VSM COLLEGE OF ENGINEERING RAMACHANDRA PURAM

MICROPROCESSOR AND MICROCONTRILLERS (R20)

III B.TECH ECE II SEM

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY KAKINADA
KAKINADA – 533 003, Andhra Pradesh, India
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

III Year – II Semester		L	T	P	C
		3	1	0	3
MICROPROCESSOR AND MICROCONTROLLERS					

UNIT–I
Introduction: Basic Microprocessor architecture, Harvard and Von Neumann architectures with examples, Microprocessor Unit versus Microcontroller Unit, CISC and RISC architectures.
8086 Architecture: Main features, pin diagram/description, 8086 microprocessor family, internal architecture, bus interfacing unit, execution unit, interrupts and interrupt response, 8086 system timing, minimum mode and maximum mode configuration.

UNIT–II
8086 Programming: Program development steps, instructions, addressing modes, assembler directives, writing simple programs with an assembler, assembly language program development tools.

UNIT-III
8086 Interfacing: Semiconductor memories interfacing (RAM, ROM), Intel 8255 programmable peripheral interface, Interfacing switches and LEDS, Interfacing seven segment displays, software and hardware interrupt applications, Intel 8251 USART architecture and interfacing, Intel 8237a DMA controller, stepper motor, A/D and D/A converters, Need for 8259 programmable interrupt controllers.

UNIT-IV
Intel 8051 MICROCONTROLLER
Architecture, Hardware concepts, Input/output ports and circuits, external memory, counters/timers, serial data input/output, interrupts. Assembly language programming: Instructions, addressing modes, simple programs. Interfacing to 8051: A/D and D/A Convertors, Stepper motor interface, keyboard, LCD Interfacing, Traffic light controls.

UNIT–V
ARM Architectures and Processors: ARM Architecture, ARM Processors Families, ARM Cortex-M Series Family, ARM Cortex-M3 Processor Functional Description, functions and interfaces, Programmers Models, ARM Cortext-M3 programming – Software delay, Programming techniques, Loops, Stack and Stack pointer, subroutines and parameter passing, parallel I/O, Nested Vectored Interrupt Controller – functional description and NVIC programmers' model.

TEXTBOOKS:

1. A.K Ray, K.M.Bhurchandhi, Advanced Microprocessor and Peripherals", Tata McGraw Hill Publications, 2000.
2. The 8051 Microcontrollers and Embedded systems Using Assembly and C, Muhammad Ali Mazidi and Janice Gillespie Mazidi and Rollin D. McKinlay; Pearson 2-Edition, 2011.
3. The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors by JosephYou.

REFERENCE BOOKS:

1. Embedded Systems Fundamentals with Arm Cortex-M based Microcontrollers: A Practical Approach in English, by Dr. Alexander G. Dean, Published by Arm Education Media, 2017.
2. Microprocessors and Interfacing – Programming and Hardware by Douglas V Hall, SSSP Rao, Tata McGraw Hill Education Private Limited, 3rd Edition, 1994.
3. Cortex -M3 Technical Reference Manual.

Course Outcomes:

At the end of this course the student will be able to:

1. Understand the architecture of microprocessor/ microcontroller and their operation.
2. Demonstrate programming skills in assembly language for processors and Controllers.
3. Analyze various interfacing techniques and apply them for the design of processor / Controller based systems.

UNIT – I: 8086/8088 MICROPROCESSORS

Introduction:

Microprocessor:

It is a semiconductor device consisting of electronic logic circuits manufactured by using either a large scale (LSI) or Very Large-Scale Integration (VLSI) Technique. It includes the ALU, register arrays and control circuits on a single chip.

The microprocessor has a set of instructions, designed internally, to manipulate data and communicate with peripherals. This process of data manipulation and communication is determined by the logic design of the microprocessor called the architecture.

The era microprocessors in the year 1971, the Intel introduced the first 4-bit microprocessor is 4004. Using this first portable calculator is designed. The following table shows the list of Intel microprocessors.

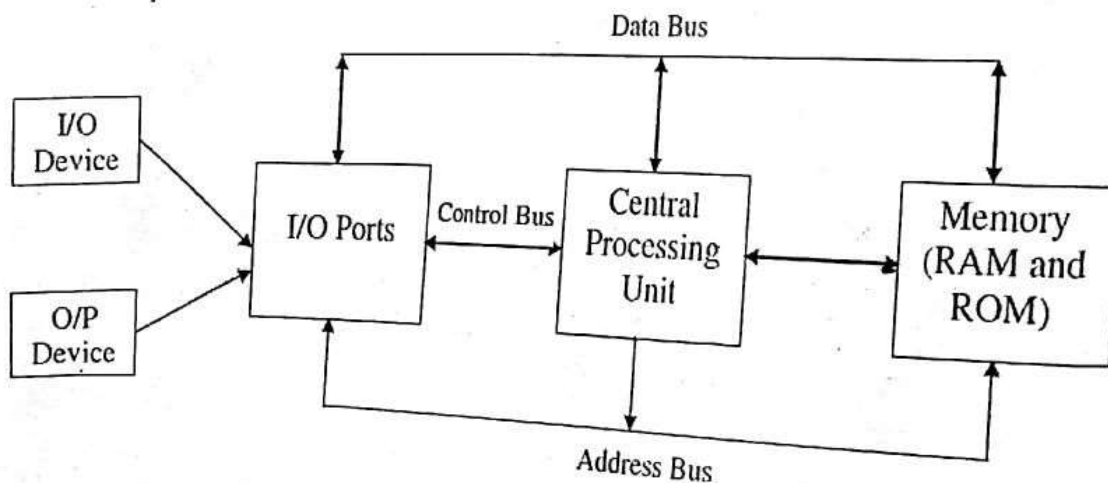
Year	Name	Bit Size
1971	4004	4
1972	8008	8
1974	8080	8
1977	8085	8
1978	8086	16
1979	8089	16
1982	80286	32
1985	80386	32
1989	80486	32
1993	80586(Pentium)	32
1995	Pentium Pro	32
1997	Pentium II	32
1999	Ecleron and Pentium III	32
2000	Pentium IV	32
2001	Intanium	64
2003	Pentium M processor	64
2005	Pentium IV and Xeon	64
2006	Pentium D 900	64

The different manufacturing companies are introduced different bit size microprocessors in the past decade is shown in the table

Company Name	Processor Name
AMD	Athlon
Cypress	CY7C601
DEC	ALPHA
Fujitsu	MBL8086
Harris	CS80C286
LSI Logic	LR 30000
National Semiconductor	NS321016N
SGS-Thomson	ST6X86
SUN-Micro	SRP1030
Texas Instruments	TMS390
Toshiba	TC85R4000
Zilog	Z80
Motorola	68000

A microcomputer system just as any other computer system, include two principal components Hardware and Software.

The memory is used to store both data and instructions that are currently being used. It is normally broken into several modules, each module containing several thousand locations. Each location may contain part or all of a datum or instruction and is associated with an identifier called a memory address. The CPU does its work by successfully inputting, or fetching instructions from memory and carrying out the tasks detected them.



Block Diagram of simple Microcomputer

Above figure shows block diagram of a simple microcomputer. The major parts are the central essing unit or CPU, memory and the input and output circuitry or Input/output. Connecting these parts three sets of parallel line is called buses and control bus. In a microcomputer the CPU is a microprocessor and is often referred to as the microprocessor unit (MPU). Its purpose is to decode the instruction and use them to control the activity within the system. It performs all arithmetic and logical computations.

Memory: Memory section usually consists of a mixture of RAM and ROM. It may also magnetic floppy disks, magnetic hard disks or optical disks, to store the data.

Input/output: The input/output section allows the computer to take in data from the outside world or send data to the outside world. Peripherals such as keyboard, video display terminals. Printers and modem are connected to the input/output section. These allow the user and computer to communicate with each other. The actual physical devices used to interface the computer buses to external systems are often called ports. An input/output port allows data from keyboard, an analog to digital converter (ADC) or some other source to be read into the computer under the control of the CPU. An output port is used to send data from the computer to some peripheral, such as a video display terminal, a printer or a digital to analog converter (DAC).

Central Processing Unit (CPU): CPU controls the operation of the computer. In a microcomputer the CPU is a microprocessor. The CPU fetches the binary coded instructions from memory, decodes the instructions into a series of simple action and carries out these actions in sequence of steps.

CPU contains an address counter or instruction pointer register which holds the address of the next instruction or data item to be fetched from memory, general purpose register, which are used for temporary storage or binary data and circuitry, which generates the control bus signals.

Address Bus: The address bus consists of 16, 20, 24 or 32 parallel lines. On these lines the CPU sends out the address of the memory locations that are to be written to or read from. The number of memory locations that the CPU can addresses is determined by the number of address lines, then it can directly address 2n memory location. When the CPU reads data from or writes data to a port, it sends the port address on the address bus.

Ex: CPU has 16 address lines can address 216 or 65536 memory locations.

Data Bus: It consists of 8, 16, 32 parallel signal lines. The data bus lines are bidirectional. This means that the CPU can read, data from memory or from a port on these lines, or it can send data out to memory or to port on these lines.

Control Bus: The control bus consists of 4 to 10 parallel signals lines. The CPU sends out signals on the control bus enable the outputs of addressed memory devices or port devices. Typical control bus signal are memory read, memory write, I/O read and I/O write.

Hardware, Software and Firmware: hardware is the given to the physical devices and circuitry of computer. Software refers to collection of programs written for the computer. Firmware is the term gi programs stored in ROM's or in other devices which permanently keep their stored information.

Introduction to 16-bit Microprocessor:

The 16-bit Microprocessor families are designed primarily to complete with microcomputers and a oriented towards high-level languages. Their applications sometimes overlap those of the 8-bi microprocessors. The have powerful instruction sets and capable of addressing megabytes of memory. The era of 16-bit Microprocessors began in 1974 with the introduction of PACE chip by National Semiconductor. The Texas Instruments TMS9900 was introduced in the year 1976. The Intel 8086 commercially available in the year 1978, Zilog Z800 in the year 1979, The Motorola MC68000 in the year 1980.

The 16-bit Microprocessors are available in different pin packages.

Ex:	Intel 8086/8088	40 pin packages
	Zilog Z8001	40 pin packages
	Digital equipment LSI-II	40 pin packages
	Motorola MC68000	64 pin packages
	National Semiconductor NS16000	48 pin packages

The primary objectives of this 16-bit Microprocessor can be summarized as follows.

1. Increase memory addressing capability
2. Increase execution speed
3. Provide a powerful instruction set
4. Facilitate programming in high-level languages.

The INTEL 8086/8088 Features:

- Direct Addressing Capability 1 M Byte of Memory
- Architecture Designed for Powerful
- Assembly Language and Efficient High-Level Languages
- 14 Word, by 16-Bit Register Set with Symmetrical Operations
- 24 Operand Addressing Modes
- Bit, Byte, Word, and Block Operations
- 8 and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal including Multiply and Divide

Range of Clock Rates:

5MHz for 8086,

8MHz for 8086-2,

10 MHz for 8086-1

- MULTIBUS System Compatible Interface
- Available in EXPRESS
  - –Standard Temperature Range
  - –Extended Temperature Range
- Available in 40-Lead CERDIP and Plastic package

It is a 16-bit Microprocessor housed in a 40-pin Dual-Inline-Package (DIP) and capable of addressing 1Megabyte of memory, various versions of this chip can operate with different clock frequencies

i. 8086 (5 MHz)

ii. 8086-2 (8 MHz)

iii. 8086-1 (10 MHz).

It contains approximately 29,000 transistors and is fabricated using the HMOS technology. The term 16-bit means that it's arithmetic logic unit, its internal registers and most of its instructions are designed to work with 16-bit binary word. The 8086 Microprocessor has a 16-bit data bus, so it can read from or write data to memory and ports either 16-bits or 8-bits at a time. The 8086 Microprocessor has 20-bit address bus, so it can address any one of 220 or 1,048,576 memory locations. Here 16-bit words will be stored in two consecutive memory locations. If the first byte of a word is at an even address, the 8086 can read entire word in one operation, if the first byte of the word is at an odd address the 8086 will read the first byte with one bus operation and the second byte with another bus operation.

Signal Description of 8086 Microprocessor:

The 8086 Microprocessor is a 16-bit CPU available in 3 clock rates, i.e. 5, 8 and 10MHz, packaged in a 40 pin CERDIP or plastic package. The 8086 Microprocessor operates in single processor or multiprocessor configurations to achieve high performance.

The pin configuration is as shown in figure. Some of the pins serve a particular function in minimum mode (single processor mode) and others function in maximum mode (multiprocessor mode) configuration.

The 8086 signals can be categorized in three groups. The first are the signals having common functions in minimum as well as maximum mode, the second are the signals which have special functions in minimum mode and third are the signals having special functions for maximum mode.

Signal	Pin	8086	Pin	MAX MODE	MIN MODE
Vss (GND)	1		40	Vcc (5P)	
AD14	2		39	AD15	
AD13	3		38	A16/S3	
AD12	4		37	A17/S4	
AD11	5		36	A18/S5	
AD10	6		35	A19/S6	
AD9	7		34	$\overline{BHE}$/S7	
AD8	8		33	MN/$\overline{MX}$	
AD7	9		32	$\overline{RD}$	
AD6	10		31	$\overline{RQ}/\overline{GT0}$	HOLD
AD5	11		30	$\overline{RQ}/\overline{GT1}$	HLDA
AD4	12		29	$\overline{LOCK}$	$\overline{WR}$
AD3	13		28	$\overline{S2}$	M/$\overline{IO}$
AD2	14		27	$\overline{S1}$	DT/$\overline{R}$
AD1	15		26	$\overline{S0}$	$\overline{DEN}$
AD0	16		25	QS0	ALE
NMI	17		24	QS1	$\overline{INTA}$
INTR	18		23	$\overline{TEST}$	
CLK	19		22	READY	
Vss (GND)	20		21	RESET	

The following signals are common for both the minimum and maximum modes.

Power supply and GND

It uses 5V DC supply at Vcc pin 40, and uses ground at Vss pin 1 and 20 for its operation.

Clock signal

Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.

ess/data bus

0-AD15, these are 16 address/data bus. AD0-AD7 carries low order byte data and AD8-AD15 carries gher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit lata.

Address/status bus

A16-A19 /S3-S6. These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.

A17/S4	A16/S3	Function
0	0	Alternate Data
0	1	Stack
1	0	Code or none
1	1	Data

S7/$\overline{BHE}$

BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus D8-D15. This signal is low during the first clock cycle, thereafter it is active.

$\overline{BHE}$	A0	Function
0	0	Whole Word
0	1	Upper byte from or to odd address
1	0	Upper byte from or to even address
1	1	None

Read ($\overline{RD}$)

It is available at pin 32 and is used to read signal for Read operation.

Ready

It is available at pin 32. It is an acknowledgement signal from I/O devices that data is transferred. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.

RESET

It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.

INTR

It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.

NMI

It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.

TEST

This signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.

MN/$\overline{\text{MX}}$

It stands for Minimum/Maximum and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-versa.

Minimum Mode Signals:

$\overline{\text{INTA}}$

It is an interrupt acknowledgement signal and id available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.

ALE

It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.

$\overline{\text{DEN}}$

It stands for Data Enable and is available at pin 26. It is used to enable Transreceiver 8286. The transceiver is a device used to separate data from the address/data bus.

DT/$\overline{R}$

It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the transceiver. When it is high, data is transmitted out and vice-a-versa.

M/$\overline{\text{IO}}$

This signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicating the memory operation. It is available at pin 28.

ands for write signal and is available at pin 29. It is used to write the data into the memory or the tput device depending on the status of $M/\overline{IO}$ signal.

HLDA

It stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal.

HOLD

This signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.

Maximum Mode Signals:

QS_1 and QS_0

These are queue status signals and are available at pin 24 and 25. These signals provide the status of instruction queue. Their conditions are shown in the following table –

QS_0	QS_1	Status
0	0	No operation
0	1	First byte of opcode from the queue
1	0	Empty the queue
1	1	Subsequent byte from the queue

$\bar{S}_0$, $\bar{S}_1$, $\bar{S}_2$

These are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory & I/O control signals. These are available at pin 26, 27, and 28. Following is the table showing their status –

$\bar{S}_2$	$\bar{S}_1$	$\bar{S}_0$	Status
0	0	0	Interrupt acknowledgement
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

LOCK

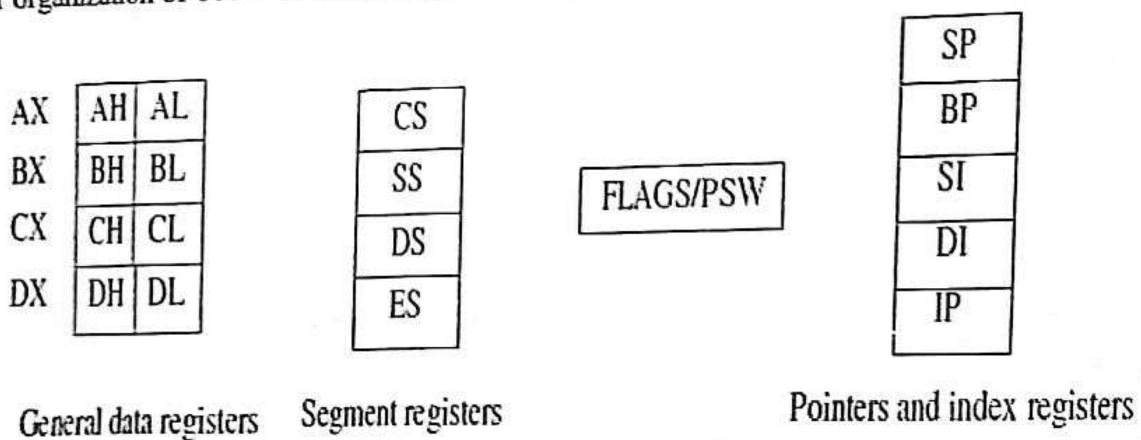
When this signal is active, it indicates to the other processors not to ask the CPU to leave the system bus. It is activated using the LOCK prefix on any instruction and is available at pin 29.

$\overline{RQ}/\overline{GT_1}$ and $\overline{RQ}/\overline{GT_0}$

These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus. When the signal is received by CPU, then it sends acknowledgment. $\overline{RQ}/\overline{GT_0}$ has a higher priority than $\overline{RQ}/\overline{GT_1}$.

Register Organization of 8086:

8086 has a powerful set of registers containing general purpose and special purpose registers. All the registers of 8086 are 16-bit registers. The general-purpose registers, can be used either 8-bit registers or 16-bit registers. The general-purpose registers are either used for holding the data, variables and intermediate results temporarily or for other purpose like counter or for storing offset address for some particular addressing modes etc. The special purpose registers are used as segment registers, pointers, index registers or as offset storage registers for particular addressing modes. The below figure shows register organization of 8086. We will categorize the register set into four groups as follows:



Register Organization of 8086 Microprocessor

General Data Registers:

The registers AX, BX, CX, and DX are the general 16-bit registers.

AX Register: Accumulator register consists of two 8-bit registers AL and AH, which can be combined together and used as a 16- bit register AX. AL in this case contains the low-order byte of the word, and AH contains the high-order byte. Accumulator can be used for I/O operations, rotate and string manipulation.

egister: This register is mainly used as a **base register**. It holds the starting base location of a ory region within a data segment. It is used as offset storage for forming physical address in case of ain addressing mode.

X Register: It is used as default counter or **count register** in case of string and loop instructions.

DX Register: **Data register** can be used as a port number in I/O operations and implicit operand or destination in case of few instructions. In integer 32-bit multiply and divide instruction the DX register contains high-order word of the initial or resulting number.

Segment Registers:

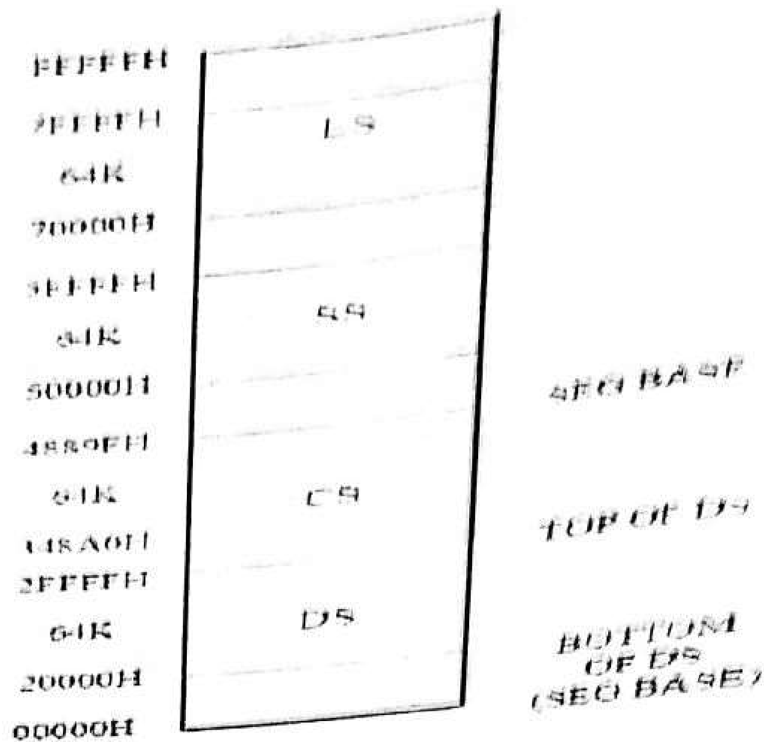
To complete 1Mbyte memory is divided into 16 logical segments. The complete 1Mbyte memory segmentation is as shown in figure. Each segment contains 64Kbyte of memory. There are four segment registers.

Code Segment (CS) is a 16-bit register containing address of 64 KB segment with processor instructions. The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions. It is used for addressing a memory location in the code segment of the memory, where the executable program is stored.

Stack Segment (SS) is a 16-bit register containing address of 64KB segment with program stack. By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment. SS register can be changed directly using POP instruction. It is used for addressing stack segment of memory. The stack segment is that segment of memory, which is used to store stack data.

Data Segment (DS) is a 16-bit register containing address of 64KB segment with program data. By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment. DS register can be changed directly using POP and LDS instructions. It points to the data segment memory where the data is resided.

Extra Segment (ES) is a 16-bit register containing address of 64KB segment, usually with program data. By default, the processor assumes that the DI register references the ES segment in string manipulation instructions. ES register can be changed directly using POP and LES instructions. It also refers to segment which essentially is another data segment of the memory. It also contains data.



Pointers and Index Registers:

The pointers contain within the particular segments. The pointers IP, BP, SP usually contain offsets within the code, data and stack segments respectively.

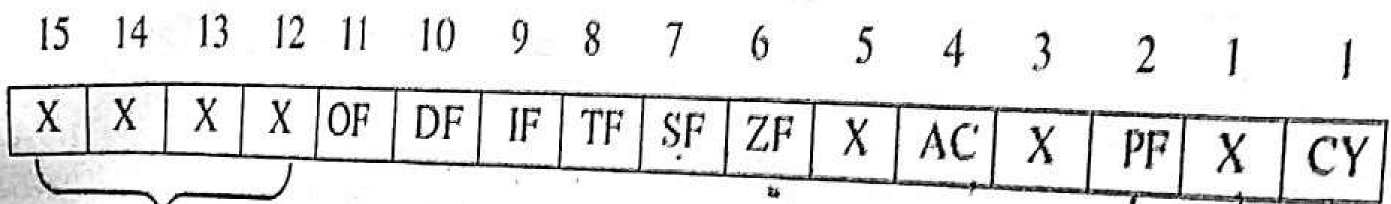
Stack Pointer (SP) is a 16-bit register pointing to program stack in stack segment.

Base Pointer (BP) is a 16-bit register pointing to data in stack segment. BP register is usually used for based, based indexed or register indirect addressing.

Source Index (SI) is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data addresses in string manipulation instructions.

Destination Index (DI) is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

Flag Register:



Flag Register of 8086

Register determines the current state of the processor. They are modified automatically by CPU mathematical operations, this allows to determine the type of the result, and to determine conditions transfer control to other parts of the program.

The 8086-flag register as shown in the fig 1.6. 8086 has 9 active flags and they are divided into two categories:

1. Conditional Flags/ Status Flags
2. Control Flags

Conditional Flags

Conditional flags are as follows:

Carry Flag (CY): This flag indicates an overflow condition for unsigned integer arithmetic. It is also used in multiple-precision arithmetic.

Auxiliary Flag (AC): If an operation performed in ALU generates a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), the AC flag is set i.e. carry given by D3 bit to D4 is AC flag. This is not a general-purpose flag, it is used internally by the Processor to perform Binary to BCD conversion.

Parity Flag (PF): This flag is used to indicate the parity of result. If lower order 8-bits of the result contains even number of 1's, the Parity Flag is set and for odd number of 1's, the Parity flag is reset.

Zero Flag (ZF): It is set; if the result of arithmetic or logical operation is zero else it is reset.

Sign Flag (SF): In sign magnitude format the sign of number is indicated by MSB bit. If the result of operation is negative, sign flag is set.

Control Flags

Control flags are set or reset deliberately to control the operations of the execution unit. Control flags are as follows:

Trap Flag (TF): It is used for single step control. It allows user to execute one instruction of a program at a time for debugging. When trap flag is set, program can be run in single step mode.

Interrupt Flag (IF): It is an interrupt enable/disable flag. If it is set, the maskable interrupt of 8086 is enabled and if it is reset, the interrupt is disabled. It can be set by executing instruction SIT and can be cleared by executing CLI instruction.

Direction Flag (DF): It is used in string operation. If it is set, string bytes are accessed from higher memory address to lower memory address. When it is reset, the string bytes are accessed from lower memory address to higher memory address.

Internal Architecture of 8086 Microprocessor:

The internal architecture 8086 microprocessor is as shown in the figure. The 8086 CPU is divided into two independent functional parts, the Bus interface unit (BIU) and execution unit (EU).

The Bus Interface Unit contains Bus Interface Logic, Segment registers, Memory addressing logic and a Six-byte instruction object code queue. The execution unit contains the Data and Address registers, the Arithmetic and Logic Unit, the Control Unit and flags.

The BIU sends out address, fetches the instructions from memory, read data from ports and memory, and writes the data to ports and memory. In other words, the BIU handles all transfers of data and addresses on the buses for the execution unit.

The execution unit (EU) of the 8086 tells the BIU where to fetch instructions or data from, decodes instructions and executes instruction. The EU contains control circuitry which directs internal operations. A decoder in the EU translates instructions fetched from memory into a series of actions which the EU carries out. The EU is having a 16-bit ALU which can add, subtract, AND, OR, XOR, increment, decrement, complement or shift binary numbers. The EU is decoding an instruction or executing an instruction which does not require use of the buses.

The Queue: The BIU fetches up to 6 instruction bytes for the following instructions. The BIU stores these prefetched bytes in first-in-first-out register set called a queue. When the EU is ready for its next instruction it simply reads the instruction byte(s) for the instruction from the queue in the BIU. This is much faster than sending out an address to the system memory and waiting for memory to send back the next instruction byte or bytes. Except in the case of JMP and CALL instructions, where the queue must be dumped and then reloaded starting from a new address, this prefetch-and-queue scheme greatly speeds up processing. Fetching the next instruction while the current instruction executes is called pipelining.

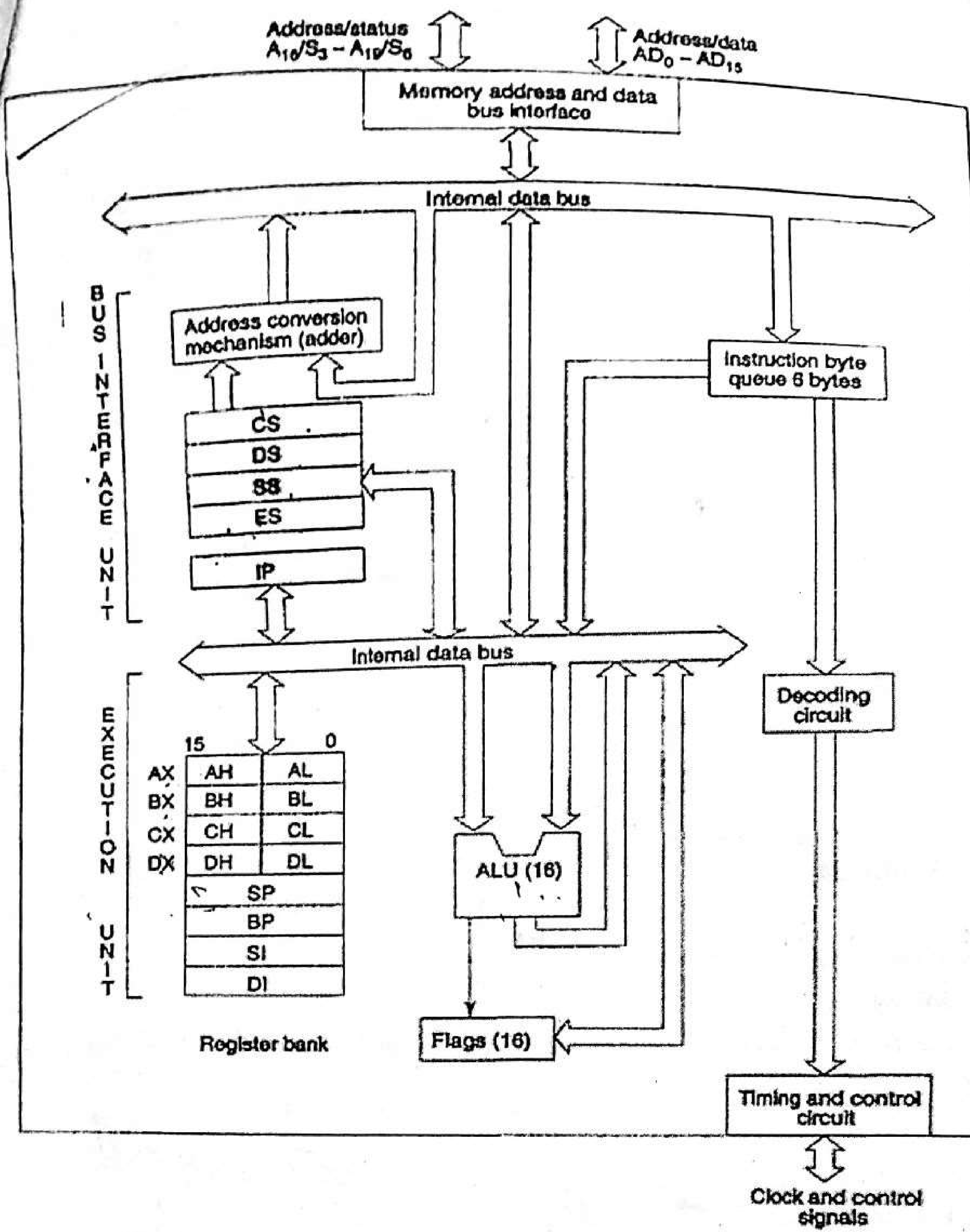


Fig 1: Internal Architecture of 8086 Microprocessor

Word Read

Each of 1 MB memory address of 8086 represents a byte wide location.16-bit words will be stored in two consecutive memory locations. If first byte of the data is stored at an even address, 8086 can read the entire word in one operation.

For example, if the 16-bit data is stored at even address 00520H is 9634H

MOV BX, [00520H]

8086 reads the first byte and stores the data in BL and reads the 2nd byte and stores the data in BH

BL= (00520H) i.e. BL=34H

BH= (00521H) BH=96H

If the first byte of the data is stored at an odd address, 8086 needs two operations to read the 16-bit data.

For example, if the 16-bit data is stored at even address 00521H is 3897H

MOV BX, [00521H]

In first operation, 8086 reads the 16-bit data from the 00520H location and stores the data of 00521H location in register BL and discards the data of 00520H location In 2nd operation, 8086 reads the 16 bit data from the 00522H location and stores the data of 00522H location in register BH and discards the data of 00523H location.

BL= (00521H) i.e. BL=97H

BH= (00522H) BH=38H

Byte Read:

MOV BH, [Addr]

For Even Address:

Ex: **MOV BH, [00520H]**

8086 reads the first byte from 00520 locations and stores the data in BH and reads the 2nd byte from the 00521H location and ignores it

BH = [00520H]

For Odd Address

Ex: **MOV BH, [00521H]**

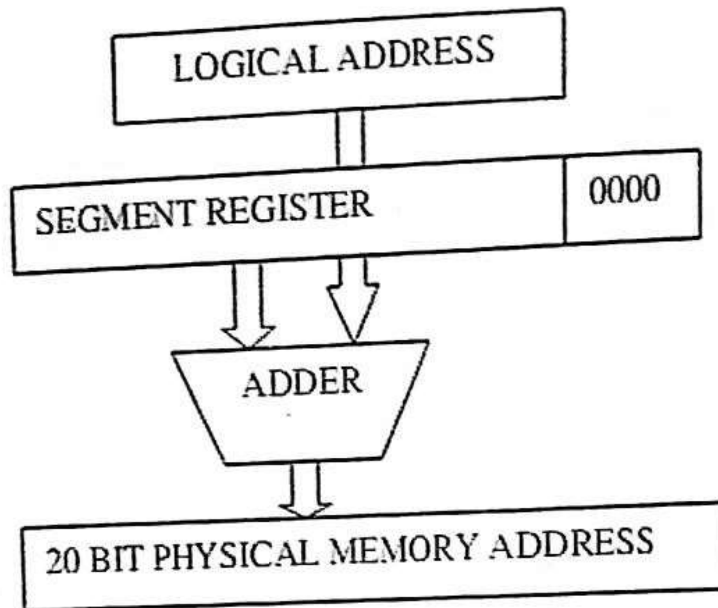
8086 reads the first byte from 00520H location and ignores it and reads the 2nd byte from the 00521 locations and stores the data in BH

BH = [00521H]

cal Address formation:

The 8086 addresses a segmented memory. The complete physical address which is 20-bits long is erated using segment and offset registers each of the size 16-bit. The content of a segment register also lled as segment address, and content of an offset register also called as offset address. To get total physical address, put the lower nibble 0H to segment address and add offset address. The figure shows formation of 20-bit physical address.

GENERATION OF 20-BIT PHYSICAL ADDRESS



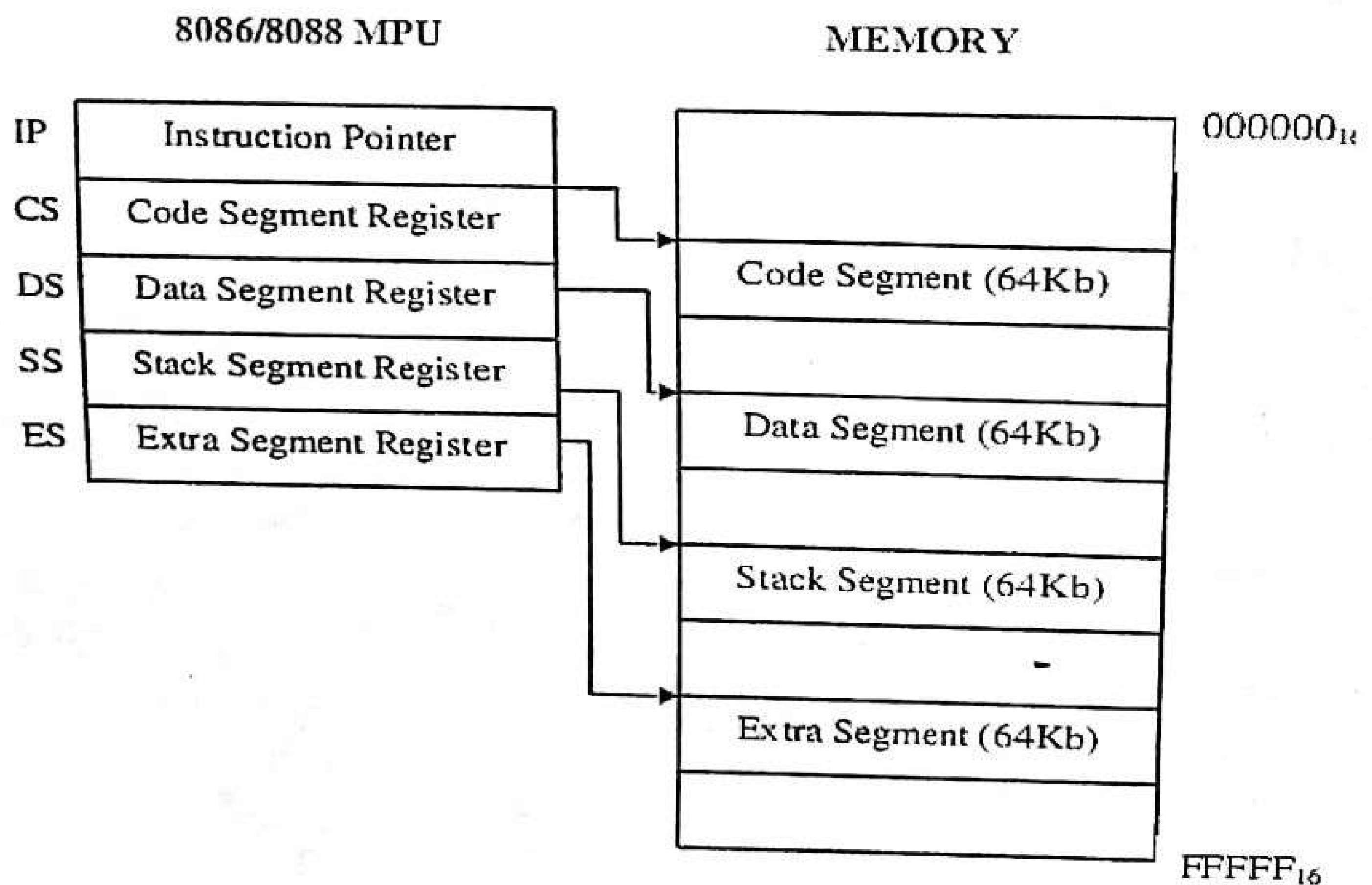
Physical Address Calculation (i):

	CS		IP
Logical Address =	2 5 0 0	:	9 5 F 3

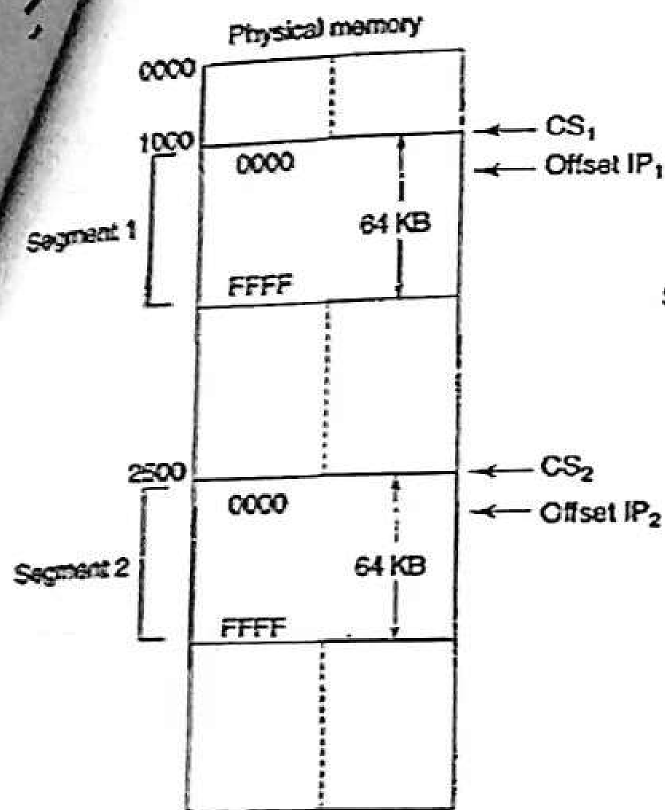
Step	Value
Start with CS.	2 5 0 0
Shift left CS.	2 5 0 0 0
Add IP. (+)	9 5 F 3
Physical address =	2 E 5 F 3

Physical Address Calculation (ii):

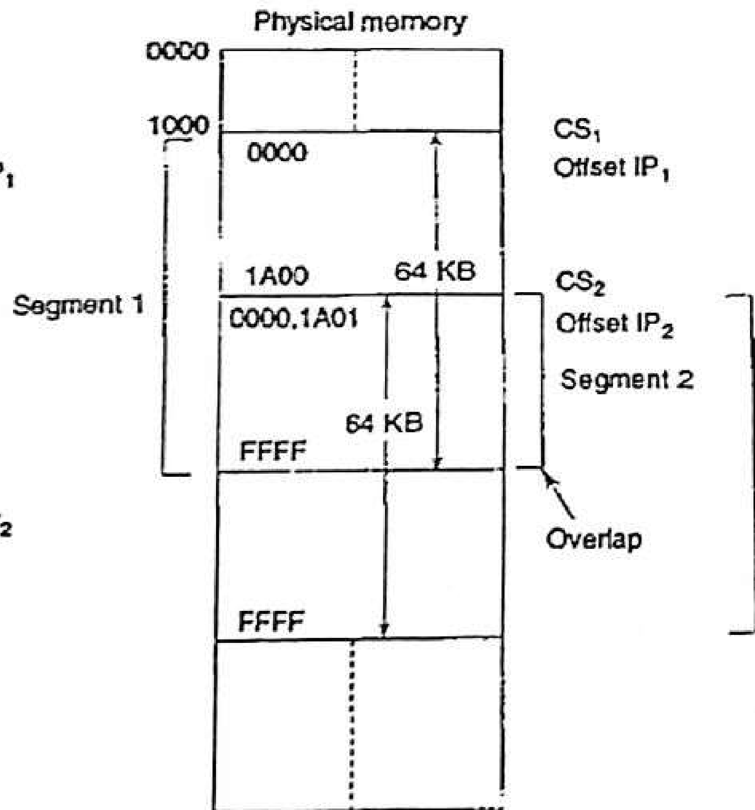
- If DS = 7FA2H and the offset is 438EH,
  - The logical address:
    7FA2:438E
  - The physical address:
    7FA20 + 438E = 83DAE
  - The upper range of the data segment:
    7FA20 + FFFF = 8FA1F
  - The lower range of the data segment:
    7FA20 + 0000 = 7FA20



ory Segmentation:



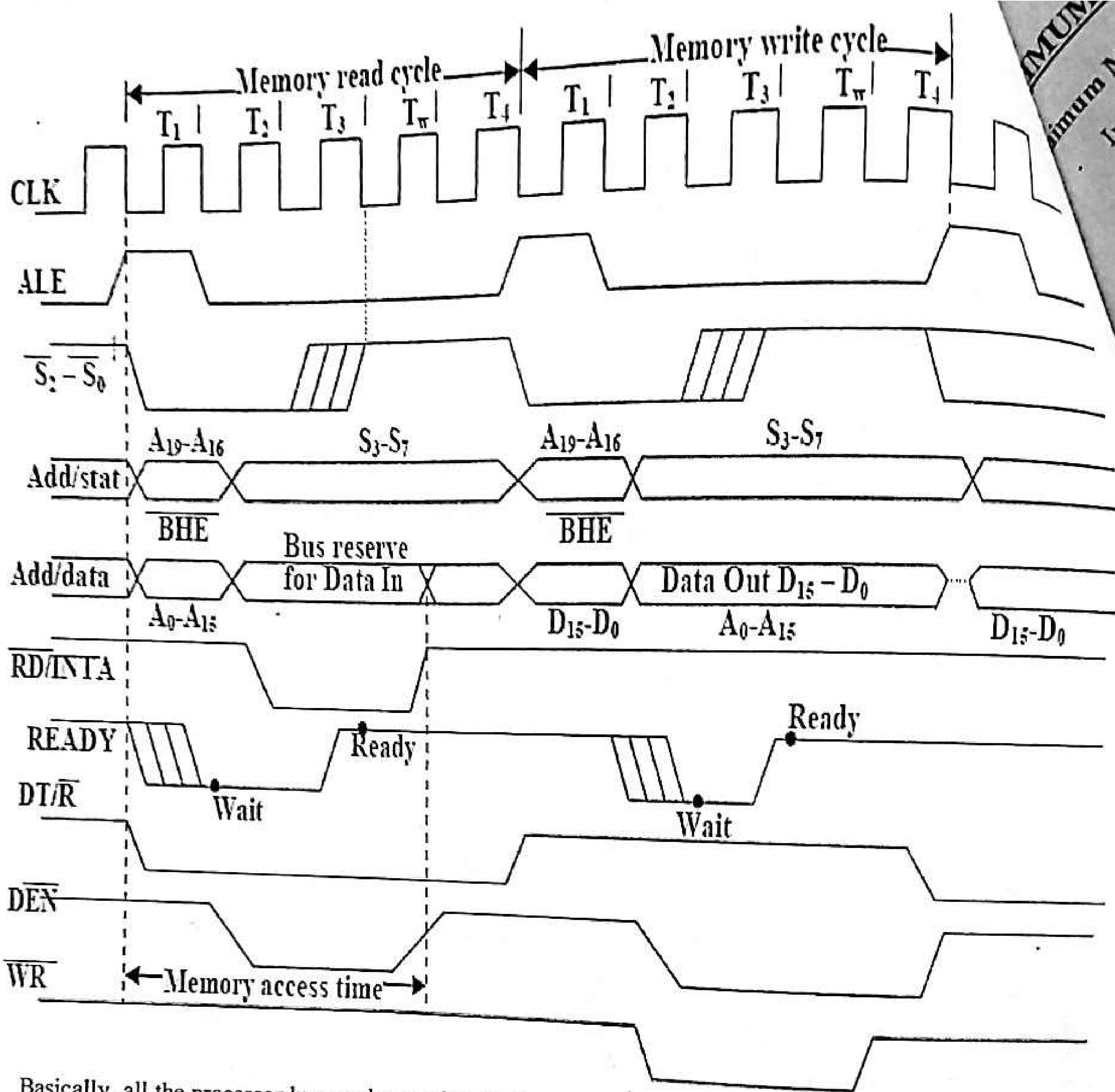
(a) *Non-overlapping Segments*



(b) *Overlapping Segments*

General Bus Operation:

- The 8086 has a combined address and data bus commonly referred as a time multiplexed address and data bus.
- The main reason behind multiplexing address and data over the same pins is the maximum utilization of processor pins and it facilitates the use of 40 pin standard DIP package.
- The bus can be de-multiplexed using a few latches and transceivers, whenever required.



- Basically, all the processor bus cycles consist of at least four clock cycles. These are referred to as T1, T2, T3, T4. The address is transmitted by the processor during T1. It is present on the bus only for one cycle. The negative edge of this ALE pulse is used to separate the address and the data or status information. In maximum mode, the status lines $\bar{S}_0$, $\bar{S}_1$, $\bar{S}_2$ are used to indicate the type of operation.
- Status bits S3 to S7 are multiplexed with higher order address bits and the BHE signal. Address is valid during T1 while status bits S3 to S7 are valid during T2 through T4.

MIUM MODE 8086 SYSTEM AND TIMINGS:

mum Mode

- In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/$\overline{MX}$ pin to logic 1.
- In this mode, all the control signals are given out by the microprocessor chip itself.

In a minimum mode 8086 system, the microprocessor 8086 is operated in minimum mode by strapping its MN/$\overline{MX}$ pin to logic1. In this mode, all the control signals are given out by the microprocessor chip itself. There is a single microprocessor in the minimum mode system. The remaining components in the system are latches, transceivers, clock generator, memory and I/O devices. Some type of chip selection logic may be required for selecting memory or I/O devices, depending upon the address map of the system.

The latches are generally buffered output D-type flip-flops, like, 74LS373 or 8282. They are used for separating the valid address from the multiplexed address/data signals and are controlled by the ALE signal generated by 8086. Transceivers are the bidirectional buffers and sometimes they are called as data amplifiers. They are required to separate the valid data from the time multiplexed address/data signal. They are controlled by two signals, namely, $\overline{DEN}$ and DT/$\overline{R}$. The $\overline{DEN}$ signal indicates that the valid data is available on the data bus, while DT/$\overline{R}$ indicates the direction of data, i.e. from or to the processor. The system contains memory for the monitor and users program storage. Usually, EPROMs are used for monitor storage, while RAMs for user's program storage. A system may contain I/O devices for communication with the processor as well as some special purpose I/O devices. The clock generator generates the clock from the crystal oscillator and then shapes it and divides to make it more precise so that it can be used as an accurate timing reference for the system. The clock generator also synchronizes some external signals with the system clock. The general system organization is shown in Fig. 1.1. Since it has 20 address lines and 16 data lines, the 8086 CPU requires three octal address latches and two octal data buffers for the complete address and data separation.

The working of the minimum mode configuration system can be better described in terms of the timing diagrams rather than qualitatively describing the operations. The opcode fetch and read cycles are similar. Hence the timing diagram can be categorized in two parts, the first is the timing diagram for read cycle and the second is the timing diagram for write cycle. Fig 1.1(a) shows the read cycle timing diagram.

- The read cycle begins in T1 with the assertion of the address latch enable (ALE) signal and also M/$\overline{IO}$ signal. During the negative going edge of this signal, the valid address is latched on the local bus. The $\overline{BHE}$ and A0 signals address low, high or both bytes.

- From T1 to T4, the M/$\overline{IO}$ signal indicates a memory or I/O operation.
- At T2 the address is removed from the local bus and is sent to the output. The bus is then tri-stated. The read ($\overline{RD}$) control signal is also activated in T2. The read ($\overline{RD}$) signal causes the addressed device to enable its data bus drivers.
- After $\overline{RD}$ goes low, the valid data is available on the data bus. The addressed device will drive the READY line high, when the processor returns the read signal to high level, the addressed device will again tri-state its bus drivers.

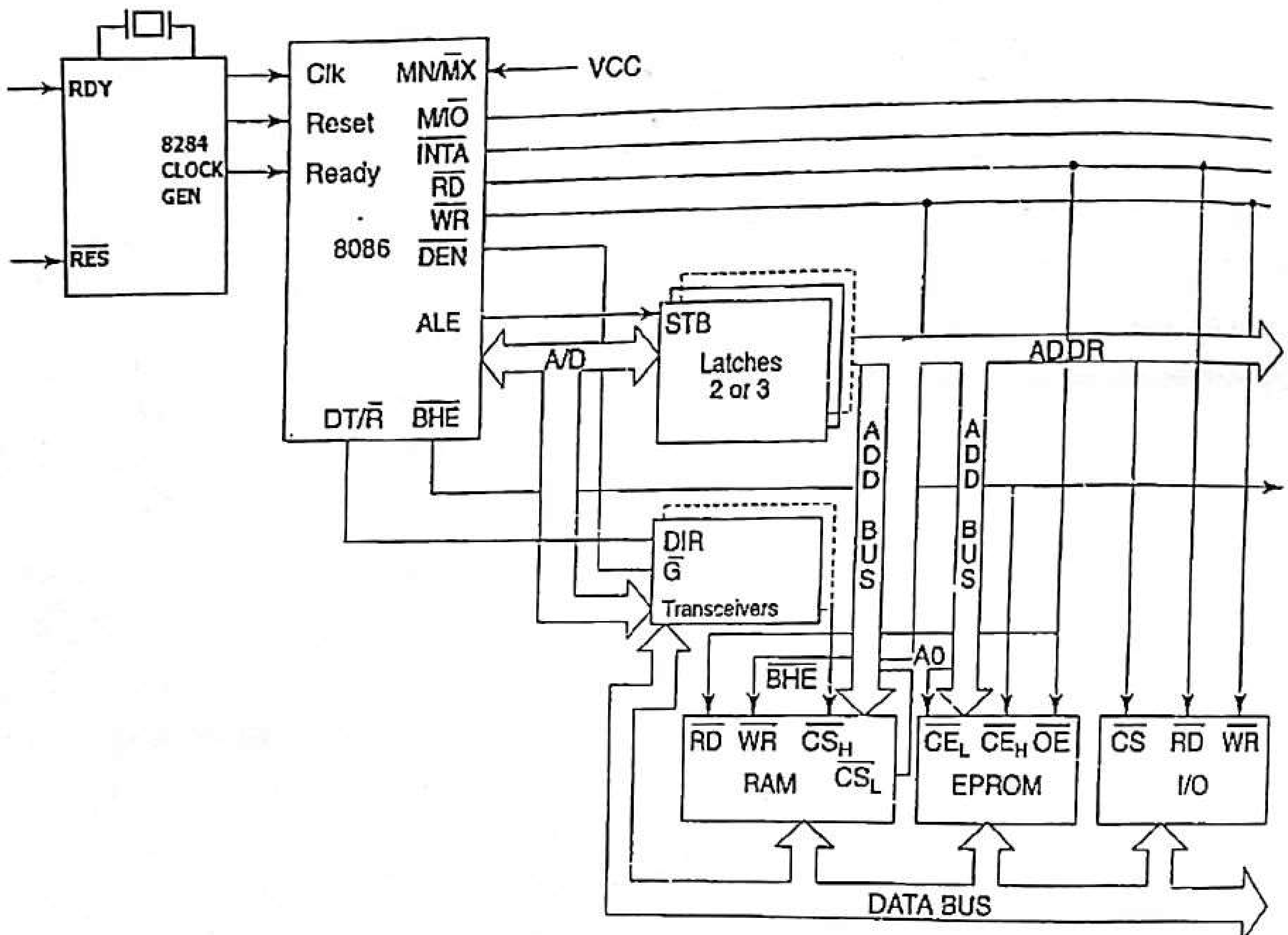


Fig 1.1: Minimum Mode 8086 System

Fig 1.1 (b) shows the write cycle timing diagram. A write cycle also begins with the assertion of ALE and the emission of the address. The M/$\overline{IO}$ signal is again asserted to indicate a memory or I/O operation.

- In T2 after sending the address in T1 the processor sends the data to be written to the addressed location.

The data remains on the bus until middle of T4 state. The $\overline{WR}$ becomes active at the beginning of T2 (unlike $\overline{RD}$ is somewhat delayed in T2 to provide time for floating).

. The $\overline{BHE}$ and A0 signals are used to select the proper byte or bytes of memory or I/O word to be read or written. The M/$\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ signals indicate the types of data transfer as specified in Table

M/$\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Transfer Type
0	0	1	I/O read
0	1	0	I/O write
1	0	1	Memory read
1	1	0	Memory write

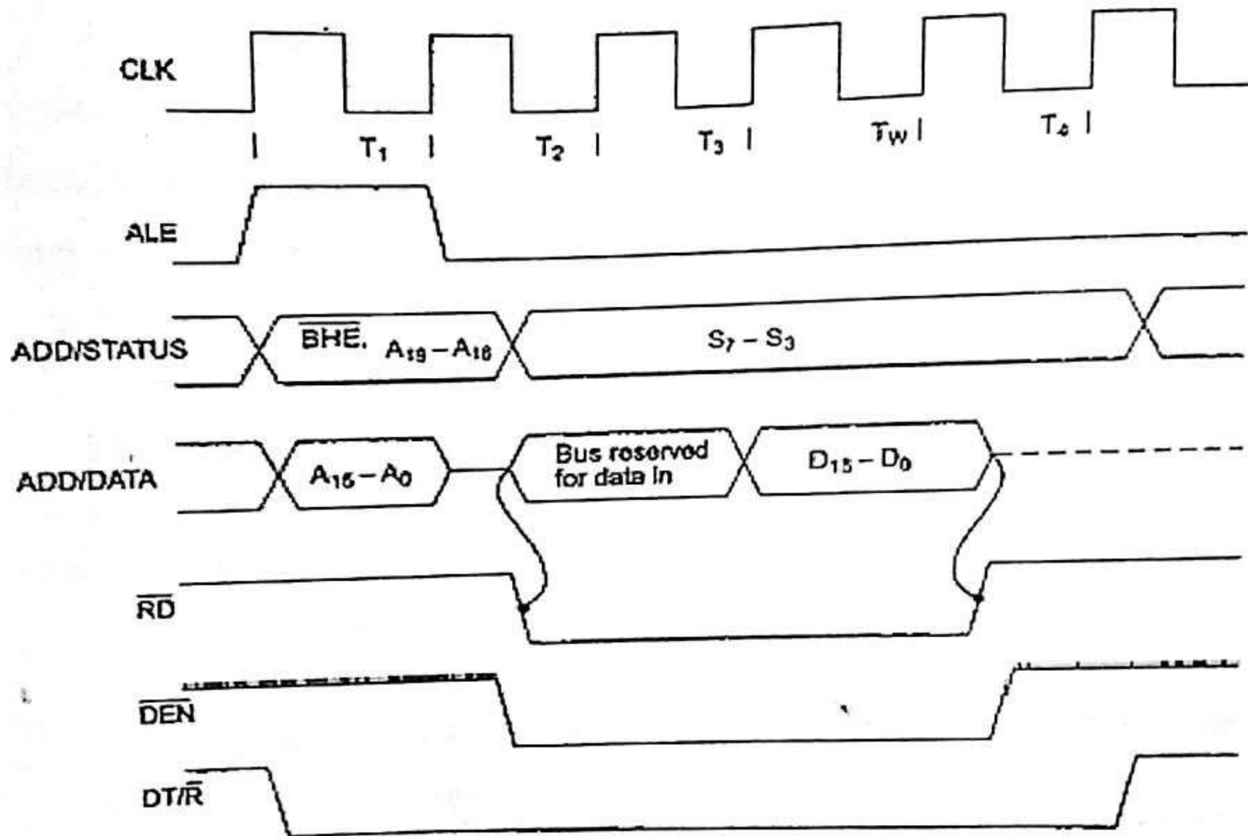


Fig.1.1 (a): Read Cycle Timing Diagram for Minimum Mode

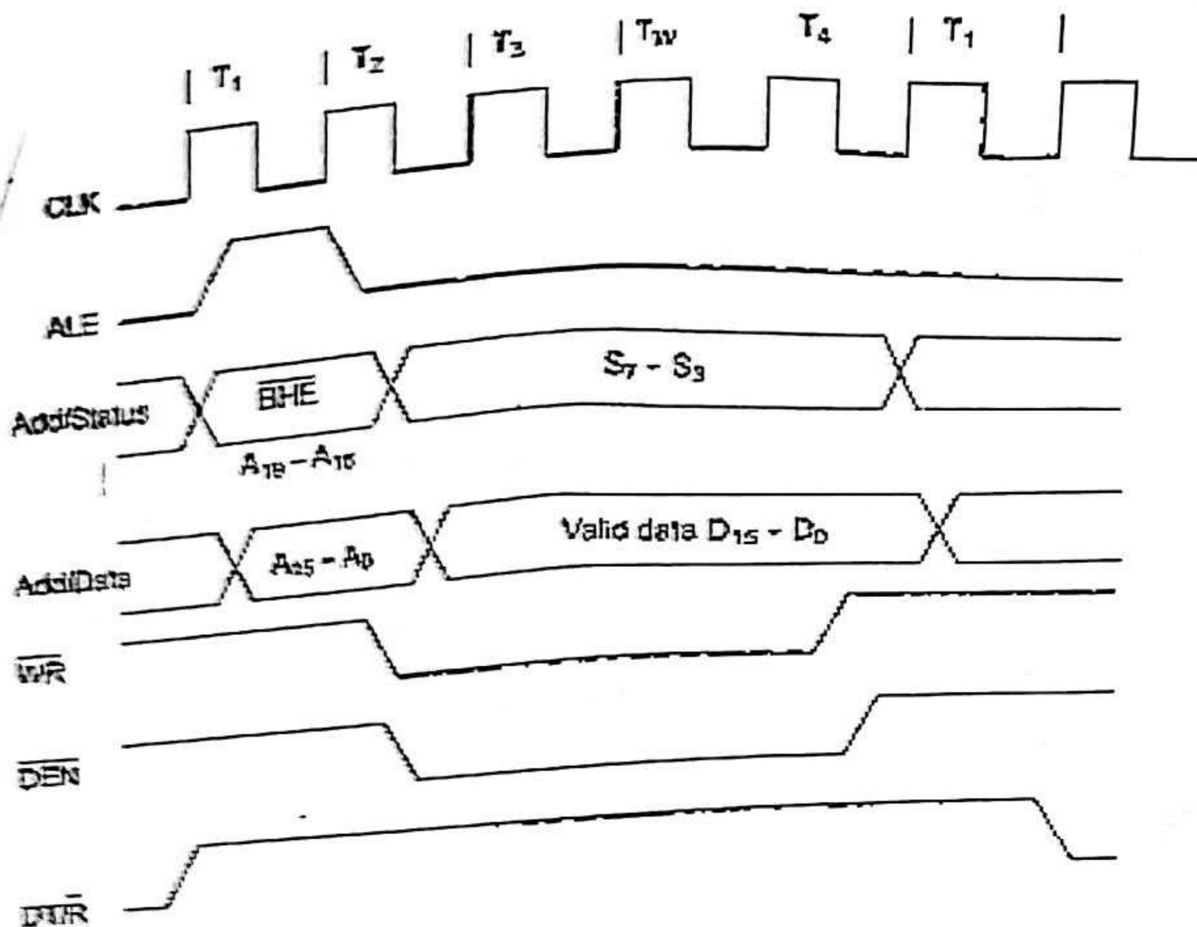


Fig.1.1 (b): Write Cycle Timing Diagram for Minimum Mode

HOLD Response Sequence

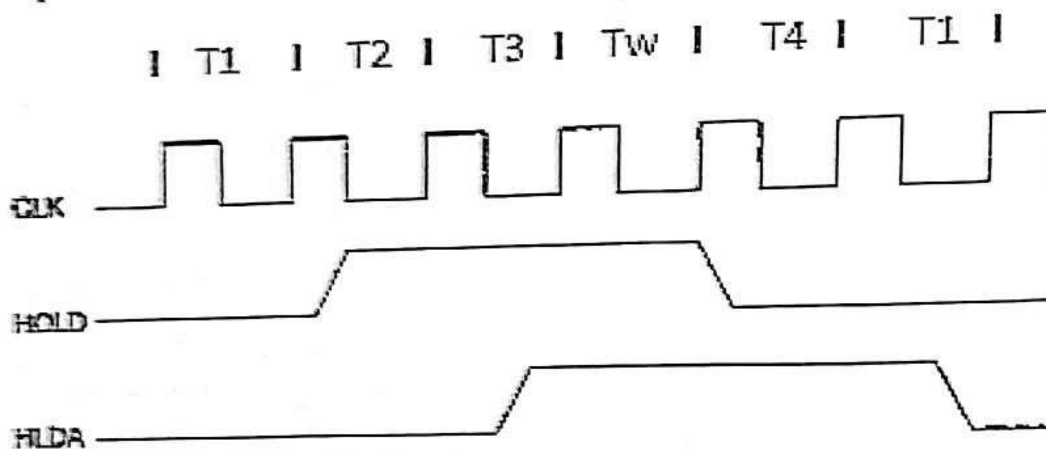


Fig 1.1(c): Bus Request and Bus Grant Timings in Minimum Mode System

The HOLD pin is checked at the end of each bus cycle. If it is received active by the processor before T4 of the previous cycle or during T1 state of the current cycle, the CPU activities HLDA in the next clock cycle and for the succeeding bus cycles, the bus will be given to another requesting master the control of the bus is not regained by the processor until the requesting master does not drop the HOLD pin low. When the request is dropped by the requesting master, the HLDA is dropped by the processor at the trailing edge of the next clock as shown in fig 1.1 (c).

...UM MODE 8086 SYSTEM AND TIMINGS:

...um Mode

In the maximum mode, the 8086 is operated by strapping the MN/$\overline{MX}$ pin to ground.

- In this mode, the processor derives the status signal $\bar{S}_2$, $\bar{S}_1$, $\bar{S}_0$. Another chip called bus controller derives the control signal using this status information.
- In the maximum mode, there may be more than one microprocessor in the system configuration.

In the maximum mode, the 8086 is operated by strapping the MN/$\overline{MX}$ pin to ground. In this mode, the processor derives the status signals $\bar{S}_2$, $\bar{S}_1$ and $\bar{S}_0$. Another chip called bus controller derives the control signals using this status information. In the maximum mode, there may be more than one microprocessor in the system configuration. The other components in the system are the same as in the minimum mode system. The general system organization is as shown in the fig1.2.

The basic functions of the bus controller chip IC 8288, is to derive control signals like $\overline{RD}$ and $\overline{WR}$ (for memory and I/O devices), $\overline{DEN}$, DT/$\bar{R}$, ALE, etc. using the information made available by the processor on the status lines. The bus controller chip has input lines $\bar{S}_2$, $\bar{S}_1$ and $\bar{S}_0$ and CLK. These inputs to 8288 are driven by the CPU. It derives the outputs ALE, $\overline{DEN}$, DT/$\bar{R}$, $\overline{MWTC}$, $\overline{AMWTC}$, $\overline{IORC}$, $\overline{IOWC}$ and $\overline{AIOWC}$. The $\overline{AEN}$, IOB and CEN pins are especially useful for multiprocessor systems. $\overline{AEN}$ and IOB are generally grounded. CEN pin is usually tied to +5V.

The significance of the MCE/$\overline{PDEN}$ output depends upon the status of the IOB pin. If IOB is grounded, it acts as master cascade enable to control cascaded 8259A; else it acts as peripheral data enable used in the multiple bus configurations. $\overline{INTA}$ pin is used to issue two interrupt acknowledge pulses to the interrupt controller or to an interrupting device.

$\overline{IORC}$, $\overline{IOWC}$ are I/O read command and I/O write command signals respectively. These signals enable an IO interface to read or write the data from or to the addressed port. The $\overline{MRDC}$, $\overline{MWTC}$ are memory read command and memory write command signals respectively and may be used as memory read and write signals. All these command signals instruct the memory to accept or send data from or to the bus. For both of these write command signals, the advanced signals namely $\overline{AIOWC}$ and $\overline{AMWTC}$ are available. They also serve the same purpose, but are activated one clock cycle earlier than the $\overline{IOWC}$ and $\overline{MWTC}$ signals, respectively. The maximum mode system is shown in fig. 1.2

Fig 1.2: Maximum Mode 8086 System

Fig. 1.2 (a): Memory Read Timing in Maximum Mode

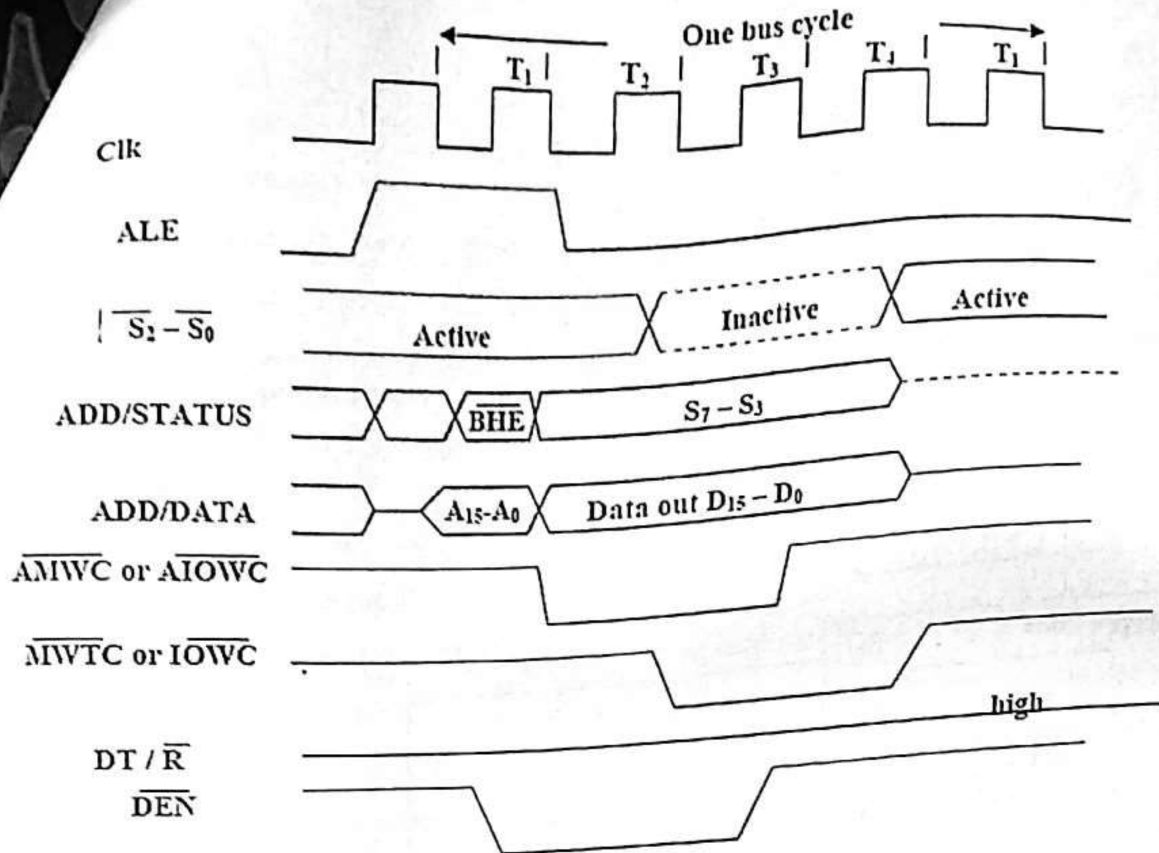


Fig. 1.2 (b): Memory Write Timing in Maximum Mode

The maximum mode system timing diagrams are also divided in two portions as read (input) and write (output) timing diagrams. The address/data and address/status timings are similar to the minimum mode. ALE is asserted in T1, just like minimum mode. The only difference lies in the status signals used and the available control and advanced command signals. The fig. 1.2 (a) shows the maximum mode timings for the read operation while the fig. 1.2 (b) shows the same for the write operation.

Timings for $\overline{RQ}/\overline{GT}$ Signals: -

The request/ grant response sequence contains a series of three pulses as shown in the timing diagram. When a request is detected for valid HOLD request are satisfied, the processor issues a grant pulse over the $\overline{RQ}/\overline{GT}$ pin immediately during the T4 (current) or T1 (next) state. When the requesting master receives this pulse, it accepts the control of the bus. The requesting master uses the bus till it requires. When it is ready to relinquish the bus, it sends a release pulse to the processor (host) using the $\overline{RQ}/\overline{GT}$ pin.

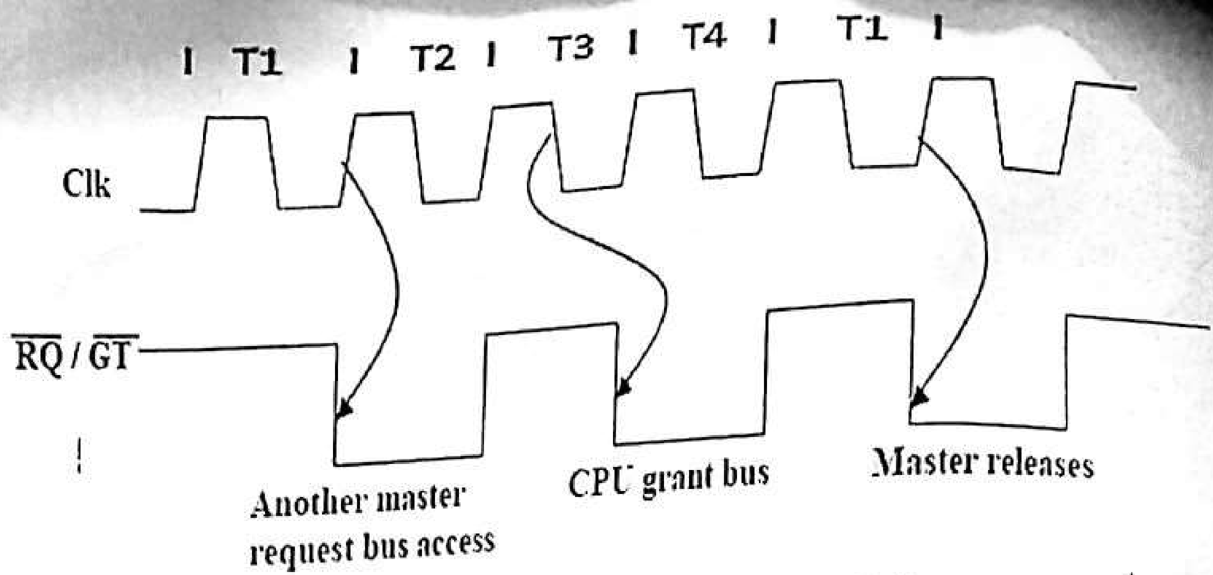
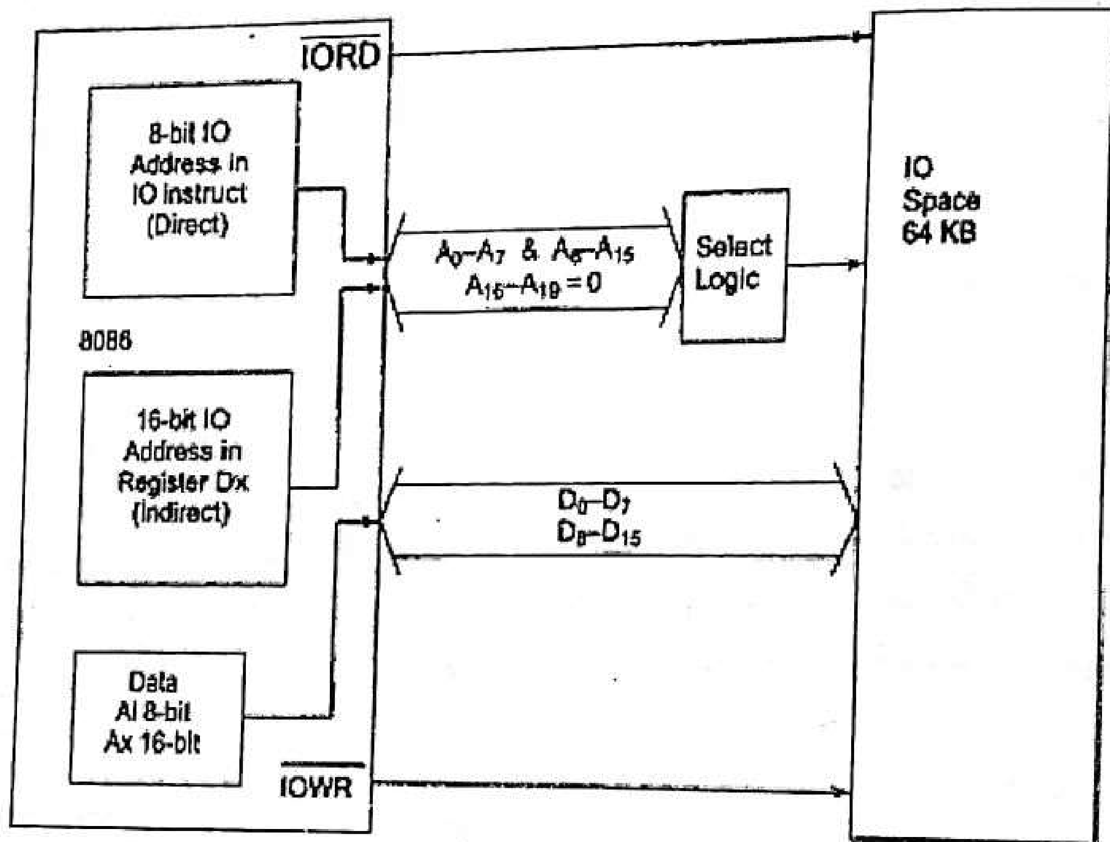


Fig1.2 (c): $\overline{RQ/GT}$ Timings in Maximum Mode

I/O Addressing Capability:



The 8086 can generate 16-bit of I/O address. Thus, it can address up to 64 Kbytes I/O locations or 32K word I/O locations. The 16-bit I/O address appears on A0 to A15 address lines; A16 to A19 lines are at logic 0 during the I/O operations. The 16-bit DX register is used as 16-bit I/O address pointer to address up to 64K devices in in-direct addressing mode. The I/O instructions with direct addressing mode can directly address one or two of the 256 I/O byte locations.

I/O ports are addressed in the same manner as memory locations. Even addressed bytes are transferred on the D7 – D0 bus lines and Odd addressed bytes on D15 – D8.

...RUPTS AND INTERRUPT SERVICE ROUTINES: 8

...rupt and its Need

The microprocessors allow normal program execution to be interrupted in order to carry out a specific task/work. The processor can be interrupted in the following ways

i) by an external signal generated by a peripheral,

ii) by an internal signal generated by a special instruction in the program,

iii) by an internal signal generated due to an exceptional condition which occurs while executing an instruction. (For example, in 8086 processors, divide by zero is an exceptional condition which initiates type 0 interrupt and such an interrupt is also called execution).

In general, the process of interrupting the normal program execution to carry out a specific task/work is referred to as interrupt.

The interrupt is initiated by a signal generated by an external device or by a signal generated internal to the processor. When a microprocessor receives, an interrupt signal it stops executing current normal program, save the status (or content) of various registers (IP, CS and flag registers in case of 8086) in stack and then the processor executes a subroutine/procedure in order to perform the specific task/work requested by the interrupt. The subroutine/procedure that is executed in response to an interrupt is also called Interrupt Service Subroutine. (ISR). At the end of ISR, the stored status of registers in stack is restored to respective registers, and the processor resumes the normal program execution from the point {instruction) where it was interrupted.

The external interrupts are used to implement interrupt driven data transfer scheme. The interrupts generated by special instructions are called software interrupts and they are used to implement system services/calls (or monitor services/calls). The system/monitor services are procedures developed by system designer for various operations and stored in memory. The user can call these services through software interrupts. The interrupts generated by exceptional conditions are used to implement error conditions in the system.

Interrupt Driven Data Transfer Scheme

The interrupts are useful for efficient data transfer between processor and peripheral. When a peripheral is ready for data transfer, it interrupts the processor by sending an appropriate signal. Upon receiving an interrupt signal, the processor suspends the current program execution, save the status in stack and executes an ISR to perform the data transfer between the peripheral and processor. At the end of ISR the processor status is restored from stack and processor resume its normal program execution. This type of data transfer scheme is called interrupt driven data transfer scheme.

The data transfer between the processor and peripheral devices can be implemented either polling technique or by interrupt method. In polling technique, the processor has to periodically poll or check the status/readiness of the device and can perform data transfer only when the device 'is ready. In polling technique, the processor time is wasted, because the processor has to suspend its work and check the status of the device in predefined intervals.

Alternatively, if the device interrupts the processor to initiate a data transfer whenever it is ready then the processor time is effectively utilized because the processor need not suspend its work and check the status of the device in predefined intervals. For an example, consider the data transfer from a keyboard to the processor.

Normally a keyboard has to be checked by the processor once in every 10 milli seconds for a key press. Therefore, once in every 10 milli seconds the processor has to suspend its work and then check the keyboard for a valid key code. Alternatively, the keyboard can interrupt the processor, whenever a key is pressed and a valid key code is generated. In this way, the processor need not waste it's time to check the keyboard once in every 10 milli seconds.

Classification of Interrupts

In general, the interrupts can be classified in the following three ways:

1. Hardware and software interrupts
2. Vectored and Non-Vectored interrupt
3. Maskable and Non-Maskable interrupts

The interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of the processor is called hardware interrupt. The 8086 processor has two interrupt pins INTR and NMI. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086.

The software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if software interrupt instruction is encountered then the processor initiates an interrupt. The 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 0 to 255.

When an interrupt signal is accepted by the processor, if the program control automatically branches to a specific address (called vector address) then the interrupt is called vectored interrupt. The automatic branching to vector address is predefined by the manufacturer of processors. (In these vector addresses the interrupt service subroutines (ISR) are stored). In non-vectored interrupts the interrupting device should supply the address of the ISR to be executed in response to the interrupt. All the 8086 interrupts are vectored interrupts. The vector address for an 8086 interrupt is obtained from a vector table implemented in the first 1kb memory space (00000h to 03FFFh).

The processor has the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to accept an interrupt is referred to as unmasking or enabling. In 8086 the interrupt flag (IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable a hardware interrupts except NMI.

The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts. The interrupts whose request has to be definitely accepted (or cannot be rejected) by the processor are called non-maskable interrupts. Whenever a request is made by non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISR. In 8086 processors, all the hardware interrupts initiated through INTR pin are maskable by clearing interrupt flag (IF). The interrupt initiated through NMI pin and all software interrupts are non-maskable.

Sources of Interrupts in 8086.

An interrupt in 8086 can come from one of the following three sources.

1. One source is from an external signal applied to NMI or INTR input pin of the processor. The interrupts initiated by applying appropriate signals to these input pins are called hardware interrupts.
2. A second source of an interrupt is execution of the interrupt instruction "INT n", where n is the type number. The interrupts initiated by "INT n" instructions are called software interrupts.
3. The third source of an interrupt is from some condition produced in the 8086 by the execution of an instruction. An example of this type of interrupt is divide by zero interrupt. Program execution will be automatically interrupted if you attempt to divide an operand by zero. Such conditional interrupts are also, known as exceptions

Interrupts Cycle of 8086:

The 8086 microprocessor has 256 types of interrupts. INTEL has assigned a type number to each interrupt. The type numbers are in the range of 0 to 255. The 8086 processor has dual facility of initiating these 256 interrupts. The interrupts can be initiated either by executing "INT n" instruction where n is the type number or the interrupt can be initiated by sending an appropriate signal to INTR input pin of the processor.

For the interrupts initiated by software instruction" INT n ", the type number is specified by the instruction itself. When the interrupt is initiated through INTR pin, then the processor runs an interrupt acknowledge cycle to get the type number. (i.e., the interrupting device should supply the type number through D0- D7 lines when the processor requests for the same through interrupt acknowledge cycle).

The kinds of interrupts and their designated types are summarized in figure by illustrating the layout of their pointers within the memory. Only the first five types have explicit definitions; the other

types may be used by interrupt instructions or external interrupts. From the figure, it is seen that the ty... associated with a division error interrupt is 0.

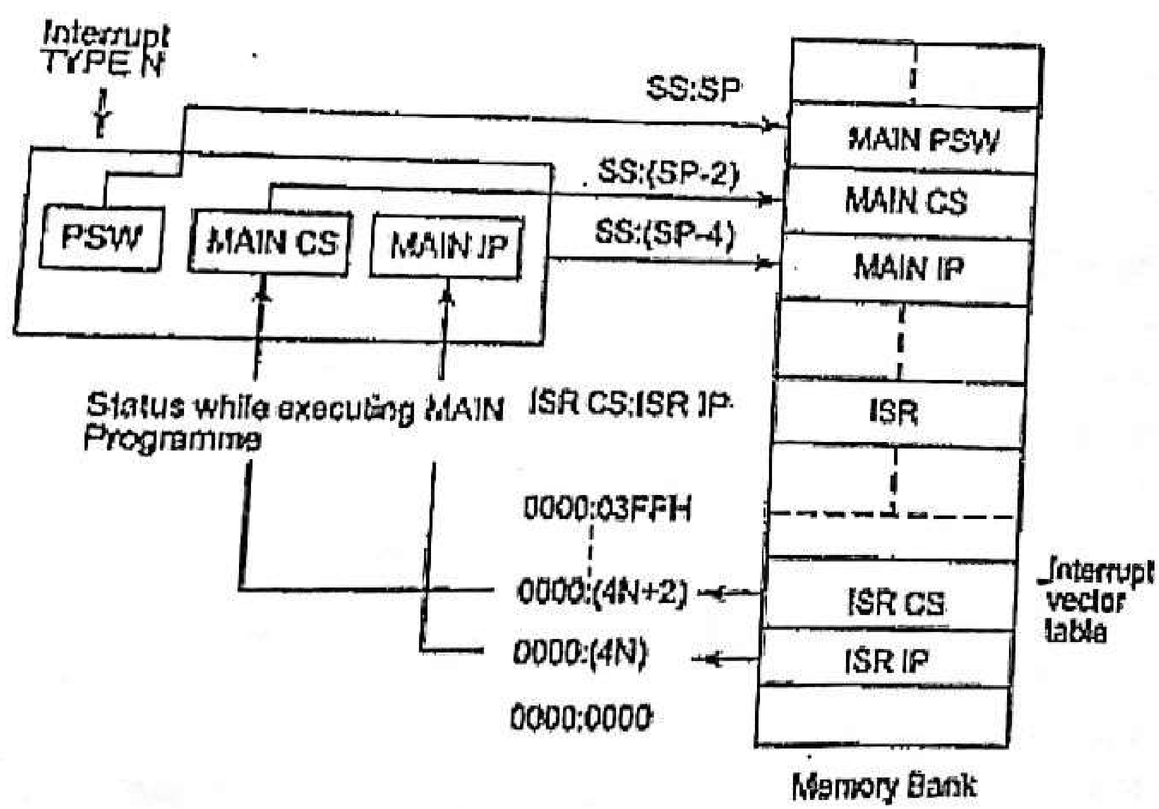
Therefore, if a division by 0 is attempted, the processor will push the current contents of the PSW, CS and IP into the stack, fill the IP and CS registers from the addresses 00000 to 00003, and continue executing at the address indicated by the new contents of IP and CS.

A division error interrupt occurs any time a DIV or IDIV instruction is executed with the quotient exceeding the range, regardless of the IF (Interrupt flag) and TF (Trap flag) status.

The type 1 interrupt is the single-step interrupt (Trap interrupt) and is the only interrupt controlled by the TF flag. If the TF flag is enabled, then an interrupt will occur at the end of the next instruction that will cause a branch to the location indicated by the contents of 00004H to 00007H.The single step interrupt is used primarily for debugging which gives the programmer a snapshot of his program after each instruction is executed.

The type 2 interrupt is the non-maskable external interrupt. It is the only external interrupt that can occur regardless of the IF flag setting. It is caused by a signal sent to the CPU through the non-maskable interrupt pin.

The remaining interrupt types correspond to interrupts instructions imbedded in the interrupt program or to external interrupts. The interrupt instructions are summarized below and their interrupts are not controlled by the IF flag.

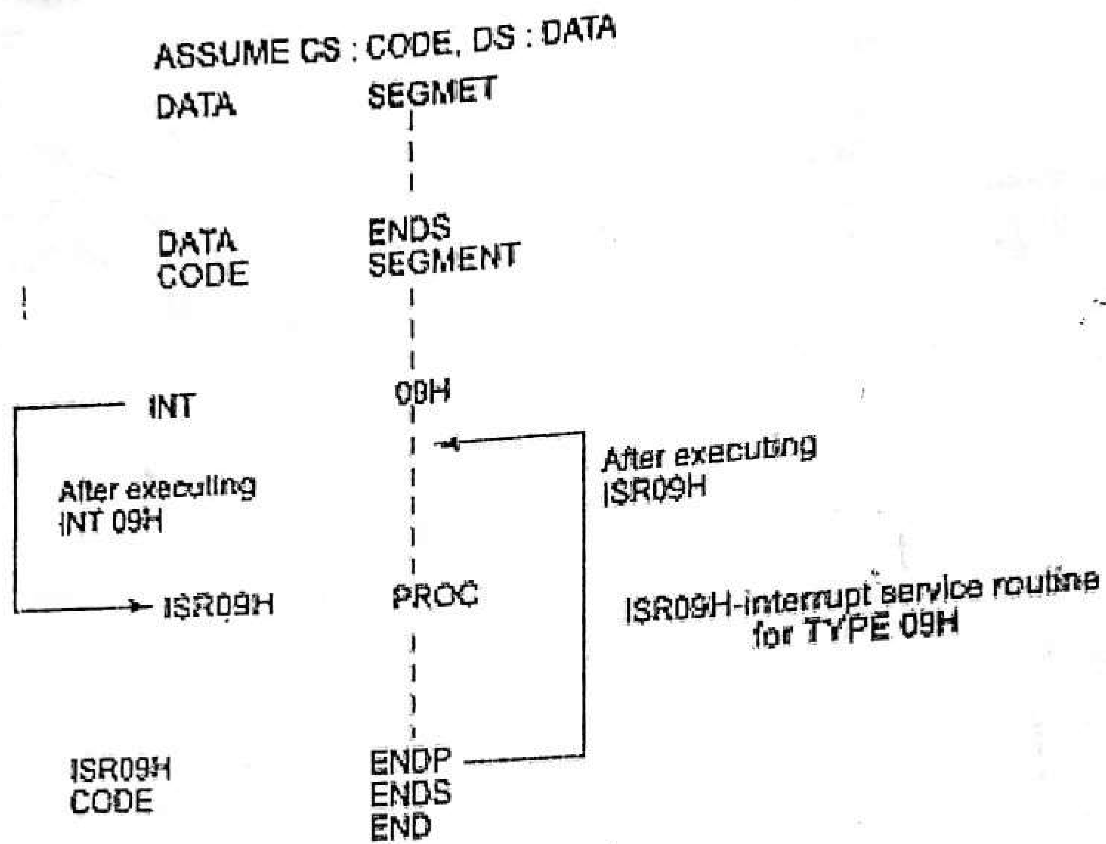


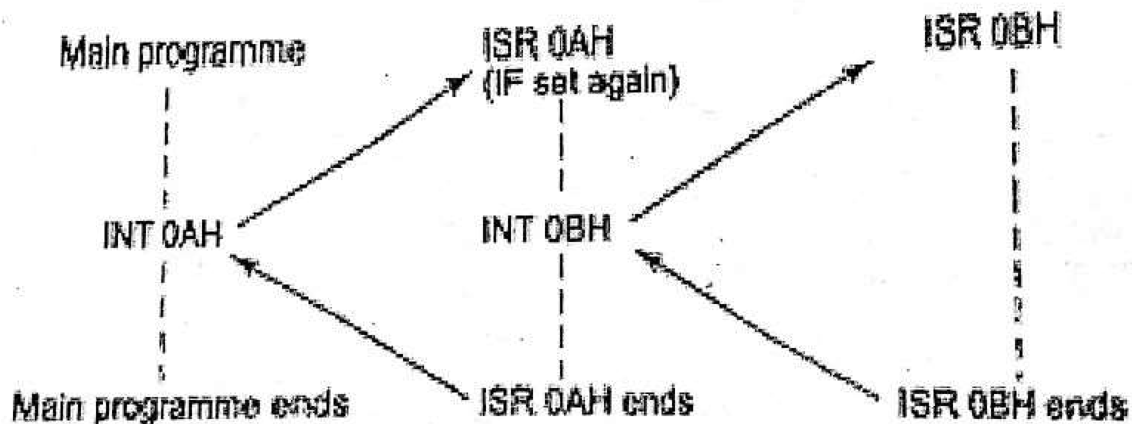
Interrupt Response Sequence

Interrupt Type No.	Contents	Address	
Pointer for type 0	New (IP) for type 0	00000	Reserved for divide error
	New (CS) for type 0		
Pointer for type 1	New (IP) for type 1	00004	Reserved for single step trap-TF must be set
	New (CS) for type 1		
Pointer for type 2	New (IP) for type 2	00008	Reserved for nonmaskable interrupt
	New (CS) for type 2		
Pointer for type 3	New (IP) for type 3	0000C	Reserved for one-byte interrupt instruction,INT
	New (CS) for type 3		
Pointer for type 4	New (IP) for type 4	00010	Reserved for overflow, INT0 instruction
	New (CS) for type 4		
		00014	Reserved for two-byte INT instructions and maskable external interrupts (00014 to 00400)
Pointer for type N	New (IP) for type N	4*N	
	New (CS) for type N		
		4*N + 4	
Pointer for type 255	New (IP) for type 255	003FC	
	New (CS) for type 255		
		00400	

Structure of Interrupt vector table of 8086

Interrupt Programming:





Transfer of Control for Nested Interrupts

UNIT – II: 8086 PROGRAMMING

Program Development Steps:

Major steps in developing an assembly language program

- Defining the Problem
- Representing program operations
- Finding the right instructions
- Writing a program

Defining the problems

- Find out the problem.

  For example: Sensing temperature, detecting fire, detecting smock, decoder-encoder, Intelligent machine controller etc.

Representing program operations

- Formula or sequence of operations used to solve a programming problem is called as the algorithm.
- There are two ways of representing algorithms:
  - Flowchart
  - Structured programming and pseudo code

Finding the right instructions

- Instructions in 8086 are mainly divided into following categories
- Data Transfer Instructions
- Arithmetic Instruction
- Bit manipulation Instruction
- String Instruction
- Program execution transfer Instruction
- Processor control Instruction

Writing a program

- We need to do the following steps to write the program effectively:
  - INITIALIZATION INSTRUCTIONS: used to initialize various parts of the program like segment registers, flags and programmable port devices.
  - STANDARD PROGRAM FORMAT: it's a tabular format containing ADDRESS, DATA OR CODE, LABELS, MNEM, OPERAND(S) and COMMENTS as the columns.

- DOCUMENTATION: you should document the program. E.g. each page of the document contains page number and name of the program, heading block containing the abstract about the program, comments should be added wherever necessary.

Instruction Set of 8086:

The 8086 instructions are categorized into the following main types.

a. Data Copy / Transfer Instructions
b. Arithmetic and Logical Instructions
c. Branch Instructions
d. Loop Instructions
e. Machine Control Instructions
f. Flag Manipulation Instructions
g. Shift and Rotate Instructions
h. String Instructions

a) Data Transfer Instruction:

These types of instructions are used to transfer data from source operand to destination operand. All the store, load, move, exchange, input and output operations belong to this category.

b) Arithmetic and Logical Instructions:

All the instructions performing arithmetic, logical, increment, decrement, compare and scan instructions belong to this category.

c) Branch Instructions:

These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instructions belong to this category.

d) Loop Instructions:

The *LOOP*, *LOOPNZ* and *LOOPZ* instructions belong to this category. These are useful to implement different loop structures.

e) Machine control Instructions:

These instructions control the machine status. *NOP*, *HLT*, *WAIT* and *LOCK* instructions belongs to this category.

f) Flag Manipulation Instructions:

All instructions which directly affect the flag register belong to this category. The instructions *CLD*, *STD*, *CLI*, *STI* etc. belong to this category.

g) Shift and Rotate Instructions:

These instructions involve the bitwise shifting or rotation in either direction with or without a count in *CX*.

h) String Instructions:

These instructions involve string manipulation operations like load, scan, compare, store etc. These instructions are only to be operated upon the string.

a) Data Copy / Transfer Instructions: -

MOV:

This instruction copies a word or a byte of data from some source to a destination.

The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number.

MOV AX, BX

MOV AX, 5000H

MOV AX, [SI]

MOV AX, [2000H]

MOV AX, 50H [BX]

MOV [734AH], BX

MOV DS, CX

MOV CL, [357AH]

Direct loading of the segment registers with immediate data is not permitted.

PUSH: Push to Stack

This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction.

Ex: - PUSH AX

PUSH DS

PUSH [5000H]

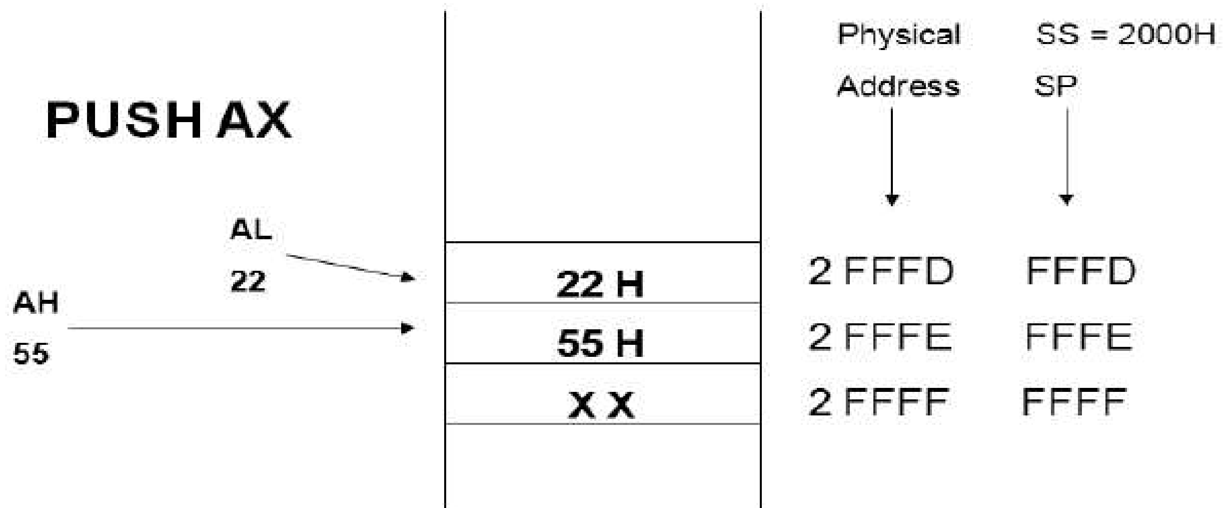


Fig: Push data to Stack memory

POP: Pop from Sack

This instruction when executed, loads the specified register/memory location with the contents of the memory location of which the address is formed using the current stack segment and stack pointer. The stack pointer is incremented by 2

Ex: - POP AX
POP DS
POP [5000H]

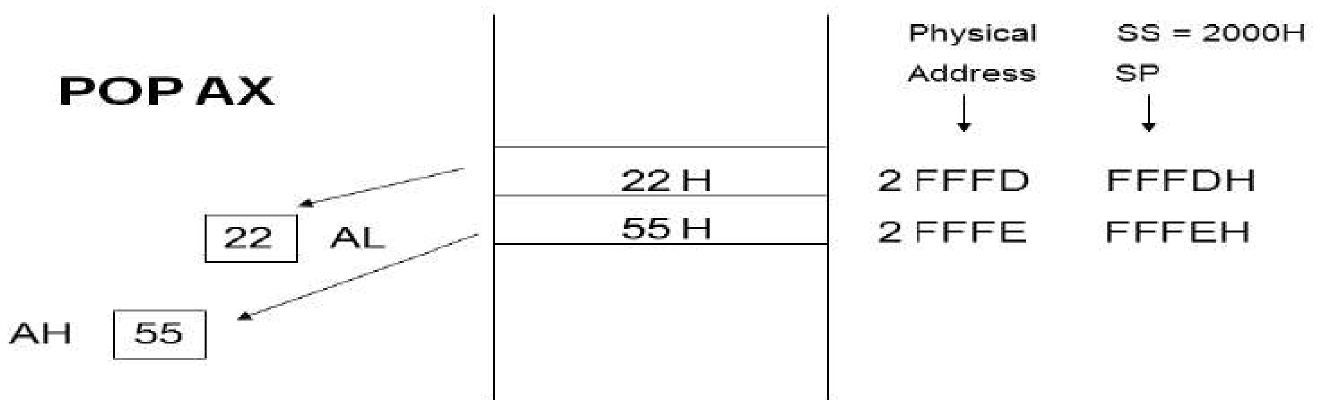


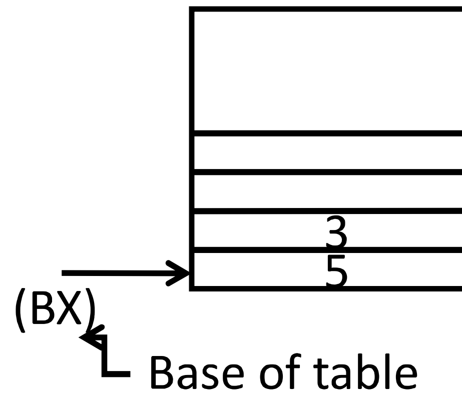
Fig: Poping Register content from stack memory

XCHG: Exchange byte or word

This instruction exchanges the contents of the specified source and destination operands

Ex: - XCHG [5000H], AX
XCHG BX, AX

XLAT: Translate byte using look-up table



Ex: - MOV BX, OFFSET TABLE

MOV AL, 00H

XLAT

(AL) ← 5

AL← [BX+AL]

Simple input and output port transfer Instructions:

IN:

Copy a byte or word from specified port to accumulator.

Ex: - IN AL,03H

IN AX, DX

OUT:

Copy a byte or word from accumulator specified port.

Ex: - OUT 03H, AL

OUT DX, AX

LEA:

Load effective address of operand in specified register.

[reg] offset portion of address in DS

Ex: - LEA reg, offset

LDS:

Load DS register and other specified register from memory.

[reg] [mem]

[DS] [mem + 2]

Ex: - LDS reg, mem

LES:

Load ES register and other specified register from memory.

[reg] [mem]

[ES] [mem + 2]

Ex: - LES reg, mem

Flag transfer instructions:

LAHF:

Load (copy to) AH with the low byte the flag register.

[AH] [Flags low byte]

Ex: - LAHF

SAHF:

Store (copy) AH register to low byte of flag register.

[Flags low byte] ← [AH]

Ex: - SAHF

PUSHF:

Copy flag register to top of stack.

[SP] ← [SP] – 2

[[SP]] ← [Flags]

Ex: - PUSHF

POPF:

Copy word at top of stack to flag register.

[Flags] ← [[SP]]

[SP] ← [SP] + 2

Ex: - POPF

b) Arithmetic Instructions:

The 8086 provides many arithmetic operations: addition, subtraction, negation, multiplication and comparing two values.

ADD:

The add instruction adds the contents of the source operand to the destination operand.

Ex: - ADD AX, 0100H

ADD AX, BX

ADD AX, [SI]

ADD AX, [5000H]

ADD [5000H], 0100H

ADD 0100H

ADC: Add with Carry

This instruction performs the same operation as ADD instruction, but adds the carry flag to the result.

Ex: - ADC 0100H
ADC AX, BX
ADC AX, [SI]
ADC AX, [5000]
ADC [5000], 0100H

SUB: Subtract

The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand.

Ex: - SUB AX, 0100H
SUB AX, BX
SUB AX, [5000H]
SUB [5000H], 0100H

SBB: Subtract with Borrow

The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand

Ex: - SBB AX, 0100H
SBB AX, BX
SBB AX, [5000H]
SBB [5000H], 0100H

INC: Increment

This instruction increases the contents of the specified Register or memory location by 1. Immediate data cannot be operand of this instruction.

Ex: - INC AX
INC [BX]
INC [5000H]

DEC: Decrement

The decrement instruction subtracts 1 from the contents of the specified register or memory location.

Ex: - DEC AX
DEC [5000H]

NEG: Negate

The negate instruction forms 2's complement of the specified destination in the instruction. The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.

Ex: - NEG AL

AL = 0011 0101 35H Replace number in AL with its 2's complement

AL = 1100 1011 = CBH

CMP: Compare

This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location

Ex: - CMP BX, 0100H

CMP AX, 0100H

CMP [5000H], 0100H

CMP BX, [SI]

CMP BX, CX

MUL: Unsigned Multiplication Byte or Word

This instruction multiplies an unsigned byte or word by the contents of AL.

Ex: - MUL BH; (AX) ← (AL) x (BH)

MUL CX; (DX) (AX) ← (AX) x (CX)

MUL WORD PTR [SI]; (DX) (AX) ← (AX) x ([SI])

IMUL: Signed Multiplication

This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX.

Ex: - IMUL BH

IMUL CX

IMUL [SI]

CBW: Convert Signed Byte to Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be sign extension of AL.

Ex: - CBW

AX= 0000 0000 1001 1000 Convert signed byte in AL signed word in AX.

Result in AX = 1111 1111 1001 1000

CWD: Convert Signed Word to Double Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be sign extension of AL.

Ex: - CWD

Convert signed word in AX to signed double word in DX: AX

DX= 1111 1111 1111 1111

Result in AX = 1111 0000 1100 0001

DIV: Unsigned division

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word by a word.

Ex: - DIV CL ; Word in AX / byte in CL
; Quotient in AL, remainder in AH
DIV CX ; Double word in DX and AX / word in CX, and Quotient in AX, remainder in DX

AAA: ASCII Adjust After Addition

The **AAA** instruction is executed after an ADD instruction that adds two ASCII coded operand to give a byte of result in AL. The **AAA** instruction converts the resulting contents of AL to a unpacked decimal digits.

AAA operation:

1) In AL If rightmost nibble is >9 (ie) A to F or Auxiliary Flag=1
   ADD 6 to rightmost nibble
2) Clear left nibble form AL.
3) In AH ADD 1
4) Set Carry and Auxiliary Carry

Ex: - ADD CL, DL ; [CL] = 34H = ASCII for 4
; [DL] = 38H = ASCII for 8
; Result [CL] = 6CH
AAA ; [AL] =02, unpacked BCD for 2
; [AH] =01, unpacked BCD for 1

AAS: ASCII Adjust AL after Subtraction

This instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. The procedure is similar to AAA instruction except for the subtraction of 06 from AL.

AAS operation:

1) AAS checks the rightmost nibble in AL If rightmost nibble is >9 (ie) A to F Or Auxiliary Flag=1, Then Subtract 6 from rightmost nibble
2) Clear left nibble in AL.
3) Subtracts 1 from AH

4) Set Carry and Auxiliary Carry

Ex: - MOV AL, 34H

SUB AL, 38H ; AX=00FC

AAS ; AX= FF06 ten's complement i.e -4 (Borrow one from AH)

OR AL, 30H ; AL=34

AAM: ASCII Adjust after Multiplication

This instruction, after execution, converts the product available In AL into unpacked BCD format.

AAM performs the following operations

1) Divides AL value by 10 (0AH)

2) Stores Quotient in AH

3) Store Remainder in AL

Ex: - MOV AL, 04 ; AL = 04

MOV BL, 09 ; BL = 09

MUL BL ; AX = AL*BL ; AX=0024H

AAM ; AH = 03H, AL=06H

AAD: ASCII Adjust before Division

This instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. In the instruction sequence, this instruction appears Before DIV instruction.

Operations done by AAD instruction

1) AAD multiplies the AH by 10(0Ah).

2) Then adds the product to AL and clears the AH

Ex: - AX 05 08

AAD result in AL 00 3A 58D = 3A H in AL

The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH. Where 3A is the hexadecimal equivalent of 58 (decimal).

DAA: Decimal Adjust Accumulator

This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL.

Ex: - AL = 53H, CL = 29H

ADD AL, CL ; AL ← (AL) + (CL)

; AL ← 53 + 29

; AL ← 7C

DAA ; AL ← 7C + 06 (as C > 9)

; AL 82

DAS: Decimal Adjust after Subtraction

This instruction converts the result of the subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only.

Ex: - AL = 75H, BH = 46H

```
SUB AL, BH      ; AL ← 2 FH = (AL) - (BH)
                ; AF = 1
DAS             ; AL ← 29 (as F>9, F - 6 = 9)
```

Logical Instructions:

AND: Logical AND

This instruction bit by bit ANDs the source operand that may be an immediate register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Ex: - AND AX, 0008H

AND AX, BX

OR: Logical OR

This instruction bit by bit ORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Ex: - OR AX, 0008H

OR AX, BX

NOT: Logical Invert

This instruction complements the contents of an operand register or a memory location, bit by bit.

Ex: - NOT AX

NOT [5000H]

XOR: Logical Exclusive OR

This instruction bit by bit XORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.

Ex: - XOR AX, 0098H

XOR AX, BX

TEST: Logical Compare Instruction

The TEST instruction performs a bit by bit logical AND operation on the two operands. The result of this AND ing operation is not available for further use, but flags are affected.

Ex: - TEST AX, BX

TEST [0500], 06H

c) Branch Instructions:

Branch Instructions transfers the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location to be transferred.

The Branch Instructions are classified into two types

i. Unconditional Branch Instructions.

ii. Conditional Branch Instructions.

Unconditional Branch Instructions:

In Unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

CALL: Unconditional Call

This instruction is used to call a Subroutine (Procedure) from a main program. Address of procedure may be specified directly or indirectly.

There are two types of procedure depending upon whether it is available in the same segment or in another segment.

i. Near CALL i.e., ±32K displacement.

ii. Far CALL i.e., anywhere outside the segment.

On execution this instruction stores the incremented IP & CS onto the stack and loads the CS & IP registers with segment and offset addresses of the procedure to be called.

RET: Return from the Procedure.

At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with Flags are retrieved into the CS, IP and Flag registers from the stack and execution of the main program continues further.

INT N: Interrupt Type N.

In the interrupt structure of 8086, 256 interrupts are defined corresponding to the types from 00H to FFH. When INT N instruction is executed, the type byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from memory block in 0000 segment.

INTO: Interrupt on Overflow

This instruction is executed, when the overflow flag OF is set. This is equivalent to a Type 4 Interrupt instruction.

JMP: Unconditional Jump

This instruction unconditionally transfers the control of execution to the specified address using an 8-bit or 16-bit displacement. No Flags are affected by this instruction.

IRET: Return from ISR

When it is executed, the values of IP, CS and Flags are retrieved from the stack to continue the execution of the main program.

d) LOOP Instructions:

LOOP Unconditionally

This instruction executes the part of the program from the Label or address specified in the instruction up to the LOOP instruction CX number of times. At each iteration, CX is decremented automatically and JUMP IF NOT ZERO structure.

Ex: -

```
          MOV CX, 0004H
          MOV BX, 7526H
Label 1:  MOV AX, CODE
          OR BX, AX
          LOOP Label 1
```

Conditional Branch Instructions:

When this instruction is executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the opcode is satisfied. Otherwise execution continues sequentially.

JZ/JE Label

Transfer execution control to address 'Label', if ZF=1.

JNZ/JNE Label

Transfer execution control to address 'Label', if ZF=0

JS Label

Transfer execution control to address 'Label', if SF=1.

JNS Label

Transfer execution control to address 'Label', if SF=0.

JO Label

Transfer execution control to address 'Label', if OF=1.

JNO Label

Transfer execution control to address 'Label', if OF=0.

JNP Label

Transfer execution control to address 'Label', if PF=0.

JP Label

Transfer execution control to address 'Label', if PF=1.

JB Label

Transfer execution control to address 'Label', if CF=1.

JNB Label

Transfer execution control to address 'Label', if CF=0.

JCXZ Label

Transfer execution control to address 'Label', if CX=0

Conditional LOOP

LOOPZ / LOOPE Label

Loop through a sequence of instructions from label while ZF=1 and CX=0.

LOOPNZ / LOOPENE Label

Loop through a sequence of instructions from label while ZF=1 and CX=0.

e) Flag Manipulation and Processor Control Instructions:

These instructions control the functioning of the available hardware inside the processor chip. These instructions are categorized into two types:

1. Flag Manipulation instructions.
2. Machine Control instructions.

Flag Manipulation instructions:

The Flag manipulation instructions directly modify some of the Flags of 8086.

i. CLC – Clear Carry Flag.

ii. CMC – Complement Carry Flag.

iii. STC – Set Carry Flag.

iv. CLD – Clear Direction Flag.

v. STD – Set Direction Flag.

vi. CLI – Clear Interrupt Flag.

vii. STI – Set Interrupt Flag.

Machine Control instructions

The Machine control instructions control the bus usage and execution

i. WAIT – Wait for Test input pin to go low.

ii. HLT – Halt the process.

iii. NOP – No operation.

iv. ESC – Escape to external device like NDP

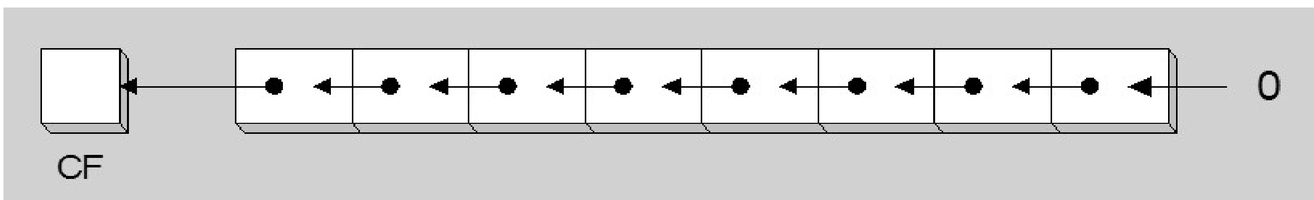
v. LOCK – Bus lock instruction prefix.

f) Shift & Rotate Instructions:

SAL/SHL: SAL / SHL destination, count

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination to the left and 0 is stored at LSB position. The MSB is shifted into the carry flag. The destination can be a byte or a word.

It can be in a register or in a memory location. The number of shifts is indicated by count.



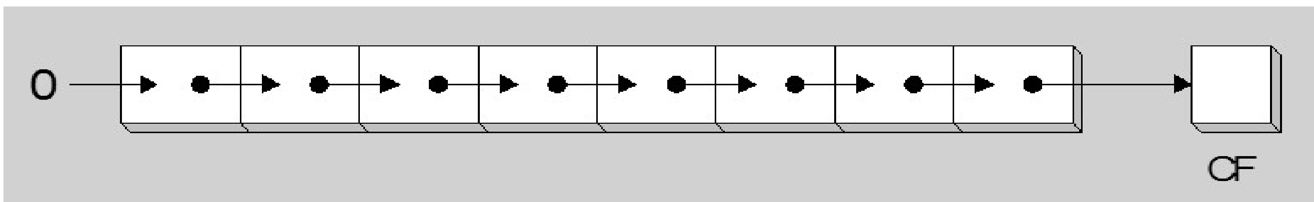
Ex: - SAL CX, 1

SAL AX, CL

SHR: SHR destination, count

This instruction shifts each bit in the specified destination to the right and 0 is stored at MSB position. The LSB is shifted into the carry flag. The destination can be a byte or a word.

It can be a register or in a memory location. The number of shifts is indicated by count.



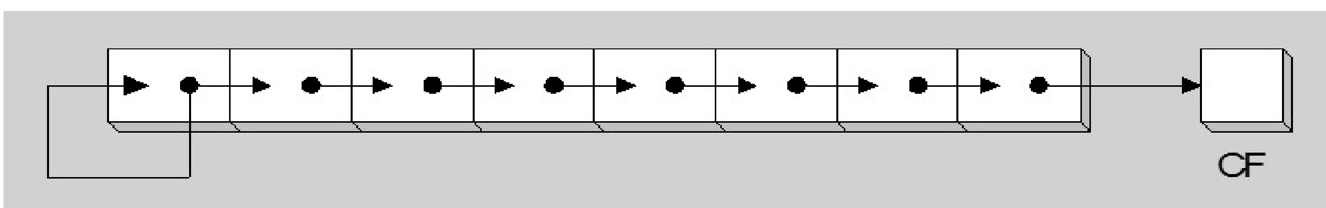
Ex: - SHR CX, 1

MOV CL, 05H

SHR AX, CL

SAR: SAR destination, count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. The LSB will be shifted into CF.



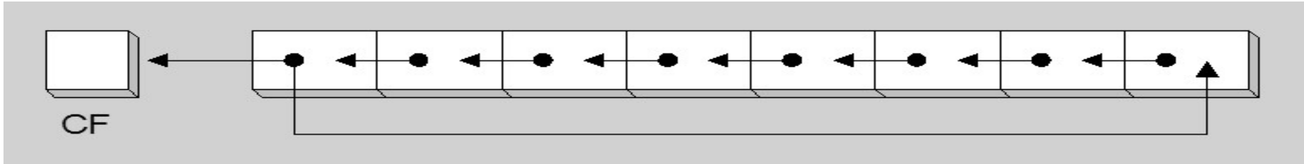
Ex: - SAR BL, 1

MOV CL, 04H

SAR DX, CL

ROL Instruction: ROL destination, count

This instruction rotates all bits in a specified byte or word to the *left* some number of bit positions. MSB is placed as a new LSB and a new CF.



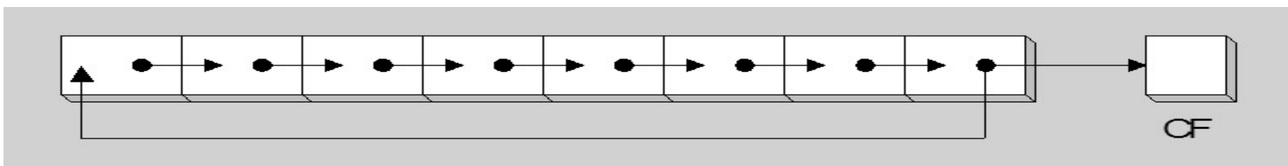
Ex: - ROL CX, 1

MOV CL, 03H

ROL BL, CL

ROR Instruction: ROR destination, count

This instruction rotates all bits in a specified byte or word to the *right* some number of bit positions. LSB is placed as a new MSB and a new CF.



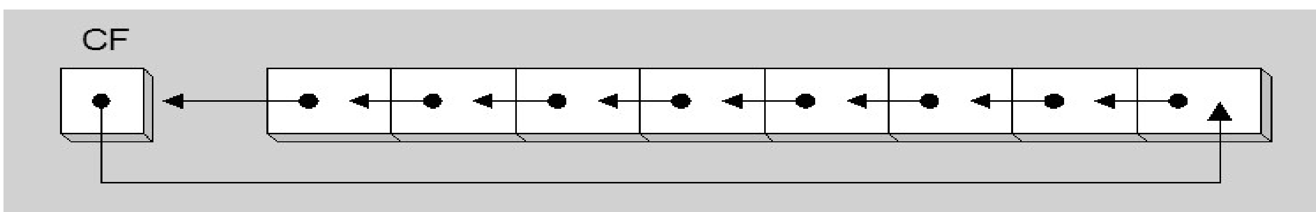
Ex: - ROR CX, 1

MOV CL, 03H

ROR BL, CL

RCL Instruction: RCL destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the *left along with the carry flag*. MSB is placed as a new carry and previous carry is place as new LSB.



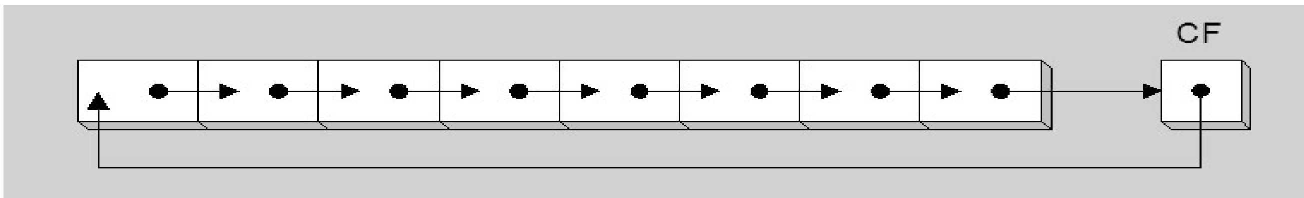
Ex: - RCL CX, 1

MOV CL, 04H

RCL AL, CL

RCR Instruction: RCR destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the right *along with the carry flag*. LSB is placed as a new carry and previous carry is place as new MSB.



Ex: - RCR CX, 1

MOV CL, 04H

RCR AL, CL

g) String Manipulation Instructions:

A series of data byte or word available in memory at consecutive locations, to be referred as Byte String or Word String. A String of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent.

The 8086 supports a set of more powerful instructions for string manipulations for referring to a string, two parameters are required.

I. Starting and End Address of the String.

II. Length of the String.

The length of the string is usually stored as count in the CX register. The incrementing or decrementing of the pointer, in string instructions, depends upon the Direction Flag (DF) Status. If it is a Byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two.

REP: Repeat Instruction Prefix

This instruction is used as a prefix to other instructions, the instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one).

i. REPE / REPZ - repeat operation while equal / zero.

ii. REPNE / REPNZ - repeat operation while not equal / not zero.

These are used for CMPS, SCAS instructions only, as instruction prefixes.

Prefix	Used with	Meaning
REP	MOVS STOS	Repeat while not end of string CX≠0
REPE/ REPZ	CMPS SCAS	CX≠0 & ZF=1
REPNE/REPNZ	CMPS SCAS	CX≠0 & ZF=0

Mnemonic	Meaning	Format	Operation	Flags affected
CLD	Clear DF	CLD	(DF) ← 0	DF
STD	Set DF	STD	(DF) ← 1	DF

MOVSB / MOVSW: Move String Byte or String Word

Suppose a string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of source string is located in the memory location whose address may be computed using SI (Source Index) and DS (Data Segment) contents.

The starting address of the destination locations where this string has to be relocated is given by DI (Destination Index) and ES (Extra Segment) contents.

Ex: - Block move program using the move string instruction

```
        MOV AX, DATA SEG ADDR
        MOV DS, AX
        MOV ES, AX
        MOV SI, BLK 1 ADDR
        MOV DI, BLK 2 ADDR
        MOV CK, N
        CLD                     ; DF=0
  NEXT: MOV SB
        LOOP NEXT
        HLT
```

CMPSB/SW: Compare String Byte or String Word

- The CMPS instruction can be used to compare two strings of byte or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero Flag is set.
- The REP instruction Prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP Prefix is False.

SCANSB/SW: Scan String Byte or String Word

This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The String is pointed to by ES: DI register pair. The length of the string s stored in CX.

The DF controls the mode for scanning of the string. Whenever a match to the specified operand is found in the string, execution stops and the zero Flag is set. If no match is found, the zero flag is reset.

LODSB/SW: Load String Byte or String Word

The LODS instruction loads the AL / AX register by the content of a string pointed to by DS: SI register pair. The SI is modified automatically depending upon DF, If it is a byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other Flags are affected by this instruction.

STOSB/SW: Store String Byte or String Word

- The STOS instruction Stores the AL / AX register contents to a location in the string pointer by ES: DI register pair. The DI is modified accordingly; No Flags are affected by this instruction.
- The direction Flag controls the String instruction execution, the source index SI and Destination Index DI are modified after each iteration automatically.
- If DF=1, then the execution follows auto-decrement mode, SI and DI are decremented automatically after each iteration.
- If DF=0, then the execution follows auto-increment mode. In this mode, SI and DI are incremented automatically after each iteration.

Ex:- Clearing a block of memory with a STOSB operation.

```
        MOV AX, 0
        MOV DS, AX
        MOV ES, AX
        MOV DI, A000
        MOV CX, OF
        CLD
AGAIN:  STOSB
        LOOPNE AGAIN
NEXT:   Clear A000 to A00F to 00H
```

Addressing Modes of 8086:

The default segment for the addressing modes using BP and SP is SS. For all other addressing modes, the default segments are DS or ES.

Addressing mode indicates a way of locating data or operands.

Different addressing modes of 8086:

1. Immediate:

In this addressing mode, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

Ex: **- MOV AX, 0050_H**

Here 0050_H is the immediate data and it is moved to register AX. The immediate data may be 8-bit or 16-bit in size.

2. Direct:

In the direct addressing mode, a 16-bit address (offset) is directly specified in the instruction as a part of it.

Ex: **- MOV AX, [1000_H]**

Here data resides in a memory location in the data segment, whose effective address is $10H \times DS + 1000H$

3. Register:

In register addressing mode, the data is stored in a register and it is referred using the particular register. All the registers except IP may be used in this mode.

Ex: **- MOV AX, BX**

4. Register Indirect:

In this addressing mode, the address of the memory location which contains data or operand is determined in an indirect way using offset registers. The offset address of data is in either BX or SI or DI register. The default segment register is either DS or ES.

Ex: - **MOV AX,** $[BX]$

The data is present in a memory location in DS whose offset is in BX. The effective address is $10H \times DS + [BX]$

5. Indexed:

In this addressing mode offset of the operand is stored in one of the index register. DS and ES are the default segments for index registers SI and DI respectively

Ex: **- MOV AX,** $[SI]$

The effective address of the data is $10H \times DS + [SI]$

6. Register Relative: In this addressing mode the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers BX, BP, SI and DI in the default either DS or ES segment.

Ex: - **MOV AX, 50H**$\left[BX\right]$

The effective address of the data is $10H \times DS + 50H + \left[BX\right]$

7. Based Indexed:

In this addressing mode the effective address of the data is formed by adding the content of a base register (any one of BX or BP) to the content of an index register (any one of SI or DI). The default segment register may be ES or DS.

Ex: - **MOV AX,**$\left[BX\right]\left[SI\right]$

The effective address is $10H \times DS + \left[BX\right] + \left[SI\right]$

8. Relative Based Indexed:

The effective address is formed by adding an 8-bit or 16-bit displacement with the sum of contents of any one of the base register (BX or BP) and any one of the index registers in a default segment.

Ex: - **MOV AX, 50H**$\left[BX\right]\left[SI\right]$

Here 50H is an immediate displacement. The effective address is $10H \times DS + \left[BX\right] + \left[SI\right] + 50H$.

9. Intrasegment Direct Mode:

In this mode, the address to which the control is to be transferred lies in the segment in which the control transfer instruction lies and appears directly in the instruction as an immediate displacement value. The displacement is computed relative to the content of the instruction pointer IP.

10. Intrasegment Indirect Mode:

This mode is similar to intrasegment direct mode except the displacement to which control is to be transferred is passed to the instruction indirectly. Here the branch address is found as the content of a register or a memory location.

11. Intersegment Direct Mode:

In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

12. Intersegment Indirect Mode:

This mode is similar to intersegment direct mode except the address to which the control is to be transferred is passed to the instruction indirectly. This information is kept in a memory block of 4 bytes: IP (LSB), IP(MSB), LS(LSR) and CS(MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode.

Assembler Directives and Operators:

The main advantage of machine language programming is that the memory control is directly in the hands of the programmer, so that, he may be able to manage the memory of the system more efficiently. On the other hand, the disadvantages are more prominent. The programming, coding and resource management techniques are tedious. The programmer has to take care of these functions hence the chances of human errors are more. The programs are difficult to understand unless one has a thorough technical knowledge of the processor architecture and instruction set.

The assembly language programming is simpler as compared to the machine language programming. The instruction mnemonics are directly written in the assembly language programs. The programs are now more readable to users than the machine language programs. The main improvement in assembly language over machine language is that the address values and the constants can be identified by labels. If the labels are suggestive, then certainly the program will become more understandable, and each time the programmer will not have to remember the different constants and the addresses at which they are stored, throughout the programs. The labels may help to identify the addresses and constants. Due to this facility, the tedious byte handling and manipulations are got rid of. Similarly, now different logical segments and routines may be assigned with the labels rather than the different addresses. The memory control feature of machine language programming is left unchanged by providing storage define facilities in assembly language programming. The documentation facility which was not possible with machine language programming is now available in assembly language.

An assembler is a program used to convert an assembly language program into the equivalent machine code modules which may further be converted to executable codes. The assembler decides the address of each label and substitutes the values for each of the constants and variables. It then forms the machine code for the mnemonics and data in the assembly language program. While doing these things, the assembler may find out syntax errors. The logical errors or other programming errors are not found out by the assembler. For completing all these tasks, an assembler needs some hints from the programmer, i.e. the required storage for a particular constant or variable, logical names of the segments, types of the different routines and modules, end of file, etc. These, types of hints are given to the assembler using some predefined alphabetical strings called assembler directives. Assembler directives help the assembler to correctly understand the assembly language programs to prepare the codes.

Another type of hint which helps the assembler to assign a particular constant with a label or initialize particular memory locations or labels with constants is called an operator. Rather, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler (MASM) or Turbo Assembler (TASM).

DIRECTIVES:

DB: Define Byte

The DB directive is used to reserve byte or bytes of memory locations in the available memory.

Example:

LIST DB 0lH, 02H, 03H, 04H

This statement directs the assembler to reserve four memory locations for a list named LIST and initialize them with the above specified four values.

MESSAGE DB 'GOOD MORNING'

This makes the assembler reserve the number of bytes of memory equal to the number of characters in the string named MESSAGE and initialize those locations by the ASCII equivalent of these characters.

VALUE DB 50H

This statement directs the assembler to reserve 50H memory bytes and leave them uninitialized for the variable named VALUE.

DW: Define Word.

The DW directive serves the same purposes as the DB directive, but it now makes the assembler reserve the number of memory words (16-bit) instead of bytes.

Example:

WORDS DW 1234H, 4567H, 78ABH, 045CH

This makes the assembler reserve four words in memory (8 bytes), and initialize the words with the specified values in the statements.

Another option of the DW directive is explained with the DUP operator.

WDATA DW 5 DUP (6666H)

This statement reserves five words, i.e. 10-bytes of memory for a word label WDATA and initializes all the word locations with 6666H.

DQ: Define Quad word

This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialize it with the specified values.

DT: Define Ten Bytes.

The DT directive directs the assembler to define the specified variable requiring 10-bytes for its storage and initialize the 10 bytes with the specified values. The directive may be used in case of variables facing heavy numerical calculations, generally processed by numerical processors.

ASSUME: **Assume Logical Segment Name**

The ASSUME directive is used to inform the assembler, the names of the logical segments to be assumed for different segments used in the program.

```
ASSUME CS: CODE, DS: DATA, SS: STACK
```

END: END of Program

The END directive marks the end of an assembly language program.

ENDS: END of Segment

This directive marks the end of a logical segment. The logical segments are assigned with the names using the ASSUME directive.

```
DATA SEGMENT
        .
        .
DATA ENDS
ASSUME CS: CODE, DS: DATA
CODE SEGMENT.
        .
        .
CODE ENDS
END
```

ENDP: END of Procedure.

In assembly language programming, the subroutines are called procedures. The ENDP directive is used to indicate the end of a procedure.

Ex: -

```
PROCEDURE STAR
        .
        .
        .
STAR ENDP
```

EVEN: Align on Even Memory Address

The EVEN directive updates the location counter to the next even address if the current location counter contents are not even, and assigns the following routine or variable or constant to that address.

```
Ex: -   EVEN
        PROCEDURE ROOT
        .
        .
        .
        ROOT ENDP
```

EQU: Equate

The directive EQU is used to assign a label with a value or a symbol. The use of this directive is just to reduce the recurrence of the numerical values or constants in a program code.

Ex: -

```
        LABEL EQU 0500H
        ADDITION EQU ADD
```

EXTRN: External and PUBLIC: Public

The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have already been defined in some other assembly language modules. While in the other module, where the names, procedures and labels actually appear, they must be declared public, using the PUBLIC directive.

If one wants to call a procedure FACTORIAL appearing in MODULE 1 from MODULE 2; in MODULE1, it must be declared PUBLIC using the statement PUBLIC FACTORIAL and in module 2, it must be declared external using the declaration EXTRN FACTORIAL. The statement of declaration EXTRN must be accompanied by the SEGMENT and ENDS directives of the MODULE 1, before it is called in MOBULE 2. Thus, the MODULE 1 and MODULE 2 must have the following declarations.

```
Ex: -   MODULEl SEGMENT

                PUBLIC FACTORIAL FAR

        MODULEl ENDS
        MODULE2 SEGMENT

                EXTRN FACTORIAL FAR


        MODULE2 ENDS
```

GROUP: Group the Related segment

The directive is used to form logical groups of segments with similar purpose or type. This directive is used to inform the assembler to form a logical group of the following segment names.

Ex: - PROGRAM GROUP CODE, DATA, STACK

ASSUME CS: PROGRAM, DS: PROGRAM, SS: PROGRAM.

LABEL: Label

The Label directive is used to assign a name to the current content of the location counter.

A LABEL directive may be used to make a FAR jump as shown below.

A FAR jump cannot be made at a normal label with a colon.

LENGTH: Byte Length of a Label

This directive is not available in TASM. This is used to refer to the length of a data array or a string.

Ex: - MOV CX, LENGTH ARRAY

This statement, when assembled, will substitute the length of the array ARRAY in bytes, in the instruction.

ORG: Origin

The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement while starting the assembly process for a module, the assembler initializes a location counter to keep track of the allotted addresses for the module. If the ORG statement is not written in the program, the location counter is initialized to 0000.

PROC: Procedure

The PROC directive marks the start of a named procedure in the statement. Also, the types NEAR or FAR specify the type of the procedure, i.e whether it is to be called by the main program located within 64K of physical memory or not.

Ex: - RESULT PROC NEAR

ROUTINE PROC FAR

SEGMENT: Logical Segment

The SEGMENT directive marks the starting of a logical segment. The started segment is also assigned a name, i.e. label, by this statement.

OPERATORS:

OFFSET: Offset of a Label

When the assembler comes across the OFFSET operator along with a label, it first computes the 16-bit displacement (also called as offset interchangeably) of the particular label, and replaces the string 'OFFSET LABEL' by the computed displacement. This operator is used with arrays, strings, lables and procedures to decide their offsets in their default segments.

Ex: -

```
DATA SEGMENT
        LIST DB 10H
DATA ENDS
CODE SEGMENT
        MOV SI, OFFSET LIST
CODE ENDS
```

PTR: Pointer

The pointer operator is used to declare the type of a label, variable or memory operand. The operator PTR is prefixed by either BYTE or WORD.

- If the prefix is BYTE, then the particular label, variable or memory operand is treated as an 8-bit quantity, while if WORD is the prefix, then it is treated as a 16- bit quantity.

Ex: -

```
MOV AL, BYTE PTR [SI]    ; Moves content of memory location addressed by SI (8-bit) to AL
INC BYTE PTR [BX]        ; Increments byte contents of memory location addressed by BX
MOV BX, WORD PTR [2000H] ; Moves 16-bit content of memory location 2000H to BX, i.e.
                           [2000H] to BL [2001 H] to BH
INC WORD PTR [3000H]     ; Increments word contents of memory location 3000H
                           considering contents of 3000H (lower byte) and 3001 H
                           (higher byte) as a 16-bit number
```

SEG: Segment of a Label

The SEG operator is used to decide the segment address of the label, variable, or procedure and substitutes the segment base address in place of ‘SEG label’.

Ex: -

```
MOV AX, SEG ARRAY   ; This statement moves the segment address
MOV DS, AX          ; of ARRAY in which it is appearing, to register AX and then to DS.
```

SHORT

The SHORT operator indicates to the assembler that only one byte is required to code the displacement for a jump (i.e. displacement is within -128 to +127 bytes from the address of the byte next to the jump opcode).

Ex: -

```
JMP SHORT LABEL
```

N. Thirumalesh, M.Tech.

UNIT - II

Assembly Language programs involving Logical, Branch & CALL Instructions :

write a program for the addition of a series of 8-bit numbers. The series contains 100 (numbers)

Sol:
```
ASSUME CS: CODE , DS: DATA
DATA SEGMENT
NUMLIST DB 52H , 23H , ....
COUNT EQU 100D
RESULT DW 01H DUP (?)
DATA ENDS
CODE SEGMENT
        ORG 200H
        MOV AX, DATA
        MOV DS, AX
        MOV CX, COUNT
        XOR AX, AX
        XOR BX, BX
        MOV SI, OFFSET NUMLIST
AGAIN:  MOV BL, [SI]
        ADD AX, BX
        INC SI
        DEC CX
        JNZ AGAIN
        MOV DI, OFFSET RESULT
        MOV [DI], AX
        MOV AH, 4CH
        INT 21H
CODE ENDS
END
```

A program to find out the largest number from a given unordered array of 8-bit numbers, stored in the location starting from a known address.

Sol:-

```
ASSUME   CS: CODE, DS: DATA
DATA  SEGMENT
LIST  DB  52H, 23H, 56H, 45H, ....
COUNT  EQU  0FH
LARGEST  DB  01H  DUP (?)
DATA  ENDS
CODE  SEGMENT
START:      MOV   AX, DATA
            MOV   DS, AX
            MOV   SI, OFFSET LIST
            MOV   CL, COUNT
            MOV   AL, [SI]
AGAIN:      CMP   AL, [SI+1]
            JNL   NEXT
            MOV   AL, [SI+1]
NEXT:       INC   SI
            DEC   CL
            JNZ   AGAIN
            MOV   SI, OFFSET LARGEST
            MOV   [SI], AL
            MOV   AH, 4CH
            INT   21H
CODE  ENDS
END   START
```

AL: ~~52~~ 56

52 > 23

52 < 56

A

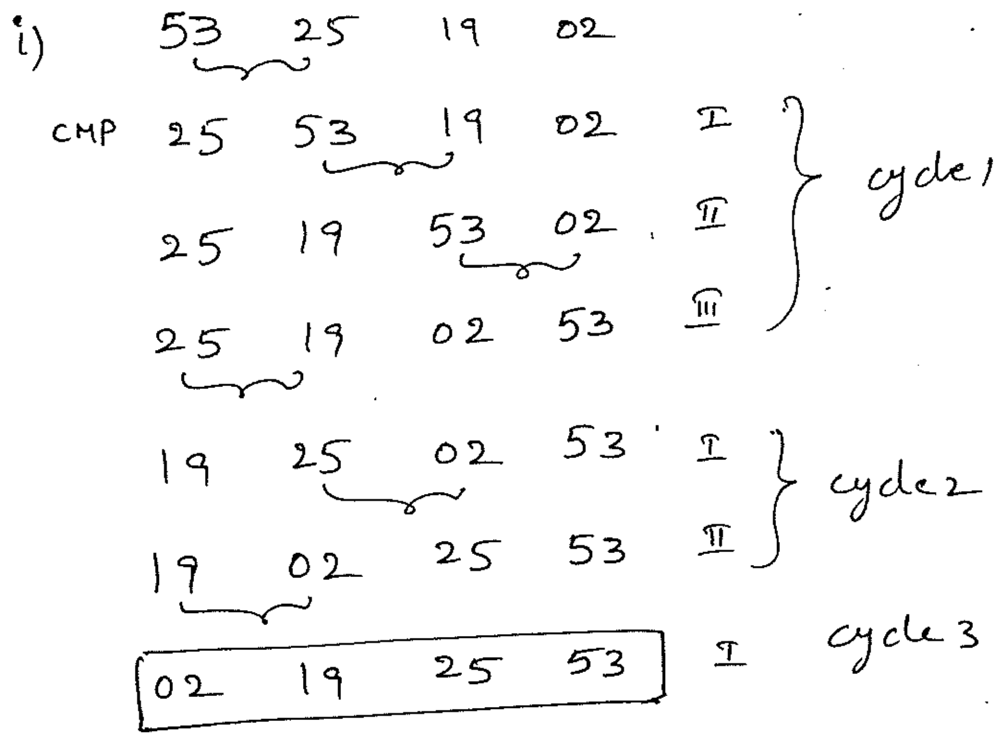
Write an ALP to arrange a given series of hexadecimal bytes in ascending order.

Sol:- Note: In order to arrange the given series of hexadecimal bytes in ascending order, here we use Bubble sorting technique

Ex:- Series of no.s 53, 25 19 02

Here n=4

- After n-1 (i.e 3) iterations we will get first largest number at the end of the series
- After n-2 (i.e 2) iterations we will get second largest number
- After n-3 (i.e 1) iteration we will get third largest number




```
DATA SEGMENT
    LIST DB 53H, 25H, 19H, 02H
    COUNT EQU 04H
DATA ENDS
```

```
ASSUME  CS: CODE, DS: DATA
CODE SEGMENT
        START:  MOV   AX, DATA
                MOV   DS, AX
                XOR   AX, AX
                MOV   DX, COUNT-1
        AGAIN0: MOV   CX, DX
                MOV   SI, OFFSET LIST
        AGAIN1: MOV   AL, [SI]
                CMP   AL, [SI+1]
                JL    PR1
                XCHG  [SI+1], AL
                XCHG  [SI], AL
        PR1:    INC   SI
                LOOP  AGAIN1
                DEC   DX
                JNZ   AGAIN0
                MOV   AH, 4CH
                INT   21H
CODE ENDS
        END START
```

#: Write an ALP to perform a one byte BCD addition (3)

Sol:- Let

```
   92H
+  59H
-------
   EBH
```

Check the lower nibble in the result

i.e B > 9, then add 6 to it.

```
   1011
+  0110
--------
 1)0001
```

After that, check the higher nibble in the result,

i.e E > 9, then add 6 to it. also carry from perivous nibble.

```
      1
   1110
+  0110
--------
 1)0101
```

Result:

cf	MSD	LSD
1	5	1

```
DATA SEGMENT
      OPR1   EQU  92H
      OPR2   EQU  59H
      RES    DB  02H DUP(00)
DATA ENDS
ASSUME CS:CODE, DS:DATA
CODE SEGMENT
      START:  MOV AX, DATA
              MOV DS, AX
```

```
        MOV   BL, OPR1
        XOR   AL, AL
        MOV   AL, OPR2
        ADD   AL, BL
        DAA
        MOV   RES, AL
        JNC   MSBO
        INC   [RES+1].
MSBO:   MOV   AH, 4CH
        INT   21H
CODE ENDS
     END START
```

(4)

#: Write an ALP to find out the number of positive numbers and negative numbers from a given series of signed numbers.

Sol:-

Note: The status of Carry flag, i.e, the MSB of the number will give the information about the sign of the numbers. during the number should ~~shift~~ left ~~carry~~

If CF = 1, the number is negative

If CF = 0, the number is positive

Ex:- i) 2579_H

i.e CF 0011 0101 0111 1001

SHL 0 0110 1010 1111 0010 ← padding

↓

sign is +ve

ii) $A500_H$

i.e CF 1010 0101 0000 0000

SHL 1 0100 1010 0000 0000 ←

↓

Sign is -ve

```
DATA SEGMENT
    LIST DW 2579H, 0A500H, 0C009H, 0159H, 0B900H
    COUNT EQU 05H
DATA ENDS
ASSUME CS: CODE, DS: DATA
CODE SEGMENT
        START: MOV AX, DATA
               MOV DS, AX
```

```
            XOR  BX, BX
            XOR  DX, DX
            XOR  AX, AX
            MOV  CL, COUNT
            MOV  SI, OFFSET LIST
AGAIN:      MOV  AX, [SI]
            SHL  AX, 01H
            JC   NEG
            INC  BX
            JMP  NEXT
NEG:        INC  DX.
NEXT:       ADD  SI, 02H
            DEC  CL.
            JNZ  AGAIN
            MOV  AH, 4CH
            INT  21H
CODE ENDS
        END START
```

te: from the above program,

After the program Execution, the result must be stored in the Register BX, & DX

BX ← No. of positive number

DX ← No. of negative numbers

A program to find out the number of even and odd numbers from a given series of 16-bit hexadecimal numbers.

Sol:-

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
LIST DW 2357H, 0A579H, 0C322H, 0C91EH, 0C000H, 0957H
COUNT EQU 006H.
DATA ENDS
CODE SEGMENT
START:   XOR  BX, BX
         XOR  DX, DX
         MOV  AX, DATA
         MOV  DS, AX
         MOV  CL, COUNT
         MOV  SI, OFFSET LIST
AGAIN:   MOV  AX, [SI]
         ROR  AX, 01
         JC   ODD
         INC  BX
         JMP  NEXT
ODD:     INC  DX
NEXT:    ADD  SI, 02
         DEC  CL
         JNZ  AGAIN
         MOV  AH, 4CH
         INT  21H
CODE ENDS
END START
```

Assembly Language Programs involving String Manipulation Instructions:

Write a program to move a string of data words from offset 2000h to offset 3000h, the length of the string is 0Fh.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
SOURCESTRT EQU 2000H
DESTSTRT EQU 3000H
COUNT EQU 0FH
DATA ENDS
CODE SEGMENT

        MOV AX, DATA
        MOV DS, AX
        MOV ES, AX
        MOV SI, SOURCESTRT
        MOV DI, DESTSTRT
        MOV CX, COUNT
        CLD
REP     MOVSW
        MOV AH, 4CH
        INT 21H
CODE ENDS
END
```

Write a program to ~~perform~~ find out whether a given byte is in the string or not. If it is in the string, find out the relative address of the byte from the starting location of the string.

Sol:- The given string is scanned for the given byte. If it is found in the string, the zero flag is set, else, it is reset. Use of SCASB instruction is quite obvious here. A count should be maintained to find out the relative address of the byte found ~~out~~.

```
ASSUME CS: CODE; DS: DATA
CODE SEGMENT
        MOV AX, DATA
        MOV DS, AX
        MOV ES, AX
        MOV CX, COUNT
        MOV DI, OFFSET STRING
        MOV BL, 00H
        MOV AL, BYTE1
SCAN1 : NOP
        SCASB [DI]
        JZ XXX
        INC BL
        LOOP SCAN1
XXX:    MOV AH, 4CH
        INT 21H
CODE ENDS
DATA SEGMENT
     BYTE1  EQU 25H
     COUNT  EQU 06H
     STRING DB 12H, 13H, 20H, 20H, 25H, 21H
DATA ENDS
END
```

Display a message "The study of microprocessors is interesting", on the CRT screen of a microcomputer.

Sol:- A program to display the string

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
MESSAGE DB ODh, OAh, "STUDY OF MICROPROCESSORS
                        IS INTERESTING", ODh, OAh, "$"
                        ; preparing string of the message
DATA ENDS
CODE SEGMENT
        MOV AX, DATA       ; Initialize DS
        MOV DS, AX
        MOV AH, 09h        ; set function value for display
        MOV DX, OFFSET MESSAGE
        INT 21h            ; point to MESSAGE & Run
        MOV AH, 4Ch        ; The interrupt
        INT 21h            ; Return to DOS
CODE ENDS
END
```

STACK STRUCTURE OF 8086:

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register.

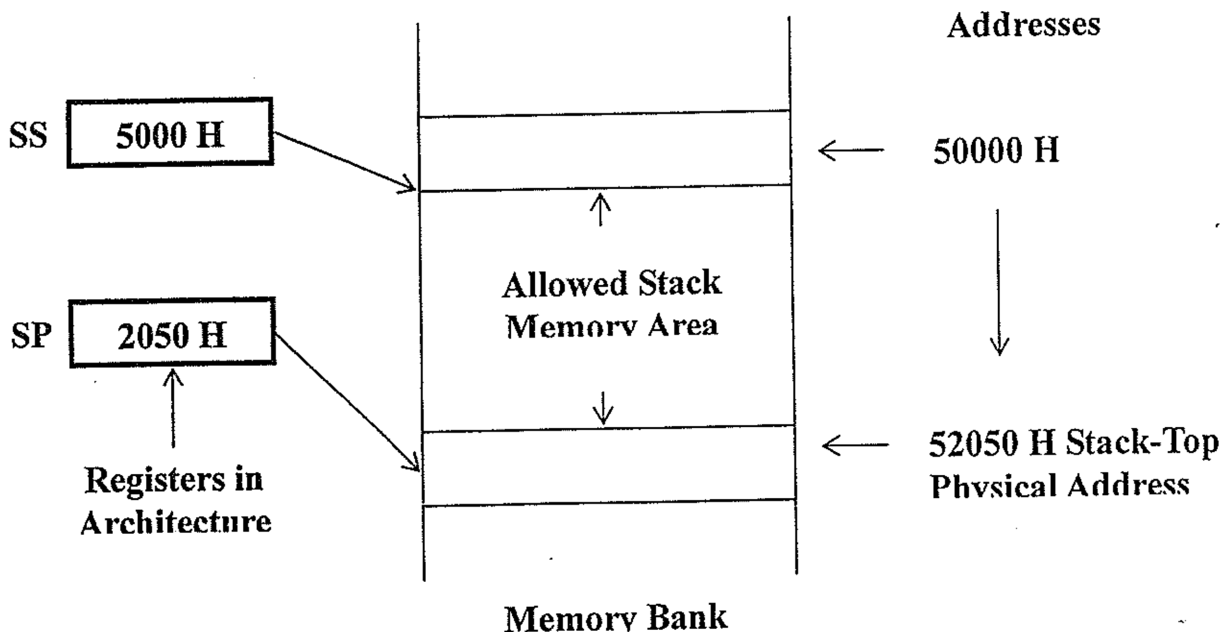
The process of storing the data in the stack is called 'pushing into' the stack and the reverse process of transferring the data back from the stack to the CPU register is known as 'popping off' the stack. The stack is essentially *Last-In-First-Out* (LIFO) data segment. This means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first. The stack pointer is a 16-bit register that contains the offset address of the memory location in the stack segment.

The stack segment, like any other segment, may have a memory block of a maximum of 64 Kbytes locations, and thus may overlap with any other segments. Stack Segment register (SS) contains the base address of the stack segment in the memory.

The Stack Segment register (SS) and Stack pointer register (SP) together address the stack-top as explained below:

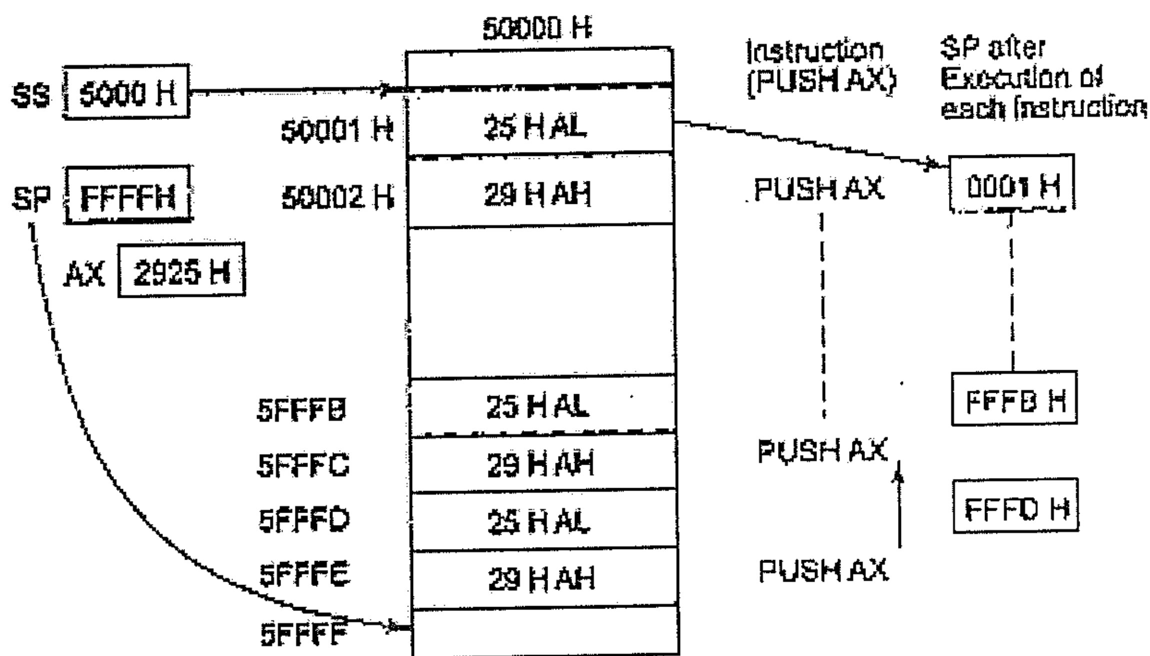
SS ⇒ 5000 H
SP ⇒ 2050 H

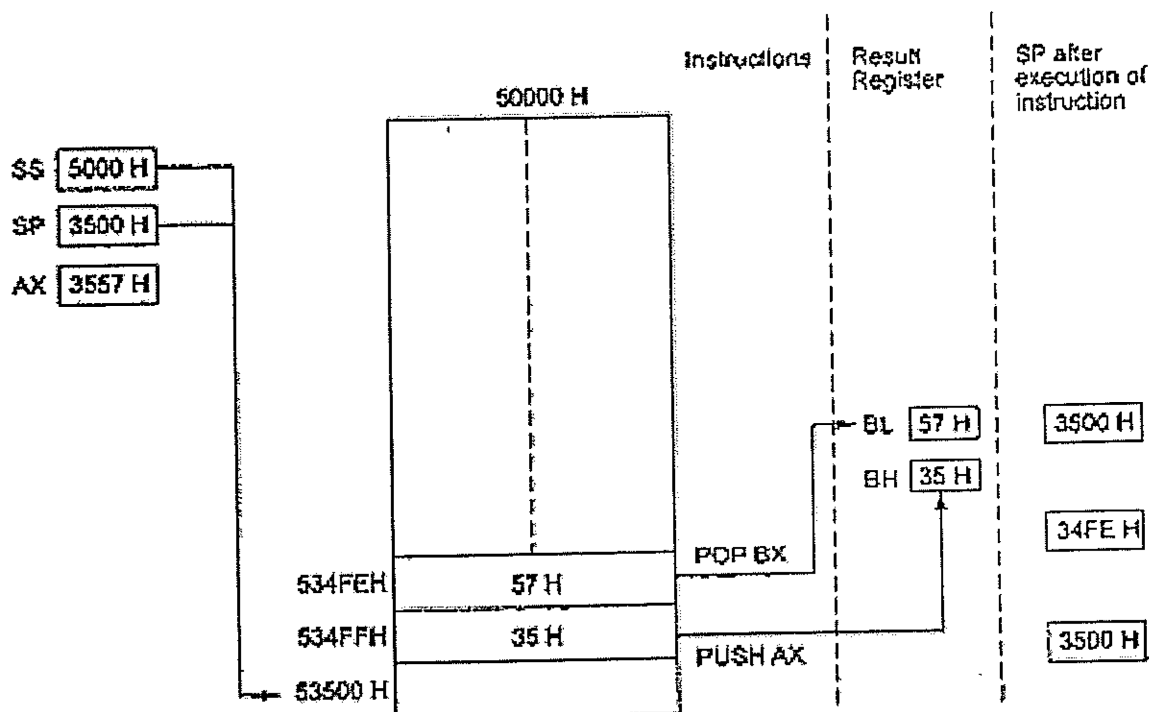
SS	⇒	0101	0000	0000	0000	
10H * SS	⇒	0101	0000	0000	0000	0000
	+					
SP	⇒		0010	0000	0101	0000
Stack-top address		0101	0010	0000	0101	0000
		5	2	0	5	0



If the stack top points to a memory location 52050H, it means that the location 52050H is already occupied with the previously pushed data. The next 16 bit push operation will decrement the stack pointer by two, so that it will point to the new stack-top 5204EH and the decremented contents of SP will be 204EH. This location will now be occupied by the recently pushed data.

Thus for a selected value of SS, the maximum value of SP=FFFFH and the segment can have maximum of 64K locations. If the SP starts with an initial value of FFFFH, it will be decremented by two, whenever a 16-bit data is pushed onto the stack. After successive push operations, when the stack pointer contains 0000H, any attempt to further push the data to the stack will result in stack overflow.





After a procedure is called using the CALL instruction, the IP is incremented to the next instruction. Then the contents of IP, CS and flag register are pushed automatically to the stack. The control is then transferred to the specified address in the CALL instruction i.e. starting address of the procedure. Then the procedure is executed.

Programming for Stack:

Program 1: Write an assembly language program to calculate squares of BCD numbers 0 to 9 and store them from 2000h offset onwards in the current data segment. The numbers and their squares are in the BCD format. Write a subroutine for the calculation of the square of a number.

Ans:

```
ASSUME CS: CODE, DS: DATA, SS: STACK

DATA SEGMENT
        ORG 2000H
        SQUARES DB 0FH DUP (?)
DATA ENDS


STACK SEGMENT
        SDATA DB 100H DUP (?)            ; Reserve 256 bytes for stack
STACK ENDS

CODE SEGMENT
        START: MOV AX,DATA               ;Initialize data segment
               MOV DS,AX
               MOV AX,STACK              ;Initialize stack segment
               MOV SS,AX
               MOV SP,OFFSET SDATA       ; Initialize stack pointer
               MOV CL,0AH                ; Initialize counter for numbers
               MOV SI,OFFSET SQUARES     ; Pointer for array of squares
               MOV AL,00H                ; Start from zero
     NEXTNUM:  CALL SQUARE               ; Calculate square procedure
               MOV BYTE PTR [SI],AH      ; Store square in the array
               INC AL                    ; Go to next number
               INC SI                    ; Increment array pointer
               DEC CL                    ; Decrement counter
               JNZ NEXTNUM               ; Stop if CL=0, else continue
               MOV AH,4CH
               INT 21H
```

CL: 10
AL: 00

```
SQUARE PROC NEAR                    ; Square is a near procedure
            MOV BH,AL
            MOV CH,AL
            XOR AL,AL
     AGAIN: ADD AL,BH               ; Successively add BH to AL
            DAA                     ; Get BCD equivalent
            DEC CH                  ; Decrement multiplier register
            JNZ AGAIN
            MOV AH,AL               ; Store the square of the number
            MOV AL,BH               ; Get back the number
            RET
SQUARE ENDP
CODE ENDS
    END START
```

Program 2: Write an ALP to change a sequence of sixteen 2-byte numbers from ascending to descending order. The numbers are stored in the data segment. Store the new series at addresses starting from 6000H. Use LIFO property of stack.

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
    LIST DW 10H
    ORG 6000H
    RESULT DW 10H
    COUNT EQU 10H
    STACKDATA DB 100H DUP (?)
DATA ENDS

CODE SEGMENT
    START: MOV AX,DATA      ;Initialize data segment
           MOV DS,AX
           MOV SS,AX
           MOV SP,OFFSET LIST
           MOV CL,COUNT
           MOV BX, OFFSET RESULT+COUNT
    NEXT:  POP AX
```

```
            MOV DX,SP
            MOV SP,BX
            PUSH AX
            MOV BX,SP
            MOV SP,DX
            DEC CL
            JNZ NEXT
            MOV AH,4CH
            INT 21H
CODE ENDS
    END START
```

PROCEDURES: -

A procedure is a set of code that can be branched to and returned from in such a way that the code is as if it were inserted at the point from which it is branched to. The branch to procedure is referred to as the *call,* and the corresponding branch back is known as the *return.* The return is always made to the instruction immediately following the call regardless of where the call is located.

Calls, Returns, and Procedure Definitions

The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. The RET instruction simply pops the return address from the stack. The registers used by the procedure need to be stored before their contents are changed, and then restored just before their contents are changed, and then restored just before the procedure is excited.

A CALL may be direct or indirect and, intra-segment or intersegment. If the CALL is intersegment, the return must be intersegment. Intersegment call must push both (IP) and (CS) onto the stack. The return must correspondingly pop two words from the stack. In the case of intra-segment call, only the contents of IP will be saved and retrieved when call and return instructions are used.

Procedures are used in the source code by placing a statement of the form at the beginning of the procedure

```
Procedure name    PROC    Attribute (i.e NEAR or FAR)
```

and by terminating the procedure with a statement

```
Procedure name ENDP
```

The attribute that can be used will be either NEAR or FAR. If the attribute is NEAR, the RET instruction will only pop a word into the IP register, but if it is FAR, it will also pop a word into the CS register.

A procedure may be in:

1. The same code segment as the statement that calls it.
2. A code segment that is different from the one containing the statement that calls it, but in the same source module as the calling statement.
3. A different source module and segment from the calling statement.

In the first case, the attribute could be NEAR provided that all calls are in the same code segment as the procedure. For the latter two cases the attribute must be FAR. If the procedure is given a FAR attribute, then all calls to it must be intersegment calls even if the call is from the same code segment. For the third case, the procedure name must be declared in EXTRN and PUBLIC statements.

Saving and Restoring Registers

When both the calling program and procedure share the same set of registers, it is necessary to save the registers when entering a procedure, and restore them before returning to the calling program.

```
MSK PROC NEAR
        PUSH AX
        PUSH BX
        PUSH CX
        POP CX
        POP BX
        POP AX
        RET
MSK ENDP
```

Procedure Communication

There are two general types of procedures; those operate on the same set of data and those that may process a different set of data each time they are called. If a procedure is in the same source module as the calling program, then the procedure can refer to the variables directly. When the procedure is in a separate source module it can still refer to the source module directly provided that the calling program contains the directive

PUBLIC ARY, COUNT, SUM

EXTRN ARY: WORD, COUNT: WORD, SUM: WORD

Program 3: Write an ALP to convert Binary number into its Decimal and then ASCII equivalent, and then display the number.

```
ASSUME CS: CODE, SS: STACK
STACK SEGMENT
        STAKDATA DB 100H
STACK ENDS

CODE SEGMENT
        START: MOV AL,59H
                CALL BTA
                MOV AH,4CH
                INT 21H

        BTA PROC NEAR
                PUSH DX
                PUSH BX
                PUSH AX
                MOV AH,00H          ; Clear AH
                ~~AAM~~ DAA         ; Convert to BCD
                ADD AX,3030H        ; Convert to ASCII
                MOV BX,AX           ; Save Result
                MOV DL,BH           ; Load first Digit (MSD)
                MOV AH,02H          ; Load function Number
                INT 21H             ; Display first Digit
                MOV DL,BL           ; Load second Digit
                INT 21H             ; Display second Digit
                POP AX
                POP BX
                POP DX
                RET
        BTA ENDP
CODE ENDS
        END START
```

INTERRUPTS AND INTERRUPT SERVICE ROUTINES:

Interrupt and its Need

The microprocessors allow normal program execution to be interrupted in order to carry out a specific task/work. The processor can be interrupted in the following ways

i) by an external signal generated by a peripheral,

ii) by an internal signal generated by a special instruction in the program,

iii) by an internal signal generated due to an exceptional condition which occurs while executing an instruction. (For example, in 8086 processors, divide by zero is an exceptional condition which initiates type 0 interrupt and such an interrupt is also called execution).

In general, the process of interrupting the normal program execution to carry out a specific task/work is referred to as interrupt.

The interrupt is initiated by a signal generated by an external device or by a signal generated internal to the processor. When a microprocessor receives, an interrupt signal it stops executing current normal program, save the status (or content) of various registers (IP, CS and flag registers in case of 8086) in stack and then the processor executes a subroutine/procedure in order to perform the specific task/work requested by the interrupt. The subroutine/procedure that is executed in response to an interrupt is also called Interrupt Service Subroutine. (ISR). At the end of ISR, the stored status of registers in stack is restored to respective registers, and the processor resumes the normal program execution from the point {instruction) where it was interrupted.

The external interrupts are used to implement interrupt driven data transfer scheme. The interrupts generated by special instructions are called software interrupts and they are used to implement system services/calls (or monitor services/calls). The system/monitor services are procedures developed by system designer for various operations and stored in memory. The user can call these services through software interrupts. The interrupts generated by exceptional conditions are used to implement error conditions in the system.

Interrupt Driven Data Transfer Scheme

The interrupts are useful for efficient data transfer between processor and peripheral. When a peripheral is ready for data transfer, it interrupts the processor by sending an appropriate signal. Upon receiving an interrupt signal, the processor suspends the current program execution, save the status in stack and executes an ISR to perform the data transfer between the peripheral and processor. At the end of ISR the processor status is restored from stack and processor resume its normal program execution. This type of data transfer scheme is called interrupt driven data transfer scheme.

The data transfer between the processor and peripheral devices can be implemented either by polling technique or by interrupt method. In polling technique, the processor has to periodically poll or

check the status/readiness of the device and can perform data transfer only when the device is ready. In polling technique, the processor time is wasted, because the processor has to suspend its work and check the status of the device in predefined intervals.

Alternatively, if the device interrupts the processor to initiate a data transfer whenever it is ready then the processor time is effectively utilized because the processor need not suspend its work and check the status of the device in predefined intervals. For an example, consider the data transfer from a keyboard to the processor.

Normally a keyboard has to be checked by the processor once in every 10 milli seconds for a key press. Therefore, once in every 10 milli seconds the processor has to suspend its work and then check the keyboard for a valid key code. Alternatively, the keyboard can interrupt the processor, whenever a key is pressed and a valid key code is generated. In this way, the processor need not waste it's time to check the keyboard once in every 10 milli seconds.

Classification of Interrupts

In general, the interrupts can be classified in the following three ways:

1. Hardware and software interrupts
2. Vectored and Non-Vectored interrupt
3. Maskable and Non-Maskable interrupts

The interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of the processor is called hardware interrupt. The 8086 processor has two interrupt pins INTR and NMI. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086.

The software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if software interrupt instruction is encountered then the processor initiates an interrupt. The 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 0 to 255.

When an interrupt signal is accepted by the processor, if the program control automatically branches to a specific address (called vector address) then the interrupt is called vectored interrupt. The automatic branching to vector address is predefined by the manufacturer of processors. (In these vector addresses the interrupt service subroutines (ISR) are stored). In non-vectored interrupts the interrupting device should supply the address of the ISR to be executed in response to the interrupt. All the 8086 interrupts are vectored interrupts. The vector address for an 8086 interrupt is obtained from a vector table implemented in the first 1kb memory space (00000h to 03FFFh).

The processor has the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to

accept an interrupt is referred to as unmasking or enabling. In 8086 the interrupt flag (IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable a hardware interrupts except NMI.

The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts. The interrupts whose request has to be definitely accepted (or cannot be rejected) by the processor are called non-maskable interrupts. Whenever a request is made by non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISR. In 8086 processors, all the hardware interrupts initiated through INTR pin are maskable by clearing interrupt flag (IF). The interrupt initiated through NMI pin and all software interrupts are non-maskable.

Sources of Interrupts in 8086

An interrupt in 8086 can come from one of the following three sources.

1. One source is from an external signal applied to NMI or INTR input pin of the processor. The interrupts initiated by applying appropriate signals to these input pins are called hardware interrupts.
2. A second source of an interrupt is execution of the interrupt instruction "INT n", where n is the type number. The interrupts initiated by "INT n" instructions are called software interrupts.
3. The third source of an interrupt is from some condition produced in the 8086 by the execution of an instruction. An example of this type of interrupt is divide by zero interrupt. Program execution will be automatically interrupted if you attempt to divide an operand by zero. Such conditional interrupts are also, known as exceptions

Interrupts Cycle of 8086:

The 8086 microprocessor has 256 types of interrupts. INTEL has assigned a type number to each interrupt. The type numbers are in the range of 0 to 255. The 8086 processor has dual facility of initiating these 256 interrupts. The interrupts can be initiated either by executing "INT n" instruction where n is the type number or the interrupt can be initiated by sending an appropriate signal to INTR input pin of the processor.

For the interrupts initiated by software instruction" INT n ", the type number is specified by the instruction itself. When the interrupt is initiated through INTR pin, then the processor runs an interrupt acknowledge cycle to get the type number. (i.e., the interrupting device should supply the type number through D0- D7 lines when the processor requests for the same through interrupt acknowledge cycle).

The kinds of interrupts and their designated types are summarized in figure by illustrating the layout of their pointers within the memory. Only the first five types have explicit definitions; the other types may be used by interrupt instructions or external interrupts. From the figure, it is seen that the type associated with a division error interrupt is 0.

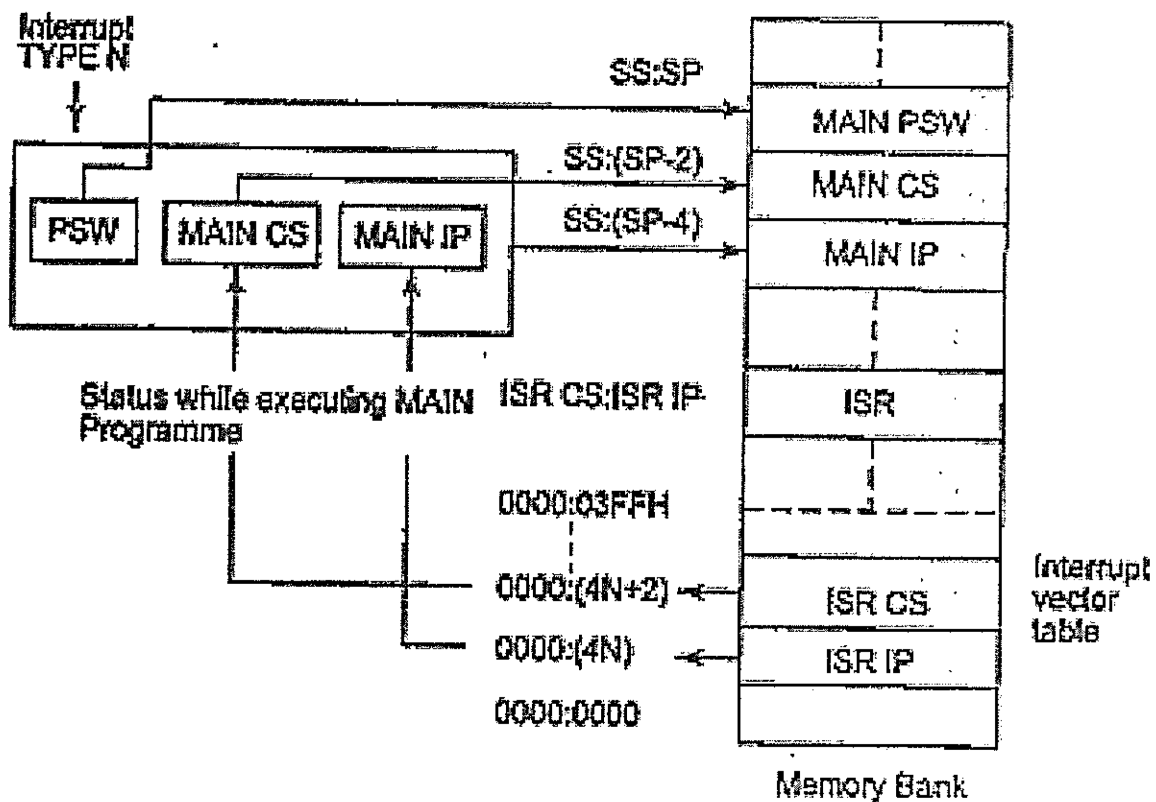
Therefore, if a division by 0 is attempted, the processor will push the current contents of the PSW, CS and IP into the stack, fill the IP and CS registers from the addresses 00000 to 00003, and continue executing at the address indicated by the new contents of IP and CS.

A division error interrupt occurs any time a DIV or IDIV instruction is executed with the quotient exceeding the range, regardless of the IF (Interrupt flag) and TF (Trap flag) status.

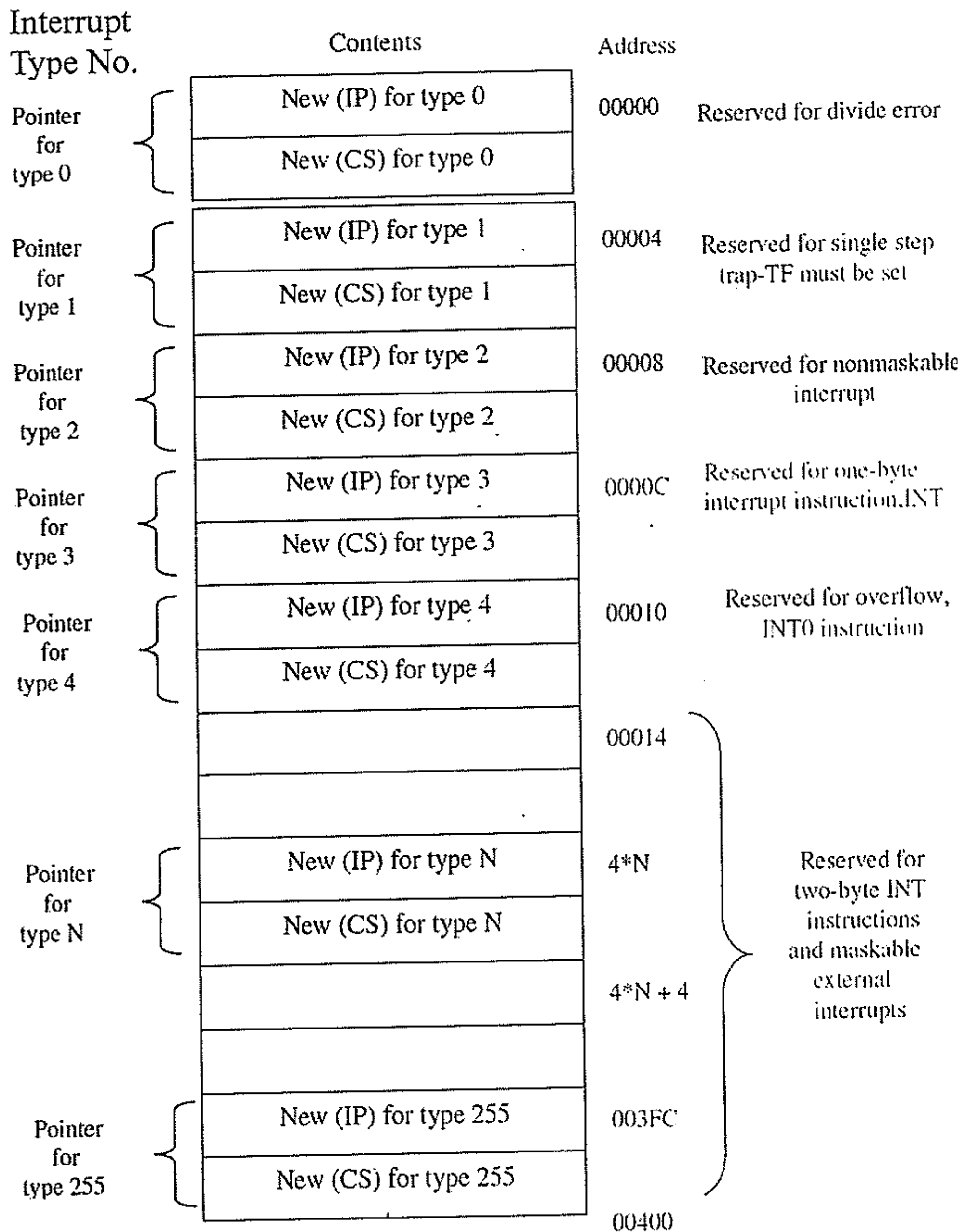
The type 1 interrupt is the single-step interrupt (Trap interrupt) and is the only interrupt controlled by the TF flag. If the TF flag is enabled, then an interrupt will occur at the end of the next instruction that will cause a branch to the location indicated by the contents of 00004H to 00007H.The single step interrupt is used primarily for debugging which gives the programmer a snapshot of his program after each instruction is executed.

The type 2 interrupt is the non-maskable external interrupt. It is the only external interrupt that can occur regardless of the IF flag setting. It is caused by a signal sent to the CPU through the non-maskable interrupt pin.

The remaining interrupt types correspond to interrupts instructions imbedded in the interrupt program or to external interrupts. The interrupt instructions are summarized below and their interrupts are not controlled by the IF flag.



Interrupt Response Sequence



Structure of Interrupt vector table of 8086

Interrupt Programming:

```
ASSUME CS : CODE, DS : DATA
DATA        SEGMET
              |
              |
              |
DATA        ENDS
CODE        SEGMENT
              |
              |
              |
INT           09H
              |
              |
              |
ISR09H      PROC
              |
              |
              |
ISR09H      ENDP
CODE        ENDS
            END
```

After executing INT 09H

After executing ISR09H

ISR09H-interrupt service routine for TYPE 09H

Main programme

INT 0AH

Main programme ends

ISR 0AH (IF set again)

INT 0BH

ISR 0AH ends

ISR 0BH

ISR 0BH ends

Transfer of Control for Nested Interrupts

Program:
Write an ALP to a create a file RESULT and store in it 500H bytes from the memory block starting at 1000:1000, if either an interrupt occurs at INTR pin with Type 0AH or an instruction equivalent to the above interrupt is executed.

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
      FILENAME DB "RESULT", "$"
      MESSAGE DB "FILE WASN'T CREATED SUCCESSFULLY", 0AH, 0DH,"$"
DATA ENDS
CODE SEGMENT
        START:  MOV AX, CODE
                MOV DS, AX               ; Set DS at code for setting IVT
                MOV DX,OFFSET ISR0A      ; Set DX at offset of ISR0A.
                MOV AX,250AH             ;Set IVT using function value 250AH
                INT 21H                  ; in AX under INT 21H
                MOV DX,OFFSET FILENAME ; Set pointer to filename.
                MOV AX,DATA              ; Set the DS at DATA for filename.
                MOV DS,AX
                MOV CX,00H
                MOV AH,3CH               ; Create file with the filename 'RESULT'
                INT 21H
                JNC FURTHER              ; If no carry, create operation is successful
                MOV DX,OFFSET MESSAGE ; else display message
                MOV AH,09H
                INT 21H
                JMP STOP
      FURTHER:  INT 0AH                  ; If the file is created successfully,
      STOP:     MOV AH,4CH               ; write into it and return to DOS prompt
                INT 21H
ISR0A PROC NEAR
        MOV BX,AX          ; Take file handle in BX,
        MOV CX,500H        ; Byte count in CX
        MOV DX,1000H       ; Offset of block in DX
        MOV AX,1000H       ; Segment value of block
        MOV DS,AX          ; in DS
        MOV AH,40H         ; Write in the file and return
        INT 21H
ISR0A ENDP
CODE ENDS
        END START
```

Program:
Write an ALP that gives display 'IRT2 is OK' if a hardware Signal appears on IRQ2 pin and 'IRT3 is OK' if it appears on IRQ3 pin of PC IO channel.

```
ASSUME CS: CODE, DS: DATA
DATA SEGMENT
        MSG1 DB "IRT2 is OK", 0AH, 0DH,"$"
        MSG2 DB "IRT3 is OK", 0AH, 0DH,"$"
DATA ENDS
CODE SEGMENT
        START:  MOV AX,CODE
                MOV DS,AX               ; Set IVT for Type 0AH
                MOV DX, OFFSET ISR1
                MOV AX,250AH            ; IRQ2 is equivalent to Type 0AH
                INT 21H
                MOV DX, OFFSET ISR2     ; Set IVT for Type 0BH
                MOV AX,250BH            ; IRQ3 is equivalent to Type 0BH
                INT 21H
        HERE:   JMP HERE
```

ISR1 display the message

```
ISR1 PROC LOCAL
        MOV AX,DATA
        MOV DS,AX
        MOV DX,OFFSET MSG1      ; Display message MSG1
        MOV AH,09H
        INT 21H
        IRET
ISR1 ENDP
```

ISR2 display the message

```
ISR2 PROC LOCAL
        MOV AX,DATA
        MOV DS,AX
        MOV AX,OFFSET MSG2      ; Display message MSG2
        MOV AH,09H
        INT 21H
        IRET
ISR2 ENDP
CODE ENDS
        END START
```

Assembly Language Program Development Tools:

1. Editor

- An editor is a program which allows you to create a file containing the assembly language statements for your program.

  Example: PC-Write, Wordstar.
- As you type in your program, the editor stores the ASCII codes for the letters and numbers in successive RAM locations.
- When you have typed in your entire program, you then save the file on the hard disk. This file is called source file and the extension is .asm.

2. Assembler

- An assembler program is used to translate the assembly language mnemonics for instructions to corresponding binary codes. When you run the assembler, it reads the source file of your program from the disk where you have saved it after editing.
- On the first pass through the source program, the assembler determines the displacement of named data items, the offset of labels, etc. and puts this information in a symbol table.
- On the second pass through the source program, the assembler produces the binary code for each instruction and inserts the offsets, etc. that it calculated during the first pass.
- The assembler generates 2 files on the floppy disk or hard disk. The first file is called object file (.obj).
- The second file generated by assembler is called the assembler list file and is given extension (.lst).

3. Linker

- A linker is a program used to join several object files into one large object file.
- The linker produces a link file which contains the binary codes for all the combined modules.
- The linker also produces a link map file which contains the address information about the linked files (.exe).

4. Locator

- A locator is a program used to assign the specific address of where the segments of object code are to be loaded into memory.
- A locator program called EXE2BIN comes with the IBM PC Disk Operating System (DOS). EXE2BIN converts a .exe file to a .bin file which has physical addresses.

5. Debugger

- A debugger is a program which allows you to load your object code program into system memory, execute the program and troubleshoot or debug it.
- The debugger allows you to look at the contents of registers and memory locations after your program runs.
- It allows you to change the contents of registers and memory locations and re-run the program.
- Some debuggers allow you to stop execution after each instruction so that you can check or alter after each register contents.
- A debugger also allows you to set a breakpoint at any point in your program. If you insert a breakpoint at any point in your program, the debugger will run the program up to the instruction where you put the breakpoint and then stop the execution.

6. Emulator

- An emulator is a mixture of hardware and software.
- It is used to test and debug the hardware and software of an external system, such as the prototype of a microprocessor based instrument. Part of the hardware of an emulator is a multi wire cable which connects the host system to the system being developed.

8251 – USART
(Universal Synchronous Asynchronous Receiver & Transmitter)

8251 is a USART (Universal Synchronous and Asynchronous Receiver and Transmitter) compatible with Intel's processors. This chip converts the parallel data into a serial stream of bits suitable for serial transmission. It is also able to receive a serial stream of bits and convert it into parallel data bytes to be read by a microprocessor.

Basic Modes of data transmission

a) Simplex

b) Half Duplex

c) Duplex

a) Simplex mode

Data is transmitted only in one direction over a single communication channel. For example, the processor may transmit data for a CRT display unit in this mode.

Ex:- radio station, earthquake sensor

b) Half Duplex mode

In this mode, data transmission may take place in either direction, but at a time data may be transmitted only in one direction. A computer may communicate with a terminal in this mode. It is not possible to transmit data from the computer to the terminal and terminal to computer simultaneously.

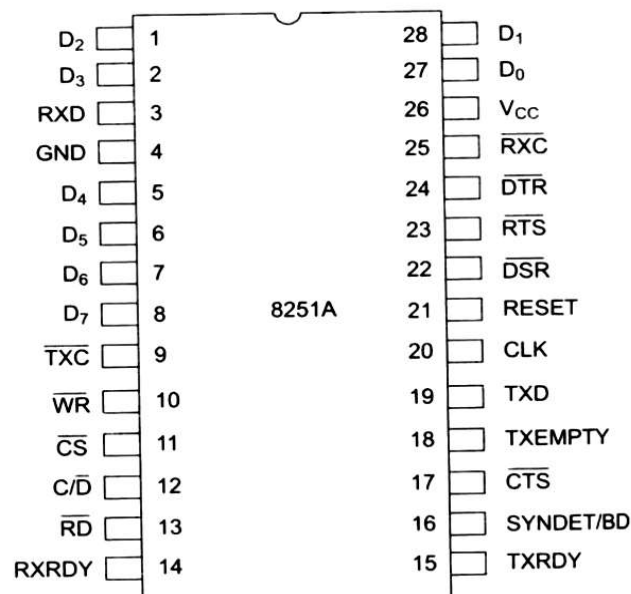
Ex:- walky-talky, push to talk (PTT) devices

c) Duplex Mode

In duplex mode, data may be transferred between two transreceivers in both directions simultaneously.

Ex:- phone conversation.

Signal Description of 8251:



D0 – D7: This is an 8-bit data bus used to read or write status, command word or data from or to the 8251A.

C / D: (Control Word/Data): This input pin, together with RD and WR inputs, informs the 8251A that the word on the data bus is either a data or control word/status information. If this pin is 1, control / status is on the bus, otherwise data is on the bus.

RD: This active-low input to 8251A is used to inform it that the CPU is reading either data or status information from its internal registers. This active-low input to 8251A is used to inform it that the CPU is writing data or control word to 8251A.

WR: This is an active-low chip select input of 825lA. If it is high, no read or write operation can be carried out on 8251. The data bus is tri-stated if this pin is high.

CLK: This input is used to generate internal device timings and is normally connected to clock generator output. This input frequency should be at least 30 times greater than the receiver or transmitter data bit transfer rate.

RESET: A high on this input forces the 8251A into an idle state. The device will remain idle till this input signal again goes low and a new set of control word is written into it. The minimum required reset pulse width is 6 clock states, for the proper reset operation.

TXC (Transmitter Clock Input): This transmitter clock input controls the rate at which the character is to be transmitted. The serial data is shifted out on the successive negative edge of the TXC.

TXD (Transmitted Data Output): This output pin carries serial stream of the transmitted data bits along with other information like start bit, stop bits and parity bit, etc.

RXC **(Receiver Clock Input):** This receiver clock input pin controls the rate at which the character is to be received.

RXD (Receive Data Input): This input pin of 8251A receives a composite stream of the data to be received by 8251 A.

RXRDY (Receiver Ready Output): This output indicates that the 8251A contains a character to be read by the CPU.

TXRDY - Transmitter Ready: This output signal indicates to the CPU that the internal circuit of the transmitter is ready to accept a new character for transmission from the CPU.

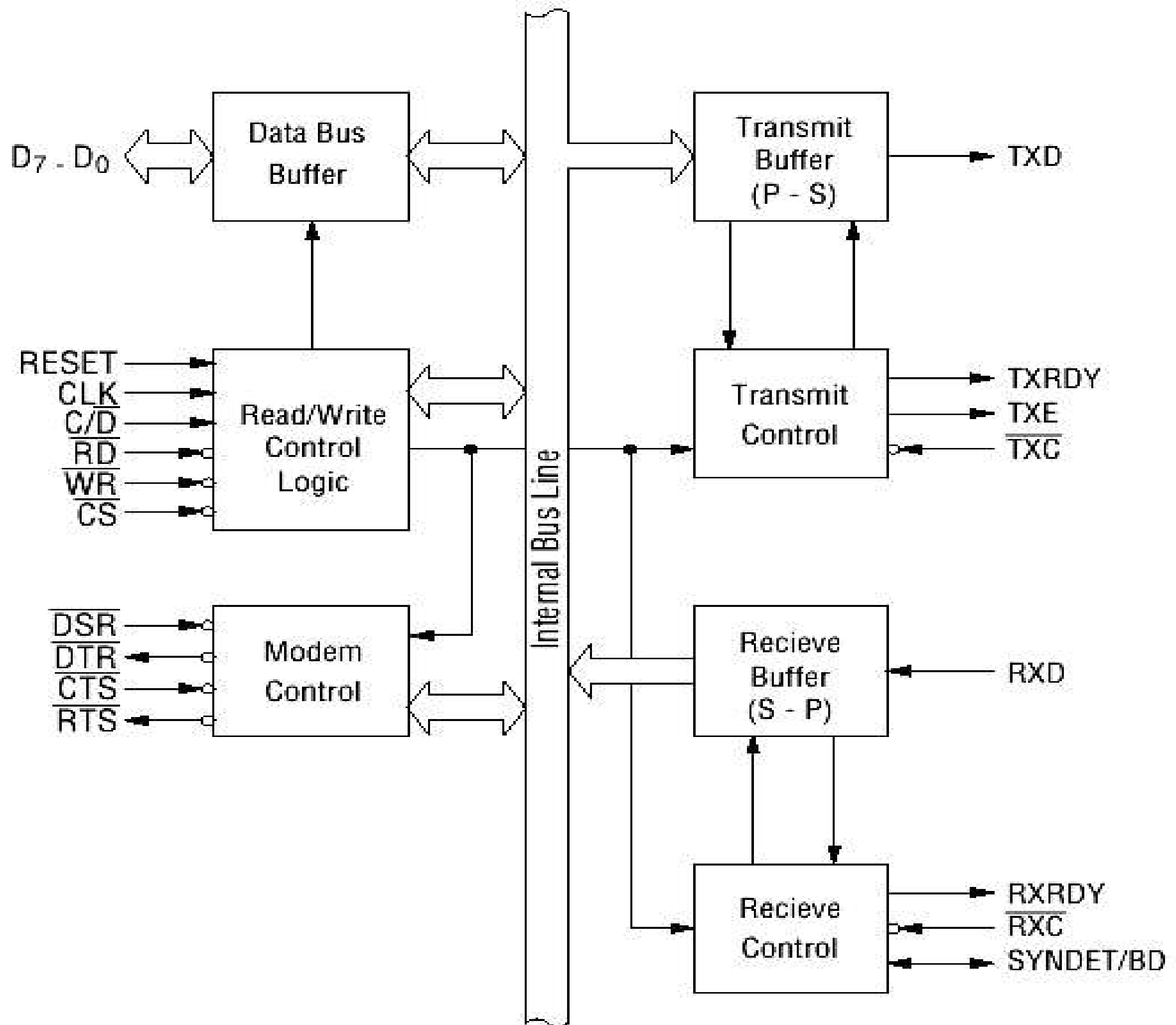
DSR **- Data Set Ready:** This is normally used to check if data set is ready when communicating with a modem.

DTR - **Data Terminal Ready:** This is used to indicate that the device is ready to accept data when the 8251 is communicating with a modem.

RTS - **Request to Send Data:** This signal is used to communicate with a modem.

TXE- Transmitter Empty: The TXE signal can be used to indicate the end of a transmission mode.

Internal Architecture of 8251A:



The data buffer interfaces the internal bus of the circuit with the system bus. The read / write control logic controls the operation of the peripheral depending upon the operations initiated by the CPU. C / decides whether the address on internal data bus is control address / data address. The modem control unit handles the modem handshake signals to coordinate the communication between modem and USART. The transmit control unit transmits the data byte received by the data buffer from the CPU for serial communication. The transmission rate is controlled by the input frequency. Transmit control unit also derives two transmitter status signals namely TXRDY and TXEMPTY which may be used by the CPU for handshaking. The transmit buffer is a parallel to serial converter that receives a parallel byte for conversion into a serial signal for further transmission.

The receive control unit decides the receiver frequency as controlled by the RXC input frequency. The receive control unit generates a receiver ready (RXRDY) signal that may be used by the CPU for handshaking. This unit also detects a break in the data string while the 8251 is in asynchronous mode. In synchronous mode, the 8251 detects SYNC characters using SYNDET/BD pin.

Operating Modes of 8251:

1. Asynchronous mode
2. Synchronous mode

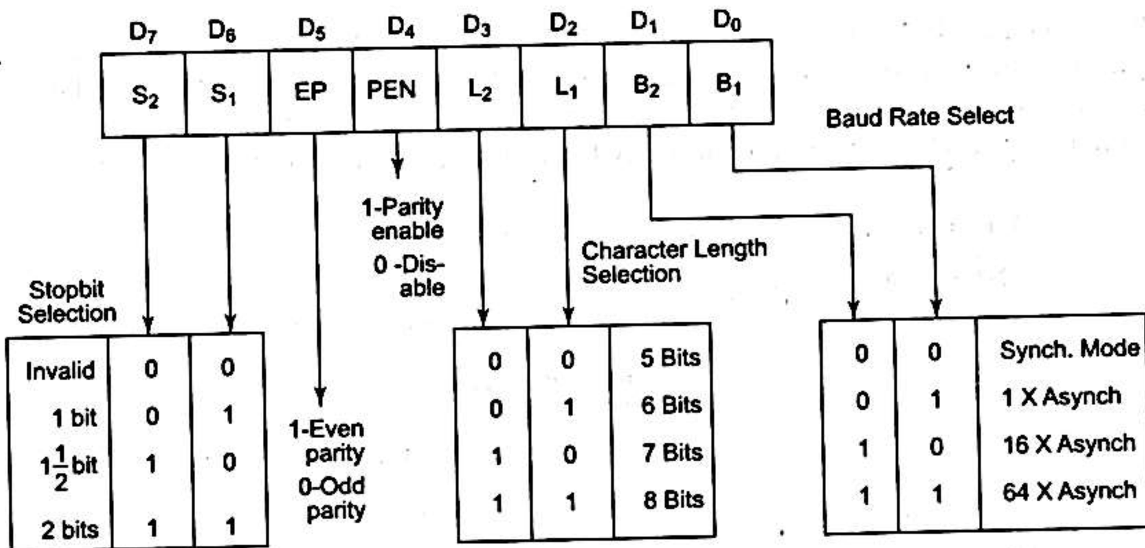
Asynchronous Mode (Transmission)

When a data character is sent to 8251A by the CPU, it adds start bits prior to the serial data bits, followed by optional parity bit and stop bits using the asynchronous mode instruction control word format. This sequence is then transmitted using TXD output pin on the falling edge of TXC.

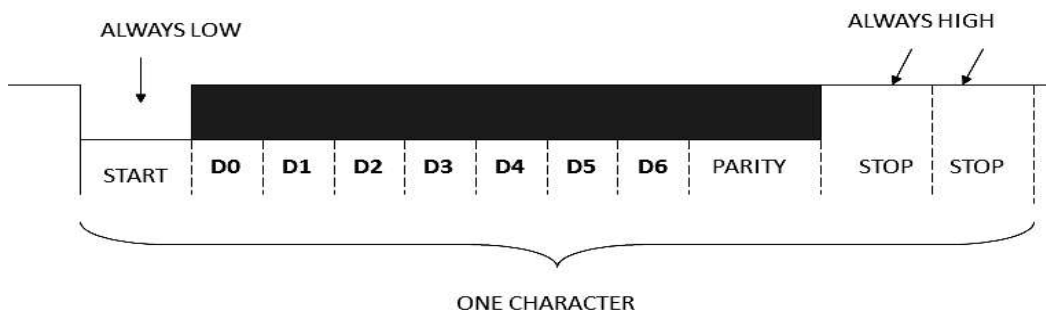
Asynchronous Mode (Receive)

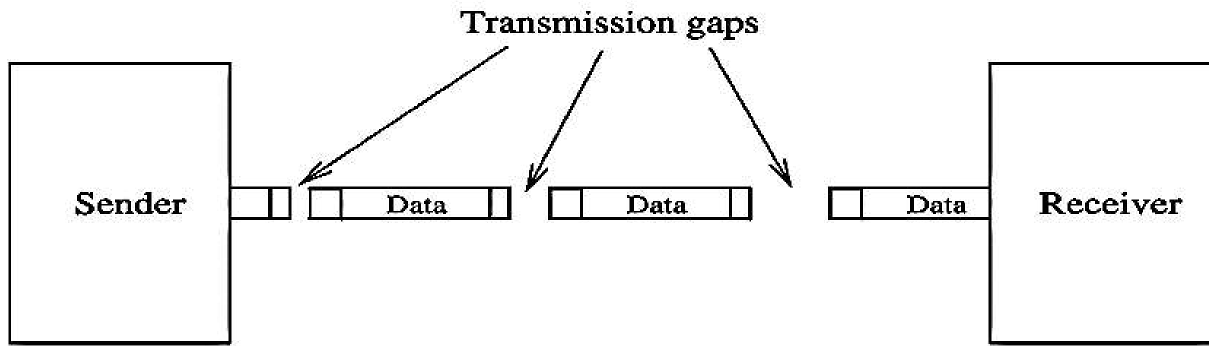
A falling edge on RXD input line marks a start bit. The receiver requires only one stop bit to mark end of the data bit string, regardless of the stop bit programmed at the transmitting end. The 8-bit character is then loaded into the into parallel I/O buffer of 8251. RXRDY pin is raised high to indicate to the CPU that a character is ready for it. If the previous character has not been read by the CPU, the new character replaces it, and the overrun flag is set indicating that the previous character is lost.

Mode instruction format for Asynchronous mode



Asynchronous Mode Transmit and Receive Formats





(a) Asynchronous transmission

Generated by 8251

Transmitter Output

TXD Marking

$D_0 - D_1$

D_x

Start Bit

Parity Bit

Stop Bit

X Programmed by character length

Does not appear on data bus

D_0 D_1

D_x

Receiver Input

RXD

Start Bit

Parity Bit

Stop Bit

Transmission Format

CPU Byte (5-8 bits/char.)

Assembled Data O/P TXD

Start Bit

Data Character

Parity Bit

Stop Bit

Receive Format

Serial Data Input

Start Bit

Data Character

Parity Bit

Stop Bit

CPU Byte (5-8 bits)

Synchronous Mode (Transmission)

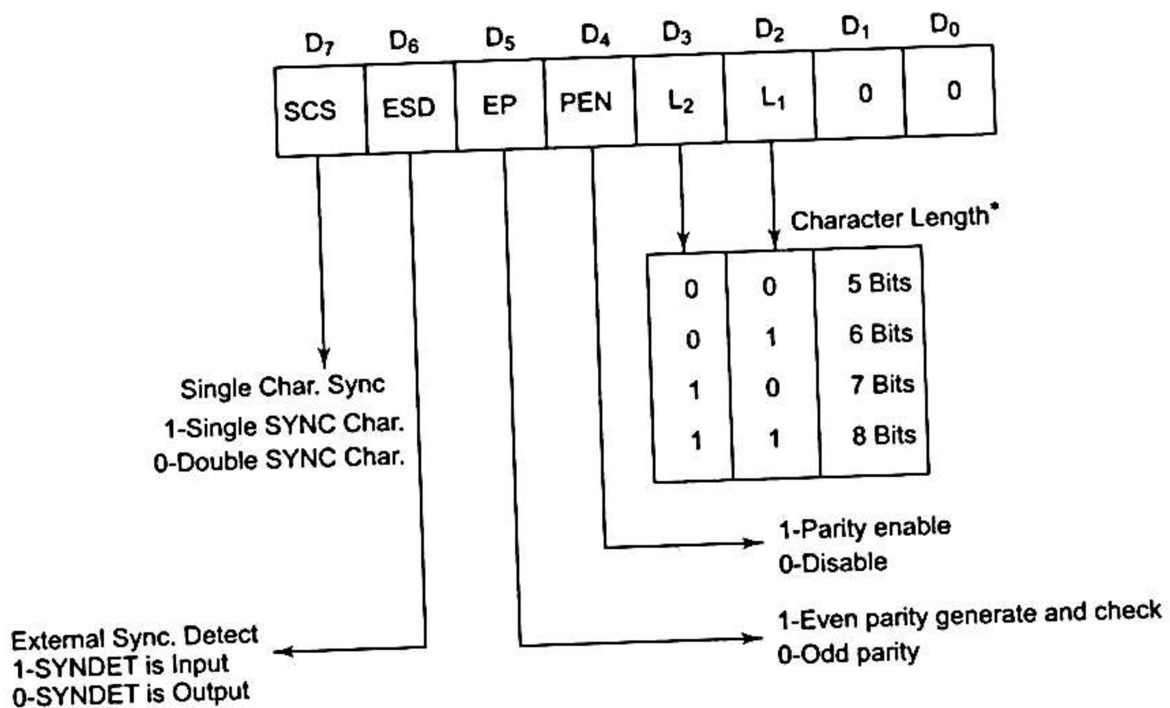
The TXD output is high until the CPU sends a character to 8251 which usually is a SYNC character. When CTS line goes low, the first character is serially transmitted out. Characters are shifted out on the falling edge of TXC. Data is shifted out at the same rate as TXC, over TXD output line. If the CPU buffer becomes empty, the SYNC character or characters are inserted in the data stream over TXD output.

Synchronous Mode (Receiver)

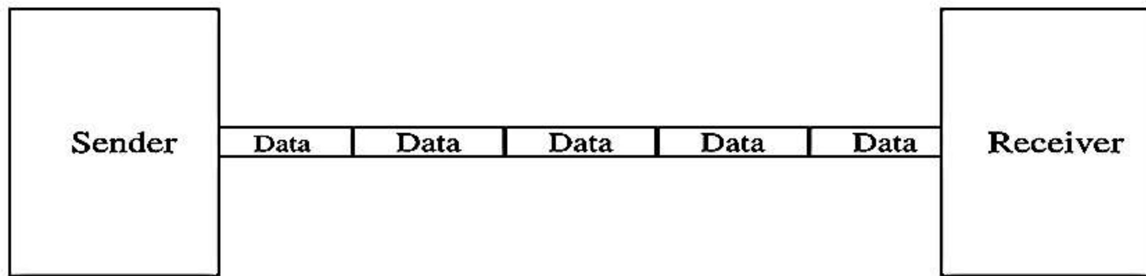
In this mode, the character synchronization can be achieved internally or externally. The data on RXD pin is sampled on rising edge of the RXC. The content of the receiver buffer is compared with the first SYNC character at every edge until it matches. If 8251 is programmed for two SYNC characters, the subsequent received character is also checked. When the characters match, the hunting stops.

The SYNDET pin set high and is reset automatically by a status read operation. In the external SYNC mode, the synchronization is achieved by applying a high level on the SYNDET input pin that forces 8251 out of HUNT mode. The high level can be removed after one RXC cycle. The parity and overrun error both are checked in the same way as in asynchronous mode.

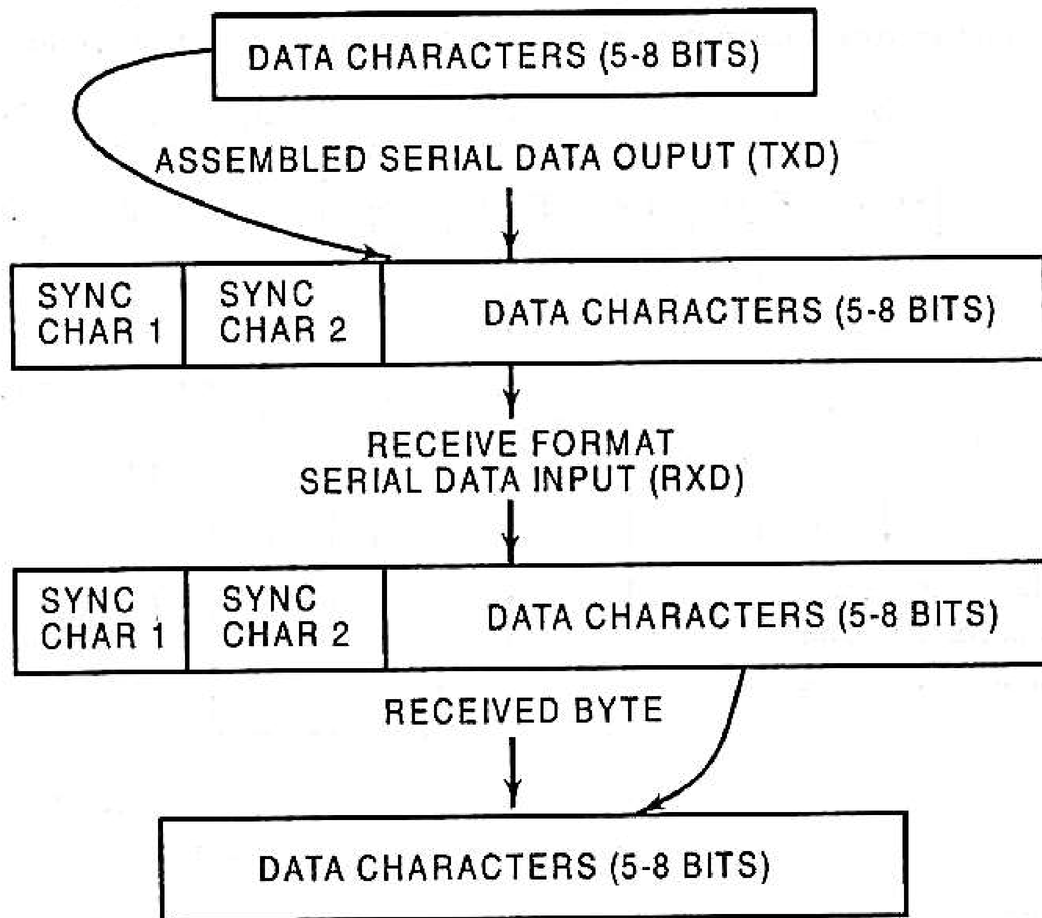
Synchronous Mode Instruction Format



Synchronous mode Transmit and Receive data format



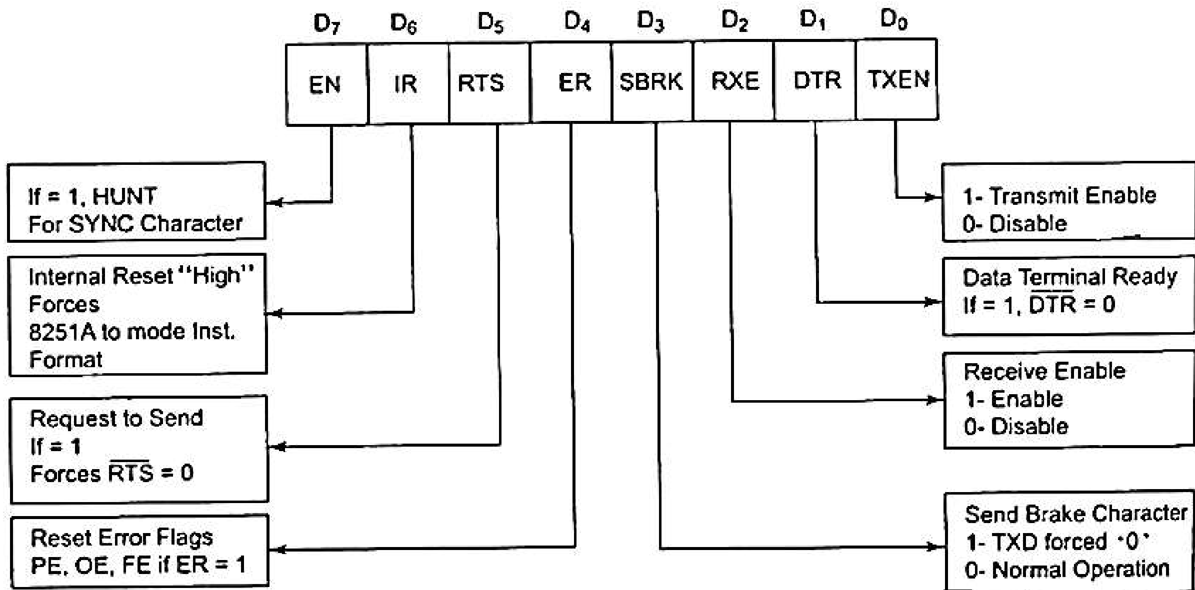
(b) Synchronous transmission



Command Instruction Definition

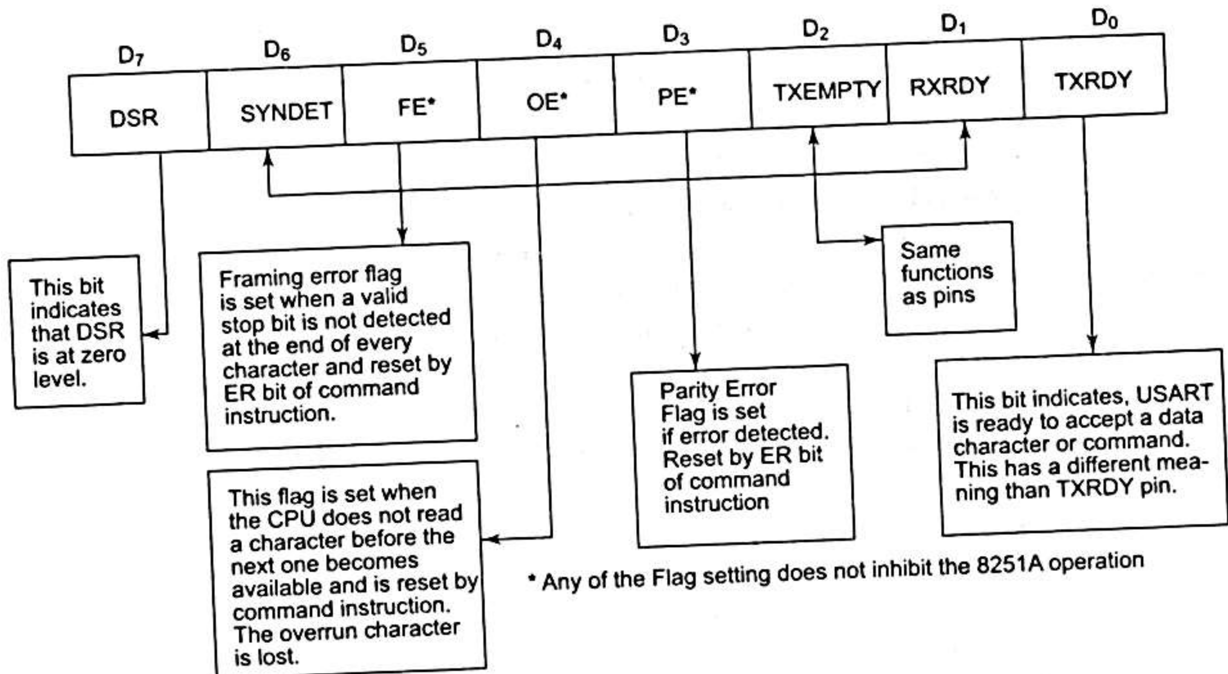
The command instruction controls the actual operations of the selected format like enable transmit/receive, error reset and modem control. A reset operation returns 8251 back to mode instruction format.

Command Instruction format



Status Read Definition

This definition is used by the CPU to read the status of the active 8251 to confirm if any error condition or other conditions like the requirement of processor service has been detected during the operation.



Interfacing 8251 with 8086:

Design the hardware interface circuit for interfacing 8251 with 8086. Set the 8251 in asynchronous mode as a transmitter and receiver with even parity enabled, 2 stop bits, 8-bit character length, frequency 160 kHz and baud rate 10 K.

(a) Write an ALP to transmit 100 bytes of data string starting at location 2000:5000H.

(b) Write an ALP to receive 100 bytes of data string and store it at 3000:4000H.

Solution:

Asynchronous mode control word for transmitting 100 bytes of data:

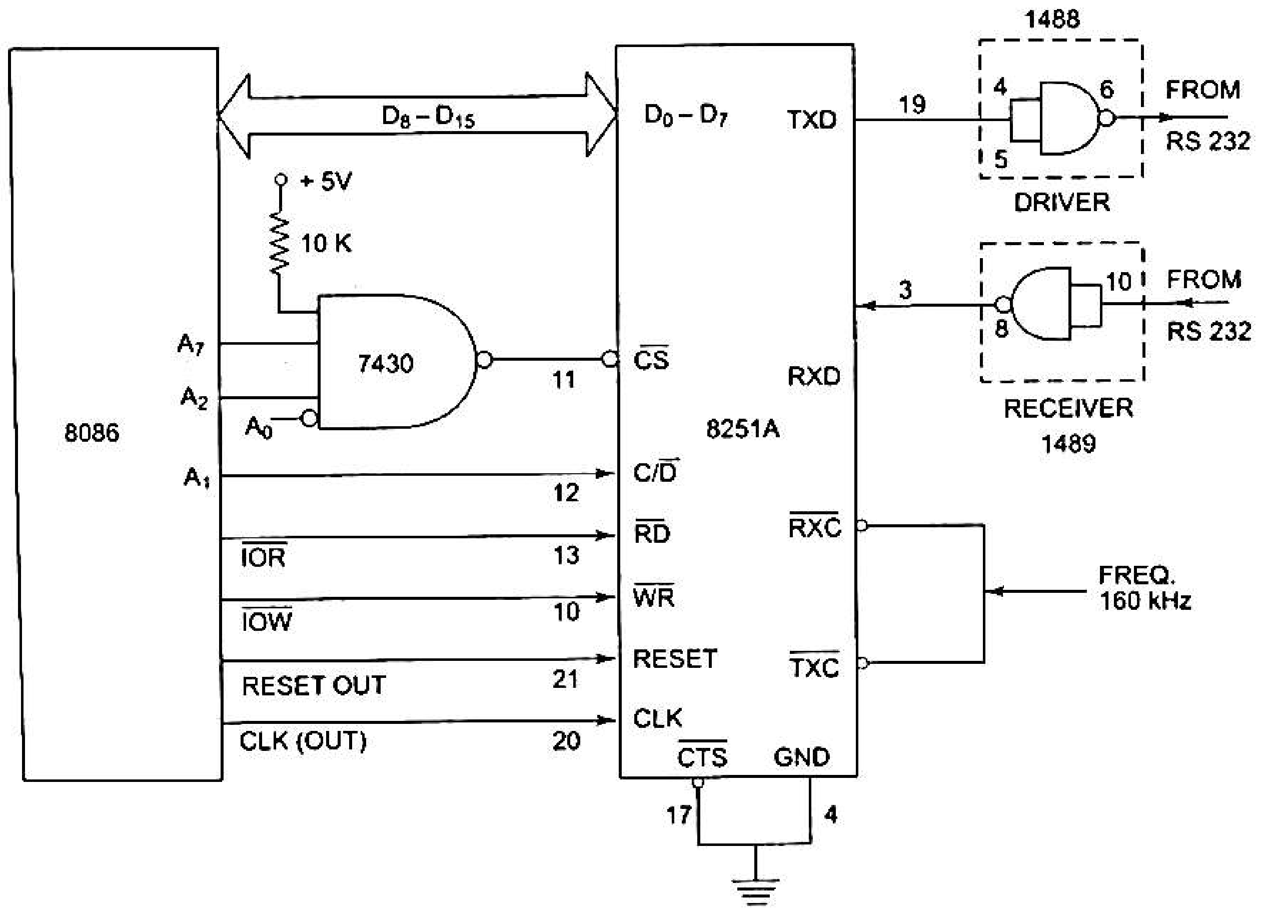
D7	D6	D5	D4	D3	D2	D1	D0
1	1	1	1	1	1	1	0 = FEH
2 Stop bits		Even Parity Enabled		8-bit format		CLK scaled	

a) ALP to initialize 8251 and transmit 100 bytes of data

```
ASSUME CS: CODE
CODE SEGMENT
        START: MOV AX, 2000H
               MOV DS, AX        ; DS points to byte string segment
               MOV SI, 5000H     ; SI points to byte string
               MOV CL, 64H       ; Length of string in CL (hex)
               MOV AL, 0FEH      ; Mode control word to D0 – D7
               OUT 0FEH, AL
               MOV AX, 11H       ; Load command word
               OUT 0FE, AL       ; to transmit enables and error reset
         WAIT: IN AL, 0FEH       ; Read status
               AND AL, 01H       ; Check transmitter enable
               JZ WAIT           ; bit, if zero wait for the transmitter to be ready
               MOV AL, [SI]      ; If ready, first byte of string data is transmitted
               OUT 0FCH, AL
               INC SI            ; Point to next byte
               DEC CL            ; Decrement counter
               JNZ WAIT          ; If CL is not zero, go for next byte
               MOV AH, 4CH
               INT 21H
CODE ENDS
        END START
```

b) An ALP to initialize 8251 and receive 100 bytes of data

```
ASSUME CS: CODE
        CODE SEGMENT
                START: MOV AX, 3000H
                       MOV DS, AX        ; Data segment set to 3000H
                       MOV SI, 4000H     ; Pointer to destination offset
                       MOV CL, 64H       ; Byte count in CL
                       MOV AL, 7EH       ; Only one stop bit for
                       OUT OFEH, AL      ; receiver is set
                       MOV AL, 14H       ; Load command word to enable
                       OUT 0FEH, AL      ; the receiver and disable transmitter
               NXTBT: IN AL, OFEH        ; Read status
                      AND AL, 38H        ; Check FE, OE and PE
                      JZ READY           ; If zero, jump to READY
                      MOV AL, 14H        ; If not zero, clear them
                      OUT OFEH, AL
              READY: IN AL, 0FEH         ; Check RXRDY, if receiver is not ready
                      AND AL, 02H
                      JZ READY           ; wait
                      IN AL, 0FCH        ; If it is ready,
                      MOV [SI], AL       ; receive the character
                      INC SI             ; Increment pointer to next byte
                      DEC CL             ; Decrement counter
                      JNZ NXTBT          ; Repeat, if CL is not zero
                      MOV AH, 4CH
                      INT 21H
        CODE ENDS
                END START
```



Interfacing 8251 with 8086

DMA Controller 8257

The Intel 8257 is a 4-channel **Direct Memory Access** (DMA) controller. It is specifically designed to simplify the transfer of data at high speeds for the Intel® microcomputer systems. Its primary function is to generate, upon a peripheral request, a sequential memory address which will allow the peripheral to read or write data directly to or from memory. Acquisition of the system bus in accomplished via the CPU's hold function.

The 8257 has priority logic that resolves the peripherals requests and issues a composite hold request to the CPU. It maintains the DMA cycle count for each channel and outputs a control signal Jo notify the peripheral that the programmed number of DMA cycles is complete. Other output control signals simplify sectored data transfers. The 8257 represents a significant savings in component count for DMA-based microcomputer systems and greatly simplifies the transfer of data at high speed between peripherals and memories.

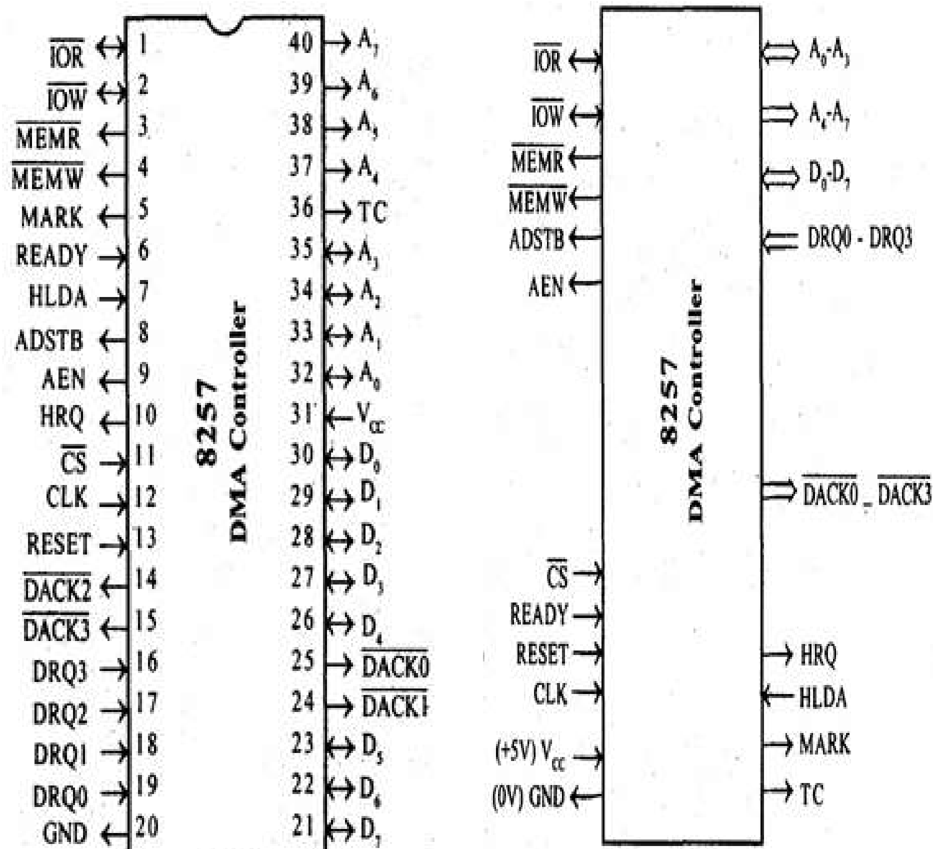
8257 features:

It is a device to transfer the data directly between IO device and memory without the CPU. So it performs a high-speed data transfer between memory and I/O device.

The features of 8257 are:

1. The 8257 has four channels and so it can be used to provide DMA to four I/O devices
2. Each channel can be independently programmable to transfer up to 64kb of data by DMA.
3. Each channel can be independently perform read transfer, write transfer and verify transfer.

Pin diagram 8257:



DRQo-DRQ3:
These are the four individual channel DMA request inputs, used by the peripheral devices for requesting the DMA services. The DRQo has the highest priority while DRQ**3** has the lowest one, if the fixed priority mode is selected.

DACKo-DACK3:
These are the active-low DMA acknowledge output lines which inform the requesting peripheral that the request has been honoured and the bus is relinquished by the CPU. These lines may act as strobe lines for the requesting devices.

Do-D7:

- These are bidirectional, data lines used to interface the system bus with the internal data bus of 8257.
- These lines carry command words to 8257 and status word from 8257, in slave mode, i.e. under the control of CPU. The data over these lines may be transferred in both the directions. When the 8257 is the bus master (master mode, i.e. not under CPU control), it uses Do-D7 lines to send higher byte of the generated address to the latch.
- This address is further latched using ADSTB signal. the address is transferred over Do-D7 during the first clock cycle of the DMA cycle. During the rest of the period, data is available on the data bus.

IOR:
This is an active-low bidirectional tristate input line that acts as an input in the slave mode. In slave mode, this input signal is used by the CPU to read internal registers of 8257.this line acts output in master mode. In master mode, this signal is used to read data from a peripheral during a memory write cycle.

IOW:
This is an active low bidirection tristate line that acts as input in slave mode to load the contents of the data bus to the 8-bit mode register or upper/lower byte of a 16-bit DMA address register or terminal count register. In the master mode, it is a control output that loads the data to a peripheral during DMA memory read cycle (write to peripheral).

CLK:
This is a clock frequency input required to derive basic system timings for the internal operation of 8257.

RESET:
This active-high asynchronous input disables all the DMA channels by clearing the mode register and tristates all the control lines.

Ao-A3:
These are the four least significant address lines. In slave mode, they act as input which select one of the registers to be read or written. In the master mode, they are the four least significant memory address output lines generated by 8257.

CS:
This is an active-low chip select line that enables the read/write operations from/to 8257, in slave mode. In the master mode, it is automatically disabled to prevent the chip from getting selected (by CPU) while performing the DMA operation.

A4-A7:
This is the higher nibble of the lower byte address generated by 8257 during the master mode of DMA operation.

READY:
This is an active-high asynchronous input used to stretch memory read and write cycles of 8257 by inserting wait states. This is used while interfacing slower peripherals..

HRQ:

- The hold request output requests the access of the system bus.
- In the non-cascaded 8257 systems, this is connected with HOLD pin of CPU.
- In the cascade mode, this pin of a slave is connected with a DRQ input line of the master 8257, while that of the master is connected with HOLD input of the CPU.

HLDA:
The CPU drives this input to the DMA controller high, while granting the bus to the device. This pin is connected to the HLDA output of the CPU. This input, if high, indicates to the DMA controller that the bus has been granted to the requesting peripheral by the CPU.

MEMR: This active –low memory read output is used to read data from the addressed memory locations during DMA read cycles.

MEMW: This active-low three state output is used to write data to the addressed memory location during DMA write operation.

ADST: This output from 8257 strobes the higher byte of the memory address generated by the DMA controller into the latches.

AEN: This output is used to disable the system data bus and the control the bus driven by the CPU, this may be used to disable the system address and data bus by using the enable input of the bus drivers to inhibit the non-DMA devices from responding during DMA operations. If the 8257 is I/O mapped, this should be used to disable the other I/O devices, when the DMA controller addresses is on the address bus.

TC:

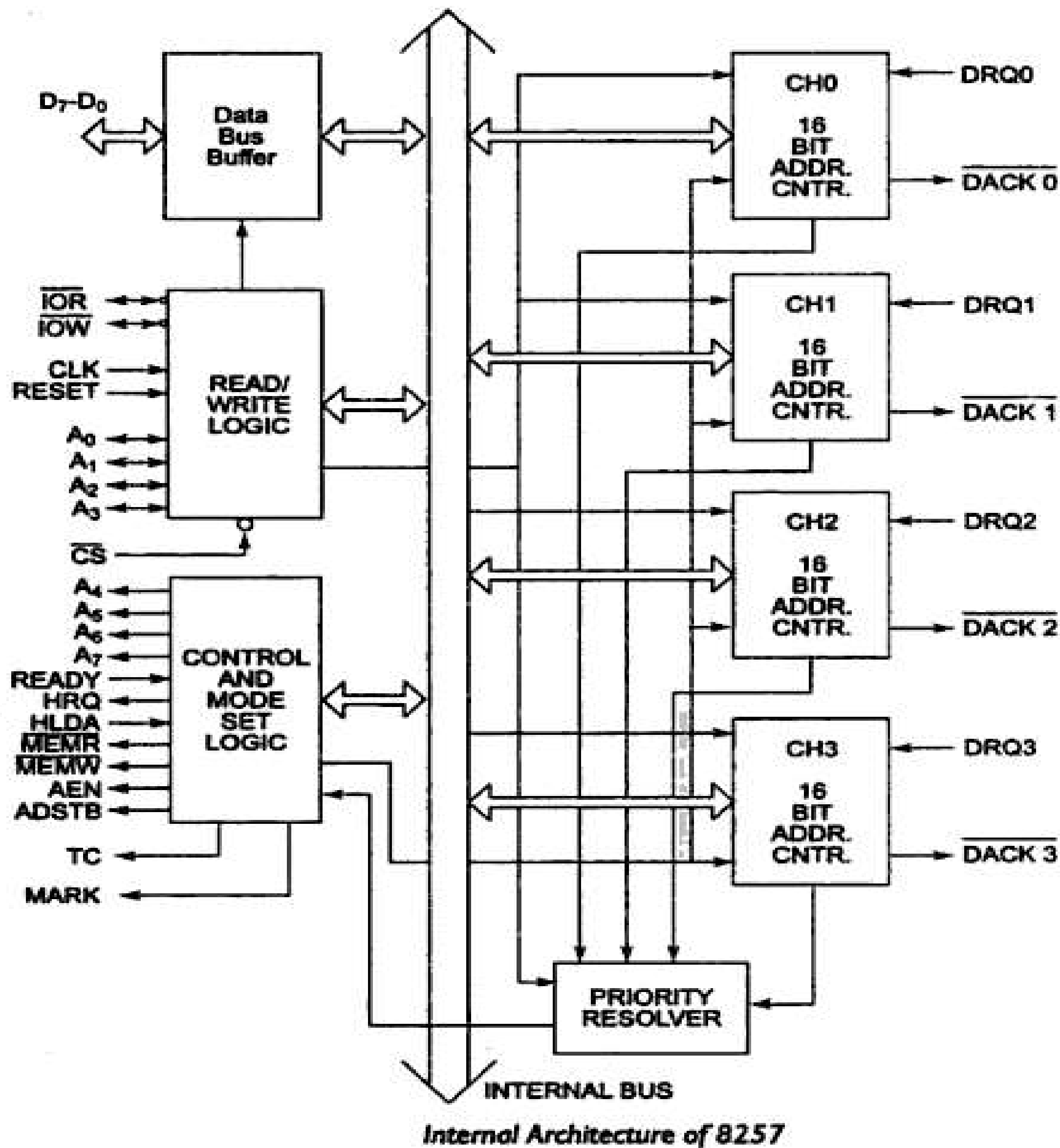
- Terminal count output indicates to the currently selected peripherals that the present DMA cycle is the last for the previously programmed data block.
- If the TC STOP bit in the mode set register is set, the selected channel will be disabled at the end of the DMA cycle.
- The TC pin is activated when the 14-bit content of the terminal count register of the selected channel becomes equal to zero.
- The lower order 14 bits of the terminal count register are to be programmed with a 14-bit equivalent of (n-1), if n is the desired number of DMA cycles.

MARK:
The modulo 128 mark output indicates to the selected peripheral that the current DMA cycle is the 128th cycle since the previous MARK output. The mark will be activated after each 128 cycles or integral multiples of it from the beginning if the data block (the first DMA cycle), if the total number of the required DMA cycles (n) is completely divisible by 128.

Vcc:
This is a +5v supply pin required for operation of the circuit.

GND:
This is a return line for the supply (ground pin of the IC).

Functional Block Diagram of 8257:



Internal Architecture of 8257

The functional blocks of 8257 are **data bus buffer**, **read/write logic**, **control logic**, **priority resolver** and **four numbers of DMA channels**.

Each channel has two programmable 16-bit registers named as **address register** and **count register**.

The 8257 is a programmable. Direct Memory Access (DMA) device which, when coupled with a single Intel® 8212 I/O port device, provides a complete four-channels DMA controller for use in Intel® microcomputer systems. After being initialized by software, the 8257 could transfer a block of data, containing up to 16.384 bytes, between memory and a peripheral device directly, without further intervention required of the CPU. Upon receiving a DMA transfer request from an enabled peripheral, the 8257:

1. **Acquires control of the system bus.**

2. **Acknowledges** that requesting peripheral which is connected to the highest priority channel.

3. **Outputs** the least significant eight bits of the memory address onto system address lines A0-A7. outputs the most significant eight bits of the memory address to the 8212 I/O port via.the data bus (the 8212 places these address bits on lines A8-A15), and

4. **Generates the appropriate memory and I/O read/ write control signals** that cause the peripheral to receive or deposit a data byte directly from or to the addressed location in memory.

8257 Modes:

The 8257 processor works on two modes:

1) Master mode

2) Slave mode

- An active-low input which enables the I/O Read or I/O Write input when the 8257 is being read or programmed in the "slave" mode.
- In the "master" mode. CS is automatically disabled to prevent the chip from selecting itself while performing the DMA function.

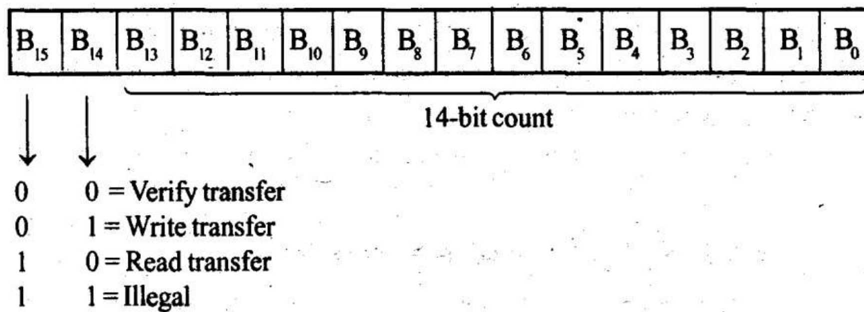
Control word:

- Address register is used to store the starting address of memory location for DMA data transfer.

- The address in the address register is automatically incremented after every operation (read/write/verify transfer).

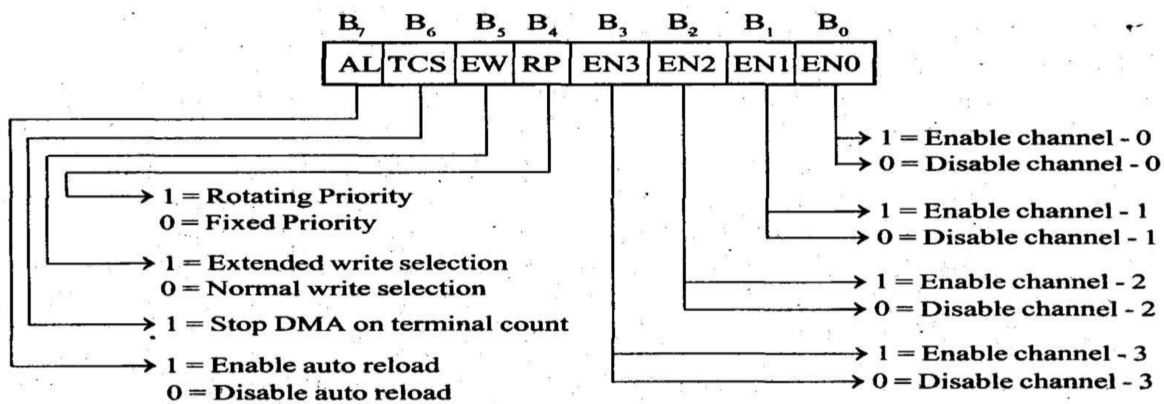
- The count register is used to count the number of byte or word transferred by DMA

The format of count register is



- 14-bits B0-B13 is used to count value and a 2-bits is used for indicate the type of DMA transfer (Read/Write/ transfer).
- In read transfer the data is transferred from memory to I/O device.
- In write transfer the data is transferred from I/O device to memory.
- Verification operations generate the DMA addresses without generating the DMA memory and I/O control signals.
- The 8257 has two eight bit registers called mode set register and status register.

The format of mode set register is



The use of mode set register is:

1. Enable/disable a channel.
2. Fixed/rotating priority
3. Stop DMA on terminal count.
4. Extended/normal write time.
5. Auto reloading of channel-2.

- The bits B0, B1, B2, and B3 of mode set register are used to enable/disable channel -0, 1, 2 and 3 respectively. A one in these bit position will enable a particular channel and a zero will disable it

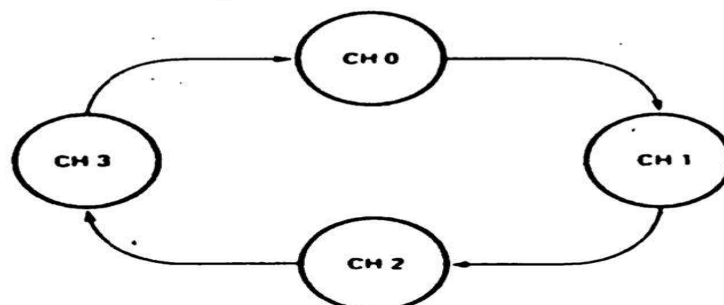
- If the bit B4 is set to one, then the channels will have rotating priority and if it zero then the channels wilt have fixed priority.
    - In rotating priority after servicing a channel its priority is made as lowest.
    - In fixed priority the channel-0 has highest priority and channel-2 has lowest priority.

- If the bit B5 is set to one, then the timing of low write signals (MEMW and IOW) will be extended.

- If the bit B6 is set to one then the DMA operation is stopped at the terminal count.

- The bit B7 is used to select the auto load feature for DMA channel-2.

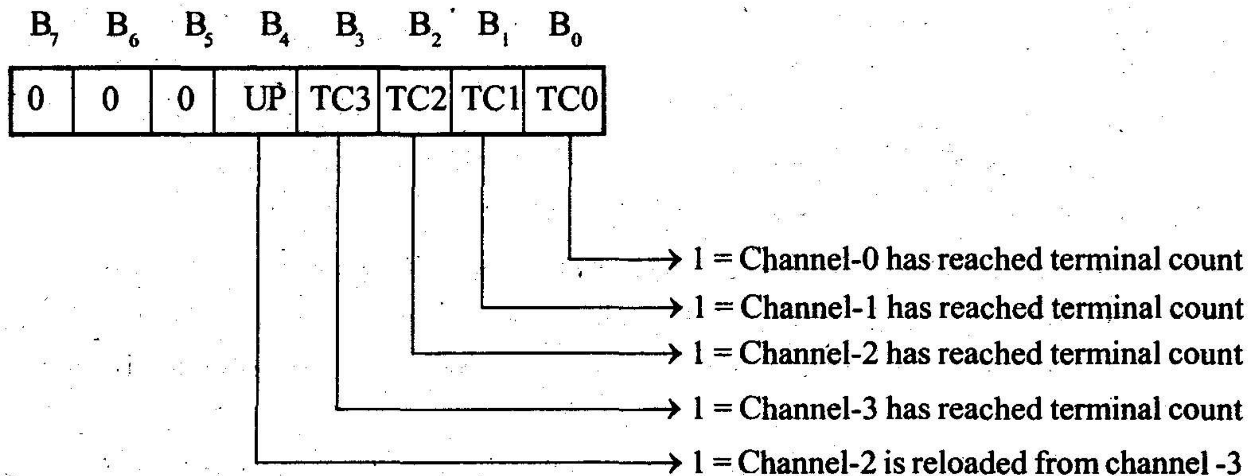
- When bit B7 is set to one, then the content of channel-3 count and address registers are loaded in channel-2 count and address registers respectively whenever the channel-2 reaches terminal count. When this mode is activated the number of channels available for DMA reduces from four to three.



If the ROTATING PRIORITY bit is not set (set to a zero), each DMA channel has a fixed priority. In the fixed priority mode, Channel 0 has the highest priority and Channel 3 has the lowest priority. If the ROTATING PRIORITY bit is set to a one, the priority of each channel changes after each DMA cycle (not each DMA request). Each channel moves up to the next highest priority assignment, while the channel which has just been serviced moves to the lowest priority assignment:

	CHANNEL→ JUST SERVICED	CH-0	CH-1	CH-2	CH-3
Priority → Assignments	Highest ↕ Lowest	CH-1 CH-2 CH-3 CH-0	CH-2 CH-3 CH-0 CH-1	CH-3 CH-0 CH-1 CH-2	CH-0 CH-1 CH-2 CH-3

The format of **status register** of 8257 is shown in fig. below:



- The bit **B0, B1, B2, and B3** of **status register** indicates the **terminal count status** of channel-0, 1,2 and 3 respectively. A one in these bit positions indicates that the particular channel has reached terminal count.
- These status bits are cleared after a read operation by microprocessor.
- The **bit B4** of **status register** is called **update flag** and a **one in this bit position indicates that the channel-2 register has been reloaded from channel-3 registers in the auto load mode of operation.**

The internal addresses of the registers of 8257 are listed in table.

8257 Register Selection

Register	Binary Address								Hexa Address
	Decoder input and enable				Input to address pins of 8257				
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
Channel-0 DMA address register	0	1	1	0	0	0	0	0	60
Channel-0 count register	0	1	1	0	0	0	0	1	61
Channel-1 DMA address register	0	1	1	0	0	0	1	0	62
Channel-1 count register	0	1	1	0	0	0	1	1	63
Channel-2 DMA address register	0	1	1	0	0	1	0	0	64
Channel-2 count register	0	1	1	0	0	1	0	1	65
Channel-3 DMA address register	0	1	1	0	0	1	1	0	66
Channel-3 count register	0	1	1	0	0	1	1	1	67
Mode set register (Write only)	0	1	1	0	1	0	0	0	68
Status register (Read only)	0	1	1	0	1	0	0	0	68

DMA operation state diagram:

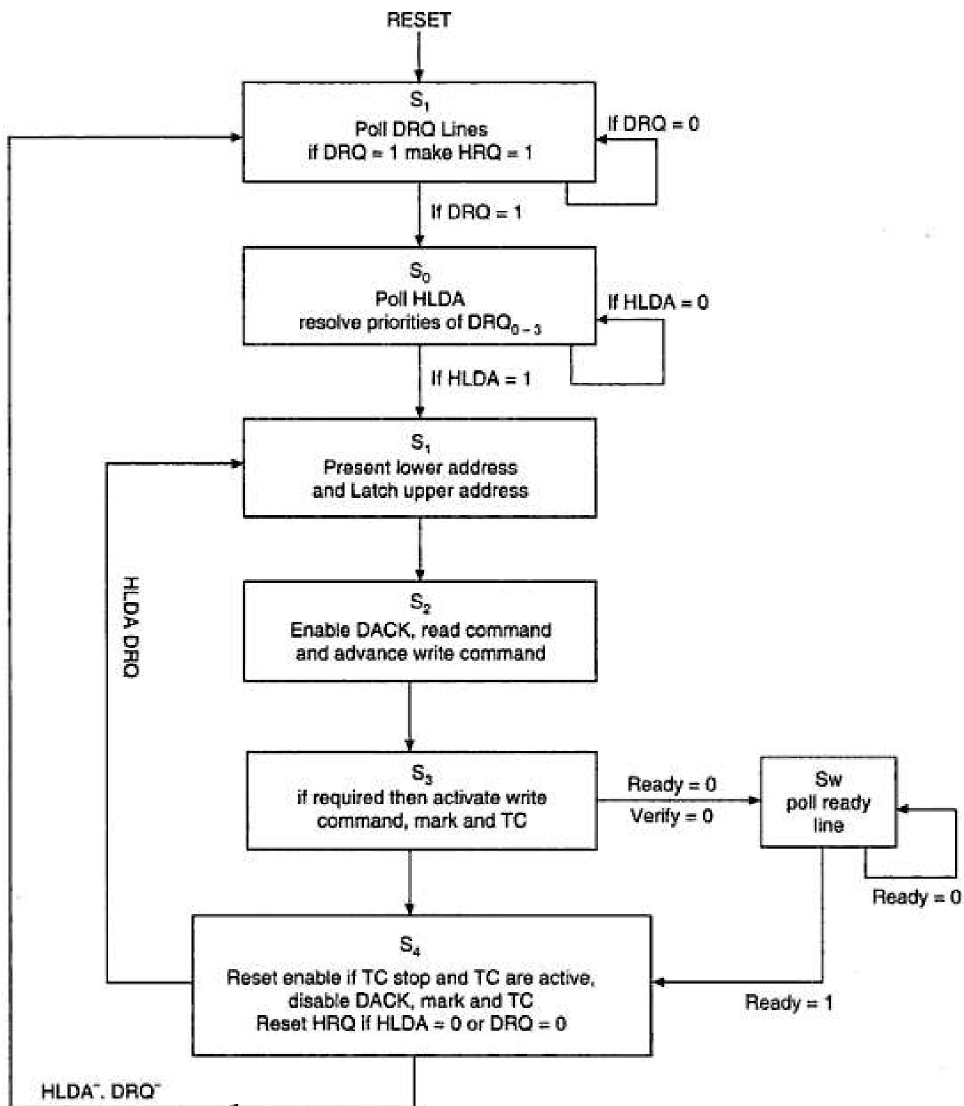


Figure 14.14 State diagram of 8257.

Single Byte Transfers

- A single byte transfer is initiated by the I/O device raising the DRQ line of one channel of the 8257. If the channel is enabled, the 8257 will output a HRQ to the CPU.
- The 8257 now waits until a HLDA is received insuring that the system bus is free for its use. Once HLDA is received the DACK line for the requesting channel is activated (LOW). The DACK line acts as a chip select for the requesting I/O device.
- The 8257 then generates the read and write commands and byte transfer occurs between the selected I/O device and memory. After the transfer is complete, the DACK line is set HIGH and the HRQ line is set LOW to indicate to the CPU that the bus is now free for use.
- DRQ must remain HIGH until DACK is issued to be recognized and must go LOW before S4 of the transfer sequence to prevent another transfer from occurring. (See timing diagram.)

Consecutive Transfers

- If more than one channel requests service simultaneously, the transfer will occur in the same way a burst does. No overhead is incurred by switching from one channel to another.
- In each S4 the DRQ lines are sampled and the highest priority request is recognized during the next transfer.
- A **burst mode** transfer in a lower priority channel will be overridden by a higher priority request. Once the high priority transfer has completed control will return to the lower priority channel if its DRQ is still active.
- No extra cycles are needed to execute this sequence and the HRQ tine remains active until all DRQ lines go LOW.

Control Override

The continuous DMA transfer mode described above can be interrupted by an external device by lowering the HLDA line. After each DMA transfer the 8257 samples the HLDA line to insure that it is still active. If it is not active, the 8257 completes the current transfer, releases the HRQ line (LOW) and returns to the idle state. If DRQ lines are still active the 8257 will raise the HRQ line in the third cycle and proceed normally. (See timing diagram.)

Ready

The 8257 has a Ready input similar to the 8080A and the 8085A. The Ready line is sampled in State 3. If Ready is LOW the 8257 enters a wait state. Ready is sampled during every wait state. When Ready returns HIGH the 8257 proceeds to State 4 to complete the transfer. Ready is used to interface memory or I/O devices that cannot meet the bus set up times required by the 8257.

Speed: The 8257 uses four clock cycles to transfer a byte of data. No cycles are lost in the master to master transfer maximizing **bus** efficiency. A 2MHz clock input will allow the 8257 to transfer at a rate of 500K bytes/second.

Memory Mapped I/O Configurations

The 8257 can be connected to the system bus as a memory device instead of as an I/O device for memory mapped I/O configurations by connecting the system memory control lines to the 8257s I/O control lines and the system I/O control lines to the 8257s memory control lines.

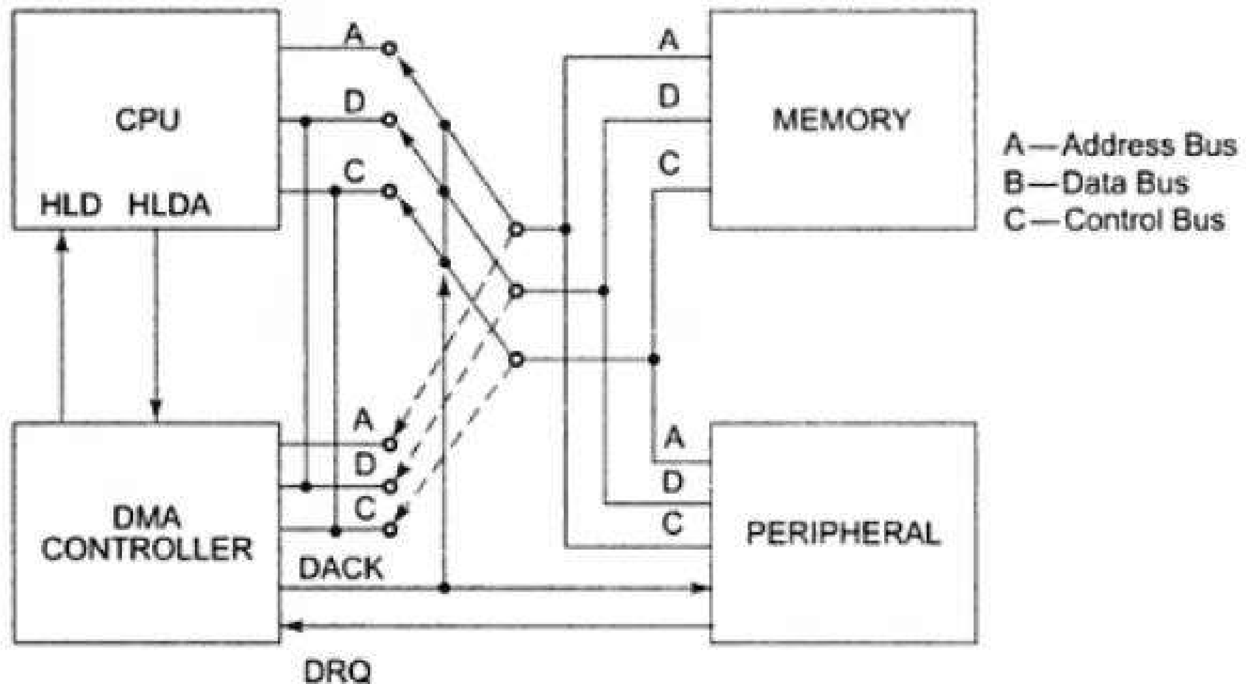
This configuration permits use of the 8080's considerably larger repertoire of memory instructions when reading or loading the 8257s registers. Note that with this connection, the programming of the Read (bit 15) and Write (bit 14) bits in the terminal count register will have a different meaning.

Interfacing 8257 with 8086

- Once a DMA controller is initialized by a CPU property, it is ready to take control of the system bus on a DMA request, either from a peripheral or itself (in case of memory-to-memory transfer).

- The DMA controller sends a HOLD request to the CPU and waits for the CPU to assert the HLDA signal. The CPU relinquishes the control of the bus before asserting the HLDA signal.

A conceptual implementation of the system is shown in Figure



- Once the HLDA signal goes high, the DMA controller activates the DACK signal to the requesting peripheral and gains the control of the system bus.

- The DMA controller is the sole master of the bus, till the DMA operation is over. The CPU remains in the HOLD status (all of its signals are tri-state except HOLD and HLDA), till the DMA controller is the master of the bus.

- In other words, the DMA controller interfacing circuit implements a switching arrangement for the address, data and control busses of the memory and peripheral subsystem from/to the CPU to/from the DMA controller.

UNIT V: INTEL 8051 MICRCONTROLLER

Introduction:

A decade back the process and control operations were totally implemented by the Microprocessors only. But now a day the situation is totally changed and it is occupied by the new devices called Microcontroller. The development is so drastic that we can't find any electronic gadget without the use of a microcontroller. This microcontroller changed the embedded system design so simple and advanced that the embedded market has become one of the most sought after for not only entrepreneurs but for design engineers also.

What is a Microcontroller?

A single chip computer or A CPU with all the peripherals like RAM, ROM, I/O Ports, Timers, ADCs etc... on the same chip. For ex: Motorola's 6811, Intel's 8051, Zilog's Z8 and PIC 16X etc…

MICROPROCESSORS & MICROCONTROLLERS:

Microprocessor:

A CPU built into a single VLSI chip is called a microprocessor. It is a general-purpose device and additional external circuitry are added to make it a microcomputer. The microprocessor contains arithmetic and logic unit (ALU), Instruction decoder and control unit, Instruction register, Program counter (PC), clock circuit (internal or external), reset circuit (internal or external) and registers. But the microprocessor has no on chip I/O Ports, Timers , Memory etc.

For example, Intel 8085 is an 8-bit microprocessor and Intel 8086/8088 a 16-bit microprocessor. The block diagram of the Microprocessor is shown in Fig.1

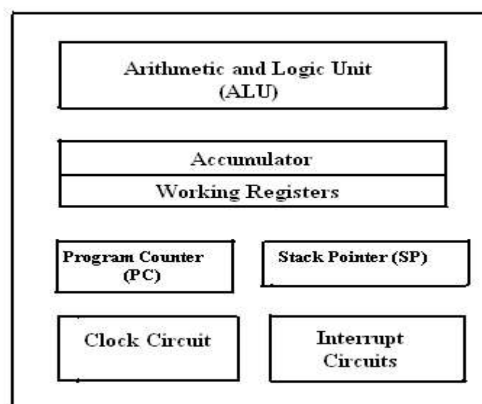


Fig.1 Block diagram of a Microprocessor.

Microcontroller:

A microcontroller is a highly integrated single chip, which consists of on chip CPU (Central Processing Unit), RAM (Random Access Memory), EPROM/PROM/ROM (Erasable Programmable Read Only Memory), I/O (input/output) – serial and parallel, timers, interrupt controller. For example, Intel 8051 is 8-bit microcontroller and Intel 8096 is 16-bit microcontroller. The block diagram of Microcontroller is shown in Fig.2.

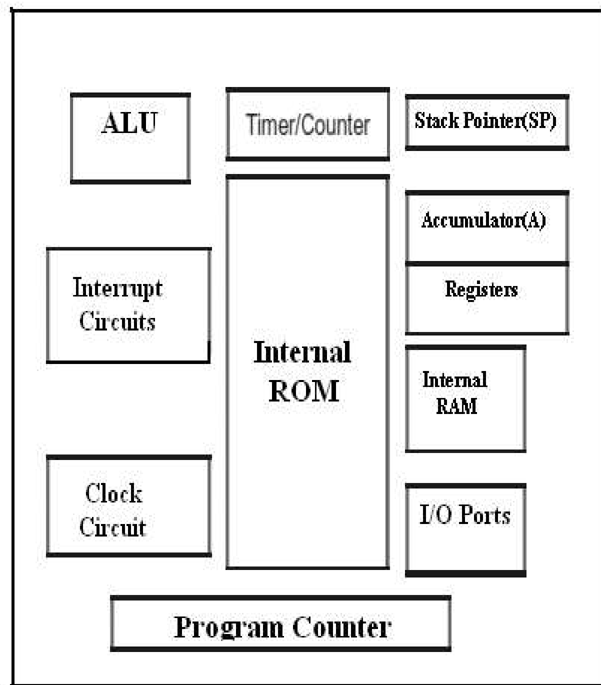


Fig.2.Block Diagram of a Microcontroller

Distinguish between Microprocessor and Microcontroller

S. No	Microprocessor	Microcontroller
1	A microprocessor is a general purpose device which is called a CPU	A microcontroller is a dedicated chip which is also called single chip computer.
2	A microprocessor do not contain onchip I/OPorts, Timers, Memories etc..	A microcontroller includes RAM, ROM, serial and parallel interface, timers, interrupt circuitry (in addition to CPU) in a single chip.
3	Microprocessors are most commonly used as the CPU in microcomputer systems	Microcontrollers are used in small, minimum component designs performing control-oriented applications.
4	Microprocessor instructions are mainly nibble or byte addressable	Microcontroller instructions are both bit addressable as well as byte addressable.
5	Microprocessor instruction sets are mainly intended for catering to large volumes of data.	Microcontrollers have instruction sets catering to the control of inputs and outputs.
6	Microprocessor based system design is complex and expensive	Microcontroller based system design is rather simple and cost effective
7	The Instruction set of microprocessor is complex with large number of instructions.	The instruction set of a Microcontroller is very simple with less number of instructions. For, ex: PIC microcontrollers have only 35 instructions**.**
8	A microprocessor has zero status flag	A microcontroller has no zero flag.

EVOLUTION OF MICROCONTROLLERS:

The first microcontroller TMS1000 was introduced by Texas Instruments in the year 1974. In the year 1976, Motorola designed a Microprocessor chip called 6801 which replaced its earlier chip 6800 with certain add-on chips to make a computer. This paved the way for the new revolution in the history of chip design and gave birth to a new entity called "**Microcontroller".** Later the Intel company produced its first Microcontroller 8048 with a CPU and 1K bytes of EPROM, 64 Bytes of RAM an 8-Bit Timer and 27 I/O pins in 1976. Then followed the most popular controller 8051 in the year 1980 with 4K bytes of ROM,128 Bytes of RAM , a serial port, two 16-bit Timers , and 32 I/O pins. The 8051 family has many additions and improvements over the years and remains a most acclaimed tool for today's circuit designers. INTEL introduced a 16 bit microcontroller 8096 in the year 1982 . Later INTEL introduced 80c196 series of 16-bit Microcontrollers for mainly industrial applications. Microchip, another company has introduced an 8-bit Microcontroller PIC 16C64 in the year 1985.The 32-bit microcontrollers have been developed by IBM and Motorola. MPC 505 is a 32-bit RISC controller of Motorola. The 403 GA is a 32 -bit RISC embedded controller of IBM.

In recent times ARM Company (Advanced RISC machines) has developed and introduced 32 bit controllers for high-end application devices like mobiles, IPods etc...

TYPES OF MICROCONTROLLERS:

Microcontrollers can be classified on the basis of internal bus width, architecture, memory and instruction set as 4-bit, 8-bit, 16-bit and 32-bit microcontrollers.

4-bit Microcontrollers: These 4-bit microcontrollers are small size, minimum pin count and low cost controllers which are widely used for low end applications like LED & LCD display drivers ,portable battery chargers etc.. Their power consumption is also low. The popular 4-bit controllers are Renasa M34501 which is a 20 pin DIP chip with 4kB of ROM, 256 Bytes of RAM,2-Counters and 14 I/O Pins. Similarly ATAM862 series from ATMEL.

8-bit Microcontrollers : These are the most popular and widely used microcontrollers .About 55% of all CPUs sold in the world are 8-bit microcontrollers only.The 8-bit microcontroller has 8-bitinternal bus and the ALU performs all the arithmetic and logical operations on a byte instruction. The well known 8-bit microcontroller is 8051 which was designed by Intel in the year 1980 for the use in embedded systems. Other 8-bit microcontrollers are Intel 8031/8052 and Motorola MC68HC11 and AVR Microcontrollers, Microchip's PIC Microcontrollers 12C5XX ,16C5X and 16C505 etc...

16-bit Microcontrollers: When the microcontroller performs 16-bit arithmetic and logical operations at an instruction, the microcontroller is said to be a 16-bit microcontroller. The internal bus width of 16-bit microcontroller is of 16-bit. These microcontrollers are having increased memory size and speed of operation when compared to 8-bit microcontrollers. These are most suitable for programming in High-level languages like C or $C^{++.}$ They find applications in disk drivers, modems, printers, scanners and servomotor control. Examples of 16-bit microcontrollers are Intel 8096 family and Motorola MC68HC12 and MC68332 families, the performance and computing capability of 16 bit microcontrollers are enhanced with greater precision as compared to the 8-bit microcontrollers.

32-Bit Microcontrollers: These microcontrollers used in high-end applications like Automotive control, Communication networks, Robotics, Cell phones ,GPRS & PDAs etc..

For EX: PIC32, ARM 7, ARM9, SHARP LH79520, ATMEL 32 (AVR) ,

Texas Instrument's –.TMS320F2802x/2803x etc... are some of the popular 32-bit microcontrollers.

COMMERCIAL MICROCONTROLLERS

There are various manufacturers who are supplying various types of microcontrollers suitable for different applications depending on the power consumption and the available features..They are given below in tables. First the various members of INTEL 51 family are given in below table.

INTEL MCS 51 Family

Microcontroller	On chip RAM (Bytes)	On chip program memory	Timers/Counters	Interrupts	Serial ports
8031	128	None	2	5	1
8032	256	None	3	6	1
8051	128	4K ROM	2	5	1
8052	256	8K ROM	3	6	1
8751	128	4K EPROM	2	5	1
8752	256	8K EPROM	3	6	1

MICROCONTROLLER DEVELOPMENT TOOLS:

To develop an assembly language program we need certain program development tools. An assembly language program consists of Mnemonics which are nothing but short abbreviated English instructions given to the controller. The various development tools required for Microcontroller programming are explained below.

1. Editor: An Editor is a program which allows us to create a file containing the assembly language statements for the program. Examples of some editors are PC writes WordStar. As we type the program the editor stores the ACSII codes for the letters and numbers in successive RAM locations. If any typing mistake is done editor will alert us to correct it. If we leave out a program statement an editor will let you move everything down and insert a line. After typing all the program we have to save the program . This we call it as source file. The next step is to process the source file with an assembler.

Ex: Sample. asm

2. Assembler: An Assembler is used to translate the assembly language mnemonics into machine language (i.e binary codes). When you run the assembler it reads the source file of your program from where you have saved it. The assembler generates a file with the extension **.hex**. This file consists of hexadecimal values encoding a sequence of data and their starting offset or absolute address.

3. Compiler: A compiler is a program which converts the high level language program like "C" into binary or machine code. Using high level languages it is easy to manage complex data structures which are often required for data manipulation. Because of its ease, flexibility and debug options now a days the compilers have become very popular in the market. Compilers like Keil, Ride and IAR workbench are very popular.

3. Debugger/Simulator: A debugger is a program which allows executes the program, and troubleshoots or debugs it. The debugger allows looking into the contents of registers and memory locations after the program runs. We can also change the contents of registers and memory locations and rerun the program. Some debuggers allow stopping the program after each instruction so that you can check or alter memory and register contents. This is called single step debug. A debugger also allows setting a breakpoint at any point in the program. If we insert a break point, the debugger will run the program up to the instruction where the breakpoint is put and then stop the execution.

A simulator is a software program which virtually executes the instructions similar to a microcontroller and shows the results. This will help in evaluating the results without committing any errors. By doing so we can detect the possible logic errors

INTEL 8051 MICRCONTROLLER:

The 8051 microcontroller is a very popular 8-bit microcontroller introduced by Intel in the year 1981 and it has become almost the academic standard now a days. The 8051 is based on an 8-bit CISC core with Harvard architecture. Its 8-bit architecture is optimized for control applications with extensive Boolean processing. It is available as a 40-pin DIP chip and works at +5 Volts DC. The salient features of 8051 controller are given below.

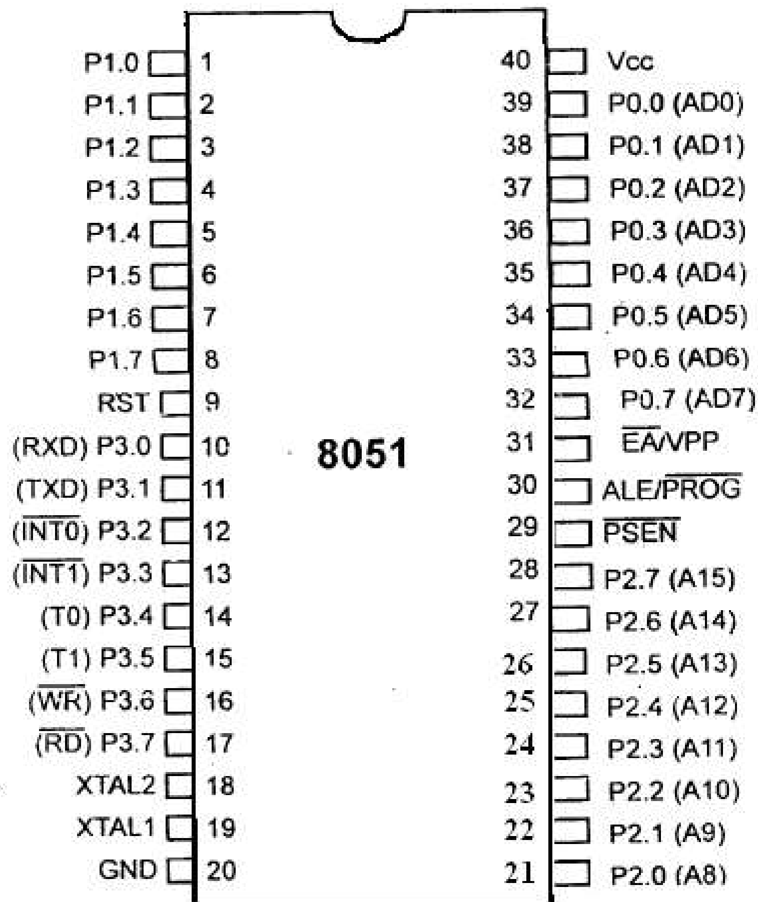
SALIENT FEATURES: The salient features of 8051 Microcontroller are

1. 4 KB on chip program memory (ROM or EPROM)).
2. 128 bytes on chip data memory (RAM).
3. 8-bit data bus
4. 16-bit address bus
5. 32 general purpose registers each of 8 bits
6. Two -16 bit timers T0 and T_1
7. Five Interrupts (3 internal and 2 external).
8. Four Parallel ports each of 8-bits (PORT0, PORT1, PORT2, PORT3) with a total of 32 I/O lines.
9. One 16-bit program counter and One 16-bit DPTR (data pointer)
10. One 8-bit stack pointer
11. One Microsecond instruction cycle with 12 MHz Crystal.
12. One full duplex Serial Communication port.

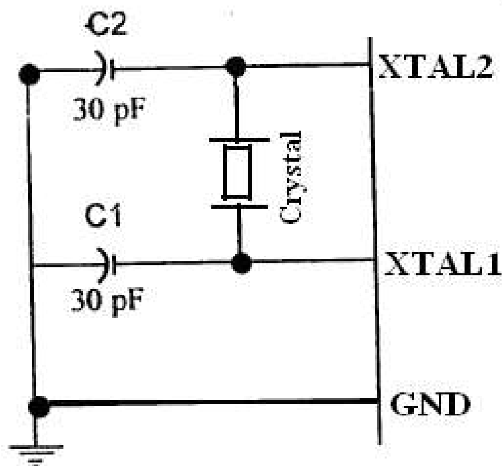
PIN Diagram of 8051 Microcontroller:

The 8051 microcontroller is available as a 40 pin DIP chip and it works at +5 volts DC. Among the 40 pins, a total of 32 pins are allotted for the four parallel ports P0, P1, P2 and P3 i.e each port occupies 8-pins .The remaining pins are VCC, GND, XTAL1, XTAL2, RST, EA, PSEN.

7



XTAL1, XTAL2: These two pins are connected to Quartz crystal oscillator which runs the on-chip oscillator. The quartz crystal oscillator is connected to the two pins along with a capacitor of 30pF as shown in the circuit. If we use a source other than the crystal oscillator, it will be connected to XTAL1 and XTAL2 is left unconnected.



RST: The RESET pin is an input pin and it is an active high pin. When a high pulse is applied to this pin the microcontroller will reset and terminate all activities. Upon reset all the registers except PC will reset to 0000 Value and PC register will reset to 0007 value.

$\overline{\textbf{EA}}$ **(External Access):** This pin is an active low pin. This pin is connected to ground when microcontroller is accessing the program code stored in the external memory and connected to Vcc when it is accessing the program code in the on chip memory. This pin should not be left unconnected.

$\overline{\textbf{PSEN}}$ **(Program Store Enable):** This is an output pin which is active low. When the microcontroller is accessing the program code stored in the external ROM, this pin is connected to the OE (Output Enable) pin of the ROM.

ALE (Address latch enable): This is an output pin, which is active high**.** When connected to external memory , port 0 provides both address and data i.e address and data are multiplexed through port 0 .This ALE pin will de-multiplex the address and data bus .When the pin is High , the AD bus will act as address bus otherwise the AD bus will act as Data bus.

P0.0- P0.7(AD0-AD7) : The port 0 pins multiplexed with Address/data pins .If the microcontroller is accessing external memory these pins will act as address/data pins otherwise they are used for Port 0 pins.

P2.0- P2.7(A8-A15) : The port2 pins are multiplexed with the higher order address pins **.**When the microcontroller is accessing external memory these pins provide the higher order address byte otherwise they act as Port 2 pins.

P1.0- P1.7: These 8-pins are dedicated for Port1 to perform input or output port operations.

P3.0- P3.7: These 8-pins are meant for Port3 operations and also for some control operations like Read, Write, Timer0, Timer1, INT0, INT1, RxD and TxD

ARCHITECTURE & BLOCK DIAGRAM OF 8051 MICROCONTROLLER:

The architecture of the 8051 microcontroller can be understood from the block diagram. It has Harvard architecture with RISC (Reduced Instruction Set Computer) concept. The block diagram of 8051 microcontroller is shown in Fig 3. below1.It consists of an 8-bit ALU, one 8-bit PSW(Program Status Register), A and B registers , one 16-bit Program counter , one 16-bit Data pointer register(DPTR),128 bytes of RAM and 4kB of ROM and four parallel I/O ports each of 8-bit width. 8051 has 8-bit ALU which can perform all the 8-bit arithmetic and logical operations in one machine cycle. The ALU is associated with two registers A & B

Block Diagram

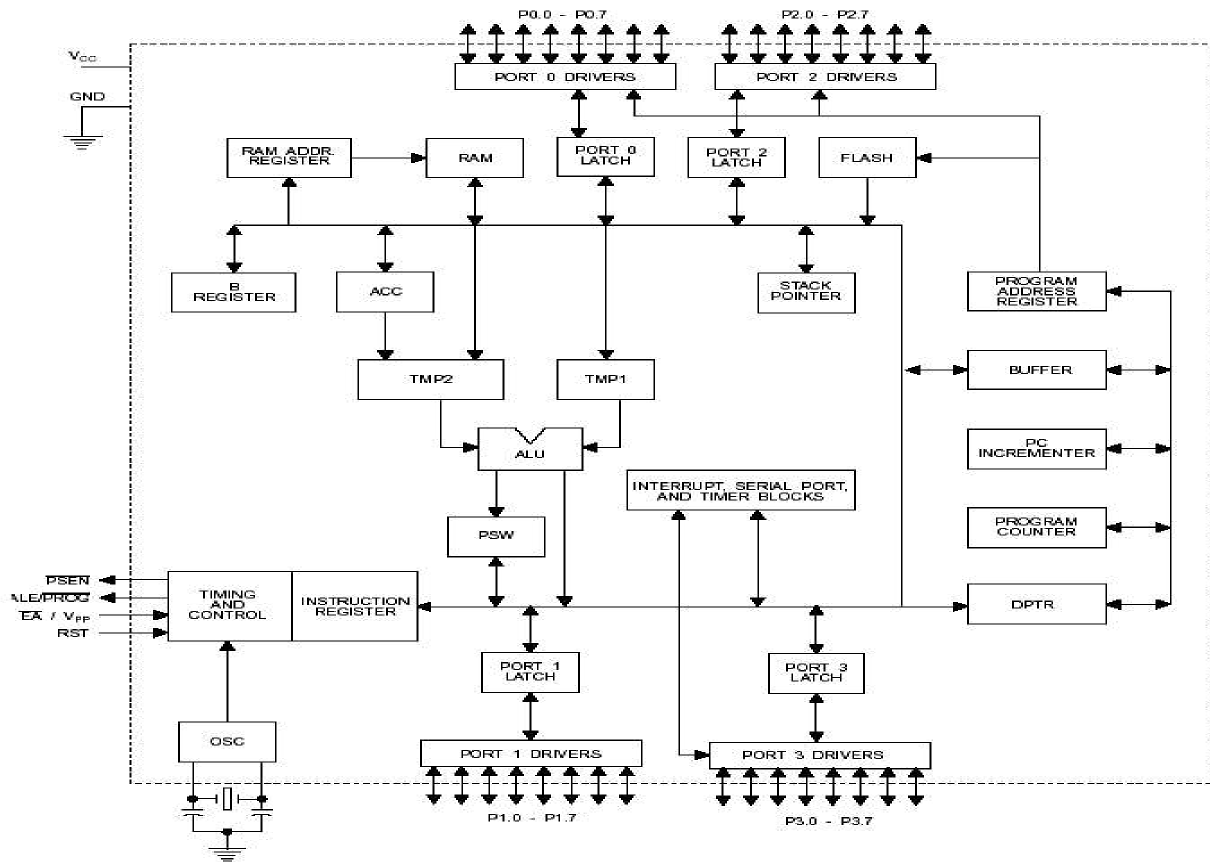


Fig.3. Internal Architecture of 8051 Microcontroller

A and B Registers : The A and B registers are special function registers which hold the results of many arithmetic and logical operations of 8051.The A register is also called the **Accumulator** and as its name suggests, is used as a general register to accumulate the results of a large number of instructions. By default it is used for all mathematical operations and also data transfer operations between CPU and any external memory.

The B register is mainly used for multiplication and division operations along with A register.

MUL AB : DIV AB.

It has no other function other than as a location where data may be stored.

The R registers: The "R" registers are a set of eight registers that are named R0, R1, etc. up to R7. These registers are used as auxiliary registers in many operations. The "R" registers are also used to temporarily store values.

10

Program Counter (PC): 8051 has a 16-bit program counter .The program counter always points to the address of the next instruction to be executed. After execution of one instruction the program counter is incremented to point to the address of the next instruction to be executed. It is the contents of the PC that are placed on the address bus to find and fetch the desired instruction. Since the PC is 16-bit width, 8051 can access program addresses from 0000H to FFFFH, a total of 6kB of code.

Stack Pointer Register (SP): It is an 8-bit register which stores the address of the stack top. i.e the Stack Pointer is used to indicate where the next value to be removed from the stack should be taken from. When a value is pushed onto the stack, the 8051 first increments the value of SP and then stores the value at the resulting memory location. Similarly when a value is popped off the stack, the 8051 returns the value from the memory location indicated by SP, and then decrements the value of SP. Since the SP is only 8-bit wide it is incremented or decremented by two. SP is modified directly by the 8051 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered.

STACK in 8051 Microcontroller: The stack is a part of RAM used by the CPU to store information temporarily. This information may be either data or an address .The CPU needs this storage area as there are only limited number of registers. The register used to access the stack is called the Stack pointer which is an 8-bit register. So, it can take values of 00 to FF H. When the 8051 is powered up, the SP register contains the value 07.i.e the RAM location value 08 is the first location being used for the stack by the 8051 controller

There are two important instructions to handle this stack. One is the PUSH and the other is the POP. The loading of data from CPU registers to the stack is done by PUSH and the loading of the contents of the stack back into a CPU register is done by POP.

EX: MOV R6, #35 H
    MOV R1, #21 H
    PUSH 6
    PUSH 1

In the above instructions the contents of the Registers R6 and R1 are moved to stack and they occupy the 08 and 09 locations of the stack. Now the contents of the SP are incremented by two and it is 0A Similarly POP 3 instruction pops the contents of stack into R3 register. Now the contents of the SP is decremented by 1

In 8051 the RAM locations 08 to 1F (24 bytes) can be used for the Stack. In any program if we need more than 24 bytes of stack, we can change the SP point to RAM locations 30-7F H. This can be done with the instruction MOV SP, # XX.

11

Data Pointer Register (DPTR): It is a 16-bit register which is the only user-accessible. DPTR, as the name suggests, is used to point to data. It is used by a number of commands which allow the 8051 to access external memory. When the 8051 accesses external memory it will access external memory at the address indicated by DPTR. This DPTR can also be used as two 8-registers DPH and DPL.

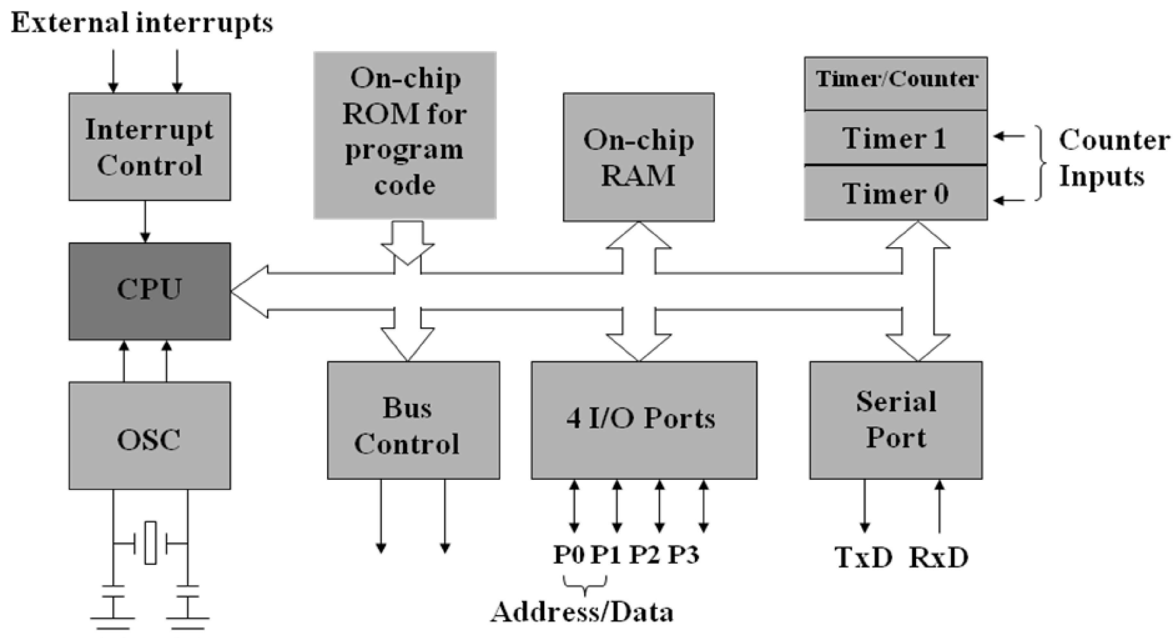
Program Status Register (PSW): The 8051 has a 8-bit PSW register which is also known as Flag register. In the 8-bit register only 6-bits are used by 8051.The two unused bits are user definable bits. In the 6-bits four of them are conditional flags .They are Carry –CY, Auxiliary Carry-AC, Parity-P, and Overflow-OV .These flag bits indicate some conditions that resulted after an instruction was executed.

CY	PSW.7	Carry Flag
AC	PSW.6	Auxiliary Carry Flag
FO	PSW.5	Flag 0 available for general purpose .
RS1	PSW.4	Register Bank select bit 1
RS0	PSW.3	Register bank select bit 0
OV	PSW.2	Overflow flag
---	PSW.1	User definable flag
P	PSW.0	Parity flag .set/cleared by hardware.

The bits PSW3 and PSW4 are denoted as RS0 and RS1 and these bits are used to select the bank registers of the RAM location. The meaning of various bits of PSW register is shown below.

The selection 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



Memory Organization:

The 8051 microcontroller has 128 bytes of Internal RAM and 4kB of on chip ROM .The RAM is also known as Data memory and the ROM is known as program memory. The program memory is also known as Code memory .This Code memory holds the actual 8051 program that is to be executed. In 8051 this memory is limited to 64K .Code memory may be found on-chip, as ROM or EPROM. It may also be stored completely off-chip in an external ROM or, more commonly, an external EPROM. The 8051 has only 128 bytes of Internal RAM but it supports 64kB of external RAM. As the name suggests, external RAM is any random access memory which is off-chip. Since the memory is off-chip it is not as flexible in terms of accessing, and is also slower. For example, to increment an Internal RAM location by 1, it requires only 1 instruction and 1 instruction cycle but to increment a 1-byte value stored in External RAM requires 4 instructions and 7 instruction cycles. So, here the external memory is 7 times slower.

Internal RAM OF 8051:

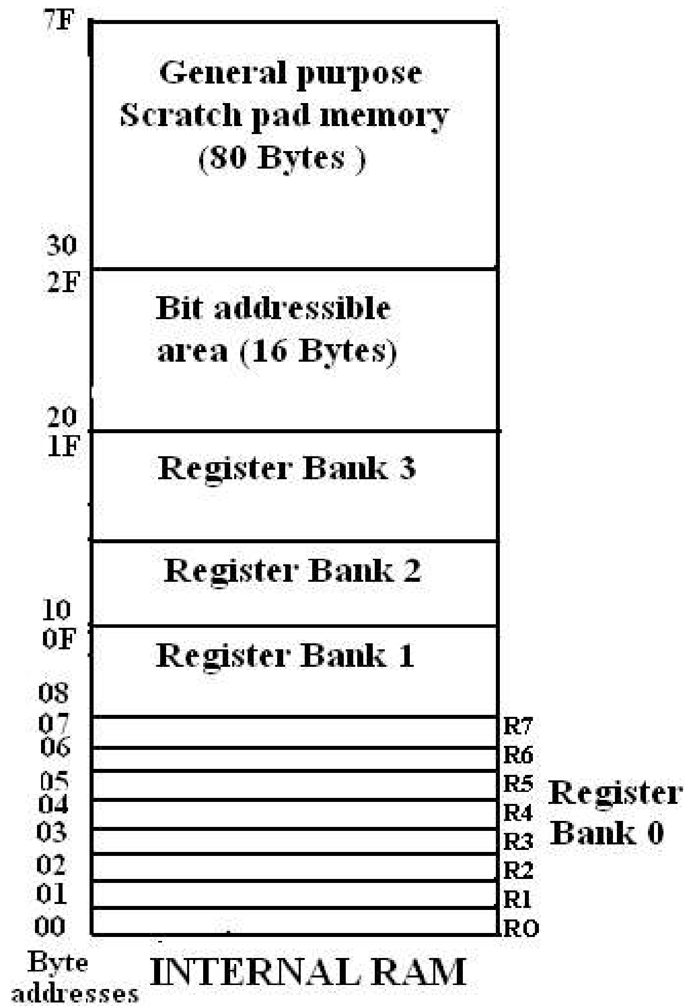
This Internal RAM is found on-chip on the 8051 .So it is the fastest RAM available, and it is also the most flexible in terms of reading, writing, and modifying it's contents. Internal RAM is volatile, so when the 8051 is reset this memory is cleared. The 128 bytes of internal RAM is organized as below.

(i) Four register banks (Bank0, Bank1, Bank2 and Bank3) each of 8-bits (total 32 bytes). The default bank register is Bank0. The remaining Banks are selected with the help of RS0 and RS1 bits of PSW Register.

(ii) 16 bytes of bit addressable area and

(iii) 80 bytes of general purpose area (Scratch pad memory) as shown in the diagram below.

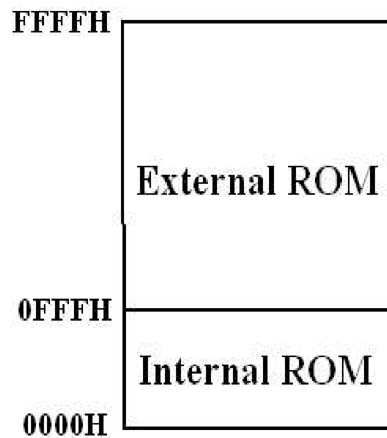
This area is also utilized by the microcontroller as a storage area for the operating stack.



The 32 bytes of RAM from address 00 H to 1FH are used as working registers organized as four banks of eight registers each. The registers are named as R0-R7 .Each register can be addressed by its name or by its RAM address.

For EX: MOV A, R7 or MOV R7, #05H

Internal ROM (On –chip ROM): The 8051 microcontroller has 4kB of on chip ROM but it can be extended up to 64kB. This ROM is also called program memory or code memory. The CODE segment is accessed using the program counter (PC) for opcode fetches and by DPTR for data. The external ROM is accessed when the EA(active low) pin is connected to ground or the contents of program counter exceeds 0FFFH.When the Internal ROM address is exceeded the 8051 automatically fetches the code bytes from the external program memory.



SPECIAL FUNCTION REGISTERS (SFRs): In 8051 microcontroller there certain registers which uses the RAM addresses from 80h to FFh and they are meant for certain specific operations. These registers are called Special function registers (SFRs). Some of these registers are bit addressable also.

The list of SFRs and their functional names are given below. In these SFRs some of them are related to I/O ports (P0, P1, P2 and P3) and some of them are meant for control operations (TCON, SCON, PCON...) and remaining are the auxiliary SFRs, in the sense that they don't directly configure the 8051.

S.No	Symbol		Name of SFR	Address (Hex)
1	ACC*		Accumulator	**0E0**
2	B*		B-Register	**0F0**
3	PSW*		Program Status word register	**0DO**
4	SP		Stack Pointer Register	**81**
5	DPTR	DPL	Data pointer low byte	**82**
		DPH	Data pointer high byte	**83**
6	P0*		Port 0	**80**
	P1*		Port 1	**90**
8	P2*		Port 2	**0A**
9	P3*		Port 3	**0B**
10	IP*		Interrupt Priority control	**0B8**

11	IE*	Interrupt Enable control	**0A8**
12	TMOD	Tmier mode register	**89**
13	TCON*	Timer control register	**88**
14	TH0	Timer 0 Higher byte	**8C**
15	TL0	Timer 0 Lower byte	**8A**
16	TH1	Timer 1Higher byte	**8D**
17	TL1	Timer 1 lower byte	**8B**
18	SCON*	Serial control register	**98**
19	SBUF	Serial buffer register	**99**
20	PCON	Power control register	**87**

The * indicates the bit addressable SFRs

Table: SFRs of 8051 Microcontroller

PARALLEL I /O PORTS :

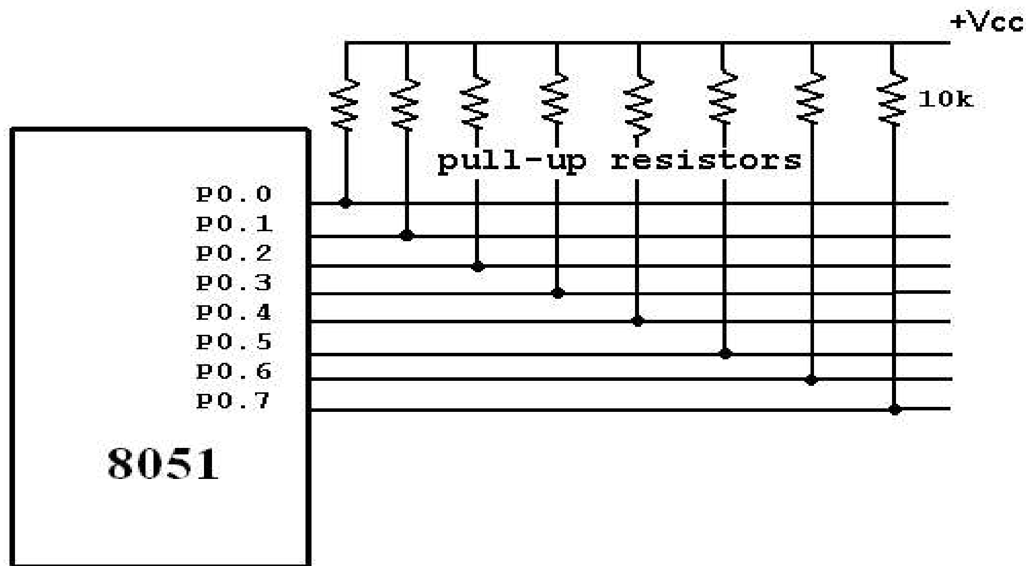
The 8051 microcontroller has four parallel I/O ports, each of 8-bits .So; it provides the user 32 I/O lines for connecting the microcontroller to the peripherals. The four ports are P0 (Port 0), P1 (Port1), P2 (Port 2) and P3 (Port3). Upon reset all the ports are output ports. In order to make them input, all the ports must be set i.e a high bit must be sent to all the port pins. This is normally done by the instruction “SETB”.

```
Ex: MOV A, #0FFH      ; A = FF
    MOV P0, A         ; make P0 an input port
```

PORT 0:

Port 0 is an 8-bit I/O port with dual purpose. If external memory is used, these port pins are used for the lower address byte address/data (AD_0-AD_7), otherwise all bits of the port are either input or output. Unlike other ports, Port 0 is not provided with pull-up resistors internally, so for PORT0 pull-up resistors of nearly 10k are to be connected externally as shown in the fig.2.

Dual role of port 0: Port 0 can also be used as address/data bus (AD0-AD7), allowing it to be used for both address and data. When connecting the 8051 to an external memory, port 0 provides both address and data. The 8051 multiplexes address and data through port 0 to save the pins. ALE indicates whether P0 has address or data. When ALE = 0, it provides data D0-D7, and when ALE =1 it provides address and data with the help of a 74LS373 latch.



Port 1: Port 1 occupies a total of 8 pins (pins 1 through 8). It has no dual application and acts only as input or output port. In contrast to port 0, this port does not need any pull-up resistors since pull-up resistors connected internally. Upon reset, Port 1 is configured as an output port. To configure it as an input port, port bits must be set i.e a high bit must be sent to all the port pins. This is normally done by the instruction "SETB". For

Ex: MOV A, #0FFH ; A=FF HEX

MOV P1, A ; make P1 an input port by writing 1's to all of its pins

Port 2: Port 2 is also an eight bit parallel port. (Pins 21- 28). It can be used as input or output port. As this port is provided with internal pull-up resistors it does not need any external pull-up resistors. Upon reset, Port 2 is configured as an output port. If the port is to be used as input port, all the port bits must be made high by sending FF to the port. For ex,

MOV A, #0FFH ; A=FF hex

MOV P_2, A ; make P_2 an input port by writing all 1's to it

Dual role of port 2: Port2 lines are also associated with the higher order address lines A8-A15. In systems based on the 8751, 8951, and DS5000, Port2 is used as simple I/O port. But, in 8031-based systems, port 2 is used along with P0 to provide the 16-bit address for the external memory. Since an 8031 is capable of accessing 64K bytes of external memory, it needs a path for the 16 bits of the address. While P0 provides the lower 8 bits via A0-A7, it is the job of P2 to provide bits A8-A15 of the address. In other words, when 8031 is connected to external memory, Port 2 is used for the upper 8 bits of the 16 bit address, and it cannot be used for I/O operations.

PORT 3: Port3 is also an 8-bit parallel port with dual function. (Pins 10 to 17). The port pins can be used for I/O operations as well as for control operations. The details of these additional operations are given below in the table. Port 3 also do not need any external pull-up resistors as they are provided internally similar to the case of Port2 & Port 1. Upon reset port 3 is configured as an output port. If the port is to be used as input port, all the port bits must be made high by sending FF to the port. For ex,

MOV A, #0FFH ; A= FF hex

MOV P_3, A ; make P_3 an input port by writing all 1's to it

Alternate Functions of Port 3: P3.0 and P3.1 are used for the RxD (Receive Data) and TxD (Transmit Data) serial communications signals. Bits P3.2 and P3.3 are meant for external interrupts. Bits P3.4 and P3.5 are used for Timers 0 and 1 and P3.6 and P3.7 are used to provide the write and read signals of external memories connected in 8031 based systems

S. No	Port 3 bit	Pin No	Function
1	P3.0	10	RxD
2	P3.1	11	TxD
3	P3.2	12	$\overline{INT0}$
4	P3.3	13	$\overline{INT1}$
5	P3.4	14	T0
6	P3.5	15	T1
7	P3.6	16	$\overline{WR}$
8	P3.7	17	$\overline{RD}$

Table: PORT 3 alternate functions

Interrupt Structure: An interrupt is an external or internal event that disturbs the microcontroller to inform it that a device needs its service. The program which is associated with the interrupt is called the **interrupt service routine** (ISR) or **interrupt handler**. Upon receiving the interrupt signal the Microcontroller, finish current instruction and saves the PC on stack. Jumps to a fixed location in memory depending on type of interruptStarts to execute the interrupt service routine until RETI (return from interrupt) upon executing the RETI the microcontroller returns to the place where it was interrupted. Get pop PC from stack

18

The 8051 microcontroller has **FIVE** interrupts in addition to Reset. They are

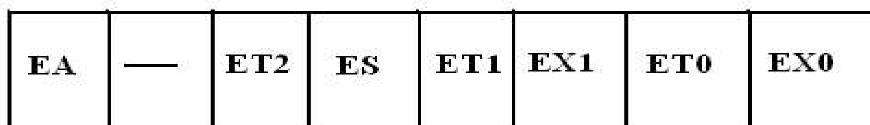
- Timer 0 overflow Interrupt
- Timer 1 overflow Interrupt
- External Interrupt 0(INT0)
- External Interrupt 1(INT1)
- Serial Port events (buffer full, buffer empty, etc) Interrupt

Each interrupt has a specific place in code memory where program execution (interrupt service routine) begins.

- External Interrupt 0: 0003 H
- Timer 0 overflow: 000B H
- External Interrupt 1: 0013 H
- Timer 1 overflow: 001B H
- Serial Interrupt : 0023 H

Upon reset all Interrupts are disabled & do not respond to the Microcontroller. These interrupts must be enabled by software in order for the Microcontroller to respond to them. This is done by an 8-bit register called Interrupt Enable Register (IE).

Interrupt Enable Register:



EA	—	ET2	ES	ET1	EX1	ET0	EX0

- EA : Global enable/disable. To enable the interrupts this bit must be set high.
- --- : Undefined-reserved for future use.
- ET2: Enable /disable Timer 2 overflow interrupt.
- ES : Enable/disable Serial port interrupts.
- ET1: Enable /disable Timer 1 overflow interrupt.
- EX1: Enable/disable External interrupt1.
- ET0: Enable /disable Timer 0 overflow interrupt.
- EX0 : Enable/disable External interrupt0

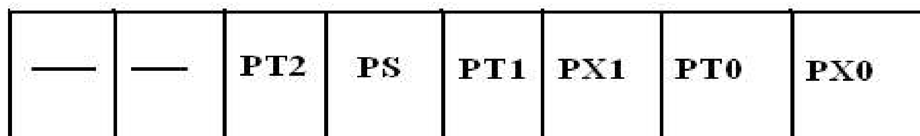
19

Interrupt Priority Register:

Upon reset the interrupts have the following priority. (Top to down). The interrupt with the highest PRIORITY gets serviced first.

1. External interrupt 0 (INT0)
2. Timer interrupt0 (TF0)
3. External interrupt 1 (INT1)
4. Timer interrupt1 (TF1)
5. Serial communication (RI+TI)

Priority can also be set to "high" or "low" by 8-bit IP register.- Interrupt priority register



—	—	PT2	PS	PT1	PX1	PT0	PX0

IP.7: reserved

IP.6: reserved

IP.5: Timer 2 interrupt priority bit (8052 only)

IP.4: Serial port interrupt priority bit

IP.3: Timer 1 interrupt priority bit

IP.2: External interrupt 1 priority bit

IP.1: Timer 0 interrupt priority bit

IP.0: External interrupt 0 priority bit

TIMERS in 8051 Microcontrollers:

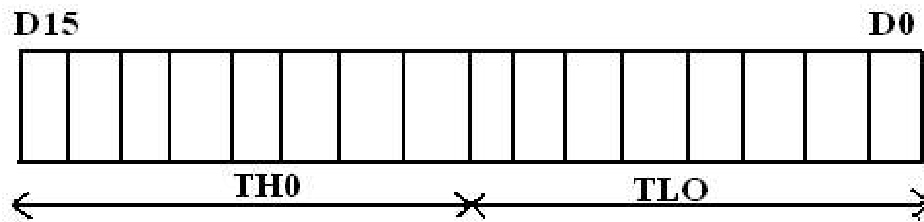
The 8051 microcontroller has two 16-bit timers Timer 0 (T0) and Timer 1(T1) which can be used either to generate accurate time delays or as event counters. These timers are accessed as two 8-bit registers TLO, THO & TL1, TH1 because the 8051 microcontroller has 8-bit architecture.

TIMER 0 : The Timer 0 is a 16-bit register and can be treated as two 8-bit registers (TL0 & TH0) and these registers can be accessed similar to any other registers like A,B or R1,R2,R3 etc…

Ex: The instruction MOV TL0,#07 moves the value 07 into lower byte of Timer0.

20

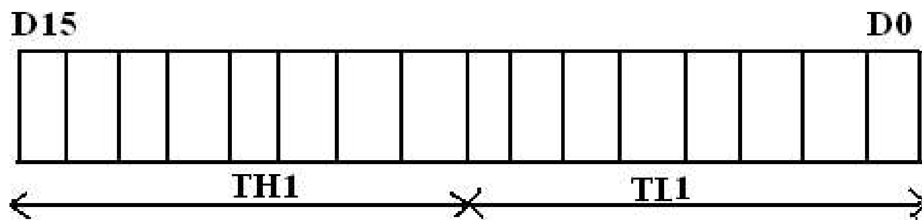
Similarly MOV R5, TH0 saves the contents of TH0 in the R5 register.



TIMER 1 : The Timer 1 is also a 16-bit register and can be treated as two 8-bit registers (TL1 & TH1) and these registers can be accessed similar to any other registers like A,B or R1,R2,R3 etc…

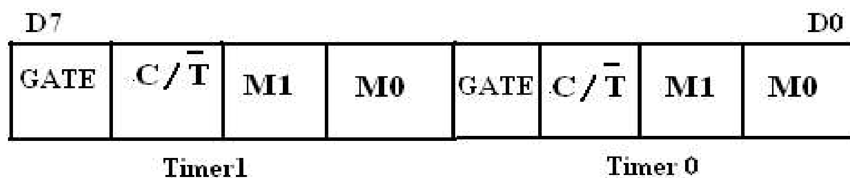
Ex: The instruction MOV TL1,#05 moves the value 05 into lower byte of Timer1.

Similarly MOV R0, TH1 saves the contents of TH1 in the R0 register



TMOD Register:

The various operating modes of both the timers T0 and T1 are set by an 8-bit register called TMOD register**.** In this TMOD register the lower 4-bits are meant for Timer 0 and the higher 4-bits are meant for Timer1.



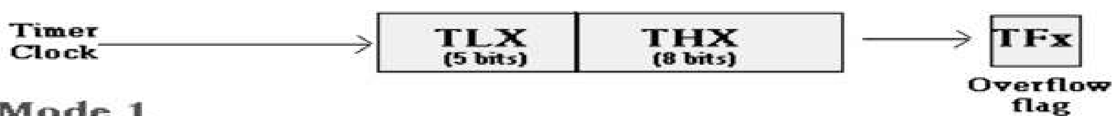
GATE: This bit is used to start or stop the timers by hardware .When GATE= 1, the timers can be started / stopped by the external sources. When GATE= 0, the timers can be started or stopped by software instructions like SETB TR0 or SETB TR1

C/T (clock/Timer): This bit decides whether the timer is used as delay generator or event counter. When **C/T = 0,** the Timer is used as delay generator and if C/T=1 the timer is used as an event counter. The clock source for the time delay is the crystal frequency of 8051.

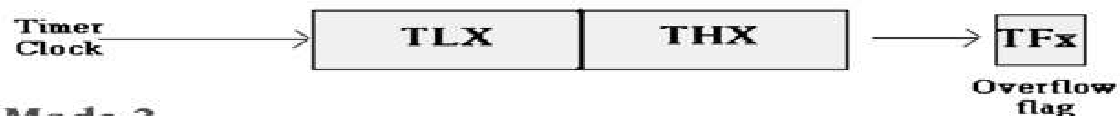
M1, M0 (Mode): These two bits are the timer mode bits. The timers of the 8051 can be configured in three modes. Mode0, Mode1 and Mode2. The selection and operation of the modes is shown below.

S. No	M0	M1	Mode	Operation
1	0	0	0	13-bit Timer mode 8-bit Timer/counter. THx with TLx as 5-bit prescalar
2	0	1	1	16-bit Timer mode.16-bit timer /counter without pre-scalar
3	1	0	2	8-bit auto reload. THx contains a value that is to be loaded into TLx each time it overflows
4	1	1	3	Split timer mode

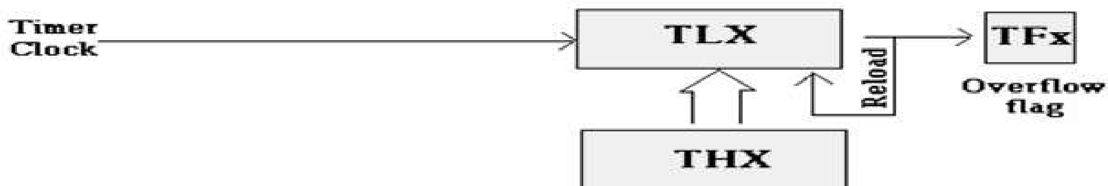
Mode 0



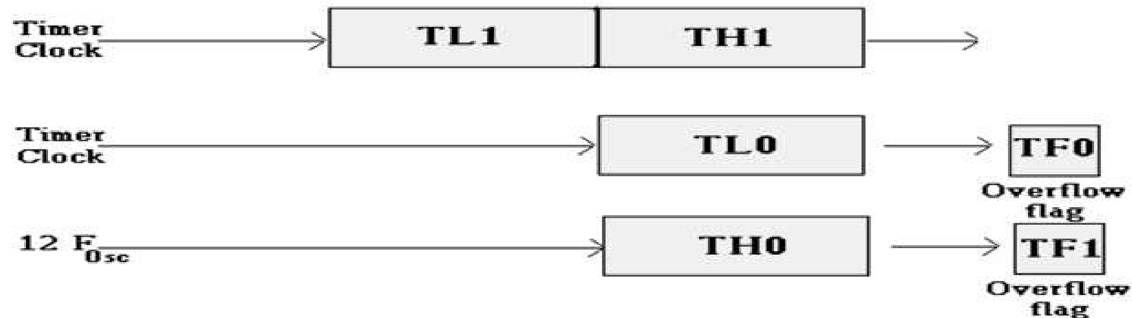
Mode 1



Mode 2



Mode 3



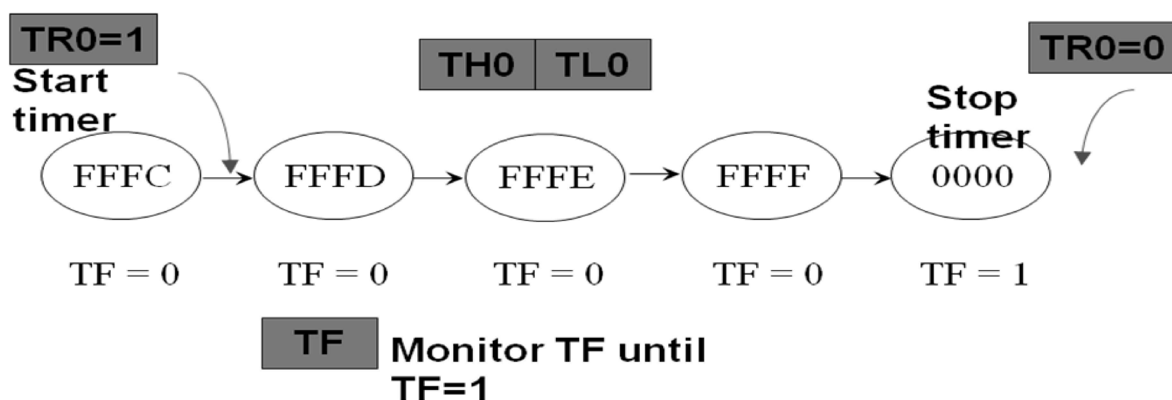
Let us understand the working of Timer Mode 1

- For this , let us consider timer 0 as an example.
- **16-bit** timer (TH0 and TL0)
- TH0-TL0 is incremented continuously when TR0 is set to 1. And the 8051 stops to increment TH0-TL0 when TR0 is cleared.

- The timer works with the internal system clock. In other words, the timer counts up each machine cycle.
- When the timer (TH0-TL0) reaches its maximum of FFFFH, it rolls over to 0000, and TF0 is raised.
- Programmer should check TF0 and stop the timer 0.

Steps of Mode 1

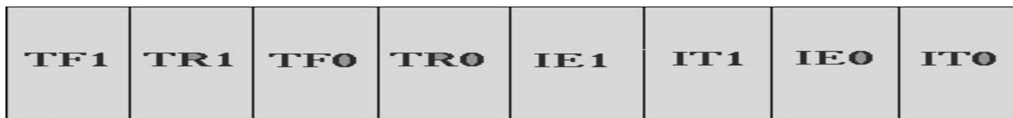
1. **Choose mode 1 timer 0**
   - **MOV TMOD,#01H**
2. **Set the original value to TH0 and TL0.**
   - **MOV TH0,#FFH**
   - **MOV TL0,#FCH**
3. **You better to clear the TF: TF0=0.**
   - **CLR TF0**
4. **Start the timer.**
   - **SETB TR0**
5. **The 8051 starts to count up by incrementing the TH0-TL0.**
   - **TH0-TL0= FFFCH,FFFDH,FFFEH,FFFFH,0000H**



6. **When TH0-TL0 rolls over from FFFFH to 0000, the 8051 set TF0=1.**
   - **TH0-TL0= FFFE H, FFFF H, 0000 H (Now TF0=1)**
7. **Keep monitoring the timer flag (TF) to see if it is raised.**
   - **AGAIN: JNB TF0, AGAIN**

8. **Clear TR0 to stop the process.**
   - **CLR TR0**
9. **Clear the TF flag for the next round.**
   - **CLR TF0**

TCON Register:



Timer control register TMOD is a 8-bit register which is bit addressable and in which Upper nibble is for timer/counter, lower nibble is for interrupts.

- **TR** (Timer run control bit)
  - TR0 for Timer/counter 0; TR1 for Timer/counter 1.
  - TR is set by programmer to turn timer/counter on/off.
    - TR=0 : off (stop)
    - TR=1 : on (start)
- **TF** (timer flag, control flag)
  - **TF0 for timer/counter 0; TF1 for timer/counter 1.**
  - **TF is like a carry. Originally, TF=0. When TH-TL roll over to 0000 from FFFFH, the TF is set to 1.**
    - TF=0 : not reach
    - TF=1 : reach
    - If we enable interrupt, TF=1 will trigger ISR.

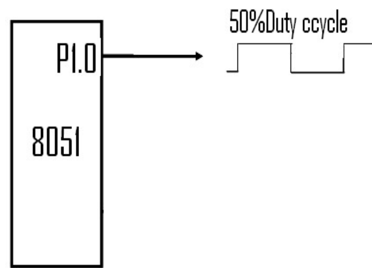
Simple applications using Ports &Timers

- Using a port, by a simple program you can generate a Square wave of any duty cycle.

```
HERE:  SETB    P1.0       (Make bit of Port 0 High)
       LCALL DELAY
       CLR P1.0
       LCALL DELAY
```

SJMP HERE ; Keep doing it

Here same delay is used for both High & low



8051-SERIAL COMMUNICATION:

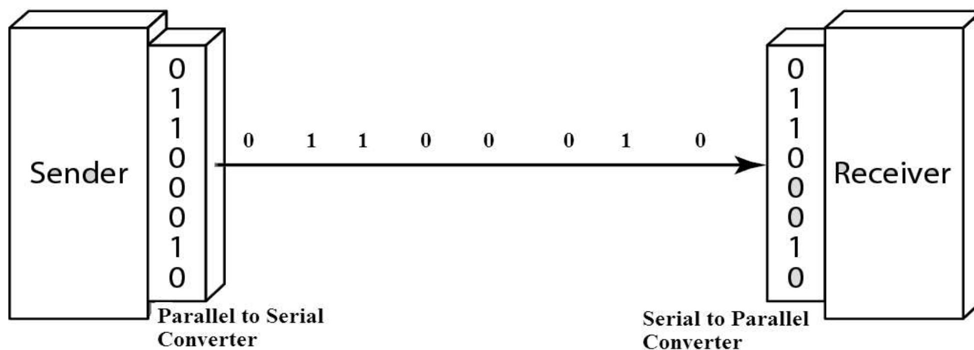
Basics of Serial communication

Data transfer between two electronic devices (Ex Between a computer and microcontroller or a peripheral device) is generally done in two ways

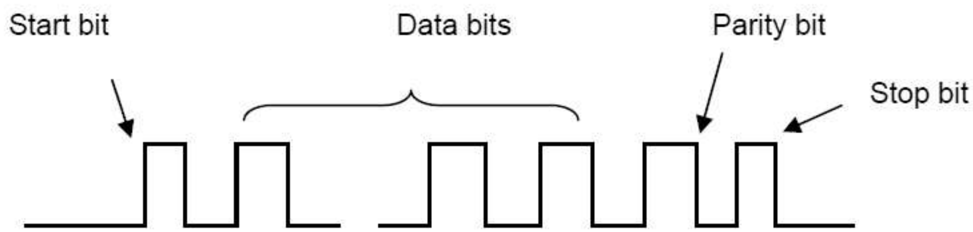
(i).Serial data Transfer and

(ii).Parallel data Transfer

Serial communication uses only one or two data lines to transfer data and is generally used for long distance communication. In serial communication the data is sent as one bit at a time in a timed sequence on a single wire. Serial Communication takes place in two methods, Asynchronous data Transfer and Synchronous data Transfer.



Asynchronous data transfer allows data to be transmitted without the sender having to send a clock signal to the receiver. Instead, special bits will be added to each word in order to synchronize the sending and receiving of the data. When a word is given to the UART for Asynchronous transmissions, a bit called the "Start Bit" is added to the beginning of each word that is to be transmitted. The Start Bit is used to alert the receiver that a word of data is about to be sent, and to force the clock in the receiver into synchronization with the clock in the transmitter.



Serial data transmission

After the Start Bit, the individual bits of the word of data are sent .Here each bit in the word is transmitted for exactly the same amount of time as all of the other bits. When the entire data word has been sent, the transmitter may add a Parity Bit that the transmitter generates. The Parity bit may be used by the receiver to perform simple error checking. Then at least one Stop Bit is sent by the transmitter. If the Stop Bit does not appear when it is supposed to, the UART considers the entire word to be corrupted and will report a Framing Error.

Baud rate is a measurement of transmission speed in asynchronous communication , it represents the number of bits/sec that are actually being sent over the serial link. The Baud count includes the overhead bits Start, Stop and Parity that are generated by the sending UART and removed by the receiving UART.

In the **Synchronous data transfer** method the receiver knows when to "read" the next bit coming from the sender. This is achieved by sharing a clock between sender and receiver. In most forms of serial Synchronous communication, if there is no data available at a given time to transmit, a fill character will be sent instead so that data is always being transmitted. Synchronous communication is usually more efficient because only data bits are transmitted between sender and receiver, however it will be more costly because extra wiring and control circuits are required to share a clock signal between the sender and receiver.

Devices that use serial cables for their communication are split into two categories.

1. DTE (Data Terminal Equipment). Examples of DTE are computers, printers & terminals.
2. DCE (Data Communication Equipment). Example of DCE is modems.

Parallel Data Transfer:

Parallel communication uses multiple wires (bus) running parallel to each other, and can transmit data on all the wires simultaneously. i.e all the bits of the byte are transmitted at a time. So, speed of the parallel data transfer is extremely high compared to serial data transfer. An 8-bit parallel data transfer is 8-times faster than serial data transfer. Hence with in the computer all data transfer is mainly based on Parallel data transfer. But only limitation is due to the high cost ,this method is limited to only short distance communications.

Differences between Serial data transfer and Parallel data transfer

S.No	Serial Communication	Parallel Communication
1	Data is transmitted bit after the bit in a single line	Data is transmitted simultaneously through group of lines(Bus)
2	Data congestion takes place	No, Data congestion
3	Low speed transmission	High speed transmission
4	Implementation of serial links is not an easy task.	Parallel data links are easily implemented in hardware
5.	In terms of transmission channel costs such as data bus cable length, data bus buffers, interface connectors, it is less expensive	It is more expensive
6	No , crosstalk problem	Crosstalk creates interference between the parallel lines.
7	No effect of inter symbol interference and noise	Parallel ports suffer extremely from inter-symbol interference (ISI) and noise, and therefore the data can be corrupted over long distances.
8	The bandwidth of serial wires is much higher.	The bandwidth of parallel wires is much lower.
9	Serial interface is more flexible to upgrade , without changing the hardware	Parallel data transfer mechanism rely on hardware resources and hence not flexible to upgrade.
10	Serial communication work effectively even at high frequencies.	Parallel buses are hard to run at high frequencies.

SERIAL COMMUNICATION IN 8051 MICROCONTROLLER

The 8051 has two pins for transferring and receiving data by serial communication. These two pins are part of the Port3(P3.0 &P3.1) .These pins are TTL compatible and hence they require a line driver to make them RS232 compatible .Max232 chip is one such line driver in use. Serial communication is controlled by an 8-bit register called SCON register,it is a bit addressable register.

SCON (Serial Control) Register:

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0	SCON.7	Serial port mode selector
SM1	SCON.6	Serial port mode selector
SM2	SCON.5	Used for multiprocessor mode communication (not applicable for 8051)
REN	SCON.4	Receive enable. Set or cleared by making this bit either 1 or 0 foe enable /disable reception.
TB8	SCON.3	9th data bit transmitted in modes 2 and 3
RB8	SCON.2	9th data bit received in modes 2 and 3.it is not used in mode 0 & mode 1.If SM2 = 0 RB8 is the stop bit .
TI	SCON.1	Transmit interrupt flag
RI	SCON.0	Receive interrupt flag.

- SM0, SM1: These two bits of SCON register determine the framing of data by specifying the number of bits per character and start bit and stop bits. There are 4 serial modes.

SM0 SM1

0 0 : Serial Mode 0

0 1 : Serial Mode 1, 8 bit data, 1 stop bit, 1 start bit

1 0 : Serial Mode 2

1 1 : Serial Mode 3

- REN (Receive Enable) also referred as SCON.4. When it is high,it allows the 8051 to receive data on the RxD pin. So to receive and transfer data REN must be set to 1.When REN=0,the receiver is disabled. This is achieved as below

SETB SCON.4

& CLR SCON.4

- TI (Transmit interrupt) is the D1 bit of SCON register. When 8051 finishes the transfer of 8-bit character, it raises the TI flag to indicate that it is ready to transfer another byte. The TI bit is raised at the beginning of the stop bit.

RI (Receive interrupt) is the D0 bit of the SCON register. When the 8051 receives data serially ,via RxD, it gets rid of the start and stop bits and places the byte in the SBUF register. Then 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.

Communication through RS232

A personal computer has a serial port known as communication port or COM Port used to connect a modem for example or any other device, there could be more then one COM Port in a PC. Serial ports are controlled by a special chip called UART (Universal Asynchronous Receiver Transmitter).

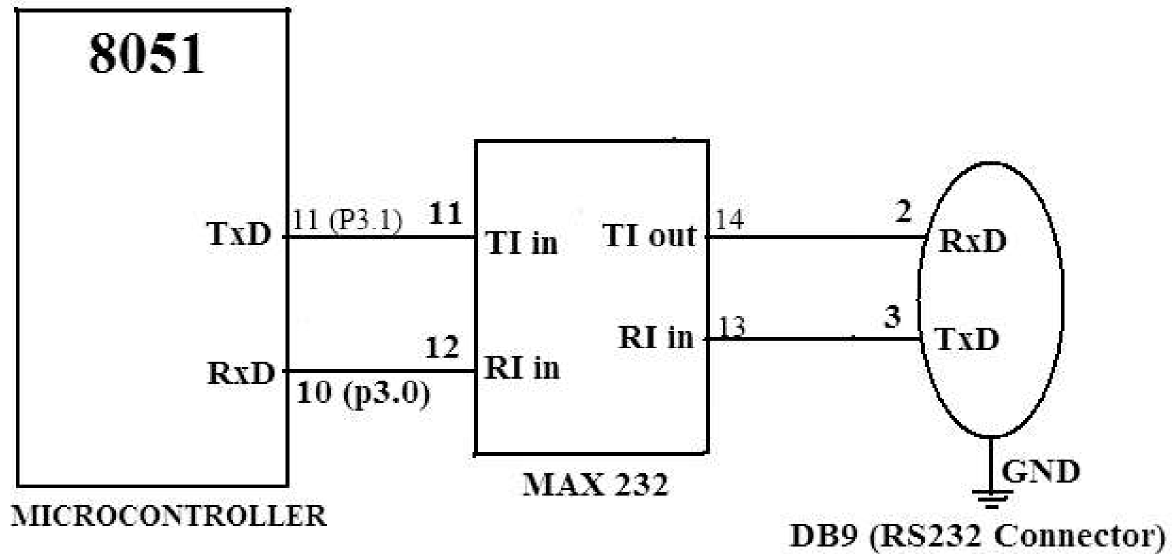
RS 232 standard describes a communication method where information is sent bit by bit on a physical channel. The **RS stands for Recommended Standard**. The information must be broken up in data words. The length of a data word is variable.

It is one of the popularly known interface standard for serial communication between DTE & DCE. This RS-232-C is the commonly used standard when data are transmitted as voltage .This standard was first developed by Electronic industries association(EIA). For the RS-232C, a 25 pin D type connector is used . DB-25P male and DB-25S female. RS-232 standard was first introduced in 1960's by Telecommunications Industry Association(TIA).

Interfacing the 8051 Microcontroller to PC:

As the RS-232 standard is developed earlier to TTL devices ,a USART such as 8251 is not directly compatible with these signal levels .Because of this ,voltage transistors called line drivers and line receivers are used to interface TTL logic with RS-232 signals . The line driver MC 1488 is used to convert RS-232 to TTL.The microcontroller is connected to the PC using the DB9 connector.

The TxD and Rx D pins are connected to the TI in and RI in pins of the MAX 232 IC and the TI out and RI in pins of the MAX IC are connected to the RxD and TxD pins of the DB9 connector as shown in the interface diagram.



ADDRESSING MODES OF 8051:

The way in which the data operands are accessed by different instructions is known as the addressing modes. There are various methods of denoting the data operands in the instruction. The 8051 microcontroller supports mainly 5 addressing modes. They are

1. Immediate addressing mode

2. Direct Addressing mode

3. Register addressing mode

4. Register Indirect addressing mode

5. Indexed addressing mode

Immediate addressing mode : The addressing mode in which the data operand is a constant and it is a part of the instruction itself is known as Immediate addressing mode. Normally the data must be preceded by a # sign. This addressing mode can be used to transfer the data into any of the registers including DPTR.

Ex: MOV A , # 27 H : The data (constant) 27 is moved to the accumulator register

ADD R1 ,#45 H : Add the constant 45 to the contents of the accumulator

MOV DPTR ,# 8245H :Move the data 8245 into the data pointer register.

MOV P1,#21 H

Direct addressing mode: The addressing mode in which the data operand is in the RAM location (00 - 7FH) and the address of the data operand is given in the instruction is known as Direct addressing mode. The direct addressing mode uses the lower 128 bytes of Internal RAM and the SFRs

MOV R1, 42H : Move the contents of RAM location 42 into R1 register

MOV 49H,A : Move the contents of the accumulator into the RAM location 49.

ADD A, 56H : Add the contents of the RAM location 56 to the accumulator

Register addressing mode :The addressing mode in which the data operand to be manipulated lies in one of the registers is known as register addressing mode.

MOV A,R0 : Move the contents of the register R0 to the accumulator

ADD A,R6 :Add the contents of R6 register to the accumulator

MOV P1, R2 : Move the contents of the R2 register into port 1

MOV R5, R2 : This is invalid .The data transfer between the registers is not allowed.

Register Indirect addressing mode : The addressing mode in which a register is used as a pointer to the data memory block is known as Register indirect addressing mode.

MOV A,@ R0 :Move the contents of RAM location whose address is in R0 into **A** (accumulator)

MOV @ R1 , B : Move the contents of B into RAM location whose address is held by R1

When R0 and R1 are used as pointers, they must be preceded by @ sign

One of the advantages of register indirect addressing mode is that it makes accessing the data more dynamic than static as in the case of direct addressing mode.

Indexed addressing mode : This addressing mode is used in accessing the data elements of lookup table entries located in program ROM space of 8051.

Ex : MOVC A, @ A+DPTR

The 16-bit register DPTR and register A are used to form the address of the data element stored in on-chip ROM. Here C denotes code .In this instruction the contents of A are added to the 16-bit DPTR register to form the 16-bit address of the data operand.

8051 Instruction Set:

- Assembly language is machine dependant.
- Each family of microprocessors or microncontrollers has its own instruction set.
- Each instruction has an 8 bit op-code with an associated mnemonic.
- Some instructions have one or two additional bytes for operand (data or addresses).
- The 8051 has 255 instructions

  Every 8-bit opcode from 00 to FF is used except for A5.

- The instructions are grouped into 5 groups
  - ✓ Arithmetic
  - ✓ Logic
  - ✓ Data Transfer
  - ✓ Boolean
  - ✓ Branching

Arithmetic Instructions:

- ADD
  - 8-bit addition between the accumulator (A) and a second operand.
    - The result is always in the accumulator.
    - The CY flag is set/reset appropriately.
- ADDC
  - 8-bit addition between the accumulator, a second operand and the previous value of the CY flag.
    - Useful for 16-bit addition in two steps.
    - The CY flag is set/reset appropriately.

Example – 16-bit Addition

ADD 1E44H to 56CAH

```
CLR   C               ; Clear the CY flag
MOV   A, #44H         ; The lower 8-bits of the 1st number
```


32

```
ADD   A, #CAH        ; The lower 8-bits of the 2nd number
MOV   R1, A          ; The result 0EH will be in R1. CY = 1.
MOV   A, #1EH        ; The upper 8-bits of the 1st number
ADDC  A, #56H        ; The upper 8-bits of the 2nd number
MOV   R2, A          ; The result of the addition is 75H
```

The overall result: **750EH will be in R2:R1**. CY = 0.

- DAA
  - Decimal adjust the accumulator.
    - Format the accumulator into a proper 2 digit packed BCD number.
    - Operates only on the accumulator.
    - Works only after the ADD instruction.
- SUBB
  - Subtract with Borrow.
    - Subtract an operand and the previous value of the borrow (carry) flag from the accumulator.
      - A ← A - <operand> - CY.
      - The result is always saved in the accumulator.
      - The CY flag is set/reset appropriately.

Example – BCD addition

ADD 34 to 49 BCD

```
CLR C           ; Clear the CY flag
MOV A, #34H     ; Place 1st number in A
ADD A, #49H     ; Add the 2nd number.
                ; A = 7DH
DAA             ; A = 83H
```

- INC
  - Increment the operand by one.
    - The operand can be a register, a direct address, an indirect address, the data pointer.

- DEC
    - Decrement the operand by one.
        - The operand can be a register, a direct address, an indirect address.
- MUL AB / DIV AB
    - Multiply A by B and place result in A:B.
    - Divide A by B and place result in A:B.

Logical Operations:

- ANL / ORL
    - Work on byte sized operands or the CY flag.
        - ANL A, Rn
        - ANL A, direct
        - ANL A, @Ri
        - ANL A, #data
        - ANL direct, A
        - ANL direct, #data
        - ANL C, bit
        - ANL C, /bit
- XRL
    - Works on bytes only.
    - CPL / CLR
    - Complement / Clear.
    - Work on the accumulator or a bit.
        - CLR P1.2
- RL / RLC / RR / RRC
    - Rotate the accumulator.
        - RL and RR without the carry
        - RLC and RRC rotate through the carry.
- SWAP A
    - Swap the upper and lower nibbles of the accumulator.
- No compare instruction.
    - Built into conditional branching instructions.

Data Transfer Instructions:

- Data is stored at the source address and moved (copied) to a destination address.
- The way these addresses are specified are determined by the addressing mode.
- There are 28 different instructions for data transfer, which can be categorized into three types:
    - MOV <dest>, <src>

34

- Push <source> or Pop <dest>
- XCH <dest>,<src>

- MOV
  - 8-bit data transfer for internal RAM and the SFR.
    - MOV A, Rn
    - MOV A, direct
    - MOV A, @Ri
    - MOV A, #data
    - MOV Rn, A
    - MOV Rn, direct
    - MOV Rn, #data
    - MOV direct, A
    - MOV direct, Rn
    - MOV direct, direct
    - MOV direct, @Ri
    - MOV direct, #data
    - MOV @Ri, A
    - MOV @Ri, direct
- MOVC
  - Move Code Byte
    - Load the accumulator with a byte from program memory.
    - Must use indexed addressing
      - MOVC A, @A+DPTR
      - MOVC A, @A+PC
- MOVX
  - Data transfer between the accumulator and a byte from external data memory.
    - MOVX A, @Ri
    - MOVX A, @DPTR
    - MOVX @Ri, A
    - MOVX @DPTR, A
- PUSH / POP
  - Push and Pop a data byte onto the stack.
  - The data byte is identified by a direct address from the internal RAM locations.
    - PUSH DPL

35

        - POP 40H
- XCH
  - Exchange accumulator and a byte variable
    - XCH A, Rn
    - XCH A, direct
    - XCH A, @Ri
- XCHD
  - Exchange lower digit of accumulator with the lower digit of the memory location specified.

    XCHD A, @Ri
    - The lower 4-bits of the accumulator are exchanged with the lower 4-bits of the internal memory location identified indirectly by the index register.
    - The upper 4-bits of each are not modified.

Boolean Operations:

- This group of instructions is associated with the single-bit operations of the 8051.
- This group allows manipulating the individual bits of bit addressable registers and memory locations as well as the CY flag.
  - The P, OV, and AC flags cannot be directly altered.
- This group includes:
  - Set, clear, and, or complement, move.
  - Conditional jumps.
- CLR
  - Clear a bit or the CY flag.
    - CLR P1.1
    - CLR C
- SETB
  - Set a bit or the CY flag.
    - SETB A.2
    - SETB C
- CPL
  - Complement a bit or the CY flag.
    - CPL 40H ; Complement bit 40 of the bit addressable memory

36

- ORL / ANL
  - OR / AND a bit with the CY flag.
    - ORL C, 20H ; OR bit 20 of bit addressable memory with the CY flag
    - ANL C, /34H ; AND complement of bit 34 of bit addressable memory with the CY flag.
- MOV
  - Data transfer between a bit and the CY flag.
    - MOV C, 3FH ; Copy the CY flag to bit 3F of the bit addressable memory.
    - MOV P1.2, C ; Copy the CY flag to bit 2 of P1.
- JC / JNC
  - Jump to a **relative address** if CY is set / cleared.
- JB / JNB
  - Jump to a **relative address** if a bit is set / cleared.
    - JB ACC.2, <label>
- JBC
  - Jump to a relative address if a bit is set and clear the bit.

Branching Instructions:

- The 8051 provides four different types of unconditional jump instructions:
  - Short Jump – SJMP
    - Uses an 8-bit signed offset relative to the 1st byte of the next instruction.
  - Long Jump – LJMP
    - Uses a 16-bit address.
    - 3 byte instruction capable of referencing any location in the entire 64K of program memory.
  - Absolute Jump – AJMP
    - Uses an **11-bit address**.
    - 2 byte instruction
      - The upper 3-bits of the address combine with the 5-bit opcode to form the 1st byte and the lower 8-bits of the address form the 2nd byte.

- The 11-bit address is substituted for the lower 11- bits of the PC to calculate the 16-bit address of the target.
  - The location referenced must be within the 2K Byte memory page containing the AJMP instruction.

– Indirect Jump – JMP

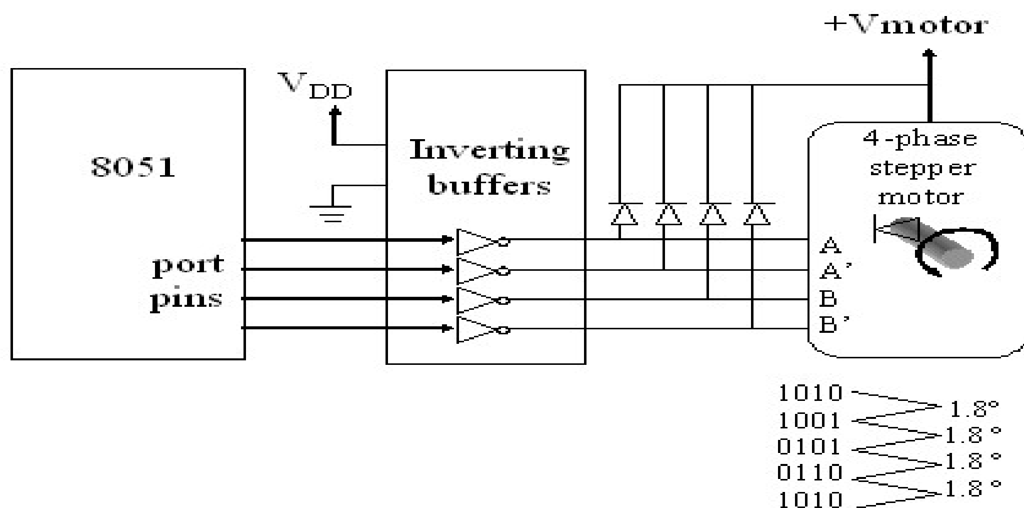
JMP @A + DPTR

- The 8051 provides 2 forms for the CALL instruction:
  - – Absolute Call – ACALL
    - Uses an 11-bit address similar to AJMP
    - The subroutine must be within the same 2K page.
  - – Long Call – LCALL
    - Uses a 16-bit address similar to LJMP
    - The subroutine can be anywhere.
  - – Both forms push the 16-bit address of the next instruction on the stack and update the stack pointer.
- The 8051 provides 2 forms for the return instruction:
  - – Return from subroutine – RET
    - Pop the return address from the stack and continue execution there.
  - – Return from ISV – RETI
    - Pop the return address from the stack.
    - Restore the interrupt logic to accept additional interrupts at the **same priority level as the one just processed**.
    - Continue execution at the address retrieved from the stack.
    - The PSW is not automatically restored.
- The 8051 supports 5 different conditional jump instructions.
  - – ALL conditional jump instructions use an 8-bit signed offset.
  - – Jump on Zero – JZ / JNZ
    - Jump if the A == 0 / A != 0
      - The check is done at the time of the instruction execution.
  - – Jump on Carry – JC / JNC
    - Jump if the C flag is set / cleared.
  - – Jump on Bit – JB / JNB
    - Jump if the specified bit is set / cleared.
    - Any addressable bit can be specified.

- Jump if the Bit is set then Clear the bit – JBC
    - Jump if the specified bit is set.
    - Then clear the bit.
- Compare and Jump if Not Equal – CJNE
  - Compare the magnitude of the two operands and jump if they are not equal.
    - The values are considered to be unsigned.
    - The Carry flag is set / cleared appropriately.
    - CJNE A, direct, rel
    - CJNE A, #data, rel
    - CJNE Rn, #data, rel
    - CJNE @Ri, #data, rel

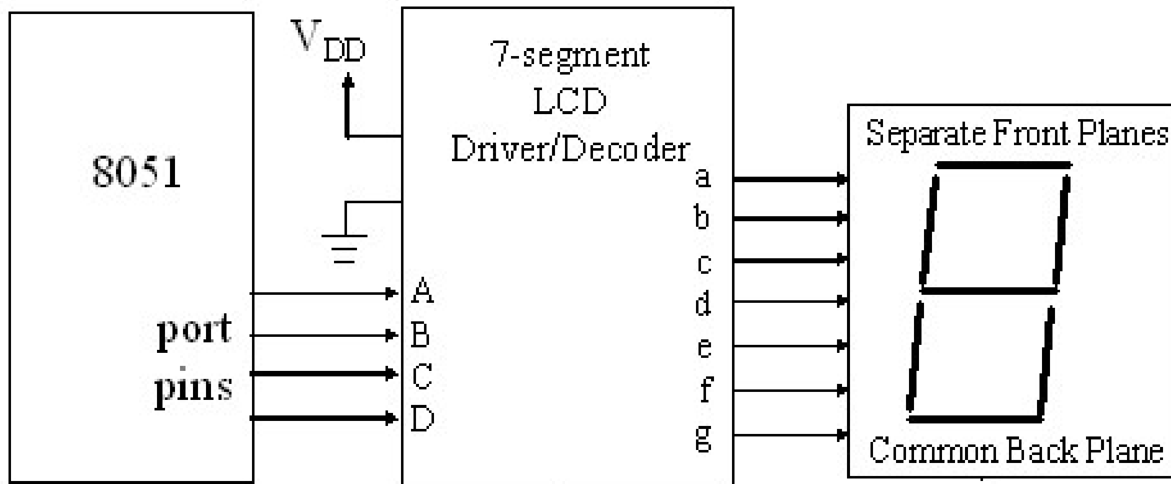
- Decrement and Jump if Not Zero – DJNZ
  - Decrement the first operand by 1 and jump to the location identified by the second operand if the resulting value is not zero.
    - DJNZ Rn, rel
    - DJNZ direct, rel
- No Operation
  - NOP

Applications of Microcontrollers

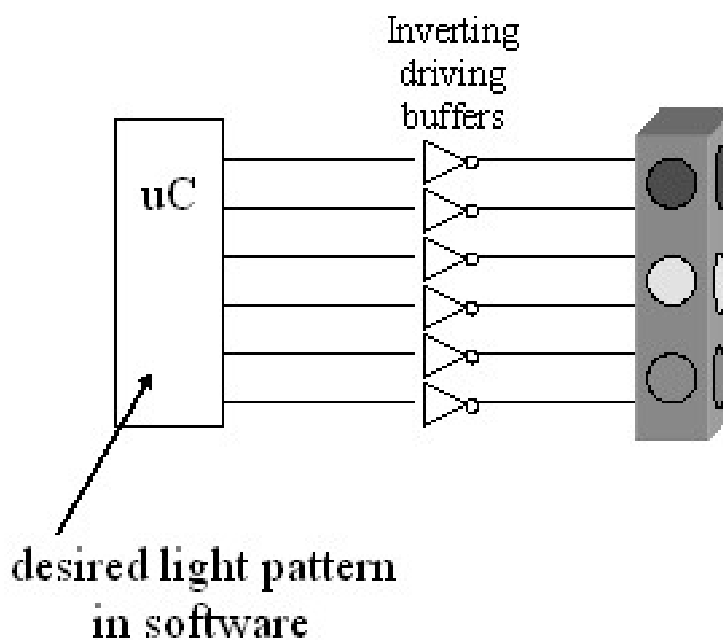


Seven segment Interfacing

Simple parallel interface – similar to LED:



Traffic light controller



40

Interfacing Keyboard/ Displays:

The key board here we are interfacing is a matrix keyboard. This key board is designed with a particular rows and columns. These rows and columns are connected to the microcontroller through its ports of the micro controller 8051. We normally use 8*8 matrix key board. So only two ports of 8051 can be easily connected to the rows and columns of the key board.

Whenever a key is pressed, a row and a column gets shorted through that pressed key and all the other keys are left open. When a key is pressed only a bit in the port goes high. Which indicates microcontroller that the key is pressed. By this high on the bit key in the corresponding column is identified.

Once we are sure that one of key in the key board is pressed next our aim is to identify that key. To do this we firstly check for particular row and then we check the corresponding column the key board.

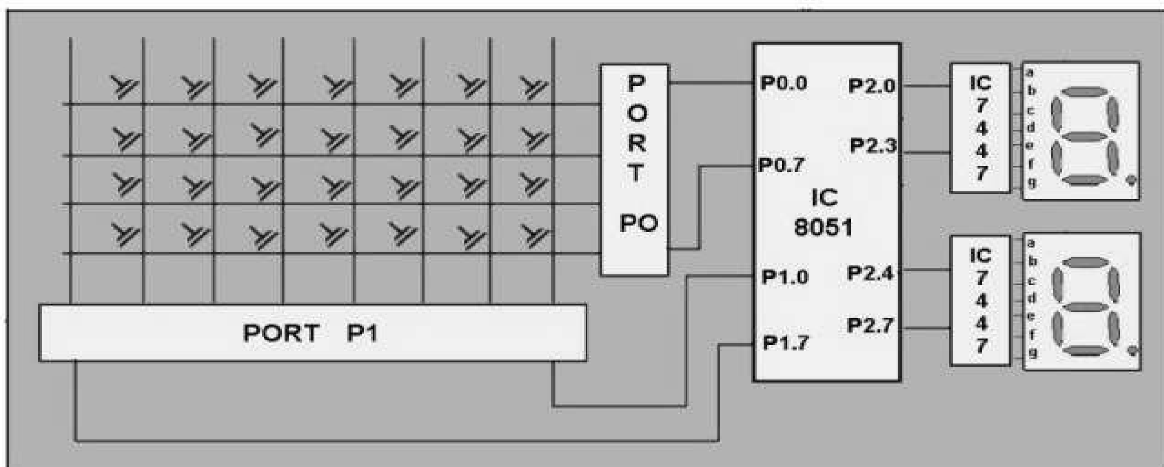
To check the row of the pressed key in the keyboard, one of the row is made high by making one of bit in the output port of 8051 high . This is done until the row is found out. Once we get the row next out job is to find out the column of the pressed key. The column is detected by contents in the input ports with the help of a counter. The content of the input port is rotated with carry until the carry bit is set.

The contents of the counter is then compared and displayed in the display. This display is designed using a seven segment display and a BCD to seven segment decoder IC 7447.

The BCD equivalent number of counter is sent through output part of 8051 displays the number of pressed key.

Fig: *Interfacing Key Board to 8051.*

The programming algorithm, program and the circuit diagram is as follows. Here program is explained with comments.

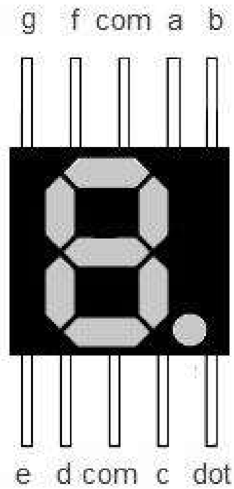


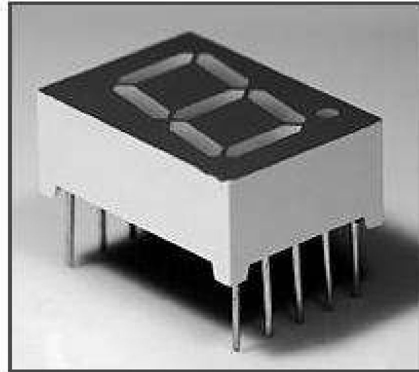
Keyboard is organized in a matrix of rows and columns as shown in the figure. The microcontroller accesses both rows and columns through the port.

1. The 8051 has 4 I/O ports P0 to P3 each with 8 I/O pins, P0.0 to P0.7, P1.0 to P1.7, P2.0 to P2.7, and P3.0 to P3.7. The one of the port P1 (it understood that P1 means P1.0 to P1.7) as an I/P port for microcontroller 8051, port P0 as an O/P port of microcontroller 8051 and port P2 is used for displaying the number of pressed key.
2. Make all rows of port P0 high so that it gives high signal when key is pressed.
3. See if any key is pressed by scanning the port P1 by checking all columns for non zero condition.
4. If any key is pressed, to identify which key is pressed make one row high at a time.
5. Initiate a counter to hold the count so that each key is counted.
6. Check port P1 for nonzero condition. If any nonzero number is there in [accumulator], start column scanning by following step 9.
7. Otherwise make next row high in port P1.
8. Add a count of 08h to the counter to move to the next row by repeating steps from step 6.
9. If any key pressed is found, the [accumulator] content is rotated right through the carry until carry bit sets, while doing this increment the count in the counter till carry is found.
10. Move the content in the counter to display in data field or to memory location
11. To repeat the procedures go to step 2.

Interfacing seven segment display to 8051:

7 segment LED display is very popular and it can display digits from 0 to 9 and quite a few characters like A, b, C, ., H, E, e, F, n, o,t,u,y, etc. Knowledge about how to interface a seven segment display to a micro controller is very essential in designing embedded systems. A seven segment display consists of seven LEDs arranged in the form of a squarish '8' slightly inclined to the right and a single LED as the dot character. Different characters can be displayed by selectively glowing the required LED segments. Seven segment displays are of two types, common cathode and common anode. In common cathode type , the cathode of all LEDs are tied together to a single terminal which is usually labeled as 'com' and the anode of all LEDs are left alone as individual pins labeled as a, b, c, d, e, f, g & h (or dot) . In common anode type, the anode of all LEDs are tied together as a single terminal and cathodes are left alone as individual pins. The pin out scheme and picture of a typical 7 segment LED display is shown in the image below.



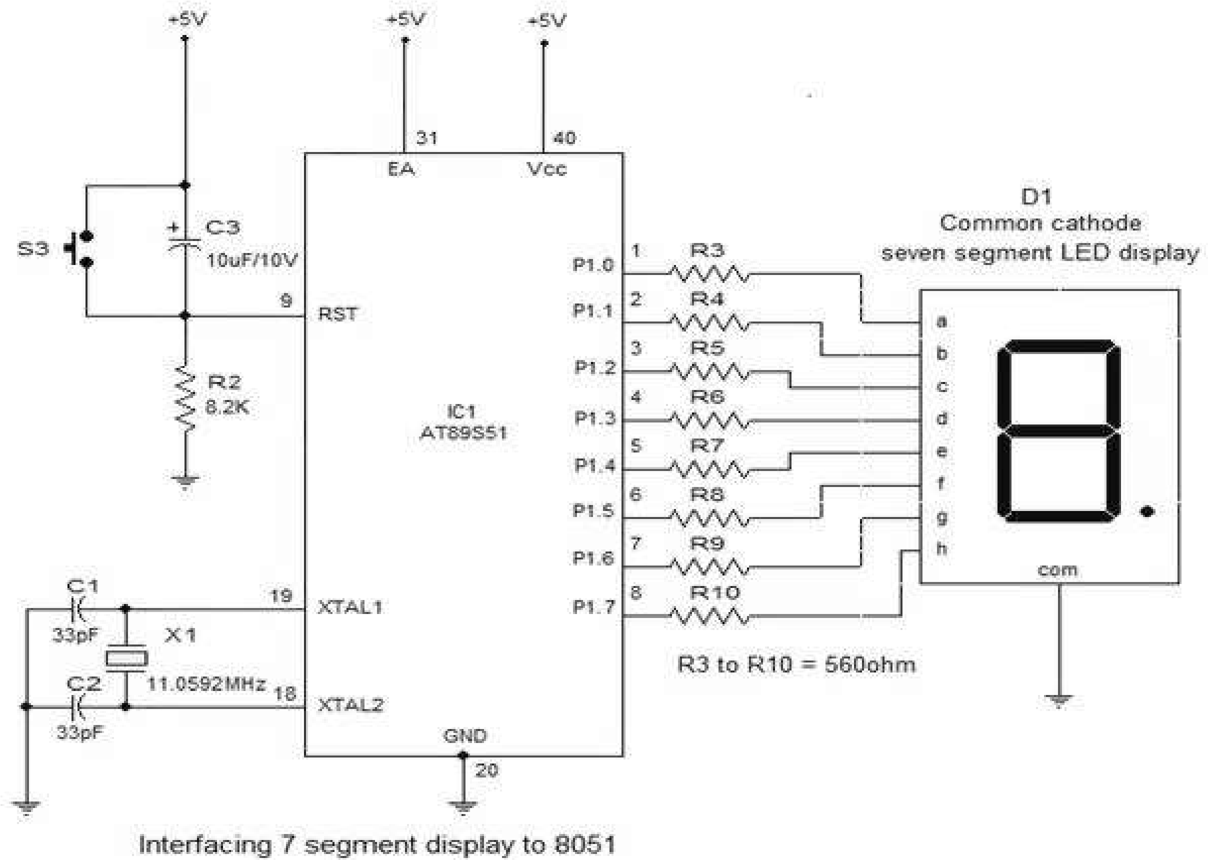


Digit drive pattern.

Digit drive pattern of a seven segment LED display is simply the different logic combinations of its terminals **'a' to 'h'** in order to display different digits and characters. The common digit drive patterns (0 to 9) of a seven segment display are shown in the table below.

Digit	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Interfacing seven segment display to 8051



Interfacing 7 segment display to 8051

The circuit diagram shown above is of an AT89S51 microcontroller based 0 to 9 counter which has a 7 segment LED display interfaced to it in order to display the count. This simple circuit illustrates two things. How to setup simple 0 to 9 up counter using 8051 and more importantly how to interface a seven segment LED display to 8051 in order to display a particular result. The common cathode seven segment display D1 is connected to the Port 1 of the microcontroller (AT89S51) as shown in the circuit diagram. R3 to R10 are current limiting resistors. S3 is the reset switch and R2, C3 forms a de-bouncing circuitry. C1, C2 and X1 are related to the clock circuit. The software part of the project has to do the following tasks.

- Form a 0 to 9 counter with a predetermined delay (around 1/2 second here).
- Convert the current count into digit drive pattern.
- Put the current digit drive pattern into a port for displaying.

All the above said tasks are accomplished by the program given below.

Program.

```
ORG 000H //initial starting address
START: MOV A,#00001001B // initial value of accumulator
        MOV B,A
        MOV R0,#0AH //Register R0 initialized as counter which counts from 10 to 0
LABEL: MOV A,B
        INC A
        MOV B,A
        MOVC A,@A+PC          // adds the byte in A to the program counters address
        MOV P1,A
        ACALL DELAY           // calls the delay of the timer
        DEC R0                //Counter R0 decremented by 1
        MOV A,R0              // R0 moved to accumulator to check if it is zero in next
                              instruction.
        JZ START              //Checks accumulator for zero and jumps to START. Done
                              to check if counting has been finished.
        SJMP LABEL
        DB 3FH                // digit drive pattern for 0
        DB 06H                // digit drive pattern for 1
        DB 5BH                // digit drive pattern for 2
        DB 4FH                // digit drive pattern for 3
        DB 66H                // digit drive pattern for 4
        DB 6DH                // digit drive pattern for 5
        DB 7DH                // digit drive pattern for 6
        DB 07H                // digit drive pattern for 7
        DB 7FH                // digit drive pattern for 8
        DB 6FH                // digit drive pattern for 9
```

```
DELAY: MOV R4,#05H // subroutine for delay
WAIT1: MOV R3,#00H
WAIT2: MOV R2,#00H
WAIT3: DJNZ R2,WAIT3
       DJNZ R3,WAIT2
       DJNZ R4,WAIT1
       RET
END
```

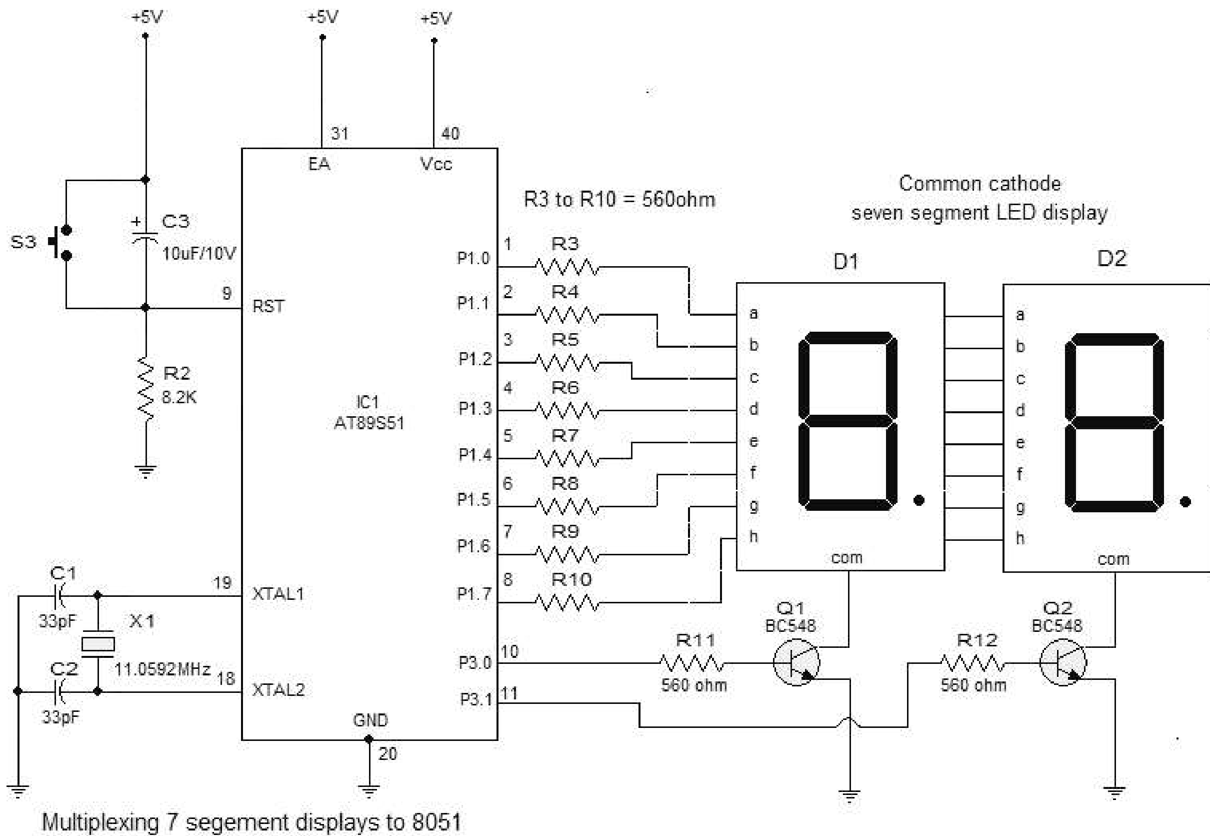
Multiplexing 7 segment displays to 8051.

Suppose you need a three digit display connected to the 8051. Each 7 segment display have 8 pins and so a total amount of 24 pins are to the connected to the microcontroller and there will be only 8 pins left with the microcontroller for other input output applications. Also the maximum number of displays that can be connected to the 8051 is limited to 4 because 8051 has only 4 ports. More over three 3 displays will be ON always and this consumes a considerable amount of power. All these problems associated with the straight forward method can be solved by multiplexing.

In multiplexing all displays are connected in parallel to one port and only one display is allowed to turn ON at a time, for a short period. This cycle is repeated for at a fast rate and due to the persistence of vision of human eye, all digits seems to glow. The main advantages of this method are

- Fewer number of port pins are required .
- Consumes less power.
- More number of display units can be interfaced (maximum 24).

The circuit diagram for multiplexing 2 seven segment displays to the 8051 is shown below.



Multiplexing 7 segement displays to 8051

When assembled and powered on, the circuit will display the number '16' and let us see how it is done. Initially the first display is activated by making P3.0 high and then digit drive pattern for "1" is loaded to the Port 1. This will make the first display to show "1". In the mean time P3.1 will be low and so do the second display will be OFF. This condition is maintained for around 1ms and then P3.0 is made low. Now both displays will be OFF. Then the second display is activated by making P3.1 high and then the digit drive pattern for "6" is loaded to the port 1. This will make the second display to show "6". In the mean time P3.0 will be low and so the second display will be OFF. This condition is maintained for another 1ms and then port 3.1 is made low. This cycle is repeated and due to the persistence of vision you will feel it as "16".

Transistor Q1 drives the first display (D1) and transistor Q2 drives the second display (D2). R11 and R12 are the base current limiting resistors of Q1 and Q2. The purposes of other components are explained in the first circuit.

Program:

```
ORG 000H                        // initial starting address
MOV P1,#00000000B               // clears port 1
MOV R6,#1H                      // stores "1"
MOV R7,#6H                      // stores "6"
MOV P3,#00000000B               // clears port 3
MOV DPTR,#LABEL1                // loads the address of line 29 to DPTR
MAIN: MOV A,R6                  // "1" is moved to accumulator
SETB P3.0                       // activates 1st display
ACALL DISPLAY                   // calls the display sub routine for getting the pattern for "1"
MOV P1,A                        // moves the pattern for "1" into port 1
ACALL DELAY                     // calls the 1ms delay
CLR P3.0                        // deactivates the 1st display
MOV A,R7                        // "2" is moved to accumulator
SETB P3.1                       // activates 2nd display
ACALL DISPLAY                   // calls the display sub routine for getting the pattern for "2"
MOV P1,A                        // moves the pattern for "2" into port 1
ACALL DELAY                     // calls the 1ms delay
CLR P3.1                        // deactivates the 2nd display
SJMP MAIN                       // jumps back to main and cycle is repeated
    DELAY: MOV R3,#02H
      DEL1: MOV R2,#0FAH
      DEL2: DJNZ R2,DEL2
            DJNZ R3,DEL1
            RET
DISPLAY: MOVC A,@A+DPTR      // adds the byte in A to the address in DPTR and loads A with data

                             present in the resultant address
RET


LABEL1: DB 3FH
        DB 06H
        DB 5BH
        DB 4FH
```

```
DB 66H
DB 6DH
DB 7DH
DB 07H
DB 7FH
DB 6FH

END
```

Interfacing A/D and D/A converters:

A/D Converter:-

ADC is the Analog to Digital converter, which converts analog data into digital format; usually it is used to convert **analog voltage** into digital format. Analog signal has infinite no of values like a sine wave or our speech, ADC converts them into particular levels or states, which can be measured in numbers as a physical quantity. Instead of continuous conversion, ADC converts data periodically, which is usually known as sampling rate. **Telephone modem** is one of the examples of ADC, which is used for internet, it converts analog data into digital data, so that computer can understand, because computer can only understand Digital data. The major advantage, of using ADC is that, we noise can be efficiently eliminated from the original signal and digital signal can travel more efficiently than analog one. That's the reason that digital audio is very clear, while listening.

In present time there are lots of microcontrollers in market which has inbuilt ADC with one or more channels. And by using their ADC register we can interface. When we select **8051 microcontroller** family for making any project, in which we need of an ADC conversion, then we use **external ADC**. Some external ADC chips are 0803,0804,0808,0809 and there are many more. Today we are going to interface 8-channel ADC with AT89s52 Microcontroller namely ADC0808/0809.

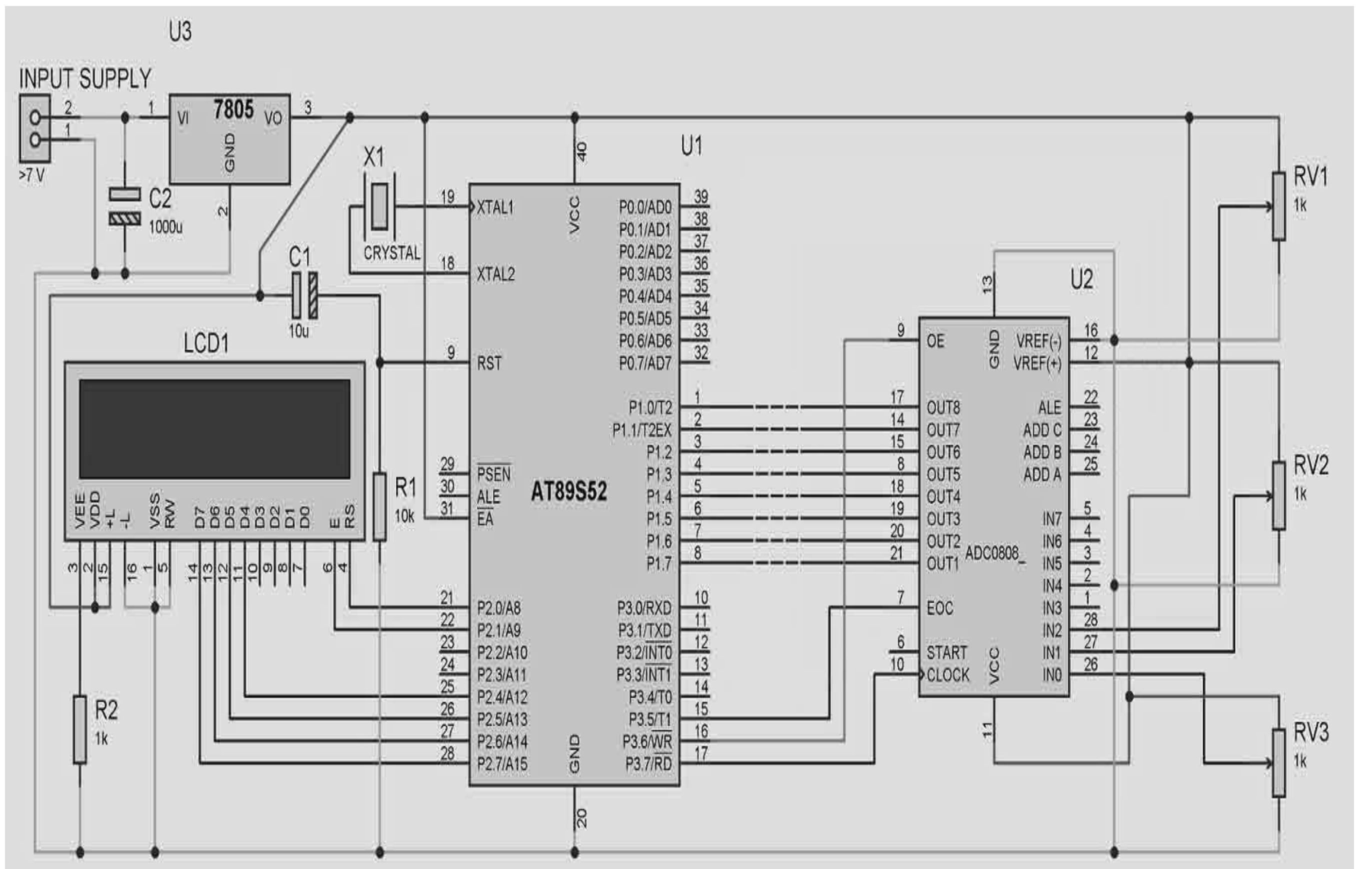
Circuit of **"Interfacing ADC0808 with 8051"** is little complex which contains more connecting wire for connecting device to each other. In this circuit we have mainly used AT89s52 as 8051 microcontroller, ADC0808, Potentiometer and LCD.

A 16x2 LCD is connected with 89s52 microcontroller in 4-bit mode. Control pin RS, RW and En are directly connected to pin P2.0, GND and P2.2. And data pin D4-D7 is connected to pins P2.4, P2.5, P2.6 and P2.7 of 89s52. ADC0808 output pin are directly connected to port P1 of AT89s52. Address line pins ADDA, ADDB, AADC are connected at P3.0, P3.1, and P3.2.

ALE (Address latch enable), SC (Start conversion), EOC (End of conversion), OE (Output enable) and clock pins are connected at P3.3, P3.4, P3.5, P3.6 and P3.7.

And here we have used three potentiometers connected at pin 26, 27, and 28 of ADC0808.

A 9 volt battery and a 5 volt voltage regulator namely 7805 are used for powering the circuit.



D/A Converter:-

The Digital to Analog converter (DAC) is a device, that is widely used for converting digital pulses to analog signals. There are two methods of converting digital signals to analog signals. These two methods are binary weighted method and R/2R ladder method. In this article we will use the MC1408 (DAC0808) Digital to Analog Converter. This chip uses R/2R ladder method. This method can achieve a much higher degree of precision. DACs are judged by its resolution. The resolution is a function of the number of binary inputs. The most common input counts are 8, 10, 12 etc. Number of data inputs decides the resolution of DAC. So if there are n digital input pin, there are 2^n analog levels. So 8 inputs DAC has 256 discrete voltage levels.

The MC1408 DAC (or DAC0808)

In this chip the digital inputs are converted to current. The output current is known as I_{out} by connecting a resistor to the output to convert into voltage. The total current provided by the I_{out} pin is basically a function of the binary numbers at the input pins D_0 - D_7 (D_0 is the LSB and D7 is the MSB) of DAC0808 and the reference current I_{ref}.

The following formula is showing the function of I_{out}

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

The I_{ref} is the input current. This must be provided into the pin 14. Generally 2.0mA is used as I_{ref}

We connect the I_{out} pin to the resistor to convert the current to voltage. But in real life it may cause inaccuracy since the input resistance of the load will also affect the output voltage. So practically I_{ref} current input is isolated by connecting it to an Op-Amp with $R_f = 5K\Omega$ as feedback resistor. The feedback resistor value can be changed as per requirement.

Generating Sine wave using DAC and 8051 Microcontroller

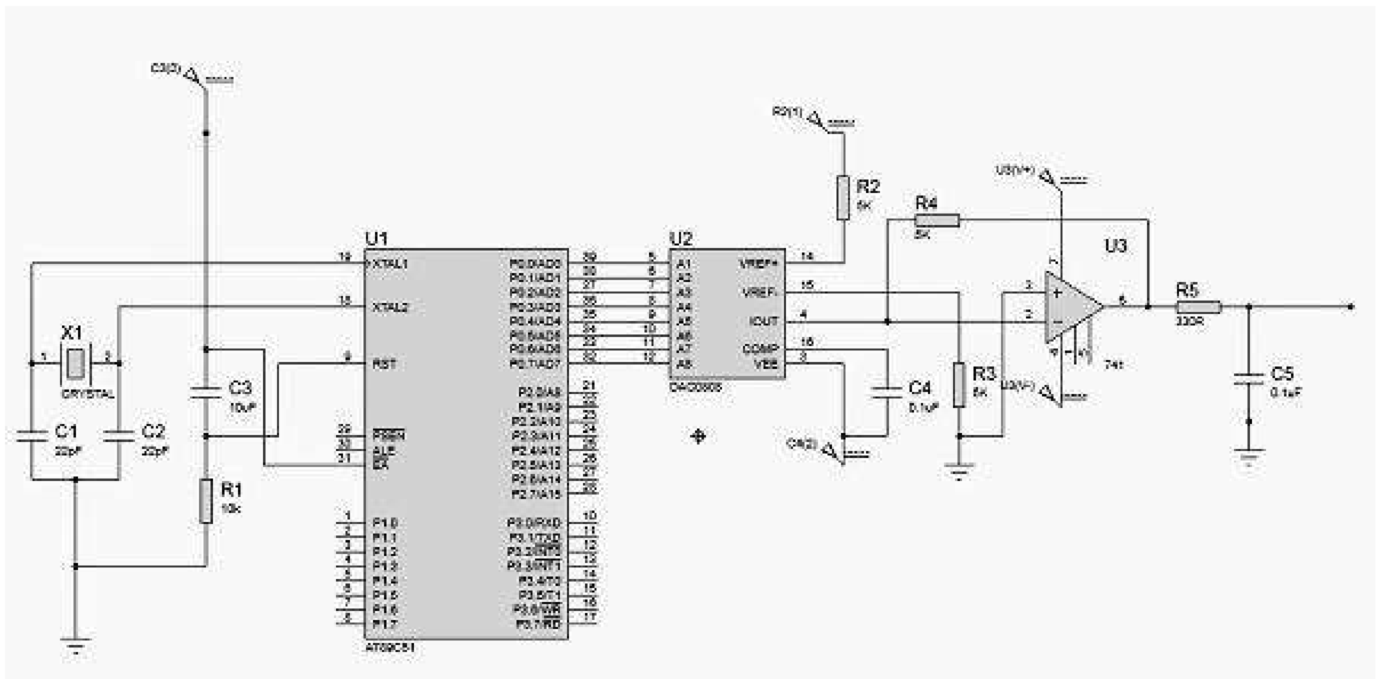
For generating sine wave, at first we need a look-up table to represent the magnitude of the sine value of angles between 0° to 360°. The sine function varies from -1 to +1. In the table only integer values are applicable for DAC input. In this example we will consider 30° increments and calculate the values from degree to DAC input. We are assuming full-scale voltage of 10V for DAC output. We can follow this formula to get the voltage ranges.

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

Let us see the lookup table according to the angle and other parameters for DAC.

Angle(in θ)	sinθ	V_{out} (Voltage Magnitude)	Values sent to DAC
0	0	5	128
30	0.5	7.5	192
60	0.866	9.33	238
90	1.0	10	255
120	0.866	9.33	238
150	0.5	7.5	192
180	0	5	128
210	-0.5	2.5	64
240	-0.866	0.669	17
270	-1.0	0	0

Angle(in θ)	sinθ	V_{out} (Voltage Magnitude)	Values sent to DAC
300	-0.866	0.669	17
330	-0.5	2.5	64
360	0	5	128



UNIT-5

ARM Architectures and Processors :

ARM Architecture, ARM Processor families, ARM Cortex-M Series family, ARM Cortex-M3 processor functional description, functions and interfaces.
Programmers Model: Modes of operation, and execution, Instruction set summary, System address map, write buffer, bit-banding, Processor core register summary, exceptions.
ARM Cortex-M3 programming- Software delay, Programming techniques, Loops, Stack and Stack pointer, subroutines and parameter passing, parallel I/O, Nested vectored Interrupt Controller- functional description, and NVIC Programmer's model.

INTRODUCTION :

ARM was formed in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer, Acorn Computer Group, and VLSI Technology. In 1991, ARM introduced the ARM6 processor family, and VLSI became the initial licensee.

The ARM Cortex™-M3 processor, the first of the Cortex generation of processors released by ARM in 2006, was primarily designed to target the 32-bit microcontroller market. The Cortex-M3 processor provides excellent performance at low gate count and comes with many new features previously avail- able only in high-end processors. The Cortex-M3 addresses the requirements for the 32-bit embedded processor market in the following ways:

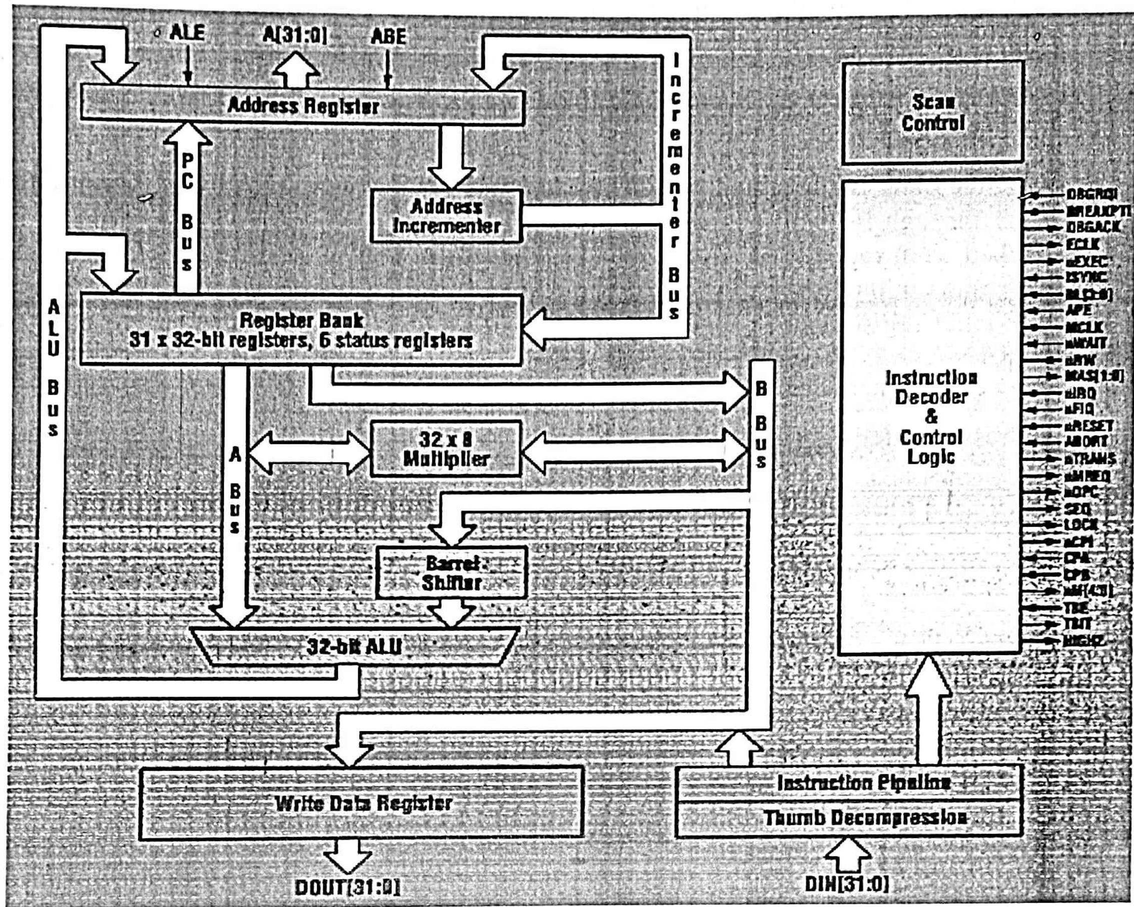
- **Greater performance efficiency**: allowing more work to be done without increasing the frequency or power requirements
- **Low power consumption**: enabling longer battery life, especially critical in portable products including wireless networking applications
- **Enhanced determinism**: guaranteeing that critical tasks and interrupts are serviced as quickly as possible and in a known number of cycles
- **Improved code density**: ensuring that code fits in even the smallest memory footprints
- **Ease of use**: providing easier programmability and debugging for the growing number of 8-bit and 16-bit users migrating to 32 bits
- **Lower cost solutions**: reducing 32-bit-based system costs close to those of legacy 8-bit and 16-bit devices and enabling low-end, 32-bit microcontrollers to be priced at less than US$1 for the first time
- **Wide choice of development tools**: from low-cost or free compilers to full-featured development suites from many development tool vendors

FEATURES OF ARM7 PROCESSOR:

- It is a 32-bit RISC-processor core (32-bit instructions)
- It consists of 37 pieces of 32-bit integer registers. Out of 37 registers 16 registers are readily available for user.
- ARM consists of Pipelined architecture . Fetch, Decode and Execute are the 3 stages of ARM processor.
- It has Cache memory. Cache memory is a small memory placed between processor and main memory to store recently accessed transactions (depending on the implementation).
- Based on memory architecture, ARM processor is a Von Neumann-type bus structure (ARM7), which has a single memory for both code and data.ARM9 has Harvard architecture which has separate memory for code and data.
- ARM processor can able to handle 8 / 16 / 32 -bit data types.
- There are 7 modes of operation namely User,FIQ,IRQ,Supervisor mode, Abort mode. System mode. Undefined mode.
- The architecture is very simple which is reasonably good speed and less power consumption ratio.
- It implements two instruction set, 32-bit ARM instruction set and 16-bit Thumb instruction set.

ARM ARCHITECTURE :

Advanced RISC Machine or Acorn RISC Machine is the architecture with different computing architectures set to be used in different environments. 32-bit and 64-bit can be used here in different computer processors. It was developed by Arm Holdings and the architecture is updated in between. This architecture is specified to be used with CPU, different chips in the system, and in different registers. Reduced Instruction Set Computing helps in creating instructions for the system to be used for several purposes. Smartphones, microcomputers, and embedded devices also use ARM architecture for the instruction set in the registers.



The architecture can be divided into A, R, and M profiles. A profile is mainly for applications, R profile is for real-time and M profile is for Microcontroller. A profile helps to maintain high performance and is designed to run the complex system in Linux or Windows. R profile checks for systems with real-time requirements and is found in networking equipment or embedded control systems. M profile is used in IOT devices and can be synchronized with small and high-power devices.

ARM7TDMI architecture is shown in above figure, which has T=Thumb instruction set, D=Debug Unit, M=MMU(Memory Management Unit) and I=Embedded Trace Core. It is also called as LOAD and STORE architecture since, the processor can access only the registers and not the memory. It consists of main blocks like Address

register, Address incrementer, Register Bank, 32x8 multiplier, Barrel shifter, ALU, Data out and Data in, Instruction decoder and control logic blocks.

1. Instruction Decoder and control logic block is used to decodes the instruction before it is processed by the ALU.
2. Register Bank is connected to ALU by two data paths, one is through A-bus and the other through B-bus it passes Barrel shifter before reaching ALU. The data will be fetched from registers(R0-R15) by ALU for processing which is called LOAD and the results will be stored in to registers after processing which is called STORE.Let, The input registers are denoted by Rm and Rn and the output registers are denoted by Rd.
3. Program counter is used to fetch the address of the next instruction and stored in address register through PC bus.
4. Barrel shifter is also called as a pre-processor.ie, if the data has to be shift left, right or to rotate, it will be done by Barrel shifter before sending it to ALU. ALU and Barrel shifter are called as combinational circuit which will take only one second for the operations.
5. Address incrementer is used to store the next address from where the instruction has to be fetched.

CONTROL SIGNALS OF ARM7 PROCESSOR:

1. **A[31:0]** : This is the 32-bit address bus. **ALE**, **ABE**, and **APE** are used to Control when the address bus is valid.
2. ABE : Address Bus Enable
3. ALE : Address Latch Enable. Address signals have to be maintained high Until the complete duration of memory access cycles.
4. **ABORT** : Memory Abort
5. **APE** : Address Pipeline Enable. This mode is used for DRAM systems.
6. **BL[3:0]** : Byte Latch enable. The values on data bus are latched on the falling Edge, when these are high.
7. **Break Point** : A conditional request for the processor to enter in to Debug state by making this signal HIGH.
8. CPA : Coprocessor Absent. If Coprocessor is able to respond the processor, this signal goes low.
9. CPB : Coprocessor Busy. When the coprocessor is ready to accept the request From processor, this signal goes low.

register, Address incrementer, Register Bank, 32x8 multiplier, Barrel shifter, ALU, Data out and Data in, Instruction decoder and control logic blocks.

1. Instruction Decoder and control logic block is used to decodes the instruction before it is processed by the ALU.
2. Register Bank is connected to ALU by two data paths, one is through A-bus and the other through B-bus it passes Barrel shifter before reaching ALU. The data will be fetched from registers(R0-R15) by ALU for processing which is called LOAD and the results will be stored in to registers after processing which is called STORE.Let, The input registers are denoted by Rm and Rn and the output registers are denoted by Rd.
3. Program counter is used to fetch the address of the next instruction and stored in address register through PC bus.
4. Barrel shifter is also called as a pre-processor.ie, if the data has to be shift left, right or to rotate, it will be done by Barrel shifter before sending it to ALU. ALU and Barrel shifter are called as combinational circuit which will take only one second for the operations.
5. Address incrementer is used to store the next address from where the instruction has to be fetched.

CONTROL SIGNALS OF ARM7 PROCESSOR:

1. **A[31:0]** : This is the 32-bit address bus. **ALE**, **ABE**, and **APE** are used to Control when the address bus is valid.
2. ABE : Address Bus Enable
3. ALE : Address Latch Enable. Address signals have to be maintained high Until the complete duration of memory access cycles.
4. **ABORT** : Memory Abort
5. **APE** : Address Pipeline Enable. This mode is used for DRAM systems.
6. **BL[3:0]** : Byte Latch enable. The values on data bus are latched on the falling Edge, when these are high.
7. **Break Point** : A conditional request for the processor to enter in to Debug state by making this signal HIGH.
8. **CPA** : Coprocessor Absent. If Coprocessor is able to respond the processor, this signal goes low.
9. CPB : Coprocessor Busy. When the coprocessor is ready to accept the request From processor, this signal goes low.

10. DBGACK : Debug Acknowledgement. When the processor is in Debug state, this signal goes high.

11. DBGRQI : Internal Debug Request. This is logical OR of DBGRQ and bit[1] of Debug control register.

12. ECLK : External Clock output.

13. HIGHZ : HIGH Impedance, when HIGH impedance instruction is loaded into the Tap Controller.

14. ISYNC : Synchronous Interrupts. Set this signal high if nIRQ and nFIQ are Synchronous to the processor clock.

15. LOCK : Locked Operation. When the processor is performing a locked memory access.

16. MAS[1:0] :Memory Access Size. It is used to indicate to the memory system the size of data transfer required for both read & write control signals.

17. MCLK : Memory Clock Input. This is the main clock for all memory access and processor operations.

18. nCPI : Not Coprocessor Instruction. When coprocessor instruction are processed, this signal goes low.

19. nFIQ : Not Fast Interrupt Request.

20. nIRQ : Not Interrupt request.

21. nM[4:0] : Not processor mode. Indicates current processor mode.

22. nMREQ : Not Memory Request.

23. nOPC : Not Opcode Fetch

24. nRESET : Not Reset

25. nRW : Not Read Write

26. nTRANS : Not Memory Translate

27. nWAIT : Not Wait

28. SEQ : Sequential Address. When the address of the next memory is close to the last memory

ARM PROCESSOR FAMILY:

Processor Name	Architecture Version	Memory Management Features	Other Features
ARM7TDMI	ARMv4T		
ARM7TDMI-S	ARMv4T		
ARM7EJ-S	ARMv5E		DSP, Jazelle
ARM920T	ARMv4T	MMU	
ARM922T	ARMv4T	MMU	
ARM926EJ-S	ARMv5E	MMU	DSP, Jazelle
ARM946E-S	ARMv5E	MPU	DSP
ARM966E-S	ARMv5E	DSP	
ARM968E-S	ARMv5E		DMA, DSP
ARM966HS	ARMv5E	MPU (optional)	DSP
ARM1020E	ARMv5E	MMU	DSP
ARM1022E	ARMv5E	MMU	DSP
ARM1026EJ-S	ARMv5E	MMU or MPU	DSP, Jazelle
ARM1136J(F)-S	ARMv6	MMU	DSP, Jazelle
ARM1176JZ(F)-S	ARMv6	MMU + TrustZone	DSP, Jazelle
ARM11 MPCore	ARMv6	MMU + multiprocessor cache support	DSP, Jazelle
ARM1156T2(F)-S	ARMv6	MPU	DSP
Cortex-M0	ARMv6-M		NVIC
Cortex-M1	ARMv6-M	FPGA TCM interface	NVIC
Cortex-M3	ARMv7-M	MPU (optional)	NVIC

Table 1.1 ARM Processor Names *Continued*

Processor Name	Architecture Version	Memory Management Features	Other Features
Cortex-R4	ARMv7-R	MPU	DSP
Cortex-R4F	ARMv7-R	MPU	DSP + Floating point
Cortex-A8	ARMv7-A	MMU + TrustZone	DSP, Jazelle, NEON + floating point
Cortex-A9	ARMv7-A	MMU + TrustZone + multiprocessor	DSP, Jazelle, NEON + floating point

CORTEX M33:

The Cortex-M33 core was announced in October 2016 and based on the newer ARMv8-M architecture that was previously announced in November 2015. Conceptually the Cortex-M33 is similar to a cross of Cortex-M4 and Cortex-M23, and also has a 3-stage instruction pipeline.

Optionally it is connected with:

- Floating-Point Unit (FPU): single-precision only IEEE-754 compliant. It is called the FPv5 extension.
- Optional Memory Protection Unit (MPU): 0, 4, 8, 12, 16 regions.
- Optional Security Attribution Unit (SAU): 0, 4, 8 regions.
- Micro Trace Buffer (MTB) (available in M0+/M23/M33/M35P).

CORTEX M35P:

The Cortex-M35P core was announced in May 2018. It is conceptually a Cortex-M33 core with a new instruction cache, plus new tamper-resistant hardware concepts borrowed from the ARM Secure Core family, and configurable parity and ECC features.

CORTEX M55:

The Cortex-M55 core was announced in February 2020 and is based on the Armv8.1-M architecture that was previously announced in February 2019. It also has a 4-stage instruction pipeline. Stack limit boundaries (available only with SAU option).

ARM CORTEX-M SERIES FAMILY:

Arm Core	Cortex M0[2]	Cortex M0+[3]	Cortex M1[4]	Cortex M3[5]	Cortex M4[6]	Cortex M7[7]	Cortex M23[8]	Cortex M33[12]	Cortex M35P	Cortex M55
ARM architecture	ARMv6-M[9]	ARMv6-M[9]	ARMv6-M[9]	ARMv7-M[10]	ARMv7E-M[10]	ARMv7E-M[10]	ARMv8-M Baseline[15]	ARMv8-M Mainline[15]	ARMv8-M Mainline[15]	Armv8.1-M
Computer architecture	Von Neumann	Von Neumann	Von Neumann	Harvard	Harvard	Harvard	Von Neumann	Harvard	Harvard	Harvard
Instruction pipeline	3 stages	2 stages	3 stages	3 stages	3 stages	6 stages	2 stages	3 stages	3 stages	4 to 5 stages
Thumb-1 instructions	Most	Most	Most	Entire	Entire	Entire	Most	Entire	Entire	Entire
Thumb-2 instructions	Some	Some	Some	Entire	Entire	Entire	Some	Entire	Entire	Entire
Multiply instructions 32x32 = 32-bit result	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Multiply instructions 32x32 = 64-bit result	No	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes
Divide instructions 32/32 = 32-bit quotient	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Saturated instructions	No	No	No	Some	Yes	Yes	No	Yes	Yes	Yes
DSP instructions	No	No	No	No	Yes	Yes	No	Optional	Optional	Optional
Single-Precision (SP) Floating-point instructions	No	No	No	No	Optional	Optional	No	Optional	Optional	Optional
Double-Precision (DP) Floating-point instructions	No	No	No	No	No	Optional	No	No	No	Optional
Half-Precisions (HP)	No	No	No	No	No	No	No	No	No	Optional
TrustZone instructions	No	No	No	No	No	No	Optional	Optional	Optional	Optional
Co-processor instructions	No	No	No	No	No	No	No	Optional	Optional	Optional
Helium technology	No	No	No	No	No	No	No	No	No	Optional
Interrupt latency (if zero-wait state RAM)	16 cycles	15 cycles	23 for NMI 26 for IRQ	12 cycles	12 cycles	12 cycles 14 worst case	15 no security ext 27 security ext	12 no security ext ?? security ext		

CORTEX – M0:

The Cortex-M0 core is optimized for small silicon die size and use in the lowest price chips.It consists of an ARM architecture of ARMV6-M, with Von-Nuemann memory architecture.It follows 3 stage instruction pipeline.It uses both Thumb 1(most) and Thumb 2(some) instructions.It takes 16 cycles for Interrupt including Zero-wait state RAM.

CORTEX – M0+ :

The Cortex-M0+ is an optimized superset of the Cortex-M0. The Cortex-M0+ has complete instruction set compatibility with the Cortex-M0 thus allowing the use of the same compiler and debug tools. The Cortex-M0+ pipeline was reduced from 3 to 2 stages, which lowers the power usage. In addition to debug features in the existing

Cortex-M0, a silicon option can be added to the Cortex-M0+ called the Micro Trace Buffer (MTB) which provides a simple instruction trace buffer.

CORTEX M1:

The Cortex-M1 is an optimized core especially designed to be loaded into FPGA chips. ARMv6-M architecture is the ARM architecture which it is using. It has 3-stage pipeline, Both the thumb instructions,Thumb-1 (most), is used except CBZ, CBNZ, IT and some of the Thumb-2 are used, only BL, DMB, DSB, ISB, MRS, MSR are used.It consists of 32-bit hardware integer multiply with 32-bit result.The Tightly-Coupled Memory (TCM): 0 to 1 MB instruction-TCM, 0 to 1 MB data-TCM, each with optional ECC(Elliptic Curve Cryptography).

CORTEX M3:

ARMv7-M architecture is used in CORTEX M3. It has 3-stage pipeline with branch speculation. Thumb-1 and Thumb-2 instructions are used entirely . If 32-bit hardware integer multiply with 32-bit or 64-bit it results in signed or unsigned, add or subtract after the multiply. 32-bit multiply is 1 cycle, but 64-bit multiply and MAC instructions require extra cycles.32-bit hardware integer divided into (2–12 cycles). It has Optional Memory Protection Unit (MPU): 0 or 8 regions.There are 1 to 241 interrupts including NMI. It requires 12 cycle interrupt latency.

CORTEX M4:

Conceptually the Cortex-M4 is a Cortex-M3 plus DSP instructions, and optional floating-point unit (FPU). A core with an FPU is known as Cortex-M4F. It uses ARMv7E-M architecture, 3-stage pipeline with branch speculation.Thumb-1 and Thumb-2 instructions used entirely. If 32-bit hardware integer multiply with 32-bit or 64-bit it results in signed or unsigned, add or subtract after the multiply. For 32-bit Multiply and MAC(Media Access Control) address are 1 cycle and for 32-bit hardware integer divide it takes 2–12 cycles.Saturation arithmetic support is available in which all operations such as addition and multiplication are limited to a fixed range between a minimum and maximum value. For DSP extension: Single cycle 16/32-bit MAC, single cycle dual 16-bit MAC, 8/16-bit SIMD arithmetic is used.1 to 241 interrupts including NMI is used.It has 12 cycle interrupt latency. Integrated sleep modes are possible. Optional floating-point unit (FPU): single-precision only IEEE-754 compliant. It is called the FPv4-SP extension.Optional memory protection unit (MPU): 0 or 8 regions.

CORTEX M7:

The Cortex-M7 is a high-performance core with almost double the power efficiency of the older Cortex-M4. It features a 6-stage superscalar pipeline with branch prediction and an optional floating-point unit capable of single-precision and optionally double-precision operations. The instruction and data buses have been enlarged to 64-bit wide over the previous 32-bit buses. If a core contains an FPU, it is known as a Cortex-M7F, otherwise it is a Cortex-M7. Entire Thumb 1 and Thumb 2 instructions will be used. It follows the same specifications of CORTEX M4 for multiplication,division, saturated arithmetic, interrupts and interrupt latency etc.

Optionally it is connected with:

1. Floating-point unit (FPU): (single precision) or (single and double-precision), both IEEE-754-2008 compliant. It is called the FPv5 extension.
2. CPU cache: 0 to 64 KB instruction-cache, 0 to 64 KB data-cache, each with optional ECC.
3. Tightly-Coupled Memory (TCM): 0 to 16 MB instruction-TCM, 0 to 16 MB data-TCM, each with optional ECC.
4. Memory Protection Unit (MPU): 8 or 16 regions.
5. Embedded Trace Macrocell (ETM): instruction-only, or instruction and data.
6. Retention Mode (with Arm Power Management Kit) for Sleep Modes.

CORTEX M23:

The Cortex-M23 core was announced in October 2016 and based on the newer ARMv8-M architecture that was previously announced in November 2015. Conceptually the Cortex-M23 is similar to a Cortex-M0+ plus integer divide instructions and Trust Zone security features, and also has a 2-stage instruction pipeline. For 32-bit hardware integer divide it takes 17 or 34 cycles. It is much slower than divide in all other cores. Stack limit boundaries are available only with SAU option(available in M23/M33/M35P).

Optionally it is connected with:

1. Memory Protection Unit (MPU): 0, 4, 8, 12, 16 regions.
2. Optional Security Attribution Unit (SAU): 0, 4, 8 regions.
3. Single-cycle I/O port (available in M0+/M23).
4. Micro Trace Buffer (MTB) (available in M0+/M23/M33/M35P).

CORTEX M33:

The Cortex-M33 core was announced in October 2016 and based on the newer ARMv8-M architecture that was previously announced in November 2015. Conceptually the Cortex-M33 is similar to a cross of Cortex-M4 and Cortex-M23, and also has a 3-stage instruction pipeline.

Optionally it is connected with:

- Floating-Point Unit (FPU): single-precision only IEEE-754 compliant. It is called the FPv5 extension.
- Optional Memory Protection Unit (MPU): 0, 4, 8, 12, 16 regions.
- Optional Security Attribution Unit (SAU): 0, 4, 8 regions.
- Micro Trace Buffer (MTB) (available in M0+/M23/M33/M35P).

CORTEX M35P:

The Cortex-M35P core was announced in May 2018. It is conceptually a Cortex-M33 core with a new instruction cache, plus new tamper-resistant hardware concepts borrowed from the ARM Secure Core family, and configurable parity and ECC features.

CORTEX M55:

The Cortex-M55 core was announced in February 2020 and is based on the Armv8.1-M architecture that was previously announced in February 2019. It also has a 4-stage instruction pipeline. Stack limit boundaries (available only with SAU option).

The WIC is not programmable, and does not have any registers or user interface. It operates entirely from hardware signals.

NESTED VECTORED INTERRUPT CONTROLLER :

Nested vector interrupt control (NVIC) is a method of prioritizing interrupts, improving the MCU's performance and reducing interrupt latency.

The term "nested" refers to the fact that in NVIC, a number of interrupts (up to several hundred in some processors) can be defined, and each interrupt is assigned a priority, with "0" being the highest priority.

BREAKPOINT UNIT:

A breakpoint enables you to interrupt your application when execution reaches a specific address. When execution reaches the breakpoint, normal execution stops before any instruction stored there is executed.

Types of breakpoints:

- Software breakpoints stop your program when execution reaches a specific address. Software breakpoints are implemented by the debugger replacing the instruction at the breakpoint address with a special instruction. Software breakpoints can only be set in RAM.

- Hardware breakpoints use special processor hardware to interrupt application execution. Hardware breakpoints are a limited resource.

MEMORY PROTECTION UNIT:

The MPU is an optional component for memory protection. The processor supports the standard ARMv7 *Protected Memory System Architecture* model. The MPU provides full support for:

- protection regions
- overlapping protection regions, with ascending region priority:
  - 7 = highest priority
  - 0 = lowest priority.
- access permissions
- exporting memory attributes to the system.

FLOATING POINT UNIT :

It is present only for Cortex-M4 Processor to perform several floating point arithmetic operations.

WATCH POINT UNIT :

A watch point is similar to a breakpoint, but it is the address of a data access that is monitored rather than an instruction being executed. Watch points are sometimes known as data breakpoints, emphasizing that they are data dependent. Execution of application stops when the address being monitored is accessed by your application. It can be set read, write, or read/write watch points.

PIPELINING:

ARM CORTEX M3 consists of 3 stage instruction pipelining architecture. They are Fetch, Decode and Execute.

INTERFACES :

There are several bus interfaces on the Cortex-M3 processor. They allow the Cortex-M3 to carry instruction fetches and data accesses at the same time. The main bus interfaces are as follows: • Code memory buses • System bus • Private peripheral bus The code memory region access is carried out on the code memory buses, which physically consist of two buses, one called I-Code and other called D-Code. These are optimized for instruction fetches for best instruction execution speed. The system bus is used to access memory and peripherals. This provides access to the Static Random Access Memory (SRAM), peripherals, external RAM, external devices, and part of the system level memory regions.

The private peripheral bus provides access to a part of the system-level memory dedicated to private peripherals, such as debugging components.

MODES OF OPERATION :

The Cortex-M3 processor has two modes and two privilege levels. The operation modes are given as,

1. Thread Mode
2. Handler Mode

In Thread mode,the processor works in normal mode. In Handler mode, the processor runs in exception handler like an interrupt handler system.

There are two privilege levels which provide a mechanism for safeguarding memory accesses to critical regions as well as providing a basic security model. They are :

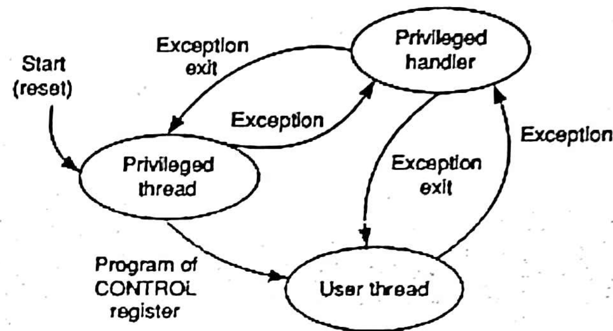
1. Privileged Level
2. User Level

When the processor is running a main program (thread mode), it can be either

in a privileged state or a user state, but exception handlers can only be in a privileged state.

When the processor exits reset, it is in thread mode, with privileged access rights. In the privileged state, a program has access to all memory ranges (except when prohibited by MPU settings) and can use all supported instructions.

Software in the privileged access level can switch the program into the user access level using the control register. When an exception takes place, the processor will always switch back to the privileged state and return to the previous state when exiting the exception handler.



A user program cannot change back to the privileged state by writing to the control register. It has to go through an exception handler that programs the control register to switch the processor back into the privileged access level when returning to thread mode.

The support of privileged and user access levels provides a more secure and robust architecture. For example, when a user program goes wrong, it will not be able to corrupt control registers in the Nested Vectored Interrupt Controller (NVIC). In addition, if the Memory Protection Unit (MPU) is present, it is possible to block user programs from accessing memory regions used by privileged processes.

The user application stack and the kernel stack memory can be separated to avoid the possibility of crashing a system caused by stack operation errors in user programs. With this arrangement, the user program (running in thread mode) uses the PSP, and the exception handlers use the MSP. The switching of SPs is automatic upon entering or leaving the exception handlers

	Privileged	User
When running an exception handler	Handler mode	
When not running an exception handler (e.g., main program)	Thread mode	Thread mode

The mode and access level of the processor are defined by the control register. When the control register bit 0 is 0, the processor mode changes when an exception takes place.

When control register bit 0 is 1 (thread running user application), both processor mode and access level change when an exception takes place.

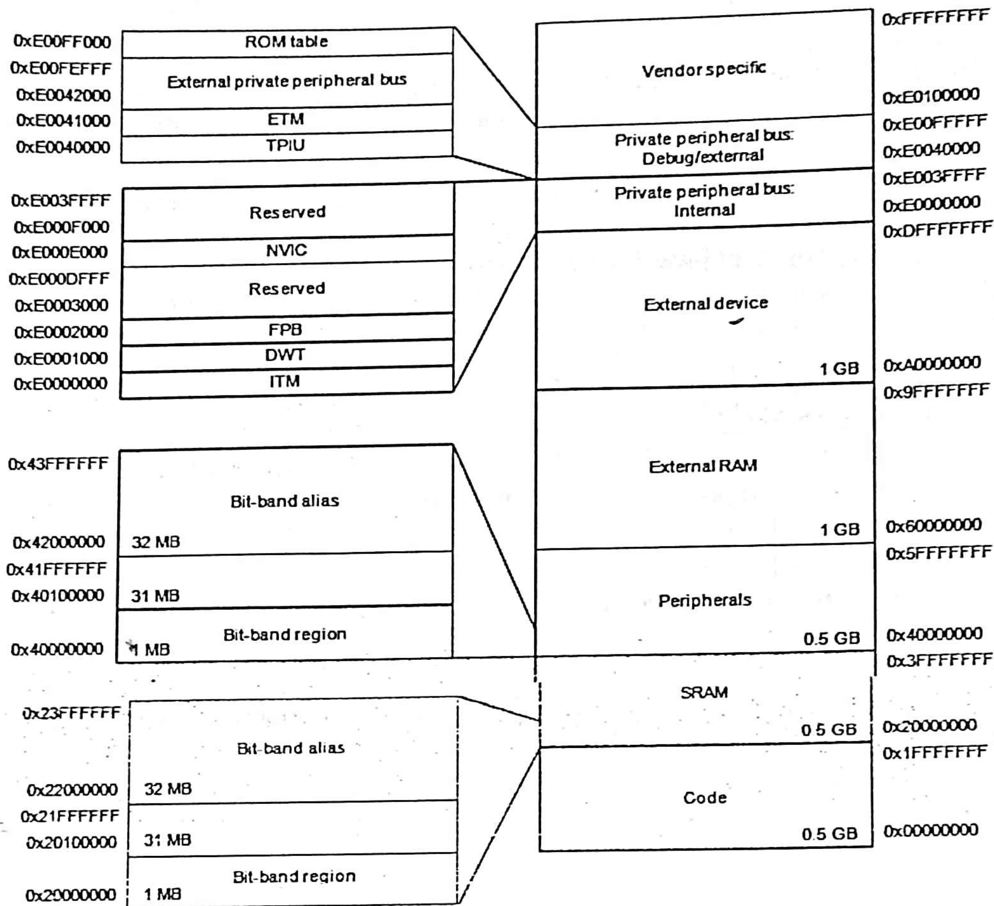
Control register bit 0 is programmable only in the privileged level. For a user-level program to switch to privileged state, it has to raise an interrupt (for example, supervisor call [SVC]) and write to CONTROL within the handler.

SYSTEM ADDRESS MAP :

The Cortex-M3 processor has a fixed memory map. This makes it easier to port soft- ware from one Cortex-M3 product to another. The Nested Vectored Interrupt Controller (NVIC) and Memory Protection Unit (MPU), have the same memory locations in all Cortex-M3 products. Nevertheless, the memory map definition allows great flexibility so that manufacturers can differentiate their Cortex-M3-based product from others.

Some of the memory locations are allocated for private peripherals such as debugging components. They are located in the private peripheral memory region. These debugging components include the following:

- Fetch Patch and Breakpoint Unit (FPB)
- Data Watch point and Trace Unit (DWT)
- Instrumentation Trace Macro cell (ITM)
- Embedded Trace Macro cell (ETM)
- Trace Port Interface Unit (TPIU)
- ROM table



The Cortex-M3 processor has a total of 4 GB of address space. Program code can be located in the code region, the Static Random Access Memory (SRAM) region, or the external RAM region. However, it is best to put the program code in the code region because with this arrangement, the instruction fetches and data accesses are carried out simultaneously on two separate bus interfaces.

The SRAM memory range is for connecting internal SRAM. Access to this region is carried out via the system interface bus. a 32-MB range is defined as a bit-band alias. Within the 32-bit-band alias memory range, each word address represents a single bit in the 1-MB bit-band region. A data write access to this bit-band alias memory range will be converted to an atomic READ- MODIFY-WRITE operation to the bit-band region so as to allow a program to set or clear individual data bits in the memory. The bit-band operation applies only to data accesses not instruction fetches. By putting Boolean information (single bits) in the bit-band region, it is possible to pack multiple Boolean data in a single word while still allowing them to be accessible individually via bit-band alias, thus saving memory space without the need for handling READ-MODIFY-WRITE in software.

Another 0.5-GB block of address range is allocated to on-chip peripherals. Similar to the SRAM region, this region supports bit-band alias and is accessed via the system bus interface. However, instruction execution in this region is not allowed. The bit-band support in the peripheral region makes it easy to access or change control and status bits of peripherals, making it easier to program peripheral control.

Two slots of 1-GB memory space are allocated for external RAM and external devices. The difference between the two is that program execution in the external device region is not allowed, and there are some differences with the caching behaviors.

The last 0.5-GB memory is for the system-level components, internal peripheral buses, external peripheral bus, and vendor-specific system peripherals. There are two segments of the private peripheral bus (PPB):

- Advanced High-Performance Bus (AHB) PPB, for Cortex-M3 internal AHB peripherals only; this includes NVIC, FPB, DWT, and ITM
- Advance Peripheral Bus (APB) PPB, for Cortex-M3 internal APB devices as well as external peripherals (external to the Cortex-M3 processor); the Cortex-M3 allows chip vendors to add additional on-chip APB peripherals on this private peripheral bus via an APB interface

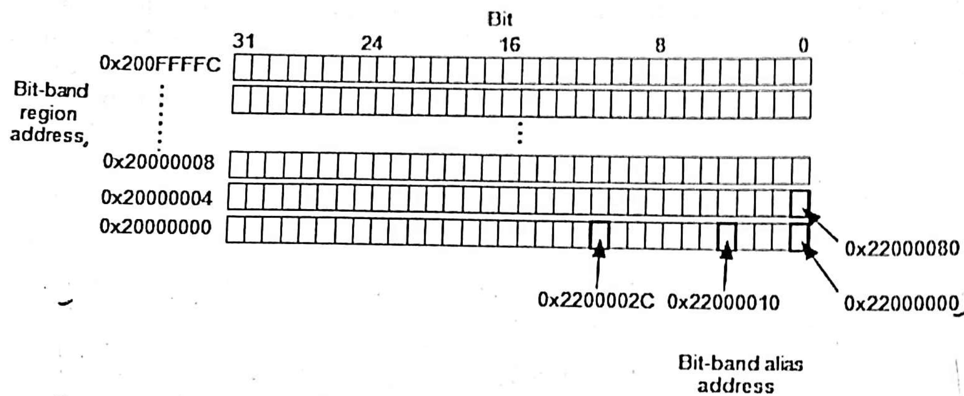
The NVIC is located in a memory region called the system control space (SCS). Besides providing interrupt control features, this region also provides the control registers for SYS- TICK, MPU, and code debugging control.

The remaining unused vendor-specific memory range can be accessed via the system bus interface.
However, instruction execution in this region is not allowed.
The Cortex-M3 processor also comes with an optional MPU. Chip manufacturers can decide whether to include the MPU in their products.

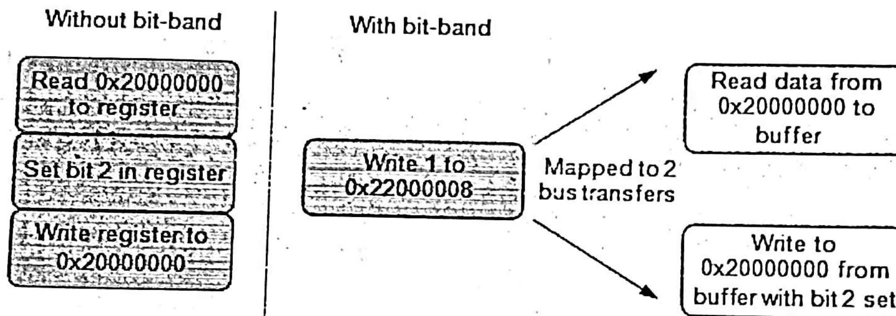
BIT – BAND OPERATIONS:

Bit-band operation support allows a single load/store operation to access (read/write) to a single data bit. In the Cortex-M3, this is supported in two predefined memory regions called bit-band regions. One of them is located in the first 1 MB of the SRAM region, and the other is located in the first 1 MB of the peripheral region.

These two memory regions can be accessed like normal memory, but they can also be accessed via a separate memory region called the bit-band alias. When the bit-band alias address is used, each individual bit can be accessed separately in the least significant bit (LSB) of each word-aligned address.



For example, to set bit 2 in word data in address 0x20000000, instead of using three instructions to read the data, set the bit, and then write back the result, this task can be carried out by a single instruction.



The assembler sequence for these two cases :

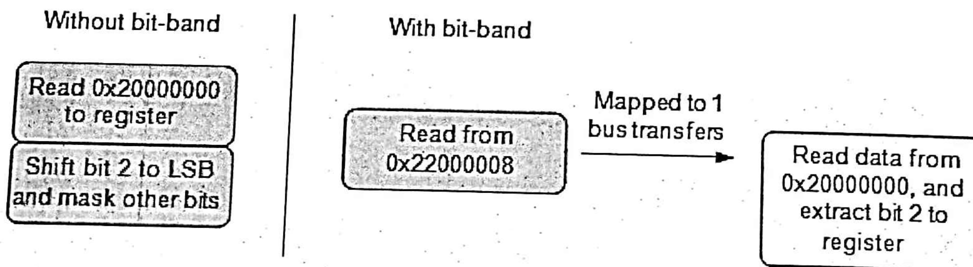
Without bit-band

```
LDR    R0,=0x20000000 ; Setup address
LDR    R1, [R0]       ; Read
ORR.W  R1, #0x4       ; Modify bit
STR    R1, [R0]       ; Write back result
```

With bit-band

```
LDR    R0, = 0x22000008 ; Setup add
MOV    R1, #1           ; Setup dat
STR    R1, [R0]         ; Write
```

Similarly, bit-band support can simplify application code if we need to read a bit in a memory location. For example, if we need to determine bit 2 of address 0x20000000,



THE BUILT-IN NESTED VECTORED INTERRUPT CONTROLLER:

The Cortex-M3 processor includes an interrupt controller called the Nested Vectored Interrupt Controller (NVIC). It is closely coupled to the processor core and provides a number of features as follows:

- Nested interrupt support
- Vectored interrupt support
- Dynamic priority changes support
- Reduction of interrupt latency
- Interrupt masking

Nested Interrupt Support:

The NVIC provides nested interrupt support. All the external interrupts and most of the system exceptions can be programmed to different priority levels. When an interrupt occurs, the NVIC compares the priority of this interrupt to the current running priority level. If the priority of the new interrupt is higher than the current level, the interrupt handler of the new interrupt will override the current running task.

Vectored Interrupt Support :

The Cortex-M3 processor has vectored interrupt support. When an interrupt is accepted, the starting address of the interrupt service routine (ISR) is located from a vector table in memory. There is no need to use software to determine and branch to the starting address of the ISR. Thus, it takes less time to process the interrupt request.

Dynamic Priority Changes Support:

Priority levels of interrupts can be changed by software during run time. Interrupts that are being serviced are blocked from further activation until the ISR is completed, so their priority can be changed without risk of accidental re entry.

Reduction of Interrupt Latency:

The Cortex-M3 processor also includes a number of advanced features to lower the interrupt latency. These include automatic saving and restoring some register contents, reducing delay in switching from one ISR to another, and handling of late arrival interrupts.

Interrupt Masking:

Interrupts and system exceptions can be masked based on their priority level or masked completely using the interrupt masking registers BASEPRI, PRIMASK, and

FAULTMASK. They can be used to ensure that time-critical tasks can be finished on time without being interrupted.

EXCEPTIONS :

The Cortex-M3 supports a number of exceptions, including a fixed number of system exceptions and a number of interrupts, commonly called IRQ. The number of interrupt inputs on a Cortex-M3 microcontroller depends on the individual design. Interrupts generated by peripherals, except System Tick Timer, are also connected to the interrupt input signals. The typical number of interrupt inputs is 16 or 32. However, you might find some microcontroller designs with more (or fewer) interrupt inputs. Besides the interrupt inputs, there is also a non maskable interrupt (NMI) input signal. The actual use of NMI depends on the design of the microcontroller or system-on-chip (SoC) product you use. In most cases, the NMI could be connected to a watchdog timer or a voltage-monitoring block that warns the processor when the voltage drops below a certain level. The NMI exception can be activated any time, even right after the core exits reset.

The list of exceptions found in the Cortex-M3 is shown:

Exception Number	Exception Type	Priority	Function
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt
3	Hard fault	−1	All classes of fault, when the corresponding fault handler cannot be activated because it is currently disabled or masked by exception masking
4	MemManage	Settable	Memory management fault; caused by MPU violation or invalid accesses (such as an instruction fetch from a nonexecutable region)
5	Bus fault	Settable	Error response received from the bus system; caused by an instruction prefetch abort or data access error
6	Usage fault	Settable	Usage fault; typical causes are invalid instructions or invalid state transition attempts (such as trying to switch to ARM state in the Cortex-M3)
7–10	—	—	Reserved
11	SVC	Settable	Supervisor call via SVC instruction
12	Debug monitor	Settable	Debug monitor
13	—	—	Reserved
14	PendSV	Settable	Pendable request for system service
15	SYSTICK	Settable	System tick timer
16–255	IRQ	Settable	IRQ input #0–239

STACK MEMORY OPERATIONS :

In the Cortex-M3, besides normal software-controlled stack PUSH and POP, the stack PUSH and POP operations are also carried out automatically when entering or exiting an exception/interrupt handler.

Basic Operations of the Stack :

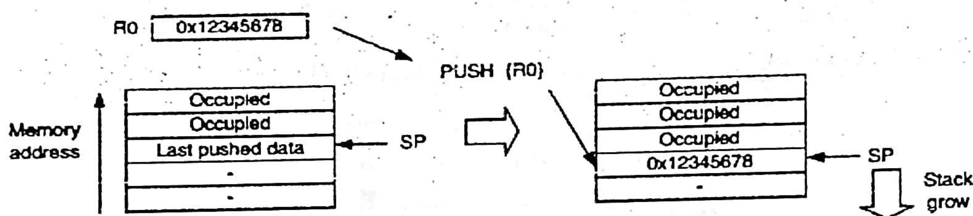
The SP is adjusted automatically in PUSH and POP so that multiple data PUSH will not cause old stacked data to be erased.

The function of the stack is to store register contents in memory so that they can be restored later, after a processing task is completed. For normal uses, for each store (PUSH), there must be a corresponding read (POP), and the address of the POP operation should match that of the PUSH operation. When PUSH/POP instructions are used, the SP is incremented/decremented automatically.

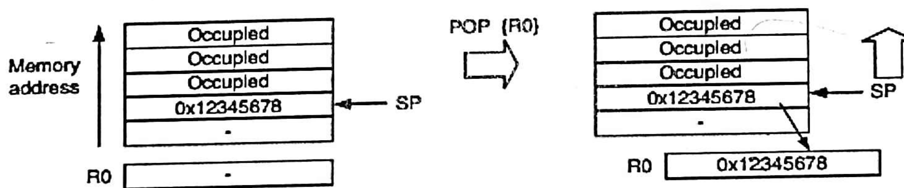
When program control returns to the main program, the R0–R2 contents are the same as before. Notice the order of PUSH and POP: The POP order must be the reverse of PUSH.

Cortex-M3 Stack Implementation:

The Cortex-M3 uses a full-descending stack operation model. The SP points to the last data pushed to the stack memory, and the SP decrements before a new PUSH operation.



Cortex-M3 Stack PUSH Implementation



Cortex-M3 Stack POP Implementation

For POP operations, the data is read from the memory location pointer by SP, and then, the SP is incremented. The contents in the memory location are unchanged but will be overwritten when the next PUSH operation takes place.

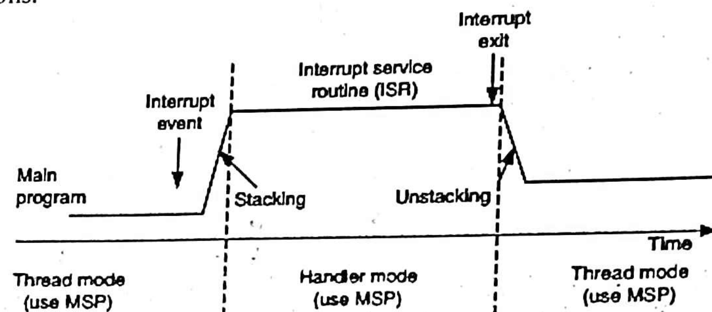
Because each PUSH/POP operation transfers 4 bytes of data (each register contains 1 word, or 4 bytes), the SP decrements/increments by 4 at a time or a multiple of 4 if more than 1 register is pushed or popped.

The Two-Stack Model in the Cortex-M3 :

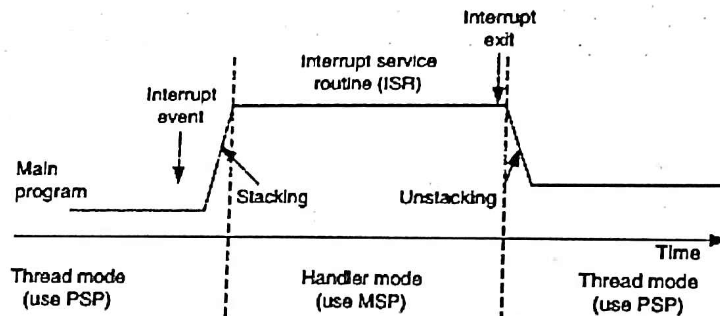
The Cortex-M3 has two SPs: 1. SP_main and 2. SP_Process. The SP register to be used is controlled by the control register bit 1.

When CONTROL[1] is 0, the SP_main is used for both thread mode and handler mode. In this arrangement, the main program and the exception handlers share the same stack memory region. This is the default setting after power-up.

When the CONTROL[1] is 1, the SP_Process is used in thread mode. In this arrangement, the main program and the exception handler can have separate stack memory regions.



CONTROL[1]=0: Both Thread Level and Handler Use Main Stack.



CONTROL[1]=1: Thread Level Uses Process Stack and Handler Uses Main Stack.

INSTRUCTION SET :

The processor does not support ARM instructions.Thumb-2 instruction set, it is now possible to handle all processing requirements in one operation state. Even interrupts are now handled with the Thumb state.

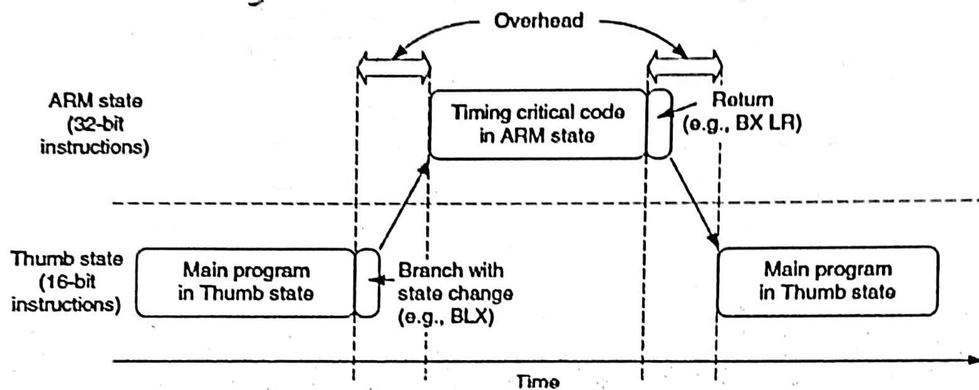
Since there is no need to switch between states, the Cortex-M3 processor has a number of advantages over traditional ARM processors,

such as:

♣ No state switching overhead, saving both execution time and instruction space

♣ No need to separate ARM code and Thumb code source files, making software development and maintenance easier

♣ It's easier to get the best efficiency and performance, in turn making it easier to write software, because there is no need to worry about switching code between ARM and Thumb to try to get the best density/performance.



Instruction are mentioned as follows:

cond	Mnemonic extension	Meaning, integer arithmetic	Meaning, floating-point arithmetic[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS[b]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC[c]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL)[d]	Always (unconditional)	Always (unconditional)	Any

Memory Access Instructions

Offset, Pre-Index, and Post-Index

IA, IB, DA, DB

!

Mnemonic	Brief Description
ADR	Load PC-relative address
CLREX	Clear exclusive
LDM{mode}	Load multiple registers
LDR{type}	Load register using immediate offset
LDR{type}	Load register using register offset
LDR{type}T	Load register with unprivileged access
LDR{type}	Load register using PC-relative address
LDRD	Load register using PC-relative address (two words)
LDREX{type}	Load register exclusive
POP	Pop registers from stack
PUSH	Push registers onto stack
STM{mode}	Store multiple registers
STR{type}	Store register using immediate offset
STR{type}	Store register using register offset
STR{type}T	Store register with unprivileged access
STREX{type}	Store register exclusive

General Data Processing Instructions

If S is specified, these instructions update the N, Z, C & V flags according to the result.

Mnemonic	Brief Description
ADC	Add with carry
ADD	Add
ADDW	Add
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear
CLZ	Count leading zeros
CMN	Compare negative
CMP	Compare
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MOV	Move
MOVT	Move top
MOVW	Move 16-bit constant
MVN	Move NOT
ORN	Logical OR NOT
ORR	Logical OR
RBIT	Reverse bits
REV	Reverse byte order in a word
REV16	Reverse byte order in each halfword
REVSH	Reverse byte order in bottom halfword and sign extend
ROR	Rotate right
RRX	Rotate right with extend
RSB	Reverse subtract
SBC	Subtract with carry
SUB	Subtract
SUBW	Subtract
TEQ	Test equivalence
TST	Test

Multiply and Divide Instructions

Mnemonic	Brief Description
MLA	Multiply with accumulate, 32-bit result
MLS	Multiply and subtract, 32-bit result
MUL	Multiply, 32-bit result
SDIV	Signed divide
SMLAL	Signed multiply with accumulate (32x32+64), 64-bit result
SMULL	Signed multiply (32x32), 64-bit result
UDIV	Unsigned divide
UMLAL	Unsigned multiply with accumulate (32x32+64), 64-bit result
UMULL	Unsigned multiply (32x32), 64-bit result

Saturating Instructions

Mnemonic	Brief Description
SSAT	Signed saturate
USAT	Unsigned saturate

Bitfield Instructions

Mnemonic	Brief Description
BFC	Bit field clear
BFI	Bit field insert
SBFX	Signed bit field extract
SXTB	Sign extend a byte
SXTH	Sign extend a halfword
UBFX	Unsigned bit field extract
UXTB	Zero extend a byte
UXTH	Zero extend a halfword

Branch and Control Instructions

Mnemonic	Brief Description
B	Branch
BL	Branch with iink
BLX	Branch indirect with link
BX	Branch indirect
CBNZ	Compare and branch if non-zero
CBZ	Compare and branch if zero
IT	If-Then
TBB	Table branch byte
TBH	Table branch halfword

Miscellaneous Instructions

Mnemonic	Brief Description
BKPT	Breakpoint
CPSID	Change processor state, disable interrupts
CPSIE	Change processor state, enable interrupts
DMB	Data memory barrier
DSB	Data synchronization barrier
ISB	Instruction synchronization barrier
MRS	Move from special register to register
MSR	Move from register to special register
NOP	No operation
SEV	Send event
SVC	Supervisor call
WFE	Wait for event
WFI	Wait for interrupt

16-BIT THUMB INSTRUCTION SET :

Some of the changes used to reduce the length of the instructions from 32 bits to 16 bits:

- ♣ reduce the number of bits used to identify the register o less number of registers can be used
- ♣ reduce the number of bits used for the immediate value o smaller number range
- ♣ remove options such as 'S' o make it default for some instructions
- ♣ remove conditional fields (N, Z, V, C)
- ♣ no conditional executions (except branch)
- ♣ remove the optional shift (and no barrel shifter operation o introduce dedicated shift instructions
- ♣ remove some of the instructions o more restricted coding

ASSEMBLER	OPERATION
LDR <Rd>, [<Rn>, <Rm>]	Load memory word from base register address + register offset
LDRB <Rd>, [<Rn>, <Rm>]	Load memory byte [7:0] from register address + register offset
LDRH <Rd>, [<Rn>, <Rm>]	Load halfword [15:0] from register address + register offset
LDRSB <Rd>, [<Rn>, <Rm>]	Load signed byte [7:0] from register address + register offset
LDRSH <Rd>, [<Rn>, <Rm>]	Load signed halfword [15:0] from register address + register offset
STR <Rd>, [<Rn>, <Rm>]	Store register word to register address
STRB <Rd>, [<Rn>, <Rm>]	Store register byte [7:0] to register address
STRH <Rd>, [<Rn>, <Rm>]	Store register halfword [15:0] to register address + register offset
LDMIA <Rn>!, <registers>	Multiple sequential memory word loads
STMIA <Rn>!, <registers>	Store multiple register words to sequential memory locations
PUSH <registers>	Push registers onto stack
POP <registers>	Pop registers from stack

ASSEMBLER	OPERATION
B <target_address>	Branch unconditional
B<cond> <target address>	Branch conditional
BL <Rm>	Branch with link
BLX <Rm>	Branch with link and exchange
BKPT <immed_8>	Software breakpoint
CBZ <Rn>,<label>	Compare zero and branch
CBNZ <Rn>,<label>	Compare not zero and branch
CPS <effect>, <iflags>	Change processor state
CPY <Rd> <Rm>	Copy high or low register value to another high or low register
IT <cond> IT<x> <cond> IT<x><y> <cond> IT<x><y><z> <cond>	Condition the following instruction, Condition the following two instructions, Condition the following three instructions, Condition the following four instructions
SEV <c>	Send event
WFE <c>	Wait for event
WFI <c>	Wait for interrupt

REGISTERS:

The processor has the following 32-bit registers:

- 13 general-purpose registers, r0-r12
- stack point alias of banked registers, SP_process and SP_main
- link register, r14
- program counter, r15
- one program status register, xPSR.

Name		Functions (and banked registers)	
R0		General-purpose register	Low registers
R1		General-purpose register	
R2		General-purpose register	
R3		General-purpose register	
R4		General-purpose register	
R5		General-purpose register	
R6		General-purpose register	
R7		General-purpose register	
R8		General-purpose register	High registers
R9		General-purpose register	
R10		General-purpose register	
R11		General-purpose register	
R12		General-purpose register	
R13 (MSP)	R13 (PSP)	Main Stack Pointer (MSP). Process Stack Pointer (PSP)	
R14		Link Register (LR)	
R15		Program Counter (PC)	

ARM CORTEX-M3 has 13 General Purpose Registers which is divided into two sections.(i) Low Registers (ii) High Registers. These registers are used for data handling purposes. Low registers are from R0 to R7 and High registers are from R8 to R12.

R13 – This register is called Stack Pointer register. It is also called banked register since, there are two registers but one register is visible to the user. The two stack registers are SP_main and SP_Process.

The default stack is SP_main, which will be only used by Handler mode. Both SP_process and SP_main can be handled by Thread mode based on the programming.

R14 – It is called Link Register. When Interrupt occurs, the return address will be stored in Link register and the program counter will move to the ISR. Other than interruption, this register acts as a normal General purpose registers.

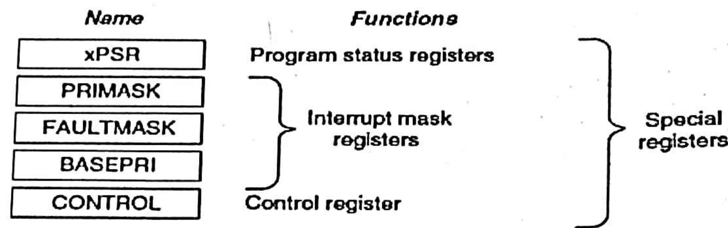
R15 – It is called Program Counter. It is used to store the address of the next instruction that has to be fetched.

Special Registers:

The Cortex-M3 processor also has a number of special registers.

- Program Status registers (PSRs)
- Interrupt Mask registers (PRIMASK, FAULTMASK, and BASEPRI)
- Control register (CONTROL)

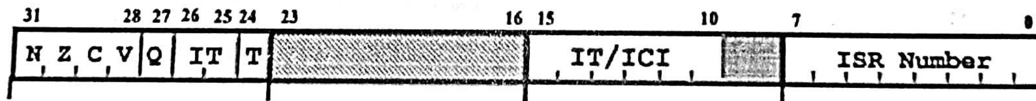
These registers have special functions and can be accessed only by special instructions. They cannot be used for normal data processing.



Special Registers and Their Functions

Register	Function
xPSR	Provide arithmetic and logic processing flags (zero flag and carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and hard fault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

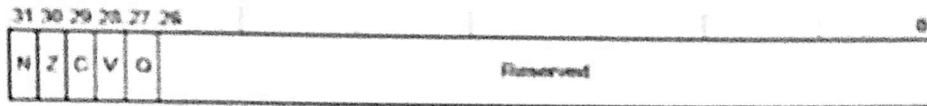
PROGRAM STATUS REGISTER:



- One Status Register consisting of
  - APSR - Application Program Status Register – ALU flags
  - IPSR - Interrupt Program Status Register – Interrupt/Exception No.
  - EPSR - Execution Program Status Register
    - IT field – If/Then block information
    - ICI field – Interruptible-Continuable Instruction information
- xPSR
  - Composite of the 3 PSRs
  - Stored on the stack on exception entry

APPLICATION PSR:

The Application PSR *(APSR)* contains the condition code flags. Before entering an exception, the processor saves the condition code flags on the stack. APSR will be access with the MSR(2) and MRS(2) instructions.



FIELD	NAME	DEFINITION
[31]	N	Negative or less than flag: 1 = result negative or less than 0 = result positive or greater than
[30]	Z	Zero flag: 1 = result of 0 0 = nonzero result.
[29]	C	Carry/borrow flag: 1 = carry or borrow 0 = no carry or borrow.
[28]	V	Overflow flag: 1 = overflow 0 = no overflow.
[27]	Q	Sticky saturation flag.
[26:0]	-	Reserved.

INTERRUPT PSR:

The *Interrupt PSR* (IPSR) contains the *Interrupt Service Routine* (ISR) number of the current exception activation.

31 ... 9	8 ... 0
Reserved	ISR NUMBER

FIELD	NAME	DEFINITION
[31:9]	-	Reserved.
[8:0]	ISR NUMBER	Number of pre-empted exception. Base level = 0 NMI = 2 SVCall = 11 INTISR[0] = 16 INTISR[1] = 17 . . . INTISR[15] = 31 . . . INTISR[239] = 255

EXECUTION PSR:

The *Execution PSR* (EPSR) contains two overlapping fields:

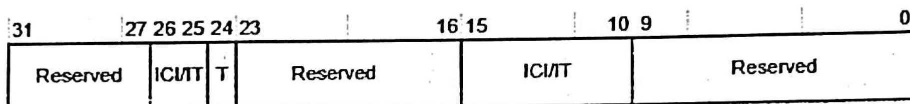
- the *Interruptible-Continuable Instruction* (ICI) field for interrupted load multiple and store multiple instructions
- the execution state field for the *If-Then* (IT) instruction, and the *Thumb state bit* (T-bit).

Interruptible-continuable instruction field:

Load Multiple (LDM) operations and *Store Multiple* (STM) operations are interruptible. The ICI field of the EPSR holds the information required to continue the load or store multiple from the point that the interrupt occurred.

If-then state field:

The IT field of the EPSR contain the execution state bits for the If-Then instruction.



The EPSR is not directly accessible. Two events can modify the EPSR:

- an interrupt occurring during an LDM or STM instruction
- execution of the If-Then instruction.

FIELD	NAME	DEFINITION
[31:27]	-	Reserved.
[26:25], [15:10]	ICI	Interruptible-continuable instruction bits. When an interrupt occurs during an LDM or STM operation, the multiple operation stops temporarily. The EPSR uses bits [15:12] to store the number of the next register operand in the multiple operation. After servicing the interrupt, the processor returns to the register pointed to by [15:12] and resumes the multiple operation. If the ICI field points to a register that is not in the register list of the instruction, the processor continues with the next register in the list, if any.

[26:25], [15:10]	IT	If-Then bits. These are the execution state bits of the If-Then instruction. They contain the number of instructions in the if-then block and the conditions for their execution.
[24]	T	The T-bit can be cleared using an interworking instruction where bit [0] of the written PC is 0. It can also be cleared by unstacking from an exception where the stacked T bit is 0. Executing an instruction while the T bit is clear causes an INVSTATE exception.
[23:16]	-	Reserved.
[9:0]	-	Reserved.