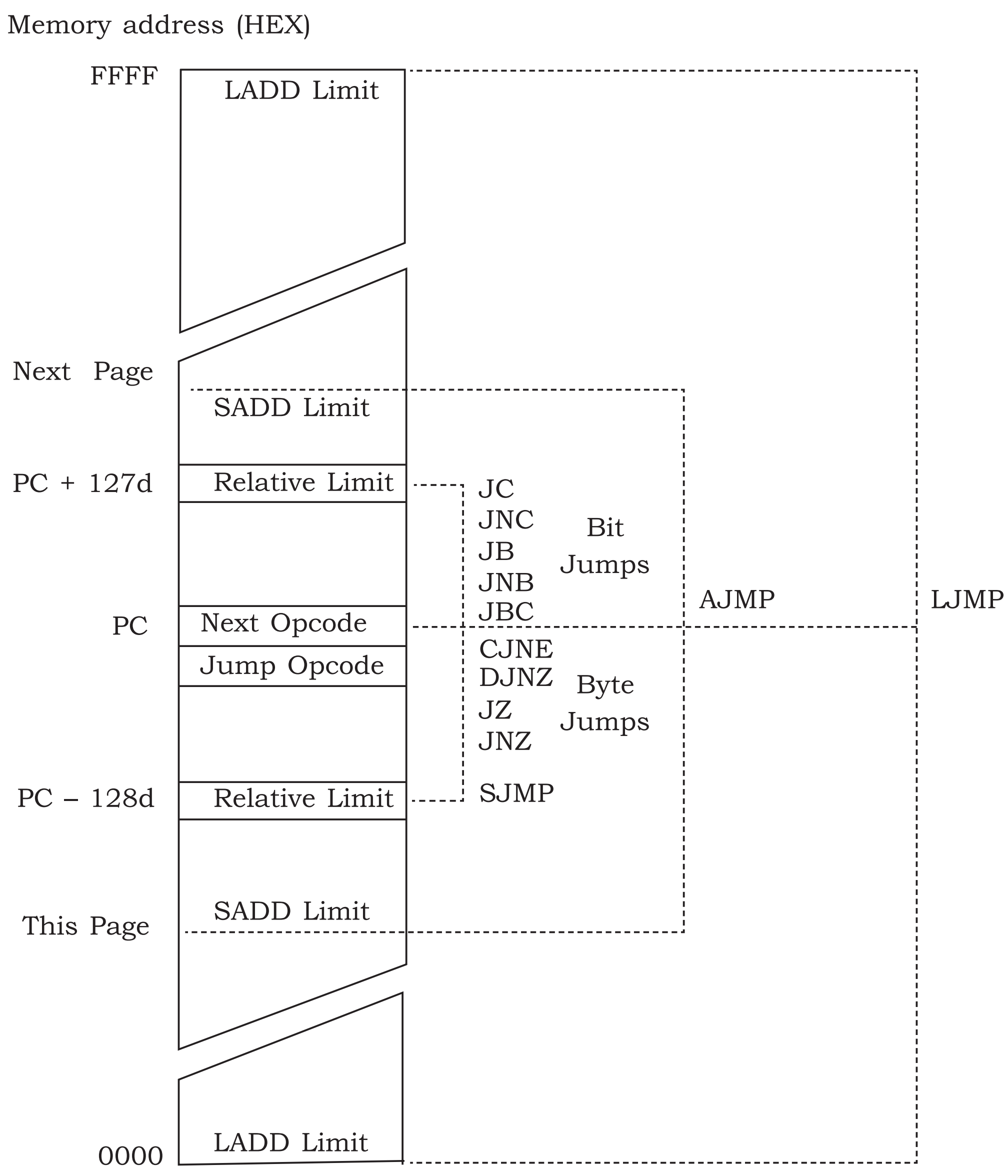


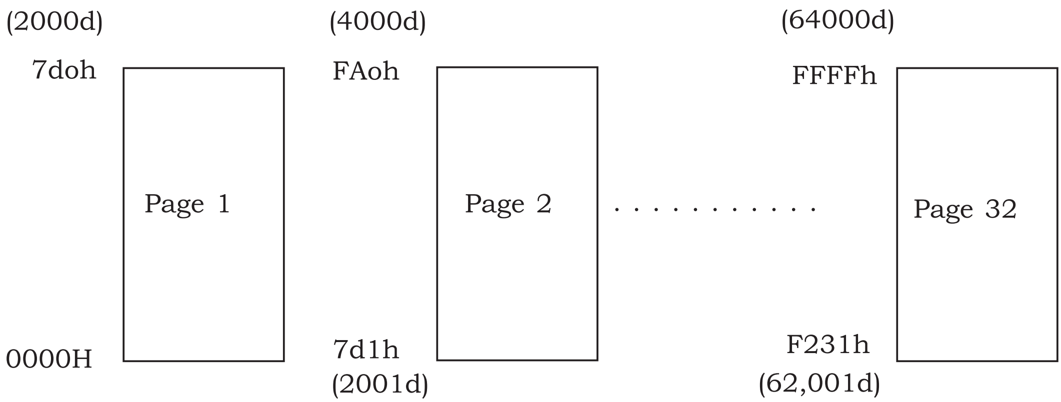
Fig. 3.2.1 Jump instruction ranges

Note: SADD: Short address, LADD: Long address

ii) Absolute range:

- In 8051, program memory is divided into logical divisions called pages each of 2k byte.
- Maximum size program memory is 64 K bytes. Size of each page is 2K bytes.

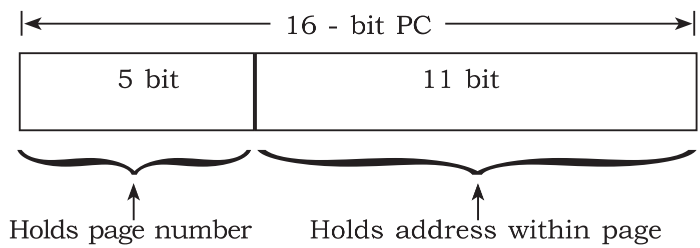
$$\text{Maximum number of pages } = \frac{64\,\text{Kb}}{2\,\text{Kb}} = 32\,\text{pages}$$



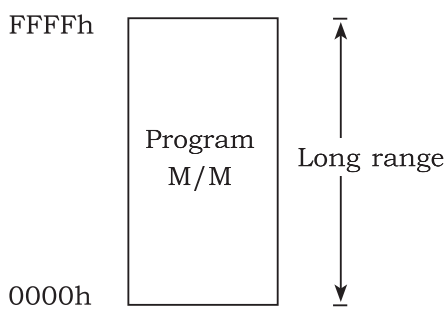
- In absolute range, Jump can be within a single page.
- The upper 5-bits of PC holds the page number and lower 11-bits holds the address within that page.

  i.e., $2^5 \rightarrow 32$ page

  $2^{11} \rightarrow 2$ Kb range



iii) Long Absolute Range:



- This range allows the Jump to any where in the memory location from 0000h to FFFFh.
- The Jump or call instructions with this range will be of 3 byte instructions in which 1st byte is opcode and 2^{nd} and 3^{rd} bytes represents the 16-bit address of target location.

3.2.1 Compare Relative range, Absolute range and Long range:

Type of Jump of CALL	Ranges	No. of bytes	Example
Relative range	–128d to +127d	2-byte instructions	JC, JNC, JB, JNB, JBC, JZ, JNZ, DJNZ, CJNE
Absolute range	Within a page (2 Kbyte)	2-byte instructions	ACALL
Long range	Anywhere within program **(0H to FFFFH)**	3-byte instructions	LCALL

3.3 Subroutine

A subroutine is a program that may be used many times in the execution of a larger program. The subroutine could be written into the body of the main program everywhere it is needed, resulting in the fastest possible code execution.

3.3.1 Call and the stack

A call instruction causes a jump to the address where the called subroutine is located. At the end of the subroutine the program resumes operation at the opcode address immediately following the call.

The stack area of internal RAM is used to automatically store the address, called the return address, of the instruction found immediately after the call. The stack pointer register holds the address of the last space used on the stack.

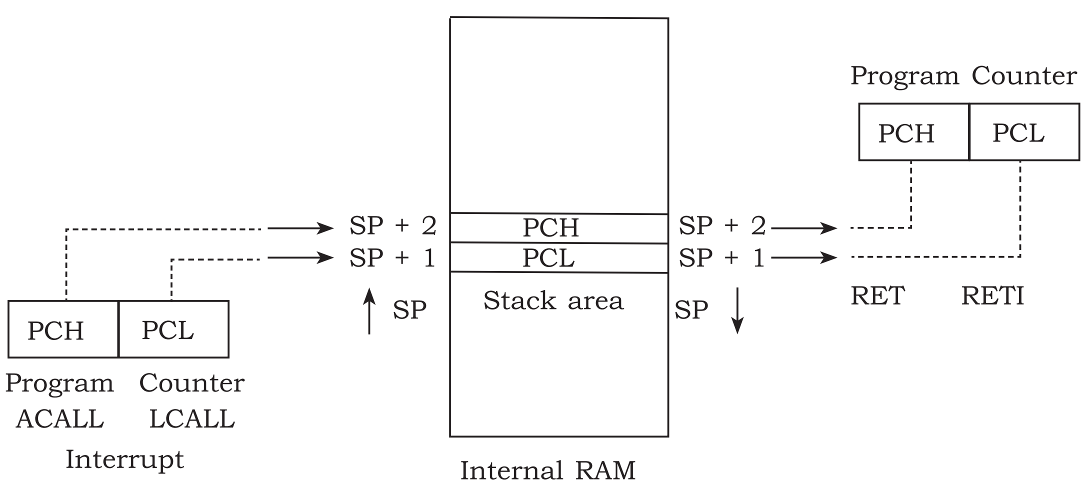


Fig.3.3.1: Storing and retrieving the return address

Figure 3.3.1 shows the following sequence of events.

1. A call opcode occurs in the program software, or an interrupt is generated in the hardware circuitry.
2. The return address of the next instruction after the call instruction or interrupt is found in the program counter.
3. The return address bytes arc pushed on the stack, low byte first.
4. The stack pointer is incremented for each push on the stack.
5. The subroutine address is placed in the program counter.
6. The subroutine is executed.
7. A RET (return) opcode is encountered at the end of the subroutine.
8. Two pop operations restore the return address to the PC from the stack area in internal RAM.
9. The stack pointer is decremented for each address byte pop.

3.4 Assembly language program examples on subroutine and involving loops Delay subroutine

Example 3.5:

Write a ALP to toggle the bits of Port 2 with a delay which depends on the value of a number in R1.

Solution:

METHOD 1	METHOD 2 (Without using counter value of R1)

<pre>   ORG 00H
UP: MOV A, #00H
        MOV P2, A
        MOV R1, #30H
        ACALL DELAY
        CPL A
        MOV P2, A
        MOV R1, #0FFH
        ACALL DELAY
        SJMP UP
 	
DELAY: NOP
REPEAT: DJNZ R1, REPEAT
        RET
        END</pre>
	<pre> ORG 00H
	        MOV A, #00H
	UP: MOV P2, A
	        ACALL DELAY
	        CPL A
	        SJMP UP
	DELAY:
	        MOV R3, #0FFH
	REPEAT DJNZ R3, REPEAT
	        RET
	        END</pre>

Example 3.6:

Factorial of an 8 bit number (result maximum 8 bit)

Solution:

The factorial of a number is given by

N! = N x (N-1) x (N-2) x.....x 3 x 2 x 1

```
    ORG 00h
    MOV A,#01H
    MOV B,A
UP: MUL AB
    INC A
    CJNE B,#06H, UP
    END
```

Result: 6 x 5 x 4 x 3 x 2 x 1 = 120d = 78H

Example 3.7:

Factorial of an 8 bit number (result more than8 bit)

Solution:

```
    ORG 00h
    MOV A,#01H
    MOV B,A
UP: MUL AB
    INC A
    CJNE B,#07H, UP
    END
```

Result: 6 x 5 x 4 x 3 x 2 x 1 = 720d = 2D0H

Example 3.8:

Block move without overlap

Solution:

```
    ORG 0000H      : Specifying starting address
    MOV R0, #50H   : move immediate data 50H into reg R0 [Source
                     location]
    MOV R1, #60H   : move immediate data into 60H reg R1
                     [Destination location]
    MOV R2, #0AH   : move immediate data into 0AH reg R2 [Counter
                     register]
```

```
UP:     MOV A, @R0      : moves contents of R0 address to accumulator A
        MOV @R1, A      : moves contents of accumulator A to contents
                          of R1
        INC R0          : Increment the R0 by one
        INC R1          : Increment the R1 by one
        DJNZ R2, UP     : decrement and jump if R2 is not equals to
                          zero
        END             : Terminate the program
```

Example 3.9:

Addition of N 8 bit numbers

Solution:

```
ORG 00h
MOV B,#00h              ; Clear B to save the result
MOV R0,#30h             ; Use R0 as a pointer to first memory
                          location
MOV A,@R0               ; Transfer data from first memory location
                          to accumulator
AGAIN:
INC R0                  ; Point to next memory location
ADD A,@R0               ; Add data and store result in
                          accumulator
JNC LOOP                ; If no carry do not increment B
INC B
LOOP:
CJNE R0,#50h,AGAIN      ;Add up to memory location 50h
END                     ; End of program
```

Example 3.10:

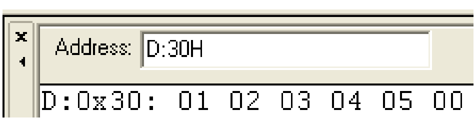
Picking smallest/largest of N 8 bit numbers.

Solution:

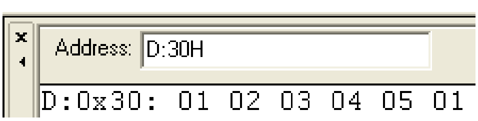
SMALLEST OF N NUMBER

```
      ORG  00H
      MOV  R3,#04H
      MOV  R0,#30H
      MOV  A,@R0
      INC  R0
UP:   MOV  B,@R0
        CJNE  A,B,NEXT
NEXT:   JC  CARRY
        MOV  A,@R0
CARRY:  INC  R0
        DJNZ  R3,UP
        MOV  @R0,A
        END
```

I/P



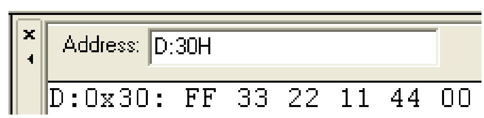
O/P



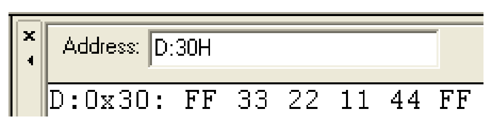
LARGEST OF N NUMBER

```
      ORG  00H
      MOV  R3,#04H
      MOV  R0,#30H
      MOV  A,@R0
      INC  R0
UP:   MOV  B,@R0
      CJNE  A,B,NEXT
NEXT:JNC  NOCARRY
          MOV  A,@R0
NOCARRY:  INC  R0
          DJNZ  R3,UP
          MOV  @R0,A
          END
```

I/P



O/P



Example 3.11:

Interfacing simple switch and LED to I/O ports to switch on/off LED with respect to switch status.

Solution:

- If SW is closed i.e. P0.0=0. Now Turn off the LED by making P2.0=1 (The cathode is high and turns off the LED)
- If SW is open i.e. P0.0=1. Now Turn on the LED by making P2.0=0 (The cathode is low and turns off the LED)

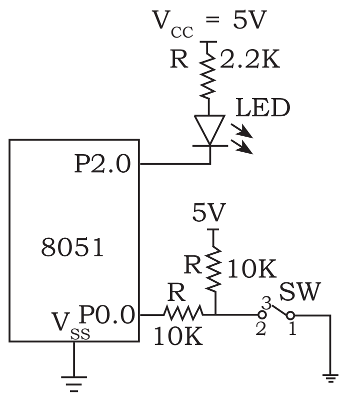


Fig.3.3.1: Shows Switch and LED interfacing to microcontroller.

ALGORITHM

1. Start
2. Make P0.0 as an input
3. Continuously monitor Switch status on pin P0.0
4. When switch is open circuit i.e. P0.0=1, Turn ON LED by clearing P2.0=0
5. Wait for some time (delay)
6. When switch is short circuit i.e. P0.0=0, Turn OFF LED by setting P2.0=1
7. Wait for some time (delay)
8. Go to step 2
9. End

<table>
<tr><th>Assembly Language Program</th><th>C Language Program
(Refer after reading Appendix 3)</th></tr>
<tr><td>

```
        ORG  00H
        SETB  P0.0

NEXT:   JB  P0.0,  ON
        SETB  P2.0
        ACALL  DELAY
        SJMP  NEXT
ON:     MOV  C,  0
        MOV  P2.0,  C
        ACALL  DELAY
        SJMP  NEXT
DELAY:
```

</td><td>

```
#include <reg51.h>
void delay(unsigned inti);
sbit led=P2^0;
sbit sw=P0^0;
void main(void)
{
        sw=1;  //make sw as an
                input
        while (1)
        {
            if (sw==1)
            led=0;
```

</td></tr>
</table>

Continued.....

```
        MOV  R0,  #255
UP2:    MOV  R1,  #255
UP1:    DJNZ  R1,  UP1
        DJNZ  R0,  UP2
        RET

        END
```

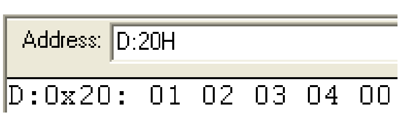
```
            delay(100);

            else
            led=1;
            delay(100);
            }
}

void delay(unsigned int count)
{
        unsigned int i, j;
        for(i=0;i<count;i++)
        for(j=0;j<1275;j++);
}
```

ASSEMBLY LANGUAGE PROGRAMS

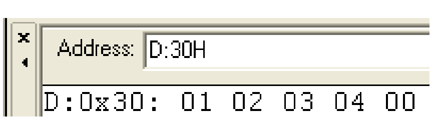
BLOCK MOVE

```
     ORG 00H
     MOV R2,#04H
     MOV R0,#20H
     MOV R1,#30H
UP:
     MOV A,@R0
     MOV @R1,A
     INC R0
     INC R1
     DJNZ R2,UP
     END
```

I/P



O/P



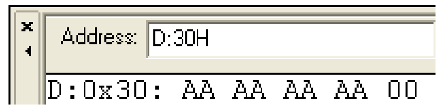
BLOCK EXCHANGE

```
     ORG 00H
     MOV R2,#04H
     MOV R0,#20H
     MOV R1,#30H
UP:
     MOV A,@R0
     XCH A,@R1
     MOV @R0,A
     INC R0
     INC R1
     DJNZ R2,UP
     END
```

I/P

Continued...

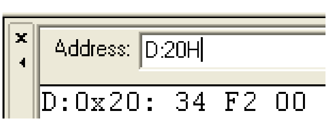
O/P

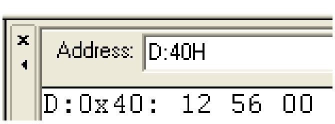


ADDITION OF TWO 16-BIT NUMBER (MULTIBYTE ADDITION)

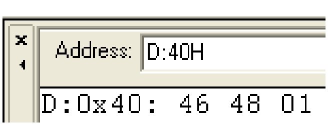
```
        ORG  00H
        MOV  R7,#02H
        MOV  R0,#20H
        MOV  R1,#40H
UP:
        MOV  A,@R0
        ADDC  A,@R1
        MOV  @R1,A
        INC  R0
        INC  R1
        DJNZ  R7,UP
        JNC  NOCARRY
        INC  R2
NOCARRY:  MOV  A,R2
        MOV  @R1,A
        END
```

I/P





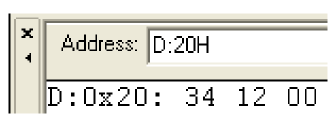
O/P

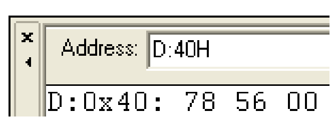


SUBTRACTION OF TWO 16-BIT NUMBER (MULTIBYTESUBTRACTION)

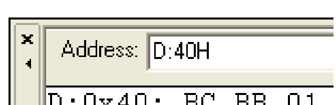
```
        ORG  00H
        MOV  R7,#02H
        MOV  R0,#20H
        MOV  R1,#40H
UP:
        MOV  A,@R0
        SUBB  A,@R1
        MOV  @R1,A
        INC  R0
        INC  R1
        DJNZ  R7,UP
        JNC  NOCARRY
        INC  R2
NOCARRY:  MOV  A,R2
        MOV  @R1,A
        END
```

I/P



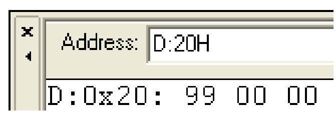
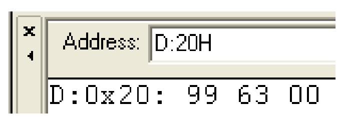


O/P



Hexadecimal to BCD	BCD to Hexadecimal

ORG 00H
MOV A,30H
MOV B,#0AH
DIV AB
MOV 33H,B
MOV B,#0AH
DIV AB
MOV 32H,B
MOV 31H,A
END
	ORG 00H
	MOV A,20H
	MOV B,#10H
	DIV AB
	MOV R2,B
	MOV B,#0AH
	MUL AB
	ADD A,R2
	MOV 21H,A
	END

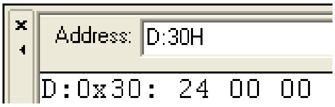
I/P  	**I/P** 
O/P  	**O/P** 

BCD to ASCII	ASCII to BCD

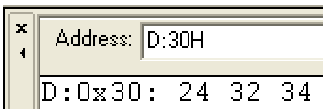
ORG 0000
MOV A,30H
ANL A,#0F0H
SWAP A
ADD A,#30H
MOV 31H,A
MOV A,30H
ANL A,#0FH
ADD A,#30H
MOV 32H,A
END
	ORG 0000H
	MOV A,30H
	SUBB A,#30H
	SWAP A
	MOV R2,A
	MOV A,31H
	SUBB A,#30H
	ADD A,R2
	MOV 32H,A
	END

Continued...

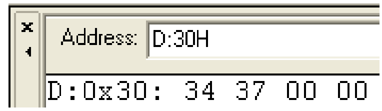
I/P



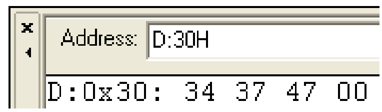
O/P



I/P



O/P



LOGIC GATES

AND, NAND, OR & NOR

```
ORG  00H
MOV  C,P1.7
ANL  C,P1.6
MOV  P1.5,C
CPL  C
MOV  P1.4,C
MOV  C,P1.7
ORL  C,P1.6
MOV  P1.3,C
CPL  C
MOV  P1.2,C
END
```

I/O:

A	B	Y=AB	Y=$\overline{AB}$	Y=A+B	Y=$\overline{(A+B)}$
0	0	0	1	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	0	1	0

HALF ADDER

```
ORG  00H
MOV  C,P1.7
ANL  C,/P1.6
MOV  P1.5,C
MOV  C,P1.7
CPL  C
ANL  C,P1.6
MOV  P1.4,C
ORL  C,P1.5
MOV  P1.3,C
MOV  C,P1.7
ANL  C,P1.6
MOV  P1.2,C
END
```

I/O:

A	B	C=AB	S=$\overline{A}B+A\overline{B}$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

XOR GATE

```
    ORG  00H
    MOV  C,P1.7
    ANL  C,/P1.6
    MOV  P1.5,C
    MOV  C,P1.7
    CPL  C
    ANL  C,P1.6
    MOV  P1.4,C
    ORL  C,P1.5
    MOV  P1.3,C
    END
```

I/O:

A	B	S=$\bar{A}B+A\bar{B}$
0	0	0
0	1	1
1	0	1
1	1	0

XOR NOR GATE

```
    ORG  00H
    MOV  C,P1.7
    CPL  C
    ANL  C,/P1.6
    MOV  P1.5,C
    MOV  C,P1.7
    ANL  C,P1.6
    MOV  P1.4,C
    ORL  C,P1.5
    MOV  P1.3,C
    END
```

I/O:

A	B	S=$\overline{AB}+AB$
0	0	0
0	1	1
1	0	1
1	1	0

ASCENDING

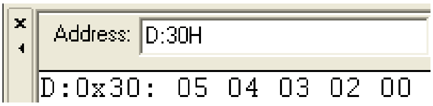
```
            ORG  00H
            MOV  R7,#03H
     MAIN:  MOV  R0,#30H
            MOV  R6,#03H
       UP:  MOV  A,@R0
            INC  R0
            MOV  B,@R0
            CJNE  A,B,NEXT
     NEXT:
            JC  NOEXCHANGE
            MOV  @R0,A
            DEC  R0
            MOV  @R0,B
            INC  R0
NOEXCHANGE:
            DJNZ  R6,UP
            DJNZ  R7,MAIN
            END
```

DESCENDING

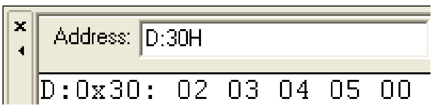
```
            ORG  00H
            MOV  R7,#03H
     MAIN:  MOV  R0,#30H
            MOV  R6,#03H
       UP:  MOV  A,@R0
            INC  R0
            MOV  B,@R0
            CJNE  A,B,NEXT
     NEXT:
            JNC  NOEXCHANGE
            MOV  @R0,A
            DEC  R0
            MOV  @R0,B
            INC  R0
NOEXCHANGE:
            DJNZ  R6,UP
            DJNZ  R7,MAIN
            END
```

Continued...

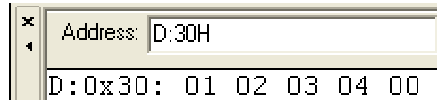
I/P



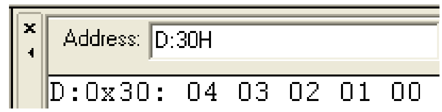
O/P



I/P



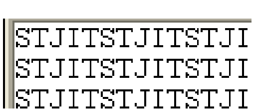
O/P



SERIAL COMMUNICATION

```
      ORG  00H
      MOV  TMOD,#20H
      MOV  TH1,#-3
      MOV  SCON,#50H
      SETB  TR1
UP:  MOV  A,  #'S'
      ACALL  SEND
      MOV  A,  #'T'
      ACALL  SEND
      MOV  A,  #'J'
      ACALL  SEND
      MOV  A,  #'I'
      ACALL  SEND
      MOV  A,  #'T'
      ACALL  SEND
      SJMP  UP

SEND:MOV  SBUF,  A
HERE:JNB  TI,  HERE
      CLR  TI
      RET
      END
```

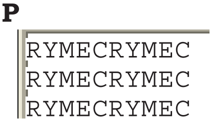
O/P



SERIAL COMMUNICATION

```
      ORG  00h
      MOV  TMOD,#20h
      MOV  TH1,#-3
      MOV  SCON,#50h
      SETB  TR1
REPEAT:
      MOV  DPTR,  #msg
UP:  CLR  A
      MOVC  A,@A+DPTR
      JZ  REPEAT
      ACALL  SEND
      INC  DPTR
      SJMP  UP
SEND:
      MOV  SBUF,A
HERE:JNB  TI,  HERE
      CLR  TI
      RET
msg: db  "RYMEC",0
      END
```

O/P



SERIAL COMMUNICATION
(Refer after reading Module-4)

```
          ORG  00H
          MOV  TMOD,  #20H
          MOV  TH1,  #-3
          MOV  SCON,  #50H
          SETB  TR1
UP:       MOV  A,  #'G'
          ACALL  SEND
          MOV  A,  #'M'
          ACALL  SEND
          MOV  A,  #'I'
          ACALL  SEND
          MOV  A,  #'T'
          ACALL  SEND
          SJMP  UP
SEND:     MOV  SBUF,  A
HERE:     JNB  TI,  HERE
          CLR  TI
          RET
          END
```

O/P

```
GMITGMIT
GMITGMIT
```

SERIAL COMMUNICATION
(Refer after reading Module-4)

```
          ORG  00h
          MOV  TMOD,#20h
          MOV  TH1,#-3
          MOV  SCON,#50h
          SETB  TR1
REPEAT:
          MOV  DPTR,  #msg
UP:       CLR  A
          MOVC  A,@A+DPTR
          JZ  REPEAT
          ACALL  SEND
          INC  DPTR
          SJMP  UP
SEND:
          MOV  SBUF,A
HERE:     JNB  TI,  HERE
          CLR  TI
          RET
          msg:  db  "GMIT",0
          END
```

O/P

```
GMITGMIT
GMITGMIT
```

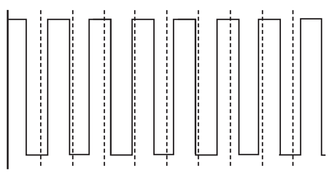
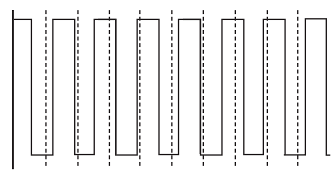
TIMER DELAY PROGRAM
(Refer after reading Module-4)

```
       ORG  00H
       MOV  TMOD,  #01H
AGAIN:
       MOV  TL0,#3EH
       MOV  TH0,#0B8H
       CPL  P1.7
       ACALL  DELAY
       SJMP  AGAIN
```

TIMER DELAY PROGRAM
(Refer after reading Module-4)

```
      ORG  00H
        MOV  TMOD,  #01H
AGAIN:
        MOV  TL0,#00H
        MOV  TH0,#00H
        CPL  P1.7
        ACALL  DELAY
        SJMP  AGAIN
```

Continued...

DELAY: SETB TR0 **HERE:** JNB TF0, **HERE** CLR TR0 CLR TF0 RET END **O/P** 	**DELAY:** SETB TR0 HERE: JNB TF0, **HERE** CLR TR0 CLR TF0 RET END **O/P** 

COUNTERS

HEX-UP COUNTER	**HEX-UP COUNTER (Refer after reading Module-4)**

ORG 00H
UP:
MOV P1,A
INC A
ACALL **DELAY**
SJMP **UP**
 	
DELAY:
MOV R0,#60H
MOV R1,#0FFH
MOV R2,#0FFH
BACK:
DJNZ R2,**BACK**
DJNZ R1,**BACK**
DJNZ R0,**BACK**
RET
END
	ORG 00H
	UP:
	MOV P1,A
	INC A
	ACALL **DELAY**
	SJMP **UP**
	
	DELAY:
	MOV TL0,#00H
	MOV TH0,#00H
	SETB TR0
	HERE: JNB TF0,**HERE**
	CLR TR0
	CLR TF0
	RET
	END

HEX-DOWN COUNTER (Refer after reading Module-4)	**HEX-DOWN COUNTER (Refer after reading Module-4)**

ORG 00H
MOV A,#0FFH
UP:
MOV P1,A
DEC A
ACALL **DELAY**
SJMP **UP**
 	
DELAY:
MOV R0,#60H
MOV R1,#0FFH
MOV R2,#0FFH
BACK:
DJNZ R2,**BACK**
DJNZ R1,**BACK**
DJNZ R0,**BACK**
RET
END
	ORG 00H
	MOV A,#0FFH
	UP:
	MOV P1,A
	DEC A
	ACALL **DELAY**
	SJMP **UP**
	
	DELAY:
	MOV TL0,#00H
	MOV TH0,#00H
	SETB TR0
	HERE: JNB TF0,**HERE**
	CLR TR0
	CLR TF0
	RET
	END

DECIMAL-UP COUNTER	**DECIMAL-UP COUNTER (Refer after reading Module-4)**

ORG 00H
UP:
MOV P1,A
ADD A,#01H
DAA
ACALL **DELAY**
SJMP UP
DELAY:
MOV R0,#60H
MOV R1,#0FFH
MOV R2,#0FFH
	ORG 00H
	UP:
	MOV P1,A
	ADD A,#01H
	DAA
	ACALL **DELAY**
	SJMP **UP**
	DELAY:
	MOV TL0,#00H
	MOV TH0,#00H
	SETB TR0

..... Continued

```
BACK:
        DJNZ  R2,BACK
        DJNZ  R1,BACK
        DJNZ  R0,BACK
        RET
        END
```

```
HERE:   JNB  TF0,HERE
        CLR  TR0
        CLR  TF0
        RET
        END
```

DECIMAL-DOWN COUNTER

```
        ORG  00H
        MOV  A,#99H
UP:
        MOV  P1,A
        ADD  A,#99H
        DAA
        ACALL  DELAY
        SJMP  UP
DELAY:
        MOV  R0,#60H
        MOV  R1,#0FFH
        MOV  R2,#0FFH
BACK:
        DJNZ  R2,BACK
        DJNZ  R1,BACK
        DJNZ  R0,BACK
        RET
        END
```

DECIMAL-DOWN COUNTER (Refer after reading Module-4)

```
        ORG  00H
        MOV  A,#99H
UP:
        MOV  P1,A
        ADD  A,#99H
        DAA
        ACALL  DELAY
        SJMP  UP
DELAY:
        MOV  TL0,#00H
        MOV  TH0,#00H
        SETB  TR0
HERE:   JNB  TF0,HERE
        CLR  TR0
        CLR  TF0
        RET
        END
```

SQUARE OF A GIVEN NUMBER

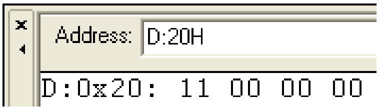
```
        ORG  00H
        MOV  A,20H
        MOV  B,A
        MUL  AB
        MOV  21H,B
        MOV  22H,A
        END
```

CUBE OF A GIVEN NUMBER

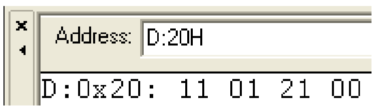
```
        ORG  00H
        MOV  A,20H
        MOV  B,A
        MUL  AB
        MOV  21H,A
        MOV  22H,B
        MOV  A,20H
```

..... Continued

I/P



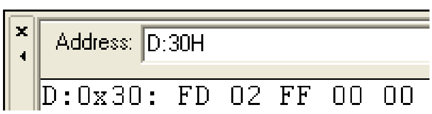
O/P



Cube I/O

I/P

O/P




```
        MOV  B,21H
        MUL  AB
        MOV  23H,A
        MOV  24H,B
        MOV  A,20H
        MOV  B,22H
        MUL  AB
        MOV  25H,A
        MOV  26H,B
        MOV  32H,23H
        MOV  A,24H
        ADD  A,25H
        MOV  31H,A
        MOV  A,26H
        ADDC  A,#00H
        MOV  30H,A
        END
```

16-BIT MULTIPLICATION

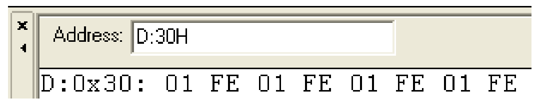
```
        ORG  00H
        MOV  A,20H
        MOV  B,22H
        MUL  AB
        MOV  30H,A
        MOV  31H,B
        MOV  A,20H
        MOV  B,23H
        MUL  AB
        MOV  32H,A
        MOV  33H,B
        MOV  A,21H
        MOV  B,22H
        MUL  AB
        MOV  34H,A
        MOV  35H,B
        MOV  A,21H
        MOV  B,23H
        MUL  AB
        MOV  36H,A
        MOV  37H,B
        MOV  43H,30H
        MOV  A,31H
        ADD  A,32H
        JNC  go
        INC  R0
go:     ADD  A,34H
        JNC  go1
        INC  R0
go1:    MOV  42H,A
        MOV  A,33H
        ADD  A,  R0
```

..... Continued

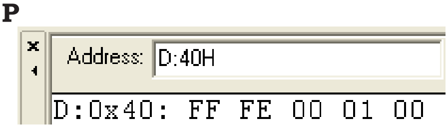
```
      JNC  go2
      INC  R1
go2:  ADD  A,35H
      JNC  go3
      INC  R1
go3:  ADD  A,36H
      JNC  go4
      INC  R1
go4:  MOV  41H,A
      MOV  A,37H
      ADD  A,R1
      MOV  40H,A
      END
```

I/P

Intermediate O/P



O/P



BYTE LEVEL LOGICAL OPERATION

```
      ORG  00H
      MOV  A,20H
      ANL  A,21H
      MOV  32H,A
      MOV  A,20H
      ORL  A,21H
      MOV  33H,A
      MOV  A,20H
      XRL  A,21H
      MOV  34H,A
      MOV  A,20H
      CPL  A
      MOV  35H,A
      MOV  A,20H
      CLR  A
      MOV  36H,A
      MOV  A,20H
      SWAP  A
      MOV  37H,A
      MOV  A,20H
      RR  A
      MOV  38H,A
      MOV  A,20H
      RL  A
      MOV  39H,A
      END
```

Appendix

A.1 Introduction to Embedded C and its applicability to 8051

- The use of C language to program microcontrollers is becoming too common and most of the time it's not easy to build an application in assembly which instead you can make easily in C. So it's important to know C language for microcontroller which is commonly known as **Embedded C**.
- The conventional 'C' language and its extensions are used for programming embedded systems, it is referred to as "Embedded C Programming".
- Embedded "C" can be considered as a subset of conventional "C" language.
- A software program called **"Cross compiler"** is used for the conversion of programs written in Embedded "C" to target processor / controller specific instructions.
- The "C" is the most common embedded language and almost 80% of embedded applications are coded in "C".
- With 8051 microcontroller we are going to use **Keil C51 Compiler**, hence we also call it **Keil C**.

Compiler

Compiler is a software tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture.

Cross Compiler

Cross compiler are software tools used in cross platform development applications. In cross platform development, the compiler running on a particular target processor / OS converts the source code to machine code for a target processor whose architecture and instruction set are different from the current development environment (OS).

Keywords

These are the reserved names used by the "C" language. All the keywords should be written in lowercase letters. ANSI "C" support 32 such keywords.

Examples: int, char, float, void, while, for, long etc.

Pros and Cons of using 8051 C

Advantages of programming in 'C' for 8051 microcontroller

1. Programming in 'C' is easier to write programs in 'C' than assembly language.
2. Programming in 'C' is easier to modify and update.

3. C code is portable to other microcontroller with little or no modification.
4. Programming in ‘C’ allows using library functions such as **sqrt**, **sine**, **scanf**, **printf** etc.
5. Programming in C” is less time consuming.

Disadvantages of programming in ‘C’ for 8051 microcontroller

1. Assembly language programs have fixed size for the HEX files produced, whereas for the same ‘C’ programs, different ‘C’ compiler produces different HEX code sizes.
2. The 8051 general purpose registers such as R0-R7, A & B are under the control of the ‘C’ compiler and are not accessed by ‘C’ statements.
3. It is difficult to calculate exact delay for ‘C’ programs.
4. Microcontroller has limited on-chip ROM and the code space. A misuse of data types by the programmer in writing ‘C’ programs can lead to a large size HEX files.

For example: Using int data type (16-bit) instead of unsigned char (8-bit) can lead to a large size HEX files. This problem does not arise in assembly language programs.

General structure of embedded C program

The first line in an 8051 C program is #include <reg51.h>. The library file reg51.h contains the definition of all the special function registers and their bits. Let us write a simple C program:

Write an 8051 C program to send values 00H -0FFH to port P1

```
#include <reg51.h>        // contains all the port registers and internal
                          // RAM of 8051declared
void main (void)
 {
   unsigned char x;       // x is allotted a space of 1 byte
   for (x=0; x<=255;x++)
   P1=x;                  // values 00H-0FFH sent to port 1
}
```

The values **00H to 0FFH** will be displayed on **port 1.**

A.2 Data Types

The data types of 8051 are

1. **unsigned char**
    - Since 8051 is an 8-bit microcontroller and the character data type is also **8-bit**. So char data type is **most widely** used in 8051 C.
    - The unsigned char is an 8-bit data type that takes a value in the range of **0 to 255** i.e. 00H to FFH.
    - Used for setting a **counter value**, to represent **ASCII character** etc.
2. **signed char**
    - The signed char is an 8-bit data type that uses the Most Significant Bit (MSB) $\mathbf{D_7}$ to represent **+ve** or **–ve** value.
    - So only **7-bits** are used to represent the **magnitude** of the number.
    - The signed char ranges from **-128 to +127**.
    - Used to represent quantity having –ve values such as **Temperature** etc.
3. **unsigned int**
    - The unsigned int is a **16-bit data type** that ranges from **0 to 65635** (0000H to FFFFH).
    - The unsigned int is used to define a 16-bit variables such as memory addresses. It is also used to **set counter** values of more than 256.
    - Since 8051 is an 8-bit microcontroller so the int data type takes **two bytes of RAM**. The **misuse** of **int** variables will result in a **larger HEX file**.
4. **signed int**
    - The signed int is a 16-bit data type that uses the MSB i.e. $\mathbf{D_{15}}$ bit to represent +ve or –ve value.
    - So, only **15-bits** are used to represent the **magnitude** of the number.
    - The signed int ranges from **-32,768 to +32,767**.
    - Used to represent **+ve** or **– ve** values.
5. **sbit**
    - The **sbit** keyword is used to access **single bit addressable registers**.
    - It allows access to the **single bit** of the **SFR registers**.
    - Its size is 1-bit i.e. either **0** or **1**
6. **bit**
    - The bit data type allows to access **single bit** of **bit-addressable memory** spaces **20H to 2FH**.
    - Its size is 1-bit i.e. either **0** or **1**.

7. sfr

- The sfr data type is used to access the **byte-size SFR registers**.
- Its size is **8-bits** i.e. 1 byte.
- RAM addresses 80H to FFH are used.

Note: The C compilers use the **signed char** and **signed int** as the **default** if we do not put the keyword unsigned in front of the char and int.

Table A.1: Data types in 8051

Data Type	Size in Bits	Data Range/Usage
unsiged char	8-bit	0 to 255
signed char	8-bit	–128 to +127
unsigned int	16-bit	Oto 65535
signed int	16-bit	–32,768 to+32,767
sbit	1-bit	SFR bit-addressable only
bit	1-bit	RAM bit-addressable only
sfr	8-bit	RAM addresses 80–FFH only

C Programs

unsigned char

Example A.1:

Write an 8051 C program to send values 00H - 0BBH to port P3

Solution:

```
#include <reg51.h>
void main (void)
  {
     unsigned char i;
     for (i=0; i<0xBB;i++)
     P3=i;
}
```

1. Pay careful attention to the size of the data
2. Try to use unsigned char instead of *int* if possible

Example A.2:

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

Solution:

```
#include <reg51.h>
void main(void)
{
unsigned char mynums[]="012345ABCDF";
unsigned char i;
for (i=0;i<11;i++)
P1=mynums[i];
}
```

Example A.3:

Write an 8051 C program to toggle all the bits of P2 continuously.

Solution:

```
//Toggle P2 forever
#include <reg51.h>
void main(void)
{   for (;;)
    {
        P2=0xAA;
        P2=0x55;
    }
}
```

signed char

Example A.4:

Write an 8051 C program to send values of –3 to +3 to port P1.

Solution:

```
#include <reg51.h>
void main(void)
{
charmynums[]={+1,-1,+2,-2,+3,-3};
unsigned char i;
```

```
for(i=0;i<6;i++)
P1=mynums[i];
}
```

Note: The negative values will be displayed in the 2's complement for as –1 = FFH, –2 = FEH, –3 = FDH, –4 = FCH and so on.

unsigned int

Example A.5:

Write an 8051 C program to toggle bit D0 of the port P0 (i.e. P0.0) 50,000 times.

Solution:

```
#include <reg51.h>
sbit MYBIT=P0^0;
void main(void)
    {
     unsigned int i;
     for (i=0;i<50000;i++)
         {
         MYBIT=0;
         MYBIT=1;
         }
     }
```

Example A.6:

Write in 8051 C program to toggle bits of PI continuously forever with some delay.

Solution:

```
# include <reg51.h>
void main (void)
{
unsigned int x;
for (;;)
   {
      Pl=0x55;
      for (x=0;x<40000;x++) ; //time delay
      Pl=0xAA;
      for (x=0;x<40000;x++);
```

```
    }
}
```

signed int

Example A.7:

Write an 8051 C program to send values of –5 to +5 to port P1.

Solution:

```
#include <reg51.h>
void main(void)
{
int mynums[]={+1,-1,+2,-2,+3,-3,+4,-4,+5,-5};
unsigned char i;
for(i=0;i<10;x++)
P1=mynums[i];
}
```

The negative values will be displayed in the 2's complement.

single bit

Example A.8:

Write an 8051 C program to toggle bit D0 of the port P0 (P0.0) 50,000 times.

Solution:

```
#include <reg51.h>

sbit MYBIT=P0^0;
void main(void)
{
        unsigned int z;
        for (i=0;i<50000;i++)
        {
            MYBIT=0;
            MYBIT=1;
        }
}
```

sbit keyword allows access to the single bits of the SFR registers

Example A.9:

Write an 8051 C program to toggle only bit P0.1 continuously without disturbing the rest of the bits of P0.

Solution:

```
#include <reg51.h>
sbit mybit=P0^1;
void main(void)
{
    while (1)
    {
        mybit =0;         // turn on P0.1
        mybit =1;         // turn off P0.1
    }
}
```

A.3 Time Delay generation

There are two ways to create a time delay in 8051 C:

1. **Using a simple for loop**
2. **Using the 8051 timers**

Time-delay generation using loops

To create time delay,three factors that can affect the accuracy of the delay

1. The instruction execution speed varies according to the number of clock periods per machine cycle i.e. different versions of microcontroller uses different machine cycles to execute instructions.

   Example: The 8051 uses 12 clock periods per machine cycle and newer generation microcontrollers uses fewer clocks per machine cycle.
2. The crystal frequency connected to the XTAL1 and XTAL2 input pins.
3. Compiler Selection.

   C compiler converts the C statements and functions to assembly language instructions. Different compilers produce different code.

Note:

- In assembly language programming, delay generated can be controlled by the user, as the number of instructions and the cycles per instructions are known.
- In case of C program, the C compiler will convert the C statements and functions

to assembly language instructions. Thus, different compilers produce different delays.

Machine cycles and clock frequency for 8051

In 8051, time for one machine cycle is 12 oscillator periods i.e. 12d.

The frequency of the crystal connected to the 8051 family can vary from 4 MHz to 30 MHz. But, the 11.0592 MHz crystal oscillator is used to make the 8051 based system compatible with the serial port.

Formulae

1. $\text{Clock Period} = \dfrac{1}{\text{Clock frequency}} = \dfrac{1}{11.0592 \times 10^6} = 0.090\,\mu s$

2. Time of one machine cycle

   Clock period × 12d = 0.090 μs × 12 = 1.085 μs

3. Time delay provided for one machine cycle

   Number of Machine cycles × Time for one machine cycle = 1 × 1.085 μs = 1.085 μs

Note:

1. The for loop i.e. for(i=0; i<1275; i++) executed on **8051** microcontroller (**12 clock per machine cycle**) with a standard crystal frequency of **11.0592 MHz** produces a time delay of approximately **12 ms**.
2. The for loop i.e. for(i=0; i<1275; i++) executed on **DS89C420** microcontroller (**1 clockper machine cycle**) with a standard crystal frequency of **11.0592 MHz** produces a time delay of approximately **1ms**.

Example A.10:

The 8051 microcontroller uses a clock frequency of 11.0592 MHz. Calculate

i) Clock period 'T'

ii) Time of one machine cycle.

iii) Time delay provided by AAH machine cycles.

Solution:

i) Clock period 'T' = $\dfrac{1}{\text{Clock frequency}} = \dfrac{1}{11.0592 \times 10^6} =$ **0.09 μs**

ii) Time of one machine cycle = 0.09 μs × 12 = **1.085 μs**

iii) Time delay provided by **AAH** = $\mathbf{(170)_{10}}$ machine cycle is

1.085 μs × 170 machine cycles = **184.45 μs**

Example A.11:

Write an 8051 C program to toggle bits of P0 continuously forever with some delay.

Solution:

//Toggle P0 forever with some delay in between "on" and "off"

```
#include <reg51.h>
void main(void)
{
unsigned int x;
for (;;)                        //repeat forever or while (1)
{
    P0=0x55;
    for (x=0;x<1275;x++);       //1 ms delay
    P0=0xAA;
for (x=0;x<1275;x++);           //1 ms delay
    }
}
```

Example A.12:

Write a 8051 C program to toggle all the bits of P0 and P1 continuously with a 1 ms delay. Use XTAL = 11.0592 MHz.

Solution:

```
#include <reg51.h>
voidmain(void)
{
    unsigned int x;
    while(1)                    //repeat forever
    {
        P0=0x55;
        P1=0x55;
        for(x=0;x<1275;x++);    //1ms delay
```

```
        P0=0xAA;
        P1=0xAA;
        for (x=0;x<1275;x++);   //1ms delay
    }
}
```

Example A.13:

Write a 8051 C program to generate a square wave of 50% duty cycle with T_{on} = T_{off} = 500 ms.

Solution:

$$\text{Duty cycle} = \frac{T_{ON}}{T_{ON} + T_{OFF}}$$

For 50% duty cycle, $T_{ON} = T_{OFF}$ = 500 ms

C Program

```
#include<reg51.h>
void delay(unsigned int);
void main( )
{while (1)          //repeat continuously
    {
        P0^1=0;     //make P0.1 pin low
        delay(500); //call delay subroutine with a parameter of 500
        P0^1=1;     //make P0.1 pin high
        delay(500); //call delay subroutine with a parameter of 500
    }               //end of while
}                   //end of main
void delay (unsigned int count)
{
    unsigned int i, j;
    for(i=0; i<count; i++)      //outer loop repeated count times
        for(j=0; j<1275; j++); //inner loop for 1 ms delay
}
```

Example A.14:

Write a 8051 C program to generate a square wave of 75% duty cycle with T =400 ms on pin P0.4.

Solution:

T = 400 ms

Duty cycle = 75% = 0.75

$$\text{Duty cycle} = \frac{T_{on}}{T} = \frac{T_{on}}{400\text{ms}}$$

T_{on} =400 × 0.75 = 300 ms

T_{off} = T – T_{on} = 400 – 300 = 100 ms.

C Program

```
include <reg51.h>
void delay (unsigned int);
void main
{ while(l)
    {
    P0^4=1;
    delay(300);
    P0^4=0;
    delay(l00);
    }
}
void delay (unsigned int count)
{
    unsigned int i,j;
    for(i=0; i<count; i++)
        for(j=0;j<1275;j++);
}
```

Example A.15:

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

Solution:

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

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

Example A.16:

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

Solution:

```
#include <reg51.h>
void delay(unsigned int);
void main(void)
{
    unsigned char mydata;
    P1=0xFF;                   //make P1 input port
    while (1)                  // repeat forever
    {
        mydata=P1;             //get a byte from P1
        delay(500);
        P2=mydata;             //send it to P2
    }
```

```
}
void delay(unsigned int count)
{
    unsigned int i, j;
    for (i=0; i<count; i++)
        for (j=0; j<1275; j++);
}
```

A.4 Accessing SFRs and bit addressable RAM

- Another way to access the **SFR RAM** space **80H to FFH** is to use the **sfr data type**. When this data type is used **no need** of using the **header file reg51.h**.
- Also, we can access a **single bit** of any **SFR** by specifying the **bit address** as shown in table A.2.
- The bit and byte addresses for the P0 to P3 ports are given in the table A.2

Table A.2: single bit addresses of ports.

P0	Addr	P1	Addr	P2	Addr	P3	Addr	Port's Bit
P0.0	80H	P1.0	90H	P2.0	A0H	P3.0	B0H	D0
P0.1	81H	P1.1	91H	P2.1	A1H	P3.1	B1H	D1
P0.2	82H	P1.2	92H	P2.2	A2H	P3.2	B2H	D2.
P0.3	83H	P1.3	93H	P2.3	A3H	P3.3	B3H	D3
P0.4	84H	P1.4	94H	P2.4	A4H	P3.4	B4H	D4.
P0.5	85H	P1.5	95H	P2.5	A5H	P3.5	B5H	D5
P0.6	86H	P1.6	96H	P2.6	A6H	P3.6	B6H	D6
P0.7	87H	P1.7	97H	P2.7	A7H	P3.7	B7H	D7

Example A.17:

Write an 8051 C program to toggle all the bits of P0, P1 and P2 continuously with a 250 ms delay. Use the sfr keyword to declare the port addresses.

Solution:

//Accessing Ports as SFRs using sfr data type

```
sfr P0=0x80;
sfr P1=0x90;
sfr P2=0xA0;
void delay(unsigned int);
void main(void)
{
    while (1)
    {
        P0=0x55;
        P1=0x55;
        P2=0x55;
        delay(250);
        P0=0xAA;
        P1=0xAA;
        P2=0xAA;
        delay(250);
    }
}
void delay(unsigned int count)
{
unsigned inti, j;
for (i=0; i<count; i++)
    for (j=0; j<1275; j++);
}
```

Another way to access the SFR RAM space 80 - FFH is to use the sfr data type

Example A.18:

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 i;
    for (i=0;i<50000;i++)
    {
        MYBIT=1;
        MYBIT=0;
    }
}
```

We can access a single bit of any SFR if we specify the bit address

Notice that there is no #include <reg51.h>. This allows us to access any byte of the SFR RAM space 80-FFH. This is widely used for the new generation of 8051 microcontrollers.

bit addressable I/O

Example A.19:

Write an 8051 C program to toggle only bit P2.5 continuously without disturbing the rest of the bits of P2.

Solution:

//Toggling an individual bit

```
#include <reg51.h>

sbit mybit=P2^5;
void main(void)
{
while (1)
{
    mybit=1;     //turn on P2.5
    mybit=0;     //turn off P2.5
}
}
```

Ports P0-P3 arc bit-addressable and we use *sbit* data type to access a single bit of P0-P3.

Use the Px^y format, where x is the port 0, 1,2, or 3 and y is the bit 0 - 7 of that port

Example A.20:

Write an 8051 C program to monitor bit P0.5. If it is high, send AAH to P1; otherwise, send FFH to P3.

Solution:

```
#include <reg51.h>
sbit mybit=P0^5;
void main(void)
{
    mybit=1;        //make mybit an input
    while (1)
    {
```

```
            if (mybit==1)
            P1=0xAA;
            else
            P3=0xFF;
        }
    }
```

Example A.21:

Write an 8051 C program to get the status of bit P1.1, save it, and send it to P0.1 continuously.

Solution:

```
#include <reg51.h>
sbit inbit=P1^1;
sbit outbit=P0^1;
bit membit;                          //use bit to declare
                                //bit- addressable memory
void main(void)
{
    while (1)
    {
        membit=inbit;       //get a bit from P1.1
        outbit=membit;      //send it to P0.1
    }
}
```

We use bit data type to access data in a bit-addressable section of the data RAM space 20 - 2FH

Example A.21:

The data pins of an LCD are connected to P1. The information is latched into the LCD whenever its Enable pin goes from high to low. Write an 8051 C program to send "RYMEC Engineering College Ballari" to this LCD.

Solution:

```
#include <reg51.h>
#define LCDData P1        //LCDData declaration
sbit En=P2^0;             //the enable pin
void main(void)
{
    unsigned char message[]= "RYMEC Engineering College Ballari";
```

```
    unsigned char z;
    for (z=0;z<33;z++)    //send 33 characters
    {
        LCDData=message[z];
        En=1;             //a high-
        En=0;             //-to-low pulse to latch data
    }
}
```

A.5 Arithmetic and Logical Operators

Arithmetic operators

The arithmetic operators in 8051 C are Additioin (+), Subtraction (-), Multiplication (*) and Division (/) and are shown in table A.3

Table A.3 Arithmetic Operators in C

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = 10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	A / B = 2

Note: A=20 & B=10

Example A.22:

Write a C program to understand all the arithmetic operators available in 8051 C.

Solution:

```
#include <reg51.h>
void main(void)
{
    unsigned char A=20;
    unsigned char B=10;
    P1=A + B;
    P1=A - B;
    P1=A * B;
    P1=A / B;
}
```

Bit-wise Logic Operators in C

❖ ***Explain the different logical operators available in 8051C.*** **5 Marks**

The 8051 C language also has several bitwise operators. The Bitwise operators affect a variable on a bit-by-bit basis. These bit-wise operators are widely used in software engineering for embedded systems and control.

Table A.3: 8051 C Bit-wise Logic Operators

Operator	Description
&	Bitwise AND
|	Bitwise OR
~	Bitwise NOT (1 s Compliment)
^	Bitwise Exclusive OR
<<	Shift Left
>>	Shift Right

Table A.4: Bit-wise Logic Operation on bit variables.

		AND	OR	EX-OR	Inverter
A	B	A&B	A | B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	–
1	1	1	1	0	–

The following shows some examples using the C logical operators.

1. 0x35 & 0x0F = 0x05 /* ANDing */

	0	0	1	1	0	1	0	1
&	0	0	0	0	1	1	1	1
	0	**0**	**0**	**0**	**0**	**1**	**0**	**1**

Result is 00000101b = 05H

2. 0x04 | 0x68 = 0x6C /* ORing */
3. 0x04 ^ 0x78 = 0x2C /* XORing */
4. ~0xAA = 0x55 /* Inverting AAH */

The bitwise operators are used to

i) Turning bits ON: Turn ON a particular bit by **ORing** with a **1**.

ii) Turning bits OFF: Turn OFF a particular bit by **ANDing** with a **0**.

iii)Toggling bits: Turning a bit OFF to ON or ON to OFF by **EXCLUSIVELY ORing** with a **1**.

Bit-wise shift operators in C

There are two bit-wise shift operators in 8051 C: shift right (>>) and shift left (<<).

The formats of bit-wise shift operators are as follows

data >> number of bits to be shifted right

data << number of bits to be shifted left

The following shows some examples of shift operators in 8051 C

1. 0x9A >> 3 = 0x13 /* shifting right 3 times */
2. 0x66 >> 4 = 0X06 /* shifting right 4 times */
3. 0x05 << 4 = 0x50 /* shifting left 4 times */

Example A.23:

The following program will explain the different logical operations. We can run on 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
}
```

Example A.24:

Write an 8051 C program to toggle all the bits of P0 and P1 continuously with a 300ms delay. Using the inverting and Ex-OR operators, respectively.

Solution:

```
#include <reg51.h>
void delay(unsigned int);
void main(void)
 {
   P0=0x55;
   P1=0x55;
   while (1)
   {
       P0=~P0;
       P1=P1^0xFF;
       delay(300);
   }
}
void delay(unsigned int count)
   {
       unsigned int i,j;
       for(i=0; i<count; i++)
           for(j=0; j<1275;j++);
   }
```

Example A.25:

Write an 8051 C program to get bit P1.2 and send it to P2.3 after inverting it.

Write a C program to read P1.2 and send it to P2.3 after inverting it.

5 Marks

Solution:

```
#include <reg51.h>
sbit inbit=P1^2;
sbit outbit=P2^3;
bit membit;
void main(void)
{
   while (1)
```

```
    {
        membit=inbit;           //get a bit from P1.2
        outbit=~membit;         //invert it and send
                                //it to P2.3
    }
}
```

Example A.26:

Write an 8051 C program to read the P1.0 and P1.1 bits and issue an ASCII character to P0 that is if P1.1 and P1.0 is 00 send '0' if 01 send '1', if 10 send '2' and if 11 send '3'.

Algorithm

1. Make P1 as input port.
2. Read P1 value.
3. Mask all bits except D0 & D1 of P1 and put the masked value in x.
4. If x=0; send '0' to P0, else if x=1; send '1' to P0, else if x=2; send '2' to P0, else send '3' to P0. (use switch statement).
5. Repeat from step 2.

Solution:

```
#include <reg51.h>
void main(void)
{
    unsigned char z;
    z=P1;                       //read P1
    z=z&0x3;                    //mask the unused bits
    switch (z)                  //make decision
    {
        case(0):
        {
            P0='0'; break;      //send ASCII 0
        }
    case(1):
```

```
{
    P0='1'; break;    //send ASCII 1
}
case(2):
{
    P0='2'; break;    //send ASCII 2
}
case(3):
{
    P0='3'; break;    //send ASCII 3
}
}}
```

Accessing Code ROM Space in 8051 C

Using ROM to Store Data

- To make C compiler use the code space (on-chip ROM) instead of **RAM** space, we can put the keyword "**code**" in front of the variable declaration.

```
unsigned char mydata[] = "HELLO"
```

  - HELLO is saved in RAM.

```
code unsigned char mydata[] = "HELLO"
```

  - HELLO is saved in ROM.

Example A.27:

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[]="RYMEC";          //RAM space
    unsigned char z;
    for (z=0;z<=5;z++)
    P1=mynum[z];
}
```

Example A.28:

Write, compile and single-step the following program on your 8051simulator. 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, i=0;
    for (x=0;x<100;x++)
    {   i++;
        mydata [x]=i;
        P0=i;
    }
}
```

Example A.29:

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

Solution:

To make the C compiler use the code space instead of the RAM space, we need to put the keyword code in front of the variable declaration

```
#include <reg51.h>
void main(void)
{
    code unsigned char mynum[]="GMIT DAVANGERE";
    unsigned char i;
    for (i=0;i<14;i++)
    P1=mynum[i];
}
```

Example A.30:

Compare and contrast the following programs and discuss the advantages and disadvantages of each one.

i)

```
#include <reg51.h>
Void main (void)
{
```

```
P1 = "H";
P1 = "E";
P1 = "L";
P1 = "L";
P1 = "O";
}
```

Solution:

Short and simple.

The individual character is embedded into the program and it mixes the code and data together. Thus, it's not flexible.

ii)

```
#include <reg51.h>
Void main (void)
{
    unsigned char mydata [ ] = "HELLO";
    unsigned char z;
    for (z=0; z<5; z++)
    P1 = mydata [z];
}
```

Solution:

The Array elements are stored in RAM, therefore the size of the array is limited.

iii)

```
#include <reg51.h>
Void main (void)
{
        code unsigned char mydata [ ] = "HELLO";
        unsigned char z;
        for (z=0; z<5; z++)
        P1 = mydata [z];
}
```

Solution:

Use a separate area of the code space for data. This allows the size of the array to be as long as you want if you have the on-chip ROM.

However, the more code space you use for data, the less space is left for your program code.

Example A.31:

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

❖ ***Write a 8051 C program to convert packed BCD to ASCII and to display it on P1 and P2*** ***5 Marks***

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;
}
```

Example A.32:

Write an 8051 C program to convert ASCII digits '4' and '7' into packed BCD and to display on port P1. ***5 Marks***

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;
}
```

Example A.33:

Write an 8051 C program to toggle all the bits of P0 for every 500ms

i) by using NOT operator

ii) by using EX-OR operator

Solution

i) By using NOT operator

```
#include <reg51.h>
void delay(unsigned int);
void main(void)
{
    P0=0x55;
    while(1)
    {
        P0=~P0;
        delay(500);
    }
}
void delay(unsigned int itime)
{
    unsigned int i,j;
    for (i=0;i<itime;i++)
        for (j=0;j<1275;j++);
}
```

ii) By using EX-OR operator

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

```
        delay(500);
    }
}

void delay(unsigned int itime)
{
    unsigned int i,j;
    for (i=0;i<itime;i++)
        for (j=0;j<1275;j++);
}
```

4

8051 Timers and Serial Port

4.1 Introduction

The 8051 has two**16-bit** timers/counters, they can be used either as

- **Timers** to generate a time delay or
- **Event counters** to count events happening outside the microcontroller.

The two timers are

i) Timer/Counter **T0** and ii) Timer/Counter **T1**

Fig.4.1.1: Timer 0 & Timer 1

- Each register can be used either for Timer or counter and can be divided into Two 8-bit registers called Timer Low (TL) and Timer High (TH) as shown in Fig. 4.1.1.
- Both timers 0 and 1 use the same register, called **TMOD** (timer mode), to set the various timer operation modes.
- **TCON** is a bit-addressable 8-bit register used for **timer control**.

4.1.1 Timer 0 register

TH0								TL0							
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Fig.4.1.2: Timer 0 register

- The 16-bit register of Timer 0 is accessed as low byte and high byte.
- The low byte register is called TL0 and the high byte register is called TH0.
- These registeres can be accessed like any other registers such as A, B, R0 to R7 etc.

Example:

```
MOV TLO, # 55H     ; Move the value 55H into TL0 register
MOV R1, TL0        ; Copy the content of TL0 into R1 register.
```

4.1.2 Timer 1 register

TH1								TL1							
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Fig.4.1.3: Timer 1 register

- The 16-bit register of Timer 1 is accessed as low byte and high byte.
- The low byte register is called TL1 and the high byte register is called TH1.
- These registeres can be accessed like any other registers such as A, B, R0 to R7 etc.

Example:

```
MOV TH1, # 55H     ; Move the value 55H into TH1 register
MOV R1, TH1        ; Copy the content of TH1 into R1 register.
```

4.2 TMOD (Timer Mode) Register

- Timer 0 & Timer 1 use the same register, called TMOD, to set the various timer operation modes.
- TMOD is an 8-bit register in which, the lower 4 bits are for Timer 0 and the upper 4 bits are for Timer 1.

(MSB) (LSB)

GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
Timer 1				Timer 0			

Fig.4.2.1: TMOD register

For Timer 1

Bit 7: **Gate (Gating control)**

- When **Gate=1**, the Timer/Counter 1 is enabled only when $\overline{\text{INT1}}$ pin is high (p 3.3) and the TR1 control bit is high (i.e. **Gate** = $\overline{\text{INT1}}$ = **TRI = 1**).

- When **Gate=0**, theTimer/Counter 1 is enabled whenever the TR1 control bit is set (i.e. **Gate=TR1=1** & regardless of the State of $\overline{INT1}$ pin).

Bit 6: **C/$\overline{T}$ (Timer or Counter selected)**

- When **C/$\overline{T}$ = 0**, **Timer mode** selected.In Timer mode, Timer 1 will increment every machine cycle.
- When **C/$\overline{T}$ = 1**,**counter mode** selected.In counter mode, Timer1 will count events (pulses) on T1 pin (P3.5).

Bit 5 & 4:**M1 & M0 (Mode bit 1 & Mode bit 0)**

T1M1	T1M0	Mode	Descriptioni
0	0	0	13 - bit timer
0	1	1	16 - bit timer
1	0	2	8 - bit auto reload
1	1	3	Split mode

For Timer 0

Bit 3:Gate (Gating control)

- When **Gate = 1**, the Timer/Counter 0 is enabled only when $\overline{INT0}$ pin is high (p3.2) and the TR1 control bit is high (i.e. **Gate =** $\overline{INT0}$ **=TR0 = 1**).
- When **Gate =0**, the Timer/Counter 0 is enabled whenever the TR0 control bit is set (i.e. **Gate=TR0 = 1** & regardless of the State of $\overline{INT0}$ pin).

Bit 2: **C/$\overline{T}$ (Timer or Counter selected)**

- When **C/$\overline{T}$ = 0**, **Timer mode** selected. In Timer mode, Timer 0 will increment every machine cycle.
- When **C/$\overline{T}$ =1**, **counter mode** selected. In counter mode, Timer 0 will count events (pulses) on T0 pin (P3.4).

Bit-1 & 0: M1 & M0 (Mode bit 1 & Mode bit 0)

T1M1	T1M0	Mode	Descriptioni
0	0	0	13 - bit timer
0	1	1	16 - bit timer
1	0	2	8 - bit auto reload
1	1	3	Split mode

Note:

- The only difference between Timer/counter is the sources of the clock pulses. When used as a timer, the clock pulses are sourced from the oscillator through the divide by 12-d circuit.
- When used as a counter, Pin T0 (P3.4) supplies pulses to counter 0, & Pin T1 (P3.5) supplies pulses to counter1.

Note:

- TMOD register configuration for Timer 0/1 in Mode 1 & Mode 2.

Table 4.2.1: TMOD register configuration

Timer & Mode	TMOD Register	TMOD value
Timer 0, Mode 1	TIMER 1: GATE C/$\overline{T}$ M_1 M_0 = × × × × TIMER 0: GATE C/$\overline{T}$ M_1 M_0 = 0 0 0 1 Timer Mode 1	01H
Timer 1, Mode 1	TIMER 1: GATE C/$\overline{T}$ M_1 M_0 = 0 0 0 1 TIMER 0: GATE C/$\overline{T}$ M_1 M_0 = × × × × Timer Mode 1	10H
Timer 0, Mode 2	TIMER 1: GATE C/$\overline{T}$ M_1 M_0 = × × × × TIMER 0: GATE C/$\overline{T}$ M_1 M_0 = 0 0 1 0 Timer operation Mode 2 of operation	02H
Timer 1, Mode 2	TIMER 1: GATE C/$\overline{T}$ M_1 M_0 = 0 0 1 0 TIMER 0: GATE C/$\overline{T}$ M_1 M_0 = × × × × Timer Mode 2 of operation	20H

Example 4.1:

Find the values of TMOD to operate as timers in the following modes.

(a) Mode 1 Timer 1 (b) Mode 2 Timer 0, Mode 2 Timer 1

(c) Mode 0 Timer 1

Solution

(a) TMOD is 00010000 = 10H

The gate control bit and C/$\overline{T}$ bit are made 0, and the unused timer (Timer 0 bit is also 0)

(b) TMOD is 01010010 = 52H

(c) TMOD is 00000000H = 00H

Example 4.2:

Indicate which mode and which timer are selected for each of the following

(a) MOV TMOD, #01H (b) MOV TMOD, #20H (c) MOV TMOD, #12H

Solution:

We convert the value from hex to binary

(a) TMOD = 00000001, mode 1 of timer 0 is selected.

(b) TMOD = 00100000, mode 2 of timer 1 is selected.

(c) TMOD = 00010010, mode 2 of timer 0, and mode 1 of timer 1 are selected.

Example 4.3:

Find the timer's clock frequency and its period for various 8051-based system, with the crystal frequency 11.0592 MHz when C/$\overline{T}$ bit of TMOD is 0.

Solution:

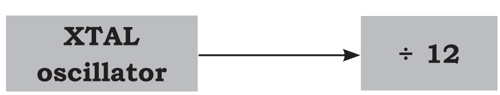


Fig. 4.2.2: Divide by 12 circuit

We know that in 8051, XTAL oscillator frequency is divide by 12 circuit as shown in figure 5.2.2.

$$\text{Frequency f} = \frac{1}{12} \times 11.0592\ \text{MHz} = 921.6\ \text{KHz}$$

$$\text{Time T} = \frac{1}{f} = \frac{1}{921.6\ \text{KHz}} = 1.085\ \mu\text{s}$$

Machine Cycle

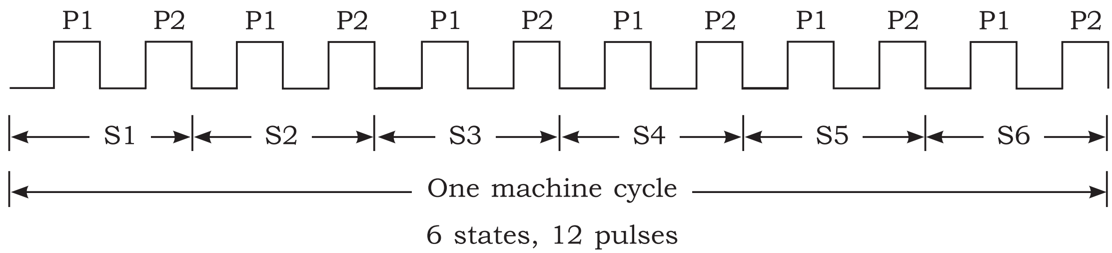


Fig. 4.2.3: Machine cycle

In 8051, **two pulses** constitute a **state** and **machine cycle** is made up of **six states**. Some instructions may require more than one machine cycle.
The time required to execute an instruction is given by

$$T = \frac{C \times 12d}{f}$$

Where, **T** is the time for instruction to be executed
f is the crystal frequency and
C is the number of machine cycles.

Example 4.4:

For 8051 microcontroller, find the time taken for an instructions which takes
i) 1 Machine cycle ii) 2 Machine cycles iii) 4 Machine cycles

Solution:

i) $T = \frac{C \times 12d}{f} = \frac{1 \times 12}{11.0592 \times 10^6} = 1.085\ \mu Sec$

ii) $T = \frac{C \times 12d}{f} = \frac{2 \times 12}{11.0592 \times 10^6} = 2.170\ \mu Sec$

iii) $T = \frac{C \times 12d}{f} = \frac{4 \times 12}{11.0592 \times 10^6} = 4.340\ \mu Sec$

4.2.1 TCON Register (Timer Control Register)

MSB LSB

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Fig.4.2.4:TCON register.

- TCON (timer control) register is a bit-addressable 8-bit register.
- The upper four bits (bit 4 to bit 7) are used to store the TF and TR bits of both timer 0 & 1.

- The lower four bits (bit 0 to bit 3) are set aside for controlling the interrupt bits.

Bit	Bit Name	Bit Function
7	**TF1**	**Timer 1 Overflow flag.** TF1=1, when timer 1 register overflows. TF1=0, when processor vectors to execute interrupt service routine located at program address **001Bh**.
6	**TR1**	**Timer 1 run control bit.** Set/Cleared by software. TR1=1, enable timer to count. TR1=0, halt timer.
5	**TF0**	**Timer 0 Overflow flag.** TF0=1, when timer 0 register overflows. TF0=0, when processor vectors to execute interrupt service routine located at program address **000Bh**.
4	**TR0**	**Timer 0 run control bit.** Set/Cleared by software. TR0=1, enable timer to count. TR0=0, halt timer.
3	**IE1**	**External interrupt 1 Edge flag.** Set to 1 when a high-to-low edge signal is received on port 3.3 ($\overline{INT1}$). Cleared when processor vectors to interrupt service routine at program address **0013h**. (Not related to timer operations).
2	**IT1**	**External interrupt 1 signal type control bit.** Set to 1 by program to enable external interrupt 1 to be triggered by a falling edge signal. Set to 0 by program to enable a low-level signal on external interrupt 1 to generate an interrupt.
1	**IE0**	**External interrupt 0 Edge flag.** Set to 1 when a high-to-low edge signal is received on port 3.2 ($\overline{INT0}$). Cleared when processor vectors to interrupt service routine at program address **0003h**. (***Not related to timer operations).***
0	**IT0**	**External interrupt 0 signal type control bit.** Set to 1 by program to enable external interrupt 1 to be triggered by a falling edge signal. Set to 0 by program to enable a low-level signal on external interrupt 0 to generate an interrupt.

Note: The instructions used for Timer Control Registers (TCON)

Table 5.2.2: Instructions for TCON registers

For Timer 0			
SETB	TR0	SETB	TCON 4
CLR	TR0	CLR	TXON 4
SETB	TF0	CLR	TCON 5
CLR	TF0	CLR	TCON 5
For Timer 1			
SETB	TR1	SETB	TCON 6
CLR	TR1	CLR	TCON 6
SETB	TF1	SETB	TCON 7
CLR	TF1	CLR	TCON 7

4.3 Timer Modes

- In 8051,timer can operate in any one of four modes: Mode 0, Mode 1, Mode 2 & Mode 3.
- The M1 & M0 bits in TMOD register determines the type of mode.
- The $C/\overline{T} = 0$, **Timer mode** selected.

4.3.1 Timer in Mode1

The following are the characteristics and operations of mode1:

1. It is a 16-bit timer; therefore, it allows value of 0000 to FFFFH to be loaded into the timer's register TL and TH
2. After TH and TL are loaded with a 16-bit initial value, the timer must be started. This is done by SETB TR0 for timer 0 and SETB TR1 for timer 1
3. After the timer is started, it starts to count up.
   - It counts up until it reaches its limit of FFFFH.
   - When it rolls over from FFFFH to 0000, it sets high a flag bit called TF (timer flag).Each timer has its own timer flag: TF0 for timer 0, and TF1 for timer 1.This timer flag can be monitored
   - When this timer flag is raised, one option would be to stop the timer with the instructions CLR TR0 or CLR TR1, for timer 0 and timer 1, respectively

4. After the timer reaches its limit and rolls over, in order to repeat the process. The registers TH and TL must be reloaded with the original value, and TF must be reset to 0.

imer 0 in Mode1

❖ ***Explain the operation of timer0 in mode 1*** ***5-Marks***

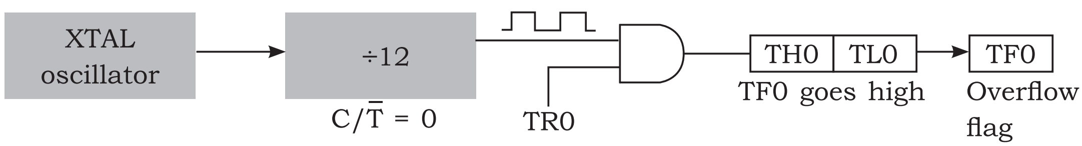


Fig.4.3.1: Block diagram of Timer 0 in Mode 1

Note: C/$\overline{T}$ = 0, TMOD = 01H & TCON =10H

Steps to program Timer 0 in Mode 1

1. Load the TMOD register with **01H** to operate in Timer 0 in Mode 1
2. Load registers TL0 and TH0 with initial count value (16-bit value i.e. 0000H to FFFFH)
3. Start the Timer 0 by setting TR0 in TCON register (**SETB TR0**)
4. Timer 0 started and it counts until it reaches its maximum value i.e. FFFFH and it rolls over to 0000H. Now it will set the TF0 bit in TCON register.

   Keep monitoring TF0 with the "**JNB TF0, here**" instruction until TF0 is set.
5. Stop the Timer 0 (Set TR0=0)
6. Clear TF0 flag for the next round.
7. Go back to Step 2 to load TH0 and TL0 again.

Timer 1 in Mode1

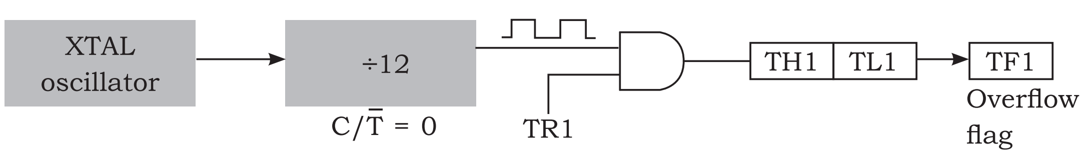


Fig.4.3.2: Block diagram of Timer 1 in Mode 1

Note: C/$\overline{T}$ = 0, TMOD = 10H & TCON = 40H

Steps to program Timer 1 in Mode 1

1. Load the TMOD register with **10H** to operate in Timer 1 in Mode 1

2. Load registers TL1 and TH1 with initial count value (16-bit value i.e. 0000H to FFFFH)
3. Start the Timer 1 by setting TR1 in TCON register (**SETB TR1**)
4. Timer 1 started and it counts until it reaches its maximum value i.e. FFFFH and it rolls over to 0000H. Now it will set the TF1 bit in TCON register.

   Keep monitoring TF1 with the "**JNB TF1, here**" instruction until TF1 is set.
5. Stop the Timer 1 (Set TR1=0)
6. Clear TF1 flag for the next round.
7. Go back to Step 2 to load TH1 and TL1 again.

4.3.2 Timer in Mode 2

The following are the characteristics and operations of mode 2:

1. It is an 8-bit timer; therefore, it allows only values of 00 to FFH to be loaded into the timer's register TH.
2. After TH is loaded with the 8-bit value, the 8051 gives a copy of it to TL Then the timer must be started. This is done by the instruction SETB TR0 for timer 0 and SETB TR1 for timer 1.
3. After the timer is started, it starts to count up by incrementing the TL register.
   - It counts up until it reaches its maximum.
   - When it rolls over from FFH to 00, it sets high the TF (timer flag).
4. When the TL register rolls from FFH to 0 and TF is set to 1, TL is reloaded automatically with the original value kept by the TH register.

- To repeat the process, we must simply clear TF and let it go without any need by the programmer to reload the original value.
- This makes mode 2 an auto-reload, in contrast with mode 1 in which the programmer has to reload TH and TL.

4.3.2.1 Timer 0 in Mode 2

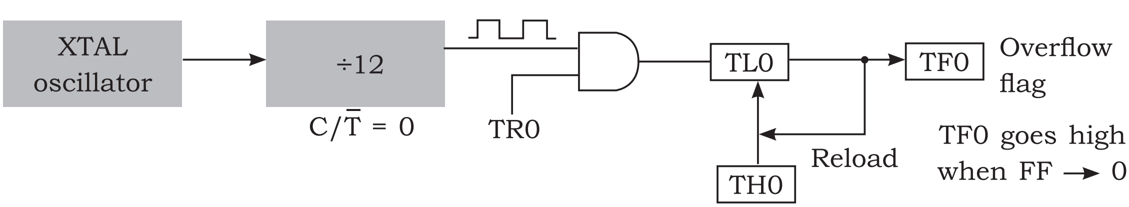


Fig.4.3.3: Block diagram of Timer 0 in Mode 2

Note: C/$\overline{T}$ = 0, TMOD = 02H & TCON = 10H

Steps to program Timer 0 in Mode 2

1. Load the TMOD register with **02H** to operate in Timer 0 in Mode 2.
2. Load initial values into TH0 register (8-bit value i.e. 00H to FFH). The TH0 content is automatically copied into TL0 register.
3. Start the Timer 0 by setting TR0 in TCON register (**SETB TR0**)
4. Timer 0 started and it counts until it reaches its maximum value i.e. FFH and it rolls over to 00H. Now, it will set the TF0 bit in TCON register and the TL0 is reloaded automatically with the initial value (i.e. TH0 value).
   Keep monitoring TF0 with the "**JNB TF0, here**" instruction until TF0 is set.
5. Clear TF0 flag for the next round.
6. Go back to Step 4, since mode 2 is auto-reload.

Timer 1 in Mode 2

1. Load the TMOD register with **20H** to operate in Timer 0 in Mode 2.
2. Load initial value into TH1 register (8-bit value i.e. 00H to FFH). The TH1 content is automatically copied into TL1 register.

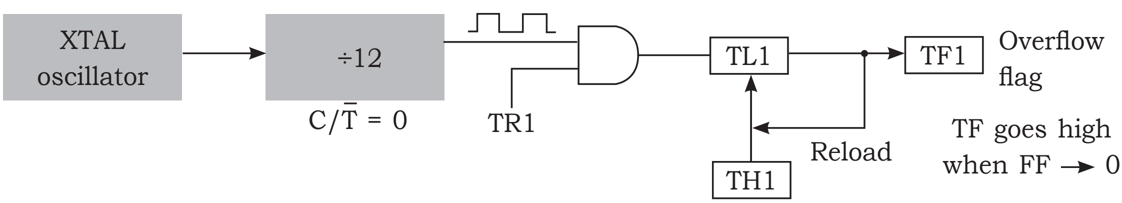


Fig.4.3.4: Block diagram of Timer 1 in Mode 2

Note: C/$\overline{T}$ = 0, TMOD = 20H & TCON = 40H

Steps to program Timer 1 in Mode 2

3. 2 Start the Timer 1 by setting TR1 in TCON register (**SETB TR1**)
4. Timer 1 started and it counts until it reaches its maximum value i.e. FFH and it rolls over to 00H. Now, it will set the TF1 bit in TCON register and the TL1 is reloaded automatically with the initial value (i.e. TH1 value).

   Keep monitoring TF1 with the "**JNB TF1, here**" instruction until TF1 is set.
5. Clear TF1 flag for the next round.
6. Go back to Step 4, since mode 2 is auto-reload.

4.4 Counter Mode

- In 8051,Timer / counter can be used as an event counter by setting **C/$\overline{T}$ =1** in TMOD register.

- In counter mode, the source of clock pulse is from external source.
- For **Counter 0** clock pulse is fed through **T0** (**P 3.4**) and **Counter 1** clock pulse is fed through **T1** (**P 3.5**).

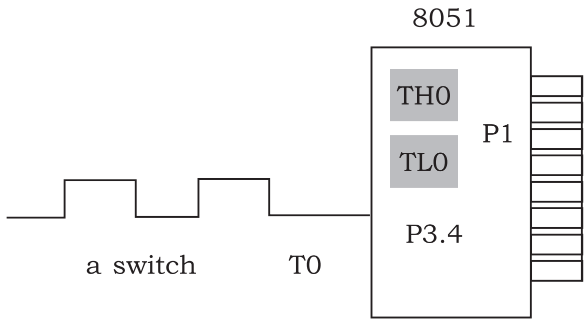


Fig. 4.4.1: Block diagram of Counter

4.4.1 Counter 0 in Mode1

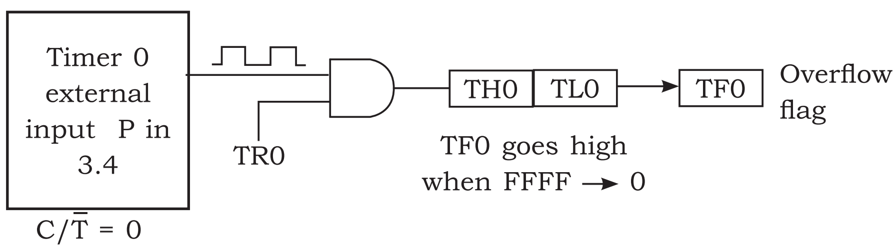


Fig.4.4.2: Block diagram of Counter 0 in Mode 1

<u>Note</u>: C/$\overline{T}$ =1, TMOD = 05H & TCON =10H

Steps to program Counter 0 in Mode 1

1. Load the TMOD register with **05H** to operate in Counter 0 in Mode 1.
2. Load registers TL0 and TH0 with initial count value (16-bit value i.e. 0000H to FFFFH)
3. Start the Counter 0 by setting TR0 in TCON register (**SETB TR0**)
4. Counter 0 started and it counts until it reaches its maximum value i.e. FFFFH and it rolls over to 0000H. Now it will set the TF0 bit in TCON register.
   Keep monitoring TF0 with the "**JNB TF0, here**" instruction until TF0 is set.
5. Stop the Counter 0 (Set TR0=0)
6. Clear TF0 flag for the next round.
7. Go back to Step 2 to load TH0 and TL0 again.

4.4.2 Counter 1 in Mode 1

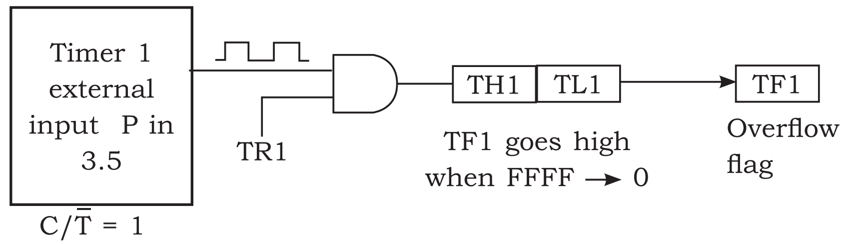


Fig.4.4.3: Block diagram of Counter 1 in Mode 1

Note: C/$\overline{T}$ = 1, TMOD = 50H & TCON = 40H

Steps to program Counter1 in Mode 1

1. Load the TMOD register with **50H** to operate in Counter1 in Mode 1.
2. Load registers TL1 and TH1 with initial count value (16-bit value i.e. 0000H to FFFFH)
3. Start the Counter1 by setting TR1 in TCON register (**SETB TR1**)
4. Counter1 started and it counts until it reaches its maximum value i.e. FFFFH and it rolls over to 0000H. Now it will set the TF1 bit in TCON register.
   Keep monitoring TF1 with the "**JNB TF1, here**" instruction until TF1 is set.
5. Stop the Counter1 (Set TR1=0)
6. Clear TF1 flag for the next round.
7. Go back to Step 2 to load TH1 and TL1 again.

Counter 0 in Mode 2

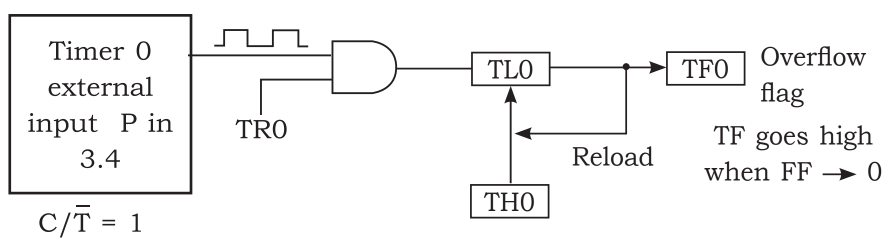


Fig.4.4.4: Block diagram of Counter 0 in Mode 2

Note: C/$\overline{T}$ = 1, TMOD = 06H & TCON = 10H

Steps to program Counter 0 in Mode 2

1. Load the TMOD register with **06H** to operate in Counter 0 in Mode 2.
2. Load initial value into TH0 register (8-bit value i.e. 00H to FFH). The TH0 content is automatically copied into TL0 register.
3. Start the Counter 0 by setting TR0 in TCON register (**SETB TR0**)

4. Counter 0 started and it counts until it reaches its maximum value i.e. FFH and it rolls over to 00H. Now, it will set the TF0 bit in TCON register and the TL0 is reloaded automatically with the initial value (i.e. TH0 value).

   Keep monitoring TF0 with the "**JNB TF0, here**" instruction until TF0 is set.
5. Clear TF0 flag for the next round.
6. Go back to Step 4, since mode 2 is auto-reload.

Counter 1 in Mode 2

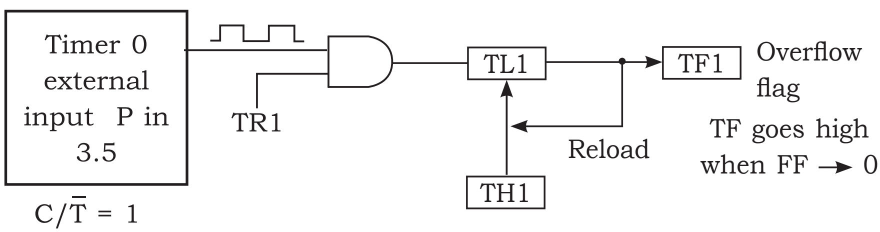


Fig.4.4.5: Block diagram of Counter 1 in Mode 2

Note: C/$\overline{T}$ = 1, TMOD = 60H & TCON = 40H

Steps to program Counter1 in Mode 2

1. Load the TMOD register with **60H** to operate in Counter1 in Mode 2.
2. Load initial value into TH1 register (8-bit value i.e. 00H to FFH). The TH1 content is automatically copied into TL1 register.
3. Start the Counter1 by setting TR1 in TCON register (**SETB TR1**)
4. Counter1 started and it counts until it reaches its maximum value i.e. FFH and it rolls over to 00H. Now, it will set the TF1 bit in TCON register and the TL1 is reloaded automatically with the initial value (i.e. TH1 value).

   Keep monitoring TF1 with the "**JNB TF1, here**" instruction until TF1 is set.
5. Clear TF1 flag for the next round.
6. Go back to Step 4, since mode 2 is auto-reload.

Difference between timers and counters

Sl No	TIMER	COUNTER
1	C/$\overline{T}$ = 0	C/$\overline{T}$ = 1
2	Timers are used to generate time delay.	Counters are used to count the events that occur outside the 8051.
3	For timer operation clock pulses are provided by crystal oscillator circuit.	For counter operation externals clock pulses are applied to pin P3.4 (T0) and P3.5 (T1).

4	Timer register incremented for every machine cycle.	The register is incremented in response to a 2 to 0 transition at its corresponding to external input pin (T0,T1).
5	A timer accumulates series events of a known interval over an interval that is being measured. The measurement of interest is typically the time elapsed between two events.	A counter accumulates an unknown quantity of external events over a known interval of time. The measurement of interest is typically frequency when the events are periodic. If the events are random, the measurement involves event density over time.
6	Maximum count rate is 1/12 of oscillator frequency.	Maximum count rate is 1/24 of oscillator frequency.

Maximum Count Value

The maximum count value of the timers in each mode is given in the table

Table 4.4.1: Maximum count value of the timer in each mode

Timer Mode	Timer size	Initial value	Maximum count value	
			Hexadecimal (H)	Decimal (d)
Mode0	13-bit	0000H	1FFFH	8191
Mode1	16-bit	0000H	FFFFH	65535
Mode2	8-bit	00H	FFH	255
Mode3	8-bit	00H	FFH	255

FORMULAE

$$\boxed{\text{Time delay} = [\text{Maximum Count Value} - (\text{initial count} + 1)] \times \left(\frac{12}{\text{Crystal Frequency}}\right)}$$

$$\boxed{\text{Initial Count} = \left[\text{Maximum Count Value} \; - \left(\text{Time delay} \times \frac{\text{Crystal Frequency}}{12}\right)\right] + 1}$$

$$\boxed{\text{Maximum delay} = \left[\text{Maximum Count Value} \; \times \left(\frac{12}{\text{Crystal Frequency}}\right)\right]}$$

Time Delay Generation & Example Programs in Assembly and C

Example 4.5:

Write a 8051 assembly and C program to generate a delay of 12 µs using Timer0 in Mode1 with XTAL frequency of 22 MHz.

Solution:

$$[\text{Initial value} — 1] = \text{Maximum value} - \text{delay} \times \frac{\text{Crystal Frequency}}{12}$$

$$= \text{FFFFh} - \frac{12\mu s \times 22\ \text{MHz}}{12}$$

$$= 65535 - 12$$

$$= 65513\ (\text{in decimals})$$

Initial value = 65513 +1 = 65514 = FFEAH

The initial value (16-bit) should be loaded into the 16-bit timer register THTL as TH1 = FF (MSB) and TL1 – EA (LSB).

For timer 0 in mode **1 TMOD = 01H**

The initial value is loaded into Timer0 register i.e. TL0 = EAH & TH0 = FFH.

Assembly Language Program (ALP)

```
        ORG 00H
        MOV TMOD,#01          ;Timer 0, Mode 1(16-bit mode)
HERE:   MOV TL0,#0EAH         ;TL0=F2H, the low byte
        MOV TH0,#0FFH         ;TH0=FFH, the high byte
        SETB TR0              ;Start timer 0
WAIT:   JNB TF0, WAIT         ;Wait till TF0=1

        CLR TR0               ;Stop timer 0
        CLR TF0               ;Clear timer 0 flag
        SJMP HERE
        END
```

C Program

```
#include <reg51.h>
void main ( )
{
```

```
    TMOD=0x01;                  //Timer0, Mode1
    while (1)
        {
        TL0=0xF2;               //Initial value FFEA i.e. TL0=EAH
        TH0=0xFF;               //TH0=FFH
        TR0=1;                  //Start Timer0
        while(TF0==0);          //Executes loop until TF0=0
                                //& comes out when TF0=1
        TR0=0;                  //Stop Timer0
        TF0=0;                  //Clear TF0 for next overflow
        }                       // end of while
}                               // end of main
```

Example 4.6:

Write an ALP and C program to generate 50% duty cycle on P1.1. Use timer 1 in mode 1 to generate the delay. Use an initial value of FFAAH for the timer and a crystal frequency of 11.0592 MHz. Also calculate the frequency of square wave.

Solution:

Assembly Language Program (ALP)

For timer 1 in mode 1 TMOD=10H

```
        ORG     00H
        MOV     TMOD,#10        ;Timer 1, Mode 1(16-bit mode)
HERE:   MOV     TL1,#0AAH       ;TL1=AAH, the low byte
        MOV     TH1,#0FFH       ;TH1=FFH, the high byte

        CPL     P1.1            ;Toggle the PORT 1 pin P1.1
        LCALL   DELAY           ;Call delay
        SJMP    HERE            ;repeat again

DELAY:  SETB    TR1             ;Start timer T1
WAIT:   JNB     TF1, WAIT       ;wait till TF1=1
```

```
        CLR     TR1             ;stop timer 1
        CLR     TF1             ;clear timer 1 flag
        RET                     ;return to main program
        END
```

C Program

```
#include <reg51.h>
void delay (void);
sbit pin=p1^1;
void main ( )
{
    TMOD=0x10;          //Tiimer1, Mode1
    while (1)
{
    Pin=~pin;           //Toggle PORT 1 pin P1.1
    TL1=0xAA;           //Initial value FFAA i.e. TL1=AAH
    TH1=0xFF;           //TH1=FFH
}
    void delay ( );
}

void delay (void)
{
        TR1=1;              //Start Timer1
        while (TF1==0);     //Executes loop until TF1=0 & comes out
                            //when TF1=1
        TR1=0;              //Stop Timer1
        TF1=0;              //Clear TF1 for next overflow
}
```

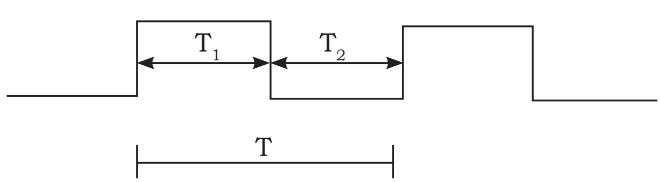


Fig.4.4.6: Square wave with 50% duty cycle.

In this example, the same delay is used for both ON time and OFF time of the square wave i.e. $T_1 = T_2$

The time period of the square wave is $\boxed{T = T_1 + T_2}$

We know that $T_1 = T_2$

$$\boxed{T = T_1 + T_1 = 2T_1 \text{ OR } 2T_2}$$

The time delay is given by

$$\text{Time Delay } T_1 = \frac{12}{\text{Crystal Frequency}} \times (\text{ final value} - \text{initial value} + 1)$$

$$= \frac{12}{11.0592\text{ M}} \times (\text{FFFF} - \text{FFAA} + 1) = 1.085\ \mu\text{s} \times 56\text{ H} = 1.085\ \mu\text{s} \times 86\text{ D}$$

$$\boxed{\text{Time Delay } T_1 = 93.31\mu\text{s}}$$

Hence frequency of square wave 'f' = $\frac{1}{2T_1} = \frac{1}{2 \times 93.31\mu\,\text{sec}} = 5.3584\text{KHz}$

Example 4.7:

Generate a square wave with an ON time of 4 ms and an OFF time of 3 ms on pin P1.1. Assume crystal frequency of 22 MHz. Use timer 1 in mode 1.

Solution:

In this example, the square wave has different ON time (4ms) and OFF time (3ms), hence the intial values for ON time & OFF time are different and need to compute separately.

Note: Timer 1 in mode 1 is an 16-bit mode. Therefore the maximim count value is FFFFH.

TMOD = 10H for Timer1 in Mode 1

<u>ON- Time initial value computation</u>

ON - time required = 4ms = 4×10^{-3}s

Hence

$$(\text{Initial value} - 1) = \text{Maximum value of mode 0} - \text{Required delay} \times \frac{\text{Crystal frequency}}{12}$$

$$= \text{FFFF} - \frac{4 \times 10^{-3} \times 22 \times 10^{6}}{12}$$

$$= \text{FFFFH} - \frac{3 \times 10^{-3} \times 22\text{ MHz}}{12}$$

$$= 65535 - 7333$$

$$= 58202 + 1$$

$$\boxed{\textbf{Initial value = E35BH}}$$

Load TL1 = 5BH and TH1 = E3H

OFF- Time initial value computation

$\text{OFF} - \text{time required} = 3 \text{ ms} = 3 \times 10^{-3} \text{s}$

Hence

$$(\text{Initial value} - 1) = \text{Maximum value of mode 0} - \text{Required delay} \times \frac{\text{Crystal frequency}}{12}$$

$$= \text{FFFFH} - \frac{3 \times 10^{-3} \times 22 \text{ MHz}}{12}$$

$$= 65535 - 5500$$

$$= 60035 + 1$$

$\boxed{\textbf{Initial value = EA84H}}$

Load TL1 = 84H and TH1 = EAH

Assembly Language Program (ALP)

```
            ORG  00H
            MOV  TMOD,#10H     ;Timer 1, Mode 1(16-bit mode)

HERE:       MOV TL1,#5BH       ;TL1=5BH, the low byte
            MOV  TH1,#E3H      ;TH1=E3H, the high byte
            SETB    P1.1       ;Make pin P1.1 high
            ACALL DELAY        ;call delay subroutine

            MOV  TL1,#84H      ;TL1=84H, the low byte
            MOV  TH1,#EAH      ;TH1=EAH, the high byte
            CLR  P1.1          ;Make pin P1.1 low
            ACALL DELAY        ;call delay subroutine

            SJMP   HERE

DELAY:      SETB TR1           ;Start Timer 1
WAIT:       JNB  TF1, WAIT     ;wait till TF1=1
            CLR  TR1           ;stop timer 1
            CLR  TF1           ;clear timer 1 flag
            RET                ;return to main program
            END
```

C Program

```
#include <reg51.h>
void delay (void);
sbit pin=P1^1;
void main ( )
{
    TMOD=0x10;          //Tiimer1, Mode1
    while (1)
    {
        TL1=0x5B;       //initial value to generate 4 ms ON time
        TH1=0xE3;
        pin=1;          //Port Pin P1.1 is made high
        delay( );
        TL1=0x84;       // initial value to generate 4 ms ON
                        // time
        TH1=0xEA;
        pin = 0;        // Port Pin P1.1 is made low
        delay( );

    }                   // end of while loop
}                       // end of main

Void delay (void)
    {
        TR1=1;          //Start Timer1
        while (TF1==0); // Executes loop until TF1=0 & comes out
                        when TF1=1
        TR1=0;          //Stop Timer1
        TF1=0;          // Clear TF1 for next overflow
    }
```

Example 4.8:

Write an ALP and C program to generate a square wave of 20 KHz on pin P2.5. Use timer 0 in mode 2 with crytal frequency of 22 MHz.

Solution:

In mode 2, the timer is 8-bit with a maximum count value of FFH.

TMOD = 020H for Timer0 in Mode 2

The total time of a square wave 20 KHz

$$T = \frac{1}{f} = \frac{1}{20\ \text{KHz}} = 50 \times 10^{-6}\,\text{s}$$

$$T = T_1 + T_2 = 50 \times 10^{-6}\,\text{s}$$

$$T_1 = 25 \times 10^{-6}\,\text{s and } T_2 = 25 \times 10^{-6}\,\text{s}$$

$$(\text{Initial value} - 1) = (\text{Maximum value of mode 2}) - \text{Required delay} \times \frac{\text{Crystal frequency}}{12}$$

$$= (\text{FF}) - 25 \times 10^{-6}\,\text{s} \times \frac{22 \times 10^{6}}{12}$$

$$= 255 - 45.88 = 209 + 1 = \text{D2H}$$

$$\boxed{\text{Initial value} = \text{D2H}}$$

```
        ORG      00H
        MOV      TMOD, #02H     ; Timer0 in Mode 2
        MOV TH0, #D2H           ; Initial value loaded in TH0 register
        SETB     TR0            ; Start Timer 0
WAIT:   JNB TF0, WAIT           ; wait till TF0=1

        CPL P2.5                ; Toggle the PORT 1 Pin P2.5
        CLR TF0                 ;clear timer 0 flag
        SJMP     WAIT
        END
```

C Program

```
#include <reg51.h>
sbit pin=P2^5;
void main ( )
{
    TMOD=0x02;          //Timer0, Mode2
    TH0=0xD2;           // Initial value loaded in TH0 register
    TR0=1;              //Start Timer0
```

```
    while(1)
          {
          pin=~pin;            // toggle Port Pin P1.1
          while (TF0==0);      // executes loop until TF0=0 &
                               comes out when TF0=1
          TF0=0;               // Clear TF0 for next overflow
          }                    // end of while loop
}                              // end of main
```

Example 4.9:

Assume that XTAL = 1.0592 MHz, Use timer Timer1 Mode2,

a. Generate square wave on pin P1.0 with initial value 05H and find the frequency of the generated square wave.

b. The smallest frequency achievable in the program, and the TH value to do that.

Solution:

a)

```
        ORG 00H
        MOV TMOD,#20h         ;T1/mode 2/8-bit/auto-reload
        MOV TH1, #5           ;TH1 = 5
        SETB TRI              ;start timer 1
BACK    JNB TF1, BACK         ;stay until timer rolls over
        CPL P1.0              ;comp,P1.0 to get hi,lo
        SJMP BACK             ;mode 2 is auto reload
        END
```

- In mode 2, we do not need to reload TH since it is auto - reload.
  Initial value = 05h
- Since this program generates a 50% duty cycle square wave,

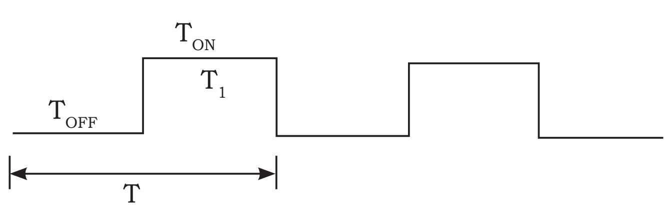


Fig. 4.4.7: Square wave with 50% duty cycle

$$\text{Time delay} = \frac{12}{\text{Crystal frequency}} - \left[\text{Max. value in Mode 2} - \text{initial value}\right]$$

$$= \frac{12}{11.0592\,\text{MHz}} - [\text{FF(h)} - 05]$$

$$= \frac{12}{11.0592\,\text{MHz}} [255 - 05]$$

$$= 1.085069\ 7\ 10^{-6}\ [250]$$

$$\text{Time delay} = 271.26\ \mu\ \text{sec}.$$

From the program, we know that,

$T_{ON} = T_{OFF} = 271.26$ µsec

$T = T_{ON} + T_{OFF} = 271.26$ µsec + 271.26 µsec

$$f = \frac{1}{T} = \frac{1}{544.67\ \mu\,\text{sec}} = 1.83957\text{KHz}$$

b)

To get smallest frequency, we need the largest time delay "T"

- Maximum time delay can be achieved when initial value = 00h

Formula:

$$\text{Maximum time delay} = \frac{12}{\text{Crystal frequency}} \left[\text{Maximum value in mode 2}\right]$$

$$= \frac{12}{11.0592\,\text{MHz}} [\text{FF(h)}]$$

$$= 1.085089 \times 10^{-6}\ (255\text{d})$$

Max. Time delay = 276.85092 µsec

We know that, $T = T_{ON} + T_{OFF}$

= 276.85 µsec + 276.85 µsec

= 553.7019 µsec and,

$$f = \frac{1}{T} = \frac{1}{553.7019\ \mu\,\text{sec}} = 1.806025 \times 10^{3}\ \text{Hz}$$

This is the smallest frequency that can be generated in Mode 2.

Example 4.10:

Assume that XTAL = 22MHz, write a program to generate a square wave of frequency 1KHz on pin P1.2. Use timer0 in Mode2.

Solution:

XTAL = 22 MHz, f = 1 KHz on P1.2 pin

We know that, $T = \frac{1}{f} = \frac{1}{1\ \text{KHz}}$

$$\boxed{T = 1\ \text{m sec}}$$

$$T = T_{ON} + T_{OFF} = 1\ \text{m sec}$$

$$\therefore T_{ON} = T_{OFF} = 0.5\ \text{msec}$$

✸ Maximum possible delay in Mode 2 $= FF(h) \times \frac{12}{\text{Crystal frequency}}$

$$= 255(d) \times \frac{12}{22\ \text{MHz}} = 0.1390909\ \text{m sec}$$

But, required delay is 0.5 msec.

✸ So, 1st generate a delay of 0.1 msec and repeat the loop for 5 times.

i.e., 0.1 msec × 5 = 0.5 msec

$$\text{counter "N"} = \frac{0.5\ \text{m sec}}{0.1\ \text{m sec}} = 5$$

Computation of Initial volume for 0.1 msec:

$$(\text{Initial vlaue} - 1) = \text{Max. Value in Mode 2} - \left[\text{Required time delay} \times \frac{\text{Crystal frequency}}{12}\right]$$

$$= FF(h) - \left[0.1\ \text{m sec} \times \frac{22\ \text{MHz}}{12}\right]$$

$$(\text{Initial vlaue} - 1) = 255(d) - [183.33]$$

$$= 72(d) + 1 = 73(d)$$

$$\boxed{\text{Initial vlaue} = 49(h)}$$

```
        ORG  00H
        MOV  TMOD,#02H          ;Timer 0, mode 2
REPT:   CPL  P1.2               ;complement P1.2
        MOV  R0, #05            ;count for multiple delays
AGAIN   MOV  TH0,#48H           ;loaf TH0 value
        SETB TR0                ;start Timer 0
BACK    JNB  TF0, BACK          ;stay until timer rolls over
        CLR  TR0                ;stop timer
```

```
CLR  TF0                   ;clear timer flag
DJNZ R0, AGAIN             ;repeat until R0=0
SJMP    REPT               ;repeat to get a train of pulses
END
```

Example 4.11:

Write an 8051 C program to toggle only pin P1.1 continuously every 250 ms. Use Timer 0 mode 2 to create the delay.

Solution:

Assume XTAL = 11.0592 MHz

$$\text{Max. Possible Time delay in mode 2} = \text{Max. value in mode } 2 \times \frac{12}{\text{Crystal frequency}}$$

$$= \text{FFH} \times \frac{12}{11.0592 \times 10^6} = 276.6927\ \mu\text{sec}$$

Required time delay is greater than maximum time delay, i.e.,

250 msec > 276.6927 μsec

$$\text{Counter "N"} = \frac{250\ \text{m sec}}{276.692\ \mu\text{sec}} = 903.52$$

In the above counter value, we are getting fraction value, so, change the counter value.

Case 1:

$$\text{Counter "N"} = \frac{250\ \text{m sec}}{50\ \text{m sec}} = 5000$$

- 1st generate a time delay of 50 msec and repeat the process for 5000 times, i.e.,

  50 μsec ×100 × 50 = 250 msec
- It can be written as, 100 × 50 = 5000

Computatioin of Initial value:

$$(\text{Initial vlaue} - 1) = \text{Max. Value in Mode } 2 - \left[\text{Required time delay} \times \frac{\text{Crystal frequency}}{12}\right]$$

$$= \text{FF(h)} - \left[50\ \mu\text{sec} \times \frac{11.0592\ \text{MHz}}{12}\right]$$

$$(\text{Initial vlaue} - 1) = 255(\text{d}) - 46.08 + 1$$

$$= 210(\text{d})$$

$$\boxed{\text{Initial vlaue} = \text{D2h}}$$

Case 2:

- Counter $"N" = \frac{250 \text{ m sec}}{25 \, \mu \text{sec}} = 10000$

$$\boxed{"N" = 10,000}$$

- It can be written as 250 × 40 = 10000

  i.e., 25 µsec × 250 × 40 = 250 msec

$$(\text{Initial vlaue} - 1) = \text{Max. Value in Mode 2} - \left[\text{Required time delay} \times \frac{\text{Crystal frequency}}{12}\right]$$

$$= \text{FF(h)} - \left[25 \, \mu\text{sec} \times \frac{11.0592 \text{ MHz}}{12}\right]$$

$$(\text{Initial vlaue} - 1) = 255(1) - 23.04 \text{ d} + 1 = 233$$

$$\boxed{\text{Initial vlaue} = \text{E9h}}$$

- We can also load equivalent -ve value of E9h,

  i.e., $\boxed{\text{E9h} = -23}$

```
#include <reg51,h>
void delay (void);
Sbit mybit=P1^5;
void main(void)
    {
    unsigned char x,y;
    while(1)
        {
        mybit=~mybit;           //toggle P1.5
        for (x=0; x=250; x++)
            for(y=0; y=40; y++)
                delay();
        }
}
void delay(void)
{
    TMOD=0×02;      //Timer0, mode 2 (8-bit auto-reload)
    TH0=-23;        //load TH0 (auto-reload value)
    TR0=1:          //turn on T0
```

```
    While(TF0==0);  //wait for TF0 to roll over
    TR0=0;          //turn off T0
    TF=0;           //clear TF0
}
```

Example 4.12

Write an 8051 c program to creat a frequency of 2500 Hz on pin P2.7. Use timer 1, mode 2 to creat the delay

Solution

$$T = \frac{1}{F} = \frac{1}{2500\ \text{Hz}}$$

$$\boxed{T = 0.4\ \text{m sec}}$$

$T = 0.4$ msec

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

$T = 0.2$ msec $+0.2$ msec

$$\boxed{T_{ON} = T_{OFF} = 0.2\ \text{msec}}$$

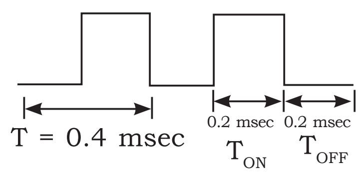


Fig: 4.4.9: Square wave

Computation of Initial value:

$$(\text{Initial vlaue} - 1) = \text{Max. Value in Mode 2} - \left[\text{Required time delay} \times \frac{\text{Crystal frequency}}{12}\right]$$

$$= \text{FF(h)} - \left[0.2\ \text{m sec} \times \frac{11.0592 \times 10^6}{12}\right]$$

$$= 255(\text{d}) - [184.32(\text{d})] + 1$$

$$= 72(\text{d})$$

$$\boxed{\text{Initial vlaue} = 48\text{h}}$$

C Program

```
#include<reg51.h>
voidd elay (void);
sbit mybit=P2^7;
void main(void)
```

```
{
    while(1)
    {
        mybit=~mybit;      //toggle P2.7
        delay();
    }
}
voiddelay (void)
{
 TMOD=0x20;                //Timer 1, mode 2(8-bit auto-reload)
 THL=-184;                 //load TH1(auto-reload value)
 TR1=1;                    //turn on T1
while (TF1==0)             //wait for TF1 to roll over
TR1=0;                     //turn off T1
TF1=0;                     //clear TF1
}
```

Example 4.13:

Assume that 1-Hz external clock is being fed into pin T1 (P3.5). Write a C program for counter 1 in mode 2 (8 -bit auto reload) to count up and display the state of the TL1 count on P1. Start the count at 0H

Solution

```
include<reg51.h>
sbit Tl=P3^5;
void main(void)
{
    Tl=l;                       //make T1 an input
    TMOD=0×60;
    TH0=0;                      //set count to 0
    while(1)                    //repeat for ever
{
    do
        {
            TR1=1;              //start timer
```

```
            P1=TL1;             //place value on pins
        }
            while(TF0==0);      //wait here
            TR1=0;              //stop timer
            TF1=0;              //clear flag
        }
}
```

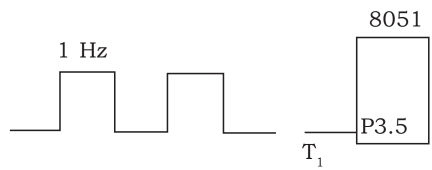


Fig: 4.4.10: Square wave

4.5 Serial Communication

Introduction

Computers transfer data in two ways:Serial communication and Parallel communication.

Serial communication

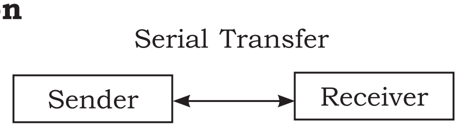


Fig. 4.5.1: Serial Communication

- Serial Communication use single data line to transfer data one bit at a time.
- It is used for long distance communication.
- Serial communication is slower than parallel communication.

  Example: IBM keyboards transfer data to the motherboard.

Parallel communication

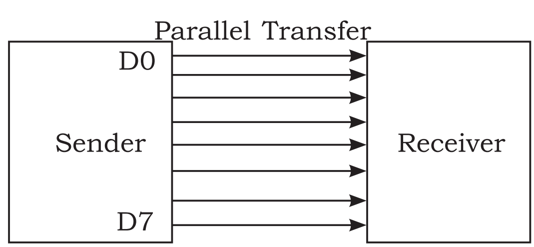


Fig. 4.5.2: Parallel Communication

- In parallel communication, number of lines required to transfer data depends on the number of bits to be transferred simultaneously.

- Parallel communication works only for shorter distance.
- Parallel communication is faster than serial communication.
  Example: Data communication from computer to printer.

Data Transfer Rates:

- The rate of data transfer in serial communication is stated in bps (bits per second). Another widely used terminology for bps is Baud rate.
- The baud rate & bps are not same. The baud rate is the MODEM terminology & is defined as the number of signal changes per second.
- In modem a single change of signal sometimes transfers several bits of data.
- For conductor wire, the baud rate & bps are the same. So for this reason we use the term bps & baud interchangeably.

Baud rate in the 8051

- The 8051 transfers & receives data serially at many different baud rates. The baud rate in the 8051 is programmable.
- A standard crystal frequency, XTAL=11.0592 MHz is used to generate the baud rate.
- The 8051 divides the crystal frequency by 12 to get the machine cycle frequency, i.e. XTAL/12 = 11.0592MHz/12 = 921.6 KHz.
- The 805l's serial communication UART circuitry divides the machine cycle frequency of 921.6 KHz by 32 i.e. 921.6 KHz/32 = 28,800 Hz, then fed to the Timerl to set the baud rate as shown in below figure 5.5.3.

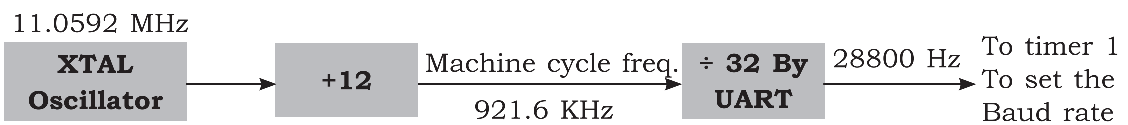


Fig. 4.5.3: UART Circuitry

- The 8051 baud rate is set by Timerl using Mode 2 (8-bit auto reload).
- To get the baud rates compatible with the PC, we must load TH1 with the values shown in below table.

Table 4.5.1: Timer 1 TH1 register values for various Baud Rates

TH1		**Baud Rate**	
DECIMAL	HEX	SMOD = 0	SMOD = 1
–3	FD	9,600	19,200
–6	FA	4,800	9,600
–12	F4	2,400	4,800
–24	E8	1,200	2,400
Note: XTAL = 11.0592 MHz			

4.6 SCON (SERIAL CONTROL) REGISTER

	0	0	0	0	0	0	0	0	Value after reset
SCON	SM0	SM1	SM2	REN	TB8	RB8	TI	RI	**Bit name**
	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	

Fig. 4.6.1: SCON Serial Port Control Register (Bit addressable)

SCON is an 8-bit register used to program the start bit, stop bit, and data bits of data framing, among other things.

<table>
<tr><th>Bit</th><th>Bit Name</th><th>Bit Function</th></tr>
<tr><td>7 & 6</td><td>SM0 & SM1</td><td>Serial Port Mode specifier
<table>
<tr><td>SM0</td><td>SM1</td><td></td></tr>
<tr><td>0</td><td>0</td><td>Serial Mode 0</td></tr>
<tr><td>0</td><td>1</td><td>Serial Mode 1, 8-bit data, 1 stop bit, 1 star bit</td></tr>
<tr><td>1</td><td>0</td><td>Serial Mode 2</td></tr>
<tr><td>1</td><td>1</td><td>Serial Mode 3</td></tr>
</table></td></tr>
<tr><td>5</td><td>SM2</td><td>This enables the multiprocessing capability of the 8051.</td></tr>
<tr><td>4</td><td>REN</td><td>Receive Enable
It is a bit-addressable register
When REN=1, it allows 8051 to receive data on RxD pin.
If REN=0, the receiver is disable.</td></tr>
<tr><td>3</td><td>TB8</td><td>Transfer bit 8
Set/Cleared by hardware to determine state of the 9th bit data transmitted in 9-bit UART (In mode 2 & 3). We make TB8=0 since it is not used in our application.</td></tr>
<tr><td>2</td><td>RB8</td><td>Receive bit 8
Set/Cleared by hardware to indicate state of the 9th bit data received -bit UART (In mode 2 & 3). We make RB8=0 since it is not used in our application.</td></tr>
<tr><td>1</td><td>TI</td><td>Transmit Interrupt
When 8051 finishes the transfer of 8-bit character
It raises TI flag to indicate that it is ready to transfer another byte.
TI bit is raised at the beginning of the stop bit.</td></tr>
</table>

0	RI	**Receive Interrupt** When 8051 receives data serially via RxD, it gets rid of the start and stop bits and places the byte in SBUF register It raises the RI flag bit to indicate that a byte has been received and should be picked up before it is lost. RI is raised halfway through the stop bit.

4.6.1 SBUF register

- SBUF is an **8-bit register** used solely for serial communication.
- The byte data to be transmitted on the serial port (TxD line) is written into SBUF register.
- Data is framed with the start and stop bits and transferred serially via the TxD line.

SBUF									Bit name
	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	

Fig. 4.6.2: SBUF register

- The SBUF can be accessed like any other register in the 8051. as

  MOV SBUF, #'E' ;load SBUF=45h, ASCII for 'E'

  MOV SBUF, A ;copy accumulator into SBUF

  MOV A, SBUF ;copy SBUF into accumulator
- SBUF holds the byte of data when it is received by 8051 RxD line.
- When the bits are received serially via RxD, the 8051 deframes it by eliminating the stop and start bits, making a byte out of the data received, and then placing it in SBUF.
- The previous data must be read before the new byte completes. Otherwise , the old data will be lost.

4.6.2 Programming the 8051 to transfer data serially

Steps used to program 8051 to transfer data serially

1. TMOD register is loaded with the value 20H, indicating the use of timer 1 in mode 2 (8-bit auto-reload) to set baud rate.
2. The TH1 is loaded with one of the values to set baud rate for serial data transfer.
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 **CLR TI** instruction.

6. The character byte to be transferred serially is written into **SBUF** register.
7. The TI flag bit is monitored with the use of instruction **“JNB TI, xx”** to see if the character has been transferred completely.
8. To transfer the next byte, go to step 5.

4.6.3 Importance of the TI 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 **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=l), indicating that the last character was transmitted and it is **ready to transfer the next character.**
5. **By monitoring the TI flag**, make sure that we are **not overloading the SBUF** register. If we write another byte into SBUF register before TI is raised, the untransmittcd 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 charactcr.
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**.

4.6.4 Programming the 8051 to receive data serially

In programming 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 **TIMER1** in **mode 2** (8-bit auto-reload) to **set** the **baud rate**.
2. The **TH1** is loaded with one of the four values to set the **baud rate** for serial data transfer (Assuming **XTAL=11.0592MHz**).
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. **RI** is cleared by the "**CLR RI**" instruction.
6. The **RI** flag bit is monitored with the use of the transmission "**JNB RI, label**" to see if an entire character has been received yet.
7. **When RI is raised, SBUF** has the byte. Its contents are moved into a safe place.
8. To receive the next character, go to step 5.

4.6.5 Importance of the RI flag

In receiving bits via its **R×D pin**, the 8051 goes through the following steps:

1. It receives the **start bit** indicating that **the next bit is first bit of the character byte** it is about to receive.
2. The **8-bit character is received one bit at a time**. When the last bit is received, a byte it is about to receive.
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 placed 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.**

4.6.6 RS 232

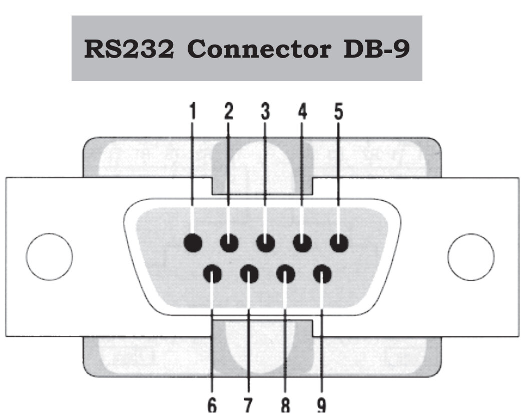


Fig. 4.6.3: DB-9 9-Pin Connector.

Table 4.6.1: IBM PC DB-9 Signals

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

- IBM introduced the DB-9 version of the serial I/O standard.
- Current terminology classifies data communication equipment as
  - i) **DTE** (Data Terminal Equipment) refers to terminal and computers that send and receive data.
  - ii) **DCE** (Data Communication Equipment) refers to communication equipment, such as modems.

Working of serial port Connecting 8051 to RS 232

The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD, RxD, and ground.

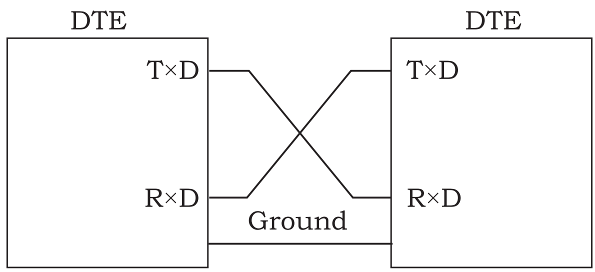


Fig. 4.6.4: Null Modem Connection

- Many of the pins of RS-232 connector are used for handshaking signals. Their descriptions are provided below and they are not supported by the 8051 UART chip.
- The RS-232handshaking signals are

1) DTR (Data Terminal Ready)

When terminal is turned on, it sends out signal DTR to indicate that it is ready for communication.

2) DSR (Data Set Ready)

When DCE is turned on and has gone through the self-test, it assert DSR to indicate that it is ready to communicate.

3) RTS (Request to Send)

When the DTE device has byte to transmit, it assert RTS to signal the modem that it has a byte of data to transmit.

4) CTS (Clear to Send)

When the modem has room for storing the data it is to receive, it sends out signal CTS to DTE to indicate that it can receive the data now.

5) DCD (Data Carrier Detect)

The modem asserts signal DCD to inform the DTE that a valid carrier has been detected and that contact between it and the other modem is established.

6) RI (Ring Indicator)

An output from the modem and an input to a PC indicates that the telephone is ringing. It goes on and off in synchronous with the ringing sound.

FORMULAE

$$\text{Baud rate} = \frac{\text{Crystal frequency}}{12 \times 32} \times \frac{1}{(256 - \text{TH1})}$$

$$\text{TH1} = 256 - \frac{\text{Crystal frequency}}{12 \times 32 \times \text{Baud rate}}$$

Example 4.14:

1. With XTAL = 11.0592 MHz, find the TH1 value needed to have the following baud rates. (a) 9600 (b) 4800 (c) 2400 (d) 1200

Solution

i) 9600 Baud Rate	ii) 4800 Baud Rate
$\text{TH1} = 256 - \frac{\text{Crystal frequency}}{12 \times 32 \times \text{Baud rate}}$ $\text{TH1} = 256 - \frac{11.0592 \times 10^6}{12 \times 32 \times 9600} = 256 - 3 = 253$ TH1 = FDH	$\text{TH1} = 256 - \frac{\text{Crystal frequency}}{12 \times 32 \times \text{Baud rate}}$ $\text{TH1} = 256 - \frac{11.0592 \times 10^6}{12 \times 32 \times 4800} = 256 - 24 = 232$ TH1 = FAH
iii) 2400 Baud Rate	**iv) 1200 Baud Rate**
$\text{TH1} = 256 - \frac{\text{Crystal frequency}}{12 \times 32 \times \text{Baud rate}}$ $\text{TH1} = 256 - \frac{11.0592 \times 10^6}{12 \times 32 \times 2400} = 256 - 12 = 244$ TH1 = F4H	$\text{TH1} = 256 - \frac{\text{Crystal frequency}}{12 \times 32 \times \text{Baud rate}}$ $\text{TH1} = 256 - \frac{11.0592 \times 10^6}{12 \times 32 \times 1400} = 256 - 24 = 232$ TH1 = E8H

TH1 register values to obtain different Baud rates

Initial Value		Baud Rate
TH1 in Hexadecimal	**TH1 in decimal**	
FD	–3	9600
FA	–6	4800
F4	–12	2400
E8	–24	1200

Note: In 8051, serial communication uses a standard crystal frequency i.e. **11.0592 MHz.**

SCON register configuration:

For serial port communication SCON is loaded with 50H as shown in figure.

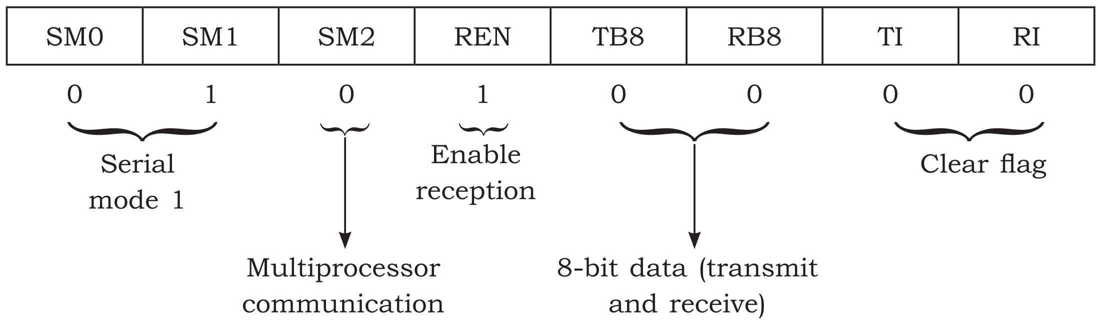


Fig. 4.6.5: SCON register configuration

TMOD register configuration:

For serial port communication TMOD register is loaded with 20H i.e. Timer1, Mode2.

Example 4.15:

Write a program for the 8051 to transfer letter "A" serially at 9600 baud, continuously.

Solution:

```
        ORH  00H
        MOV      TMOD,#20H      ;timer 1,mode 2(auto reload)
        MOV      TH1,#-3        ;9600 baud rate
        MOV      SCON,#50H      ;8-bit, 1 stop, REN enabled
        SETB     TR1            ;start timer 1
AGAIN:  MOV      SBUF,#”A”      ;letter “A” to transfer
HERE:   JNB TI,  HERE           ;wait for the last bit
        CLR      TI             ;clear TI for next char
        SJMP     AGAIN          ;keep sending A
        END
```

C-Program

```
#include <reg51.h>
void main(void)
{
        TMOD=0x20;              //use Timer 1, mode 2
```

```
            TH1=0xFD;                    //9600 baud rate
            SCON=0x50;
            TR1=1;
            while (1)
        {
            SBUF='A';                    //place value in buffer
            while (TI==0);
            TI=0;
        }
    }
```

Example 4.16:

Write an ALP and C program to serially transmit the message "ECE" continuously at a baud rate of 9600, 8-bit data and 1 stop bit.

Solution:

ALP	C Program
ORG 00H	#include <reg51.h>
MOV TMOD, #20H	void send(unsigned char);
MOV TH1, #-3	void main()
MOV SCON, #50H	{
SETB TR1	TMOD=0x20;
AGAIN MOV A, # "E"	TH1=0xFD;
ACALL **SEND**	SCON=0x50;
MOV A, # "C"	TR1=1;
ACALL **SEND**	while (1)
MOV A, # "E"	{
ACALL **SEND**	send('E');
SJMP **AGAIN**	send('C');
SEND: MOV SBUF, A	send('E');
WAIT: JNB TI, **WAIT**	}
CLR TI	}
RET	void send(unsigned char msg);
END	{
	SBUF=msg;
	while(TI==0);
	TI=0;
	}

Alternate ALP and C Program

ALP	C Program
ORG 00H	#include <reg51.h>
MOV TMOD, #20H	void main()
MOV TH1, #-3	{
MOV SCON, #50H	unsigned char i;
SETB TR1	unsigned char msg[ ] = "ECE";
MOV DPTR, #MESSAGE	TMOD=0x20;
AGAIN: CLR A	TH1=0xFD;
MOVC A, @A+DPTR	SCON=0x50;
JZ **GO**	TR1=1;
MOV SBUF, A	while (1)
	{
WAIT: JNB TI, **WAIT**	for(i=0; i<5; i++)
CLR TI	{
INC DPTR	SBUF=msg[i];
GO: SJMP **GO**	while (TI==0);
MESSAGE: DB "ECE",0	TI=0;
END	}
	}
	}

Example 4.17:

Write an ALP and C program to serially transmit the message "HELLO" continuously at a baud rate of 9600, 8-bit data and 1 stop bit.

Solution:

ALP	C Program
ORG 00H	#include <reg51.h>
MOV TMOD, #20H	void send(unsigned char);
MOV TH1, #-3	void main()
MOV SCON, #50H	{
SETB TR1	TMOD=0x20;
AGAIN MOV A, # "H"	
ACALL **SEND**	

......Continued

`MOV A,#"E"`	`TH1=0xFD;`
`ACALL SEND`	`SCON=0x50;`
`MOV A,#"L"`	`TR1=1;`
`ACALL SEND`	`while (1)`
`MOV A, #"L"`	`{`
`ACALL SEND`	`send('H');`
`MOV A, #"0"`	`send('E);`
`ACALL SEND`	`send('L');`
`SJMP AGAIN`	`send('L');`
	`send('0');`
`SEND: MOV SBUF, A`	`}`
`WAIT: JNB TI, WAIT`	`}`
`CLR TI`	`void send(unsigned char msg);`
`RET`	`{`
	`SBUF=msg;`
`END`	`while(TI==0);`
	`TI=0;`
	`}`

Example 4.18:

Write an ALP and C program to serially transmit the message "GMIT DAVANGERE" continuously at a baud rate of 9600, 8-bit data and 1 stop bit.

Solution:

ALP	C Program
`ORG 00H`	`#include <reg51.h>`
`MOV TMOD, #20H`	`void main( )`
`MOV TH1,#-3`	`{`
`MOV SCON,#50H`	`unsigned char i;`
`SETB TR1`	`unsigned char msg[ ] =`
`repeat: MOV DPTR,#msg`	`"GMIT DAVANGERE";`
`up: CLR A`	`TMOD=0x20;`
`MOVC A,@A+DPTR`	`TH1=0xFD;`
`JZ repeat`	`SCON=0x50;`
`ACALL send`	`TR1=1;`
`INC DPTR`	`while (1)`

......Continued

`        SJMP up`
`send:   MOV SBUF,A`
 	
`here:   JNB TI, here`
`        CLR TI`
`        RET`
`msg: db "GMIT DAVANGERE",0`
`        END`
	`        {`
	`            for(i=0; i<14; i++)`
	`            {`
	`                SBUF=msg[i];`
	`                while (TI==0);`
	`                TI=0;`
	`            }`
	`        }`
	`}`

Example 4.19

Write a program to transfer a letter 'Y' serially at 9600 baud continuously and also to send a letter 'N' through port0, which is connected to a display divide

Solution

```
        ORG 00h
        MOV TMOD,#020H    ; timer 1. mode 2
        MOV TH1,#-3       ; 9600 baud rate
        MOV SCON,#50H     ; 8 bits, 1stop, REN enabled
        SETB TRI          ; START TIMER 1
AGAIN:
        MOV SBUF, #"Y"    ; transfer *Y* serially
HERE:  JNB TI, HERE       ; WAIT FOR transmission to be over
        CLR TI            ; clear Tl for next transmission
        MOV P0, #"N"      ; move "N" to P0 for parallel transfer
        SJMP AGAIN        ; repeat
        END
```

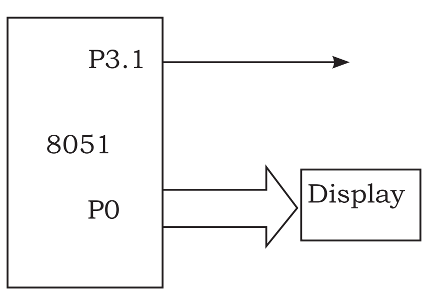


Fig. 4.6.6: 8051 PORT P0 interfaced with display

Example 4.20:

Write an ALP and C program to receive serial data at 4800 baud rate and send it to port 0.

Solution

ALP

```
        ORG 00H
        MOV TMOD, #20H  ;Timer1, Mode2
        MOV TH1, #-6    ;4800 baud rate
        MOV SCON, #50H  ;serial mode 1
        SETB TR1        ;start timer 1
AGAIN:  CLR RI          ;clear RI flag
WAIT:   JNB RI, WAIT    ;wait till character is received
        MOV A, SBUF     ;received character is moved into Accumulator
        MOV P0, A       ;copy Accumulator content into P0
        SJMP AGAIN      ;repeat for next character
        END
```

C Program

```
#include <reg51.h>
void main()
{
        unsigned char rdata;
        TMOD=0x20;              //use Timer 1, mode 2
        TH1=0xFA;               //4800 baud rate
        SCON=0x50;              //serial mode 1
        TR1=1;                  //start timer 1
        while (1)
        {
                while (RI==0);          //wait to receive data
                rdata=SBUF;             //read received data
                P0=rdata;               //send received data to P0
                RI=0;                   //clear RI flag
        }                               //end of while
}                                       //end of main
```

Example 4.21:

Write a Cprogram for the 8051 to transfer the letter "C" serially at 9600 baud continuously. Use S bit data and 1 stop bit.

Solution

```
#include<reg51.h>
void mainO
        {
            TMOD=0x20;
            THl=0xFD;
            SCON=0X50;
            TR1=1;
            while(l)
            {
                SBUF='C';
                while(TI=0);
                TI=0;
            }
}
```

```
        ORG  00H
        MOV  TMOD,#20h
        MOV  THl,#-3
        MOV  SCON,#50h
        SETB  TR1
up:
        MOV  SBUF,#'C'
        here:  JNB  TI  here
        CLR  TI
        SJMP  up
        END
```

Example 4.22:

find an 8051 C program to transfer the message "GOOD MORNING" serially at 9600 baud, 8-bit data, 1 stop bit.

Solution

```
        ORG  00H
        MOV  TMOD,#20H
        MOV  THl,#-3
        MOV  SCON,  #50H
        SETB  TR1
repeat:
        MOV  DPTR,  msg
up:     CLR  A
        MOVC  A,@A+DPTR
        JZ  repeat
        ACALL  send
```

```
#include<reg51.h>
void main()
{
        unsigned char i;
        unsigned char msg[]="GOOD
        MORNING";
        TMOD=Ox20;
        THl=0xFD:
        SCON=0X50;
        TR1=1:
        while(l)
```

......Continued

INC DPTR	{
SJMP **up**	for(i=0; i< 12; i++)
send:	{
MOV SBUF, A	SBUF=msg[i];
here: JNB TI, **here**	while (TI==0);
CLR TI	TI=0:
RET	}
msg: db **"GOOD MORNING"**,0	}
END	}

Example 4.23:

Write an 8051 C program to receive byte of data serially and put them on P1. Set the baud rate to 4800, 8-bit data, 1 stop bit.

Solution

#include<reg51.h>	ORG 00h
void main()	MOV TMOD,#20h
{	MOV TH1,#-6
TMOD=0x20;	MOV SCON,#50h
TH1=-6;	SETB TR1
SCON=0X50;	**up:** CLR RI
TR1=1;	**here:** JNB RI, **here**
while(1)	MOV A, SBUF
{	MOV P1,A
RI=0;	SJMP **up**
while(RI==0);	END
P1=SBUF;	
}	
}	

OR

#include<reg51.h>	ORG 00h
void main()	MOV TMOD,#20h
{	MOV TH1,#-6
TMOD=0x20;	MOV SCON,#50h
TH1=-6;	SETB TR1

......Continued

```c
        SCON=0X50;
        TR1=1;
        while(1)
        {
            while(RI==0);
            P1=SBUF;
            RI=0;
        }
}
```

```asm
up:
here:   JNB  RI, here
        MOV  A, SBUF
        MOV  P1,A
        CLR  RI
        SJMP up
END
```

NOTE: Both programs are correct.

Example 4.24:

Write an 8051 ALP and C program to transfer serially the character "H" continuously at a baud rate of 9600.

Solution

```c
#include<reg51.h>
void main()
{
        TMOD=0x20;
        TH1=0xFD;
        SCON=0X50;
        TR1=1;
        while(1)
        {
            SBUF='H';
            while(TI==0);
            TI=0;
        }
}
```

```asm
        ORG  00h
        MOV  TMOD,#20h
        MOV  TH1,#-3
        MOV  SCON,#50h
        SETB TR1
up:
        MOV  SBUF,#'H'
here:   JNB  TI, here
        CLR  TI
        SJMP up
        END
```

Example 4.25:

Write an 8051 ALP and C program to transfer serially the character "H" continuously at a baud rate of 19200 with a crystal frequency of 11.0592 MHz.

Solution

C Program	ALP

`#include<reg51.h>`
`void main()`
`{`
`TMOD=0x20;`
`TH1=0xFD;`
`SCON=0X50;`
`TR1=1;`
`PCON = PCON | 0X80;`
`while(1)`
`{`
`SBUF='H';`
`while(TI==0);`
`TI=0;`
`}`
`}`
	`ORG 00h`
	`MOV A,PCON`
	`SETB ACC.7`
	`MOV PCON,A`
	`MOV TMOD,#20h`
	`MOV TH1,#-3`
	`MOV SCON,#50h`
	`SETB TR1`
	`up:`
	`MOV SBUF,#'H'`
	`here: JNB TI, here`
	`CLR TI`
	`SJMP up`
	`END`

Example 4.26:

Write an 8051 ALP and C program to transfer serially the character "H" to serial #1 continuously at a baud rate of 9600.

Solution

C Program	ALP

`#include<reg51.h>`
`sfr SBUF1=0XC1;`
`sfr SCON1=0XC0;`
`sbit TI1=0XC1;`
`void main()`
`{`
`TMOD=0x20;`
`TH1=0xFD;`
`SCON1=0X50;`
`TR1=1;`
`while(1)`
`{`
`SBUF1='H';`
`while(TI1==0);`
`TI1=0;`
`}`
`}`
	`SBUF1 EQU 0C1h`
	`SCON1 EQU 0C0h`
	`TI1 BIT 0C1h`
	
	`ORG 00h`
	`MOV TMOD,#20h`
	`MOV TH1,#-3`
	`MOV SCON1,#50h`
	`SETB TR1`
	`up:`
	`MOV SBUF1,#'H'`
	`here:` `JNB TI1, here`
	`CLR TI`
	`SJMP up`
	`END`

Example 4.27:

Write an 8051 ALP and C program to receive data serially and it in RAM memory location 60h. Set a baud rate of 9600.

Solution

```c
        #include<reg51.h>
        void main()
{
        unsigned char value;
        TMOD=0x20;
        TH1=-3;
        SCON=0X50;
        TR1=1;
        while(1)
        {
            RI=0;
            while(RI==0);
            value = SBUF;
        }
}
```

```asm
        ORG 00h
        MOV TMOD,#20h
        MOV TH1,#-3
        MOV SCON,#50h
        SETB TR1
up:     CLR RI
here:   JNB RI, here
        MOV A, SBUF
        MOV 60h,A
        SJMP up
        END
```

Example 4.28:

Write an 8051 ALP and C program to receive data serially and it in RAM memory location 60h. Set a baud rate of 9600. Use serial #1.

Solution

```c
#include<reg51.h>
sfr SBUF1=0XC1;
sfr SCON1=0XC0;
sbit TI1=0XC1;
void main()
{
        unsigned char value;
        TMOD=0x20;
        TH1=-3;
        SCON1=0X50;
```

```asm
        SBUF1 EQU 0C1h
        SCON1 EQU 0C0h
        TI1 BIT 0C1h
        ORG 00h
        MOV TMOD,#20h
        MOV TH1,#-3
        MOV SCON1,#50h
        SETB TR1
up:     CLR RI1
        here: JNB RI1, here
```

......Continued

`        TR1=1;`	`        MOV A, SBUF1`
`        while(1)`	`        MOV 60h,A`
`        {`	`        SJMP up`
`            RI1=0;`	`        END`
`            while(RI1==0);`	
`            value = SBUF1;`	
`        }`	
`}`	

Example 4.29:

Write an 8051 ALP and C program to transfer serially the character "H" continuously at a baud rate of 19200 with a crystal frequency of 11.0592 MHz.

Solution

```
#include<reg51.h>
void main()
{
        TMOD=0x20;
        TH1=0xFD;
        SCON=0X50;
        TR1=1;
        PCON = PCON | 0X80;
        while(1)
        {
          SBUF='H';
          while(TI==0);
          TI=0;
        }
}
```

```
        ORG 00h
        MOV A,PCON
        SETB ACC.7
        MOV PCON,A
        MOV TMOD,#20h
        MOV TH1,#-3
        MOV SCON,#50h
        SETB TR1
up:
        MOV SBUF,#'H'
        here: JNB TI, here
        CLR TI
        SJMP up
        END
```

Example 4.30:

Write an 8051 C program to send two different strings to the serial port. Assuming that SW is connected to pin P2.0, monitor its status and make a decision as follows:

SW=0: send your first name

SW=1: send your last name

Assume XTAL=11.0592 MHz, baud rate of 9600,8-bit data, 1 stop bit.

Solution

```
#include<reg51.h>
sbit SW=P2^0;                           //input switch
void main(void)
{
            unsigned char i;
            unsigned char fname [] =”ALI”;
            unsigned char lname [] ="AKBAR";
            TMOD=0x20;
            TH1=0xFD;
            SCON=0x50;
            TR1=1;
            if (SW==0)                  //check switch
            {
                for (i=0;i<3;i++)
                {
                     SBUF=fname[i];
                     while (TI==0);
                     TI=0;
                }
            }
        else
                {
                for (i=0 ; i<5 ; i++)         //write name
                     {
                         SBUF=lname[i];   //place value in buffer
                         while (TI==0);   //wait for transmit
                         TI=0;
                     }
            }
}
```

Example 4.31:

Write an 8051 C program to send 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; 9600 baud rate
SW=1; 19200 baud rate
Assume that XTAL=11.0592MHz for both cases.

Solution

```
#include<reg 51.h>
sbit SW=P2^0;                        //input switch
void main(void)
        {
               unsigned char i;
               unsigned char mess1 [] ="Normal Speed";
               unsigned char mess2 [] ="High Speed";
               TMOD=0X20;          //use timer1, 8bit auto-reload
               TH1=0XFD;           //9600 for normal speed
               SCON=0X50;
               TR1=1;             //star timer
               if (SW==0)
           {
           for (i=0;i<12;i++)
               {
                   SBUF=mess1[i];
                   while (TI==0);
                   TI=0;
               }
           }
           else
               {
                   PCON=PCON|0X80; //for high speed of 19200
                   for (i=0;i<10;i++)
                   {
                      SBUF=mess2[i];
                      while (TI==0);
                      TI=0;
               }
           }
        }
```

5

8051 Interrupts and Interfacing Applications

5.1 Introduction to Interrupts

An interrupt is the occurrence of a condition (an event) that causes a temporary suspension of a program while the event is serviced by another program (Interrupt Service Routine **'ISR'**).

OR

An interrupt is an external or internal event that interrupts the microcontroller to inform it that a device needs its service.

5.1.1 Interrupt &Polling methods

A single microcontroller can serve several devices. There are two ways to do that

1. Interrupts
2. Polling.

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 handler.

Polling

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 each one is serviced.

5.1.2 Comparison between interrupt and polling method

Sl. No.	INTERRUPTS	POLLING
1	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.	Microcontroller can monitor the status of several devices and serve each of them as certain conditions are met.After that, it moves on to monitor the next device until each one is serviced.
2	Each device can get the attention of the microcontroller based on the priority assigned to it.	The polling method cannot assign priority since it checks all devices in a round-robin fashion.
3	Microcontroller can also ignore (MASK) a device request for service.	Microcontroller cannot ignore (MASK) a device request for service.
4	Microcontroller time is used efficiently.	Microcontroller time is not efficiently used.

5.1.3 Steps in executing an interrupt

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.

5.1.4 Different types of interrupt

The 8051 has six sources of interrupts and only five interrupts are available to the user. The RESET interrupt is not available to the user.

RESET: When the RESET pin is activated, the 8051 jumps to address location 0000H., i.e., program counter is loaded with 000H (PC = 00H)

Timer Interrupts: There are two timer interrupts

1. TF0 for Timer 0 and its interrupt vector address is 000BH.
2. TF1 for Timer 1 and its interrupt vector address is 001BH

External Hardware Interrupts

There are two external hardware interrupts and also referred to as EX1 and EX2.

1. $\overline{\text{INT0}}$ (P3.2) and its interrupt vector address is 0003H.
2. $\overline{\text{INT1}}$ (P3.3) and its interrupt vector address is 0013H

Serial communication interrupt

It has a single interrupt that belongs to both receive and transmit and its interrupt vector address is 0023H.

Table 5.1.1: Interrupt Vector Table

Interrupt	ROM Location (Hex)	Pin
Reset	0000	9
External hardware interrupt 0 ($\overline{\text{INT}}$0)	0003	P3.2 (12)
Timer 0 interrupt (TF0)	000B	
External hardware interrupt 1 ($\overline{\text{INT}}$1)	0013	P3.3 (13)
Timer 1 interrupt (TF1)	001B	
Serial COM interrupt (RI and TI)	0023	

5.1.5 IE and IP registers

Interrupt enabling disabling and priority setting

- Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated. The interrupts must be enabled by software in order for the microcontroller to respond to them.
- There is a register called IE (interrupt enable) that is responsible for enabling (unmasking) and disabling (masking) the interrupts.

5.1.6 Interrupt Enable (IE) registers

D7							D0
EA	–	ET2	ES	ET1	EX1	ET0	EX0

Fig. 5.1.1 : Interrupt Enable register

EA	IE.7	Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
-	IE.6	Not implemented, reserved for future use.*
ET2	IE.5	Enables or disables Timer 2 overflow or capture interrupt (8052 only).
ES	IE.4	Enables or disables the serial port interrupt.
ET1	IE.3	Enables or disables Timer 1 overflow interrupt.
ET1	IE.2	Enables or disables external interrupt 1.
ET0	IE.1	Enables or disables Timer 0 overflow interrupt.
EX0	IE.0	Enables or disables external interrupt 0.

Note: User software should not write is to reserved bits. These bits may be used in future flash microcontrollers to invoke new features.

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.

5.1.7 Interrupt Priority

When the 8051 is powered up, the priorities are assigned according to Table 4.7.2. In reality, the priority scheme is nothing but an internal polling sequence in which the 8051 polls the interrupts in the sequence listed and responds accordingly

Table 5.1.2: 8051 Interrupt Priority upon Reset

Highest to Lowest Priority	
External Interrupt 0	($\overline{INT0}$)
Timer Interrupt 0	(TF0)
External Interrupt 1	($\overline{INT1}$)
Timer Interrupt 1	(TF1)
Serial Communication	(RI + TI)

For example, that if external hardware interrupts 0 and 1 are activated at the same time, external interrupt 0 is responded to first. Only after $\overline{INT0}$ has been serviced, $\overline{INT1}$ serviced, since $\overline{INT1}$ has the lowest priority.

5.1.8 Priority Setting

Interrupt priority is done by programming a register called Interrupt Priority (IP) Register. Upon power-up reset, the IP register contains all 0s, making the priority sequence based. To give a higher priority to any of the interrupts, we make the corresponding bit in the IP register high.

5.1.9 Interrupt Priority (IP) Register

D7							D0
–	–	PT2	PS	PT1	PX1	PT0	PX0

Fig. 5.1.2: Interrupt Priority Register

Priority bit = 1 assigns high priority. Priority bit = 0 assigns low priority.

—	IP.7	Reserved
—	IP.6	Reserved
PT2	IP.5	Timer 2 interrupt priority bit (8052 only)
PS	IP.4	Serial port interrupt priority bit
PT1	IP.3	Timer 1 interrupt priority bit
PX1 .	IP.2	External interrupt 1 priority bit
PT0	IP.1	Timer 0 interrupt priority bit
PX0	IP.0	External interrupt 0 priority bit

Note: User software should never write 1s to unimplemented bits, since they may be used in future microcontrollers to invoke new features.

Each interrupt source can be programmed to have one of the two priority levels by setting (high priority) or clearing (low priority) a bit in the IP (Interrupt Priority) Register. A low priority interrupt can itself be interrupted by a high priority interrupt, but not by another low priority interrupt. If two interrupts of different priority levels are received simultaneously, the request of higher priority level is served. If the requests of the same priority level are received simultaneously, an internal polling sequence determines which request is to be serviced.

Example 5.1:

Discuss what happens if interrupts INT0, TF0, and INT1 are activated at the same time. Assume priority levels were set by the power-up reset and the external hardware interrupts are edge-triggered.

Solution:

If these three interrupts are activated at the same time, they are latched and kept internally. Then the 8051 checks all five interrupts according to the sequence listed in Table 4.7.2. If any is activated, it services it in sequence. Therefore, when the above three interrupts are activated, IE0 (external interrupt 0) is serviced first, then timer 0 (TF0), and finally IE1 (external interrupt 1).

Example 5.2:

(a) Program the IP register to assign the highest priority to $\overline{\textbf{INT}}$1 (external interrupt 1),

(b) Discuss what happens if $\overline{\textbf{INT}}$0 $\overline{\textbf{INT}}$1 , and TF0 are activated at the same time. Assume the interrupts are both edge-triggered.

Solution:

(a) MOV IP,#00000100B ;IP.2=1 assign $\overline{\text{INT}}$1 higher priority. The instruction SETB IP.2 also will do the same thing as the above line since IP is bit-addressable.

(b) The instruction in Step (a) assigned a higher priority to I$\overline{\text{INT}}$1 than the others; therefore, when $\overline{\text{INT}}$0, $\overline{\text{INT}}$1, and TF0 interrupts are activated at the same time, the 8051 services $\overline{\text{INT}}$1 first, then it services $\overline{\text{INT}}$0, then TF0. This is due to the fact that $\overline{\text{INT1}}$ has a higher priority than the other two because of the instruction in Step (a). The instruction in Step (a) makes both the $\overline{\text{INT}}$0 and TF0 bits in the IP register 0. As a result, the sequence in **Table 4.7.2** gives a higher priority to I$\overline{\text{INT}}$0 over TF0.

Example 5.3:

Assume that after reset, the interrupt priority is set the instruction MOV IP,#00001100B. Discuss the sequence in which the interrupts are serviced.

Solution:

The instruction "MOV IP #00001100B" (B is for binary) and timer 1 (TF1)to a higher priority level compared with the reset of the interrupts. However, since they are polled according to **Table 5.1.2,** they will have the following priority.

Highest Priority External Interrupt 1 ($\overline{\text{INT}}$1)

Timer Interrupt 1 (TF1)

External Interrupt 0 ($\overline{\text{INT}}$0)

Timer Interrupt 0 (TF0)

Lowest Priority Serial Communication (RI+TI)

Example 5.4:

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
```

Example 5.5:

Discuss the interrupt priority order achieved by the execution of MOV IP,#11H instruction ***5-Marks***

Ans. IP = 0 0 0 **1** 0 0 0 **1** i.e. **PS=1** & **PX0=1**

- After executing ***MOV IP,#11H*** *instruction*, two interrupts i.e. Serial port interrupt and External interrupt 0 are assigned with priority levels
- The external interrupt 0 has highest priority. So it will be served first.
- The serial port interrupt has lowest priority and hence served after external interrupt 0.

5.1.9 8051 Interrupt Numbers

8051 compiler have extensive support for the interrupts.

- They assign a unique number to each of the 8051 interrupts.
- It can also assign a register bank to an ISR. This avoids code overhead due to the pushes and pops of the R0-R7 registers.

Table 5.1.3: 8051 Interrupt Numbers

Interrupt	Name	Numbers used by 8051 C
External Interrupt 0	($\overline{INT}0$)	0
Timer Interrupt 0	(TF0)	1
External Interrupt 1	($\overline{INT}1$)	2
Timer Interrupt 1	(TF1)	3
Serial Communication	(RI + TI)	4

Advantages

1. Multitasking
2. Efficient, better than polling.
3. Great flexibility.
4. All peripheral devices have to be handled by tasks
5. Data transfer by polling.
6. Hardware details encapsulated in dedicated routines.
7. Can enable interrupts before the servicing of an individual interrupt is complete reducing interrupt latency.
8. Handles interrupt with different priorities.
9. Uses a single jump and saves valuable cycles to go to the ISR.

Disadvantages

1. Sometimes requires additional hardware.
2. Degradation of processor performance (busy wait)
3. Kernel has to be modified when adding devices.
4. More inter-processor communication.
5. Task must know low level details of the drive.
6. Lower priority interrupts can block higher priority interrupts.
7. Trends to be more complex.
8. Each ISR has a mechanism to set the external interrupt mask to stop lower-priorities interrupts from halting the current ISR, which add extra code to each ISR.

Differentiate between RET and RETI

RET

- The RET instruction returns the program from a subroutine.

- RET pops the return address from the stack and continue execution there and making the 8051 return to where it left. The high byte and low byte address of PC from the stack and decrements the SP by 2.
- The execution of the instruction will result in the program to resume from the location just after the CALL instruction.
- No flags are affected.

Example:

- Suppose SP=0BH originally and an interrupt is detected during the instruction ending at location 0213H
    - ▶ RAM Internal locations 0AH and 0BH contain the values 14H and 02H respectively.
    - ▶ The RET instruction leaves SP = 09H and returns program execution to location 0214H.

RETI

- The RETI instruction returns the program from an interrupt subroutine.
- RETI pops the high byte and low byte address of a PC from the stack and restores the interrupt logic to accept additional interrupts.
- SP decrements by 2 and no other registers are affected. However the PSW is not automatically restored to its pre-interrupt status.
- After the RETI, program execution will resume immediately after the point at which the interrupt is detected.
- No flags are affected.

Example:

- Suppose SP=0BH originally and an interrupt is detected during the instruction ending at location 0213H
    - ▶ RAM Internal locations 0AH and 0BH contain the values 14H and 02H respectively.
    - ▶ The RETI instruction leaves SP=09H and returns program execution to location 0214H.

❖ ***Explain why we cannot use RET instead of RETI as the last instruction of an ISR.***

Solution:

- RET and RETI perform the same actions of popping off the top two bytes of the stack into the program counter, and making the 8051 return to where it left.

- However, RETI also performs an additional task of clearing the interrupt-in-service flag, indicating that the servicing of the interrupt is over and the 8051 now can accept a new interrupt on that pin.
- If we use RET instead of RETI as the last instruction of the interrupt service routine, then it will simply block any new interrupt on that pin after the first interrupt, since the pin status would indicate that the interrupt is still being serviced.
- In the case of TF0, TF1, TCON.1 and TCON.3, they are cleared due to the execution of RETI.

5.1.10 Enabling or disabling of Interrupts.

1. Using bit instructions

Method 1	Method 2	Comments
SETB EA	SETB IE. 7	; EA=1 ;enable interrupts
SETB ET0	SETB IE. 1	; ET0 = 1;enable timer 0 interrupt
SETB ET1	SETB IE. 3	; ET1 = 1;enable timer 1 interrupt
SETB EX0	SETB IE. 0	; EX0= 1;enable external 0 interrupt
CLR ES	CLR IE. 4	; ES=0 ;disable serial interrupt
CLR EX1	CLR IE. 2	; EX1=0 ;disable external 1 interrupt

2. Using byte instruction

```
MOV IE, #10001011B 'OR' MOV   IE,   #8DH
```

Note: Bypassing interrupt vector table

```
    ORG 0000H   ;wake-up ROM reset locationi
    LJMP TO     ;bypass interrupt vectortable
                ; the wake-up program
    ORG    30H
TO:
       ....
    END
```

Example 5.4:

Write a C program using interrupts to do the following:

(a) Generate a 10000 Hz frequency on P2.1 using TO 8-bit auto-reload,

(b) Use timer 1 as an event counter to count up a 1-Hz pulse and display it on P0. The pulse is connected to EX1. Assume that XTAL = 11.0592 MHz. Set the baud rate at 9600.

Solution:

```
include <reg51.h>
abit WAVE = P2^1;
```

```
unsigned char cnt;
void timer0() interrupt 1
        {
            WAVE=~WAVE;        //toggle pin
        }
void timer1() interrupt 3
        {
            cnt++;             //increment counter
            P0=cnt;            //display value on pins
void main()
            {
            cnt=0;             //set counter to zero TMOD - 0x4 2;
            TMOD=0x-46;        //10000 Hz
            IE=0x86;           //enable interrupts
            TR0=1;             //start timer 0
            TR1=1;             //start timer 1
            while(1);          //wait until interrupted
1/10000 Hz = 100 µs
100 µs/2 = 50 µs
50 µs/1.085 ms = 46
```

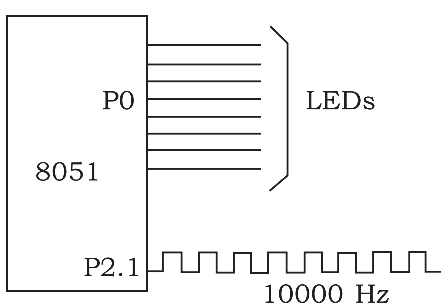


Fig.5.1.3: Generating square wave on P2.1

Example 5.:7

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
```

Note: In C Program, to code ISR, we use standard syntax as shown below. (For timer 0, interrupt number is '**1**').

Keyword to use interrupt

```
void time0 (void) interrupt 1
{
ISR of timer 0
}
```

Interrupt Number

Example 5.8:

Write a program that displays a value of *Y at port l and 'N' at port 3 and also generates a square wave of 10 kHz with Timer 0 in mode 2 at port pin P0.1. XTAL=22 MHz.

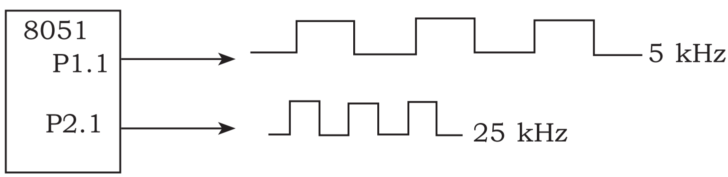
Solution:

```
;--- upon wake up, go to main,avoid using memory space allocated to
interrupt vectoi table
      ORG 0000H
      LJMP MAIN            ;bypass interrupt vector table
;---ISR for Timer 0 to generate square wave
      ORG 000BH            ;Timer 0 interrupt vector
      CPL P0.1
      RETI
;---the main program for initialization
      ORG 003OH            ;a location after the interrupt vectors
MAIN: MOV TMOD,#02H        ;Timer 0, mode 2 (auto-reload)
```

```
        MOV  THO, #0B6H       ;move count value into THO
        MOV  IE,#82H          ;enable interrupt timer 0
        SETB  TR0             ;start Timer 0
BACK:   MOV  P1,#'Y'          ;display'Y'at port PJ
        MOV  P3,#'N'          ;display'N'at port P3
        SJMP  BACK            ;keep doing this until interrupted
        END
```

Example 5.9:

Write a program to generate two square waves one of 5 KHz frequency at pin P1.1 and another of frequency 25 kHz at pin P2.1 Assume XTAL = 22 MHz.



Solution:

```
;Tested for an AT89C51 with a crystal frequency of 22 MHz.
        ORG  0000H            ;avoid using the interrupt vector table
        LJMP  MAIN
;--ISR  for Timer 0
        ORG  000B             ;Interrupt vector for Timer 0
        CPL P1.1
        RETI
;---ISR for Timer 1
        ORG 001BH             ;Interrupt vector for Timer 1
        CPL P2.1
        RETI
;---main program for initialization
        ORG 0030H
MAIN:   MOV TMOD,#22H         ;both timers are initialized for Mode 2
        MOV IE,#8AH           ;enable the Timer 0 and Timer 1 Interrupts
        MOV THO,#048H         ;count value for 5 KHz square wave
        MOV TH1,#0B6H         ;count value for 25 KHz square wave
        SETB  TR0             ;start Timer 0
```

```
        SETB  TR1              ;start Timer 1
WAIT:   SJMP  WAIT             ;keep waiting for the roll off of either
                                timer
        END
```

Example 5.10:

Two switches are connected to pins P3.2 and P 3.3. When a switch is pressed, the corresponding line goes low. Write a program to
(a) light all LEDs connected to port 1, if the first switch is pressed.
(b) light all LEDs connected to port 2, if the second switch is pressed.

Solution:

```
;Tested for an AT89C5 with a crystal frequency of 22 MHz.
```

Pin 3.2 is the pin for Interrupt 0, and pin 3.3 is the pin for Interrupt 1.

```
;upon wake up,go to main
      ORG 0000H         ;bypass interrupt vector table
      LJMP MAIN
;------------------the ISR for interrupt INTO
      ORG 0003H         ;interrupt vector for Interrupt 0   '
LED1: MOV P1,#0FFH      ;turn on LEDs of portl
      MOV R0,#255
      DJNZ R0,LED1      ;keep the LEDs ON for a short time RETI
;------------------ISR for INT 1
      ORG 0013H         ;Interrupt vector for Interrupt 1
LED2  MOV P2, #0FFH     ;tum on LEDs of port 2
      MOV RO,#255
      DJNZ R,LED2       ;keep the LEDs ON for a short time RETI
      ;------------------main program for initialization
      ORG 0030H
MAIN: MOV IE,#85H       ;enable INTO and INT1 HERE:
      SJMP HERE
      END
```

The LEDs will remain ON if the corresponding switch is kept pressed.

Example 5.11:

Generate from all pins of Portl, a square wave which is half the frequency of the signal applied at INTO pin.

Solution:

```
;Tested for an AT89C51 with a crystal frequency of 22 MHz.
```

Every negative edge at Pin 3.2 will cause the INT0 (vectored to location 0003) interrupt to be activated.

```
      ORG 0000H
      LJMP MAIN
;--ISR for hardware interrupt INT0
      ORG 0003H
      CPL PI
      RETI
      ORG 0030H
MAIN: SETB TCON.0            ;make INTO an edge-triggered interrupt
      MOV IE,#81H            ;enable hardware interrupt INT0
HERE: SJMP HERE
      END
```

Example 5.12:

Write a C program that continuously gets a single bit of data from P1.1 and sends it to P1.2, while simultaneously creating a square wave of 200 ms period on pin P1.3. Use timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

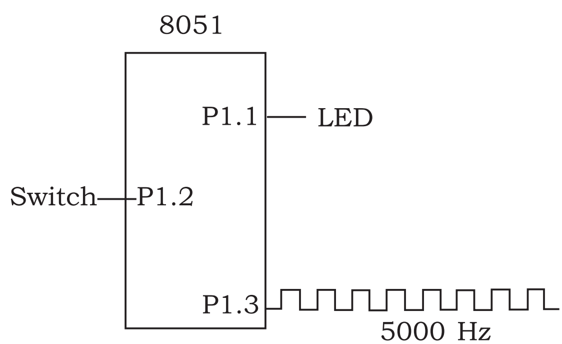
Solution:

We will use timer 0 in mode 2 (auto-reload). One half of the period is 100 µs.

100/1.085 µs = 92, and TH0 = 256 - 92 = 164 or A4H

200 µs/2 = 100µs

100 µs / 1.085 µs = 92




```
#include <reg51.h>
sbit SW   = Pl^2;
sbit rec  = Pl^1;
sbit WAVE = Pl^3;
void timer0(void) interrupt 1
      {
      WAVE = ~WAVE;     //toggle pin
      }
void main()
      {
```

```
        SW = 1;             //make switch input
        TMOD = 0x02;
        THO = 0xA4  ;       //THO = -92
        IE = 0x82;          //enable interrupts for timer 0
        while(1)
              {
                 rec = SW;//send switch to LED
              }
}
```

Example 5.13:

Write a C program using interrupts to do the following:

(a) Generate a 10000 Hz frequency on P0.1 using TO 8-bit auto-reload,

(b) Use timer 1 as an event counter to count up a 1-Hz pulse and display it on P1. The pulse is connected to EX1

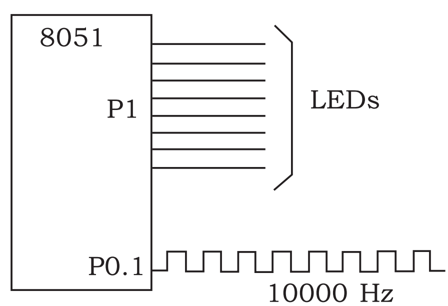
Assume that XTAL = 11.0592 MHz. Set the baud rate at 9600.

Solution:

1 / 10000 Hz = 100 μs

100 μs / 2 = 50 μs

50 μs / 1.085 μis = 46




```
#include <reg51.h>
sbit WAVE = P0^1;
unsigned char cnt;
void timer0() interrupt 1
{
        WAVE = ~WAVE;       //toggle pin
}
void timerl() interrupt 3
{
        cnt++;              //increment counter
        PI = cnt;           //display value on pins
}
void main()
{
        cnt =0;             //set counter to zero
        TMOD = 0x42;
        TH0 = 0x-46;        //10000 Hz
        IE = 0x86;          //enable interrupts
```

```
      TR0 = 1;          //start timer 0
      TR1 = 1;          //start timer 1
      while(1);   //wait until interrupted
}
```

Example 5.14:

Write a C program that continuously gets a single bit of data from P1.7 and sends it to P1.0, while simultaneously creating a square wave of 200 μs period on pin P2.5. Use Timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

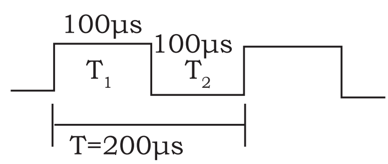
Solution:

We will use timer 0 mode 2 (auto-reload).

$$\boxed{T = T_1 + T_2}$$

$$T = 200\mu s$$

$$T_1 = T_2 = \frac{T}{2} = 100\mu s$$



Initial Count value:

$$TH0 = \text{Final value} - \frac{\text{Required delay} \times \text{crystal freqyency}}{12}$$

$$= 256 - \frac{100\mu s \times 11.0592\,\text{MHz}}{12} = 256 - 92 = 164 = \text{A4H}$$

$$\boxed{TH0 = \text{A4H}}$$

```
#include <reg51.h>
sbit SW=P1^7;
sbit S_data=P1^0;
sbit S_WAVE=P2^5;
void timer0(void) interrupt 1
{
    S_WAVE=~S_WAVE;             //toggle pin
}
void main()
{
    SW=1;                       //make switch input
    TMOD=0x02;
```

```
    TH0=0xA4;                    //TH0=-92
    IE=0x82;                     //enable interrupt for timer 0
    while (1)
    {
        S_data=SW;               //send switch to LED
    }
}
```

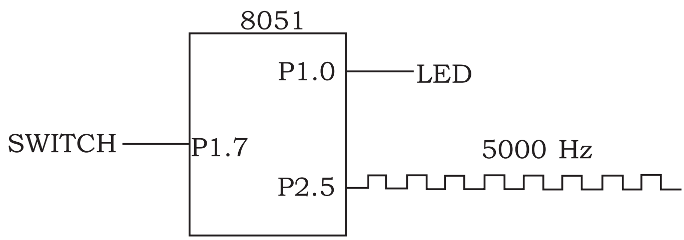


Fig. 5.1.4: Generating square wave on P2.5

The frequency of the square wave generated is 5 KHz.

Example 5.15:

Write a C program using interrupts to do the following:

(a) Receive data serially and send it to P0

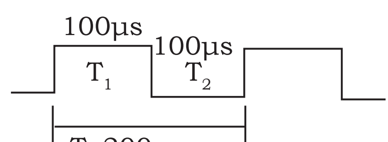
(b) Read port P1, transmit data serially, and give a copy to P2

(c) Make timer 0 generate a square wave of 5 kHz frequency

Assume that XTAL = 11.0592 MHz. Set the baud rate at 4800

Solution:

$$\text{Time period } T = \frac{1}{f} = \frac{1}{5\text{KHz}} = 200\mu s$$



$$\boxed{T = T_1 + T_2}$$

$$T = 200\mu s$$

$$T_1 = T_2 = \frac{T}{2} = 100\mu s$$

Initial Count value:

$$\text{TH0} = \text{Final value} - \frac{\text{Required delay} \times \text{crystal freqyency}}{12}$$

$$= 256 - \frac{100\mu s \times 11.0592\,\text{MHz}}{12} = 256 - 92 = 164 = \text{A4H}$$

$$\boxed{\text{TH0} = \text{A4H}}$$

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

```
sbit WAVE=P0^1;
void timer0() interrupt 1
{
    WAVE=~WAVE;                     //toggle pin
}
    void serial0() interrupt 4
{
    if (TI==1)
    {
        TI=0;                       //clear interrupt
    }
    else
    {
        P0=SBUF;                    //put value on pins
        RI=0;                       //clear interrupt
    }
}

void main()
{
    unsigned char x;
    P1=0xFF;                        //make P1 an input
    TMOD=0x22;
    TH1=0xF6;                       //4800 baud rate
    SCON=0x50;
    TH0=0xA4;                       //5 kHz has T=200us
    IE=0x92;                        //enable interrupts
    TR1=1;                          //start timer 1
    TR0=1;                          //start timer 0
    while (1)
    {
        x=P1;                       //read value from pins
        SBUF=x;                     //put value in buffer
        P2=x;                       //write value to pins
    }
}
```

Example 5.16:

Write a C program using interrupts to do the following:

(a) Generate a 10 KHz frequency on P2.1 using T0 8-bit auto-reload

(b) Use timer 1 as an event counter to count up a 1-Hz pulse and display it on P0. The pulse is connected to EX1.

Assume that XTAL = 11.0592 MHz. Set the baud rate at 9600.

Solution:

```
#include <reg51.h>
sbit WAVE =P2^1;
unsigned char cnt;
void timer0 ( ) interrupt 1
{
        WAVE=~WAVE;                 //toggle pin
}
void timer1() interrupt 3
{
        cnt++;                //increment counter
        P0=cnt;               //display value on pins
}
void main()
{
        cnt=0;                //set counter to 0
        TMOD=0x42;
        TH0=0x-46;            //10 KHz
        IE=0x86;              //enable interrupts
        TR0=1;                //start timer 0
        while (1);            //wait until interrupted
}
```

Example 5.17:

Write an ALP and C Program to generate a square wave of 5 KHz with Timer0 Mode2 at port pin P2.1 using interrupt mode. Also display a value of "A' at PORT 0. Use XTAL = 22 MHz.

Solution:

We know that

$$\boxed{T = T_1 + T_2}$$

$$T = \frac{1}{f} = \frac{1}{5KHz} = 2\times10^{-4}s$$

$$T_1 = T_2 = \frac{T}{2} = 1\times10^{-4}s$$

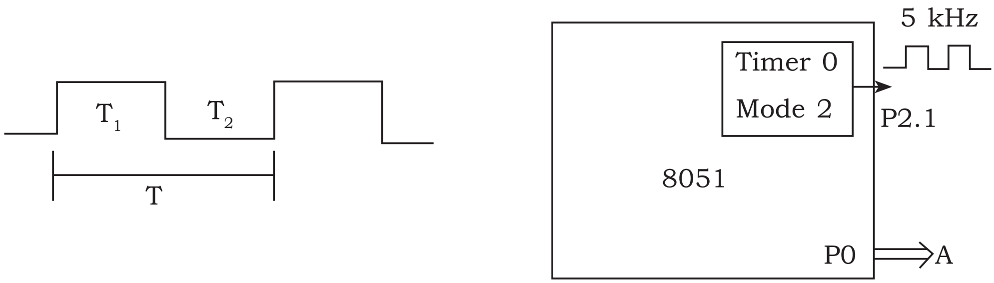


Fig. 5.1.5: Generating square wave on P2.1

Initial Count value:

$$TH0 = \text{Final value} - \frac{\text{Required delay}\times\text{crystal freqyency}}{12}$$

$$= 256 - \frac{1\times10^{-4}s\times22\times10^{6}}{12}$$

$$= 256 - 183$$

$$= 73$$

$$\boxed{TH0 = 49H}$$

IE register configuration

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

Enable interrupts

Enable timer 0

Fig. 5.1.6: IE register configuration

Load IE = 82H

The instruction used is **MOV IE,#82H** OR **SETB EA and SETB ET0**

ALP

```
         ORG   00H
         SJMP  MAIN                  ;jump to main program
         ORG   000BH                 ;ISR for Timer0
         CPL   P2.1                  ;Toggle pin P2.1 for square wave
         RETI                        ;return from interrupt
         ORG   30H                   ;main program written from 30H
MAIN:    MOV   TMOD, #02H            ;Timer0, Mode2
         MOV   TH0, #49h             ;initial value to generate 5 KHz
         MOV   IE,#82H               ;Enable ET0=1 & EA=1
         SETB  TR0                   ;start timer 0
         MOV   P0,#'A'               ;Display 'A' at PORT P0
         SJMP  AGAIN
         END
```

C Program

```
#include <reg51.h>
void timer0(void) interrupt 1 // ISR for interrupt number 1
{
    P2.1=~P2.1;                 //Toggle pin P2.1 for square wave
}

void main( )
{
    TMOD=0x02;                  //use Timer 0, mode 2
    IE=0x82;                    //Enable ET0=1 & EA=1
    TH0=0x49;                   //initial value to generate 5 KHz
    TR0=1;                      //start timer 0
    while (1)
    {
        P0="A";                 //send "A" to P0
    }                           //end of while
}                               //end of main
```

Example 5.18:

Write an ALP and C Program to generate two square waves of 25 KHz and 5 KHz at pins P0.1 and P0.2 respectively using timer 0 & timer 1 in Mode 2 in interrupt mode. Use XTAL = 22 MHz.

Solution:

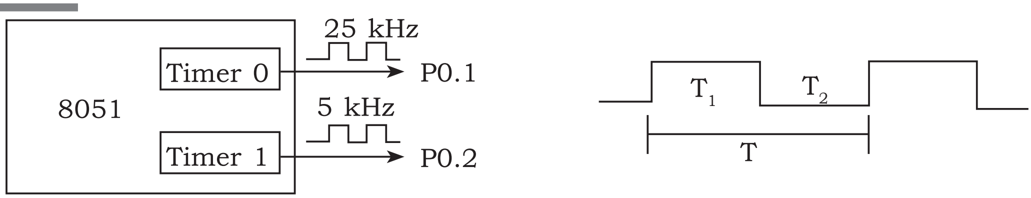


Fig. 5.1.7: Generating square wave on P0.1 & P0.2

For 5 KHz

We know that

$$T = \frac{1}{f} = \frac{1}{5KHz} = 0.2 \text{ ms}$$

$$\boxed{T = T_1 + T_2}$$

$$T_1 = T_2 = \frac{T}{2} = 0.1 \text{ ms}$$

Initial Count value:

$$TH1 = \text{Final value} - \frac{\text{Required delay} \times \text{crystal freqyency}}{12}$$

$$= 256 - \frac{1 \times 10^{-4} \text{s} \times 22 \times 10^{6}}{12} = 256 - 183 = 73$$

$$\boxed{TH1 = 49H}$$

For 25 KHz

We know that

$$\boxed{T = T_1 + T_2}$$

$$T = \frac{1}{f} = \frac{1}{25KHz} = 4 \times 10^{-5} \text{s} \qquad T_1 = T_2 = \frac{T}{2} = 2 \times 10^{-5} \text{s}$$

Initial Count value:

$$TH0 = \text{Final value} - \frac{\text{Required delay} \times \text{crystal freqyency}}{12}$$

$$= 256 - \frac{2 \times 10^{-5} \text{s} \times 22 \times 10^{6}}{12} = 256 - 37 = 219\text{d}$$

$$\boxed{TH0 = DBH}$$

IE register configuration

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

EA = 1: Enable interrupts
ET1 = 1: Enable timer 1 interrupts
ET0 = 1: Enable timer 0 interrupts

Fig. 5.1.8: IE register configuration

Load IE = 8AH

The instruction used is **MOV IE,#8AH OR SETB EA, SETB ET1 and SETB ET0**

ALP

```
        ORG   00H
        SJMP  MAIN          ;jump to main program
        ORG   000BH         ;ISR for Timer 0
        CPL   P0.1          ;Toggle pin P0.1 for square wave
        RETI                ;return from interrupt
        ORG   001BH         ;ISR for Timer 1
        CPL   P0.2          ;Toggle pin P0.2 for square wave
        RETI                ;return from interrupt
        ORG   30H           ;main program written from 30H
MAIN:   MOV   TMOD, #22H    ;Timer0 & Timer1 in Mode2
        MOV   IE,#8AH       ;Enable ET0=1, ET1=1 & EA=1
        MOV   TH0, #0DBH    ;initial value to generate 25 KHz
        MOV   TH0, #49h     ;initial value to generate 5 KHz
        SETB  TR0           ;start timer 0
        SETB  TR1           ;start timer 1
AGAIN:  SJMP  AGAIN
        END
```

Note: The instruction **SJMP HERE** executes continuously until an interrupt occurs, it jumps to Timer 0 or Timer 1 ISR and services it and then return to the main program i.e. **SJMP HERE**

C Program

```
#include <reg51.h>
void timer0(void) interrupt 1 // ISR for interrupt number 1
{
    P0.1=~P0.1;                 //Toggle pin P0.1 for square wave
}
void timer1(void) interrupt 3 // ISR for interrupt number 3
{
    P0.2=~P0.2;                 //Toggle pin P0.2 for square wave
}
void main( )
{
    TMOD=0x22;            //use Timer 0, Timer 1 in mode 2
    IE=0x8A;              //Enable ET0=1, ET1=1 & EA=1
    TH0=0xDB;             //initial value to generate 25 KHz
    TH1=0x49;             //initial value to generate 5 KHz
    TR0=1;                //start timer 0
    TR1=1;                //start timer 1
    while (1);            //wait for indefinite time
}                         //end of main
```

5.2 ADC0804 (Analog to Digital Converter)

5.2.1 Introduction

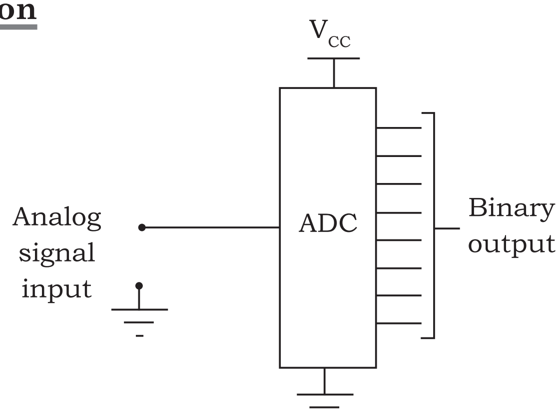


Fig. 5.2.1: Block diagram of ADC

- The ADC consists of an analog input, digital outputs, control lines and power supply as shown in figure. 6.3.1.
- ADC converts an input analog signal to a digital output.
- ADCs are mainly classified into

  i. Serial ADCs and

  ii. Parallel ADCs

5.2.2 Pin diagram of ADC0804

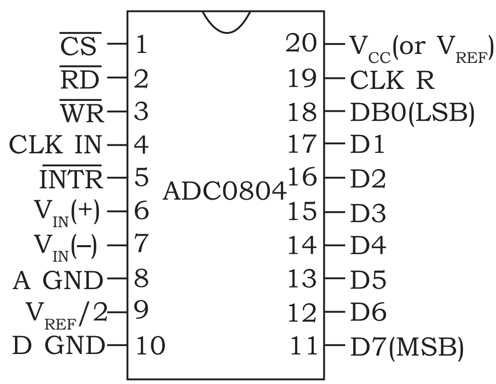


Fig. 5.2.2: Pin diagram of ADC0804

Table 5.2.1: Pin details of ADC0804

Pin No.	Pin Name	Pin Function
1	$\overline{CS}$ Chip Select	Chip select is an active low input used to activate the ADC0804 chip.
2	$\overline{RD}$	It is an input signal and active low. The ADC converts the analog input to its binary equivalent and holds it in an internal register. When $\overline{CS}$ = 0, if a high-to-low pulse is applied to the $\overline{RD}$ pin, the 8-bit digital output shows up at the D0-D7.
3	$\overline{WR}$	It is an active low input used to inform the ADC0804 to start the conversion process. If $\overline{CS}$ = 0 when $\overline{WR}$ makes a low to high transition, the ADC0804 starts converting the analog input value of Vin to an 8-bit digital number.

4& 19	CLK IN & CLK R	• CLK IN is an input pin connects to an external source when an external clock is used for timing. The ADC0804 chip has an internal clock generator. • The CLK IN & CLK R pins are connected to a resistor and capacitor to use internal clock generator. • The clock frequency is given by $f = \frac{1}{1.1\,RC}$
5	$\overline{INTR}$	• This is an active low output pin. • When $\overline{INTR}$ = 0, indicate end of conversion (EOC) to 8051.
6 & 7	$V_{IN(+)}$& $V_{IN(-)}$	• $V_{IN(+)}$ pin is connected with analog input to be converted to digital • $V_{IN(-)}$ pin is connected to ground • These are the differential analog input given by $V_{IN} = V_{IN(+)} - V_{IN(-)}$
8	A GND	Analog ground for analog signal
9	$V_{ref/2}$	It is an input voltage used for the reference voltage. If this pin is open, the analog input voltage for ADC is in the range of 0V to 5V.
10	D GND	Digital ground
18-11	D0-D7	D0-D7 are the 8-bit digital data output pins. $D_{out} = \frac{V_{in}}{\text{Step Size}}$
20	V_{CC}	VCC=5V. It is also used as a reference voltage when Vref/2 input is open.

5.2.3 Timing diagram for data conversion by ADC0804 chip

The following steps must be followed for data conversion by the ADC0804 chip.

1. Make CS=0 and send a low-to-high pulse to pin WR to start the conversion.

2. Keep monitoring the INTR pin. If INTR is low, the conversion is finished and we can go to the next step. If INTR is high, keep polling until it goes low.
3. After the INTR has become low, we make CS=0 and send a high-to-low pulse to the RD pin to get the data out of the ADC0804 IC chip. The timing diagram for this process is shown in figure. 5.2.3

Note: CS is set to low for both $\overline{RD}$ and $\overline{WR}$ pulses

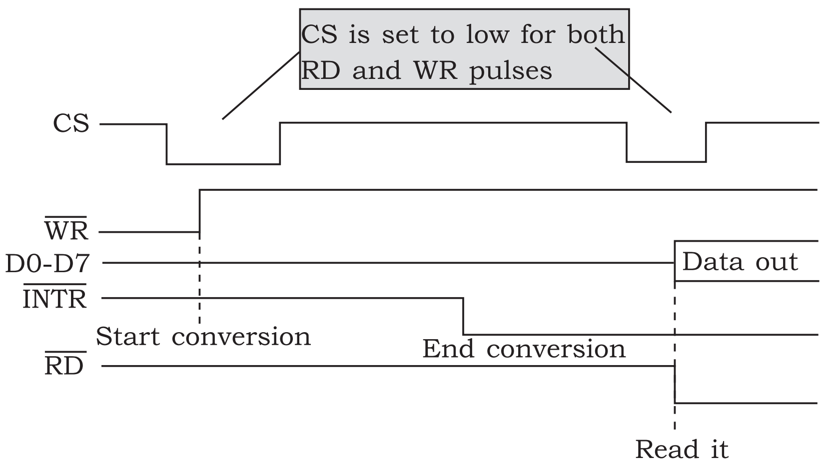


Fig. 5.2.3: Read and Write Timing for ADC0804

Example 5.19:

Write the schematic, algorithm and a program to interface a ADC 0804 to 8051

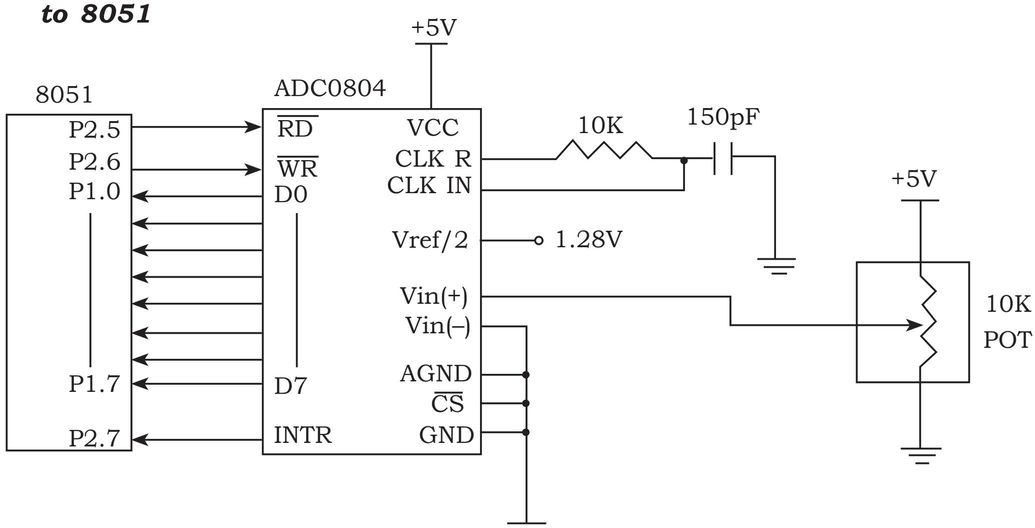


Fig. 5.2.4: 8051 connection to ADC0804 with self-clocking

ALGORITHM

1. Set P1 as input port and P2.7 as input pin
2. Send a start of conversion to ADC. $\overline{WR}$ = low to high signal.
3. Wait for end of conversion by reading in the INTR pin (INTR = EOC = 0)
4. Enable $\overline{RD}$ =0, so that converted data is put on D_0-D_7 lines
5. Read in the digital data on D_0-D_7 using P1
6. Call hexadecimal to ASCII conversion.
7. Call data display to display the ASCII values on LCD.

Assembly language interfacing programming

```
        RD      BIT  P2.5       ;RD
        WR      BIT  P2.6       ;WR (start conversion)
        INTR    BIT  P2.7       ;end-of-conversion
        MYDATA  EQU  P1         ;P1.0-P1.7=D0-D7 of the ADC804
        MOV     P1,#0FFH        ;make P1 » input
        SETB    INTR
BACK:   CLR     WR              ;WR-0
        SETB    WR              ;WR=1 L-to-H to start conversion
HERE:   JB      INTR,HERE       ;wait for end of conversion
        CLR     RD              ;conversion finished,enable RD
        MOV     A,MYDATA        ;read the data
        ACALL   CONVERSION      ;hex-to-ASCII conversion
        ACALL   DATA_DISPLAY    ;display the data
        SETB    RD              ;make RD=1 for next round
        SJMP    BACK
```

C language interfacing programming

```
#include <reg51.h>
sbit RD = P2^5;
sbit WR = P2^6;
Sbit INTR = P2^7;
sfr MYDATA = P1;
void main{)
```

```
{
        unsigned char value;
        MYDATA = OxFF;                      //make PI and input
        INTR = 1;                           //make INTR and input
        RD = 1;                             //set RD high
        WR = 1;                             //set WR high
        while(1)
            {
                WR = 0;                     //send WR pulse
                WR = 1;                     //L-to-H(Start Conversion)
                while(INTR == 1);           //wait for EOC
                RD = 0;                     //send RD pulse
                value = MYDATA;             //read value
                ConvertAndDisplay(value);   //Display value on LCD
                RD = 1;
            }
}
```

5.3 LCD Interfacing

5.3.1 Introduction

LCD is finding widespread use replacing LEDs

- 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.

5.3.2 LCD Pins

The pins of alphanumeric LCD module which help in interfacing with the microcontroller are given in table 6.2.1.

Table 5.3.2: LCD Commands Code

Pin	Symbol	I/O	Descriptions
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	DBO	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

used by the LCD to latch information presented to its data bus

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

5.3.3 LCD Commands

Table 5.3.1: LCD Commands Code

Code (Hex)	Command to LCD Instruction
1	Clear display screen
2	Return home
4	Decrement cursor (shift cursor to left)
6	Increment cursor (shift cursor to right)
5	Shift display right
7	Shift display left
8	Display off, cursor off
A	Display off, cursor on
C	Display on, cursor off
E	Display on, cursor blinking
F	Display on, cursor blinking
10	Shift cursor position to left

14	Shift cursor position to right
18	Shift the entire display to the left
1C	Shift the entire display to the right
80	Force cursor to beginning of 1st line
C0	Force cursor to beginning of 2nd line
38	2 lines and 5x7 matrix

5.3.4 LCD Timing for READ

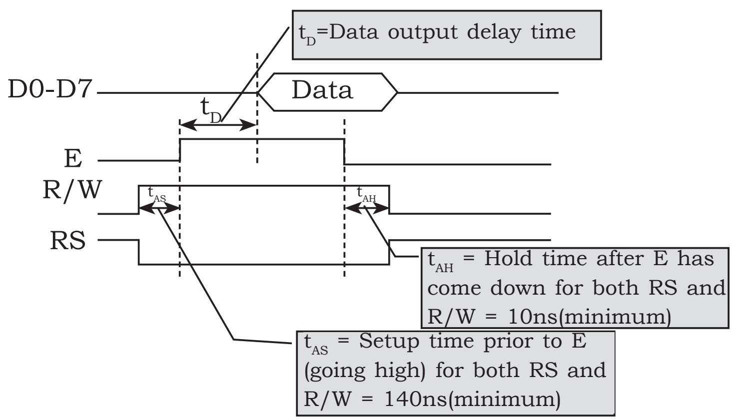


Fig. 5.3.1: LCD Timing for READ

Note*:* Read requires an L-to-H pulse for the E pin

5.3.5 LCD Timing for WRITE

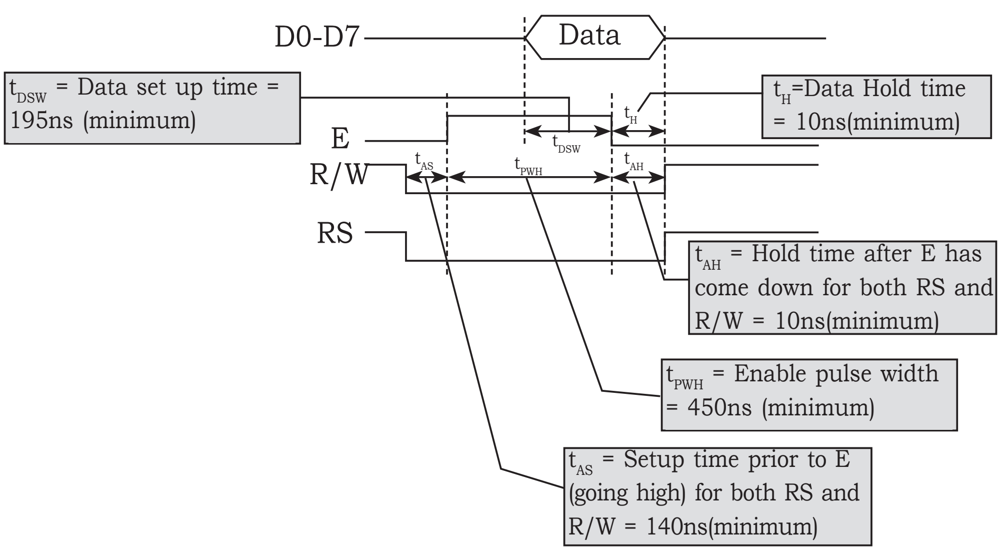


Fig. 5.3.2: LCD Timing for write

Example 5.14:

Write the schematic, algorithm and a program to interface a alphanumeric LCD to 8051 and to display 'INDIA'

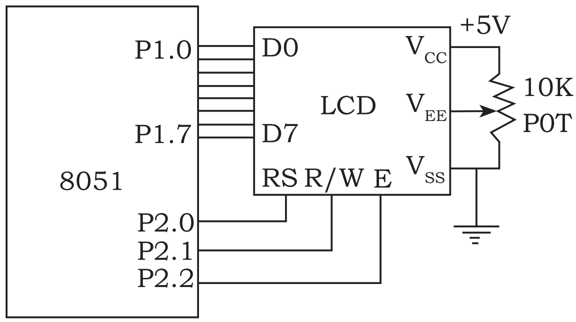


Fig. 5.3.3: LCD Interfacing with 8051

ALGORITHM

1. Initialize the LCD using the lcdcmd subroutine.
2. Send the commands 38H, 0EH, 01H, 06H and 86H to PORT P1
3. Make RS = 0 & R/W = 0
4. Make enable Pin E=1 for a 1 ms.
5. Make enable Pin E=0
6. Return to calling program
7. Send the ASCII value **'I', 'N', 'D', 'I', 'A'** to PORT P1 and call lcddata subroutine with 250 ms delay.
8. Make RS = 1 & R/W = 0
9. Make enable Pin E=1 for a 1 ms.
10. Make enable Pin E=0
11. Return to calling program

Assembly language interfacing programming

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.

;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 00H
MOV A,#38H              ;Initialize 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,#84H              ;cursor at line 1, pos. 4
ACALL COMNWRT           ;call command subroutine
ACALL DELAY             ;give LCD some time
MOV A,#'I'              ;display letter I
ACALL DATAWRT           ;call display subroutine
ACALL DELAY             ;give LCD some time
MOV A,#'N'              ;display letter N
ACALL DATAWRT           ;call display subroutine
ACALL DELAY             ;give LCD some time
MOV A,#'D'              ;display letter D
ACALL DATAWRT           ;call display subroutine
ACALL DELAY             ;give LCD some time
MOV A,#'I'              ;display letter I
ACALL DATAWRT           ;call display subroutine
ACALL DELAY             ;give LCD some time
MOV A,#'A'              ;display letter A
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
```

C language interfacing programming

```
#include <reg51.h>
sfr ldata=0x90;                     //P1=LCD data pins
sbit RS=P2^0;                       // Register Select
sbit RW=P2^1;                       // Read (1) & Write (0)
sbit E=P2^2;                        // LCD enable

void lcdmd(unsigned char i);
void lcddata(unsigned char i);
void delay(unsigned char i);

void main()
{
        lcdcmd(0x38);
        delay(250);
        lcdcmd(0x0E);
```

```
        delay(250);
        lcdcmd(0x01);
        delay(250);
        lcdcmd(0x06);
        delay(250);
        lcdcmd(0x86);
        delay(250);
        lcddata('I');
        delay(250);
        lcddata('N');
        delay(250);
        lcddata('D');
        delay(250);
        lcddata('I');
        delay(250);
        lcddata('A');
}
void lcdcmd(unsigned char value)
{
    ldata = value;                  //put the value on the pins
    RS=0;
    RW=0;
    E=1;                            //strobe the enable pin
    delay(1);
    E=0;
    return;
}
void lcddata(unsigned char value)
{
    ldata = value;                  //put the value on the pins
    RS=1;
    RW=0;
    E=1;                            //strobe the enable pin
    delay(1);
    E=0;
    return;
}
void delay(unsigned int count)
{
    unsigned int i, j;
```

```
        for(i=0;i<count;i++)
            for(j=0;j<1275;j++);
    }
```

Example 5.15:

Write the schematic, algorithm and a program to interface a alphanumeric LCD to 8051 and to display 'HELLO'

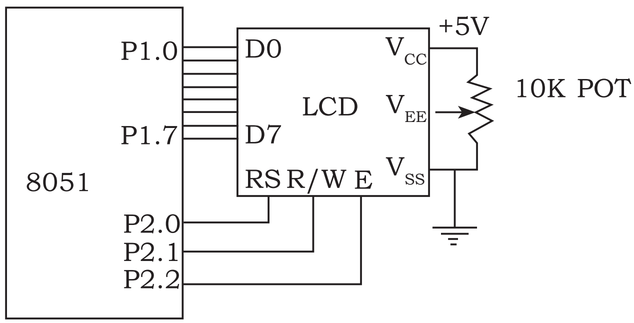


Fig. 5.3.4: LCD Interfacing with 8051

ALGORITHM

1. Initialize the LCD using the lcdcmd subroutine.
2. Send the commands 38H, OEH, 01H, 06H and 86H to PORT P1
3. Make RS = 0 & R/W = 0
4. Make enable Pin E=1 for a 1 ms.
5. Make enable Pin E=0
6. Return to calling program
7. Send the ASCII value **'H', 'E', 'L', 'L', 'O'** to PORT P1 and call lcddata subroutine with 250 ms delay.
8. Make RS = 1 & R/W = 0
9. Make enable Pin E=1 for a 1 ms.
10. Make enable Pin E=0
11. Return to calling program

Assembly language interfacing programming

```
;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 00H
MOV A,#38H              ;initialize 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,#'H'              ;display letter H
ACALL DATA_DISPLAY
MOV A,#'E'              ;display letter E
ACALL DATA_DISPLAY
MOV A,#'L'              ;display letter L
ACALL DATA_DISPLAY
MOV A,#'L'              ;display letter L
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
;read command reg and check busy flag
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
```

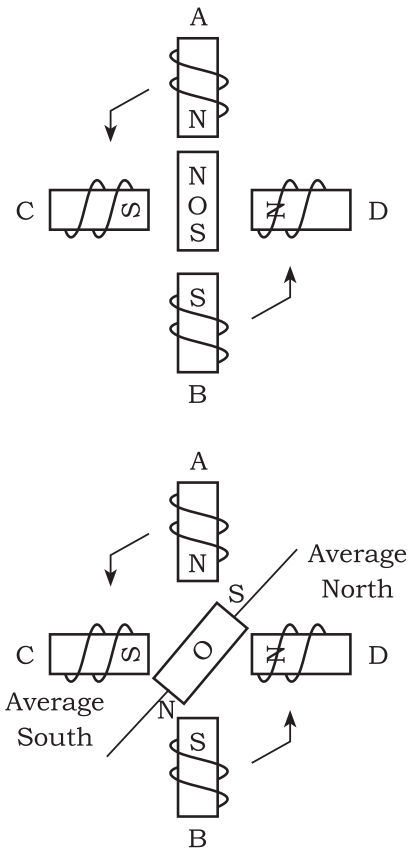
C language interfacing programming

```
    #include <reg51.h>
    sfr ldata=0x90;                    //P1=LCD data pins
    sbit RS=P2^0;                      // Register Select
    sbit RW=P2^1;                      // Read (1) & Write (0)
    sbit E=P2^2;                       // LCD enable
    void lcdmd(unsigned char i);
    void lcddata(unsigned char i);
    void delay(unsigned char i);
    void main( )
    {
       lcdcmd(0x38);
       delay(250);
       lcdcmd(0x0E);
       delay(250);
       lcdcmd(0x01);
       delay(250);
       lcdcmd(0x06);
       delay(250);
       lcdcmd(0x86);
       delay(250);
       lcdcmd('H');
```

```
    delay(250);
    lcdcmd('E');
    delay(250);
    lcdcmd('L');
    delay(250);
    lcdcmd('L');
    delay(250);
    lcdcmd('O');
}

void lcdcmd(unsigned char value)
{
    ldata=value;                    //put the value on the pins
    RS=0;
    RW=0;
    E=1;                            //strobe the enable pin
    delay(1);
    E=0;
    return;
}
void lcddata(unsigned char value)
{
    ldata=value;                    //put the value on the pins
    RS=1;
    RW=0;
    E=1;                            //strobe the enable pin
    delay(1);
    E=0;
    return;
}

void delay(unsigned int count)
{
    unsigned int i, j;
    for(i=0;i<count;i++)
        for(j=0;j<1275;j++);
}
```

5.4 Stepper Motor

5.4.1 Introduction



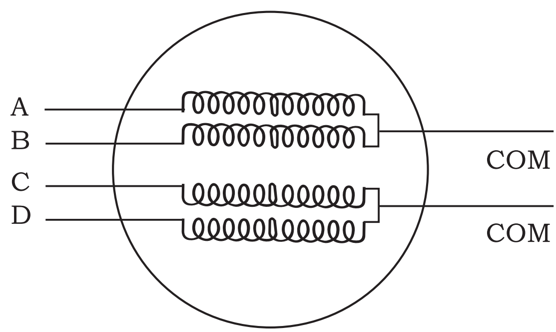


Fig. 5.4.1 ROTOR Alignment ***Fig. 5.4.2: Stator Windings configuration***

- A stepper motor translates electrical pulses into mechanical movements. Stepper motors have a permanent magnet rotor called shaft surrounded by a stator.
- A stepper motor or step motor or stepping motor is a brushless DC electric motor that divides a full rotation into a number of equal steps. The motor's position can then be commanded to move and hold at one of these steps.

The most common stepper motor has four stator windings that are paired with center-tapped common as shown in figure. This type of stepper motor is commonly referred to as a four-phase or unipolar stepper motor. The center tap allows a change of current direction in each of two coils when a winding is grounded, thereby resulting in a polarity change of the stator. Notice that while a conventional motor shaft runs freely, the stepper motor shaft moves in fixed repeatable increment, which alone one to move it to a precise position.

The direction of the rotation is dictated by the stator poles. The stator poles are determined by the current sent through the wire coils. As the direction of the current is changed, the polarity is also changed causing the reverse motion of the rotor. The stepper motor has a total of 6 leads: 4 leads representing the four stator winding and 2 common for the center-tapped leads.

The figure explains the stepper motor control system.

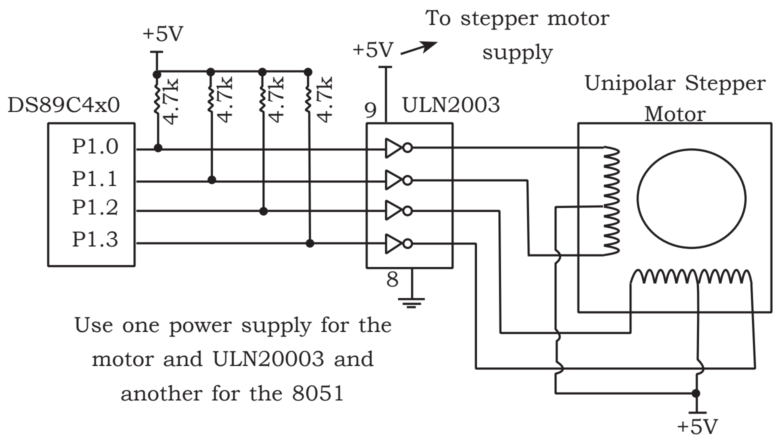


Fig. 5.4.3 Circuit diagram of stepper motor control system

A stepper motor is brushless and synchronous motor which divides the complete rotation into number of steps. Each stepper motor will have some fixed step angle and motor rotates at this angle. The main principle of this circuit is to rotate the stepper motor step wise at a particular step angle. The ULN2003A IC is used to drive stepper motor as the controller cannot provide current required by the motor.

The circuit mainly consists of:

- 8051 micro controller
- ULN2003A
- Stepper Motor

The Motor is connected to the Port 2 of the microcontroller through a driver IC. The ULN2003A is a current driver IC. It is used to drive the current of the stepper motor as it requires more than 60mA current. It is an array of Darlington pairs. It consists of seven pairs of Darlington arrays with common emitter. The IC consists of 16 pins in which 7 are input pins, 7 are output pins and remaining are VCC and Ground. The first four input pins are connected to the microcontroller. In the same way, four output pins are connected to the stepper motor.

Stepper motor has 6 pins. In these six pins, 2 pins are connected to the supply of 12V and the remaining are connected to the output of the stepper motor. Stepper motor rotates at a given step angle. Each step in rotation is a fraction of full cycle.

The stepper motors will have stator and rotor. Rotor has permanent magnet and stator has coil. The basic stepper motor has 4 coils with 90 degrees rotation step. These four coils are activated in the cyclic order. The figure shown gives the direction of rotation of the shaft of the stepper motor. There are different methods to drive a stepper motor. Some of these are explained below.

Full Step Drive (1.8^0/Step): In this method two coils are energized at a time. Thus, here two opposite coils are excited at a time.

Half Step Drive (0.9^0/Step): In this method coils are energized alternatively. Thus it rotates with half step angle. In this method, two coils can be energized at a time or single coil can be energized. Thus it increases the number of rotations per cycle.

Table 5.4.1: Full-Step: 4-Step Sequence

Clockwise	Step	Winding A	Winding B	Winding C	Winding D	Counter-Clockwise
↓	1	1	0	0	1	↑
	2	1	1	0	0	
	3	0	1	1	0	
	4	0	0	1	1	

Table 5.4.2: Half-Step: 8-Step Sequence

Clockwise	Step	Winding A	Winding B	Winding C	Winding D	Counter-Clockwise
↓	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	

Table 5.4.3: Wave drive 4-Step sequence

Clockwise	Step	Winding A	Winding B	Winding C	Winding D	Counter-Clockwise
↓	1	1	0	0	0	↑
	2	0	1	0	0	
	3	0	0	1	0	
	4	0	0	0	1	

Stepper motor controller circuit advantages:

- It consumes less power.
- It requires low operating voltage.

Stepper Motor Applications:

- Robotics
- Mechatronics
- Position Control etc.
- Dot matrix printers
- Disk drives

5.4.2 Stepper motor controller

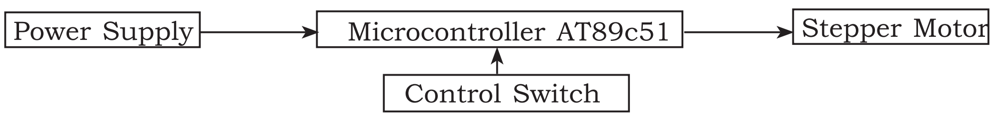


Fig. 5.4.4: Block diagram of a stepper motor controller system.

A stepper motor controller (SMC) is an interfacing circuit which is used in between the stepper motor & the load, which is to be controlled. A SMC based 8051 microcontroller to control the rotation of a DC stepper motor in clockwise and anti-clockwise directions is shown in the figure 6.65. The controller is simple and easy to construct, and can be used in many applications including machine control and robotics for controlling the axial rotation.

Example 5.16:

Sketch the schematic for interfacing a stepper motor to 8051

Solution:

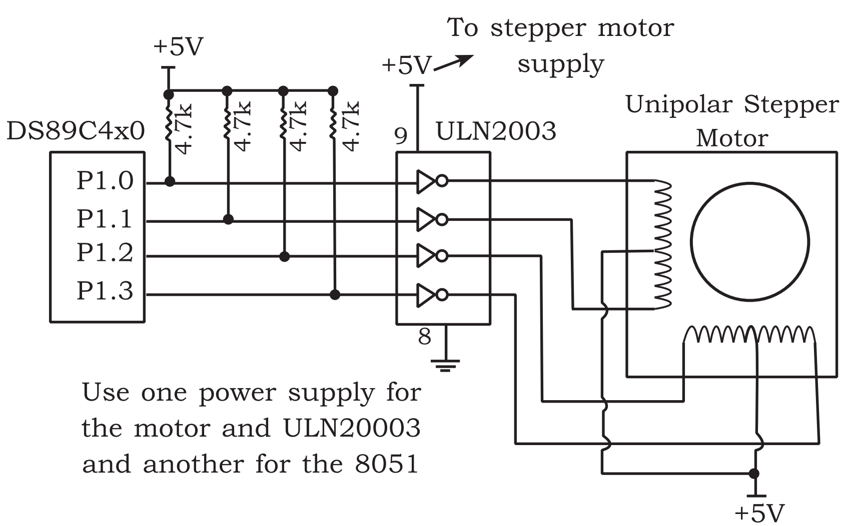


Fig. 5.4.5: Circuit diagram of stepper motor control system

Example 5.17:

Write the schematic, algorithm and a program to interface a stepper motor to 8051 and to rotate the motor in clock wise direction using normal 4 step sequence

Solution:

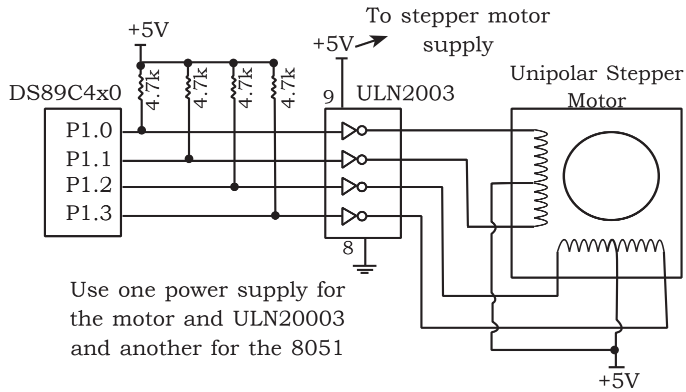


Fig. 5.4.6: Circuit diagram of stepper motor control system

Table 5.4.4: Full-Step: 4-Step Sequence

Clockwise	Step	Winding A	Winding B	Winding C	Winding D	Counter-Clockwise
↓	1	1	0	0	1	↑
	2	1	1	0	0	
	3	0	1	1	0	
	4	0	0	1	1	

- To generate the full-step 4-step sequence, the initial values to be used for clockwise are **66H, 33H, 99H** and **CCH**.
- To generate the full-step 4-step sequence, the initial values to be used for counter-clockwise are **CCH, 99H, 33H** and **66H**.

Algorithm

1. For **clockwise direction**, load the sequence **66H** into **P1**
2. Call 100 msdelay
3. Load the sequence **33H into P1**
4. Call 100 msdelay
5. Load the sequence **99H into P1**
6. Call 100 msdelay
7. Load the sequence **CCH into P1**
8. Call 100 msdelay
9. Repeat from step 1

Assembly language interfacing programming

Method 1	Method 2
ORG 00H	ORG 00H
BACK: MOV A,#66H	**BACK:** MOV P1, #66H
MOV P1,A	ACALL **DELAY**
RR A	MOV P1, #33H
ACALL **DELAY**	ACALL **DELAY**
	MOV P1, #99H
SJMP **BACK**	ACALL **DELAY**
	MOV P1, #CCH
DELAY:MOV R0,#255	ACALL **DELAY**
UP1: MOV R1,#255	SJMP BACK
UP2: DJNZ R1,**UP2**	**DELAY:** MOV R0,#255
DJNZ R0,**UP1**	**UP1:** MOV R1,#255
RET	**UP2:** DJNZ R1,UP2
END	**DJNZ** R0,UP1
	RET
	END

C language interfacing programming

```
#include <reg51.h>
void delay(unsigned int i);
void main ( )
{
         while (1)
         {
             P1 = 0x66;
             delay (100);
             P1 = 0x33H;
             delay (100);
             P1 = 0x99;
             delay (100);
             P1 = 0xCC;
             delay (100);
         }
}
void delay(unsigned int count)
{
         unsignedint i, j;
         for(i=0;i<count;i++)
         for(j=0;j<1275;j++);
}
```

Example 5.18:

Write the schematic, algorithm and a program to interface a stepper motor to 8051 and to rotate the motor in anti-clock wise direction using wave drive sequence

Solution:

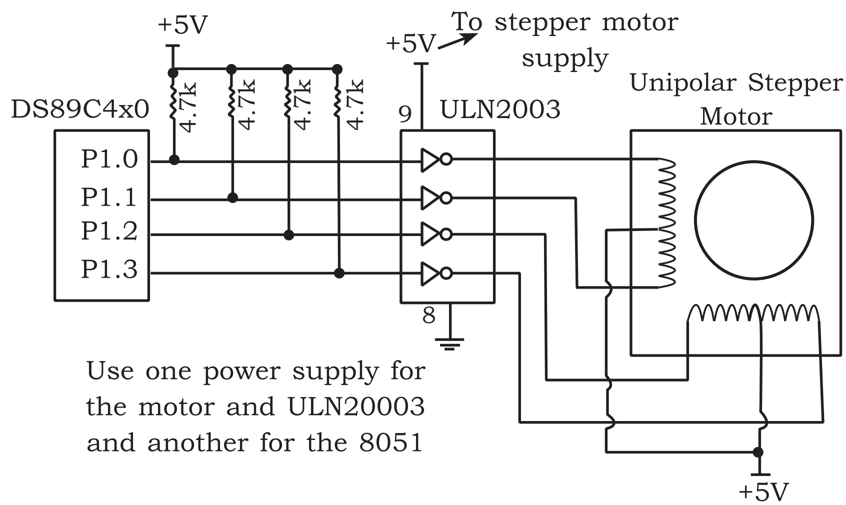


Fig. 5.4.7: Circuit diagram of stepper motor control system

Table 5.4.5: Wave drive 4-Step sequence

Clockwise	Step	Winding A	Winding B	Winding C	Winding D	Counter-Clockwise
↓	1	1	0	0	0	↑
	2	0	1	0	0	
	3	0	0	1	0	
	4	0	0	0	1	

In wave-drive 4-step sequence, the initial values to be used for **clockwise** are **8H, 4H, 2H and 1H.**

In wave-drive 4-step sequence, the initial values to be used for **counter-clockwise** are **1H, 2H, 4H and 8H**.

Algorithm

1. PORT 1 is used as output port
2. Load the sequence **01H into P1**
3. Call 100 msdelay
4. Load the sequence **02H into P1**
5. Call 100 msdelay

6. Load the sequence **04H into P1**
7. Call 100 msdelay
8. Load the sequence **08H into P1**
9. Call 100 msdelay
10. Repeat from step 2

Assembly language interfacing programming

<table>
<tr><th>Method 1</th><th>Method 2</th></tr>
<tr><td><pre>
        ORG 00H
BACK: MOV A,#01H
        MOV P1,A
        RL   A
        ACALL   DELAY

        SJMP   BACK

DELAY: MOV R0,#255
UP1:   MOV R1,#255
UP2:   DJNZ R1,UP2
        DJNZ R0,UP1
        RET
        END
</pre></td><td><pre>
        ORG 00H
BACK: MOV P1, #01H
        ACALL   DELAY
        MOV   P1, #02H
        ACALL   DELAY
        MOV   P1, #04H
        ACALL   DELAY
        MOV   P1, #08H
        ACALL   DELAY

        SJMP   BACK

DELAY: MOV R0,#255
UP1:   MOV R1,#255
UP2:   DJNZ   R1,UP2
        DJNZ   R0,UP1
        RET
        END
</pre></td></tr>
</table>

C Program

```
#include <reg51.h>
void delay(unsigned int i);
void main ( )
{
   while (1)
   {
      P1=0x11;
      delay (100);
      P1=0x22;
      delay (100);
```

```
        P1=0x44;
        delay (100);
        P1 = 0x88;
        delay (100);
    }
}
void delay(unsigned int count)
{
unsigned int i, j;
for(i=0;i<count;i++)
    for(j=0;j<1275;j++);
}
```

Example 5.19:

A switch is connected to pin P2.7. Write a C program to monitor the status of SW and perform the following:

i. If SW = 0, the stepper motor moves clockwise.

ii. If SW = 1, the stepper motor moves counterclockwise.

Solution:

- To generate the full-step 4-step sequence, the initial values to be used for clockwise are **66H, 33H, 99H** and **CCH**.
- To generate the full-step 4-step sequence, the initial values to be used for counter-clockwise are **CCH, 99H, 33H** and **66H**.

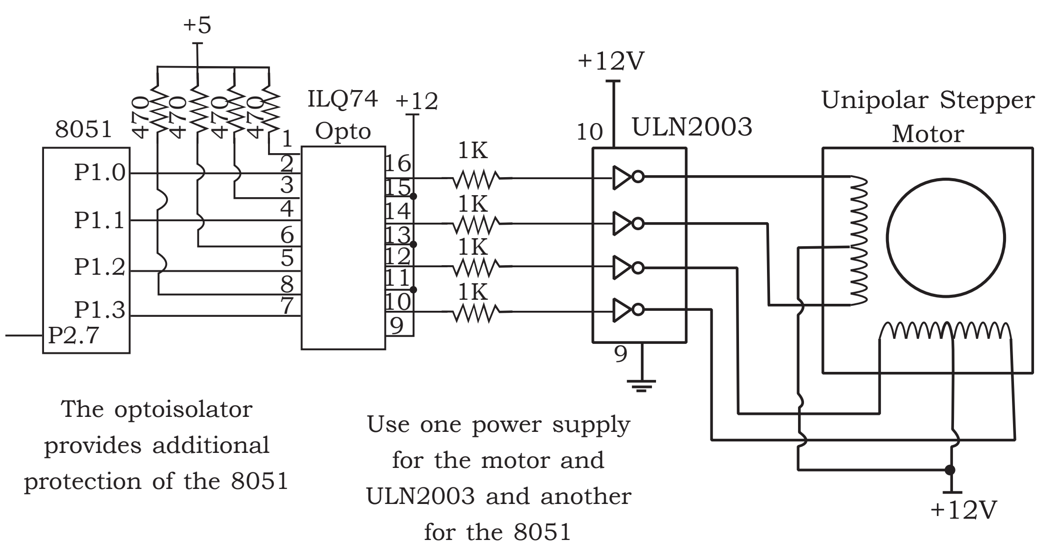


Fig. 5.4.8: Circuit diagram of stepper motor control system

Algorithm

1. If SW = 0, rotate the motor in clockwise direction
2. Load the sequence **66H into P1**
3. Call 100 msdelay
4. Load the sequence **33H into P1**
5. Call 100 msdelay
6. Load the sequence **99H into P1**
7. Call 100 msdelay
8. Load the sequence **CCH into P1**
9. Call 100 msdelay
10. Else (i.e. if SW =1)
11. Rotate motor in **counterclockwise** direction **i.e**. load the sequence **CCH** into **P1**
12. Call 100 msdelay
13. Load the sequence **99H into P1**
14. Call 100 msdelay
15. Load the sequence **33H into P1**
16. Call 100 msdelay
17. Load the sequence **66H into P1**
18. Call 100 msdelay
19. Repeat from step 1

Assembly language interfacing programming

Method 1	Method 2
ORG 00H	ORG 00H
SETB P2.7	SETB P2.7
MOV A, #66H	
NEXT: JNB P2.7, **CLOCKWISE**	**BACK:** JNB P2.7, **CLOCKWISE**
RL A	
MOV P1, A	MOV P1, #CCH
ACALL **DELAY**	ACALL **DELAY**
SJMP **NEXT**	MOV P1, #99H
CLOCKWISE: RR A	ACALL **DELAY**
MOV P1, A	MOV P1, #33H
ACALL **DELAY**	ACALL **DELAY**
SJMP **NEXT**	MOV P1, #66H
DELAY: MOV R0, #255	ACALL **DELAY**
UP2: MOV R1, #255	SJMP **BACK**

Continued...

UP1:	DJNZ R1, **UP1**	**CLOCKWISE:**	
	DJNZ R0, **UP2**		MOV P1, #66H
	RET		ACALL **DELAY**
	END		MOV P1, #33H
			ACALL **DELAY**
			MOV P1, #99H
			ACALL **DELAY**
			MOV P1, #CCH
			ACALL **DELAY**
			SJMP **BACK**
		DELAY:	MOV R0,#255
		UP1:	MOV R1,#255
		UP2:	DJNZ R1,**UP2**
			DJNZ R0,**UP1**
			RET
			END

C language interfacing programming

```
#include <reg51.h>
sbit SW=P2^7;
void delay(unsigned int i);
void main ( )
{
        SW= 1;
        while (1)
        {
            if (SW==0)
            {
                P1 = 0x66;
                delay (100);
                P1 = 0x33;
                delay (100);
                P1 = 0x99;
                delay (100);
                P1 = 0xCC;
                delay (100);
            }
        else
```

```
            {
                P1 = 0xCC;
                delay (100);
                P1 = 0x99;
                delay (100);
                P1 = 0x33;
                delay (100);
                P1 = 0x66;
                delay (100);
            }
        }
}
void delay(unsigned int count)
{
        unsignedint i, j;
        for(i=0;i<count;i++)
        for(j=0;j<1275;j++);
}
```

Example 5.20:

Write a program to rotate a motor 80^o in the clockwise direction. The motor has a step angle of 2^o. Use the 4-step sequence.

Solution:

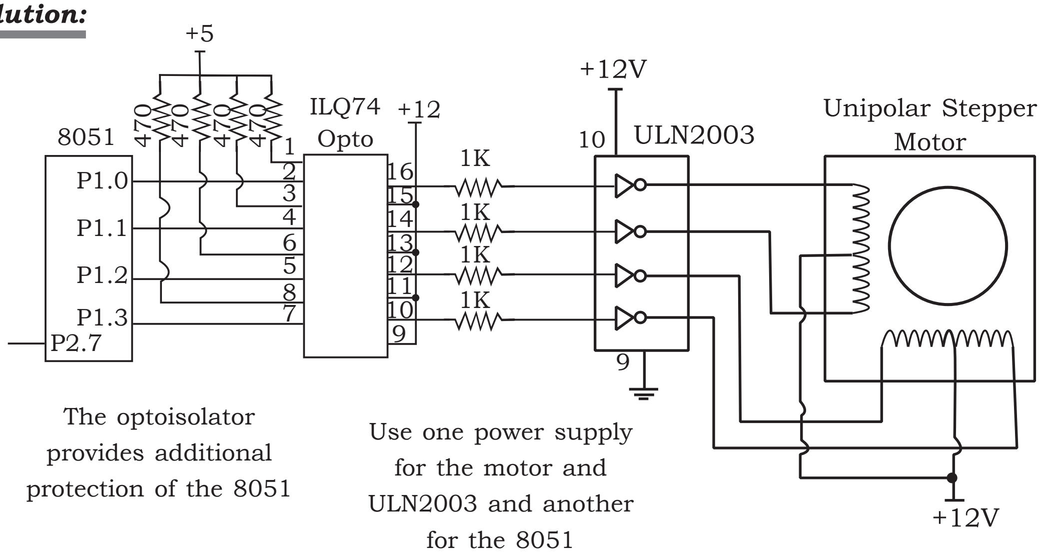


Fig. 5.4.9: Circuit diagram of stepper motor control system

Number of steps to rotate the motor for 80° is

$$\frac{80^\circ}{\text{Step Angle}} = \frac{80^\circ}{2^\circ} = 40 \text{ Steps}$$

Algorithm

1. Initialize a counter with 40 steps
2. Load the sequence **66H into P1**
3. Call delay
4. For clockwise direction rotate the phase sequence right
5. Decrement counter (R0) and repeat from step 1 until counter is zero
6. If counter is zero i.e. R0=0, Stop.

Assembly language interfacing programming

```
        ORG     00H
        MOV A,  #66H
        MOV R0, #40
UP:     RR  A
        MOV P1, A
        ACALL DELAY
        DJNZ R0, UP
DELAY:  MOV R2, #255
UP2:    MOV R1, #255
UP1:    DJNZ R1, UP1
        DNJZ R2, UP2
RET
END
```

Clanguage interfacing programming

```
#include <reg51.h>
void delay(unsigned int i);
void main ( )
{
        unsigned char counter, sequence, i;
        counter=40;
        sequence=0x66;
        for(i=0; i<counter: i++)
        {
            P1=sequence;
            sequence=sequence>>1;
            delay(100);
        }
```

```
        while (1);
}
void delay(unsigned int count)
{
        unsignedint i, j;
        for(i=0;i<count;i++)
        for(j=0;j<1275;j++);
}
```

Example 5.21:

Write a program to rotate a motor 80° in the clockwise direction. The motor has a step angle of 2°. Use the 4-step sequence.

Solution:

Algorithm

1.	Initialize a counter with 40 steps
2.	Load the sequence **66H into P1**
3.	Call delay
4.	For clockwise direction rotate the phase sequence right
5.	Decrement counter (R0) and repeat from step 1 until counter is zero
6.	If counter is zero i.e. R0=0, Stop.

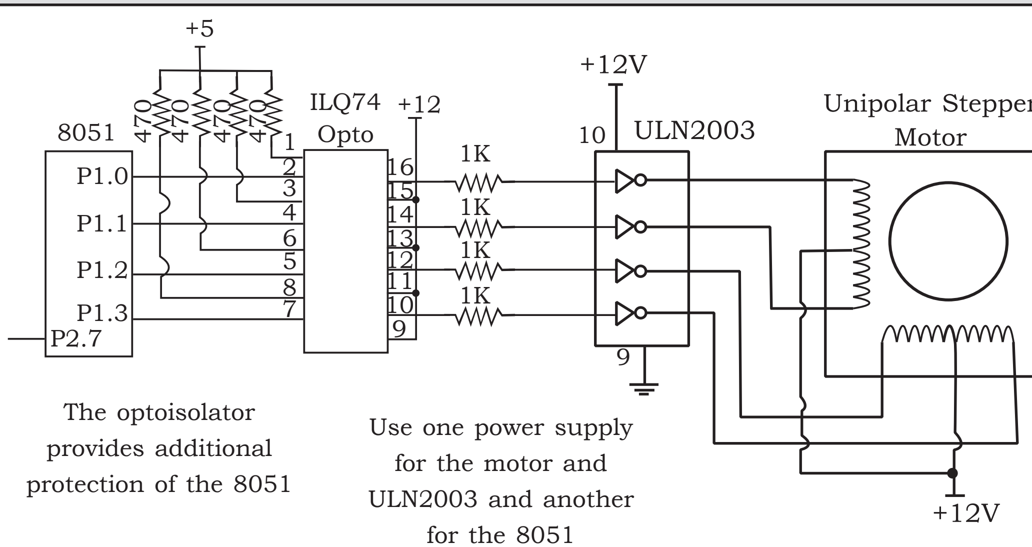


Fig. 5.4.10: Circuit diagram of stepper motor control system

Assembly language interfacing programming

```
ORG     00H
MOV A, #66H
```

```
            MOV R0, #40
UP:         RR  A
            MOV P1, A
            ACALL DELAY
            DJNZ R0, UP
            SJMP EXIT
DELAY:      MOV R2, #255
UP2:        MOV R1, #255
UP1:        DJNZ R1, UP1
            DJNZ R2, UP2
            RET
EXIT:
            END
```

C language interfacing programming

```
#include <reg51.h>
void delay(unsigned int i);
void main ( )
{
        unsigned char counter, sequence, i;
        counter=40;
        sequence=0x66;
        for(i=0; i<counter: i++);
        {
        P1=sequence;
        sequence=sequence>>1;
        delay(100);
        }
}
void delay(unsigned int count)
{
        unsignedint i, j;
        for(i=0;i<count;i++)
        for(j=0;j<1275;j++);
}
```

Example 5.21:

Write a program to rotate the stepper motor continuously

i. Clockwise using the wave drive 4-step sequence

ii. Clockwise using the half-step 8-step sequence.

Use the sequence values saved in the program ROM location.

Solution:

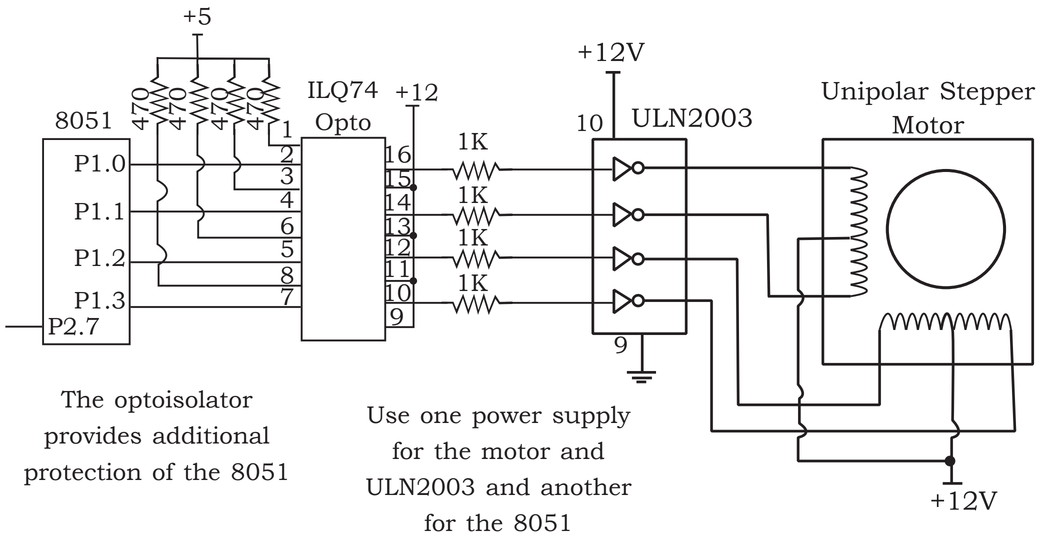


Fig. 5.4.11: Circuit diagram of stepper motor control system

i. Clockwise using the wave drive 4-step sequence

Algorithm

1. Initialize a counter R0 for 044 steps
2. Initialize the starting address of the sequence table
3. Get value from the sequence table and output to port P1
4. Increment the DPTR pointer
5. Decrement counter (R0) and repeat from step 3 until counter is zero
6. If counter is zero i.e. R0=0, repeat from step 1.

Assembly language interfacing programming

```
            ORG     00H
UP:         MOV R0, #04
            MOV DPTR, #0200H
REPEAT:     CLR A
            MOVC A,@A+DPTR
            MOV P1, A
            ACALL DELAY
            INC DPTR
            DJNZ R0, REPEAT
            SJMP UP

DELAY:      MOV R2, #255
UP2:        MOV R1, #255
```

```
UP1:        DJNZ R1, UP1
            DJNZ R2, UP2
            RET

            ORG 0200H
            DB  8,4,2,1
            END
```

C language interfacing programming

```
#include <reg51.h>
void main ( )

{
        code unsigned char sequence[]={08,04,02,01};
        unsigned char i;
        while(1)
        {
            for(i=0;i<4;i++)
            {   P1=sequence[i];
                delay(100);
            }
        }
}
void delay(unsigned int count)
{
        unsignedint i, j;
        for(i=0;i<count;i++)
        for(j=0;j<1275;j++);
}
```

ii. Clockwise using the half-step 8-step sequence.

Algorithm

1. Initialize a counter R0 for 08 steps
2. Initialize the starting address of the sequence table
3. Get value from the sequence table and output to port P1
4. Increment the DPTR pointer
5. Decrement counter (R0) and repeat from step 3 until counter is zero
6. If counter is zero i.e. R0=0, repeat from step 1.

Assembly language interfacing programming

```
            ORG  00H
UP:         MOV  R0, #08
```

```
            MOV   DPTR, #0100H
REPEAT:     CLR   A
            MOVC  A,@A+DPTR
            MOV   P1, A
            ACALL DELAY
            INC   DPTR
            DJNZ  R0, REPEAT
            SJMP  UP

DELAY:      MOV   R2, #255
UP2:        MOV   R1, #255
UP1:        DJNZ  R1, UP1
            DJNZ  R2, UP2
            RET

      ORG   0100H
      DB    09, 08, 0CH, 04, 06,02,03,01
            END
```

C language interfacing programming

```
#include <reg51.h>
void main ( )
{
        code unsigned char sequence[]={09,08,0x0C,04,06,02,03,01};
        unsigned char i;
        while(1)
        {
            for(i=0;i<8;i++)
            {   P1=sequence[i];
                delay(100);
            }
        }
}
void delay(unsigned int count)
{
        unsigned int i, j;
        for(i=0;i<count;i++)
        for(j=0;j<1275;j++);
}
```

8051 Programs

8051 Programs

1. **Show the stack contents, SP contents and contents of any register affected after each step of the following sequence of operation.**

Solution:

PROGRAM	SP	STACK CONTENT	REGISTER CONTENT
MOV SP,#70H	70H	-	-
MOV R5,#30H	70H	-	R5=30H
MOV A,#44H	70H	-	A=44H
ADD A,R5	70H	-	A=74H
MOV R4,A	70H	-	R4=74H
PUSH 4	71H	74H	R4=74H
PUSH 5	72H	30H	R5=30H
POP 4	71H	-	R4=30H

2. **Write a program to logically OR the contents of ports P1 & P2 and put the result in external RAM location.**

Solution:

```
ORG 00H
MOV DPTR,#0100H
MPV P1,#0FFH
MOV P2,#0FFH
MOV A,P1
MOV R0,P1
ORL A,R2
```

```
MOVX @DPTR,A
END
```

3. Find the sum of the values 79h,F5h & E2h; put the sum in registers R0 (low byte) & R5 (higher byte).

Solution:

```
ORG 00H
MOV R0,#00H
MOV A,#79H
ADD A,#0F5H
ADDC A,#0E2H
MOV R5,A
JNC NOCARRY
INC R0
NOCARRY: END
```

4. Write a program to add 10 BCD numbers stored in successive M/M locations from 20H in internal RAM locations & store the result at address 40H & 41H.

Solution:

```
ORG 00H
MOV R0,#20H
MOV R2,#0AH
MOV R3,#00H
MOV A,@R0
UP: INC R0
ADDC A,@R0
DA A
JNC NOCARRY
INC R3
DJNZ R2,UP
MOV 41H,A
NOCARRY: MOV 40H,R3
END
```

5. Find the status of carry flag after the following code sequence.

Solution:

i) CLR A	ii) CLR C	iii) CLR C
ADD A,#0FFH	JNC OVER	JC OVER
JNC OVER CPL C	SETB C	CPL C
OVER:	OVER:	OVER:

i) CLR A	ii) CLR C	iii) CLR C
ADD A,#0FFH	JNC OVER	JC OVER
JNC OVER CPL C	SETB C	CPL C
OVER:	OVER:	OVER:
carry=0	carry=0	carry=1

6. Write an ALP to add two input data of 16-bit result in 4 distinct addressing modes.

Solution:

Immediate	Direct	Register	Indirect Addressing
ORG 00H	ORG 00H	ORG 00H	ORG 00H
MOV A,#0FFH	MOV A,40H	MOV R0,#0FFH	MOV R0,#40H
ADD A,#0FFH	ADD A,41H	MOVR1,#0FFH	MOV R1,#51H
END	MOV 51H,A	MOV A,R0	MOV A,@R0
	JNC NOCARRY	ADD A,R1	INC R0
	INCR2	MOV R3,A	ADD A,@R0
	NOCARRY: MOV 50H,R2	JNC NOCARRY	MOV @R1,A
	END	INCR2	JNC NOCARRY
		NOCARRY:END	INC R2
			NOCARRY: MOV A,R2
			DEC R1
			MOV @R1,A
			END

7. Write a program to put the number 34H in registers R4, R5, R6 & R7 using different addressing modes.

Solution:

Immediate	Direct	Register	Indirect Addressing
ORG 00H	ORG 00H	ORG 00H	ORG 00H
MOV R4,#34H	MOV 40H,#34H	MOV A,#34H	MOV R0,#40H
MOV R5,#34H	MOV R4, 40H	MOV R4,A	MOV 04H,@R0
MOV R6,#34H	MOV R5, 40H	MOV R5,A	MOV 05H,@R0
MOV R7,#34H	MOV R6, 40H	MOV R6,A	MOV 06H,@R0
END	MOV R7, 40H	MOV R7,A	MOV 07H,@R0
	END	END	END

8. **Write an ALP in 8051 to perform the following operation: Z = (X1 + Y1) * (X2 + Y2) where, X1, X2, Y1 and Y2 are the 8-bit hexadecimal numbers stored in the RAM locations. Write a subroutine for the addition and assume that each addition result with 8 bit number.**

Solution:

Let X 1,Y 1,X2 & Y2 are stored at memory locations 30H, 31H, 32H and 33H, respectively and results will be stored at memory locations 34H and 35H.

```
ORG 00H
MOV R0, #30H
ACALL ADDITION
MOVR1, A
INC R0
ACALL ADDITION
MOV B, R1
MUL AB
MOV 34H, A
MOV 35H, B
SJMP EXIT
ADDITION:
MOV A,@R0
INC R0
ADD A,@R0
RET
EXIT:
END
```

9. Write an ALP to clear both R0 & R1 of Bank 1 & Bank 2 without using their direct address.

Solution:

```
ORG 00H
SETB PSW.3
CLR PSW.4
MOV R0,#00H
MOVR1,#00H
CLR PSW.3
SETB PSW.4
MOV R0,#00H
MOVR1,#00H
END
```

10. Write an ALP to clear R0 of bank 1 & bank 3 without using its address.

Solution:

```
ORG 00H
SETB PSW.3
CLR PSW.4
MOV R0,#00H
SETB PSW.3
SETB PSW.4
MOV R0,#00H
END
```

11. Write a program to set the carry flag to 1, if the number in reg. A is even and reset the carry flag to 0, if the number in reg. A is ODD. Use the assembly language of 8051.

Method-1	**Method-2**	**Method-3**
ORG 00H	ORG 00H	ORG 00H
MOV A,30H	MOV A,30H	MOV A,30H
RRC A	RRC A	ADD A,#00H
JNC NEXT	JC NEXT	JB PSW.0, NEXT
SETB C	CLR C	CLR C
SJMP EXIT	SJMP EXIT	SJMP EXIT
NEXT: CLR C	NEXT: SETB C	NEXT: SETB C
EXIT: END	EXIT: END	EXIT: END

12. Write the result statement after execution of each instruction.

Solution:

The internal RAM address of SP is 81H i.e., Stack pointer is loaded with address 30H (SP=30H)

Program	Register	Result
MOV 81H,#30H0ACH	30H	30H
MOV R0,#0ACH	R0	0ACH
PUSH 00H	SP& 31H	31H & 0ACH
PUSH 00H	SP & 32H	32H & 0ACH
POP 01H	R1	0ACH
POP 80H	P0	0ACH
MOV A,#0FFH	A	FFH
XRL A,80H	A	53H
POP 82H	P2(82H)	0ACH
POP 83H	P3(83H)	0ACH
MOVX @DPTR,A	ADDRESS	53H

13. What are the final numbers in A, B and OV flag after the execution?

Solution:

MOV A,#7BH	
MOV 0F0H,#02H	0F0H IS A. ADDRESS OF B REGISTER. ie., MOV B,#02H
MUL AB	RESULT =00F6H. ie., B=00H & A=F6H
MOV B,#0FEH	B=FEH
MUL AB	A=F6H & B=FEH. RESULT IS F414. ie., A=14H & B=F4H. OV=1
END	

Result: A=14H, B=F4H & OV=1.

14. Name the addressing modes of the following instructions.

i) MOVC A,@A+DPTR ii) MUL AB iii) MOV B,#0FFH iv) SUBB A,45H

Solution:

Instructions	Addressing Mode
MOVC A,@A+DPTR	INDEXED
MUL AB	REGISTER
MOV B,#0FFH	IMMEDIATE
SUBB A,45H	DIRECT

15. Write a C-program to toggle all bits of P0 & P2 continuously with 250 msec delay. Use inverting operator.

Solution:

```
#include<reg51.h>
void delay(unsigned int);
void main()
{
 P0=0x55;
 P2=0x55;
 while(1)
         {
         P0= P0;
         P2= P2;
         delay(250);
         }
}
void delay(unsigned int count)
{
unsigned int i,j;
for(i=0;ijcount;i++)
for(j = 0;ji1275;j ++);
}
```

16. A switch (sw) is connected to P2.0 port pin. Write a C-program to send out the values 44H serially one bit at a time via P1.0,depending upon the switch condition: when SW=0; LSB should go out first, when SW=1; MSB should go out first.

Solution:

```
#include<reg51.h>
sbit Pin0=P1 ^0;
sbitALSB = ACC^0;
sbitAMSB = ACC^7;
sbitSW = P 2^0;
voidmain( )
```

```
{
  unsigned char mydata = 0x44;
  unsigned char i;
  ACC = mydata;
  if (SW ==0)
          {
          for (i = 0; i < 8; i + +)
                  {
                   pin = ALSB;
                  ACC = ACC>> 1;
                  }
          }
else
          {
          for (i = 0; i < 8; i + +)
                  {
                  pin = AMSB;
                  ACC = ACC << 1;
                  }
          }
}
```

18. Write an 8051 C-program to toggle all the bits of P1, P2 & P0 continuously with a 250 msec delay. Use SFR keyword to declare the port address.

Solution:

```
sfr P0=0x80;
sfr P1=0x90;
sfr P2=0xA0;
void delay(unsigned int);
void main()
  {
  while(1)
          {
           P0=0x55;
           P1 = 0x55;
```

```
            P2=0x55;
            delay(250);
            P0=0xAA;
            P1 = 0xAA;
            P2=0xAA;
            delay(250);
           }
 }
void delay(unsigned int count)
 {
 unsigned int i,j;
 for(i=0;iicount;i++)
 for(j = 0;ji1275;j ++);
 }
```

19. Write an 8051 C-program to convert a given hex-data 0FFH into its equivalent decimal data and display the result digits on P0, P1 & P2.

Solution:

```
#include<reg51.h>
void main()
{
  unsigned char x, bindata, dl, dm, dh;
  bindata=0xFF;
  x=bindata/10;
  dl=bindata%10;
  dm=x%10;
  dh=x/10;
  P0=dl;
  P1=dm;
  P2=dh;
}
```

20. Write a C-program in 8051 to convert packed BCD 0x39 to ASCII and display the bytes on P1 & P2.

Solution:

```
#include<reg51.h>
void main()
{
  unsigned char x,y,z;
  unsigned char mydata=0x39;
  x=mydata & 0x0F;
  P1=x | 0x30;
  y = mydata & 0xP0;
  y = y » 4;
  P2 = y | 0x30;
}
```

21. Explain with an example, bit-wise logical operator for 8051-C.

Solution:

A	B	AND	OR	EX-OR	INVERTER (B)
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	-
1	1	1	1	0	-

```
void main()
{
  P0=0x55&0x0F;
  P1 = 0x04| 0x68;
  P2 = 0x54^0x78;
  P3 = ~0x55;
  P1 = 0x9A » 4;
  P2 = 0x06 « 4;
}
```

22. Write an 8051 c-program to toggle the bit of P1 ports continuously with a 250 msec delay.

Solution:

```
#include<reg51.h>
void delay(unsigned int);
```

```
void main()
{
  while(1)
        {
          P1 = 0x55;
          delay(250);
          P1 = 0xAA;
          delay(250);
        }
}
void delay(unsigned int count)
{
  unsigned int i,j;
  for(i=0;ijcount;i++)
  for(j = 0;ji1275;j ++);
}
```